NAME: VIVIAN KERUBO MOSOMI

REGISTRATION NUMER: SCT212-0062/2021

COMPUTER ARCHITECTURE – LAB 1

**Question 1**

Context, Objectives and Organization Worksheet 1- week 3 This worksheet covers material from Lectures 1 (introduction to CA and technology trends), 2 (CPU performance equations) and 3 (the first half of the section on Instruction Set Architecture). The main goals of this tutorial is to give you some quantitative experience with the CPU performance equation, and to stimulate a discussion where your group can explore some issues related to modern instruction set design. This tutorial consists of two activities. The first involves quantitative problem-solving using exercises taken from the 2nd and 3rd editions of the H&P book. These can be done individually or in groups of two. The second main activity involves qualitative discussions in small groups followed by an exchange of ideas with the rest of the group. 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.

**Unoptimized Version**

Clock Rate - 5% higher

Loads/Stores – 30% which is 0.3

Clock cycle – 1

**Optimized Version**

Clock Rate – 5% lower than optimized

Reduces 2/3 as many loads/stores of the unoptimized version.

Clock cycle – 1

Loads/Stores = (2/3) \* 0.30(loads/stores of the unoptimized version) which is 0.20

**Answer**

CPU Time = (Instruction Count \* Clock per Instruction) / Clock Rate

Other instructions, for both optimized and unoptimized version = (1 – 0.30) = 0.70

Instruction count for the optimized version = Loads/Stores + Other Instructions

Calculated as: 0.20 + 0.70 = 0.90

**Calculating CPU time:**

i)For unoptimized version **=** Instruction Count / Clock Rate

ii) For optimized version:

(Instruction count = 0.90) / (0.95) =

**Compare Performance**

The ratio:

CPU Time (Unoptimized) / CPU Time (Optimized) = (Unoptimized Instruction Count/ Unoptimized Clock Rate) \*(0.95 Unoptimized Rate ) / (0.90 Unoptimized Instruction Count) which gives

0.95 / 0.90 = 1.056

(1.056 – 1) \* 100 = 5.6

The optimized version is 5.6% faster than the unoptimized version

**Question 2**

**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 ADD by ADD Rx,0(Rb) Ry,Ry,Rx 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.**

**i). What percentage of the loads must be eliminated for the machine with the new instruction to have** **at least the same performance**

Execution Time = Total Instructions / Clock Rate where:

Unoptimized clock rate = 1.05

Optimized clock rate = 1

Let:

Iu – Total number of instructions in the unoptimized version

Io – Total number of instructions in the optimized version

30% of instructions are loads in the unoptimized version and the optimized version reduces 2/3 of loads. Thus load of optimized = 2/3 loads of unoptimized (Lu)

Total instruction counts after optimization:

Io​=(Iu​−Lu​) +Lo​

Io=(Iu−0.30Iu) + 2/3(0.30Iu)

Io=(0.70Iu) +(0.20Iu) ​= 0.90Iu

Iu/1.05​​ = Io​​/1.00

Replacing Io with 0.90Iu thus:

Iu/1.05 = 0.90Iu/1.00

Cancelling out Iu from both sides and remaining with:

1/1.05 = 0.9524

0.9524 = 0.90

0.9524 – 0.90 = (0.0524 \* 100) = 5.24%

5.24% of loads must be eliminated for the machine with new instruction to have the same performance

**ii) 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:

# Loading a value from memory

LOAD R3, 0(R1)

# Using the value in an addition

ADD R2, R2, R3

# Using the same value in a subtraction

SUB R4, R4, R3

The above ADD cannot be replaced because the loaded value,R3, is used in multiple places and not just the ADD operation

**Discussion 1**

**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 80s). If they are still RISC, then what features in the instruction set best defines the simplicity of an ISA? (e.g. memory access instructions, fixed and simple instruction encoding, register-oriented instructions, simple data types, etc?).**

**Discussion on whether modern RISC processors are no longer RISC (As envisioned in the 80s)**

**i) RISC in the 80s**

When computer scientists realized that most programs used only a small subset of the available instructions in complex instruction set computer architecture (CISC). Researchers search as John Cocke at IBM, proposed to simplify the instruction set and use more registers to store data thus reducing the need to memory access and increasing the speed of execution.

Features of RISC at that time:

1.Simple, fixed length instructions

Instructions had a fixed size, often 32 bits for easier decoding unlike CISC which had variable length instructions. Simple instructions implied that the CPU could fetch and execute them faster.

2.Load-Store architecture

Memory operations were separate from arithmetic and logical operations which meant that:

* Data had to be loaded into registers before computation.
* Results had to be stored into memory after computation.

3.Register-Oriented Design

Early RISC designs had more registers than CISC processors. This was so as to keep frequently used data in registers instead of slow memory. Instead of performing operations directly on memory, RISC used registers.

