

Design guided by measured performance.

Covered:

- Design Principles (1.6)
- Components of CPU Performance (Quantitative Principles) (1.6)
- Benchmarks (1.5)

(Numbers refer to book sections.)

Principles computer designers apply widely.

- Make the common case fast.
Obviously.
- Amdahl's Law: Don't make common case too fast.
As speed of one part increases...
...impact on total performance drops.
- Locality of Reference.
Temporal: It might happen again soon.
Spatial: It might happen to your neighbors soon too.

Make the common case fast.

Consider a system with many parts.

Consider design options for a part:

Simple Design:

- Can be quickly explained, compactly represented.

- Design/validation completed quickly.

- Resulting design may operate slowly.

Sophisticated Design:

- Design must consider multiple situations, has lengthy description.

- Design time longer.

- Validation time (if done properly) much longer.

When design time is limited by a deadline:

- Use simple, but correct designs for all parts.
- Then make the most common cases fast.

Amdahl's Law: don't make common case too fast.

Speedup as a Performance Measure

Many systems evaluated by execution time.

New design compared to old one by how much faster it is, its *speedup*.

Let t_T denote the speed of the old system and $t_{T'}$ the speed of the new one.

Then speedup is defined to be $S_T = \frac{t_T}{t_{T'}}$.

For example, if $S_T = 3$ then new system takes one third the time.

Consider a system with multiple parts ...
... one of which is being considered for improvement.

Amdahl's law relates speedup of part to speedup of system.

Let t_B and $t_{B'}$ denote speed of part B before and after improvement.

Let S_B denote speedup of part B.

Define $f = \frac{t_B}{t_T}$ to be the fraction of time contributed by B.

Speedup can be written: $S_T = \frac{t_T}{(1-f)t_T + t_{B'}}$

Applying $t_{B'} = \frac{t_B}{S_B} = \frac{ft_T}{S_B}$ yields $S_T = \frac{t_T}{(1-f)t_T + ft_T/S_B}$

Cancelling, $S_T = \frac{1}{(1-f) + f\frac{1}{S_B}}$

When f is close to zero, impact of S_B is small.

The first two principles are “common sense”.

However, locality is a characteristic of executing programs ...

... which has held and is expected to continue to hold.

Because many designs work only when locality is present ...

... if it all of a sudden programs did not exhibit locality ...

... computers would run them many, many times slower!

Locality usually applied to memory addresses issued by processor.

Temporal: there's a good chance that an address used will be used again soon.

Spatial: once an address is used there's a good chance a nearby address will be used.

For examples, analyze execution of almost any program.

CPU Performance Decomposed into Three Components:

- Clock Frequency (ϕ)
Determined by technology and influenced by organization.
- Cycles per Instruction (CPI)
Determined by organization and instruction mix.
- Instruction Count (IC)
Determined by program and ISA.

These combined to form *CPU Performance Equation*

$$\boxed{t_T = \frac{1}{\phi} \times \text{CPI} \times \text{IC}},$$

where t_T denotes the execution time.

Given a program there are two ways instructions could be tallied:

- *Static Instruction Count*

The number of instructions making up the program.

- *Dynamic Instruction Count (IC)*

The number of instructions executed in a run of the program.

For estimating performance, dynamic instruction count is used.

Example, C program that computes $a = \sum_{i=0}^9 i$.

(For simplicity, treat each line as an instruction.)

IC	Line	Code
1	1	<code>a = 0;</code>
1	2	<code>for(i = 0;</code>
11	3	<code> i < 10;</code>
10	4	<code> i++)</code>
10	5	<code> a = a + i;</code>

Static count: 5 (number of lines).

Dynamic count: 33.

Example, assembler program that computes $a = \sum_{i=0}^9 i$.

Written in Simplescalar assembler.

Loop executes slightly differently than previous one.

IC			
1	move	r5, r0	! r0 is always zero.
1	move	r3, r0	
	L23:		! Branch label.
10	addu	r5, r5, r3	! Add unsigned.
10	addu	r3, r3, 1	
10	slt	r2, r3, 10	! r2 = r3 < 10
10	bne	r2, r0, L23	! Branch to L23 if r2 not equal 0.

Static count: 6 (number of instructions).

Dynamic count: 42.

CPUs implemented using synchronous clocked logic.

