Concept and implementation of an extended return stack to enhance subroutine and exception handling in FORTH

Andrew Read

June 2014

andrew81244@outlook.com

Abstract

A conceptual generalization of the FORTH return stack is obtained by complementing it with one or more additional stacks that are tightly synchronized in operation according to some precise logical rules. With additional stacks specifically deployed to support subroutine and exception handling, a model is obtained whereby CATCH and THROW can be implemented as single machine language instructions and a number of other features emerge that enhance the flexibility, speed, and robustness of subroutine and exception handling in FORTH. The extended return stack model has been successfully implemented on the N.I.G.E. Machine.

1 Introduction

This paper discusses the concept and implementation of an extended return stack in FORTH. The traditional stack data structure comprises an array of cells in memory with a movable stack pointer. Push and pop operations are defined for placing data onto the stack or removing data from the stack. Typical FORTH implementations utilize two stacks: the parameter stack (generally used for program data) and the return stack (generally used for flow control and holding a subroutine return address). This paper describes a conceptual generalization to the return stack and its implementation on the N.I.G.E. Machine.

The key conceptual idea is that additional stacks can be linked to return stack in a structured arrangement. These additional stacks can be leveraged to support enhanced flexibility, speed, and robustness of subroutine and exception handling in FORTH.

The N.I.G.E. Machine is a complete computer system implemented on an FPGA development board [1]. It comprises a 32 bit softcore processor optimized for the FORTH language, a set of peripheral hardware modules, and FORTH system software. The N.I.G.E. Machine was presented at EuroFORTH in 2012 and 2013 [2, 3] and is available open source [4]. It follows in the footsteps of a number of significant FORTH processors [5, 6, 7, 8, 9, 10].

2 Review of prior work

Flexibility, speed and robustness are critical in all aspects of computer science. Much prior work has been done to apply these topics to subroutine and exception handling in FORTH.

Klaus Schleisiek conducted some of the first research into error trapping in FORTH and noted that "everything to do with a subroutine belongs on the return stack" [27, 28]. Klaus's comment inspired the author's thinking on the topic and is the intellectual germ of this project. Brad Rodriguez was also one of the pioneers of FORTH exception handling in the era before CATCH and THROW, including applying stack frames to FORTH [29, 30]. The development of the standard words, CATCH and THROW has been explained by Michael Milendorf who also provides a reference implementation [16].

Many authors have discussed ways to enhance the reliability of FORTH software. The ideas presented in this paper build off several:

Jaanus Pöial studied stack effects at a conceptual level and developed an algebraic formalism for validating FORTH code [21, 22].

Paul Bennett and Malcolm Bulger discussed the certification of high integrity software and explained that in order to be able to fully certify an application, then it is first required to certify the programming surface. They made the point that this would only need certifying once, and then could be used as the platform for many products [12]. This is especially relevant if the programming surface can be built into the base hardware.

Anton Ertl introduced a construct that guarantees that the cleanup code associated with resource usage is always completed, and demonstrated a more efficient implementation approach for cleanup code than using a full exception frame [13].

Nick Nelson contrasted approaches in the search for reliability of a large and complex FORTH system, including the idea of developing system which tries to struggle on despite programming errors [17].

Bill Stoddart and Peter Knaggs have contributed significantly to making FORTH robust [23, 24, 25, 26].

Considering the topic of subroutine local variables, Bailey, Sotudeh and Ould-Khaoua identified that local variable management and its efficient support in hardware is a prime concern in developing efficient stack based computation [11]. Anton Ertl stressed that the appropriate use of local variables has the potential to significantly unburden the data stack [14].

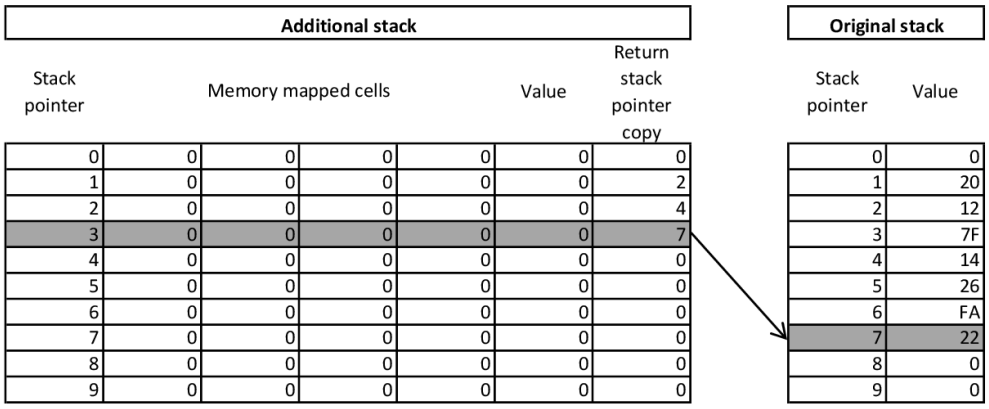
The use of a third stack for local variable storage is not new. Philip Koopman pointed out that this idea has often been proposed, but he suggests a better solution to support local variables may be a frame pointer into a software-managed program memory stack [19].

Lastly, regarding flexibility in subroutine handling, Glassanenko discussed programming techniques using return stack manipulations such as the implementation of new control structures and backtracking [15].

3 Two linked stacks as a conceptual model of subroutine and exception handling behavior

The motivation for starting with a conceptual scheme (rather than with a design specification for some desired output) was to find a straightforward logical model of the behaviors that occur in subroutine and exception handling as an intellectual goal in itself. The approach taken to finding a suitable model was an iterative series of thought experiments and pen and paper investigations. (The author accepts that this approach has the weakness of missing a formal proof. Therefore, as with the N.I.G.E. Machine overall, structured and extensive testing is a prerequisite to use in any critical system.)

This section presents the key features of the logical model that was found to represent subroutine and exception handling (the extended return stack) at a purely conceptual level. The section



Stack pointer	Memory mapped cells				Value	Return stack pointer copy
0	0	0	0	0	0	0
1	0	0	0	0	0	2
2	0	0	0	0	0	4
3	0	0	0	0	0	7
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0

Stack pointer	Value
0	0
1	20
2	12
3	7F
4	14
5	26
6	FA
7	22
8	0
9	0

Figure 1: Illustration of basic conceptual scheme of the extended return stack. The current position of the stack pointer is highlighted in gray.

following will explain exactly how this conceptual scheme has a direct relationship with subroutine and exception handling in FORTH.

