Embedded processor architecture matters – for efficiency

Imsys AB
info@imsystech.com

Abstract *– Processor architecture development history, reasons why a different architecture is more efficient with energy, transistors, and memory*

Keywords *– Energy efficiency, Embedded systems, Internet of Things.*

I. INTRODUCTION

General purpose efficiency improvement through hardware architecture – is that possible?

You'd think it shouldn't be. You'd think everything has been tried before during the many decades of general purpose computer development.

But computing is nowadays very different from what it used to be – especially for embedded systems, which didn't exist when the traditional architectures were created.

It's time to do things differently. For many years there was no opportunity to do that. There is now, and the reasons for doing it are stronger than ever. Energy efficiency has recently become more important than performance. It is the limiting factor both for big servers and new battery-operated or energy harvesting devices.

II. WHY WE USE ANCIENT INSTRUCTION SETS

Development of new instruction set architectures was stifled already in the seventies due to all legacy assembly code that was dependent on existing instruction sets.

Then came the RISC era, which did away with that problem (RISC = Reduced Instruction Set Computer). But the first RISC architectures couldn't really be improved upon, if you would stick to the optimization principles of the RISC (and not add special execution units outside the basic core). The instruction sets of MIPS and ARM, from the early eighties, were therefore good enough and are still used, while newer RISC architectures – despite having been created by much bigger companies – became commercial failures. Many of them have disappeared altogether (e.g. Fairchild Clipper, AMD 29000, Motorola 88000, DEC Alpha, Intel i860, i960), while the old CISC (i.e. non-RISC) architectures survived.

A. CISC

CISC architectures survived, but there was no development of new CISC architectures during the years when many thought RISC would take over. Although the RISC era had reduced the importance of legacy code, no new important CISC – in the meaning "Complex Instruction Set Architecture Computer" – has been created since 1979 (Motorola 68000), despite the unfathomable development of IC density, cost, and performance, of languages and tools, and of the applications of computers as well as embedded systems.

Or, rather, no important instruction set hardware architecture has appeared. Several "virtual machines" have been created – some of these are extremely "CISC" if you see them as processor instruction sets – and this is a trend that continues. Code for such a VM is meant to be interpreted by an interpreter program, but special processors have been built for hardware execution of VM code, especially for the Java language (earlier also for the lower level Forth language). However, these processors have been more or less limited to their respective native language and have not become important.

B. RISC

In the early eighties, following the development of the Unix operating system and the C language (a hardware independent but rather low level language), computer scientists aimed at a new start, abandoning assembly level programming and at the same time achieving high performance for C-language computer programs of the type that was typical for office computers at the time. This was the RISC principle. The basic idea was this:

- Make each instruction specify at most one, single-cycle, operation, and only on internal register contents. (Note that multiplication was not included; like division it was performed by a sequence of instructions.)
- Make the width of memory, addresses, instructions, arithmetic logic, and data registers all the same (data was 32 bit integers, no bytes or fractions).
- Define the instructions such that they look alike and their processing can go through the same sequence of specialized phases (one of which is the execution of the arithmetic operation), like in a factory assembly line.
- Pipeline the hardware, with a station for each phase, such that one instruction per clock cycle enters the pipeline. Instructions for read/write in memory don't fit the pattern and pass through the stations as empty slots. One instruction is finished per cycle, provided there is no jump instruction, which invalidates any already fetched instructions that follow it in the program.
- Let an optimizing compiler take care of the creation of the massive object code of inefficiently coded elementary operations that this principle leads to.

This turned out to be a success, but it made computer scientists and development engineers busy ever since with problems caused by its disadvantages. The engineers added hardware multipliers outside the pipeline, also complete floating-point units if needed, and cache memory.

The need for cache for the RISC has steadily increased, because the logic-to-memory speed gap has increased by

more than an order of magnitude, due to IC technology development. How to handle that problem in an optimal way has become a major area of computer science.

This has probably not left much time for questioning the foundation for the RISC-type architecture.

