sTTAck: Stack Transport Triggered Architecture

Aliaksei V. Chapyzhenka

Belarusian State University of Informatics and Radioelectronics
Department of Computer Engineering
P.Brovky 6, 220027 Minsk, Belarus
alex@radiopage.com.by

Abstract. *Simplicity of Forth language allowed to build Forth based processors. But the common sequential computational approach has performance limitation. The traditional approach to enhance the performance of processor is to increase the number of functional units that work concurrently.*

In this paper we examine stack's properties and propose to extend stack paradigm for parallel computing. The main goal of this article is to define new architecture so-called sTTAck. The main advantages of this architecture are the simplicity and flexibility of hardware and large freedom of its software utilisation.

Keywords: *Instruction Level Parallelism, VLIW, Stack, Forth*

Introduction

The Forth language has found it's main application in embedded systems, where the simplicity is the key property. A number of its principles allows to construct simple, flexible and scalable hardware/software systems.

The hard requirements are imposed upon modern embedded systems [PLC97]. The direct hardware implementation of primitives of Forth [Koo89] gives a noticeable purchase in performance, but for further increasing it is necessary to apply an internal parallelism of algorithms.

However hard requirements to the price, compactness and power consumption allow using multiprocessor implementation only as the last measure. First of all, it is necessary to trouble of performance of single processing node by increasing the number of function units, which operate concurrently.

Two different approaches: superscalar and *very long instruction word* (VLIW) [Fish83] are the parts of unified Instruction Level Parallelism (ILP) theory. VLIW are easier extensible for high performance ranges because they lack much of the superscalar hardware required for scheduling of resources. However all scheduling optimisation functions made at the software level and the binary compatibility loses.

Applying of the VLIW principles for the Forth processors requires extending of dual stack model up to multistack case and building of the paralleling translator converting sequential intermediate representation of code to parallel.

The parallel multistack architecture so-called sTTAck became a basis for experiments in this area. The project is based on the principles and framework of *transport triggered architecture* (TTA) [Cor98]. The hardware stack machines add some useful properties to TTA.

1. Primitives of Forth

One foundation of the Forth language speaks that the all variety of extensions of the dictionary may be constructed from base set of words - *primitives* that are taking into account of all nuances of the concrete hardware representation.

Primitives can be written in codes of the existing processor, and the further extension of the dictionary will be processor-independent. The base dictionary of the eFORTH translator is the example of that. The compact kernel of it is realised for some different processors to ensure program portability at the source text level. The portability of the code level can be realised by means of stack virtual machine.

Processors of the various architectures and the different destination have the features permitting to achieve the peak performance for the determined class of the tasks. It can be both extended computational functions such as floating point operations and special mechanisms of data access. The aspiration to use them in the unfoundedly increases number of primitives granting access to these particular resources. The result is the unreasonable propagation of number of primitives and that worse we have the loss of compatibility at all levels.

Hardware implementation of primitives

Natural path for speed-up of Forth codes execution is to found the instruction set of the processor on basis of Forth primitives. Forth-processors follow this way [Koo89]. The aim is to minimise a set of primitives has begun a basis of the MISC project [TiM95]. There are two ways of further enhanced of the performance of a single processor node.

The first way is to execute chain of primitives concurrently, as it NOVIX makes, grouping them in one command; or as it is in PicoJava [Way96], where the special hardware scans an instruction stream to find the permit combination for concurrent execution.

Second way is to try to split primitives on elementary operation, more workable for the concurrent execution. For this goal, it is necessary to analyse properties of stack as a main data handling mechanism.

2. Stack's properties

Stack as the data structure is widely used. Furthermore, it is possible to use stack as a local computing environment. Modern processors of general and special purpose apply stacks. The hardware stack support is a main means of the acceleration of the stack operations. Some RISC processors use the mechanism of stack frames for access into main register file.

Another approach is applied in stack machines. The key property of these machines is the visibility of all stack operations on the architectural level. In addition to the hardware stack machines (stack-processor) the virtual stack machines are widely used too. There are some properties inseparably linked with stack:

Restorability is the possibility to return in the point where current state was stored. Program counter (PC) is a main variable what determines current state of program flow. But usually the processor has other state registers: flags, status and environment variables. All these variables may be stored in stack for later using.

