A Comparative Analysis of Modern Instruction Set Architectures: ARMv9-A, x86-64, MIPS32, and RISC-V

Nadim Alamleh (0247841), Faris Omise (0241387), Zaid Shehadeh (0216065), Hussam mohammed (0207944)
Department of Computer Engineering,
The University of Jordan, Amman, Jordan

***Abstract*—This research investigates four different ISAs: ARMv9-A (AArch64), x86-64, and MIPS32, with RISC-V being the primary reference for comparison. ISAs are compared in terms of their register organization, addressing modes, arithmetic, logical, and memory access instructions, and encoding. It also demonstrates how the architectural decisions made by each ISA influence hardware implementation factors such as processor performance, code density, decoder complexity, and energy efficiency, while considering the trade-offs of the design decision. Additionally, the research examines what application is most appropriate for each ISA in various application domains, including embedded systems, servers, desktop computers, mobile devices, IoT devices, and high-performance computing systems.**

I. Introduction

The instruction set architecture (ISA) is an abstract interface between the hardware and the machine code that includes all information necessary to write a low-level program that will run according to the design of the registers, instruction set, memory architecture, addressing modes, and data types. It could be described as the grammar and vocabulary of the machine language that the processor can decode and execute. ISAs are split into two main types: CISC and RISC. Complex Instruction Set Computers, or CISC for short, have many instructions and addressing modes with different levels of complexity. This can lead to shorter programs, but more complex instruction formats and addressing modes can increase implementation and decoding complexity due to their intricacy. RISC, or Reduced Instruction Set Computers, however, have fewer and simpler instructions and addressing modes that are of fixed size. This can lead to longer programs, but simpler instructions can reduce implementation and decoding complexity.

Creating a new ISA is expensive and difficult, so it would make sense to keep reusing an ISA to save effort. However, different ISAs have been developed due to many differing design objectives and computing requirements. Some ISAs prioritize performance, while others focus on energy efficiency. For example, the original MIPS user-level integer instruction set comprised just 58 instructions and was straightforward to implement as a single-issue, in-order pipeline, while Intel's x86 comprises 1300 instructions and dozens of special-purpose registers and multiple address-translation schemes [1]. This illustrates how important the ISA design is in a modern computing system, since it can affect hardware complexity, performance, energy efficiency, code density, compiler optimization, and application suitability.

This research compares four different ISAs: ARMv9-A (AArch64), x86-64, and MIPS32, with RISC-V being the focal point of comparison. For each ISA, design philosophy, application suitability, and architectural characteristics will be discussed, and a deep architectural analysis and instruction set comparison will be conducted to highlight the fundamental engineering trade-offs inherent in their respective microarchitectural implementations.

II. Overview of the Selected ISAs

A. Design Philosophy and Architectural Classification

The architectural spectrum is defined by the contrasting design philosophies of the CISC and RISC methodologies. **x86-64** is an architecture that lies at the absolute CISC extreme. Its lineage dates back to Intel's 16-bit 8086 processor, then evolved into the 32-bit IA-32 standard, and finally becoming the 64-bit AMD64 (adopted by Intel as Intel 64) [2]. One of its main characteristics is maintaining backwards compatibility. The architecture tolerates immense hardware decoder complexity in exchange for highly expressive, variable-length instructions that allow a single macro-operation to address memory, compute a result, and write back simultaneously [2].

ARMv9-A (AArch64)'s main attributes, however, are processing performance, very low power consumption, implementation size, strict energy efficiency, and backwards compatibility. It uses RISC technologies and a rigid load-store memory model [3]. The AArch64 state deliberately omits the mixed 16-bit/32-bit Thumb encoding that characterized earlier ARM architectures to enforce a cleaner, more regular 32-bit instruction format [3]. To maintain software continuity across varying system needs, it exposes two distinct execution states—AArch64 providing 64-bit general-purpose registers, and AArch32 preserving legacy 32-bit execution.

