MicroCore

an Open-Source, Scalable,

Dual-Stack, Harvard Processor

Synthesisable VHDL for FPGAs

Klaus.Schleisiek@hamburg.de
www.microcore.org

Introduction

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.

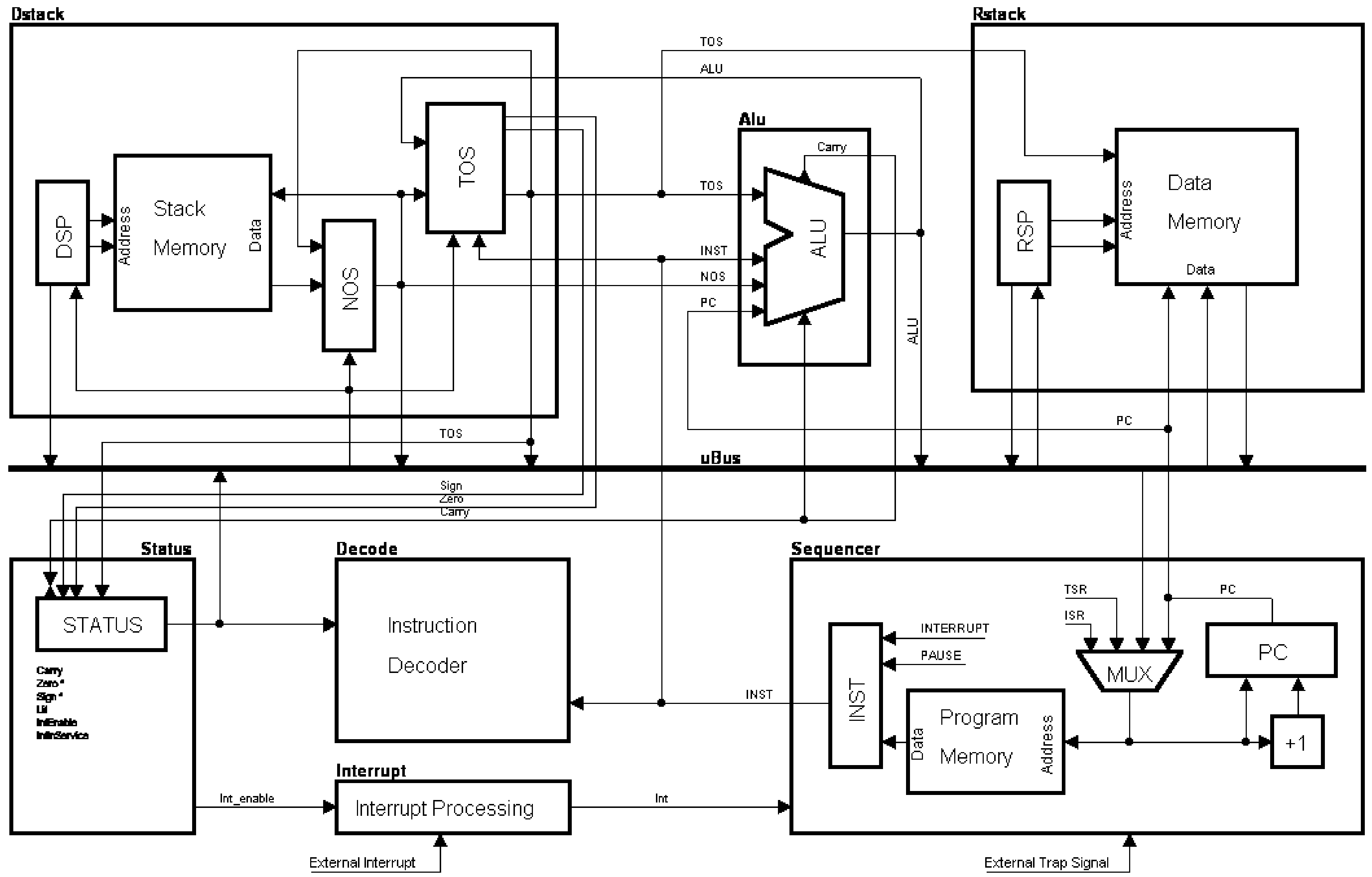
The FPGA approach also makes the user independent from product discontinuity problems that haunt the hi-rel industry since the dawn of the silicon age. Finally: putting the core into FPGAs puts an end to one of the high-level programming language paradigms, namely the aspect of hoped-for portability. 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.

Table of Content

MicroCore Simple Kernel Architecture

K. Schleisiek, 26.6.01, Rev. 1.40



1 Hardware Architecture

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

The architecture diagrams depict all busses which are needed, not showing the control signals which are generated in the Instruction Decoder from instruction register INST and status register STATUS as inputs.

