MicroCore

a
scalable,
dual Stack,
Harvard Processor
for embedded Control
that fits into FPGAs easily

Klaus.Schleisiek AT hamburg.de

Using an FPGA based simple and extensible processor core as the foundation of a system eventually frees the programmer from the limitations of any static processor architecture, be it CISC, RISC, WISC, FRISC or otherwise. No more programming around known hardware bugs. A choice can be made as to whether a needed functionality should be implemented in hardware or software; simply, the least complex, most energy efficient solution can be realised while working on a specific application. Of course, using FPGAs is a hefty blow for MIPS ratings. But, building on an FPGA, time critical and perhaps complex functions can be realised in hardware in exactly the way needed by the application offloading the processor from sub-optimal inner software loops.

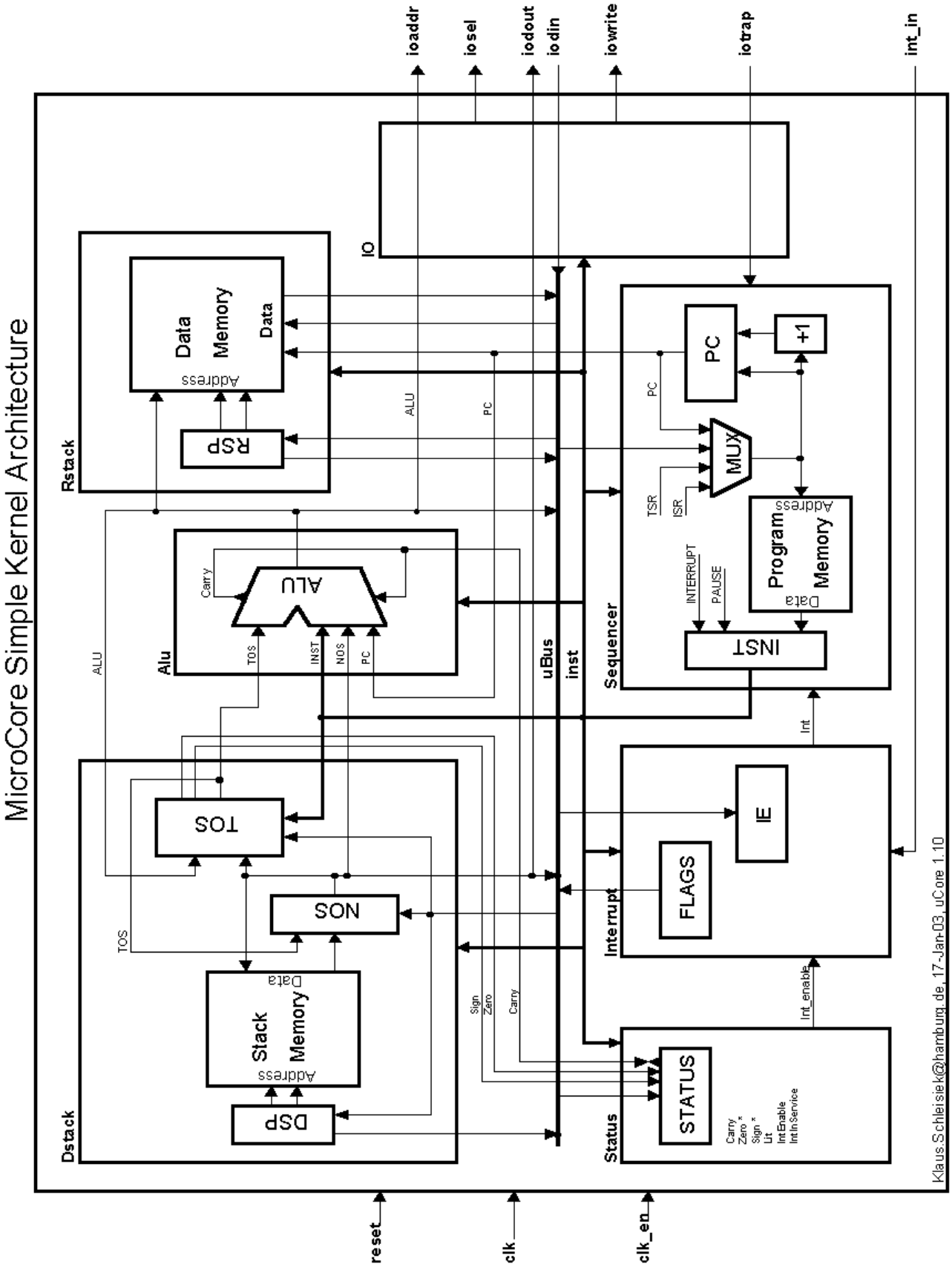
The FPGA approach also makes the user independent from product discontinuity problems that haunt the hi-rel industry since the dawn of the silicon age. Finally: putting the core into FPGAs puts an end to one of the high-level programming language paradigms, namely the aspect of (hoped-for) portability. Once I can realise my own instruction set, I am no longer confronted with the need to port the application to any different architecture and henceforth, the only reason to adhere to a conventional programming style is the need to find maintenance programmers. Remains the need for a vendor independent hardware description language to be portable w.r.t. any specific FPGA vendor and family. To date, MicroCore has been realised in VHDL, using the MTI simulator and the Synplify and Leonardo synthesisers targeting Xilinx and Altera FPGAs. For clarity, VHDL declarations are appended to this paper to define the basics of the MicroCore architecture. For more and up-to-date information, please refer to "www.microcore.org".

MicroCore is not confined to executing Forth programs but it is rooted in the Forth virtual machine. MicroCore has been designed to support Forth as its "Assembler". Support for local variables (relative return-stack addressing) is cheap and seems to be all that is needed to soup up MicroCore for C. Its fitness for Java needs to be explored.

Hamburg, 18-Nov-01.

