

9-Oct-06 (1)



## CSC2510 - Computer Organization

Lecture 6: Pentium IA-32

Philip Leong

9-Oct-06 (2)



## Introduction

- Intel is by far the most successful computer architecture to date
- Describe Intel architecture (IA) 32-bit machines (hence IA-32)
- First IA-32 was the 80386 (1985), then 80486 (1989), Pentium (1993), Pentium Pro (1995), Pentium II (1997), Pentium III (1999), Pentium 4 (2000), Intel Core (2006)

9-Oct-06 (3)



## Environment

- In this course we will use the MinGW environment ([www.mingw.org](http://www.mingw.org))
- GNU toolchain (GNU assembler and linker)
- C runtime library for I/O
- See course homepage for more links as well as MinGW install instructions
  - A good summary of IA-32 instructions is available from textbook and/or <http://www.cs.princeton.edu/courses/archive/fall06/cs318/docs/pc-arch.html>

9-Oct-06 (4)



## GNU Assembler (gas)

- Because GAS was invented to support a 32-bit unix compiler, it uses standard AT&T syntax
- This syntax is neither worse, nor better than the Intel syntax. It's just different.
- When you get used to it, you find it much more regular than the Intel syntax.
- **We are going to give lectures using Intel syntax but exercises will use GAS (a real life issue).**

9-Oct-06 (5)



## Registers and Addressing

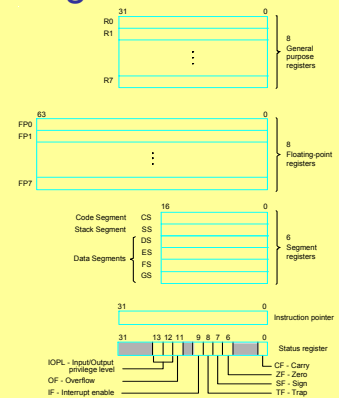
- Memory is byte addressable using 32-bit addresses
- Instructions operate on data operands of 8 or 32 bits (byte and doubleword)
- Little endian
- Multiple byte data operands may start at any byte address (no alignment necessary)

9-Oct-06 (6)



## IA-32 Registers

- 8x 32-bit general purpose registers
- 8x floating point registers (doubleword or quadword) with extension to 80-bits internally for increased accuracy
- Memory divided into **segments** and controlled by segment registers
- Instruction pointer = program counter



9-Oct-06 (7)



## Segments

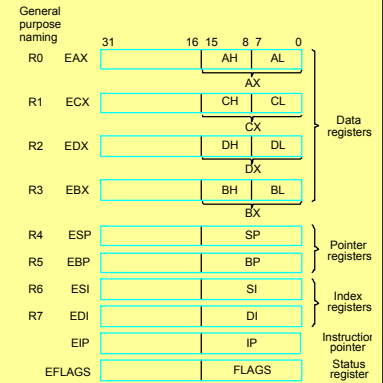
- Segments
  - Code segment holds program instructions
  - Data segment holds data operands
  - Stack segment holds processor stack
- Status register
  - Condition codes CF, ZF, SF, OF
  - Program execution mode bits (IOPL, IF, TF) associated with IO and interrupts

9-Oct-06 (8)



## Register Names

- Registers in early processors map to IA32 registers
- Grouped into data, pointer and index registers
- The E-prefix means a 32-bit version of the register
- We will use this naming convention



9-Oct-06 (9)



## IA-32 Addressing modes

Name	Assembler syntax	Addressing function
Immediate	Value	Operand = Value
Direct	Location	EA = Location
Register	Reg	EA = Reg that is, Operand = [Reg]
Register indirect	[Reg]	EA = [Reg]
Base with displacement	[Reg + Disp]	EA = [Reg] + Disp
Index with displacement	[Reg * S + Disp]	EA = [Reg] * S + Disp
Base with index	[Reg1 + Reg2 * S]	EA = [Reg1] + [Reg2] * S
Base with index and displacement	[Reg1 + Reg2 * S + Disp]	EA = [Reg1] + [Reg2] * S + Disp

Value	= an 8- or 32-bit signed number
Location	= a 32-bit address
Reg, Reg1, Reg2	= one of the general purpose registers EAX, EBX, ECX, EDI, ESI, ESP, EBP, with the exception that ESP cannot be used as an index register
Disp	= an 8- or 32-bit signed number, except that in the Index with displacement mode it can only be 32 bits.
S	= a scale factor of 1, 2, 4, or 8

9-Oct-06 (10)



## Examples

- Immediate
  - MOV EAX,25 (decimal)
  - MOV EAX,3FA00H (the H suffix means hexadecimal)
  - NUM EQU 25  
MOV EAX,NUM
  - MOV EBX,OFFSET LOC (where LOC is an address label)
- Direct
  - MOV EAX,[LOC] (brackets not needed if LOC is an address label)

9-Oct-06 (11)



## Examples

- Register
  - MOV EAX,EBX
- Register indirect
  - MOV EAX,[EBX]
- Immediate, direct, register and register indirect are the basic addressing modes. The ones that follow give more flexibility.

