BCT 2408 Computer Architecture-II: Lab 1 Solution

Name: Jeff Gicharu
Reg No: SCT212-0053/2021

This document provides the solutions to the problems presented in Lab 1, focusing on the CPU performance equation and Instruction Set Architecture (ISA) concepts.

E1: Performance Impact of Procedure Call Optimization (individual or groups of 2, 15 mins)

Problem *(Based on H&P 2/e 1.6 p.61)*

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 (CPI = 1).

Which is faster? Justify your decision quantitatively.

- **Answer:**
  Let the subscripts 'unop' denote the unoptimized version and 'op' denote the optimized version. We use the CPU performance equation: CPU Time = Instruction Count (IC) * CPI * Clock Cycle Time (T_cycle).
  Given Information:
  1. Clock Rate Relation: Clock Rate_unop = 1.05 * Clock Rate_op. This implies T_cycle_op = 1.05 * T_cycle_unop. (Alternatively, T_cycle_unop = T_cycle_op / 1.05 ≈ 0.952 * T_cycle_op. The solution file uses T_cycle_unop = 0.95 * T_cycle_op, which corresponds to the *optimized* version having a 5% higher clock rate, or the *unoptimized* clock rate being 5% lower. Let's follow the solution file's interpretation for consistency: T_cycle_unop = 0.95 *

T_cycle_op).

2. Load/Store Fraction (Unoptimized): IC_ld/st, unop = 0.30 * IC_unop.
3. Other Instructions Fraction (Unoptimized): IC_other, unop = (1 - 0.30) * IC_unop = 0.70 * IC_unop.
4. Load/Store Count (Optimized): IC_ld/st, op = (2/3) * IC_ld/st, unop = (2/3) * 0.30 * IC_unop = 0.20 * IC_unop.
5. Other Instructions Count (Optimized): IC_other, op = IC_other, unop = 0.70 * IC_unop.
6. CPI: CPI_unop = CPI_op = 1.

Calculate Total Instruction Count for Optimized Version:IC_op = IC_ld/st, op + IC_other, opIC_op = (0.20 * IC_unop) + (0.70 * IC_unop) = 0.90 * IC_unop.Calculate CPU Times:CPU Time_unop = IC_unop * CPI_unop * T_cycle_unopCPU Time_unop = IC_unop * 1 * (0.95 * T_cycle_op) = 0.95 * IC_unop * T_cycle_op.CPU Time_op = IC_op * CPI_op * T_cycle_opCPU Time_op = (0.90 * IC_unop) * 1 * T_cycle_op = 0.90 * IC_unop * T_cycle_op.Compare CPU Times:Since 0.90 * IC_unop * T_cycle_op is less than 0.95 * IC_unop * T_cycle_op, the optimized version has a lower CPU time.Calculate Speedup:Speedup = CPU Time_unop / CPU Time_opSpeedup = (0.95 * IC_unop * T_cycle_op) / (0.90 * IC_unop * T_cycle_op)Speedup = 0.95 / 0.90 ≈ 1.056.*Therefore, the **optimized version is faster** by approximately 5.6%.*

E2: Performance Impact of Register-Memory Addressing Mode (individual or groups of 2, 10 mins)

***Problem** (Based on H&P 2/e 2.6 p.164)*

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 (cycle time) of the CPU to increase by 5%. Use the instruction frequencies for the gcc benchmark on the load-store machine from Table 1 (provided in the PDF). The new instruction affects only the clock cycle time 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?
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.

- **Answer (1): Percentage of Loads to Eliminate**
  Let 'unop' be the original load-store machine and 'op' be the machine with the new ADD Reg, Mem instruction. We want CPU Time_op ≤ CPU Time_unop.
  IC_op * CPI_op * T_cycle_op ≤ IC_unop * CPI_unop * T_cycle_unop.
  Given Information:
  1. Clock Cycle Time Relation: T_cycle_op = 1.05 * T_cycle_unop.
  2. CPI Relation: CPI_op = CPI_unop. (Let's assume CPI=1 for simplicity, although the exact value cancels out).
  3. Load Frequency (Unoptimized): From Table 1, Freq_load = 22.8% = 0.228. So, IC_load, unop = 0.228 * IC_unop.
  4. ADD Frequency (Unoptimized): From Table 1, Freq_add = 14.6% = 0.146.

  Let X be the fraction of original LOAD instructions that are eliminated by being merged into the new ADD Reg, Mem instruction.
  - Number of eliminated LOADs = X * IC_load, unop = X * 0.228 * IC_unop.
  - Number of eliminated ADDs (that followed those loads) = X * IC_load, unop = X * 0.228 * IC_unop.
  - Number of new ADD Reg, Mem instructions = X * IC_load, unop = X * 0.228 * IC_unop.

  Calculate IC_op:The number of instructions removed is the number of eliminated LOADs.IC_op = IC_unop - (Number of eliminated LOADs)IC_op = IC_unop - (X * 0.228 * IC_unop) = IC_unop * (1 - 0.228 * X).Apply Performance Condition:IC_op * CPI_op * T_cycle_op ≤ IC_unop * CPI_unop * T_cycle_unop[IC_unop * (1 - 0.228 * X)] * CPI_unop * (1.05 * T_cycle_unop) ≤ IC_unop * CPI_unop * T_cycle_unopDivide both sides by IC_unop * CPI_unop * T_cycle_unop:(1 - 0.228 * X) * 1.05 ≤ 11.05 - 0.2394 * X ≤ 10.05 ≤ 0.2394 * XX ≥ 0.05 / 0.2394X ≥ 0.2088*Therefore, at least **20.88%** (or approximately 21-22% as per the solution file's calculation) of the original LOAD instructions must be eliminated (and converted to the new format) for the modified machine to have at least the same performance*.
- **Answer (2): Situation Preventing Replacement**
  The replacement LOAD Rx, 0(Rb) followed by ADD Ry, Ry, Rx with ADD Ry, 0(Rb) is not semantically equivalent if the destination register Ry is the *same* as the loaded register Rx.

