Extreme Forth – rev3, March 2022

Stephen Pelc
MicroProcessor Engineering Ltd.
133 Hill Lane
Southampton SO15 5AF
UK

stephen@mpeforth.com

Stephen Pelc is the managing director of MicroProcessor Engineering, a provider of hardware, software and firmware development tools since 1981. Stephen confesses to having programmed in DIBOL, Fortran and Algol-60 amongst other languages. He has also run a community arts centre and designed bank note sorting machines.

The articles Modern Forth and Extreme Forth were originally written in 2008. They have been slightly updated here.

The previous article, "Modern Forth (http://www.ddj.com/embedded/210600604)", focused on the impact of modern Forth compiler design on current register oriented CPUs. This article is about the relationship between software and silicon, and about a search for simplicity to improve performance and reduce chip size and power consumption.

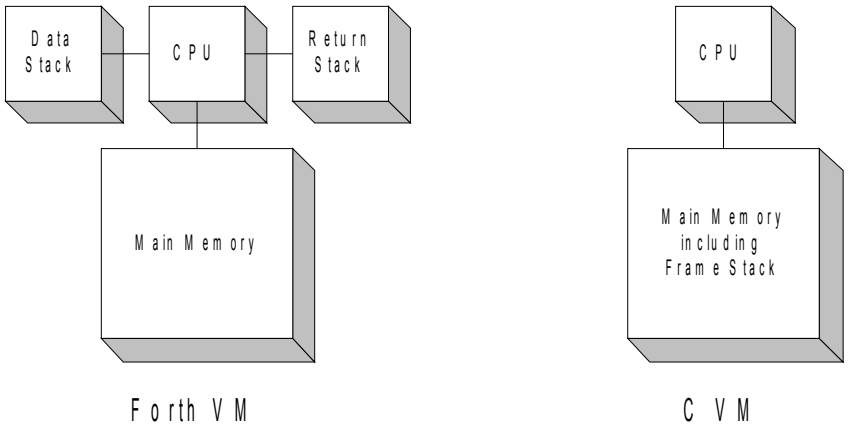
For those of you whose software life is based around C and other Pasgol languages, shift your perspective of Forth and start thinking of it as a two stack silicon machine. Forth compilers for conventional CPUs just map this model onto a register oriented model. We will alsosee how to map the C virtual machine (VM) onto a two stack VM.

Chips designed to run Forth well have been produced for over 30 years, including the Novix NC4000, the Harris/Intersil RTX2000, and Silicon Composers SC32. There has been a flurry of cores for implementation in FPGAs, including MicroCore (http://www.microcore.org). Today, the state of the art is the 144 core GA144 chip from http://www.GreenArrayChips.com. Later in this article, we will look at the C18 core and interconnects used in the SEAforth chips by Intellasys which preceded the GA144.

First, we will look at changes to the canonical Forth VM to achieve the goals of performance and code size.

Revisiting the Forth virtual machine

We will look again at the Forth and C VMs, see where the Forth VM is weak, and discover how to adjust it to improve execution of both languages. This will lead to some understanding of why the IntellaSys C18 core is as it is.



The canonical Forth virtual machine is weak in several areas.

1) It does not execute C well, which is important for commercialexploitation of general purpose silicon stack machines. C requires a frame pointer for access to local variables and buffers in main memory. The two stacks are not in addressable memory.

2) It is weak for DSP operations, which restricts performance in embedded applications without changes to the VM or increased compiler complexity.

