Software/Hardware Co-design: Crypto MicroCore

MicroCore is a dual stack, Harvard architecture with three memory areas that can be accessed in parallel. One special feature of MicroCore is the possibility to define more complex Forth words by creating new CPU instructions that later can be implemented via VHDL in FPGA hardware. In our project[1] we developed a new version of the MicroCore C-compiler based on lcc, a fully ANSI-C compliant compiler. The compiler generates from C-Code MicroCore instructions for the 1.71 version which are sent to the target hardware with the MicroForth-loader. We built in optimizations for the global stack allocations, first suggested by the group of Chris Bayley at the University of York. We tested our compiler with the BLAKE hash algorithm, implemented both in compiled MicroCore code and with BLAKE-optimized instructions directly coded in hardware via VHDL.

Markus Knecht, Willi Meier, Klaus Schleisiek, Carlo U. Nicola | carlo.nicola@fhnw.ch

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). All instructions without exception are 8 bit 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 as long as the needed address space can be represented. Figure 1 shows the MicroCore's structure [7].

One special feature of MicroCore is the possibility to define more complex Forth words by creating new CPU instructions that later can be implemented via VHDL in FPGA hardware. The synchronous simple design of MicroCore is tailored toward an easy and cheap implementation in FPGA. Version 1.71 of MicroCore (the current specification) introduces a new and better subset of Forth as a machine language especially tailored for cryptological calculations. A simple VHDL interpreter allows the cross compiler to load "CONSTANTS.VHD", which defines hardware features and opcode mnemonics. MicroCore/MicroForth thus evolves to a co-design environment. Every change in the hardware is unambiguously brought forward to the cross compiler. It is guaranteed by design that names of hardware characteristics are spelled the same in VHDL and in Forth, and that they have identical values.

The control part of MicroCore (see Fig. 2) is represented by a decentralized Moore-FSM (Finite State Machine) whose next state is determined by the decoder. Decentralized means that semi-autonomous parts of the data path (i.e. the local- and data-stack) are directly fed by the decoder. Why decentralized? For two reasons: firstly, the logic of the FSM is simpler, and secondly, the VHDL description maps more gracefully into the hardware components.

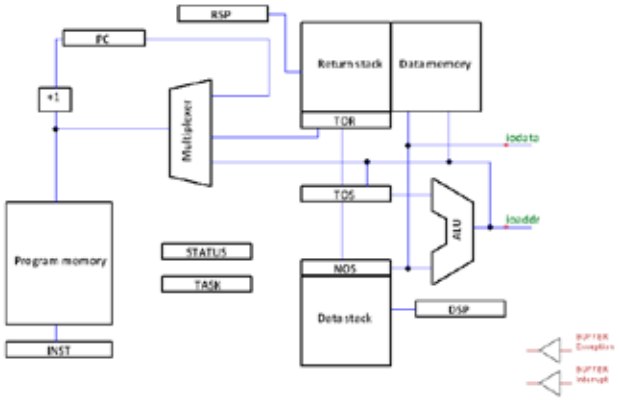

Figure 1: Overview of MicroCore

The data-path (Fig. 3) regulates the flow of operands to or from the two stacks. We have not littered the picture with all the necessary multiplexers in order to better illustrate the underlying structure. TOR, PC, DSP and RSP (Table 1 lists the most important abbreviations) each have a primitive n-bit-adder, because of the increment/decrement and push/pop instructions associated with them. Please, notice that manipulations of DSP, RSP, PC and INST occur simultaneously and can be done within a single clock cycle. All registers, stacks, and internal busses are n-bit wide. Since a MicroCore's instruction is 8 bit long we can store $k = n/8$ instructions per clock cycle in INST, transforming in fact the register INST into a small cache.

The data-path is made up of the data-stack, the ALU and of the data memory and return stack. Registers TOS and NOS feed the ALU. TOR allows not only for a decrement-and-branch instruction that can be nested, but also for complex math instructions like multiply, divide or vector multiply. Adding the TASK register will support a multi-tasking OS with a base-index address-

1 This work was partially funded by the Hasler Foundation Grant no. 12002 "An optimized CPU architecture for cryptological functions".

