P24 MISC Microprocessor User's Manual

Dr. C. H. Ting

Hsinchu, Taiwan, Republic of China

Updated by Walter Cook 1/23/01 to reflect adaptation
to the ACTEL 54SX72A.

Caltech document number STEREO-CIT-004.A

Carry bit documentation corrected 10/9/01 (WRC)

Chapter 1. INTRODUCTION

1.1 General Information

P24 is “Minimal Instruction Set Computer” design patterned after Mr. Chuck Moore's MuP21. P24 has a 24-bit CPU core with dual stack architecture intended to efficiently execute Forth-like instructions. The processor design is simple to allow implementation within field programmable gate arrays.

P24 employs a RISC-like instruction set with four 6-bit instructions packed into 24 bit words. The most significant bit of each instruction designates an I/O Buss operation when set. For I/O Buss instructions (the I/O buss will be referred to as the G buss) the second most significant bit specifies a write when set, read when cleared, while the remaining four bits specify the G buss address. For non-G buss instructions the most significant bit is cleared and the remaining 5 bits specify 32 possible instructions, 31 of which are implemented.

Following is a list of unique features of P24:

* 24-bit address and data buses
* 6-bit RISC-like CPU instructions
* 4-deep instruction cache
* 17-deep data stack
* 33-deep return stack
* Uses about 75% of 54SX72A registers and logic modules
* Current implementation runs at 10 MHz

1.2 Architecture of P24 CPU

P24 has the following registers:

A Address Register, supplying address for memory read and write
I Instruction Latch, holding instructions to be executed
P Program Counter, pointing to the next program word in memory
R Top of Return Stack
S Top of Data stack
T Accumulator for ALU

All registers are 24 bit wide.

(Original text. Applies to Dr. Ting's VHDL model, but not to the current Actel implementation: T, R, S, and A are all 25 bits wide. The most significant bit in T, T(24) is the carry produced by the 24-bit adder. This carry bit is preserved as data in T is transferred to other registers and to the stacks. The preservation of carry bit greatly simplifies the logic processing of data, and allows interrupts to be services when the next program word is fetches from the memory, without having to save the carry bit and restore it on return.)

Unlike the original specification by Dr. Ting, the current ACTEL implementation Does not provide the extra bit width needed to preserve the carry bit. The carry bit is rather held in a dedicated flip-flop and is preserved only until the next operation affecting the carry bit.

P24 has two stacks:

S_stack Data stack, 17 levels deep
R_stack Return stack, 33 levels deep

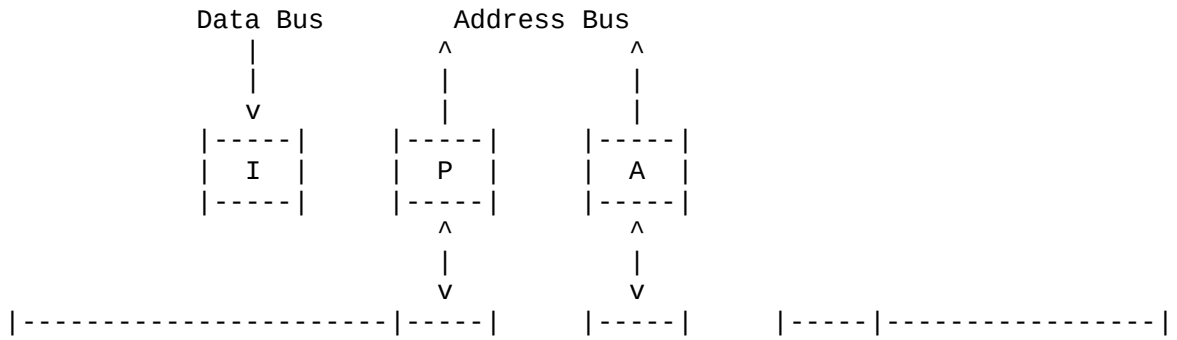
The return stack is used to preserve return addresses on subroutine calls. The data stack is used to pass parameters among the nested subroutine calls. With these two stacks in the CPU hardware, P24 is optimized to support the Forth programming language.

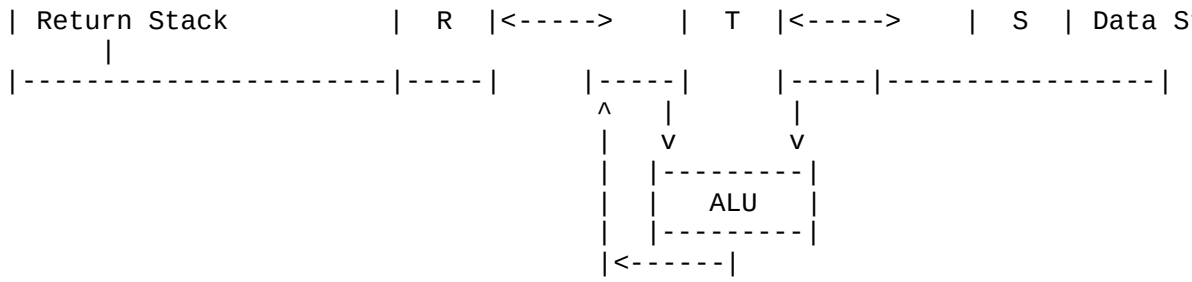
The 24-bit P24 CPU sports a small, RISC-like instruction set. Four 6-bit instructions are packed into one 24-bit word, and are executed consecutively after a word is fetched from memory. The P24 CPU has a two stack architecture that is easily programmed in Forth. The data stack is 17 levels deep (including T), and the return stack is 33 levels depth.

The following diagram shows the architecture of the P24 processor. It shows the registers, the stacks, and the data paths among them.

Not shown in the diagram is the connection between T register and the external data bus. When reading data from memory, the A register supplies the memory address to the address bus, and data is latch from the data bus into the T register. When writing data into memory, the address is supplied by A register, and data is written to the data bus from the T register.

Figure 1. The architecture of P24





1.3 Functional Block Diagram of P24 (This section applies to Dr. Ting's VHDL definition and has not been updated to correspond to the ACTEL implementation which is somewhat different.)

These data path diagrams should be read with the CPU24.VHD file.

The instruction decoding logic simply apply the proper control signals to the following register loading and multiplexer selecting signals:

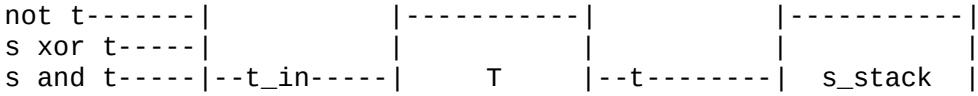
Clr	Master reset
Clk	Master clock, 0-40 MHz
t_sel	Select input to T register
tload	If set, load t_in into T register
spop	If set, pop the data stack
spush	If set, push T on the data stack
a_sel	Select input to A register
aload	If set, load a_in inot A register
r_sel	Select input to R register
rload	If set, load r_in into R register
rpop	If set. Pop the return stack
rpush	If set, push R on the return stack
p_sel	Select input to P register
pload	If set, load P_in into P register
m_sel	Select output to Address bus
iload	If set, load instruction from data bus to I register
reset	Clear the machine instruction counter
slot	Output of machine instruction counter to select instruction

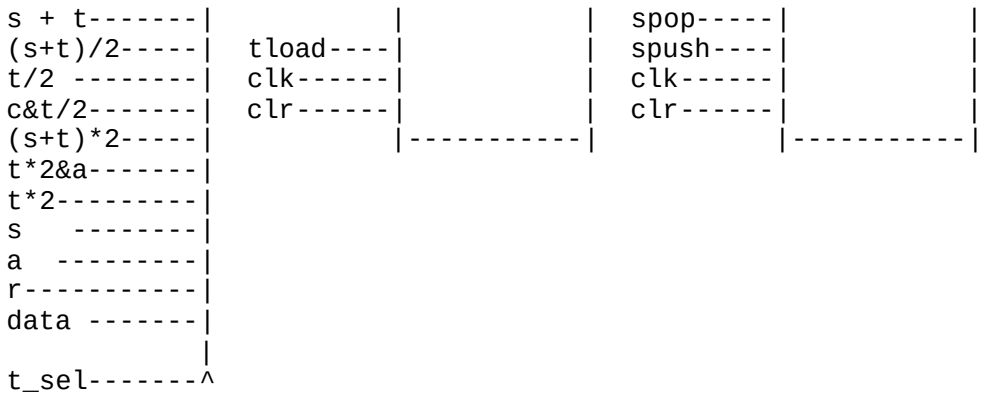
The synchronous program execution unit clocks the slot signal, which selects the proper 6-bit instructions in the I register to produce the above control signals. At the rising clock edge, the selected data are latched into the proper register and stacks. All data signals must stabilize before the next rising clock edge strikes.

The architecture is very simple and components are very similar to one another. It should be very easy to do a good layout, and the routing should not be difficult.

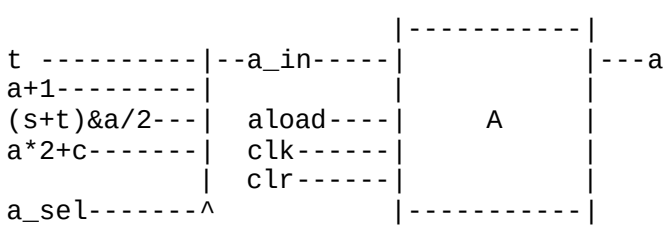
Figure 2. The block diagrams of P24 components

The T and Data Stack Data Path

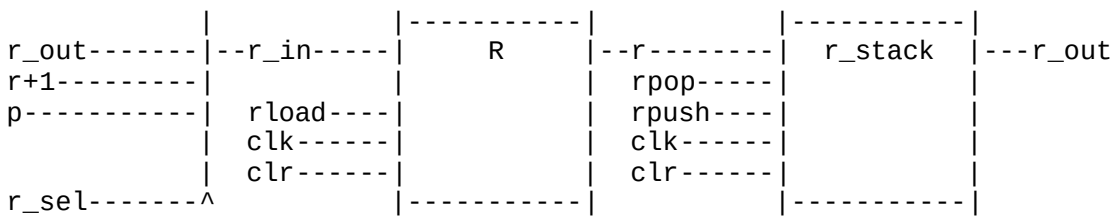




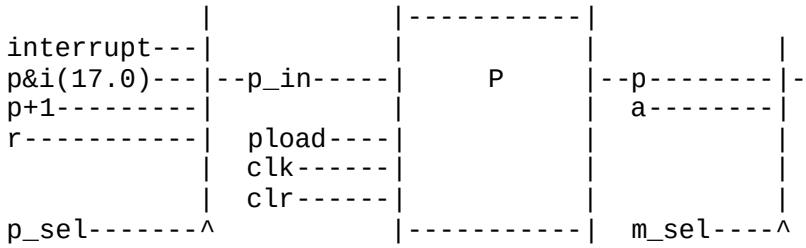
The A Register and A-Mux



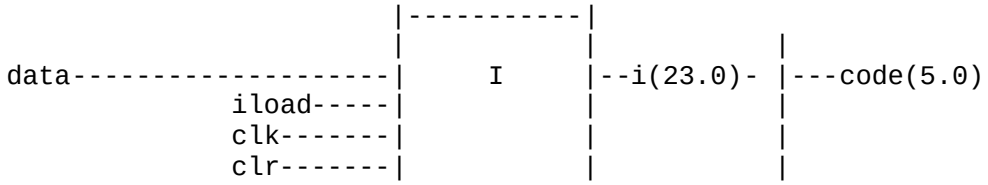
The Return Stack Data Path

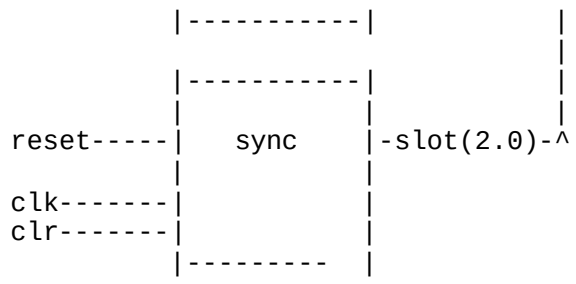


The Program Counter Data Path



The Instruction Latch and Decoder Data Path





On power-up, all registers and the stacks are cleared to zero when "clr" is held high. When "clr" is lowered to zero, the master clock "clk" will start the CPU from memory location 0, as the initialized P register is pointing to.

Chapter 2. Device Characteristics

2.1 Input and Output Signals

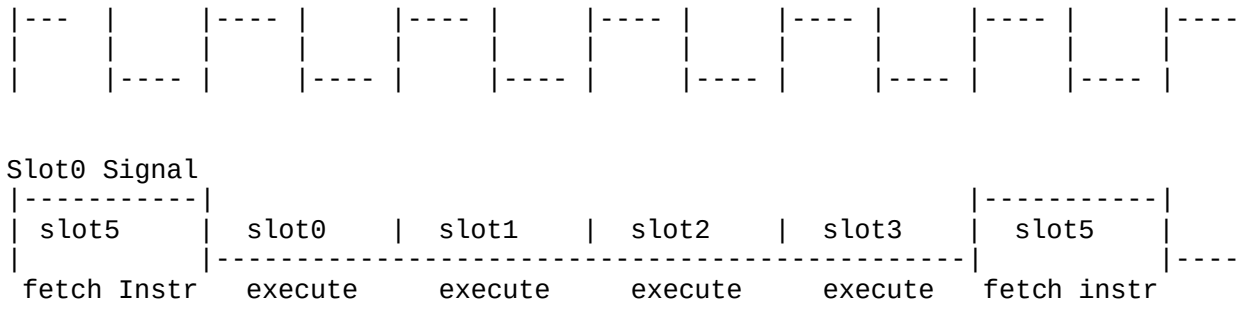
P24 is very flexible in packaging, depending on the memory configuration. These are the signals normally brought to I/O pins. In certain applications, the memory is included on chip and the address bus and data bus do not have to be brought out.

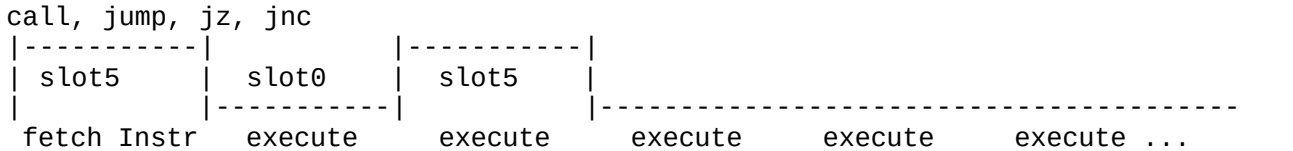
CLK	1-40 MHz master clock
A0-23	Address bus to RAM, SRAM and I/O devices
D0-23	Data bus for RAM, SRAM and I/O devices
CLR	Low system reset (active low)
Vdd	5V power supply
Vss	Ground
WE	Write enable (active low)
INT0-4	External interrupt inputs
UART_IN	RS232 serial input pin
UART_OUT	RS232 serial output pin

2.2 Timing

All time periods noted in the following timing diagrams are in periods of the master clock.

Figure 3. Timing of P24 instruction executions





NOP and RET instructions can be in any of the four slots. When these two instructions are executed, slot5 will be forced into the next slot, and the next instruction words will be fetched and then executed.