All instructions without exception are 8-bits wide, and they are stored in the program-memory ROM. Due to the way literal values of any magnitude can be composed from sequences of literal instructions, all data-paths and memories are scaleable to any word width without any change in the instruction set.

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, and of the registers NOS (Next-Of-Stack), and TOS (Top-Of-Stack), which are in between the data-stack and the ALU.

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

The ALU's inputs come from TOS on one side and from NOS, INST or PC on the other side. Therefore, TOS+PC will be available for relative branching, and TOS+INST for post-incrementing memory addresses.

The upper end of the Data Memory is utilised as the return-stack under control of the Return-Stack-Pointer RSP. The address range which is physically used by the return-stack doubles as memory mapped I/O (not shown in the diagrams).

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

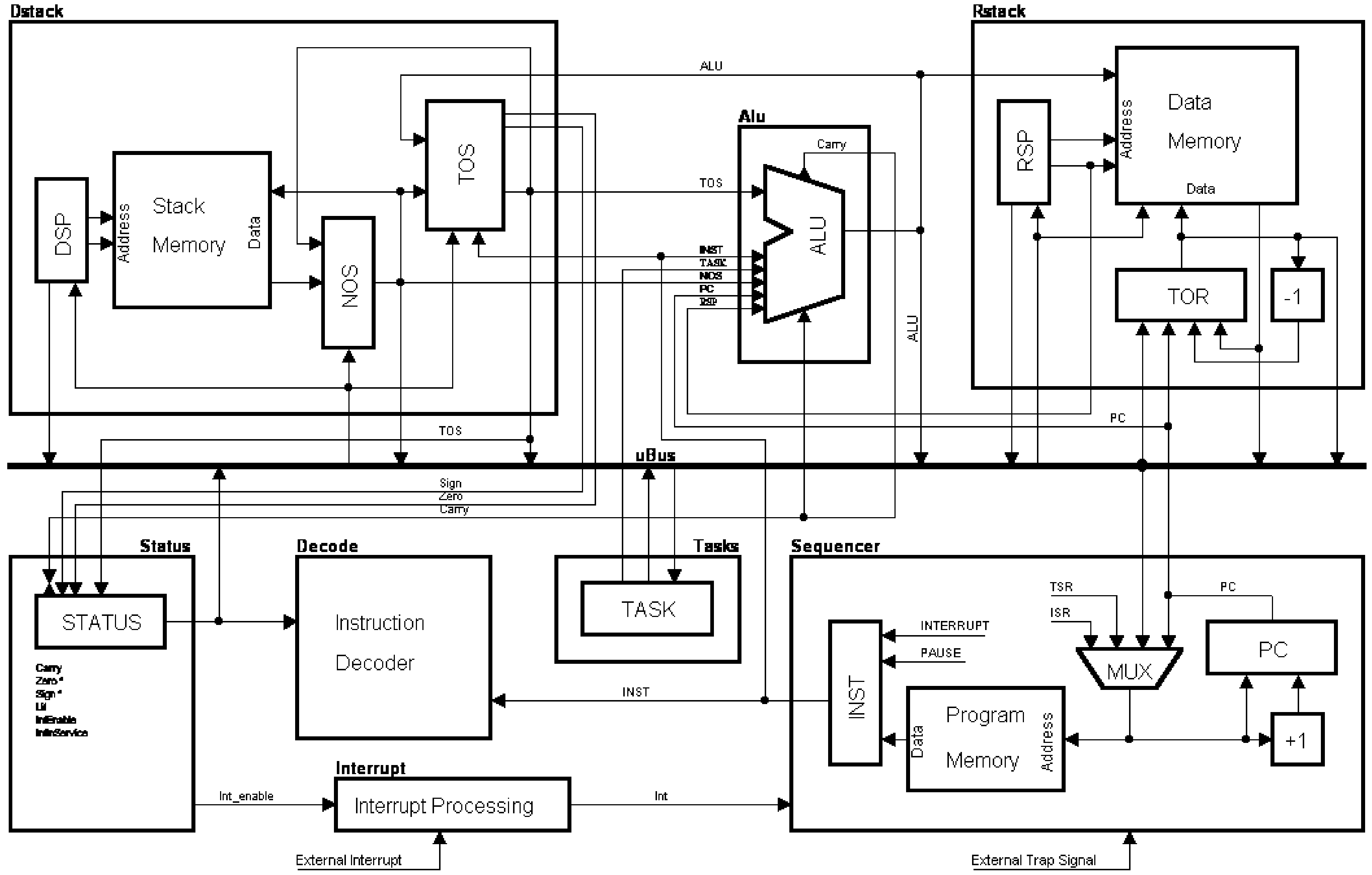
- The Program Counter PC for a sequential instruction,
- the ALU for a relative branch or call,
- the TOS register for an absolute branch or call,
- the return-stack inside the Data Memory for a return instruction,
- the fixed Interrupt Service Routine address ISR as part of an interrupt acknowledge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

The STATUS register has been singled out as a separate entity because it is composed of status bits generated from several sources.

