COMPUTER ARCHITECTURE LAB 1

VICTOR MWAI SCT212-0202/2021

MARCH 13, 2025

Computer Architecture LAB 1 Solutions

E1: Performance Comparison of Optimized vs. Unoptimized Versions

Problem Statement

After analyzing the use of high-level language constructs, it was found that procedure calls are expensive operations. A new optimization reduces the number of loads and stores associated with procedure calls. Given:

- The unoptimized version has a 5% higher clock rate.

- 30% of its instructions are loads and stores.

- The optimized version executes only 2/3 of the original loads and stores, while all other instructions remain unchanged.

- All instructions (including loads and stores) execute in one clock cycle.

Solution

Step 1: Define Performance Metrics

The execution time formula is:
Execution Time = (Instruction Count × CPI) / Clock Rate

Step 2: Compute Instruction Counts

Assume the unoptimized version executes 100 instructions:
- 30% are loads/stores = 30 instructions
- The optimized version executes 2/3 of these loads/stores = 20 fewer instructions
- Total optimized instruction count = 80

Step 3: Compare Execution Times

Execution time for unoptimized:
ET_unoptimized = 100 / CR

Execution time for optimized:
ET_optimized = 80 / (0.95CR)

Speedup:
Speedup = (ET_unoptimized) / (ET_optimized) = (100/80) × 0.95 = 1.1875

Conclusion

The optimized version is 18.75% faster, despite a slightly lower clock rate, because it executes fewer instructions.

E2: Effect of Register-Memory Addressing Mode on Performance

Problem Statement

A proposed register-memory addressing mode removes separate load instructions, replacing:
LOAD Rx, 0(Rb)
ADD Ry, Ry, Rx
with:
ADD Ry, 0(Rb)

Solution

Since the new instruction increases the clock period by 5%, we calculate how many loads must be eliminated to maintain the same performance.

Step 1: Define Performance Metrics

Performance = 1 / Execution Time, where:

execution time is (instruction count x CPI)/Clock rate

Since CPI remains the same, the clock rate drops by 5%, making it 0.95 of the original

Step 2: Determine Required Load Reduction

Let L be the fraction of eliminated loads. To maintain performance:
$(1 - L) \times 0.95 = 1$
Solving for L:
$L = 1 - (1/0.95) = 5.3\%$

Step 3: Identify a Case Where Replacement Fails

If a load instruction is followed by another instruction that modifies the same register before the value is used elsewhere, the replacement is not possible.

Example

LOAD R1, 0(RB)

ADD R2, R1, R3

For the case above, if R1 is used before modification in another instruction, replacing LOAD with ADD changes program behavior.

Conclusion

At least 5.3% of loads must be eliminated to maintain performance. Some cases cannot be replaced due to dependence, that is, replacement is not possible if the loaded value is used elsewhere before modification.

D1: Are Modern "RISC" Processors Still RISC?

Yes, but with evolution. Originally, RISC meant simple instructions, fixed encoding, and load/store design. Key features include:

- **Register-based execution** - Most instructions still operate on registers, with memory access limited to explicit load/store operations.
- **Fixed-length instructions** - Modern RISC architectures like ARM and RISC-V still use fixed instruction sizes, simplifying decoding.
- **Efficient pipelines** - Instructions execute in one or a few cycles, supporting high-speed execution.

However, modern RISC also includes:

- **More instructions** = More Complexity – Modern RISC architectures now include vector processing, floating-point operations, and specialized instructions (e.g., ARM's NEON, MIPS DSP extensions)
- **Some micro-coding** - Some processors now use microcode, blurring the line between RISC and CISC.
- **Variable-length encoding in some cases** - While core RISC designs use fixed instructions, compressed instruction sets (e.g., ARM Thumb, RISC-V C extension) introduce variable-length encoding, a feature once associated with CISC.

Conclusion

Modern RISC maintains its fundamental characteristics but has adapted to modern computing needs.

D2: Is Intel's x86 Now a RISC or Still a CISC?

Modern x86 behaves like RISC at the hardware level but remains CISC in software. Key observations:

- Internally, x86 uses RISC-like micro-operations and pipelining:

- **Micro-operations** break complex instructions into **simpler, fixed-size** instructions.
- **Pipelining and Out-of-Order Execution** improve performance as modern x86 CPUs use deep pipelines and execute instructions in parallel.

- **Register-based execution** is prioritized that is, while x86 started with memory-to-memory operations, modern CPUs rely heavily on register-based execution (like RISC).

  - Externally, x86 still has variable-length CISC instructions and legacy complexity.

- **Variable-length instructions** (software still sees CISC instructions).
- **Legacy compatibility** (decades-old features remain).
- **Complex decoders** are required for instruction translation.

Conclusion

x86 is RISC-like at the microarchitecture(hardware) level but remains CISC at the instruction set(software) level.