The ACTEL implementation contains a prioritized interrupt controller. If an interrupt is pending an extra slot, slot4, is added to the sequence following slot3. During slot4 the program counter is pushed to return stack and the interrupt vector is placed in the program counter. Currently 7 interrupt lines, labeled int0 – int6 are implemented and only two are used. Int0 vectors to address 1, int2 to address 2, etc. The highest priority int0 is currently assigned to the UART receive function, while the next highest priority int1 is assigned to the UART transmit function. Once an interrupt is serviced via execution of slot4, servicing of interrupts is automatically disabled until the execution of an RTI instruction. Immediately after the RTI execution, the highest priority pending interrupt (if any) will be serviced.

When executing a right shift instruction SHR, the sign bit T(23) is preserved. Bits T(23..1) are shifted to the right by one bit. Bit T(0) is latched onto the UART_OUT pin, and UART_IN pin is latched into the carry bit T(24). This very simple mechanism allows a simple RS232 serial port to be built in P24 core. As the serial port is the only peripheral device required by eForth, this simple serial port opens a window for the user to access the resources provided by P24, and supports a powerful embedded Forth system to control and to program the P24 system. (This simple UART was very valuable in bringing up the ACTEL implementation, but is no longer used.)

Chapter 3. P24 Instruction Set

3.1 Instructions

The P24 instruction set can be best explained using the register and data flow diagram as shown in Figures 1 and 2. The T register is the center of the ALU, which takes data from the T and S registers and routes the results back to the T register. The contents of T can be moved to the A register, pushed on the data stack S, and pushed on the return stack S.

The T register connects the data stack and the return stack as a large shift register. Data can be shifted towards the return stack by the PUSH instruction, and shifted towards the data stack by the POP instruction.

Register A holds a memory address, which is used to read data from memory into the T register, or write the data in T register to external memory. The address in A can be auto incremented, so that P24 can conveniently access data arrays in memory.

P is the program counter and it holds the address of the next instruction to be fetched from the memory. After an instruction is fetched, P is auto incremented and ready to read the next instruction. When a CALL instruction is executed, the address in P is pushed on the return stack. When a return (RET) instructions is executed, the previously saved address in R is popped back into P. The execution sequence interrupted by CALL can now be resumed.

P24 is a microprocessor with 24-bit instructions. Each instruction contains up to 4 6-bit machine codes. The instruction fields in a program word can be shown as follows:

```
Bits: 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

     | Instruction 1    | Instruction 2   | Instruction 3   | Instruction 4   |
```

There are 64 possible instructions in a 6-bit field. Half of these are used for G-Buss access, and are specified by a one in the most significant bit of the six-bit field. For G-Buss access instructions the next most significant bit specifies write if set, read if cleared, while the remaining 4 bits are the G-Buss address. The G-Buss is intended to provide fast access to on-chip application specific functions such a timers, i/o registers, UART, general purpose registers, etc. The non-G-Buss instructions are of four classes:

```
0     Transfer Instructions
1     Memory Access Instructions
2     ALU Instructions
3     Register Instructions
```

JUMP, CALL, JZ and JNC instructions must appear in Slot0 of a program word, ie. bits 23-18. The last 18 bits 17-0 contain the address inside the current 256K word page. They can access code within the current page. To reach other pages of memory, you will have to push a 24-bit address on the return stack and execute the RET instruction.

The transfer instructions thus has the following forms:

```
      JUMP  aaaaaa aaaaaa aaaaaa
      CALL  aaaaaa aaaaaa aaaaaa
      JZ    aaaaaa aaaaaa aaaaaa
      JNC   aaaaaa aaaaaa aaaaaa
```

The conditional jump instruction JZ is used to implement the IF, WHILE, and UNTIL words in Forth in that it does pop the number being tested in T. The conditional jump instruction JNC causes a jump if the carry bit T(24) is cleared. It is useful in multiple precision math operations. JNC does not pop the T register, so its contents can be tested again.

Table 1. P24 Machine Code

Code	Name	Function
Transfer Instructions		
00	JUMP	Jump to 18 bit address. Must in Slot0.
01	RET	Subroutine return.

```
02    JZ          Jump if T is 0.  Must in Slot0.
03    JNC         Jump if carry is reset.  Must in Slot0.
04    CALL        Call subroutine.  Must in Slot0.
05    NEXT        Jump if R is not 0. Post-decrement R.
                  Pops R if R is 0. Must be in Slot0.
06    TIMES       Repeat instruction word if R is not 0. Post-decrement R.
                  Pops R if R is 0.
07    RTI         Return from interrupt

Memory Access Instructions
09    LDP         Push memory at A to T.  Increment A.
0A    LDI         Push in-line literal to T.
0B    LD          Push memory at A to T.
0D    STP         Pop T to memory at A.  Increment A.
0F    ST          Pop T to memory at A.

ALU Instructions
08    RR8         Rotate right T by 8 bits.
0C    NIP         Pop S, (Equivalent to SWAP DROP)
10    COM         Complement all bits in T.
11    SHL         Shift T left 1 bit.
12    SHR         Shift T right 1 bit.
13    MUL         Multiplication step.
14    XOR         Pop S and Exclusive OR it to T.
15    AND         Pop S and AND it to T.
16    DIV         Division step.
17    ADD         Pop S and add it to T.

Register Instructions
18    POP         Pop R to push T.
19    LDA         Push A to T.
1A    DUP         Duplicate T.
1B    OVER        S to T, push original T.
1C    PUSH        Pop T to push R.
1D    STA         Pop T to A.
1E    NOP         Do nothing.
1F    DROP        Pop T.

0E                Reserved
```

3.2 P24 Instructions

JUMP (SKIP, ELSE, AGAIN, REPEAT)

Code: 0

Usage: 00000 aaaaaa aaaaaa aaaaaa

Stack Effects: none
Carry: no change

Function:

Jump to the 18 bit address in the bit field 17-0 in the current 256K word page of memory. It must be in slot 0 of a word.

Restriction:

This instruction allows the program to be redirected to any location within an 256K word page of memory. It does not cross page boundaries. To jump to locations outside of a memory page, one has to push the target address on the return stack and execute the RET instruction to effect a long jump. This restriction also applies to CALL, JZ and JNC. See also RET.

Coding Example:

```
CODE 50us
        2 ldi skip
CODE 100us
        1 ldi
        then
        sta -138 ldi
        begin lda add
        -until
        drop
        ret
```

SKIP makes an unconditional jump to THEN, to let 50us sharing the delay loop with 100us.

RET (;)

Code: 1

Usage:
```
000001 xxxxxx xxxxxx xxxxxx
cccccc 000001 xxxxxx xxxxxx
cccccc cccccc 000001 xxxxxx
cccccc cccccc cccccc 000001
```

Stack Effects: (-- ; R: a --)
Carry: no change

Function:

Pop the address of the top of the return stack into the program counter P, thus resume the execution sequence interrupted by the last CALL instruction. Besides terminating a subroutine, this instruction may be used to effect a long jump to a location outside of the current memory page.

This instruction can be placed in any slot of a word. The instructions before return are executed. The instructions following return are ignored.

Coding Example:

In the subroutine thread model, RET is used to terminate all code words and colon words. The Forth word ; simply compiles a RET to end a Forth word.

JZ (IF, WHILE, UNTIL)

Code: 2

Usage: 000010 aaaaaa aaaaaa aaaaaa

Stack Effects: (n --)
Carry: no change

Function:

Conditionally jump to the 18 bit address in the bit field 17-0 in the current 256K word page of memory, if the T register contains a 0. It must be in slot 0 of a word.

The T register is destroyed and the data stack is popped back to T. This instruction is different from JNC, which does not pop the data stack and removes T.

Coding Example:

```
CODE ?DUP ( w -- w w | 0 )
   dup
   if dup ret then
   ret
```

JNC (-UNTIL, -IF, -WHILE)

Code: 3

Usage: 000011 aaaaaa aaaaaa aaaaaa

Stack Effects: (n -- n)
Carry: no change

Function:

Conditionally jump to the 18 bit address in the bit field 17-0 in the current 256K word page of memory, if the Carry flag (Bit 24 of T) is reset. It must be in slot 0 of a word.

The T register and the data stack are preserved. This instruction is different from the instructions JZ, which pop the data stack and removes T.

Coding Example:

To test the negative flag T(23), it is shifted into carry T(24) and tested using JNC compiled by -IF.

```
CODE ABS ( n -- +n )
   dup shl
   -if drop com 1 ldi add
       ret
   then
```

```
   drop ret
```

CALL

Code: 4

Usage: 000100 aaaaaa aaaaaa aaaaaa

Stack Effects: (-- ; R: -- a)
Carry: no change

Function:

Call a subroutine whose address is in the bit field 17-0 in the current 256K word page of memory. It must be in slot 0 of a word

The address of the next word is pushed on the return stack. When a return instruction in the subroutine is encountered, this address is popped off the return stack and the next word is executed to resume the interrupted execution sequence.

Restriction:

This instruction allows the program to call to any subroutine within the current 256K page of memory. It does not cross page boundaries.

Coding Example:

All Forth words are compiled as subroutine calls. This is the most efficient way to build lists in Forth.

NEXT

Code: 5

Usage: 000101 aaaaaa aaaaaa aaaaaa

Stack Effects: (--)
Carry: no change
Function:

If R is non-zero jump to the 18 bit address in the bit field 17-0 in the current 256K word page of memory and post-decrement R. If R is zero, pop R. It must be in slot 0 of a word.

Coding Example:

```
: FUN ( n -- ) FOR R@ . NEXT ;  ( prints the numbers 0 through n in reverse order )
```

TIMES

Code: 6

Usage:
```
000110 cccccc cccccc cccccc
cccccc 000110 cccccc cccccc
cccccc cccccc 000110 cccccc
cccccc cccccc cccccc 000110
```

Stack Effects: (--)
Carry: no change
Function:

If R is non-zero jump back to the beginning of the current instruction word and post-decrement R. If R is zero, pop R.

Coding Example:

```
CODE LSHIFT ( n1 n2 –- n1*2^n2 )
      zero com add push       ( subtract one from n2 and push to R )
      shl times ret           ( shift n1 left n2 times, then return )
```

RTI

Code: 7

Usage:
```
000111 xxxxxx xxxxxx xxxxxx
cccccc 000111 xxxxxx xxxxxx
cccccc cccccc 000111 xxxxxx
cccccc cccccc cccccc 000111
```

Stack Effects: (-- ; R: a --)
Carry: no change

Function:

Pop the address of the top of the return stack into the program counter P, to resume execution at completion of an interrupt service routine. Re-enables slot4 interrupt servicing.

This instruction can be placed in any slot of a word. The instructions before RTI are executed. The instructions following return are ignored.

Coding Example:

```
: RCV 1 ASSIGN    ( uart receive interrupt service routine, see multi3 file )
      lda –IF –1 ELSE 0 THEN              ( save A and carry on data stk )
      G1@ RCVFULL? IF DROP ELSE RCV! THEN ( service interrupt )
      WAKE OPERATOR !                     ( service interrupt )
      shl drop sta rti ;                  ( restore carry and A, then rti )
```

LDP

Code: 9

Usage:
```
001001 cccccc cccccc cccccc
cccccc 001001 cccccc cccccc
cccccc cccccc 001001 cccccc
cccccc cccccc cccccc 001001
```

Stack Effects: (-- n)
Carry: no change

Function:

Fetch the contents of a memory location whose 24-bit address is in the A register and push that number onto the data stack. The address in the A register is then incremented to facilitate accessing the next memory. It is most useful in reading values from a table in the memory.

This fetch instruction is different from the @ instruction in Forth, which uses the address on the top of the data stack.

This instruction also resets the carry flag (Bit 24) in the T register.

Coding Example:

Increment T
```
sta ldp drop lda
```

Otherwise,
```
cccccc cccccc ldi add
000000 000000 000000 000001
```
costs 6 slots.

LDI

Code: 0A

Usage:
```
001010 cccccc cccccc cccccc
nnnnnn nnnnnn nnnnnn nnnnnn

cccccc 001010 cccccc cccccc
nnnnnn nnnnnn nnnnnn nnnnnn

cccccc cccccc 001010 cccccc
nnnnnn nnnnnn nnnnnn nnnnnn

cccccc cccccc cccccc 001010
nnnnnn nnnnnn nnnnnn nnnnnn
```

Stack Effects: (-- n)
Carry: no change

Function:

Fetch the contents of the next word and push that number onto the data stack. The program counter PC is incremented passing the next word. This instruction allows a program to enter numbers onto the data stack for later use.

This instruction also resets the carry flag (Bit 24) in the T register.

Coding Example:

Push 1 2 3 4 on data stack:

```
Ldi ldi ldi ldi
1
2
3
4
```

LD

Code: 0B

Usage:
```
001011 cccccc cccccc cccccc
cccccc 001011 cccccc cccccc
cccccc cccccc 001011 cccccc
cccccc cccccc cccccc 001011
```

Stack Effects: (-- n)
Carry: no change

Function:

Fetch the contents of a memory location whose 24-bit address is in the A register and push that number onto the data stack. The address in the A register is not modified.

This fetch instruction is different from the @ instruction in Forth, which uses the address on the top of the data stack.

This instruction also resets the carry flag (Bit 24) in the T register.

Coding Example:

STP

Code: 0D

Usage:
```
001101 cccccc cccccc cccccc
cccccc 001101 cccccc cccccc
cccccc cccccc 001101 cccccc
cccccc cccccc cccccc 001101
```

Stack Effects: (n --)
Carry: no change

Function:

Pop the number off the data stack and store it into the memory location whose 24-bit address is in Register A. The address in the A register is then incremented to facilitate the next memory access. It is most useful in storing values to a table in the memory.

This store instruction is different from the ! instruction in Forth, which uses the address on the top of the data stack.

Coding Example:

See the copying program shown in LDP.

ST

Code: 0F

Usage:
```
001111 cccccc cccccc cccccc
cccccc 001111 cccccc cccccc
cccccc cccccc 001111 cccccc
cccccc cccccc cccccc 001111
```

Stack Effects: (n --)
Carry: no change

Function:

Pop the number off the data stack and store it into the memory location whose 24-bit address is in Register A. The address in the A register is not modified.

This store instruction is different from the ! instruction in Forth, which uses the address on the top of the data stack.

Coding Example:

```
CODE ! ( n a -- )
      sta st ret
```

RR8

Code: 08

Usage:
```
001000 cccccc cccccc cccccc
cccccc 001100 cccccc cccccc
cccccc cccccc 001100 cccccc
cccccc cccccc cccccc 001000
```

Stack Effects: (n1 – n2)
Carry: no change

Function:

All 24 bits in the T register are rotated right by 8 bits. The least significant byte of T moves to the position of the most significant byte. Useful for fast accessing of byte data and data formatting/packing.

Coding Example:

```
: BYTE# ( n1 n2 –- n3 )       ( n3 is byte number n2 of n1, for  )
      ?DUP                    ( n2 expected equal 0, 1 or 2 )
      IF    zero com add push
            rr8 times
      THEN
      FF and ;
```

NIP

Code: 0C

Usage:
```
001100 cccccc cccccc cccccc
cccccc 001100 cccccc cccccc
cccccc cccccc 001100 cccccc
cccccc cccccc cccccc 001100
```

Stack Effects: (n1 n2 -- n2)
Carry: no change

Function:

Pop S, leaving T unchanged.

```
Coding Example:               ( You hopefully never need this one. )
                              ( A good candidate for replacement with )
                              ( something more useful. Suggest not using. )
```

COM

Code: 10

Usage:
```
010000 cccccc cccccc cccccc
cccccc 010000 cccccc cccccc
cccccc cccccc 010000 cccccc
cccccc cccccc cccccc 010000
```

Stack Effects: (n1 – n1*)
Carry: no change

Function:

Complement all 24 bits in the T register. This is a one's complement operation.

