

What You Should Consider When Choosing a Processor IP Core

Roddy Urquhart



Designing microprocessors is
**hard, expensive, and
takes years.**

But does it have to?

No...

Codasip® was founded on a simple belief – that we could bring together the brilliance of microprocessor architects, RTL engineers, and software engineers and capture it in tools that made design simpler, faster, and less expensive.

Codasip Studio™ was born 2014 with the mission of automating processor design. At the same time RISC-V® was gaining traction, so it was natural for Codasip to embrace it as an Open Instruction Set Architecture (ISA).

After 50 years, Moore's law, Dennard Scaling, and Amdahl's law are failing. The semiconductor industry must change, and processor paradigms must change with it.

The Future is in

- Heterogeneous compute
- Customized solutions
- New leaders that disrupt

www.codasip.com

| 1 INTRODUCTION

Most integrated circuits include at least one processor core and makes use of embedded software. In the case of more complex systems-on-chip (SoC), there may be application processors running the main software, and operating system plus multiple specialised subsystems handling functions such as communications, security, and sensors. Requirements for processing vary considerably and there is a wide choice of processor IP cores available to choose from. Processor IP vendors often vie with each other by showing PPA numbers (performance, power, and area) but such indicators need context and can potentially be misleading. So, what do you need to consider when choosing a processor IP core?

In this article we investigate the following key considerations:

- Instruction set architecture (ISA)
- Performance
- Complexity
- Operating systems
- Subsystem silicon area
- Licensing model

| 2 INSTRUCTION SET ARCHITECTURE (ISA)

One of the most important considerations you face in choosing a processor is the ISA. Thirty years ago, many integrated device manufacturers had their own microcontroller architectures, each with their own ISAs. However, at that time the PC market was dominated by the X86 architecture which saw off all challengers and supported a huge range of application software.

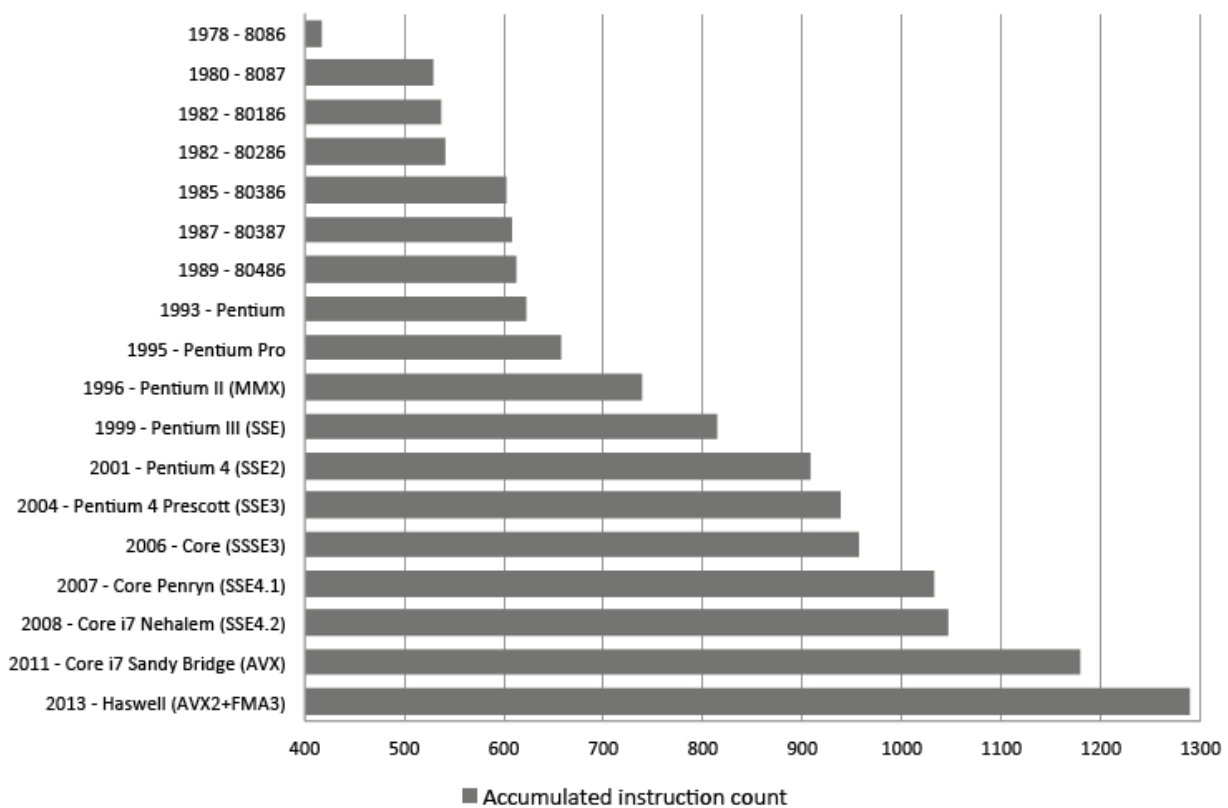
With the emergence of processor IP companies in the 1990s, more and more semiconductor companies adopted 3rd party processor IP with Arm becoming the dominant architecture for both mobile phones and embedded. This led to many companies developing products with the same processor cores as their competitors which limited the differentiation of their chip products. However, with widespread use of the Arm ISA, an impressive ecosystem of software developed.