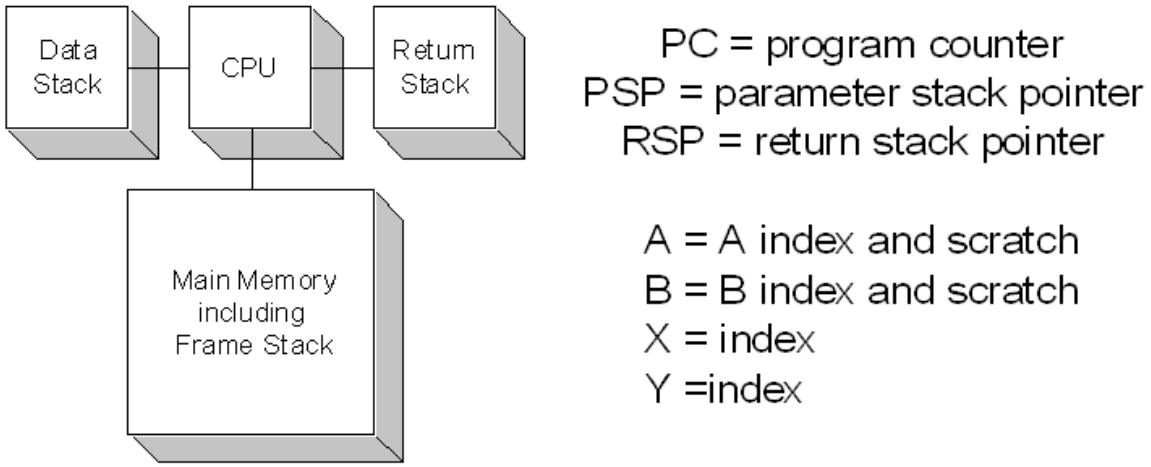
3) Without index operations, dealing with complex data structures is cumbersome, especially when a base address is passed as an argument to a word/function.

DSP operations often require three or four parameters to be manipulated, for example:

1) source address, destination address and length,

2) first source address, second source address, destination address and length.

Canonical Forth requires ugly source code to deal with these situations. Several silicon implementations provide index and scratch registers, and others have provided more access to the top of the return stack. Using the top of the return stack as a loop counter has been common for a long time, e.g. the **`FOR ... NEXT`** loop structure.



A possible new Forth VM and registers

The Forth community has long talked about TOS (top of data stack), NOS (next/second on data stack) and TOR (top or return stack). These are not quite enough for DSP operations. Chuck Moore's current silicon includes A and B registers which are used both as index registers and for scratch storage. Efficient execution of C requires a frame pointer, and a spare index register is always useful. We end up with the model above.

```
The A and B registers are used as scratch locations and for stepping
through memory using auto-increment and auto-decrement addressing
modes. The X and Y index registers have base+index addressing and
can be used as frame and thread-local storage pointers. The X and Y
registers are important for general purpose CPUs, and are not
implemented in the IntellaSys C18 and GreenArrays cores.
```

The impact of the A and B registers can be seen in this biquad filter implementation by Gary Bergstrom for a 16 bit embedded system. Gary commented on the previous article about Forth's return stack not getting in the way of parameters: "This has to be one of the most underrated points in Forth. Factoring words in Forth is natural and the lack of return addresses interspersed with the data allows this to be very efficient. In most languages you can't factor to the degree that you can in Forth without having severe run-time speed consequences. You can't keep passing data to lower and lower layers without building new stack frames, with the same data repeated in them, again and again."

```
$4000 constant +1.      \ -- n
\ Integer +1 in 2.14 fractional arithmetic format.
: *.        \ fr1 fr2 -- fr3
\ Fractional multiply.
  +1. */  ;
: 1STEP+    \ sum -- sum'
\ Perform a multiply/accumulate step, incrementing both
\ pointers.
  B@+ A@+ *. +  ;
: 1STEP-    \ sum -- sum'
\ Perform a multiply/accumulate step, incrementing the
\ coefficient pointer and decrementing the data pointer.
  B@+ A@- *. +  ;
: SHIFT2    \ fr --
\ The last step of the filter. The current data item
\ is shifted into the next data slot and replaced by fr.
  A@ SWAP A!+ A!+ ;
: (BIQUAD)  \ frx -- fry
\ The core of the biquad filter operation.
  DUP >R  B@+ *.              \ initial sum = B0*input
  1STEP+ 1STEP-  R> SHIFT2
  1STEP+ 1STEP-  ;
: BIQUAD    \ fx addr-filt addr-coef -- fry
\ A single order biquad filter.
  >B >A  (BIQUAD) DUP SHIFT2  ;
: 2xBIQUAD  \ fx addr-filt addr-coef -- fry
\ A second order biquad filter.
  >B >A (BIQUAD) (BIQUAD) DUP SHIFT2  ;
```

