Instruction Sets

Tuesday, January 3, 2017 11:35 AM

Lectures 4 - 6:

Computer architecture and computer programmer don't have much overlap, except the machine instruction set.

- From designer POV:
  - Machine instruction set provides functional requirements for the processor
    - Implementing processor is a task that involves implementing machine instruction set
- Programmer POV:
  - if assembly language, user is aware of register and memory structure
  - Function of ALU
  - Types of data directly supported by the machine

Machine instruction: the operation of the processor is determined by the instructions it executes.
Machine instruction set: the collection of different instructions the processor can execute.

Elements of a Machine Instruction:

1. Operation code: specifies the operation to be performed (ADD, I/O)
   a. This operation is specified by a binary code, called **opcode** (operation code)
2. Source operand reference: operands that are inputs for the operation.
3. Result operand reference: operation may produce result
4. Next instruction reference: tells the processor where to fetch the next instruction after the execution of current instruction is complete.
   a. Can be virtual or real address, depending on architecture.
   b. Typically no reference to next instruction. When explicit reference is needed, memory address must be supplied

- *Source and Result operands*
  - Main or virtual memory: memory address must be supplied
  - Processor register: processor contains one or more registers that may be referenced by machine instructions, where instruction contains number of desired register
  - Immediate: the value of the operand is contained in a field in the instruction being executed
  - I/O device: the instruction must specify the I/O module and device for the operation.
    - If memory-mapped, then main or virtual memory address

Instruction cycle:

1. **Fetch the instruction**: The next instruction is fetched from the memory address that is currently stored in the program counter (PC), and stored in the instruction register (IR). At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.
2. **Decode the instruction**: During this cycle the encoded instruction present in the IR (instruction register) is interpreted by the decoder.
3. **Evaluate effective address**: In case of a memory instruction (direct or indirect) the execution phase will be in the next clock pulse. If the instruction has an indirect address, the effective address is read from main memory, and any required data is fetched from main memory to be processed and then placed into data registers (Clock Pulse: T_3). If the instruction is direct, nothing is done at this clock pulse. If this is an I/O instruction or a Register instruction, the operation is performed (executed) at clock Pulse.
4. **Fetch operands:** read memory data
5. **Execute the instruction**: The control unit of the CPU passes the decoded information as a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction such as reading values from registers, passing them to the ALU to perform mathematical or logic functions on them, and writing the result back to a register. If the ALU is involved, it sends a condition signal back to the CU. The result generated by the operation is stored in the main memory, or sent to an output device. Based on the condition of any feedback from the ALU, Program Counter may be updated to a different address from which the next instruction will be fetched.
6. **Store result:** write back memory data

Instruction representation
Similar to a packet in networks, an instruction is represented by a sequence of bits.

4 bits	6 bits	6 bits
Opcode	Operand reference	Operand reference

16 bits total - simple instruction format

Binary representations of machine instructions are difficult, so a symbolic representation of machine instructions is used. This is NOT assembly (though it looks like it)

Opcodes are represented by abbreviations called mnemonics that indicate operation.

Examples:

ADD	Add
SUB	Subtract
MUL	multiply
DIV	divide
LOAD	Load data from memory
STOR	Store data to memory

Operands can also be represented symbolically.

ADD R, Y -> add the value contained in data location Y to contents or register R

*Y is an address of a location in memory. Operation is performed, however, on content of location, and not address

Each symbolic opcode has a FIXED binary representation. Therefore, when writing code a simple program would:

- Accept symbolic input
- Convert opcodes and operand references to binary
- Construct binary machine instructions

High level language is clear and concise and expresses operations in algebraic form, using variables Machine language, however, expresses operations in a basic form involving the movement of data to and from registers.