STATUS	Status register with eight 1-bit flags: C = Carry, OVFL = Overflow, IE = Interrupt Enable, IIS = Interrupt In Service, LIT = Literal, N = Sign flag, ZF/DIV = Zero Flag, TIMES = Counter for TIMES instruction.
INST	Instruction register
TOS, NOS, DSP	Top Of Stack register, Next Of Stack register, Data Stack Pointer. All work together with the data-stack (DS).
TOR, RSP	Top Of Return stack register, Return Stack Pointer. This set of registers works with the return-stack (RS).
PC	Program Counter register
TASK	Register whose content points to the Task Control Block (TAB).

Table 1: Glossary of the abbreviations we use to describe the data path

ing mode into the task control block. Indexed addressing into the return stack gives single cycle access to local variables. The data stack is realized by a single port RAM used as stack under the control of the DSP, and the topmost stack item is held in the TOS register. Typically, the size of the data-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 external world when set. In addition, program memory may be mapped into the lower part of the IO address space for von Neumann style read and write access in two cycles during the development phase. If the most significant address bit is not set, data-memory and return stack RAM is selected. The return-stack grows towards lower addresses and typically occupies the upper end of data memory under the control of *Return-Stack-Pointer* RSP. Both data and program memory may use internal FPGA block-RAM as "caches" and therefore, MicroCore can run as a "single chip controller" inside an FPGA without any external memory

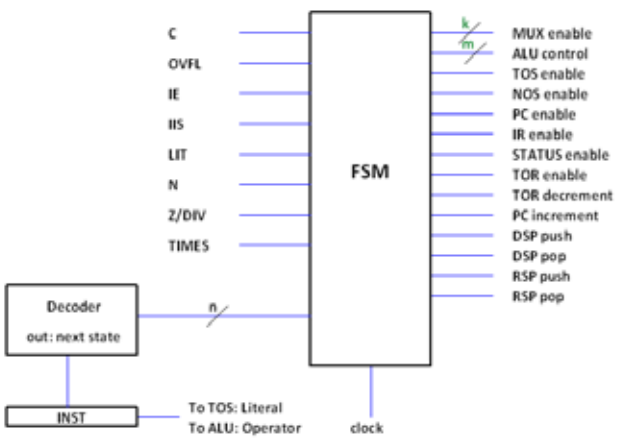

Figure 2: The control unit. n, m, k: number of control lines dependent in general from the width in bits of the data bus and the CPI (Clock Per Instruction).

needs. Several data memory access instructions are available:

- Absolute addressing with the address in the TOS register and a three-bit-signed pre-increment/decrement in the instruction (ld, st);
- Indexed addressing relative to the RSP for local variable access (lld, lst);
- Indexed addressing relative to the TASK register (tld, tst).

After each memory access, the absolute memory address that had been accessed will remain in TOS. Data transfer takes place between the second data-stack element NOS and memory/IO.

The Sequencer generates the program-memory address for the next instruction, which can have a number of sources:

- the PC for a sequential instruction;
- the ALU for a relative branch or call;
- the TOS register for an absolute branch or call;
- the TOR register for a return instruction;
- the INSTruction register for an immediate call (soft instruction);
- the fixed *Interrupt Service Routine* address (ISR) as part of an interrupt acknowledge cycle;
- the fixed *Exception Service Routine* address (ESR) during an exception cycle;
- the fixed *Overflow Service Routine* address (OSR) for a conditional service routine on overflow.

In the code, *uBus* has been defined as a record of signals that are needed in several entities: the data memory and I/O signals, a selection address for the internal registers, an array of all register outputs so as to simplify adding application specific registers. For better and easier code maintenance, instruction decoding and status register bit processing have been centralized since version 1.50 in a single file "uCore.vhd".

In Figure 4 we show the MicroCore development board we used throughout this work.