Coding Example:

To generate a -1 in T register:

```
                        zero com
```

OR has to be synthesized from COM, and AND using:
A or B = not(not(A) and not(B))

```
CODE OR ( n n - n )           ( this looks pretty awkward, maybe )
```

```
   com push com              ( the last available opcode or NIP )
   pop and com ret           ( should be replaced with OR )
```

SHL

Code: 11

Usage:
```
010001 cccccc cccccc cccccc
cccccc 010001 cccccc cccccc
cccccc cccccc 010001 cccccc
cccccc cccccc cccccc 010001
```

Stack Effects: (n -- 2n)
Carry: Bit 23 of T is shifted into carry

Function:

Shift all lower 24 bits in the T register to the left by 1 bit. The lowest Bit-0 is cleared.

Coding Example:

Multiply T by 3: dup shl add
Multiply by 5: dup shl shl add
Multiply by 6: dup shl add shl

SHL allows the negative bit of T(23) to be tested as carry T(24):

```
CODE 0< ( n - f )
   shl
   -if drop -1 ldi ret
   then
   dup xor ( 0 ldi )
   ret
```

SHR

Code: 12

Usage:
```
010010 cccccc cccccc cccccc
cccccc 010010 cccccc cccccc
cccccc cccccc 010010 cccccc
cccccc cccccc cccccc 010010
```

Stack Effects: (n -- n/2)
Carry: no change

Function:

Shift the contents of the T register right by one bit. Bit-0 is shifted to the bit-banged UART serial output. The sign (Bit23) is preserved.

Coding Example:

SHR is used to implement a simple UART. The lowest bit in T, T(0) is shifted out to the UART serial output pin, and the UART serial input pin is loaded into carry for testing.

```
CODE EMIT ( c -- )
        $7F ldi and
        shl $FFFF01 ldi xor
        $0A ldi
        FOR shr 100us NEXT
        drop ret
CODE KEY ( -- c )
        $FFFFFF ldi
        begin   shr
        -until
        repeat   ( wait for start bit )
        50us
        7 ldi
        FOR
          100us shr
          -if $80 ldi xor then
        NEXT
        $FF ldi and
        100us ret
```

MUL

Code: 13

Usage: 010011 cccccc cccccc cccccc
cccccc 010011 cccccc cccccc
cccccc cccccc 010011 cccccc
cccccc cccccc cccccc 010011

Stack Effects: (n1 n2 -- n1 n3)
Carry: unchanged

Function:

Conditionally add the S register on the data stack to the T register if Bit-0 in A is set. If Bit-0 in A is reset, T register is not modified. The T-A register pair is now shifted to the right by one bit.

This MUL instruction is useful as a multiplication step in implementing a fast software multiplication routine. Repeating this instruction 24 times will multiply A and S and produce a 48-bit product in the T-A pair. (T is normally initialized to zero prior to the multiply sequence. However any non-zero initial value in T adds to the final result in the T-A pair.)

Coding Example:

Multiply two 24-bit unsigned integers. Multiplicand is in S. Multiplier is in A.

```
                        mul mul mul mul
                        mul mul mul mul
```

```
                        mul mul mul mul
                        mul mul mul mul
                        mul mul mul mul
                        mul mul mul mul
```

The 48-bit product is in T-A register pair and the multiplicand in S is preserved.

Primitive multiplication routines are thus defined:

```
CODE UM* ( u u -- ud )
   sta 0 ldi
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   push drop lda pop
   ret
```

XOR

```
Code:                   14

Usage:                  010100 cccccc cccccc cccccc
                        cccccc 010100 cccccc cccccc
                        cccccc cccccc 010100 cccccc
                        cccccc cccccc cccccc 010100

Stack Effects:          ( n1 n2 -- n3 )
Carry:                  unchanged
```

Function:

Pop S on the data stack and exclusive-OR it to the T register. All 24 bits in T are affected.

Coding Example:

To clear T to zero:

```
                        dup xor     ( now use more transparent “drop zero” )
```

To generate a zero in T register:

```
                        dup dup xor ( now use faster “zero” )
```

T is duplicated twice to save its contents. The two duplicated copies of T are XOR'ed together. All the reset bits remained reset. All set bits get reset. Thus a 0 is created in T.

It costs 5 slots to produce a -1:

```
                        Ldi cccccc cccccc cccccc
                        -1
```

vs

```
                        dup dup xor com   ( now use faster “zero com” )
```

AND

Code: 15

Usage:

```
010101 cccccc cccccc cccccc
cccccc 010101 cccccc cccccc
cccccc cccccc 010101 cccccc
cccccc cccccc cccccc 010101
```

Stack Effects: (n1 n2 -- n3)
Carry: unchanged

Function:

Pop S on the data stack and AND it to the T register. All 24 bits in T are affected.

Coding Example:

DIV

Code: 16

Usage:

```
010110 cccccc cccccc cccccc
cccccc 010110 cccccc cccccc
cccccc cccccc 010110 cccccc
cccccc cccccc cccccc 010110
```

Stack Effects: (n1 n2 -- n1 n3)
Carry: unchanged

Function:

Add the S register on the data stack to the T register. If the addition produces a carry place the sum in T, otherwise leave T unchanged. The T-A register pair is now shifted to the left by one bit. Carry is shifted into A(0).

This DIV instruction is useful as a division step in implementing a fast software division routine. Repeating this instruction 25 times will divide a 48 bit number originally in the T-A register pair by the negative of the number in S, leaving the result in A and remainder in T.

Coding Example:

Divide a 48-bit positive integer by a positive divisor. The negated divisor is in S.

```
                        div div div div
                        div div div div
                        div div div div
                        div div div div
```

```
                        div div div div
                        div div div div
                        div shr
```

(Note: I think that this last shr undoes the most recent shl that is part of div, aligning the remainder properly in T. Also I think this division actually only works properly for 47 bit unsigned numbers in T-A. -- WRC)

Primitive division routines are thus defined:

```
CODE UM/MOD ( ud u -- ur uq )
   com 1 ldi add sta
   push lda push sta
   pop pop
   skip
CODE /MOD ( n n -- r q )
   com 1 ldi add push
   sta pop 0 ldi
   then
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div 1 ldi xor shr
   push drop pop lda
   ret
```

ADD

```
Code:                   17

Usage:                  010111 cccccc cccccc cccccc
                        cccccc 010111 cccccc cccccc
                        cccccc cccccc 010111 cccccc
                        cccccc cccccc cccccc 010111

Stack Effects:          ( n1 n2 -- n1+n2 )
Carry:                  change according to n1 and n2
```

Function:

Pop S on the data stack and add it to the T register.

Coding Example:

The primitive addition in eForth is thus defined:

```
CODE UM+  ( n n - n carry )        ( don’t use this if you want speed – WRC )
   add
   -if 1 ldi ret
   then
   dup dup xor ( 0 )
   ret
```

POP

Code: 18

Usage: 011000 cccccc cccccc cccccc
cccccc 011000 cccccc cccccc
cccccc cccccc 011000 cccccc
cccccc cccccc cccccc 011000

Stack Effects: (-- n ; R: n --)
Carry: unchanged

Function:

Pop the R register on the return stack to the T register. Original contents in T are pushed on the data stack.

Coding Example:

```
Exchanging A and T      lda push sta pop
Exchanging A and R      lda pop sta push
Increment T             sta ldp drop lda        ( now use “one add” )
Decrement T             dup dup xor com add     ( now use “zero com add” )
```

LDA

Code: 19

Usage: 011001 cccccc cccccc cccccc
cccccc 011001 cccccc cccccc
cccccc cccccc 011001 cccccc
cccccc cccccc cccccc 011001

Stack Effects: (-- a)
Carry: unchanged

Function:

Copy the contents in the A register to the T register. The original content of the T register is pushed on the data stack. With LDA and STA, the A register can serve as a scratch pad register to save and restore the contents of the T register.

Coding Example: (see example for POP)

DUP

Code: 1A

Usage: 011010 cccccc cccccc cccccc
cccccc 011010 cccccc cccccc

```
                        cccccc cccccc 011010 cccccc
                        cccccc cccccc cccccc 011010

Stack Effects:          ( n -- n n )
Carry:                  unchanged

Function:

Duplicate T register and push it on the data stack.

Coding Example:

Decrement T             dup dup xor com add     ( now use “zero com add” )


OVER

Code:                   1B

Usage:                  011011 cccccc cccccc cccccc
                        cccccc 011011 cccccc cccccc
                        cccccc cccccc 011011 cccccc
                        cccccc cccccc cccccc 011011

Stack Effects:          ( n1 n2 –- n1 n2 n1 )
Carry:                  unchanged

Function:

S is transferred into T register.  The original contents in the T register is
pushed onto the data stack.

Coding Example:

CODE 2DUP ( n1 n2 – n1 n2 n1 n2 )
   over over ret


PUSH

Code:                   1C

Usage:                  011100 cccccc cccccc cccccc
                        cccccc 011100 cccccc cccccc
                        cccccc cccccc 011100 cccccc
                        cccccc cccccc cccccc 011100

Stack Effects:          ( n -- ; R: -- n )
Carry:                  unchanged

Function:

Pop S on the data stack and store it to the T register.  The original contents
in the T register is pushed onto the return stack.

Coding Example:
```

```
CODE ROT ( w1 w2 w3 -- w2 w3 w1 )
   push push sta pop
   pop lda ret
```

STA

Code: 1D

Usage:
```
011101 cccccc cccccc cccccc
cccccc 011101 cccccc cccccc
cccccc cccccc 011101 cccccc
cccccc cccccc cccccc 011101
```

Stack Effects: (a --)
Carry: no change

Function:

Pop S on the data stack and store it to the T register. The original contents in the T register is copied into the A register. This instruction initializes the A register so that it can be used to fetch data from memory or store data into memory.

Coding Example:

```
CODE ! ( n a -- )
      sta st ret
```

NOP

Code: 1E

Usage:
```
011110 xxxxxx xxxxxx xxxxxx
cccccc 011110 xxxxxx xxxxxx
cccccc cccccc 011110 xxxxxx
cccccc cccccc cccccc 011110
```

Stack Effects: (--)
Carry: no change

Function:

No operation. This instruction will force the execute state to slot 5, to get the next word to be fetched and executed. (Actually this is what the NOP SHOULD do, but in the current ACTEL implementation the NOP instead passes control to the next instruction slot.)

Coding Example: usually inserted by assembler.

DROP

Code: 1F

```
Usage:                  011111 cccccc cccccc cccccc
                        cccccc 011111 cccccc cccccc
                        cccccc cccccc 011111 cccccc
                        cccccc cccccc cccccc 011111

Stack Effects:          ( n -- )
Carry:                  unchanged
```

Function:

Pop S on the data stack and store it to the T register. The original contents in the T register are lost.

Coding Example: see example for jump.

Chapter 3.1 G Buss Instructions

G@

```
Code:                   2G, where G is 4bit G-Buss address

Usage:                  10gggg cccccc cccccc cccccc
                        cccccc 10gggg cccccc cccccc
                        cccccc cccccc 10gggg cccccc
                        cccccc cccccc cccccc 10gggg

Stack Effects:          ( n -- )
Carry:                  unchanged
```

Function:

Read G-buss address gggg and store in T, push original contents of T to data stack.

Coding example:

```
CODE KEY ( -- c )       ( waits for a serial character and returns it )
        begin g0@ $10000 ldi and until g1@ ret
```

Note: Presently assembler words are implemented only for reading Gbuss addresses 0, 1, 2, and 3. The corresponding assembler words are “g0@, g1@, zero, and one”. See discussion below.

G!

```
Code:                   3G, where G is 4bit G-Buss address

Usage:                  11gggg cccccc cccccc cccccc
                        cccccc 11gggg cccccc cccccc
                        cccccc cccccc 11gggg cccccc
                        cccccc cccccc cccccc 11gggg

Stack Effects:          ( n -- )
```

```
Carry:                  unchanged

Function:

Pop S from data stack into T. Original contents of T are written to G-buss
address gggg.

Coding example:

CODE EMIT ( c -- )      ( waits for previous transmit to complete, then )
                        ( sends character )
        begin g0@ $20000 ldi and until g1! Ret
```

Chapter 3.2 G-Buss Peripherals (Interrupt Register and UART)

Currently, G-buss peripherals include only an interrupt control and status register at address 0, the UART send/receive data register at address 1, and read-only “zero” and “one” at addresses 2 and 3 respectively.

The interrupt control register at address 0 is defined as follows:

Bit0(lsb)	r/w	global interrupt enable (1 to enable)
Bit1	r/w	enable for interrupt 0 (highest priority)
Bit2	r/w	enable for interrupt 1
Bit3	r/w	enable for interrupt 2
Bit4	r/w	enable for interrupt 3
Bit5	r/w	enable for interrupt 4
Bit6	r/w	enable for interrupt 5
Bit7	r/w	enable for interrupt 6
Bits8-15		not used, read as 0
Bit16	r	status of interrupt line 0
Bit17	r	status of interrupt line 1
Bit18	r	status of interrupt line 2
Bit19	r	status of interrupt line 3
Bit20	r	status of interrupt line 4
Bit21	r	status of interrupt line 5
Bit22	r	status of interrupt line 6
Bit23	r	logical OR of enabled interrupt lines

The interrupt enable bits are all initialized to 0 by power-on reset. Note that after performing an interrupt, the interrupt controller will (without clearing bit 0) disable further interrupt servicing until after the next RTI instruction is executed. Interrupt lines 0 and 1 are currently dedicated to the UART receive and transmit functions respectively. Interrupt line 0 is set after a character is received by the UART and is cleared by reading the data at Gbuss address 1 (least significant byte). Interrupt line 1 is set after the UART has completed a character transmission and is cleared by writing a character to Gbuss address 1 (least significant byte). (An example of the programming of the UART for interrupt driven i/o is in file multi3.f. Examples of polled i/o are the KEY and EMIT words shown above.).

Gbuss addresses 2 and 3 are currently read-only, returning 0 and 1 respectively. (Special assembler words “zero” and “one” compile the instructions to read Gbuss addresses 2 and 3.)

Gbuss addresses 2 and 3 are available for write-only applications. Gbuss addresses 4-15 are entirely available.

Typical Gbuss peripherals that might be added depending on application include:

(1) A timer to produce periodic interrupts at a programmable interval.
(2) A FIFO for buffering event data, producing an interrupt when full or half full.
(3) Scratch-pad registers.
(4) Hardware single-step multiplier, with operands taken from pair of scratch pad registers.
(5) I/O ports.
(6) Registers to control and monitor app-specific on-chip functions.
(7) Additional UARTs.

