A Forth Machine for the S-100 System

R. C. Grewe and R. D. Dixon

Wright State University
Dept. of Computer Science
Dayton, OH 45435

Abstract

A three-board processor which contains a Forth interpreter, executing on AMD-2900 bit-slice components, high speed memory and hardware stacks, has been designed and built for use in the S-100 bus environment. The Forth system operates independently of the S-100 system processor and memory, and can only be accessed from the bus by ports. The S-100 system processor is free to do I/O and the existing system uses the CP/M operating system to support the operation of the Forth system.

Introduction

The straightforward design of the Forth virtual machine and the availability of public domain implementations of both the virtual machine and a usable computing environment, make Forth a good target for a student machine design project. This paper is the description of the result of such a project by a senior computer engineering student using available systems and components. In that sense the design may not be optimal, but it is simple and easy to understand; and the performance is promising enough that it indicates more serious designs along this line would be useful. Our design is somewhere between the specialization of the Rockwell FORTH processor described by Dumse [3,4] and the FORTH Engine of Winkel [6].

The paper is organized into six sections. The first covers the basic design of the system, and the second covers the communication between the Forth and the host system. The third section examines the performance of this system relative to some other familiar systems. The forth and fifth cover the structure of the microcode and an example. The last covers what we learned and some improvements we will make on future designs.

The Basic Design

This Forth system, which is called RUFOR, is built on three standard S-100 wire wrap boards. The CPU and the high speed stacks share a single board; the system control unit resides on a second board; the memory and S-100 bus interface take up a third. Two busses, cabled across the three boards, provide communication between RUFOR's control unit, CPU, memory and S-100 interface.

The block diagram of the RUFOR system is shown in Figure 1. The CPU is based on a 16-bit ALU consisting of cascaded 4-bit microprocessor slices (AM2901). The CPU uses bipolar technology which is capable of a 125-ns cycle time. (Because we do not have high speed memory chips available, the cycle time is currently 500-ns.)

The Forth primitives are micro-coded in a 148K-bit control store, and the CPU processes instructions using a pipelined architecture as described in [1, page 6-168]. This