Typical Clock Cycle

- When clock switches from low to high work starts.
- While clock is high work proceeds.
- When clock goes from high to low work should be complete.

Clock frequency determined by *critical path*.

Critical Path:

Logic doing most time consuming work (in a cycle).

If clock frequency is too high work will not be completed ...
... and so system will not perform properly.

For high clock frequencies, keep critical paths short.

Cycles (clocks) per Instruction (CPI)

Oversimplified definition: (CPI)

Number of cycles needed to execute an instruction.

Better definition: (CPI)

Number of cycles to execute some code divided by number of instructions.

Difference:

Interested in rate at which instructions executed in program ...

... not time time for any one instruction.

Analogy: A restaurant chef preparing meals

Chef may be simultaneously preparing several orders ...

... some take more time than others ...

... some combinations are more time consuming, etc.

Knowing crawfish étouffée takes 30 minutes to prepare ...

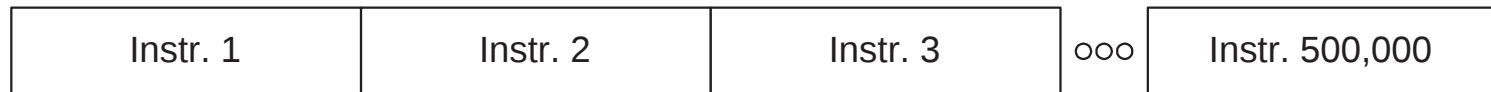
... cannot alone predict how long it would take to serve 20 diners.

Possible Execution (Once upon a time.)

In program order ...

... one at a time.

Time/cycles:	0	1	2	3	4	5	6	7	8	9	10	11	1,999,996
Time/mms:	0				80				160				39,999,920



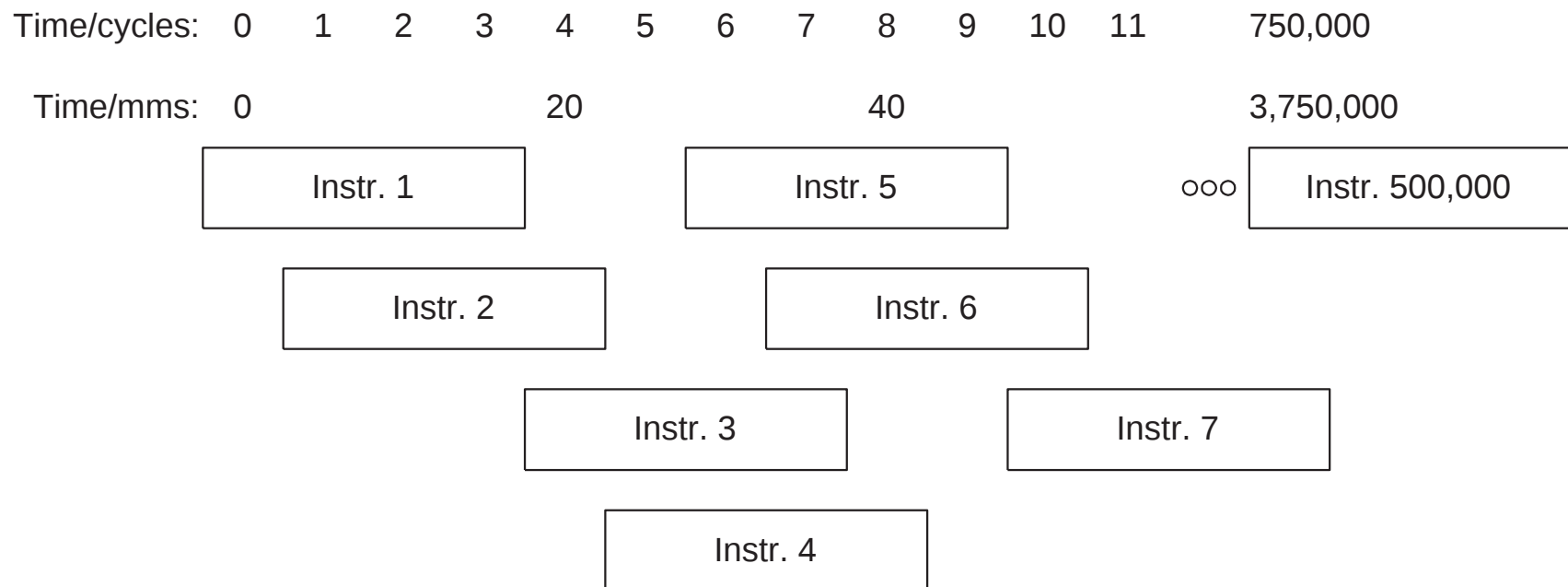
Execution time: $IC \times CPI \times \text{clock period}$.