For decades, Arm was regarded as being independent of semiconductor companies offering a sort of "Swiss neutrality". This neutrality was undermined by the acquisition by SoftBank which meant that Arm was no longer independent. This has been exacerbated by tensions between Arm and Arm China. Even if the acquisition by Nvidia does not go through, Arm has lost its neutral appearance by being potentially owned by a licensee. The MIPS architecture is already owned by Wave Computing. Many semiconductor companies will feel uncomfortable at the risk of depending on a key technology from a rival semiconductor company.

When choosing an ISA there are many things you will want to consider. Three of the most important include:

- a) Complexity
- b) Software ecosystem
- c) Openness and neutrality

Over the years, there have been both efforts to increase ISA complexity and to reduce it. A study by the University of Campinas showed that the X86 architecture grew more than threefold between 1986 and 2013. Similar growth has taken place with Arm's ISAs.



Intel x86 ISA Growth (1978-2013). Source ISA Aging: A Case Study, University of Campinas.

In the opposite direction there was the University of California Berkeley's RISC project which started in 1980 and used just 32 instructions but exploited pipelining and register windows to achieve performance. This concept was commercialised by Sun Microsystems as SPARC. In 1981 the Stanford University MIPS project developed a small ISA with a mix of 16-bit and 32-bit instructions.

More recently, in 2010 Krste Asanović of UC Berkeley started a "3-month project" to develop a clean slate ISA. This was motivated by the limitations of proprietary ISAs including:

- Protection by patents – **cost**
- Linkage between ISA ownership and processor designs – **single supplier**
- Complexity of many ISAs – **frequently not needed**
- ISAs can die with the company owning them – **risk over time**

This became the open RISC-V ISA standard.

A key characteristic of RISC-V is that it is not a one-size-fits-all ISA. It recognises that

different applications require different levels of ISA complexity. There is a very small base instruction set (just 47 instructions for 32-bit processors), then a range of optional standard extensions which can be used if needed. Lastly there is provision for creating non-standard custom instructions.



- Required for a given width
- Key to software portability
- Extend processor
- Only use necessary subsets
- Application-specific
- Focused & custom
- Variable length
- Only use necessary instructions

RISC-V Modular ISA. Source Cudasip.

This work © 2021 by Cudasip Group is licensed under CC BY-NC-SA 4.0.

Unlike proprietary commercial ISAs, RISC-V enables you to match the ISA complexity to the needs of your design. For example, a high-end Cortex-A processor from Arm will have over 800 instructions but are they all needed? With RISC-V you would use the base ISA plus precisely those extensions required such as [C], [G], and [P].

Availability of software is a key consideration for you to think about. Some software is only available on a limited number of ISAs – for example, data centre software is mainly available on X86, with Amazon putting software on Arm relatively recently. With embedded software, a lot of software is available in source code so can be ported to almost any ISA while some is available in binaries. For Arm there is a wide choice of software in binary form but for other proprietary ISAs there is a much smaller choice.

With RISC-V, if embedded software is compiled to the base instruction set or to a common combination of standard extensions, such as RV32IMC or RV64GC, it can readily be delivered in binary form regardless of the target processor. The RISC-V community is requiring the use of [G][C] extensions for Linux.

Finally, it is useful to think about the openness of an ISA. Proprietary architectures are usually linked with a single company, and are therefore dependent on the health and survivability of that company. If Arm and MIPS are owned by semiconductor companies, they are no longer perceived to be neutral by other companies developing integrated circuits.

The RISC-V open ISA is owned by the independent RISC-V International organisation in Switzerland and not by a single company. Thus, even if a company delivering RISC-V processor IP becomes insolvent, there are alternative suppliers of RISC-V cores that you can use. Investing in software targeting RISC-V cores is a low-risk choice for you to make.

| 3 PERFORMANCE

Looking at any processor IP, you will find that their vendors emphasise PPA (performance, power & area) numbers. In theory, these should provide you with a level playing field for comparing different processor IP cores, but in reality, the situation is more complex. Let's consider what 'performance' is.

The first thing to think about is what aspect of performance do you care about? Do you care more about the absolute throughput that you want (performance per second) or the performance per MHz? In an application such as machine vision, which is continuously running and requiring the use of complex algorithms, it is likely that you will care about the absolute throughput. However, if you have a wireless sensor node with a low duty cycle, when the node wakes up you will want it to be active for as few clock cycles as possible. This means you will care about how much computation you achieve per mJ energy.

About 40 years ago, computers were compared on the basis of MIPS (millions of instructions per second) although the problem is – what is an 'instruction'? Instructions vary considerably in complexity from one architecture to another, thus an operation will generally require fewer cycles in a CISC processor than a RISC one. MIPS were only helpful when comparing products with similar architectures and were called "meaningless indices of performance" by some!

Another thing to think about is the type of computation that you expect to care most about. Is it integer operations – and if so, which ones – or say floating point computations? In the past MFLOPS (million floating point operations per second) was a popular measure. But again, what is an 'operation'? And to what precision?