Klaus Schleisiek

Table of Content


MicroCore Simple Kernel Architecture
ioaddr
iosel
iodout
iodin
iowrite
iotrap
int_in
IO
Rstack
Data Memory
Address
Data
RSP
ALU
PC
Alu
Carry
ALU
TOS
INST
NOS
PC
uBus
inst
Sequencer
PC
+1
MUX
TSR
ISR
Program Memory
Address
Data
INTERRUPT
PAUSE
INST
Int
Interrupt
IE
FLAGS
Int_enable
Dstack
TOS
NOS
TOS
Stack Memory
Address
Data
DSP
Sign
Zero
Carry
Status
STATUS
Carry
Zero *
Sign *
Ut
IntEnable
IntInService
Klaus.Schleisiek@hamburg.de, 17-Jan-03, uCore 1.10
reset
clk
clk_en

1 Hardware Architecture

MicroCore is a dual-stack, Harvard architecture with three memory areas that can be accessed in parallel: Data-stack (RAM), Data-memory and return-stack (RAM), and Program-memory (ROM).

The architecture diagram shows all busses that are needed. Each entity generates its own control signals from the current instruction INST and the STATUS register.

All instructions without exception are 8-bits wide, and they are stored in the program-memory ROM. Due to the way literal values can be concatenated from sequences of literal instructions, all data-paths and memories are scalable to any word width without any change in the object code as long as the magnitude of the numbers processed are representable. In essence, on a given object code the processor performs arithmetic modulo the synthesised data path width.

The data paths are made up of the data-stack Dstack, the ALU and of the data-memory and return-stack Rstack as well as of uBus, the registers NOS (Next-Of-Stack) and TOS (Top-Of-Stack), that are in between the data-stack and the ALU, and the ioBus.

The data-stack is realised by a RAM used as Stack under control of the Data-Stack-Pointer DSP, and the two topmost stack items are held in registers NOS and TOS. Most often the size of the Stack Memory needed will be small enough to fit inside the FPGA.

IO is data-memory mapped and the most significant address bit selects the outer world when set. When not set, it selects data-memory and return-stack RAM. The return-stack occupies the upper end of Data Memory under control of Return-Stack-Pointer RSP.

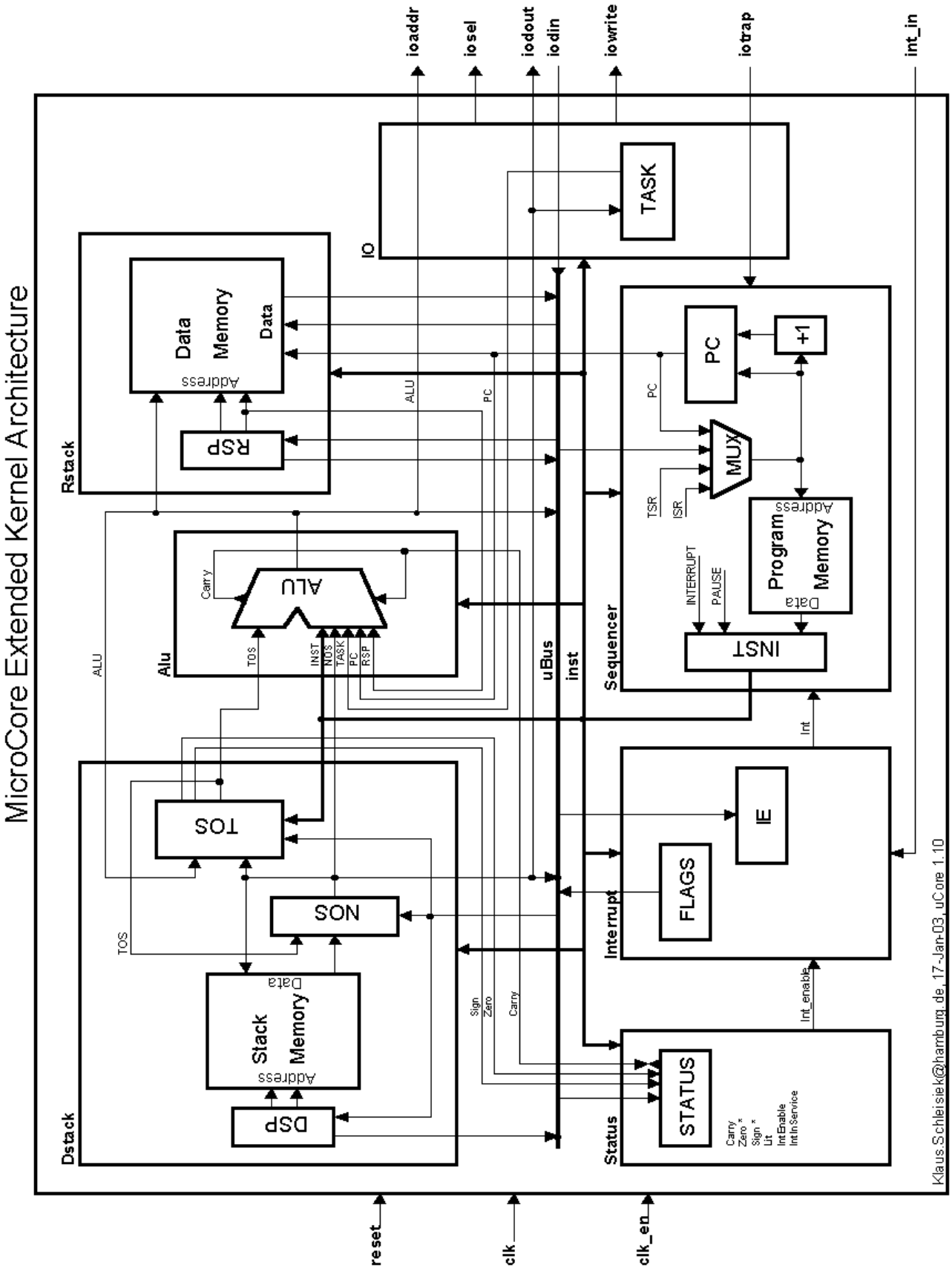
The Sequencer generates the Program Memory address for the next instruction, which can have a number of sources:

- The Program Counter PC for a sequential instruction,
- the ALU for a relative branch or call,
- the TOS register for an absolute branch or call,
- the Data Memory for a return instruction,
- the fixed Interrupt Service Routine address ISR as part of an interrupt acknowledge cycle, or
- the fixed Trap Service Routine address TSR for the IOTRAP trap signal or the PAUSE instruction.

