CS10 – Computer Architecture and Organization

My Notes

Leonard Mohr

Student at Foothill College

leonardmohr@gmail.com

Contents

Section 1

Computer Abstractions and Technology

Subsection 1.1

Introduction

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

Figure 1. We can describe storage in binary or decimal notation. We are much more familiar with the decimal term. Note also the size difference between the two: binary term gets progressivly larger.

1.1.1 Program Performance

One of the main goals for both the hardware designer and the software designer is to improve performance. This can be achieved in different ways (just think about how quick sort is much faster than bubble sort, and the apple M1 processor is much faster than the intel processors they replaced).

Hardware or software component	How this component affects performance	Where is this topic covered?
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed	Other books!
Programming language, compiler, and architecture	Determines the number of computer instructions for each source-level statement	Chapters 2 and 3
Processor and memory system	Determines how fast instructions can be executed	Chapters 4, 5, and 6
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	Chapters 4, 5, and 6

Figure 2. Here we can see the various ways program performance can be improved.

Check Yourself

1. As mentioned earlier, both the software and hardware affect the performance of a program. Can you think of examples where each of the following is the right place to look for a performance bottleneck?

   a) The algorithm chosen

   > If a program that sorts a really long list of names is taking a long time, you might want to look at the algorithm being used.

   b) The programming language or compiler

> If your program could is not language dependent, you could look at using a compiled language (like C), as opposed to an interpreted language like Python.

c) The operating system

> If one program runs well, but two at a time don't, then maybe the operating system isn't distributing it's resources efficiently.

d) The processor

> If the computer in general is using a lot of energy, you might want to look at the processor. Similarily, if you are doing compute heavy tasks such as vdeo editing, you might need a processor upgrade.

e) The I/O system and devices

> If it takes a long time to write to a hard drive, you might look at upgrading to a SSD.

Subsection 1.2

Eight Great Ideas in Computer Architecture

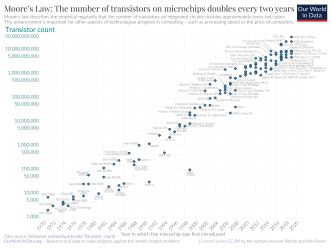

Figure 3. Moore's Law in action.

1. **Moore's Law**
   Moore's Law resulted from a 1965 prediction that integrated circuit (IC) resources would double every 18-24 months.

2. **Use Abstraction to Simplify Design**
   To increase productivity, by design both computer architects and programmers try use abstractions to hide the lower-level details and provide a simpler to work with. For example, the operating system abstracts away the complexity of the memory system so that programs are provided with a much simplified view of the memory, and the details are handled by the operating system.

3. **Make the Common Case Fast**
   Better performance gains can be made if you optimize for what the prgram is going to do most often.

4. **Performance via Parallelism**
   We can make a program a lot faster by performing multiple tasks at once. This is especially true with the advent of multi-core processors.

5. **Performance via Pipelining**
   If you are moving a lot of bricks from one place to another (using just manpower) it would be a lot more efficient to set up a line of people, and pass the bricks down the line, than to just have everyone running back and forth. The same principle can be used in computers.

6. **Performance via Prediction**
   When a processor encounters an if statement, it might say that on average the result is true, and procede like it is, so that it keep going, and then the if qualifier can be processed at a later date.

7. **Hierarchy of Memories**
   By using different types of memory, from really small and really fast, to really large and slow, a memory hierarchy is created. Caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy.

8. **Dependability via Redundancy**
   Because we are sad when a computer dies, we want to prevent the death of the computer. To do this we make them dependable. This can be achieved by including redundant components that can both take over in the event of a failure as well as detect when a failure has occured.

Subsection 1.3
Below Your Program

The Operating System is a great example of abstraction. Some of its most important functions are:

- Handling basic input and output operations
- Allocating storage and memory
- Provided for protected sharing of the computer among multiple applications using it simultaneously.

Another example of abstraction is high-level programming languages like C. When computers first came about, programmers programmed in binary; they wrote their programs in 1's and 0's (the language of the computer). Since that was tedious, they invented the assembler which would convert assembly language into binary code. Next came the compiler which would convert higher order languages into assembly.

Another benefit of the programming languages is it allows one to chose the specific language that is best for the task. Also, programs don't have to be written for a specific processor, since the compiler and assembler can package it for different computers.

Subsection 1.4
Under the Covers

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. This organization is independent of hardware technology: you can place every piece of every computer, past and present, into one of these five categories.

