**ELVIS MBURU**

**SCT212-0062/2020**

**BCT 2408: COMPUTER ARCHITECTURE**

***E1: H&P (2/e) 1.6 p.61 – individual or groups of 2, 15 mins.***

*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.

### **Quantitative Comparison**

We are analyzing two versions of the CPU:

* **Unoptimized Version:**
  + 30% of instructions are loads/stores.
  + Clock rate is **5% higher** than the optimized version.
  + CPI = 1 for all instructions.
* **Optimized Version:**
  + The number of loads/stores is reduced by **1/3**.
  + Other instructions are unchanged.
  + Clock rate is **5% lower** (slower clock cycle).

Let’s define:

* III = total instruction count of the unoptimized version.
* Loads/stores = 0.3I
* Other instructions = 0.71

In the optimized version:

* Loads/stores = (2/3)×0.3I=0.2I
* Other instructions = 0.7I

**Total optimized instruction count:**

0.2I+0.7I=0.9I

### **Execution Time Calculation**

**CPU Performance Equation:**

Execution Time=(Instruction Count)×(CPI)×(Clock Cycle Time)

Let’s assume:

* Clock cycle time of the unoptimized version = C
* Clock cycle time of the optimized version = 1.05C (since the clock is 5% slower).

#### **Unoptimized:**

* Instruction count = I.
* CPI = 1.
* Clock cycle time = C.

Execution Timeunopt=I×1×C=IC

**Optimized:**

* Instruction count = 0.9I.
* CPI = 1.
* Clock cycle time = 1.05C.

Execution Timeopt=0.9I×1×1.05C

### **Which Version Is Faster?**

We compute the **speedup:**

Speedup=Execution Timeunopt  / Execution Timeopt=IC / 0.945IC≈1.058

**The optimized version is ~5.8% faster.**

## **Discussion**

### **Trade-offs Between Instruction Count and Clock Rate**

This analysis shows that **reducing the instruction count** can yield a meaningful performance boost **even if** it results in a **slower clock.** In this case, although the optimized version had a **5% slower clock,** the 10% reduction in instruction count (from III to 0.9I0.9I0.9I) compensated for it and produced a **net performance gain of ~5.8%.**

This highlights a **fundamental trade-off**:

* **Higher Clock Rate:** Can improve performance but may require simpler instructions and pipelines.
* **Lower Instruction Count:** Reduces the total work the CPU does, but may come with a cost in clock speed or pipeline complexity.

### **Implications for Modern ISA Design (RISC vs. CISC)**

Modern CPU designs (e.g., ARM, x86) grapple with exactly these trade-offs:

* **RISC:** Traditionally focuses on simple instructions, keeping CPI low and clock rates high. However, more instructions are often needed to perform a task.
* **CISC:** Uses complex instructions that can reduce instruction count (like our optimization here), but these may complicate the hardware and potentially limit clock speed.

Today’s CPUs often **blur the lines:**

* **x86 (CISC):** Complex instructions are translated internally into **micro-ops** that behave like RISC instructions.
* **ARM (RISC):** Has added many complex instructions and optimizations over time.

This analysis shows why **modern ISAs must balance:**

* Reducing **instruction count** where possible (e.g., SIMD/vector extensions, fused instructions).
* Maintaining high **clock rates** by keeping pipelines and instruction execution simple internally.

In essence, performance gains can come from either side of the equation, and **modern CPU designs aim to optimize both simultaneously.**

**E2: H&P (2/e) 2.6 p.164 – individual or groups of 2, 10 mins.**

*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 instruction to have at least the same performance?

CPU Performance Equation:

Execution Time=(Instruction Count)×(CPI)×(Clock Cycle Time)

Goal:

For the new machine to match performance:

(New Instruction Count)×(1)×(1.05C)=(Original Instruction Count)×(1)×(C)

Let:

* I = total instruction count (before).
* R= reduction in instruction count (due to eliminating some loads).

We have:

1.05(I−R)=I

Solving:

1.05I−1.05R=I

- 1.05R=−0.05I-1.05R

R = (0.05/1.05)I ≈ 0.0476I≈4.8% of total instructions.

Apply to Loads:

From the gcc benchmark:

* Loads = 22.8% of total instructions.

We need a 4.8% total instruction count reduction. To find what % of the loads this represents:

X × 22.8% = 4.8

X = 4.8% / 22.8%≈21%

2. Show a situation in a multiple instruction sequence where a load of a register (say 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.  
  
**Example:**

LOAD R1, 0(R2)

ADD R3, R3, R1

MUL R4, R1, R5

### **Why Can’t We Replace It?**

In this sequence:

1️ The **LOAD** puts the value from memory at address 0(R2) into R1.

2️ The **ADD** uses R1 to update R3.

3️ **Importantly, the next instruction (MUL)** **also uses R1.**

If we tried to **replace the LOAD + ADD** with a **single:**

ADD R3, 0(R2)

* This would remove the **explicit LOAD**.
* But now, **R1 no longer holds the loaded value**, which the **MUL** needs!

The original code ensures R1 contains the memory value for **both the ADD and the MUL**, but the replacement only updates R3 and does not keep the loaded value in R1.

### **Key Reason:**

The loaded value is used by more than one instruction, not just by the ADD.