CPI is number of cycles for each instruction.

To Run Faster: Overlap Instructions (*Pipelined Execution*)

Result must be same as one-at-a-time execution ...

... not too difficult to achieve.



Execution time at best: $IC \times \text{clock period}$...

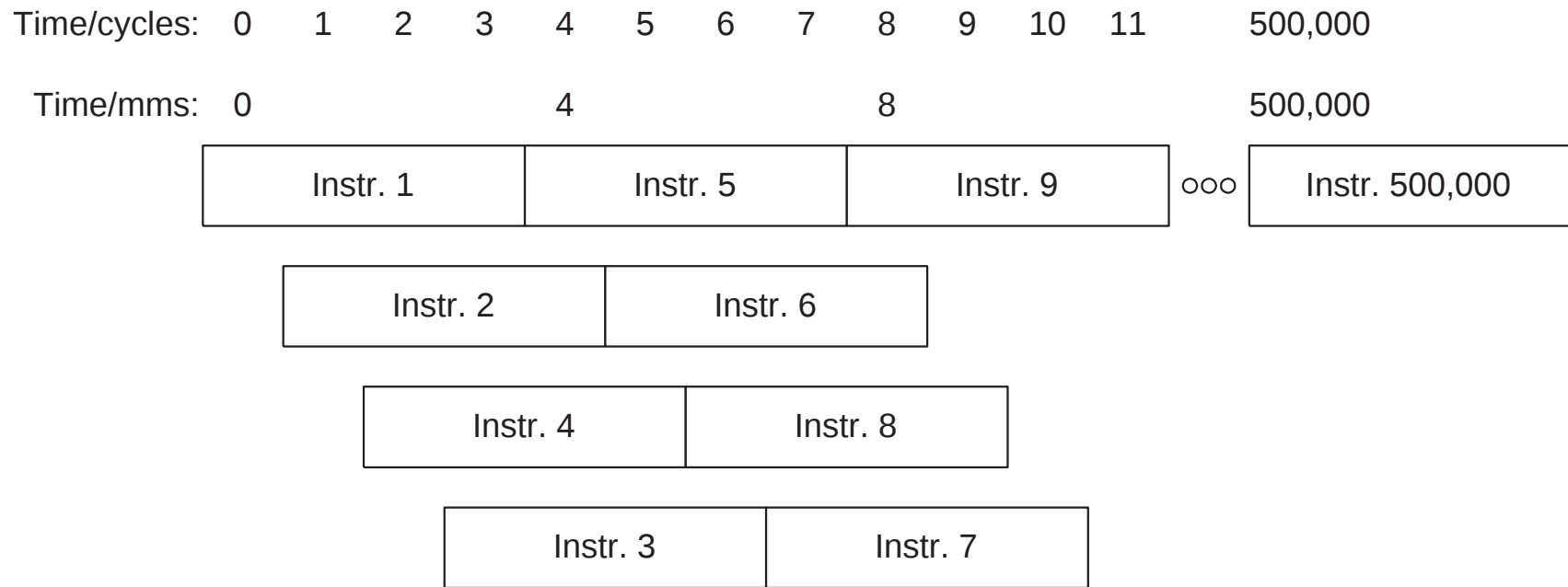
... assuming 1 cycle to start each instruction ...

... instruction can start each cycle.

Execution: Pipelined, Out of Order

To Run Even Faster: Overlap Instructions and Start Out of Order

Sometimes skip an instruction and execute it later.