9-Oct-06 (12)



## Other addressing modes

- Base with displacement
  - MOV EAX,[EBP+60] (word)
  - MOV EAL,[EBP+4] (byte)
- Base with index and displacement
  - MOV EAX,[EBP+ESI\*4+200]
- Have both of these modes as base with displacement can be encoded with 1 less byte

9-Oct-06 (13)

## Base with index + disp

(a) Base with displacement mode, expressed as  $[EBP] + 60$

(b) Base with displacement and index mode, expressed as  $[EBP] + [ESI] \cdot 4 + 200$

9-Oct-06 (14)

## IA-32 Instructions

- `MOV dst,src` ( $dst \leftarrow [src]$ )
- `ADD dst,src` ( $dst \leftarrow [dst] + [src]$ )
- `SUB dst,src` ( $dst \leftarrow [dst] - [src]$ )
- `JG LOOPSTART` (G means greater than 0)
- `MOV EBX,OFFSET LOCATION`
  - Loads address of label LOCATION into EBX
  - What if it is not a fixed address? Use load effective address which is computed dynamically. `LEA EBX,[EBP+12]`

9-Oct-06 (15)

## Operands

- Note only one operand can be in memory so  $C \leftarrow [A] + [B]$ 
  - `MOV EAX,A`
  - `ADD EAX,B`
  - `MOV C,EAX`

9-Oct-06 (16)

## A simple program

<code>LEA EBX,NUM1</code>	Initialize base (EBX) and counter (ECX) registers.
<code>MOV ECX,N</code>	
<code>MOV EAX,0</code>	Clear accumulator (EAX)
<code>MOV EDI,0</code>	and index (EDI) registers.
<code>STARTADD: ADD EAX,[EBX + EDI * 4]</code>	Add next number into EAX.
<code>INC EDI</code>	Increment index register.
<code>DEC ECX</code>	Decrement counter register.
<code>JG STARTADD</code>	Branch back if $[ECX] > 0$ .
<code>MOV SUM,EAX</code>	Store sum in memory.

9-Oct-06 (17)

## Improve program

- The LOOP instruction
  - `LOOP STARTADD`
  - Decrements ECX and branches to STARTADD if  $ECX \neq 0$
  - Equivalent to
    - `DEC ECX`
    - `JG STARTADD`
- Can use a single register rather than EDI,ECX
  - Should choose ECX as it can be used with LOOP
  - Do the loop backwards

9-Oct-06 (18)

## Improved version

<code>LEA EBX,NUM1</code>	Load base register EBX and adjust to hold $NUM1 - 4$ .
<code>SUB EBX,4</code>	
<code>MOV ECX,N</code>	Initialize counter/index (ECX).
<code>MOV EAX,0</code>	Clear the accumulator (EAX).
<code>STARTADD: ADD EAX,[EBX + ECX * 4]</code>	Add next number into EAX.
<code>LOOP STARTADD</code>	Decrement ECX and branch back if $[ECX] > 0$ .
<code>MOV SUM,EAX</code>	Store sum in memory.

(b) More compact program

9-Oct-06 (19)



## Intel/gas differences

	GNU Syntax (AT&T)	Intel
Immediate operands	Preceded by "\$" e.g. <code>push \$4</code> <code>movl \$0x000a, %eax</code>	Not denoted e.g. <code>push 4</code> <code>mov ebx, 000ah</code>
Register operands	Preceded by "%" e.g. <code>eax</code>	Not denoted e.g. <code>eax</code>
Argument order (e.g. add the address of C variable 'foo' to register EAX)	source[, source], dest e.g. <code>addl \$foo, %eax</code>	dst, source[, source] e.g. <code>add eax, foo</code>
Single-size operands	Explicit with operand size opcode(b, w) e.g. <code>movl foo, %eax</code>	Implicit with register name, byte ptr, word ptr, or dword ptr e.g. <code>movsb, foo</code>
Address a C variable 'foo'	<code>foo</code>	<code>[foo]</code>
Address memory pointed by a register (e.g. EAX)	<code>(%eax)</code>	<code>[eax]</code>
Address a variable offset by a value (e.g. esp)	<code>foo(%eax)</code>	<code>[eax+foo]</code>
Address a value in an array 'foo' of 32-bit integers	<code>foo(, %eax, 4)</code>	<code>[eax*4+foo]</code>
Equivalent to C code *ptr	<code>!(%eax)</code>	<code>[EAX holds the value of y, then [eax+1]</code>

9-Oct-06 (20)



## Machine Instruction Format

- General format of instructions is as below
- Ranges from 1 to 12 bytes
- Most instructions only require 1 opcode byte
- Instructions that only use one register to generate effective address of operand only take 1 byte
- Need to be able to figure out length of instruction

OP code	Addressing mode	Displacement	Immediate
1 or 2 bytes	1 or 2 bytes	1 or 4 bytes	1 or 4 bytes

9-Oct-06 (21)



## One byte instructions

- Instructions that only use one register to generate effective address of operand only take 1 byte
  - E.g. `INC EDI`, `DEC ECX`
  - Registers specified by 3-bit codes in the single opcode byte

9-Oct-06 (22)



