# RISC vs. CISC at 46

## What was it about, and does it matter today?

Edition 1.05 - 25/7/26

# Contents



Front page: https://blachford.info/computer/RISC_VS_CISC_at_46/index.html

PDF Version: https://blachford.info/computer/RISC_VS_CISC_at_46/RISC-vs-CISC_at_46.pdf

Note: This article is mainly intended for those with an interest in microprocessors and uses standard industry terms. A glossary is included to explain some of these if you are unfamiliar with them.

# Part 1: What is RISC?

## Why Take a RISC?

*Reduced Instruction Set Computer* (RISC) was a processor design philosophy that started in the late 1970s in opposition to the increasingly complex designs of the day [MAIN].

The paper that coined the term RISC was published way back in 1980 [RISC].
In the following years, RISC processors would come to dominate the workstation, server and embedded markets. Today it dominates the entire CPU industry, apart from the desktop and servers, but it's making inroads there.

What people think RISC was about seems somewhat divorced from its origins. There is talk of RISC principles, pure RISC, everything must be done in a single cycle etc. I've even heard the idea that modern RISC processors are actually now *Complex Instruction Set Computer* (CISC)s because they have many more instructions these days. These are largely references to implementation details [RISC-I], or a misconception of what RISC originally was.

The name doesn't help, Reduced Instruction Set Computer sounds like the goal was just about removing instructions. In reality the acronym was devised by David Patterson and Carlo Sequin on the way to a funding meeting because it sounded like "*high risk*" [RETRO].

RISC really started with three tenets:

- **1: Remove complex microcoded instructions**
- **2: Use internal registers for operations**
- **3: Remove surplus instructions**

The original paper argued that these three changes would make for smaller, simpler, and faster processors. The concept did involve reducing the number of operations a processor had, but it was really about making processors smaller and faster by removing complexity.

In addition to the above tenets, I would add a fourth. This was there from the beginning, but I don't think its importance would become obvious until implementations began to appear:

- **4: Simplify decode**

## Tenet 1: Remove Microcoded Instructions

Microcode dates back to the beginning of computers when they were large, the hardware rudimentary, and memory was very small and expensive.

Microcode was software routines inside the processor that implemented instructions, essentially mini programs complete with conditional jumps and subroutines, that ran in the processor [MCODE]. This did not operate in the way you might think however. Instructions were not decoded then executed. Microcode instructions directly controlled the hardware, with bit patterns in different parts of the microcode instructions controlling different parts of the processor.

Microcode enabled more complex operations to be added to the instruction set relatively easily, increasing capabilities while saving memory. For example, some processors didn't have hardware multipliers but you can implement a multiply in software using multiple adds. If you were to implement that algorithm in microcode you could then include a multiply instruction in the processor's instruction set.

Another use of microcode was to enable compatibility across different machines. To reuse the previous example, high end machines might have a hardware multiplier, and the instructions to use it. Lower end machines without the hardware might use microcode. This meant you could use the same software on both machines, despite the hardware differences.

Assembly code programming was much more common in the 1970s and before, so instruction sets were made to be programmer-friendly [68K]. Instructions were made to be "orthogonal", that is, any operation can use any instruction format. This meant there were lots of different ways of doing the same operation, and each of these were represented by individual instructions. Processors were also adding high level instructions. Functions that you might find in a software library were implemented as microcode inside the processor. Some machines went all out and included things like the OS and compiler in hardware [HLLCA].

Microcode makes a lot of sense for ease of programming and code density, but as more and more instructions were added, it made the processors ever-more complex. When memory was slow, it was initially a way to speed up processors, but as memory got cheaper and faster, and processor complexity increased, this turned on its head.

The complex instructions added an unavoidable overhead, they had to be decoded and identified, then the microcode engine would have to be initiated and then the microcode program would then be executed. All of these operations were sequential, there was no way to do other operations at the same time. Even basic pipelining was difficult.

On top of this, there was additional complexity when memory management operations were involved or interrupts occurred. What do you do if an interrupt comes in in the middle of executing a microcode program? This required intermediate processor state to be saved and restored, adding another level of complexity to already complex processors.

All this complexity meant that processors took longer and longer to design and debug. When the machines did ship, they would still have bugs. The bugs became so bad that processor vendors took to including special memories in the processor just to hold microcode patches. Bizarrely, this gave us the floppy disc. IBM invented it as a way to distribute microcode patches [RETRO].

The delays pushed out development times from months to years. The original RISC paper notes that both the Z8000 and 68000 microprocessors were 1-2 years late because of their microcode complexity.

High level instructions were added to make things easier and faster for programmers, but it's not clear if this always worked. When the VAX's INDEX instruction was replaced by a set of simpler instructions, it ran 45 percent faster [RISC].

Microcode is an example of a powerful technology that was a good solution to the problems of its time. However, by the late 1970s, computers were far more sophisticated and memory was becoming faster and cheaper. Compiled languages were also becoming more common and they tended to avoid complex instructions. The technical environment had changed and thus the problems that would present, and the solutions to them, had changed.

The original RISC paper argued removing microcode would make processors simpler, easier to develop, and faster.

## Tenet 2: Use Internal Registers for Operations

One of the tenets of RISC is to do operations using internal registers in place of using memory directly. Some CISC instructions let you specify the memory address of data alongside the operation in instructions.

RISC processors on the other hand use what is known as a *load-store* architecture.
The data is explicitly loaded into the processor registers before any operations are performed on it. This means more instructions, but it's faster. In fact, experiments on programming existing systems found that they could get speed-ups just by using the registers more [801]. Further experiments showed that when a compiler was designed to use registers more with simpler instructions, it would be significantly faster, even on existing hardware.

## Tenet 3: Remove Surplus Instructions

The third tenet of RISC was to remove under-used instructions.

As more and more higher level languages were being used, it was found that all the nice orthogonal different versions of instructions added for the benefit of assembly programmers, were not being used.

On top of that, the complex instructions that went to microcode were also not used by the compilers.

Compilers tended to use the fastest instructions they could use, preferably those that would be fastest across as many versions of the processor as possible. This meant compilers tended to use only a small subset of the instructions available in the instruction set. Many of the more complex instructions were not used at all.

The consequence of this was you could simply remove the majority of the processor instructions and no one would notice them missing. Well, maybe apart from the odd irate assembly programmer.

The RISC processors did exactly this and that is partly where the name comes from. However, just reducing the number of instructions appears to be one of the least important aspects of the RISC philosophy. It was much more about removing complexity and reversing the direction away from adding high level language features.

**What is an Instruction Anyway?**
I would like to clarify a point here, since this seems to cause some confusion.
I'll define the terms as I use them:

> **Operations**
> The basic hardware functions a processor performs. For example: A 32 bit add.
>
> **Instructions**
> The commands you use to initiate operations. These are what the *Instruction Set Architecture* (ISA) defines.

There may be a single 32 bit add operation, but many different instructions that use that add.
Tenet 3 essentially says: "Why have 8 different add instructions when the compiler only uses 2?"
This is what is meant by "reduced instruction set".

Note that RISC was *never* about reducing the number of hardware operations.

## Tenet 4: Simplify Decode

When RISC processors moved into real development from theory, the ideas and end results shifted somewhat. Beyond removing microcode, surplus instructions and using registers, it soon became about removing complexity in general, especially in the instruction set. Decode in particular was targeted, though this was not mentioned in the original RISC paper.

The developers of the MIPS-X research processor took up a maxim to cover this:

"*The goal of any instruction format should be:*

*1. Simple decode,*
*2. Simple decode,*
*and 3. Simple decode.*

*Any attempts at improved code density at the expense of CPU performance should be ridiculed at every opportunity."*

While this may seem a touch extreme, it worked. The results of the MIPS-X processor resulted in 25% larger binaries, but delivered enormous performance gains [MIPS-X]. These results were also born out in real commercial designs, the DEC Alpha had larger binaries even compared to other RISC designs, but it was the performance leader for much of the time it was in production.