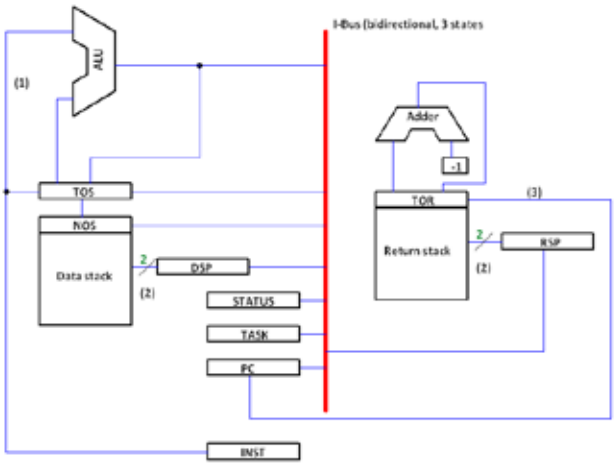

Figure 3: The data path. (1) Because of ++!, ...: 3 bit increment value will be added to TOS; (2) push and pop multiplexer lines; (3) direct way to save next instruction's address during call, int, ...; ALU top input: RSP, NOS, TASK, PC (not all connections are shown in this simplified representation).

Figure 4: The MicroCore development board. The large chip is the Lattice LFXP2-17E-5Q208C FPGA into which we implemented the different versions of MicroCore. Size: 94 x 70 mm.

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

Figure 5: The format of an instruction

MicroCore Instruction Set Architecture (ISA)

All MicroCore instructions are 8 bit wide. If the 7th bit is set, the instruction is interpreted as a 7 bit literal (LIT). LIT can be joined to form arbitrary long integers, which are stored in the operand stack. Two LITs build a LITERAL (a 16-bit number in two's complement format). It is a compiler's task to preserve the correct sign when joining many LIT together. The data size is thus independent from that of the instruction. As shown in Figure 5 all instructions are partitioned in fields, whose width in bit remains always constant. The instruction set architecture is based upon the Forth language.

The MicroCore's ISA implements a rich subset of the Forth language. Why, Forth? Forth has evolved in the last thirty years through natural selection in the embedded systems niche, a simple but powerful group of instructions that are well understood and whose side effects are empirically well known. We shall show in Tables 2-4 how to encode the three opcode's format fields.

With the encodings in Tables 2, 3, and 4 we can now synthesize meaningful Forth words, as Table 5 shows. The symbol - in the column LIT means that the previous instruction was an opcode; whereas when the symbol changes to +, that it was a LIT. In this case we get a new LIT on TOS. A * means, as always, don't care.

The regular structure of the three fields (Type, Stack, Group) and the constant bit format of all instructions simplify enormously the complexity of the decoder. However, we hasten to add that the encoding of the Forth words is not entirely satisfactory: for example, we had to use the stack field NONE to push or pop data into the local stack. This represents not only a breach in the orthogonality of the ISA, but more seriously, a supplementary hardware layer in the decoder.

Code	Name	Action
00	BRA	Branches, calls and returns
01	ALU	Binary and unary operations
10	MEM	Data memory and register access
11	USR	Free for application specific extensions. As to version 1.71, 26 user defined extensions are possible.

Table 2: Type field encoding

Code	Name	Action
00	NONE	Dependent on type field encoding
01	PUSH	TOS → NOS → Stack
10	POP	Stack → NOS → TOS
11	BOTH	Dependent on type field encoding

Table 3: Stack field encoding

Code	Binary Operation	Unary Operation	Condition	Register
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

Table 4: Group field encoding. The group field determines operations, conditions, and registers in dependence on type field encoding. The different type fields behave as follows: ALU specifies binary and unary operations; BRA specifies the conditions under which a jump is executed; MEM specifies a register name.

For the crypto version of MicroCore we introduced some new instructions:

ROT32: Instead of *times* we define rot32: ROT32 (32b n - 32b')

Label (<name> -), C: Compiles <name> into the dictionary as a constant, which holds the current program memory address. If <name> is the destination of a preceding GOTO, ?GOTO, or CALL, pending forward references will be resolved.

GOTO (<name> -), C: Compiles an unconditional branch to <name>. <name> may be a label or colon definition. GOTO supports forward referencing, i.e. the name of the label or colon definition that follows may be defined later on.

?GOTO (<name> -), C (zero: n -) (sign: n -) (carry: -) (ovfl: -): Compiles a conditional branch to <name>. <name> may be a label or colon definition. ?GOTO supports forward referencing, i.e. the name of the label or colon definition that follows may be defined later on.