C. DSP

A different kind of processor, the DSP (Digital Signal Processor), was created for embedded applications that needed digital signal processing, something that the RISC cannot do efficiently since the typical elementary DSP operation is multiplication of fractions rather than addition of integers. The pipelines are totally different, as are the cycle times, data formats, and requirements on storage and input/output.

Many modern products contain both a RISC and a DSP. Software for DSP usually consists mainly of relatively small algorithms, often standardized and written once and for all in optimized assembly code. RISC, on the other hand, is not made for assembly code and not efficient at DSP-type work, but is good as the target of an optimizing compiler, i.e. suitable for large general purpose high level language (including C language) programs.

When both are used in a system, the DSP does the performance critical part of the work, while the RISC runs most of the software. A system with two processors working together means increased hardware complexity and cost, but is here motivated by the differences in architecture between these processor types.

III. COMPUTERS: WHY RISC THREATENED TO REPLACE CISC, BUT THEN DIDN'T

CISC (Complex Instruction Set Computer) is an acronym inspired by RISC, but refers to older, traditional general purpose computer architectures, where different instructions can have different widths, do any number of elementary operations and memory accesses, and require different numbers of clock cycles to execute.

The complexity typically necessitates a microprogram in read-only memory, inside the processor, directly controlling its inner workings, i.e. mainly all the steps of the fetching and execution of each machine instruction of the compiled program. This is an additional level of programming, but it is very special programming, done once and for all by the processor design engineers, and this microprogram defines the instruction set architecture of the CISC.

In the mid eighties it became possible to put a RISC processor together with multiplier and small cache on one chip, and that made the Unix workstations a very cost-efficient alternative to the minicomputers, which therefore quickly disappeared. The actual reason for the demise of the minicomputers was, however, not their architecture – instead it was the bipolar IC technology, which was now replaced by CMOS with much higher integration, dramatically reducing the number of components. Nevertheless, it was seen as a big success for the RISC.

The personal computers had CISC architectures like the minicomputers, but they were not threatened immediately. They already used CMOS microprocessors, they were less expensive than the workstations, and their performance had not yet become important. And soon the CISC processors, notably the X86 of the IBM PC, could also fit cache memory on the processor chip (and with their better code density they didn't need quite as much).

Then with increased performance Apple and Microsoft introduced graphical user interface and "desktop publishing", and more performance was suddenly badly needed. The RISC developers began their fight for the desktop computer CPU socket, which many thought they would win. Few people (but Maurice Wilkes, legendary computer architect who pioneered microprogramming, was among those few) realized that the advantage of RISC had only been of transient nature.

The RISC lost the war in the end, when personal computers gradually replaced UNIX workstations, and Apple, in 2005, switched from PowerPC to the old CISC architecture X86.

RISC has suffered disappointments in the server segment too, during the last decade. Intel Itanium didn't manage to replace X86. Like the Sun SPARC, used in workstations and servers, it is now said to have an uncertain future. And sales are increasing of the archetype of CISC, the IBM mainframe family now called System z, binary compatible with S/360 – from 1964.

And in embedded systems, Motorola never could replace the CISC architecture 68K by the RISC PowerPC. Instead the 68K lives on, modernized, as Freescale Coldfire.

IV. EMBEDDED: DIFFERENT FROM COMPUTERS

Today we are, in embedded systems, usually not dependent on the high C-code performance that a RISC processor with an optimizing compiler can offer. In fact, the ARM processors in mobile phones often use a lower clock frequency than what is possible, which means that they have the inefficiency of the RISC principle without utilizing its advantage.

Most embedded systems use small microcontrollers that are neither RISC nor CISC – their instructions are not restricted by a deep pipeline, but the machines are simple enough to do without microcode. They will survive. They are ideal for applications where a small on-chip program is sufficient and where no extreme performance is needed.

But for products that communicate on networks, which usually means they have large programs in external memory, the architectures used in these controllers are too simple to be the best choice. Such more advanced embedded systems also often have some performance-critical special function, of a kind that neither a simple controller nor an architecture like that of a computer processor is well adapted for. This could be graphics, audio or video processing, data compression/decompression, or secure communication. Such

devices are becoming more and more common, and they often use a RISC for the complex software plus a coprocessor for the performance critical functions. The mobile phone is the first very high volume example, but other products will follow in the expected future "Internet of Things".

