UFO – Ubiquitous Forth Object

Chris Bailey
Dept. Computer Science
University of York
chrisb@cs.york.ac.uk

Stephen Pelc
Microprocessor Engineering
133 Hill Lane
Southampton SO15 5AF
stephen@mpeltd.demon.co.uk

Abstract

The UFO will form a combination of CPU and operating system technology, namely a stack-machine CPU core, and a Forth operating system. However both of these components will have novel properties, in that a scaleable target instance can be generated, according to a trade-off between gate-count, speed, power, and computational performance. Simpler machines can emulate more complex ones, but at the cost of lower performance than a physical implementation of the more complex architecture.

Introduction

The desire to produce a dual stack machine in silicon remains within all embedded Forth programmers, yet does the world really need another CPU design when so many are freely available? To this question we answer "yes", because there are still many unsolved issues about stack machine design, in particular super-scalar systems. As CPU clock speeds increase, it should be noted that interrupt response has not improved in the last ten years.

So we took the plunge and decided to design a scalable CPU which can be implemented as a VHDL/Verilog element in a larger System-On-Chip (SOC) design.

The outcome will be a CPU node which can exist in its simplest form as a bare-skeletal machine, yet still capable of supporting a Forth environment albeit with poor performance relative to a fuller system. In this case speed is not of the essence, but minimalism succeeds. The CPU core will be developed such that additional contexts can be enabled with successively more CPU capability, such that the Forth environment will gain in speed at the cost of gate count and power consumption.

Consequently, software and behaviour of the MicroMote will appear uniform, and low-level machine differences will be transparent to the programmer and user except with respect to overall performance. The purpose will be to demonstrate a ubiquitous node capable of supporting very small, or more complex tasks with hardware scaled to the objective in each case.

Embedded systems

There are four main areas in which embedded systems differ from desktop computing.
1) Importance of interrupt response time
2) Importance of deterministic response time
3) Economics of code size
4) Importance of branch behaviour

In the case of stack machines, all comparisons are against an RTX2000 CPU at 10MHz (circa 1988). The state of the art for an embedded system is taken from a 60MHz Philips LPC2106 ARM-based microcontroller without caches but with a memory accelerator (2003/4).

Interrupt response time

I define this as the time taken for the CPU hardware and software to save context BEFORE starting useful work. To this must be added the context restore time AFTER performing useful work. The code below shows the figures for a 10MHz RTX against a 60MHz ARM7TDMIS.

Entry

```
RTX2000     400ns     0 bytes     0 instructions
ARM         432ns     20 bytes    5 ins, 27 cycles @ 16ns
```

Exit

```
RTX20000    200ns     2 bytes     1 instruct
ARM         400ns     16 bytes    4 ins, 25 cycles
```

Note that these figures exclude any overheads for compiler-generated code.

The figures show that despite a 6:1 to 40:1 clock rate increase in 15 years, where typical top-end embedded CPUs run at 200-800MHz, interrupt response times have not improved. The ARM is among the better performers in embedded RISC CPUs. As shown by Koopman et al, interrupt response (entry) times in current CISC CPUs can exceed 400 clock cycles. When caches and MMUs are involved, the situation becomes even worse.

```
PROC IRQ_entry  \ --              \ 4 cycles
\ --- save state ---
  stmfd rsp ! { r0-r12, link }    \ 3 + 14 mems
  mrs   r3, SPSR                  \ 1
  stmfd rsp ! { r3 }              \ 2 + 1 mem
\ --- call high level handler ---
l: IRQ_call
  bl    <action>                  \ 3
\ --- restore state ---
  ldmfd rsp ! { r3 }              \ 3 + 1 mem
  msr   SPSR, _c _f r3            \ 1
  ldmfd   rsp ! { r0-r12, link }  \ 3 + 14 mems
  sub .s  pc, link, # 4           \ 3
end-code
```

Determinism

Many embedded applications sample regular signals (heartbeat, 50/60Hz mains, audio etc.). It is imperative that sampling periods are at fixed intervals to reduce phase jitter in the

sampling. Modern CPUs achieve high clock speed using caches and long pipelines. Both of these have adverse impact on determinism. See Koopman et al for the numbers.