### What Is CISC?

This may seem an odd question to ask but one thing that can be slightly confusing about the RISC vs. CISC debate is that CISC was not given a definition at the time. RISC was a reaction to increasing complexity of designs, The RISC-I paper [RISC-I] points to 3 systems: DEC VAX, Intel iAPX-432 and IBM System 38. Notably, the Intel 8086 was listed as one of the simpler designs!

In a retrospective of the RISC-I paper, the authors state:

> "*I think the trade press concluded that any commercial computer that wasn't a RISC must be a CISC, and hence the confusion*" [RETRO].

**Computer Architecture: A Quantitative Approach** defines CISC as:

> *A Complex Instruction Set Computer (CISC) is a computer architecture in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions* [CISC].

John Mashey argued that new architectures from 1986 onwards shared a set of characteristics that previous designs did not. He gave more detail on the concrete differences between RISC and CISC processors [Mashey].

## RISC in Theory and Practice

While CISC wasn't really defined, It's also true to say RISC never had a tight definition either. The tech press would eventually invent their own definition, correct or not and this led to things like single-cycle instructions being part of the definition, or even the idea of *pure* RISC.

In reality, RISC is a design philosophy, an approach to processor design. Single-cycle instructions were part one or two implementations but never required. Without a concrete definition, there is not really such a thing as a *pure* RISC, though perhaps it could be a processor that follows the RISC philosophy very closely.

# Part 2: How the RISC Concept Came About

## Building 801: Where RISC began

RISC was the result of multiple developments, the biggest being IBM's experimental 801 processor(s) led by John Cocke [801].

The 801 initially started in 1974 as a project to build a telephone switch that could handle 300 calls per second. This required 12 mips *Million Instructions Per Second* (MIPS) when the fastest IBM computer at the time could only do 3.5 mips.

The project looked at what was required and found it didn't really require that many instructions, so they designed a very stripped back processor for the task. The telephone switch project was terminated the following year, but the design was advanced enough that it was expected to perform very well in a number of areas. Consequently, it was kept going with the aim of producing a more general purpose design.

Using a simple design and only concentrating on a limited set of instructions, made for a relatively simple but very fast processor, with many operations able to be completed in just 1 clock cycle.

Research showed that the majority of code typically used a fraction of an instruction set and most of this (> 50%) was based on just 5 instructions. This meant that this simple machine was capable of doing what the much more complex machines could do. You just had to code it yourself instead of relying on microcode. A compiler could generate the same sequence of operations as instructions and on this simpler, faster design. It would then run faster than the complex machines with microcode.

John Cocke wrote:

> "*Imposing microcode between a computer and its users imposes an expensive overhead in performing the most frequently executed instructions.*" [801]

The 801 was a pure load-store machine with 16 registers, it had pipelining and a split instruction / data cache. There was no microcode and instructions used a single fixed 3 byte format.

Some lessons were learned from the first machine so a second version was designed. One finding was analysis of the code and performance showed the compiler was spilling registers to memory so the second version increased the number to 32 registers and a fixed 4 byte instruction size. This set a pattern that would be used by many subsequent commercial RISCs. Even the recent *Advanced Performance Extensions* (APX) proposal for x86 includes 32 registers [APX].

The 801 processor was extremely fast for its day, some of this is due to its load-store nature. Bizarrely, in some cases, code running on a 801 simulator running on an IBM 370/168 would run faster than native code on the same machine! This sounds incomprehensible, but if the simulator was mapping directly to the faster, simpler instructions in the 370, and the existing native compilers used the slower, complex instructions, it does make some sense.

The 801 compiler was then ported to the 370 but only used simple instructions and kept the load-store method of working. The resulting code ran up to 3 times faster than the existing 370 compilers. So, it turns you can get substantial performance benefits just by treating a CISC machine like a RISC machine.

## The Original RISC Paper

The 801 was an internal project at IBM with little published about it initially, but other projects were coming to similar conclusions. Some of these led to the original RISC paper.

### David Patterson vs. VAX

After his PhD, David Patterson spent some time working on DEC's highly complex microcode. He proposed building a single chip VAX computer but it would require a way to patch the often buggy microcode. His paper for this was rejected on the ground that such a store would be a waste of silicon, he appeared to agree so he began looking at simpler instruction sets without microcode [RETRO].

### David Ditzel vs. SYMBOL

Another inspiration for the original RISC paper was David Ditzel's work with the SYMBOL computer. This was an example of a *High Level Language Computer Architecture* that used hardware to replace software. It implemented the OS, compiler, and even a text editor in 20,000 hardware chips [HHLCA].

The conclusions of working with this machine were multifold, it was slow, buggy (it took several *years* to debug) and difficult to optimise for. A conference presentation by Ditzel sums it up: "*Fascination studying SYMBOL (Ditzel/Patterson) eventually led both to religious conversion to RISC*" [Ditzel]

This ultimately led to the 1980 paper "The Case for the Reduced Instruction Set Computer" by David Patterson and David Ditzel [RISC] that introduced the RISC and CISC concepts to the world and gave us the acronyms.

## Other RISCy Developments

Others beyond the authors of the original RISC paper also came to the conclusion that simpler machine were better:

### Bell Labs Iterative Processor Development

Stephen Johnson at Bell labs did a project to to optimise a processor by iterating over a process of proposing designs then building a compiler and measuring the results. This also resulted in a relatively simple RISC-like design. [SCJ]

### Andrew Tanenbaum's EM-1

Andrew Tanenbaum (of Minix fame) proposed a simpler machine in 1978.
His research was designing a machine as a compiler target. This was based on analysing thousands of lines of code and designing a machine around it. This resulted in a simple instruction set that while variable-length, would have been relatively easy to decode. [EM-1]

### The Pre-RISC RISC

It turned out that the idea of simplification for performance was not new, some had already been done ...a long time ago. [6600]

The Seymour Cray designed CDC 6600 supercomputer used 2 sizes of instructions, a simple instruction set and no complex microcode way back in 1964. It was the fastest computer in the world until 1969 when the CDC 7600 came out. The backronym "Really Invented by Seymour Cray" is not without some merit!

Some consider the 6600 as non-RISC because it doesn't use a single length length of instructions, however I don't think this is quite true. The 6600 uses a fixed 60 bit packed format with some variations on the number of instructions within, so technically yes, they're more than one length but what's important is that decoding is predictable and simple. This fits perfectly with the RISC philosophy.

## Making a RISC (or two)

### RISC-I

There was a lot of resistance to the initial RISC proposals and the 801 wasn't public at this point so the RISC proponents couldn't point to that work as an example. Building a processor would act to publicly prove the concept.

After returning to Berkeley from DEC, David Patterson supervised research into a new processor design based on the RISC concept. This became known as the Berkeley RISC-I [RISC-I]. This was the first time a University had designed a processor so it was an ambitious project to say the least [RETRO].

The RISC-I was followed by RISC-2, and it would go on to serve as the inspiration for the Sun SPARC processor.

### Stanford MIPS

Not to be outdone, Stanford University also decided to design their own RISC chip known as the *Microprocessor without Interlocked Pipeline Stages* (MIPS)*.

The designers decided to directly commercialise this project and founded MIPS Computer Systems.

* That won't be confused with the benchmark, honest.

# Part 3: The Rise of RISC

## Little RISCy goes to market

RISC first appeared on the market in the form of IBM 801 derivatives for embedded use. However, it wasn't until 1986 when the ball really got rolling when the first RISC processors appeared from HP, MIPS, and Acorn (the first ARM development board).

