Chapter 14
68000 Assembly

JForth and 68000 Assembly Language

JForth offers an extensive set of tools for operating at the assembly language level. These include:

1. The JForth 68000 Reverse Polish Notation (RPN) Assembler
2. The JForth 68000 Motorola-Style (Forward-Parsing) Assembler
3. The JForth 68000 Disassembler.

This chapter includes documentation for each; some knowledge of 68000 assembly language topics is assumed; for additional technical information on the 68000, refer to a Motorola 680xx Programmers Reference Manual.

JForth Register Utilization

To write successful assembly language programs, a basic understanding of how JForth uses the CPU registers is necessary. This includes which registers are available for general use, and how to push numbers onto the data stack and pop them off.

7 CPU registers are available for general use by the programmer. These include D0, D1, D2, D3, D4, A0 and A1. These may be altered at will, but are not preserved across calls to other JForth words, so relevant registers should be saved and restored if you do so.

JForth requires the remaining 9 registers for itself; these are either unavailable for use by the programmer, or useable only in prescribed ways (such as pushing numbers onto the stack and popping them off).

The following chart describes the JForth CPU register utilization:

Register	JForth Name	Free to Use?	Used in JForth as:
D0	--	Yes	--
D1	--	Yes	--
D2	--	Yes	--
D3	--	Yes	--
D4	--	Yes	--
D5	ILOOP	No	F83 Loop Index #1
D6	JLOOP	No	F83 Loop Index #2
D7	TOS	No	Top of Stack
A0	TEMP0	Yes	--
A1	TEMP1	Yes	--
A2	LOC	No	Pointer to LOCAL stack frame
A3	+64K	No	Pointer to relative addr 64K
A4	ORG	No	Pointer to relative addr 0
A5	UP	No	Pointer to base of User Vars
A6	DSP	No	Pointer to 2nd data stack item
A7	RP	No	Return Stack/System Stack

Note that JForth provides alternate names for the reserved registers that are more descriptive of their specific functions. These are recognized by both assemblers, and may be optionally included in the output of the disassembler. See each respective section for additional information on register names.

JForth caches the topmost data stack item in the CPU register D7, also named TOS (Top-Of-Stack). The second item on the stack (as well as any others) is stored on an actual stack in memory and is pointed to by register A6, or DSP (Data-Stack-Pointer). Because of this arrangement, pushing a number on the data stack is a two-step process:

1. Move the contents of the TOS register out to the data stack, adjusting DSP value to reflect a new element.
2. Load the new number into TOS.

The process of popping a number off of the stack is even easier, requiring only one step:

1. Move what DSP is pointing to into the TOS register, adjusting DSP value to reflect 1 less element.

Examples of how to do this are given in each assembler section; for now, remember that these operations involve the TOS register (D7) and the DSP register (A6).

One other reserved register can be useful to the programmer, but is ONLY READ FROM, and NEVER CHANGED! This is register A4, also called the ORG register, and contains an absolute pointer to the beginning of the JForth image, the address that JForth considers 0, relative to itself. The value of ORG is often added or subtracted from addresses to convert between the 'real' Amiga ones (ABSOLUTE) and the JForth ones (RELATIVE), as in the words >ABS and >REL.

JForth 68000 Forth Style Assembler (RPN)

Compiling the RPN Assembler

Since the RPN Assembler is available as a MODULE, it is not usually required that the program be compiled. Nonetheless, the Reverse Polish assembler can be compiled into the resident dictionary by entering:

```
DETACHMODULE ASSEM
INCLUDE JF:ASM
```

RPN Assembler Usage

Here is an example of a simple RPN-assembled word.

```
CODE DOUBLE  ( n -- n*2 , Double top of stack.)
    TOS DN  TOS DN  ADD  ( Add TOS to itself.)
END-CODE
7 DOUBLE .
```

The word CODE activates the Reverse-Polish Assembler; subsequent input is interpreted in the ASSEMBLER context. One ADD instruction follows; source operand first, destination operand second, finally the opcode. The whole sequence ends with the END-CODE operator, which deactivates the ASSEMBLER after installing an RTS instruction as the final 68000 directive in this small program (ALL JForth words MUST end in an RTS). Please note that if an RTS is specifically included as the last instruction in the word being assembled, END-CODE will NOT append another one.