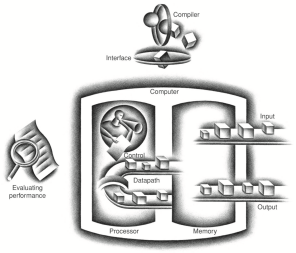

Figure 4. Here we see the flow of information in a computer. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

1.4.1 Parts of the Computer

- Integrated Circuit: Also called a chip. A device combining dozens to millions of transistors.
- Central Processing Unit (CPU): Also called the processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.
- Datapath: The component of the processor that performs arithmetic operations
- Control: The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.
- Memory: The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

- Dynamic Random Access Memory (DRAM): Memory built as an integrated circuit; it provides random access to any location Access times are 50 nanoseconds and cost per gigabyte in 2012 was $5 to $10.
- Cache Memory: Consists of a small, fast memory that acts as a buffer for the DRAM memory.
- Static Random Access Memory: SRAM is faster but less dense, and hence more expensive, than DRAM.
- Instruction Set Architector: The interface between the hardware and the lowest-level software. This includes all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.
- Application Binary Interface (ABI): The user portion of the instruction set plus the operating system interfaces used by application programmers. It defines a standard for binary portability across computers.

Subsection 1.6

Performance

When talking about computers, we often want to look at the performance of the computer. But how can we define performance?

Definition 1

Response Time
Also referred to as **execution time**. This is the total time required for the computer to complete a task, including disk access, memory aaccesses, I/O activities, operating system overhead, CPU execution time, and so on.

Definition 2

Throughput / Bandwidth
This measures the number of tasks computed per unit time.

For the time being, we will mostly be looking at response time. So that a higher number represents a machine with better performance we will say that

$$\text{Performance}_{\text{X}} = \frac{1}{\text{Execution time}_{\text{X}}}.$$

Also, we often want to say that computer "X is n times as fast as Y", to compute n we say:

$$\frac{\text{Performance}_{\text{X}}}{\text{Performance}_{\text{Y}}} = n.$$

CPU execution time or simply **CPU time** is the actual time the CPU spends computing for a specific task.

1.6.1 Clock Rate and Clock Period

Designers refer to the length of a **clock period** both as the time for a complete clock cycle (e.g., 250 picoseconds) and as the *clock rate* (e.g., 4 gigahertz, or 4GHz), which is the inverse of the clock period.

For example a clock rate of

$$\begin{aligned} 4\text{GHz} &= 4,000,000,000 \frac{\text{cycles}}{\text{sec}} \\ &\equiv \frac{1 \text{ cycle}}{4,000,000,000 \text{ second}} \\ &= 0.00000000025 \frac{\text{cycle}}{\text{second}} \\ &= \frac{1}{250} \frac{\text{cycle}}{\text{picosecond}}. \end{aligned}$$

We can relate clock cycles and clock cycle time to CPU time:

$$\begin{matrix}\text{CPU execution time} \\ \text{for a program}\end{matrix} = \begin{matrix}\text{CPU clock cycles} \\ \text{for a program}\end{matrix} \times \text{Clock cycle time}$$

or alternatively,

$$\begin{matrix}\text{CPU execution time} \\ \text{for a program}\end{matrix} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate.}}$$

1.6.2 Instruction Performance

In addition to thinking about clock cycles and CPU time, we also need to think about number of instructions there are in a program and the time each instruction takes:

$$\text{CPU clock cycles} = \text{Instructions for a Program} \times \frac{\text{average clock cycles}}{\text{instruction}}$$

Clock Cycles per Instruction (CPI) is the average number of clock cycles per instruction for a program or program fragement.

1.6.3 The Classic CPU Performance Equation

Putting everything from above together we have

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

and

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}.$$

We also have

$$\text{Time} = \text{Seconds / Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle.}}$$

Processors these days can vary their clock rates. For example Intel Core i7 chips temporarily increase clock rate by 10% until the chip gets too warm. Thus we need to use the average clock rate for a program.

Figure 5.

Subsection 1.10

Fallacies and Pitfalls

1.10.1 Amdahl's Law

Definition 3 **Amdahl's Law**
A rule stating that the performance enhancement possible with a given improvement

is limited by the amount that the improved feature is used:

$$\begin{aligned}\text{Execution time after improvement} = &\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} \\ &+ \text{ Execution time unaffected}\end{aligned}$$

where Amount of improvement $= n$.

We thus see that there is only so much benefit that can be achieved by improving a part of the program that is rarely used. For this reason we want to make the common case fast!

1.10.2 MIPS

One should be wary about using only a subset of the performance equation as a performance metric; one can't determine performance just by looking at clock rate, instruction count, or CPI alone.

An alternative to time is **MIPS (million instructions per second)**:

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}.$$

There are a couple problems with MIPS:

- MIPS specifies the instruction execution rate but does not take into account the capabilities of these instructions. Therefore we can't compare computers with different instruction sets.
- MIPS varies between programs on the same computer; thus a computer cannot have a single MIPS rating.

$$\text{MIPS} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}.$$

1.10.3 Check Yourself

Consider the following performance measurements for a program:

Measurement	Computer A	Computer B
Instruction count	10 billion	8 billion
Clock rate	4 GHz	4 GHz
CPI	1.0	1.1

Which computer is faster and which has the higher MIPS rating?