Today, synthetic benchmarks are widely used with processor IP cores. They have the following characteristics:

- a) They are relatively small and portable,
- b) They are representative of commonly used relevant applications,
- c) They are reproducible and transparent,
- d) They can be applied to a range of processors fairly,
- e) They express the benchmark result as a single number.

A benchmark that has been popular for the last 36 years is the **Dhrystone** benchmark. Its name is a play on words comparing it with the once-popular Whetstone benchmark. While Whetstone focused on floating point operations, Dhrystone focused on integer and string operations. The Dhrystone benchmark results are generally quoted as DMIPS (the Dhrystone score divided by that of a nominally 1MIPS machine). The benchmark has been criticised because modern compilers can optimize away parts of the work meaning that it partly tests compiler rather than processor performance.

Another popular synthetic benchmark for embedded applications since 2009 has been EEMBC's **CoreMark®** which aims to undertake operations that are representative of embedded integer processing needs. These include list processing, matrix operations, finite state machines and CRC.

A small group led by UCB's Dave Patterson has criticised all synthetic benchmarks and have instead proposed the Embench as an alternative using real programs. Time will tell if this gains mindshare in the industry.

As you can see, there are various benchmark systems out there, each suited for measuring a slightly different type of performance. So how do you assess performance when choosing processor IP for your project? If your embedded software has similar operations to a synthetic benchmark, then that benchmark may give you useful initial guidance quickly and simply. However, such benchmarks are normally quoted per MHz for example CoreMark/MHz. The per MHz figure is normally a good indication for a low power application where you are looking for good results per clock cycle. However, if you are looking for high absolute performance, this may be misleading. Instead, you should consider say the CoreMarks achievable at your target clock frequency.

If your main issue is floating point performance, bear in mind that DMIPS and CoreMark are integer benchmarks. You would be better comparing cores based on a floating-point benchmark such as Whetstone or LINPACK. However, Whetstone is infrequently used at present and is rather elderly.

Another pitfall to be aware of – especially with more complex cores – is that Dhrystone and CoreMark may be small enough to fit into the cache. If run from the cache, the performance information is rather meaningless if your real application is larger.

Ultimately, it is strongly recommended to invest the time in running realistic software on a processor core to assess whether the core gives you the performance you need. Even if synthetic benchmarks give you quick pointers, there is no substitute for using real application software. If you are looking at RISC-V, then profiling your software to understand where the computational bottlenecks are can also lead to assessing whether adding custom instructions can give you improvements in performance and code size.

| 4 COMPLEXITY

The more complex a processor core, the larger the area and the power consumption. But increasing complexity is not a single dimension as processors can be more complex in different ways. In selecting a processor IP core it is important to choose the right sort of complexity for your project.

Some ways you can think about complexity include:

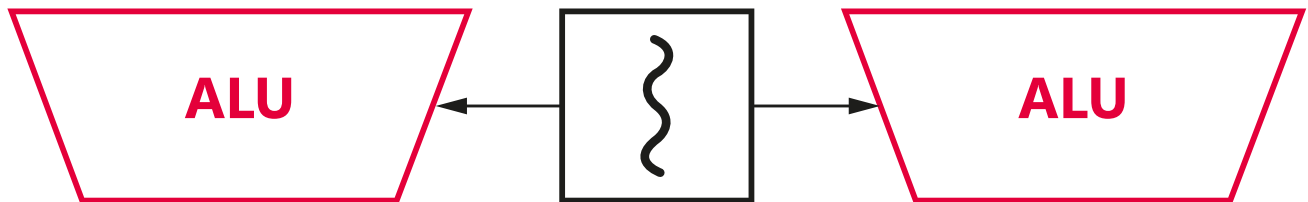
- Wordlength
- Execution units
- Privilege/protection
- Virtual memory
- Security features

Generally, the smaller the word length, the smaller the core and the lower the power, however this is not always the case. An 8-bit core such as 8051 is comparable in gate count to the smallest 32-bit cores, but the power consumption is usually worse. An 8-bit core requires more memory accesses and with less computation per clock cycle requires more cycles, and therefore more dynamic power to complete a computation.

Processor cores vary considerably in the complexity of their execution units. The simplest are basic single ALUs requiring many common operations to be implemented by simple instructions – for example using shift and add to implement a multiplication. It is therefore commonplace for cores to have a hardware multiplier and divider. In the event of you needing good floating-point performance, adding a hardware floating point unit will provide significantly better performance. This is available for CodaSip's L30F, L50F, H50F, and A70 RISC-V cores. With embedded cores, adding a floating point unit can roughly double the core area versus an equivalent integer core.

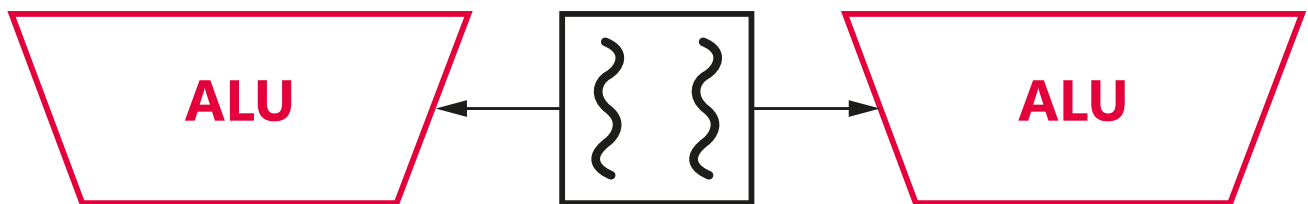
So far, we have assumed a single computational thread and scalar processing units which execute one instruction at a time. Superscalar architectures have instruction-level parallelism being able to fetch multiple instructions and despatch them

to different execution units to give you enhanced performance. For example, the Western Digital EH1 and EH2 SweRV Cores have two execution units. A dual-issue core processing one thread can theoretically have up to double the performance of a single-issue core.



