CS 221 Computer Architecture

Week 8: Classifying Instruction Sets

Fall 2001

Course Schedule

W1	Sep 11- Sep 14	Introduction	
W2	Sep 18- Sep 21	Information Representation (1)	(Chapter 3)
W3	Sep 25- Sep 28	Information Representation (2)	(Chapter 3)
W4	Oct 2- Oct 5	Computer Architecture Pep/6	(Chapter 4)
W5	Oct 9- Oct 12	Assembly Language	(Chapter 5)
W6*	**Oct 16- Oct 19**	**High-Level Languages (1)**	**(Chapter 6)**
W7*	**Oct 23- Oct 26**	**High-Level Languages (2)**	**(Chapter 6)**
	Midterm*		
W8	Oct 30- Nov 2	Instruction Sets	(Tanenbaum)
W9	Nov 6 - Nov 9	Storage Management	(Chapter 9)
W10*	**Nov 13- Nov 16**	**Devices and Interrupts**	**(Chapter 8)**
W11	Nov 20- Nov 23	Combinational Logic	(Chapter 10)
W12	Nov 27- Nov 30	Sequential Logic & Review	(Chapter 11)

**Marked classes will be taught by Dave Dove, same place same slot*

Announcements

- **Assignment 3 will be posted on the web today. You can find all details on the web site.**

  Due: **Thu Nov. 15th, 11:00**
  Where: **Drop Box CISC221 Goodwin Hall**

Plan for Week 8

- **Classifying Instruction Sets**
  - By number of operands
  - By type of operations
  - By number of instructions
- **Assignment 3 will be posted on the web.**
- **Readings:**
  - Tanenbaum 5.1 handout

I.
Classifying Instruction Sets by Operands

Instruction Sets

- **We have experienced programming in Assembly Language at Level 5**
  - But assembly is really just human readable machine language
  - So we have experienced programming using the PEP/6 instruction set
  - The opcode set that implements the CPU calculations
- **So we can now ask the question: are all instruction sets like PEP/6?**
  - The answer is no.

Classifying Instruction Sets

- **So what makes instruction sets different?**
  - Or rather, how can we classify them?
  - One way to classify them is by counting the number of operands required.
  - Another way is to classify them according to the number or type of instructions used.
- **Classifying by number of operands**

  In today's class, we will be discussing:

  - Zero-operand machines
  - One operand machines
  - Two operand machines
  - Three operand machines

Zero Operand Machines

- **Most instructions on a zero-operand machine have no explicit operands at all.**
- **Zero-operand machines are stack-based**
  - most instructions take their operands from the top of a hardware stack
  - and push the results back onto the stack.
- **To get the data items onto the stack and to store the results there need to be at least two one-operand commands:**
  - `push address` command that fetches a value from memory and pushes it onto the stack
  - `pop address` command that removes the top element from stack and stores it in memory.

An Example: Java VM

- **The Java Virtual Machine is a stack-based (software) processor**
  - Java needs to run on a large number of hardware CPUs
  - It uses a stack-based machine because it provides a clean model of computation and poses no restrictions on hardware.
- **To see how a zero-operand machine works, here's an example of the JVM byte codes produced to evaluate an arithmetic expression:**

```
// Java method:
//
public double quadraticPart(double a, double b,
  double c)
    return ((b * b - 4 * a * c)/(2 * a));
```


Compiled Byte Code

```
Byte codes implementing the method on the Java Virtual Machine:

Output of "javap -c":          Interpretation

   0 dload_3                   push b
   1 dload_3                   push b
   2 dmul                      multiply
   3 ldc2_w #28 <Double 4.0>   push 4.0
   6 dload_1                   push a
   7 dmul                      multiply
   8 dload 5                   push c
  10 dmul                      multiply
  11 dsub                      subtract
  12 ldc2_w #26 <Double 2.0>   push 2.0
  15 dload_1                   push a
  16 dmul                      multiply
  17 ddiv                      divide
  18 dreturn                   return popped value
```


How does this change stack?