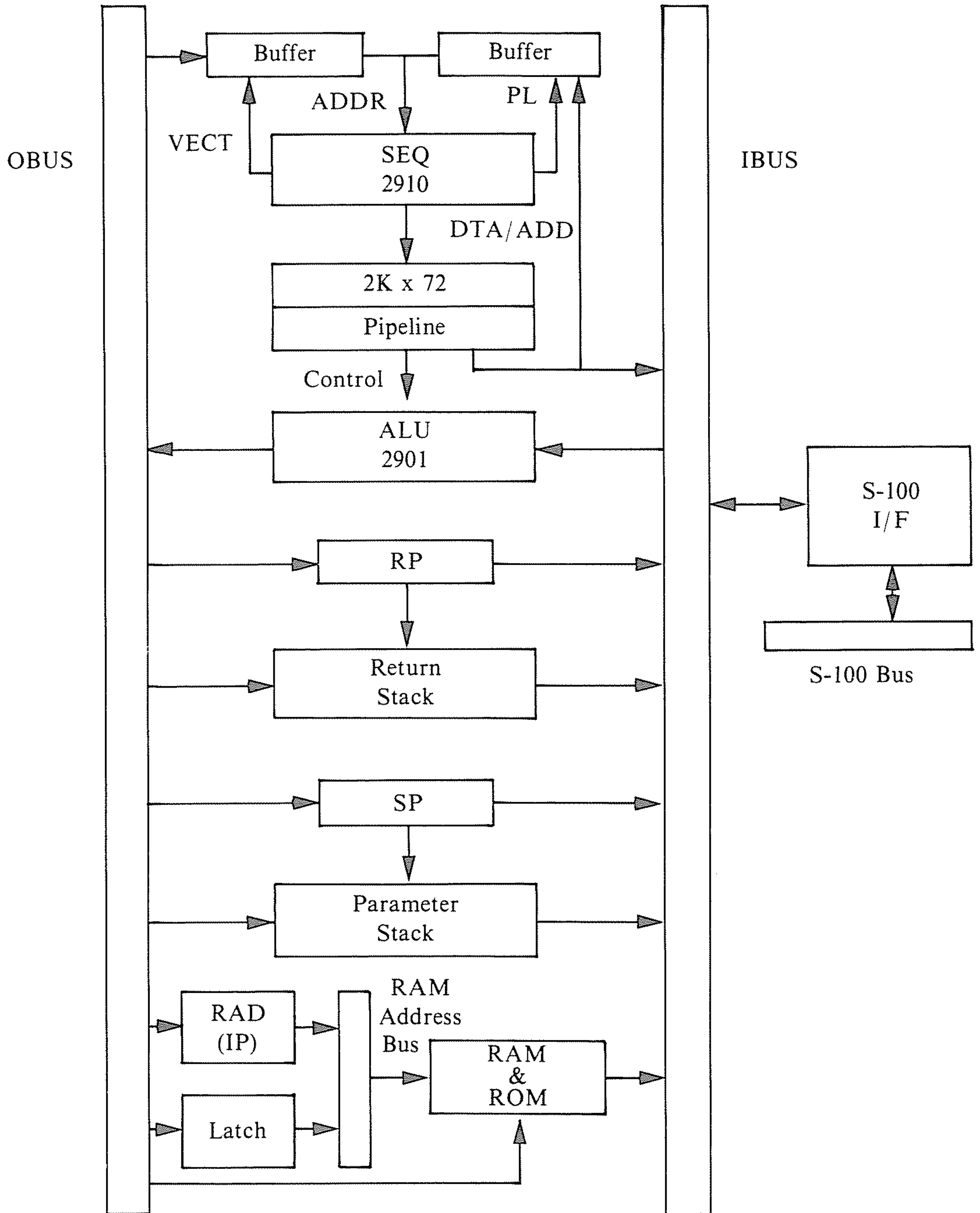

Figure 1. The System Block Diagram of RUFOR.

design improves overall performance by allowing the fetch of the next micro-instruction to occur during the execution of the current one. The control store is made of 250-ns EPROM's (55-ns PROM's are necessary to achieve a 125-ns cycle time), with each micro-instruction coded in a 72-bit horizontal microword which is divided into several independent control fields.

This separation of control fields allows a simple specification of parallel hardware operations. The structure of the hardware facilitates the execution of many Forth operations in a single microcycle.

Two stacks, containing 256 16-bit words accessible in 35-ns and placed on the CPU board, enhance the execution of Forth instructions. Separation of these stacks from user memory was necessary to gain the parallelism used to complete stack to stack and memory to stack transfers in a single cycle.

Additionally, the top entry of each stack resides in a general purpose register on the control board. This gives the CPU parallel access to the top two elements of each stack. This access allows the completion of Forth arithmetic and logic operations in one cycle. Figure 2 shows the basic configuration of these components.