Recursion as a desirable language feature has been introduced for the subroutine call in the late 50s. The non-recursive languages had a many problems with a directly or indirectly self calling and other complex methods of the flow control. The solution of the recursion problem is in the use of stacks for storing of the current state. The using of stacks as a control structure for recursion calls make possible to realise basic types of control.

Reentrancy is a possibility of multiple uses of the same code by different threads of control. The correct decomposition of the program allows receiving the more compact and effective code.

Locales are the property of all stack allocations.

Stacks also may be calcified by the purpose: return stack, expression evaluation stack, local variable stack, parameter stack and other.

Stack as the data structure

From a theoretical viewpoint, stacks are important as the most basic and natural means that can be used in processing well structured code. The stack allocation order is natural for many dynamical types of data. Two simple operations are always defined for stack: **pop, push**. The action of these operations is opposite.

Stack as a local computing environment.

In addition to the data storage functionality it is possible to use of the stack contents as the computing environment. In principle it is quite enough to add two operations (read, write) that provide the access to the upper elements of the stack.

The choosing of hierarchy of stack's memory

The stack represents the certain hierarchy where the top element has the most operating accessibility in writing and reading, and the lower element can be accessible by sequential using of command POP. Other elements occupy intermediate position. The problem is in the creation of the effective hierarchy of stack's memory. Let's esteem three main modes of accessing to stack elements.

Stack Frame mechanism is applied in some RISCs [PaS81], when the instruction gives an access not to the global registers but to their local maps, for example:

```
add L4, L4, L2
```

When the subroutine is called the new local frame is allotted. When it's restored the frame is released. The special logic follows for the exchange between the register file and the main memory. If the register file is filled up, the oldest values are unloaded. When we approach to the low bound of the register file the logic performs uploading of data from the memory.

Local variables mechanism differs from the previous subjects. We see that the memory access of the stack is carried out by means of instruction which load the local variable on the top of a stack and store of the top in a local variables, as it is made in Java VM [Sun95]. All operations are performed on the stack's top:

```
iload_4  iload_3  iadd  istore_5
```

Stack manipulation mechanism is the most advanced in Forth-processors. Its work is effective only with two or tree upper elements of stack. The others is in the local memory and the access to them is carried out by means of stack manipulations:

```
>R ROT OVER + -ROT R>
```

The Hardware Stack as Functional Unit

The storing of the procedure's return addresses is the most typical stack application. Modern machines usually have some sort of hardware support for return address stack. If the call/return time is a critical factor, the stack realises as an independent functional unit.

The Hardware stack support is a main means of acceleration the stack operation. The hardware stacks are used both in general-purpose processors, and in digital signal processors. For example is the ADSP-21xx [ADSP] which have a four stacks. But each of them are keep only one special form of the date. However the instruction set of this processor is a register based. This is not a stack-processor.

Stack Machine

The machine that use stacks as their primary data handling mechanism is named Stack Machine [Koo89]. It is considered that the programming of the stack machines is easier than conventional machines, and that stack machines programs run more reliably than other programs. As shown in [Koo93], stack machines are also much more efficient in running certain types of programs than register-based machines, particularly programs that are well modularised.

In addition to the hardware stack machines the virtual stack machines are widely used also. For example the JavaVM [Sun95] developed by Sun Microsystems, Inc. that allows to execute the same code on the different platforms. The JavaVM allows building the compact portable and reliable code.

The key property of Stack-processors is the visibility of stack operation at the architectural level. This allows use the stacks as main storage instead of the registers.

The existent stack-processors are demonstrated their good properties in the general purpose tasks but they have a very low performance in the digital signal processing.

Optimising compilers for register machines perform register allocation to improve code efficiency. Similarly, stack machines would profit from stack allocation, i.e., mapping variables to the stack [MaE98].

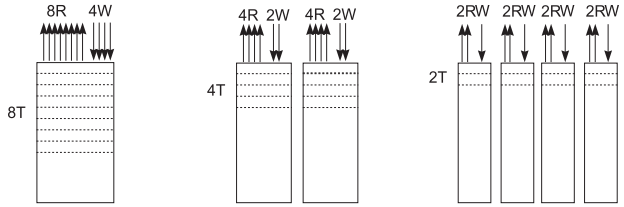
All modern programming high-level languages use stacks, but the development of the stack concept in general is connected with Forth language. The Forth language openly let to have two stacks (stack of parameters and stack of returns) as a main universal operating data structure. The first completely stack-processor was created for hardware supports of this language.