If a has the value 6.0, b 7.0 and c 8.0:

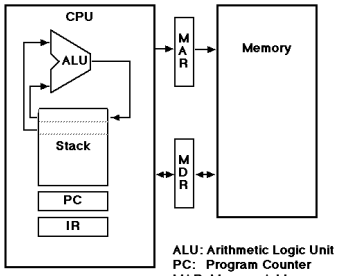
push b	top->	7.0
		?
push b	top->	7.0
		7.0
		?
multiply	top->	49.0 (pops top 2 values off the stack, and pushes their product)
		?
push 4.0	top->	4.0
		49.0
		?
push a	top->	6.0
		4.0
		49.0
		?
multiply	top->	24.0
		49.0
		?
push c	top->	8.0
		24.0
		49.0
		?

How does this change stack? (2)

multiply	top->	192.0
		49.0
		?
subtract	top->	-143.0
push 2.0	top->	2.0
		-143.0
		?
push a	top->	6.0
		2.0
		-143.0
		?
multiply	top->	12.0
		-143.0
		?
divide	top->	-11.92
		?

return popped value top-> ?

Architecture of 0-Operand Machine



ALU: Arithmetic Logic Unit
PC: Program Counter
MAR: Memory Address Register
MDR: Memory Data Register
IR: Instruction Register

0-Operand Architecture (2)

- **The program counter contains the address of the next instruction**
  - The Instruction Register contains the current instruction
- **The Arithmetic Logic Unit pops the top two items off the stack as its operands and pushes the result onto the stack.**
- **When the CPU needs to access memory, it puts the address of the memory location it wishes to read or write into the memory address register.**
  - During memory read and write operations, this address sets the corresponding address lines of the computer's bus, the collection of wires that connect the various components of the computer.

0-Operand Architecture (3)

- **For a push operation, the CPU tells the memory it wishes to read data by "asserting" some control line or lines on the bus.**
  - The memory fetches the value stored at the address signalled on the bus address lines, and returns the value using the data lines of the bus.
  - Those data lines are connected to the Memory Data Register of the CPU.
  - Finally, the CPU copies the value from the MDR onto the top of the stack.
- **For a pop operation, the CPU copies the value at the top of the stack into the MDR**
  - asserts the memory write signal while the destination address is on the bus.
  - The data lines on the bus get set according to the data value in the MDR, and the memory writes that value into the specified address.
- **This Machine Uses Little Memory!!!!**

One-Operand Machines

- **One-operand machines have a special register, usually called the accumulator, as the implicit operand of all instructions. The other operand is in the memory.**

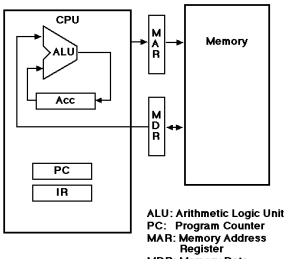
```
A := 2 * B + C  becomes

LOAD 2      ; Accumulator := 2
MULT B      ; Accumulator := Accumulator * Memory[B]
ADD C       ; Accumulator := Accumulator + Memory[C]
STORE A     ; Memory[A] := Accumulator
```

where B, C, and A would be replaced by addresses

- **The original von Neumann design was an accumulator machine, and the one-operand design dates from those early days of computing when registers were extremely expensive.**

One-Operand Architecture



ALU: Arithmetic Logic Unit
PC: Program Counter
MAR: Memory Address Register
MDR: Memory Data Register
IR: Instruction Register
Acc: Accumulator

- This diagram shows something that will be important in the comparison of RISC and CISC designs that we are working toward.
- While the instruction set may have operations that perform operations with values from memory, those values are actually fetched into a register, the Memory Data Register, before the operations occur. It is this register that is wired into the CPU's circuitry to add, subtract, et cetera, not every location in memory.