In this example the **A** and B registers are set up by the words **A** and **B** in **BIQUAD**. These registers are now parameters to the lower layers with no parameter passing overhead. Use of these registers has removed the need for local variables while permitting additional factorisation. They have also considerably reduced stack manipulation in both the source code and the compiled code. Because parameter passing is efficient, what would be inline code in other languages is encapsulated as factors, which in turn reduces code size. The importance of code density will become apparent in the next section.

The X and Y registers above show their worth in larger systems for indexed addressing into structures in memory. In a conventional Forth system they will be used to access local variables and buffers, and to provide a pointer to thread-local storage. One of them will be used as a frame pointer by a C compiler.

These changes to the Forth VM improve code density and performance in Forth. They also permit two stack machines to run C efficiently. A more in-depth look at this VM will appear in the EuroForth 2008 conference proceedings and on the EuroForth conference website in October 2008.

Extreme stack machines

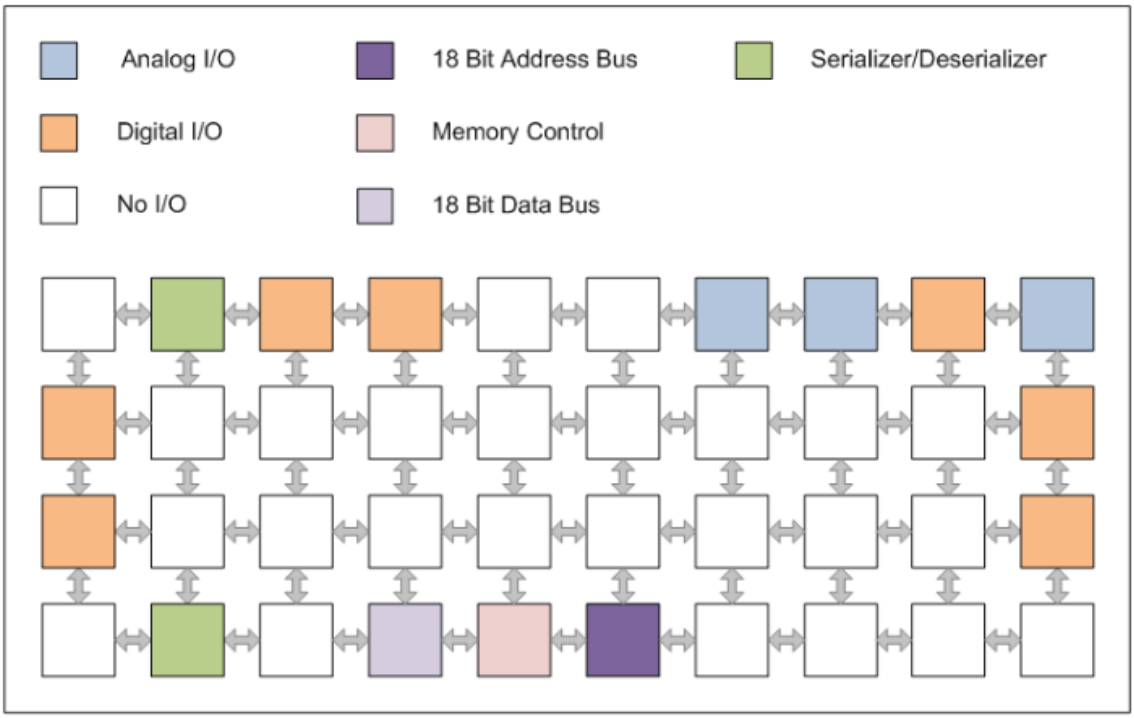
Chuck Moore is the founding father of Forth. He has always sought simplicity in implementing software and silicon. Savage minimalism is apparent in the design of the SEAforth chips produced by IntellaSys, for whom Chuck Moore is the CTO. Everything about these chips is different. The

result is astonishing processing power in a tiny chip that uses very little power. The SEAforth 40C18 has gone to production and will be shipping in December 2008. A full tool chain is available based around VentureForth, a Forth system tuned to the needs and capabilities of the SEAforth chips. As of 2022, IntellaSys no longer exists, and the production chip is the 144 core GA144 from **`http://greenarraychips.com`**.

