By Eric Verhulst,
Founder and CEO,
Eonic Systems NV

Proposal for a System-on-a-Chip Model

With RISC architectures no longer being as reduced as they used to be, CISC being build on top of RISC principles, DSP embracing RISC as well as VLIW principles, we see DSP capability being added to almost any architecture. Embedded applications must cope with shortening times to market, reliability, increasing complexity and flexibility to change the runtime algorithms. Taking a software point of view based on multi-tasking and resulting in very efficient but modular asynchronous architectures can solve these issues.

ENGINEERING: THE ART OF BUILDING BLACK BOXES

Our world is full of systems and objects that for the part of the world that uses these, are often "black boxes". They serve a purpose or execute a certain function while the user doesn't need to know how they work. From the economical point of view, the black boxes bear a cost in relation to their purpose and the manner of how they provide that purpose. That's where the engineer comes in. He was educated and hired to design and build "better", read more economical black boxes. The concept of a black box in real life is wide, covering from bridges over machines to "abstract" pieces of software. In fact, the range is too wide for any engineer to be able to master them all. So engineers get a domain specific education and he becomes a specialist. That specialization is also a curse, as it locks the engineer in his field and in "how things have always been done". Specialization is marvelous for improving existing systems. It is a hindrance if a given "system architecture" has been exhausted and only a radical new architecture can produce a major step forward. We will try to illustrate this with the history of (digital) Computing, an engineering field that, based on the von Neumann architecture, is still largely in place.

THE VON NEUMANN ARCHITECTURE

The first digital computers were, just like their analogue counter parts, custom wired for each use. This not being very flexible, von Neumann conceived the idea to change the architecture by adding "memory". The memory held on the one hand a "program", a sequence of instructions that at every step changed the hardware to execute a particular function while on the other hand it stored "data", the input or the results of the program. Central to this architecture was a clocked "sequencer" that automatically incremented the "program counter" to follow the logic of the program instructions. To this date, all processor architectures still follow this paradigm. Processor performance being measured by how many instructions one can execute per time unit; processor engineers have continuously found tricks to improve that figure. A first fundamental one is to increase the clock speed. This worked quite well. On average, we have witnessed a speed increase every 18 months. It is a matter of mostly making the circuits smaller and making the electrons move faster. That is the specialization field of semiconductor engineers. At the next higher level, improvements were possible as well. One of the fundamental bottlenecks in the von Neumann machine is the transport delay between the memory and the actual processor. Processor architectures found numerous tricks that while not fundamentally changing the concept, allowed to increase the performance, at least if the program fitted the model:

- Complex instructions: by adding more hardware circuits, one can execute the equivalent of several simpler instructions in one step. The drawback of the approach is that often a lot of the circuits will not be used, while the execution delays limit the potential speed-up.
- Simpler instructions: by carefully analyzing programs, engineers found that most often the same type of fundamental operations was needed. Often fairly simple, one can then implement them in small, optimized circuits, allowing an increase of the clock frequency. Called "RISC" (Reduced Instruction Set Computers), this architecture has the drawback that often the memory has trouble keeping up with the processor. Hence, large caches (often complex circuits) and large register banks were introduced.
- Breaking up the instructions into a number of smaller internal ones that can be pipelined. This allows to "feed" the processor with the next instruction while the previous one is still being processed. However as we use higher level language to program, this makes the job of writing a compiler harder. One must take care of inter-instruction data dependencies and the pipelines can "stall" if a branch instruction is used.
- Often, combined with the RISC idea, the processor was redesigned to allow to execute multiple instructions at the same time. The ultimate in that approach is called a "VLIW" (Very Large Instruction Word) processor and as the state of the art we can consider some of the multi-media processors or the latest C62 DSP from Texas Instruments. The basic architecture suffers from the same problem as the pipelined one.