There was some debate at the time about if RISC could outperform CISC because the higher number of instructions that RISC processors required. This debate was ended in 1987 where the first SPARC based Sun workstation appeared at 3 times faster than the previous CISC Motorola 68020 based model. The 68030 appeared the same year but the SPARC was still twice as fast.

At the time, Motorola pretty much owned the workstation market with the 68K line. The bigger workstation vendors jumped ship to the faster RISC designs pushing 68K down to the desktop and embedded market, though it did continue to see some use in workstations along with some Macs, Amigas and STs.

Motorola did their own RISC processor, the 88000 series, and this was chosen by both Apple and Steve Jobs' NeXT Computer [88K]. However, the initial 88100 was a very expensive multi-chip system and both companies elected to wait until the next generation 88110, a single chip version. This chip was late and by then IBM had approached Apple and Motorola to team up for PowerPC. Both Apple and NeXT would then jump to PowerPC, but NeXT abandoned hardware before their PowerPC machine shipped.

Aside from being fast, the RISC processors were cheap to make because they relatively simple to develop and physically small. To put it into perspective, The ARM 2 is competitive with the initial Intel 386s, yet the ARM had one ninth of the transistors and was only one third of the size, and that's with the 386 a process node ahead.

## If You Can't Beat Them, Join Them

From 1986 and onwards into the 90s RISC took over the workstation and server market. Intel got bigger and bigger on the desktop but didn't have a look-in on the workstation market. Intel's attempt at build their own answer to the RISC onslaught was a RISC/VILW processor called the i860. It was reportedly very fast, but rather complex to program. It was not a great success but it turned out to be surprisingly good for graphics and was used as an graphics accelerator by both NeXT and Silicon Graphics.

## The Ultimate RISC

In 1992 after a number of internal RISC projects, DEC, producer of the decidedly CISC VAX machines introduced the Alpha 21064 EV4 processor where EV either denotes the silicon process or is a reference to an experiment involving and electrically charged pickle! [EV]. The Alpha was an absolute monster of a processor coming in at almost 200MHz when most processors were not even at 100MHz. DEC would rapidly ramp its clock speed over the next few years leaving all the other vendors playing catch up.

## CISC gets RISCy

As development continued on the RISC side, CISC vendors did not stand still. They started introducing some of the concepts and features associated with RISC designs into their CISC processors. Adding these kinds of features to a CISC architecture was thought to be impossible, but that didn't stop smart engineers.

The 486 and 68040 both introduced pipelining, this provided large per-clock performance improvements over their predecessors.

The successors to the 486 and 040, the Pentium and 68060 used more RISC like techniques, and especially superscalar execution, to boost performance.

The 68060 in particular is quite literally a RISC processor with CISC fetch & pre-decode stage. It removed microcode entirely and puts instructions through a pre-decode stage that outputs fixed length instructions. To quote the 68060 User Manual: "*Fixed format instructions are dispatched to dual four-stage pipelined RISC operand execution engines*" [060UM].

Intel was similar but not quite so RISCy with the Pentium. Startup NexGen did something more like the 060 and created a processor with an x86 decoder and a RISC core. This was sold as the Nx586. NexGen was later bought by AMD.

## Side Note: and Speed Demons and Brainiacs

RISC was a single design philosophy but from early on, there were always some different approaches to building RISC processors. The SPARC (based on the Berkeley RISC 1/2) included a feature called *register windows* to speed up context switching. None of the other mainstream RISC designs used this technique.

One big difference resulted in the *Brainiac vs. Speed Demon* debate. The early IBM Power processors ran at a relatively low frequency but included Out-of-Order execution to achieve high performance.

Other vendors went with simpler designs and relied on a high clock frequency to achieve their performance. The flagship for this approach was the DEC Alpha which was the fastest processor on the market for quite some time. However, this debate didn't last too long, the third generation Alpha 21264 also want Out-of-Order in 1998.

This debate was also repeated on the desktop when Intel's Pentium 4 went with a very high clock. Intel eventually abandoned this approach when it became obvious that very high clocks require rather high power consumption. Graphs of the day pointed out that temperatures were scaling up so quickly that they'd eventually hit the temperature of the surface of the Sun. The following Core processors didn't require such high clocks for performance and were low power enough to find a home in laptops.

## A RISCy Game

RISC proceeded to take over the workstation and server markets and make inroads into the embedded market (which 68K still ruled). Games consoles started using RISC processors which could provide high performance at a low price:

- **Sony PlayStation 1 / 2 - MIPS**
- **Nintendo N64 - MIPS**
- **Nintendo Game Cube - PowerPC**
- **Nintendo Game Boy Advance - ARM**
- **Sega Saturn - 2 x SH-2**
- **Sega Dreamcast - SH-4**
- **Atari Jaguar - 2 x Custom RISC**

## ARM Gets Embedded

In the 1990s many companies were designing and building their own processors RISC or otherwise. One company however did things somewhat differently. UK desktop computer company Acorn developed the *Acorn RISC Machine* (ARM) processor for a successor to the BBC Micro computer that had been highly successful in the education market in the UK in the 80s.

The first ARM was designed to be low power to save cost on the packaging, but turned out to be even lower power that expected. This fact serves Arm well to this day and became a defining feature of the processors and indeed Arm the company.

Pressure from Acorn's competitors and a deal with Apple for a CPU for the Newton led to Arm being split off as an independent company (now called Advanced RISC Machine). Unlike most CPU vendors, Arm didn't build it's own processors. They designed the processors and then licensed the designs to other companies.
This led to Arm getting into some of the bigger companies with their own designs and replacing them from the inside. Those companies would then become Arm vendors. Arm vendors essentially became a salesforce for Arm.

Arm did reasonably well as a small company displacing 8 and 16 bit processors, but they hit a problem. The 8 and 16 bit processors often used smaller instructions that meant code could be very small, Arm only used fixed 32 bit instructions so compiled code was bigger. This meant anyone considering Arm might have to increase memory in their device, increasing product costs. This came up in a visit to Nintendo who found using Arm would have made game cartridges use more memory making them uneconomic [ARM-H].

Arm's then chief architect Dave Jaggar proposed a solution to this: Add a 16 bit instruction mode called Thumb. This idea initially met with a great deal of resistance within Arm as it was considered as going against RISC ideals. This however was for a good purpose and in this case code density was critically important. The proposal eventually won out and It was added without adding a great deal of complexity to instruction decoding, i.e. it was done in a RISC way [ARM-H].

The resulting ARM 7TDMI processor went on to be highly successful after the then No.1 mobile phone manufacturer Nokia started using it. The rest of the Mobile industry copied Nokia and Arm took off. Arm is in pretty much every mobile phone now, and almost everything else too!

# Part 4: Death of a CISC

Despite the 68K series taking on many RISC characteristics over time, the inherent complexity of implementing the CISC 68K instruction set remained. In the 1990s these ultimately made the Motorola 68K series processors uncompetitive against RISC competitors in workstation, desktop, and eventually embedded markets. This can be illustrated by comparing the final top end 68K, the 68060 to some of its contemporary processors, specifically a number of MIPS processors:

## 68060 vs. R8000

Both of these processors were released in 1994. They were not intended for the same market so are quite different, however there are some similarities and its the specific areas of complexity I'm comparing.

The Motorola 68060 is essentially a superscalar RISC processor with a CISC decoder. The pipeline is divided into 2 halves, the first 4 stages fetch and pre-decode 2 instructions per cycle. The second 4 stages then decode and execute the instructions.

The MIPS R8000 has a 5 stage pipeline and also does fetch and pre-decode, but this is done in a single pipeline stage. However, this stage fetches and decodes 4 instructions at a time.

