Previous Lecture

We learnt about:

Registers and some simple instructions and instruction formats.

We briefly mentioned Segment Registers.

However, in our programs we are using the ‘flat’ memory model so segment registers are basically irrelevant.

All segment registers point to the same memory block.

This Lecture

We will now take a closer look at the addressing modes, and we will start to take a look the various loop and jump instructions.

Some of the ways we can implement structures such as the “if” statement, and the “for” and “while loops” in assembler.

Addressing Modes

We will see:

That the 8086 offers the programmer a wide number of choices when referring to a memory location.

Many believe that the number of addressing modes contained in a microprocessor is a measure of its power.

Many of the addressing modes are used to generate a physical address in memory.

One of the segment registers (CS,DS,ES,SS,FS,GS) will always supply the first 16-bits of the address in the old processors but are fixed in 32-bit mode.

The second 16-bit address is formed by a specific addressing mode operation.

Addressing Modes

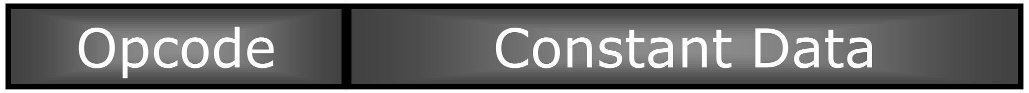
Addressing mode	Pentium	UltraSPARC	JVM
Immediate	●	●	●
Direct	●		
Register	●	●	
Register	●		
Indexed	●	●	●
Based-Indexed	●	●	
Stack			●

Don't need many addressing modes for an effective ISA
- Immediate, Direct, Register and Indexed modes are usually enough

Most code will be generated by a compiler
- Fewer choices means the compiler can be simpler

Complex addressing modes may reduce the number of instructions, but may be hard to implement efficiently

Immediate Addressing



This is the simplest addressing mode. The instruction contains *data* to be used by the instruction rather than the address of the data

- E.g. “Load ECX with the value 1024 ”

```
MOV ECX,1024
```

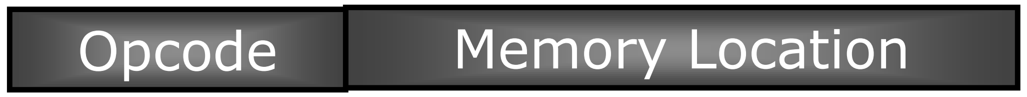
The data is available when the instruction is fetched

- Don’t need to get data from the memory

Useful for dealing with constants

- But the constant may be limited by the size of the instruction

Direct Addressing



The instruction contains the address of the data to be used by the instruction

- E.g. “Load register EAX with the data at memory location 1234”

  ```
  MOV EAX, [1234]
  ```

  The instruction will always access exactly the same memory location

Need to know the desired address at compile-time

- Useful for dealing with global variables

Register Addressing

Opcode	Register Specification

Like direct addressing, but the data is in a register rather than a memory location

- E.g. “Add registers EBX and ECX and put the result in register EAX”

  `ADD EBX, ECX`

This mode is very common

- RISC machines (e.g. UltraSPARC II) are specifically designed to operate in this mode
- Register-to-register operations are very fast

Frequently combined with other modes

Register Indirect Addressing

Opcode	Register Specification

The register contains the address of the data to be used

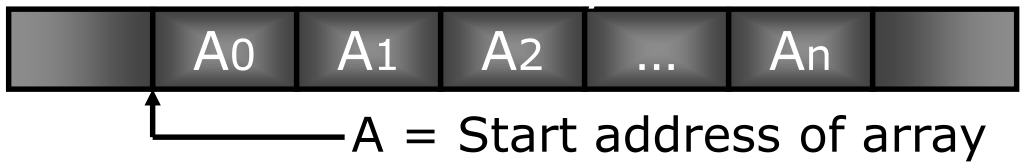
- E.g. “Set the memory location pointed at by a register to zero”

```
MOV [EBX], 0
```

Useful for accessing sequential data structures

- E.g. Arrays

Accessing Array Elements using Indirect Addressing

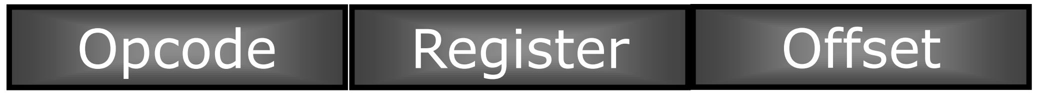


```
  MOV EBX, A      ; Get address of array
  MOV ECX, 10     ; Number of elements in the array

STEP:             ; Start of loop
  MOV [EBX], 0    ; Set current location to zero
  INC EBX         ; move to next location
  CMP EBX, ECX    ; if the current != end, then
  JLE STEP        ; we need to process more elements
```

