Using MicroCore

Report on a
Deeply Satisfying
Instantiation

kschleisiek @ send.de
www.microcore.org

Abstract

Three years ago I presented the "MicroCore" architecture, a dual-stack Harvard processor core that took Forth as a guideline to define its instruction set, adapting architecture innovations of the Transputer. Then, MicroCore, or "uCore" for short, only existed as simulated VHDL code.

Since then we have used uCore to realize our latest generation 4 channel seismic seabed data acquisition recorder. It turned into a proof of concept beyond expectations. Software development has been quicker than anticipated repeatedly.

This is mainly due to the use of the Exception Mechanism, which replaces about half of what conventionally would be handled by interrupts, reducing interface software complexity substantially. Furthermore, a small number of special purpose instructions reduces software complexity even further, reducing power consumption at the same time.

To sum it up: Full control over all aspects of system design, namely processor architecture, operating system, and application programming not only allows to optimize the hardware/software interface w.r.t. simplicity, elegance and power efficiency, but at the same time the cost of future hardware advances - previously abhorred as "software porting" - is drastically reduced, because the instruction set of the "soft core" processor is not affected by a change of the underlying FPGA technology.

1 Prototyping board "uCore100"

As a first step to using uCore in real applications a prototyping board "uCore100" has been realized (sponsored by Forth Gesellschaft eV) consisting of an XC2S200 FPGA, 512k program and 256k x 32 data RAM (10 ns), 2M x 16 Flash, an uncommitted USB port, and lots of uncommitted I/O pins brought out on pin headers and a 96 pin edge connector. Using a 24 MHz crystal, uCore executes one instruction every 80 ns.

uCore100 has been used to fire up uCore for the first time and to develop the debugger. Initially the "umbilical interface" of the debugger was connected to the PC printer port (centronics) that allowed rapid parallel upload of the program memory with asynchronous handshake in both directions and it served to develop the debugging environment on the host PC. Later on, in preparation for migrating to the actual application hardware, a UART (COM interface) was used as umbilical, transferring 32 bit data items as a sequence of bytes ("start" byte followed by four data bytes), followed by a handshake byte in the opposite direction to signal completion for every single transaction.

Later on it turned out that this approach could be carried over to the actual application operating in two "modes": In "user mode", the UART would be used to implement KEY and EMIT as usual. In "debug mode", the UART would be used to communicate with the debugger transferring 32 bit data items that reach through to KEY and EMIT when bits 31 through 8 are zero.

1.1 Software Development Environment

The development environment (debugger) runs under Linux and is written in GCC for the sake of portability. A Forth cross-compiler that is beefed up by uCore specific instructions serves as a macro assembler loading on Gforth and Win32For, and a mating disassembler allows to inspect the produced code. It is able to produce VHDL code to aid in hardware simulation as well as binary object code and a symbol table file for the interactive debugger.

The umbilical interface of the debugger is supported by appropriate hardware on the target system and it allows to initialize the program memory with or without a reset of the processor core. One of the single cycle user instruction vectors is used as breakpoint that can be "pushed" through the code under control of the debugger when instructions are single stepped. On uCore itself a conventional debug monitor listens to the umbilical waiting for an instruction address to jump to. When ready, it returns a completion or an error code to the host.

