COS 140: Foundations of Computer Science

CPU Organization and Assembly Language

Fall 2018

Homework, etc.

- Reading
  - Chapter 14
  - Appendix A is there as well
- Homework: Exercises at end of Chapter 14, due 10/17 (later than usual)
- Prelim I: Friday, 10/12
  - Detailed questions (e.g., problems) on material up through RAID
  - Conceptual questions on all material up to & including today's material

CPU

Central Processing Unit

- Part of the computer that carries out the instructions of programs
- Controls the other two parts (memory, I/O).
- This class:
  - CPU organization
  - Machine (and assembly) language introduction

Components of the CPU

Arithmetic and logic unit (ALU): where actual computation takes place
Control unit (CU): controls moving information around in CPU, placing data in registers, getting new instructions from memory, ALU function, etc.
Registers: memory for CPU operation

Registers

- Registers virtually all CPUs have:
  - program (or instruction) counter
  - program status word (PSW)
  - instruction register

Design Issues for Registers

- User accessible or protected?
  - Want to protect the machine, other users from the user
  - PSW – probably protected
  - Instruction counter: generally protected, but some special instructions for changing (e.g., branch, subroutine instructions)
  - Data registers: user accessible (via instructions)

Design Issues for Registers

- General-purpose or special-purpose?
  - General-purpose ⇒ flexibility
  - Special-purpose ⇒ reduce number of operands or operand size
  - Mixed: some special-purpose registers that are used as general-purpose registers by some instructions
  - Different size registers for different purposes
- Size?
  - Need to hold largest operand required
  - Sometimes use multiple registers for single operand
- Who saves them when needed (on interrupt) – software or hardware?

Machine Language

- *Instruction set:* instructions CPU can carry out
- Each: *Op code* & ≥ 0 *operands*
- Operands include addresses and data.
- Op codes, operands: binary numbers—of course.
- Instuctions are the *machine language*
- Difficult for humans to use!

Assembly Language

- Symbolic version of machine language.
- Op codes represented by *mnemonics* – e.g.,
  - Machine language: addition op code might be 001001
  - Assembly language: might represent as `ADD`
  - Operands can be symbolic: names for constants, memory locations
  - *Assembler* translates assembly language program $\rightarrow$ machine language

Instruction Execution

- ☐ *Fetch* instruction from memory
- ☐ *Decode* instruction
- ☐ Fetch operands (if any)
- ☐ Carry out instruction
- ☐ Store results (if any)

Instruction Format

- ☐ Specification of action to be done: op code
- ☐ Addresses of any operands
- ☐ Address of any result to be stored
- ☐ E.g., add register 0 to register 1
  - Assembly lang: `ADD R0,R1`
  - Instruction: $01000 \mid 000 \mid 0000 \mid 00 \mid 01$
  - 5 bit op code, 3 bits unused, 4 bits for mode (direct, direct), R0, R1

Types of Instructions

- ☐ Data transfer
- ☐ Arithmetic operations
- ☐ Logical operations
- ☐ Transfer of control
- ☐ System control

Data Transfer Instructions

- ☐ Move data from place to place
- ☐ Source, destination could be registers, memory
- ☐ Different addressing modes possible (see later)
- ☐ Size of transfer: on some machines, amount can be large

Arithmetic Operations

- Almost all CPUs (i.e., their ALUs) provide `ADD`, `SUB`, `MULT`, and `DIV`
- Why include `SUB` – could use `ADD` and complement second op?
- Usually provide for signed integers, other types (e.g., floating point)
- Usually have to move the operand's data to location where ALU expects it (e.g., a register)

Logical operations

- Usually provide `AND`, `OR`, `NOT`, `XOR`, `EQUAL`, `SHIFT`, `ROTATE`, maybe arithmetic shift (sign bit propagated)
- Why include all these, when you can do it all with `NAND`?
- What about complement? Can you do it with XOR? with NOT?
- How would you *mask* (clear) particular bits?
- How would you set particular bits?

Transfer of control

- Branch (jump) instructions
  - Unconditional and conditional
  - Absolute or relative
- Subroutine call and return

System control

- Protected operations – only executable by some processes (e.g., operating system)
- Use to access protected registers, protected memory, etc.
- Input/output instructions are usually protected

Number of Instructions

- One philosophy: have relatively few, simple instructions
  - Allows instructions to be optimized
  - Requires less chip space ($\Rightarrow$ cheaper, or more other things can be put on chip)
  - Components can be closer to each other ($\Rightarrow$ faster)
  - Reduced instruction set computers (RISC)
  - Examples: SPARC chips in Sun machines, PowerPC in older Macs

Number of Instructions

- Other philosophy: have a large number of instructions, including special-purpose ones (e.g., multimedia, etc.)
  - Makes it easier for programmers
  - Requires less memory, less disk space for programs
  - Complex instruction set computers (CISC)
  - Examples: Intel family of processors

Data Types

- Numbers:
  - Integers (sign-magnitude, 2's complement, packed decimal)
  - Floating point
  - Size: single, double, etc.
- Characters:
  - Encoded as ASCII or Unicode characters these days
  - Single byte (ASCII) or two or more bytes (Unicode) in length
  - Strings of characters: length + string or null-terminated
- Logical data
- General bit strings: e.g., for multimedia, etc.