Dual issue core. Source Codaip.

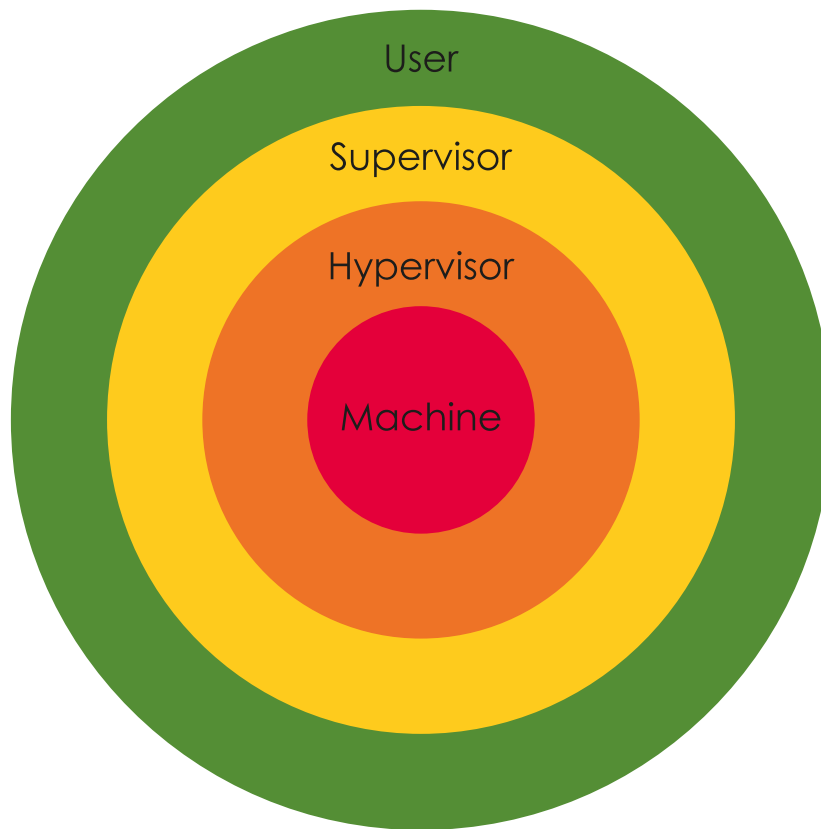
However, a thread can stall, making both execution units temporarily inactive. If there are two hardware threads (harts), if one thread stalls, the other can still continue execution.



Core with dual hardware threads. Source Codaip.

Another area of complexity for you to consider is privilege modes – generally the more modes, the more complex the core logic. Many embedded applications run in machine mode, which means that the code has full access to the core – like root privilege in Linux. Such code must be completely trusted to avoid negative consequences. In more sophisticated applications, a range of privileges such as machine, supervisor, and user may be offered.

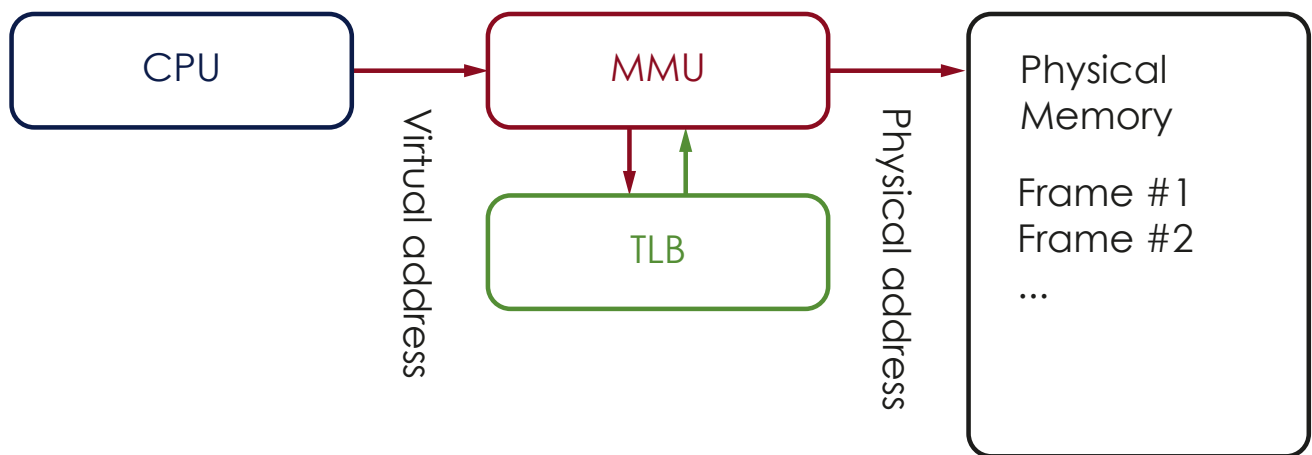
Normal applications will run in user mode with the greatest amount of protection and some software requiring greater privilege will use supervisor mode. Linux requires all three modes. The RISC-V specification also defines physical memory protection (PMP) which prevents less privileged modes from accessing certain memory spaces.



