Introduction

- Intel is by far the most successful computer architecture to date
- Describe Intel architecture (IA) 32-bit machines (hence IA-32)

Environment

- In this course we will use the MinGW environment (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/cos318/docs.pc-arch.html

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).

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)

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
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

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

IA-32 Addressing modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler syntax</th>
<th>Addressing function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Value</td>
<td>Oper and = Value</td>
</tr>
<tr>
<td>Direct</td>
<td>Location</td>
<td>EA = Location</td>
</tr>
<tr>
<td>Register</td>
<td>Reg</td>
<td>EA = Reg</td>
</tr>
<tr>
<td>Register indirect</td>
<td>[Reg]</td>
<td>EA = [Reg]</td>
</tr>
<tr>
<td>Indexed with displacement</td>
<td>[Reg + Disp]</td>
<td>EA = [Reg] + Disp</td>
</tr>
<tr>
<td>Indexed with index and displacement</td>
<td>[Reg1 + Reg2 S + Disp]</td>
<td>EA = [Reg1] + [Reg2 S] + Disp</td>
</tr>
<tr>
<td>Value</td>
<td>= an 8- or 32-bit signed number</td>
<td></td>
</tr>
<tr>
<td>Location</td>
<td>= a 32-bit address</td>
<td></td>
</tr>
<tr>
<td>Reg, Reg1, Reg2</td>
<td>= one of the general purpose registers EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, with the exception that ESP cannot be used as an index register</td>
<td></td>
</tr>
<tr>
<td>Disp</td>
<td>= an 8- or 32-bit signed number, except that in the indexed with displacement mode it can only be 32 bits</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>= a scale factor of 1, 2, 4, or 8</td>
<td></td>
</tr>
</tbody>
</table>

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)

Examples

- Register
  - MOV EAX,EBX
- Register indirect
  - MOV EAX,[EBX]

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
### Base with index + disp

![Diagram of base with index + disp](image)

- 1000 bytes
- Base register EBP
- Operation address (EA) = [EBP] + 60

### IA-32 Instructions

- MOV dst, src (dst ← [src])
- ADD dst, src (dst ← [dst] + [src])
- SUB dst, src (dst ← [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]

### Operands

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

### A simple program

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA EBX, NUM1</td>
<td>Initialize base (EBX) and counter (ECX) registers.</td>
</tr>
<tr>
<td>MOV ECX, N</td>
<td>Initialize counter/index (ECX) registers.</td>
</tr>
<tr>
<td>MOV EAX, 0</td>
<td>Clear the accumulator (EAX).</td>
</tr>
<tr>
<td>STAR TADD: ADD EAX, [EBX + ECX 4]</td>
<td>Add next number into EAX.</td>
</tr>
<tr>
<td>INC EDI</td>
<td>Increment index register.</td>
</tr>
<tr>
<td>JG STAR TADD</td>
<td>Decrement counter register.</td>
</tr>
<tr>
<td>MOV SUM, EAX</td>
<td>Store sum in memory.</td>
</tr>
</tbody>
</table>

### Improve program

- The LOOP instruction
  - LOOP STARTADD
  - Decrement ECX and branches to STARTADD if ECX ≠ 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

### Improved version

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA EBX, NUM1</td>
<td>Load base-register EBX and adjust to hold NUM1 = 4.</td>
</tr>
<tr>
<td>MOV ECX, N</td>
<td>Initialize counter (ECX).</td>
</tr>
<tr>
<td>MOV EAX, 0</td>
<td>Clear the accumulator (EAX).</td>
</tr>
<tr>
<td>STARTADD: ADD EAX, [EBX + ECX 4]</td>
<td>Add next number into EAX.</td>
</tr>
<tr>
<td>LOOP STARTADD</td>
<td>Decrement ECX and branch back if [ECX] &gt; 0.</td>
</tr>
<tr>
<td>MOV SUM, EAX</td>
<td>Store sum in memory.</td>
</tr>
</tbody>
</table>

(b) More compact program
**Intel/gas differences**

**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

<table>
<thead>
<tr>
<th>OP code</th>
<th>Addressing mode</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 or 2 bytes</td>
<td>1 or 2 bytes</td>
<td>1 or 4 bytes</td>
<td>1 or 4 bytes</td>
</tr>
</tbody>
</table>

**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

**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

**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)

**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
A full program

Statements that generate machine instructions:

MAIN:
- LEA EBX, NUM1
- SUB EBX, 4
- MOV ECX, 0
- STARTADD:
  - ADD EAX, [EBX + ECX * 4]
  - LOOP STARTADD
- MOV SUM, EAX

Assembler directives:
- .data
  - DD allocates 32-bit locations for variables
- .code
- MAIN
- END MAIN

Conditional jumps

DEC ECX
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

Compare Instruction

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

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)

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

Digit Packing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA EBP, LOC</td>
<td>EBP points to first byte.</td>
</tr>
<tr>
<td>MOV AL[EBP]</td>
<td>Load first byte into AL.</td>
</tr>
<tr>
<td>SHL AL,4</td>
<td>Shift left by 4 bit positions.</td>
</tr>
<tr>
<td>MOV BL[EBP+1]</td>
<td>Load second byte into BL.</td>
</tr>
<tr>
<td>AND BL, 0FH</td>
<td>Clear high-order 4 bits to zero.</td>
</tr>
<tr>
<td>OR AL, BL</td>
<td>Concatenate the BCD digits.</td>
</tr>
<tr>
<td>MOV PACKED, AL</td>
<td>Store the result.</td>
</tr>
</tbody>
</table>
**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

**MEMORY MAPPED I/O**

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

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

**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