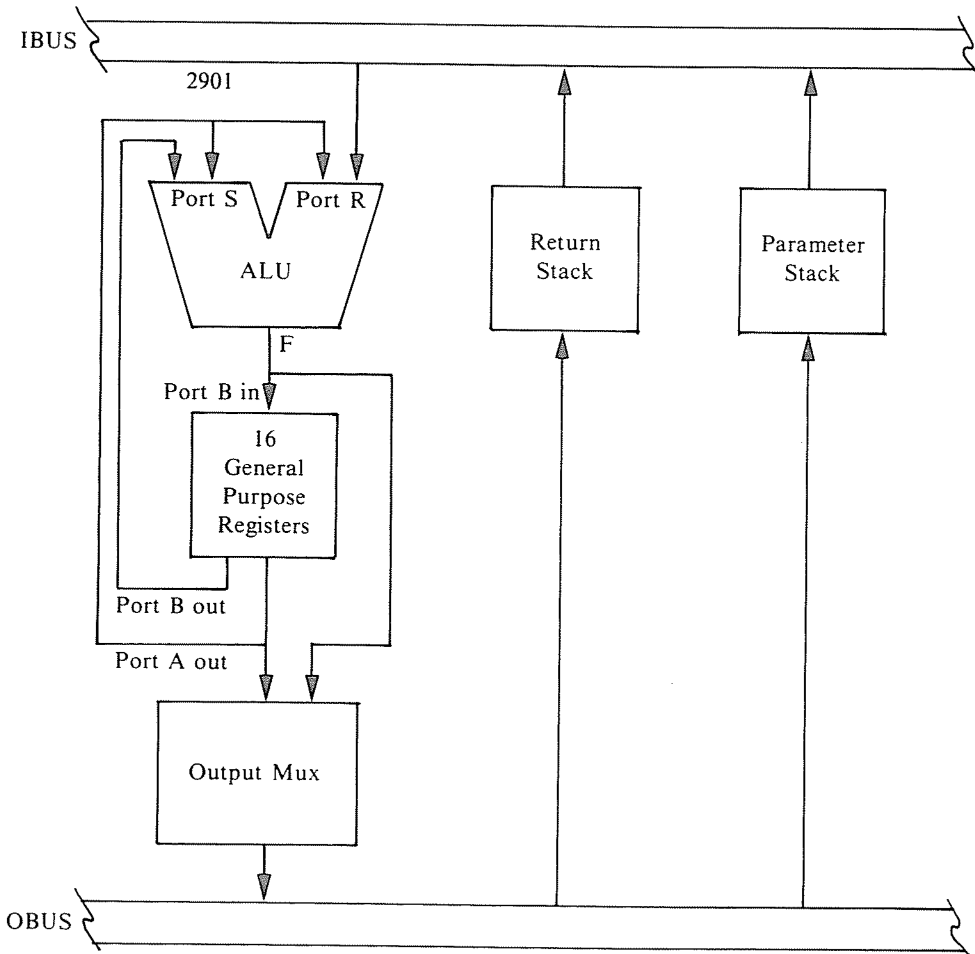

Figure 2. The ALU Flow Diagram

The RUFOR cache memory board was constructed to accept 32K-bytes of 55-ns RAM. (Currently the board contains 8K-bytes of 200-ns RAM.) RUFOR can transfer data between two memory locations in two cycles. User variables and dictionary entries are kept in this memory. Space has been reserved for a few disk buffers in cache but this has not yet been implemented. The S-100 system memory is currently used to handle all disk buffers and the code for operating the disk.

4K-bytes of EPROM on RUFOR's memory card hold definitions of 200 pre-defined Forth instructions so that RUFOR is a fully operational Forth machine on power-up and does not need to be down-line loaded from the host. Thus RUFOR's cache memory is all available for user definitions. The EPROM is expandable to 32K-bytes to allow more frequently used programs, e.g. an editor, to be placed in EPROM. The coding in the EPROM's follows John James' PDP-11 fig-Forth [5].

Communications Between RUFOR and the S-100 System

Communication with RUFOR from the S-100 system is accomplished by a port addressed data/control register and a separate port addressed status register. The status register contains only one meaningful bit, called a start bit, which is used to handshake data/control bytes between the host processer and RUFOR. Proper data/control transfers from the host are accomplished by the host first writing to the data/control register and then writing a 1 to the start bit. This assures RUFOR that the data/control byte is valid prior to the reception of the start bit. To start RUFOR, the host writes RUFOR's starting address, START, into the data-control register and sets the start bit. (START is less than 256 since the S-100 port is 8-bits.) Once in operation, RUFOR's program will request service from the host by writing an 8-bit control byte into the data/control register, and then resetting the start bit. Since our host system does not use interrupts, it must poll this start bit to find out when RUFOR needs service. This strictly half duplex protocol must be maintained for proper operation of the system.

The following are 1-byte control codes RUFOR uses in communicating with the host:

R.BYE	;RUFOR encountered a BYE instruction, execution terminated.
R.ODD	;RUFOR tried a word operation on an odd address, execution terminated.
R.RESET	;RUFOR received a hardware S-100 bus reset, execution terminated.
R.EMIT	;RUFOR encountered an EMIT instruction. After host acknowledgement RUFOR will place the outgoing character into the data/control register.
R.KEY	;RUFOR encountered a KEY instruction. The host must transfer a character into the data/control register and acknowledge by setting the start bit.
R.TERM	;RUFOR encountered a ?TERMINAL instruction. The host must test its terminal and transfer a zero if no character has been received, else transfer the character code into the data/control register and then set the start bit.

The disk handling protocol for RUFOR has not been programmed and so the loading of user programs into RUFOR is a little awkward. The host program that communicates with RUFOR (RUFORCOM) is also written in fig-FORTH [2]. Thus, one loads the host Forth and RUFORCOM and an editor if needed. The screens which contain RUFOR programs are edited on the host and the editor is exited. RUFORCOM is entered and instructed to load those screens to RUFOR. RUFOR receives those definitions as though they were being typed from the terminal and enters them into its dictionary. RUFORCOM then enters a transparent mode where the host terminal appears to belong to RUFOR. Since we have only used RUFOR as a test system, this has proved satisfactory.