The STATUS register has been shown as a separate entity. In the code however, it is composed of status bits generated from several sources and therefore, it is spread across the entire design as stBus.

The Interrupt Processing unit takes care of synchronising and masking a scalable number of static external interrupt sources.

Both, instruction decoding and status register bit processing has been decentralised because it makes the code easier to understand, maintain, modify, and extend.


MicroCore Extended Kernel Architecture
ioaddr
iosel
iodout
iodin
iowrite
iotrap
int_in
IO
TASK
Rstack
Data Memory
Address
Data
RSP
ALU
PC
Alu
Carry
ALU
TOS
INST
NOS
TASK
PC
RSP
uBus
inst
Sequencer
PC
+1
MUX
TSR
ISR
Program Memory
Address
Data
INTERRUPT
PAUSE
INST
Int
Dstack
TOS
NOS
Stack Memory
Address
Data
DSP
Interrupt
IE
FLAGS
Int_enable
Status
STATUS
Carry
Zero *
Sign *
Lit
IntEnable
IntInService
Sign
Zero
Carry
Klaus.Schleisiek@hamburg.de, 17-Jan-03, uCore 1.10
reset
clk
clk_en

2 Instruction Architecture

Each instruction is always 8 bits wide. Scalability is achieved on the object code level because all literal values are composed from literal instructions that can be concatenated. Refer to [3 instruction structures] for a discussion of the literal representation used, which is characterised by its "prefix" nature dubbed "Vertical instruction set with literal prefixes" in the paper. To my knowledge, this type of code has been invented by David May for the Transputer.

It has two advantages and one drawback compared to other instruction set structures:

Each instruction is "self contained" and therefore, this type of code can be interrupted between any two instructions, simplifying interrupt hardware and minimising interrupt latency to the max.

Long literals can be composed of a sequence of literal instructions that are concatenated in the TOS register. Therefore, this type of instruction architecture is independent of the data-word width.

Prefix code has the highest instruction fetch rate compared to the two other instruction types discussed in the paper. Therefore, it is not really the technology of choice for demanding real-time applications. A way out would be to fetch several instructions per memory access but that introduces unpleasant complexity for branch destinations.

Keeping in mind that MicroCore is about putting a very simple and small processor core into FPGAs for simple, embedded control, the latter drawback is tolerable because the instruction fetch delay, even using external ROM, will hardly dominate total processor delay because all processor logic will be contained in an FPGA and therefore, it will be substantially slower than an ASIC implementation anyway.

The instruction

7 $80	6 $40	5 $20	4 $10	3 $8	2 $4	1 $2	0 $1
Lit/Op	Type		Stack		Group		

2.1 Lit/Op Bit

1: 7-bit Literal (signed)
0: 7-bit Opcode

The Lit/Op field is a semantic switch:

When set, the remaining 7 bits are interpreted as a literal nibble and transferred to the Top-of-Stack (TOS) register. When the previous instruction had been an opcode, the literal nibble is sign-extended and pushed on the stack. If its predecessor was a literal nibble as well, the 7 bits are shifted into TOS from the right. Therefore, the number of literal nibbles needed to represent a number depends on its absolute magnitude.

When not set, the remaining 7 bits are interpreted as an opcode. Opcodes are composed of three sub-fields whose semantics are almost orthogonal: Type, Stack, and Group. Not all possible bit combinations of these fields have a meaningful semantic easing instruction decoding complexity.

2.2 Type field

Code	Name	Action
00	BRA	Branches, Calls and Returns
01	ALU	Binary and Unary Operators
10	MEM	Data-Memory and Register access
11	USR	Not used by core, free for user extensions

BRAnches are conditioned by the group field and they consume the content of TOS, using either TOS or TOS+PC as destination address. Although elegant, the fact that each branch has to pop the stack to get rid of the destination address makes the implementation of Forth's IF, WHILE, and UNTIL complicated. (N.B. This has been the most challenging problem for the Forth cross-compiler). Calls push the content of the PC on the return-stack while branching. Returns pop the return-stack using it as the address of the next instruction.

ALU instructions use the stack as source and destination for arithmetic operations. Unary operations only use TOS, binary operations use TOS and Next-of-Stack (NOS) storing the result in TOS.

MEMory instructions refer to the data memory when the most significant bit of TOS, which holds the address, is not set. When set, it refers to input/output operations with the outer world. The return-stack occupies the upper end of data-memory. Eight registers can be accessed directly using the Group field.

32 USeR instructions are free for any application specific functions, which are needed to achieve total system throughput.

2.3 Stack field

Code	Name	Action
00	NONE	Type dependent
01	POP	Stack->NOS->TOS
10	PUSH	TOS->NOS->Stack
11	BOTH	Type dependent

POP pops and PUSH pushes the data stack. The stack semantics of the remaining states NONE and BOTH depend on type and on external signals INT_IN and IOTRAP. This is where the opcode fields are non-orthogonal creating instruction decoding complexity, which is gracefully hidden by the synthesiser.

2.4 Group field

Code	Binary-Ops ALU	Unary-Ops ALU	Conditions BRA	Registers MEM
000	ADD	NOT	NEVER	STATUS
001	ADC	SL	ZERO	TOR
010	SUB	ASR	NSIGN	RSTACK
011	SBC	LSR	NCARRY	LOCAL
100	AND	ROR	PAUSE	RSP
101	OR	ROL	INT	DSP
110	XOR	ZEQU	NZERO	TASK
111	NOS	CC	ALWAYS	FLAGS / IE

The semantics of the group field depend on the type field and in the case of ALU also on the stack field.