I do not have the space to do more than outline the capabilities of such a device and to discuss some of the software issues involved in writing applications for it. Technical information is available for the similar GA144 from the `GreenArrayChips` website.

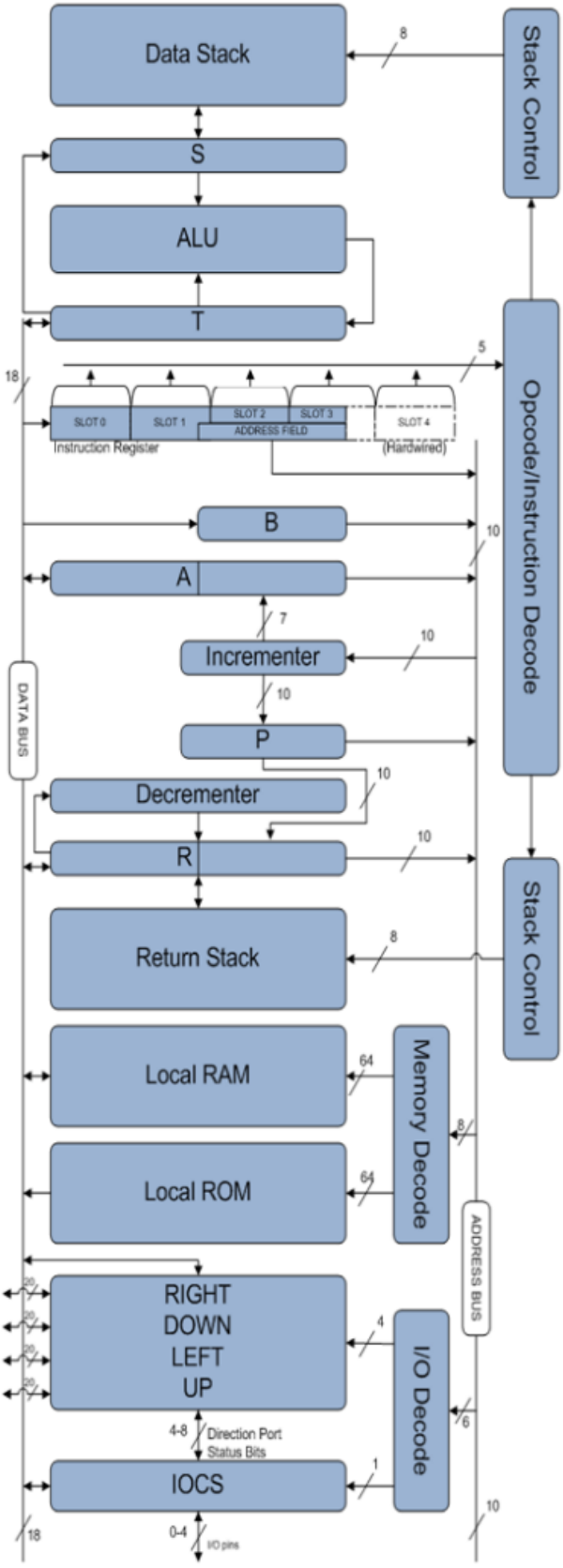
Each 40C18 chip has 40 cores. Each core contains a C18 stack machine, ROM, RAM and interconnects to its nearest neighbours. The ROM for each core contains a BIOS that can be used by application code in RAM. A BIOS for an edge core can contain code that emulates a serial port, an SPI port, or a DRAM controller. Interconnects on cores at the edge of the chip can include I/O. The inner cores lacking I/O provide functions needed by their neighbours.