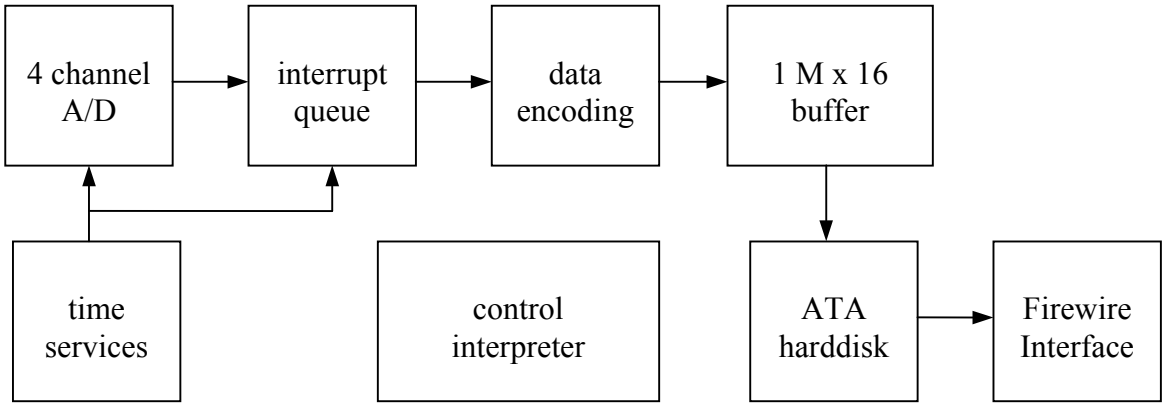
2 Geolon-MCS system architecture

Geolon-MCS is a four channel (hydrophone and 3-component geophone) "seismocorder" to be used in autonomous seismic acquisition systems on the seabed. Its A/D converters produce 24 bit numbers @ a SNR of 130 dB and it consumes 500 mW power storing the data on automotive HDDs that can be read out via a firewire interface.

The hardware is partitioned into

- power management producing +/- 2.7V, 3.3V, and 5V from the 6 - 15V battery input
- digital control with 256kx8 program RAM, 256kx16 data and return stack RAM, 1Mx16 data buffer RAM, an RS232 interface for interactive control, an ATA HDD interface, a firewire interface.

- A/D converters with differential pre-amplifiers for the hydrophone and the geophones, sigma delta modulators, 24 bit digital filters, reference voltage generation and analog power conditioning.
- high precision microprocessor controlled crystal oscillator producing both a 1pps timing pulse and a 12.288 MHz clock signal.



Data flow architecture of Geolon-MCS

An unexpected yet obvious benefit of using a soft processor core in an FPGA: There is no processor chip that consumes pc-board real estate nor any wiring and therefore, all of the digital electronics did fit on one boards instead of two boards before.

2.1 uCore Implementation

Before porting uCore to the MCS hardware, it was prototyped on the uCore100 prototyping board i.e. physical data memory was assumed to be only 16 bits wide, and the debug umbilical was re-implemented to use the RS232 serial interface instead of the parallel centronics port. In addition, 8 parallel instantiations of the stacks were realized and our proven co-operative multi tasker with both priority and round-robin scheduling was ported to "uCore-Forth". Fortunately enough, the very first implementation of uCore on the new hardware talked to the debugger immediately.

Characteristics of the uCore instantiation for Geolon-MCS:

Data path width	32 bits
Program memory	256k external RAM, 2k internal "cache"
Data memory	128k external 55ns RAM, 1k internal "cache". Used for variables and the return stack. The external RAM is only 16 bits wide physically and therefore, each fetch and store takes two cycles.
Data buffer	2MB external 55 ns RAM 16 bits wide to allow DMA access to the 16 bit IDE interface.
Clock frequency	12.288 MHz = 160 ns instruction execution cycle
RTOS	co-operative multi tasker with 8 physical data and return stack areas, 12 usec task switch
FPGA	XC2S200E "Spartan 2" technology, 35% of logic resources used for uCore

2.2 Booting

Another advantage of the soft core approach is the total freedom to implement any booting strategy. Given the MCS' hardware capabilities the following boot process has been realized:

Once the FPGA has been configured off of its external serial flash memory, an internal flip-flop "boot" is preset to '1'. (How this can be realized seems to be synthesizer specific, unfortunately.) While "boot" is true, a small boot program is mapped into the program memory space that switches on the HDD and copies a certain number of sectors from the disk into the "normal" program memory. When this has been done, a branch to address zero (the reset vector address) is executed, which resets the boot flip-flop and starts execution of the program that has been read off the disk. The boot flip-flop is not affected by a processor reset: It will remain false once reset by the initial branch to zero. This boot program consumes about 120 instructions and it is "hard coded" into the FPGA configuration by the synthesizer.

