IA-32 (x86) Architecture

History

As technology improved over the years, there developed a race to get the first (usable) processors on a single integrated circuit.

When able to place approximately 10,000 transistors on a single IC, then we have just about enough circuitry to put a (simple) processor on a this single IC.

The Intel 8086 was Intel's entry in the race. On the way to getting their processor out (on the market) as fast as possible, they made some unusual design decisions.

```
Year
1974    8080 8-bit architecture with 8-bit bus

1978    8086 16 bit architecture w/ 16-bit bus

        8088 - like 8086, 16-bit architecture, but
               only had an 8-bit (internal) bus
        selected for IBM PC -- golden handcuffs

1980    8087 FPU

1982    80286 24-bit weird addresses

1985    80386 32b registers and addresses

1989    80486,
1993    Pentium,
1995    Pentium Pro -- few changes

1997    MMX
```

Backward Compatibility

SECTION NOT YET WRITTEN!

Current implementations

Pentium Pro and after

Instruction decode translates machine code into "RISC OPS" (like decoded MIPS instructions)

An **execution unit** runs RISC OPS

+ Backward compatiblity
- Complex decoding
+ execution unit as fast as RISC like MIPS

The Pentium Architecture

- It is **not** a load/store architecture.
- The instruction set is huge! We go over only a fraction of the instruction set. 16bit, 32bit operations on memory and registers decoding nightmare: a single machine code instruction can be from 1 to 17 bytes long w/ prefixes & postfixes. But, mainline (most common) 386 instructions not terrible
- There are lots of restrictions on how instructions/operands are put together, but there is also an amazing amount of flexibility.

Registers

The Intel architectures as a set just do not have enough registers to satisfy most assembly language programmers. Still, the processors have been around for a LONG time, and they have a sufficient number of registers to do whatever is necessary.

For our (mostly) general purpose use, we get

32-bit	16-bit	8-bit (high part of 16)	8-bit (low part of 16)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

and

EBP	BP
ESI	SI
EDI	DI
ESP	SP

There are a few more, but we will not use or discuss them. They are only used for memory accessability in the segmented memory model.

Note that it is unusual to be able to designate part of a register as an operand. This evolved, due to the backward compatibility to previous processors that had 16-bit registers.

Using the registers:
As an operand, just use the name (upper case and lower case both work interchangeably).

EBP is a frame pointer.
ESP is a stack pointer.

ONE MORE REGISTER:
Many bits used for controlling the action of the processor and setting state are in the register called EFLAGS. This register contains the condition codes:

```
OF  Overflow flag
SF  Sign flag
ZF  Zero flag
PF  Parity flag
CF  Carry flag
```

The settings of these flags are checked in conditional control instructions. Many instructions set one or more of the flags.

The use of the EFLAGS register is implied (rather than explicit) in instructions.

Accessing Memory

There are 2 memory models supported in the Pentium architecture. (Actually it is the 486 and more recent models that support 2 models.)

In both models, memory is accessed using an address. It is the way that addresses are formed (within the processor) that differs in the 2 models.

FLAT MEMORY MODEL

-- The memory model that every one else uses.

SEGMENTED MEMORY MODEL

-- Different parts of a program are assumed to be in their own, set-aside portions of memory. These portions are called segments.

-- An address is formed from 2 pieces: a segment location and an offset within a segment.

Note that each of these pieces can be shorter (contain fewer bits) than a whole address. This is much of the reason that Intel chose this form of memory model for its earliest single-chip processors.

-- There are segments for:

code
data
stack
other

-- Which segment something is in can be implied by the memory access involved. An instruction fetch will always be looking in the code segment. An instruction to push data onto the stack always accesses the stack segment.

Addressing Modes

Some would say that the Intel architectures only support 1 addressing mode. It looks (something like) this:

```
effective address = base reg + (index reg x scaling factor) + displacement

   where
     base reg is EAX, EBX, ECX, EDX or ESP or EBP
     index reg is EDI or ESI
     scaling factor is 1, 2, 4, or 8
```

The syntax of using this (very general) addressing mode will vary from system to system. It depends on the preprocessor and the syntax accepted by the assembler.

For our implementation, an operand within an instruction that uses this addressing mode could look like

```
[EAX][EDI*2 + 80]
```

The effective address calculated with be the contents of register EDI multiplied times 2 added to the constant 80, added to the contents of register EAX.

