Chapter 5::
Target Machine Architecture

Programming Language Pragmatics

Michael L. Scott

Copyright © 2009 Elsevier

Assembly-Level View

- As mentioned early in this course, a compiler is simply a translator
  - It translates programs written in one language into programs written in
    another language
    - This other language can be almost anything
    - Most of the time, however, it's the machine language for some available computer

Copyright © 2009 Elsevier

Assembly-Level View

- As a review, we will go over some of the material most relevant to language implementation, so that we can better understand
  - what the compiler has to do to your program
  - why certain things are fast and others slow
  - why certain things are easy to compile and others aren't

Copyright © 2009 Elsevier

Assembly-Level View

- There are many different programming languages and there are many different machine languages
  - Machine languages show considerably less diversity than programming languages
  - Traditionally, each machine language corresponds to a different computer ARCHITECTURE
  - The IMPLEMENTATION is how the architecture is realized in hardware

Copyright © 2009 Elsevier

Assembly-Level View

- Formally, an architecture is the interface to the hardware
  - what it looks like to a user writing programs on the bare machine.
- In the last 20 years, the line between these has blurred to the point of disappearing
  - compilers have to know a LOT about the implementation to do a decent job

Copyright © 2009 Elsevier

Assembly-Level View

- Changes in hardware technology (e.g., how many transistors can you fit on one chip?) have made new implementation techniques possible
  - the architecture was also modified
  - *Example*: RISC (reduced instruction set computer) revolution ~20 years ago
- In the discussion below, we will focus on modern RISC architectures, with a limited amount of coverage of their predecessors, the CISC architectures

Copyright © 2009 Elsevier

Workstation Macro-Architecture

- Most modern computers consist of a collection of DEVICES that talk to each other over a BUS
- From the point of view of language implementation:
  - the most important device is the PROCESSOR(S)
  - the second most important is main memory
  - other devices include: disks, keyboards, screens, networks, general-purpose serial/parallel ports, etc.

Copyright © 2009 Elsevier

Workstation Macro-Architecture

- Almost all modern computers use the (von Neumann) stored program concept:
  - a program is simply a collection of bits in memory that the computer *interprets* as instructions, rather than as integers, floating point numbers, or some other sort of data
- What a processor does is repeatedly
  - fetch an instruction from memory
  - decode it - figure out what it says to do
  - fetch any needed operands from registers or memory
  - execute the operation, and
  - store any result(s) back into registers or memory

Copyright © 2009 Elsevier

Workstation Macro-Architecture

- This set of operations is referred to as the <u>fetch-execute cycle</u>
  - The computer runs this cycle at a furious pace, never stopping, regardless of the meaning of the instructions
    - You can point a processor's instruction fetch logic at a long string of floating point numbers and it will blithely begin to execute them; it will do *something*, though that something probably won't make much sense
    - Operating systems are designed so that when the computer has nothing useful to do it is pointed at an infinite loop that it can execute furiously, but harmlessly

Copyright © 2009 Elsevier

Workstation Macro-Architecture

- The crucial components of a typical processor include a collection of FUNCTIONAL UNITS:
  - hardware to decode instructions and drive the other functional units
  - hardware to fetch instructions and data from memory and to store them back again if they are modified
  - one or more arithmetic/logic units (ALUs) to do actual computations
  - registers to hold the most heavily-used parts of the state of the computation
  - hardware to move data among the various functional units and registers

Copyright © 2009 Elsevier

Memory Hierarchy

- Memory is too big to fit on one chip with a processor
  - Because memory is off-chip (in fact, on the other side of the bus), getting at it is much slower than getting at things on-chip
  - Most computers therefore employ a MEMORY HIERARCHY, in which things that are used more often are kept close at hand

Copyright © 2009 Elsevier

Memory Hierarchy

	typical access time	typical capacity
registers	0.2–0.5ns	256–1024 bytes
primary (L1) cache	0.4–1ns	32K–256K bytes
secondary (L2) cache	4–10ns	1–8M bytes
tertiary (off-chip, L3) cache	10–50ns	4–64M bytes
main memory	50–500ns	256M–16G bytes
disk	5–15ms	80G bytes and up
tape	1–50s	effectively unlimited

Figure 5.1 The memory hierarchy of a workstation-class computer. Access times and capacities are approximate, based on 2008 technology. Registers must be accessed within a single clock cycle. Primary cache typically responds in 1–2 cycles; off-chip cache in more like 20 cycles. Main memory on a supercomputer can be as fast as off-chip cache; on a workstation it is typically much slower. Disk and tape times are constrained by the movement of physical parts.