Similar to **ARMv9-A**, **MIPS32** also uses RISC technologies, but on a stricter and purer level. It is a deliberately minimalist, load-store architecture that heavily targets embedded computing spaces, consumer electronics, and digital signal processing (DSP) applications [1]. The architecture relies on simplifying internal hardware to drive high base clock frequencies, deliberately avoiding complex instructions. It enforces strict memory

alignment (byte-addressed but fully aligned) and restricts its instruction set to exactly three geometric formats [4].

RISC-V introduces a modern paradigm to classic RISC principles through an explicit, royalty-free *modularity-first* philosophy. Instead of freezing a massive, monolithic instruction set that forces hardware vendors to implement rarely used legacy instructions, RISC-V specifies a minimal Base Integer ISA (e.g., RV32I or RV64I) [5]. System designers can then cleanly augment this base by composing optional standard extensions: the M-extension for integer multiply/divide, F and D for floating-point operations, V for scalable vector processing, and C for 16-bit compressed encodings [6]. This modular abstraction produces a highly simplified, clean datapath.

B. Licensing Models

Licensing operates as a first-order architectural constraint, dictating who may legally fabricate, customize, and commercialize the silicon. **x86-64** is strictly proprietary. While the full instruction specification is publicly documented, implementing the ISA in custom silicon requires a restrictive licence from Intel or AMD [2]. **ARMv9-A** operates on a similar proprietary model controlled by Arm Ltd. Companies such as Apple, Qualcomm, NVIDIA, and Samsung hold architectural licenses that permit them to design custom microarchitectures, provided the resulting cores strictly satisfy the ISA compliance specifications [7].

MIPS32 is historically proprietary and features a largely static ecosystem today. In contrast, **RISC-V** is a fully open standard [5]. Any academic institution, startup, or large commercial entity may design, fabricate, modify, and sell RISC-V processors without paying royalties or negotiating legal licenses, democratizing custom silicon design.

C. Target Application Domains

x86-64 completely dominates **desktop computing**, **laptops**, and **cloud datacenters**. This dominance is driven by its raw single-thread throughput, large multi-level caches, and indispensable hardware virtualization support [8]. **ARM** controls virtually all mobile computing due to its efficient power-to-performance ratio, and has expanded aggressively into HPC and supercomputing. Examples of these are Fugaku's A64FX-powered supercomputer and Microsoft's Azure Cobalt 100 server processors [9]. **MIPS32** remains heavily ingrained in networking equipment and **real-time consumer DSP** devices due to its highly deterministic pipelines and minimal execution latency [4]. **RISC-V**'s royalty-free modularity has established it as the premier ISA for **custom silicon, AI hardware accelerators, and deep IoT endpoints** where power and area must be strictly minimized without licensing overhead [6].

III. Instruction Set Comparison

A. Instruction Encoding, Formats, and Length

Instruction encoding dictates decoder complexity, pipeline latency, and binary code density. Table I details the encoding strategies across the four architectures.

x86-64 utilizes a highly complex variable-length byte stream ranging from 1 to 15 bytes. A single instruction may contain up to four optional legacy prefix bytes, a 1- to 3-byte opcode, an optional ModRM byte (for addressing form), an optional Scale-Index-Base (SIB) byte, an optional displacement ranging from 1 to 4 bytes, and an optional immediate operand of up to 4 bytes [2]. The REX prefix must be placed immediately before the opcode to unlock 64-bit operand sizes and access the extended R8-R15 registers. Because instruction boundaries are impossible to identify without sequentially parsing the entire stream, decoding is fundamentally serialized.

ARMv9-A, MIPS32, and RISC-V all enforce fixed 32-bit widths for their base instruction sets. This structural guarantee means instruction boundaries are always known at fetch time, which allows the processor front-end to fetch and decode multiple instructions in parallel. **MIPS32** restricts itself to exactly three geometric formats: R-Type (utilized for integer register-to-register operations consisting of opcode, rs, rt, rd, shamt, and funct code), I-Type (for data transfers, conditional branches, and immediate math), and J-Type (for unconditional jumps with a 26-bit address field) [1]. **RISC-V** continues on this design philosophy by requiring the source registers (rs1, rs2) and the destination register (rd) to occupy identically placed bit positions across all six of its instruction formats (R, I, S, B, U, J). This strict uniformity allows hardware to read the register file speculatively in parallel with opcode decoding [5].

