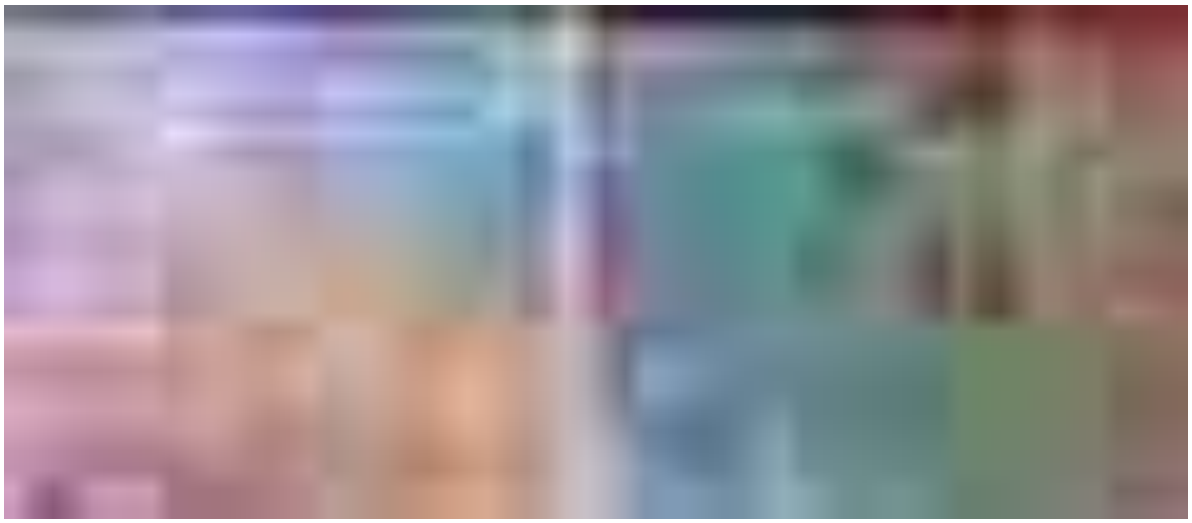
medium.com

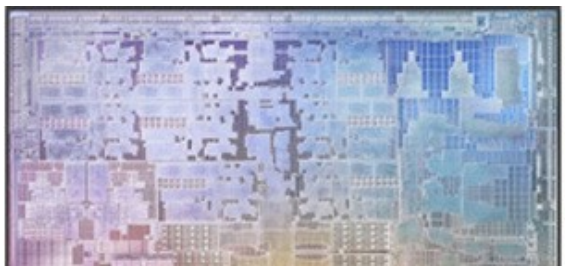
What Does RISC and CISC Mean in 2020? - The Startup - Medium

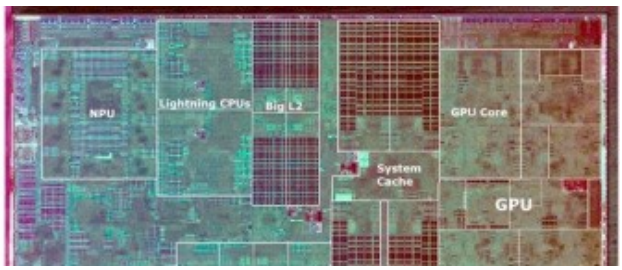
Erik Engheim

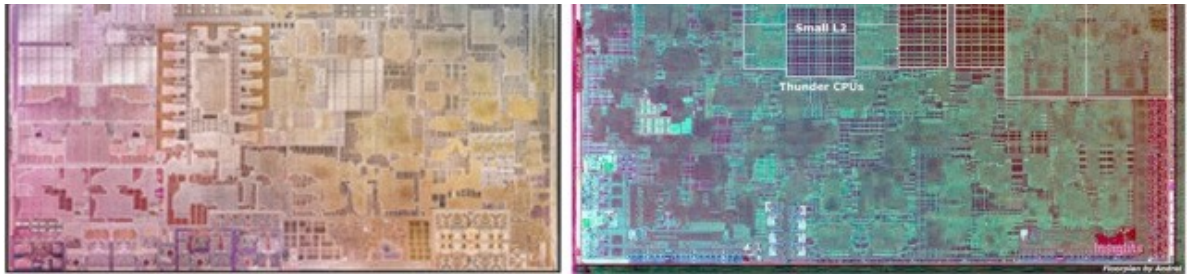
32-41 minutes

Many today say the difference between RISC and CISC has become irrelevant. Is that true? And if not what exactly is the difference between modern RISC and CISC CPUs?