Copyright © 2009 Elsevier

Memory Hierarchy

- Some of these levels are visible to the programmer; others are not
- For our purposes here, the levels that matter are *registers* and *main memory*
- Registers are special locations that can hold (a very small amount of) data that can be accessed very quickly
  - A typical RISC machine has a few (often two) sets of registers that are used to hold integer and floating point operands

Copyright © 2009 Elsevier

Memory Hierarchy

- It also has several special-purpose registers, including the
  - program counter (PC)
    - holds the address of the next instruction to be executed
    - usually incremented during fetch-execute cycle
  - processor status register
    - holds a variety of bits of little interest in this course (privilege level, interrupt priority level, trap enable bits)

Copyright © 2009 Elsevier

Data Representation

- Memory is usually (but not always) *byte-addressable*, meaning that each 8-bit piece has a unique address
  - Data longer than 8 bits occupy multiple bytes
  - Typically
    - an integer occupies 16, 32, or (recently) 64 bits
    - a floating point number occupies 32, 64, or (recently) 128 bits

Copyright © 2009 Elsevier

Data Representation

- It is important to note that, unlike data in high-level programming languages, memory is untyped (bits are just bits)
- *Operations* are typed, in the sense that different operations *interpret* the bits in memory in different ways
- Typical DATA FORMATS include
  - instruction
  - integer (various lengths)
  - floating point (various lengths)

Copyright © 2009 Elsevier

Data Representation

- Other concepts we will not detail but covered in other courses
  - Big-endian vs. little-endian (for details see Figure 5.2)
  - Integer arithmetic
    - 2's complement arithmetic
  - Floating-point arithmetic
    - IEEE standard, 1985