Consider a simple, traditional stack that has been extended by making available an additional stack and creating certain logical connections between the two. In this paper the diagrammatic convention will be that the original stack is shown on the right and the additional stack on the left. The additional stack is structured as follows (figure 1):

The additional stack is at least two cells wide. Each cell can be read independently. The data contents of all cells change simultaneously when the stack pointer moves. The first cell (by convention shown as the rightmost cell in this paper) of the additional stack holds a copy of the stack pointer to the original stack. This stack pointer copy is read and written on certain events as described below. The second cell of the additional stack is the conventional part of the stack; it is where data that is PUSHed to the stack will be placed. The remaining cells of the additional stack are mapped to registers at fixed locations in system memory. This is configured in such a way that regardless of the position of the stack pointer, the top of stack values are always found at the same physical addresses in system memory.

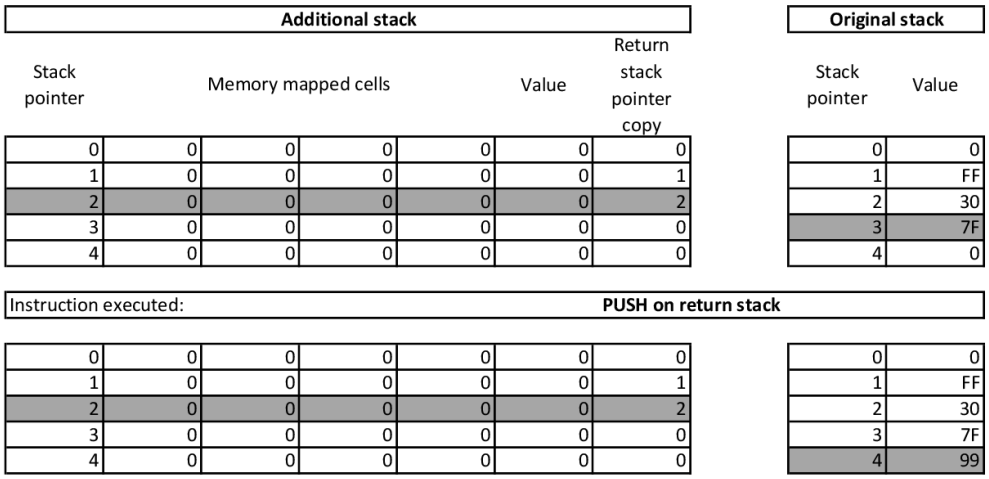
The rules for interaction between the original stack and the additional stack are as follows (illustrated in figures 2, 3, 4, 5, 6):

1. Push and pop operations are separately available for each stack. (Looking ahead, it is the mapping of these push and pop operations on separate stacks to an appropriate set of machine language instructions that makes for efficient support of subroutine and exception handling).

2. When the original stack is pushed then there is no impact on the additional stack.

3. When the original stack is popped then there is no impact on the additional stack unless the value of the original stack pointer after the pop would be less than the copy value held on the additional stack. In this case the additional stack is simultaneously also popped.

4. When the additional stack is pushed then two additional operations occur simultaneously: firstly the current value of the original stack's stack pointer is pushed to the first (rightmost)



Additional stack							Original stack	
Stack pointer	Memory mapped cells				Value	Return stack pointer copy	Stack pointer	Value
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	0
Instruction executed:						PUSH on return stack		
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	99

Figure 2: Illustrating the rule that when the original stack is pushed then there is no impact on the additional stack. The "before" state of the stacks is shown above and the "after" below.

cell on the additional stack and secondly the original stack is a pushed with the same value as was pushed to the additional stack.

5. When the additional stack is popped then the original stack's stack pointer is reset to the copy value held on the additional stack before the pop.

The conceptual scheme may be extended with multiple additional stacks as follows:

Each new additional stack is added on the "left", so that it has the same relationship with the current "leftmost" additional stack as the first additional stack has with the original stack according to the rules above. When an additional stack is pushed, all of the stacks on its right are simultaneously pushed with the same value. When an additional stack is popped, the reset of stack pointers flows to each stack on its right in a chain sequence until the reset of the stack pointer of the original stack. When a stack is popped to such a position that it causes a pop of the additional stack on its "left", then it is not necessary to propagate that behavior further to the left.

4 High level application to the FORTH programming language

To apply this conceptual model to the FORTH programming language the following arrangement is made. The FORTH return stack is the "original stack" in the terminology of the previous section and two additional stack are added (figure 7). The first additional stack is termed the subroutine stack and the second additional stack is termed the exception stack. PUSH and POP operations on each of the three stacks are mapped to FORTH primitives as follows: >R and R> operate on the return stack. EXECUTE and EXIT operate on the subroutine stack and CATCH and THROW operate on the exception stack (table 1).

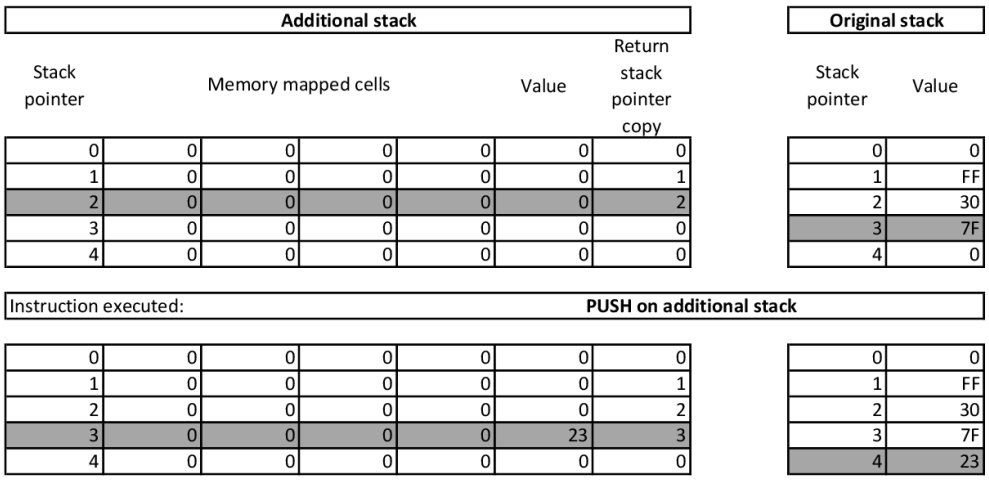
The first cell on the subroutine stack holds a copy of the return stack's stack pointer. The second cell on the subroutine stack is reserved for storing the return address of a subroutine call. There are at least an additional 16 cells on the subroutine stack that are memory-mapped to the system

Additional stack							Original stack	
Stack pointer	Memory mapped cells				Value	Return stack pointer copy	Stack pointer	Value
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	0
Instruction executed:						**POP on return stack**		
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	99

Figure 3: Illustrating the rule that when the original stack is popped then there is no impact on the additional stack provided that the value of the original stack pointer after the pop is not less than the copy value held on the additional stack.