B. Register Organization, Architecture, and Conventions

The organization of the register file directly impacts compiler optimization capabilities and pipeline hazard management. **x86-64** provides 16 general-purpose 64-bit registers, addressable at 8-, 16-, 32-, or 64-bit granularity through aliased names (e.g., AL, AX, EAX, RAX). Several registers retain implicit roles. Under the System V AMD64 ABI (used in Linux and macOS), specific registers (RDI, RSI, RDX, RCX, R8, R9) are mandated for passing the first six integer arguments. However, the Microsoft x64 convention assigns only RCX, RDX, R8, and R9 to arguments and requires the caller to reserve 32 bytes of shadow space per call [2], [10].

ARMv9-A provides a highly orthogonal file of 31 general-purpose 64-bit registers (X0–X30), accessible in 32-bit form as W0–W30, with the lower significant half storing the 32 bits. A dedicated 64-bit program counter (PC), stack pointer (SP), and Exception Link Registers (ELR_EL#) exist independently. AArch64 also provides 32 128-bit SIMD/FP registers (Q0–Q31) that can be accessed depending on bit usage as 128, 64, 32, 16, or 8-bit registers [3].

MIPS32 provides 32 general-purpose 32-bit registers ($0–$31) and 32 floating-point registers. Software conventions strictly structure this file: $zero always holds the constant 0, $s0–$s7 are callee-saved temporaries, $t0–$t9 are caller-saved non-preserved temporaries, and $ra holds the return address [1]. **RISC-V** similarly provides 32 general-purpose registers (x0-x31) with x0 permanently hardwired to zero. The temporary registers are x5-x7 and x28-x31, which are stored

TABLE I
COMPREHENSIVE COMPARISON OF INSTRUCTION ENCODING

Architectural Feature	x86-64 (CISC)	ARMv9-A (RISC)	MIPS32 (RISC)	RISC-V (RISC)
Encoding Style	Variable-length byte stream	Fixed-length word	Fixed-length word	Fixed (+ C-ext 16-bit)
Instruction Size	1 to 15 bytes	32 bits strictly	32 bits strictly	32 bits / 16 bits
Encoding Complexity	Extremely High	Low to Moderate	Very Low	Very Low
Instruction Formats	Dozens, highly variable	Structured, fixed	3 geometric: R, I, J	6 geometric: R, I, S, B, U, J
Reg. Field Position	Variable across opcodes	Structured	Structured	Absolutely constant across all formats
Compaction Extension	Not applicable	Not applicable	MIPS16e ASE	C-Extension

by the caller, and the saved registers are x9 and x18-x27 which are stored by the callee.

The defining architectural contrast among these ISAs is the memory access model. x86-64 is a *register-memory* architecture: arithmetic instructions can operate directly on memory addresses. The three RISC architectures enforce a strict *load/store* model: memory is accessible only through dedicated load and store instructions, cleanly decoupling the address generation unit from the ALU. Table II summarizes these register configurations.

C. Addressing Modes and Hardware Indexing

x86-64 has many addressing modes available for use. Its general memory operand form `imm(rbase, rindex, scale)` specifies a base register, an index register scaled by a multiplicative factor (1, 2, 4, or 8), and a constant displacement [8]. The `LEA` (Load Effective Address) instruction calculates this scaled-index expression without accessing memory, acting as a rapid multi-operand arithmetic shortcut.

```
; Standard Memory Access: Load from address rbx + (rcx*4) + 8
mov  eax, [rbx + rcx*4 + 8]

; LEA Instruction: compute rbx + (rcx*4) without memory access
; Highly utilized by GCC/Clang for fast address mathematics and array indexing
lea  rax, [rbx + rcx*4]
```

Listing 1. x86-64: Scaled-index addressing and LEA instruction explicitly bypassing memory reads

ARMv9-A supports 64-bit virtual addresses translated by an MMU. Its load/store addressing modes require a 64-bit base address stored in a general-purpose register or SP, with an optional 12-bit scaled unsigned immediate offset or 9-bit unscaled signed offset. It also supports base plus a 64-bit register offset or 32-bit extended register offset, PC-relative literal loads, and pre/post-indexing with automatic writeback [3].