- ?GOTO compiles 0<>branch
- 0= ?GOTO compiles 0=branch

Forth	LIT	Implementation	Comments
NOP	*	BRA NONE NEVER	No Operation; 0 → LIT
>R	*	MEM NONE RSTACK	Stack → NOS → TOS → R-Stack
R>	*	MEM BOTH RSTACK	R-Stack → TOS → NOS → Stack
OVER	*	ALU PUSH NOS	
+	*	ALU POP ADD	Stack → NOS<+>TOS → TOS
-	*	ALU POP SUB	Stack → NOS<->TOS → TOS
AND	*	ALU POP AND	Stack → NOS<AND>TOS → TOS
BRANCH	-/+	BRA POP ALWAYS	Absolute 3, relative 2 clock cycles
CALL	-/+	BRA BOTH ALWAYS	Absolute 3, relative 2 clock cycles
EXIT	*	BRA NONE ALWAYS	
L@	*	MEM BOTH LOCAL	LS[RSP + relAddr] → TOS

Table 5: Forth words encoding

- 0< ?GOTO compiles s-branch
- 0< 0= ?GOTO compiles ns-branch
- carry? ?GOTO compiles carry? IF ELSE GOTO THEN
- carry? 0= ?GOTO compiles nc-branch
- ovfl? ?GOTO compiles ovfl? IF ELSE GOTO THEN
- ovfl? 0= ?GOTO compiles no-branch

CALL(<name> -), C: Compiles an unconditional call to <name>. <name> may be a label or colon definition. CALL supports forward referencing, i.e. the name of the label or colon definition that follows may be defined later on.

A synchronized 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] and therefore, the instruction whose address is in PC (which has been reset to zero!) will be fetched during the first cycle (which is the NOP instruction).

Exceptions

MicroCore knows only one type of exception; hardware interrupts. They are synchronized with the internal clock and are answered at the end of the actual clock cycle, if the IE bit in STATUS register is set. During the first clock cycle after granting the interrupt request, MicroCore:

1. stores the already correct next instruction's address into PC without any increment, and
2. loads the hard-wired instruction BRA BOTH INT into IR. This call instruction selects the hard-wired address of the interrupt handler.

During the second clock cycle, after granting the interrupt request, MicroCore:

1. executes the instruction BRA BOTH INT. That means two things: transfer of the content of the STATUS register on the Stack and of the content of the PC on the L-Stack; and
2. loads the first instruction of the interrupt handler, since its address was already prepared during the first cycle.

In summary, only the first INT-cycle must be performed 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. During the STATUS register's transfer on the stack, the IIS bit is automatically set. New interrupt requests are from now on granted, only if we explicitly set IE and reset IIS.

Whenever an interrupt source is asserted, whose corresponding interrupt-enable bit is set in the IE-register, 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 de-assert its interrupt signal as part of its interrupt service routine.

Software Development

An interactive software development environment for uCore is rather straightforward and it has been realized under Linux and Windows on top of gforth under GNU GPL conditions. A "debuggable uCore" has an additional umbilical interface that can be controlled by a two-wire UART (RxD, TxD) interface on the host computer. The program memory, which must be realized as a RAM, can be loaded through this interface. After loading the application, a small debug monitor takes control and is exchanging messages with the host.

The following tasks can be done under control of a host computer:

- loading a program into uCore's program memory;
- resetting uCore;
- single-step debugging uCore with breakpoints;
- observing variables, buffer areas, semaphores, and the task list.

It loads on top of gforth (Windows and Linux), an open source 32 bit implementation of Forth. It can produce a binary image and a symbol table file

for the debugger, a VHDL file for simulation, and a MEM file for FPGA blockRAM configuration. Because the Forth systems are 32-bit systems, the cross-compiler only supports numbers of up to 32-bits signed magnitude. For larger data path widths, the cross compiler has to be extended accordingly if larger numbers need to be compiled.

It is a short but rather complex piece of code. Several peep hole optimizations have been implemented and the cross-compiler is of production quality.

lcc for MicroCore

We adapted the excellent ANSI-scc[2] [8] for generating MicroCore code. The scc is based on the lcc[3]. We highly recommend our few readers to get their hands on a copy of Hanson and Fraser's book [3] that describes in greater details the workings of lcc. We will here describe the interface between scc and Microcore in some detail, and only briefly skim (as need arises!) over the inner workings of the compiler.

