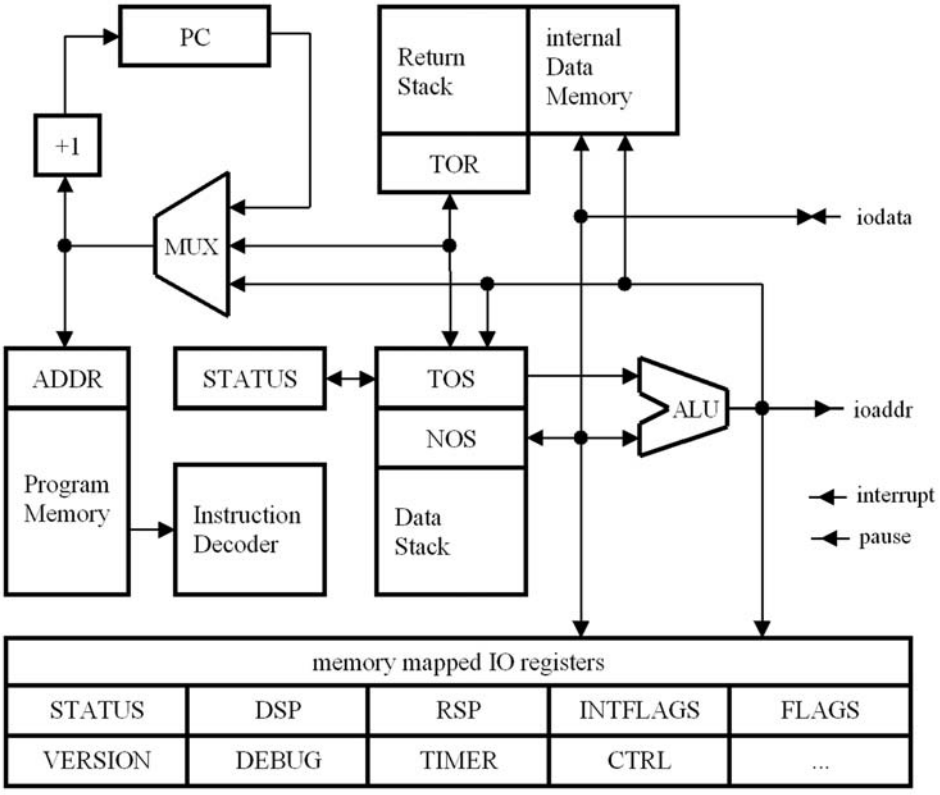
µCore Overview

µCore (pronounced "microCore") is a processor written in VHDL for synthesis into FPGAs. It is optimized for embedded real time control. µCore is an embodiment of the virtual machine of the Forth programming language [1] and hence Forth is µCore's assembler.

Characteristics

1. Dual stack (data stack & return stack)
2. Harvard architecture (8 bit program memory & independent data memory)
3. Configurable word width for data memory & stacks
4. Postfix instruction set architecture (literal operands/addresses precede rather than follow instructions that need them in the instruction stream)
5. Single cycle execution, no pipelining
6. Interrupt & Pause traps
7. Multiple data and return stack areas for efficient multitasking
8. Deterministic program execution
9. Hardware/Software co-design environment
10. Malleable instruction set



µCore block diagram

μCore Overview

1 Dual Stack

μCore does not have general purpose registers, it has a "data stack" for numerical computations and a "return stack" for program flow control.

The data stack resides in a private RAM area managed by the DataStackPointer (DSP). Its top two items are held in registers (TopOfStack - TOS and NextOfStack - NOS).

The return stack is mapped into the data memory managed by the ReturnStackPointer (RSP). Its top item is held in a register (TopOfReturnstack - TOR).

The pros-and-cons of a stack versus register architecture:

1) Parameter passing.
   Both architectures quite often suffer from the fact that the arguments for a subroutine call are not in the registers resp. not in the stack order, in which the subroutine expects them. As a result, registers have to be re-shuffled resp. the stack has to be re-ordered.
   The number of registers is finite and hence the number of arguments that may be passed to a subroutine. The stack has a finite depth in hardware as well, but if needed, the bottom part of the stack may be swapped out and in of data memory by a background process at runtime.
2) Interrupt processing
   The stack architecture holds all arguments on stacks already and therefore, no registers need to be saved and restored.
3) Code optimization
   "Register allocation" became a research topic long ago whereas "stack allocation" is an esoteric topic. But three generations of research happened and the results are already of production quality.

   When μCore's assembler μForth is used, the programmer her- or himself is responsible for stack allocation and stack re-ordering.
4) Instruction set impact
   A register architecture needs address bits in the instruction word whereas a stack architecture does without. A subroutine always takes its arguments from the top of stack downwards and leaves its results on the stack.

Looking at the technical merits, the stack architecture has several advantages especially for real time applications. Its major disadvantage is its unfamiliarity.

2 Harvard Architecture

μCore's instructions and the program memory are 8 bits wide. The configurable width data memory (see below) is independent of the fixed width program memory. Two instructions, which allow writing and reading program memory, may or may not be instantiated and therefore, μCore can be made safe from corrupting its program during runtime.