Many such products will be wireless and thus need to be energy efficient. The above-mentioned processor types all have serious shortcomings in such applications, for reasons that should now be clear.

RISC was developed for computers, and a typical embedded system is very different from a computer. Note that memory size may be on the order of a millionth, power consumption a thousandth, i.e. "very different" is an understatement. And embedded systems are many – billions of units sold per year – they shouldn't need to use an architecture developed for something else.

V. WHY WAS RISC CHOSEN IN SMARTPHONES?

Again, historical happenstance played a part. As CMOS silicon processing and synthesis tools improved in the first half of the nineties, ASIC (Application Specific Integrated Circuit) technology went from simple gate arrays to standard cell chips that offered higher complexity and flexibility. Suppliers of e.g. consumer electronics products could suddenly develop their own advanced IC components. New types of products, like the PDA (Personal Digital Assistant) could then be very much improved, without waiting for the traditional IC companies to offer suitable components.

The important thing here was that a standard cell ASIC could contain a processor core. The first, and for a long time the only, processor that was available as licensable IP core for such custom chip designs had a RISC architecture that had been designed and used by a British small computer company in the eighties, and was offered from the new company ARM through essentially all ASIC suppliers. Thus, this processor was selected, by default, by the British PDA maker Psion when they developed their EPOC32 operating system, which was later renamed Symbian. Ericsson, Nokia, and Motorola shortly thereafter chose this OS – and thereby this CPU architecture – when they formed a joint venture with Psion in order to strengthen their chances against Microsoft's plans for Windows-based smartphones.

ARM thus came to dominate in mobile phones, but that application has also dominated ARM's sales. Developers of set-top boxes and advanced games chose MIPS, which was used in computer workstations and also became available for IP licensing. ARM and MIPS are chosen in computer-like embedded systems, but they are usually augmented by "accelerator" logic blocks, or DSP coprocessors, when high performance for graphics and media processing is needed, and it is mainly the smaller RISC versions that are being chosen.

StrongARM/Xscale (which Intel sold to Marvell) and the bigger processor cores from ARM never became commercial successes, and where more computing power is needed than in smartphones they now face a strong competitor in Intel Atom, with the old CISC architecture X86.

There is a wide range of applications where no efficient solution has yet been well established

Now ARM is extending their line of processor cores downwards, hoping that their RISC architecture can compete with the "8-bit" or "16-bit" microcontrollers. You might think that this is a wiser strategy since most embedded applications don't need very high performance, and the old microcontroller architectures are not ideal for high-level language which is increasingly needed, e.g. when network communication is involved. But, on the other hand, an application that doesn't need performance shouldn't use an architecture that has serious inherent disadvantages (size, code size, energy consumption) caused by its speed optimization.

There is a wide gap in performance, and cost, between the simple 8/16-bit microcontrollers and X86-based systems, and many important applications could benefit from an architecture optimized for that wide mid-range. A different processor architecture is needed to fill that gap, and it is not RISC.

VI. WHAT'S WRONG WITH THE OLD ARCHITECTURES?

The old CISC instruction sets were created at a time when they needed to be suitable for assembly coding. They were at least partly made for a human programmer.

The RISC was a simplification of the instruction set architecture, keeping only simple instructions that would be useful for an optimizing C compiler, and having a strict (but inefficient) instruction format that allowed the machine to have the highest possible speed for the simplest operations (e.g. add or subtract) on 32-bit integers in registers.

Neither of this is what processors need to be good at in modern applications. Instead it is input/output, byte and bit handling for communication and graphics and compression/decompression, multiplication-accumulation of fractions for audio and video, and other special processing for cryptography, and for virtual machine interpretation for high level languages like Java.

All of these kinds of processing are awkward to do for a RISC, and this cannot be helped except by adding resources externally. The basic RISC core itself cannot be modified to be efficient for these modes of processing. It is simply too constrained by its defining characteristics, those that once gave it high Dhrystone benchmark results at low hardware cost.