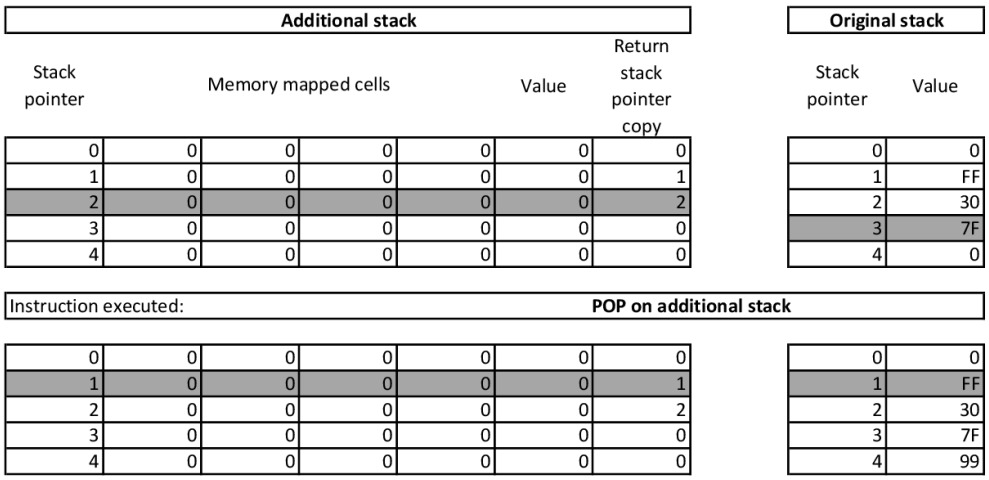
Additional stack							Original stack	
Stack pointer	Memory mapped cells				Value	Return stack pointer copy	Stack pointer	Value
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	0
Instruction executed:						**POP on return stack**		
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	99

Figure 4: Illustrating the rule that when the original stack is popped then the additional stack is also popped if the value of the original stack pointer after the pop is less than the copy value held on the additional stack.



Additional stack							Original stack	
Stack pointer	Memory mapped cells				Value	Return stack pointer copy	Stack pointer	Value
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	0
Instruction executed:						PUSH on additional stack		
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	23	3	3	7F
4	0	0	0	0	0	0	4	23

Figure 5: Illustrating the rule that when the additional stack is pushed then two additional operations occur: firstly the current value of the original stack's stack pointer is copied to the first (rightmost) cell on the additional stack and secondly the original stack is also pushed with the same value as was pushed to the additional stack.



Additional stack							Original stack	
Stack pointer	Memory mapped cells				Value	Return stack pointer copy	Stack pointer	Value
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	0
Instruction executed:						POP on additional stack		
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	99

Figure 6: Illustrating the rule that when the additional stack is popped then the original stack's stack pointer is reset to the copy value held on the additional stack.

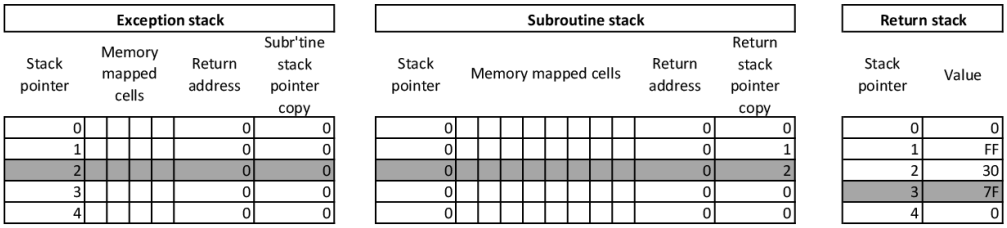


Figure 7: Application of the extended return stack model to FORTH with an exception, subroutine and return stack. The additional memory-mapped cells on the subroutine and exception stacks are shown narrower for clarity of the diagram only.

address space via registers in such a way that the top-of-stack values are always visible at fixed addresses regardless of the position of the stack pointer. These cells are used to hold subroutine local variables. Because of the way the subroutine stack operates with EXECUTE and EXIT, the correct set of local variables will always be available at the relevant memory addresses. (The choice of 16 cells for local variables is made to comply with the minimum ANSI FORTH requirement, but in the case of the N.I.G.E. Machine sufficient FPGA BLOCK RAM is available to allow for more cells if desired, table 2.)

The first cell on the exception stack holds a copy of the subroutine stack's stack pointer. The second cell on the exception stack holds a copy of the exception return address. The exception stack also has a number of additional cells that are memory-mapped to fixed addresses. Just as with subroutine local variables on the subroutine stack, these memory mapped cells are used to hold local variables that have scope within a single exception.

Instruction	**Exception stack**	**Subroutine stack**	**Return stack**
PUSH	CATCH	EXECUTE	>R
POP	THROW	EXIT	R>

Table 1: Mapping of PUSH and POP operations on the exception stack, subroutine stack and return stack to FORTH primitives

The operation of the extended return stack as concerns execution flow control will be discussed in relation to the following code example (reordered to match the flow of the text). The stack effects are also illustrated in figure 8.

```
:  mainloop
    ' innerloop CATCH
    if ." Error code" .  then
;

:  innerloop
    sub1
\ code omitted
;

:  sub1
    sub2
    255 THROW
\ code omitted
;

:  sub2
    10 >R
;
```

Within the FORTH word `mainloop` the first action is to call the word `innerloop` by way of a CATCH statement. CATCH is mapped to a PUSH instruction on the exception stack. The effect of the CATCH is to increment the exception stack pointer. Because the exception stack is the "leftmost" additional stack, a PUSH to this stack also increments the subroutine stack pointer and the return stack pointer. The first cell of the exception stack is pushed with the value of the subroutine stack pointer as it is before being incremented. The second cell of the exception stack is pushed with the return address for this CATCH statement. Since a PUSH operation has also been applied to the subroutine stack, the first cell of the subroutine stack is pushed with the value of the return stack pointer as it is before being incremented. The second cell of the subroutine stack is pushed with the return address for this CATCH statement. The return address for this CATCH is also pushed onto the return stack. At this point control has been passed to the word `innerloop`.

