Stack Structure for the efficient execution of the FORTH Language Instruction Set

Alvaro Bernal N.

Abstract

This paper presents an alternative design approach of a stack structure for the efficient execution of high level language instructions. The stack system was designed to execute a FORTH directed instruction set. The instruction set to be executed, the structure of a basic cell and the control logic are described. A 32 bit word length prototype, with 12 ns access time, using an 1.2 μm double level metal CMOS technology, was designed.

I. Introduction

In most applications, data are conveniently organized in a stack structure. Usually, the stack is implemented through a set of registers whose contents are managed on a last-in-first-out (LIFO) basis. A stack is commonly used with arithmetic and logic operations, comparison blocks and for saving the return address for subprogram calls. Also, it facilitates the evaluation of expressions and minimizes the control overhead needed to store data [1].

The processors with oriented stack architecture, also called stack machines, use a stack extensively for temporary data storage and execution of all operations on the data stored there. This extensive use of stacks leads to the creation of efficient high-level language compilers [2], resulting in very efficient machine code execution. The processor operational units find all the data in a predefined location which is implied by the operation code itself, and no explicit address is required to be decoded, resulting in faster instruction execution.

Forth is a structured programming language [3], where most operations communicate only through a stack. This language uses a two stack programming model: the operand stack and a control flow stack. In this job, we present the stack structure used in a FORTH directed microprocessor with stack-based architecture being designed in the Integrated Systems Laboratory of São Paulo University.

The cell used in the operand stack design, leads to a structure which permits both a sequential addressing, like as RAM structure and to shift the data stored into the stack, like as shift register bank, allowing to execute some composite Forth instructions in one instruction cycle, so, this fact represents the main characteristic of this configuration.

II. The stack design approaches

The FORTH two stack programming model is supported by two stack caches in the data path. Both operand and return stacks consist of thirty-two words of 32 bits register. Most Forth primitives take operands from one or both stacks, push or pop the stacks and return a result to one of the stacks.

There are two common ways to implement stacks, first, using a shift registers bank and second, like a RAM structure.

A. Shift Register Approach

Lets consider a typical shift register approach. The stack is implemented using a n-array of shift registers (SR) $S_1,....S_n$, and only four control signals must be generated for each bit cell. The control signals allow to execute three basic operations:

PUSH: (S_{i+1} <------ S_i , $1 < i < n-1$)
POP : (S_i <------- S_{i+1} , $1 < i < n-1$)
NOP : (S_i <------- S_i , $1 < i < n$)

In a LIFO structure, this configuration allows to push and pop information only for the top cell. In case of a FIFO structure, the information can be pushed on the top cell, and poped on the bottom cell. The main disadvantage of this configuration consists of the high power consumption when a POP or PUSH operation is executed, due to the fact that all previous stored data must be shifted. See figure 1 [4].

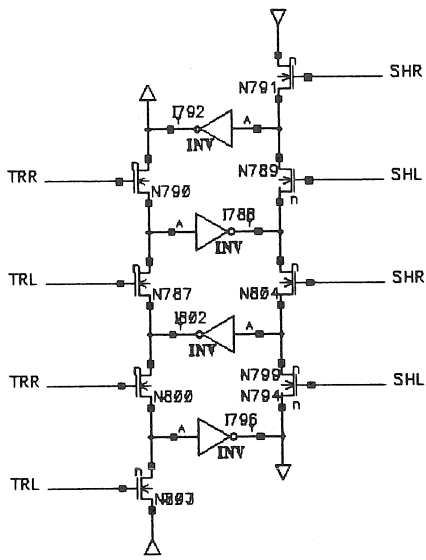


Figure 1. Array of shift register

B. RAM approach

Another possibility to implement a stack, is like a memory structure. Only few pointers are needed to indicate the actual top address. The stack pointer contains the address of the most recent stored item. The operations PUSH and POP would be executed as:

PUSH:
S_i <-- S_{i+1}, $1 < i < n$
RET[S_i] <-- SP

POP:
SP <--- RET[S_i]
S_i <--- S_{i-1}, $1 < i < n$

In the shift register approach, the pointer always is addressing the same top position and the data are shifted down or up. In the RAM approach, the data are stored in fix positions and the access to any location can be realized through the variable pointer. This configuration allows to read the last stored item but does not allow to shift data into the structure. Furthermore, due to the fact that the addressing is sequential, not is allowed to have random data access from the structure.