Furthermore it is now desirable to achieve small size of compiled high-level language programs, and low power consumption. In a SoC (System on Chip), where processor core and program are on the same die, code density is especially important, since the program typically needs more silicon area than the processor. Both the simple controllers and the RISCs have bad code density, when compared to e.g.

the Java Virtual Machine. Note that bad code density also means high power consumption in the memory.

The RISC has poor utilization of energy and silicon, which is the price paid for its high frequency of simple operations. CMOS development steadily increases this disadvantage. The increasing speed gap between logic and memory has necessitated cache memory, sometimes in multiple levels, and in some cases also other added hardware based on speculation (i.e. the results of its work is not always used).

It should therefore be increasingly important to look for other ways of optimization, which consider the needs of modern applications and also the fact that the very high speed of CMOS today is often sufficient to achieve the needed performance without using cache and speculation.

VII. TRADITIONAL PERFORMANCE NO LONGER AS IMPORTANT

The incredible increase in logic speed of CMOS is at last moving the focus away from pure speed. We now have consumer electronic products with what used to be supercomputer performance.

Power consumption (for a given performance level) has also been dramatically reduced, but that has not in the same way made it less important – on the contrary; power consumption was not talked about in the eighties but is now of great and increasing importance. For computers this is because the increasing density and speed has made the power consumption per mm2 reach levels that make it difficult to remove the heat. For embedded applications the increasing efficiency of CMOS logic and memory has opened for a huge growth in wireless, battery-operated devices, these are becoming more and more sophisticated, and batteries are not improving fast enough.

High efficiency - for Dhrystone, or byte handling, or signal processing, or Java interpretation, or graphics and imaging – can be accomplished in straightforward ways by different kinds of hardware optimization. This has been done in different processor types. However, a processor combining these optimizations would not be an efficient general-purpose processor. Instead it would be one that is always using only part of its resources, i.e. it would be inefficient, at least with silicon area and leakage current.

VIII. HOW SHOULD AN EFFICIENT GENERAL PURPOSE EMBEDDED PROCESSOR BE DESIGNED?

To be efficient with silicon area – i.e. cost – most of the machine should be utilized for all processing, i.e. it should not have big parts that are seldom used. Its hardware resources must be able to execute efficiently also those tasks that are now often taken care of by added hardware (e.g. floating point, DSP, interpretation, graphics) were intended for. This is made possible by the increased speed of CMOS logic, but to be able to do it the hardware must be more flexible. The processor needs to be efficient both at DSP algorithms and at high level language execution by virtual machine interpretation, which are very different kinds of work.

Also, cache should be avoided if possible. Cache is very expensive, normally it uses more silicon than the entire logic core of the RISC processor, and it increases performance only for certain types of processing.

DSP processors also often use fast memory buffers bridging the speed gap, but they are different from the caches of RISC processors, Also, the data formats of the RISC and DSP are different, and these machines are optimized for different kinds of operations, requiring different number of logic levels, i.e. having different cycle time – the RISC and DSP designs simply cannot be combined into one processor.

The data path should not favor one data format in such a way that other common formats are inefficiently handled. This happens e.g. when operations have to be performed on 32bit words even though only one byte position is of interest, or if 32bit words have to be shifted in order to get a certain byte in a certain position.

In general, to be energy efficient the processor should avoid activity that is not necessary for the task at hand. This also means that it should not use a deep pipeline, since the energy spent on filling the pipeline is often thrown away because of jumps in the program sequence.

It should also avoid the overhead activity caused by a limited instruction repertoire. Note that a normal instruction set will not, even in a “hotspot” and although the execution resources exist in the machine, allow memory access and arithmetic operation and jump simultaneously, simply because there are no instructions that can specify such a combination of actions. The minimized machine that can fulfill the above requirements must be very flexible, i.e. it should have many degrees of freedom in the control of what its data path and sequence control logic can do – and this flexibility should not always be limited by a predefined and limited set of instructions, since that requires the execution of more cycles than those actually needed to get the work done. Thus, ideally it should be possible to fully control what the logic does in each and every cycle, independently of what it does in other cycles.