Of the binary operators NOS is used to realise SWAP and OVER.

Unary operations are detailed below.

Of the conditions, NEVER is used to realise NOP, DUP and DROP. NZERO supports the use of the Top-Of-Return-stack as a loop index. PAUSE and INT are conditions to aid in processing external events INT_IN and IOTRAP.

Of the registers, TOR is used to implement R@, whereas RSTACK implements >R and R>.

3 Instruction Semantics

In the following tables the LIT-field is marked with - and +.

This indicates the following two cases:
‘-’: The previous instruction has also been an opcode; TOS holds the top-of-stack value.
‘+’: The previous instruction(s) have been literals; TOS holds a "fresh" literal value.

3.1 BRA instructions

LIT	Stack	act	Operation	Forth operators / phrases

*	none	none	conditional return from subroutine
			When Cond=ZERO or NZERO
			Stack -> NOS -> TOS
			When Cond=INT
			Stack -> NOS -> TOS -> STATUS
				`EXIT NOP`
				`IRET`
				`?EXIT`
				`0=EXIT`

-	pop		conditional branch to Program[TOS] Stack -> NOS -> TOS	`absolute_BRANCH` `DROP`
+	pop		conditional branch to Program[PC+TOS] Stack -> NOS -> TOS	`relative_BRANCH`
*	push		TOS -> TOS -> NOS -> Stack	`DUP`

-	both	pop
		push
			conditional call to Program[TOS]
			Stack -> NOS -> TOS
			Except when Cond=INT or PAUSE
			Call to Program[ISR] or Program[TSR]
			STATUS -> TOS -> NOS -> Stack
				`absolute_CALL`
				
				`INTERRUPT`
				`PAUSE`

+	both	pop
		push
			conditional call to Program[PC+TOS]
			Stack -> NOS -> TOS
			Except when Cond=INT or PAUSE
			Call to Program[ISR] or Program[TSR]
			STATUS -> TOS -> NOS -> Stack
				`relative_CALL`
				
				`INTERRUPT`
				`PAUSE`

3.2 ALU instructions

Stack	act	Operation	Forth operators / phrases
none	none	NOS <op> TOS -> TOS	`OVER_SWAP_- SWAP`
pop		Stack -> NOS <op> TOS -> TOS	`+ - AND OR XOR DROP`
push		NOS <op> TOS -> TOS TOS -> NOS -> Stack	`2DUP_+ OVER`
both	none	TOS <uop> -> TOS	`0= 2* ROR ROL 2/ u2/`

3.3 MEM instructions

Stack	act	Operation	Forth operators / phrases
none	pop	Stack -> NOS -> TOS -> Register LOCAL := Stack -> NOS -> Data[RSP+TOS] TASK := Stack -> NOS -> Data[TASK+TOS]	`>R` store into local variables store into task variables
pop		Stack -> NOS -> Data[TOS+<inc>] TOS + <inc> -> TOS	`++!` pre-incrementing data memory or I/O store
push		Data[TOS+<inc>] -> NOS -> Stack TOS + <inc> -> TOS	`++@` pre-incrementing data memory or I/O fetch
both	push	Register -> TOS -> NOS -> Stack LOCAL := Data[RSP+TOS] -> NOS -> Stack TASK := Data[TASK+TOS] -> NOS -> Stack	`R@, R>` fetch from local variables fetch from task variables

4 Basic Forth Operations

A single instruction is composed of a triple of a type, stack, and group mnemonic. The following table lists often used Forth atoms and their realisation using the MicroCore instruction architecture.

Forth	LIT	Implementation	Remarks
NOP	-	BRA NONE NEVER	
>R	-	MEM NONE RSTACK	
R>	-	MEM BOTH RSTACK	
R@	-	MEM BOTH TOR	
DUP	-	BRA PUSH NEVER	
DROP	-	BRA POP NEVER	
SWAP	-	ALU NONE NOS	
OVER	-	ALU PUSH NOS	
N ++@	-	MEM PUSH N, pre-increment	N = signed 3-bit number
@	x	MEM PUSH 0 ALU POP NOS	2-cycle
N ++!	-	MEM POP N, pre-increment	N = signed 3-bit number
!	x	MEM POP 0 ALU POP NOS	2-cycle
+	-	ALU POP ADD	
-	-	ALU POP SUB	
AND	-	ALU POP AND	
OR	-	ALU POP OR	
XOR	-	ALU POP XOR	
INVERT	-	ALU BOTH NOT	
2*	-	ALU BOTH SL	
2/	-	ALU BOTH ASR	
u2/	-	ALU BOTH LSR	
CALL	+	BRA BOTH ALWAYS	absolute 3-cycle, relative 2-cycle
EXIT	-	BRA NONE ALWAYS	
?EXIT	-	BRA NONE ZERO	
BRANCH	+	BRA POP ALWAYS	absolute 3-cycle, relative 2-cycle
?BRANCH	+	BRA POP ZERO ALU POP NOS	additional DROP needed in both cases for Forth's IF
NEXT	+	BRA POP NZERO	used for FOR ... NEXT loop
LITERAL	+	no additional instruction needed, just loading LIT-nibbles in succession. When LIT=0 a Literal-nibble triggers a PUSH operation and initialises TOS. When LIT=1, the Literal-nibble is shifted into TOS.	#-cycles depending on its magnitude - fragmented into 7-bit nibbles
1+	1	ALU NONE ADD	2-cycle
1-	-1	ALU NONE ADD	2-cycle
0=	-	ALU BOTH ZEQU	
=	-	ALU POP SUB ALU BOTH ZEQU	2-cycle

Now we have about 30 meaningful Forth instructions and many opportunities for peephole optimisation across the two preceding instructions (in order to detect e.g. "OVER OVER <op>"). In addition, there are additional useful opcodes like NC-BRANCH and NS-BRANCH, which are usually not present in Forth.