While I deliberately simplified the matters above, one can see a general trend: micro-parallelism or an architecture where several things can happen in parallel at the instruction level. This works best if all instructions are mostly independent from each other and produce no side effects. This is called orthogonality. The C62

Copyright 1999 by Real-Time Magazine 99-2 (http://www.realtime-info.com)

can be considered as a design that applied this architecture with good result: running at 200 MHz, the processor can deliver a peak performance of 1600 Mips, albeit only for a short while. Thanks to the small orthogonal instruction set, the circuits are fairly simple, the power consumption is modest and the compiler can generate fairly optimal code. Nevertheless, the von Neumann architecture, as also demonstrated on other advanced processors, like Digital's Alpha is reaching its limits. We are at a point where the memory can no longer keep up with the processor. While it is generally claimed that the technology improvements were mainly used to increase the memory density and not the speed, the fundamental aspect that memory is organized in rows and columns and often is external to the processor, has not changed much. First level and secondary level caches themselves operating with wait states are expensive Band-Aids. They reduce the problem on a statistical basis, but not on a fundamental basis. While the desktop market might tolerate the situation, the embedded market, much more driven by cost and power constraints, and with the requirement for predictable real-time behavior, needs a better solution. Most processor designers tried to alleviate the problem by adding faster internal memory. However, the current technology will probably never put enough there. Is there a way out?

THE SOFTWARE GAP

If we look at the successive processor architectures engineers have come up with, we can see a clear bias. Most improvements are at the circuit level. This bottom-up approach stems from the history. Hardware was needed to execute programs, so hardware improvements have always given a better performance. As this is already a difficult job, specialization was paramount, often leaving software in the background. Another reason is that processors are still perceived as machines that can execute a particular calculation very fast, hence the emphasis on executing a particular stream of instructions very fast, e.g. scientific computation. Hence, a lot of engineering work on the software side, at least in terms of processor architecture, is focused on building compilers that translate high level language programs in fast executing streams of instructions. Is that the purpose and the solution to building "black boxes"? While some engineers worked on the software black boxes, building object oriented programming systems, they have concentrated on the programming methodology on processors as they were, seldom looking at better performance as a goal.

Therefore, we have to take a step back and ask the fundamental question, what is software all about? Initially, we have to agree, software was first of all about programming the machine and programming languages evolved from simple assembler level into high end object oriented ones. Today, software is or should be about taking a real-world problem, writing a program that models it and, eventually, executing it one or multiple processors. If engineering is about building black boxes that execute a certain function, we shouldn't need to know about how the internals of the black boxes operate. Hence, a software engineer should be able to concentrate on his modeling task, rather than on how the black box will execute the model. Of course, at some point that might be helpful to improve the performance but only in a second step. The trick to make it happen is to design the hardware to execute or support the software model, cleaned from all historical hardware references better. Can it be done?

Keeping the system in balance: Step 1: parallel processing

Engineers have also come up with another improvement to increase the performance: macro parallelism. Two good reasons for it. Firstly, by putting more processors on the program, it should be possible to get a better performance. Secondly, at the expense of more "real estate", one can achieve a better balance between the processor and memory speed. This was and still is a good idea. It has however been largely been superseded by the new processor architectures. While originally processors could share memory in a balanced way because each of them executed the instructions in 3 to 4 cycles, current processors run at a much higher frequency and execute several instructions in parallel in a single cycle. Even then, this approach became very quickly inefficient if more processors are added. In addition, this ignores a more fundamental problem. How can a program be partitioned to run in parallel? How can we assure that all the parallel program parts keep the memory in a consistent state? One can think of a bag of tricks or Band-Aids to solve it, often in a program dependent way. These tricks not only make the programming a lot more complex, they also add considerably to the complexity of the hardware, thereby ignoring one of the original goals: more performance for less.