The difference I want to illustrate here is the simplicity the R8000 has in getting its instructions. It reads 128 bits into an instruction buffer. It doesn't need to calculate any instruction lengths or positions, these are already known because the instructions are a fixed length.
Instructions also cannot cross cache lines or memory pages, so problems associated with those are non-existent.

The 68060 avoided cache and memory page issues by pulling multiple instructions into a relatively large buffer. It then has to partly decode an instruction to get its length so it can fetch instructions from the buffer. The 68K series has a myriad of instruction formats to deal with so there are potentially thousands of instructions of various different lengths. In the 68060 these are pre-decoded and converted into fixed length instructions for execution. The fetch / pre-decode unit occupies approximately 20% (40 $mm^{2)}$ of the entire 200 $mm^2$ chip area.

## 68060 vs. R4400

RISC chips didn't need anything nearly as big and complex as the 68060 fetch / pre-decode unit so they could either make a smaller (cheaper) chip or use the area for something else.

Another contemporary of the 68060 is the MIPS R4400. Its relatively simplicity enables a much higher clock rate (it scaled to 250MHz) and twice the cache, it gave twice the performance on the SPEC Int 92 benchmark at 150MHz. This wasn't even the fastest processor at the time, the Alpha 21064A at 275 MHz gave numbers twice as high as the R4400 on SPEC int 92 [SPEC].

## 68060 vs. R4200

Since the 68060 was ultimately positioned for the embedded market, a comparison with the MIPS R4200 is valid as it was also designed for embedded use at the same time.

The R4200 had similar performance to the 68060 but ran at lower power and sold for around $50, just 1/6 of the price of the 68060.

Within 2 years, DECs implementation of ARM, the StrongARM had appeared with double the performance and power usage under 0.5 Watts. It cost $29 in volume.

## Was the 68060 a Bad Design?

No, the 68060 was certainly a worthy successor to the 68040, and was competitive (per clock) with Intel's Pentium which was considerably more expensive. At the time I was using a 68030 accelerated Amiga, a 68060 would have been awesome!

The problem was, the inherent complexity in the 68K's CISC instruction set had a high cost in the size and complexity of the processor design. Without the same level of complexity, RISC chips could be clocked much faster. More importantly, because of their simplicity they could be much smaller and sell for a much lower price. CISC designs simply became uncompetitive.

CISC may have had an advantage for code density, but there's no point saving a little money on the memory system if the cost of the CPU is multiple times higher.

The 68060 did get some embedded design wins and was in production for several years, it was also briefly used in an Amiga where it provided a good performance boost over the 68040. Later revisions of the chip proved highly overclockable going well beyond the fastest 75MHz version produced [Lightroom], and are still sought after by classic Amiga enthusiasts to this day.

## Backwards Compatibility was not Important in 68K Markets

Binary compatibility wasn't important on workstations as recompilation was usual.
It also wasn't very important in the embedded market where processor requirements can differ wildly.Backwards compatibility could have been important as the 68K was present in Macs at the time, however Apple had already decided to go RISC [88K].

Commodore and Atari also both used 68Ks but neither used the 68060. Commodore were planning to go with a RISC processor in a follow up to the Amiga called *Hombre* but the company didn't survive long enough to make it [HOMBRE].

This lack of a compatibility requirement in the markets 68K made the 68K line a lot more vulnerable to the RISC advance. Had backwards compatibility been more important, the next generation probably would have likely taken the same direction as the Pentium Pro and would have regained some ground back against the RISC processors.

## Backwards Compatibility Saved x86

One area where binary compatibility is critical is the desktop, it's no coincidence this is the one area where a CISC instruction set lives on to this day.  Without binary compatibility, it was very difficult for any of the RISC competition to challenge the dominance of x86 in desktops.

The DEC Alpha did have a technology called FX32! that could emulate an x86, in some cases faster than the x86 processors of the time.  Had Alpha been pushed down into the desktop market this could have changed things, but DEC kept the Alpha in the high end market instead ...and went bankrupt.

## Market Differences Made 68K Vulnerable

As much as technical differences were important, it was the markets that the 68K line served that made its downfall inevitable.  For x86, it's the markets that those processors served that ultimately kept it going.

## 68K Went Really RISCy

The 68K line didn't end with the 68060. The 68K was even more RISC-ified into the Coldfire processors with pretty major changes to the instruction set enabling much higher clock frequencies. These would continue in active development for another decade.

# Part 5: Revenge of the CISC

## The Pentium Pro

While intel had been including RISC techniques in the 486 and Pentium, they went full in on the Pentium Pro. It was aimed straight at the at RISC competition and was briefly the fastest processor on the market for integer processing.

The Pentium Pro is essentially a CISC decoder with an *Out of Order* (OOO) RISC execution engine. The CISC instructions are broken into RISC-like micro-ops that go into the execution engine. That said, the x86 instruction set is complicated so it's not as RISCy as the RISC competition. For example, microcode is still present for some operations in x86 devices.

The Pentium Pro architecture would go on to be highly successful, its derivatives would be the mainstay of Intel processors for many years to come. When Intel tried to go all speed demon with the Pentium 4 and things got too hot, they returned to the Pentium Pro architecture derivatives and sold even more of them.

## An EPIC RISC: Itanium

In the 1990s Intel and HP started work on a new kind of processor based on a *Very Long Instruction Word* (VLIW). While RISC designs move work to the compiler, VLIW goes even further getting the compiler to also schedule code execution. This turns out to be a damn-near impossible task for general purpose code and despite billions being poured into it and support from many computer vendors, Itanium never really took off.

After Compaq bought DEC, they cancelled the future Alpha versions in favour of the Itanium. Itanium also briefly replaced HP's PA-RISC and MIPS, but Itanium's lacklustre performance soon brought them (and Alpha) back to life.

Itanium found some success in high-end servers, with its main user being co-developer HP. It did stay in production for the best part of 20 years and went through quite a few revisions over its lifetime. The last versions were released in 2017.

## RISC Won the Microarchitecture War

By the Mid 1990s, the old school CISC processors that RISC was developed in answer to were dead. Many of the ideas that powered RISC designs had been adopted by the CISC vendors leaving no one developing the old school type CISC designs, at least in the desktop/workstation market.

Intel became so big no one could seriously compete against them. The AIM (Apple IBM Motorola) alliance tried with PowerPC, but once Apple shut down the clone market, the numbers weren't there to continue development. Within a decade, even Apple had moved to x86.

The CISC x86 instruction set remained of course, but even that would change.

## CISCy gets even more RISCy: x86-64

While Intel decided to go left-field with the Itanium, AMD stuck firm with x86 and decided to enhance it. The AMD64 (aka x86-64) specification added 64 bit capabilities along with more registers, moving x86 more towards a load-store style architecture. In addition to the changes in previous generations, this was another tenet of RISC being adopted within a CISC design.

## x86-64 Takes Over

As x86-64 became more powerful, and sold more and more, they gradually took over the workstation and low end server spaces, pushing RISC designs up into higher end servers.

RISC designs were increasingly expensive to develop and as the performance differences shrank, many vendors eventually switched to x86-64. This killed off the Alpha, PA-RISC and MIPS processors, though MIPS would continue in the embedded market for many more years.

The performance difference between CISC and RISC began to close when CISC started using the same techniques as RISC. CISC eventually won out over the RISC workstation processors by essentially becoming RISC themselves and then beating them economically with sheer numbers from the much larger desktop PC market.

## Arm Took Embedded

While x86-64 took over the workstation and server market, in the embedded market, many different kinds of processors from CISC to RISC continued to flourish. With an er, army of low-cost low-power designs, seemingly everyone licensed Arm based designs, Arm took over a major chunk of the market. To date it is estimated that 350 Billion Arm processors have been made, over half of all processors ever made.