RPN Assembler Register Names

Since they conflict with hex numbers, the normal names for the registers (A0, D4, etc.) cannot be used by the RPN Assembler. Instead, a convention has been adopted whereby the RPN-acceptable name is formed as a single word, beginning with the NUMBER of the register, followed by DR for data registers and AR for address registers. For example, 0AR represents address register 0, 4DR is data register 4, and so on.

Additionally, all of the JForth functional names such as TOS and DSP are available.

Motorola Addressing Modes and RPN Assembler Equivalence

This table describes the various addressing modes available.

Forth address mode		Motorola example	description
DN	(REG ---)	d0	== data register direct
AN	(REG ---)	a1	== address register direct
A@	(REG ---)	(a1)	== address register indirect
A@+	(REG ---)	(a2)+	== A@ then inc reg by size
-A@	(REG ---)	-(a3)	== dec by size, then A@
AN+W	(AREG N---)	9(a1)	== address + word indirect
AN+R+B	(AREG REG BYTE---)	5(a2,d3)	== addr+reg+byte indirect
ABS.W	(N---)	1000	== absolute address word
ABS.L	(N.LSW N.MSW---)	2000	== absolute address long
PC+W	(W ---)	7(pc)	== pc + word
PC+R+B	(REG BYTE---)	9(pc.a2)	== pc + reg + byte
#	(N-OR-D---)	#4	== immediate data

Here are some examples of using the different addressing modes with a MOVE instruction, and the Motorola-style equivalent.

```
Source             Destination   Opcode      Motorola equiv.
0DR DN               0AR AN        MOVE    | MOVE.L D0,A0
0AR A@               1AR A@+       MOVE    | MOVE.L (A0),(A1)+
0AR -A@              1AR 50 AN+W   MOVE    | MOVE.L -(A0),50(A1)
1AR 0DR 30 AN+R+B    100 ABS.L     MOVE    | MOVE.L 30(A1,D0.L),100
2000 ABS.W           50 PC+N       MOVE    | MOVE.L 2000,50(PC)
34 #            0DR 100 PC+R+B     MOVE    | MOVE.L #34,100(PC,D0.L)
```

RPN Assembler Support Words.

A data-length may be specified with one of these 3 operators:

```
BYTE -- declares 8 bit data-size
WORD -- declares 16-bit data-size
LONG -- declares 32-bit data-size  ( default )
```

The following examples illustrate the use of the size specifiers:

```
     Reverse Polish                  Motorola
$ 7F #  0DR DN        BYTE AND       AND.B  #$7F,D0
0DR DN  0AR AN        WORD MOVE      MOVE.W D0,A0
0AR 0DR WORD 0 AN+R+B BYTE TST       TST.B  0(A0,D0.W)
```

These two words generate local labels for branching...

```
MARK ( --- )  ( BR# --IN-- )    ( create a label number )
BR:  ( #br -- dest-addr #br  )  ( post fix create a label )
```

The only difference is the manner in which they accept the argument specifying the branch location, either PRE- or POST-FIX notation. This example illustrates both:

```
CODE TEST-SIGN  ( n1 -- result , -1=negative 0=zero 1=positive )
         TOS DN    TST      ( test register contents )
         2 BEQ    ( just leave if its zero by branching to '2' )
         5 BMI              ( if negative, branch to 5 )
         1 #  TOS DN MOVEQ  ( positive if here, set result = 1 )
         2 BRA              ( and exit )
MARK 5   -1 # TOS DN MOVEQ  ( negative if here, set result = -1 )
2 BR:    RTS
END-CODE
```

Note that the numbers used to specify the branch locations need not be sequential, but each one may only specify one location (paired only once with BR: or MARK) within the CODE/END-CODE combination. The same numbers may again be used in the next CODE word.

You may find words in the ASM source code that are not explained in this chapter.   These are JForth Assembler INTERNAL words that should NOT be called from other programs.

Non-Standard Opcodes

Those 68000 opcodes that reference the status register (SR), user stack pointer (USP), or condition codes register (CCR) are available in the JForth RPN Assembler as custom opcodes which accept a single source or destination operand. Examples...