## Immediate mode encoding

- `MOV EAX, 820`
  - 5 bytes
  - One byte opcode to specify move operation, that a 32-bit operand is used and the name of the destination register
  - 4-byte immediate value of 820
- `MOV DL, 5`
  - 2 bytes as immediate value is 8-bits
- `MOV DWORD PTR [EBP+ESI*4+DISP], 10`
  - `DWORD PTR (doubleword)` specifies a 32-bit operation

9-Oct-06 (23)



## Displacement fields

- One operand of a two-operand instruction usually a register. Other can be register or memory
- Two exceptions where both can be in memory
  - Source operand is immediate and destination is in memory
  - Push/pop
- When both operands are in registers, only one addressing mode byte needed
  - `ADD EAX, EBX` encoded in 2 bytes
  - One for opcode and the other for addressing mode (specifies the two registers)

9-Oct-06 (24)



## Displacement fields

- `MOV ECX, N`
  - 6 bytes
  - Opcode
  - Addressing mode (specifies direct mode and destination register)
  - Four bytes for address N
  - A **direction bit** in the opcode specifies which operand is the source
- `ADD EAX, [EBX+EDI*4]`
  - 3 bytes
  - Opcode
  - Two addressing mode bytes as two registers used to generate effective address of source operand

9-Oct-06 (25)



## A full program

```

Assembler directives {
    .data
    NUM1 DD 17, 3, -51, 242, -113
    N DD 5
    SUM DD 0
    .code
Statements that generate machine instructions {
    MAIN : LEA EBX, NUM1
          SUB EBX, 4
          MOV ECX, N
          MOV EAX, 0
    STARTADD : ADD EAX, [EBX + ECX * 4]
              LOOP STAR TADD
              MOV SUM, EAX
Assembler directive END MAIN

```

- .data and .code specify start of data and code in the program
- DD allocates 32-bit locations for variables
- MAIN specifies the start of the program
- END indicates the end of the MAIN procedure

9-Oct-06 (26)



## Conditional jumps

DEC ECX  
JG STARTADD

- JG relates to results of the most recently executed data manipulation instruction (in this case DEC ECX)
- Jumps if result was > 0 (ECX > 0 in this case)
- Assume STARTADD was 1000 and address after the JG is 1007, relative address is -7 and is stored in the instruction
- Since only one byte is used, jump range is -128 to 127. A 4-byte offset is used if the address is outside this range
- Other jumps such as jump if equal to 0 (JZ or JE), jump if sign bit is set (i.e. result was negative JS) etc

9-Oct-06 (27)



## Compare Instruction

- CMP dst,src
  - [dst]-[src]
  - Only used to set condition codes
  - Neither operand changed

9-Oct-06 (28)



## Unconditional jump

- JMP ADDR
  - One byte or four byte offset forms just list JG
  - More powerful addressing modes also allowed e.g. JMP [JUMPTABLE+ESI\*4] (index with displacement)

9-Oct-06 (29)



## Logical/Shift/Rotate

- AND EBX,EAX
  - E.g. if EAX=0000FFFFH and EBX=02FA62CAH what is the result?
- NOT EBX
- SHL dst,count
  - Also SHR, SAL (same as SHL), SAR
- ROL, ROR, RCL, RCR

9-Oct-06 (30)



## Digit Packing

```

LEA  EBP,LOC      EBP points to first byte.
MOV  AL,[EBP]     Load first byte into AL.
SHL  AL,4         Shift left by 4 bit positions.
MOV  BL,[EBP+1]   Load second byte into BL.
AND  BL,0FH       Clear high-order 4 bits to zero.
OR   AL,BL        Concatenate the BCD digits.
MOV  PACKED,AL    Store the result.

```

9-Oct-06 (31)



## I/O Operations

- READWAIT BT INSTATUS,3  
JNC READWAIT  
MOV AL,DATAIN
- ; BT transfers INSTATUS bit 3 to the carry flag
- WRITEWAIT BT OUTSTATUS,3  
JNC WRITEWAIT  
MOV DATAOUT,AL

9-Oct-06 (32)



## MEMORY MAPPED I/O

---

	LEA	EBP,LOC	EBP points to memory area.
READ:	BT	INSTATUS,3	Wait for character to be
	JNC	READ	entered into DATAIN.
	MOV	AL,DATAIN	Transfer character into AL.
	MOV	[EBP],AL	Store the character in memory
	INC	EBP	and increment pointer.
ECHO:	BT	OUTSTATUS,3	Wait for display to
	JNC	ECHO	be ready.
	MOV	DATAOUT,AL	Send character to display.
	CMP	AL,CR	If not carriage return,
	JNE	READ	read more characters.

---

Figure 3.44. An IA-32 program that reads a line of characters and displays it.

9-Oct-06 (33)



## ISOLATED I/O

- IN and OUT instructions used only for I/O
- Addresses for these instructions are in a separate address space to other instructions
- IN REG,DADDR and OUT DEVADDR,REF
  - REG must be AL or EAX
  - 16-bit address space, if 0-255, specified in the opcode
  - Otherwise use DX for DADDR e.g. IN REG,DX