Apple has now released the M1 Apple Silicon chips, and after all the fuss you may be wonder what makes it different from an intel or AMD processor? You have probably heard M1 referred to as an ARM processor and that ARM is a so called RISC processor, unlike x86 processors from intel and AMD.

Read More: Why Is Apple's M1 Chip So Fast?

If you try to read up about the difference between RISC and CISC microprocessors, you will hear a lot of people tell you that RISC and CISC doesn't matter anymore. That they are essentially the same. But is that really true?

Okay, so you are confused and you want some straight answers. Well, then this article is the right place for you to start.

I have plowed through a ton of comments and writing on this, sometimes by the engineers themselves that created these chips, so you don't have to waste time on this.

First I will start with some of the basics you got to understand before we begin to answer some of the deeper questions such as RISC vs CISC. I am putting in headlines, so you can skip over the stuff you already know.

Here are some of the topics I will cover in this article:

- What is a CPU?
- What is an Instruction Set Architecture (ISA)

- Why pick one ISA over another?
- How are RISC and CISC instruction sets different?
- CISC Philosophy
- RISC Philosophy
- Pipelining
- Load / Store Architecture
- Compressed Instruction Sets
- Microcode vs Micro-operations
- How is a Micro-operation different from a RISC instruction?
- Hyper-threading / Hardware threads
- Does RISC vs CISC still make sense?

What is a Microprocessor (CPU)?

Let us just clarify what a microprocessor is. You probably already have some idea, otherwise you would not have clicked on this article.

The CPU is basically the brain of the computer. It reads instructions from memory, telling the computer what to do. These instructions are just numbers which have to be interpreted a specific way.

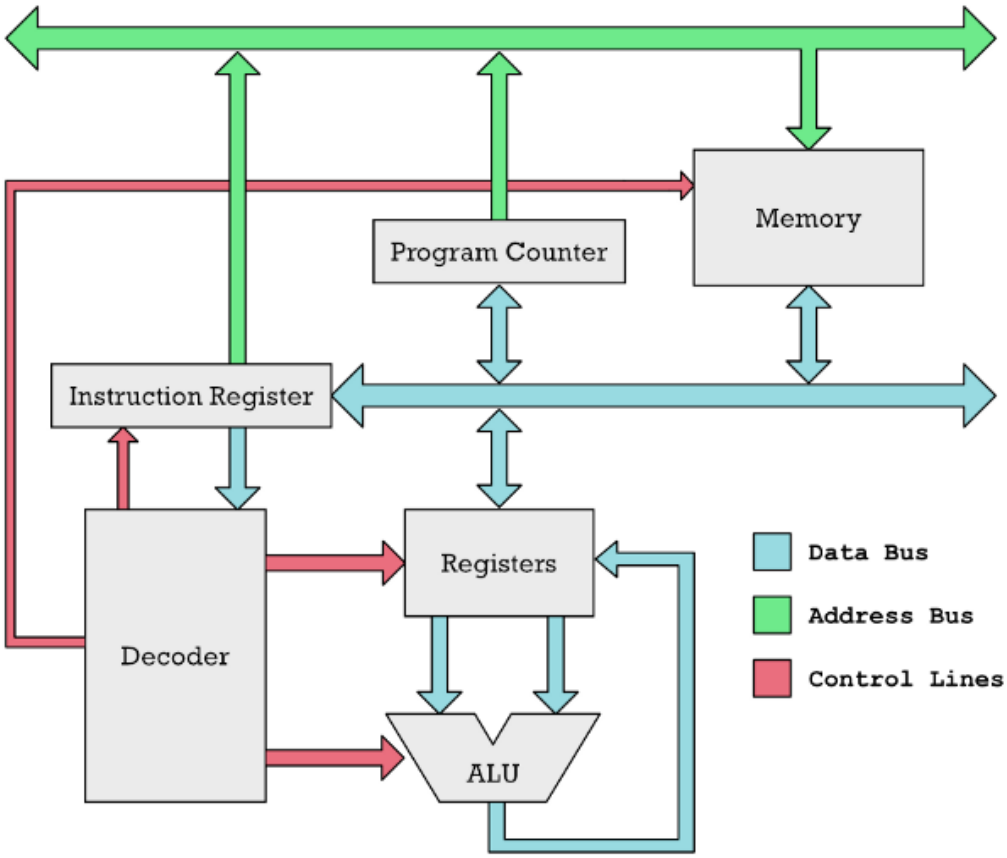
There is nothing in memory that marks a number off as being just a number or specifically an instruction. Instead it is up to the makers of the operating system and programs to make sure instructions and data is put into the locations where the CPU expects to find program code and data.