Privilege mode hierarchy. Source Cudasip.

This work © 2021 by Cudasip Group is licensed under CC BY-NC-SA 4.0.

Virtual memory also requires your processor to have additional resources such as a memory management unit (MMU) and translation lookaside buffer (TLB) to handle translating virtual memory addresses to physical addresses. This brings additional costs in terms of area and power dissipation without improving processor throughput. Nevertheless, virtual memory is necessary for using rich operating systems such as Linux which enable more complex software to be used.



Core with virtual memory. Source Cudasip.

Finally, hardware security features add logic. This may be in the form of providing encryption/decryption to memories, providing error correction coding and tamper detection blocks.

So, when choosing a processor core, work out what sort of execution units, memory management, privilege, and security you really need. That combination will determine the complexity of the core.

| 5 OPERATING SYSTEMS

For each embedded product, you need to consider whether you need an operating system; and if so, what type of an OS. Operating systems vary considerably, from real-time operating systems with a very small memory footprint to general-purpose rich OSes such as Linux with a rich set of features.

Choosing a proper type of operating system for your product – and consequently working out the required features of the embedded processor – depends significantly on whether you face a hard-real-time requirement. Safety-critical and industrial systems such as an anti-lock braking system or motor control will have hard maximum response times. At the other end of the spectrum, consumer systems such as audio or gaming devices may be able to tolerate buffering, if the average performance is adequate. Such systems are said to have soft real-time requirements.

A hard-real-time requirement can be achieved by writing so called bare-metal software that directly controls the underlying hardware. Bare-metal programming is typically utilized when the processor resources are very limited, the software is simple enough, and/or the real-time requirements are so tight that introduction of a further abstraction layer would complicate meeting these hard-real-time requirements. The disadvantage to this approach is that such bare-metal software needs to be written as a single task (plus interrupt routines), making it difficult for programmers to maintain the software as its complexity grows.

When dealing with more complex embedded software, it is often advantageous to employ a Real-Time Operating System (RTOS). It allows the programmer to split the embedded software into multiple threads whose execution is managed by the small, low overhead “kernel” of the RTOS. The use of the multi-threaded paradigm enables developers to create and maintain more complex software while still allowing for enough reactivity. RTOSes typically operate with a concept of “priority” assigned to individual threads. The RTOS can then “pre-empt” (temporarily halt) lower-priority threads in favour of those with higher priority, so that the required real-time constraints can be met. The use of an RTOS often becomes necessary when adopting complex libraries or protocol stacks (such as TCP/IP or Bluetooth) as this third-party software

normally consists of multiple threads already. Today there is a wide choice of open-source and commercially licensed RTOSes.

The embedded processor requirements of a simple RTOS, such as FreeRTOS or Zephyr, are truly modest. It is sufficient to have a RISC-V processor with just machine mode (M) and a timer peripheral. These RTOSes can therefore run on any of the CodaSip RISC-V cores or Western Digital SweRV Cores. However, rigorous software development is needed as machine mode offers unconstrained access to all memory and peripherals with associated risks. Extra protection is possible through a specialized RTOS such as those developed for functional safety, like SAFERTOS, or for security.

If a processor core supports both machine (M) and user (U) privilege modes and has physical memory protection (PMP), you can establish separation between trusted code (with unconstrained access) and other application code. With PMP, the trusted code sets up rules for each portion of the application code, saying which parts of memory (or peripherals) it is allowed to access. PMP can for instance be used to prevent third-party code from interfering with the data of the rest of the application, or to detect stack overflows. FreeRTOS does offer memory protection support for cores with PMP. Employing PMP therefore increases the safety and security of a system, but at the cost of additional hardware required for its support.

If your application requires a more advanced user interface, sophisticated I/O and networking, such as in set-top boxes or entertainment systems, an RTOS is likely to be too simplistic. The same applies if you need complex computations, requirements for a full process isolation and multitasking, filesystem & storage support, or a full separation of application code from hardware via device drivers. Systems like these generally have soft real-time requirements and can be best served if you use a general-purpose rich operating system such as Linux. As noted previously, Linux requires multiple RISC-V privilege modes – machine, supervisor, and user modes (M, S, U) – as well as a memory management unit (MMU) for virtual-to-physical address translation. Also, the memory footprint of such system is significantly larger compared to a simple RTOS.

Finally, if your embedded systems requires both hard real-time responses and features of a rich operating system like Linux, it is common to design them with two communicating processor subsystems, one supporting an RTOS and the other running Linux.

6 SUBSYSTEM SILICON AREA

The area of any part of a design contributes both to your silicon cost and to your power consumption. A simplistic following of the “A” in a processor IP vendor’s PPA numbers can be misleading. Your processor core is never in isolation but is part of a subsystem additionally including at least instruction memory, data memory and peripherals, and possibly caches. In most cases, instruction memory will be dominant and the processor area much less important.

