

## Processor Architectures

---

Handout written by Maggie Johnson and revised by Julie Zelenski.

### Architecture Vocabulary

Let's review a few relevant hardware definitions:

*register:* a storage location directly on the CPU, used for temporary storage of small amounts of data during processing.

*memory:* an array of randomly accessible memory bytes each identified by a unique address. Flat memory models, segmented memory models, and hybrid models exist which are distinguished by the way locations are referenced and potentially divided into sections.

*instruction set:* the set of instructions that are interpreted directly in the hardware by the CPU. These instructions are encoded as bit strings in memory and are fetched and executed one by one by the processor. They perform primitive operations such as "add 2 to register i1", "store contents of o6 into memory location 0xFF32A228", etc. Instructions consist of an operation code (opcode) e.g., load, store, add, etc., and one or more operand addresses.

*CISC:* Complex instruction set computer. Older processors fit into the CISC family, which means they have a large and fancy instruction set. In addition to a set of common operations, the instruction set has special purpose instructions that are designed for limited situations. CISC processors tend to have a slower clock cycle, but accomplish more in each cycle because of the sophisticated instructions. In writing an effective compiler back-end for a CISC processor, many issues revolve around recognizing how to make effective use of the specialized instructions.

*RISC:* Reduced instruction set computer. Many modern processors are in the RISC family, which means they have a relatively lean instruction set, containing mostly simple, general-purpose instructions. Even basics like multiply and divide may not be present (you have to do repeated addition/subtraction/shift/ etc.) But the clock cycle can be cranked to unbelievable speeds allowing comparable or even better performance than CISC chips overall. (RISC is one of the many innovations we can attribute to our Stanford president). In writing a

compiler back-end for a RISC processor, there are fewer instruction choices to consider, but the scheduling of instructions and use of registers becomes critical because of the fast clock speed.

## Memory Hierarchy

All computers need storage to store and work on data during processing. Ideally, this storage would be quick to access and large in size. But fast memory is expensive and small; slow memory is cheaper and large. To provide the illusion of both fast and large, the memory system of modern computers is organized in a hierarchical way. At the very top of the hierarchy are the CPU registers. Main memory is below that, one or two levels of fast cache memory may be inserted in between those two levels. Via virtual memory, the hard disk can expand the capacity of main memory. A typical memory hierarchy might have five levels:

- registers:* Your average modern processor has 16 to 128 on-board registers, each register can hold a word of memory (typically 32 bits although 64-bit machines are becoming more common). The total register space is thus just a couple hundred bytes total. Time to access registers is perhaps a few nanoseconds. Registers are the most expensive. You cannot expand the available space by adding registers to a chip; they are fixed as part of its design.
- L1 cache:* A level-1 cache is on the same chip as the processor and usually cannot be expanded. Typical L1 caches are 32-64 KB. Time to access the L1 cache is usually tens of nanoseconds. L1 caches are pricey, but not as expensive as registers.
- L2 cache:* A level-2 cache is usually a separate chip from the processor and a typical size is 1-2 MB. The L2 cache size is usually expandable, although expensive. Access time is usually in the range of 10s of nanoseconds and cost is around \$50 per megabyte.
- main memory:* Main memory is what you think of when you say your computer has 128MB of memory. These are DRAM modules that are usually socketed into the board and expansion is a matter of adding modules or replacing with higher-density modules. Today's computers usually have 128-1024 MB of main memory. Access time is in the 50-100 ns range and cost is dropping drastically. (Currently around a dime or two per megabyte.)
- hard disk:* When using your hard disk as back-up, you can further extend memory into the range of many gigabytes. Disks are cheap, just a

fraction of a penny per megabyte, but slow, access times are measured in 10s or 100s of milliseconds.

Summary: you get what you pay for...