## Other RISCs Kept Going

In 2005 Sun benefited from a rebound with the UltraSPARC T1 "Niagara" processor. While other vendors were starting to do designs with dual core each with dual threads, the T1 had 8 cores with 4 threads each. The result was a server processor that, for web serving, blew everything else out of the water. Others soon caught up and IBM's POWER processors have had many heavily-threaded cores for several generations now. There had been similar thoughts going around the industry for some time. Some years before the UltraSPARC T1, Compaq's DEC / Alpha group had had proposed a similar many-core design codenamed "Piranha" but it never went into production [Piranha].

SPARC and successors went through various versions with a final variant SPARC64 XII released in 2017 by Fujitsu. However, Fujitsu are now phasing these out in place of Arm based designs.

IBM's Power still lives on. The latest variant, the 30 billion transistor Power 11 shipped in 2025.

RISC-V, an open source design, appeared in 2010 primarily aimed at the embedded market. This is gradually picking up steam and has now sold billions of units [Patterson].

# Part 6: Return of The RISC

CISC x86-64 had the desktop, workstations and a big chunk of the server market.  Intel had the fastest processors, the best silicon process, huge numbers and huge profits.  Their lead looked unassailable, but in the 2010s a most unexpected challenger arose.

## The Mobile Challenger

The Mobile devices were no threat to the PC performance. Small low powered chips just couldn't challenge the 100W+ monsters in desktop PCs ...until they did.

In the 2010s ever finer geometries and modern low power transistors enabled techniques such as superscalar or Out-of-Order execution to be used even in chips designed for mobile phones. Arm started using these same techniques that the RISC and CISC vendors had used in the 1990s.

Mobile processor performance grew very rapidly over the 2010s unlike the PC market where generational performance increases were shrinking.  With increasingly better capabilities and devices like the iPhone appearing, followed by tablets, mobile devices even began to replace PCs as personal computing devices.

## What Went Wrong for Intel?

Intel was an engineering marvel, x86 might not have had the best ISA, but some of the best engineering and the best silicon process kept it rolling. They had the PC market sewn up, but then they made some missteps.

When Steve jobs came calling, Intel didn't produce a mobile chip for Apple. Intel was used to big profits and it wasn't clear to Intel if Apple could sell enough phones to make the big money. More importantly, Intel decided not to invest in *Extreme UltraViolet* (EUV) technology for making chips. This would prove to be a catastrophic decision. Silicon process had kept intel in front for years, this would eventually see them get stuck at the 10nm node while the rest of the industry advanced.

Intel was still the industry 800 pound gorilla and still made a ton of money, but these missteps would eventually catch up.

Contract silicon manufacturer TSMC had been growing rapidly in the background. They produced the vast numbers of processors for the embedded and mobile markets and had been catching up to Intel on silicon process. When Intel got stuck at 10nm, the rest of the industry and especially TSMC kept going, and then pulled ahead.

## AMD Gets Game

Intel's most direct competitor AMD had been in semi-permanent second place.  They went to TSMC, caught up, and then pulled ahead leading to greater success on the server and desktop spaces. Leading edge games require fast processors and AMD have also done well here. At time of writing, if you want a gaming CPU, it'll probably be an AMD part.

## Arm Closes In

As Moore's law continued, the gargantuan mobile device market and TSMC's success combined with Arm using advanced design techniques. The result was mobile processors that had previously trailed desktop designs by many years, started to catch up.

The first Arm had originally been developed as a desktop processor but Arm mostly concentrated on embedded and mobile markets*. Some Arm based laptops did start to appear in the 2010s, but these were mainly Chromebooks, or low spec nameless brands you get on ebay running Android.

Apple in particular started designing their own Arm architecture processors very aggressively for performance. Others would in turn design processors to catch up so performance went up across all mobile processors. I was personally quite shocked when the Geekbench score for an iPhone purchased in 2018 matched the single core numbers for a MacBook Pro (Core i7) purchased only a couple of years earlier.

Today desktop vendors have concentrated on multi-threaded performance. Mobile phone processors pose no threat to this, however, in some benchmarks, mobile phone processors post similar or better single-threaded benchmark numbers than current top end x86-64 processors [Bench]. It's no coincidence that Apple was able to put a mobile processor in the MacBook Neo.

* I believe there was always something for the Acorn fans, so it can be argued Arm technically never left the desktop space.

## Revenge of the RISC

In 2020 it all changed, Apple's introduced their own, in-house designed M series processors based on the Arm instruction set.  RISC had returned to the desktop, and it returned with a bang.  x86-64 laptops were big and power hungry, they dropped the clock speed when you unplugged them to save battery. Apple's laptops ran fast, and continued at full speed, even on battery.*

*FYI I'm using a MacBook pro with an M1 Pro to write this article.

Apple's success didn't go unnoticed. Qualcomm had been selling Arm based laptops for some time but they were fairly low spec compared to the x86-64 machines.  That changed when they introduced the Snapdragon X Elite series processors, these were closer to the M series processors in performance.  That said, things are complicated in the PC space. A lot of PC software was not native on Arm and emulators didn't run everything properly.  It has taken time to get things ported across and debugged.

...and there's more. Nvidia now has the Arm based RTX Spark [X925] processors and rumours continue about AMD developing an Arm based laptop chip called Sound Wave.  RISC has returned to the desktop.

## How Did RISC Get Ahead Again?

Modern top end processors from both CISC and RISC camps are remarkably similar.  They are all complex Out-of-Order machines with all sorts of extensions such as SIMD, DSP, crypto, security, virtualisation, and of course these days, AI.

If CISC designs adopted many of the same techniques as RISC designs, how have modern RISC designs managed pull ahead of x86-64 in performance again? What difference is left?

There is one area where modern CISC and RISC designs are still substantially different: The instruction set.

# Part 7: Complexity Theory

RISC was designed for simplicity. When they went to design physical processors, they simplified the instruction set to make the instruction decode process run faster and make it easier to design, usually using fixed length instructions and a limited set of addressing modes.

Note: while this says *decode*, it really means instruction fetching, instruction decoding, and also to a degree instruction execution.

CISC designs use variable length instructions and sometimes highly complex addressing modes. On these designs handling just a single instruction could be extraordinarily complicated. This complexity was magnified by the effects of virtual memory, memory caches and interrupts [Mashey].

## Complexity in Instruction Fetch

One of the lauded advantages of CISC instruction sets is they have good code density due their use of variable length instructions. This is true, but variable length instructions can make just fetching an instruction complicated.

If instructions are variable lengths, you have the immediate problem that you don't know how much data to fetch. The obvious thing to do is fetch the maximum possible and start decoding it. In most cases the instruction length should be easy to find out, but not always. On some CISC designs there are some instruction formats so complex that you wouldn't be able to work out the instruction length until well into the decoding process [Mashey]! This makes life somewhat difficult if you want to process more than one instruction at a time.

Variable length instructions also have the problem that some won't be aligned in memory, that is a 4 byte instruction won't be aligned on a 4 byte address. This means you might require multiple memory accesses just to fetch a single instruction.

Caches and virtual memory makes this even more complex. Your instruction might be across more than one cache line so when fetched, it'll require 2 reads. Of course, one of those cache lines might have been evicted in which case you then need a memory read too.

Not only can the instruction go across a cache line, it might also go across memory pages. This means you'll need 2 MMU lookups to before you can access the memory containing the instruction. The MMU might not contain one of the addresses required in which case it has to be fetched from memory before you can fetch the memory containing the relevant part of the instruction.
If that sounds complicated, remember that MMU and memory accesses can trigger processor exceptions (faults) that must be handled. In addition to this, an interrupt can occur at any point during the decode process and you have to work out how to handle it.