RUFOR's Performance

This section provides a list of some of the Forth nucleus operations and the number of microcycles RUFOR takes to execute them. Also listed is the number of memory cycles needed to execute these operations on a PDP-11 [5]. The NEXT macro which is included in all other operations is listed first and its time is not included in the figures given for the other operations. For example, the time for AND is 1 in the table but is really 1 plus the 3 cycles for NEXT which is 4 cycles.

Operation	**RUFOR**	**PDP-11**
NEXT	3 *	5
LIT	2 *	3
BRANCH	2	2
0BRANCH	4	5
(LOOP)	5	10
(LOOP+)	7	13
(DO)	3 *	9
I	2 *	3
AND	1	5
OR	1	3
XOR	1	4
SP@	3	3
SP!	3	3
RP!	3	2
>R	2 *	3
R>	2 *	3
R	2 *	3
;S	2	2
0=	2	5
+	1	3
D+	4	14
SWAP	3	9
DUP	2 *	3
DROP	1	2
@	3	4
C@	3	7
!	3	7
C!	6	7
DOCOL	3	3
DOCON	2	3
DOVAR	3 *	2
USER	3	5
1+	1	2
=	3	8
ROT	5	14

The * beside RUFOR's values indicate those values that could be reduced one cycle by some minor changes in RUFOR's hardware.

Some benchmarks on simple loops were run on RUFOR, a 4-megahertz Z-80 [2], and an LSI-11/2 [5]. In these tests RUFOR's NEXT macro had two extra cycles inserted for

debugging and its clock was set at 2-megahertz. The basic loop was:

```
: TEST 100 0 DO 32000 0 DO .... LOOP LOOP ;
```

For TEST1 .... was empty;
for TEST2 .... was 2 2 + DROP ;
for TEST3 .... was I 4508 C! ;
for TEST4 .... was I 4509 C! .

TEST	RUFOR	Z80	PDP-11
TEST1	16 sec.	181 sec.	64 sec.
TEST2	41	484	193
TEST3	55	447	181
TEST4	65	447	181

In these tests, RUFOR behaves as predicted. Estimating that the inner loop takes all the time, we have that it takes 16/3,200,000 seconds or 5 microseconds to execute (LOOP) one time. (LOOP) takes 5 cycles plus the time for NEXT, which is 3 cycles and 2 cycles for debugging code. 10 cycles at 2-megahertz is, indeed, 5 microseconds.

If we remove the debugging code and replace RUFOR's memory with chips that allow a cycle time of 8-megahertz, the corresponding figure for RUFOR in TEST1 is 3.2 seconds which gives a performance ratio of 60 over the Z-80 and 20 over the LSI-11/2.

The tests using C! were included because RUFOR is a word oriented machine and these instructions behave as poorly as any. Note that the performance is not as good for the odd bytes as this requires some shifting.

Winkel [6] reports a performance factor of about 100 for his 3-megahertz FORTH Engine over a 1-megahertz 6809, which is better than RUFOR could do with a comparable clock. The implementation with a standard bit-slice processor has appeal but Winkel asserts that this method is too slow. We may abandon the AM2901 in favor of other components in our future efforts, which will be towards the implementation of a 32-bit token threaded code machine.

The Structure of the Microcode

RUFOR's microwords are 72 bits wide, divided into fields as shown in Figure 3. Most of the fields are directly wired to the components of the system with little intermediate decoding. Only the data field has two distinct uses which generate different interpretations. A short description of each field follows.

Register Definitions

The AM2901 ALU has two input ports, R and S. The register file of the AM2901 has two output ports, A and B, so that any pair of registers may be routed to the ALU on any cycle. The first two fields of the microword are used to make that selection. It is also true that the register selected from port A may be used as an output, as shown in Figure 2. In the example of microcode assembler given later, the symbols GR0, GR1, ... GR15 are used to indicate the 16 registers. U is another name for GR8 and W is GR7.