ORG TOS 0 AN+R+B MOVE-FROM-SR   (move status reg to addr on stack)

$ 0f #   ANDI-CCR   (zero   all but lower 4 bits of CCR)

The entire list follows. IMPORTANT!!! Those marked as "Privileged" should ONLY be used in interrupt code!!!

Opcode	Privileged
MOVE-TO-USP	X
MOVE-FROM-USP	X
MOVE-TO-SR	X
MOVE-FROM-SR	X
MOVE-TO-CCR	
ANDI-SR	X
EORI-SR	X
ORI-SR	X
ANDI-CCR	
EORI-CCR	
ORI-CCR	

Standard Opcode Mnuemonics

These function according to the Motorola standard, except as noted.

```
LINK MOVEM MOVEP TRAP CMPM MOVEQ
RTR TRAPV RTS RTE RESET NOP STOP SWAP UNLK EXT TAS ABCD SBCD
CLR NEG NOT TST NEGX CMPI ORI ANDI SUBI ADDI EORI MOVE
AND OR SUB ADD EOR CMP MULS MULU CHK DIVS DIVU LEA SUBQ ADDQ
PEA JMP* JSR* NBCD EXG ROR ROL LSR LSL ROXR ROXL ASR ASL ADDX SUBX
Bxx** (branch on condition) DBxx** (decrement & branch on
condition)
```

```
BSR***
SCC SLS SCS SLT SEQ SMI SF SNE SGE SPL SGT ST SHI SVC SLE SVS

  * - Use  ] <name> [  instead of Jxx (invokes compiler to CALL
word)
 ** - accepts a local label, for example:   BNE  1$
*** - Usage:   ' <wordname> BSR    (forces PC-relative call)
```

More RPN Examples

```
( Demonstrate accessing a Variable )
VARIABLE MY-VAR
CODE SHIFT-MY-VAR! ( N -- , Shift data in MY-VAR N times left. )
\
\ invoke compiler; generate variable address...
    ] MY-VAR [               ( -- N MY-VAR )
\
\ get a copy of N into d0...
    DSP A@  0DR DN  MOVE     ( -- N MY-VAR ) ( d0=shift count )
\
\ do the shift...
    0DR DN  ORG TOS 0 AN+R+B  ASL   ( -- N MY-VAR )
\
\ drop both numbers from the stack...
    CELL #  DSP AN  ADD       ( -- MY-VAR ) ( same as NIP )
    DSP A@+ TOS DN  MOVE      ( -- )        ( same as DROP )
END-CODE
```

This example demonstrates local branches, and also invoking the compiler (via] and [)to create a call to another Forth word. Note how D0 is saved while calling EMIT.

```
CODE PLOT# ( N -- , emit N dashes )
      TOS DN   0DR DN   MOVE   ( keep count in D0 )
      DSP A@+  TOS DN   MOVE   ( drop count and reload TOS)
1 BR: TOS DN   DSP -A@  MOVE   ( save TOS )
      ASCII - # TOS DN  MOVEQ  ( load '-' character into TOS )
      0DR DN   7AR -A@  MOVE   ( save D0 on return stack)
      ] EMIT [                 ( emit dash )
      7AR A@+   0DR DN  MOVE   ( restore D0 )
      1 # 0DR DN SUBQ          ( decrement loop counter )
      1 BNE                    ( loop until done )
END-CODE
```

Additional RPN Assembler Features

RPN Assembler Usage in Colon Definitions

As long as you do not use the branching operators (Bcc, DBcc, BR: and MARK), you can activate the RPN assembler during normal compilation, as in the following example.   (If you use the assembler as a MODULE and not compiled resident in your dictionary, you need to insure the ASSEM module is loaded via GETMODULE ASSEM for this feature).

```
: SUM*2 ( a b -- [a+b]*2 )
   +  [ ALSO ASSEMBLER  TOS DN TOS DN LONG ADD   PREVIOUS ] ;
```

