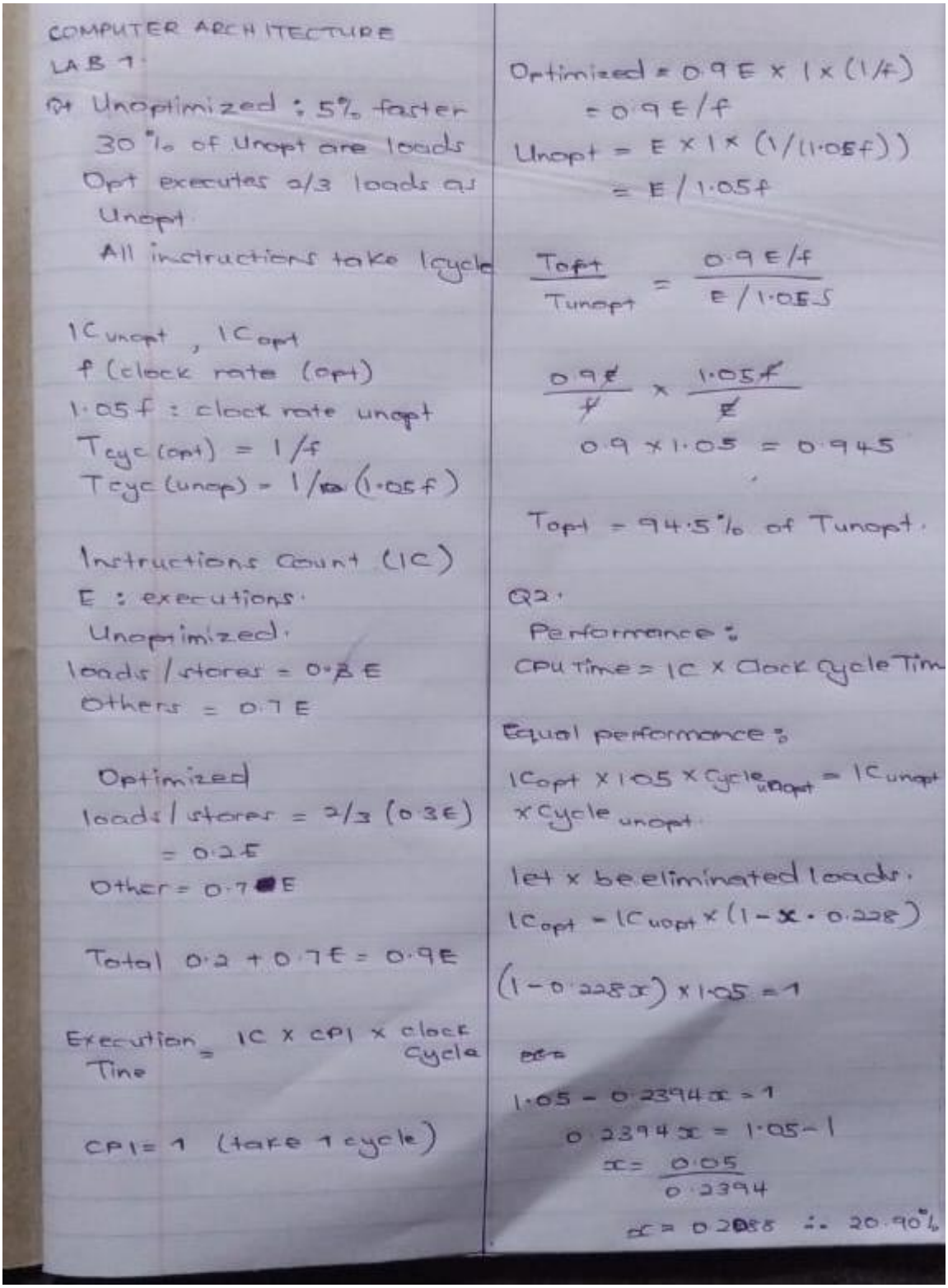
WANJOHI ELVIS MUTHUMBI. SCT212-0043/2021

COMPUTER ARCHITECTURE. LAB 1. Q1 & Q2



COMPUTER ARCHITECTURE
LAB 1.