Any of the above potential issues can occur, so your fetch unit must have the logic to handle them. However, it's possible a number of them occur at the same time. So you must have logic to handle all possible combinations. Note that this is not an optional extra, if it can happen, your processor MUST handle it, no matter how rare or complicated it might be. If not, you'll get random crashes.

The above is about fetching just 1 instruction, a CISC processor might have many different lengths of instructions and many different formats. Modern processors fetch many instructions at once.

## Complexity in Decode

After the instruction is fetched it must be decoded.
There can be many different instruction formats with the different bits used to indicate (at its simplest):

Instruction format identifier.
Sources (operands).
The operation.
Destination of the result.

The addressing modes are where it gets complicated. These can specify that you need to calculate one or more of the addresses of the sources or destination with a base, offsets and scaling. You can also add in pre- or post-increments, and indirect addressing (where you use an address to lookup another address where you get the actual data from).

Things can get very complicated indeed, just for a single instruction. Of course, there are many different possible operations and each might have a whole set of possible addressing modes.

All this complexity is how you make up an orthogonal instruction set which is easy for humans to use, however all of this must be somehow be implemented in logic in the hardware of the processor, this must be designed, implemented and tested. It's no wonder CISC processors were often late to market, imagine debugging the logic for this.

## Complexity in Execution

CISC instructions can be highly complex involving many steps to execute, but even simple operations can be surprisingly complicated.

For example, on the 68K processors you can use an instruction like:

Add value in Register 1 to value at address X and store the result in register 2.

That sounds fairly straightforward, it's a register fetch, a memory fetch, an add, and a store. Except, what happens if there's an interrupt when you are fetching memory? Do you abandon the fetch and restart? Do you let the instruction complete before handling the interrupt? But, what if the memory access caused an exception? How do you handle this and the interrupt and in what order?
Once you've figured out how to handle these sorts of problems you have to design the logic to implement it. Note: This is an example of a **simple** CISC instruction.

Most of the 68K line stops the instruction mid flow, saves the processor state (including any temporary state) and restarts after the interrupt. This is very complicated and slow way of handling an interrupt since this involves storing and reloading a lot of intermediate state.

For a little more complexity, what if your instruction is in a loop accessing data in an array and rather than an add it's a divide. You can use an increment mode to automatically move the address through the array. This makes life easy for the programmer but you now have a different problem. Divides are a relatively slow operation and were incredibly slow back in the day (taking more than 100 cycles to complete a single divide). So, if an interrupt comes in you'd probably want to abort, otherwise the interrupt latency would be very high. However, you now have the problem that the address you are looking will be wrong when the operation is restarted, so you now need logic that knows you are using a pre-increment mode and you have to de-increment the address by the right amount before restarting the operation. However, what if the Pre-increment had not completed before the interrupt?

Again, this is a simple example. As noted above, instructions on CISC machines could be highly complex involving multiple address and pre/post increment modes. Any memory access can cause an exception and an interrupt can come in at any time. Granted some of these may be edge cases, but that doesn't matter. As noted above, If the situation can occur, the processor MUST be able to handle it.

## RISC Simplifies

RISC processors got around all this complexity by breaking things up into simpler, fixed-length instructions.
Instruction fetch became a great deal simpler by using fixed length (4 byte) and memory aligned instructions. The fetch unit has no need to calculate the length of instructions because they're always 4 bytes. To fetch an instruction just fetch 4 bytes, to fetch 2 instructions just fetch 8 bytes. The instructions were aligned so they couldn't be partly loaded, go across a cache line or a memory page.

For decode it's also a lot simpler. Instructions would only do individual operations. There were no complex instruction formats or complex addressing modes. Instruction execution was also simpler with instructions either doing a memory access or performing an operation, not both.

All this and the removal of microcode led to much simpler hardware that could run faster, provided better performance, required less power and cost less to make.
The above points about complexity of fetch, decode, and execution are what made most of the CISC processors uncompetitive, and ultimately killed them off.

# Part 8: Back to the Present

While modern CISC (that is x86-64) processors use many RISC principles in their design. They ultimately are still stuck with the old instruction set. This was improved in x86-64 but still has the fundamental CISC way of working.

Like the 68060 this complexity has an impact on the design on the processors. Of the 3 forms of complexity mentioned previously, this especially impacts fetch and decode. Fetch is most likely done via a large buffer. Instruction lengths have to be worked out so this involves partially decoding the instruction.

Once instruction sizes are established and instructions are fetched, the hardware must work out exactly which of the myriad instruction formats it is. Once the instruction format is established the is instruction sent to the relevant decoder where it is decoded and broken into micro-ops for execution.

Execution is somewhat different as it involves putting the micro-ops into a large out-of-order execution engine. It is in the execution stages that modern RISC and CISC processors are likely to be the most similar. One exception however is the CISC processors will use more temporary registers and these also need to be tracked and renamed and tracked. Though I don't think this will make that much of a difference. Status bits may prove more challenging but some RISC designs also have at least a few of these.

## Modern RISC vs. Modern CISC

There are several RISC families still going but IBM is mostly in servers while RISC-V is mostly in embedded. For desktops and workstations it's pretty much just Arm64 (officially called AArch64).

While modern RISC chips are highly complex [M1RE], they used fixed length instructions and limited numbers of instruction formats. The fetch and decode stages in these processors can be a great deal simpler than x86-64.

Fetch is done via an instruction buffer. Some top end Arm64 processor cores fetch and issue up to 10 instructions per cycle (Apple M4, M5, Nvidia Olympus [OLY]). Modern RISC designs also use micro-ops but the instructions being broken up are not nearly as complex as x86-64 instructions, and many instructions aren't broken up at all.

On the CISC side, Both AMD's Zen 5 and Intel's Lion cove / Cougar cove have 8 wide decoders, however the width of the decoder however may not be directly comparable as individual CISC instructions can be more complex and can conceivably get more done. That said, x86-64 decoders, at least in the past would be grouped with simple and complex decoders, with more of the simple decoders.

## What Difference Does This Make?

The CISC decode requires hardware in x86-64 processors to deal with a level of complexity that Arm64 or (other RISC processors) just don't require.

This extra hardware is highly complex hardware.  Hardware that calculates the lengths of instructions in parallel or predict the location of further instructions all requires power.  Adding more to handle multiple instructions per cycle only requires even more complex logic.

Running this soft of hardware at a high clock speed only serves to increases power requirements even more. I think this is the fundamental limitation that x86-64 vendors are now hitting.

The RISC decode process is so much simpler that RISC processors don't require this much complex hardware and thus don't need to power it.  They can either allocate the power to something else to boost performance, or just run at lower power levels.

Intel used to have the best silicon process in the industry and consequently made the fastest single-threaded processors. They had this lead for many, many years. This was a circular advantage - the best silicon meant they could make the fastest chips, they made more money, so Intel invested in better silicon process and this allowed them to make better chips... Intel broke that circle when they didn't invest in EUV technology, leaving them stuck on the 10nm node for several years.

In 2021 Intel brought back previous CTO Pat Gelsinger save the company. He bet on getting Intel's silicon process technology back even with TSMC. After spending a great deal of money and several years of high accelerated development, Intel's 18A process is now technically the most advanced in the industry. The engineering heroism worked but it doesn't appear to be providing much of a performance advantage as of yet. However, even if it was, silicon processes and fabrication plants have become so expensive that Intel now have to make chips for other companies, including competitors. Apple, SpaceX and Nvidia have all been mentioned [FAB].

## The Great Silicon Equaliser

The silicon process has become the great equaliser and this first enabled AMD to pull ahead of Intel. It then enabled Arm64 vendors (notably Apple) to produce laptop chips with similar or better single threaded performance than x86-64 desktop parts [Bench].