Chapter 4. P24 Metacompiler

(This section has not been updated to reflect changes made for the ACTEL implementation. See the comments in files meta24i.f, ok24i.f, kern24i.f, and ef24i.f for a log of the changes. The main changes are use of the hardware UART, inclusion of target-resident assembly words and the repositioning of certain buffers and system variables used by Forth. Multi-tasking words and buffered interrupt-driven serial i/o words are in the file multi3.f and are not currently present at boot. Multi-tasking words are documented by comments in the file multi3.f. In addition, some words were added in meta24i.f to facilitate prom burning etc. – see “prom.” and “image.” The details of the boot method have changed. Presently booting can occur from EEPROM or serially. To be documented later. Changes near the end of meta24i.f relate to booting. -- WRC)

Metacompiler is a term used by Forth programmers to describe the process of building a new Forth system on an existing Forth system. The new Forth system may run on the same platform as the old Forth system. It may be targeted to a new platform, or to a new CPU. The new Forth system may share a large portion of the Forth code with the old system, and hence the term metacompilation. In a sense, a metacompiler is very similar to a conventional cross assembler/compiler.

The P24 eForth metacompiler is contained in the source code file META24.F, which runs under Win32Forth, a public domain Forth for Windows 95/98/2000/NT. It calls on the following files to build the P24 eForth system:

ASM24.F	P24 assembler
KERN24.F	Kernel words in P24 eForth
EF24.F	High level words in P24 eForth
K24.F	Words to replace assembly macros

In this chapter, I take the source code in META24.F and explain the functions of the Forth words which build the eForth system.

4.1 Start Up the Metacompiler

```
comment:
meta24.f  07nov00cht, change for P24, p24c
          02dec00cht, interpreter ok, debugging compiler

This meta-compiler was originally written by Chuck Moore to build
Forth systems for the MuP21 microprocessor.  It can be easily
changed and used to compile code for any CPU.

This file loads all the source code and construct an image of P24
which can be ported to VHDL for Xilinx XCV300/1000 FPGA.  It runs
under Win32Forth, a public domain Forth system authored by Andrew
McKewan and Tom Zimmer.  It can be downloaded from the web, at
www.forth.org, under the category of Compiler/Windows.  Click on
the download button, and it will be downloaded to your computer
and automatically installed.

Run Win32Forth from your desktop, or from Start/Program/Win32Forth and
you will see a window opened.  Open the WinView editor from the File
menu.  Open the file META24.F through the directory tree.  Then
click back the Win32Forth window and type:
        fload meta24
Youu will see the list of all the words compiled into the P24
target image.  Type
        showram
to show the image dumped out in hexidecimal.  Type
        bram
to see the image dumped in a form acceptable by Xilinx VHDL.  Cut
and paste this image into your VHDL code and synthesize the P24
system.
```

4.2 Tools of Metacompiler

```
 \ create two vocabularies. ASM24 will contain the assembler woprds
 \ and the words in the P24 target.  SIM24 will contain words which
 \ build a cycle-based simulator to exercise code in the P24 target.

VOCABULARY ASM24
VOCABULARY SIM24

\ following are tools words added to the baseline Forth system
ONLY FORTH ALSO DEFINITIONS

\ turn off the warnings normally supplied by Win32Forth on
\ duplicated names and stack changes.  They clutter the symbol
\ table.
HEX
WARNING OFF
' NOOP IS STACK-CHECK

\ type 'debugging? on' and you can pace the meta-compiler by
\ hitting SPACE for the next steps.  Hit RET to stop.
\ This is useful when you want to locate errors before a full
\ compilation.
variable debugging?
debugging? off
```

```
\ .HEAD prints teh name of a new word to be compiled.
: .head ( addr -- addr )
   >IN @ 20 word count type space >IN !
   dup .
   ;

\ CR is redefined so you can step through the compilation by
\ setting debuggin? on.
: CR CR
   debugging? @
   if .s KEY 0D = abort" done"
   then
   ;

\ Here is a group of Forth words which clash with words in the
\ target.  You can use the aliases to ensure that you still
\ has the behavior of the original Forth words.
' '      alias forth'
' dup    alias forthDUP
' drop   alias forthDROP
' over   alias forthOVER
' swap   alias forthSWAP
' @      alias forth@
' !      alias forth!
' and    alias forthAND
' +      alias forth+
' -      alias forth-
' word   alias forthWORD
' CR     alias CRR
' .(     alias forth.(
' count  alias forthCOUNT

\ Chuck Moore preferred this name for XOR.
: -OR   XOR ;

\ RAM is a large array to hold the binary image of P24 target.
\ P24 is a 24-bit CPU.  One 24 bit program word is compiled into
\ a 32-bit word in this array.
\ RESET clears the RAM array.
\ RAM@ ( a ) uses a word address to fetch a program word in RAM.
\ RAM! ( n a ) stores a word n into RAM at address a.
CREATE ram  8000 ALLOT
: RESET   ram 8000 ERASE ;   RESET
: RAM@   4 * ram +  @ ;
: RAM!   4 * ram +  ! ;

\ FOUR displays four consecutive words in target.
\ SHOW displays 128 words in target from address a.  It also returns
\ a+128 so you can SHOW the next block of 128 words.
\ SHOWRAM displays the entire image, 2048 words.
: FOUR   4 0 DO  DUP RAM@ 7 U.R  1+ LOOP ;
: SHOW ( a)   10 0 DO  CR  DUP 7 .R SPACE
      FOUR SPACE FOUR  LOOP ;
: showram 0 0c 0 do show loop drop ;

\ UD. displays a 24-bit word in 8 digits with leading zeros.
```

```
\ B. displays one byte in two digits.
\ C. displays nibble in one digit.

\ STRING. displays the attribute init string in the VHDL format
\ required by Xilinx Foundation synthesizer.  The string
\ "attribute INIT_" is temporarily replaced by "qq" to avoid
\ lines broken by Forth output routine.  When the whole
\ attribute blocks are pasted into the VHDL code, "qq" must
\ be globally replaced by "attribute INIT_".

\ EIGHT displays one line of memory attribute for VHDL
\ BLOCKRAM displays one block of memory attributes for VHDL
\ BRAM dumps the entire memory blocks for VHDL
: ud. 0 <# # # # # # # # # #> type ;
: b.  0 <# # # #> type ;
: c.  0 <# # #> type ;
: string. ( a ) 8 / 10 /mod swap
\        ." attribute INIT_" b.
         ."  qq" b.
         ."  of memory" 0F and c.
         ." :label is " 22 emit ;
: eight   8 + dup 8 0 DO  1 - DUP RAM@ ud.  LOOP DROP ;
: blockram ( a)   10 0 DO  CR DUP string.
      eight 22 emit 3B emit LOOP cr ;
: BRAM base @ hex 0 0F 0 do blockram loop drop base ! ;
```

4.3 Calling Other Building Blocks

```
 \ Now we compile the structured, optimizing assembler for P24.
 CR .( include asm24 )
 include ok24

 \ Now we compile the kernel portion of the P24 eForth system.
 $18 org
 CR .( include eforth kernel )
include kern24

\ This set of words will be used to build high level control
\ structures in the body of eForth system:
\       BEGIN ... AGAIN
\       FOR ... NEXT
\       FOR AFT ... THEN NEXT
\ LIT let LDI to assemble a literal
\ $LIT compiles a counted ASCII string, packed three bytes to
\ a 24-bit program word.
: again ( a -- )
   jump ;
: for ( -- a )
   push begin ;
: next ( a -- )
   doNEXT jump ;
: <next> next ;
: aft ( a -- a' a" )
   forthDROP begin 0 jump begin forthSWAP ;
: LIT ( d -- )
   ldi ;
```

```
: $LIT ( -- )
   22 forthWORD forthCOUNT
   forthDUP ,B ( compile count )
   0 DO
      forthCOUNT ,B ( compile characters )
   LOOP
   forthDROP ;

\ ;; terminates a high level colon word with a ret.
\ WAIT pauses the execution.  Restart by any key.  This is
\ a cheap breakpoint mechanism, now replaced by the simulator.
' EXIT alias ;;
\ ' WAIT alias ;;               \ debugger

\ CREATE builds a new array word.  doVAR returns the array address.
\ VARIABLE builds a variable in P24 target.
: CREATE makeHead begin .head CONSTANT doVAR DOES> forth@ call ;
: VARIABLE CREATE 0 #, ;

\ Ready to compile the high level portion of the P24 eForth.
CR .( include eforth24 )
include ef24

\ Compile Forth words used as macros in assembler, but now needed
\ so the Forth interpreter and compiler have access to these
\ functions.
CR
include k24
```

4.4 Boot Code

```
\ Build the boot code starting at location 0.  This piece of code
\ initializes the variables in RAM memory and then jumps to COLD.
0 ORG
10 LIT 704 LIT 6 LIT
forth' COLD >body forth@ LIT
push push
anew H forth@
   push sta ldp push
   lda pop pop sta
   stp lda
<next>
pops pops ret

\ Build the table of initial values for the variables to be
\ copied to RAM memory on booting.
10 ORG
730 #,
0A #,
lastH forth@ #,
780 #,
lastH forth@ #,
forth' $INTERPRET >body forth@ #,
forth' QUIT >body forth@ #,
```

Chapter 5 The Optimizing P24 Assembler

This ASM24.F file contains a Structured, Optimized Assembler for P24 CPU. It packs up to four machine instructions into one 24-bit program word. It also builds structures similar to those in high level Forth. The structures are build in a single pass, without labels.

P24 eForth adopts the Subroutine Thread Model, in which the colon words contain lists of subroutine calls, instead of lists of addresses. Using this model, the assembler assumes the duties of the compiler. Another advantage of the Subroutine Thread Model is that machine instructions can be assembled in-line with the colon words.

5.1 Assembly Tools

```
\ Put all the assembly words and words in the P24 target into
\ the ASM24 vocabulary.
ONLY FORTH ALSO ASM24
ASM24 DEFINITIONS

\ H points to the next free location in the target image to
\ receive new code or data.
\ LOC marks a target location to be reference later. It is not
\ used in the P24 system.
VARIABLE H
: LOC   CONSTANT  DOES> @  H ! ;

\ LASTH contains the link field address of the last word
\ to build the linked list of Forth words.
variable lastH 0 lastH !        \ init linkfield address lfa

\ NAMER! stored n into the next location in target.
\ It was useful if you want to assemble names in a separated
\ names dictionary.
: nameR! ( n -- )
   H @ RAM!                     \ store double to code buffer
   1 H +!                       \ bump nameH
   ;

\ COMPILE-ONLY marks the current word so it can only be compiled
\ in a colon word.  It cannot be executed interactively.
\ IMMEDIATE marks the current word so it will be executed in
\ a colon word.  All structure building words are so marked.
: compile-only 400000 lastH @ RAM@ -OR lastH @ RAM! ;
: immediate    800000 lastH @ RAM@ -OR lastH @ RAM! ;

\ Derived from Chuck Moore's P21 20 bit assembler
\ HI selects one of four masks to assemble a machine instruction
\ into one of the four slots in a 24-bit program word.
\ HW points to the program word into which new machine instructions
\ are to be assembled.  H may advance from HW as literal values
```

```
\ are assembled following the program word.
\ BI points to one of the 3 bytes in a 24-bit program word.  It
\ allows the assembler to pack 3 ASCII characters into one word.
VARIABLE Hi   VARIABLE Hw
VARIABLE Bi ( for packing)

\ ALIGN forces the next instruction to be assembled into the next
\ word.
\ ORG ( a ) changes H to a, to start assembling at a new location.
: ALIGN  10 Hi ! ;
: ORG    DUP . CR H !  ALIGN ;

\ MASK contains four mask patterns to assemble a machine instruction
\ into one 6-bit slot of a program word.  The mask is selected by
\ HI.
\ #, ( n ) assembles n into the next free location pointed by H.
\ Advance H afterwards.
\ ,W ( n ) assembles a machine instruction to the next free slot
\ in the current program word pointed to by HW.
\ ,I ( n ) assembles a machine instruction.  It the current word
\ is full, assemble the instruction into the next word.
\ ,B ( c ) packs a character c into the current word.  Uses BI to
\ determine the character postion in a 24-bit word.  Pack it to
\ the next word if the current word is full.
CREATE mask  FC0000 ,  3F000 ,  FC0 ,  3F ,
: #,     FFFFFF AND H @ RAM!  1 H +! ;
: ,w     Hw @ RAM@  -OR  FFFFFF AND Hw @ RAM! ;
: ,I     Hi @ 10 AND IF  0 Hi !  H @ Hw !  0 #,  THEN
         Hi @ mask + @ AND  ,w  4 Hi +! ;
: ,B   ( c ) Bi @ 0 = IF 1 Bi ! H @ Hw ! 0 #, 10000 * ,w EXIT THEN
             Bi @ 1 = IF 2 Bi ! 100 * ,w EXIT THEN
             0 Bi ! ,w ;
```

5.2 Transfer Instruction Assembler

```
 \ INST A defining word to define a single slot machine instruction.
\ When the machine instruction word is executed, it assembles the
\ desired machine instruction into the current program word.  If
\ the current word is full, start a new program word.  The constant
\ contained in the machine instruction word has four identical
\ machine code patterns in the four slots, so that the word ,I
\ can select one of them and add it to the current program word.
\ NOP machine instruction word has 1E in all four slots.  They
\ make up the 24-bit pattern 79E79E.
: INST   CONSTANT   DOES> @  ,I ;
79E79E INST nop

\ ANEW starts a new program word by filling the current word with
\ NOPs.  It is required when we have to assemble a 4-slot
\ instruction like CALL, BZ, or BNZ.
: anew   BEGIN Hi @ 10 AND 0= WHILE nop REPEAT 0 Bi !
         H @ Hw ! ;

\ JMP A defining word to assemble a 4-slot long instruction.  The
\ machine instruction thus defined will take an address on the
\ stack and assemble the least significant 18 bits of the address
```

```
\ into the address field of the instruction.
: JMP    CONSTANT  DOES> @  anew SWAP 3FFFF AND -OR #, ALIGN ;

\ BEGIN starts a new program word and marks its address on stack.
\ -;' terminates a colon word by changing the last subroutine
\ call into a jump.  This is tail-recursion, which saves the
\ return instruction at the end of a colon word.
\ LDI ( n ) assembles a Load-Immediate machine instruction and
\ add the literal value to the next word.
: begin  anew  H @ ;
: -;'    Hw @ RAM@ DUP $FC0000 AND 100000 =
         IF 100000 -OR Hw @ RAM! ELSE DROP THEN ;
: ldi    28A28A ,I  #, ;

\ CALL assembles a 18-bit call instruction to a location in the
\ current page of 256K words.
\ JUMP assembles a 18-bit jump instruction to a location in the
\ current page of 256K words.
\ BZ assembles a 18-bit conditional branch to a location in the
\ current page of 256K words.  Branch on T=0.
\ BNC assembles a 18-bit conditional branch to a location in the
\ current page of 256K words.  Branch on Carry Not Set.
\ UNTIL assembles a branch on T=0.
\ -UNTIL assembles a branch on no carry.
100000 JMP call
0 JMP jump   80000 JMP bz    C0000 JMP bnc
             80000 JMP until  C0000 JMP -until

\ The following words build structures in assembly code words,
\ much like those in the high level code.  Since we use the
\ Subroutine Thread Model for colon words, these structure words
\ will be used in the colon words as well.  The structures are:
\       IF ... THEN
\       IF ... ELSE ,,, THEN
\       SKIP ... THEN
\       BEGIN ... AGIAN
\       BEGIN ... UNTIL
\       BEGIN ... WHILE ... REPEAT
\ -IF, -WHILE, -UNTIL are similar to IF, WHILE, UNTIL, except
\ that they assemble BNC instead of BZ.
: if     begin 0 bz ;
: -if    begin 0 bnc ;
: skip   begin 0 jump ;
: then   DUP >R >R  begin  3FFFF AND  R> RAM@ -OR  R> RAM! ;
: else   skip SWAP then ;
: while   if  SWAP ;
: -while -if  SWAP ;
: repeat jump then ;
: again  jump ;
```

