Components of CPU Performance and Performance Equation

02-1

Why is my computer fast (or slow)?

Would it help to improve ☐?

CPU performance equation is one way to start answering these questions.

02-2

CPU Performance Decomposed into Three Components:

- *Clock Frequency* (ϕ)
  Determined by technology and influenced by organization.
- *Clocks per Instruction* (CPI)
  Determined by ISA, microarchitecture, compiler, and program.
- *Instruction Count* (IC)
  Determined by program, compiler, and ISA.

These combined to form *CPU Performance Equation*

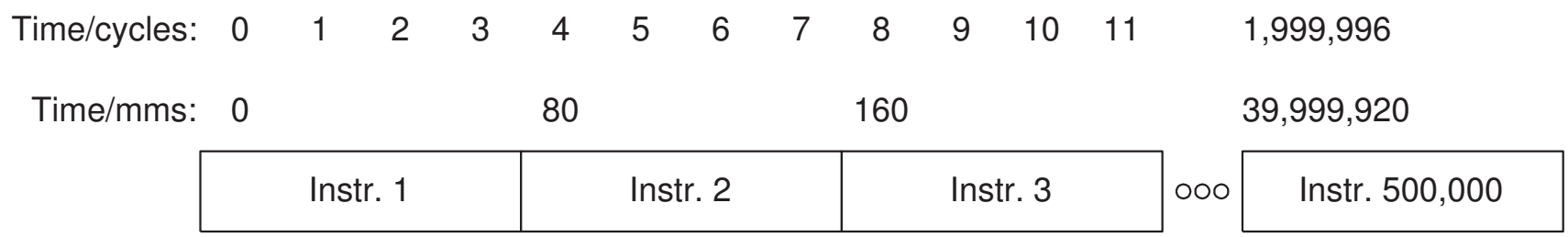
$$\boxed{t_{\mathrm{T}} = \frac{1}{\phi} \times \mathrm{CPI} \times \mathrm{IC}},$$

where t_{T} denotes the execution time.

02-3

CPU Performance: Simple System

Execution in program order . . .
. . . one at a time.



$\mathrm{IC} = 500,000$; $\phi = 50\,\mathrm{kHz}$; $\mathrm{CPI} = 4$.

Execution time: IC $\times$ CPI. $\times$ clock period.

Here (**and only here)** CPI is number of cycles for each instruction.

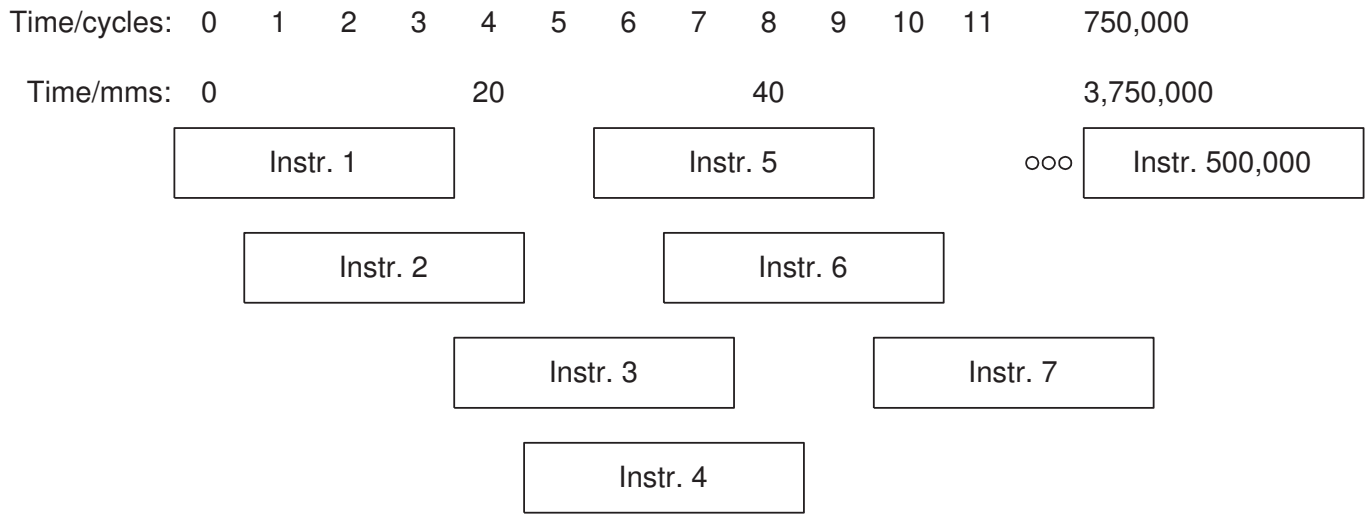
EE 4720 Lecture Transparency. Formatted 9:14, 6 March 2013 from lsli02.
02-3

Execution: Pipelined, In Order

02-4

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

Result must be same as one-at-a-time execution . . .
. . . not too difficult to achieve.