5 Core Registers

5.1 STATUS

Bit	Name	Access	Description
0	C	R/W	The Carry-Flag reflects the result of the most recent ADD, SUB, ADC, SBC, SL, ASR, LSR, ROR, and ROL instructions. On subtraction, it is the complement of the borrow bit.
1	IE	R/W	Interrupt-Enable-Flag
2	IIS	R/W	The Interrupt-In-Service-Flag is set at the beginning of an interrupt-acknowledge cycle. It is reset by the IRET (Interrupt-RETurn) instruction. When IIS is set, interrupts are disabled. When the Status-register is read, IIS always reads as '0'.
3	LIT	R	The LITeral-Status-Flag reflects the most significant bit of the previous instruction.
4	N	R	The Negative-Flag reflects the content of the most-significant-bit of TOS or of NOS when LIT=1
5	Z	R	The Zero-Flag reflects the content of TOS or of NOS when LIT=1

Z and N reflect the actual state of the top "number" on the stack. This may be in TOS (when LIT=0) or in NOS (when LIT=1) because e.g. a target address may be in TOS.

For the ordering of the bits it has been taken into consideration that "masks" for masking off flags can be loaded with only one literal nibble. This is important for the C- and IE-flags, see below.

5.2 TOR

Top-Of-Return-stack. This allows access to the return-stack without pushing or popping it.

5.3 RSTACK

Return-STACK. When RSTACK is used as a destination, a return-stack push is performed. When it is used as a source, a return-stack pop is performed.

5.4 LOCAL

This register-addressing mode (MEM NONE LOCAL and MEM BOTH LOCAL) is included in order to support C and its local local variable mentality. These can be placed in a return-stack frame. The actual data memory address is the sum of RSP+TOS and this "addressing mode" is the only method for access into the return-stack, because its address range doubles as memory mapped I/O.

5.5 DSP

Data-Stack-Pointer. It is used to implement the data-stack and it can be read and written to support multitasking.

5.6 RSP

Return-Stack-Pointer. It is used to implement the return-stack that is located at the upper end of the data memory and it can be read and written to support multitasking and stack-frame linkage.

5.7 FLAGS (read) / IE (write)

This is a pair of registers – FLAGS for reading, IE (Interrupt Enable) for writing.

An interrupt condition exists as long as any bit in FLAGS is set whose corresponding bit in IE has been set previously. Interrupt processing will be performed when the processor is not already executing an interrupt (IIS-status-bit not set) and interrupts are enabled (IE-status-bit set).

Typically at the beginning of interrupt processing (after calling the hard-wired interrupt handler address ISR) the FLAGS-register will be read. One specific bit is associated with each potential interrupt source. When a certain interrupt has been asserted, its associated bit will be set. All interrupts are static and therefore, it is the responsibility of the interrupt service routine (ISR) of a specific interrupt to reset the interrupt signal of the external hardware before the end of the ISR.

IE (Interrupt Enable) is a register, which can only be written, and it holds one enable bit for each interrupt source. Setting or resetting interrupt enable bits is done in a peculiar way, which could be called "bit-wise writing":

When IE is written, the least significant bit determines whether individual IE-bits will be set ('1') or reset ('0'). All other bits written to IE select those enable bits, which will be affected by the write operation. Those bits that are set ('1') will be written to, those bits that are not set ('0') will not be changed at all. This way individual interrupt enable bits may be changed in a single cycle without affecting other IE-bits and without the need to use a "shadow variable".

5.8 TASK

The TASK register can be read and written via memory mapped I/O (address = -1). It holds an address that points at the Task Description Block (TDB) of the active task. The implementation of the multitasking mechanism is operating system dependent. Variables that are local to a task can be accessed via the MEM NONE TASK (store) and MEM BOTH TASK (fetch) instructions. It works similar to ++@ and ++!. However, the data memory address is the sum of TASK+TOS.

If the TASK register is not used for multitasking support, it constitutes a general base register for a pre-incrementing base-offset addressing mode.

6 Unary operations

SL 0 -> LSB, MSB -> C

ASR MSB -> MSB-1, LSB -> C

LSR 0 -> MSB, LSB -> C

ROR C -> MSB, LSB -> C

ROL C -> LSB, MSB -> C

ZEQU When TOS=0, TOS <- -1 otherwise TOS <- 0
(A "luxury", because it can be synthesised using the ?BRANCH instruction but it is an often used instruction in condition flag computations)

CC ComplementCarry Carry <- not Carry

7 Booting

Given MicroCore's hardware architecture, this is a very simple:

A RESET signal resets all registers to zero. Because the code for a NOP { BRA NONE NEVER } happens to be all zeros, the processor just fetches the instruction pointed to by the PC register (which had also been reset to zero) in the first cycle. Therefore, the reset vector happens to be at memory address zero.

8 Interrupts

8.1 The Interrupt Mechanism

At first, interrupt requests are synchronised.
In the succeeding cycle(s) the following mechanism will unfold by hardware design:

1st cycle:

The current program memory address will be loaded into the PC un-incremented.

The instruction BRA BOTH INT will be loaded into the INST register instead of the output of the program memory.

2nd cycle:

Now, BRA BOTH INT will be executed that performs a CALL to the ISR-address, which is a constant address, selected by the program address multiplexer and STATUS is pushed on the data stack at the same time.

Therefore, only the first INT-cycle must be performed by special hardware. The second cycle (INT-instruction) is executed by an instruction that is forced into the INST register during the first Interrupt acknowledge cycle.