Therefore, another way of parallel processing is to develop architectures that distribute the memory. Each processor having its own local memory can run at maximum speed, and is connected with other processors using dedicated communication links. Two problems remain: the first again is how to split the program in parallel parts and secondly, processor now replaces the processor to memory latency to processor communication latency. In addition, the communication, being foreign to the semantics of a processor architecture, must be dealt with as well, including routing functions using dedicated I/O hardware. To improve the performance, most designers opt for DMA and crossbar switches to use with the communication links.

Keeping the system in balance: Step 2: multi-tasking

As one can now see, improving the system's performance is a question of keeping the system balanced. Today's processors run so fast that the real bottleneck is I/O in all of its manifestations:

- I/O between the processor and its memory;
- I/O between communication processors;
- and for embedded systems: I/O for data input or output.

In all cases, I/O can be separated in three main parts: set-up, data-transfer and transfer termination.

Copyright 1999 by Real-Time Magazine 99-2 (http://www.realtime-info.com)

The solution is as elegant and as simple as the one used for improving the performance of the processor itself: increase the parallelism. At the processor level, we have found micro-parallelism. Now that we start using processors as components, we need more macro-parallelism. Just as micro-parallelism increased the performance by having overlapping instructions, but increasing the (relative) latency, macro-parallelism does the same. Having several executing application threads running in parallel allows to share the processor while the communication is happening. The question is now how we can best introduce this multitasking approach. (I personally prefer to use the term multitasking rather than multi-threading because of connotations with a single memory space.) One thing should have become clear: programming methods that take the sequential von Neumann machine as the starting point and then try to use the same approach to program inherently parallel machines are essentially trying to achieve an unnatural match. In the history of computing this has resulted in significant side-tracks, such as parallelizing FORTRAN compilers, a case of reverse engineering; cache coherency in shared memory, a problem that is purely the result of the choice to use shared memory in the first place and the myth of parallel algorithms.

BACK TO BASICS: A COHERENT PROGRAMMING APPROACH

As outlined above, computing systems are essentially black boxes that execute a model of the real-world. Many real-world systems are naturally modeled by a hierarchical decomposition of interacting black boxes. In order to execute the model as a computer program, programming languages pose another problem as well. Evolved from machine instructions, onto which successive layers of abstractions were added to simplify the programming, they nevertheless still reflect the sequential operation of the underlying von Neumann machine. What is really needed is an approach that obeys following boundary conditions:

- It must be a natural programming model;
- It must cope with the problems of micro-parallelism;
- It must cope with the problems of macro-parallelism;
- It must be a universal solution for having well balanced systems where processor bandwidth is matched with the I/O bandwidth.

The model that we propose exists and was initially applied to macro-parallelism first. In fact, under the guise of real-time kernels, industry has been using it for years albeit in conjunction with a scheduler, with the goal to obtain a predictable real-time behavior rather than a coherent programming approach. The theoretical foundations that turned this approach into a well founded programming paradigm has been laid by several authors, but by C.A.R. Hoare in particular with his CSP (Communicating Sequential Processes). I will shortly explain its principles and illustrate how a practical implementation was obtained in the Virtuoso RTOS. I will then speculate how the same principles can be applied to micro-parallelism.

CSP

A CSP program is defined as a set of processes. Each process is a function or rather a set of one or more sequential instructions operating on a local workspace and living in principle forever. Processes interact exclusively through communication channels. This also means that any local variable is only visible inside the scope of the process and not outside of it. The channels are fully synchronizing, which means that two communicating processes must both reach the communication point before they can proceed further. If not they remain blocked until that other process reaches the communication point. I will illustrate CSP with some occam program segments. Occam was the first programming language that implemented the CSP principles.

