Microprocessors

The ia32 User Instruction Set

Jan 31st, 2002
User Instruction Set

This is the instructions that can be used by applications programs

Operating in protection ring 3

The most restricted instruction set
Addressing

Addresses are specified by a

- Segment number, where the segment dictates the starting address and length of the segment.
- A 32-bit offset within the segment

In user mode, a program cannot change what segments are allocated or accessible.

Segments can be marked read-only, or execute-only
Addressing, continued

Usually, an program operates with DS and SS set to a standard data segment and CS set to a standard code segment.

Segment registers are never modified or explicitly referenced.

All addressing then behaves like a simple 32-bit linear address, where code addresses and data addresses are separate (cannot modify code).
Addressing, continued

Addresses (offsets) are 32-bits and fit in 32-bit register, segments are ignored, so we can think of these offsets as simply addresses.

In remainder of these slides we will use that convention

More on segmentation when we talk about the system level instruction set
Basic model of instructions

Instructions typically operate

- Register to register
- Register to memory
- Memory to register
- But not memory to memory
- There are some memory/memory instructions
The MOV instruction

Moves data

- From register to register
- From register to memory
- From memory to register

It thus encompasses load and store

The memory address if given allows the full set of addressing modes
Addressing modes

Direct memory (32-bit address in instruction)
Simple indexing
Indexing + offset (8 or 32 bits)
Double indexing
Double indexing + offset (8 or 32 bits)
Double indexing with scaling
Double indexing with scaling and offset
Scaling multiplies one index reg by 2,4,8
The ADD instruction

ADD can operate

- \( \text{reg1} := \text{reg1} + \text{reg2} \)
- \( \text{mem} := \text{mem} + \text{reg2} \)
- \( \text{reg1} := \text{reg1} + \text{mem} \)

Full range of addressing modes if mem

Same instruction for signed/unsigned

- Sets flags as a result

Can be used on any of the 8 registers
Flag Registers

CF, set if carry out from MSB
ZF, set if result is zero
OF, set if signed overflow
  Carry in to sign /= Carry out of sign
SF, set if result sign flag is 1
Other special purpose flags
Other arithmetic and logic

SUB – subtract, like ADD
ADC/SBB – input the CF at low order
- Used for multiple precision
OR, AND, XOR
CMP (like SUB but no result set)
- Just flags
TEST (like AND, but on result set)
- Just flags
The flags after a CMP

CMP is basically subtract

CF set if opnd2 > opnd1 unsigned

ZF set if operands equal

OF set if subtraction caused overflow

SF set if result sign bit is set

Combinations of these flags provide testing of the entire range of signed and unsigned comparison results
An example of test after CMP

Consider comparison of two signed values.
If the subtraction does not overflow, then the result is negative if \( \text{opnd2} > \text{opnd1} \).
But if the result overflow, the sign is wrong.
Therefore condition for \( \text{opnd2} > \text{opnd1} \) is:
\[ SF \text{ xor} \ OF \]
Full set of shifts

Logical (shift in zero bits)
Arithmetic (shift in sign bits)
Double shift (uses EAX/EDX)
Operand can be 1, or value in ECX
Can shift register or memory
8-Bit operands

All arithmetic and logical operations can operate on the 8-bit registers AL, BL, CL, DL, AH, BH, CH, DH

This is mostly for back compatibility but can be useful, e.g. for 8-bit arithmetic with carry/overflow detection

Instructions for sign/zero extending 8-bit to 32-bits (these are generally useful)
16-bit operands

For further back compatibility, if a special 16-bit operand prefix precedes the instruction, then the instruction is interpreted in old 8086 style, and can operate on AX,BX,CX,DX,SP,BP,SI,DI. This is the low order 16-bits of each register.
Alignment Requirements

Data does not have to be aligned, it can be on any byte boundary.

But machine operates more efficiently if data is naturally aligned

- E.g. 32-bit integer on four byte boundary
- Which means last 2 bits of address are 00
Jump Instructions

The unconditional jump instruction can jump anywhere in the 32-bit code segment (full 32-bit offset in instruction)

There is also a 7 byte jump instruction that includes a segment number

But we don’t often need to change code segments for a jump, so seldom used.
Branch Instructions

The conditional branch instructions allow an 8-bit offset (-128 to +127 bytes)

Can always combine with jump

Instead of BNE target which is out of range

Use

BE next JMP target next:

Assembler can take care of this

Idea is to keep branch instructions small
Branch Conditions

Full set of branch conditions

Direct test of flag

\[ \text{JO target} - \text{jumps if OF flag is set} \]

Combinations of flags for comparisons

\[ \text{JL target} - \text{jumps if opnd1} < \text{opnd2 signed} \]

\[ (\text{corresponds to jump if OF xor SF}) \]

\[ \text{Full set of conditions for signed/unsigned} \]
ESP is used as a hardware stack pointer
PUSH and POP instructions push data onto this stack, or POP data off, from/to register or memory
Used for temp storage, saving of registers and passing of parameters to procedures
Stack builds down (PUSH decrements ESP)
CALL Instruction

Most usual form is 5 bytes, opcode plus 32-bit offset into code segment.

- Return point automatically pushed onto stack
- RET instruction returns via pushed return

Can also give segment number for intersegment calling

- Can be used to call system routines (more on that when we discuss system instructions)
The Calling Sequence

Arguments are pushed on to stack
- Series of push instructions one for each arg

Procedure establishes stack frame using
- PUSH EBP (save old frame pointer)
- MOV EBP,ESP (set new frame pointer)
- SUB ESP, 44 (establish local frame)
What a Stack Frame Looks Like

High memory at top of slide
  ⚫ first pushed parameter
  ...
  ⚫ last pushed parameter
EBP ⚫ saved EBP (old frame pointer)
  ⚫ first variable
  ...
ESP ⚫ last local variable
Address params with positive offsets from EBP
Address locals with negative offsets from EBP
Returning from a Procedure

Remove stack frame
- MOV ESP,EBP

Restore EBP
- POP EBP
- Above two instructions can be given as a single instruction LEAVE

Return
- RET if caller strips parameters
- RET 8 to strip 8 bytes after return
More on calling sequences

STDCALL

This is the PASCAL calling sequence, parameters are pushed from left to right and the caller is expected to pop the parameters.

C

In the C calling sequence, parameters are pushed from right to left, and the caller does NOT pop the parameters (allows variable number of parameters).