First looking at computer A.

$$\begin{aligned}\text{Time}(A) &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}} \\ &= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}} \\ &= \frac{10 \times 10^9 \text{ instructions} \times \frac{1 \text{ cycle}}{\text{instruction}}}{4 \times 10^9 \frac{\text{cycles}}{\text{second}}} \\ &= 2.5 \text{ seconds}.\end{aligned}$$

$$\begin{aligned}\text{MIPS}(A) &= \frac{\text{Instruction Count}}{\text{Execution Time } \times 10^6} \\ &= \frac{10 \times 10^9 \text{ instructions}}{2.5 \text{ seconds } \times 10^6} \\ &= \frac{4 \times 10^3 \text{ million instructions}}{\text{second}}.\end{aligned}$$

Now looking at computer B.

$$\begin{aligned}\text{Time}(B) &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}} \\ &= \frac{\text{Instruction Count } \times \text{ CPI}}{\text{Clock Rate}} \\ &= \frac{8 \times 10^9 \text{ instructions} \times \frac{1.1 \text{ cycles}}{\text{instruction}}}{4 \times 10^9 \frac{\text{cycles}}{\text{second}}} \\ &= 2.2 \text{ seconds}.\end{aligned}$$

$$\begin{aligned}\text{MIPS}(B) &= \frac{\text{Instruction Count}}{\text{Execution Time } \times 10^6} \\ &= \frac{8 \times 10^9 \text{ instructions}}{2.2 \text{ seconds } \times 10^6} \\ &\approx \frac{3.6 \times 10^3 \text{ million instructions}}{\text{second}}.\end{aligned}$$

We thus see that Computer A has a higher MIPS score but runs slower.

Section 2

Instructions: Language of the Computer

Subsection 2.2

Operations of the Computer Hardware

2.2.1 MIPS Operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

2.2.2 MIPS Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Note how just about all operations have exactly three operands. This conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number.

Subsection 2.3

Operands of the Computer Hardware

2.3.1 Memory Operands

Let's say we have the following C statement

```
1  g = h + A[8];
```

What will be the associated MIPS code if g and h are in registers $\$s1$ and $\$s2$, and that the base address of the array is in $\$s3$.

```
1  lw    $t0, 32($s3)     # Temporary reg $t0 gets A[8]
2  add   $t0, $s2, $t0    # Temporary reg $t0 gets h + A[8]
3  sw    $t0, 48($s3)     # A[12] <—— $t0
```

Note that MIPS uses byte addressing, and so to get to the 8th index, you need to add $8 * 4$ since the size of each array index is 4 bytes. Also note that words must start with addresses that are multiples of 4.

Alignment Restriction is the requirement that data be aligned in memory on natural boundaries.

2.3.2 Constant or Immediate Operands

Let's say for example that we want to add 5 to some register for whatever reason, instead of loading that from a memory location into a temporary register, and then adding the temporary register to the desired register we can use the instruction add immediate:

```
1  addi    $s3, $s3, 4        # $s3 = $s3 + 4
```

By including this constant opperation the processor can operations much faster and using less energy. Because more than half of MIPS arithmetic instructions have a constant as an operand when running the SPEC CPU2006 benchmarks this is an example of making the common case fast.

Overflow occurs when after performing an arithmatic operation on two numbers results in a number that can't be stored in the number of bits available to register (32 in case of MIPS).

Subsection 2.4

Signed and Unsigned Numbers

2.4.1 Unsigned Numbers

In any number base, the value of the ith digit d is

$$d \times \text{ Base}^i.$$

Thus for example, in binary

$$\begin{aligned} 101 &= (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) \\ &= 4 + 0 + 1 \\ &= 5. \end{aligned}$$

2.4.2 Twos Complement

The idea of twos complement is that the most significant bit indicates the sign of the number, 1 if negative and 0 if positive (zero being treated as a positive number). But instead of merely reprsenting the sign, it represents the negative equivalent value of that number. So for example

$$\begin{aligned} 1 \text{ represents } &-1 \\ 10 \text{ represents } &-2 \\ 100 \text{ represents } &-4 \end{aligned}$$

and the other bits are their normal positive values:

$$\begin{aligned} 11 \text{ represents } &-1 \\ 111 \text{ represents } &-1 \\ 101 \text{ represents } &-3 \\ 110 \text{ represents } &-2. \end{aligned}$$

To achieve two's complement representation you

1. Start with the equivalent positive number
2. Invert all bits (change every 0 to 1, and every 1 to 0)

3. Add 1 to the inverted number, ignoring overflow.

The reason this works is because $x + \bar{x} = -1$ and therefore $\bar{x} + 1 = -x$, where $\bar{x}$ is x inverted. You can use this process to convert from negative to positive and vice versa:

- Twos complement representation of -5

$$\begin{aligned} 0101 &= 5 \\ 1010 &= \bar{5} && \textit{Inverse of 5} \\ 1011 &= -5 && \textit{-5 represented in twos complement} \end{aligned}$$

- Twos complement representation of 5

$$\begin{aligned} 1011 &= -5 \\ 0100 &= \overline{-5} \\ 0101 &= 5. \end{aligned}$$

In two's complement, if you want to use more bits to represent the same number, you just use **sign extension** (repeat the most significant bit):

$$\begin{aligned} 0111 &\equiv 7 \equiv 0000\ 0111 \\ 1011 &\equiv -5 \equiv 1111\ 1011. \end{aligned}$$