E.g. following code shows two processes in parallel that after initializing the variables exchange them and then multiplies the result by 2:

```
PROC P1, P2 :
CHAN OF INT c1, c2 :
PAR
  P1
  P2
PROC P1
  INT a :
  a := 1
  c1 ! a     // put a on channel c1
PROC P2
  INT B
  c1 ? b     // read from channel b and
  assign the value to b
  b := b*2    // multiply by 2
```

This program is a “parallel” version of b:=a.
We could have it written as:

```
PAR
  SEQ
    INT a :
    a := 1
    c1 ! a
  SEQ
    INT b :
    c1 ? b
    b := b*2
```

Or even shorter (at least functionally equivalent):

```
  SEQ
    INT a, b :
    b := a*2
```

Or graphically:

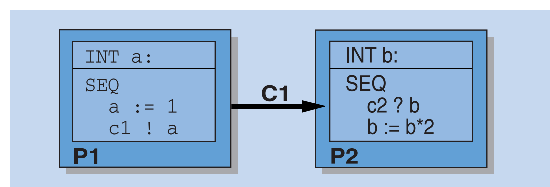

Figure 1.

Some points to remark:

- The order of writing the code for all statements in the scope of a PAR construct is irrelevant. Like in dataflow, the processes become activate when the data is available. The order of execution and hence the real-time behavior is not defined.
- The data is fully protected inside a process. Hence,

Copyright 1999 by Real-Time Magazine 99-2 (http://www.realtime-info.com)

no accidental overwriting of data will occur as the code can be checked at compile time for absence of scope violations.

- If a channel is mapped onto a HW device (e.g. a communication link), the logical behavior of the processes and of the program will remain the same.
- The sequential assignment and logically equivalent program is to be regarded as an optimized version that will only execute on a single processor with local memory.

How can this program now be implemented? For that we need to create a list that keeps track of the processes and their status and a list of the active channels. Whenever an executing process reaches a communication point, synchronization is verified and if needed, the status of the waiting process is updated. At this point, it should be clear that we also need something like a scheduler or kernel that controls the execution of the processes. This also raises the question on how the context (the status) of a process is preserved across a channel operation. As each process is sequential, the context can be small. In effect the only information that needs to be saved across the channel operation are the variables that are communicated. For this reason, occam used round-robin scheduling. This was sufficient for the simple semantics, but as pointed out not enough for predictable real-time behavior. From another point of view, one can look upon a process as a virtual processor that shares the CPU resources with other processes thanks to the scheduler that conserves its state. This is elegant, because it conserves the original model, each process being a little von Neumann processor. In any case, we have pointed out some features that the original von Neumann architecture never supported:

- the concept of a context;
- the concept of communication.

Following the experience gained with a more general purpose model used in the Virtuoso RTOS, we will try to indicate how from the optimized software architecture, we can get some hints on how these extra functionalities can better be supported in hardware.

THE VIRTUOSO PROGRAMMING SYSTEM

Virtuoso's Virtual Single Processor model

When occam came onto the world, and when we looked at it, we found its limitations to cumbersome for industrial use. Real-world programming often requires a higher level of abstraction and more subtle ways of expressing what a real-world application is supposed to be doing. Simultaneously, using occam for programming parallel systems was tedious because of following main reasons:

- communication was by lack of a routing layer strictly local on each processor. Processes on different processors could communicate through an explicit mapping of a software channel on a hardware communication link. This made any topology change very hard because the routing had to be done at the application level.
- communication channels carry strict bitstreams. Hence, any higher level protocol had to be provided by the application. In conjunction with the explosion in channel names, this made programming error prone, especially in terms of deadlocks

Hence, at Eonic Systems, we decided that we needed:

- a richer, more industrially accepted programming language;
- we needed pointers in the language for performance reasons, hence C;
- we needed pre-emptive scheduling for predictability reasons.