Register Definition A (4 bits)	Register Definition B (4 bits)	Source Operands (3 bits)	ALU Functions (4 bits)	Destination Control (7 bits)
CC Mux Select Field (4 bits)	2910 Microprogram Controller (4 bits)	Output Bus Source (2 bits)	Output Bus Destination (3 bits)	Input Bus Source (4 bits)
Input Bus Destination (2 bits)	RAM Address Select Field (2 bits)	Counter Control (6 bits)	Data (16 bits)	Discrete Bit (1 bit)

Figure 3. The Microword Field Structure

Source Operands

The ALU may select the inputs to its ports, R and S, from among the register ports, A and B; a zero word, Z; an internal register, Q; and direct data, D, from the IBUS. The symbols AQ, AB, ZQ, ZB, ZA, DA, DQ, DZ have the straightforward meanings for this field, with the first letter being the choice for R, the second for S.

ALU Functions

This field determines which of a small set of arithmetic and logic functions will be performed on a given cycle. They are:

```
ADDRS   ; R + S
ADDRS1  ; R + S + 1
SUBRS   ; R - S
SUBRS1  ; R - S - 1
SUBSR   ; S - R
SUBSR1  ; S - R - 1
OR      ; R OR S
AND     ; R AND S
EXOR    ; R EXCLUSIVE OR S.
```

Destination Control

This field is used to control the loading of the internal register file, and determines which value to place onto the OBUS, either the result of the operation being performed, or the contents of the register addressed from Port A. The most frequently used operations are:

```
NOP.F   ; output the result of the current operation
LDB.B   ; load the register addressed from Port B with the result of the current
          operation, and output that result
LDB.A   ; load the register addressed from Port B with the result of the current
          operation, and output the register addressed from Port A.
```

Condition Code Multiplexor Select Field

The AM2901 outputs status bits after each ALU operation. These bits, and several other hardware status bits are fed to a multiplexor. This field is used to select which one of these inputs will be passed to the AM2910 Controller for use in conditional branch instructions. The field selections are:

ZERO	; the result was zero
LT	; the result was less than zero
GT	; the result was greater than zero
COUT	; the ALU's carry out
LE	; the result was less than or equal to zero
GE	; the result was greater than or equal to zero
IB15	; bit 15 of the IBUS; this is used to check for negative numbers without requiring use of the ALU
OB15	; bit 15 of the OBUS; this is used to check for negative numbers without requiring use of the ALU
OB0	; bit 0 of the OBUS; this is used to check for odd/even addresses without requiring use of the ALU
BUSY	; the start bit from the S-100 interface.

The AM2901 Microprogram Controller

The controller can take conditional jumps to locations specified by various inputs and has some loop management facilities and subroutine jumps. Those instructions are given in [1, page 6-159].

The Output Bus Source

This field controls which signals drive the output bus (OBUS). Currently the only device driving the output bus is the ALU, but this field was included for further hardware enhancements.

The Output Bus Destination

This field is sent to a decoder that selects which one of the devices connected to the OBUS is to be loaded. They are:

LDSP	; load the parameter stack pointer
WTSP	; write the parameter stack
LDRP	; load the return stack pointer
WTSP	; write the return stack
LDIP	; load the instruction pointer
WTRAM	; write to the location in RAM addressed by IP or Latch
LATCH	; load the Latch
OFFOD	; off condition. No loading occurs

The Input Bus Source

This field is also sent to a decoder which enables exactly one of the devices which drives the input bus (IBUS). They are:

RDSP	; read the parameter stack pointer
RDRP	; read the return stack pointer
RDIP	; read the instruction pointer
RDPL	; read the data field in the pipeline
OFFIS	; off condition. No device enabled
SP	; read the parameter stack
RP	; read the return stack
RDRAM	; read the RAM addressed by IP or Latch
RDCNTL	; read the data/control interface register

Input Bus Destination

This field is used only to cause the data/control interface register to be loaded. The symbolic name of this action is LD.IF.

RAM Address Selection Field

As shown in Figure 1, there are two registers which can be used to generate an address for RAM. They are the instruction pointer, IP, and another register, called Latch, which are selected by the names ADDIP and ADDLCH respectively.

