INTEL IA-32 ARCHITECTURE

Introductory programming

Richard Hayden (with thanks to **Naranker Dulay**)
rh@doc.ic.ac.uk

http://www.doc.ic.ac.uk/~rh/teaching
or
https://www.doc.ic.ac.uk/~wl/teachlocal/arch1
or
CATE

Integer addition and subtraction

Instruction	Operation	Notes
`add dst, src`	dst = dst + src	Add
`sub dst, src`	dst = dst – src	Subtract
`cmp dst, src`	dst – src	Compare and set `EFLAGS` 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; register, memory** or **immediate**
- Arithmetic instructions also set `EFLAGS` bits, e.g. **zero flag (ZF), sign flag (SF), carry flag (CF), overflow flag (OF).** These are used by branching instructions

Summary

Today

- Arithmetic and bit-level expressions
- Overflow and division by zero
- Booleans and comparison
- “if” statements and loops

Next week

- Parameter passing
- Local variables

Integer multiply

Instruction	Operation
`imul reg, imm`	reg = reg * imm
`imul destreg, srcreg`	destreg = srcreg * destreg
`imul reg, mem`	reg = reg * mem
`imul destreg, srcreg, imm`	destreg = srcreg * imm
`imul reg, mem, imm`	reg = mem * imm

- Operands can be **word** or **doubleword** sized
- Multiply instructions set both the **carry** and **overflow** flags if the result does not fit into the destination register

Integer divide

Instruction	Operation
`idiv opr` (8-bit)	al = ax **div** opr ah = ax **mod** opr
`idiv opr` (16-bit)	ax = (dx:ax) **div** opr dx = (dx:ax) **mod** opr
`idiv opr` (32-bit)	eax = (edx:eax) **div** opr edx = (edx:eax) **mod** opr

- Operands must be **registers** or **memory** only – to divide by an immediate value, you must load it into a register or memory first

Example expression (1)

- Assume we have the following global variables:

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

- Assume **16-bit 2's complement** values here
- We want to make the following assignment in assembly:

```
alpha = (alpha * beta + 5 * gamma) * (alpha – beta)
```

Other instructions

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

- Quick way to multiply/divide by powers of 2.
- `n` must be an immediate or the byte register `cl`

Instruction	Operation	Notes
`cbw`	ax = al	Convert byte to word
`cwde`	eax = ax	Convert word to doubleword
`cdq`	edx:eax = eax	Convert double to quadword

- Extend a signed integer by filling the extra bits of the destination register with the sign bit of the source register. Under 2's complement, this preserves the sign of the value.

Example expression (2)

```
alpha = (alpha * beta + 5 * gamma) * (alpha – beta)
```

Example expression (2)

alpha = (alpha * beta + 5 * gamma) * (alpha – beta)

```
mov    ax, [alpha]              ; ax = alpha
```

ax	0007						
bx							

Example expression (2)

alpha = (alpha * beta + 5 * gamma) * (alpha – beta)

```
mov    ax, [alpha]              ; ax = alpha
imul   ax, [beta]               ; ax = alpha * beta
```

ax	0007	001C					
bx							

Example expression (2)

alpha = (alpha * beta + 5 * gamma) * (alpha – beta)

```
mov    ax, [alpha]              ; ax = alpha
imul   ax, [beta]               ; ax = alpha * beta
imul   bx, [gamma], 5           ; bx = gamma * 5
```

ax	0007	001C					
bx			FFF1				

Example expression (2)

alpha = (alpha * beta + 5 * gamma) * (alpha – beta)

```
mov    ax, [alpha]              ; ax = alpha
imul   ax, [beta]               ; ax = alpha * beta
imul   bx, [gamma], 5           ; bx = gamma * 5
add    ax, bx         ; ax = (alpha * beta + 5 * gamma)
```

ax	0007	001C		000D			
bx			FFF1				

Example expression (2)

alpha = (alpha * beta + 5 * gamma) * (alpha – beta)

```
mov   ax, [alpha]              ; ax = alpha
imul  ax, [beta]               ; ax = alpha * beta
imul  bx, [gamma], 5           ; bx = gamma * 5
add   ax, bx      ; ax = (alpha * beta + 5 * gamma)
mov   bx, [alpha]              ; bx = alpha
```

ax	0007	001C		000D			
bx			FFF1		0007		

Example expression (2)

alpha = (alpha * beta + 5 * gamma) * (alpha – beta)

```
mov   ax, [alpha]              ; ax = alpha
imul  ax, [beta]               ; ax = alpha * beta
imul  bx, [gamma], 5           ; bx = gamma * 5
add   ax, bx      ; ax = (alpha * beta + 5 * gamma)
mov   bx, [alpha]              ; bx = alpha
sub   bx, [beta]               ; bx = alpha - beta
```