A simple instruction such as X = X + Y, would require 3 instructions in machine language:

1. Load a register with contents of memory location stored in X
2. Add contents of memory location stored in Y to that register
3. Store contents of the register in memory location X

Have to be able to translate high-level language into machine language.
So, we can categorize instruction types based on the 4 basic functions of computing:

- Data processing: arithmetic and logic instructions
- Data storage: movement of data into our out of register and/or memory locations
- Data movement: I/O instructions
- Control: test and branch instructions
    - Branch instructions are used to branch to a different set of instructions depending on the decisions made
    - Test instructions are used to test the value of a data word or the status of a computation

Number of Addresses
typical way to describe processor architecture in by number of addresses contained in each instruction

What addresses would we need in an instruction?

- 1 source operand (unary) or 2 source operands (binary)
- Destination operand address
- next instruction address (can be implicitly obtained from program counter)

Most architectures have either one, two, or 3 operand address, with next instruction being implied

If we wanted to compute $Y = \frac{A - B}{C + (D \cdot \mathrm{E})}$ it would depend entirely on the type of architecture

***may need to use *T* as a temporary storage for intermediate results in 3-address instructions
not common because of length of instruction format
**one address must do double duty when using 2-address instructions (both operand and result)
This reduces space requirement. Have to add different instructions (like MOVE) to not alter operand
*second address must be implicit. Uses processor register known as accumulator (AC). This register contains operand and is used to store result
@technically, you can do this with 0 registers by using a stack.

Number of address	Symbolic Representation	Interpretation
3	OP A, B, C	A <- B OP C
2	OP A, B	A <- A OP B
1	OP A	AC <- AC OP A
0	OP	T <- (T-1) OP T

Pros and Cons of address per instruction:
Fewer address:

PRO

- More primitive, so less complex processor
- Instructions are shorter length

CON

- Programs contain more total instructions, leads to longer and complex programs
- Execution time is longer
- Only has one general-purpose register available, the accumulator

Multiple address instructions:

PRO

- Multiple general purpose registers
- Faster execution (because register reference is faster than memory reference)

CON

- More complex design

Truthfully, most machines employ a mixture of two- and three- address instructions. Machines also offer variety of instruction formats

Instruction Set Design
Integral part of computer design. Instruction set itself defines many of the functions performed by the processor and has a significant effect on the implementation of the processor.
--The instruction set is the programmer's means of controlling the processor

Fundamental design issues when designing an instruction set:

- Operation repertoire: How many and which operations to provide, and how complex operations should be
- Data types: the various types of data upon which operations are performed
- Instruction format: instruction length (in bets), number of addresses, size of various fields, etc.
- Registers: Number of processor registers that can be referenced by instructions and their use
- Addressing: the mode(s) by which address of an operand is specified

TYPES OF OPERANDS

Most important general categories of data are **Addresses**, **Numbers**, **Characters**, and **logical data**

Numbers
Numeric data types include: binary integer or binary fixed point, binary floating point, and decimal.
Binary has been discussed already, so let's discuss decimal.
Typically, user will represent decimal in ***packed decimal*** or BCD (binary coded decimal).
BCD encodes decimal digit into unique 4-bit sequence. Packed decimal stores BCD-encoded digits using one byte for each two digits. To form numbers, usually strung together in multiples of 8.

Can also represent sign bits using packed decimal where
1100 = positive and **1101 = negative.**

Characters

Text or character strings. Computers cannot easily transmit this in character form. So characters must be represented as a sequence of bits. Early example includes Morse code
Most common code today is International Reference alphabet (IRA) or as we like to call it, American Standard Code for Information Exchange (ASCII). Each character represented by unique 7-bit pattern. Some of these are control characters (for printing, communication, etc.)

Even though 7-bit code, usually transmitted using 8 bits per character, with eighth bit set to 0 or used as a parity bit for error detection.