Example:
Consider the sequence:
LOAD R1, 0(R5) # Load value from Mem[R5] into R1
ADD R1, R1, R1 # Add the loaded value to itself, store in R1

Assume R5 contains address 1000, Mem[1000] contains the value 4.

- After LOAD R1, 0(R5), R1 contains 4.
- After ADD R1, R1, R1, R1 contains 4 + 4 = 8.

Now consider the proposed replacement:ADD R1, 0(R5)   # Add value from Mem[R5] to the *original* value of R1
Assume R1 initially held some unrelated value, say 47, and R5 and Mem[1000] are as before.

- The ADD R1, 0(R5) instruction would read the original value of R1 (47) and the value from Mem[1000] (4), add them, and store the result in R1.
- Result: R1 would contain 47 + 4 = 51.

This result (51) is incorrect compared to the original sequence's result (8). The replacement fails when the destination of the ADD is the same register being loaded.

D1: Discussion - Modern RISC vs CISC (groups of 4, 15 mins)

Problem

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?)

- **Answer (Discussion Points):**
  - **Are Modern RISC Still RISC?** While the *number* of instructions in modern RISC ISAs (like ARMv8, RISC-V, MIPS) has grown significantly compared to early RISC designs, they generally retain the core *philosophies* that defined RISC:
    - **Load-Store Architecture:** ALU operations primarily work on registers, requiring explicit load/store instructions to move data to/from memory. This remains a key differentiator from many classic CISC designs that allowed memory operands for ALU instructions.

        - **Fixed-Length Instructions (mostly):** Many RISC ISAs still prioritize fixed-length instructions (e.g., 32-bit for MIPS/RISC-V base) or simple variable-length schemes (e.g., ARM Thumb, RISC-V C extension) which simplify instruction fetch and decode compared to complex CISC encodings (like x86).
        - **Simple Addressing Modes:** RISC ISAs tend to favor a smaller set of simpler addressing modes compared to the wide variety found in some CISC ISAs.
        - **Focus on Pipelining:** The design principles facilitate efficient pipelining.
    - **Defining Features of Modern RISC Simplicity:** Rather than just the instruction count, the defining features of RISC simplicity today are arguably:
        - **Load-Store Nature:** This is fundamental. It simplifies pipeline design as memory access is isolated to specific instructions.
        - **Instruction Encoding Regularity:** Even with extensions, the base instruction formats are often simpler and more regular than CISC, aiding faster decoding.
        - **Emphasis on Register Operands:** ALU operations predominantly use registers, simplifying data forwarding and reducing pipeline stalls compared to memory-based operands.
    - **Conclusion:** While modern RISC ISAs are more complex than their ancestors (adding SIMD, floating-point, specialized instructions), they largely adhere to the foundational principles that distinguish them from traditional CISC ISAs, particularly the load-store nature and relative instruction format simplicity. The term "RISC" still applies, but the definition has evolved beyond a strict, low instruction count.

D2: Discussion - Classifying Modern x86 (groups of 4, 10 mins)

Problem

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, they are RISC. Others say that, since at the software interface (compiler) they are seen as CISC, they are CISC. Discuss at what level ISA complexity should be measured. What are the implications of considering the ISA at each level? Are the latest Intel processors RISC?

- **Answer (Discussion Points):**
  - **Level of Measurement:** ISA complexity can be viewed at two main levels:
    - **Architectural Level (Software/Compiler View):** This is the interface defined by the instruction set manual (e.g., the x86 instruction set). At this level, x86 is undeniably **CISC**. It features variable-length instructions, complex addressing modes, instructions that perform complex operations (like string manipulation), and memory operands for many ALU instructions. Compilers target this complex ISA.
    - **Microarchitectural Level (Internal Hardware View):** This is how the ISA is *implemented* internally. Modern x86 processors (like Intel Core/Xeon, AMD Ryzen) employ **RISC-like techniques**. They translate complex x86 instructions into simpler, fixed-length internal operations called **micro-operations (µops)**. These µops are then processed using techniques common in high-performance RISC designs: deep pipelining, register renaming, out-of-order execution, superscalar issue, and sophisticated branch prediction, operating on a large set of physical registers.
  - **Implications:**
    - **Architectural Level:** Defines software compatibility, compiler complexity, and code density. Maintaining the CISC x86 architecture ensures backward compatibility for a vast software ecosystem. However, it makes compiler design more challenging and instruction decoding complex.
    - **Microarchitectural Level:** Determines the actual performance, power efficiency, and hardware complexity. Using RISC techniques internally allows designers to achieve high performance despite the complex front-end ISA. It requires significant hardware resources for decoding and managing the µops.
  - **Are Latest Intel Processors RISC?**
    - From the **software perspective (ISA)**, they are **CISC**. They execute the x86 instruction set.
    - From the **internal hardware implementation perspective (microarchitecture)**, they employ many **RISC principles** to execute those CISC instructions efficiently.
    - **Conclusion:** It's most accurate to say they have a **CISC architecture implemented using a RISC-like microarchitecture**. They are not purely RISC because the fundamental software-facing interface remains CISC. The RISC elements are an implementation detail hidden from the programmer/compiler.