Single threaded performance may not seem important these days as modern processors add more and more cores, however, it very much still matters, probably more than you think.

Every application has sequential sections that can only run on a single core and this limits the performance of the entire application. The performance of any algorithm running on multiple cores is determined by the single threaded performance of a processor. This was first presented by computer architect Gene Amdahl in 1967 and became known as Amdahl's law. [Amdahl]

Not only is the single threaded performance level of the M5 highest on the market, it is achieved at considerably lower power consumption than x86-64 processors.  x86-64 processors have clock rates up to 6GHz and peak power consumption figures up to 170W for AMD and up to 250W for Intel's highest desktop models.

Apple's M5 Max has been measured at up to 90W under a sustained heavy load - and that includes the GPU!

To get the throughput that an Arm64 can achieve, the x86-64 processors must either increase the number of decoders, costing complexity and power, or up the clock rate, also increasing power.

Simpler decode makes for lower power and faster processors. There is the x86-64 ecosystem advisory group [EAG], who are putting forward proposals to increase performance, in the process making x86-64 even more RISCy [APX]. This will help, but it isn't clear how x86-64 vendors can fight against Arm's simpler decode advantage.

## RISC on RISC

Arm came up from below (and partly above) to attack x86-64. Arm however is now under pressure from another contender attacking Arm from below. Another modern RISC design, RISC-V, is increasing having success in the embedded market.

RISC-V is the fifth CPU developed at Berkeley. This processor is also public and open source [Patterson].

# Conclusion: RISC 46 Years on

I began this article with what I described as the 4 tenets of RISC, 46 years later, are these still relevant?

1. **Remove complex microcoded instructions**
2. **Use internal registers for operations**
3. **Remove surplus instructions**
4. **Simplify decode**

## 1. Remove complex microcoded instructions

Nobody is making processors with the type of complex microcode that the original RISC paper talked about. The move towards making processors implement high level language features appears long dead.

## 2. Use internal registers for operations

Internal registers are used exclusively in RISC instructions but not in x86-64 instructions where an instruction can include an operation and a memory access.
x86-64 have been breaking instructions into micro-ops since at least the Pentium Pro. Instructions that include memory operations are broken apart so internally x86-64 processors are essentially load-store machines. For example, a load will executed separately into a temporary register, when the data is loaded the operation is executed. If there is a memory write that would also be executed as a separate operation.

## 3. Remove surplus instructions

Plenty of *special* instructions or data types have been added over the years but these are to typically speed up specific algorithms. Other features have also been added over time such as SIMD units or security features but these increase functionality. There has been no adding of complex addressing modes just because they can.

## 4. Simplify decode

An explained in previous sections the RISC processors have a much easier time fetching and decoding instructions. I think this is the biggest difference between modern RISC and CISC designs. In my opinion, it is the complexity decoding the x86-64 instruction set that is giving processor designers an increasingly hard time improving performance, allowing Arm64 (a RISC design) to pull ahead.

## Conclusion

The type of processor designs that RISC was a reaction to, are long dead.

x86-64, the only remaining mainstream CISC architecture (which wasn't really what CISC originally meant) uses many of the RISC concepts. Processors today are a great deal more complex now than early RISC designs, but the complexity is mostly not in the areas that slowed the original CISC processors.  So, yes, in my opinion that tenets of RISC are alive and well and every bit as relevant today.

RISC has taken over the vast majority of the CPU market.  CISC may remain king on the desktop, but it's not really CISC and today, RISC processors are also making inroads on the desktop.

# Addendum: Can RISC Take a CISC?

## CISC's Final Advantage

I have talked about the complexity in CISC processors and their disadvantages, but not all of CISC is bad. One feature of CISC instruction sets that their advocates focus on is that they tend to have very good code density.  The code they run tends to be small and thus fits better into cache.

For embedded processors code density is very important as memory is at a premium. On the other hand, this appears to have relatively little effect on the desktop for most workloads. I suspect it could be providing a little extra performance.

Can RISC processors also somehow get this advantage?

## Variable Length RISC?

Some RISC instruction set architectures such as Arm's Thumb, MIPS16e, and RISC-V C extensions use a mixed 32/16 bit instruction set to achieve better code density. This makes code smaller and it is still simple to decode, but typically these techniques have limitations.

## Can You Take a Complex Feature of CISC and Make it Simple?

Is it possible for RISC take a leaf out of the CISC playbook and get even better code density, but without the associated complexity?  Can you make the variety of variable-length instructions CISC processors use, simple to fetch and decode?

I think there is a way to do this, using a technique from the CDC 6600 in 1964: Pack different lengths of instructions into a long fixed word.

## Proposal: Variable Length Instructions in a Fixed-Length, Packed Format

The aim of this proposal is to provide a way to use variable length instructions but keep the decoding simplicity of RISC.

Firstly, instructions are packed into a fixed length long instruction *package*, at for example, 128 bits. The package provides a way to fit variable length instructions together but the package itself cannot roll over a cache line or memory page.

A bit field indicates how the package is divided into instructions. For example, 4 bits would give you 16 *package-formats,* that is, 16 different ways to pack instructions.

The package-formats are predefined so the instructions will always be in set locations that are defined in the package-format. There is no requirement to calculate the length or position of instructions.

An unpacker stage is required that extracts the instructions and expands each to a full 32 bit (or bigger) fixed-length RISC format. Because the instruction locations are known by the unpacker, this is relatively simple and importantly, instruction extraction can be done in parallel. This means you can use variable length instructions but without much of the associated complexity.

Package formats can be pre-designed to include different lengths, including non-powers of two. Addressing modes can also be defined by the format. Both of these remove the need to use bits in the instructions to indicate these, thus improving code density further. This is all fixed in the package-format so is relatively easy to implement.

As found during the 801 research [801], most code uses a small number of the total available instructions. If the most commonly used are made as small as possible, there can potentially be some very good code density gains. Andrew Tanenbaum encouraged this in his work [EM-1] but not in a packed format.

I think this example serves as an illustration of the RISC philosophy. It's not about the features you want to implement, it's how you do it. Variable length instructions are typically seen as a CISC feature, however in this proposal I show how they can be done in a manner compliant with the RISC philosophy.

# References

**[MAIN]**
This is a long article, so here's some mellow 80s synth music to listen to while you're reading, from an appropriately named band, (FYI I prefer the restored version of the album).
https://mainframe-music.info/index.html

**[RISC]**
The Case for the Reduced Instruction Set Computer - D. Patterson / D. Ditzel.
https://people.cs.umass.edu/~emery/classes/cmpsci691st/readings/Arch/RISC-patterson.pdf

**[RISC-I]**
Paper describing the Berkeley RISC-I processor.
RISC I: A Reduced Instruction Set VLSI Computer - D. Patterson / C. Sequin.
https://people.eecs.berkeley.edu/~kubitron/courses/cs252-S07/handouts/papers/p216-patterson.pdf

**[RETRO]**
Retrospective:  RISC 1: Reduced Instruction Set Computer - D. Patterson / C. Sequin.
https://www.researchgate.net/publication/220771227_Retrospective_RISC_I_A_Reduced_Instruction_Set_Computer

**[MCode]**
8086 microcode is complex stuff - and the 8086 is relatively simple!
How the 8086 processor's microcode engine works - K. Shirriff.
https://www.righto.com/2022/11/how-8086-processors-microcode-engine.html

**[68K]**
Design Philosophy Behind Motorola's MC68000 - T. Starnes.
http://www.easy68k.com/paulrsm/doc/dpbm68k1.htm

**[HLLCA]**
Retrospective on High-Level Language Computer Architecture -  D. Ditzel / D. Patterson.
https://people.eecs.berkeley.edu/~kubitron/cs252/handouts/papers/p97-ditzel.pdf

**[801]**
The evolution of RISC technology at IBM - J. Cocke / V. Markstein.
https://acg.cis.upenn.edu/milom/cis501-Fall11/papers/cocke-RISC.pdf

**[MIPS-X]**
Architectural Tradeoffs in the Design of MIPS-X - P. Chow / M. Horrowitz.
https://archive.org/details/DTIC_ADA182299

**[CISC]**
Definition from the Computer Architecture Bible:
Computer Architecture: A Quantitative Approach - J. Hennessy / D. Patterson.