Parallel Stack Access.

Stack is a sequential device. For providing higher bandwidth it is necessary to extend stack paradigm. The functional units of the ILP processor must have N-r ports for reading and N-w ports for storing current values. One can mark two different approaches to expanding stack.

First principle consists in representing the stack's top as multi-port register file. The special unit is doing operation load/store in main stack's memory automatically. This mechanism has been repeatedly used in some processors [MO98]. But the complexities of hardware register file realisation expect another solutions.

Another solution is multistack architecture. When the several simple stacks are connecting into one processor.



3. The sTTAck concept

ILP roots of the architecture

To achieve performance goals, modern processors use a general class of architectural features known as Instruction Level Parallelism (ILP) to exploit the fine-grained parallelism available in most algorithms. In most cases modern DSPs and GPPs has a non-regular architecture, and consequently is poorly accommodated to automatic creation of an effective code [LDK98]. Another processor type has more regular architecture - Very Long Instruction Word (VLIW). New processor's generation created on this base have an increased performance [Tex97], [Cas94]. The significant successes are reached in creation of compilers for such processors [SSO97] too. One VLIW-stack-processor is known [Pay96].

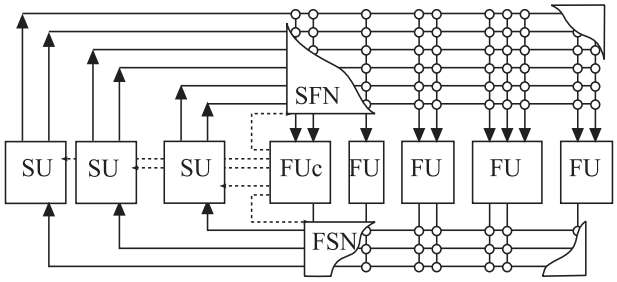
As a further step it is possible to consider the new class: transport-triggered architecture (TTA). The key property of TTAs is the visibility of all data transports (within the processor) at the architectural level.

Proposed architecture can be classified as modified TTA, with split read-write network.

The key feature is the hard binding of stacks and personal transport buses. As the result we deal with more uniform buses by structure (read-only and write-only).

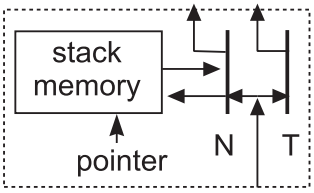
Component parts of the architecture

The are four types of units in the sTTAck architecture.



Functional units (FUs) aimed for data processing, external access and control operations. The number of FUs, destination, number of inputs (FUi) and outputs (FUo) can have wide range. Adding of inputs and outputs can extend the functionality of unit. It is necessary to found a compromise between complicated multifunctional FUs and simple single-functional units. It is obligatory the control device (FUc) in any processor.

Stack Units (SUs) are equal and aimed for operative storage and giving an access to local variables. For what each stack have input and two outputs and a possibility to execute stack manipulations. The number of stacks (S) also can be different. The internal structure of SU is rather simple. Two registers: Top and Next element of stack; little block of stack memory and automatically modified stack pointer.



S-F Network (SFN) - Interconnection network connecting SU's outputs to FU's inputs by dimension *2S*Fui*B*, where *B* - uniform bit precision of the processor. The network keeps its state itself executing switch commands and makes fundamental rule: *only one input line can be connected on every-one output.* In conflict situation oldest connection is dropped. The connection matrix may be reduced.

F-S Network (FSN) - Interconnection network connecting FU's outputs to SU's inputs by dimension *S*Fui*B*. This network works similarly.

Architectural Interactions

We can mark four interactions in the proposed architecture.

Global interaction. Any of FU may have interacted with an outer world. Every port's or memory interface can be organised as FU. This interaction has the greatest scale and the time of interchanging can be much more than time of command cycle.

Network interaction between SU and FU via SFN and FSN. All units take part in this co-operation but only unlike unit's types may interchange so. Just this interaction defines the command cycle time.