8.2 Handling Multiple Interrupt Sources

Whenever an interrupt source whose corresponding interrupt enable bit is set in the IE-register is asserted its associated bit in the FLAGS-register will be set and an interrupt condition exists. An interrupt acknowledge cycle will be executed when the processor is not currently executing an interrupt (IIS-bit not set) and interrupts are globally enabled (IE-bit of the STATUS-register set).

Please note that neither the call to the ISR-address nor reading the FLAGS-register will clear the FLAGS register. It is the responsibility of each single interrupt server to reset its interrupt signal in the external hardware as part of its interrupt service routine.

9 Multitasking

The transputer has been a very innovative processor, which focused on multitasking that was entirely realised in hardware. Nice as this feature and the underlying philosophy of its programming language Occam may be, it crippled the transputer for traditional programming languages. This in turn did make the transputer difficult to understand and market. It never became really popular although its users were happy with it.

Nevertheless, hardware support for multitasking seems to be an attractive feature greatly simplifying software engineering for complex systems. Analysing the real needs w.r.t. multitasking support it occurred to me that a full-blown task switch mechanism in hardware is not really needed. Instead, a mechanism that would allow to access resources that may not be ready yet using fetch and store without the need to explicitly query associated status flags beforehand is all that is needed to hide multitasking pains from the application programmer.

Therefore, MicroCore has a PAUSE instruction and a TRAP mechanism to support multitasking or, to be less ambitious, to deal with busy resources. Fortunately, it turned out that the implementation of this mechanism in MicroCore comes almost for free and therefore, it is build into the core from the very beginning. If not used for multitasking, it is a nice basis for a breakpoint debugger.

9.1 TRAP signal

An additional external control signal has been added: TRAP. When the processor intends to access a resource, the resource may not be ready yet. In such an event, it can assert the TRAP signal before the end of the current execution cycle (before the rising CLK edge). This disables latching of the next processor state in all registers but the INST register that loads the PAUSE instruction (BRA BOTH PAUSE) instead of the next instruction from program memory.

In the next processor cycle, BRA BOTH PAUSE will be executed calling the TSR-address (Task Service Routine). Similar to an interrupt, the STATUS register is pushed on the data stack at the same time.

The TSR-address will typically hold a branch to code, which will perform a task switch depending on the operating system. Please note that the return address pushed on the return-stack is the address of the instruction following the one that caused the TRAP. Therefore, before re-activating the trapped task again, the return address on the return-stack must be decremented by one prior to executing the IRET instruction (BRA NONE INT) in order to re-execute the instruction, which caused the trap previously. Please note that no other parameter reconstruction operation prior to re-execution has to be made because the TRAP cycle fully preserves all registers but the INST register.

The TRAP mechanism is independent from the interrupt mechanism. It adds one cycle of delay to an interrupt acknowledge when both an interrupt request and a TRAP signal coincide.

In essence, the TRAP mechanism allows to access external resources without having to query status bits to ascertain the availability/readiness of a resource. This greatly simplifies the software needed for e.g. serial channels for communicating with external devices or processes.

10 Data Memory Access

On a data memory access, TOS holds the base address and NOS holds/receives the value to be exchanged with memory. Pre-incrementing access operators `++@` and `++!` have been defined for a pre-incrementing implementation. The group field is used as a signed increment spanning the range from -4 .. 3 and after the memory access, the incremented address remains in TOS.

Alternatively, relative addressing into the return-stack may be used using the LOCAL "register". The actual memory address is the output of the ALU-adder, adding the offset in TOS and the RSP. After the memory access, TOS holds the physical address (pointer) of the memory access. As a further alternative, relative addressing into the data memory can be performed relative to the TASK register that points to the beginning of a block of memory that may e.g. hold variables that are local to a task.

11 Software Development

An interactive software development environment for MicroCore is rather straightforward and in essence, it has been realised before when I worked on the IX1 field bus processor.

A "debuggable MicroCore" has an additional Centronics interface, which connects to a PC serving as the host. (Yes, I know, a USB port would have been more appropriate nowadays..) The program memory, which must be realised as a RAM, can be loaded across this interface. After loading the application, a very simple debug kernel takes control exchanging messages with a host computer.

11.1 Forth Cross-Compiler

It loads on top of Win32Forth because that is a free 32-bit system. It produces a binary image for the program memory as well as a VHDL file, which behaves as the program memory in a VHDL simulation. (Please note that the cross-compiler is implemented in such a way that it only supports literals up to 31 bits signed magnitude.)

It is a short but rather complex piece of code and my 4th iteration on implementing a Forth cross-compiler in Forth.

The most challenging aspect was compiling MicroCore's branches, which, as relative branches, are preceded by a variable number of literal nibbles. The cross-compiler at first tries to get away with one literal nibble for the branch offset. If it turns out that this is not sufficient space for the branch offset at the closing ELSE, THEN, UNTIL, or REPEAT, the source code is re-interpreted again, leaving space for the required number of literal nibbles in front of the branch opcode.

Another challenge is compiling Forth's IF, WHILE, and UNTIL because, after the branch, you still have the flag dangling on the stack. Therefore, a DROP has to be inserted after the IF and the THEN

in an IF ... THEN phrase. There's still a lot of room for optimisations and the compiled code sometimes looks sub-optimal with sequences of DUP DROP sprinkled across the code as an indication of a BEGIN ... UNTIL backward branching loops.

11.2 C Cross-Compiler

A first implementation has been realised for an earlier version of MicroCore at the technical university of Brugg/Windisch, Switzerland. The compiler is based on the LCC compiler, and a MicroCore back-end was created that takes the syntax tree as input.

It turned out that the LOCAL addressing mode is all that is needed to come to grips with C's local variable mentality. Actually, the LOCAL addressing mode does not really add any additional functionality to the core but it is a mechanism to come to grips with state-of-the-art C-compiler technology. The LOCAL addressing mode with its additional hardware consumption could go away as soon as an optimiser has been realised that is capable of transforming local variable accesses into appropriate data-stack manipulations.