It should be noted that a heavy interrupt load affects both entry and exit performance. In the worst case, the whole of the interrupt routine must be loaded from main (slowest) memory and displaces the background task which in turn must be reloaded from main (slowest) memory.

Economics of code size

Silicon costs go up as the fourth power of dimension (yield, chips/wafer etc) and power consumption goes up with the number of active transistors per clock within the same geometry. In the single chip microcontroller area, code size affects memory in terms of on/off chip Flash, and also in terms of RAM usage. Many die photos demonstrate that the silicon area of on-chip memory exceeds that of the CPU and peripherals.

Importance of branch behaviour

Typical microcontroller code shows that branches and flow of control occur roughly every 5 instruction (approx 20% of the code). A Pentium 4 with a 20 stage pipeline, and misses in all caches and branch prediction buffers can suffer a 30 cycle penalty for a missed branch (20 cycles in the pipeline, 10 in the memory). The ARM above has a 4 cycle worst case (1 for decode, 3 for memory), whereas the RTX has a fixed 2 cycle overhead in all cases.

Conclusions

In the area of hard real-time computing ("late ansers are wrong answers"), current CPU and memory architectures have done little or nothing to improve interrupt response and determinism. Current CPUs often have worse characteristics than those of 15 years ago, despite the huge increase in CPU clock rate.

Commercial stack machines have shown impressive determinism (Novix NC4000, Harris RTX2000, Silicon Composers SC32) and have been used for medical imaging, hard disc drives and satellite applications. MPE's RTXcore CPU for FPGAs was developed because the client could find no other way to achieve the required determinism.

With the increasing use of virtual machine code such as the Java VM, silicon stack machines have become an increasing attractive area of research. To date, although it has been shown several times that such machines can be manufactured in FPGA, there has been no active research exploring what can be achieved for hard real-time systems while investigating the impact of modern silicon technology and potential superscalar stack architectures. This project offers us the chance to research the areas of code density, determinism and performance; it also is managed in a way to provide FPGA implementations and the all-important software too chains in a form that can be taken to market rapidly.

The UFO engine

We proposed to develop a synthesisable stack machine with the ability for the licensed developer to adjust the hardware/software boundary during synthesis. What this means is that the architecture will have an instruction set architecture which remains consistent as far as any overlying software and compilers are concerned. At the hardware level, the physical implementation of particular instructions could be hardwired, implemented as on-chip ROM routines, or as vectors to software in memory. Consequently the hardware cost of implementing the CPU can be scaled up or down. This has benefits for ubiquitous applications domains, since a standard software architecture can run transparently on several different grades of hardware, from low-footprint low gate count, through to high performance larger gate-count implementations.

This means that integration of extra hardware (e.g. second UART) could be traded off against CPU performance where gate capacity of an FPGA is tight, or indeed where an ASIC die area is to be minimised, along with power consumption etc. Currently if a particular FPGA were to be used for a ubiquitous node, the main choices are which soft-CPU to use, and what peripheral IP cores are to be integrated.

For Ubiquitous/Pervasive computing, the ability to reduce hardware cost and power consumption together is a much better solution than simply having a fixed size CPU and reducing the clock rate to tradeoff power vs application processing - this reduces power but not gates and transistors used.

