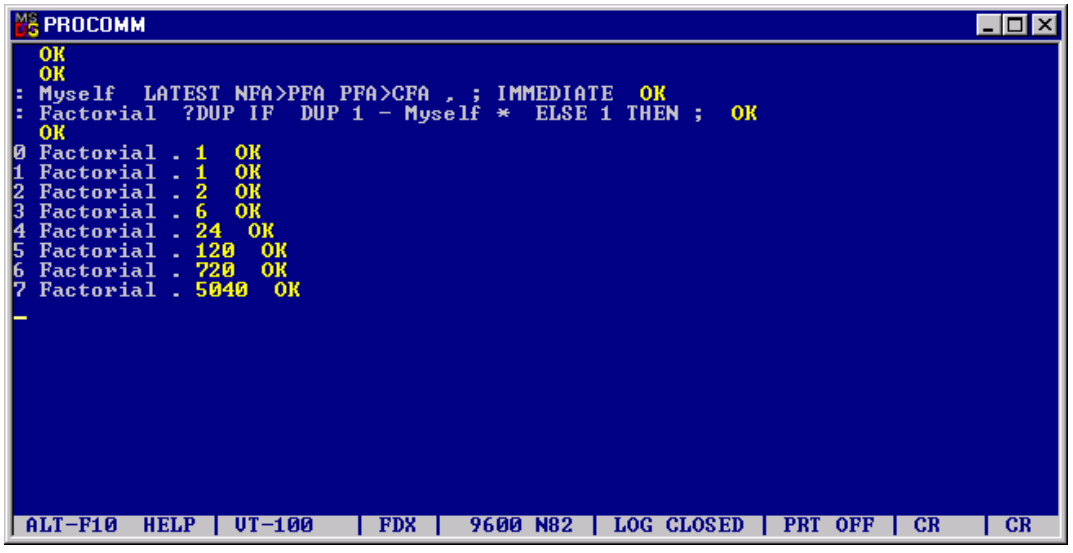
Mark 1 FORTH Computer

This computer has no microprocessor. The CPU is discrete TTL logic.

I bought my first TTL data book in 1979. I was learning 6502 machine code at the time and dreamt of building a simple TTL CPU. I sketched some circuit ideas; but that was as far as it went. Now, 25 years later, I've finally done it! Working evenings and weekends, the Mark 1 took a month to design, 4 months to build and a month to program. Here's the result:



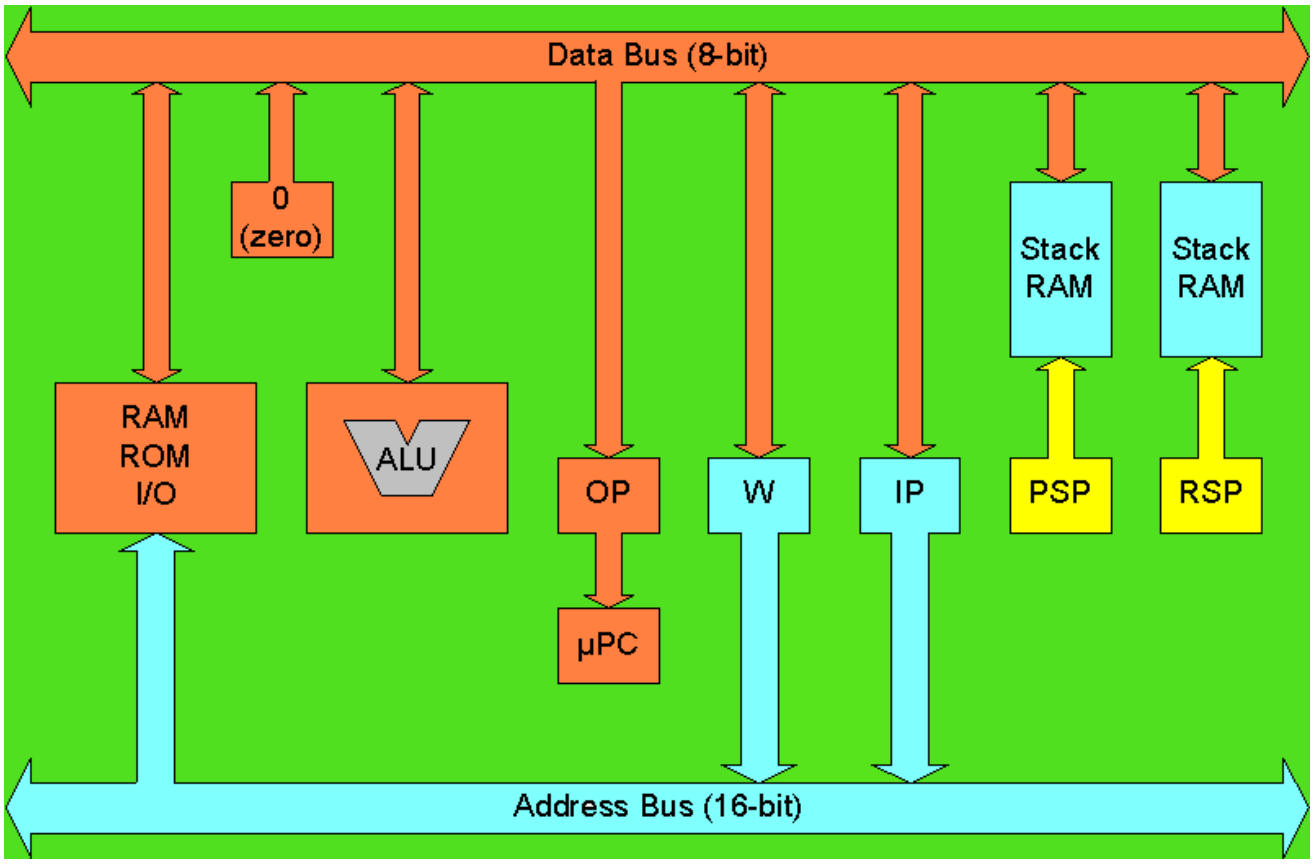
`Myself` is a standard way to implement recursion in FORTH. Even if you're not familiar with FORTH, if I tell you it's a stack-based language, and uses reverse polish notation (RPN), you might be able to figure out how this works.

Specification

Technology	TTL (HCMOS)

Clock	1 MHz
Data Bus	8-Bit
Address Bus	16-Bit
Software	fig-FORTH

System Overview



Module	Width (bits)	Description	Comments
ALU	8	Arithmetic and logic unit	The ALU data path is a bottleneck. It takes four clock cycles to load the inputs, set the ALU function, and read the result. This is the least satisfactory aspect of the whole design.
OP	8	Operand register	OP is loaded into the uppermost 8 bits of µPC. The lower 4 bits are reset to zero.
µPC	12	Microcode program counter	
W	16	FORTH Working register	The 16-bit index registers, IP and W, support increment, decrement, and can address memory.
IP	16	FORTH Instruction Pointer	
PSP	8	FORTH Parameter stack pointer	The stack pointers, RSP and PSP, are 8-bit up/down counters feeding the A1-A9 address inputs of the stack RAMs. The least significant address input (A0) selects the upper or lower byte. Logically, the stacks are 16-bits wide by 256 words deep. The FORTH word length is 16 bits.
RSP	8	FORTH Return stack pointer	
Stack RAM	16	Dedicated stack RAM	

0	8	Force 00H on data bus

µ-Instruction Format

The Mark 1 is a micro-programmed machine with a highly encoded "vertical" microcode. The microinstruction (µ) is only 8-bits wide. One normally thinks in terms of "horizontal" microcodes, which are wider and less encoded. Some are very wide indeed. The Mark 1 is more like a RISC processor.

