Real machine architectures

part 1: general overview

Machine language instructions

• Consist of:
  – op codes: specify operation to be executed
  – operands: specify data (value or location)

• Addressing modes:
  – Direct (001 in Pep/8): value of operand is address
    for data storage (HLL version: \( x = y \))
  – Immediate (000 in Pep/8): value of operand is
    data (HLL version: \( x = 14 \))

Instruction formats

• Architectures are differentiated by:
  – number of bits per instruction
  – number of operands per instruction (0, 1, 2 or 3
    are most common)
  – types of operations (in terms of both function and
    memory access
  – types of data instruction can process
  – how operands are stored in CPU (stack vs.
    registers)

Instruction types, continued

• Operand location
  – register to register
  – register to memory
  – memory to register
  – memory to memory

Design decisions for instruction sets

• Format of instructions must be decided before
  many other decisions can be made
• The instruction set must match the
  architecture
• The architecture could last for decades

ISAs measured by:

• Amount of space program requires
• Complexity of instruction set
  – amount of decoding necessary to execute
    instruction
  – complexity of task each instruction performs
  – instruction length / # of instructions
### Design considerations

- **Short instructions**
  - take up less memory space
  - can be fetched / decoded quickly
  - less bits per instruction means
    - less total instructions possible
    - limited size and number of operands

- **Fixed-length instructions**
  - Easy to decode
  - Waste space
- Memory organization affects instruction format
  - (to access ASCII character data, you need byte-addressable memory, even if word size is 16, 32 or 64 bit)
- Fixed-length doesn’t necessarily mean fixed number of operands (allowing operand field to vary is called expanding op codes)

- **Many different addressing modes**
  - Pep/8 has 8, with 3 bits allotted for specification
  - Pep/7 had 4 – used just 2 bits, with a 5-bit op code

- **Registers vary by:**
  - number
  - organization
  - special vs. general purpose

- **Byte-level organization**: in multiple-byte words, how should bytes be stored in byte-addressable machine: little-endian vs. big-endian
  - “Endian” refers to byte order: which byte in the word has the MSB, which has the LSB
  - most UNIX machines big-endian, most PCs little-endian

- **Hardware smackdown: big-endian vs. little-endian**
  - Intel: little endian
  - Motorola: big endian
  - Illustration: 4-byte integer
    ```plaintext
    byte 3 | byte 2 | byte 1 | byte 0
    -- if machine is LE, base address + 0 = byte 0
    -- if machine is BE, base address + 0 = byte 3
    ```
Hardware smackdown: big-endian vs. little-endian

- **BE advantages:**
  - more natural to people; easier to read hex dumps
  - with MSB first, can test bit address(0) to determine sign of stored number
  - ints & strings are stored consistently
  - most bit-mapped graphics are mapped with MSB as leftmost bit
- **BE disadvantages:**
  - conversion from 32-bit int address to 16-bit int requires addition
  - can’t write words on non-word boundaries

Big-Endian vs. Little-Endian: quick recap

- In a big-endian machine, bytes used to store a data item are arranged left to right, so that the MSB is found at the leftmost position (first byte of address, the “big end”)
- Little-endian is just the opposite; bytes are arranged right to left, with the MSB as the first bit of the last byte (the “little end”)
- Note that, in either case, **bits within each byte** are arranged left to right – so a little-endian integer isn’t exactly the same thing as a big-endian integer backwards

Hardware smackdown: big-endian vs. little-endian

- **LE advantages:**
  - high-precision arithmetic is faster & easier
  - allows for read/writes at odd addresses
- **LE disadvantages:**
  - to determine sign of stored number, need to know length of number, skip over bytes to find one with sign bit
  - have to reverse byte order when working with graphical objects

Byte ordering & data movement

- Computer networks are big endian:
  - Little endian machines must convert integers (e.g. network device addresses) before they can be passed over the network
  - Little endian machines must also convert integers retrieved from the network to the native mode for the machine