$\text{IC} = 500,000$; $\phi = 200\,\text{kHz}$; $\text{CPI} = \frac{750000}{500000} = 1.5$.

Execution time at best: IC $\times$ clock period . . .
. . . assuming 1 cycle to start each instruction and . . .
. . . instruction can start each cycle. (Slower in illustration.)

EE 4720 Lecture Transparency. Formatted 9:14, 6 March 2013 from lsli02.

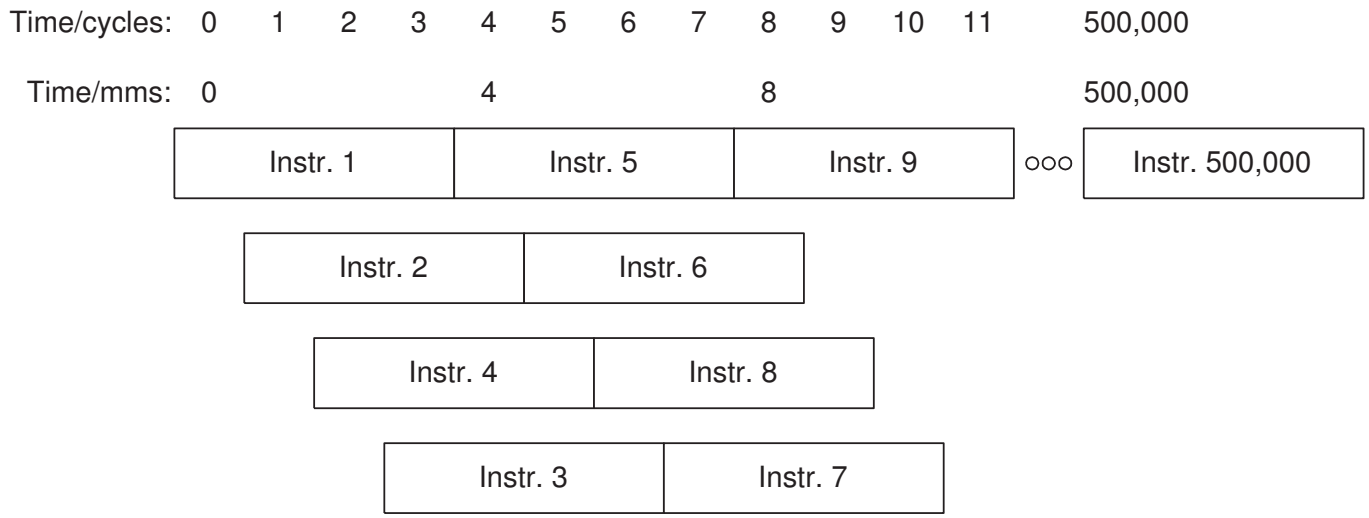
02-4

Execution: Pipelined, Ideal Out of Order

02-5

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

Sometimes skip an instruction and execute it later.



$\mathrm{IC} = 500,000$; $\phi = 200\,\mathrm{kHz}$; $\mathrm{CPI} = 1$.

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

EE 4720 Lecture Transparency. Formatted 9:14, 6 March 2013 from lsli02.