During cold boot, μCore may always write into program memory and therefore, it may fetch its program from an external, non-volatile memory. The boot loader is a μForth program that is compiled before synthesis.

3 Configurable Data Memory & Stack Word Width

In μCore the word width of the data memory and the stacks is configurable and must be defined before synthesis as needed by the application. Most of the time, μCore does not deal with "bytes" but only with entire memory "cells", which are word width wide. Exceptions are 16 or 32 bit word widths, which may be instantiated with or without byte addressing.

The instantiated word width may be any odd or even number of bits. 12 bits is a practical minimum, because it limits the program memory's address range to 4096 instructions. The maximum width is only limited by the available FPGA resources. This configurability is rendered possible by μCore's postfix instruction set and Harvard architecture (see below).

4 Postfix Instruction Set Architecture

μCore's postfix instruction set extends Forth's postfix syntax into the hardware realm. The principle has been inherited from the Transputer, and it makes the configurable data word width possible.

When instructions include immediate numbers (literal, offset, address etc.), the number of bits needed for these numbers depends on the processor's data word width. In μCore, literals and opcodes are kept separate and literals precede rather than follow opcodes that need them. Numbers of any magnitude may be constructed by sequences of literal instructions.

μCore's instructions are 8 bits wide. An instruction's most significant bit determines whether it is interpreted as an opcode (MSB not set) or as a literal (MSB set). A "lit flag" in the status register is set by literals and reset by opcodes. This leads to four different cases for instruction interpretation:

lit flag	MSB	interpretation
0	0	An opcode that does not need a literal.
0	1	A literal following an opcode. The least significant 7 bits are pushed on the stack as a 2s-complement number in the range -64 .. 63, and the lit flag is set.
1	0	An opcode following a literal. If it is just a noop, the literal remains on the stack as a number and the lit flag is reset.
1	1	A literal following another literal. The literal in TOS is shifted 7 bit positions to the left and the 7 least significant bits of the literal instruction are appended. This covers the following number ranges: 1 lit -64 .. 63 2 lits -8192 .. 8191 3 lits -1048576 .. 1048575 etc. This way numbers of any magnitude can be constructed.

Opcodes do neither include literal information nor register addresses due to the stack. Therefore, each of the remaining 127 opcodes can be used for different semantics. This is plenty. The "core" opcode set has 47, the "extended" and "float" set another 29 opcodes. This leaves room for 42 application specific opcodes.

Opcodes like branch and call do even have different semantics depending on the lit flag. When it is set, they take the number in TOS as a branch offset. Otherwise, they take it as an absolute target address.

Instructions (literals as well as opcodes) are always self-contained and therefore, interrupts can be accepted after each instruction.

5 Single Cycle Execution, No Pipelining

In general, a single instruction needs one active clock transition to execute.

On a hardware level (VHDL), μCore's registers have a clock and a clock enable input. The enable input allows embedding μCore into environments with a clock frequency that is above μCore's timing requirements. Constant **cycles** in architecture_pkg.vhd defines the number of clock cycles, which are needed to execute one instruction. In addition, the top-level enable input may be used to temporarily halt μCore. Typically, μCore's worst-case asynchronous signal delay is less than 40 ns (25 MHz) for Xilinc, Altera, and Lattice FPGAs with 90 nm technology.

Reading the FPGA's internal blockRAM memories requires the execution of a sequence of two uninterruptible instructions. The first instruction latches the memory address inside the blockRAM, the second instruction pushes the memory's data output on the stack.

To this end a general mechanism allows to chain sequences of uninterruptible instructions. It can also be used for read-modify-write instructions like **+!**.

6 Interrupt & Pause Traps

Interrupts are a well known concept in computing and almost every existing processor implements it. Hardly known is its Janus-faced sibling, the Pause, another inheritance from the Transputer. A Pause will e.g. be raised by a UART, when the processor intends to read a character, which has not been received yet. The difference between interrupts and pauses is as follows:

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

On an interrupt the processor pushes the status register on the data stack, raises the InterruptInService (IIS) status bit that disables further interrupts, and executes a call to the interrupt trap location. The status register is a collection of single bit flags that characterize the processor state besides the PC.

Globally, interrupts may be enabled or disabled via the "InterruptEnable" (IE) status bit. Individual interrupts sources are enabled or disabled writing the Intflags register, which holds a flag for each interrupt source. Reading Intflags allows locating an interrupt's source(s).

Interrupt processing is terminated by the IRET instruction, which returns from the interrupt service routine and restores the status register from the data stack, thereby resetting the IIS bit again.

Pause: An event did **not** happen that was expected by the software.

On a pause the processor aborts the current instruction, which caused the pause signal to be raised, does not increment the Program Counter (PC), and executes a call to the pause trap location.

In a single task system, the pause trap would immediately execute an EXIT (return from subroutine), thereby returning to the instruction that caused the pause previously. This loop repeats until the pause signal will no longer be raised.

In a multitask system, the pause trap would call the scheduler, and another task can be given the opportunity to run. Eventually, the task that caused the pause will be activated again and continue normal execution as soon as the reason for raising the pause signal has vanished, i.e. the missing event did happen.