Q1 Unoptimized : 5% faster
30% of Unopt are loads
Opt executes 2/3 loads as Unopt
All instructions take 1 cycle

IC_{unopt} , IC_{opt}
f (clock rate (opt))
1.05f : clock rate unopt
$T_{cyc(opt)} = 1/f$
$T_{cyc(unop)} = 1/(1.05f)$

Instructions Count (IC)
E : executions.
Unoptimized.
loads / stores = 0.3 E
Others = 0.7 E

Optimized
loads / stores = 2/3 (0.3E)
= 0.2E
Other = 0.7E

Total 0.2 + 0.7E = 0.9E

Execution Time = IC x CPI x clock cycle

CPI = 1 (take 1 cycle)

Optimized = 0.9E x 1 x (1/f)
= 0.9E/f
Unopt = E x 1 x (1/(1.05f))
= E / 1.05f

$\frac{T_{opt}}{T_{unopt}} = \frac{0.9E/f}{E/1.05f}$

$\frac{0.9E}{f} \times \frac{1.05f}{E}$

$0.9 \times 1.05 = 0.945$

T_{opt} = 94.5% of T_{unopt}.

Q2.
Performance :
CPU time = IC x Clock Cycle Tim

Equal performance :
$IC_{opt} \times 1.05 \times Cycle_{unopt} = IC_{unopt} \times Cycle_{unopt}$

let x be eliminated loads.
$IC_{opt} = IC_{uopt} \times (1 - x \cdot 0.228)$

$(1 - 0.228x) \times 1.05 = 1$

$1.05 - 0.2394x = 1$
$0.2394x = 1.05 - 1$
$x = \frac{0.05}{0.2394}$
$x = 0.2088 \therefore 20.90\%$

DISCUSSION 1.

Yes, they deviate from the original 1980s RISC ideals:

1. Increased Instruction Count: The initial RISC philosophy, espoused by the MIPS and Berkeley RISC initiatives, centered around a limited, fixed-size instruction set. Modern RISC implementations like ARM and RISC-V, while still adhering to many of the principles of RISC, have a significantly larger instruction set and addressing modes than the original implementations. This expansion has been due to the need to support a wider range of applications as well as improve performance in areas (e.g., multimedia, cryptography).
2. More Advanced Instructions: While still generally simpler than CISC instructions, modern RISC sets sometimes feature instructions that perform more sophisticated operations than the rudimentary load/store, arithmetic, and control flow instructions of the early RISC era. These include targeted SIMD instructions, atomic memory instructions, and more complex addressing modes than base + offset.
3. Variable Length Encoding (to some extent): In the quest for uniformity, a few modern RISC designs have introduced mechanisms allowing for flexibility in instruction length to accommodate a greater opcode space or immediate operand values. That is somewhat a deviation from the strict fixed-length encoding of early RISC designs.

No, they still adhere to fundamental RISC principles:

Despite the increased complexity, modern "RISC" designs usually still adhere to the simple principles that launched the RISC revolution:

1. Load-Store Architecture: Still a mainstay. Data processing operations primarily occur between registers. Memory access is limited to load and store instructions only. Instruction formats and pipeline design become simpler as a consequence.
2. Emphasis on Register-to-Register Operations: The majority of instructions manipulate data held in registers, leading to faster execution compared to memory operands.
3. Simplified Addressing Modes: Even though there may be more modes than in the initial RISC designs, there are comparatively fewer and simpler modes than in conventional CISC designs. Such complicated addressing computations usually are performed through sequences of comparatively simple arithmetic instructions.
4. Fixed and Simple Instruction Encoding (mostly): While some modern RISC designs could have slight variations, instruction encoding is generally mechanical, with instruction fetching and decoding becoming simpler and faster compared to CISC's variable and complex encoding.
5. Pipelining Focus: The RISC structures were created keeping in mind that they should emphasize efficient pipelining. Uniformity of the instruction formats and simplicity of the operations allow more and deeper pipeline efficiency, and this is useful for high levels of performance.