III. Stack Structure

Our stack system has been designed mixing the most interesting concepts from both shift register and RAM approaches. So, the operand stack allows partial random access and the return stack is a pure LIFO. Figure 2 shows the block diagram of the operand stack, bellow we describe them:

A. Return Stack

The return stack was implemented as a pure LIFO structure. The memory cell has one read port and one write port. An array of thirty-two words of 32 cells was used (total 1024 bits). The return stack stores the return address when a call instruction is executed, this fact permits a very fast sub-routine return mechanism.

In the return stack's design, a basic six transistor static CMOS memory cell was used [5]. In this case the PMOS-load transistors are small, since they act only as a latch, and must not drive the bit line. Due to the fact that the bit line are precharged to one, the NMOS pull-down transistor was designed properly in order to drive fast enough these line.

The address and control blocks of the return stack consist of an adder/sub, a 5 to 32 bits decoder and a read/write logic. Flags from the add/sub block trigger the appropriate trap instruction in the control unit. See figure 2(a).

B. Operand Stack

The operand stack is used to store operands, to call arguments or to hold some values, it is interconnected by three buses to other elements of the operative part, for this reason the instruction set is based on a triadic model.

From the control section of the operand stack, two big blocks can be observed, first one allows partial random access to the stack, so, like as [6], it is possible simultaneously to access the top position and one of the three adjacent locations. The stack pointer contains the address of the most recent stored item, and three pointers more indicate the second (S_{n-1}), third (S_{n-2}) and fourth (S_{n-3}) locations down, related to the stack pointer.

The possibility to access the top four positions, leads to a single-cycle implementation for some of the Forth stack instructions. So, this means that one primitive of the language can frequently be implemented with one instruction. So, if a instruction will be executed, the first data may be at the top stack location, the second will either one, or two, or three locations down, and the result data is then stored in one of these positions.

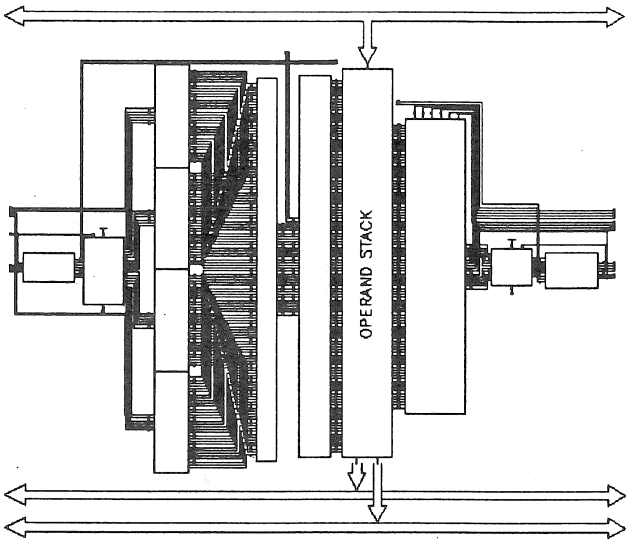


Figure 2. Operand stack block diagram.

Two read commands permit simultaneously transfer two operands from the stack through two buses to the operator unit. In a stack oriented architectures the result of an operation is transferred from the operator through another bus to the stack. By the way, all instructions use as data two of the last four operands stored into the stack.

Furthermore, the possibility to shift or rotate one position, the data in the four top locations in just one clock cycle, represents the main feature of this implementation. As a matter of fact, this structure allows to execute FORTH composite instructions in one instruction cycle.

The shift/rotate operations have a variable depth and can operate on two, three or four items. The control logic has to active only those pointers which define the appropriate locations. The second block of the control logic select the appropriate locations to execute the variable-depth rotate operation.. The stack was implemented as a variable-four-position shift register function.