The latter was partially implemented in the original occam but restricted to two priority levels because of hardware implementation limits, which is often only good enough for soft real-time systems. In addition, the communication over the links is FIFO-ed so that all prioritization is lost as soon as the program communicates off-chip. The result of these requirements was the Virtual Single Processor (VSP) model, implemented in the Virtuoso RTOS. It main characteristics are as follows:

- the unit of execution is a task with a full context;
- all services have the same logical behavior whether called locally (on the same processor) or whether from another processor;
- the unit of distribution is a “kernel object” (task, semaphores, mailboxes, queues, resources, etc.);
- the scheduling is locally pre-emptive, in order of priority;
- the communication is prioritized, using packet switching (the latter is a trade-off between maximum throughput and predictable communication latency).

Initially we selected to port an existing microcontroller kernel. However, when we started to implement the topology independence, we discovered fundamental problems with the semantics of the services. E.g. using binary semaphores is not side-effect free or messages cannot pass pointers. The same applies for most RTOS on the market because they were designed to operate on local or shared memory architectures. Therefore, we redefined the semantics so that the logical behavior would be the same whether a service required the cooperation of local only or one or more distributed kernel objects. The overall result is that Virtuoso programs can now be moved easily from single processor to large parallel processing systems (and vice versa), without any change in the source code. This means that user has never to program the communication explicitly, unless he wants to for optimization reasons. Due to the prioritized packet switching the hard real-time characteristics are also, within the boundaries of the unavoidable communication delays, conserved when going parallel. Many distributed real-time systems exhibit a FIFO based communication mechanism and that is for hard real-time not acceptable.

Copyright 1999 by Real-Time Magazine 99-2 (http://www.realtime-info.com)

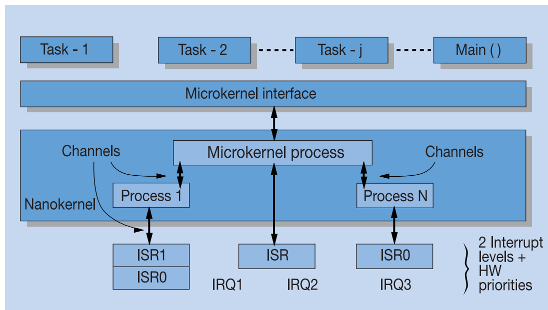

Figure 2. Virtuoso 4.0 generic architecture

Virtuoso internal architecture

While we outlined the principle of operation above, it would have been unusable if the internal architecture had not been optimized. After several iterations, we come up with following four levels, that I will illustrate with the Virtuoso 4.0. architecture. Details of these levels can be found in the references.

Task level

This level is also called the microkernel level, because tasks are pre-emptively scheduled by the microkernel.

The process level

Processes are managed by the nanokernel. While the microkernel itself is a nanokernel process, the other nanokernel processes are most often system level drivers.

The ISR1 level

ISR1 is the name given in Virtuoso to the interrupt service level whereby interrupts are globally enabled while simultaneously interrupt handling can be prioritized and even nested.

The ISR0 level

This is the default level where interrupts are handled by the hardware. As outlined above, this can have a serious impact on the interrupt latency.

Functional requirements mapping

The levels as defined above were selected and have a functionality that corresponds with the real needs of an application. This can be illustrated by following a dataprocessing flow as found in DSP applications. At ISR0, the data will be acquired and the hardware will be acknowledged (if needed). Often only 2 or 3 registers are needed. The shorter the interrupt service routine, the more interrupts per seconds can be processed and the higher the potential datarate, unless extra hardware provides buffering. If no extra hardware is present, this can be done in software at e.g. the ISR1 level. If the interrupt is low enough, one can pass the buffering to the higher nanokernel or microkernel level. At ISR1, this requires a few more registers to be saved, while at nanokernel and microkernel level, a full process or task context will be swapped. Note that the minimum level needed will be determined not by the datarate, but by a combination of the interrupt rate, the number of samples read at each interrupt and by the datasize that is being processed. Therefore, e.g. higher level processing like FFT can often be done at the task level with no serious overhead as each FFT takes between 100 to a 1000 microseconds depending on the processor.