The first action of the word `innerloop` is to call the word `sub1` with an implied EXECUTE statement. EXECUTE is mapped to a PUSH instruction on the subroutine stack. The effect of the EXECUTE is to increment the subroutine stack pointer and, since the subroutine stack is "left" of the return stack, to also increment the return stack pointer. The first cell of the subroutine stack is pushed with the value of the return stack pointer as it is before being incremented. The second cell of the subroutine stack is pushed with the subroutine return address. The subroutine return address is also placed onto the return stack. The exception stack is not affected by a subroutine call, consistent with the conceptual scheme as outlined. At this point control has been passed to the word `sub1`. The first action of the word `sub1` is to call the word `sub2`. The mechanism involved is the same again with the result that both the subroutine and return stack pointers are incremented a second time.

The word `sub2` places the value 10 on the return stack above the current return address before calling the implied EXIT. In a typical FORTH implementation exiting a subroutine after placing an arbitrary value on the top of the return stack could result in an unstable condition because the arbitrary value could be taken as the subroutine return address. However in the extended return stack arrangement as described here the EXIT statement is mapped to a POP instruction on the subroutine stack. The effect of the POP on the subroutine stack is to return execution to the return address as read from the subroutine stack (the second cell position), reset the return stack's stack pointer to the value as read from the subroutine stack (the first cell position) and then decrement the subroutine stack pointer. As a result EXIT will return the flow of execution to `sub1` and the return stack pointer will return to the position that it was in when the subroutine call to `sub2` was originally made.

The following action is to call the word THROW with the value 255 on the top of the parameter

stack. THROW with a non-zero parameter is mapped to a POP operation on the exception stack. The effect of THROW at this point is to return the flow of control to the exception return address that was copied on to the exception stack when the corresponding CATCH was called. In addition, the subroutine stack is reset to the position that it was in when CATCH was called since the copy of the subroutine stack pointer that was copied to the exception stack at that time is now written back to the subroutine stack pointer. Following this, the return stack is also reset to the position that it was in when CATCH was called as the copy of the return stack pointer on the reset subroutine stack is written back to the return stack.

5 Additional FORTH requirements for subroutine and exception handling

The conceptual scheme for additional return stacks as described in the previous two sections provides almost all of the functionality that is needed to implement exception handling in FORTH. However to implement fully all of the required ANSI FORTH functionality in CATCH and THROW, some further mechanisms are required in addition to the additional stacks as described in the previous section.

Subroutine calls and exception handling could be operated by the additional stacks without any need for the return address of a subroutine call to be placed on the top of return stack. This is because as described, both the subroutine stack and the exception stack hold return addresses for subroutine and exception calls. However for comparability with existing FORTH code a copy of the subroutine return address should be placed on the return stack. There is a further point here. FORTH subroutines may remove the subroutine return address from the top of the return stack so that when an EXIT instruction is subsequently encountered, flow control is returned to the caller of the caller of that subroutine. This behavior, known as backtracking, may be used in implementations of CREATE DOES> and in other applications [15]. Since the conceptual scheme for the additional stacks already requires that if a stack is popped above the level of the copy of its stack pointer held by the additional stack to its right, traditional FORTH code that takes advantage of backtracking can run without alteration in the extended return stack scheme. (Note that backtracking in this form is environmentally dependent.)

Another issue that needs to be dealt is that EXIT (including an implied exit at the end of a FORTH word definition) needs to place a value of zero on the parameter stack if the subroutine was called with CATCH, but does not place any value on the parameter stack if the subroutine was called with EXECUTE (including an implied execute when a FORTH word is complied inside a definition). The conceptual scheme as discussed in the previous sections maps EXIT to a POP operation on the subroutine stack with flow control passing to the return address held on the subroutine stack. The difficulty is that subroutine stack doesn't "know" whether the current FORTH word was called with EXECUTE or CATCH.

A solution is achieved by having the return address that is stored on the exception stack to be one instruction ahead of the return address that is stored on the subroutine stack and arranging that a CATCH machine language instruction will always be followed by a ZERO machine language instruction (i.e. the instruction that places ZERO on the top of the parameter stack). As a result EXIT will direct execution flow, via the subroutine return address, to the ZERO instruction in a subroutine that was called by CATCH. On the other hand THROW, when called with a non-zero parameter, will direct execution flow via the exception return address and to the next following instruction and therefore skip the ZERO instruction. This is illustrated in the machine language excerpt below:

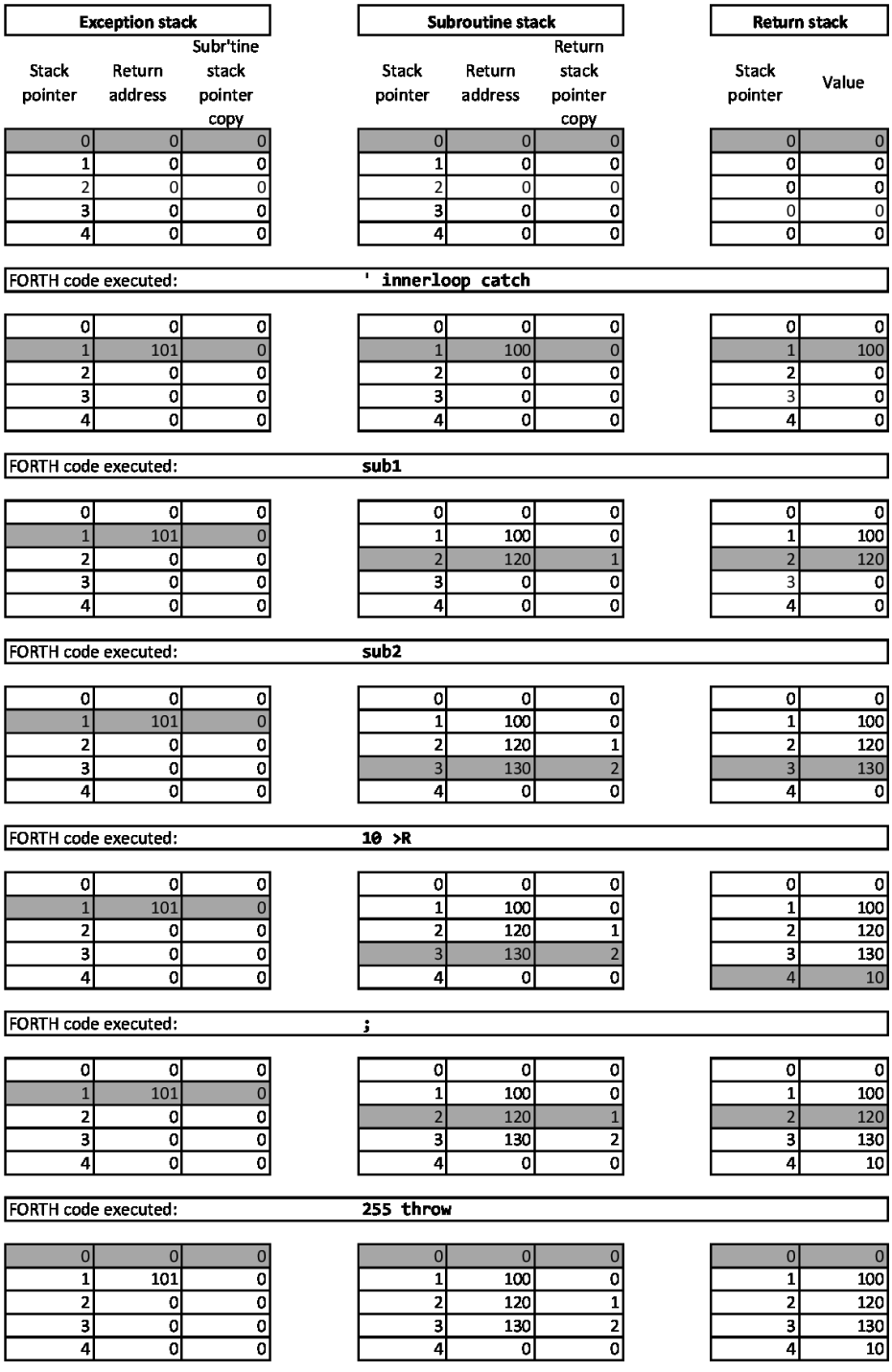


Figure 8: Illustration of the effect of some FORTH code on the exception, subroutine and return stacks. The current position of each stack pointer is show in grey. For clarity the memory-mapped cells for local variable storage on the subroutine and exception stacks have been omitted from the diagram.

```
PC address   Instruction              Comment
100          #.l          <sub addr>  ; load the subroutine address on stack
105          CATCH
106          ZERO                     ; 106 is the return address for EXIT
107          <next>                   ; 107 is the return address for THROW
```

Although the first two cells on the subroutine and control stack are used for control purposes, both of the stacks are multiple cells wide and the remaining cells are intended to be used for variable storage. As previously mentioned, the conceptual scheme and its implementation is such that the contents of the cells at the top of the subroutine and exception stacks are always available at fixed memory locations regardless of the value of the return stack pointer.

The remaining available cells on the subroutine stack are used to hold variables that are local to a subroutine call. As the subroutine stack is pushed down with EXECUTE or CATCH instructions, new variable storage is exposed. As the subroutine stack is popped up with EXIT or THROW instructions, the set of local variables reverts to those applicable to the appropriate subroutine level. In order to take best advantage of the cells on the subroutine stack that provide storage for locals it may be arranged that each time the subroutine stack is pushed, these cells will be written with a default value of zero.

The remaining available cells on the exception stack are used to hold variables that have scope within a particular exception. Suitable candidates for variables to hold on the exception stack are discussed in section 7.1. It should be arranged that each time the exception stack is pushed, these cells will be written with a copy of the cells immediately above them on the exception stack. In this way exception level variables will persist through subsequent CATCH statements unless they are explicitly changed.

The ANSI FORTH standard requires that THROW (with a non-zero parameter) also restore the parameter stack pointer to the value that it held at the time of the relevant CATCH statement. To accommodate this behavior one cell on the exception stack should hold a copy of the parameter stack pointer and automatically be updated at the time of a CATCH statement. Additional logic should reset the parameter stack pointer to the value as held on the exception stack at the time of a non-zero THROW. In this way CATCH and THROW can produce all of the required ANSI FORTH functionality simply by operation of the additional stacks without the need for supporting FORTH code. This is how the additional return stacks have been implemented on the N.I.G.E. Machine.

6 Implementation on the N.I.G.E. Machine

A number of design enhancements have been made to the N.I.G.E. Machine since it was demonstrated at EuroFORTH 2013. These are briefly summarized here for context. Firstly the overall design was ported from a Xilinx Spartan 3E FPGA on the Diligent Nexys 2 development board to a Xilinx Artix 7 FPGA on a Digilent Nexys 4 development board. In the process of porting the design the direct memory access (DMA) controller that mediates access to the off-chip 16 M byte pseudo-static dynamic RAM (PSDRAM) chip was completely re-written to conform to the AXI-4 protocol, the system clock speed was increased from 50MHz to 100MHz, and the program memory space was increased from 48K bytes to 128K bytes. (16M bytes of data memory is additionally available in PSDRAM). The native FAT file system software was also upgraded to support SD cards formatted with partition tables (as is common with higher capacity micro-SD cards). Finally a new video mode was added with 1024*768 resolution. A revised user manual has been produced that includes a quick start guide and documentation of the system features.

The subroutine and exception stacks described in section 4 and the further FORTH functionality described in section 5 were successfully implemented on the Nexys 4 version of the N.I.G.E. Machine. Three new machine language instructions were created: CATCH, THROW and RESETSP.