There are extremely few times where a high-level language compiler can utilize such a complex addressing mode. It is much more likely that simplified versions of this mode will be used.

SOME ADDRESSING MODES

-- **register mode** -- The operand is in a register. The effective address is the register (wierd).

Example instruction:

```
mov  eax, ecx
```

Both operands use register mode. The contents of register ecx is copied to register eax.

-- **immediate mode** -- The operand is in the instruction. The effective address is within the instruction.

Example instruction:

```
mov  eax, 26
```

The second operand uses immediate mode. Within the instruction is the operand. It is copied to register eax.

-- **register direct mode** -- The effective address is in a register.

Example instruction:

```
mov  eax, [esp]
```

The second operand uses register direct mode. The contents of register esp is the effective address. The contents of memory at the effective address are copied into register eax.

-- **direct mode** -- The effective address is in the instruction.

Example instruction:

```
mov  eax, var_name
```

The second operand uses direct mode. The instruction contains the effective address. The contents of memory at the effective address are copied into register eax.

-- **base displacement mode** -- The effective address is the sum of a constant and the contents of a register.

Example instruction:

```
mov  eax, [esp + 4]
```

The second operand uses base displacement mode. The instruction contains a constant. That constant is added to the contents of register esp to form an effective address. The contents of memory at the effective address are copied into register eax.

-- **base-indexed mode** -- (Intel's name) The effective address is the sum of the contents of two registers.

Example instruction:

```
mov  eax, [esp][esi]
```

The contents of registers esp and esi are added to form an effective address. The contents of memory at the effective address are copied into register eax.

Note that there are restrictions on the combinations of registers that can be used in this addressing mode.

-- PC relative mode -- The effective address is the sum of the contents of the PC and a constant contained within the instruction.

Example instruction:

```
jmp  a_label
```

The contents of the program counter is added to an offset that is within the machine code for the instruction. The resulting sum is placed back into the program counter. Note that from the assembly language it is not clear that a PC relative addressing mode is used. It is the assembler that generates the offset to place in the instruction.

Instruction Set

Generalities:

-- Many (most?) of the instructions have exactly 2 operands. If there are 2 operands, then one of them will be required to use register mode, and the other will have no restrictions on its addressing mode.

-- There are most often ways of specifying the same instruction for 8-, 16-, or 32-bit oeprands. Note that on a 32-bit machine, with newly written code, the 16-bit form will never be used.

Meanings of the operand specifications:

```
reg - register mode operand, 32-bit register
reg8 - register mode operand, 8-bit register
r/m - general addressing mode, 32-bit
r/m8 - general addressing mode, 8-bit
immed - 32-bit immediate is in the instruction
immed8 - 8-bit immediate is in the instruction
m - symbol (label) in the instruction is the effective address
```

Data Movement

```
mov   reg, r/m                ; copy data
      r/m, reg
      reg, immed
      r/m, immed

movsx reg, r/m8               ; sign extend and copy data

movzx reg, r/m8               ; zero extend and copy data

lea   reg, m                  ; get effective address
   (A newer instruction, so its format is much restricted
    over the other ones.)

    EXAMPLES:

    mov EAX, 23  ; places 32-bit 2's complement immediate 23
                 ; into register EAX
    movsx ECX, AL  ; sign extends the 8-bit quantity in register
                   ; AL to 32 bits, and places it in ECX
    mov [esp], -1  ; places value -1 into memory, address given
                   ; by contents of esp
    lea EBX, loop_top ; put the address assigned (by the assembler)
                      ; to label loop_top into register EBX
```