The 8-bit µ-instruction (µ) is encoded as follows:

	µ7	µ6	µ5	µ4	µ3	µ2	µ1	µ0
Move LSB	0	0	Source			Destination		
Move MSB	0	1	Source			Destination		
Decrement	1	0	0	0	0	0	Register	
Disable IRQ	1	0	0	0	0	1	x	x
Increment	1	0	0	0	1	0	Register	
Enable IRQ	1	0	0	0	1	1	x	x
Jump Direct (zero page)	1	0	0	1	Address			
Set ALU function	1	0	1	0	Function			
Jump Indirect (µPC←OP*16)	1	0	1	1	x	x	x	x
Conditional skip	1	1	Test		Distance			

The source and destination fields of the move instructions are coded as follows:

	Destination	Source
000	W	W
001	IP	IP
010	TOS	TOS
011	R	R
100	Memory[W]	Memory[W]
101	OP	Memory[IP]
110	ALU input A	Zero
111	ALU input B	ALU output

TOS = Top of parameter stack; R = Top of return stack

The Mark 1 executes 1 µ-instruction per clock cycle (1MHz).

Decoding is done centrally using 74HC138 1-of-8 decoders. Decoded control signals are distributed via the back plane. Simple gating is then required at card level to complete the decoding.

The conditional is a skip not a branch. It inhibits loading of another µ-instruction for a specified number of cycles if the test is true. The skip distance is decremented to zero at which point normal execution resumes.

The "µPC←OP*16" instruction (1011xxxx) a.k.a. XOP loads the uppermost 8-bits of the program counter (µPC) from the operand register. It's a form of indirect jump.

The other jump instruction (1001xxxx) has a 4-bit operand and can only reach the first 16 bytes of the µ-ROM.

The 74181-based ALU requires 8 control signals. These are decoded from the 4-bit ALU function field in the µ-instruction using a 7x16 diode matrix ROM. The shaded squares indicate positions where diodes are fitted:

	OP	S0	S1	S2	S3	M	FLAG	D6	D7
0	ADD	1	0	0	1	L	x	H	H
1	ADC	1	0	0	1	L	x	x	L
2	SUB	0	1	1	0	L	x	L	H

3	SBB	0	1	1	0	L	H	x	L
4	ASL	0	0	1	1	L	x	H	H
5	ROL	0	0	1	1	L	x	x	L
6									
7									
8 (a)	0<	1	1	1	1	H	L	x	x
8 (b)	A	1	1	1	1	H	x	x	x
9	B	0	1	0	1	H	x	x	x
10	AND	1	1	0	1	H	x	x	x
11	OR	0	1	1	1	H	x	x	x
12	NOT	0	0	0	0	H	L	x	x
13	XOR	0	1	1	0	H	x	x	x
14	A=B	1	0	0	1	H	x	x	x
15									

The control signals are transmitted from the diode matrix to the ALU via the data bus.

M is hard-wired to µ3.

D6 and D7 control the carry input as follows:

D6	D7	Carry IN
X	L	Previous carry OUT
H	H	0
L	H	1

FLAG selects sign or overflow testing. Sign testing routes the most significant bit of the ALU result to the conditional test multiplexer. Overflow testing required the addition of a quad-XOR gate. The 74181 does not generate an overflow signal (the later 382 variant does). This was not used in the end because FORTH implements signed comparison by testing the sign of the result after subtraction.

Backplane

The Mark 1 is housed in a 3U 19" IEC297 sub-rack with a 64-way DIN 41612 backplane. The bus layout is shown below. The "A" row resembles a standard 8-bit microprocessor bus. The "C" row carries the µ-instruction and various decoded control signals. The fully-bussed pins (1, 2, 31, & 32) carry power supply and clocks.