5.3 Single Slot Instructions

```
\ Here is the list of all the single slot machine instructions.
\ RET returns from subroutine call.
\ LDP loads a word and pushes it to T.  Address of word is in A.
\     A is auto-incremented.
```

```
\ LD load a word and pushes it to T.  Address of word is in A.
\ STP pops a word from T and stores it to memory.  Address of
\     word is in A.  A is auto-incremented.
\ ST pops a word from T and stores it to memory.  Address of
\     word is in A.
\ COM compliments T, ones compliment.
\ SHL left shifts T by one bit.
\ SHR right shifts T by one bit.  Arithmetic right shift.
\ MUL multiply step.
\ XOR pops data stack and XORs it to T.
\ AND pops data stack and ANDs it to T.
\ DIV divide step.
\ ADD pops data stack and ADDs it to T.
\ POP pushes T on data stack, and pops return stack to T.
\ LDA pushes T on data stack, and copy A to T.
\ DUP pushes T on data stack.
\ PUSH pushes T on return stack, and pops data stack to T.
\ STA Copies T to A and pops data stack to T.
\ DROP pops data stack to T.

                41041 INST ret
249249 INST ldp   2CB2CB INST ld    34D34D INST stp   3CF3CF INST st
410410 INST com   451451 INST shl   492492 INST shr   4D34D3 INST mul
514514 INST xor   555555 INST and   596596 INST div   5D75D7 INST add
618618 INST pop   659659 INST lda   69A69A INST dup
71C71C INST push  75D75D INST sta ( 79E79E INST nop ) 7DF7DF INST drop
```

5.4 Miscellaneous Assembly Tools

```
 \ POPS alias of DROP, useful after high level DROP is defined.
 \ PUSHS alias of DUP, useful after high level DUP is defined.
 : pops drop ;
: pushs dup ;

\ LJUMP a long jump to a 24-bit address.  Pushes the address
\ on the return stack and then execute RET.
: ljump  ' >body @ ldi           \ get address of target word
   push  ret ;                    \ long jump

\ (MAKEHEAD) builds a header for a new word.  First builds the
\ link field using LASTH, and then packs the name count and name
\ into the name field, three bytes per word.
\ MAKEHEAD builds a header while retains the word pointer so
\ that the name string is still available to be printed.
: (makeHead)
   anew
   20 word                       \ get name of new definition
   lastH @ nameR!                \ fill link field of last word
   H @ lastH !                   \ save nfa in lastH
   forthdup c@ ,B                 \ store count
   count 0 do
      count ,B                \ fill name field
   loop forthdrop anew
   ;
: makeHead
   >IN @ >R                      \ save interpreter pointer
```

```
   (makeHead)
   R> >IN !                    \ restore word pointer
   ;

\ $LIT packs a counted string into the next available space
\ in the target image, three bytes to a word.
: $LIT ( -- )
   anew
   22 WORD
   forthDUP c@ ,B ( compile count )
   count 0 DO
      count ,B ( compile characters )
   LOOP
   forthDROP anew ;

\ ': builds a new subroutine in the target without a header.
\ CODE builds a new code word in the target with a header.
\ :: builds a new colon word in the target with a header.  As
\ we are using the Subroutine Thread Model, :: is the same as
\ CODE, and colon words shared all the structure building tools
\ with code words.
: ':     begin  .head CONSTANT  DOES> @  call ;
: CODE makeHead ': ;
: :: CODE ;
```

Chapter 6. The P24 Kernel

The KERN24.F file contains most of the words which are written in assembly for speed considerations. P24 eForth is optimized as all the words which can be written in assembly are so done. However, much more optimization is achieved by a set of macros, which try to convert the most commonly used high level Forth words into machine instructions and packs these machine instruction as tightly as possible. The end results are that the code size is significantly reduced and the execution speed greatly increased.

The use of macros will be further explained along with the code.

The code words in this file alse serve as programming examples for the optimal use of the P24 CPU. It is worth you while to study them carefully, and use them as templates when you like to convert high level application words into assembly.

The Forth Virtual Engine is:

T	top of stack	
S	data stack	16 levels
R	return stack	16 levels

Both the data and return stacks are in CPU.

The A register is used by the memory fetching and storing instructions to provide address to the external memory. When not used to address memory, A can be used as a scratched register.

In the MUL and DIV instructions, the A register serves as the extension to the T register to hold the lower half of the partial product or the divident.

Subroutine thread model eliminates IP, doColon, and EXIT.

16 levels of stacks are enough for most applications. They will wrap around when exhausted.

Memory allocation:

0	Boot code
10	Initial variables
18	Kernel
9A	Forth words
700	RAM, variables
710	Text buffer
730	TIB
780	User dictionary
7FF	End of memory

6.1 System Variable

```
\ All the system variables are defined as macros.  They will be
\ assembled as literals in the form of LDI instructions.  On
\ execution, they will return their respective addresses on the
\ data stack.  It is assumed that the target system has RAM starting
\ from location $700.  For a different target system, you have
\ to change the locations in these macros.

\ HLD points to buffer for output numeric string.
\ SPAN variable to hold the length of input text string.
\ >IN offset to the text string currently being interpreted.
\ #TIB length of the input text string
\ 'TIB location of the terminal input buffer.
\ BASE base for number conversions
\ CONTEXT pointer to start dictionary searches
\ CP points to the top of the dictionary
\ LAST points to the name field of the last word
\ 'EVAL points to $INTERPRET or $COMPILE to evaluate words
\ 'ABORT points to error recovery routine
\ TEXT points to text buffer to unpack strings
\ tmp a scratch pad variable.
hex
CRR .( System variables ) CRR
: HLD 700 ldi ;         \ scratch
: SPAN 701 ldi ;        \ #chars input by EXPECT
: >IN  702 ldi ;        \ input buffer offset
: #TIB 703 ldi ;        \ #chars in the input buffer
: 'TIB 704 ldi ;        \ TIB
: BASE 705 ldi ;       \ number base

CRR
: CONTEXT 706 ldi ;    \ first search vocabulary
: CP 707 ldi ;         \ dictionary code pointer
: LAST 708 ldi ;       \ ptr to last name compiled
```

```
: 'EVAL 709 ldi ;       \ interpret/compile vector
: 'ABORT 70A ldi ;
: TEXT 710 ldi ;         \ unpack buffer
: tmp 70B ldi ;        \ ptr to converted # string
```

6.2 Assembly Macros for Code Optimizing

```
\ Many Forth words have corresponding P24 machine instructions
\ or can be represented by a short sequence of P24 machine
\ instructions.  Instead of representing them in subroutines,
\ they are defined as macros, which invoke the assembler
\ mneumonics to pack as many machine instructions to program
\ words.
\ Obviously, if a Forth words can be translated to less than
\ four machine instrucitons, there are gains in shorter code
\ sizes and faster execution speed.  However, there are also
\ significant gains when a Forth word is defined as a 4 machine
\ instruction macro, because it may continue the packing from
\ the previous word to the next word.
\ These macros together with the machine instructions
\       DUP, DROP, AND, XOR
\ tend to pack the code tightly.
CR .( macro words ) CR
: EXIT ret ;
: EXECUTE ( a ) push ret ;
: ! ( n a -- ) sta st ;
: @ ( a - n ) sta ld ;
: R> ( - n ) pop ;
: R@ ( - n ) pop dup push ;
: >R ( n ) push ;
: SWAP ( n1 n2 - n2 n1 )
   push sta pop lda ;
: OVER ( n1 n2 - n1 n2 n1 )
   push dup sta pop
   lda ;
: 2DROP ( w w  -- )
   drop drop ;
: + ( w w -- w ) add ;
: NOT ( w -- w ) com ;
:  NEGATE ( n -- -n )
   com 1 ldi add ;
: 1- ( a -- a )
   -1 ldi add ;
: 1+ ( a -- a )
   1 ldi add ;
: BL ( -- 32 )
   20 ldi ;
: +! ( n a -- )
   sta ld add st
   ;
: - ( w w -- w )
   com add 1 ldi add
   ;
```

6.3 Forth Words Coded in Assembler

```
 \ Following words are complicated and have to be defined as
 \ code word.
 \ doVAR starts a variable or an array.  It returns the address
 \ following doVAR.
\ doNEXT terminates a FOR-NEXT loop.  It decrements the counter
\ on the return stack.  It exits the loop when the count is 0.
CR .( kernel words ) CR
CODE doVAR
   pop ret
CODE doNEXT
   pop pop dup                \ decrement count
   if -1 ldi add push
      push ret             \ if index is not 0, loop back
   then
   drop 1 ldi add                  \ index is 0, exit loop and continue
   push ret

\ Following are Forth words which are too long for macros,
\ yet still easily expressible in machine instructions.
\ They are all commonly used Forth words.
\ UM+ ( n n -- sum carry ) is a special word in eForth to
\ provide carry in addition.  However, it is not used here
\ because carry is readily accessible using -if or BNC.
\ Note that BZ removes the flag tested from the stack, while
\ BNC does not disturb the data stack.  Thus BZ can be used
\ to code IF directly, and BNC will let -IF to test T repeatedly
\ using the SHL instruction.
CR
CODE 0< ( n - f )
   shl
   -if drop -1 ldi ret
   then
   dup xor ( 0 ldi )
   ret
CODE OR ( n n - n )
   com push com
   pop and com ret
CODE UM+  ( n n - n carry )
   add
   -if 1 ldi ret
   then
   dup dup xor ( 0 )
   ret
CODE ?DUP ( w -- w w | 0 )
   dup
   if dup ret then
   ret
CODE ROT ( w1 w2 w3 -- w2 w3 w1 )
   push push sta pop
   pop lda ret
CODE 2DUP ( w1 w2 -- w1 w2 w1 w2 )
   dup push push
   dup sta pop lda pop
   ret

CR
```

```
CODE DNEGATE ( d -- -d )
   com push com 1 ldi
   add
   -if pop ret
   then
   pop 1 ldi add ret
CODE ABS ( n -- +n )
   dup shl
   -if drop com 1 ldi add
       ret
   then
   drop ret

CR
CODE = ( w w -- t )
   xor
   if dup dup xor ret then
   -1 ldi ret
CODE 2! ( d a -- )
   sta push stp
   pop st ret
CODE 2@ ( a -- d )
   sta ldp ld ret
CODE COUNT ( a -- a+1 n )
   sta ldp push lda
   pop ret
```

6.4 Packing and Unpacking Text Strings

```
 \ B> adds one byte at b to the word at a.  It shifts the
 \ existing data in a left by 8 bits.  Returns b+1 and a,
 \ and is ready to pack in the next byte.
\ B> is used by PACK$ to pack a byte string into a packed
\ string.
\ >B unpacks three bytes in a and puts them at b.  Returns
\ a+1 and b+3 so it is ready to unpack the next word.  The
\ first byte unpacked is also return as a count, which is
\ useful when this word is the first word of a packed string.
\ >B is called by UNPACK$ to convert a packed string to a
\ counted byte string.
CR ( pack B> and unpack >B strings )
CODE B> ( b a -- b+1 a )
   push sta ldp push
   lda pop pop sta
   ld
   shl shl shl shl
   shl shl shl shl
   add st lda ret
CODE >B ( a b -- a+1 b+3 count )
   push sta ldp push
   lda pop pop ( a+1 n b ) sta
   dup push
   $FF ldi and pop
   $FFFF00 ldi and $FF ldi xor
   shr shr shr shr
   shr shr shr shr
```

```
   dup push
   $FF ldi and pop
   $FFFF00 ldi and $FF ldi xor
   shr shr shr shr
   shr shr shr shr
   $FF ldi and dup push
   stp stp stp ( a+1 c )
   lda pop ret
```

Chapter 7. High Level Words in P24 eForth

The file EF24.F contains all the high level words in P24 eForth. This implementation follows closely the eForth model. The following set of words are removed because they are not absolutely necessary for embedded applications. In this implementation, the size constrain is severe, and the existence of every word must be justified rigorously.

Words removed from the eForth model:

```
        CATCH, THROW, PRESET, XIO, FILE, HAND, I/O
        CONSOLE, RECURSE, USER, VER, HI, 'BOOT
```

Most of the user variables are eliminated:

```
        SP0, RP0, '?KEY, 'EMIT, 'EXPECT, 'TAP, 'ECHO
        'PROMPT, CSP, 'NUMBER, HANDLER, CURRENT, NP
```

Only these user variables remain and are macros:

```
        HLD, SPAN, >IN, #TIB, 'TIB, 'EVAL, BASE, tmp
        CP, CONTEXT, LAST, 'ABORT, TEXT
```

The P24 eForth system can be summarized in the following words and their pseudo code:

COLD boots Forth, print sign-on message and jump to QUIT

QUIT repeats the sequence: accepts a line of text and executes the commands in sequence. The pseudo code is:

```
     : QUIT BEGIN QUERY EVAL AGAIN ;
```

QUERY accepts one line of text of 80 characters or terminated by a carriage-return.

EVAL parses out tokens in the text and evaluates them:

```
     : EVAL BEGIN TOKEN WHILE 'EVAL @EXECUTE REPEAT .OK ;
```

TOKEN parses out one word from the input text.

'EVAL contains $INTERPRET in the interpret mode or $COMPILE in the compiling mode.

@EXECUTE executes either $INTERPRET or $COMPILE.

.OK prints out the "OK" message.