Integer Arithmetic

```
add   reg, r/m                ; two's complement addition
      r/m, reg
      reg, immed
      r/m, immed

inc   reg                     ; add 1 to operand
      r/m

sub   reg, r/m                ; two's complement subtraction
      r/m, reg
      reg, immed
      r/m, immed

dec   reg                     ; subtract 1 from operand
      r/m

neg   r/m                     ; get additive inverse of operand

mul   eax, r/m                ; unsigned multiplication
                              ; edx||eax <- eax * r/m

imul   r/m                    ; 2's comp. multiplication
                              ; edx||eax <- eax * r/m
       reg, r/m               ; reg <- reg * r/m
       reg, immed             ; reg <- reg * immed
```

```
div   r/m                    ; unsigned division
                             ; does edx||eax / r/m
                             ; eax <- quotient
                             ; edx <- remainder

idiv   r/m                   ; 2's complement division
                             ; does edx||eax / r/m
                             ; eax <- quotient
                             ; edx <- remainder

cmp   reg, r/m               ; sets EFLAGS based on
      r/m, immed             ; second operand - first operand
      r/m8, immed8
      r/m, immed8            ; sign extends immed8 before subtract


   EXAMPLES:

   neg [eax + 4]    ; takes doubleword at address eax+4
                    ;   and finds its additive inverse, then places
                    ;   the additive inverse back at that address
                    ;   the instruction should probably be
                    ;      neg  dword ptr [eax + 4]

   inc ecx          ; adds one to contents of register ecx, and
                    ;   result goes back to ecx
```

Logical

```
not   r/m                    ; logical not

and   reg, r/m               ; logical and
      reg8, r/m8
      r/m, reg
      r/m8, reg8
      r/m, immed
      r/m8, immed8

or    reg, r/m               ; logical or
      reg8, r/m8
      r/m, reg
      r/m8, reg8
      r/m, immed
      r/m8, immed8

xor   reg, r/m               ; logical exclusive or
      reg8, r/m8
      r/m, reg
      r/m8, reg8
      r/m, immed
      r/m8, immed8

test  r/m, reg               ; logical and to set EFLAGS
      r/m8, reg8
      r/m, immed
      r/m8, immed8


   EXAMPLES:

   and edx, 00330000h   ; logical and of contents of register
                        ;   edx (bitwise) with 0x00330000,
                        ;   result goes back to edx
```

Floating Point Arithmetic

Since the newer architectures have room for floating point hardware on chip, Intel defined a simple-to-implement extension to the architecture to do floating point arithmetic. In their usual zeal, they have included MANY instructions to do floating point operations.

The mechanism is simple. A set of 8 registers are organized and maintained (by hardware) as a stack of floating point values. ST refers to the stack top. ST(1) refers to the register within the stack that is next to ST. ST and ST(0) are synonyms.

There are separate instructions to test and compare the values of floating point variables.

```
finit                          ; initialize the FPU

fld   m32                      ; load floating point value
      m64
      ST(i)

fldz                           ; load floating point value 0.0

fst   m32                      ; store floating point value
      m64
      ST(i)

fstp  m32                      ; store floating point value
      m64                      ;   and pop ST
      ST(i)

fadd  m32                      ; floating point addition
      m64
      ST, ST(i)
      ST(i), ST

faddp ST(i), ST                ; floating point addition
                               ;   and pop ST
```

Control Instructions

All conditional control instructions in the Intel architectures are called jumps. Their machine code is similar to the MIPS branch instructions.

Just some of the many control instructions:

```
jmp   m              ; unconditional jump
jg    m              ; jump if greater than 0
jge   m              ; jump if greater than or equal to 0
jl    m              ; jump if less than 0
jle   m              ; jump if less than or equal to 0
```

Note that a control instruction takes a single operand, which specifies the jump target. The conditional control instructions look at the condition code bits (in the EFLAGS register) to make a decision on whether to take the jump or not.

The condition code bits are set by separate instructions. Several arithmetic and logical instructions set some of the condition code bits. There are also specific instructions to compare operands and set the condition code bits based on the comparison (examples: cmp, test).

Some sample code, for fun:

Pentium code to add 1 to each element of an array of integers.

Assume that there is an array of 100 integers in memory. The label associated with the first element is int_array.

Comments are placed to the right, and preceded by a semicolon (;).

```
      lea EAX, int_array      ; like la in MIPS, EAX is pointer
      mov ECX, 100            ; register ECX contains counter
loop_top:
      cmp ECX, 0              ; must set condition codes
      je  all_done            ; uses condition codes to branch
      inc [EAX]               ; a register direct addressing mode!
      add EAX, 4              ; updates pointer
      dec ECX                 ; update counter
      jmp loop_top            ; unconditional branch to loop_top

all_done:
```

Some things to notice about this code:

-- You can figure it out, although you only know MIPS assembly language! That is because most assembly languages look similar.