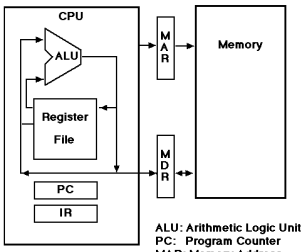
Two-Operand Machines

- **Two-operand instruction sets are now the most common.**
  - For example on the Intel x86 architecture:

```
MOV [i], 10    ; i := 10
MOV [j], 5     ; j := 5
MOV AX, [i]    ; AX register := i
ADD AX, [j]    ; AX := AX + j
MOV [k], AX    ; k := AX
INT 21h        ; return to OS
END
```

- **The 2-operand machine has some 1- and 0-operand instructions**
  - Here there is the INT instruction, which traps into the OS
- **The result of operations on a two-operand machine is stored into one of the operands (the first one, in the Intel assembly syntax).**
- **In orthogonal instruction sets the choice of addressing modes for the operands is independent of each other and of instruction.**
  - Few instruction sets are perfectly orthogonal. Most have restrictions such as requiring that at most one operand be in memory (the other being in a register)

2-Operand Architecture



ALU: Arithmetic Logic Unit
PC: Program Counter
MAR: Memory Address Register
MDR: Memory Data Register
IR: Instruction Register

- The intention is to show that operands to the ALU can come from memory, via the memory data register, as well as from the registers within the CPU. Similarly, results from ALU operations may be written to a register or to the memory, via the MDR.

3-Operand Machines

- **Three operand instruction sets allow you to explicitly specify the destination for the result of an operation, independently of its source operands.**

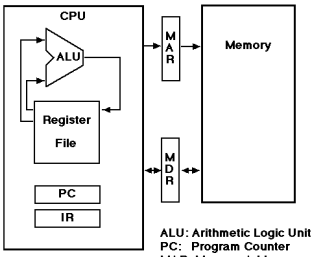
```
# Code for a 3-operand machine, the MIPS R5000 in an SGI O2.
# The code was produced by the SGI compiler from C:
#         return ((b * b - 4 * a * c)/(2 * a));
# The numbers prefixed by $ signs are register numbers.
# a is passed in $4, b in $5, c in $6,
# and $2 is used to return the result form the function.

        mul     $14, $5, $5
        mul     $15, $4, 4
        mul     $24, $15, $6
        subu    $25, $14, $24
        mul     $8, $4, 2
        div     $2, $25, $8
        j       $31
        .end    quadraticPart
```


3-Operand Machines

- **Three-operand instructions were only common on large machines with word sizes (and instruction formats) of 64 bits**
  - Since you need many bits to encode three operands, if the operands may include addresses anywhere in memory.
- **Now, though, three-operand machines have proliferated in the form of the load/store architecture.**
  - In load/store machines the only instructions which access memory are
  - instructions to load a register with the value at an address,
  - instructions to store the value in a register at a particular address
  - branch and subroutine instructions.
- **Load-Store machines are almost synonymous for 3-Operand machines. They have a stack machine-like architecture**
  - Except that a collection of general registers replaces the stack

3-Operand Architecture



ALU: Arithmetic Logic Unit
PC: Program Counter
MAR: Memory Address Register
MDR: Memory Data Register
IR: Instruction Register

- All ALU instructionsmust have all their operands in registers.
- This is of course FAST!!!
- But expensive
- In practice, 3-operand machines tend to be load/store machines, and vice versa.
- That's because limiting the operands of most instructions to the set of CPU registers makes it possible to encode three operands into a reasonably-sized instruction format.
- With 32 registers, for example, you need only 5 bits to encode a register number, for a total of 15 bits of operands.

Differences in Speed

- **Both two- and three-operand machines are a response to the disparity in speed between operations within the CPU and accesses to memory.**
  - Having lots of registers lets you keep local variables and parameters entirely in memory.
  - Having a three-operand instruction format instead of a two-operand format helps make good use of these registers
  - It is not necessary to overwrite one of the operands with the result, so you can keep intermediate values without needing extra instructions.
- **To compute A := B + C on a two-operand machine, without losing the values in B and C, you'd need two instructions:**

```
MOVE A B    ; A := B
ADD A C     ; A += C
```

- **One instruction would suffice on a three-operand machine:**

```
ADD A B C   ; A := B + C
```


II. Classifying Instruction Sets by Instructions

Classifying Instruction Sets

- **We have seen how we can categorize differences between instruction sets by the number of operands used.**
- **There are, however other ways:**
  - Classification by Instruction Type
  - Classification by Number of Instructions
- **Today, we will discuss these alternative ways of classification**