Stack-based CPUs do not permit to address in a simple manner the single slots of the stack where the parameters for a procedure call are temporarily stored. All allocation algorithms that work well with register-based CPUs are useless for stack-based CPUs, since registers are individual random addressable slots of memory. The problem arises as how to allocate efficiently stack memory's slots for inter- and intra-procedure calls. Koopman [4] shows how to optimize the former, and Shannon [8] how to optimize the latter. Both algorithms are implemented in scc. For a good overview of the problem of global stack allocation, see [2].

The lcc consists of a front end and a back end. The former produces a meta representation of a C-program and the latter describes the ISA. The meta representation of the C-program is unfortunately not totally independent from the back end. A special structure (the interface) is responsible for the exchange of information between the two. We need to specify for example the data's alignment and type, the byte order (little or big endian), and for a typical lcc, the set of registers. In our case we must find a way to accommodate the stack structure of MicroCore within this register structure.

We will now discuss the changes we made in scc, to adjust it to the ISA of MicroCore.

The Meta representation

The C-Code is translated into a sequence of *Directed Acyclic Graphs* (DAGs) in such a way that each C-function is represented by a forest of DAGs. The DAG's nodes represent the primitive operations. Figure 6 shows the DAG-tree for the simple C-statement y = x[2] + 4. The processing of this

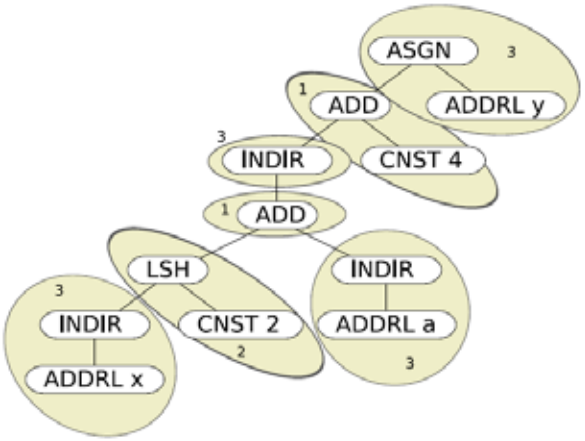

Figure 6: A typical DAG tree for the C statement: y = x[2] + 4. Picture adapted from [8].

DAG-tree necessitates following rules (for more details, see [6]):

```
stmt:   ASGNI4(stk, lAddr)     "#\n"       2
stk:    INDIRI4(lAddr)         "#\n"       2
stk:    CNSTI4                 "%a\n"      1
stk:    LSHI4(stk, stk)        "shift\n"  1+
                              (DATA_WIDTH+1)/2
```

DAG's nodes are assembled with three criteria: (1) type of operation, (2) data type and (3) size of data type. The following example illustrates the principle:

```
ADD + I + 4  ->  ADDI4
```

The size is always given in number of bytes. In our example the size is 4 bytes. Each node may have a variable number (between 0 and 2) of kids. Kids are nodes or terminal leaves. The interested reader will find in [6] the complete table with all types of nodes lcc supports.

We have defined following specific non-terminals for MicroCore:

- const: a constant;
- zero: constant 0;
- const1: constant 1;
- lAddr: the address of a local variable;
- addr: the address of a global variable, of a branch or of a dereferenced pointer (that means that all pointers contain absolute addresses.);
- jAddr: the address of a function;
- stk: represents a stack slot.