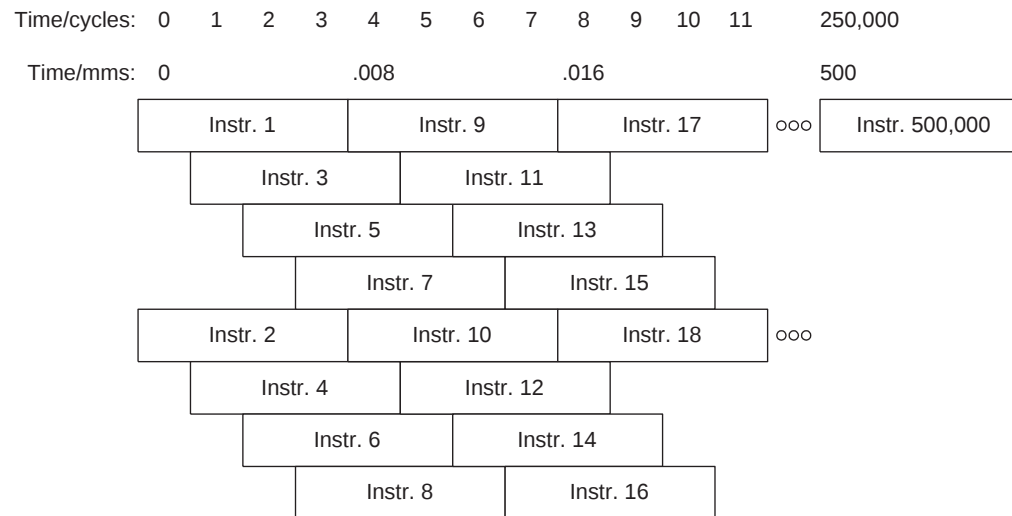


Execution time at best: $IC \times \text{clock period} \dots$
... assuming 1 cycle to start each instruction ...
... instruction can start each cycle.

Same ideal execution time as pipelined, In-Order, but better on average.

To Run Fastest¹: Overlap, Out-of-Order, Start n per Tick (n -Way Superscalar).

Requires about n times as much hardware.



Execution time at best: $\frac{1}{n} \times \text{IC} \times \text{clock period} \dots$
 $\dots$ assuming 1 cycle to start each instruction $\dots$
 $\dots$ instruction can start each cycle.

¹ Using a conventional serial instruction set architecture.

Tradeoffs between Clock Frequency, CPI, and Instruction Count

Increasing Clock Frequency ...

... reduces the work that can be done in a clock cycle ...

... forcing designers to choose higher-CPI designs.

Reducing IC (by adding “powerful” instructions to ISA) ...

... may force implementors to increase CPI or lower clock frequency.

Balancing these is an important skill in computer design.

Since the ISA is usually fixed, IC is less of a factor.

Company X is considering two clock frequencies for its next processor, 500 MHz or 300 MHz. A 500 MHz implementation would execute instructions at 1.7 CPI, the 400 MHz part at 1.1 CPI. Which would be faster?.

500 MHz execution rate: $500 \times 10^6 / 1.7 = 294.1 \times 10^6$

400 MHz execution rate: $400 \times 10^6 / 1.1 = 363.3 \times 10^6$.

The lower clock rate would nevertheless execute faster.

Perhaps because at 500 MHz too much work had to be split into multiple cycles.

An ISA contains many instructions ...

... execution characteristics differ.

E.g., division takes longer than add.

To account for this instruction count can be partitioned.

For example, $IC = IC_1 + IC_2 + IC_3$,

where IC_x is the number of instructions of class x in the execution of some program.

Choosing Instruction Classes

Option: a class for every instruction in the ISA.

This would be tedious to work with.

Option: classes for instructions sharing execution characteristics.

Example: a class for all memory instructions, class for all integer instructions, etc.

Since instructions in class similar, no need to consider separately.

Similarly, CPI can be partitioned by class.

The CPU Performance Equation can then be written:

$$t_T = \frac{1}{\phi} \sum_i IC_i \times CPI_i.$$

Design Tradeoffs Using Instruction Classes

Change may affect one class but not another.

CPU Performance Equation, $\frac{1}{\phi} \sum_i IC_i \times CPI_i$, shows impact.

Impact of changes of different instructions can be estimated.

Note: unlike case without instruction classes ...

... impact computed depends on program being analyzed.

E.g., you're out of luck ...

... if $IC_1 = 0$ and change reduced only CPI_1 .

(The classless CPI is really an average for a typical program.)

Instruction counts by class can guide designer's effort:

First consider instructions in class i , where $IC_i \geq IC_x$.

Design changes can even influence IC (without changing ISA):

Suppose CPI_1 was reduced to a really low value.

Then programmer might re-write code so $IC_1 \gg 0$.

Result might be faster execution with modified program.

But performance benefit must be high enough to justify re-coding.

Assumption

IC is based on output of a good compiler.