$INTERPRET (a) searches the dictionary for a word of the text string at a. If the word exists, execute it. Else, convert the string into a number on the stack. Failing to convert the string to a number, prints an error message and abort to QUIT.

```
     : $INTERPRET NAME? IF EXECUTE ELSE NUMBER?
         IF ELSE ERROR THEN THEN ;
```

$COMPILE (a) searches the dictionary for a word of the text string at a. If the word exists, compile it.

```
     Else, convert the string to a number and compile the
     number as a literal.  Failing the conversion, prints
     a message and abort to QUIT.
     : $COMPILE NAME? IF , ELSE NUMBER?
        IF LITERAL ELSE ERROR THEN THEN ;
NAME? calls 'find' to locate a word of the name parsed out
     out the input text string.
NUMBER? ( a ) converts the text string at a to a number.
ERROR prints the offending text string and aborts to QUIT.
LITERAL ( n ) compiles n as a literal into the current word
     being compiled.

The above words serve as a top-down map of the eForth operating
system.  The eForth system source code builds up to QUIT and
COLD.  Most words in EF24.F are necessary in the building
process.  The eForth system can be viewed as a very sophisticated
application of P24.  Most applications are much simplier than
eForth system.  You can model your application code to eForth,
and use all the tools contained therein.
```

8.1 Serial Port

```
\ 50us delays 52 us, half of a bit at 9600 baud.
\ 100us delays 104 us, one bit frame at 9600 baud.

\ EMIT ( c ) sends character c to the serial output port.

\ KEY ( -- c ) waits for a character from the serial input port.
\ The serial ports are actually connected to the T register.

\ The No-Cost UART
\ On executing SHR instruction, the least significant bit in
\ T, T(0), is shifted to a flip-flop, whose output is
\ connected to the serial output port.  At the same time
\ the state of the serial input port is latched into the
\ carry bit, which is bit T(24).  Repeating SHR 8 times,
\ a character is sent out.  One character is captured by
\ waiting for the start bit on the serial input port, and then
\ test the port at the intervals of 100 us.
\ One must be very careful in using the SHR instruction.
\ In order not to disturb the output port, you should always
\ set T(0) to a 1 before executing SHR.  This way, the serial
\ output port stays at the mark level.

CRR .( Chararter IO ) CRR
CODE 50us
        2 ldi skip
CODE 100us
        1 ldi
        then
        sta -138 ldi
        begin lda add
        -until
        drop
        ret
CODE EMIT ( c -- )
```

```
        $7F ldi and
        shl $FFFF01 ldi xor
        $0A ldi
        FOR shr 100us NEXT
        drop ret
CODE KEY ( -- c )
        $FFFFFF ldi
        begin   shr
        -until
        repeat   ( wait for start bit )
        50us
        7 ldi
        FOR
          100us shr
          -if $80 ldi xor then
        NEXT
        $FF ldi and
        100us ret
```

8.2 Simple Utility Words

```
 \ These common functions are too complicated to code in machine
 \ instructions, and are left in the high level form.
CRR .( Common functions ) CRR
:: U< ( u u -- t ) 2DUP XOR 0< IF SWAP DROP 0< EXIT THEN - 0< -;'
::  < ( n n -- t ) 2DUP XOR 0< IF      DROP 0< EXIT THEN - 0< -;'
:: MAX ( n n -- n ) 2DUP      < IF SWAP THEN DROP ;;
:: MIN ( n n -- n ) 2DUP SWAP < IF SWAP THEN DROP ;;
:: WITHIN ( u ul uh -- t ) \ ul <= u < uh
  OVER - >R - R> U< -;'
```

8.3 Division

```
 \ UM/MOD and /MOD share the same body to do division of a 48-bit
 \ divident by a 24 bit divisor, using the DIV machine instruction.
 \ The higher half of the divident is placed in T and the lower
\ half is placed in A.  The divisor is negated and placed on the
\ data stack below T.  The negated divsor is added to T in the
\ adder.  If a carry is generated, indicating that T is big enought
\ to subtract the divisor, The sum is accepted into T, and then T-A
\ combination is shifted left by one bit.  The most significant bit
\ in A is shifted into T(0), and Carry is shifted into A(0).
\ If the adder does not generate a carry, the subtraction will not
\ be done.  The T-A combination is shifted left by one bit, and
\ a 0 is shifted into A(0).

\ The above divide step DIV instructions is repeated 25 times to
\ generate the proper quotient in A.  The remainder is in T, if it
\ is shifted right by one bit.

\ The only restriction in this division procedure is that the divisor
\ and the divident must be positive.  It cannot handle negative
\ divisor or negative divident.  This is not a serious limitation
\ because the special word M/MOD does signed division by first
\ convert both divisor and divident to postive numbers for division
```

```
\ operations, and then place appropriate signs in front of quotient
\ and remainder.

\ UM/MOD, /MOD, /, and MOD all assume that divisors and dividents
\ are positive.  In the eForth system, this is not a problem.
\ Nevertheless, users must be aware of this limitation when writing
\ code which must handle negative numbers.
CRR .( Divide ) CRR
CODE UM/MOD ( ud u -- ur uq )
   com 1 ldi add sta
   push lda push sta
   pop pop
   skip
CODE /MOD ( n n -- r q )
   com 1 ldi add push
   sta pop 0 ldi
   then
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div 1 ldi xor shr
   push drop pop lda
   ret
CODE MOD ( n n -- r )
   /MOD
   drop ret
CODE / ( n n -- q )
   /MOD
   push drop pop ret
:: M/MOD ( d n -- r q ) \ floored
  DUP 0<  DUP >R
  IF NEGATE >R DNEGATE R>
  THEN >R DUP 0< IF R@ + THEN R> UM/MOD R>
  IF SWAP NEGATE SWAP THEN ;;
```

8.4 Multiplication Words

```
 \ UM* multiplies two unsigned 24-bit integers and produces a
 \ 48-bit product.  The multiplier is placed in A register, and
 \ the multiplicant is placed on the data stack below T.  T is
 \ cleared to zero.  The MUL machine instruction looks at A(0)
\ bit.  If it is a one, the multiplicant is added to T, and
\ the T-A combination is shifted to the right by one bit.
\ Carry us shifted into T(23).  It A(0) is a zero, the multiplicant
\ is not added.  The T-A combination is shifted to the right, and
\ a zero is shifted into T(23).
\ After the MUL instruction is repeated 24 times, a 48-bit product
\ is produced in the T-A combination.  T has the more significant
\ half and A has the less significant half of the product.

\ Both UM* and * do the unsigned multiplication.  M* does signed
\ multiplication.  For correctness, * should call M* to do the
\ multiplicant.  However, here * calls UM* for speed.  You should
```

```
\ be aware of this property in your applications.  As the eForth
\ system only does unsigned multiplications, it is not a problem.
CRR .( Multiply ) CRR
CODE UM* ( u u -- ud )
   sta 0 ldi
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   push drop lda pop
   ret
:: * ( n n -- n ) UM* DROP ;;
:: M* ( n n -- d )
  2DUP XOR 0< >R  ABS SWAP ABS UM*  R> IF DNEGATE THEN ;;
:: */MOD ( n n n -- r q ) >R M* R> M/MOD -;'
:: */ ( n n n -- q ) */MOD SWAP DROP ;;


8.5 Memory Access Words
8.6
\ >CHAR filters out non-printable characters for TYPE.
\ It thus ensures that TYPEing a non-printable character
\ will not choke the printer.
CRR .( Bits & Bytes ) CRR
:: >CHAR ( c -- c )
  $7F LIT AND DUP $7F LIT BL WITHIN
  IF DROP ( CHAR _ ) $5F LIT THEN ;;

CRR .( Memory access ) CRR
:: HERE ( -- a ) CP @ ;;
:: PAD ( -- a ) CP @ 50 LIT + ;;
:: TIB ( -- a ) 'TIB @ ;;

CRR
:: @EXECUTE ( a -- ) @ ?DUP IF EXECUTE THEN ;;
:: CMOVE ( b b u -- )
  FOR AFT >R DUP @ R@ ! 1+ R> 1+ THEN NEXT 2DROP ;;
:: FILL ( b u c -- )
  SWAP FOR SWAP AFT 2DUP ! 1+ THEN NEXT 2DROP ;;


8.6    String Packing and Unpacking Words

 \ PACK$ packs the string at b with length u into memory located
 \ at a, three bytes to a 24-bit program word.  It calls B> to
 \ do the packing.  This packing function greatly reduces the
\ total size of the P24 code image.  The packing also speeds
\ up the dictionary searches because three bytes are compared
\ at once.  The system scratch variable TMP is used to store
\ the byte count which directs the bytes to their proper
\ location.  After the byte string is fully packed, the last
\ packed program word is left justified and empty slots are
\ filled with NUL bytes.
:: PACK$ ( b u a -- a ) \ null fill
  dup push
```

```
  1 ldi tmp sta st
  sta dup push st
  lda pop
  FOR AFT ( b a )
    B>
    tmp sta ld
    IF ld 1 ldi xor
      IF dup dup xor st
         1 ldi add
      ELSE 2 ldi st
      THEN
    ELSE 1 ldi st
    THEN
  THEN NEXT
  tmp sta ld
  IF ld 2 ldi xor
     IF sta ld
        shl shl shl shl
        shl shl shl shl
       st lda
     THEN
     sta ld
     shl shl shl shl
     shl shl shl shl
     st lda
  THEN
  drop drop pop
  ;;

\ UNPACK$ unpacks a packed string at a into a counted byte string
\ at b.  It calls >B to unpack a 24-bit word into three bytes.
\ It allows names of words to be printed, and in-line packed strings
\ to be accessed as byte strings.
:: UNPACK$ ( a b -- b )
  DUP >R ( save b )
  >B $1F LIT AND 3 LIT /
  FOR AFT
    >B DROP
  THEN NEXT
  2DROP R>
  ;;
```

8.6 Number Output Words

```
 \ All numbers in P24 are stored internally as 24-bit binary patterns.
 \ To make the numbers visual to the user, they are converted to
 \ strings of digits to be printed.  A number is converted one digit
 \ at a time.  It is divided by the value stored in BASE, and the
\ remainder is converted to a digit by DIGIT.  The quotient is
\ divided further by BASE to build a complete numeric string
\ suitable for printing.  The output numeric string is built
\ backward below the memory buffer at PAD, using HLD as the pointer
\ moving backward.  Additional formating characters can be inserted
\ into the output string by HOLD.

\ This numeric output mechanism is extremely flexible and can produce
```

```
\ numbers in a wide variety of formats for tables and arrays.  It also
\ allows the user to display numbers in any reasonable base, like
\ decimal, hexidecimal, octal, and binary, among other non-conventional
\ bases.
CRR .( Numeric Output ) CRR \ single precision
:: DIGIT ( u -- c )
  9 LIT OVER < 7 LIT AND +
  ( CHAR 0 ) 30 LIT + ;;
:: EXTRACT ( n base -- n c )
  0 LIT SWAP UM/MOD SWAP DIGIT -;'
:: <# ( -- ) PAD HLD ! ;;
:: HOLD ( c -- ) HLD @ 1- DUP HLD ! ! ;;
:: # ( u -- u ) BASE @ EXTRACT HOLD -;'
:: #S ( u -- 0 ) BEGIN # DUP WHILE REPEAT ;;
CRR
:: SIGN ( n -- ) 0< IF ( CHAR - ) 2D LIT HOLD THEN ;;
:: #> ( w -- b u ) DROP HLD @ PAD OVER - ;;
:: str ( n -- b u ) DUP >R ABS <# #S R> SIGN #> -;'
:: HEX ( -- ) 10 LIT BASE ! ;;
:: DECIMAL ( -- ) 0A LIT BASE ! ;;
```

8.7 Number Input Words

```
 \ Numbers are entered into P24 as strings of digits, delimited by
 \ spaces and other white characters like CR, TAB, NUL, etc.
 \ Numeric strings are converted to internal binary form by
 \ multiply the digits, most significant digit first, by the value
\ in BASE and accumulate the product until the digits are exhausted.

\ NUMBER? does the conversion.  It allows a leading $ to
\ indicate that the numeric string is in hexidecimal.  It also
\ allows a leading - sign for negative numbers.
CRR .( Numeric Input ) CRR \ single precision
:: DIGIT? ( c base -- u t )
  >R ( CHAR 0 ) 30 LIT - 9 LIT OVER <
  IF 7 LIT - DUP 0A LIT  < OR THEN DUP R> U< -;'
:: NUMBER? ( a -- n T | a F )
  BASE @ >R  0 LIT OVER COUNT ( a 0 b n)
  OVER @ ( CHAR $ ) 24 LIT =
  IF HEX SWAP 1+ SWAP 1- THEN ( a 0 b' n')
  OVER @ ( CHAR - ) 2D LIT = >R ( a 0 b n)
  SWAP R@ - SWAP R@ + ( a 0 b" n") ?DUP
  IF 1- ( a 0 b n)
    FOR DUP >R @ BASE @ DIGIT?
      WHILE SWAP BASE @ * +  R> 1+
    NEXT DROP R@ ( b ?sign) IF NEGATE THEN SWAP
      ELSE R> R> ( b index) 2DROP ( digit number) 2DROP 0 LIT
      THEN DUP
  THEN R> ( n ?sign) 2DROP R> BASE ! ;;

\ This is the set of words displaying characters to the output
\ device.
\ DO$ is an internal system word which unpacks a packed string compiled
\ in-line with program words.  It digs up the starting address of the
\ packed string on the return stack, unpacks the string to location a,
\ and then move the return address passing the packed string.  Then,
```

```
\ the execution can continue, skipping the packed string in-line.

\ $"| is compiled before a packed string.  It unpacks the string and
\ returns the address of the TEXT buffer where the unpack string is
\ stored.
\ ."| is also compiled before a packed string.  It unpacks the string
\ and displays it on the output device.
CRR .( Basic I/O ) CRR
:: SPACE ( -- ) BL EMIT -;'
:: CHARS ( +n c -- )
  SWAP 0 LIT MAX
  FOR AFT DUP EMIT THEN NEXT DROP ;;
:: SPACES ( +n -- ) BL CHARS -;'
:: TYPE ( b u -- )
  FOR AFT DUP @ >CHAR EMIT 1+
  THEN NEXT DROP ;;
:: CR ( -- ) ( =Cr )
  0A LIT 0D LIT EMIT EMIT -;'
:: do$ ( -- a )
  R> R@ TEXT UNPACK$
  R@ R> @ $3FFFFF LIT AND $30000 LIT / 1+ +
  >R SWAP >R ;;

CRR
:: $"| ( -- a ) do$ -;'
:: ."| ( -- ) do$ COUNT TYPE -;'
::  .R ( n +n -- )
  >R str      R> OVER - SPACES TYPE -;'
:: U.R ( u +n -- )
  >R <# #S #> R> OVER - SPACES TYPE -;'
:: U. ( u -- ) <# #S #> SPACE TYPE -;'
::  . ( n -- )
  BASE @ 0A LIT  XOR
  IF U. EXIT THEN str SPACE TYPE -;'
:: ? ( a -- ) @ . -;'

8.7   Word Parser
 \ TOKEN parses out the next word in the input stream, delimited by
 \ spaces.  The word is packed and placed on the top of the dictionary,
 \ so that it can be used to do dictionary searches, and becomes the
\ name field if the word just happed to be the name of a new
\ definition.

\ PARSE allows the user to specify the delimiting character to parse
\ out the next word in the input stream.  It calls 'parse' to do the
\ dirty work.

\ 'parse' scans the input stream and skips the leading blanks if
\ SPACE is the delimiting character.  The parsed word starts with
\ the next non-delimiting character and is terminated by the next
\ delimiting character.  It returns b the beginning address of the
\ parsed word, u the length of the remaining characters in the input
\ stream, and delta the length of the parsed word.  It is a very
\ long word with many nested and interlaced structures.  It is a
\ challenge even to the very experienced Forth programmers.
CRR .( Parsing ) CRR
:: (parse) ( b u c -- b u delta ; <string> )
```

```
  tmp ! OVER >R DUP \ b u u
  IF 1- tmp @ BL =
    IF              \ b u' \ 'skip'
      FOR BL OVER @ - 0< NOT
        WHILE 1+
      NEXT ( b) R> DROP 0 LIT DUP EXIT \ all delim
        THEN  R>
    THEN OVER SWAP  \ b' b' u' \ 'scan'
    FOR tmp @ OVER @ -  tmp @ BL =
      IF 0< THEN WHILE 1+
    NEXT DUP >R
      ELSE R> DROP DUP 1+ >R
      THEN OVER -  R>  R> - EXIT
  THEN ( b u) OVER R> - ;;
:: PARSE ( c -- b u ; <string> )
  >R  TIB >IN @ +
  #TIB @ >IN @ -
  R> (parse) >IN +! ;;
:: TOKEN ( -- a ;; <string> )
  BL PARSE 1F LIT MIN 2DUP
  DUP TEXT ! TEXT 1+ SWAP CMOVE
  HERE 1+ PACK$ -;'
:: WORD ( c -- a ; <string> )
  PARSE HERE 1+ PACK$ -;'
```