ax	0007	001C		000D			
bx			FFF1		0007	0003	

Example expression (2)

alpha = (alpha * beta + 5 * gamma) * (alpha – beta)

```
mov   ax, [alpha]              ; ax = alpha
imul  ax, [beta]               ; ax = alpha * beta
imul  bx, [gamma], 5           ; bx = gamma * 5
add   ax, bx      ; ax = (alpha * beta + 5 * gamma)
mov   bx, [alpha]              ; bx = alpha
sub   bx, [beta]               ; bx = alpha - beta
imul  ax, bx             ; ax = ax * (alpha - beta)
```

ax	0007	001C		000D			0027
bx			FFF1		0007	0003	

Example expression (2)

alpha = (alpha * beta + 5 * gamma) * (alpha – beta)

```
mov   ax, [alpha]              ; ax = alpha
imul  ax, [beta]               ; ax = alpha * beta
imul  bx, [gamma], 5           ; bx = gamma * 5
add   ax, bx      ; ax = (alpha * beta + 5 * gamma)
mov   bx, [alpha]              ; bx = alpha
sub   bx, [beta]               ; bx = alpha - beta
imul  ax, bx             ; ax = ax * (alpha - beta)

mov   [alpha], ax              ; alpha = ax
```

ax	0007	001C		000D			0027
bx			FFF1		0007	0003	

Integer overflow

- Most arithmetic operations can produce an overflow, for example, signed byte additions 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 using the **conditional jump on overflow** instruction:

```
add    ah, bh     ; add will set EFLAGS.OF on overflow
jo     ov_label   ; Jump to ov_label if overflow

       …
ov_label:    code here to handle overflow
```

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

- `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 for specific bit patterns**. Exactly when does the following code jump to `is_zero`?

```
test   bh, bh
jz     is_zero
```

Integer divide by zero

- Another erroneous condition which usually causes an interrupt to occur (we will learn about these later)

- Can guard against this by checking the divisor for zero before doing the division. Specifically, we check the **zero flag** in the `EFLAGS` register using the **conditional jump if equal** instruction:

```
cmp    bh, 0      ; compare divisor with zero
je     zd_label   ; Jump to zd_label if divisor is zero
idiv   bh         ; otherwise go ahead with division

       …
zd_label:    code here to handle div by zero
```

Boolean expression

- We'll represent booleans using a full byte, 0 for false, true otherwise

```
man    resb    1
rich   resb    1
okay   resb    1
```

- `okay = (man && rich) || (! man)`

Boolean expression

- We'll represent booleans using a full byte, 0 for false, true otherwise

```
man     resb    1
rich    resb    1
okay    resb    1
```

- `okay = (man && rich) || (! man)`

```
mov     al, [man]           ; al = man
```

Boolean expression

- We'll represent booleans using a full byte, 0 for false, true otherwise

```
man     resb    1
rich    resb    1
okay    resb    1
```

- `okay = (man && rich) || (! man)`

```
mov     al, [man]           ; al = man
and     al, [rich]          ; al = man && rich
```

Boolean expression

- We'll represent booleans using a full byte, 0 for false, true otherwise

```
man     resb    1
rich    resb    1
okay    resb    1
```

- `okay = (man && rich) || (! man)`

```
mov     al, [man]           ; al = man
and     al, [rich]          ; al = man && rich
mov     ah, [man]           ; ah = man
```

Boolean expression

- We'll represent booleans using a full byte, 0 for false, true otherwise

```
man     resb    1
rich    resb    1
okay    resb    1
```

- `okay = (man && rich) || (! man)`

```
mov     al, [man]           ; al = man
and     al, [rich]          ; al = man && rich
mov     ah, [man]           ; ah = man
not     ah                  ; ah = ! man
```

Boolean expression

- We'll represent booleans using a full byte, 0 for false, true otherwise

```
man     resb    1
rich    resb    1
okay    resb    1
```

- `okay = (man && rich) || (! man)`

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

Jump instructions

Instruction	Flag condition	Notes
`jmp label`	Unconditional	Jump
`je/jz label`	ZF = 1	Jump if zero (equal)
`jne/jnz label`	ZF = 0	Jump if not zero (not equal)
`jg label`	ZF = 0 and SF = 0	Jump if greater than
`jge label`	SF = 0	Jump if greater than or equal to
`jl label`	SF = 1	Jump if less than
`jle label`	ZF = 1 or SF = 1	Jump if less than or equal to