12 Project Status

The VHDL code has been released. The identical code could be synthesised using the Synplify and the Leonardo synthesisers targeting Xilinx and Altera FPGAs. A simulator based on C code is in a prototype stage

Implementation experiments on Xilinx XC4000/Spartan devices reveal routing difficulties. But MicroCore can be easily realised on the Virtex family. Here are some implementation results for the Xilinx XCV50-4. This includes synthesis and place&route. Please note that the architecture is fully scalable: Any data word width you like - but less than 12 bits does not make sense. For the synthesis example, MicroCore has been compiled for an internal data-stack 16 elements deep, a return-stack 256 elements deep, an external program ROM and an external data memory RAM and two interrupt sources. Most examples are for multiplexed busses. To compute maximum clock frequency, add worst case access delay of your RAM and ROM to the specified delay predictions of the synthesiser. The synthesis result is for minimal gate count and no timing optimisation.

data path	RAM	ROM	simple		extended	
width	size	size	% used	delay [ns]	% used	delay [ns]
12	2k	4k	33	40.0	36	48.0
16	32k	64k	40	39.0	45	53.0
16 tri_state	32k	64k	34	46.0	39	57.0
24	64k	64k	51	44.0	55	58.0
24	1M	2M	53	42.0	58	55.0
32	1M	2M	64	44.5	69	60.0
32 tri_state	1M	2M	54	50.0	61	62.0

The Forth cross-compiler is operational for up to 31 bit signed literals. It’s already of production quality. Some more effort could be spent on peephole optimisations.

The C cross-compiler is in a prototype stage producing code for an obsolete version. Another design iteration is needed.

A debug interface for MicroCore based on a Centronics port exists. It can be used as a model for a USB interface implementation.

The interactive debugger itself on the host PC has not been started yet but its basic design will be ported from the IX1 design environment.

A prototyping board has been built in the framework of another research project and awaits the first actual implementation of MicroCore. Forth Gesellschaft eV is prepared to finance a small prototyping board for a 16-bit model implementation with a USB port as debug interface.

13 Legal Issues

I have applied for a patent for the MicroCore architecture. This is not because I want to restrict access, but because I want to remain in control of it.

Since the world does not wait for yet another processor architecture, I figured that I might as well give it away for free. Therefore, MicroCore should be used in the spirit of the licensing terms of the Free Software Foundation applied to a hardware design. Actually, the terms of the MicroCore license are even more liberal than the GPL terms. The only right I reserve for myself is the exclusive right to appoint institutions that may verify conformance of derived work with the original MicroCore model.

"Open" or "Free" Software is about – well – software. MicroCore is hardware. What's the difference? The protection and control that the Free Software Foundation is able to exert on the use of its material is based on copyright protection. This gives the foundation enough power to save e.g. GNU from microsoftisation, i.e. subtle changes that will make it incompatible with the original. GNU, Linux and the rest is such an immense heap of uniquely concatenated characters that it is next to impossible to realise something close but incompatible, which would not infringe copyright.

The situation for MicroCore is radically different: As the name implies, it is simple. Once you have explained the architecture and instruction set to an experienced VHDL programmer, he will come up with an original implementation in three months or less without infringing on the copyright of the original VHDL model. This is why I have applied for patent protection.

When MicroCore catches on, I am prepared to transfer the patent rights to a public, non-profit organisation.

14 Acknowledgements

I would like to thank a number of people, without which MicroCore would be different or not exist at all, namely:
Chuck Moore, who invented Forth and pioneered Forth hardware with the design of the NC4000.
Norbert Ellenberger, who backed the design of the FRP1600 that paved the way for the IX1.
Christophe Lavarenne who introduced the Transputer innovations to me. Adolf Krüger, without whom the literal accumulator would probably still be in a separate register instead of on the stack.