Each core runs asynchronously - there is no common clock or crystal. When a CPU reads or writes data to/from a neighbour that is not ready, it just goes to low power sleep until the neighbour completes the transfer. Each core runs at 600+ MHz, giving a total processing power of up to 26 billion operations per second. Programs are loaded from an external EEPROM or Flash.



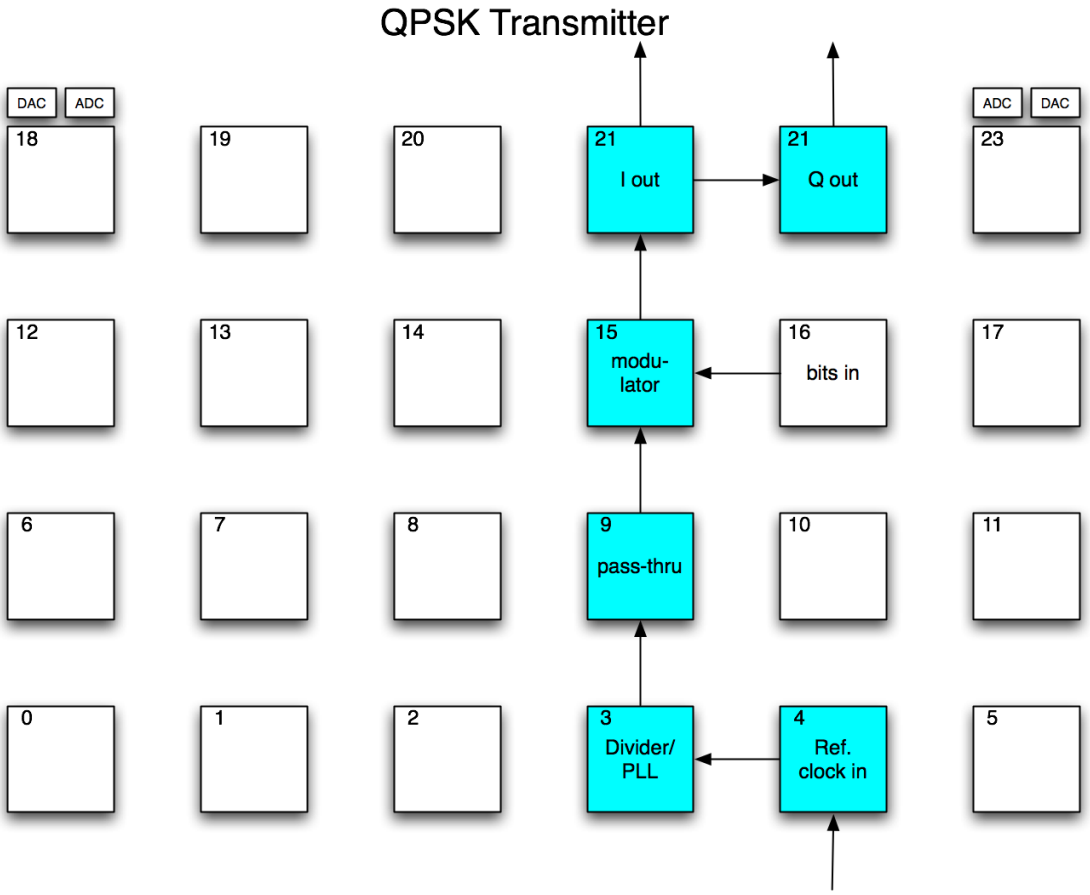
That was the good news. Each core is tiny with an 18 bit data bus, and 64 18 bit words each of ROM and RAM. Instructions are packed four to a word. The core implements a subset of the A and B register system discussed in the previous section. The B register is optimised to be a pointer. The two stacks cache the top items (two for the data stack and one for the return stack), and then contain an additional eight items arranged as a circular list. Every time a stack is pushed or popped, the next or previous item is selected. An odd but useful side effect of this is that a pop does not actually destroy data, it just goes to the bottom of the stack. Devious programmers takeadvantage of this feature to use the stacks as small data caches.