2.3 FPGA Technology Issues

The choice of FPGA technology was not trivial. It was clear that we would use Xilinx FPGAs because of prior knowledge of the software tools (in retrospect that appears as an unduly restriction on potentially better FPGA choices). At first, the new Spartan 3 architecture appeared as a natural choice, because it offers multiply cells that would have allowed to implement a 32x32->64 bit single cycle multiply instruction. BUT: Spartan 3 is 90 nm silicon technology with a dramatic increase in static power consumption compared to older technologies and the FPGA would have consumed 200 mW without even applying a clock signal. Therefore, we settled for the XC2S200E FPGA that does all the control needed in the MCS at a total power consumption of 40 mW.

3 Exceptions / Interrupts

The most fruitful concept that uCore inherited from the transputer is its "exception mechanism" and we have made rewarding use of it. Let me first clarify what it is by comparing it to the well known interrupt mechanism:

Interrupt: An event did happen that was **not** expected by software.
Exception: An event did **not** happen that was expected by software.

And this is how the debug interface uses the exception mechanism: There is a DEBUG_REGISTER in the FPGA that serves as an input and output for the umbilical to the host. When uCore has nothing better to do, it just does a DEBUG_REGISTER @. Everything is fine when there has been information placed in the register by the umbilical.

But what happens if nothing is there that could be fetched? Or, to rephrase the sentence above, if the "umbilical" event did **not** happen that was expected by the "@"?

The address decoding electronics of the DEBUG_REGISTER (memory mapped) inside the FPGA "know" whether there is a new item in the register or not. If not, it asserts the exception input of uCore during the execution of the fetch instruction. This keeps all registers inside uCore in their previous state with the exception of the instruction register, which is fed the "exception instruction", which consequently will execute a call to the exception vector during the following cycle. Therefore, the processor behaves as if the fetch would not have been executed at all. When execution of the exception vector instruction starts, the return stack holds the program memory

address one instruction past the fetch due to the call performed. Therefore, the simplest form of exception service routine is the phrase `R> 1- BRANCH`, which will attempt to "re-execute" the fetch instruction that raised the exception before in a very tight loop. Therefore, the processor would stall at the fetch instruction until a new value becomes available via the umbilical. No more querying of status flags and all the IFs and WHILEs we are so accustomed with in real time systems programming, which clutter the code, make it obscure, and difficult to maintain.

It will come as no surprise that in a multi tasking system the exception service routine consists of a call to PAUSE of a co-operative multi tasker, thereby putting the task that wants new input from the DEBUG_REGISTER to sleep, doing something else for a while.

The exception mechanism applied to the umbilical was the first use we made of it and when the system did not crash for hours even in the presence of a 10 ms timer interrupt we gained confidence that we "got it right".

3.1 Timing Services

Conventional wisdom always realizes some regular interrupt (e.g. every 10 ms) that increments variable TIMER, forming the basis for timing services e.g. using the following set of words:

```
: ahead    ( ticks -- time.ahead )  Timer @ + ;

: timeout? ( ticks -- ticks f )     dup Timer @ - 0< ;

: continue ( ticks -- )     BEGIN  pause timeout? UNTIL drop ;

: sleep    ( ticks -- )     ahead continue ;
```

In the MCS this has been realized more elegantly using the exception mechanism instead, eradicating one interrupt source. A memory mapped register TIMER is incremented every 30 usec (because: why not? PAUSE is executed in 12 usec after all!).