Stack interaction between elements of stack without using of interconnection network. This set of operation includes all typical manipulations with two top elements and stack memory that can be pushed or popped. This interaction doesn't modified data itself but make its distribution easier.

Functional interaction accompanies all function data changing into FU's. As well as stack manipulations it doesn't use the interconnection network for work. Each function is characterised the time of propagation.

Instruction set

Possibly, the matter of the command format must be study most carefully. While it is possible to offer only the most common approach.

The long instruction word (LIW) is divided on S subinstructions, each of them is referred to the determined stack and subnetwork linking it with FUs. We have some field with different purpose into each subinstruction. Let's examine, what operating can be executed concurrently in account on one stack:

a. Connecting of one of FU's output with SU's input line. *FUout(i) -> stack.*
b. Connecting of output T with FU's input. *T -> FUinp(i)*
c. Connecting of output N with FU's input. *N -> FUinp(i)*
d. Stack manipulations - steady set of operations caused movement of data into stack and its capture through input. *Stack(Memory <-> N <-> T) <-*
e. The field of conditional execution. If this field is enable the state of stack's input determines whether a subinstruction for is executed or not.

The real instruction format

Direct storage and transferring of microinstructions may be very expansive. It is necessary to analyse the redundancy for elaboration of the packed instruction format.

Simple decoder or microprogram memory may be designed in each concrete case.

```
: Vector.Summ ( A N -- S )
0 SWAP
FOR
  OVER @ + SWAP CELL+ SWAP
NEXT
NIP ;
```

Figure 4.1

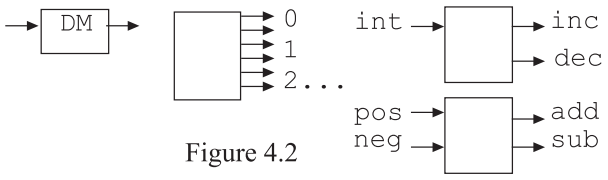


Figure 4.2

```
1:              (    pc ) |               ( N A ) |     pos.neg.0          ||
2: dup    .dec  ( L1 pc ) |   int.DM.dec  ( N A ) | cup        .DM ( 0    ) ||
3: if drop dup  ( L1 pc ) | nip cup .inc  ( A n') | cup      .add ( s Ai ) ||
4:              ( L1 pc ) | nip cup .dec  ( n'A') | 2drop cup .DM ( s'   ) ||
5: 2drop                  | 2drop                                ( S    ) ||
```

Figure 4.3

4. Features of the architecture

Let's examine the small example of the implementation of the word computing the sum of all elements of a vector (see figure 4.1). Three stacks and small number of units (figure 4.2) are enough in our case. In figure 4.3 the implementation of the word on an assembler is shown. Let's esteem features of the architecture on this example.

Each line of an assembler text is a single instruction. The character | is a separator between subinstructions for different stacks and the character || is the end of an instruction.

Stack0 (first column) is present at any configuration of the processor and has distinctive features. His top always contains current program counter (PC) and is used by the control unit. All control operations are referred to this stack:

```
drop - return
nip  - rdrop
cup  - call
...
```

IF statement. In contrast to majority of traditional architectures where only the branch operations have a conditional execution mode in sTTAck architecture (as in TTA) each subinstruction can be made conditional individuality. IF statement is enable the field of the conditional execution.

The commutations are the commands that are responsible for control operations. Its format is: `outN.outT.inp` where names of inputs and outputs are indicated through the dot in the right order.

The literal is addressed as the FU with outputs *0,1 … N*. It is inexpedient deriving of the large numbers (more than 32) thus. If the subroutine receives the address of a data frame on the stack the short literal can index this frame. Thus, the long literal and branch addresses can be obtained. The code, which has not long constants, has the greater degree of reentrancy. In connection with the larger number of stacks, the subroutines can receive/restore more arguments, without complex stack manipulations. As the parameters can be passed on any stack, their notation is required.

As it is shown in example there is no complete analogy between words of a Fort and commands of the processor, and it is required the special optimiser parallelize translator for code generation.