The Interrupt Processing unit takes care of synchronising and masking an application specific number of external interrupt sources.

MicroCore Extended Kernel Architecture

K. Schleisiek, 26.6.01, Rev. 1.41



The extended architecture adds "nice to have" capabilities:

1. Feeding the return-stack address from RSP into the ALU allows to compute TOS+RSP for return-stack relative indexed addressing. This is all that is needed to support local variables in C.
2. Adding a TASK register that also feeds into the ALU allows to compute TOS+TASK for base-indexed addressing of e.g. the task-descriptor-block of the active task in a multitasking environment.
3. Adding a Top-Of-Return-stack (TOR) register and a decrementer allows to realise very fast FOR ... NEXT loops.

All these extension add capabilities to the simple architecture and they may be selectively implemented according to the needs of the application. Please note that all capabilities of the extended architecture are supported in the standard instruction set.

2 Instruction Architecture

Each instruction is always 8 bits wide. Scalability is achieved on the source-code level because literal values may be compiled into different object code depending on the data-word width. 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 Michael D. May and used in the Transputer for the first time.

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

Every 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 which are concatenated in the TOS register. Therefore, this type of code 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 when using an external ROM will hardly dominate total processor delay because all FPGA based processor logic 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 it was preceded by a literal nibble instead, 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	Unused by core, free for user extensions

BRAnches are group field conditioned and they consume the TOS using the content of 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. I feel that this was the most challenging problem of 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 for 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, which hosts the return-stack at its upper end. Fetch and stores use TOS for the address and the data is transfered between NOS and the Data-memory. A fetch sort of pushes the stack, leaving the address in TOS in place. A store sort of pops the stack, leaving the address in TOS in place. In both cases, the address in TOS may be auto-incremented or auto-decremented under control of the group-field which is interpreted as a 3-bit signed number. Absolute fetch and stores, which map into the return-stack area access memory mapped IO instead. Eight registers can be accessed directly.

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 interrupt and pause. This is where the opcode fields are non-orthogonal creating instruction decoding complexity, which is gracefully hidden by the VHDL synthesiser.

2.4 Group field

Code	Binary-Ops ALU	Unary-Ops ALU	Conditions BRA	Registers MEM
000	ADD	NOT	NEVER	STATUS
001	SUB	SL	ZERO	TOR
010	ADC	ASR	SIGN	RSTACK
011	SBC	LSR	CARRY	LOCAL
100	AND	ROR	PAUSE	RSP
101	OR	ROL	INT	DSP
110	XOR	ZEQU	DBR	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. DBR (Decrement-and-BRanch) supports the use of the Top-Of-Return-stack as a loop index. PAUSE and INT are conditions to aid in processing external events interrupt and pause.

Of the registers, TOR is used to implement R@ and 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 literal nibbles; 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 DBR
			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`

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-increment or `!++` post-increment
push		Data[TOS+<inc>] -> NOS -> Stack TOS + <inc> -> TOS	`++@` pre-increment or `@++` post-increment

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 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 version	N = signed 3-bit number
N @++	-	MEM PUSH N, post-increment version	N = signed 3-bit number
@	x	MEM PUSH 0 ALU POP NOS	2-cycle
N ++!	-	MEM POP N, pre-increment version	N = signed 3-bit number
N !++	-	MEM POP N, post-increment version	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!
NEXT	+	BRA POP DBR	atypical R-stack behaviour
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 numerical value - 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 CARRY-BRANCH and NEGATIVE-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, ROL instruction.
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 destination address has been loaded into TOS.

For the ordering of the bits one should take into consideration which "masks" for masking off flags can be loaded with only one literal nibble. This is important for the C- and IE-flags.

5.2 TOR

Top-Of-Return-stack. This allows access to the return-stack without pushing or popping it. The TOR-register must not necessarily be present because it is only needed when TOR is used for the DBR condition (Decrement-and-BRanch). An alternative and much cheaper implementation hardware wise to support software loops is its implementation as a branch on not-zero.