A			**C**		
1	+5V				
2	CLK1				
3	D0	**DATA**	3	µ0	**µ**
4	D1		4	µ1	
5	D2		5	µ2	
6	D3		6	µ3	
7	D4		7	µ4	
8	D5		8	µ5	

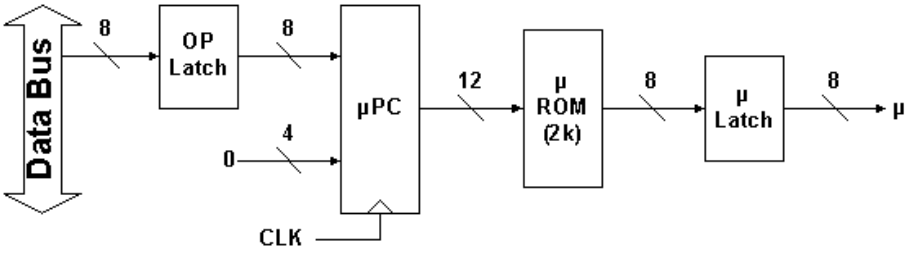
9	D6		9	µ6		
10	D7		10	µ7		
11	A0		11	µ210=101	Dest=OP	
12	A1		12	µ210=110	Dest=ALU A	
13	A2		13	µ210=111	Dest=ALU B	
14	A3		14	µ210=000	W	
15	A4		15	µ210=001	IP	
16	A5		16	µ210=010	Dest=TOS	PSP
17	A6		17	µ210=011	Dest=R	RSP
18	A7	**ADDRESS**	18	SRC=111	ALU	
19	A8		19	SRC=000	W	
20	A9		20	SRC=001	IP	
21	A10		21	SRC=010	TOS	
22	A11		22	SRC=011	R	
23	A12		23	µ=1000xxxx	INC / DEC	
24	A13		24	µ=1001xxxx	JUMP #	
25	A14		25	µ=1010xxxx	ALU Function	
26	A15		26	µ=1011xxxx	JUMP OP	
27	MR	Memory Read	27	M@IP	Address = IP	
28	MW	Memory Write	28	M@W	Address = W	
29	RESET		29	LO	LO-byte	
30	IRQ		30	HI	HI-byte	
31	CLK2					
32	0V					

The clocks are in quadrature. CLK1 ~~rises~~falls at the beginning of the machine cycle. CLK2 is used to generate write-enable signals for the RAM and I/O.

All control signals are active low.

Microcode sequencer

The sequencer has a 12-bit micro-program counter (µPC). The uppermost 8-bits can be loaded from the OP Latch, effecting a jump to one of 256 microcode routines. Each routine starts on a 16-byte µ-page boundary.



Opcodes, the machine language of the macro-machine, are loaded into the OP latch from the data bus under micro-program control. They can be fetched from memory using one of the index registers as a program counter. A simple *micro-interpreter* consists of the following 3 µ-instructions:

```
OP←Memory[Index]     Load OP latch from memory
Index←Index+1        Increment "program counter"
µPC←OP*16            Execute microcode routine
```

How do these 3 µ-instructions get executed? One possibility is to append them to the end of every µ-routine. A slower but more space-efficient option is to append a jump to them. Mark 1 microcode has a jump specifically for this.

Virtual machine

It's possible to customise the the instruction set and thereby create a "virtual machine".

Opcodes can have zero, one, or more operands. The µ-routines consume operands by incrementing the program counter. During development, I used this two-operand POKE instruction to test the UART. This expects a 16-bit address followed by a data byte:

```
Poke:    Index.Lo ← Memory [PC]        ; Address LO
         PC ← PC+1
         Index.Hi ← Memory [PC]        ; Address HI
         PC ← PC+1
         Temp ← Memory [PC]            ; Data byte
         PC ← PC+1
         Memory [Index] ← Temp         ; Do the POKE
         Jump Next
```

The Mark 1 was designed to support the FORTH virtual machine. The following FORTH primitives are micro-programmed:

```
EXIT LIT EXECUTE BRANCH 0BRANCH (LOOP) (DO) U* U/ AND OR XOR LEAVE R> >R R 0= 0< + D+ MINUS DMINUS OVER DROP SWAP DUP @ C@ ! C! (DOES)
```