This requires a high instruction rate, as in RISC machines. Furthermore, the high flexibility requires wider instructions than in the RISC. Won’t that require even bigger and more expensive instruction cache?

No, only if the intention is to compile to this level of instructions. But it isn’t. As mentioned above, DSP algorithms are relatively few and small (compared to computer programs in general), and this is true also for special routines for graphics and cryptography. And also, in fact, for virtual machine interpretation.

IX. VIRTUAL MACHINES

High level language source code can be compiled to a very efficient (semantically dense) object code that the machine interprets. This is commonly done for Java, but can be done for virtually any language. This dense code is more adapted

to the compiler than to the hardware that executes it. Usually it is interpreted in real time by a software program – the interpreter reads each ”instruction” of the object code, executes it, reads the next one, etc. The interpreter, together with other software needed for this to work, is called a virtual machine (VM). For Java the ”instructions” produced by the compiler are called JVM bytecodes, because most of them consist of only one byte (even though their execution may involve multiple memory accesses and arithmetic operations).

Also, digital signal processing (DSP) is just as important as general C or Java code today. DSP algorithms are very seldom unique custom software sequences, instead they are standard routines that are usually assembly coded. The same is valid for cryptographic routines and graphics. Also many other performance-critical routines are standardized today, they are part of libraries that have been written once and for all and may be part of the language, like floating point operations, or of commonly used standard libraries, of functions like square root and trigonometrics.

Note that the virtual machine ”instructions”, e.g. JVM bytecodes, as well as common DSP algorithms and standard crypto and math functions etc. could be subroutines programmed in the above-mentioned wider instructions specifying the activity in each cycle, and that these subroutines could reside inside the processor itself, and thus the object code that is stored in main memory would consist of nothing but calls to these routines, and that these calls could be coded as efficiently as possible...

It is this technique that is called microprogramming, and it is typical of CISC machines, e.g. the old IBM S/360 family or Intel X86 or Motorola 68000. In a traditional CISC, however, the microprogram is entirely in read-only memory (ROM) and is only used for the execution of the instructions of an ancient CISC instruction set.

X. IMSYS DESIGN CHOICES

To achieve what CISC does well, and more of it, Imsys opted for a large microprogram, which is also partly writable and can therefore be continuously developed, like system software.

To avoid the CISC weakness of legacy native code dependence, Imsys decided on an instruction set architecture made for compilers rather than for assembly programmers. The Java VM bytecodes are part of the instruction set, which means that Java and many other higher level languages are very efficiently supported. It also contains additional opcodes that are needed for similar compilation of C and C++. Due to this, software compiled from C is just as compact as if it had been written in Java.

The RISC design on the other hand has its strength in its high frequency of elementary arithmetic operations. The Imsys processor achieves this for the microcode, using pipelined microinstructions. Furthermore the flexibility of the microinstructions have often allowed the microprogrammers at Imsys to interleave parallel sequences in the microprogram, increasing utilization and speed. Microcoding also saves many cycles by simply eliminating unproductive jumps and moves.

The Imsys processor – the opposite of a RISC in every respect – uses microcode for JVM bytecode instructions as well as for common DSP algorithms, data compression/decompression, graphics/imaging, crypto, and also for some peripheral functions that would normally require extra hardware, such as Ethernet MAC and LCD display refresh. It uses classical principles for processor architecture, but some aspects of the design are proprietary and patented.

Like the design of the hardware, microprogramming is done by the processor designers and requires special skills and tools. These are more advanced now than when the CISCs were designed and the microprogram of the Imsys processor is much bigger that those of the CISCs were.

The design choices lead to high efficiency, with cost (silicon area) and with energy consumption per unit of useful work done. This has been shown by independent benchmarks.

The processor is best suited for embedded systems where 8/16-bit microcontrollers are not sufficient, where the program size is at least hundreds of KB and memory of at least a couple of MB is needed.

DSP performance, if not extreme, further strengthens its case against RISC, since it can handle that without the help of a coprocessor. The high I/O throughput is also important in some applications.

Thus, it should be a good choice for applications that fall within the wide gap in cost and performance mentioned above.

2010-05-26 Imsys AB