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 variable preference, which can be placed in a return-stack frame. It works similar to pre-incrementing ++@ and ++!. However, the data memory address is the sum of RSP+TOS.

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 which is located at the upper end of the data memory and it can be read and written to support multitasking and stack-frames for C-support.

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 also been set. 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. It is the responsibility of the interrupt service routine (ISR) of a specific interrupt to reset its interrupt signal 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 which are set ('1') will be written to, those bits which 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.

5.8 TASK

The TASK register can be read and written via memory mapped I/O (address = -1). It holds an address which 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 pre-incrementing ++@ 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-computation)

CC ComplementCarry Carry <- not Carry

7 Booting

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

A synchronised RESET signal resets all registers to zero with the exception of the INST register. Instead, INST loads the code for a NOP {BRA NONE NEVER} which happens to be all zeros as well, and therefore, during the first cycle (which executes the NOP instruction) the instruction whose address is in PC (which had been reset to zero!) will be fetched.

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 which performs a CALL to the ISR-address, which is a constant address, selected by the program address multiplexer and the STATUS register is pushed on the data-stack at the same time.

Therefore, only the first INT-cycle must be realised by special hardware. The second cycle (INT-instruction) is executed by an instruction which 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 its interrupt service routine.

9 Multitasking

The transputer has been a very innovative processor indeed, which was focused on multitasking, which is completely realised in hardware. Nice as this feature and its underlying philosophy may be, this lead to ramifications in order to simplify the necessary hardware support which did make the transputer difficult to market and eventually, despite a lot of money from British and European taxpayers, it was a commercial failure.

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 which would allow to access resources which 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 is very cheap 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 loading 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 has to 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 fully independent of the interrupt mechanism. It only 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

Data memory access operators `++@` and `++!` have been defined for a pre-incrementing implementation, access operators `@++` and `!++` have been defined for a post-incrementing implementation and this has been carried through to the cross-compiler.

From a programmers point of view, the pre-incrementing implementation is easier to handle because you only have to worry about a potential address offset when you need it, whereas in the post-incrementing implementation you have to worry about an offset in the preceding memory access.

From a hardware point of view, the post-incrementing implementation is more efficient because it takes the memory address directly from TOS. In the pre-incrementing implementation, the address is the output of the ALU-adder and therefore, it will arrive at the data memory later. If this memory is fast compared to one processor cycle this may not result in an overall performance degradation nevertheless. Alternatively, an additional processor cycle may be added to the memory access operators using the CLK_EN signal of the core.

When LOCAL data access is realised, the same problem exists. The memory address is the output of the ALU-adder, adding the offset in TOS and the RSP. In this case, the pre-incrementing implementation may be chosen because the timing problem does exist anyway.

In addition, most branch and call instructions will be relative, using the adder in order to compute the destination address. Because the offset is build up in TOS prior to the branch/call instruction the address computation can already be started in the previous cycle. This has been realised in the post-incrementing MicroCore model.

I believe that pre-/post-increment timing constraints will only be an issue when MicroCore is realised as an ASIC anyway. In FPGA implementations the (external) RAM can be expected to be fast enough.

Ideally, the cross-compiler would be made smart enough to compile the proper code based on a pre-incrementing syntax even if the implementation is post-incrementing. Otherwise, a change of implementation would break the code.

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 "debugable MicroCore" has an additional Centronics interface, which connects to a PC serving as the host. 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 the host, using the Centronics interface as umbilical.

11.1 Forth Cross-Compiler

It exists and it loads on top of Win32Forth because it’s 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. (Unfortunately, the cross-compiler is written in such a way that the current implementation only supports a maximum data-word width of 31 instead of 32 bits.)

It is a short but rather complex piece of code and my 4th or so 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. It took several attempts and months to solve the problem. Now, 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 remaining 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 remains of 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 has been implemented.

It turned out that the LOCAL addressing mode is all that is needed to come to grips with C's local variable preference. 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 which is capable of transforming local variable accesses into appropriate data-stack manipulations.

Once MicroCore actually exists, running on a prototype board, another iteration will be made.

12 Project Status

The VHDL code exists and can be released. The identical code could be synthesised using the Synplify and the Leonardo synthesisers targeting Xilinx and Altera FPGAs.