TIMER @ works as expected: It returns the current content of the TIMER register. But TIMER ! behaves quite differently: When the "time.ahead" value on the stack (in NOS) is larger than the current content of TIMER, an exception is raised and therefore, program execution does not continue past the "attempted" store instruction. Based on this hardware mechanism CONTINUE can be realized much simpler and it may even pay off to define it as a macro:

```
: continue ( ticks -- )     Timer ! ;
```

3.2 Semaphores

It did not take long to realize that the exception mechanism can be used to realize "real" semaphores in hardware. To this end, the SEMAPHORES register has been implemented, and each single bit can be used as a semaphore, which synchronizes the software with specific events.

Executing e.g. **`#sema_ide Semaphores !`** will raise an exception as long as the #sema_ide bit of the Semaphores register is set. Otherwise, execution will just continue.

As of today, these semaphores have been implemented in the MCS:

1 Constant #sema_ide
will be set by storing a command into the hard disk command register, and it will be reset by the hard disk interface interrupt on completion of an ATA command. Again, this completely replaces one interrupt source of previous systems.

2 Constant #sema_adc
will be set when storing a command into the SPI interface of the A/D converters as well as when passing this semaphore, and it will be reset when the A/D converter is ready to accept the next command, which is realized by a state machine that autonomously polls the A/D converters. This saves cumbersome status polling.

4 Constant #sema_reset
will be set when the external reset signal is raised, and it will be reset when the external reset line has settled to its inactive state again. This way all peripheral interfaces (A/D converters, firewire interface etc) can be reset under program control, and before re-initializing these interfaces, the processor would wait on this semaphore. Actually, this semaphore was realized as a bug fix for a buggy peripheral chip.

8 Constant #sema_buf
will be set when the data buffer is full and it will be reset when the data buffer pointer has been set to the beginning of the buffer again. Refer to the discussion of the "Data Encoding and Storage" special instructions below.

3.3 Events with timeout

```
7 POP USR Opcode: event ( ticks semaphor.bit -- ticks )
```

This is the definition of "User" instruction EVENT (but I am still looking for a better name). EVENT combines both of the above described mechanisms: timing and semaphores. It waits on semaphore semaphor.bit but only until ticks has elapsed. I.e. an exception will be raised if both conditions hold: Time ticks has not been reached yet and semaphor.bit is still set in Semaphores.

If semaphor.bit in Semaphores is not set, execution continues after EVENT and the carry bit will be reset. If ticks has elapsed, execution continues after EVENT as well but the carry bit will be set. Therefore, a branch on carry (a single uCore instruction) will differentiate between these two cases. EVENT would be used like this:

```
50 ms ahead #sema_adc event drop  carry IF adc_error THEN  ...
```

4 Special Instructions

Another chance for code simplification is the implementation of special purpose instructions, which may be very application specific. Besides simplifying the code and making it more readable and more reliable they will save energy as well, because the number of instructions that need to be executed in order to achieve a wanted data transformation will be reduced perhaps up to a point where the processor may be clocked at half the speed or even less.

4.1 Complex Math

These are not special purpose instructions in the narrow sense but I am including them here, because I did not describe them before. These are "math step operators" which have to be executed repeatedly for each single bit of the data path width to generate a valid result. All of these instructions operate on three registers at once: the two topmost items of the data stack and the top of return stack.

```
MULTS NONE ALU Opcode: mults
```

Before executing MULTS repeatedly, the multiplier, multiplicand, and product has to be set up in their appropriate register and afterwards the stack has to be cleaned up. This creates an overhead of four instructions. Therefore, it takes data_path_width + 4 cycles to realize the Forth word UM*.

```
0DIVS NONE ALU Opcode: 0divs  \ sets up the registers
UDIVS NONE ALU Opcode: udivs  \ unsigned division step
LDIVS NONE ALU Opcode: ldivs  \ final division step
```

Setting up the registers for division is more complex compared to multiplication and therefore, instruction 0DIVS has been implemented. In addition, division requires a last irregular "correction step" that is performed by LDIVS. Taking these three instructions, it also takes data_path_width + 4 instructions to realize the Forth word UM/MOD.