We can use *indirect addressing* to access all the elements of an array at address *A* in turn by using a register to hold the current address

Indexed Addressing



Access memory at a constant offset from a memory address stored in a register

- E.g. “Load EAX with the data at the address stored in BX plus 8 bytes”

```
MOV EAX, [EBX + 8]
```

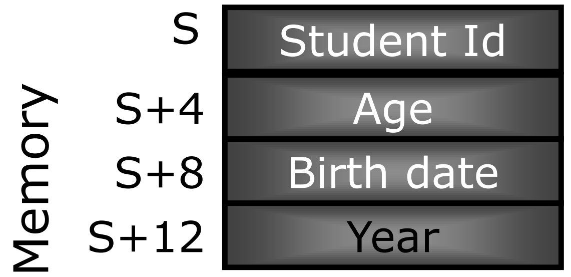
Useful for accessing record structures

Also used for local variables and arrays

Accessing a Structure using Indexed Addressing

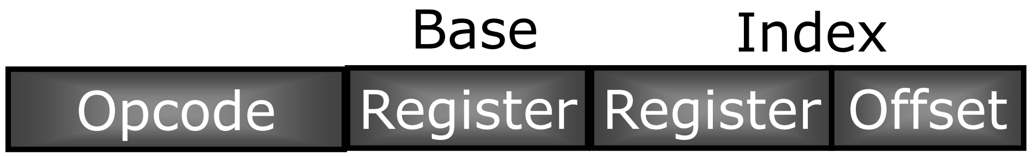
Suppose we have a student record with four numerical fields at address *S*

We can access the individual elements using *indexed addressing*



```
MOV EBX, S       ; Get address of record
MOV EAX, [EBX]   ; Get student's ID from record
MOV EAX, [EBX+4] ; Get Student's age from record
MOV EAX, [EBX+8] ; Get Student's birth date from record
MOV EAX, [EBX+12]; Get student's year from record
```

Based-Indexed Addressing



Access memory at a constant offset from a memory address computed by adding two registers

- E.g. “Load EAX with the data at the memory address found by adding the contents of EBX and ECX and adding 4 bytes”

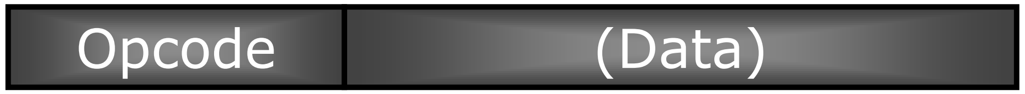
```
MOV EAX, [EBX + ECX + 4]
```

“Generalized” addressing mode

- Can set one register or the offset to zero to get indexed addressing or indirect addressing

Useful for accessing record structures, arrays and arrays of records

Stack Addressing



All operands are taken from the stack

- No addresses are needed for most instructions

The instruction only needs to specify the operation to perform

- May also need to specify constants or memory locations in the instruction too
- E.g to load a constant

Used in the JVM

Summary

The number of addresses used in an instruction is fundamental to the design of the processor

The different types of addressing mode available are useful in different context

Not all processors support all addressing modes

Loops and Jumps

Loop and Jump Instructions

At some time during execution of a program we need to make decisions and want the program to change its direction of execution.

This is usually as a result of some condition such as a flag changing.

In other words we want to ***Jump*** to another section of code.

Compare (CMP) Instruction

The CMP instruction performs an implied subtraction of two operands. Neither of which are actually altered during the process.

CMP changes the *Overflow, Sign, Zero, Carry, Auxiliary Carry and Parity Flags,* according to the value the destination operand would have had if a subtraction had occurred.

CMP con’t

CMP Results	Zero Flag	Carry Flag
Destination < Source	0	1
Destination > Source	0	0
Destination = Source	1	0

CMP is valuable because it provides the basis for most conditional logic structures.
For instance if you follow a CMP with a conditional jump instruction you can create the assembler equivalent of a IF statement.

Jump Instructions

The 80x86 family have a rich supply of jump instructions. A full list is given in the next couple of slides.

Note there are many different types defined, some are redundant in that they simply provide a different name for an existing instruction.

Conditional Jumps

JZ	Zero	Z-flag = 1
JNZ	Zero	Z-flag = 0
JA	Above	(CF and ZF) =0
JB	Below	CF = 1
JAE	Above or equal	CF =0
JBE	Below or equal	(CF or ZF) = 1
JNC	No Carry	CF = 0
JG	Greater	ZF = 0 and SF = OF
JL	Less	SF <> OF

Conditional Jumps CON’T

JGE	Greater or Equal	SF = OF
JLE	Less or Equal	(ZF = 1) or (SF <> OF)
JO	Overflow	OF = 1
JNO	Not Overflow	OF = 0
JS	Sign	SF = 1
JNS	No sign	SF = 0
JPO	Parity Odd	PF = 0
JPE	Parity Evan	PF = 1
JCXZ	CX is equal to zero	none