Unless you have followed Chuck Moore's work over the last ten years or so, you will find this design radically different from anything else. It's actually just another CPU whose subtleties you have to learn.


Data Stack
8
Stack Control
S
ALU
T
5
18
SLOT 0
SLOT 1
SLOT 2
SLOT 3
SLOT 4
ADDRESS FIELD
Instruction Register
(Hardwired)
Opcode/Instruction Decode
B
10
A
7
Incrementer
10
DATA BUS
10
P
10
Decrementer
10
R
Return Stack
8
Stack Control
Local RAM
64
Memory Decode
8
Local ROM
64
ADDRESS BUS
20
20
20
20
RIGHT
DOWN
LEFT
UP
4
I/O Decode
6
4-8
Direction Port
Status Bits
IOCS
1
10
18
0-4
I/O pins

With a maximum of a few hundred opcodes in each core, your major problem is not programming each core, it's floor planning – how the cores interconnect. Chuck Moore says that he can program a core in a day. People with more experience programming SEAforth than me repeatedly say that getting the floor plan and interconnects right is the major part of programming a SEAforth chip.

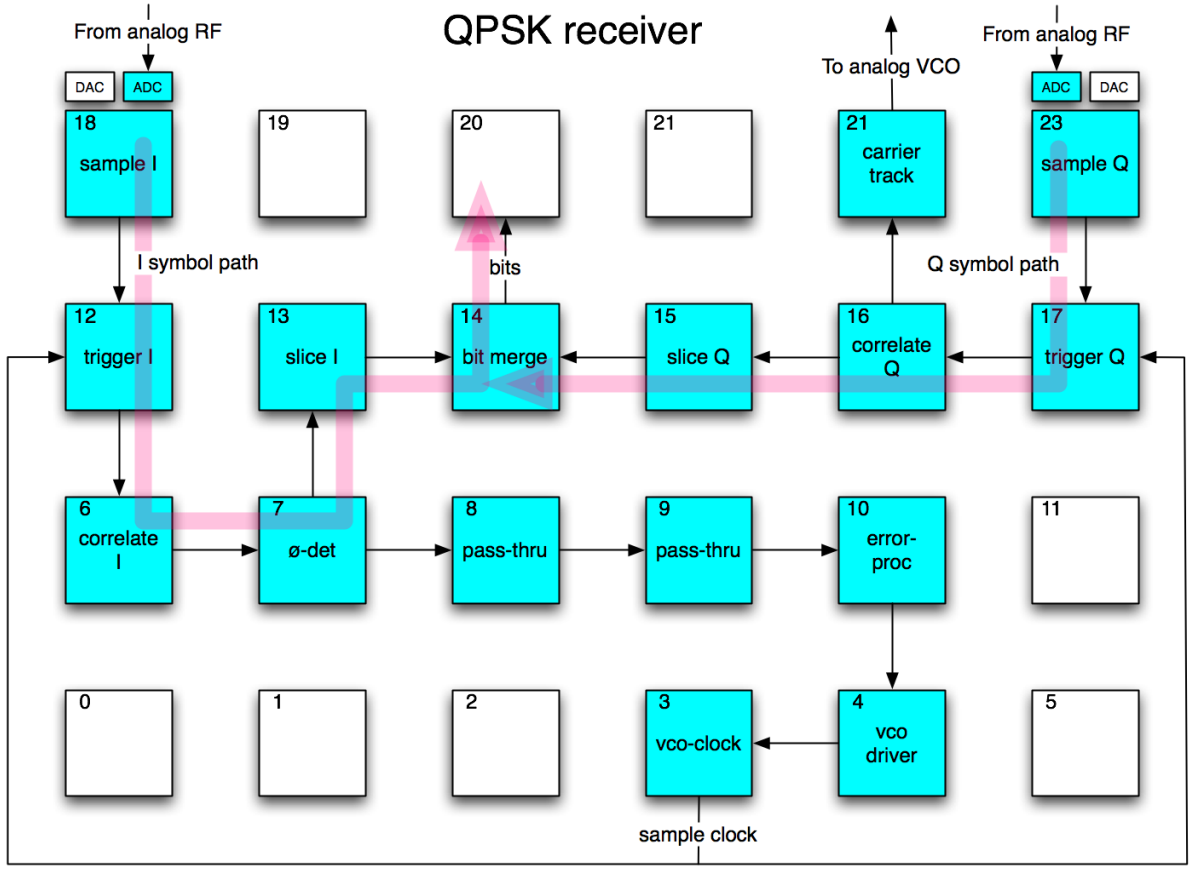
You have to decide how to partition the cores and explore the consequences of how each core is used. To illustrate the issues of floor planning, we will look at a QPSK transmitter and receiver system implemented on the previous IntellaSys 24 core device. Compared to a conventional microcontroller, the SEAforth devices have very few peripherals in hardware, the main ones being digital-to-analogue and analogue-to-digital converters. The ring of cores around the edge of the chip are programmed to perform functions that would normally be performed in hardware. Later versions of the SEAforth family will have enough cores to approach the "one core per pin" model.