The push operation causes the stack pointer to increment and a pop causes the pointer to decrement. The stack pointer is incremented/decremented through an adder/sub block, starting from the S_1 first position. This block generates the overflow and the underflow flags in the S_n and S_1 positions, respectively.

The size of the operand stack has been studied through benchmark programs. From this analysis, a stack size of 32 words would reduce the stack overflows to less than 1% [7]. Each of the registers used in the data stack requires a cell with one write port and two read ports connected to three buses. In the next item, the basic cell used is described.

IV. The operand stack customized Cell

The cell design was based on the traditional cross-coupled inverter with pass transistors inserted between them. Therefore, it consists of two separate left and right shift registers, each one using recirculating-type cells. A bidirectional shift register is very simple to implement in a MOS technology thanks to the possibility of using pass transistor [8]. The pass transistors allow to shift up or shift down the stored data into the stack. Furthermore, the cell has three ports with access to three buses architecture [9]. So, two-read ports and one-write port were considered. See figure 3(a) for more details.

The operand stack employs a three port CMOS cell using a weak inverter and two read bit lines. A good feature of this cell is that simultaneous read operations on the stack can be performed. Each port of the cell has an independent word select, global control, data and bit lines. See figure 3(b).

Due to the regular structure of the array, a well defined layout strategy of the global control, power and ground lines for the stack system can be achieved. All the word lines are running perpendicular to the data lines as is shown in figure 4(b).

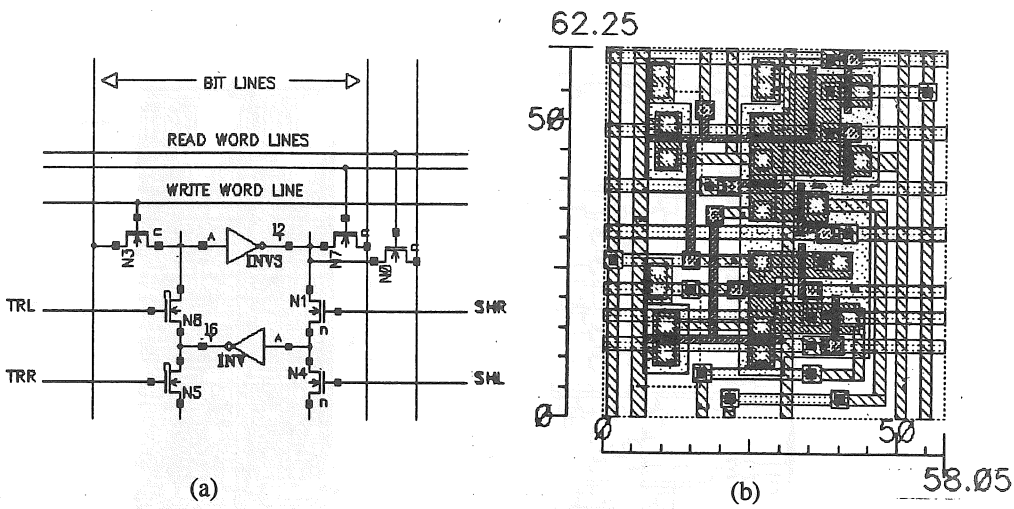


Figure 3. Operand stack basic cell. (a) Electrical diagram. (b) Layout

A. Read-Write Operations

The microprocessor global control is clocked by four non overlapping phases, the cell allows to execute the read-write operations in two consecutive phases of the cycle instruction. Due to the fact that write operation is executed by writing data stored in the bit line, only one N-ytpe pass transistor is involved in the operation, so, to transfer the data into the recirculating-type cell, the write and appropriate row select signals must be activated.

When a read operation is performed, an independient precharge pulse for each bit line is generated. By precharging the bit lines to one-level, only one NMOS transistor to access the cell is required, and only the N-type of the inverter has to be large enough to drive the long bit lines to zero. The PMOS load of the inverters can be kept small.

In the read-write operations those pass transistors which interconnect the memory cells remain inactive and the stack operates as a conventional RAM structure.