Classifying by Instruction Type

- **One can classify instruction sets by examining the type of instructions used**
  - What do the instructions do?
  - For any particular machine, one can outline the functionality of the instruction set
  - Note that this is categorization of instructions within an instruction set
- **There are roughly six different types of operations performed by instructions:**
  1. **Data movement**
  2. **Arithmetic operations**
  3. **Logical operations**
  4. **Comparative operations**
  5. **Flow of control**
  6. **Input/output**

Classifying by Instruction Type (2)

- **Data movement instructions**
  - Provide functionality to move data from and to main memory
  - LOADA, STOREX, etc.
- **Arithmetic operations:**
  - Provide functionality to calculate arithmetic expressions.
  - ADDA, SUBX, etc.1
- **Logical operations:**
  - Provide functionality to perform logical operations on data.
  - NOTA, ANDA, etc.
- **Comparative operations:**
  - Provide functionality to compare data for program flow control.
  - COMPA, COMPX

Classifying by Instruction Type (3)

- **Flow of control:**
  - Provide functionality to alter program behavior, e.g., on the basis of data
  - BR, BRN, JSR, RET, etc.
- **Input/output:**
  - Provide functionality to input and output data.
  - CHARI, CHARO

- **Note that the above is a categorization of instructions within an instruction set**
  - However, it provides a framework for making comparisons between instruction sets.
  - In particular, it is our entry point for making the distinction between computers known as CISC machines and those known as RISC machines

Classification by the Number and Complexity of Instructions

- **Our comparison by type of instructions provided leads to our final classification: by complexity and number of instructions.**
- **There have long been two camps defending what instructions types should be provided:**
  - CISC camp defended use of a large set of complex instructions: **C**omplex **I**nstruction **S**et **C**omputer.
  - RISC camp defended use of a small set of simple instructions: **R**educed **I**nstruction **S**et **C**omputer.

Reduced Instruction Set Computing

- **Reducing the number of instructions reduces the complexity of the chip makes individual instructions execute faster**
  - Even though more instructions might be required to accomplish a task.
  - For example, decoding the instruction would take less time if there is less instructions.
  - The holy grail of RISC was to make every instruction execute in 1 chip-clock per instruction
- **Given an unlimited number of transistors to work with, a processor can be made into a 1-clock-per-instruction device.**
  - But the transistors on a CPU have to fit into a limited amount of real estate, so transistor counts are limited.
- **RISC's original goal was to limit the number of instructions on the chip so that each could be allocated enough transistors to make it execute in one clock cycle.**

Reduced Instruction Set Computing (2)

- **A CISC CPU could multiply 5 by 10 like this:**

```
mov     ax,10
mov     bx,5
mul     bx
```

- **But a RISC chip without a mul instruction might do it like:**

```
mov     ax,0        ;initialize ax
mov     bx,10
mov     cx,5
Begin:
add     ax,bx
loop    Begin    ;loop cx times
```

- **Modern RISC chips do have multiply instructions, but it's illustrative nonetheless.**
- **The RISC version uses more instructions, but if the instruction timings are low enough, the RISC chip might do in 20 clocks what the CISC chip requires 100 clocks to do.**

Modern Computing

- **Today, transistor count is no longer a problem**
  - A Pentium contains more than 3 million transistors in an area that is roughly a half-inch square. The sheer number of transistors, combined with architectural improvements to the chip itself, allows a Pentium to execute two instructions per clock cycle.
- **Use of Pipelines and cache allows efficient processing of instructions**
  - The Pentium has two independent pipelines, execute the Von Neumann Cycle in parallel, allowing an average of 1 instructions per clock cycle.

Instruction 1

Fetch, decode, generate operand address, execute, store

Instuction 2

Fetch, decode, generate operand address, execute, store

- **So CISC chips have become faster, and RISC chips are allowed to have more instructions: the difference has blurred**

Back to Instruction Types

- **Looking at the various categories of instructions again…**
- **Machines vary in the range of arithmetic operations they perform in hardware.**
  - For example, the Pep/6 has a very rudimentary set of arithmetic operations: it can add and subtract integers
  - Most machines these days have hardware instructions for integer multiplication and division, and many have special floating point coprocessors. However, RISC emphasizes floating point, while CISC emphasizes integers
- **Machines differ in how they perform comparisons.**
  - Many, like the Pep/6, have status bits which are set by most operations, as well as by a COMPr-like instruction meant expressly for comparing two values.
  - On pipelined machines setting the condition codes with every instruction can cause problems
  - either the condition flags are set only by certain instructions designated to set them (SPARC, PowerPC)
  - or the condition codes are replaced by a comparison instruction that compares the values in two registers and sets the value in a third to indicate the result (MIPS).