The size of your instruction memory will be influenced by the target instruction set, the compiler and the compiler switches used. In the case of RISC-V, the choice of optional standard extensions and custom extensions can greatly influence the code size.

ISA	Core size (kgates)	Increase over base	Code size (kbytes)	Decrease over base
RV32I (base)	16.0	× 1.0	232	× 1.0
RV32IM	26.2	× 1.6	148	× 1.6
RV32IM+DSP	38.7	× 2.4	64	× 3.6

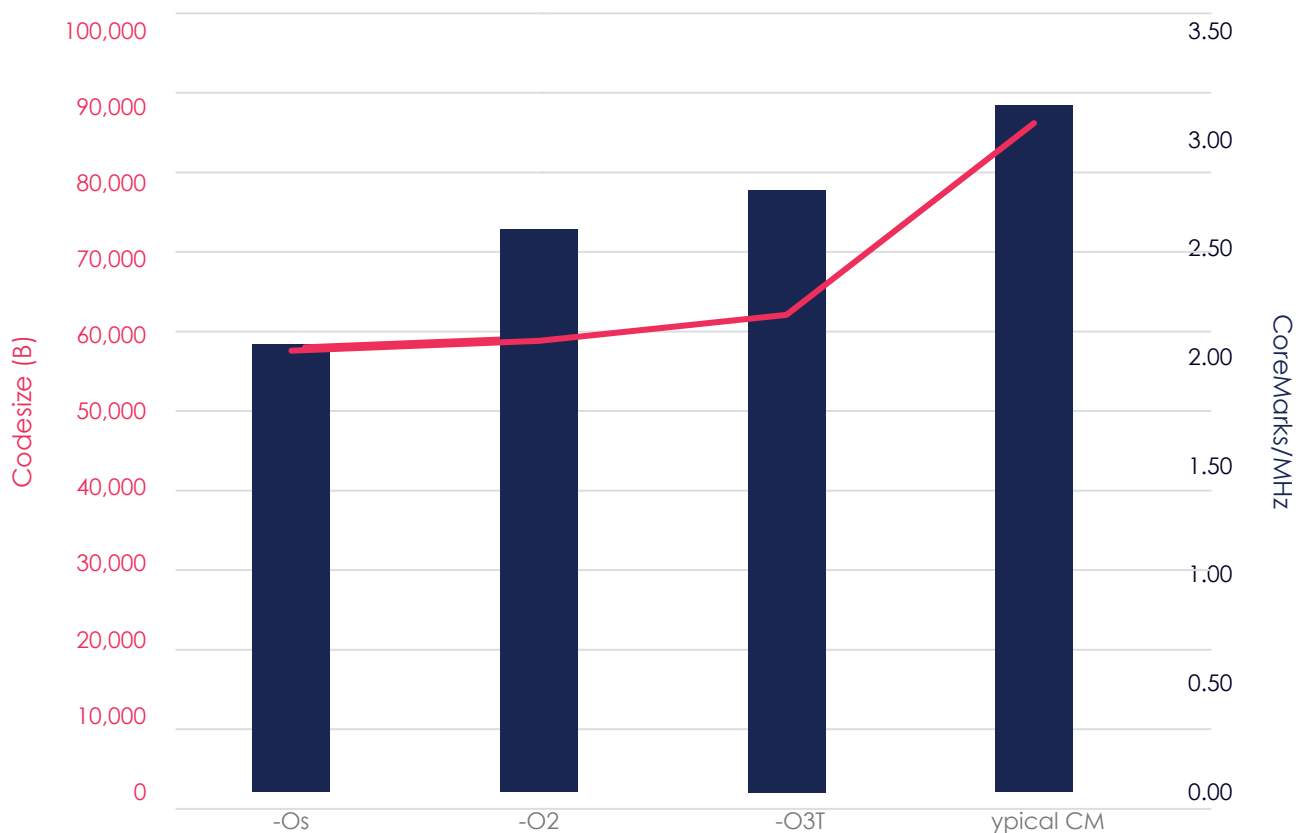
Source: Implementing RISC-V for IoT applications, Dan Ganousis & Vijay Subramaniam, Design Automation Conference 2017

To illustrate this, the table above shows the effect of adding extensions to both core and code size. In this example, Microsemi used a CodaSip RISC-V L30 core to implement an audio processing application. Starting with just the 32-bit base instruction set, they had an unacceptably high code size and cycle count. Some improvement was achieved by adding multiplication [M] extensions but the breakthrough was using custom DSP instructions. These led to a 3.6× reduction in code size at the price of a 2.4× increase in core size compared with the base core. With instruction memory dominating the area, this was a good trade-off; furthermore, the performance goals were readily achieved.

With typical vendor PPA data, synthetic benchmarks such as CoreMark/MHz are often quoted with a complex set of compiler switches. But in practice, embedded software is probably going to be compiled using common switches such as -Os or -O3.