The 24 core chip is a 6 by 4 array. The reference clock input (core 4), and the two outputs (cores 21 and 22) have to be at an edge. Because of other functions needed in other cores, the path though the chip includes one core (core 9) which simply passes data. Core 16 provides data to the modulator in core 15. What we see here is that each core performs a simple task, and then passes data to the next

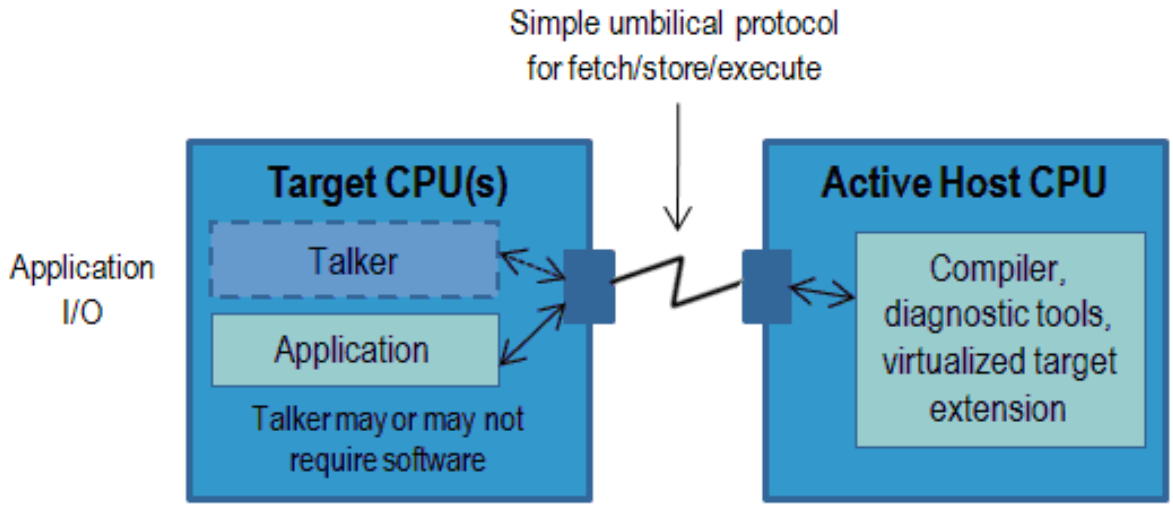
core in the chain. In this example the partitioning is relatively simple. The receiver is a different matter.

Here we see the designer taking account of where the analogue-to-digitalconverters are placed on the chip. The majority of application code runs from RAM. This permits a programmer to redefine the function of a core during run-time. One example of this is for selecting the modulation in a multi-mode receiver. Because the SEAforth 40C18 is a low power device, it is ideal for software defined radio (SDR) applications. All this begs the question of how programs are loaded into RAM.