RPN Assembler Macros

Similarly, to do Macros in JForth RPN assembler, simply write a colon definition that has assembler words compiled into it. (This cannot be done if you need to branch using BR: or MARK. Also, the RPN assembler MUST be resident in the dictionary, and NOT loaded as a MODULE. See the above section *Compiling the RPN Assembler* to achieve that configuration).

```
: M.DOUBLE   [ ALSO ASSEMBLER ]  TOS DN  TOS DN  ADD
             [ PREVIOUS ] ;  IMMEDIATE
```

When this word is used within another definition (as follows), it will invoke the assembler to create the ADD instruction inline.

```
: DOUBLE-MY-VAR ( -- , double the contents of a variable MY-VAR )
  MY-VAR @  ( -- my-var-data )   \ get value in MY-VAR
  M.DOUBLE  ( -- my-var-data*2 ) \ compile above ADD inst inline
  MY-VAR !  ( -- )  ;            \ put it back
```

The RPN assembler is unique in this ability to create macros; this is not currently possible with the Forward Assembler. Unfortunately, the disadvantage of not being able to use the RPN assembler as a MODULE usually outweighs the benfit of this somewhat obscure feature.

The RPN Assembler as a MODULE

In JForth V1.3 and higher, the RPN Assembler may be compiled and saved as a MODULE (as is the default for the release version, in MOD:ASSEM.MOD). The standard procedure for recompiling the com:JForth image, JF:LOADJFORTH, automatically regenerates the .MOD file if MODULE support is included.

When implemented as such, the CODE and ;CODE keywords will automatically load the ASSEM module from MOD: if needed.

When the RPN Assembler is being used in module form, its directives may NOT be used to build assembler macros, described above.

Motorola-Style (Forward-Parsing) Assembler

As many knowledgeable 68000 assembly-level programmers will admit, a non-standard reverse-polish syntax for a Forth assembler seems an additional burden to learn, simply to program in this 'lowest-of-levels'. While the inherent 'macro-ability' and interactivity of the RPN assembler is an important gain, the Forward-Assembler environment features familiar Motorola formats as well as support for 'interactively' accessing elements of the JForth dictionary.

You can see additional examples of Forward Assembler usage in the files JF:HASHING, JIFF:UNPACKING, CL:STARTJFORTH.ASM.

Please note that the Forward Assembler does not currently support the BLOCK environment; it may only be invoked within standard ascii text files accessed with INCLUDE, or from the keyboard.

Compiling the Forward Assembler

Since the Foward Assembler is available as a MODULE, it is not usually required that the program be compiled. Nonetheless, the Forward Assembler may be compiled into the resident dictionary by entering:

```
DETACHMODULE ASSEM
INCLUDE JF:FORWARD-ASM
```

Forward Assembler Usage

The Forward-Assembler is invoked with the "ASM" keyword, followed by the name of the word to be created (similar to the use of ":" when compiling a HIGH-LEVEL word). Assembly-language mnemonics then follow, one statement per line.

Finally, the last line begins (and ends) with the "END-CODE" operator, terminating the assembly-mode and resolving/verifying the just-created word. An 'RTS' instruction is automatically assembled by this operation (ALL JForth words MUST end with RTS!) if necessary.

This is illustrated by the following simple example, called ADD2, which adds the top two items on the stack...

```
ASM ADD2   ( a b -- a+b )
   ADD.L  (A6)+,D7
END-CODE
```

The overall syntax adheres closely with Motorola standards; the following fields, based on character position, are defined:

```
LABEL  OPCODE  OPERAND      COMMENT (rest of line is ignored)
5$:    add.l   (a0)+,d0     add what A0 points to with D0
       move.l  d0,d1        copy the sum into d1
       bne.s   5$           do it again if 'non-zero'
```

The following notes apply to each specific field...

The Forward Assembler Label Field

The first column of each line marks the beginning of the Label Field.

The first character of this field must be one of 3 things: whitespace (blank or TAB), the beginning of a Forth comment, or begin a properly-formatted local label.

The Forward-Assembler supports the use of Motorola-style local labels. A declaration begins in column 1, and consists of the label VALUE followed by "$:" (for example... 18$:).

Currently, only one class of instruction may operate on declared labels; those which conditionally BRANCH on a tested condition. BEQ (branch-if-equal) and DBEQ (decrement-&-branch-until-equal) are examples of this class.