Mapping the Virtuoso architecture onto hardware support

As already outlined, the INMOS transputer was originally build to execute occam programs and as such had hardware support for round-robin scheduling processes and link communication. The instruction set included instructions to support these features directly. E.g. creating a process was a single instruction and except the address, writing or reading from a software channel looked the same as reading or writing to a hardware link and required a single instruction. The context at all times was very small (3 to 4 registers for fixed point) to reduce the context switching overhead. No pipelined instructions were used. As a result, context switching was fast (typically one microsecond at 25 MHz). The underlying idea behind this architecture was simple: by making processes lightweight, they can be small and numerous, hence more overlapping would occur between processing and communication, increasing the system's efficiency. From a functional point of view, the Virtuoso software architecture is very similar to the transputer architecture, albeit we arrived at it from a different point of view. We wanted first of all to achieve low latency communication and ended with the nanokernel architecture. The performance obtained in software on standard processors and DSP is similar as on the transputer, but at the expensive of a much larger development effort. This is because the native register sets are much larger, instructions can be pipelined, links have to be programmed at a low level and must be made to work together with the DMA engines. Very often, this results in the detection of small silicon bugs, that while they have little impact on the performance, they have a heavy impact on the debugging cycle. Can the Virtuoso architecture be supported in the hardware with no impact on the performance gained from the microparallel architectures? We believe the answer is positive if the four level model is followed.

The essential to see is that multi-tasking is a must, unless for pure synchronous dataflow applications. It is a must as it provides the essential benefits from the object oriented point of view: modularity, abstraction, information hiding and as shown, it can be used as a building block to program parallel systems. It is a must to hide the communication latency. On the other we could clearly see the need to separate the processing from the I/O handling with the separation between the levels being characterized by the register context. Hence, why not increase the hardware parallelism again and create a level of nano-parallellism? At the same time, we can clearly see that low level interrupt programming even needs less context and is in fact totally out of sync with the background processing? This approach might also solve the memory access latency discussed in the beginning of this paper: multitask to mask the I/O latencies.

Therefore the following ideas are proposed to the processor architecture designers. The idea is to design a processor as a set of cooperating, asynchronously operating CPUs, each dedicated to a specific function. This makes sense if we see how much effort is needed in software to reduce the overhead of having

the main and only CPU take care of all. We can distinguish following modules:

- main CPU, optimized for computational through-put processing;
- interrupt engine, optimized for low latency I/O;
- memory management, optimized to move data and protect memory;
- communication links, optimized for fast background communication;
- a scheduler, that is probably more efficient to implement in software on each of the CPUs as the instruction can be tuned for it;

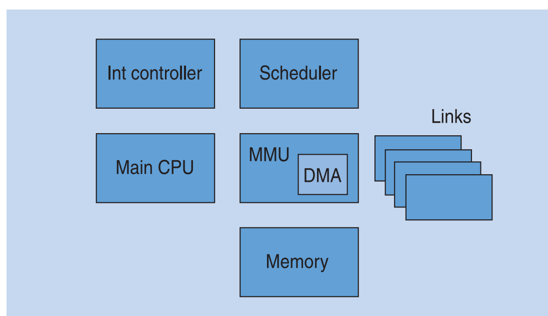

Figure 3. New Processor Architecture

- in larger systems, one can also expect to find reprogrammable logic and several DSP or RISC ALU's as front or end processors for the main processor.