In such cases, replacing with a single memory-based ADD would break the program’s logic because the loaded value isn't available for later instructions.

**D1: Discussion – in groups of 4, 15 mins.**

In the early years of the RISC versus CISC dispute, the total number of different instructions and their variations in the ISA was a common indication of the “simplicity” of an ISA (lesser the number, greater the simplicity). Modern RISC instruction sets contain almost as many

instructions as old CISC instruction sets. Discuss whether modern “RISC” processors are no longer RISC (as envisioned in the 80’s). If they are still RISC, then what features in the instruction set best define the simplicity of an ISA? (e.g. memory access instructions, fixed and simple instruction encoding, register-oriented instructions, simple data types, etc?  
  
  
**Are Modern “RISC” Processors Still RISC?**

In the 1980s, RISC (Reduced Instruction Set Computer) was defined by:

* Small number of simple instructions.
* Load/store architecture (memory access only via specific instructions).
* Fixed-length, simple instruction encoding.
* Few simple addressing modes.
* Heavy reliance on registers (register-register operations).

### What’s Changed?

Today, modern RISC ISAs (like ARMv8, RISC-V):

* Have many more instructions (sometimes hundreds, with SIMD, crypto, and other extensions).
* Support complex operations (e.g., fused multiply-add, load/store with update).
* Include variable-length instruction sets (e.g., ARM’s Thumb).

This blurs the line between the original RISC and CISC vision, especially since the total instruction count now rivals old CISC designs.

### What Has Stayed RISC?

Despite these changes, key RISC principles still hold:

Memory access:

* Still a load/store architecture. Even complex instructions don’t allow full register-memory-register operations like x86 does.

Encoding:

* Often fixed-length and simple encodings (e.g., 32-bit fixed instructions in RISC-V), making it easier to pipeline and decode.

Register-oriented:

* Most operations are still register-to-register, not direct memory-to-memory.

Simple data types:

* Generally focus on basic data types (integers, floats) with minimal implicit state changes.

### Conclusion:

Even though modern RISC architectures have grown in complexity:

* They retain core RISC characteristics in terms of architecture design philosophy.
* The increase in instructions reflects evolving needs (like multimedia, security) rather than abandoning RISC principles.

So, while they don’t look as “minimalist” as in the 1980s, modern RISC CPUs are still RISC by virtue of:

* Simple, efficient execution model.
* Load/store separation.
* Predictable instruction formats.

### Key Features Defining RISC Simplicity Today:

* Load/store architecture.
* Fixed/simple instruction encoding.
* Register-based operations.
* Minimal side effects.
* Simple addressing modes.

**D2: Discussion – in groups of 4, 10 mins.**

Even though the Intel x86 ISA is a clear example of a CISC ISA, modern implementations of it (e.g. Core and Xeon) use many RISC ideas: register-based micro-instructions, pipelining, simple branch micro-instructions, fixed length micro-instructions, etc. Some say that, since at the low level the the latest Intel processors behave like a RISC, it is RISC. Others say that, since at the software interface (compiler) they are seen like a CISC, they are CISC. Discuss at what level we should measure the complexity of ISA? What are the implications

of considering the ISA at each level? Are the latest Intel processors RISC?  
  
  
**Where Should We Measure ISA Complexity?**

The **ISA (Instruction Set Architecture)** defines the interface between **software (compilers, programs)** and **hardware.** The debate boils down to:

* **Software-visible ISA:**
  + What the **compiler/programmer sees** (e.g., x86 with complex instructions, many addressing modes).
* **Microarchitecture (internal):**
  + How the **CPU actually executes** the instructions (e.g., breaking x86 into simpler micro-ops, using RISC-like execution).

### **Intel Example:**

* **x86 ISA (Software Level):**
  + Complex instructions (e.g., string operations, memory-to-memory arithmetic).
  + Variable-length encoding (from 1 to 15 bytes!).
  + Rich set of addressing modes.
* **Modern x86 Microarchitecture:**
  + Translates x86 instructions into **RISC-like micro-ops** internally.
  + Uses **pipelining, fixed-length micro-ops, out-of-order execution, etc.**

## **Key Question:**

**Is x86 RISC just because its internals behave like RISC?**

### **Most Agreed View:**

**We should measure ISA complexity at the software interface level.**

Here’s why:

1️ **ISA = Contract between Software & Hardware:**

* The ISA defines how software is written and compiled.
* Even if the CPU is RISC-like inside, the compiler still has to deal with the **CISC interface** (e.g., x86).

2️ **Portability & Compatibility:**

* What matters for binary compatibility is the **ISA software sees**, not how it’s implemented internally.

3️ **Microarchitecture is Flexible:**

* Any ISA can be implemented in many ways.
* A CISC ISA can use RISC-style micro-ops, and a RISC ISA could, in theory, have complex execution units.

### **Implications:**

* **Classifying by Software-Level ISA:**
  + x86 remains **CISC** because compilers/programmers still target a **complex, rich instruction set.**
* **Classifying by Microarchitecture:**
  + Modern x86 CPUs **behave like RISC inside,** showing that **RISC techniques are valuable** even when the ISA is CISC.

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

**At the microarchitecture level:**

* Yes, they use many **RISC-like principles** internally.

**At the ISA/software level:**

* No, they are still **CISC,** because that's what developers/compilers interact with.