- There are also unsigned versions: **jump above (`ja`), jump above or equal (`jae`), jump below (`jb`) and jump below or equal (`jbe`)**
- Similar instructions exist for other flag registers, e.g. `jo` which you saw earlier

Boolean expression

- We'll represent booleans using a full byte, 0 for false, true otherwise

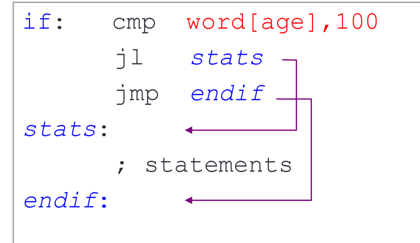
```
man     resb    1
rich    resb    1
okay    resb    1
```

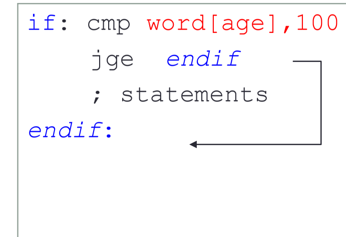
- `okay = (man && rich) || (! man)`

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

If statement (1)

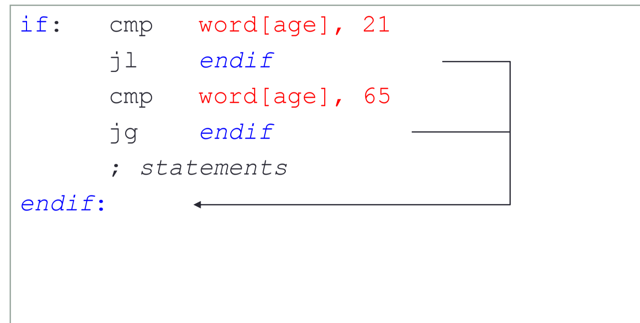
```
if  (age < 100) {
  statements
}
```





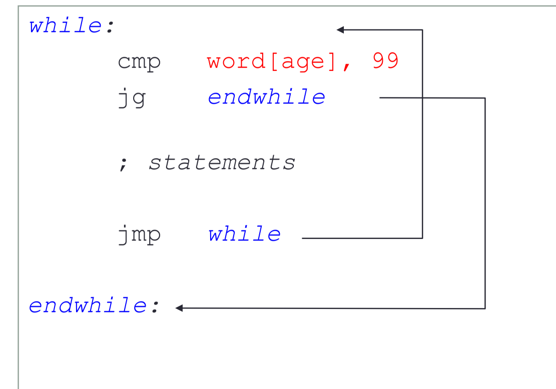
If statement (2)

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



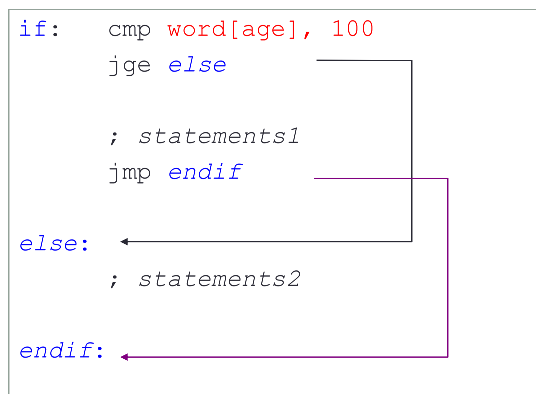
While loop

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



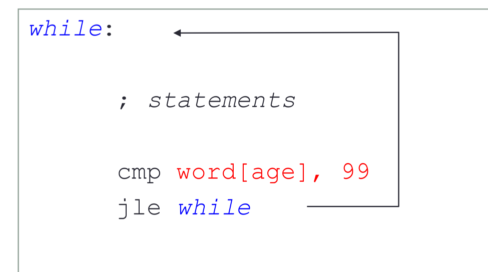
If-then-else statement

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



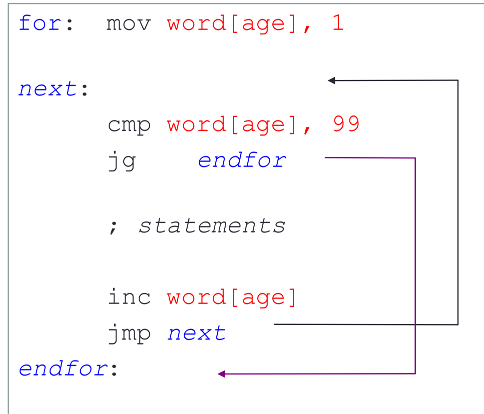
Do-while loop

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



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:
```