Here are some synthesis results for the Xilinx XC40xxE-4 family, which is slow according to today's standards. Please note that the architecture is fully scaleable: Any data word width you like (but under 12 bits does not make sense). The clock frequency is for a synthesis result with minimised gate count and no timing optimisation. For the synthesis example, MicroCore has been compiled for an internal data-stack 16 elements deep, an external program ROM and an external data memory RAM and two interrupt sources. CLB stands for "Configurable Logic Block", which is the atomic design entity of Xilinx FPGAs.

data word width	RAM/ROM size	CLBs	Clock [MHz]
12	4k	191	12.7
16	64k	234	11.7
24	64k	297	11.3
24	16M	307	12.0
32	64k	362	11.2
32	4G	386	10.4

The Forth cross-compiler is operational for up to 31 bits data word width. It's already of production quality. Some more effort could be spent on additional peephole optimisations.

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

The debug interface for MicroCore based on a Centronics port for host communication needs integration of a few design changes, which have been made for the sake of synthesiser portability.

The debugger itself on the host PC has not been started yet but its basic design can 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.

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 may be used in the spirit of the licensing terms of the Free Software Foundation applied to a hardware design.

"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 which 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. At present, you can use it in the spirit of the Free Software Foundation's licensing terms. I will work on specific licensing terms adapted to MicroCore, but that is not a top priority, it’s rather a boring necessity.

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 started it all with the design of the NC4000. Norbert Ellenberger, who backed the design of the FRP1600 which 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 := 12;
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 status_width : NATURAL :=  6;
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)
```

```
--------------------------------------------------------------------------
-- 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_SIGN   : std_logic_vector(2 DOWNTO 0) := "010";
CONSTANT op_CARRY  : 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_DBR    : 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 MicroCore - philosophical background (from 1997)

Similar to the Micro-Kernel approach in building real-time operating system kernels (e.g. Chorus Systeme SARL), I propose a Micro-Architecture approach to building a processor core. Therefore, the name of the project will be MicroCore.

In the MicroCore project, an initial building set of subsystems will be defined which can be composed into a processor core that fits into contemporary FPGAs. I will call these subsystems "Micro Cells". In terms of granularity, they are one level below the OMI Macro Cells and in terms of classical digital hardware nomenclature, they are on the LSI (Large Scale Integration) or MSI (Medium Scale Integration) level. In VHDL terms, a Micro Cell is an Entity.

Here is a non-exhaustive list of Micro Cells, not all of which necessarily have to be realised during the MicroCore project:

- Stack
- Return-stack (Stack with stackable decrement-and-branch-Register)
- Program Sequencer (including Program counter, subroutine mechanism)
- ALU
- 3-state bus controller (glitch free)
- Memory-Controller (adjusting logical versus external physical memory width)
- Stack-Frame Controller (to support C)
- Virtual Memory Controller (program controlled Cache)
- DMA controller
- Interrupt Controller
- Timer/Counter
- Test- and Debug-Interface (using JTAG protocol, if this is simple enough)
- FPGAbus to connect additional FPGAs as peripheral I/O devices
- RS232 interface

The Micro Cells will be realised as an abstract, simulation-efficient behavioural HDL description such that an efficient simulation of the macro architecture can be performed and used for hardware/software co design. Then for each Micro Cell that is going to be realised in hardware a synthesisable VHDL implementation must be developed. This implementation could be technology specific to take e.g. FPGA specific constraints into account.

I see two research challenges:

- To find a consistent interface philosophy such that the Micro Cells can be easily "plugged together" without the need for glue-logic.
- To find a good "factorisation" for the Micro Cells, such that they are reusable to realise a wide variety of Macro-Architectures.

Judging from the software engineering process, these two challenges can only be mastered by iterative refinement and in that respect the result of the MicroCore project will constitute a prototype after which a more "elegant" solution could be specified.

In the MicroCore project, I would like to realise a “tiny” processor core with a 12-bit word width and, accordingly, a maximum program size of 4k. The macro architecture itself should be designed such that it is scalable to also allow for 16 bit and 24 bit versions with maximum program sizes of 64k and 16M respectively (This has its major influence on the instruction-set structure to be used).

A multi-tasking real-time kernel for this “tiny” processor will be licensable from DESY where such a kernel had been developed back in the PDP-8 days and successfully ported to the IX1 two years ago.

MicroCore will have to include a survey of existing FPGA families in order to define implementation constraints, which will lead to portable VHDL realisations.

17 Bibliography

[3 instruction structures] Xiaoming Fan, Holger Heitsch, Tomasz Malitka, Bernd Rosenthal, and Klaus Schleisiek "Three Instruction Set Structures for a Stack Processor", Proceedings euro4th 1995, mail to: Klaus.Schleisiek@hamburg.de

17.1 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 DBR
1.45	22.6.01	PAUSE-Instruction realised instead of BREAK, Patent-Application
1.45a	21.11.01	Open Document for Dagstuhl 2001 euro4th Conference