Microprocessors (CPUs) do very simple things. Here is an example of the kind of instructions a CPU follows:

```
load r1, 150
load r2, 200
add  r1, r2
store r1, 310
```

This is a human readable form of what would just be a list of numbers in the computer. For instance `load r1, 150` would in a typical RISC CPU be represented by a single 32-bit number. That means a number with 32 digits, where each digit must be a 0 or 1.

`load` in first line, moves the contents of memory location 150 to register `r1`. Your computer memory (RAM) is a collection of billions of numbers. Each of these numbers has an address (location) so that the CPU has a way of picking the right one.



A simplified diagram of the operations of a CPU. Instructions are moved into instruction register where they are decoded. Decoder activated relevant parts of CPU to make operations happen.

Next, you may wonder what a **register** is. This is actually a pretty old concept. Old mechanical cash registers in convenience stores also had the concept of *registers*. Back then a register was some sort of mechanical contraption that held the number you want to operate on. Often it would have an *accumulator register* which you could add values to. It would keep track of the sum.

Your electronic calculator is the same. What you see most of the time on the display is the contents of the accumulator. You do a

bunch of computations that affect the contents of the accumulator.

CPUs are the same. They have a number of registers often given simple names such as A, B, C or r1, r2, r3, r4 etc. CPU instructions usually do operations on these registers. They may add two numbers together stored in different registers.

In our example `add r1, r2` adds the contents of `r1` and `r2` together and store the result in `r1`.

Finally we want to store the result in memory (RAM) using `store r1, 310`, which stores the result in the memory cell with address 310.

What is an Instruction Set Architecture (ISA)?

As you can imagine there is a limited number of instructions that a CPU understands. If you are familiar with regular programming where you can define your own functions, machine code is *not* like that.

There is a *fixed* number of instructions that the CPU understands. You as a programmer cannot add to this set.

There are a large number of different CPUs, and they don't all use the same instruction set. That means they will not interpret numbers for instructions the same way.

In one CPU architecture `501012` may mean `add r10, r12` while in another architecture it may mean `load r10, 12`. The combination of instructions a CPU understands and the registers it knows about is called the *Instruction Set Architecture* (ISA).

Intel and AMD chips e.g. both understand the x86 ISA. While e.g.

the chips Apple use in their iPhone and iPad devices such as the A12, A13, 14 etc all understand the ARM ISA. And now we can add M1 to that list.

These chips are what we call Apple Silicon. They use the ARM ISA, like many other mobile phones and tablets. Even gaming consoles such as Nintendo and the worlds current fastest super computer uses the ARM ISA.

x86 and ARM is not interchangeable. A computer program will be compiled for a particular ISA, unless it is a JavaScript, Java, C# or similar. In this case the program is compiled to byte code which is a CPU-like ISA, but for a CPU that doesn't really exist. You need a Just in Time Compiler or interpreter to translate this made up instruction set to the actual instruction set used on the CPU in your computer.

This means that most current programs on the Apple Macs will not run out of the box on the new Apple Silicon based Macs. Current programs are made up of x86 instructions. To solve this programs have to be recompiled for the new ISA. And Apple has an ace up their sleeve with Rosetta 2, which is a program that will translate x86 instructions to ARM instructions before it is run.

Why Change to an Entirely Different ISA?

Now the next question is. Why use a new ISA for their Macs? Why couldn't Apple just make their Apple Silicon understand the x86 instructions? No recompile or translation with Rosetta 2 needed.

Well, it turns out all instruction sets are not created equal. The ISA of a CPU heavily influence how you can design the CPU itself. The particular ISA you use can complicate or simplify the job of

creating a high performance CPU or a CPU that consumes little power.

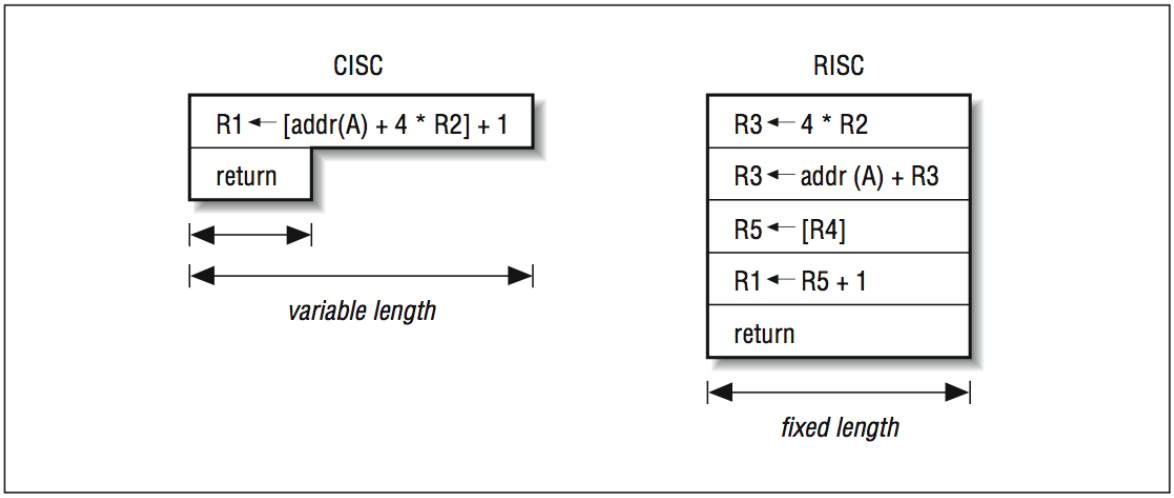
The second issue is licensing. Apple cannot freely make their own CPUs with an 86 ISA. It is part of Intel's intellectual property and they don't want competitors. The ARM company in contrast doesn't actually build their own CPUs. They just design the ISA and provide reference designs for CPUs implementing this ISA.

