**TANIA MAINA.**

**SCT212-0179/2021.**

**BCT2408: COMPUTER ARCHTECTURE.**

**EXERCISE 1.**

CPU performance equation:

Execution Time = (Instruction count \* CPI) / Clock rate

where:

Instruction Count (IC) is the total number of executed instructions.

Cycles Per Instruction (CPI) is the average number of clock cycles per instruction.

Clock rate is the processor speed in cycles per second.

Let:

1. In\_unopt be the total instruction count for the unoptimized version.
2. In\_opt be the total instruction count for the optimized version.
3. L\_unopt be the number of load and store instructions in the unoptimized version.
4. L\_opt be the number of load and store instructions in the optimized version.

* The unoptimized version runs 5% faster than the optimized one this means if the optimized version has a clock rate of X, then the unoptimized version has a clock rate of 1.05X.
* In the unoptimized version, 30% of the instructions are load and store instructions and all instructions take 1 cycle thus CPI = 1. Therefore; L\_unopt = 30% of I\_unopt = 0.3 I\_unopt
* The optimized version executes 2/3 as many loads and stores as the unoptimized one. Therefore; L\_opt = (2/3) \* L\_unopt = (2/3) \* (0.3 I\_unopt) = 0.2 I\_unopt.

Since other instruction counts remain unchanged:

The number of instructions removed by optimization is: (L\_unopt – L\_opt). Therefore:

I\_opt = I\_unopt - (L\_unopt - L\_opt)

I\_opt = I\_unopt - (0.3 I\_unopt - 0.2 I\_unopt)

I\_opt = I\_unopt - 0.1 I\_unopt = 0.9 I\_unopt

Thus, the optimized version executes 90% of the instructions compared to the unoptimized version.

*To compute the execution time:*

Since CPI = 1;

Unoptimized Version Execution Time

Time\_unopt = (In\_unopt \* 1) / 1.05X = (In\_unopt) / 1.05X

Optimized Version Execution Time

Time\_opt = (In\_opt \* 1) / X = (0.91In\_unopt) / X

To determine which version is faster:

Time\_opt / Time\_unopt = (0.91In\_unopt / X) / (In\_unopt / 1.05X)

Cancel In\_unopt and X from both sides:

Time\_opt / Time\_unopt = (0.9/1.05) = 0.857

Since 0.857 is less than 1 this means that the optimized version is faster.

To calculate the speedup:

Speedup = Time\_opt / Time\_unopt = (1 / 0.857) = 1.167. This means that the optimized version is 16.7% faster than the unoptimized version.

EXERCISE 2.

1. Instruction Frequencies from Table 1:

Load: 22.8%

Add: 14.6%

The proposed instruction combines a LOAD and ADD into a single ADD instruction with a memory operand.

CPI is unchanged.

The performance of a CPU is determined by:

CPU Time = (Instructions \* CPI \* Clock cycle time) / Total program execution

The new instruction increases the clock period by 5%. Therefore:

Original clock cycle time: T

New clock cycle time: 1.05T

For performance to remain the same:

Original execution time = New execution time

Original instruction count \* CPI \* T = New instruction count \* CPI \* 1.05T

Cancel out CPI since it remains the same

Original instruction count \* T = New instruction count \* 1.05T

Cance T from both sides

Original instruction count = New instruction count \* 1.05

New instruction count / Original instruction count = 1 / 1.05 = 0.952

This means that the total instruction count must decrease by 4.8% for performance to remain unchanged.

Load instructions originally contribute to 22.8% of all instructions.

If X is the fraction of loads eliminated, then:

New total instructions = Original instructions \* (1 − 0.228X)

Setting this equal to 95.2% of the original instruction count:

1 − 0.228X = 0.952

1 – 0.952 = 0.228X

0.048 = 0.228X

X = 0.048 / 0.228 = 0.2105

X = 21.05%. This means at least 21.05% of load instructions must be eliminated for performance to remain the same.

2. When the value is needed multiple times. If a LOAD instruction is followed by multiple instructions using the loaded value, we cannot replace all occurrences.

For example:

LOAD R1, 0(R2)  
ADD R3, R3, R1  
SUB R4, R4, R1

In the example, R1 is loaded from memory and used in both ADD and SUB. If we replace the LOAD + ADD with ADD R3, 0(R2), we still need to load R1 for SUB. A separate LOAD instruction is still required, meaning we cannot eliminate this LOAD entirely.

**DISCUSSION 1**.

Modern RISC processors still adhere to the core principles of RISC design despite incorporating features that were traditionally associated with CISC architectures. The number of instructions in modern RISC architectures such as ARM and RISC-V has significantly increased by including SIMD extensions, vector processing, and cryptographic operations. The fundamental characteristics of RISC include; fixed-length instruction encoding, a load/store architecture where memory operations are separate from arithmetic operations, and register-based computation. On the other hand, CISC processors often feature variable-length instructions and memory-to-memory operations.

Modern RISC architectures leverage compiler optimization to streamline execution and this ensures that the complexity remains at the software level rather than within the processor itself. RISC processors maintain simplicity in instruction decoding, which enhances performance. Execution pipelines have evolved to support out-of-order execution, speculative execution and superscalar processing, these advancements are however not inherently tied to CISC design but rather to overall improvements in processor efficiency. Therefore, while modern RISC processors have grown more complex in response to performance demands, they still fundamentally align with the RISC philosophy of streamlined execution, fixed-length instructions, and a register-oriented architecture, demonstrating that they are still RISC at their core.

**DISCUSSION 2.**

Modern Intel processors present a unique blend of CISC and RISC characteristics, making their classification dependent on the level of analysis. At the software interface, Intel CPUs are undoubtedly CISC, as they support the x86 instruction set, which includes complex, variable-length instructions, memory-to-memory operations, and a vast array of legacy instructions for backward compatibility. This means that from the perspective of compilers and programmers, Intel processors retain all the features of traditional CISC architectures. However, at the microarchitectural level, Intel CPUs translate these complex CISC instructions into smaller, fixed-length RISC-like micro-operations before execution. These micro-operations, which operate primarily on registers, enable efficient pipelining, out-of-order execution, and instruction-level parallelism which are distinct features of RISC design.

Despite this internal transformation, Intel processors cannot be classified as purely RISC, as their external interface remains fundamentally CISC. The need for backward compatibility forces Intel to retain CISC features at the instruction set level, even if the execution pipeline behaves more like a RISC processor. Therefore, the classification of modern Intel processors depends on the level: if evaluated based on their instruction set, they are CISC; if evaluated based on their execution strategy, they resemble RISC. This hybrid approach allows Intel to maintain compatibility with older software while leveraging RISC-like efficiencies for performance optimization.