Forth model

My original plan was to build a subroutine-threaded FORTH. High-level definitions were to be called explicitly, primitives were to be compiled inline:

```
: Foo DUP SWAP DROP ;                  Foo: DB OP_DUP, OP_SWAP, OP_DROP, OP_EXIT

: Bar OVER Foo ROT ;                   Bar: DB OP_OVER, OP_CALL
                                            DW Foo
                                            DB OP_ROT, OP_EXIT
```

I abandoned this idea because most FORTHs use indirect threading and I wanted a full-featured standard FORTH with all the usual compiler facilities. Many compiling words are tightly coupled to the indirectly threaded model.

Indirectly threaded code is a list of execution tokens. An execution token is a code field address. The code field is a pointer to machine code. This presented a problem on the mark 1 because it has separate macro and micro address spaces. What should go in the code field? My solution was to shorten it to 1 byte and store the opcode:

Subroutine-threaded	Indirectly-threaded	Mark 1
Foo: DB OP_DUP DB OP_SWAP DB OP_DROP DB OP_EXIT Bar: DB OP_OVER DB OP_CALL DW Foo DB OP_ROT DB OP_EXIT	cfa_Foo: DW Enter pfa_Foo: DW cfa_DUP DW cfa_SWAP DW cfa_DROP DW cfa_Exit cfa_Bar: DW Enter pfa_Bar: DW cfa_OVER DW cfa_Foo DW cfa_ROT DW cfa_Exit cfa_Exit: DW pfa_Exit pfa_Exit: .. code .. cfa_DUP: DW pfa_DUP pfa_DUP: .. code ..	cfa_Foo: DB OP_ENTER pfa_Foo: DW cfa_DUP DW cfa_SWAP DW cfa_DROP DW cfa_Exit cfa_Bar: DB OP_ENTER pfa_Bar: DW cfa_OVER DW cfa_Foo DW cfa_ROT DW cfa_Exit cfa_Exit: DB OP_EXIT cfa_DUP: DB OP_DUP cfa_SWAP: DB OP_SWAP

In Mark 1 assembly language, the FORTH *inner interpreter* (NEXT) looks like this:

```
NEXT:       mov w.l, [ip]       ; W ← XT, IP ← IP+2
            inc ip
            mov w.h, [ip]
            inc ip

            mov op, [w]         ; OP ← [CFA]
            inc w               ; W ← PFA
            xop                 ; µPC ← OP*16
```

It follows the convention of invoking primitives with the PFA in W as required by ENTER:

```
ENTER:      dec rsp             ; Push IP
            mov rs, ip
            mov ip, w           ; IP ← PFA
            jmp NEXT

EXIT:       mov ip, rs          ; Pop IP
            inc rsp
            jmp NEXT
```

Note: My microcode assembler expands 16-bit moves into a pair of 8-bit µ-instructions.

Notice how the last 3 µ-instructions in NEXT resemble the simple micro-interpreter described earlier. This was used to advantage in the implementation of multiplication and division.

Math primitives

The math primitives U* and U/ were split into 3 opcodes to optimise performance:

```
cfa_UMUL:    DB OP_MUL_BEGIN, 16 DUP(OP_MUL_BIT), OP_MUL_END
cfa_UDIV:    DB OP_DIV_BEGIN, 16 DUP(OP_DIV_BIT), OP_DIV_END
```

I use the Microsoft Assembler (MASM) to create ROM images for the macro memory space. The syntax "16 DUP()" tells MASM to repeat the enclosed byte 16 times. It's equivalent to:

```
cfa_Uxxx:    DB OP_XXX_BEGIN
pfa_Uxxx:    DB OP_XXX_BIT, OP_XXX_BIT, OP_XXX_BIT, OP_XXX_BIT
             DB OP_XXX_BIT, OP_XXX_BIT, OP_XXX_BIT, OP_XXX_BIT
             DB OP_XXX_BIT, OP_XXX_BIT, OP_XXX_BIT, OP_XXX_BIT
             DB OP_XXX_BIT, OP_XXX_BIT, OP_XXX_BIT, OP_XXX_BIT
             DB OP_XXX_END
```

First, NEXT calls _BEGIN with the address of the PFA (i.e. the first _BIT) in W. _BEGIN and _BIT end by jumping to the 3rd from last instruction in NEXT. This executes _BIT 16 times incrementing W as it goes. W acts as the loop counter or temporary program counter. Finally, _END jumps to the high-level NEXT.