B. Shift Operations

In this structure, four additional control signals for each bit cell are required for executing the shift operations. The SHR(Shift Right), SHL(Shift Left), TRR(Transfer Right) and TRL(Transfer Left) signals are controlled to perform the rotate operations. When a shift down operation is being executed, the TRL, SHR and SHL signals are turned off and TRR is turned on. In the next cycle, TRL and SHR are turned on, TRR and SHL are inactived. The shift-rotate and read operations could not be excuted simultaneously, for this reason, three phases of clock are required to execute some instructions .

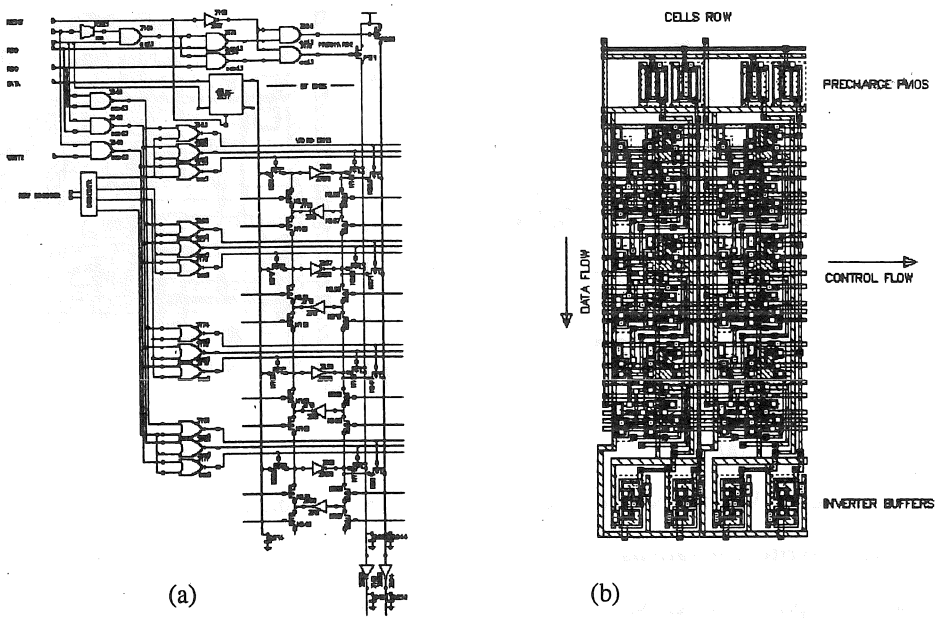
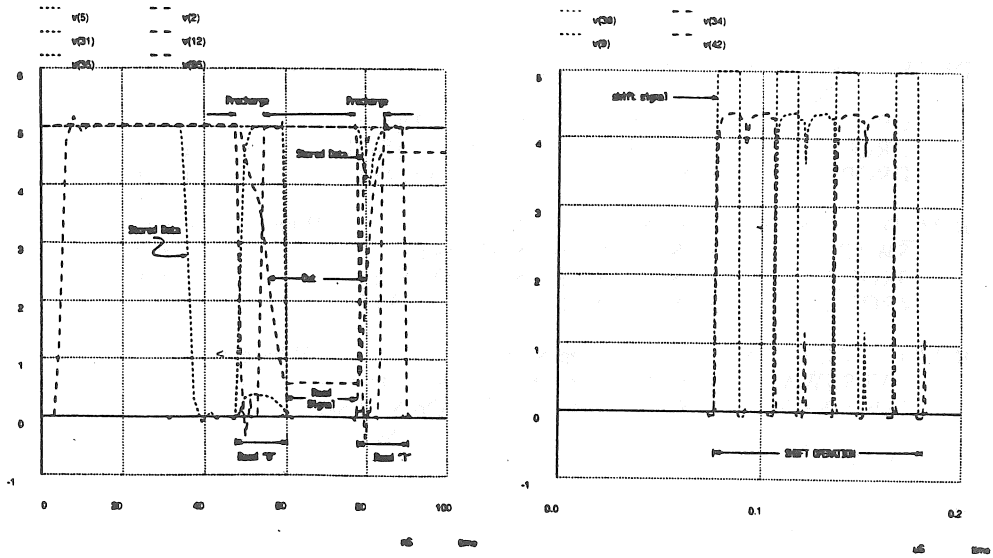


Figure 4. (a) Simulation circuit. (b) Layout strategy

Simulations were done using the SPICE 3D2 electrical simulator considering a bit line capacitances of 1.4P, the circuit used to simulate the cell is shown in figure 4(a). In figure 5(a) the results of the electrical simulation for write, precharge and read operations are shown. Figure 5(b), depicts the simulation results for shift operations.