15 MicroCore Basics in VHDL

```
-------------------------------------------------------------------
-- microcore bus widths
-------------------------------------------------------------------

CONSTANT data_width       : NATURAL := 12; -- from 12 .. 32 or more
CONSTANT inst_width       : NATURAL :=  8;
CONSTANT data_addr_width  : NATURAL := 11; -- "top" half address space is I/O
CONSTANT prog_addr_width  : NATURAL := 12;
CONSTANT ds_addr_width    : NATURAL :=  4;
CONSTANT rs_addr_width    : NATURAL :=  4;
CONSTANT interrupts       : NATURAL :=  2;

-------------------------------------------------------------------------
-- microcore busses
-------------------------------------------------------------------------

SUBTYPE data_bus      IS std_logic_vector(data_width-1 DOWNTO 0);
SUBTYPE inst_bus      IS std_logic_vector(inst_width-1 DOWNTO 0);
SUBTYPE data_addr     IS std_logic_vector(data_addr_width-1 DOWNTO 0);
SUBTYPE prog_addr     IS std_logic_vector(prog_addr_width-1 DOWNTO 0);
SUBTYPE ds_addr       IS std_logic_vector(ds_addr_width-1 DOWNTO 0);
SUBTYPE rs_addr       IS std_logic_vector(rs_addr_width-1 DOWNTO 0);
SUBTYPE int_bus       IS std_logic_vector(interrupts-1 DOWNTO 0);

-------------------------------------------------------------------------
-- status register
-------------------------------------------------------------------------

CONSTANT s_c_bit      : NATURAL :=  0;  -- carry bit
CONSTANT s_ie_bit     : NATURAL :=  1;  -- Interrupt Enable bit
CONSTANT s_iis_bit    : NATURAL :=  2;  -- InterruptInService bit
CONSTANT s_lit_bit    : NATURAL :=  3;  -- LIT bit of the previous instruction
CONSTANT s_n_bit      : NATURAL :=  4;  -- Sign-bit of top data element (TOS or sometimes NOS)
CONSTANT s_z_bit      : NATURAL :=  5;  -- Zero-bit of top data element (TOS or sometimes NOS)
CONSTANT status_width : NATURAL :=  6;

-------------------------------------------------------------------------
-- physical addresses
-------------------------------------------------------------------------

CONSTANT addr_isr : std_logic_vector(3 DOWNTO 0) := "0100";
CONSTANT addr_tsr : std_logic_vector(3 DOWNTO 0) := "1000";

-------------------------------------------------------------------------
-- op codes
--                    TYPE
-------------------------------------------------------------------------

CONSTANT op_BRA    : std_logic_vector(1 DOWNTO 0) := "00";
CONSTANT op_ALU    : std_logic_vector(1 DOWNTO 0) := "01";
CONSTANT op_MEM    : std_logic_vector(1 DOWNTO 0) := "10";
CONSTANT op_USR    : std_logic_vector(1 DOWNTO 0) := "11";

-------------------------------------------------------------------------
--                    STACK
-------------------------------------------------------------------------

CONSTANT op_NONE   : std_logic_vector(1 DOWNTO 0) := "00";
CONSTANT op_POP    : std_logic_vector(1 DOWNTO 0) := "01";
CONSTANT op_PUSH   : std_logic_vector(1 DOWNTO 0) := "10";
CONSTANT op_BOTH   : std_logic_vector(1 DOWNTO 0) := "11";

-------------------------------------------------------------------------
--                    GROUP
-------------------------------------------------------------------------

CONSTANT op_ADD    : std_logic_vector(2 DOWNTO 0) := "000";
CONSTANT op_ADC    : std_logic_vector(2 DOWNTO 0) := "001";
CONSTANT op_SUB    : std_logic_vector(2 DOWNTO 0) := "010";
CONSTANT op_SBC    : std_logic_vector(2 DOWNTO 0) := "011";
CONSTANT op_AND    : std_logic_vector(2 DOWNTO 0) := "100";
CONSTANT op_OR     : std_logic_vector(2 DOWNTO 0) := "101";
CONSTANT op_XOR    : std_logic_vector(2 DOWNTO 0) := "110";
CONSTANT op_NOS    : std_logic_vector(2 DOWNTO 0) := "111";
```

```
CONSTANT op_NOT    : std_logic_vector(2 DOWNTO 0) := "000";
CONSTANT op_SL     : std_logic_vector(2 DOWNTO 0) := "001";
CONSTANT op_ASR    : std_logic_vector(2 DOWNTO 0) := "010";
CONSTANT op_LSR    : std_logic_vector(2 DOWNTO 0) := "011";
CONSTANT op_ROR    : std_logic_vector(2 DOWNTO 0) := "100";
CONSTANT op_ROL    : std_logic_vector(2 DOWNTO 0) := "101";
CONSTANT op_ZEQU   : std_logic_vector(2 DOWNTO 0) := "110";
CONSTANT op_CC     : std_logic_vector(2 DOWNTO 0) := "111";

CONSTANT op_NEVER  : std_logic_vector(2 DOWNTO 0) := "000";
CONSTANT op_ZERO   : std_logic_vector(2 DOWNTO 0) := "001";
CONSTANT op_NSIGN  : std_logic_vector(2 DOWNTO 0) := "010";
CONSTANT op_NCARRY : std_logic_vector(2 DOWNTO 0) := "011";
CONSTANT op_PAUSE  : std_logic_vector(2 DOWNTO 0) := "100";
CONSTANT op_INT    : std_logic_vector(2 DOWNTO 0) := "101";
CONSTANT op_NZERO  : std_logic_vector(2 DOWNTO 0) := "110";
CONSTANT op_ALWAYS : std_logic_vector(2 DOWNTO 0) := "111";

CONSTANT op_STATUS : std_logic_vector(2 DOWNTO 0) := "000";
CONSTANT op_TOR    : std_logic_vector(2 DOWNTO 0) := "001";
CONSTANT op_RSTACK : std_logic_vector(2 DOWNTO 0) := "010";
CONSTANT op_LOCAL  : std_logic_vector(2 DOWNTO 0) := "011";
CONSTANT op_RSP    : std_logic_vector(2 DOWNTO 0) := "100";
CONSTANT op_DSP    : std_logic_vector(2 DOWNTO 0) := "101";
CONSTANT op_TASK   : std_logic_vector(2 DOWNTO 0) := "110";
CONSTANT op_FLAGS  : std_logic_vector(2 DOWNTO 0) := "111";
CONSTANT op_IE     : std_logic_vector(2 DOWNTO 0) := "111";

------------------------------------------------------------------------
--  some instructions needed as constants
------------------------------------------------------------------------

CONSTANT NO_OP     : inst_bus := '0' & op_BRA & op_NONE & op_NEVER;
CONSTANT INT_OP    : inst_bus := '0' & op_BRA & op_BOTH & op_INT;
CONSTANT PAUSE_OP  : inst_bus := '0' & op_BRA & op_BOTH & op_PAUSE;
```

16 Bibliography

[3 instruction structures] Xiaoming Fan, Holger Heitsch, Tomasz Malitka, Bernd Rosenthal, and Klaus Schleisiek "Three Instruction Set Structures for a Stack Processor", Proceedings euroForth 1995, mail to: Klaus.Schleisiek@hamburg.de

16.1 Revision History

Version	Date	Remarks
1.40	21.1.01	First description after unifying TOS, LIT and ADDR registers
1.41	2.2.01	Unary CC-Instruction added
1.42	11.5.01	Remarks on pre-increment, post-increment data RAM addressing
1.43	23.5.01	FLAGS and IE register, multiple interrupts
1.44	2.6.01	Multitasking support added. Change in BRA NONE ZERO and NZERO
1.45	22.6.01	PAUSE-Instruction realised instead of BREAK, Patent-Application
1.45a	18.11.01	Paper for the 2001 euroForth Conference (Dagstuhl Castle)
1.46	11.1.03	Adaptation for the release of uCore_1.10