The following illustrates a test & branch condition. Here, the tight loop will be repeated until register D0 equals zero...

```
1$:    subq.l  #1,d0    subtract 1 from d0...
       bne     1$       if not equal to zero, branch to 1$

... <<< EXECUTION CONTINUES HERE WHEN D0 = 0 >>>
```

Please note that label values MUST BE UNIQUE, but only between the same ASM and END-CODE combination. Once another ASM has begun, the values may be reused. Also, any DECIMAL value may be used; the programmer is not restricted to using sequential values.

The Forward Assembler Opcode Field

This field contains standard Motorola-style 68000 mnemonics in either case, and, where necessary, the size-specifying suffix .B, .W, .S, or .L.

A few JForth-specific commands are also scanned for to provide greater flexibility:

If the opcode field contains CALLCFA the next text is considered the name of a JForth word, and a call to that word will be compiled in the most efficient manner possible (do NOT use the JSR or JMP instruction to reference NAMED words...use CALLCFA). The following example reads two variables, DIVIDEND and DIVISOR, calls / to divide them and leaves the result on the stack:

```
ASM DIV2  ( -- n , divide the variables )
        callcfa   DIVIDEND         ( -- DVNDvar )
        move.l    0(org,tos.l),tos ( -- DVDN )
        callcfa   DIVISOR          ( -- DVDN DVSRvar )
        callcfa   @                ( -- DVDN DVSR , same as move.l..)
        callcfa   /                ( -- QUO )
END-CODE
```

It is possible to explicitely force a PC-relative call to be assembled to another named word as long as it is within +-32K of the calling instruction. This is useful to build interrupt code where the ORG register is not yet setup.

```
        move.l    #[ascii -],d0   put a '-' character in d0
        bsr       ProcessChar     call some char processor routine
```

If the opcode field contains FORTH{, all text up to the following } character (or end-of-line) will be sent to the JForth INTERPRETER and executed. This example invokes the compiler to create a reference to an Amiga Library function:

```
        move.l    tos,-(dsp)   ( -- prevTOS )
        move.l    #$40,tos     ( -- $40 )
        FORTH{  ] callvoid dos_lib Delay [  }   ( -- )
```

The Forward Assembler Operand Field

This field describes any operands needed by opcode. All standard Motorola-style effective address formats are recognized, as well as a few JForth-specific patterns which are here described.

The Forward Assembler, as supplied in JForth versions 3.0 and earlier, does not directly support references to the SR (Status Register), CCR (Condition Codes Register) or the USP (User Stack Pointer), but access to the related RPN assembler operatives is always possible. For example...

```
   \ Zero all but lower 4 bits of CCR...
   FORTH{ ASSEMBLER   $ 0f  #  ANDI-CCR ( andi #$f,ccr )    PREVIOUS
   }
```

A special construct affords access to most JForth entities; in places where a numeric argument is expected, the [and] characters can be used to delimit a string to be submitted to the JForth INTERPRETer (similar to the FORTH{ operative, described above). This is useful to apply defined data CONSTANTs, but can be used for other purposes. The following example checks if the value in TOS is equal to a pre-defined CONSTANT (in this case, an Amiga mask describing memory):

```
   cmp.l  #[MEMF_CHIP],tos     is the value in tos = mask?
```

Another example illustrates how this feature can also be used to pre-calculate operand 'literal' values:

```
   move.l #[8 1024 *],tos      put '8K' in TOS
```

A third example reads the 4th element of a 32-bit array, base address in a0, into register d0:

```
   move.l [3 cells](a0),d0     read a0 plus 12, indirect
```

Any valid sequence of Forth commands may exist between the [and] characters, as long as it meets the following criteria:

1. The expression has an overall stack diagram of (-- n1) .

2. Due to the parsing requirements of the Motorola assembly format, the text between the [and] delimiters may NOT contain any of the following 6 characters: ( ) , . [ ]

NOTE: You should never use the [and] characters to generate an ADDRESS for an instruction. An example of this would be:

```
   move.l  #[' NOOP],tos
```

Such code will run in the JForth dictionary, but is not compatible with the CLONE program. You can use FORTH{ and the compiler directive ALITERAL to create code which will push an address on the stack in a CLONEable way:

```
        FORTH{  ' NOOP  ] ALITERAL [   }
```

NOTE: between the ASM and END-CODE, numbers are INTERPRETed in DECIMAL; preceed hex numbers immediately with a $ character, for example $FFFE.