Compiler is tuned for a particular implementation.

Two Cases

1. Same ISA, different implementation.
2. Different ISA, (and of course) different implementation.

Case 1: Same ISA, different implementation.

Newer implementation may have lower CPI on existing code ...
... but even better performance attainable by recompiling ...
... which may *increase* CPI.

Compiler writer selects instructions based on performance of implementation.

Consider two implementations:

Implementation A: add CPI 1 cycle, mul CPI 5 cycles.

Implementation B: add CPI 1 cycle, mul CPI 2 cycles.

```
! Call original value of r1, x.  Code computes 6x.
```

```
! Code For Implementation A
```

```
add  r1, r1, r1 ! r1 = 2x
```

```
add  r2, r1, r1 ! r2 = 4x
```

```
add  r1, r1, r2 ! r1 = 6x
```

```
! Code For Implementation B.
```

```
mul  r1, r1, 6  ! r1 = 6x.
```

Implementation A: IC = 3, CPI = 1 (Computing CPI will be covered later.)

Implementation B: IC = 1, CPI = 2.

Implementation B is faster despite higher CPI.

Code compiled for B will run slowly on A.

Case 2: Different ISA, (and of course) different implementation.

Major tradeoffs in complexity and speed.

Consider two implementations:

Implementation A: CPI: load, 2; add and store, 1.

Implementation B: CPI: add (doing load and store), 4.

```
! Code for implementation A.
load  r1, [r2]      ! Load r1 with data at address in r2.
add   r3, r1, r4    ! r3 = r1 + r4
store [r2], r3      ! Store r3 at address in r2.

! Code for implementation B.
add [r2], r4, [r2]
```

Execution time same.

Implementation A: $IC = 3$, $CPI = \frac{4}{3}$.

Implementation B: $IC = 1$, $CPI = 4$.

Benchmark:

program used to evaluate performance.

Uses

- Guide computer design.
- Guide purchasing decisions.
- Marketing tool.

Guiding Computer Design

Measure overall performance.

Determine characteristics of programs.

E.g., frequency of floating-point operations.

Determine effect of design options.

Important: Choice of programs for evaluation.

Optimal but unrealistic:

The exact set of programs customer will run.

Problem: computers used for different applications.

Therefore, must model typical users' workload.

Options:

Real Programs

Programs chosen using surveys, for example.

- + Measured performance improvements apply to customer.
- Large programs hard to run on simulator. (Before system built.)

Kernels

Use part of program responsible for most execution time.

- + Easier to study.
- Not all program have small kernels.

Toy Benchmarks

Program performs simplified version of common task.

- + Easier to study.
- May not be realistic.

Synthetic Benchmarks

Program “looks like” typical program, but does nothing useful.

+ Easier to study.

– May not be realistic.

Commonly Used Option

Overall performance: real programs

Test specific features: synthetic benchmarks.

Definition: a named set of programs used to evaluate a system.

Typically:

- Developed and managed by a publication or non-profit organization.
E.g., Standard Performance Evaluation Corp., PC Magazine.
- Tests clearly delineated aspects of system.
E.g., CPU, graphics, I/O, application.
- Specifies a set of programs and inputs for those programs.
- Specifies reporting requirements for results.

What Suites Might Measure

- Application Performance
E.g., productivity (office) applications, database programs.
Usually tests entire system.
- CPU and Memory Performance
Ignores effect of I/O.
- Graphics Performance

Example, SPEC 95 Suites

Respected measure of CPU performance.

Managed by Standard Performance Evaluation Corporation,...

...a non-profit organization funded by computer companies.

Measures CPU and memory performance on integer and FP code.

Uses common Unix programs such as perl, gcc, compress.

Requires that results on each program be reported.

Programs compiled with publicly available compilers and libraries.

Programs compiled with and without expert tuning.

SPEC 95 Suites and Measures

CINT95 suite of integer programs run to determine:

- SPECint95, execution time of tuned code.
- SPECint_base95, execution time of untuned code.
- SPECint_rate95, throughput of tuned code.

CFP95 suite of floating programs run to determine:

- SPECfp95, execution time of tuned code.
- SPECfp_base95, execution time of untuned code.
- SPECfp_rate95, throughput of tuned code.

Other Examples

BAPCO Suites, measure productivity app. performance on Windows 95.

TPC, measure “transaction processing” system performance.

WinMARK, graphics performance.