**[Mashey]**
A deep dive into CISC and RISC instruction set complexity - J. Mashey.
https://homepages.cwi.nl/%7Erobertl/mash/RISCvsCISC

**[APX]**
Proposal for upping the number of x86-64 registers to 32. - X86 Ecosystem advisory group.
https://x86ecosystem.org/blog/apx-201/

**[Ditzel]**
Conference slides - D. Ditzel.
https://microarch.org/micro55/media/keynote_ditzel.pdf

**[SCJ]**
Iterative processor development at Bell labs - S. Johnson.
https://hoult.org/1979_Johnson_A_32_bit_Processor_Design.pdf

**[EM-1]**
Tanenbaum Proposes Simpler Architecture in 1978.
Implications of Structured Programming for Machine Architecture - A. Tanenbaum.
https://research.vu.nl/ws/files/110789436/11056

**[6600]**
CDC 6600, the first supercomputer - HandWiki.
https://handwiki.org/wiki/CDC_6600

**[88K]**
Apple / NeXT 88110 development - Stories of Apple.
https://www.storiesofapple.net/the-88110-cpu-and-the-risc-workstations-that-never-were.html

**[EV]**
Electric Vlasic (pickle).
Characterization of Organic Illumination Systems - Various.
https://web.archive.org/web/20080706151313/http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-13.pdf

**[060UM]**
M68060 User's Manual - Motorola 1994.
https://www.nxp.com/docs/en/data-sheet/MC68060UM.pdf

**[ARM-H]**
A History of The ARM Microprocessor - D. Jaggar (Talks at Google).
https://www.youtube.com/watch?v=_6sh097Dk5k

**[SPEC]**
List of SPEC 92 / 95 results from early/mid 90s.
https://www.cs.toronto.edu/pub/jdd/spectable

**[Lightroom]**
Amiga Lightroom Benchmarks (FYI: My 2.8GHz Raspberry Pi 5 runs this in 47 seconds).
https://drive.google.com/file/d/1_VxGqSt2fMQbDxdZNO-SvA5Th0gfgp3n/view?pli=1

**[HOMBRE]**
Hombre, the future Amiga that never was :-(
https://grokipedia.com/page/Amiga_Hombre_chipset
Full Specs:
https://archive.org/details/Hombre_201808/Hombre_Presentation_part_1/

**[Piranha]**
Multi-core Alpha.
Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing - Various.
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/isca00.piranha.pdf

**[Patterson]**
Lex Fridman interviews David Patterson (includes RISC vs. CISC, RISC-V, and Wrestling).
https://www.youtube.com/watch?v=naed4C4hfAg

**[X925]**
Deep dive on the X925 core used in the RTX Spark.
Arm's Cortex X925: Reaching Desktop Performance - C. Lamb.
https://chipsandcheese.com/p/arms-cortex-x925-reaching-desktop

**[Bench]**
See below.

**[M1RE]**
Apple M1 Reverse Engineered - caohuajia.
https://github.com/caohuajia/apple-m1/blob/main/vol1%20M1%20Explainer.nb.pdf

**[OLY]**
Nvidia Olympus Architecture
https://nvdam.widen.net/s/nmw5vblpqd/nvidia_vera_cpu_architecture_whitepaper?nvid=nv-tblg-543584

**[FAB]**
Intel make chips for Apple, SpaceX and Nvidia.
https://www.macrumors.com/2026/07/13/apple-agreed-intel-chips-trump-tariff-talks/

**[Amdahl]**
Amdahl's law
https://lawsofsoftwareengineering.com/laws/amdahls-law/

**[EAG]**
x86 ecosystem advisory group.
https://x86ecosystem.org/

**[Bench]**
Contemporary benchmark links:

SPECrate2026 integer base rate-1 (Estimated Performance):
https://benchview-hjc-im.translate.goog/?_x_tr_sl=auto&_x_tr_tl=en&_x_tr_hl=en-US&_x_tr_pto=wapp&_x_tr_hist=true#dataset=data%2FSPECrate2026_int_base.csv
Above is linked from David Huang's Blog:
https://blog.hjc.im/spec-cpu-2017

PassMark Single Thread:
https://www.cpubenchmark.net/single-thread/

Cinebench 2024 Scores:
https://www.cpu-monkey.com/en/cpu_benchmark-cinebench_2024_single_core

Cinebench 2026 Single-threaded:
https://www.cpu-monkey.com/en/cpu_benchmark-cinebench_2026_single_thread

Geekbench 6 Single-threaded:

https://www.cpu-monkey.com/en/cpu_benchmark-geekbench_6_single_core

Early Nvidia Vera benchmarks, single + multi-threaded:
https://www.phoronix.com/review/nvidia-vera-benchmarks

# Glossary

Note: Some of these terms are difficult to define as different vendors often use different terms and to add to the confusion they are often used interchangeably.

**AArch64 / Arm64**
The 64 bit instruction set defined by Arm. AArch64 is the official name, Arm64 is more informal.

**Addressing mode**
This determines how a value is interpreted in an instruction.

**ARM / Arm**
The company that defines the Arm architecture (i.e. instruction set architecture) and designs and licenses Arm processors (and more). Other companies such as Apple and Qualcomm design their own processors based on the Arm instruction set.
Note: "ARM" changed to "Arm" in a 2017 rebrand.

**Cache**
A small on-chip memory that is used to speed up access to data and instructions.

**Cache line**
A sub-division of memory in a cache. The smallest contiguous block of memory that can be transferred between memory and cache.

**Code density**
How much memory a piece of code takes up. Variable length instructions typically have higher code density.

**Decode / decoder**
The stage where an instruction is interpreted. This determines what operation it represents.

**Exception / fault**
An exception is an internal event triggered by the processor while executing an instruction. It tells the processor to stop what it's doing and deal with what caused the exception.
A fault is a type of exception. See Interrupt.

**Fetch**
The stage where a processor gets (fetches) an instruction (or instructions) from memory into an internal buffer.

**Instruction format**
Describes the structure of an instruction.

**Instruction set / *Instruction Set Architecture* (ISA) / architecture**
The defined set of processor instructions and any effects they have.

**Interrupt**
An external signal that tells the processor to stop what it's doing and deal with what caused the interrupt. See exception.

**Micro-architecture**

The design or layout of the physical chip.

**Micro-op (Micro-operation)**
The basic operations that the processor performs. Many kinds of microprocessors, including RISC processors break instructions into micro-ops.

**Memory page**
Memory is typically divided into multiple blocks called pages. This is used to enable multiple-users, multitasking, and security.

**Microcode**
A type of code executed inside the processor. Usually involved in implementing an instruction, especially complex instructions.

**Out-of-Order execution**
Technique used by modern microprocessors to increase performance.
involves executing instructions out of sequence where possible.

**PAL code**
DEC's name for a technique similar to micro-code but typically much simpler.
Used in many processors, including RISsC processors.

**Pipeline**
A series of stages of execution in a processor operating like an assembly line. This enables multiple instructions to be in different stages of execution simultaneously.

**Register**
A small fast storage location inside a processor. Typically 32 or 64 bits long.

**Superscalar**
A processor that can issue more than one instruction per clock cycle.

**Variable length instructions**
Instructions whose size varies. These typically have high code density.