Consider compiling the CoreMark benchmark using different switches in the common GCC compiler. In this case the target was a Codaip RV32IMC RISC-V core with a 3-stage pipeline. The chart below shows CoreMark/MHz and code size measures for different compiler settings. The last example is one that is typical of vendor performance data where many switches are used for CoreMark.

```
CM = "-O3 -flto -fno-common -funroll-loops -finline-functions -falign-  
functions=16 -falign-jumps=8 -falign-loops=8 -finline-limit=1000 -fno-  
if-conversion2 -fselective-scheduling -fno-tree-dominator-opts -fno-  
reg-struct-return -fno-rename-registers --param case-values-threshold=8  
-fno-crossjumping -freorder-blocks-and-partition -fno-tree-loop-if-convert  
-fno-tree-sink -fgcse-sm -fgcse-las -fno-strict-overflow"
```



Compiler switches aimed at improving performance may disproportionately impact code size and instruction memory area

In this example, the CoreMark/MHz score grows as the switches change from left to right. However, it is interesting to note that the most complex set of switches increases the code size by 40 % over '-O3' while the performance only improves by 14 %. Not every example will behave in this way but your choice of compiler switches influence both performance and code size.

Vendor benchmarks frequently make use of loop unrolling or inlining functions both of which have a detrimental effect on code size. Consider a simple loop with a multiply-accumulate.

```

for (i=0; i<8; i++) {
    sum += a[i] * b[i];
}

```

→

```

sum += a[0] * b[0];
sum += a[1] * b[1];
sum += a[2] * b[2];
sum += a[3] * b[3];
sum += a[4] * b[4];
sum += a[5] * b[5];
sum += a[6] * b[6];
sum += a[7] * b[7];

```

→

```

mul temp, a[0], b[0]
add sum, sum, temp
mul temp, a[1], b[1]
add sum, sum, temp
mul temp, a[2], b[2]
add sum, sum, temp
mul temp, a[3], b[3]
add sum, sum, temp
mul temp, a[4], b[4]
add sum, sum, temp
mul temp, a[5], b[5]
add sum, sum, temp
mul temp, a[6], b[6]
add sum, sum, temp
mul temp, a[7], b[7]
add sum, sum, temp

```

Illustration of loop unrolling showing the original loop, unrolled C and unrolled assembler

Unrolling the loop results in significantly more code especially when you look at the assembler. You would be very unlikely to want to compile your application in that way. Instead, you would almost certainly want to keep your code size down and avoid switches such as -funroll. Indeed, you might want to consider creating a multi-accumulate instruction to help reduce your code size and improve performance.

In choosing a core, it is important to be realistic about what compiler switches you would like to use and to ensure that the switches used for any performance benchmark data match those you would use for assessing code size.

| 7 LICENSING MODEL

Everybody is familiar with commercial licensing from traditional processor IP vendors such as Arm, Cadence, or Synopsys. But in discussing the RISC-V Open Instruction Set Architecture (ISA), there is widespread confusion of terminology, with RISC-V often being described as “open source”. Some have even accused vendors of commercial RISC-V IP such as Codaip or Andes as not being in the ‘spirit of RISC-V’. But what is reality?

Let's look at definitions briefly. An **open standard** like C, Verilog or HTTP is defined by a document that is maintained by an independent organisation. Thus, C is maintained by ISO, Verilog by IEEE, and HTTP by IETF. These organisations maintain the technical standards using a set of impartial rules. Such open standards are generally freely accessible.

With **open source**, the source code for a software package or the hardware description language source for a hardware block are made available using a license. Open-source licenses vary from restrictive ones, such as copyleft license, to permissive ones such as Apache. An open-source license defines rights for using, studying, modifying, and distributing the code. A copyleft license will require that any modifications be open-sourced while a permissive license will not.

The RISC-V is an **open standard**, and the ISA does not define any microarchitecture or business model. Therefore, a RISC-V microarchitecture can be licensed either as a **commercial IP** license, or as an **open-source** one. Nothing is prescribed.

If you think of a classic commercial processor IP license, you are generally paying for:

- The right to use the vendor's ISA,
- The right to use the vendor's microarchitecture,
- A warranty,
- Vendor commitment to fix errors,
- Indemnification.