MIPS32 and **RISC-V** restrict addressing strictly to a base register plus a constant displacement. Hardware auto-increment and indexed (base+register) memory accesses are intentionally omitted. This omission simplifies the address generation hardware—requiring no multiplier, scale selector, or writeback data paths—at the expense of requiring additional ALU instructions generated by the compiler to compute array indices [1], [6].

D. Memory Access and Data Transfer Operations

x86-64 uses `MOV` for data transfer between registers, memory, and immediates, and `MOVZX`/`MOVSX` if zero/sign extensions are needed. `PUSH` and `POP` implicitly decrement and increment the stack pointer [2]. **ARMv9-A** employs a structured load/store taxonomy: `LDR` loads a 32-bit word or 64-bit doubleword, `LDRB` loads a byte, `LDRSB` loads a signed byte, `LDRH` for halfwords, and `LDRSH` for signed halfwords. It also incorporates unscaled offset variants like `LDUR`, `LDURB`, and `STUR` [3]. **MIPS32** limits access to `lw`, `lh`, `lb` and their store counterparts (`sw`, `sh`, `sb`). **RISC-V** follows an identical load/store taxonomy (`lw`, `ld`, `lh`, `lb`).

The structural divergence is evident when executing a basic Load-Add-Store sequence. As illustrated below, x86-64 leverages its CISC nature to compress the operation, while RISC models expose the underlying operations:

```
; --- x86-64 (Register-Memory: Fused Operation) ---
; Fuses memory read, ALU addition, and memory write into a single instruction
add  [rbx], rcx          ; Mem[rbx] = Mem[rbx] + rcx

; --- MIPS32 (Load/Store Model: Three explicit instructions) ---
lw    $t0, 0($s1)        ; I-Type: load word via displacement
daddu $t1, $t0, $s2      ; R-Type: register-register add
sw    $t1, 0($s1)        ; I-Type: store result back to memory

; --- RISC-V RV32I (Load/Store Model) ---
lw   t0, 0(s1)           ; Load word via displacement
add  t1, t0, s2          ; Register-register add
sw   t1, 0(s1)           ; Store result back to memory
```

Listing 2. RISC vs. CISC Memory Operation Execution illustrating the code density of Fused Operations

TABLE II
COMPREHENSIVE COMPARISON OF REGISTER ORGANIZATION AND MEMORY MODELS

Architectural Feature	x86-64	ARMv9-A	MIPS32	RISC-V
GP Registers	16 × 64-bit	31 × 64-bit	32 × 32-bit	32 × 32/64-bit
Hardwired Zero Reg.	Not present (Uses XOR)	XZR / WZR	$zero (R0)	x0
FP / SIMD Registers	32 ZMM (512-bit)	32 × 128-bit (SIMD/FP)	32 FPR (32/64-bit)	Via F, D, V Extensions
Sub-register Access	8, 16, 32, 64-bit via aliasing	32-bit (Wn), 64-bit (Xn)	Not supported	Not supported
Memory Model	Register-Memory	Load/Store	Load/Store	Load/Store

E. Arithmetic, Logical, and Shift Execution

x86-64 covers all integer basics (`ADD`, `SUB`, `ADC`, `SBB`, `INC`, `DEC`, `NEG`, `CMP`, `IMUL`, `MUL`, `IDIV`, `DIV`). Logical operations (`AND`, `OR`, `XOR`, `NOT`) accept register, memory, or immediate operands. The `TEST` instruction performs a bitwise AND but only updates condition flags without storing the result, commonly used to check if a register is zero. Shift instructions (`SHL`, `SHR`, `SAR`, `ROL`, `ROR`) specify shift amounts via an immediate or the `CL` sub-register [2].