DISCUSSION 2

The "level" on which we quantify ISA complexity is purely a matter of our perspective and the questions we're trying to resolve:

The Software Interface (Compiler/Programmer's View): This is the architectural level that software communicates with directly. Complexity here is quantified by:

Number and Types of Instructions: A large and diverse instruction set with many addressing modes and special instructions would be more complex.
Instruction Form: Variable-length instructions with complex encoding schemes bring in complexity.
Addressing Modes: Numerous and complex ways of addressing memory operands contribute to complexity.
Instruction Semantics: Instructions involving multiple, complex operations packed in a single step add complexity.
Micro-architectural Level (Processor Implementation):

This refers to how the processor works internally with the ISA. Complex here has to do with:

Instruction Decoding: How difficult is it to translate ISA instructions into processor internal execution?
Execution Pipeline: How complex does the pipeline need to be to support different instruction types and execution requirements?
Control Logic: How complex does the control unit need to be to direct the execution of the ISA?
Implications of Assessing the ISA at Each Level:

Software Interface Level:

Compiler Design: A complex ISA can significantly complicate compiler design.
Compilers have to strategically select and sequence instructions from a massive inventory, with complex addressing modes and special instructions, in order to
generate effective code. A complete ISA would actually allow for more concise code sequences for certain uses.
Assembly Language Programming: Programmers programming in assembly explicitly can make it harder to learn and use a challenging ISA due to the large number of instructions and how different they are from one another.
Code Density: CISC ISAs usually aim for higher code density
by encasing more intricate operations within fewer bytes. This can have ramifications on instruction cache performance and footprint (though in modern caches it is mitigated somewhat).
Portability: A software for a complex, platform-specific ISA tends to be less portable to other architectures.

Micro-architectural Level:

Processor Design Complexity: A direct implementation of a complex ISA in hardware can lead to a more complex and possibly less efficient processor design. Early RISC designs attempted to minimize the hardware complexity by having a more regular and predictable instruction set.
Performance Potential: Novel micro-architectures, even for CISC ISAs, can achieve high performance by breaking down intricate instructions into easy-to-handle micro-operations that can be pipelined and executed in parallel effectively. The underlying micro-architecture generally plays a greater performance role than ISA complexity at the software level.

Power Consumption: A sophisticated micro-architecture can use more power in directly executing a sophisticated ISA. But modern designs employ a number of power-saving techniques regardless of the sophistication of the ISA at the software level.

Translation Overhead (for CISC implementations): When a CISC instruction is translated into a set of simpler micro-operations, there is an initial overhead for this translation. But modern processors employ sophisticated decoders and micro-op caches to minimize this overhead.

Are the New Intel Processors RISC?

At the level of the processor to the software interface, the new Intel processors are unmistakably CISC. Compilers and programmers still deal with the x86 instruction set with its variable-length instructions, lots of addressing modes, and subtle operations.
The ISA on which software is operating beneath is still CISC.

At the micro-architectural level, recent Intel processors extensively utilize RISC concepts in their architecture. They break down complex x86 instructions into sequences of simpler, fixed-length micro-operations (usually called μops). These μops are processed by a micro-architecture which has the following features:

Register-based operations: μops primarily operate on internal registers.
Pipelining: μops pass through a deep and complex pipeline.
Simple micro-instructions: μops perform simple operations.
Fixed-length micro-instructions: This simplifies internal processing.
Load-store architecture (at the μop level): Memory access is typically done by dedicated load and store μops.