(a) (b)

Figure 5. Simulation results. (a) Read-write operation. (b) Shift operation

V. The Instruction Set

In this section, instruction set to be executed by the stack system is described. In this visualization method we consider that the item on the rigth denote the top of the stack (S_n), S_{n-1} represents the location directly below it and so on. An single manipulation on the operation stack can simplify a program and decrease his execution time. The instruction set will be executed by controlling the stack pointer and the shift control signals. The stack system can execute both unary and binary instructions.

Unary Instructions

DUP(i)	($S_1,S_2.....S_n$ ---> $S_1,S_2.....S_n,Op[S_i]$).	Where $n-3 \leq i \leq n$
DUP	($S_1,S_2....S_n$ --- $S_1,S_2,...S_n,Op[S_n]$)	$i = n$
OVER1	($S_1,S_2....S_n$ --- $S_1,S_2,...S_n,Op[S_{n-1}]$)	$i = n-1$
OVER2	($S_1,S_2....S_n$ --- $S_1,S_2,...S_n,Op[S_{n-2}]$)	$i = n-2$
OVER3	($S_1,S_2....S_n$ --- $S_1,S_2,...S_n,Op[S_{n-3}]$)	$i = n-3$

PICK(i)	($S_1,S_2.....Sn$ ---> $S_1,S_2.....S_{n-1},Op[S_i]$)	Where $n-3 \leq i \leq n$
OVERSWAP	($S_1,S_2....S_n$ ---> $S_1,S_2....S_{n-1},Op[S_n]$)	$i = n$
DROPDUP	($S_1,S_2....S_n$ ---> $S_1,S_2....S_{n-1},Op[S_{n-1}]$)	$i = n-1$
PICK2	($S_1,S_2....S_n$ ---> $S_1,S_2....S_{n-1},Op[S_{n-2}]$)	$i = n-2$
PICK3	($S_1,S_2....S_n$ ---> $S_1,S_2....S_{n-1},Op[S_{n-3}]$)	$i = n-3$

DROP(i) ($S_1,S_2.....S_n$ ---> $S_1,S_2....S_{n-2},Op[S_i]$). Where $n-3 \leq i \leq n$

SWAPDUP	($S_1,S_2....S_n$ ---> $S_1,S_2....S_{n-2},Op[S_n]$)	$i = n$
DROP	($S_1,S_2....S_n$ ---> $S_1,S_2....S_{n-2},Op[S_{n-1}]$)	$i = n-1$
DROP2	($S_1,S_2....S_n$ ---> $S_1,S_2....S_{n-2},Op[S_{n-2}]$)	$i = n-2$
DROP3	($S_1,S_2....S_n$ ---> $S_1,S_2....S_{n-2},Op[S_{n-3}]$)	$i = n-3$

ROT_D(i) ($S_1,S_2..S_n$ ---> $S_1..S_{i-1},S_{i+1}..S_n,Op[S_i]$) Where $n-3 \leq i \leq n-1$

SWAP	($S_1..S_n$ --> $S_1..S_{n-2},S_n,Op[S_{n-1}]$).	$i = n-1$
3ROT_D	($S_1..S_n$ --> $S_1..S_{n-3},S_{n-1},S_n,Op[S_{n-2}]$).	$i = n-2$
4ROT_D	($S_1..Sn$ --> $S_1..S_{n-4},S_{n-2},S_{n-1},S_n,Op[S_{n-3}]$)	$i = n-3$

ROT(i) ($S_1,S_2...$ S_n ---> $S_1..S_{i-1},Op[S_i],S_{i+1}..$) Where $n-3 \leq i \leq n$