Byte ordering & data movement

- Any program that reads/writes file data must be aware of byte ordering
  - For example, Windows BMPs were developed on a little endian machine; an application on a big endian machine that reads a BMP must reverse byte order
  - PhotoShop, JPEG, MacPaint, Sun raster files: big endian
  - GIF, PC Paintbrush, RTF: little endian

Internal CPU storage

- 3 choices for data storage in CPU:
  - Stack architecture:
    - Use stack to execute instructions; operands stored at top of stack
    - No random access
  - Accumulator architecture:
    - Minimum of internal complexity; short instructions
    - One (implicit) operand stored in accumulator
    - Involves high volume of memory traffic
  - General Purpose register: see next slide
General Purpose Register (GPR)

- Set (>1) of GPRs
- Most common architecture in use today
- Registers are faster than memory; easier for other parts of the CPU to handle register data (than data from memory)
- Cheaper hardware tends to mean an increased number of registers in the CPU
- GPRs mean longer instructions, because register(s) must be specified; takes more time to fetch/decode longer instructions

Classification of GPR architectures

- Memory to memory (VAX):
  - Instruction uses 2-3 operands, stored in memory
  - Instructions can perform operations without involving registers
- Register to memory (Intel, Motorola): at least one operand must be in a register
- Load-store (SPARC, MIPS, Alpha, PowerPC): Requires movement of data to registers before any operations performed

Operand number / instruction length

- Instructions can be formatted 2 ways:
  - Fixed-length: fast, but wastes space
  - Variable-length: more complex to decode, but saves space
- Real-life compromise often involves 2-3 instruction lengths (so fixed, but variable)

Von Neumann machines

- Computers that store program instructions as well as data (stored program is the Big Idea)
- Data & instructions share memory space
- All modern computers fit this description

Program loading

- Instructions are placed in contiguous memory; address of second instruction depends on length of first
- PC (or IP) and stack pointer (SP) are initialized when program is loaded
  - PC: holds address of next instruction (or start address if we’re at the beginning)
  - SP: hold address of stack top (in Pep/8 SP is set to 7FF8 – stack grows up)

Fetch/execute cycle

- Stored program executes in a continuous loop called the fetch/execute cycle
- Steps:
  - fetch instruction
  - decode instruction
  - increment program counter (PC) or instruction pointer (IP)
  - execute instruction
  - repeat
Fetch instruction

- Program is loaded in memory
- Start address is stored in a register
- Start address value is used to calculate offset when PC (or IP) is incremented
  - CPU interprets PC value as address
  - In Pep/8, CPU grabs op code & stores in first byte in instruction register (IR)

Decode

- Extract op code
- Extract operand specifier(s)
- Extract addressing mode
- In Pep/8, if operation is not unary, CPU fetches next word (remaining 16 bits of instruction) & stores operand specifier in the last 2 bytes of the IR

Increment / Execute

- Increment adds appropriate offset value to PC
- Execute:
  - CPU performs instruction stored in IR
  - Op code specifies what instruction to execute

Repeat

- CPU returns to fetch operation unless the last instruction performed was a STOP
- Program termination may also occur if illegal operation is attempted – most commonly this involves a forbidden addressing mode

Von Neumann bottleneck

- Total number of memory fetches required can be the main determining factor in a program’s speed
- Memory fetches are usually the slowest part of an operation
- Caching & pipelining – strategies for resolving the bottleneck

Errors & their causes

- Error messages (runtime, not compiler) result when machine can’t perform a step in the fetch/execute cycle
- The next two slides describe two of the most common runtime errors
Addressing error

- Fetch step can’t access address specified by operand; possible reasons:
  - address doesn’t exist
  - hardware error at address
  - incrementing PC caused overflow

Illegal instruction error

- Decode step wasn’t successful; possible reasons:
  - illegal or meaningless op code
  - often result of data being interpreted as instruction

Von Neumann characteristics

- Where data & instructions share memory, one can be misinterpreted as the other – hence, illegal instruction
- But the architecture carries a significant advantage: one program can be viewed as data for another program
  - Makes high level languages possible (compilers, interpreters, even assemblers)
  - Makes operating systems possible