Copyright © 2009 Elsevier

Data Representation

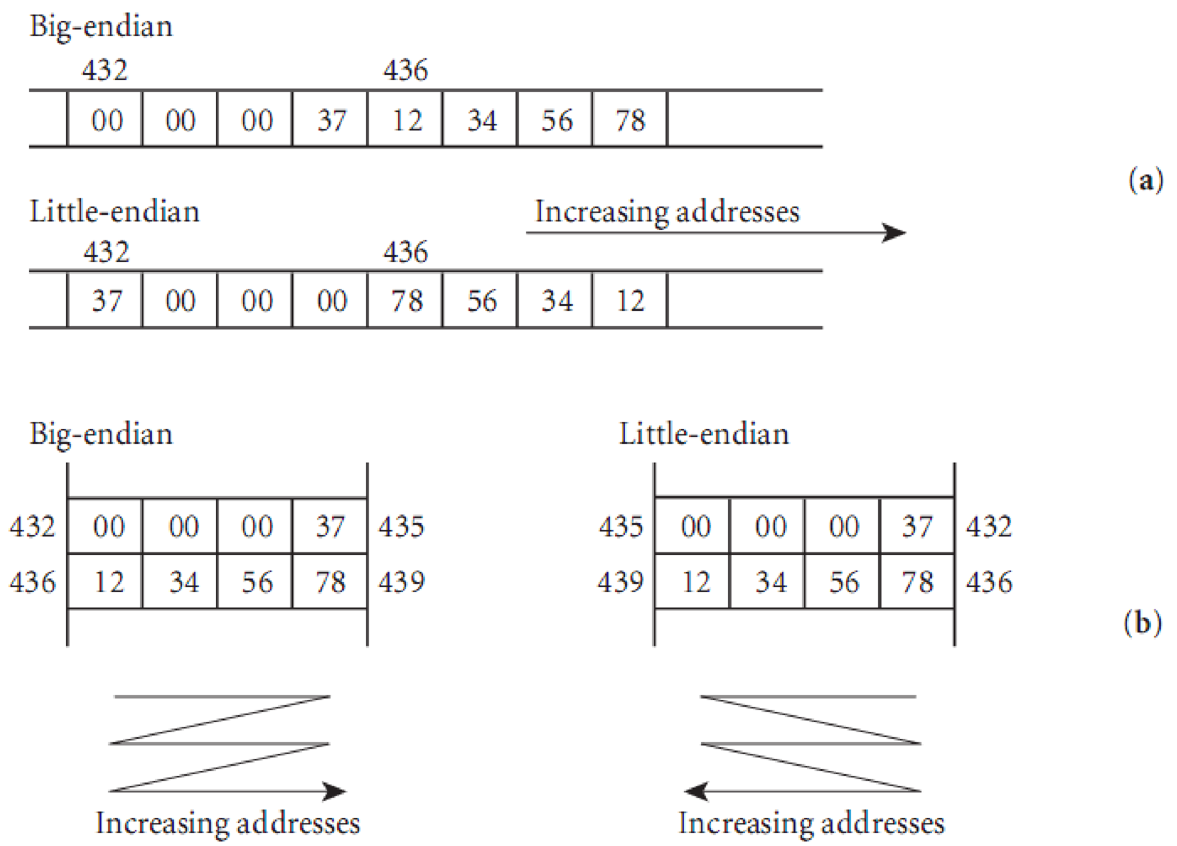

Figure 5.2 Big-endian and little-endian byte orderings. (a) Two four-byte quantities, the numbers 37_{16} and $12\,34\,56\,78_{16}$, stored at addresses 432 and 436, respectively. (b) The same situation, with memory visualized as a byte-addressable array of words.

Copyright © 2009 Elsevier

Instruction-Set Architecture

- The set of instructions executed by a modern processor may include:
  - data movement (load, store, push, pop, movem, swap - registers)
  - arithmetic and logical (negate, extend, add, subtract, multiply, divide, and, or, shift)
  - control transfer (jump, call, trap - jump into the operating system, return - from call or trap, conditional branch)

Copyright © 2009 Elsevier

Instruction-Set Architecture
Addressing Modes

- Instructions can specify many different ways to obtain their data (Addressing Modes)
  - data in instruction
  - data in register
  - address of data in instruction
  - address of data in register
  - address of data computed from two or more values contained in the instruction and/or registers

Copyright © 2009 Elsevier

Instruction-Set Architecture
Addressing Modes

- On a RISC machine, arithmetic/logic instructions use only the first two of these ADDRESSING MODES
  - load and store instructions use the others
- On a CISC machine, all addressing modes are generally available to all instructions
  - CISC machines typically have a richer set of addressing modes, including some that perform
    - multiple indirections, and/or
    - arithmetic computations on values in memory in order to calculate an effective address

Copyright © 2009 Elsevier

Architecture Implementation

- As technology advances, there are occasionally times when some threshold is crossed that suddenly makes it possible to design machines in a very different way
  - One example of such a *paradigm shift* occurred in the mid 1980s with the development of RISC (reduced instruction set computer) architectures

Copyright © 2009 Elsevier

Architecture Implementation

- During the 1950s and the early 1960s, the instruction set of a typical computer was implemented by soldering together large numbers of discrete components that performed the required operations
  - To build a faster computer, one generally designed extra, more powerful instructions, which required extra hardware
  - This has the unfortunate effect of requiring assembly language programmers to learn a new language

Copyright © 2009 Elsevier

Architecture Implementation

- IBM hit upon an implementation technique called MICROPROGRAMMING:
  - *same* instruction set across a whole line of computers, from cheap to fast machines
  - basic idea of microprogramming
    - build a *microengine* in hardware that executed a interpreter program *in firmware*
      - interpreter implemented IBM 360 instruction set
    - more expensive machines had fancier microengines
      - more of the 360 functionality in hardware
      - top-of-the-line machines had everything in hardware

Copyright © 2009 Elsevier

Architecture Implementation

- Microprogramming makes it easy to extend the instruction set
  - people ran studies to identify instructions that often occurred in sequence (e.g., the sequence that jumps to a subroutine and updates bookkeeping information in the stack)
  - then provided new instructions that performed the function of the sequence
  - By clever programming in the firmware, it was generally possible to make the new instruction faster than the old sequence, and programs got faster.