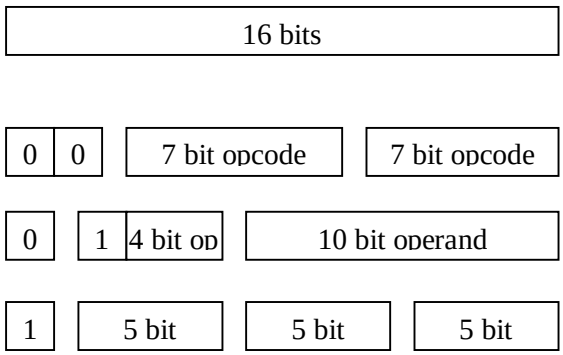
From a commercial point of view, the CPU must be supported with compilers for C/C++ as well as Forth. Not only must a C compiler be available, it must generate good quality code. As has been shown several times, only minor adjustments to the canonical Forth stack machine are required to support C well, both in terms of additional registers and in terms of additional instructions.

Instruction set design

The UFO instruction set design is compromise to permit small systems to run with an eight bit memory bus, while permitting larger systems to run with a 32 bit bus. Because TCP/IP networking is assumed for all UFO nodes, there is an assumption of a big-endian byte-addressed machine whose stack cells are 32 bit units.

We had no desire to use variable length instruction encoding at first. However, the requirement for indexed addressing modes lead us to using a fixed 16-bit instruction payload containing subunits.

16-bit Payload decomposition



It can be seen that only one or two bits are needed to determine the sub-type. In the case of an 8-bit fetch model, all of the decoding bits are ion the first fetch, allowing any operation to commence with no further decoding at the start of the third clock cycle. In the case of the 5-op format, the first operation can start after the first fetch.

The 4-bit operations would be reserved for immediate operand functions such as :-

0 Lit <10 bits>
1 LOC <1,1,8> <R/W> <W/L> <offset8> access to local (FP+n)
2 CONST <1,9> <W/L> <offset9> access const at (PC+n)

3 LDPI <1,1,(1+7)> <XP/YP> <B/{W/L}> <8offset(byte)/7offset(Long/word)>
4 STPI <1,1,(1+7)> <XP/YP> <B/{W/L}> <8offset(byte)/7offset(Long/word)>

5 BrZ <10 bits>
6 BrNZ <10 bits>
7 ZCALL <10 bits> Zero Page Call (for kernel routines etc)

8 +PP <2,8 bits> Where PP = PC,XP,YP,FP (for pointer stepping PP+=8b)
9 -PP <2,8 bits> Where PP = PC,XP,YP,FP (does bru if pp=PC)

10 PXIA <10 bits> push XP+10 bit
11 PYIA <10 bits> push YP+10 bit

12 undefined
13 undefined
14 undefined
15 undefined

The 7-bit opcodes would provide the main instruction set implementations, whilst 5-bit ops would duplicate the first 32 opcodes in a short format. These would include often used primitive operations such as the following example scheme:-

0 Add
1 Sub
2 Inc
3 Dec
4 And
5 Or
6 Exit
7 Dup
8 Drop
9 Swap
10 @.B
11 !.B
12
13
14
15
16
17 Skip (skip +2 payloads)
18 Tne
19 Teq

24-31 Constants 0 to 7

For example the sequence Lit 7 FP- Exit might be used to deallocate a variable frame and exit. It would encode into one 16-bit payload.

As a result the packing scheme allows the following permutations:-

	Payload variants (Max cases)	Other comments
8-bit	3 primitive ops 2 standard ops (7bit) 1 embedded immediate op	Two fetches required for each payload First op executes in first fetch cycle First op executes in 2nd cycle Single op executes in second cycle
16-bit	3 primitive ops 2 standard ops (7bit) 1 embedded immediate op	One payload per memory fetch First op executes in first cycle First op executes in first cycle First op executes in first cycle
32-bit	6 primitive ops 4 standard ops (7bit) 2 embedded immediate op plus combinations of above 3 cases	Two Payloads per memory fetch First op executes in first cycle First op executes in first cycle First op executes in first cycle
64-bit	12 primitive ops 8 standard ops (7bit) 4 embedded immediate op plus combinations of above 3 cases	As for 32-bit