ARMv9-A arithmetic operates on W (32-bit) and X (64-bit) registers. Logical instructions include `AND`, `EON` (Exclusive OR with the second operand inverted), `EOR`, `ORR`, `ORN`, and `MVN` (Inverse). Shift instructions include `ASR` (arithmetic shift right), `ASRV` (where the shift amount is held in a register), `LSL`, and `LSR` [3]. **MIPS32** handles integer arithmetic and logic via instructions such as `ADD`, `ADDU`, `SUB`, `AND`, `OR`, `XOR`, and `NOR`. Shift operations (`SLL`, `SRL`, `SRA`) define the shift amount using the 5-bit `shamt` field within the R-Type format or dynamically via a register [1]. **RISC-V** minimizes its catalog by omitting redundant instructions, for instance, the `NOT` operation is executed programmatically via `XORI` with an immediate value of -1 [6].

F. Control Flow, Branching, and Condition Testing

Table III outlines control flow capabilities. **x86-64** and **ARMv9-A** execute conditional branches by testing a central **condition flags register** updated by previous ALU operations. In x86-64, `CMP rax, rbx` subtracts the values and sets the Zero Flag (ZF), Sign Flag (SF), Carry Flag (CF), and Overflow Flag (OF). A subsequent branch (`JG`, `JL`, `JE`, `JNE`) tests these flags. ARM uses similar logic via the `B.cond` family (`B.EQ`, `B.NE`, `B.GE`, `B.LT`) [2], [3]. However, this shared architectural state creates Write-After-Write (WAW) and Write-After-Read (WAR) hazards that complex out-of-order execution engines must resolve through extensive flag renaming hardware. Furthermore, x86-64 provides structural looping instructions like `LOOP` (which decrements RCX and branches) and `ENTER`/`LEAVE` to setup and teardown function stack frames.

RISC-V deliberately eliminates the condition flags register entirely. Branch instructions (`BEQ`, `BNE`, `BLT`, `BGE`, `BLTU`, `BGEU`) directly compare two source registers specified within the instruction itself, bypassing flag hazards entirely. `JAL` (Jump and Link) stores PC+4 in a destination register and jumps PC-relative within ±1MB, while `JALR` uses a base register plus a 12-bit immediate for function returns [5]. **MIPS32** provides `BEQ`/`BNE` for equality, but for magnitude comparisons it utilizes compare-and-set instructions (`SLT`, `SLTI`, `SLTU`) to write a boolean result (0 or 1) into a general register, which is subsequently tested by `BEQ`/`BNE` [1]. Unconditional jumps use the J-Type format utilizing a 26-bit address field.

G. Memory Consistency Models and Concurrency

An advanced differentiator that significantly impacts multi-threaded software and hardware design is the underlying Memory Consistency Model. **x86-64** strictly adheres to the **x86-TSO** (Total Store Order) model [11]. TSO enforces a relatively strong memory ordering guarantee: while loads can be reordered past previous independent stores (facilitated by hardware store buffers), all other memory operations (Load-Load, Store-Store, Load-Store) appear to execute sequentially to all cores. This strong guarantee greatly simplifies concurrent programming, as software developers and compiler writers can rely on natural ordering without excessive use of memory barrier instructions [11]. However, it places an immense burden on the hardware, requiring complex snooping protocols, associative store buffers, and aggressive speculation mechanisms to maintain performance without violating TSO constraints.

Conversely, **ARMv9-A** and **RISC-V** adopt **Weakly Ordered Memory Models** [3], [5]. In these models, the hardware is granted maximum flexibility to reorder memory operations (loads and stores) independently to optimize pipeline utilization and mask memory latency. Unless explicit memory barrier instructions (such as `DMB`, `DSB`, or `ISB` in ARM, and `FENCE` in RISC-V) are inserted by the compiler, multi-threaded applications may observe memory updates out of order. While this pushes the complexity burden onto the compiler back-end and software developers, it drastically simplifies the hardware memory controller and coherence interconnect, saving significant die area and power. **MIPS32** similarly operates under a weak memory model, requiring explicit `SYNC` instructions to serialize data transfers across multiprocessor boundaries [1].

