RUME DILLIANCE OCHIENG

SCT212-0076/2020

BCT2408

Computer Architecture - Lab 1

# E1: H&P (2/e) 1.6 p.61

**Problem**

After graduating, you are asked to become the lead computer designer at Hyper Com- puters, Inc. Your study of usage of high-level language constructs suggests that procedure calls are one of the most expensive operations. You have invented a scheme that reduces the loads and stores normally associated with procedure calls and returns. The first thing you do is run some experiments with and without this optimization. Your experiments use the same state-of-the-art optimizing compiler that will be used with either version of the computer. These experiments reveal the following information:

* The clock rate of the unoptimized version is 5% higher.
* 30% of the instructions in the unoptimized version are loads or stores.
* The optimized version executes 2/3 as many loads and stores as the unoptimized version. For all other instructions the dynamic counts are unchanged.
* All instructions (including load and store) take one clock cycle. Which is faster? Justify your decision quantitatively.

*Solution*

CPU performance equation: *CPUTime* = *IC* ∗ *CPI* ∗ *ClockTime*

We know

*ClockTimeunop* = 0*.*95 ∗ *ClockTimeop*

*ICld/st,unop* = 0*.*3 ∗ *ICunop*

*ICld/st,op* = 0*.*67 ∗ *ICld/st,unop ICothers,unop* = *ICothers,op*

*CPI* = 1

Thus:

*CPUTimeunop* = *ICunop* ∗ *ClockTimeunop* = 0*.*95*ICunop* ∗ *ClockTimeop* (1)

*CPUTimeop* = *ICop* ∗ *ClockTimeop*

but,

*ICop* = 0*.*7 ∗ *ICunop* + 0*.*3 ∗ 0*.*67 ∗ *ICunop* = 0*.*9 ∗ *ICunop*

then,

*CPUTimeop* = 0*.*9 ∗ *ICunop* ∗ *ClockTimeop* (2) Comparing 1 and 2 we see that the optimized version is faster.

*Speedup* = 0*.*95 ∗ *ICunop* ∗ *ClockTimeop*

0*.*9 ∗ *ICunop* ∗ *ClockTimeop*

= 1*.*06

# E2: H&P (2/e) 2.6 p.164

*Problem*

Several researchers have suggested that adding a register-memory addressing mode to a load-store machine might be useful. The idea is to replace sequences of:

LOAD Rx,0(Rb)

ADD Ry,Ry,Rx

by

ADD Ry,0(Rb)

Assume this new instruction will cause the clock period of the CPU to increase by 5%. Use the instruction frequencies for the gcc benchmark on the load-store machine from Table 1. The new instruction affects only the clock cycle and not the CPI.

1. What percentage of the loads must be eliminated for the machine with the new in- struction to have at least the same performance?
2. Show a situation in a multiple instruction sequence where a load of a register (Rx) followed immediately by a use of the same register (Rx) in an ADD instruction, could not be replaced by a single ADD instruction of the form proposed.

|  |  |
| --- | --- |
| **Instruction** | **Frequency** |
| load | 22.8% |
| store | 14.3% |
| add | 14.6% |
| sub | 0.5% |
| mul | 0.1% |
| div | 0% |
| compare | 12.4% |
| load imm | 6.8% |
| cond. branch | 11.5% |
| uncond. branch | 1.3% |
| call | 1.1% |
| return, jump ind. | 1.5% |
| shift | 6.2% |
| and | 1.6% |
| or | 4.2% |
| other (xor, not) | 0.5% |

Table 1: Instruction frequencies for gcc (cc1) (from H&P (2/e), Figure 2.26, p.105)

*Solution*

CPU performance equation: *CPUTime* = *IC* ∗ *CPI* ∗ *ClockTime*

We know:

*ClockTimeop* = 1*.*05 ∗ *ClockTimeunop*

*CPIop* = *CPIunop*

## a.

but,

*CPUTimeop* = *CPUTimeunop*

⇒ *ICop* ∗ 1*.*05 ∗ *ClockTimeunop* = *ICunop* ∗ *ClockTimeunop*

⇒ *ICop* = 0*.*95 ∗ *ICunop* (3)

*ICop* = *ICunop* − *R* (4)

where *R* is the number of instructions removed, then combining 3 and 4,

*ICunop* − *R* = 0*.*95 ∗ *ICunop*

⇒ *R* = 0*.*05 ∗ *ICunop* (5)

but (from figure 2.32, pg. 138),

*ICld,unop* = 0*.*228 ∗ *ICunop* (6)

combining 5 and 6 ,

*R* = 0*.*05 ∗ 4*.*39 ∗ *ICld,unop*

⇒ *R* = 0*.*22 ∗ *ICld,unop*

## b.

Consider the following code:

LOAD R1, 0(R5)

ADD R1, R1, R1

and that r1 has the initial value 47, r5 has a value of 1000 and that the data value in memory [1000] is 4. Then replacing the code with:

ADD R1, 0(R5)

will give the incorrect result of 51, instead of the correct result 8. In other words, in situations in which Rx and Ry refer to the same register, the replacement will not be semantically equivalent.