Without claiming to have a fully finished concept, we outline some of the ideas behind it:

- Separate interrupt handling from the actual processing. This can be done by even adding a small separate CPU (but a subset of the main one) on the processor that operates with a simple instruction set, can only address a very small memory (a few Kbytes should be enough), has a small register set that can be automatically saved to memory in a single instruction and communicates with the main CPU by sharing some of its memory. The small register set also means that interrupt handlers can be written in a high level language. For performance reasons this means that we can have very high interrupt rates, while the main processor keeps running at full speed, not being interrupted. This also means that interrupt latency is no longer affected by the main CPU pipeline latencies.
- Separate the memory management from the actual processing. Note that the same mechanism as for the interrupt handling can be used. E.g. to buffer the DMA driven access to external memory. The mechanism can be used to include memory protection and management as well, e.g. paging and caching, essential features for more general purpose and safety critical types of processing. Note however that while small, above CPUs (or should be call them ALUs?) should have the same word length registers as the main CPU. It would be best if these units had there own separate instructions and we had more than one of them.
- Communication links combine the functionalities of above mentioned units. An often-neglected design issue is reliability and the capability to recover from errors. E.g. often links are designed as a distributed state machine that is running on the connected two processors. However to provide robustness, we need to be able to detect transient and permanent communication failures, isolate these errors and recover from them. As an example of an elegant and simple design that implements these features, we refer to the recently adopted IEEE-1355 link standard.
- Allow for nesting and prioritization of the interrupt handlers (see previous section).
- Separate the register sets hardware wise in two sets, while each set can be saved automatically to memory in a single cycle. The first set is together with an orthogonal instruction set optimized for control instructions and list manipulation as often used in kernel software or the higher level state machine of the application. The second set is optimized for compute intensive operations, e.g. indexing arrays, multiply-accumulates, etc. While we could adopt the hardware approach of the INMOS transputer, fixing higher levels functionalities to rigidly in hardware has its drawbacks as can be seen from the difficulty we often had in programming around the "nice to have" hardware features processor manufacturers so generously provided. Orthogonality is the key.
- As we have now a very asynchronous architecture, composed of multiple functional units, we can not only run these at different frequencies (probably multiples of the same base frequency) to reduce power consumption, but we also need a mechanism that efficiently supports the (de)scheduling. This is probably the most difficult part. What we essentially need is a mechanism that is hidden but like "test and branch" instructions allows to deschedule very fast whenever an operation can be delayed for a relatively long time. E.g. when a DMA transfer is set up the completion of it can take several 100 of cycles. What is desired is that the program segment is automatically rescheduled when the DMA transfer terminates. In current CPUs this is handled through interrupts, and unless one accepts polling (!?), this is tedious to program. However this can work. In the Tera supercomputer, the CPU was even designed to switch context on every cycle, with as the goal to mask the memory latencies. The question is whether that is not taking it too far while it ignores the need for pre-emptive scheduling.
- This architecture implies that the high level language compiler should actually combine several code generators if we had a classical monolithic CPU. By separating the functional requirements along the lines of the registers and instructions we need, a single compiler can be used albeit it must be made aware of the level for which it is generating code. The main CPU should have all of the subsets supported.
- To reduce the memory latency, a maximum amount of internal memory should be added. The above granulated architecture is likely to be fairly simple because each sub-unit can be kept simpler. This

Copyright 1999 by Real-Time Magazine 99-2 (http://www.realtime-info.com)

was already demonstrated by e.g. the C62 where the CPU die size is small, partly due to the orthogonal design. This leaves more silicon space for memory. Given the evolution towards 200 Million transistors on a single chip within a few years, that will become less of an issue, although applications will always be able to use more memory.

- For the time being we ignored here issues like run-time errors conditions, emulation support, supervisor modes and other important system level issues. However most of these can be considered as software interrupts. Therefore, the main CPU could still have a classical interrupt mode, but only for handling exceptions. One could also tackle the issue by suspending the main CPU when the exceptions occur and then let the one of the peripheral units, with direct access to the main CPU registers, handle it.

Note the some parallel DSP like the TMS320C40 and 21060 already exhibit some of these features, except that all peripheral units are implemented as fixed function units (with a very modest degree of re-programmability through dedicated registers) or in software (through the RTOS). Similar tendencies are found in some ASIC designs that combine a RISC core, a MMU, one or more DSP core and dedicated peripheral circuits.