TABLE III
COMPREHENSIVE COMPARISON OF CONTROL FLOW AND BRANCH MECHANISMS

Architectural Feature	x86-64	ARMv9-A (A64)	MIPS32	RISC-V
Conditional Branch	Via flags (JE, JNE, JG, JL...)	Via flags (B.EQ, B.NE, B.GE...)	BEQ, BNE + SLT/SLTI	Direct compare (BEQ, BNE, BLT)
Unconditional Jump	JMP	B (PC-rel, $\pm$128MB)	J (26-bit target)	JAL (PC-rel, $\pm$1MB)
Procedure Call	CALL	BL (stores PC+4 in X30)	JAL ($ra = PC+4)	JAL (rd = PC+4)
Register Jump/Return	JMP reg, RET	BR Xn, BLR Xn, RET Xn	JR $ra	JALR x0, ra, 0
Condition Source	Central Flags Reg.	Central Flags Reg.	General Register	Inside Branch Inst.

IV. ARCHITECTURAL ANALYSIS AND ENGINEERING TRADE-OFFS

A. Decoder Complexity and Micro-Operation Translation

A fundamental architectural trade-off exists between instruction expressiveness and hardware decoder simplicity. The x86-64 decoder is extremely complex. Because instruction lengths range from 1 to 15 bytes, boundaries cannot be predicted. Modern implementations employ a strict two-level strategy: the front-end parses the byte stream into macro-instructions, then translates them into fixed-width micro-operations (μops) that behave like RISC instructions internally [12]. These μops are cached in a Decoded Instruction Cache (often called a Loop Stream Detector) to bypass the expensive decode penalty on execution loops. The μop-to-macro ratio in GCC-generated x86 code is typically below 1.3, indicating that compilers heavily favor simple instructions, rendering the complex decode hardware largely redundant for everyday execution [13]. Deep out-of-order (OOO) pipelines, pattern history tables, and massive branch target buffers are mandatory to maintain throughput. A mispredicted branch forces a pipeline flush, which costs the processor 10 to 20 wasted cycles [12].

ARMv9-A leverages its 32-bit alignment to bypass the sequential boundary-scanning bottleneck entirely. For example, the Cortex-A710 microarchitecture utilizes a streamlined two-stage front-end: instructions are fetched and decoded into internal macro-operations (MOps), register-renamed, and later split into micro-operations (UOps) for dispatch across **thirteen distinct issue pipelines** [14]. This structural uniformity allows higher sustained fetch bandwidth per clock cycle.

MIPS32 possesses the simplest decoder. Advanced cores such as the **74K family** push this simple baseline to extreme pipeline depths to maximize clock frequency, featuring a **17-stage Address Generation (AGEN) pipeline** and a **16-stage ALU pipeline**. The pipelines stages are equipped with advanced out-of-order dispatch engines and 256-entry bimodal branch predictors to hide instruction latency and memory bottlenecks [4].

RISC-V's constant-bit-position register fields allow speculative register file reads during the decode phase, shortening the critical path. The absence of flags, branch delay slots, and complex addressing modes dramatically simplifies stall-detection, forwarding logic, and pipeline management for both in-order and superscalar implementations [5].

B. Code Density and Compiler Optimization Limitations

Empirical studies reveal x86-64's average instruction length in real execution environments is 3.1–3.4 bytes [15], shorter statically than fixed 4-byte RISC words. However, the RISC-V C-extension (handling roughly 60% of dynamic instructions) and the MIPS16e ASE deliver code density that is highly competitive with, and often superior to, x86-64 and ARM Thumb-2 for integer workloads [1], [6].

From a compiler perspective, the **RISC-V** compiler, while benefiting from an orthogonal 32-register file, must execute heavier instruction scheduling to compensate for the lack of complex addressing modes and auto-increment operations, requiring explicit address arithmetic instructions to be injected into the instruction stream [6].