# D1: Discussion

***Are Modern “RISC” Processors Still RISC?***

The core tenets of Reduced Instruction Set Computing or RISC, set in the 1980s, focused on efficiency and minimalism in the execution of instructions. This minimalism was marked by:

* Fixed-Length Instructions: All instructions are the same size, simplifying instruction decoding.
* LOAD/STORE Architecture: The memory is accessible only through load and store operations, while other operations are restricted to registers.
* Few Addressing Modes: There are only a few simple addressing modes.
* Restricted Instruction Set: A minimal, but extremely efficient set of commands.
* Single-Cycle Implementation: All instructions are executed in one clock cycle.

Today's RISC processors like ARM, RISC-V, and PowerPC have also been greatly improved. They have many more instructions, like traditional CISC processors, but are nonetheless RISC for many essential reasons:

***Why Modern RISC is Still RISC***

Load/Store Architecture

Memory is still accessible only through specific operations (load and store). Arithmetic and logic operations are all performed within registers, holding to the RISC philosophy.

Fixed and Simple Instruction Encoding

RISC processors maintain a fixed instruction length even with the rise in the number of instructions, keeping the decoding process simpler.

Register-Based Instructions:

Most are executed directly on registers instead of memory addresses, lowering the level of instruction complexity.

Parallelism is bedevilled

RISC architectures remain pipeline-optimized, with a focus on instruction throughput over the complexity of a single instruction.

Simplicity in Instruction Implementation:

Although modern RISC ISAs contain more instructions, every instruction is considered to be relatively simple and takes a few cycles to execute. Simple operations are not broken down.

***Why Modern RISC is Less “Simple” Than 80s RISC:***

Expanded Instruction Set

New instructions have been introduced to manage multimedia processing, cryptographic operations, and other specialized operations. These instructions add to the number of instructions but are aimed to run efficiently with few cycles.

Variable Instruction Lengths:

Certain recent RISC processors (such as ARM Thumb and RISC-V compressed instructions) embrace variable-length coding to enhance code density, deviating to some extent from the rigid fixed-length philosophy.

Microcoded Instructions:

Some modern RISC processors use microcode for certain complex instructions, a CISC-like feature that reduces simplicity but enhances functionality.

***Defining Simplicity in Modern RISC ISAs***

With respect to contemporary RISC architectures, the following are the features that best characterize “simplicity”:

Load/Store Architecture

Separating memory access from computations strictly is perhaps the most characteristic aspect of this simplicity.

Consistent Instruction Encoding:

With more instructions, sticking to a consistent or predictable coding scheme keeps the decoding logic for the instructions simple.

Limited Addressing Modes:

The decreasing the number of addressing modes keeps the instruction set simple and consistent.

Register-Based Design:

Minimizing direct memory operations and prioritizing operations in registers minimizes the number of cycles per instruction.

Simple data types

Fewer, more general data types minimize the amount of specialized instructions.

# D2: Discussion

***Are Modern Intel Processors RISC or CISC***

Newer generation processors like Core and Xeon have made modern processors like them a unique case in the RISC vs. CISC discussion. From the software interface point of view, they stick to the classic x86-style CISC architecture, with long, variable-length operations, many addressing modes, but also multi-cycle operations. Internally, they utilize a RISC-like microarchitecture.

***Levels of ISA Complexity:***

In order to properly analyze whether the processors of Intel are RISC or CISC, we should differentiate between two levels:

Architectural (Software Interface) Level

This is the level accessible to compilers and to assembly language coders.

Intel processors expose the classic x86 CISC instruction set to software.

The ISA at this level has complex, variable-length instructions, multiple addressing modes, and specialized instructions (e.g., SSE, AVX).

Microarchitecture (Internal Implementation) Level:

It is at this level that instructions are executed internally.

Today's Intel processors convert x86 CISC instructions to micro-operations (µops), which are RISC-like in character.

They are simple, fixed-length, and register-centric, much like traditional RISC concepts.

All these are RISC-inspired methods, including pipeline, register renaming, out-of-order execution, and branch prediction.

Which Level Specifies ISA Complexity?

If the ISA is to be defined in terms of the software interface, then Intel processors are distinctly CISC. The x86 instruction set is still complex, having variable length, multi-step addressing modes, and a rich number of instructions.

When defining the ISA based on the internal microarchitecture, the processors of Intel look like RISC, since the CISC instructions are decomposed internally into simpler, fixed-size µops that look like RISC instructions.

Hence, the response varies according to perspective:

From the programmer's point of view: Intel processors remain CISC, in that they present a high-level, complex instruction set to software.

microarchitectural level-execution engine: These processors are RISC-like in that they break down high-level instructions to simpler, fixed-sized operations.

Implications of considering ISA at every level

Software Level Interface (CISC View):

Compiler Design: Needs more advanced instruction decoding logic.

Support for backward compatibility for older versions of software using legacy CISC instructions.

Complex instructions can require several cycles to decode, which may add to the latency.

Microarchitecture Level (RISC View):

Execution Efficiency: Reduced µop length to enable deeper pipelining, greater parallelism, and improved exception prediction.

Power Optimization: Simple µops can be performed in less than a single cycle, thus lowering power requirements.

Scalability: Fixed-size µops are simpler to implement for out-of-order execution and superscalar techniques.

***Are Modern Intel Processors RISC?***

No, they are neither purely RISC, but neither are they purely CISC. More fittingly, they are best described as being externally CISC with an internally implemen- tation that is RISC-like.

* On the software level, they are still CISC, following the x86 instruction set with the associated added complexity.
* On the internal execution level, they embrace RISC concepts through decoding of the high-level instructions into simple, fixed-size µops that are similar to RISC instructions.