The full set of MicroCore's specific rules are shown in [6]. Here we illustrate them with a few examples. From the fact that all MicroCore's pointers may contain addresses or data, we derive the following rules:

```
addr:   INDIRP4(sread)     "\n"        0
addr:   INDIRP4(lAddr)     "#\n"       2
addr:   INDIRP4(addr)      "@\n"       1
```

2 scc = stack C compiler compiler
3 lcc = lean C compiler

Where:

```
lAddr: ADDRLP4 "%a"          1
/* No \n so must be printed by assig/indir */
addr:  CNSTP4 "#\n"          1
sread: COPYP "dup\n"         range(a, 1, 1)+1
sread: COPYP "over\n"        range(a, 2, 2)+1
sread: COPYP "2over\nnip\n" range(a, 3, 3)+2
sread: COPYP "any\n"         range(a, 0, 0)+1
```

For example, *addr: INDIRP4(addr) "@\n"* fetches data from a constant address of variable size. When scc sees the token '\#' in a rule, then it stops the evaluation of all the sub-trees which grow from that node, and it jumps to the function *emit2()* that calculates the exact size of the address (see [6]). The fourth column contains the cost (correct or guessed clock cycles per instruction) of the DAG-node. The compiler tries to construct a DAG-tree with minimal cost.

Optimizations

As we already observed, a stack-based CPU does not have single named registers that we can access in a random manner. On the contrary, we can access the slots of a stack only through stack operators like *swap*, *over*, etc. This fact makes all register allocation algorithms useless. We will not discuss here the equivalent algorithms for stack allocation, as they are very well explained in [4] and [2]. We give below a non-trivial example, which shows the working of these algorithms by generating MicroCore Forth code.

We examine the program mutual recursion to show how the intra- and inter-procedure optimization from [4] and [8] works.

```
int mRecFun(int cU, int cD){
    if(cD == 0){
        return cU;
    } else if(cD%2 == 0){
        return mRecFun2(++cU,--cD);
    } else {
        return mRecFun(++cU,--cD);
    }
}
int mRecFun2(int cU, int cD){
    if(cD == 0){
        return cU;
    } else if(cD%2 == 1){
        return mRecFun(++cU,--cD);
    } else {
        return mRecFun2(++cU,--cD);
    }
}
```

First we give in Listing 1 the MicroCore code without the local/global stack optimizations. The

```
( MicroCore(TM) uForth ) decimal : _mRecFun
    0 >r rsp@ 3 - rsp!
      ( allocate 3 local variables )
    6 l@  ?GOTO LABEL_2
        ( jump if not equal )
    5 l@ ( return int )
    GOTO LABEL_1
Label LABEL_2
    6 l@ 2 mod  ?GOTO LABEL_4
        ( jump if not equal )
    6 l@ 1- 3 l!
    3 l@ 6 l!
    3 l@ >r ( push int parameter 1 )
    6 l@ 1+ 3 l!
    3 l@ 6 l!
    3 l@ >r ( push int parameter 2 )
    CALL _mRecFun2  ( call to function )
    rdrop rdrop ( deallocate 2 parameters )
    ( tos ) 1 l!
    1 l@ ( return int )
    GOTO LABEL_1
Label LABEL_4
    6 l@ 1- 3 l!
    3 l@ 6 l!
    3 l@ >r ( push int parameter 1 )
    6 l@ 1+ 3 l!
    3 l@ 6 l!
    3 l@ >r ( push int parameter 2 )
    RECURSE ( recursive call to function )
    rdrop rdrop ( deallocate 2 parameters )
    ( tos ) 1 l!
    1 l@ ( return int )
Label LABEL_1
    rdrop rdrop rdrop rdrop
        ( deallocate 3 local Variables )
;
```

Listing 1: MicroCore code without stack optimizations

```
decimal : _mRecFun ( 2 parameters )
    swap swap dup 0 -
    ?GOTO lbl_2 ( jump if not equal )
    drop ( return int )
    GOTO lbl_1
  Label lbl_2
    dup 2 mod 0 -
    ?GOTO lbl_4 ( jump if not equal )
    swap 1+ swap swap swap 1-
    swap swap
    CALL _mRecFun2 ( call to function )
    ( return int )
    GOTO lbl_1
  Label lbl_4
    swap 1+ swap swap swap 1-
    swap swap
    RECURSE ( recursive call to function )
    ( return int )
  Label lbl_1
;
: _mRecFun2 ( 2 parameters )
    swap swap dup 0 -
    ?GOTO lbl_7 ( jump if not equal )
    drop ( return int )
    GOTO lbl_6
  Label lbl_7
    dup 2 mod 1 -
    ?GOTO lbl_9 ( jump if not equal )
    swap 1+ swap swap swap 1-
    swap swap
    _mRecFun ( return int )
    GOTO lbl_6
  Label lbl_9
    swap 1+ swap swap swap 1-
    swap swap
    RECURSE ( recursive call to function )
    ( return int )
  Label lbl_6
;
```

Listing 2: MicroCore code with stack optimizations

multiple use of l@ means that the local variables are temporarily stored outside the stack, thus increasing the time to access them. The optimized version of the program is given in Listing 2.
All the temporarily defined local variables remain on the stack and are accessed through normal stack operations like -*rot*, *dup* or *drop*.