C. SIMD, Vectorization, and Data-Level Parallelism

Processing performance depends heavily on how effectively an architecture supports data-level parallelism. **x86-64** provides the heavily layered **AVX-512** instruction set, which utilizes expansive 512-bit ZMM registers [2]. Even though the AVX-512 provides massive throughput for scientific workloads, the complexity of decoding these instructions, along with the dynamic power drawn by the 512-bit wide ALUs, frequently forces x86 processors to throttle their clock speeds to remain within thermal design power (TDP) constraints [12].

ARMv9-A presents an exceptionally robust alternative via the Scalable Vector Extension 2 (SVE2). It allows for scalable vector processing, which gives the ability for the compiler to vectorize workloads and execute operations on multiple pieces of data simultaneously. It accelerates complex mathematical and multimedia workloads [3], [16]. The ISA does not mandate a specific register width. Instead, code is compiled once and the hardware dynamically determines execution width (ranging from 128 bits up to 2048 bits depending on the specific chip implementation). This provides binary portability across low-power edge devices and massive cloud servers [17]. **RISC-V** adopts a mathematically similar approach via its **V-Extension**, which is also vector-length agnostic, allowing highly scalable designs tailored for AI workloads [6]. **MIPS32** addresses data parallelism in a much tighter footprint via the **DSP ASE Rev2** extension. Rather than massive vector registers, it provides SIMD execution on 8-bit and 16-bit fractional data, utilizing multiple 64-bit accumulators to double Multiply-Accumulate

(MAC) execution speeds at a negligible silicon cost of 2.7% of the total die area [4].

D. Energy Efficiency, Leakage, and Power Dynamics

x86-64 processors suffer from a high static power floor imposed by the massive CISC decode infrastructure, physical register renaming arrays, and large reorder buffers (ROB). While modern chips employ Dynamic Voltage and Frequency Scaling (DVFS) and hierarchical C-states to clock-gate execution units, the inherent static leakage logic renders them structurally unsuitable for strictly battery-operated environments [13].

ARMv9-A minimizes power via its RISC load/store foundation and mechanisms like the Wait-For-Event (WFE), which allows an entry to a low-power state, until an event occurs. It is a mechanism used to improve energy efficiency during periods when the processor is waiting for work to become available [3]. **MIPS32** demonstrates extreme efficiency in embedded RTOS systems: the multi-pipeline nMPRA4 model consumes merely **0.432 W at 33 MHz**. It achieves this by multiplexing pipeline registers for concurrent task contexts rather than duplicating entire hardware cores which saves massive logic resources and dynamic power [18]. **RISC-V**'s modularity allows the physical excision of unused modules (e.g., FPU or Vector units) directly from the silicon die, driving leakage current to absolute minimums for IoT endpoints [5].

V. Application Suitability

A. Desktop, Server, and High-Performance Computing

x86-64 is deep in the desktop computing market, cloud servers, and HPCs due to strong single-thread performance, massive multi-core scaling, VT-x/AMD-V hardware virtualization, and the x86-TSO memory consistency model. The x86-TSO model greatly simplifies concurrent multi-threaded programming by natively forbidding intra-core store reordering, making large-scale database synchronization more predictable [11]. **ARMv9-A** is rapidly infiltrating the server space (e.g., Azure Cobalt 100) and **HPC** (Fugaku supercomputer, NVIDIA Grace), driven strictly by the necessity for high compute-per-watt energy proportionality in thermal-constrained datacenters [9].

B. Mobile and Embedded Networking

ARMv9-A controls the **mobile and tablet** domains. Its power-efficient pipelines coupled with heavy SIMD throughput (NEON/SVE2) handle complex image signal processing without requiring discrete graphics processors [17]. **MIPS32** remains the architecture of choice for **networking infrastructure** (routers/switches) and **real-time DSP**. Its DSP ASE Rev2 extension doubles Multiply-Accumulate (MAC) execution speeds for audio/video decoding at a negligible silicon cost of 2.7% of the total die area. Furthermore, the implementable nHSE hardware scheduler engines unify task and interrupt priority spaces, minimizing scheduling jitter to levels required by high-precision control systems [4], [18].