Hence ARM lets you do almost exactly what you want. This is what Apple wants. They want to create tailor made solutions for their computers with specialized hardware handling things like machine learning, encryption and face recognition. If you go with x86 you have to do all that on external chips. For efficiency reasons Apple wants to do all that stuff in one large integrated circuit, or what we call a System on a Chip (SoC).

This is a development that began on smart phones and tablets. They are too small to have lots of different separate chips on some big motherboard. Instead they must integrate everything typically spread out over one motherboard into one chip, that contains CPU, GPU, memory and other specialized hardware.

Now this trend is arriving on laptops and likely later on desktop PCs. Tight integration gives better performance, and here x86 with their inflexible licensing scheme is a big disadvantage.

But let us not derail what this article is primarily about: RISC vs CISC. Instruction Set Architectures tend to follow different core philosophies for how the ISA is defined. x86 is what we call a CISC architecture. While ARM follows the RISC philosophy. That makes a big difference. So let us delve into the key difference.



CISC instructions can in principle be any length. E.g. theoretical length of some x86 instructions are infinite (but clamped at 15 bytes for practical reasons). RISC instructions tend to be of fixed length.

CISC stands for Complex Instruction Set Computer, while RISC stands for Reduced Instruction Set Computer.

Explaining what the difference is today, is harder than when RISC first came out because both RISC and CISC processor have stolen ideas form each other, and there has been a heavy marketing campaign with the interest of bluring the distinction.

A Detour on Marketing Disinformation

Paul DeMone writes this article back in 2000, which gives some

idea of the marketing pressure that developed early on.

Back in 1987, the top of the line x86 processor was an intel 386DX, while a top of the line RISC processor was MIPS R2000.

Despite the fact that the intel processor had more transistors, 275 000 vs 115 000 on the MIPS and had twice as much cache, the x86 processor was completely demolished in performance tests.

Both processors ran at 16 MHz clock rate, but the RISC processor had 2–4 times higher performance, depending on benchmark used.

Thus it is not strange that by the early 90s it had become a generally accepted idea that RISC processors had vastly better performance.

Intel thus started getting a perception problem in the market. They had problems convincing investors and buyers that their outdated CISC design could beat a RISC processor.

Thus Intel began marketing their chips as being RISC processors, with a simple decoding stage in front which turned CISC instructions into RISC instructions.

Thus Intel could present themselves in an attractive way: They would say with our chips you still get technologically superior RISC processors but our RISC processors understands x86 instructions which you already know and love.

But let us clarify this right away: There are no RISC internals in x86 chips. That is just a marketing ploy. Bob Colwells, one of the main creators of the intel Pentium Pro, regarded as the chip with RISC inside makes this clear himself.

You will however see this falsehood propagated all over the

internet, because Intel was really good at pushing this marketing ploy. It works because there are some half truths to it. But in order to really understand RISC vs CISC you really have to begin by dispensing with this myth.

Thinking that a CISC processor can have a RISC processor inside it will just confuse you about the difference between RISC and CISC.

The CISC Philosophy

Okay with that rant out of the way, let us look at what RISC and CISC is really about. Both are about philosophies in how you design a CPU.

Let us begin by looking at the CISC philosophy. CISC is harder to pin down as the chips we label CISC have a large variety of designs. But we can still talk about some common patterns.

Back in the late 1970s when CISC processor began development, memory was still really expensive. Compilers were also really bad and people tended to write programs in assembly code by hand.

Because memory was expensive, you needed to find ways of conserving it. The way of doing this was to come up with really powerful CPU instructions which could do a lot of stuff.