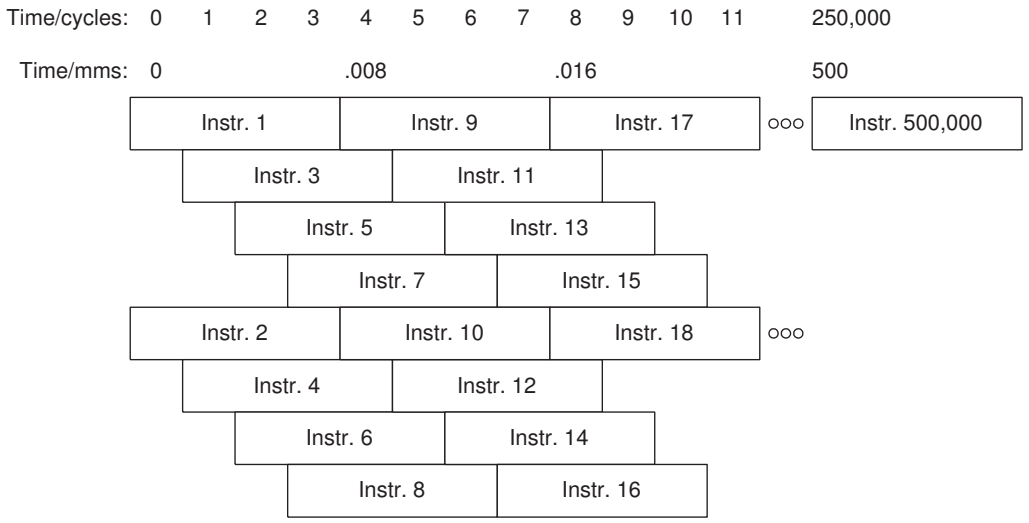
02-5

Execution: Pipelined, Ideal Out of Order, Superscalar

02-6

To Run Fastest[1]: Overlap, Out-of-Order, Start n per Tick (*n-Way Superscalar*).

Requires about n times as much hardware. (Below, $n = 2$.)



$\text{IC} = 500,000$; $\phi = 500\,\text{MHz}$; $\text{CPI} = \frac{1}{2}$.

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

[1] Using a conventional serial instruction set architecture.

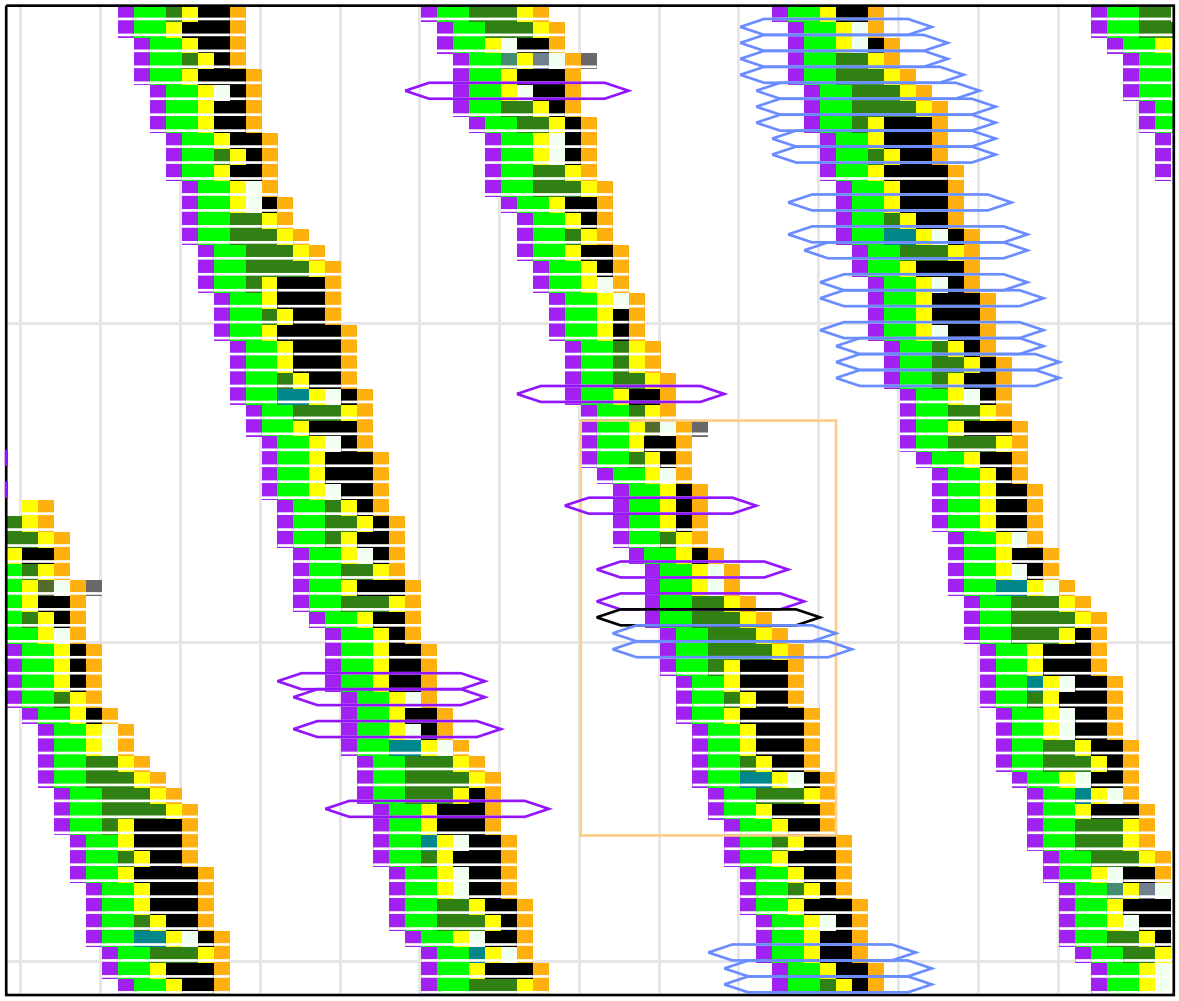
Execution: Pipelined, Out of Order, Superscalar

