Machine and Assembly Language Principles

Instruction Categories

- Data transfer instructions
- Integer arithmetic and logic instructions
- Shift and rotate instructions
- Control transfer instructions
- Bit manipulation instructions
- System control instructions
- Floating point instructions
- Special function unit instructions

Data transfer instructions

Types of transfer:

- register to register
- register to or from memory
- memory to memory

Multiple addressing modes available for memory

Size of operands usually 1, 2, 4, or 8 bytes,

Some machines allow moving MULTIPLE data items in one instruction.

- Exchange instructions
- Push/pop multiple operands
- String copy and other operations
- Special move instructions

Multiple Data Transfers

Exchange - one instruction to exchange 2 operands

Example (80X86): XCHG src1, src2

Otherwise would require 3 instructions

In some machines this operation is ATOMIC

Move multiple - move to or from several registers. Examples:

- (80486): PUSHA, POPA - to push or pop all general-purpose registers
- (680X0): MOVEM ea, list of registers (implemented as a mask in machine code)

Saves several instructions on saving sets of registers

String operations and special moves

(nearly) UNLIMITED data size.
Uses registers for source and destination pointers
Uses count registers
Examples:

- (80X86): (REP) MOVS
- (80X86): CMPS
- (VAX): Move string

Conditional termination possible (REPNZ in 80486)

Translate while moving.

- (80X86): XLAT ; AL gets [EBX+AL]
- (VAX): translate string using table

Move to/from control registers

Arithmetic, logical, and shift

Seen in exercise sessions - how to **extend**?

For example, need to implement long, or even **unlimited** precision integer operations.

For ADD, SUBTRACT, SHIFT - use carry flag , or extend flag

Examples (80X86):

```
ADD     EAX, [X_LOW]
ADC     EBX, [X_MID]
ADC     ECX, [X_HIGH]

SHL     dword [X_LOW], 1
RLC     dword [X_HIGH], 1
```

(MOTOROLA 68000)

```
ADD.L   X_LOW, D0
ADDX.L  X_HIGH, D1
```

Bit manipulation instructions

Flag manipulation

Flags change as a result of ALU operations
Sometimes can be manipulated directly

Examples (80X86)

- Carry flag: STC, CLC, CMC
- Direction flag: CLD, STD

Instructions to move to/from flag register:
(80X86): LAHF, SAHF

Push and pop flag register:
(80X86): PUSHF, POPF

Used to save state, but can be used to manipulate flags:

```
LAHF
AND     AH, mask
SAHF
```

Bit and Bit Field Manipulation

Individual bit Manipulation - (80X86):

```
BT      dst, bit  ; Test bit
BTC     dst, bit  ; Test and complement
BTS     dst, bit  ; Test and set
BTR     dst, bit  ; Test and reset
```

Bit string operations: scan bits (80X86)

```
BSF     dst, src  ; Bit scan forward (find first 1)
BSR     dst, src  ; Bit scan reverse
```

Bit field operations: compressed structs.
Examples (MOTOROLA 680X0):

```
BFCLR, (ea)o:w            ; Clear bit field
BFEXTS (ea)o:w, Dn      ; Sign extend
BFFFO  (ea)o:w, Dn      ; Find first 1
BFINS  Dn, (ea)o:w        ; Insert to field
```

Also has: BFCHG (invert), BFEXTU (zero extend), BFSET, BFTST

Program flow control

Branching addressing modes:
Relative, absolute, indirect, others

Jump and branch: JMP, BRA

Conditional jump and branch:
Jccc (80X86)

Other conditional operations:
SETcccc (80X86)

Condition types (examples - 80X86):

- Arithmeric: Z (E), NZ (NE), GT, LT, LE, LT, S, NS, O, NO
- Other: PE, PO, CXZ

Hybrids: do operation, test and conditional jump.
Examples (80X86): LOOP, LOOPZ, LOOPNZ
(680X0): DBcc Dn, label (decrement, branch on cc)

Support for precodures

Procedure called from more than one spot, need to save return address. Where?