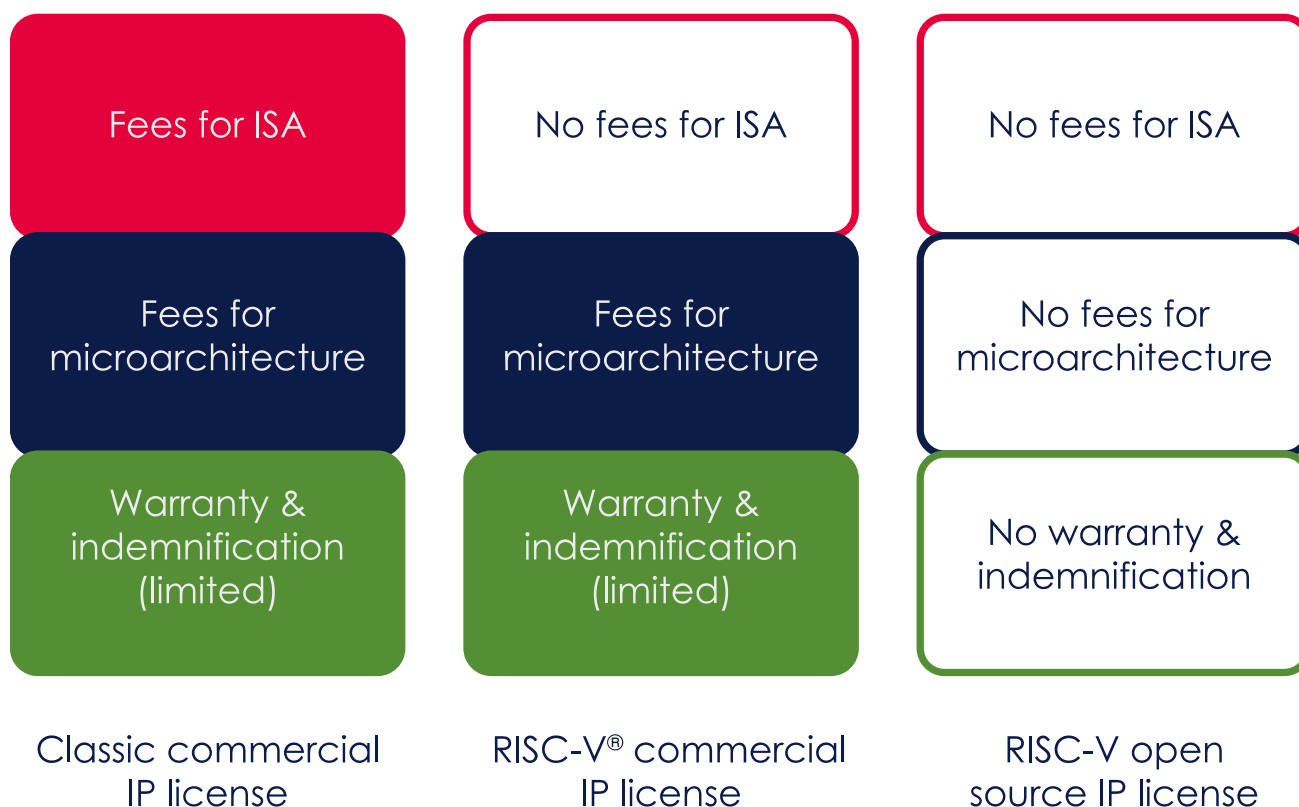
In practice, the warranty is usually time-bound, and the indemnification is limited.

However, for the licensee, the vendor has some commitments to fix a design if bugs are found, which is valuable particularly on a tight schedule. If a licensee is accused of patent infringement, intellectual property indemnification mean that the vendor will either defend the accusation or settle it on behalf of the licensee.

Classic IP vendors have jealously guarded their own ISAs as well as their microarchitectures. A normal license bundles the use of the ISA with the microarchitecture and there are no rights to modify the deliverables. Very rarely such vendors have offered an architectural license which has enabled the licensee to use the ISA with their own microarchitecture, but such licenses have commanded very large fees. One reason why RISC-V is very disruptive is that with a free and open ISA, one of the most valuable possible deliverables has no license fee.

Given that RISC-V does not prescribe microarchitecture or how it is licensed, there are both commercially licensed and open-source RISC-V IP cores. With an open-source license, you pay no license fee for the microarchitecture, but you also do not get all the benefits of a commercial license. Generally, deliverables have no warranty and are accepted "as is". Similarly, there is not the indemnification that exists with a commercial license. If bugs are found, then either the licensee or the open source community needs to fix them.

With a commercially licensed RISC-V cores, the only fees are associated with the microarchitecture as the RISC-V ISA is licensed free of charge. With this license, you get the warranty, indemnification and bug fixing commitments normally associated with a commercial license.



Alternative licensing models. Source Codaip.

This work © 2021 by Codaip Group is licensed under CC BY-NC-SA 4.0.

So, which is the right choice for RISC-V? Both commercial licenses and open-source licenses have advantages and disadvantages. You need to weigh up what is best for your design project.

At Codaip, we offer commercial RISC-V IP licenses. In the past, commercial and open-source licenses have been seen as bitter competitors. However, in the software world, companies such as Microsoft have embraced both models. Microsoft offers commercial licenses, supports open source projects, and has cloud-based business models. Codaip is convinced that commercial and open-source RISC-V licenses can co-exist and complement each other.

8 CONCLUSION

Choosing your processor cores is a complex task requiring a clear assessment of what your underlying needs are. Superficial assessment of measures such as vendor PPA data or benchmarks can be misleading and end up leading to sub-optimal choices of processor.

The starting point should be a clear understanding of what your SoC project needs. Would you like an open ISA or one controlled by a single company? Are you concerned about absolute throughput or computation per clock cycle? Do you face hard real-time requirements, or do you need a rich operating system? What computational units are needed to achieve your planned performance? What software do you need to run on your processor core and what combination of ISA and compiler will allow you to achieve the overall subsystem area you need? Lastly do you care more about getting a warranty, commitment to fix potential bugs and indemnification than avoiding a license fee?

Thinking through these issues will help guide you to making a good processor choice. If you need further advice, we'd be very happy to discuss your needs – please contact us here or e-mail us at contact@codasip.com.



Roddy Urquhart
Sr. Marketing Director

rurquhart@codasip.com
www.codasip.com