-- The 2-address instruction set does not generate a larger number of instructions for this example (than a 3-address instruction set would, like MIPS). It does do the same number of memory accesses.

Intel MMX (Optional)

MultiMedia eXtension to Intel Arch.
[source: Peleg & Weiser, IEEE Micro, Aug. 96]

Motivation

Q: Why might people want to buy newer, faster PCs?
A: Processing audio and video

Let's make audio and video perform better

Method 1: add special-purpose card
Method 2: make regular microprocessor perform better at audio/video

Intel's MMX follows Method 2
The goal is 2x performance in audio, video, etc.

Key observation: precision of data required << 32 bits

For video,
Red/Green/Blue might use 8 (16) bits each for 256 (64K) colors per pixel (picture element)

Key technique: pack multiple low-precision items into a 64-bit floating-point register add instructions to manipulate them

(This is an example of a general technique called "single instruction multiple data", or SIMD)

MMX Datatypes ----------

```
    * 1 x 64 bit quad word
    * 2 x 32 bit double-word
    * 4 x 16 bit word
    * 8 x 8  bit byte
```

MMX Instructions

```
Example, ADDB (B stands for byte)

      17   87  100 ... 5 more
    + 17   13  200 ... 5 more
    ---- ---- ---- ...
      34  100   44 = 300 mod 256 ==> wraparound
             255 = max value ==> saturating
```

This can be used to do arithmetic/logical operations on more than 1 pixel's worth of data in 1 instruction.

Also MOV's == load & stores

Example:

```
16 element dot product (from matrix multiply)

[a1 a2 ... a16]* [b1 b2 ... b16]^T = a1*b1 + b2*b2 + ... + a16*b16

comparision with Intel IA-32 gives:

->  32 loads
->  16 *
->  15 +
->  12 loop ctrl
   ---
    76 instructions
    int ==> 200 cycles
    fp  ==> 76 cycles

Intel MMX assuming 16b values

->  16 instructions
->  12 cycles (6x better than fp)
```

Other Instructions

PACK/UNPACK -- putting multiple values in single register & back

MASK

```
   Example,  "make 0xff if equal"

        15   15 100  120  101   76   15   15
        15   15  15   15   15   15   15   15
       ------------------------------------
        FF   FF  00   00   00   00   FF   FF
```

Why? Mask for weatherman!

- * film weatherperson in front of blue background (0x15)
- * wthmsk = use above mask instruction

```
                        wthmsk==FF -- no weatherperson
                        wthmsk==00 -- weatherperson

                image = (~wthmsk & weatherperson ) | (wthmsk & weathermap)
```

(What happens if weatherperson wears suit of color 15?)

MMX Constraints

- Instruction Set Architecture extensions, but perfect backward compatibility
- 100% Operating System compatible (no new registers, flags, exceptions)
- Independent Software Vendor (ISV) support (bit in CPUID instruction so applications can test for MMX and include code for both)

IA-64/Merced (Optional)

Motivation

- IA-32 has 32-bit addresses
- 2^32 ==> 4G bytes of memory Current large servers want more!
- Near future medium servers will want more ... Someday desktops will want more?

What to do?

1. Kludge IA-32 to support > 32-bit addresses
2. Do new instruction set with binary compatibility strategy
   (a) have new chips also support IA-32
   (b) use binary translation, etc.

Intel claims to be doing 2a, but has only partially revealed plans. (as of Nov '98)

New instruction set architecture: IA-64

- 64-bit addresses
- Mode for running old code
- First implementation is called "Merced"
- Has extra large instructions so that 128b = 4 * 32b holds

```
instrn0 instrn1 instrn2 template
```

  "template" gives "relationships" between instuctions.
  Example: whether instrn1 shares no registers or memory locations w/ instrn0
- Uses "templates" and "predication".
  Each instruction is "predicated"
  Example,

```
if $1 < $2
then
    $3 = $4
else
    $5 = $6
endif
```

  Is normally:

```
      bge $1, $2, else
      mov $3, $4
      b endif
else: mov $5, $6
endif:
```

  With predication:

```
setlt p0, $1, $2
if (p0==TRUE) mov $3, $4
if (p0==FALSE) mov $5, $6  /* three instructions & no branches */
```

Aren't you glad we did not teach 354 with IA-64/Merced?

Copyright © Karen Miller, 2006