Alias for Conditional Jumps

JZ	**JE**	Jump if equal
JNZ	**JNE**	Jump if Not Equal
JA	**JNBE**	Jump if Not Below or Equal
JB	**JNAE**	Jump if Not Above or Equal
	JC	Jump if Carry
JAE	**JNB**	Jump if Not Below
JBE	**JNA**	Jump if Not Above
JNC	**No alias**	
JG	**JNLE**	Jump if Not Less or Equal
JL	**JNGE**	Jump if Not Greater nor Equal
JGE	**JNL**	Jump if Not Less
JLE	**JNG**	Jump if not Greater
JO	**no alias**	
JNO	**no alias**	
JS	**no alias**	
JNS	**no alias**	
JPO	**JNP**	Jump if Not Parity
JPE	**JP**	Jump if Parity
JCXZ	**no alias**	

SEVEN ‘Survival kit’ Jumps

JMP	Jump (no matter what)
JZ	Jump if the result was zero
JNZ	Jump if the result was Not Zero
JA	Jump if the result was above zero
JB	Jump if the result was Below zero
JG	Jump if the result Greater than zero
JL	Jump if the result was less than zero

Conditional Jumps con’t

Note:

- By default MASM requires the destination of the jump to be within the current procedure.
- Prior to Intel386 All of the conditional jump instructions were coded as short jumps, allowing them a relative range of only +127 to -128 bytes from there current location. IA-32 processors have no such limits.
- None of the jumps affect the flags.

If Statements

We can combine logical operations and conditional jumps to build "if" statements

```
if (a < b) then X = 1 else X = 2;
    MOV EAX, A              ; Load value of A
    CMP EAX, B              ; Compare it with B
    JGE ElseCase            ; If A >= B then "else"
    MOV X, 1                ; Otherwise do "then"
    JMP EndIf               ; Jump to end of if
  ElseCase:
    MOV X, 2                ; Do "else" case
  EndIf:
```

CMP DESTINATION, SOURCE

YOU COMPARE THE DESTINATION TO SOURCE

CMP EAX, VALUE

JG TARGET

IF EAX IS LARGER THAN VALUE THEN JUMP

For Loops

We can combine logical operations and conditional jumps to build “for” loops

```
for (i=1; i<=10; i++) x = x + i;


  MOV ECX, 1         ; initialise counter
ForLoop:
  ADD X, ECX         ; do loop body
  INC ECX      ; increment counter
  CMP ECX, 10        ; do termination test
  JLE ForLoop  ; repeat if CX <= 10
```

Also useful for repeating an instruction several times

The **LOOP** Instruction

LOOP *label*

- Decrement (e)**CX** and branch to the label if **CX** is not zero
- If **CX** =0 initially, will repeat 65535 times

Previous example rewritten:

```
   MOV ECX, 10          ; initialise counter
ForLoop:
   ADD X, ECX           ; do loop body
   LOOP ForLoop         ; repeat if ECX > 1
```

But we had to *reverse* the loop to use this instruction

The LOOP instruction

This instruction will decrement register (E)CX and perform a jump to the target address if (E)CX does not equal zero.

LOOP is very useful for routines that need to repeat a specific number of times.

(E)CX must be loaded prior to entering the section of code terminated by loop.

LOOP instruction example

```
            mov     ECX,10
Back:  -------
            -------
            -------
            -------
            loop    Back
```

This will loop 10 times before exiting the routine

Nested Loops

The next slide shows an example of a nested loop.

The outer loop will execute 5 times.

The inner loop will execute 50 times.

The **push** and **pop** instructions are used to preserve the contents of the outer loop counter, via the use of the stack

Nested Loop example

```
                mov     CX,5
OUTER:          push    CX
                mov     CX,10
INNER:          -------
                -------
                -------
                -------
                loop    INNER
                pop     CX
                loop    OUTER
```

LOOPE/LOOPZ

Loop if equal and Loop if Zero.

This instruction is similar to LOOP but a secondary condition must also be met.

In addition to decrementing CX LOOPZ also examines the state of the zero flag. If set and CX not equal to zero LOOPZ will jump to the target.

If CX equals zero, or the zero flag gets cleared within the loop, the loop will terminate.

LOOPE is an alternate name for this instruction.

There is an instruction LOOPNE/LOOPNZ which is the opposite of LOOPE/LOOPZ.

While Loops

Very similar to for loops, but we don't need a loop counter

```
A=0; while (A < 100) A = A + 1;
  MOV A, 0          ; initialise A
WhileLoop:
  CMP A, 100        ; If test fails then
  JGE WhileEnd      ;    we can stop
  INC A             ; Otherwise do body
  JMP WhileLoop     ;    and repeat
WhileEnd:
         ...
```