The peephole optimization is carried out by the Forth cross-compiler whose responsibility is the transfer of the MicroCore Forth code to the target.

The cryptological function in Hardware

We used the BLAKE hash algorithm [1] in order to locate those cryptological functions that consume the most CPU cycles. The whys of this choice are explained elsewhere [5][6]. BLAKE security was evaluated by NIST in the SHA-3 process as having a "very large security margin", and the cryptanalysis published on BLAKE was noted as having "a great deal of depth". The BLAKE hash function comes in a family of four hash functions: BLAKE-224, BLAKE-256, BLAKE-384 and BLAKE-512. BLAKE-256 is based on 32-bit words, whereas BLAKE-512 on 64-bit words, from which the other members of hash functions are derived.

BLAKE uses only three basic operations: integer addition, xor '⊕', and rotation '<<<' (i.e., a circular shift given by a prescribed shift amount).

The G-function is the core of BLAKE and the source of its security against differential attacks [1]. Each G-function of BLAKE, $G_i(a,b,c,d)$ at round r operates on four selected words a,b,c,d of a 4 x 4 square of state words, and sets:

$$
\begin{aligned}
a &:= a+b+(m_{\sigma(r,2i)} \oplus c_{\sigma(r,2i+1)}) \\
d &:= (d \oplus a) \lll 16 \\
c &:= c+d \\
b &:= (b \oplus c) \lll 12 \\
a &:= a+b+(m_{\sigma(r,2i+1)} \oplus c_{\sigma(r,2i)}) \\
d &:= (d \oplus a) \lll 8 \\
c &:= c+d \\
b &:= (b \oplus c) \lll 7
\end{aligned}
$$

Here, σ denotes a specified permutation of numbers between 0 and 15, and the m_k denotes message words. The G-function performs six integer additions, six xors, and four word rotations. Apart from the permutation σ, the time consumption of each of the operations involved in the computation of a G-function has to be inspected. The rotation amounts in the second and fourth assignments differ, and are not always a power of two.

Operation	Seconds for 10^8 operations
Integer addition	0.18
xor	0.18
rotation	0.20

Table 6: Experimental results for BLAKE basic operations

Performance with gcc

For performance measurements, a computer with the following technical characteristics has been used: CPU Intel(R) Core i7 M 640 at 2.80 GHz; 2 GiB DIMM-Memory at 1333 MHz, 4 MiB external Cache, 320 GiB of permanent storage.

As the G-function is composed of the operations, (unsigned) integer addition, xor and rotation, the time consumption of these operations is measured and compared individually. To obtain reproducible and measurable quantities, each measurement has been executed 10^8 times (see Table 6). We give in Listing 3 the reference implementation of the G-function.

The measurements have been done using the C-function *clock()*. Timings for differing rotation amounts in the rotation operation don't vary significantly. Note however, that rotation amounts that are a multiple of 8 can be implemented by just reordering bytes, which is often faster than shifting the words. For measurements, rotation has been implemented as in the official BLAKE document, as an ANSI C-preprocessor macro. As a result, rotation is the most time consuming operation (see Table 6). When looking at assignments, the first and fifth assignments are more time consuming than the others, as they involve more basic operations plus a permutation of 16 words.