This also helped Assembly programmers as they could more easily code programs, as there would always be some single instruction that did what they wanted.

This started after a while to get really complicated to deal with. Designing decoders for all these instructions became a hassle. One way they solved this with initially was to invent Microcode.

In program you know you can avoid repeating common tasks by placing them in a separate subroutine (function), which you could call repeatedly.

The idea of Microcode was similar. For each machine code instruction in the ISA you make a little program stored in special memory inside the CPU made up of much simpler instructions called Microcode.

Thus the CPU would have a small set of simple Microcode instructions. They could then add lots of advance ISA instructions by simply adding new little Microcode programs inside the CPU.

This had the added benefit that the memory holding these Microcode programs was Read-Only Memory (ROM), which at the time was much cheaper than RAM. Hence reducing RAM usage at the expense of increasing ROM usage was an economic tradeoff.

So everything looked really nice in CISC land for a while. But eventually they started running into problems. All these Microcode programs became a problem to deal with. Sometimes they would make a programming mistake. Fixing a microcode program with a bug, was a much bigger hassle than fixing a bug in a regular program. You cannot access and test that code like you access and test regular software.

Some people started thinking: there has to be a simpler way of dealing with this mess!

The RISC Philosophy

RAM started getting cheaper, compilers got better and people were not programming as much in assembly code anymore.

This change in the technological landscape caused the emergence of the RISC philosophy.

They had started observing by analyzing programs that the complex instructions that was added to CISC was not used very much by people.

Compiler writers also found it hard to pick one of these complex instructions. Instead they preferred to combine simpler instructions to get the job done.

You could say we had a case of the 80/20 rule. Roughly 80% of the time was spent running 20% of the instructions.

So the RISC idea was: Let us ditch the complex instructions and replace them with fewer simple instructions. Instead of debugging and fixing Microcode programs which is hard. You leave it to compiler writers to solve the issues.

There is some contention in what the word *Reduced* should be interpreted as in relation to RISC. It has been interpreted as reducing the number of instructions. But a more sensible interpretation is that it means a reduction in instruction complexity. Meaning the instructions themselves are made simpler. This does not necessarily mean simple seen from the perspective of the user, but rather simpler to implement in hardware and preferably utilizing few CPU resources at the same time.

RISC code is not necessarily easier to write for a human. I made that mistake years ago when I thought I would save time writing my assembly code using PowerPC instructions (an IBM RISC architecture with tons of instructions). It gave me a lot of extra work and frustrations.

One of the key rationals behind RISC was that people had stopped handwriting assembly code and you needed an instruction set which was easy for compilers to deal with. RISC is optimized for compilers not necessarily for humans.

Although particular RISC instruction sets can in some ways feel easier to write for a human because there are a lot fewer instructions to learn. On the other hand you typically have to write more instructions than if you used a CISC instruction set.

Pipelining: A RISC Innovation

Another core idea of RISC was pipelining. Let me give you a simple analogy to give a motivation for what this is about.

Think of shopping in the grocery store. Now this works a bit different in every country, but I am basing this on how it works in my native Norway. You can divide the activities at the cash register in multiple steps:

1. Put the items you bought on the conveyer belt, and get them scanned.
2. Use the payment terminal to pay for the goods just scanned.
3. Pack the goods you just paid for into bags.

Food vector created by pch.vector — www.freepik.com

If this happened in a non-pipeline fashion, which is how most CISC processor operated initially, then the next customer would not begin putting their groceries on the conveyer belt until you had packed up all your groceries and left.

This is inefficient, because the conveyer belt could be used to scan items while you pack. Even the payment terminal could be used while you pack. Resources are thus under utilized.

We can think of each step as taking a clock cycle or time unit. That means it takes 3 time units to process each customer. Thus in 9 time units you have only processed 3 customers.

However we could pipeline this process. Once I start operating the payment terminal, the next customer could put their food products on the conveyor belt.

When I start packing, this customer could use the payment terminal which I just finished using. A third customer could at this point begin putting their groceries on the conveyor belt.

The result of this approach is that every time unit somebody will be finish packing and leaving with their groceries. Thus in 9 time units you would process 6 customers. As time increases, you would get close to processing one customer in just 1 time unit, all thanks to pipelining. That is a 9x speedup.

We could describe using the cash register as having a latency of 3 time units, but a throughput of one shopping per 1 time unit.

In microprocessor terminology this would mean 1 instruction has a latency of 3 clock cycles, but an average throughput of 1 instruction per clock cycle.

Now there was a bunch of assumption I put into this example, which was that every stage in the checkout took equal amounts of time. That putting groceries on the conveyor belt took about the same time as operating the payment terminal or packing the groceries.