CATCH completes execution in 2 clock cycles (table 3). THROW completes execution in 3 cycles for a non-zero parameter and in 1 cycle if the parameter is zero. RESETSP resets all of the parameter stack pointer, the return stack pointer, the subroutine stack pointer and the exception stack pointer to zero. It completes execution in 1 clock cycle. The intended use of RESETSP is to return the machine to a known configuration upon reset. Three machine language instructions were removed from the instruction set, partly so that their opcodes could be reused in the three new instructions and partly because their use would interfere with the proper operation of the extended return stack. The removed instructions were: RSP@, RSP!, PSP! which respectively read and wrote the return stack pointer and wrote the parameter stack pointer. The original instruction PSP@ remains in the instruction set and is used within the implementation of the FORTH word PICK.

The subroutine and exception stacks were implemented as FPGA BLOCK RAM. The subroutine stack is 17 cells wide (544 bits) and 512 cells deep. The first 32 bit cell is subdivided into a 9 bit cell that holds a copy of the return stack pointer and a 23 bit cell that holds the subroutine return address. There are 16 cells available for the storage of subroutine local variables. 36 K bytes of BLOCK RAM are allocated to the subroutine stack. The exception stack is 9.5 cells wide (304 bits) and 512 cells deep. The first 32 bit cell is also subdivided into a 9 bit cell that holds a copy of the subroutine stack pointer and a 23 bit cell that holds the exception return address. There is next a 16 bit wide cell that holds a copy of the parameter stack pointer. There are 8 cells available for the storage of exception variables. (The value of 8 cells was chosen somewhat arability with the expectation that it can easily be adjusted depending on future needs). 19 K bytes of BLOCK RAM are allocated to the exceptions stack, so that the total BLOCK RAM used for both additional stacks is 55 K bytes.

The N.I.G.E. Machine uses microcode to control the datapath. In this scheme the lowest 5 bits of each machine language opcode are interpreted as an address and access a BLOCK RAM element. The data value returned from the BLOCK RAM element at that address are the control lines used to configure the multiplexers in the datapath . To accommodate the extended return stack the number of control lines was extended from 14 to 21. For example, 3 bits are used to control the subroutine stack pointer. Of the allowed 8 possible configurations available in 3 bits, 5 configurations are used: no change, decrement, increment, reset to the copy value held on the exception stack, reset to zero. The exception stack pointer and return stack pointer are similarly controlled by microcode. An additional configuration was added to the control of the parameter stack pointer: reset to the copy held on the exception stack. This configuration is used by THROW.

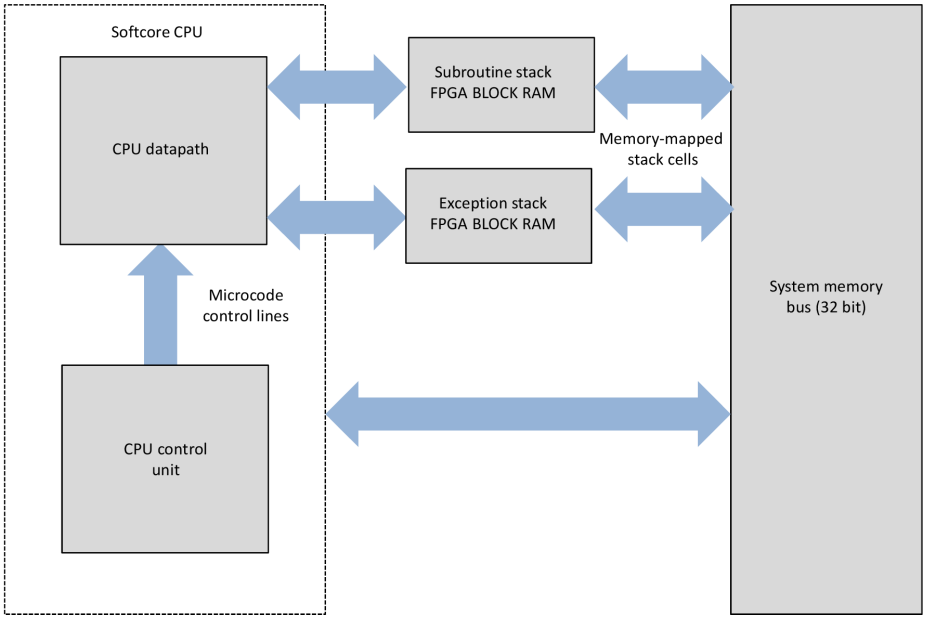


Figure 9: System diagram of the extended return stack implementation on the N.I.G.E. Machine

In order to make the subroutine and exception stacks available within the system memory space VDHL modules were created that interface the system memory space address lines with the BLOCK RAM elements holding the stacks. In each case the interface operated so that regardless of the position of the subroutine and exception stack pointers, the top of stack values are found at fixed memory locations. (The interface does not allow any other access from the system memory space into the subroutine and exception stacks, except to the at top of the stack values). These modules are also responsible for setting the newly exposed cells on the subroutine stack to zero and for "copying down" the values on the exception stack when a PUSH occurs, as described in the previous section.

Lastly, THROW was configured so that it would also signal the CPU to cancel any current interrupt condition. Thus if a non-zero THROW occurs within interrupt code the interrupt condition will be canceled when flow control is returned to the exception address. This is important since in the N.I.G.E. Machine an interrupt condition blocks further interrupts from occurring.

Making CATCH and THROW machine language instructions comes at the expense of implementing the subroutine and exception stacks in hardware. With the Artix 7 FPGA (device XC7A100T), the additional resources consumed are quite modest (table 2).

Version of N.I.G.E. Machine	BLOCK RAM	FPGA fabric logic
With extended return stack	32%	8.7%
Without extended return stack	22%	7.0%

Table 2: Artix 7 FPGA resource consumption comparing versions of the N.I.G.E. Machine with and without the extended return stack.

All of the required functionality was therefore achieved by adding BLOCK RAM elements to serve as the additional stacks, by extending the microcode used to control the datapath, by adding the required multiplexers to control the stacks within the datapath, and by making the top of stack values on the subroutine and exception stacks available in the system memory space. There was no need to resort to any unstructured "glue logic" to complete the design. Consequently the timing performance of the N.I.G.E. Machine was not affected and the design can be comfortably implemented at a clock speed of 100MHz.