OVERSWAP	($S_1,S_2..S_n$ --> $S_1..S_{n-3},S_{n-2},S_{n-1},Op[S_n]$)	$i = n$
2SWAPOVER	($S_1,S_2..S_n$ --> $S_1..S_{n-3},S_{n-2},Op[S_{n-1}],S_n$)	$i = n-1$
3ROT	($S_1,S_2..S_n$ --> $S_1..S_{n-3},Op[S_{n-2}],S_{n-1},S_n$)	$i = n-2$
4ROT	($S_1,S_2..S_n$ --> $S_1..Op[S_{n-3}],S_{n-2},S_{n-1},S_n$)	$i = n-3$

ROT_U(i) ($S_1...S_n$ ---> $S_1..S_{i-1},Op[S_n],S_i..S_{n-1}$) Where $n-3 \leq i \leq n-1$

SWAPR	($S_1,S_2..S_n$ --> $S_1..S_{n-3},S_{n-2},Op[S_n],S_{n-1}$)	$i = n-1$
3ROT_U	($S_1,S_2..S_n$ --> $S_1..S_{n-3},Op[S_n],S_{n-2},S_{n-1}$)	$i = n-2$
4ROT_U	($S_1,S_2..S_n$ --> $S_1..Op[S_n],S_{n-3},S_{n-2},S_{n-1}$)	$i = n-3$

SHF(i) ($S_1...S_n$ ---> $S_1...Op[S_i],S_{n-3},S_{i\pm1},S_n$) Where $n-2 \leq i \leq n-1$

SHF2	($S_1,S_2...S_n$ ---> $S_1..Op[S_{n-1}],S_{n-3},S_{n-2},S_n$)	$i = n-1$
SHF3	($S_1,S_2...S_n$ ---> $S_1..Op[S_{n-2}],S_{n-3},S_{n-1},S_n$)	$i = n-2$

The **binary instructions** only add one second operand to the operation. The instruction cycle is not increased.

VI. Conclusions

The implementation of a stack cache for an application-specific in the FORTH directed microprocessor with stack-based architecture, has been presented. The operand stack uses a cell with three port access which allows shift up or down the data stored in the stack.

Data are transferred directly into the stack operand through one write port. The two read ports allow simultaneously get two operands and carry-out to the microprocessor operational units.

Furthermore, the possibility to shift the data one position up or down, is a main feature of this implementation. The presented solution proved to be very efficient to perform composite FORTH instructions in one instruction cycle. The stack system presented was designed in 1.2 µm CMOS technology, the address and control blocks were designed using the standard cell library from ES2 silicon foundry. The size of the cell was 59 µm x 63 µm and it provides a highly regular structure leading to a well defined layout strategy.

VIII. References

[1] P.J. Koopman, Jr, Stack Computers-The new wave, Ellis Horwood Limited, John Wiley & sons, New York.

[2] W. P. Salman, O Tisserand, B. Toulout, "FORTH", Macmillan Computer Science Series, Editions EYROLLES, París, 1983.

[3] J.S. James, What is FORTH ? A tutorial Introduction, Byte Publications, August, 1980.

[4] P.E. Danielsson, A Variable-Length Shift-Register, Correspondence, IEEE Transactions on Computers, Vol c-32, No 11, November, 1983.

[5] K. O'Connor, The Twin-Port Memory Cell, IEEE Journal of Solid States Circuits, Vol sc-22, No 5, October, 1987.

[6] S.C. Lee, J.R. Hayes, Development of a FORTH Language Directed Processor using Very Large Scale Integrated Circuitry, John Hopkins APL Technical Digest, Vol 10,Number 3, 1989.

[7] D. L. Miller, Stack Machines and Compiler Design, Byte Publications, April, 1987.

[8] A. Mukherjee, "Introduction to NMOS and CMOS VLSI System Design", Prentice Hall, 1986.

[9] M. L. Anido, D. J. Allerton, E. J. Zaluska, A three-Port/Three-access Register File for Concurrent Processing and I/O Communication in a RISC-LIKE Graphics Engine, ACM, 1989.