8.8 Dictionary Search

```
 \ 'find' follows the linked list in the dictionary, and compares
 \ the names of each compiled word with the packed string stored
 \ at a.  va points to the starting name field of the dictionary.
 \ If a match is found, it returns the execution address (code
\ field address) and the name field address of the matching word
\ in the dictionary.  If it failed to find a match, it returns
\ the address of the packed string and a 0 for a false flag.

\ 'find' runs through the dictionary very quickly, because it
\ compares the length and the first two characters in the names.
\ Most Forth words are unique in these three characters.  For
\ words with the same lengths and identical first two characters,
\ 'find' calls SAME? to determine whether the remaining characters
\  of the packed strings match.
\ NAME> converts a name field address na to a code field address xt.
\
CRR .( Dictionary Search ) CRR
:: NAME> ( na -- xt )
  DUP @ $3FFFFF LIT AND
  $30000 LIT / + 1+ ;;
:: SAME? ( a a u -- a a f \ -0+ )
  $30000 LIT /
  FOR AFT OVER R@ + @
    OVER R@ + @ - ?DUP
    IF R> DROP EXIT THEN
  THEN NEXT
  0 LIT ;;
:: find ( a va -- xt na | a F )
  SWAP         \ va a
```

```
  DUP @ tmp !  \ va a  \ get cell count
  DUP @ >R     \ va a  \ count
  1+ SWAP      \ a' va
  BEGIN @ DUP  \ a' na na
    IF DUP @ $3FFFFF LIT AND
      R@ XOR \ ignore lexicon bits
      IF 1+ -1 LIT
      ELSE 1+ tmp @ SAME?
      THEN
    ELSE R> DROP SWAP 1- SWAP EXIT \ a F
    THEN
  WHILE 1- 1-  \ a' la
  REPEAT R> DROP SWAP DROP
  1- DUP NAME> SWAP ;;
:: NAME? ( a -- xt na | a F )
  CONTEXT find -;'
```

8.9 Terminal Input

```
 \ ^H processes the Back Space encountered in the input stream.  It
 \ backs up the character pointer and erased the character preceeding

\ the Back Sapce.
\ TAP echoes an input character and deposit it into the terminal
\ input buffer.
\ kTAP Detects a Carriage Return to terminate the input stream.  It
\ also calls ^H to process a Back Space, and TAP to process ordinary
\ characters.
\ These words allows the interpreter to handle a human user on the
\ terminal smoothly, and friendly.
CRR .( Terminal ) CRR
:: ^H ( b b b -- b b b ) \ backspace
  >R OVER R> SWAP OVER XOR
  IF ( =BkSp ) 8 LIT EMIT
     1-         BL EMIT \ distructive
     ( =BkSp ) 8 LIT EMIT \ backspace
  THEN ;;
:: TAP ( bot eot cur c -- bot eot cur )
  DUP EMIT OVER ! 1+ ;;
:: kTAP ( bot eot cur c -- bot eot cur )
  DUP ( =Cr ) 0D LIT XOR
  IF ( =BkSp ) 8 LIT XOR
    IF BL TAP ELSE ^H THEN
    EXIT
  THEN DROP SWAP DROP DUP ;;

\ QUERY accepts a line of characters typed in by the user and
\ put them in the terminal input buffer for interpreting or
\ compiling.  The line is terminated at the 80th input
\ character or a Carriage Return.
\ 'accept' waits for input characters and place them in the
\ terminal input buffer at b with length u.  It returns the
\ same buffer address b with the length of the character string
\ actually received.
\ EXPECT receives the input stream and stores the length in the
\ variable SPAN.
```

```
CRR
:: accept ( b u -- b u )
  OVER + OVER
  BEGIN 2DUP XOR
  WHILE  KEY  DUP BL -  5F LIT U<
    IF TAP ELSE kTAP THEN
  REPEAT DROP  OVER - ;;
:: EXPECT ( b u -- ) accept SPAN ! DROP ;;
:: QUERY ( -- )
  TIB 50 LIT accept #TIB !
  DROP 0 LIT >IN ! ;;
```

8.10 Error Handling Words

```
 \ ABORT actually executes QUIT, which is defined much later.
 \ Here it is defined as a vectored execution word which gets
 \ the execution address in the system variable 'ABORT.  This
 \ mechanism also gives the user some flexibility in how the
\ application should handle an error condition.

\ abort" aborts after a warning message is displayed.

\ ERROR prints the character string store in the TEXT buffer
\ before aborting.  The TEXT buffer contains the word just
\ parsed out of the input stream.  This is the word which
\ the interpreter/compiler fail to recognize.  The natural
\ error message is this word followed by a ? mark.
CRR .( Error handling ) CRR
:: ABORT ( -- ) 'ABORT @EXECUTE ;;
:: abort" ( f -- )
  IF do$ COUNT TYPE ABORT THEN do$ DROP ;;
:: ERROR ( a -- )
  SPACE TEXT COUNT TYPE
  $3F LIT EMIT CR ABORT
```

8.11 Text Interpreter

```
 \ $INTERPRET interprets the word just parsed out of the input
 \ stream.  It searches the dictionary for this word.  If a match
\ is found, executes it, unless the word is marked as a
\ compile-only word.  It a match is now found in the dictionary,
\ convert the word into a number.  If successful, the number is
\ left on the data stack.  If not successful, exit with ERROR.
CRR .( Interpret ) CRR
:: $INTERPRET ( a -- )
  NAME?  ?DUP
  IF @ 400000 LIT AND
    ABORT" $LIT compile only" EXECUTE EXIT
  THEN DROP TEXT NUMBER?
  IF EXIT THEN ERROR
:: [ ( -- )
  forth' $INTERPRET  >body forth@ LIT 'EVAL !
  ;; IMMEDIATE
:: .OK ( -- )
  forth' $INTERPRET >body forth@ LIT 'EVAL @ =
```

```
  IF ."| $LIT  OK" CR
  THEN ;;
:: EVAL ( -- )
  BEGIN TOKEN DUP @
  WHILE 'EVAL @EXECUTE \ ?STACK
  REPEAT DROP .OK -;'

CRR .( Shell ) CRR
:: QUIT ( -- )
  ( =TIB) $730 LIT 'TIB !
   [ BEGIN QUERY EVAL AGAIN

CRR .( Compiler Primitives ) CRR
:: ' ( -- xt )
  TOKEN NAME? IF EXIT THEN
  ERROR
:: ALLOT ( n -- ) CP +! ;;
:: , ( w -- ) HERE DUP 1+ CP ! ! ;;
:: [COMPILE] ( -- ; <string> )
  '  $100000 LIT OR , -;' IMMEDIATE

CRR
:: COMPILE ( -- ) R> DUP @ , 1+ >R ;;
:: LITERAL $29E79E LIT , ,
  -;' IMMEDIATE
:: $," ( -- ) ( CHAR " )
  22 LIT WORD @ 1+ ALLOT -;'

CRR .( Name Compiler ) CRR
:: ?UNIQUE ( a -- a )
  DUP NAME?
  IF TEXT COUNT TYPE ."| $LIT  reDef "
  THEN DROP ;;
:: $,n ( a -- )
  DUP @
  IF ?UNIQUE
    ( na) DUP DUP NAME> CP !
    ( na) DUP LAST ! \ for OVERT
    ( na) 1-
    ( la) CONTEXT @ SWAP ! EXIT
  THEN ERROR
```

8.12 Compiler

```
 \ $COMPILE compiles the word just parsed out of the input
\ stream.  It searches the dictionary for this word.  If a match
\ is found, compiles it, unless the word is marked as an
\ immediate word.  An immediate word is executed by the compiler.
\ If a match is not found in the dictionary, convert the word into
\ a number.  If successful, the number is compile as a literal.
\ If not successful, exit with ERROR.
CRR .( FORTH Compiler ) CRR
:: $COMPILE ( a -- )
  NAME? ?DUP
  IF @ $800000 LIT AND
    IF EXECUTE
```

```
    ELSE $3FFFF LIT AND $100000 LIT OR ,
    THEN EXIT
  THEN DROP TEXT NUMBER?
  IF LITERAL EXIT
  THEN ERROR
:: OVERT ( -- ) LAST @ CONTEXT ! ;;
:: ; ( -- )
  $5E79E LIT , [ OVERT -;' IMMEDIATE
:: ] ( -- )
  forth' $COMPILE >body forth@ LIT 'EVAL ! ;;
:: : ( -- ; <string> )
  TOKEN $,n ] -;'
```

8.13 Debugging Tools

```
 CRR .( Tools ) CRR
 :: dm+ ( b u -- b )
   OVER 7 LIT U.R SPACE
   FOR AFT DUP @ 7 LIT U.R 1+
   THEN NEXT ;;
 :: DUMP ( b u -- )
   BASE @ >R HEX  8 LIT /
   FOR AFT CR 8 LIT 2DUP dm+
   THEN NEXT DROP R> BASE ! ;;

 CRR
:: >NAME ( xt -- na | F )
  CONTEXT
  BEGIN @ DUP
  WHILE 2DUP NAME> XOR
    IF 1-
    ELSE SWAP DROP EXIT
    THEN
  REPEAT SWAP DROP ;;
:: .ID ( a -- )
  TEXT UNPACK$
  COUNT $01F LIT AND TYPE SPACE -;'

CRR
:: SEE ( -- ; <string> )
  ' CR
  BEGIN
    20 LIT FOR
      DUP @ DUP FC0000 LIT AND
      DUP
      IF 100000 LIT XOR THEN
      IF U. SPACE
      ELSE 3FFFF LIT AND >NAME
        ?DUP IF .ID THEN
      THEN 1+
    NEXT KEY 0D LIT =  \ can't use ESC on terminal
  UNTIL DROP ;;
:: WORDS ( -- )
  CR CONTEXT
  BEGIN @ ?DUP
  WHILE DUP SPACE .ID 1-
```

```
  REPEAT ;;
CODE .S ( dump all 17 stack items )
  PAD sta stp
  stp stp stp stp
  stp stp stp stp
  stp stp stp stp
  stp stp stp stp
  DROP PAD $10 LIT
  FOR DUP ? 1+ NEXT
  DROP PAD @ CR -;'
```

8.14 Start Up

```
 CRR .( Hardware reset ) CRR
 ::   DIAGNOSE     ( - )
      $65 LIT
 \ 'F'  prove UM+ 0<         \ carry, TRUE, FALSE
      0 LIT 0< -2 LIT 0<     \ 0 FFFF
      UM+ DROP               \ FFFF ( -1)
      3 LIT UM+ UM+ DROP     \ 3
     $43 LIT UM+ DROP       \ 'F'
\ 'o' logic: XOR AND OR
     $4F LIT $6F LIT XOR    \ 20h
     $F0 LIT AND
     $4F LIT OR
\ 'r' stack: DUP OVER SWAP DROP
      8 LIT 6 LIT SWAP
      OVER XOR 3 LIT AND AND
      $70 LIT UM+ DROP       \ 'r'
\ 't'-- prove BRANCH ?BRANCH
      0 LIT IF $3F LIT THEN
      -1 LIT IF $74 LIT ELSE $21 LIT THEN
\ 'h' -- @ ! test memeory address
      $68 LIT $700 LIT !
      $700 LIT @
\ 'M' -- prove >R R> R@
      $4D LIT >R R@ R> AND
\ 'l'  -- prove 'next' can run
      1 LIT $6A LIT FOR 1 LIT UM+ DROP NEXT
      ;;

CRR
:: COLD ( -- )
   diagnose
   CR ."| $LIT P24 v"
   66 LIT <# # # ( CHAR . ) 2E LIT HOLD # #> TYPE
   CR QUIT
```

8.15 Control Structure Words

```
 CRR .( Structures ) CRR
 :: IF ( -- A )    HERE $80000 LIT , -;' IMMEDIATE
:: FOR ( -- a )   $71E79E LIT , HERE -;' IMMEDIATE
:: BEGIN ( -- a ) HERE -;' IMMEDIATE
:: AHEAD ( -- A ) HERE 0 LIT , -;' IMMEDIATE
```

```
CRR
:: AGAIN ( a -- ) , -;' IMMEDIATE
:: THEN ( A -- )  HERE SWAP +! ;; IMMEDIATE
:: NEXT ( a -- )  COMPILE doNEXT , -;' IMMEDIATE
:: UNTIL ( a -- ) $80000 LIT + , -;' IMMEDIATE
CRR
:: REPEAT ( A a -- ) AGAIN THEN -;' IMMEDIATE
:: AFT ( a -- a A )  DROP AHEAD BEGIN SWAP ;; IMMEDIATE
:: ELSE ( A -- A )   AHEAD SWAP THEN -;' IMMEDIATE
:: WHILE ( a -- A a ) IF SWAP ;; IMMEDIATE
```

8.16 Redefine Macro Words

```
 CRR .( macro words ) CRR

 CODE EXIT pop drop ret
 CODE EXECUTE push ret
 CODE ! sta st ret
 CODE @ sta ld ret

 CRR
 CODE R> pop sta pop lda push ret
 CODE R@ pop sta pop dup push lda push ret
CODE >R sta pop push lda ret

CRR
CODE SWAP
   push sta pop lda ret
CODE OVER
   push dup sta pop
   lda ret
CODE 2DROP
   drop drop ret

CRR
CODE +  add ret
CODE NOT com ret
CODE  NEGATE
   com 1 ldi add ret
CODE 1-
   -1 ldi add ret
CODE 1+
   1 ldi add ret

CRR
CODE BL
   20 ldi ret
CODE +!
   sta ld add st
   ret
CODE -
   com add 1 ldi add
   ret

CRR
CODE DUP dup ret
```

```
CODE DROP drop ret
CODE AND and ret
CODE XOR xor ret
CODE COM com ret
```

8.17 Final System Words

```
 CRR
 :: ABORT" ( -- ; <string> ) COMPILE abort" $," ;; IMMEDIATE
 :: $" ( -- ; <string> ) COMPILE $"| $," ;; IMMEDIATE
 :: ." ( -- ; <string> ) COMPILE ."| $," ;; IMMEDIATE
 :: CODE ( -- ; <string> ) TOKEN $,n OVERT -;'
 :: CREATE ( -- ; <string> ) CODE doVAR ;;
:: VARIABLE ( -- ; <string> ) CREATE 0 LIT , -;'

CRR
:: .( ( -- ) 29 LIT PARSE TYPE -;' IMMEDIATE
:: \ ( -- ) #TIB @ >IN ! ;; IMMEDIATE
:: ( 29 LIT PARSE 2DROP ;; IMMEDIATE
:: IMMEDIATE $800000 LIT LAST @ @ OR LAST @ ! ;;

CRR
```

Chapter 9. P24 Cycle-Based Simulator

An accurate and fast logic simulator is extremely valuable in the design and testing of a new CPU. It is also very useful in separating the hardware design from software development, so that hardware and software can be developed simultaneously. This P24 simulator served well in the process of building the P24 CPU and the eForth system which proves that the hardware-software system works correctly.