The syntax for local variables implemented in the N.I.G.E. Machine system software follows VFX FORTH [20] and is documented in more detail in the N.I.G.E. Machine user reference manual [4]. The compiler utilizes a recognizer to identify local variable names ahead of searching the main dictionary, and is aware of the fixed memory addresses where local variables are stored on the subroutine stack.

7 Discussion

This section will examine the advantages and limitations of the extended return stack as compared with traditional approaches to subroutine and exception handling.

7.1 Flexibility

Using the exception stack to hold variables that have scope within a single CATCH statement ensures that if a THROW occurs all of these variables are guaranteed to be restored to their values prior to the CATCH. This restoration happens as an atomic operation inside a single machine language instruction. The feature can be leveraged for considerable utility as is illustrated by the following example. Anton Ertl has shown several models for a word `hex.` that prints a number in hexadecimal without changing BASE [13]. A new model that can be adopted with the extended return stack is as follows:

```
:  hex.-helper
    hex     \ the variable BASE is located on the exception stack
    u.
;

:  hex.
    ['] hex.-helper catch throw    \ no exception frame needed with extended
return stack.  CATCH is as fast at EXECUTE
;
```

The word `hex.` calls `hex.-helper` using CATCH, thus pushing the exception stack. Within `hex.-helper` the word `hex` stores the value of 16 to the variable BASE, which is located on the exception stack. Regardless of how `hex.-helper` exits, either at the implicit EXIT statement or due to a THROW during `u.`, the exception stack will be popped at that time and the value of BASE will be restored to the value that it held prior to the call to `hex.-helper`.

This model can be extended to a more general case with a word `debug.` that prints a number to the RS232 port in hexadecimal without changing BASE or redirecting output. The point is

that any variables that have scope within a single exception will be automatically restores to their former values upon EXIT or THROW.

```
:  debug.-helper
    hex     \ the variable BASE is located on the exception stack
    >remote    \ the character output vector is updated
    u.
;

:  debug.
    ['] debug.-helper catch throw
;
```

Anton Ertl has developed various models for region based memory allocation [18]. This system could likely be used to support region based memory allocation if the critical references are held on the exception stack. A complete model for region based memory allocation using the extended return stack is a topic for further research.

An additional point of flexibility concerns local variables. FORTH implementations typically favor VALUE flavored locals because these can be implemented using index offset load/store instructions. VARIABLE flavored locals may be less suitable for typical FORTH implementations because they require a memory address to be explicitly calculated on each reference. By contrast, the extended return stack makes it straightforward and efficient to implement VARIABLE flavored locals because the memory addresses of the local variables are fixed. The relocation of local variables onto the subroutine stack also removes the possibility of interference with DO LOOP operations that may occur when the return stack is used for local storage.

However the extended return stack approach to local variables has its constraints. Firstly the number of local variables available is limited to the cells provided in hardware on the subroutine stack. In the current implementation of the N.I.G.E. Machine there are 16 cells available for local variables on the subroutine stack and 8 on the exception stack and this could be extended relatively easily. Secondly, since most FORTH subroutines do not use local variables the approach of keeping them on the subroutine stack may seem wasteful of memory resources. However this is more of a trade-off decision, since the prize obtained by adopting this approach is the ability to implement CATCH and THROW as single machine language instructions. Depending on the chosen FPGA, this trade-off may not be a significant concern (table 2).

In the N.I.G.E. Machine system software, the variable BASE, and vectors for redirecting keyboard and screen input/output to the RS232 port are held on the exception stack.

7.2 Speed of code execution

Fast subroutine execution is important in FORTH because of the highly factored nature of FORTH code. The speed of subroutine execution on the N.I.G.E. Machine is unchanged by the implementation of the extended return stack. Fast exception handling may also be important. Although CATCH and THROW were not implemented on the N.I.G.E. Machine prior to the extended return stack, considering the reference implementation of CATCH and THROW it is likely that executing CATCH in 2 clock cycles and THROW in 3 clock cycles will be an order of magnitude faster than implementing these constructs in software (table 3).

Version of N.I.G.E. Machine	EXECUTE	EXIT	CATCH	THROW
With extended return stack	2	2	2	3
Without extended return stack	2	2	n/a	n/a

Table 3: Speed of subroutine and exception execution measured in clock cycles the N.I.G.E. Machine with and without the extended return stack implemented. CATCH and THROW were not implemented on the original N.I.G.E. Machine

Speed of local variable access is also important: arguably local variables are expected by programmers to be the fastest-to-access storage available. This will be the case if local variables are held in CPU registers rather than system memory, as is likely to be the case with implementations of the C programming language or with certain native FORTH implementations. However if local variables are held in registers then there will be a time penalty on subroutine entry and exit due to the need to save and restore the register set. Alternatively, holding local variables in system memory dispenses with this penalty, but access to system memory will likely be significantly slower than access to registers. The extended return stack offers the best of both worlds. Specifically allocated local variable storage on the subroutine stack means that there is no requirement to save or restore a register set. At the same time access to local variables is directly mediated by FPGA fabric logic. On the N.I.G.E. Machine all load/store operations to local variables complete execution in 2 clock cycles (as is the case with access to the N.I.G.E. Machine's BLOCK RAM in general).

7.3 Robustness / fault tolerance

Four arguments are made why the extended return stack concept, implemented in hardware, offers significant benefits for robustness and fault tolerance for FORTH programmes:

The hardware based extended return stack provides an absolute guarantee that variables held on the exception stack will be returned to their former (prior to CATCH) values upon EXIT or THROW. This occurs as an atomic operation within a single machine language instruction. This guarantee means that no further software problem solving is needed to ensure the safe handling of these variables in the event of an exception. This is valuable in high integrity software both as a feature in its own right and because, as Paul Bennett and Malcolm Bugler explain [12], once a programming surface is certified then it serves as a extensible platform for further applications.

A subroutine EXIT will execute correctly even if the subroutine has left spurious values on the top of the return stack (for example by leaving a DO LOOP without UNLOOP). This was demonstrated in section 4. Whilst it might raise a concern for the moral hazard of programmer complacency in managing the return stack, in critical situations the avoidance of the serious error that would have occurred otherwise may be a significant benefit. As Nick Nelson points out, a system that struggles on despite programming errors is a valid strategy for avoiding failures [17].