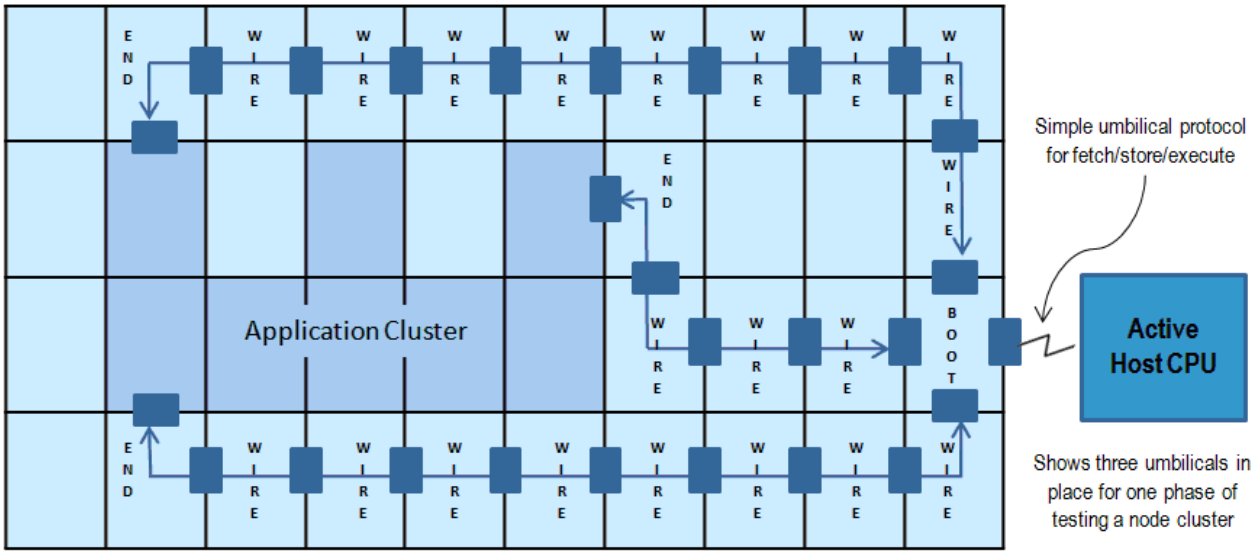


The architecture of the core places the I/O ports in memory from which code can be executed. If the BIOS code sets the core to execute an instruction from an interconnect port and there is no data, the core goes to sleep until the data is available. If the data is an opcode set (up to four instructions), the instructions are executed. It is possible in these four instructions to read a data block into core memory or to pass it on to another port. The latter is known colloquially as a "port pump". Port pumps need not use any local RAM space. When a chip is reset, the BIOS in one or more edge cores looks for data, e.g. from a serial SPI Flash device. The start of this stream triggers a port pump to transfer data (code) to a neighbouring core which in turn passes the stream to its neighbours.

We can take advantage of this mechanism to aid development and debugging. The compiler remains interactive when your code has compiled, and is connected to a core on the target device through an umbilical link.



The use of interactive compilers and umbilical debugging links is common practice in cross-compiled Forth systems. These same techniques are used to load and execute programs in multicore chips. Because port pumps can be essentially non-invasive, they can be used for debugging as well as for loading programs. Once you have selected a boot core for the umbilical link, you can get to any core you want to debug. However, there is not quite a free lunch.



In practice, you work by debugging the nodes furthest away, and pull back to the boot/umbilical link core. Because each core is asynchronous, the simulator cannot be cycle accurate across multiple cores. As with any software, you have to do some testing on the real target. You have to design testability into the system as well as design the code.

Debugging is a major issue in multicore programming, regardless of the interconnect system, and is a work in progress for all multicore architectures. The approach used in the IntellaSys SEAforth chips provides small low-power multicore chips that can be programmed by mortals like us. It is the change in thought processes in going from single core programming to multicore programming that causes the learning curve.

Acknowledgements

The staff at IntellaSys provided a great deal of support over the last year, in particular Chet Brown, Debbie Davis, Dean Sanderson and Dylan Smeder. Chuck Moore started all this a while ago, and has shown what can be done with so little.

Gary Bergstrom has provided trenchant comments and input for this and the previous article. His code is published with permission.