Counter Control

The SP, RP, and IP can all be incremented and decremented in parallel with other device accesses. Unfortunately, we implemented only the post-fetch increments and decrements for the pointers associated with these files.

```
INCIP     ; increment the instruction pointer
DECIP     ; decrement the instruction pointer
INCSP     ; increment the parameter stack pointer
DECSP     ; decrement the parameter stack pointer
INCRP     ; increment the return stack pointer
DECRP     ; decrement the return stack pointer
INCIP.RP  ; increment the instruction and return stack pointers
INCSP.IP  ; increment the instruction and parameter stack pointers
INCSP.RP  ; increment the parameter stack and the return stack pointers
DECIP.SP  ; decrement the instruction and parameter stack pointers
```

Data Field

On the non-branch instructions, NOBRCH, this 16-bit field can be used to place constants onto the IBUS. For branch instructions, BRANCH, this field is broken into two subfields. A 2-bit field, called the Branch Condition Select Field, which controls the polarity of the condition code inputs to the Controller, has three options:

```
ALWAYS    ; branch always
IF.TRU    ; branch if condition code is true
IF.FLS    ; branch if condition code is false
```

and a 12-bit Next Address Field which is used as the control store address if a branch is to be taken.

Discrete Bit

This field is used only to clear the start bit in the S-100 interface.

A Microcode Example

The ability of RUFOR's architecture can be best illustrated with an example using the microcode assembler. In this example, the FORTH instruction + is shown in both RUFOR's microcode format, and in PDP-11 assembly code:

```
ADD: BRANCH   GR5,GR5,DA,ADDRS,LDB.F,,CJP,,,SP,,,INCSP,ALWAYS,NEXT,
ADD: ADD      (SP)+,(SP)
```

Both take one instruction to execute, but the difference lies in the fact that RUFOR's one instruction takes only one clock cycle vs. multiple memory cycles for the PDP-11 instruction.

In order to add the two top items of the parameter stack, the following must occur:

1) fetch ADD instruction
2) fetch item pointed to by SP, update SP
3) fetch item pointed to by SP, add to previously fetched item, write the result back to location pointed to by SP.

RUFOR accomplishes all three steps within one microcycle. In the above example, the top item of the parameter stack is located in GR5. Thus, RUFOR reads the top item of the stack by addressing port A with GR5, then adds that item to the second element of the stack by enabling the parameter stack onto the IBUS and applying the correct information to the function bits of the ALU. The result is then written back to the top of the stack by addressing port B with GR5, and selecting the load option with the destination control field set to LDB.F. The stack pointer is then updated after the add is completed by using a post-increment capability (SP is held in a up/down counter). The branch condition control field value, ALWAYS, assures control is then returned to the next instruction to be executed by branching to the routine NEXT within the same clock cycle.

Lessons Learned

The architecture of RUFOR is centered around efficient computation using the top two words on the stack. RUFOR is a word machine, and it does not do well with byte oriented instructions. The addition of multiplexors between memory and the busses would improve performance and make coding easier.

The S-100 interface is half duplex and is a bottleneck. Interrupts on both the host and RUFOR would improve that situation.

The addition of fast hardware necessary to do increments and decrements prior to fetches on both stack and memory pointers would save cycles on several critical primitives.

The latch which is used as an alternate pointer to memory should be upgraded to the level of the IP and used as the W register. This would allow NEXT to be done in two cycles instead of three.

The present implementation of RUFOR is noisy, with a hardware error occurring randomly about every 15 minutes. We will build future copies in a separate box and cable only the communications with the host system to an interface card in the host.

References

1. *Bipolar Microprocessor Logic and Interface Data Book*, Advanced Micro Devices, 1981, Sunnyvale, CA.
2. Duncan, Ray, Z80 FORTH Version 1.14, Laboratory Microsystems, Los Angeles, CA.
3. Dumse, Randy M., Rockwell FORTH Processor, *Proceedings of the 1982 FORML Conference*, pp. 3–12.
4. Dumse, Randy M., The R65F11 FORTH Chip, *FORTH Dimensions*, July/Aug. Vol. 5, #2, p. 25.
5. James, John, *PDP-11 Forth*, Forth Interest Group, San Carlos, CA.
6. Winkel, David, The FORTH Engine, *FORTH Dimensions*, Sept./Oct. 1981 Vol. 3, #3, pp. 78–79.

Manuscript received August 1984.

Mr. Russ Grewe is an undergraduate student of computer engineering at Wright State University. He currently works at Digital Technology.

Robert Dixon received his doctorate from Ohio State University in 1962. He has been teaching at Wright State since 1964. His current interests include real-time systems, and hardware and software support systems for natural language interfaces.