### Processor Architectures

A processor's architecture refers to the way in which its memory and control is organized. Most early computers were designed as *accumulator machines*. The processor has a single register called the accumulator where arithmetic, logic and comparison operations occur. All other values and variables are stored in memory and transferred to the accumulator register when needed. These machines are also referred to as one-address machines since the instruction set uses an opcode and a single operand address. Accumulator machines are rarely seen anymore; they reached their heyday in the era of the PDP-8, Zilog Z-80, and Intel 8080.

```
load, MEM(y)           ;; load value of y into accum
add, MEM(z)            ;; add value of z into accum y + z
store, MEM(x)          ;; store value of accum in x (thus x = y + z)
```

Many of today's processors are *general register machines*, which replace the single accumulator with a small set of general registers in which arithmetic, logic and comparison operations occur. Motorola 68K, Intel Pentium, PowerPC, etc. are all general register processors. CISC machines commonly have 8 to 16 general registers, a RISC machine might have two or three times that. Some general register machines are two-address machines since the instruction set uses an opcode and at most two operand addresses. Others use three-address code where the result and two operands can be specified in the instructions. Encoding more addresses into the instruction leaves fewer bits for other information (opcode, subtypes, etc.) which depending on the word size of the machine may be a difficult constraint.

```
load R1, MEM(y)        ;; load value of y into register #1
load R2, MEM(z)        ;; load value of z into register #2
add R1, R2              ;; add reg #1 to reg #2, store result in reg#1
store R1, MEM(x)       ;; store result in z (thus x = y + z)
```

A variant on the general register machine is the *register window machine*, which has a facility for saving and restoring groups of registers to memory, allowing the machine to appear as though it had many more registers that are physically available on the chip, in a sort of "virtual memory" like fashion. Sun Sparc and Knuth's MMIX are examples of register window architectures.

A *stack machine* has no accumulator or registers and instead uses only a stack pointer that points to the top of a stack in memory. All operations except data movement are performed on operands at the top of the stack. Thus, these are referred to as zero-

address machines. The Java virtual machine is an example of a stack machine, so are many HP calculators.

```

push MEM(y)           ;; load value of y onto top of stack
push MEM(z)           ;; load value of z onto top of stack
add                   ;; pop top 2 operands, add, and push result
store MEM(x)          ;; store top in z (thus x = y + z)
pop                   ;; remove result from stack

```

Which of the architectures do you think is the fastest? Which is in most common usage?

### Addressing Modes

The *addressing mode* refers to the way an operand is specified. In other words, "How does the add instruction identify which numbers to add?" There are dozens of possible addressing modes, but they all generally fall into one of the categories below. Under what circumstances would each of these be useful?

*direct*: a register contains the operand value

*indirect*: a register contains an address that specifies the location holding the value

*indexed*: an indirect reference (address) that is offset by some constant or by the contents of another register

Some of the more exotic ones might include indirect with offset and update which is available on the PowerPC, to access an indexed value and update the offset by some constant increment or decrement. Such a specialized instruction would be useful in translating a loop that accesses each member of an array in turn.

### A Simple Assembly Language

In order to understand the back-end of a compiler, we need to review a bit of assembly language. Let's start by defining a simple assembly language for an accumulator machine. Our mythical machine has a single ALU register R with three condition codes (GT, EQ, LT) used for comparison results. The instructions are encoded using a 4-bit opcode, and we have exactly 16 different instructions in our set:

op code	op code mnemonic	meaning
0000	load x	$R = \text{Mem}(x)$
0001	store x	$\text{Mem}(x) = R$
0010	clear x	$\text{Mem}(x) = 0$
0011	add x	$R = R + \text{Mem}(x)$
0100	increment x	$\text{Mem}(x) = \text{Mem}(x) + 1$
0101	subtract x	$R = R - \text{Mem}(x)$
0110	decrement x	$\text{Mem}(x) = \text{Mem}(x) - 1$

0111	compare x	Sets condition code of ALU if Mem(x) > R then GT = 1 if Mem(x) = R then EQ = 1 if Mem(x) < R then LT = 1
1000	jump x	set PC (program counter) to location x
1001	jumpgt x	set PC to x if GT = 1
1010	jumpeq x	set PC to x if EQ = 1
1011	jumplt x	set PC to x if LT = 1
1100	jumpneq x	set PC to x if EQ = 0
1101	in x	read standard input device, store in Mem(x)
1110	out x	output Mem(x) to standard output device
1111	halt	stop program execution

A *pseudo-op* is an entry that does not generate a machine language instruction but invokes a special service of the assembler. Pseudo-ops are usually preceded by period.