Addressing Modes

Addressing Modes

- Different instructions have different addressing *modes*
- Determines where the data corresponding to the operands is
- E.g.: think of operand as *name* of data's location
- Modes: immediate, direct, indirect, register, register indirect, displacement, others
- Issues:
  - Which one should CPU use for instruction?
    - Sometimes indicated by op code
    - Sometimes indicated by *mode bits* that can be set
  - How many bits are required for instruction?
  - How many memory references are required to find *effective address* (where the data is actually located)?

Immediate Addressing

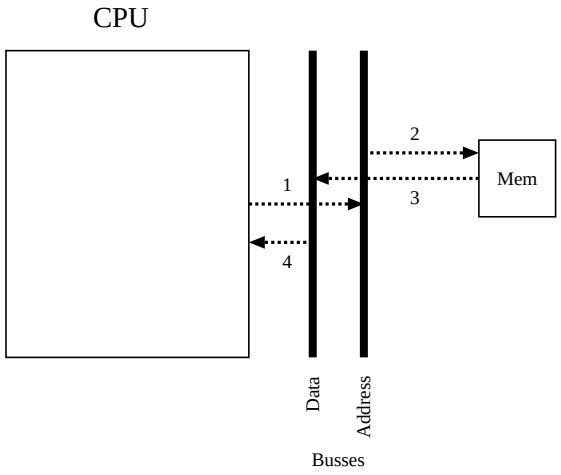
- Operand's data is the operand itself: i.e., in instruction
- E.g., NEG #5 might mean "negate 5, leave result in register 0"
- How many bits needed?
  - Generally: fixed-size *field* in the instruction for operand
  - Generally limited to integers (2's complement) or characters
- Operand memory accesses: none

Direct Addressing

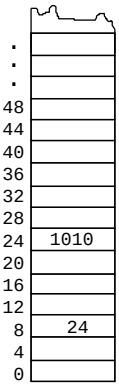
- Operand contains *memory address* of the data
- E.g.: `NEG 8`: negate the value of the data *located* at memory location 8
- Number of bits needed?
  - Varies between computers, maybe between instructions
  - With n bits devoted to address, can refer to 2^n addresses in memory
  - Up to size of word

Direct Addressing

- Operand memory accesses: one per operand



Indirect Addressing

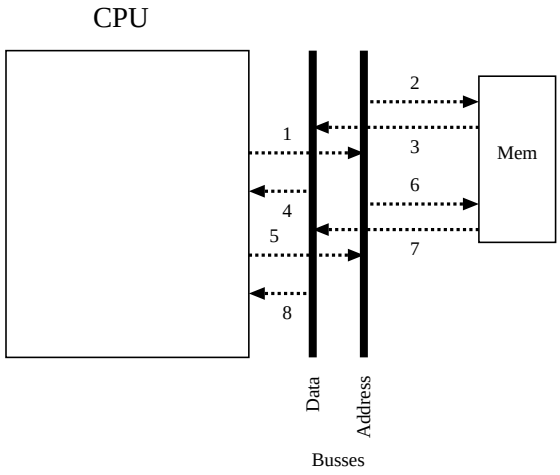


- Operand contains the *address of the address* of the data
- E.g.: `NEGI 8`: if address 8 contains 24, then the actual data that would be negated would be what's stored at memory location 24

- Number of bits needed – same as for direct addressing

Indirect Addressing

- Operand memory accesses needed: two per operand



Register Addressing

- Implicit ⇐ op code specifies which register is used
- Explicit ⇐ operand specifies which register
- E.g.: `NEG R5` – negate the data contained in *register* 5
- Number of bits needed? Usually few, since number of registers limited.
- Operand memory accesses: none
  - Only requires access to register, which is quick
  - Only really efficient if register used over and over – otherwise, have to load the register, which could be more inefficient than direct addressing
- RISC machines make heavy use of register addressing – e.g., PowerPC forces this for all arithmetic operations

Register Indirect Addressing

- Register contains *address* of the data
- E.g.: `NEGI R5` – if register 5 contains the value 24, then the data would be found at address 24
- Number of bits needed? Same as register addressing
- Operand memory accesses: one per operand access
- When would this be useful?

Other Addressing Modes

- Relative displacement addressing: used for jumps (branches) relative to program counter
- Base register displacement addressing: have a *base register* to which all addresses are added

Instruction Set Design

Designing Instruction Sets

- Which operations to support
  - Memory-mapped I/O or special instructions?
  - How many instructions to support? RISC or CISC?
- Format of instructions:
  - Size of instruction: part of word, word, multiple words?
  - All instructions same size or not?
  - Fields within instruction?
  - All instructions with same format or not?
  - Address modes?
  - Number of registers?
  - Addressing granularity?

Abstraction

Abstraction hierarchy