Power-on reset

Reset forces µPC to 000H. The first 8 bytes of the µ-ROM contain the following:

```
RESET:       mov w, 0            ; W ← 0002h
             inc w
             inc w

             dis                 ; Disable IRQ

             mov ip.l, [w]       ; IP ← Cold start vector
             inc w
             mov ip.h, [w]

Next:        ...
```

This initialises the high-level instruction pointer (IP) from a cold start vector at location 0002h in main memory. It then drops through into NEXT.

The high-level ROM assembly begins like this:

```
.Model Tiny
.Code
             Include OPS.INC
             ORG 0

             DW 0FFFFh           ; Reserved for IRQ vector
             DW Reset            ; Cold-start vector

Reset        DW UART_Init
             ...
```

Offset 0000h is reserved for the interrupt vector.

Software development

I wrote a single-pass assembler for Mark 1 microcode. This generates Intel Hex images of the µ-ROM, and a list of opcodes formatted as MASM EQU statements. High-level ROMs are created using the Microsoft MASM assembler. The /TINY command-line switch forces linker version 6.15 to generate binary .COM files, which are then converted to Intel Hex.

Burning EPROMs soon became tedious and I wrote a ROM-resident monitor to accept Intel Hex downloads via the serial port. This is how FORTH was originally loaded; but the latest version is ROM-resident. I now have a PC-based simulator for debugging, FORTH is fairly stable, and there's less need for the monitor.

My original FORTH, posted here in 2003, reversed the stack order of quotients and remainders left by division words. This has been corrected. Here's the latest code:

Asm.cpp	µ Assembler
ROM.ASM ROM.HEX	µ-ROM
OPS.INC	MASM include file
Boot.ASM	Monitor
Forth.ASM Forth.HEX	fig-FORTH
Bin2Hex.cpp	Binary to Intel Hex conversion tool
build.bat	Build script
Sim.zip	Simulator

UPDATED
December 2006

This implementation of fig-FORTH is based on the original May 1979 Installation Manual for the 6502 by Bill Ragsdale. It deviates from the standard in the following ways:

- ANSI names for header navigation words e.g. PFA>CFA
- ?DUP instead of -DUP
- Vocabulary support omitted
- CFA is 1-byte wide

Most fig-compliant code should run with little or no alteration. See the examples (e.g. DOER-MAKE) in the simulator zip.

STOP PRESS - Mark 1 Cloned!

Aaron Tang, a student at the Universiti Teknologi Petronas in Malaysia, has built a Mark 1 FORTH Computer; and one of his classmates, Aidil Jazmi, has cloned Bill Buzbee's Magic-1. Visit their UTP Cloners page to read all about it. Aaron first contacted me in March 2006 with a few questions, and by October 2006 his computer was working. He used the same eurocards, mounted in a similar rack to mine, and followed my layout very closely; but, whereas I mostly used pen-wiring, Aaron's computer is wire-wrapped. Well done to both Aaron and Aidil.

More homemade computers

You'll find more homemade computers on my links page.
Please visit the other sites on the web ring (below) and don't forget to have a look at my Mark 2 FORTH Computer.

Homebuilt CPUs WebRing

Home | Previous | Next | Random

JavaScript by Qirien Dhaela

Join the ring?

David Brooks, designer of the Simplex-III homebrew computer, has founded the Homebuilt CPUs Web Ring. To join, drop Dave a line, mentioning your page's URL. You'll need to copy this code fragment into your page.

---

andrew@aholme.co.uk