If the times varies a lot for each stage, this does not work well. E.g. if somebody has a lot of groceries on the conveyor belt, then the payment terminal and packing area remain unused for many time unites, dropping the efficiency of the whole thing.

RISC designers understood this. Thus they tried to standardize how long each instruction is and split up what an instruction does into stages which take roughly the same time. That way each resource inside the CPU can remain utilized to the max constantly as instructions get processed.

E.g. if we look at the ARM RISC processor, it has a 5-stage pipeline for processing instructions:

- **Fetch** instruction from memory and update program counter to be able to fetch next instruction next clock cycle.
- **Decode** instruction. Meaning figure out what it is supposed to do. That means activating various electric wires to toggle on different parts of the CPU we are using to perform the instruction.
- **Execute** involves using the Arithmetic Logic Unit (ALU) or perform shift operations.
- **Memory** Access data in memory if relevant. That is what a `load`

instruction would do.

- **Write back** results of previous operation to relevant register.

  The ARM instructions have sections dealing with each of these parts, and each stage typically takes 1 clock cycle. This makes it easy to push ARM instructions through a pipeline.

  E.g. because each instruction has the same size, the **Fetch** stage knows how to get the next instruction. It doesn't need to decode first.

  With CISC instructions is this tricky. Instructions can be variable length. So you don't really know until you decode parts of the instruction where the next instruction will be.

  The second problem is that CISC instructions can have arbitrary complexity. Making multiple memory accesses and doing a whole host of things which means you cannot easily divide a CISC instruction into cleanly separate parts which can execute in a staged fashion.

  Pipelining was the killer feature, which really caused early RISC processors to destroy their CISC counterparts in performance.

Load/Store Architecture

To keep the number of cycles needed for each instruction relatively uniform and predictable to make everything pipeline friendly, RISC ISAs separate loading and storing to memory clearly from other instructions.

In CISC e.g. an instruction may load data from memory, perform an addition, multiplication or whatever and write the result back to memory.

In the RISC world this is generally a big no-no. RISC operations such as add, shift, multiply etc generally only work on registers. They don't access memory.

This is important for pipelines to work. Otherwise instructions in pipelines can get all sorts of dependencies between them.

Multiple Registers to Avoiding Memory Bloat

A big challenge for RISC relative to CISC is that with simpler instructions, more instructions are needed and memory, while not expensive, is slow. If a RISC program ends up consuming a lot more memory than a CISC program it can end up running a lot slower because the CPU is constantly waiting for slow memory reads.

RISC designers made some observations to solve this problem. They observed that a lot of instructions are really just moving data in and out of memory, to prepare for different operations. By having a larger number of registers, they could cut down on the number of times they had to write data back to memory.

This did require improvements in compilers. Compilers had to analyze programs well and understand when they could keep variables in a register and when it had to be written back to memory. Juggling a ton of registers became an important thing for compilers to make RISC CPUs run fast.

Because RISC instructions were simpler. There was not lots of different addressing modes to deal with e.g. there was more bits available among the 32-bits for a whole instruction to specify the number of a register.

This is important to understand. A CPU could easily have

hundreds of registers. That is not a big deal. It does not require a lot of transistors. The problem is having enough bits available in the instruction to address each of these. x86 e.g. could only spare 3 bits to specify each register. That only gave $2^3 = 8$ registers. RISC CPUs saving a lot of bits used for addressing modes could afford 5 bits to specify registers, giving them $2^5 = 32$ of them. Obviously this varies. But it seems to have been commonly the case.

I keep writing **addressing mode** without explaining what it is. Basically it is a different ways of fetching data. E.g. by specifying a constant representing an address, using an absolute or relative address. Using the contents of a register as an address etc. CISC CPUs had instructions which allowed you to e.g. add numbers found in memory locations in addition to values encoded in the instruction itself or in another register. RISC makes this simple. Only load and store instructions do memory access.

Compressed Instruction Sets

Compressed instruction sets is a relatively new idea in the RISC world to reduce the memory bloat issue that RISC faces in competition with CISC.

Because it is new, ARM had to retrofit this as ARM was not originally designed with this in mind while the modern RISC-V ISA has a compressed instruct set designed in from the beginning.

This is a twist on the CISC idea. Remember CISC could have really short and really long instructions.

RISC cannot easily add short instructions as it complicates how the pipelines work. Instead designers came up with the idea of

compressed instructions.

Basically this means that a frequently used subset of the normal 32-bit instructions are fitted into 16-bit instructions. Thus each time the RISC processor fetches an instruction, it is actually potentially fetching two instructions.

In e.g. RISC-V there will be a special flag that indicates whether it got a compressed instruction or not. It if is compressed, it will decompress to two separate 32-bit instructions.