Copyright © 2009 Elsevier

Architecture Implementation

- The microcomputer revolution of the late 1970s (another *paradigm shift*) occurred when it became possible to fit a microengine onto a single chip (microprocessor):
  - personal computers were born
  - by the mid 1980s, VLSI technology reached the point where it was possible to eliminate the microengine and still implement a processor on a single chip

Copyright © 2009 Elsevier

Architecture Implementation

- With a hardware-only processor on one chip, it then became possible to apply certain performance-enhancing tricks to the implementation, but only if the instruction set was very simple and predictable
  - This was the RISC revolution
  - Its philosophy was to give up "nice", fancy features in order to make common operations fast

Copyright © 2009 Elsevier

Architecture Implementation

- RISC machines:
  - a common misconception is that small instruction sets are the distinguishing characteristic of a RISC machine
  - better characterization: RISC machines are machines in which at least one new instruction can (in the absence of conflicts) be started every cycle (hardware clock tick)
    - all possible mechanisms have been exploited to minimize the duration of a cycle, and to maximize the number of functional units that operate in parallel during a given cycle

Copyright © 2009 Elsevier

Architecture Implementation

- Reduced cycle time comes from making all instructions simple and regular
  - Simple instructions never have to run slowly because of extra logic necessary to implement the complicated instructions
  - Maximal parallelism comes from giving the instructions a very regular, predictable format
    - the interactions between instructions are clear and the processor can begin working on the next instruction before the previous one has finished

Copyright © 2009 Elsevier

ELSEVIER

Compiling for Modern Processors

- PIPELINING is probably the most important performance enhancing trick
  - It works kind of like this:

TIME →

fetch instr	*decode* instr	*fetch* data	*execute*	*store* data		
	fetch instr	*decode* instr	*fetch* data	*execute*	*store* data	
		fetch instr	*decode* instr	*fetch* data	*execute*	*store* data

Copyright © 2009 Elsevier

Compiling for Modern Processors

- The processor has to be careful not to execute an instruction that depends on a *previous* instruction that hasn't finished yet
  - The compiler can improve the performance of the processor by generating code in which the number of dependencies that would *stall* the pipeline is minimized
  - This is called INSTRUCTION SCHEDULING; it's one of the most important machine-specific optimizations for modern compilers

Copyright © 2009 Elsevier

Compiling for Modern Processors

- Loads and load delays are influenced by
  - Dependences
    - Flow dependence
    - Anti-dependence
    - Output dependence
- Branches
  - since control can go both ways, branches create delays

Copyright © 2009 Elsevier

Compiling for Modern Processors

- Usual goal: *minimize pipeline stalls*
- Delay slots
  - loads and branches take longer than *ordinary* instructions
  - loads have to go to memory, which is slow
  - branches disrupt the pipeline
  - processor have interlock hardware
  - early RISC machines often provided *delay slots* for the second (maybe third) cycle of a load or store instruction, during which something else can occur
  - the instruction in a branch delay slot gets executed whether the branch occurs or not

Copyright © 2009 Elsevier

Compiling for Modern Processors

- Delay slots (continued)
  - the instruction in a load delay slot can't use the loaded value
  - as pipelines have grown deeper, people have generally realized that delay slots are more trouble than they're worth
  - most current processor implementations interlock all loads, so you don't have to worry about the correctness issues of load delay
  - some machines still have branch delay slots (so they can run code written in the late '80s)
    - later implementations usually provide a *nullifying* alternative that skips the instruction in the slot if static branch prediction is wrong

Copyright © 2009 Elsevier

Compiling for Modern Processors

- Unfortunately, even this *start a new instruction every cycle* characterization of RISC machines is inadequate
  – In all honesty, there is no good clear definition of what RISC means
- Most recent RISC machines (and also the most recent x86 machines) are so-called SUPERSCALAR implementations that can start *more* than one instruction each cycle