.data	builds signed integers
FIVE: .data 5	generate binary representation for 5 put it in next available memory location make label "FIVE" equivalent to that memory address

Data is usually placed after all program instructions, so the format of an executable program is:

```
.begin
    assembly instructions
halt
    data generation pseudo-ops
.end
```

### Example Assembly Programs

Some simple arithmetic:

<i>Source program</i>	<i>Generated assembly</i>
b = 4	.begin
c = 10	load FOUR
a = b + c - 7	store b
	load TEN
	store c
	load b
	add c
	subtract SEVEN
	store a
	halt

```

a:      .data  0
b:      .data  0
c:      .data  0
FOUR:   .data  4
TEN:    .data 10
SEVEN:  .data  7
        .end

```

An example with an if/else, and using input and output:

<i>Source program</i>	<i>Generated assembly</i>
Read x	.begin
Read y	in x
if x >= y then	in y
Print x	load y
Else	compare x
Print y	jumplt PRINTY
	out x
	jump DONE
	PRINTY: out y
	DONE:  halt
	x:      .data  0
	y:      .data  0
	.end

An example of a simple loop:

<i>Source program</i>	<i>Generated assembly</i>
Set sum to 0	.begin
Read n	clear sum
repeat until n < 0 {	in n
add n to sum	AGAIN: load ZERO
Read n	compare n
}	jumplt NEG
Print sum	load sum
	add n
	store sum
	in n
	jump AGAIN
	NEG:  out sum
	halt
	sum:  .data  0
	n:    .data  0
	ZERO: .data  0
	.end

## The Assembler

Before we can execute the program, we must convert the assembly to machine code. The assembly code is already machine-specific, so it doesn't need much conversion, but it is expressed using symbolic names (e.g., load, store, add, etc.) and labels. These need to be translated to actual memory locations and the instructions need to be binary encoded in the proper format. The *assembler* is the piece that does this translation. The assembler's tasks are:

- 1) convert symbolic addresses to binary: each symbol is defined by appearing in label field of instruction or as part of `.data` pseudo-op. In an assembler, two passes over the code are usually made; the first pass is for assigning addresses to each instruction and for building a symbol table where it stores each symbol name and the address it just assigned.

```

                .begin
LOOP:          in x           0 (location to this instruction)
                in y           2
                load x         4
                compare y      6
                jumpgt DONE    8
                out x          10
                jump LOOP      12
DONE:          out y          14
                Halt           16
x:             .data 0        18
y:             .data 0        20

```

After the first pass the symbol table contains:

Symbol	Address
LOOP	0
DONE	14
X	18
Y	20

- 2) During the second pass, the source program is translated into machine language: the opcode table is used to translate the operations, and the symbol table is used to translate the symbolic addresses. Converting the symbolic opcodes to binary is basically just a lookup of name to find the matching code and then encoding the rest of the instruction (operands, subtype, etc.).

For example, if the instruction format is 4-bit opcode followed by 12-bit address, the first instruction is encoded as:

```

in x           1101  0000 0001 0010
                opcode  operand address

```

- 3) perform assembler service requested by pseudo-ops, such as laying out the contents of the data segment and initializing any necessary values.
- 4) output translated instructions to a file. During the second pass, the assembler converts data constants to binary and writes each instruction to a file. This file is called an object file. Some sort of header is usually included in the file to identify the boundaries of the data and text segments, which will be needed by the linker and loader.

### **The Linker**

The linker is a tool that takes the output file(s) from the assembler and builds the executable. The linker is responsible for merging in any library routines and resolving cross-module references (i.e., when a function or global variable defined in one translation unit is accessed from another). An executable is said to be *statically* linked if all external references are resolved at link-time and the executable contains all necessary code itself. For executables that need large libraries (e.g., Xwindow routines), it is common to *dynamically* link the executable to those libraries so that the executable does not contain the contents of the library, but instead references to it will be resolved at runtime on an as-needed basis.

### **The Loader**

The *loader* is the part of the operating system that sets up a new program for execution. It reads instructions from the object file and places them in memory. An *absolute* loader places each line from the object file at the address specified by the assembler (usually just a sequence from 0 upwards). (You might ask yourself: how might a relocatable loader work?) When this is completed, it places the address of the first instruction in the program counter register. The loader also usually sets up the runtime stack and initializes the registers. At that point the processor takes over, beginning the "fetch/decode/execute" cycle.

### **Bibliography**

- A. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Reading, MA: Addison-Wesley, 1986.
- J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.
- J.P. Hayes, Computer Architecture and Organization. New York, NY: McGraw-Hill, 1988.
- A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.
- G.M.Schneider, J. Gersting, An Invitation to Computer Science, New York: West, 1995.
- J. Wakerly, Microcomputer Architecture and Programming. New York, NY: Wiley, 1989.