This is important, because it means the rest of the CPU can operate as normal. It sees the same nice uniform 32-bit instructions where all the different stages are encoded in predictable standard locations.

Hence compressed instructions doesn't really add any new instructions. This relies heavily on smart assemblers or compilers. A compressed instruction has fewer bits available and hence cannot possibly do all the variation of things a 32-bit instruction can do.

Hence a compressed instruction may only be able to access the 8 most used registers. Not all 32. I may not be able to load equally large number constants or memory offsets.

Thus it is up to the assembler or compiler to figure out whether a particular pair of instructions can be packed together or not. The assembler will have to look for opportunities to perform compression.

So while this looks a bit like CISC, it really isn't. The rest of the CPU, the pipeline etc sees the same 32 bit instructions as usual.

On ARM you even have to switch mode to execute compressed

instructions. The compressed instruction set on ARM is called Thumb. So this is quite different from CISC. You would not perform a mode change to run a short instruction.

Compressed instruction sets have been a game changer for RISC. Some RISC variants manage to use fewer bytes than x86 to for the same programs using this strategy.

Larger Caches

Cache is a special form of really fast memory that you can put on your CPU die. This will of course take away valuable silicon real-estate needed by your CPU, so there are limits to how much cache you can add.

The idea of cache is that most real world programs run a small subset of the program far more frequently than the rest of the program. Often small parts of the program are repeated countless times, such as in the case of loops.

Thus by putting these frequently used parts of your overall program into cache you can greatly speed up your programs.

This was the early RISC strategy when RISC programs took more memory than CISC programs. Because RISC CPUs were simpler, they required fewer transistors to implement. That left more silicon real-estate which could be used for other things such as cache.

Thus by having larger caches RISC CPUs compensated for their programs being somewhat larger than CISC programs.

However with instruction compression this is of course not really true anymore.

CISC Strikes Back — Micro-operations

Of course CISC didn't sit still and let itself be beat to pulp by RISC. Both Intel and AMD came up with strategies to emulate some of the advantages of RISC.

In particular they needed a way to be able to pipeline their instructions and that was never going to work well with traditional CISC instructions.

The solution was to make the internals of the CISC CPU more RISC like. The way this was achieve was by having the instruction decoded break CISC instruction up into multiple simpler instructions called Micro-operations.

Like RISC instructions these Micro-operations are easier to put into a pipeline because they have fewer dependencies between each other and execute in a more predictable number of cycles.

How Are Micro-operations different from Microcode?

Microcode are small ROM programs which execute to mimic a more complex instruction. However unlike Micro-operations they are not put into a pipeline. They are not made for that purpose.

In fact Microcode and micro-ops usually exist side by side. In a CPU using micro-ops, the microcode programs will instead of executing directly be used instead to generate a series of micro-ops which are pushed into a pipeline for later execution.

Keep in mind that a Microcode in a traditional CISC CPU would need to actually perform decoding and execution. As they execute they take control of different CPU resources such as the ALU,

registers etc.

In modern CISC, they would finish faster because they are not using any CPU resources. They are simply used to produce a series of Micro-ops.

How are Micro-operations different from RISC Instructions

This is a common confusion. People think a micro-op is the same as a RISC instruction. They are not at all the same.

A RISC instruction exists at the ISA level. It is what a compiler targets. They are concerned with describing what you want to do. And we try to optimize them to not use too much memory etc.

A micro-op in contrast is entirely different. Micro-ops tend to be large. They can be more than 100 bits. It doesn't matter how large they are because only a few of them exist temporarily. That is different from RISC instructions which make up whole programs consuming potentially giga bytes of data. They cannot be arbitrarily long.

Micro-ops are specific to a particular CPU model. Each bit tends to specify completely specific parts of the CPU to enable or disable when they execute.

Basically if you make an instruction larger you don't need to do any decoding. Every bit can correspond to a particular hardware resource in the CPU.

Thus different CPUs with the same ISA can have different micro-ops internally.

In fact many high end RISC processor today turn their instructions

into micro-ops. That is because micro-ops tend to be even simpler than a RISC instruction. But this is not a requirement. A lower performance ARM processor may not use micro-ops while a higher performance ARM processor with exactly the same instruction-set may use it.

The RISC advantage still exists. CISC ISA instructions were not designed to be easy to pipeline. Hence breaking those instructions into micro-ops is a complex and messy task, that may not always work great. Translating RISC instructions to micro-ops will typically be more straight forward.

In fact some RISC processor use Microcode for some of their instructions just like CISC CPUs. One example of this is for saving and restoring registers in relation to subroutine calls. When a program jumps to another subroutine to do a task, that subroutine will use a number of register to perform local calculations. The code calling the subroutine doesn't want its registers to randomly change, so it must often save them to memory.