Although the literature does not suggest that exception processing is currently a bottleneck for FORTH programs[13], the extended return stack offers very fast exception processing in hardware (i.e. as fast as an ordinary subroutine call and return). This assurance on performance may encourage programmers to increase their use of CATCH and THROW, this improving software integrity.

As a final point, since the exception stack does not rely on a global variable to anchor its execution, the possibility that this variable could be corrupted, with catastrophic consequences for subsequent exception flow control, is avoided.

However on a practical level, and as also noted in section 3, before the N.I.G.E. Machine could be used in any critical systems an extensive program of structured testing (or some other approach) would be needed to certify the integrity of the N.I.G.E. Machine itself.

8 Conclusion

The extended return stack starts with a conceptual scheme for additional stacks that is not dependent on any particular hardware or FORTH implementation. The straightforward way in which this structure is able to handle exception and subroutine processing, and the one-to-one correspondence of the CATCH, THROW, EXECUTE, EXIT, >R, and R> FORTH primitives with PUSH and POP operations on the exception, subroutine and return stacks suggests that the conceptual stack scheme has a natural correspondence to the underlying logic structure of exceptions and subroutines in FORTH. The additional tweaks to this conceptual scheme that are needed to fully implement the requirements of ANSI FORTH are not extensive.

Implementation on the N.I.G.E. Machine was straightforward because the N.I.G.E. Machine's softcore is microcode based. The additional functionality is obtained by extending the number of control lines set by microcode and adding appropriate multiplexers to the datapath. The FPGA resource requirements for the extended return stack are minimal on an Artix 7.

The availability of variables on the exception stack that are guaranteed to be restored to their pre-CATCH value upon EXIT or THROW may be a genuine innovation. In addition, the implementation of CATCH and THROW as single machine language instructions makes exception processing very fast. Overall this paper has argued that the extended return stack offers significantly enhanced flexibility, speed, and robustness of subroutine and exception handling in FORTH.

Further work is intended in three areas:

- preparing additional verifications that the additional return stack design works correctly for subroutine and exception handling in all corner cases
- developing applications for variable storage on the exception stack, for example complementing with Anton Ertl's models for region based memory allocation [18]
- seeking further ways in which the extended return stack could drive further improvements in robustness and fault tolerance of FORTH software

The author sincerely wishes to thank the anonymous academic reviewers for their time and effort in providing feedback. Their comments on content and calibration have been very helpful in clarifying the author's thinking and improving the presentation of the paper.

References

[1] The author, video demonstrations https://www.youtube.com/channel/UCz_LqPfKT0r2rEID7AvChw

[2] The author, "The N.I.G.E. Machine: an FPGA based micro-computer system for prototyping experimental scientific hardware", in *EuroFORTH*, 2012

[3] The author, "Optimizing memory access design for a 32 bit FORTH processor", in *EuroFORTH*, 2013

[4] The author, Github open source repository https://github.com/Anding/N.I.G.E.-Machine

[5] James Bowman , "J1: a small Forth CPU Core for FPGAs" in *EuroFORTH*, 2010

[6] K. Schleisiek, "MicroCore," in *EuroFORTH*, 2001.

[7] B. Paysan, "b16-small – Less is More," in *EuroFORTH*, 2004.

[8] E. Hjrtland and L. Chen, “EP32 - a 32-bit Forth Microprocessor,” in Canadian Conference on Electrical and Computer Engineering, pp. 518–521, 2007.

[9] E. Jennings, “The Novix NC4000 Project,” Computer Language, vol. 2, no. 10, pp. 37–46, 1985.

[10] Rible, John, "QS2: RISCing it all," Proceedings of the 1991 FORML Conference, Forth Interest Group, Oakland, CA (1991), pp. 156-159.

[11] C. Bailey, R. Sotudeh, and M. Ould-Khaoua, “The Effects Of Local Variable Optimisation In A C-Based Stack Processor Environment.”, in *EuroFORTH*, 1994

[12] Paul E. Bennett, Malcolm Bugler, “Certification of High Integrity Software”, in *EuroFORTH*, 1998

[13] M. Anton Ertl, “Cleaning up after yourself”, in *EuroFORTH*, 2008

[14] M. Anton Ertl, “Ways to Reduce the Stack Depth”, in *EuroFORTH*, 2011

[15] M.L.Gassanenko, “Open Interpreter: Portability of Return Stack Manipulations”, in *EuroFORTH*, 1998

[16] Michael Milendorf, “CATCH and THROW”, in *EuroFORTH*, 1998

[17] N.J. Nelson, “Crash Never”, in *EuroFORTH*, 2011

[18] M. Anton. Ertl, “Region-based Memory Allocation”, in *EuroFORTH*, 2013

[19] P. J. Koopman, Jr., “Stack computers: the new wave”, Halsted Press, 1989

[20] Stephen Pelc, “VFX FORTH for Windows”, MPE, 2011

[21] Jaanus Pöial, “The algebraic specifications of stack effects for Forth programs”, FORML, 1990

[22] Jaanus Pöial, “Multiple stack effects of Forth programs”, EuroFORML, 1991

[23] Bill Stoddart and Peter Knaggs, “The Cell Type”, Proc. 1991 Rochester Forth Conf.

[24] Bill Stoddart and Peter Knaggs, “Formal Forth”, Proc. 1991 Rochester Forth Conf.

[25] Bill Stoddart and Peter Knaggs, “The Event Calculus: Formal Specification of Real Time Systems by means of Diagrams and Z Schemas”, 5th International Conference on putting into practice method and tools for information system design, 1992, Institute Universitaire de Technologies, Nantes, France

[26] Bill Stoddart and Peter Knaggs, “Type inference in Stack Based Languages”, Formal Aspects of Computing 5(4):289-98, Springer International

[27] Klaus Schleisiek, "ERROR TRAPPING: a Mechanism for Resuming Execution at a Higher Level.", 1983 FORML Conference Proceedings, pp. 151-154, San Jose, CA: FORTH Interest Group, 1984

[28] Klaus Schleisiek, "Error Trapping and Local Variables", 1984 FORML Conference Proceedings, CA: FORTH Interest Group, 1985

[29] Brad Rodriguez, "A Forth Exception Handler", SIGForth Newsletter Vol. 1 No. 2 (Summer 1989)

[30] Brad Rodriguez, "Stack Frames in Forth", SIGForth Newsletter Vol. 1 No. 4 (Winter 1989)