This P24 simulator faithfully replicates the logic behavior of the P24 CPU on a cycle by cycle basis. As the P24 CPU is composed of a set of registers and two stacks, and the registers and stacks acquire new contents only on the rising edge of the master clock, it is very simple to emulate this behavior.

Each register and each level in the two stacks are represented by two 32-bit words. The first word contains the current value of the register, and the second word contains the value to be latched into the register on the next rising edge of the master clock. This simple mechanism very conveniently replicates the behavior of a synchronously clocked flip-flops, and forms the basis of the P24 simulator.

Two large arrays are opened to host these 32-bit word sets. The FROM array contains the current values of all the registers and all the stack levels, and the TO array contains the new values to be stored into the registers and stacks on the next clock. The rising edge of the clock forces the entire TO array to be copied into the FROM array, and these is functionally one machine cycle. The multiplexers in P24 are replaced by Forth words which perform the logic functions and update values in the TO array.

The Slot Machine, which fetches a program word from memory, and sequences the execution of the four machine instructions in this word, is simulated by a 32-bit counter. The less significant 3 bit in this counter steps through slots 0 to 4 in 5 clock cycles. Then this 3-bit field is cleared to zero and the upper 29-bit counter is incremented. Therefore, the upper 29-bit field in the counter gives an accurate program word count.

The most interesting feature of this P24 simulator is that it can vector the KEY and EMIT function to the equivalent Windows function, so that the simulator can actually run P24 eForth interactively, and produces the identical output as the actual P24 computer would do on a terminal. Now that it was proven that the simulator runs identically to the actual P24 computer, the simulator can be used for software development, in place of a real P24 computer.

The simulator code is in SIM24.F. It must be loaded after META24.F, which builds a P24 eForth system in the array RAM. The simulator reads program words from the RAM memory and execute the instructions contained in these program words.

9.1 The Registers and the Stacks

```
\ Put all simulator words in SIM24 vocabulary.  They are thus distinguished
\ from words of the same names in the FORTH and ASM24 vocabularies.
ONLY FORTH ALSO SIM24 DEFINITIONS

\ Stacks are limited to 16 levels, and act like circular buffers
\ Program are limited to 32KB, or 8KW, the size of RAM array
\ CLOCK has a 29-bit program word count filed and a 3-bit SLOT field
\ The SLOT field sequences program word fetch and execution of up to
\ four instructions in the program word.
\ (REGISTER) a pointer to switch between FROM array and TO array
\ BREAK breakpoint address
\ REGISTER the base address of either FROM or TO array
\ FROM forces accessing registers in the FROM array
\ TO forces accessing registers in the TO array
DECIMAL

15 CONSTANT LIMIT       ( stack depth )
$1FFF CONSTANT RANGE   ( size of memory array )
VARIABLE CLOCK          ( slot is in the last 3 bits )
VARIABLE (REGISTER)     ( where registers and stacks are )
VARIABLE BREAK          ( address of break point )

( On the rising edge of clock, copy TO array to FROM array. )
: REGISTER  (REGISTER) @ ;
: FROM  PAD (REGISTER) ! ;
: TO    PAD $180 + (REGISTER) ! ;

\ P   program counter
\ T   accumulator
\ R   top of return stack
\ A   address register
\ I   instruction latch
\ I0-4      machine instruction storage
\ RP  return stack pointer
\ SP  data stack pointer
```

```
\ RSTACK    returns address of top of return stack
\ SSTACK    returns address of top of data stack
: P     REGISTER ;
: T     REGISTER 4 + ;
: R     REGISTER 8 + ;
: A     REGISTER 12 + ;
: I     REGISTER 24 + ;
: I0    REGISTER 28 + ;
: I1    REGISTER 29 + ;
: I2    REGISTER 30 + ;
: I3    REGISTER 31 + ;
: I4    REGISTER 32 + ;
: RP    REGISTER 33 + ;
: SP    REGISTER 34 + ;
: RSTACK  RP C@ LIMIT AND CELLS REGISTER + $40 + ;
: SSTACK  SP C@ LIMIT AND CELLS REGISTER + $80 + ;
```

9.2 Machine Cycles

Here are a set of words supporting the simulator.

```
\ CYCLE     simulate rising edge of master clock. Copy TO array to FROM array.
\ NEXT      forces fetching the next program word
\ RPUSH     push a integer d on return stack
\ RPOPP     pop return stack and leave the integer on the Forth stack
\ SPUSH     push a integer d on data stack
\ SPOPP     pop data stack and leave the integer on the Forth stack
\ CONTINUE  fetch next program word and deposit the 4 machine instructions
\           in I1-I4
: CYCLE TO P FROM P $180 CMOVE 1 CLOCK +! ;
: NEXT  CLOCK @ 7 OR CLOCK ! ;
: RPUSH ( d -- , push d on return stack )
       FROM R @ RP C@ 1 + LIMIT AND TO RP C! RSTACK ! R ! ;
: RPOPP ( -- d , pop d from return stack )
       FROM R @ RSTACK @ RP C@ 1 - LIMIT AND TO RP C! R ! ;
: SPUSH ( d -- , push d on data stack )
       FROM T @ SP C@ 1 + LIMIT AND TO SP C! SSTACK ! T ! ;
: SPOPP ( -- d , pop d from data stack )
       FROM T @ SSTACK @ SP C@ 1 - LIMIT AND TO SP C! T ! ;
: continue
       FROM P @ DUP 1+ TO RANGE AND P !
       RAM@ DUP I !
       64 /MOD SWAP I4 C!
       64 /MOD SWAP I3 C!
       64 /MOD SWAP I2 C!
       63 AND I1 C!
       ;
```

9.3 Machine Instructions

Machine instructions in the simulator take current values in the FROM registers and stacks and compute the desired new values and deposit them in the TO registers. Their functions in the real P24 CPU are performed by multiplexers and logic circuits. Nevertheless, these instructions truthfully describe the behavior of all the machine instructions.

```
 : jmp   FROM I @ RANGE AND TO P ! NEXT ;
 : call  FROM P @ RPUSH jmp ;
: ret   RPOPP TO RANGE AND P !
        NEXT ;
: jz    SPOPP $FFFFFF AND IF NEXT EXIT THEN
        jmp ;
: jnc   FROM T @ $1000000 AND IF NEXT EXIT THEN
        jmp ;
: ld    FROM A @ RANGE AND RAM@ SPUSH ;
: ldp   ld
        FROM A @ 1+ TO A ! ;
: ldi   FROM P @ 1+ RANGE AND TO P !
        FROM P @ RANGE AND RAM@ SPUSH ;
: st    SPOPP FROM A @ RANGE AND RAM! ;
: stp   st
        FROM A @ 1+ TO A ! ;
: com   FROM T @ $FFFFFF AND $FFFFFF XOR TO T ! ;
: shr   FROM T @ 2/ $FFFFFF AND TO T ! ;
: shl   FROM T @ 2* $1FFFFFF AND TO T ! ;
: mul   FROM A @ 1 AND
        IF SSTACK @ T @ + $1FFFFFF AND
        ELSE T @ THEN
        DUP 1 AND >R 2/ TO T !
        FROM A @ $FFFFFF AND 2/ R> IF $800000 OR THEN TO A ! ;
: andd  SPOPP TO T @ AND $FFFFFF AND T ! ;
: xorr  SPOPP TO T @ XOR $FFFFFF AND T ! ;
: div   FROM SSTACK @ $FFFFFF AND T @ $FFFFFF AND +
        DUP $1000000 AND DUP >R
        IF ELSE DROP T @ THEN $FFFFFF AND
        2* ( diff) A @ $800000 AND IF 1+ THEN TO T !
        FROM A @ 2* $FFFFFF AND R> IF 1+ THEN TO A ! ;
: add   SPOPP $FFFFFF AND TO T @ $FFFFFF AND + TO T ! ;
: popr  RPOPP SPUSH ;
: pushs FROM T @ SPUSH ;
: lda   FROM A @ SPUSH ;
: pushr SPOPP RPUSH ;
: sta   SPOPP TO A ! ;
: pops  SPOPP DROP ;
: nop   NEXT ;

\ GET forces the simulator to get a key from the keyboard under Windows OS
\ PUT forces the simulator to send a character to the display window
: get   KEY DUP $1B = ABORT" done"
        SPUSH ret ;
: put   SPOPP $7F AND EMIT ret ;

\ EXECUTE decodes a machine instruction and performs the required operations.
HEX

: execute ( code -- )
        DUP  0 = IF DROP jmp  EXIT THEN
        DUP  1 = IF DROP ret  EXIT THEN
        DUP  2 = IF DROP jz   EXIT THEN
        DUP  3 = IF DROP jnc  EXIT THEN
        DUP  4 = IF DROP call EXIT THEN
```

```
        DUP  6 = IF DROP get  EXIT THEN
        DUP  7 = IF DROP put  EXIT THEN
        DUP  9 = IF DROP ldp  EXIT THEN
        DUP 0B = IF DROP ld   EXIT THEN
        DUP 0A = IF DROP ldi  EXIT THEN
        DUP 0D = IF DROP stp  EXIT THEN
        DUP 0F = IF DROP st   EXIT THEN
        DUP 10 = IF DROP com  EXIT THEN
        DUP 11 = IF DROP shl  EXIT THEN
        DUP 12 = IF DROP shr  EXIT THEN
        DUP 13 = IF DROP mul  EXIT THEN
        DUP 14 = IF DROP xorr EXIT THEN
        DUP 15 = IF DROP andd EXIT THEN
        DUP 16 = IF DROP div  EXIT THEN
        DUP 17 = IF DROP add  EXIT THEN
        DUP 18 = IF DROP popr EXIT THEN
        DUP 19 = IF DROP lda  EXIT THEN
        DUP 1A = IF DROP pushs EXIT THEN
        DUP 1C = IF DROP pushr EXIT THEN
        DUP 1D = IF DROP sta  EXIT THEN
        DUP 1E = IF DROP nop  EXIT THEN
        DUP 1F = IF DROP pops EXIT THEN
        CR . ." illegal code" ABORT
        ;
```

9.4 Instruction Execution

```
\ .STACK displays the contents of a stack.
\ .SSTACK displays the contents of data stack
\ .RSTACK displays the contents of return stack.
\ .REGISTERS displays contents of all the relevant registers
\ S show all the registers and stack at this cycle
\ SYNC executes the current machine instruction using CLOCK to determine which
\      slot is being executed.
\ C runs one clock cycle and displays all the registers and stacks.
\ RESET clear both FROM and TO arrays, simulating the hardware reset.
\ G run and stop at the address given on the Forth stack.  This is a much more
\   efficient way to set breakpoints and then run till breakpoint is triggered.
\   It allows the user to execute a large portion of the program and stop only
\   on specified location.
\ PUSH push a new integer into the T register and push the data stack
\ POP discard the contents in T and pop data stack back into T.
 : .stack ( add # ) 0 ?DO DUP @ . 4 - LOOP DROP CR ;
 : .sstack ." S:" T @ . SSTACK SP C@ .stack ;
 : .rstack ." R:" R @ . RSTACK RP C@ .stack ;
 : .registers ."  P=" P @ . ."  I=" I @ .
         ." I1=" I1 C@ . ." I2=" I2 C@ .
        ." I3=" I3 C@ . ." I4=" I4 C@ . CR
        ." A=" A @ . CR ;
: S  CR ." CLOCK=" CLOCK @ . .registers
        .sstack .rstack ;

: sync  CLOCK @ 7 AND
        DUP 0 = IF   continue DROP EXIT THEN
        DUP 1 = IF   I1 C@ execute DROP EXIT
                THEN
```

```
        DUP 2 = IF   I2 C@ execute DROP EXIT
                THEN
        DUP 3 = IF   I3 C@ execute DROP EXIT
                THEN
        DUP 4 = IF   I4 C@ execute THEN
        DROP NEXT ;
: C     sync CYCLE S ;
: reset FROM P 300 ERASE 0 CLOCK ! ;
reset

: G     ( addr -- )
        CR ." Press any key to stop." CR
        BREAK !
        BEGIN sync P @ BREAK @ =
              IF CYCLE C EXIT
              ELSE CYCLE
              THEN
              KEY?
        UNTIL ;
: PUSH  ( d ) pushs TO T ! ;
: POP   pops ;
```

9.5 User Interface

This simulator has very simple text based user interface. The most used commands are C for single steps, RUN to continue stepping with any key and terminated by ESC. If the target address is know, then G is a convenient choice. P allows the user to start simulating at any address.

```
: D     P @ 1- FOUR FOUR ;
 : M     SHOW ;
 : RUN   CR ." Press ESC to stop." CR
         BEGIN C KEY 1B = UNTIL ;
 : P     RANGE AND DUP FROM P ! TO P ! ;

 : HELP  CR ." eM24 Simulator, copyright eMAST Technology, 2000"
        CR ." C: execute next cycle"
        CR ." S: show all registers"
        CR ." D: display next 8 words"
        CR ." addr M: display 128 words from addr"
        CR ." addr P: start execution at addr"
        CR ." addr G: run and stop at addr"
        CR ." RUN: execute, one key per cycle"
        CR ;
```

9.6 Simulating Running Forth System

The simulator is the most effective in debugging short sequences of program words to verify that the sequences are executed correctly. After the P24 machine instructions are verified, one can use the G command to execute a long stretch of program and break only at specific locations. This allows large segment of programs to be tested.

The simulator can run eForth system if KEY and EMIT are vectored to the keyboard input and screen display in Windows. This is accomplished by defining two new

machine instructions GET and PUT with the proper Windows interface. GET and PUT is then patched into KEY and EMIT in the target memory. Now, executing:

```
      800 G
```

will start the P24 eForth running, because 800 is a location it will never reach. In the meantime, the user can interact with the eForth in the simulator like using any other eForth system.

It is possible to build this simulator into a full P24 program development system by vectoring input streams from text files maintained under Windows. This, however, will have to wait in the next revision of the simulator.

```
 ( patch KEY and EMIT to run eForth interactively )
 180000 B7 RAM!
 1C0000 AA RAM!
```

9.7 Running P24 Simulator

From Windows, load Win32Forth by clicking its icon on the desktop, or run it in the Start/Programs/Win32Forth/Win32Forth. Win32Forth opens a window. Click File/OpenFile and navigate to the directory in which all the P24 files are stored. Select one of the files, say META24.F, and an WinView window is opened, displaying the META24.F file.

Go back to the Win32Forth window, and type:

```
      Fload meta24
```

You will see a list of names and compiled addresses scrolling on the screen. This list of names and addresses are very useful in running the simulator. You can interpret the addresses and determine which word is being executed, and you can select specific words to simulate.

You can inspect the compiled target image by typing:

```
      0 show
      show
      show ...
```

to dump the memory, 128 words at a time. SHOW will change the starting address so that you can use it to dump consecutive blocks of target memory without giving the address explicitly.

Load simulator file by typing:

```
      Fload sim24
```

Type HELP to see all the useful commands in the simulator. Then use C, RUN, G, P commands to step through programs you want to debug.

Type BRAM to dump the target memory in a form acceptable to the VHDL synthesizer in the Foundation FPGA development system. The eForth system can be synthesized with the P24 core, and run in XCV300 FPGA or its large cousins from Xilinx.

P24 system is still undergoing modifications and enhancements. Check with eMAST Technolgoy or Offete Enterpries for latest updates.