Copyright © 2009 Elsevier

Compiling for Modern Processors

- If it's a CISC machine, the number of instructions per second depends crucially on the mix of instructions produced by the compiler
  - the MHz number gives an upper bound (again assuming a single set of functional units)
  - if it's a "multi-issue“ (superscalar) processor like the PowerPC G3 or Intel machines since the Pentium Pro, the upper bound is higher than the MHz number

Copyright © 2009 Elsevier

Compiling for Modern Processors

- As technology improves, complexity is beginning to creep back into RISC designs
- Right now we see "RISC" machines with on-chip
  - vector units
  - memory management units
  - large caches

Copyright © 2009 Elsevier

Compiling for Modern Processors

- We also see "CISC" machines (the Pentium family) with RISC-like subsets (single-cycle hard-coded instructions)
- In the future, we might see
  - large amounts of main memory
  - multiple processors
  - network interfaces (now in prototypes)
  - additional functions
    - digital signal processing

Copyright © 2009 Elsevier

Compiling for Modern Processors

- In addition, the 80x86 instruction set will be with us for a long time, due to the huge installed base of IBM-compatible PCs
  - After a failed attempt to introduce its own RISC architecture (the i860), Intel has for the last three years been working with HP on the RISC-like Merced, or IA64, architecture, which will remain provide a compatibility mode for older x86 programs

Copyright © 2009 Elsevier

Compiling for Modern Processors

- In a sense, code for RISC machines resembles microcode
  - Complexity that used to be hidden in firmware must now be embedded in the compiler
  - Some of the worst of the complexity (e.g. branch delay slots) can be hidden by the assembler (as it is on MIPS machines)
    - it is definitely true that it is harder to produce good (fast) code for RISC machines than it is for CISC machines

Copyright © 2009 Elsevier

Compiling for Modern Processors

- **Example**: the Pentium chip runs a little bit faster than a 486 if you use the same old binaries
  - If you recompile with a compiler that knows to use a RISC-like subset of the instruction set, with appropriate instruction scheduling, the Pentium can run *much* faster than a 486

Copyright © 2009 Elsevier

Compiling for Modern Processors

- Multiple functional units
  - superscalar machines can issue (start) more than one instruction per cycle, if those instructions don't need the same functional units
  - for example, there might be two instruction fetch units, two instruction decode units, an integer unit, a floating point adder, and a floating point multiplier

Copyright © 2009 Elsevier

Compiling for Modern Processors

- Because memory is so much slower than registers, (several hundred times slower at present) keeping the *right* things in registers is extremely important
  - RISC machines often have at least two different classes of registers (so they don't have to support all operations on all registers) which the compiler has to keep track of

Copyright © 2009 Elsevier

Compiling for Modern Processors

- Some (SPARC) have a complicated collection of overlapping REGISTER WINDOWS
- Finally, good register allocation sometimes conflicts with good instruction scheduling
  - code that makes ideal use of functional units may require more registers than code that makes poorer use of functional units
  - good compilers spend a *great* deal of effort
    - make sure that the data they need most is in register

Copyright © 2009 Elsevier

Compiling for Modern Processors

- Note that instruction scheduling and register allocation often conflict
- Limited instruction formats/more primitive instructions
  - Many operations that are provided by a single instruction on a CISC machine take multiple instructions on a RISC machine
  - For example, some RISC machines don't provide a 32-bit multiply; you have to build it out of 4-bit (or whatever) multiplies

Copyright © 2009 Elsevier

Compiling For Modern Machines

- To make all instructions the same length
  - data values and parts of addresses are often scaled and packed into odd pieces of the instruction
  - loading from a 32-bit address contained in the instruction stream takes two instructions, because one instruction isn't big enough to hold the whole address *and* the code for *load*
    - first instruction loads part of the address into a register
    - second instruction adds the rest of the address into the register and performs the load

Copyright © 2009 Elsevier

Summary

- There are currently four (4) major RISC architectures:
  - ARM (Intel, Motorola, TI, etc)
  - SPARC (Sun, TI, Fujitsu)
  - Power/Power PC (IBM, Motorola. Apple)
  - MIPS (SGI, NEC)
- Currently there is growing demand for 64-bit addressing (Intel, AMD)

Copyright © 2009 Elsevier