Logical Data

Logical data: *n*-bit unit as consisting of *n* 1-bit items of data, each having the value 0 or 1.
We have 2 advantages: wish to store array of Boolean or binary data items where each item can only take on true (1) and false (0); wish to manipulate the bits of a data item (for instance, shifting in floating-point operations)

INTEL x86 AND ARM DATA TYPES

What is Intel x86?
Has a complex instruction set computer (CICS) architecture. Made famous by Intel in 1978 with their 16-bit 8086 processor. Subsequent processors were built from this foundation, which is why it's called x86 (other processors ended in 86 as well). The ISA of x86 has evolved to remain backwards compatible. There are now thousands of instructions in the instruction set, as no instructions are subtracted. Only technology and organization has changed over years. Uses little Endian--least significant byte is stored in the lowest address (from right-to-left order similar to arithmetic)

*Big Endian is from left-to-right order similar to writing in Western culture. These names come from Gulliver's travels and is about how groups break eggs (whether they break it on the big end or the little end). True story

What is ARM?
Advanced RISC machine (originally Acorn RISC machine). With RISC in title, uses RISC architecture. Whereas majority of personal computers use CISC, smartphone, portable technology, and embedded systems use RISC almost exclusively.

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

Example of CISC: a multiply instruction called MULT, that takes two numbers, multiplies them, and stores the result in a register. No need to explicitly load numbers into registers. i.e. MULT A, B

Example of RISC:
A RISC-based computer design approach means processors require fewer transistors, which means less heat, power use, and complexity.
RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

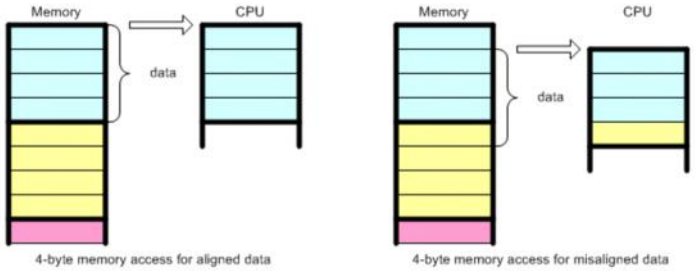
LOAD X, A
LOAD Y, B
PROD X, Y
STORE A, X

OK. Now back to original point…
x86 can deal with data types of 8 (byte), 16 (word), 32 (doubleword), 64 (quadword) and 128 (double quadword). These are general data types. Also can support specific data types like

- Integer -- signed binary value in two's complement representation
- Packed BCD
- Near and far Pointer -- represents the offset within a segment
- Ordinal -- unsigned integer
- Floating point

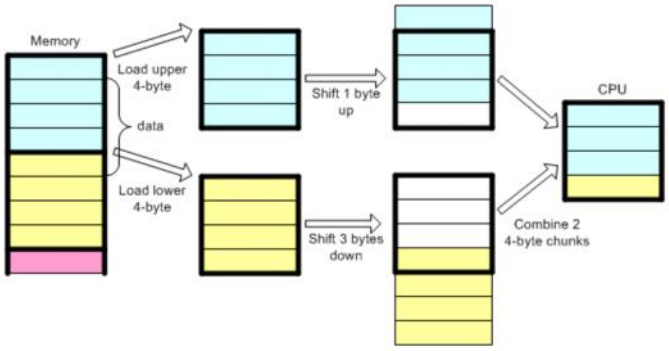
Words don't need to be aligned at even-numbered addresses; doublewords don't need to be aligned at addresses evenly divisible by 4, etc.

Why align? Issue of speed and performance.



Memory mapping from memory to CPU cache

If the data is misaligned of 4-byte boundary, CPU has to perform extra work to access the data: load 2 chucks of data, shift out unwanted bytes then combine them together. This process definitely slows down the performance and wastes CPU cycle just to get right data from memory.



Misaligned data slows down data access performance

ARM data types of 8 (byte), 16 (halfword), and 32 (word) bits in length. Alignment is critical as halfword and word access should be halfword and word aligned, respectively.
3 alternatives options supported for alignment issues:

1. Default case - address is treated as truncated
2. Alignment checking - alignment fault for attempting unaligned access.
3. Unaligned access - processor uses 1 or more memory accesses to generate required transfer

Usually, don't have floating point hardware, which saves power and area. This is computed in software. Architecture is bi-endian. You can set E-bit, which defines which Endian to load and store data.

