1. ***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***  
Compare two versions of a CPU:

* Unoptimized: Higher clock rate (5% faster), 30% loads/stores.
* Optimized: Executes ⅔ as many loads/stores, same CPI (1 cycle/instruction).

Solution:

1. Define Variables:
   * Let clock rate of optimized =  C.
   * Unoptimized clock rate =  1.05C.
   * Let total instructions in unoptimized =  I.
2. Instruction Breakdown:
   * Loads/stores in unoptimized:  0.3I
   * Other instructions:  0.7I
   * Optimized loads/stores:  2/3 × 0.3I = 0.2I
   * Total optimized instructions:  0.7I + 0.2I = 0.9I
3. Execution Time:
   * Unoptimized:  I/1.05C
   * Optimized:  0.9I/C
4. Performance Ratio:

Optimized Time/Unoptimized Time= (0.9I/C) / (I/1.05C) = 0.9 × 1.05 = 0.945

Since 0.945<1, the optimized version is 5.5% faster.

Therefore, the optimized version is faster by approximately 5.5%.

1. ***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.***

***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.***

1. Percentage of Loads to Eliminate:

* Let original clock cycle = T New clock cycle = 1.05T
* Let *x* = fraction of loads eliminated.
* Total instructions saved = 0.228xT
* New instruction count = T−0.228xT=T(1−0.228x)

Performance Equation: (1−0.228x) ×1.05≤1

⟹   x≥0.05/0.2394 = 20.89%

At least 20.89% of loads must be eliminated.

2. Example Where Replacement Fails:

LOAD Rx, 0(Rb)

ADD Rz, Rb, R4

ADD Ry, Ry, Rx

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 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?)***
2. Core RISC Principles Retained:
   * Load-Store Architecture: Data operations (e.g., ADD, MUL) only work on registers, not memory.
   * Fixed-Length Instructions: Simplifies pipelining and decoding (e.g., ARM A32, RISC-V base ISA).
   * Orthogonal Instruction Set: Uniform register usage and addressing modes.
3. Evolution of Modern RISC:
   * Increased Instruction Count: Modern RISCs (e.g., ARMv9, RISC-V) add instructions for:
   * Microarchitecture Advancements: Out-of-order execution, speculative execution, and multi-issue pipelines (common in high-performance RISC CPUs like Apple M-series).

* Classic RISC (e.g., MIPS I) prioritized minimalism (e.g., 32 instructions).
* Modern RISC focuses on pipeline efficiency and hardware simplicity, even with more instructions. For example:
  + RISC-V allows modular extensions (e.g., "F" for floating-point), keeping the base ISA simple.
  + ARM Thumb-2 uses variable-length instructions but retains RISC principles internally.

Conclusion:  
Yes, modern RISC ISAs remain true to the original philosophy. Simplicity is defined by hardware-friendly design (e.g., fixed encoding, load-store discipline) rather than raw instruction count.

1. ***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?***
2. x86 as a CISC ISA:
   * Complex Instructions: Single instructions like REP MOVSB (block memory copy) or PCLMULQDQ (carry-less multiplication).
   * Variable-Length Encoding: Instructions range from 1 to 15 bytes.
   * Memory-Memory Operations: e.g., ADD [mem], [mem].
3. RISC Under the Hood:
   * Micro-Op Translation: Modern x86 CPUs (e.g., Intel Core, AMD Zen) decode CISC instructions into RISC-like micro-ops.
   * Example: ADD [mem], [mem] splits into LOAD → ALU → STORE micro-ops.
   * Benefits:
     + Enables deep pipelining, superscalar execution, and efficient out-of-order scheduling.
     + Reduces power consumption compared to legacy CISC designs.

* Software View (CISC): Compilers and programmers interact with x86’s complex ISA.
* Hardware View (RISC): Micro-ops simplify execution but require complex decoders and µOp caches.
* Legacy Burden: x86 maintains backward compatibility, limiting radical simplification.
* Performance Tradeoffs:
  + Pros: High performance via micro-ops and advanced pipelines.
  + Cons: Decoder complexity and power overhead.

Conclusion:  
Intel x86 is CISC at the ISA level but uses RISC-inspired microarchitecture techniques. Complexity should be measured at the software interface (ISA), as that defines compatibility and programming effort.