Performance with scc

We only show an example of the performance measurements before and after the hardware implementation of the *ROT32(x,n)* function. The full set of measurements is available upon request. We

```
#define ROT32(x,n) ((((x) << (32-(n))) | ((x) >> (n))))
#define ADD32(x,y) (((u32)((x) + (y))))
#define XOR32(x,y) (((u32)((x) ^ (y))))

#define G32(a,b,c,d,i) \
  v[a] = (ADD32(v[a],v[b]) + XOR32(m[sigma[round][2*i]], c32[sigma[round][2*i+1]])); \
  v[d] = ROT32(XOR32(v[d],v[a]),16); \
  v[c] = ADD32(v[c],v[d]); \
  v[b] = ROT32(XOR32(v[b],v[c]),12); \
  v[a] = (ADD32(v[a],v[b]) + XOR32(m[sigma[round][2*i+1]], c32[sigma[round][2*i]])); \
  v[d] = ROT32(XOR32(v[d],v[a]), 8); \
  v[c] = ADD32(v[c],v[d]);
```

Listing 3: Reference implementation of the G-function

```
Literal:        Bit 7=1           Opcode:         Bit 7=0

Forth Stack Description

Type     Stack     Instr.      LIT     Before     After     Notes      Number of Cycles (CPI)

BRA: Branch Functions (32 possibilities: 32 used)
...
BRA      BOTH      ROT32       *       n          --        ROT32      1
...
```

Listing 4: Function ROT32 defined as new Micro-Forth word

measured one full round of the loop at the heart of BLAKE containing all the $G_i(a,b,c,d)$ functions:

```
...
/*  do 12 rounds */
for(round=0; round < NB_ROUNDS32; ++round) {
    _ledsOn_();

    /* column step */
    G32(0, 4,  8, 12, 0);
    G32(1, 5,  9, 13, 1);
    G32(2, 6, 10, 14, 2);
    G32(3, 7, 11, 15, 3);

    /* diagonal step */
    G32(0, 5, 10, 15, 4);
    G32(1, 6, 11, 12, 5);
    G32(2, 7,  8, 13, 6);
    G32(3, 4,  9, 14, 7);

    _ledsOff_();
}
...
```

All times are measured with a Tektronix TDS 2014 oscilloscope with a resolution of 2 GHz directly on the MicroCore board (see Fig. 4). 12 rounds without hardware optimization take 2876 µs. With hardware optimization they only take 1700 µs, which results in a speedup of 1.69. The experiments show that the choice of parameters for $G_i(a,b,c,d)$ does not influence very much its performance. Namely, the average duration of a $G_i(a,b,c,d)$ function within the loop is: 1700/12/8 = 17.7 µs.

Discussion

Our measurements confirm that the apparently trivial function *ROT32(x,n)* is the performance bottleneck of BLAKE. It is interesting to note that this function is used very often in different hash-functions. Therefore, we decided to implement it in VHDL and to define it as new Micro-Forth word (and as part of the MicroCore ISA) as given in Listing 4.

It is quite remarkable that this new instruction has a CPI of 1. We have thus demonstrated that MicroCore is a very elegant platform that permits a continuous transition between hardware and software optimization.

Acknowledgement

We thank Dr. Crispin Bayley of York University for the source code of the global allocation algorithms.

References

[1] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan, SHA-3 proposal BLAKE, http://www.131002.net/blake/, 2010. 14, 15.

[2] Chris Bayley and Mark Shannon, Register allocation for stack machines, Proceedings of the 22nd EuroForth Conference (2006), 13–20. 10, 12.

[3] Christopher W. Fraser and David R. Hanson, A retargetable C compiler: design and implementation, Addison-Wesley, 1995. 10.

[4] Philip Koopman Jr., A preliminary exploration of optimized stack code generation, Proceedings of the rochester Forth Conference (1992), –. 10, 12, 13.

[5] Markus Knecht, Willi Meier, and Carlo U. Nicola, A space- and time-efficient implementation of the Merkle tree traversal algorithm, http://arxiv.org/abs/1409. 4081, 2014. 14.

[6] Carlo U. Nicola, Markus Knecht, Willi Meier, and Klaus Schleisiek, Microcore: An optimized architecture for cryptological functions, Tech. report for the Hasler Foundation, FHNW, October 2014. 11, 12, 14.

[7] Klaus Schleisiek, MicroCore 1.66 open, scalable, dual stack, Harvard processor for embedded control, June 2011. 2.

[8] Mark Shannon, A c compiler for stack machines, Master's thesis, University of York, Department of Computer Science, 2006. 10, 11, 13.