TYPES OF OPERATIONS
Opcodes may vary from machine to machine. However, same general types of operations:

- Data transfer
- Arithmetic
- Logical
- Conversion

- I/O
- System control
- Transfer of control

Data transfer: most fundamental type of machine instruction. Must specify several things:
- Location of source and destination operand (register to register, register to memory, memory to memory, etc.)
- Length (or amount) of the data to be transferred
- Mode of addressing must be specified. (we'll discuss more on this in chapter 13)

Sample instructions based on IBM EAS/390

Operation Mnemonic	**Name**	**Number of bits transferred**	**Description**
L	Load	32	Transfer from memory to register
LH	Load halfword	16	Transfer from memory to register
LR	Load	32	Transfer from register to register
LER	Load (short)	32	Transfer from floating-point register To floating point register
LD	Load (long)	64	Transfer from memory to floating-point register
ST	Store	32	Transfer from register to memory
STC	Store character	8	Transfer from register to memory

Data transfer operations are simplest type, especially if register to register. If operand is in memory, then must perform the following actions:
1. Calculate the memory address based on address mode
2. If virtual memory, translate from virtual to real memory address
3. Determine whether addressed item is in cache
4. If not, issue a command to memory module

Arithmetic:
Typical add, subtract, multiply, and divide. Also provides single-operand instructions like Absolute, Negate, Increment, and decrement

Logical:
Manipulating individual bits of a word or other addressable units.
Logical shift-bits of a word are shifted left or right
Arithmetic shift-treats data as signed integer and does NOT shift the sign bit.
Rotate-preserves all the bits being operated on; cyclic shift

Conversion:
Change the format or operate on the format of data. Example: convert from decimal to binary

I/O:
Input and output. Instructions include reading and writing to devices

System control:
Those that can be executed ONLY while processor is in a certain privileged state or is executing a program in a special privileged area of memory.
Typically reserved for the operating system
Include instructions such as:
- Jump (branch) -- unconditional
- Jump conditional
    - Branch if result is positive
    - Branch if result is negative
    - Branch if result is zero
    - Branch if overflow occurs

- Return--replace contents of PC from known location
- Skip--increment PC to skip next instruction

Procedure call instructions - self-contained computer program. At any point procedure can be invoked or *called.* Similar to a function/method in higher-level languages. **Modularity** by breaking larger program into smaller ones.
Uses a CALL instruction to branch to routine and then RETURN to get back.

Processor must save return address:
- Register
- Start of called procedure
- Top of stack <-- powerful approach

Stack can also save parameters that need to be passed to called procedure. The entire set of parameters and return address stored for procedure invocation is known as *stack frame*

x86 offers four instructions to support procedures: CALL, ENTER, LEAVE, RETURN

Flags
Bits in special registers set by certain operations and used in conditional branch instructions

Status bit	Name	Description
C	Carry	Carrying out of left most bit after arithmetic operation
P	Parity	Parity of least significant byte. 1 indicates even; 0 is odd
A	Auxiliary carry	Carrying between half-bytes of an 8-bit arithmetic or logic operation
Z	Zero	Result is zero
S	Sign	Indicates the sign of the result
O	Overflow	Arithmetic overflow after and addition or subtraction for twos Complement arithmetic

ADDRESSING MODES

Need multiple modes to be able to reference large range on locations in main memory. These have tradeoffs between address range and flexibility, versus complexity and number of memory references.

Common addressing modes:
- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

Almost all processors utilize multiple addressing modes. Different opcodes can have different addressing modes. Also, one or more bits in the instruction format can be used as a mode field.

EA = effective address -- main memory address or register (if no virtual memory)

Mode	**Algorithm**	**Principal advantage**	**Principal disadvantage**
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity

Assume instruction is LDAC

LDAC #10
LDAC 10
LDAC (10)
LDAC R ← usually a letter for registers
LDAC @R
LDAC 10(X)

Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

LDAC @R
LDAC 10(X)
PUSH

a letter for registers

*direct may use # when associating values
*indirect addressing - address field is usually less than the word length
*register indirect can use @ symbol to distinguish it's referring to register

Indexing: address references a main memory address, and the referenced register contains a positive displacement from that address.