This is such a frequent occurrence that adding a specific instruction to save multiple registers to memory was too tempting. Otherwise these instructions can eat up a lot of memory. Since this involves repeated memory access, it makes sense to add this as a microcode program.

However not all RISC processors do this. RISC-V e.g. tries to say more pure and don't have special instructions like this. RISC-V chips are very focused on having an ISA optimized for pipelining. Following the RISC philosophy more strictly tends to make pipelining more effective.

Hyper-threading or Hardware Threads

Another trick CISC employed to get back at RISC was to use hyper-threading.

Remember micro-ops aren't easy to make that cleanly. Your pipeline will still likely not be filled up in as regular interval as a RISC pipeline.

The trick then is to use hyper-threading. A CISC CPU would take in multiple streams of instructions. Both instruction streams would be hacked to pieces and turned into micro-ops.

Because this process is imperfect you will get a number of gaps in the pipeline. But by having an extra stream of instructions you can push in other micro-ops into these caps and thus keep the pipeline full.

This tactic is in fact also useful on RISC processor, because not every RISC instruction can execute in the same time frame. Memory access e.g. will often take longer time. The same goes for saving and restoring registers with those complex microcoded instructions some RISC processors use. There will also be jumps in the code which will cause gaps in the pipeline.

Hence more advance and high performance RISC CPUs such as the IBM POWER CPUs will use hardware threads as well.

However my understanding is that hyper-threading is a trick that benefits CISC more. Because the production of micro-ops is less perfect on CISC they get more gaps to fill and hence hyper-threading can boost performance more.

If your pipelines stay full, there is nothing to gain from hyper-threading/hardware-threads.

Hardware threads can also present a security risk. Intel has faced security problems with hardware threads because one instruction stream can influence another. The details of this I don't know. But apparently some vendors choose to turn off Hardware threads for this reason.

Hardware threads as a rule of thumb give about 20% speed boost. So a 5 core CPU with hardware threads perform similar to a 6 core CPU without hardware threads. But as I said this number will depend a lot on your CPU architecture.

Anyway that is one of the reasons that a number of high performance ARM chip makers such as Ampere, makes an 80-core CPU, the Ampere Altra, which does not use hardware threading. In fact I am not sure if any ARM processor use hardware threading.

The Ampere is for use in data centers where security would be important.

Does the RISC vs CISC distinction still make sense?

Yes, despite what people are saying these are still fundamentally different philosophies. It may not matter much for high end chips as they have such high transistor count that the complexity in chopping up complex x86 instructions is dwarfed by everything else.

Yet these chips still look quite different and you approach them in a different manner.

Some RISC characteristics do not make as much sense anymore.

RISC instruction sets are not necessarily all that small anymore. Of course it depends a lot on how you count.

Looking at RISC-V may give a good sense of the difference. RISC-V is built upon the idea of being able to tailor make particular chips where you can choose which instruction set extensions you are using.

However there is still a core minimal instruction set, and this is very RISC like:

- Fixed size
- Instructions designed to use specific parts of CPU in a predictable manner for facilitate pipelining.
- Load/Store architecture. Most instructions operate on registers. Loading and storing to memory is generally done with specific instructions only for that purpose.
- Lots of registers to avoid frequent memory access.

CISC instruction in contrast are still variable length. People can argue that micro-ops are RISC like, but micro-code is an implementation detail very close to hardware.

One of the key ideas of RISC was to push a lot of heavy lifting over to the compiler. That is still the case. Micro-ops cannot be re-arranged by the compiler for optimal execution.

Time is more critical when running micro-ops than when compiling. It is an obvious advantage in making it possible for advance compiler to rearrange code rather than relying on precious silicon to do it.

While RISC processors have gotten more specialized instructions

over the years, e.g. for vector processing. They still lack the complexity of memory access modes that many CISC instructions have.

The sources for this article which I used to write it is collected in this previous medium article.

I would also point to the following sources:

- RISC vs. CISC Still Matters by Paul DeMone.
- Instruction Set Architecture.
- RISC
- Intel 8086
- On ARM performance by Xavier Tobin.
- RISC-V compressed instruction set format.
- Video presentation of how well RISC-V instruction compression works, by looking at benchmarks.
- Classic RISC Pipeline. Goes into more details in how RISC instruction sets are designed to work well with pipelines. What is done in each stage etc.
- Status register. Not a topic I discussed but interesting in learning more about tradeoffs for different RISC architectures and pipelining. Many RISC processors e.g. don't have status flags for arithmetic operations, only a general purpose one.