5. Development tools

Mainly the development tools are based on the principles of TTA framework [Cor98]. The hardware and software models have the unified configuration file and FU libraries. The hardware model is responsible for ASIC synthesis, and the objective estimation of hardware complexity, power consumption and time parameters. Only VHDL model for FPGA synthesis is carried out now. The software development tools (assembler, HLL translator and debugger) are using for generate the code which is parallel on the instruction level. It gives statistics on hardware utilisation. The FORTH-based system is developed now.

The different set of FUs and number of stacks can be necessary for different applications. The iterative process of a configure should power up analysis of the hardware complication of the synthesized model and outputs of the program.

Implementation of the high-level language translator is a separate challenge. It is possible to offer only the main directions of research here.

Translation from intermediate representation.

The application can be compiled in the sequential representation, be stored and be passed in it. Such shape can become the convoluted threaded code. At the node with a specific configuration (number of stacks and set of units) the translator will convert the code into a parallel object code. It can be used JAVA byte-code alternatively. The optimisation of stack allocation [Mai98], make base-block longer by use of implementation of the code of often used

words, instead of their call, loop unrolling, speculative execution are only brief list of perspective translator's functions.

Conclusions and Further Work

The persist attempts of hardware implementation of Forth-processors is the visual proof of vitality of this idea.

We have presented some of the more interesting details of sTTAck architecture. The realisation of proposed architecture and software tools for it are very interesting field of researches.

The big further work under model lie ahead. Testing of the model on the FPGA expected. Generation of efficient machine code for sTTAck demands for new optimisation techniques.

Acknowledgments

References

[PLC97] P.G.Paulin, C.Liem, M.Cornero, F.Nacabal, and G.Goossens, "Embedded software in real-time signal processing systems: Application and architecture trends," Proc. IEEE, Vol. 85, No. 3., pp.419-435.

[Koo89] P.J.Koopman, Stack Computers. Ellis Horwood, Chichester, 1989.

[Fish83] J.A.Fisher, Very Long Instruction Word Architectures. 253 Yele University, 1983.

[Cor98] H.Corporaal, Microprocessor Architectures from VLIW to TTA, Wiley&Sons, Inc., New York, 1998. http://cs.et.tudelft.nl/~heco/

[TiM95] C.Ting, C.Moore, MuP21 - A High Performance MISC Processor, Forth Dimentions Jan. 1995. http://www.dnai.com/~jfox/mup21.html

[MaE98] M.Maierhofer, A. Ertl, Local Stack Allocation, Compiler Construction CC'98, Springer LNCS 1383, pages 189-203. http://www.complang.tuwien.ac.at/papers/

[Koo93] P.Koopman, Usenet Nuggest: Why Stack Machines? Computer Architecture News, Vol.21, No. 1, March 1993.

[Pay96] B.Paysan, A Four Stack Processor, diploma, 1996. http://www.jwdt.com/~paysan/4stack.html

[Ptsc] Patriot Scientific Corporation, The PSC1000 ShBoom Processor. See: www.ptsc.com/shboom/

[Sun95] The Java Virtual Machine Specification, Sun Microsystems, 1995. http://java.sun.com/docs/books/vmspec/

[PaS81] D.A.Patterson, C.H.Sequin Proc. Ann. Symp. Comput. Archit. 8th, Minneapolis, Minnesota, 443-457, 1981.

[LDK98] S.Liao, S.Devadas, K.Keutzer, S.Tjiang, S. Wang, Code Optimization Techniques in Embedded DSP Microprocessors. Kluwer Academic Publishers, 1998.

[ADSP] Analog Devices. Various ADSP User's Manuals. http://www.analog.com

[Tex97] Texas Instruments. TMS320C62xx CPU and Instruction Set Reference Guide, 1997. http://www.ti.com/sc/docs/dsps/dsphome.htm

[Cas94] B. Case, Philips Hope to Displace DSPs with VLIW. Microprocessor Report, 8(16), 5 Dec. 1994, pp.12-15.

[SSO97] R.Sakellariou, E.Stohr, M.F.P.O'Boyle, Compiling Multimedia Applications on a VLIW Architecture. 1997.

[Way96] P.Wayner, Sun Gamblets on JAVA Chips. Byte Magazine, November 1996.

[MO98] H.McGhan, M.O'Connor. PicoJava: A Direct Execution Engine For Java Bytecode. IEEE Computer, October 1998, pp. 22-30.