Back to Instruction Types

- **Machines differ in support for input/output instructions**
  - The Pep/6 is a port-mapped system. It has special I/O instructions (CHARO and CHARI) to move data to output devices and get them from input devices (
  - In a memory-mapped systems the I/O devices look like memory to the CPU, so you get data from an input device by loading it from a particular memory location, and write data to an output device by storing it to a particular location.
- **Machines differ support provided for high-order language control structures like for loops and subroutines**
  - Here we get to a difference between classical CISC and RISC machines.
  - CISC machines, like the Digital VAX, the Motorala 68000, and the Intel x86, provide several complex instructions designed to map closely to high-order language concepts.
  - Classic RISC machines, like the MIPS and SPARC architectures, have only simple flow-of-control instructions.

An Example

- **The CISC VAX, for example, has a procedure calling instruction that**
  - saves a return address and transfers control to the first instruction in the subroutine,
  - takes care of adjusting the stack pointer to allow for passed parameters
  - saves a set of registers onto the stack.
  - The VAX also has a six-operand index instruction which not only calculates the address of the ith entry in an array, but will also trap into the OS if the subscript is out of range.
- **The RISC MIPS subroutine calling and return instructions, for example, are actually a bit simpler than the Pep/6's instructions**
  - they save the return address in a register instead of on the stack
  - leave it to the program to save that register's value to the stack before a nested subroutine call.

Back to Instruction Types

- **Machines differ in the variety of instructions available for data movement**
  - RISC machines are load/store machines: the only instructions that can access memory are load and store. All other operations take place between a large set of registers.
  - On CISC machines, most instructions can have at least one of their operands in memory. Several CISC computers, including the Intel x86, have special instructions for copying bytes from one location in memory to another.
- **CISC machines also have lots of ways of specifying the locations of the data in memory.**
  - The CISC VAX CPU has at least 13 addressing modes
- **RISC architectures have few addressing modes.**
  - The MIPS architecture has only one addressing mode for accessing data in memory, register plus a constant displacement, and it is up to the program to set the value in the register to synthesize other modes.

Context leading to CISC

- **CISC machines developed at about the time that main memory began to be implemented with semiconductor chips**
  - Difference in speed between the CPU operations and memory accesses was acute, and memories were expensive to build and thus small.
  - The use of assembly language for writing substantial programs made it a design goal to have an instruction set that would be readable and convenient for humans.
  - New high-order languages with poor compilers led CISC designers to introduce instructions that would facilitate the execution of high-level language programs.
- **All of these factors added up to a desire to have individual machine instructions do more**
  - programs would require fewer instructions so running them would require fewer memory access and less memory.
  - human assembly programmers could get more done with each line of code
  - elaborate instructions would make it simpler for compilers to generate code.

Comparing RISC vs. CISC

- With that context in mind, can look at the following comparison table. Aspects that are visible to the programmer appear first, while the aspects that are entirely questions of the hardware implementation come later.

CISC	RISC
Any instruction may reference memory	Only loads and stores reference memory
Many instructions and addressing modes	Few instructions and addressing modes
Variable format instructions	Fixed format instructions
Complex instructions taking multiple cycles	Simple instructions taking one cycle
Instructions interpreted by the microprogram	Instructions executed by the hardware
Not pipelined or less pipelined	Highly pipelined
Single register set	Multiple register sets
Complexity is in the microprogram	Complexity is in the compiler

- The first three aspects of CISC instruction sets--many instructions and addressing modes, variable-format instructions, and the ability of any instruction to reference memory--follow directly from the design goal of providing flexible power to assembly language programmers and compiler writers, while keeping program code small.

Questions?