4.Single-Cycle Execution for Most Instructions

RISC instructions were designed to execute in one clock cycle, unlike CISC where some instructions took multiple cycles.

**ii) Emergence of RISC**

In the 2010s, a new open-source RISC instruction set design emerged, called RISC-V. RISC-V was developed by researchers at UC Berkeley, with the aim of creating a simple, modular, and scalable instruction set that could be used for a wide range of devices and purposes. RISC-V is based on the core principles of RISC, such as:

1. Simple Instruction Set

RISC-V follows the same fixed-size instruction format. It avoids complex multi-cycle instructions, just like the early RISC processors.

2. Load-Store Architecture

Memory access is separate from computations, sticking to the original RISC load-store model.

Instructions operate only on registers, reducing memory bottlenecks.

3. Large number of registers

RISC-V supports registers, like early RISC processors. Thus, enhancing performance in modern applications by reducing memory access and improving speed.

4.Pipelining and Parallelism

Instructions are designed to be executed in a single cycle whenever possible, efficient for pipelining hence improving overall performance.

From the comparison above, modern RISC is still RISC envisioned in the 80s.

**Features in the instruction set that best define simplicity of an ISA**

1.Fixed-Length, Simple Instruction Encoding

Instructions have a fixed size e.g 32 bit, making decoding predictable and fast

2. Load-Store Architecture

Memory access is only done through Load and Store Operations. All arithmetic and logical operations happen in registers. This keeps memory access and computation separate.

3. Register-Based Operations

Most RISC processors have a large number of registers. Instead of performing operations directly on memory, data is first loaded into registers. Example: Instead of adding memory locations, a simple addition operation can entail:

LOAD R1, memory\_location\_1

LOAD R2, memory\_location\_2

ADD R3I, R1, R2

4. Single-Cycle Execution

RISC instructions are designed to execute in a single clock cycle, thus optimizing CPU designs for high performance.

5. Fewer and Simpler Data Types

RISC typically supports only a few basic data types such as integers and floating-point numbers, simplifying CPU design and making data processing more efficient.

**Discussion 2**

**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?**

**i) At what level should we measure the complexity of ISA?**

The complexity of an Instruction Set Architecture can be viewed in two levels:

1. Software Interface (Compiler Perspective) –

This level is what the programmer sees. The ISA is the interface between the software (compilers, applications) and the hardware (processor). It defines the instructions the processor understands and how the software interacts with the hardware.

2. Hardware Implementation

This level is how the CPU actually executes information. The processor's internal implementation (microarchitecture) is how the instructions are actually executed. This can involve techniques like pipelining and micro-operations.

**ii) What are the implications of considering the ISA at each level?**

a) At the software level, the view of programmers.

For this level, the focus is on how instructions are written, compiled and understood by software.

* Software Compatibility & Portability

A complex ISA can support a wide range of instructions, making it easier to run different types of software without modification. Simpler ISAs (like RISC) require more compiler optimization, while more complex ISAs (like CISC) might handle certain tasks in fewer instructions.

* Ease of Programming

A simple ISA means fewer types of instructions, making assembly-level programming easier while a complex ISA might provide more built-in functionality, reducing the need for multiple instructions for a single task.

* Impact on Compiler Design

Complex ISA requires more effort from compilers to efficiently translate high-level code into machine instructions while a simpler ISA shifts more responsibility to the compiler, meaning more optimization must be done in software.

Thus, considering ISA at the software level helps maintain software compatibility and defines how easy it is for programmers and compilers to interact with the CPU.

b) At the hardware level (Execution)

For this level, the focus is on how instructions are executed inside the CPU and not how they are compiled as in the software level.

* Pipeline Efficiency & Performance

A simpler ISA (like RISC) allows for faster, more efficient execution because instructions are uniform and optimized for pipelining while complex ISA requires additional decoding steps to break down instructions into simpler operations.

* Flexibility & Optimization

If an ISA appears complex at the software level, CPUs can internally optimize execution to be more like a simple processor. This allows designers to strike a balance between compatibility and efficiency.

**iii) Are the latest INTEL Processors RISC?**

The latest INTEL processors are x86 architecture.

x86 uses a complex instruction set computer (CISC) design. Thus, supports a wide range of instructions that can perform multiple tasks in one operation, making it versatile but potentially more power-hungry.

**For** x86, m**emory addressing is still more complex than RISC** – Intel processors allow **direct memory operations**, which is a key CISC feature.

Although, internally, Intel processors break down CISC instructions into RISC-like micro-operations before executing them. These micro-operations are fixed-length, register-based, and optimized for pipelining—just like RISC instructions.

Even though Intel processors use RISC-like execution internally, they still cannot be called RISC processors because they support a wide range of instructions that can perform multiple tasks which is a CISC-feature and they allow direct memory operations.