4.2 Byte handling

These instructions are also pretty much general purpose and they serve to operate on bytes of data. They could be emulated by executing single bit shift instructions repeatedly but they have been realized as single cycle instructions for the sake of efficiency. Their semantics is self explanatory given their names.

```
UP   NONE ALU Opcode: 256*  ( n -- n*256 )
DOWN NONE ALU Opcode: u256/ ( u -- u/256 )
6    BOTH USR Opcode: 256/  ( n -- n/256 )
```

4.3 Pseudo DMA

```
0 PUSH USR Opcode: ide@  ( bufaddr -- bufaddr+1 )
0 POP  USR Opcode: ide!  ( bufaddr -- bufaddr+1 )
```

These two instructions are specific to transferring 16 bit data between the IDE hard disk interface and the buffer memory. With the buffer memory address on the stack, IDE@ transfers 16 bits of data from the disk to the buffer, and IDE! transfers 16 bits from the buffer to the disk. At the same time, the address is incremented by one, ready for the next transfer. One sector on the disk is 512 bytes or 256 words long and therefore, executing 256 of these "DMA instructions" in a row will transfer one complete disk sector in 256 processor cycles, while the processor may be interrupted in between at any time with no latency.

4.4 Data Encoding and Buffering

Data encoding in the MCS is quite unique and optimized for robustness. The disk is treated as a serial storage device as if it were a tape and the data are stored continuously without any block structure. Instead, each data item is tagged with a variable number of tagging bits, which are stored in the most significant bits of each data item. This way, even if a disk sector may not be readable any more, only this sector of information will be lost and the tag code is constructed in such a way that the decoding process can re-synchronize to the data stream quite rapidly.

Sampled data is encoded as follows: The current sample is subtracted from the previous sample. If the signed difference fits into seven bits, it will be stored as a byte with tag code '0'. If the difference fits into 14 bits, it will be stored as a 16 bit word using tag code '10'. Otherwise, the full 24 bit signed number will be stored as a 32 bit item with tag code '11110'. Besides data, other codes are used to tag time and status information. Most importantly, a 32 bit synchronization token exists, which is used to re-synchronize the decoder. No matter how confused the decoder is, after the synchronization token it will be properly aligned to the data stream again.

Computationally, two problems exist using this strategy:

1. Encoding the data.
2. Storing the variable width data in the 16 bit wide buffer memory.

Both processes are computationally intensive given traditional processor instructions and in previous systems, these two inner loops constituted the limiting factor on the maximum data rate that could be processed. To make the benefits of the special instructions more obvious, I will show the code needed with and without the instructions, which have been realized for this purpose.

This is the uCore code needed for the data encoding and buffering process using standard instructions:

```
: split  ( 32b -- l16b h16b ) dup $FFFF and swap u256/ u256/ ;

: wsplit ( 16b -- l8b h8b )   dup $FF and swap u256/ ;

: buf8! ( 8bit -- )
  >r idebuf_ptr @ 2/
  carry IF    ld swap r> 256* or swap !
        ELSE  r> $FF and swap !
        THEN
  idebuf_ptr ld swap 1+
  dup [ #idebuf_top 2* ] literal u>
  IF  idebuf_flush drop #idebuf_addr 2*  THEN
  swap !
;
: buf16! ( 16bit -- ) wsplit buf8!  buf8!  ;

: buf24! ( 24bit -- ) split  buf8!  buf16! ;

: buf32! ( 32bit -- ) split  buf16! buf16! ;
```

These four words take care of writing the variable width encoded data items to the buffer and the buffer will be written to the disk when it is full using IDEBUF_FLUSH. The current buffer location is held in variable IDEBUF_PTR.