C. IoT, Custom Silicon, and AI Accelerators

RISC-V is the definitive architecture for **custom AI accelerators and deep IoT**. Its royalty-free standard allows designers to inject domain-specific instructions (e.g., custom tensor matrix operations) natively into the ISA without licensing barriers. The minimal RV32E base core delivers ultra-low-power footprints impossible to achieve under the monolithic catalogs of x86-64 or ARM [5].

VI. Conclusion

This comparative analysis demonstrates that no single ISA is universally superior—architectural efficacy is strictly domain-dependent. The **x86-64** architecture offers the principal strength of raw single-thread performance and complete backward compatibility, which makes it the preferred choice for legacy desktop environments and heavy enterprise cloud servers. However, its primary limitation is the immense decoder complexity and high power consumption inherent to its CISC design, rendering it fundamentally unsuitable for battery-operated devices.

ARMv9-A achieves the optimal balance between high processing throughput and energy efficiency. Its major strength lies in its sophisticated RISC pipeline and advanced power management, with the main market share being in mobile computing and power-conscious datacenters. Its primary limitation remains its proprietary licensing model, which restricts custom silicon modifications.

MIPS32 provides absolute structural simplicity and highly deterministic execution. Even though it has limited scalability and an aging ecosystem, due to its minimal silicon footprint and predictable pipelines, specific embedded systems, networking routers, and real-time DSP applications use the architecture.

Finally, **RISC-V** introduces a paradigm shift through its open-standard, royalty-free modularity. Its primary limitation is the heavy reliance on compiler optimization to compensate for the lack of complex addressing modes. However, next generation custom silicon, secure IoT devices, and specialized hardware accelerators for AI depend on the architecture due to its clean datapath and extensibility.

The selection among these ISAs requires a calculated engineering trade-off between decoder simplicity, energy constraints, licensing budgets, and legacy software compatibility.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[2] Intel Corp., "Intel 64 and IA-32 Architectures Software Developer's Manual,".

[3] Arm Limited, "Arm Architecture Reference Manual (DDI0487)," .

[4] MIPS Technologies, "Architectural Strengths of the MIPS32 74K Core Family," 2008.

[5] A. Waterman, "Design of the RISC-V Instruction Set Architecture," Ph.D. dissertation, EECS Dept., UC Berkeley, 2016.

[6] A. Waterman *et al.*, "The RISC-V Instruction Set Manual, Vol I," Tech. Rep. UCB/EECS-2014-54, UC Berkeley, 2014.

[7] A. Tanenbaum and T. Austin, "ARM Architecture and RISC Applications,".

[8] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed. Pearson, 2015.

[9] M. Hossain *et al.*, "Redefining Computing: Rise of ARM," *World J. Adv. Res. Rev.*, vol. 21, no. 1, 2024.

[10] Microsoft Corp., "x64 calling convention," Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170

[11] P. Sewell *et al.*, "x86-TSO," *Commun. ACM*, vol. 53, no. 7, 2010.

[12] A. Fog, "The Microarchitecture of Intel, AMD and VIA CPUs," 2024.

[13] E. Blem *et al.*, "Power Struggles: Revisiting RISC vs. CISC," in *Proc. IEEE HPCA*, 2013.

[14] Arm Limited, "Arm Cortex-A710 Core Technical Reference Manual,".

[15] V. M. Weaver and S. A. McKee, "Code Density Concerns," in *Proc. IEEE ICCD*, 2009.

[16] MDPI. [Online]. Available: https://www.mdpi.com/2674-0729/4/4/44

[17] Authors, "ARM Architecture: From Mobile to HPC," *Chips*, vol. 4, 2025.

[18] Authors, "Hardware RTOS: Custom Scheduler Implementation," in *Proc. Int. Conf. Embedded Systems*, 2022.