02-7

Data from a real program, perl. CPI is 0.44.

Processor can start four instructions per cycle.

Colors show the steps in processing an instruction, yellow is execution.



```
S_regmatch+104
0x001610d8    stw  %o4, [ %l0 + 16 ]
0x001610dc    cmp  %o1, 2
0x001610e0    bne  +3i  {S_regmatch+109}
0x001610e4    lduw  [ %l0 + 24 ], %l6
0x001610ec    mov  %o0, %i5
0x001610f0    mov  %i1 , %l3
0x001610f4    cmp  %i1, 0
0x001610f8    be  -100i  {S_regmatch+12}
0x001610fc    sethi  %hi(0x1f0800), %g2
0x00161100    lduh  [ %l3  + 2 ], %g2
0x00161104    ldub  [ %l3 + 1 ], %o0
0x00161108    sll  %g2 , 2, %g2
0x0016110c    add  %l3 , %g2 , %i1
0x00161110    cmp  %i1 , %l3
0x00161114    bne  +3i  {S_regmatch+122}
0x00161118    mov  %o0, %l0
0x00161120    cmp  %o0, 77
0x00161124    bgu  +3509i  {S_regmatch+3632
0x00161128    sethi  %hi(0x219c00), %g2
0x0016112c    sethi  %hi(0x160c00), %g1
0x00161130    sll  %o0, 2, %g3
0x00161134    add  %g1, 512, %g2  {0x160e00
0x00161138    lduw  [ %g3 + %g2 ], %g3
0x0016113c    jmp  %g3 + %g2
```


Time 42,084,435

Grid 20 insn X 5 cyc

02-7

Component of CPU Performance: Instruction Count

02-8

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.