We will leave it to the specialists in the field to investigate whether this is a feasible architecture or not. Other alternatives are feasible as well, but we believe that this approach is consistent with a system level design approach, whereby the hardware is mapped onto the full software "black box" needs and not the other way around.

CONCLUSION

While simplifying along the way and rather speculative at the end, we have tried to show how the original sequential von Neumann machine as a paradigm is no longer adequate as a programming paradigm to build systems in the large. We have first shown that based on the CSP model of C.A.R. Hoare, we have an adequate model for programming in a scaleable way larger parallel systems. This model saves the von Neumann machine by embedding it into a virtual single processor. We have seen however that the current generation of processors could be improved a lot to support this model much better. ■

RECOMMENDED REFERENCES

No specific references are used, but the reader is referred to some reading material that inspired the author or that he considers worth reading.

[1] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985

[2] K. Mani Chandy & Jayadev Misra, Parallel Program Design, a foundation (the UNITY viewpoint). Addison Wesley, 1988.

[3] J.B. Worthsworth, Software developmengt with Z. Addison-Wesley, 1992.

[4] A Classical Mind, Essays in homour of C.A.R. Hoare. Ed. A.W. Roscoe, Prentice Hall, 1994.

[5] Parallel Programming and Java, World Occam Transputer User Group 20, Twente, Ed. A. Bakkers, IOS Press, 1997.

[6] J.P. Lehoczky, Lui Sha, J.K. Strosnider and Hide Tokuda. Foundations of real-time computing. Scheduling and resource management. Kluwer Academic Press, 1991. Kluwer has a very fine series on books on real-time and a unique magazoine devoted to the subject, Real-Time Systems. ed. J.A. Stankovic.

[7] D.M. Harland, Rekursiv, object oriented computer architecture, Ellis Horwood Ltd., 1988.

[8] INMOS, occam2 Reference manual, Prentice Hall, 1988.

[9] INMOS, Transputer instrauction set, Prentice Hall, 1988.

[10] A.W. Roscoe & Naiem Dathi, The Pursuit of Deadlock Freedom, Oxford University Compuyting Laboratory (Technical Monograph PRG-57), 1986.

[11] IEEE 1355-1995. Standard for Heterogeneous InterConnect (HIC) (Low cost Low Latency Scalable Serial Interconnect for Parallel System Construction), IEEE Standards Dept. Please consult the IEEE WEB site and http://www.ieee1355_organisation.org.

[12] E. Verhulst, RTXC/MP, a distributed real-time kernel defined for a virtual single processor. ICSPAT, Boston. 1992.

[13] E. Verhulst, Virtuoso: providing sub-microsecond context switching on DSPs with a dedicated nanokernel. ICSPAT, Santa Clara, 1993.

[14] Eonic Systems, Virtuoso Reference Manual, 1991-1997.

Eric Verhulst has a degree in "Civil Engineering – Telecommunications and Ballistics" at the Royal Military Academy in Brussels, 1979. After a military carreer, he founded Eonic Systems in 1989 and remained CEO until now. Eric is also IEEE Member and CEO of DSP Valley.

It's hard to compete without the right tools.

NEW ! Celeron™ & Pentium III processor in-Circuit - Emulation

Since 1991, American Arium has worked closely with Intel to create powerful new development tools for each new Intel processor as it was introduced. So whether you're using Intel's **Pentium** , **Pentium pro or Pentium II** processor, there's an American Arium tool designed to get your project to market ahead of schedule and under budget.

Pentium is a registered trademark of Intel Corporation.
Celeron is a trademark of Intel Corporation.

american arium
www.arium.com
E-mail: info@arium.com

P.O. Box 7054
5980 AB Panningen
The Netherlands

Tel. (+31) 77 307 84 38
Fax. (+31) 77 307 84 39
e-mail: info@logic.nl

LOGIC TECHNOLOGY

www.logic.nl

There's always a Logic solution