Using these buffer storage primitives, the data may now be encoded and saved:

```
: abs  ( n -- u )   neg IF  0 swap-  THEN ;

: encode ( sample channel# -- )
  under           \ val val ch
  samples +       \ val val addr
  ld >r swap r> ! \ val old    store val in samples buffer
  under -         \ val difference
  dup abs         \ val difference |difference|
  dup $40 <   IF drop $7f and buf8!
                 drop EXIT
              THEN
  dup $1000 < IF drop $1FFF and $C000 or buf16!
                 drop EXIT
              THEN
  $80000 <    IF $FFFFF and $E00000 or buf24!
                 drop EXIT
              THEN
  drop $7FFFFFF and $F0000000 or buf32!
;
```

This mess of complex code can be drastically simplified implementing five special instructions, four primitives for storing together with an internal register and one primitive for the data coding.

```
1 POP  USR Opcode: buf8!   ( 8b -- )
2 POP  USR Opcode: buf16!  ( 16b -- )
1 NONE USR Opcode: buf24!  ( 24b -- 16b )
2 NONE USR Opcode: buf32!  ( 32b -- 16b )
```

These are the four primitive buffer storage operators that take care of the variable length data and the byte alignment problems that occur are managed by a temporary byte register inside the FPGA. BUF_POINTER is an internal register that delivers the address into the buffer memory area and it is incremented by these operations appropriately.

But what happens if the data buffer is full? The end of the data buffer is an absolute hardware constant synthesized into the FPGA. Therefore, the FPGA "knows" when it attempts to increment the BUF_POINTER past the buffer end. At that moment, `#sema_buf` (see: Semaphores above) is reset, which unlocks the task that writes the buffer to the disk and an exception is raised when one of the four buffer storage operators is executed. Once the complete buffer has been written to disk, `#idebuf_addr BUF_POINTER !` will set the buffer pointer back to the beginning of the buffer memory area and it will set `#sema_buf` again, locking the "writeback" task and unlocking the buffer storage operators again at the same time.

Data coding is handled by just one single instruction TAG, which is much easier to implement in VHDL than its uCore Forth counterpart.

```
ENCODE NONE ALU Opcode: tag  ( sample previous -- sample code )
```

TAG expects the current and the previous sample on the stack and it produces the code according to the encoding rules. The information whether the code constitutes a byte, a 16 bit or a 32 bit word is encoded in the carry and overflow flags. Piecing these special instructions together results in a simply elegant ENCODE routine:

```
: encode ( val ch# -- )
   samples + ld >r tag
   carry IF  buf8!  ELSE  ovl IF  buf32!  THEN  buf16!  THEN
   r> !
;
```

This is a real power saver because it allows to run the processor at one fourth its previous speed, reducing the dynamic power consumption for the digital control by a factor of 2 to 3.

5 Lessons Learned

The migration from uCore 1.10 (as published on www.microcore.org) to uCore 1.20 was still a simulation exercise without really putting the code to the hardware test. Nevertheless, the integration of a "top of return stack" register, which was driven by the implementation of the complex math step operators made it clear that there was something fundamentally wrong with the way that the data memory/return stack was connected to the core. uCore 1.20 is the result of a major restructuring of the memory interface. Now it is simple to use the data memory interface to interface to both internal registers and external interface chips in much the same way that we are accustomed to in conventional processors.

Still, uCore 1.20 was a simulated "paper exercise" but it served as a viable basis to instantiate uCore on the prototyping board.

Porting this design to the Geolon-MCS hardware and gaining experience with it in solving a real problems led to a number of internal changes that have mainly to do with factoring the VHDL code to ease adaptation to application specific requirements. "Porting this code back" to a generalized uCore 1.30 has yet to be done.

But one of the key lessons learned in these exercises does sound familiar to every software engineer:

Don't try to anticipate generality when solving a specific problem. The evolving reality tends to be completely different from your imaginations. Just solve the problem at hand in as readable a manner as possible. You will have to re-write the software anyway when adding new functionality. This is true for VHDL code as well.