Using the pause mechanism, resource locking can be completely realized in hardware. E.g. an ADC with an integrated 8-channel multiplexer in a multitasking environment can then be programmed in μForth:

```
<channel_number> ADC !
ADC @ Sample !
```

or in C[1] as follows:

```
ADC = <channel_number>;
Sample = ADC;
```

Storing the <**channel_number**> into the memory mapped **ADC** interface will initiate conversion of that channel and set the ADC's semaphor flag in the flags register. Should the ADC be in use by another task, the semaphor will have been set and a pause will be raised until eventually the semaphor will have been reset. In the next line, the conversion result will be read from the **ADC**, stored in varable **Sample**, and the semaphor will be reset. Should the ADC be still busy converting when a read attempt is made, pause will be raised until eventually the ADC has finished conversion.

This way the hardware takes care of both mutual exclusion of the ADC resource as well as waiting for the AD-conversion to finish without having to test flags in software loops.

7 Multiple Data- and Return-Stack Areas for Efficient Multitasking

The number of data- and return-stack memory areas can be configured so that each task has its own set of stacks. This speeds up task scheduling, because only internal registers TOS, NOS, TOR, DSP, and RSP need to be redirected. A task switch @ 25 MHz takes 7 μsec to put the current task to sleep and start another one.

[1] A μCore back-end for the LCC C compiler has been implemented for an earlier 32 bit version of μCore. In addition, LCC itself has been modified for stack- rather than register-allocation. This work has been done at Fachhochschule Aargau, Windisch (CH), supported by the Hasler Foundation.

8 Deterministic Program Execution

μCore's program execution is fully deterministic. There is no pipeline. There is no cache memory that may have to be loaded before the next instruction can be executed. Therefore, μCore's time to execute a certain sequence of instructions is predictable, which is a prerequisite for verifyable real-time systems.

9 Hardware/Software Co-Design Environment

μCore's design environment consists of the following elements:

- VHDL source code for μCore on a target system,
- the μForth cross-compiler and debugger running on a host computer,
- an RS232 umbilical link that connects host and target,
- an interactive command line interpreter to inspect and control the target,
- a disassembler,
- and a single-step tracer.

Both the VHDL hardware description as well as the μForth cross-compiler share a common file **architecture_pkg.vhd**, which characterizes μCore's architecture and opcodes. Therefore, the cross-compiler will always be in-sync with the hardware description.

The cross-compiler is able to produce program memory initialization code for the VHDL simulator and therefore, μForth program execution can be observed in the VHDL simulator.

A umbilical link consisting of a 2-wire UART (rxd, txd) connects μCore's debugger with the mating debugger on the host and allows to

- load object code into μCore for execution,
- control μCore interactively Forth style from a command line,
- display the data stack and dump data memory to the host's display,
- upload/download data memory areas without interfering with μCore's program execution,
- single step code on a subroutine level displaying the data stack at each step.

10 Malleable Instruction Set

μCore's instruction set consists of 52 "core", 22 "extended", 5 "float" , and 5 "byte" instructions. In addition, 26 "software traps" are available, which do a single cycle call to fixed program memory locations.
44 opcodes are unused or software traps and may be used for application specific instructions.

In the core version, the extended instructions will be emulated by macros or subroutines. Therefore, a core version has the same functionality as the extended version but it runs slower consuming fewer FPGA resources.

Adding a new opcode to μCore is rather simple and consists of three steps:

1. Define the binary code for **op_newname** as a constant in **architecture_pkg.vhd**.
2. Add a new WHEN clause to the instruction decoder's CASE statement in **uCntrl.vhd** for the semantics of **op_newname**.
3. Define a name for **op_newname** in **opcodes.fs** to make it known to μForth.

μCore Overview

The core instruction set

Data stack:	drop, dup, ?dup, swap, over, rot
Return stack:	>r, r>, r@
Branches:	branch, 0=branch, next, call, exit, iret
Data memory:	ld, st
Unary arithmetic:	not, 0=, 0<, mshift, mashift
Binary arithmetic:	+, +c, -, swap-, and, or, xor, um*, *, um/mod
Flags:	status-set, ovfl?, carry?, time?, <
Traps:	reset, interrupt, pause, break, dodoes, data!

11 Implementation

μCore has been ported to these 90 nm FPGA families:
Xilinx (XC2S), Lattice (XP2), Altera (EP2), and Actel/Microsemi (A3PE).

Reference implementations on a Lattice LFXP2-8 prototyping board with minimal external IO and hardware multiplier have been made with different μCore configurations. The clock's timing constraint in the synthesizer and in the place-and-route tool had been set to 25 MHz.

Instruction set	word width	SLICES	data memory	program memory	maximum clock
core	16	988	6k	8k	33 MHz
extended	16	1199	6k	8k	30 MHz
core	27	1259	4k	8k	33 MHz
extended	27	1608	4k	8k	28 MHz
extended and floating point	27	1808	4k	8k	26 MHz
core	32	1432	3k	8k	33 MHz

Literature

[1] Leo Brodie: "Thinking Forth", http://thinking-forth.sourceforge.net/