LAB 1 Assignment

Question 1
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 Computers 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.

Clock rate of optimised = CR (optimised)

Clock rate of unoptimized = 1.05 x CR (optimised)

Clock Cycle Time = 1/Clock Rate

Clock Cycle Time of optimised = 1/CR (optimised)

Clock Cycle Time of unoptimized = 1/1.05 x CR (optimised)

Loads and stores of unoptimized = 0.3 x IC (unoptimized)

Loads and stores of optimized = 2/3 of 0.3 x IC (unoptimized) = 0.2 of IC (unoptimized)

Other instructions of unoptimized = 1 - (0.3 x IC (unoptimized)) = 0.7 x IC (unoptimized)

Other instructions of optimized = 0.7 x IC (unoptimized)

Since All instructions (including load and store) take one clock cycle. CPI = 1

Using **CPU Performance Formula = IC x CCT x CPI**

IC = Loads and stores instructions + Other Instructions

CPU Time of unoptimized = (0.3 x IC (unoptimized) + 0.7 x IC (unoptimized)) x 1/1.05 x CR (optimised) = 0.9524

CPU Time of optimised = (0.2 of IC (unoptimized) + 0.7 x IC (unoptimized)) x 1/CR (optimised) = 0.9

Ratio = CPU Time of unoptimized/ CPU Time of optimised = 0.9524/0.9 = 1.058

Since the ratio is greater than 1, the CPU Time of unoptimized is larger making it slower than the **optimised version which is faster**.

Question 2

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 Time = **= IC x CCT x CPI and CPI = 1**

CCT (new) = 1.05 x CCT (old)

CPU Time(old) = IC (old) x CCT (old)

Fraction of load to remove = (X x 0.228 x IC (old))

IC (new) = IC (old) - (X x 0.228 x IC (old))

CPU Time(new) = IC (old) - (X x 0.228 x IC (old)) x (1.05 x CCT (old))

Equate the 2 CPU times and solve for x. Since IC (old) and x CCT (old) appear on both sides of the equation, so they cancel out.

1 = (1 - 0.228x) * 1.0

x = 0.0476 / 0.228

x = 0.209 or 20.9%

At least **20.9% of the loads must be eliminated** for the machine with the new instruction to have at least the same performance.

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.

LOAD R1, 0(R2);

STORE R1, 0(R3);

ADD R4, R4, R1;

We cannot replace the LOAD R1, 0(R2) and ADD R4, R4, R1 with ADD R4, 0(R2 because the STORE R1, 0(R3) instruction needs the original value loaded into R1 before it's overwritten by a combined load-and-add.

The STORE instruction creates a data dependency that prevents the optimization. The value loaded into R1 is used for two different purposes: storing to memory and adding to R4.

D1: Discussion

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?

Modern RISC (Reduced Instruction Set Computing) processors have evolved significantly since their early days in the 1980s. While they were originally designed with a small and simple instruction set, modern RISC architectures, such as ARM and RISC-V, now include a large number of instructions, comparable to older CISC (Complex Instruction Set Computing) architectures. Despite this increase, RISC processors still follow key principles such as a load/store architecture, fixed instruction length, and register-based operations, ensuring efficient execution and pipelining.

The increase in instructions in modern RISC ISAs is mainly due to specialized extensions for tasks like floating-point arithmetic, cryptography, and SIMD operations. However, the core instruction set remains simple and adheres to RISC principles, unlike CISC architectures that feature complex addressing modes and variable-length instructions. Additionally, RISC designs prioritize uniform instruction execution, where most operations complete in a single clock cycle, allowing for better performance and easier pipelining.

Therefore, modern RISC processors are still fundamentally RISC, despite their expanded instruction sets. The defining features of simplicity in a RISC ISA include load/store memory access, fixed instruction encoding, register-oriented operations, and efficient pipelining. These characteristics distinguish them from CISC designs, ensuring that modern RISC architectures remain efficient, scalable, and optimized for performance.

D2: Discussion

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 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?

The complexity of an ISA (Instruction Set Architecture) can be measured at different levels, each with its own implications. At the software interface level, which includes the view of compilers and programmers, Intel's x86 ISA remains a CISC. It supports variable-length instructions, complex addressing modes, and many specialized instructions, requiring more complex decoding and execution logic. From this perspective, x86 maintains its CISC nature because compilers must generate and optimize code according to these high-level, complex instructions.

However, at the hardware implementation level, modern Intel processors behave like RISC internally. They use micro-op translation, where CISC instructions are broken down into simpler, fixed-length micro-operations that resemble RISC instructions. These micro-operations are then executed using techniques common in RISC processors, such as pipelining, register renaming, and superscalar execution. While this RISC-like backend improves efficiency, it does not change the fact that x86 remains a CISC ISA at the software level.

The choice of level at which we measure complexity has key implications. If we focus on the software interface, x86 is clearly CISC, meaning compilers and programmers must handle its complexity. If we focus on hardware execution, it behaves like RISC, meaning that its efficiency and performance benefit from RISC principles. However, Intel processors still require a complex front-end decoder to translate CISC instructions into RISC-like operations, introducing overhead. Therefore, while modern Intel processors use RISC techniques internally, their ISA remains CISC, as it defines how software interacts with the hardware.