Scheme in old machines (CDC): word prior to procedure
Advantages: simple, allows multiple levels
Disadvantages: no recursion!
Most machines use a **stack**.

Call - push return address (IP) on stack and jump Examples:

- (80X86): CALL addr ; Push IP and jump
- (VAX): calls addr, mask ; Push PC and registers, jump

Return - pop return address form stack
Examples:

- (80X86): RET ; Pop IP
- (80X86): RET cnt ; also add cnt to SP
- (VAX): rets ; restore registers and PC

Input - Output Operations

From programmer's perspective: read or write IO device registers - by read or write of **IO ports**.

Examples (Intel 80X86):

```
OUT   port-number, AL ; (or AX, or EAX)
IN    AL, port-number
```

Device registers usually in **IO address space**. This means:

- Address bus emits address in IO space
- Control lines signify IO read or write

Hardware - usually at least total of 4 registers at 2 addresses:

- Address for data in/out registers
- Address for control and status

Example - serial IO port.

Memory Mapped IO

IO can share address/data busses or use special IO bus.

Likewise, can use special ID address space, or share with the memory address space.

Some "memory address space" reserved for IO.

Device registers activated by "memory access", with decoders enabled by memory access control signals.

From programmer's perspective, access to memory mapped IO looks just like move to/from memory!

Caveat: but this is NOT exactly like memory access.
For example, what you write to an address is NOT necessarily what you get when you read the address!

Machine control and interrupts

Remember that in a real machine, especially servers:

- Multiple IO devices
- Multi process **and** multi user
- Sometimes also multi **processor**

How to achieve the above? Hardware must support:

- Protection and security support
- Interrupts: internal and external
- Atomic operations and bus locking
- Special control operations

Protection and security

Much of this issue is handled by the **operating system**.

Some hardware support essential:

- Memory block tagging or even mapping
- At least 2 machine modes, one for operating system, one for user.

More on memory protection and mapping - in operating systems course.

Machine mode: determined by 1 or more flag bits in CPU.

In **user mode** many instructions **inaccessible**:

- Cannot just change mode flag!
- Limits access to memory and IO

Example (Motorola 680X0):
S=0 is user mode, S=1 is **supervisor** mode.

Interrupts - internal and external

Internal interrupts:

- By using a deliberate, legal, instruction.

```
INT    n    ; Intel 80X86
TRAP   n    ; Motorola 680X0
```

- Generated internally by CPU due to **exceptions** (faults). Usually causes program termination.
  - Arithmetic error: divide by 0, floating point overflow
  - Memory related: bus error, alignment error, segmentation violation
  - Illegal istruction
  - Protection violation

Deliberate (software) interrupts (also called **traps**, supervisor calls).
Controlled access to privileged parts of machine, via **system calls**.

Interrupts (cont)

External interrupts: external signal.

Used by IO devices and internal controller. Usually does **not** cause program termination. Can occur at *any time*.

- IO devices: Keyboard character available. Disk seek complete. Input buffer full.
- Internal controller: Page fault. Timer interrupt.

All interrupts behave like a special **call**:

1. Save state (IP, PSW)
2. Change machine mode to supervisor
3. Jump to address according to **interrupt vector**

Code at interrupt vector: **interrupt handler**

To return - use special instruction (IRET in 80X86).

Atomic operations

Data in memory shared between processes may need to be accessed.

Interrupts can occur at **any time** - may cause race conditions.

One possibility - disable interrupts. But:

- Not available in user mode
- Not all interrupts can be disabled (NMI)
- Does not work in multi processor systems

Solution: read-modify-write bus-locked instructions.

- TAS (test and set in Motorola 680X0)
- LOCK XADD (exchange and add in Intel 80X86)

When executed, cause asserting a (bus) LOCK control signal.

Special control operations

Breakpoint and other debug control

System timer control

Floating point processor control

Co-processor control

Interrupt controller operations

DMA control

Memory management unit control

Cache control