CSCI 4717 – Computer Architecture

Instruction Set Design

Reading: Sections 5.2 and 5.3 of the Art of Assembly

Today

- Execution of an instruction
- Theory of instruction set design
- Trade offs of different types of instruction sets

Turing Complete

- A simplistic definition: A machine is *Turing complete* if it can produce the result of any calculation.
- Is a machine Turing complete?
  - Is it capable of everything that a Turing machine is capable of?
  - Does it have a Turing-complete instruction set?
    - Conditional branching
    - Read and write memory
    - Addition
    - Negation
  - Most all machines are Turing complete

One Instruction Set Computer

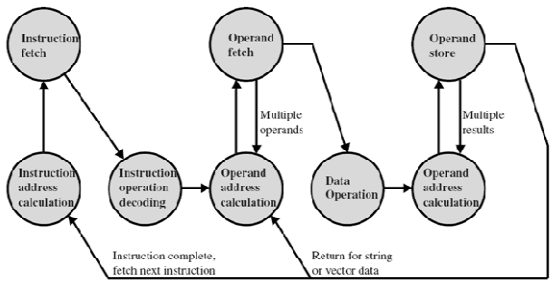
- Subtract and branch if negative

```
subneg   a, b, c    ; b = b – a
                    ; branch to c if b < 0
```

- Move machine (memory-mapped ALU and PC)

```
move     a, b       ; copy memory location
                    ; b to memory location
                    ; a
```

Execution of a Machine Instruction



Instruction Set Design

- Instruction set design needs to be well-addressed from the start
  - Attractive to programmers
  - Backwards compatible
  - Support architectural performance efforts

Instruction Set Design (continued)

Approaches

- kitchen sink: include all the instructions you can think of
- minimalist: include only the instructions you absolutely need to get by
- balance: start with a simple instruction set while leaving room for expansion

Problems with Kitchen Sink

- Limited number of transistors
  - 8086 processor had transistor budget of 29,000 transistors
  - 10-Core Xeon Westmere-EX had transistor budget of 2,600,000,000
- Cost: the more transistors → the higher the cost

Problems with Kitchen Sink (continued)

- Expandability: difficult to predict what instructions will be needed in future, so leave room
- Legacy Support: if you include something that's not used that often, it will still need to be supported for backwards compatibility
- Complexity: too complex and early adopters (assembly language programmers and compiler writers) won't be drawn to it

Basic Instruction Design Goals

- Machine code performs well in terms of size and speed
- Instruction size is kept reasonably small
- Programmer's needs are fulfilled

Instruction Types

- Set should include common instructions:
  - Data transfer - LOAD, STORE, MOVE, PUSH
  - Data processing - ADD, SUBTRACT, AND, NOT, NEG
  - Program control - JMP, BEQ, BNE
- Group instructions by common types of operands, e.g., ADD and SUB have the same operand needs, but not the same as a NOT or a DIV

In-Class Example

- Assume a processor has 4 registers (aside from PC, MAR, and MBR)
  - Accumulator A (ACCA)
  - Accumulator B (ACCB)
  - Pointer X (IX)
  - Stack pointer (SP)
- Issues
  - Addressing modes
  - Number of non-register addresses per instruction
  - Fixed or variable length machine code
- Create a list of instructions you want in your processor

Encoding an Instruction

- Each instruction combined with each address mode must have unique numeric opcode
- For an n-bit number, there are 2^n possible opcodes
- Decoder speed: Randomly assigning opcodes to instructions is bad idea
  - Byte-long instruction allows for 256 opcodes
  - Requires 8-line to 256-line decoder (256 8-input NAND gates)
  - Uses a lot of transistors

Encoding an Instruction (continued)

- Partition instruction so that sections of instruction represent different items or "sub-opcodes"
  - Some bits identify operation
  - Some bits identify addressing mode

Encoding an Instruction – Example

- Eight basic operations: (3-line to 8-line decoder)
- Two operands each of which could be four values (two x 2-line to 4-line decoders)
- Instruction becomes three fields: three bit operation, two bit source operand, two bit destination operand

Size of an Instruction Opcode

- Instruction size must be a multiple of the memory word size
- Useless to go below the data bus width
- Large instruction size → consumes extra memory
- Compact instruction size → not enough flexibility for the programmer
- Solution → variable size instructions

Size of an Instruction Opcode (continued)

Number and type of addresses (operands) may also affect instruction size.

- CISC designers do not consider operands (except register references) as part of the opcode
- RISC designers make processors that tend to have all register-based operations and therefore consider the operands as part of the opcode

Size of an Instruction Opcode (continued)

- 8-bit opcode (256 instructions) – not much room for expandability
- 16-bit opcode (65,536 instructions) – enough opcodes, but twice the memory requirements
- Solution:
  - Analyze typical applications to determine most used instructions – assign those to 8-bit opcodes
  - Use 16-bit or even 24-bit opcodes for least used opcodes

Example

- Byte patterns for 8-bit instructions:
  - 01xxxxxx
  - 10xxxxxx
  - 11xxxxxx
- Byte patterns for multi-byte instructions:
  - 001xxxxx xxxxxxxx– two-byte opcodes
  - 000xxxxx xxxxxxxx xxxxxxxx – three-byte opcodes
- This gives us $3 \times 2^6 = 192$ 8-bit opcodes, $2^{(5+8)} = 8{,}192$ 16-bit opcodes, and $2^{(5+16)} = 2{,}097{,}152$ 24-bit opcodes.

Problems with Large Number of Operand Choices

- The 80x86 MOV instruction requires a two-byte opcode because of the number of possible operands.
- Two of the most common MOV instructions, "mov(disp, eax)" and "mov(eax, disp)" are used so much that they are given their own special 8-bit opcode

Saving Opcodes

- It may be worthwhile to set aside some of the opcodes for future expandability from each of the three opcode sizes
- For example, reserve one block each from 8-bit (64 opcodes), 16-bit (4,096 opcodes), and 24-bit (1,048,576 opcodes)
- Example of what not to do: 68HC11 adding an additional index register

Problems with Variable-Length Opcodes

- Decoding is more difficult and slower
- Adds extra step (determining instruction size) to decode process
- Causes instruction execution time to vary from instruction to instruction creating problems with pipelining

Example: MIPS Instruction Format

Register-Immediate Instruction

31–26	25–21	20–16	15–0
Operation	Src reg	Dest reg	Immediate data

Branch Instruction

31–26	25–21	20–16	15–0
Operation	Src reg 1	Src reg 2	Branch/address displacement

Register-to-Register Instruction

31–26	25–21	20–16	15–11	10–6	5–0
Operation	Src reg 1	Src reg 2	Dest reg	Shift	Function

Jump/Call Instruction

31–26	25–0
Operation	Target address