Instruction Counts

02-9

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

Written in Simplescalar assembler.

```
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.

EE 4720 Lecture Transparency. Formatted 9:14, 6 March 2013 from lsli02.

02-9

Component of CPU Performance: Clock Frequency

02-10

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.

Component of CPU Performance: CPI

02-11

Cycles (clocks) per Instruction (CPI)

Oversimplified definition: *CPI:*

Average number of cycles needed to execute an instruction.

Better definition: *CPI:*

Number of cycles to execute some code divided by number of instructions. This is approximately the average number of cycles between instruction initiations (instruction starts).

Difference between simple and better definition:

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

. . . not time time for any one instruction.

Review of CPU Performance Equation

02-12

$$\boxed{t_{\mathrm{T}} = \frac{1}{\phi} \times \mathrm{CPI} \times \mathrm{IC}},$$

where t_{T} denotes the execution time.

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

Interaction of Execution Time Components

02-13

Tradeoffs between Clock Frequency, CPI, and Instruction Count

Increasing Clock Frequency ...

... reduces the work that can be done in a clock cycle ...
... and possibly limiting instruction overlap, therefore increasing CPI.

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.

IC v. CPI Tradeoffs

02-14

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.

IC v. CPI Tradeoffs, continued.

02-15

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.

02-16

Consider two implementations:

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

Implementation B: `add` CPI impact 1 cycle, `mul` CPI impact 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.

IC v. CPI Tradeoffs, continued.

02-17

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

Major tradeoffs in complexity and speed.

Consider two implementations:

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

Implementation B: CPI impact: `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.

Technological Change

02-18

Golden Handcuffs:

The need to maintain compatibility in a successful product line.

Famously, Intel's IA-32. (Popularly referred to as 80x86.)

The ISA is the handcuffs...

... and technological change brings the desire to move your arms.

02-19

Technological Change and Computer Designer

Technology determines "raw materials" for designer.

Raw material: number of gates and their speed.

ISA lifetime can be decades.

Raw materials greatly change over this time.

So, design ISA for now and future.

02-20

How technological advancement affects processor.

Logic Speed, Clock Rate

No changes to organization or ISA.

Number of Transistors Available for Logic

Changes to organization and possible changes to ISA.

Memory Size

Change ISA to use larger address space.

Can use ISA having larger instruction codings.

Memory Speed Compared to Processor Speed

Include more sophisticated caching in organization.

Benchmarks

02-21

Benchmark:
Program used to evaluate performance.

Uses

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

02-22

Using Benchmarks to Guide Computer Design

Measure overall performance.

Determine characteristics of programs.

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

Determine effect of design options.

Choosing Benchmark Programs

02-23

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.

Benchmark Types

02-24

Benchmark Classifications

Based on how benchmark is to be used.

Real Programs:

Programs chosen using surveys, for example.

Example: Photoshop (Image editing program.)

+ 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.

Example: Photoshop code for shrinking an image.

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

02-25

Microbenchmarks:

Code written to test a specific feature of a system.

Example: Measure maximum number of FP divisions per second.

\+ Useful for tuning specific features during implementation development.

\- One might get too fixated on narrow feature.

Toy Benchmarks:

Programs written casually, without insuring that they measure something useful.

Example: The pi program used in class.

\+ Easier to write.

– Not realistic.

Commonly Used Benchmark Categories

Overall performance: real programs

Test specific features: microbenchmarks.

Benchmark Suites

02-26

Benchmark Suite:
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.

02-27

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

02-28

Example, SPEC CPU2006 Suites

Measures CPU and memory performance on *integer* and *FP* programs.

Respected measure of CPU performance.

Managed by Standard Performance Evaluation Corporation,. . .
. . .a non-profit organization funded by computer companies and other interested parties.

Uses common programs such as perl, gcc, gzip.

Requires that results on each program be reported.

Programs compiled with publicly available compilers and libraries.

Programs compiled with and without expert tuning.

02-29

SPEC CPU2006 Suites and Measures

Suite of integer programs run to determine:

- SPECint2006, execution time of tuned code.
- SPECint_base2006, execution time of untuned code.
- SPECint_rate2006, throughput of tuned code.
- SPECint_rate_base2006, throughput of untuned code.

Suite of floating programs run to determine:

- SPECfp2006, execution time of tuned code.
- SPECfp_base2006, execution time of untuned code.
- SPECfp_rate2006, throughput of tuned code.
- SPECfp_rate_rate2006, throughput of untuned code.

SPECcpu Benchmarks

02-30

SPEC CPU Suite Goals

Measure CPU and memory system.

Avoid benchmarks making lots of disk I/O, etc.

Measure potential of newest implementations and ISAs.

Tester compiles benchmark using own tools.

Trustworthiness of Suite.

Suite developed by competitors, and other interested parties.

Trustworthiness of Results.

Easy for anyone to duplicate test results, so erroneous results quickly exposed.

02-31

SPEC Testing Procedure

Defined by *Run & Reporting Rules*.

Carried out by tester (not SPEC).

Given:

A computer to test.

A copy the SPEC benchmarks.

Software to compile.

02-32

Test Procedure

Get:

System Under Test (SUT):
The computer on which benchmarks are to be run.

A copy of the SPECcpu benchmark suite.

Prepare a config file:

Name of system, build tools, etc.

Location of compiler.

Portability switches.

Optimization switches.

02-33

Run script:

Script will..

Compile benchmarks, profile, compile again.

Run benchmarks three times, verify outputs.

Generate reports.

Evaluate results:

If not satisfied

Try different optimization switches.

Substitute different compilers, libraries, etc.

Convince customers that for them SPECcpu results are irrelevant.