Example of Accessing Structure Members

Here is an example illustrating how to access members of an Amiga structure. Structure members return their offset when referenced so we can use them directly between [and]. Make sure you use the proper size MOVE. Notice the MOVE.W for the width. Notice also that we clear the high bits of D7 first because MOVE.W onlt sets the low bits. This example also illustrates conversion between absolute and relative addresses. This next word takes a relative window address and prints its title and width.

```
ASM  WINDOW.INFO  ( window -- )
   MOVE.L   D7,A0       \ relative window address to A0
   ADD.L    ORG,A0      \ convert to absolute address
   CLR.L    D7          \ clear high bits of TOS
   MOVE.W   [WD_WIDTH],D7     ; get width in TOS
   MOVE.L   D7,-(DSP)   \ save TOS
   MOVE.L   [WD_TITLE](A0),D7 ; get title in TOS
   SUB.L    ORG,D7      \ convert to relative for JForth
   CALLCFA  0COUNT      \ ( -- width address count )
   CALLCFA  TYPE
   CALLCFA  SPACE
   CALLCFA  .           \ print width
END-CODE
```

We can test this word by assembling it then entering:

```
INCLUDE JU:AMIGA_GRAPH
GR.INIT  GR.OPENTEST
GR-CURWINDOW @ WINDOW.INFO
GR.CLOSECURW  GR.TERM
```

Example of Referencing Variables from an Interrupt

If you need to reference a variable from an interrupt routine, you cannot use

```
CALLCFA  VAR1  \ NOT legal in interrupt routines
```

because that assumes that the 68000 registers are setup for JForth use. In an interrupt, that will not be true. In an interrupt you cannot use the JForth data stack, or call any word using CALLCFA, or call any word that uses the data stack. Here is an example of referencing a VARIABLE and calling a subroutine using legal PC relative addressing:

```
variable VAR1
ASM INCVAR  \ increment VAR1
   LEA      [VAR1 HERE - 2-](PC),A0
   ADD.L    #1,(A0)
END-CODE
ASM   INTROUTINE
   BSR      INCVAR      \ PC relative call so OK
   CLR.L    D0          \ tell Amiga we’re done
END-CODE
```

See the files JD:DEMO_INTERRUPT and JD:HIGH_INTERRUPT for more examples.

The Forward Assembler as a MODULE

In JForth V1.3 and higher, the Forward Assembler may be compiled and saved as a MODULE (as is the default for the release version, in MOD:ASSEM.MOD). JF:LOADJFORTH, the standard procedure for recompiling the com:JForth image, automatically regenerates the .MOD file anew if MODULE support is included.

When implemented as such, the ASM keyword will automatically load the ASSEM module from MOD: if needed.

DISM - JForth Disassembler

Compiling the Disassembler

Since the Disassembler is available as a MODULE, it is not usually required that the program be compiled. Nonetheless, the Disassembler can be compiled into the resident dictionary by entering:

```
DETACHMODULE DISASSEM
INCLUDE JF:DISM
```

Disassembler Output

DISM, and its related forms, sends output to the standard EMIT device. It is of the form:

```
1D14  move.l  tos,-(dsp)       2D07         "-."
1D16  moveq.l #$20,tos         7E20         "~ "
1D18  bsr.w   26D0  = EMIT     6100 09B6    "a..."
1D1C  rts                      4E75         "Nu"
```

The first column indicates the address of the code. This may be displayed in several modes (see DISM, RISM, ADISM and RELDISM below), but normally represents the JForth relative address.

The second column is the assembly language mnemonic.

The third column displays the operands (for options, see DISM-NAMES, below).

In the fourth row is shown the actual hex code of the opcode and related operands.

By default, the fifth column displays ascii-equivalent characters, helpful in identifying strings. (for options, see DISM-CYCLES, below).

"Automatic" Disassembly Features

All forms of the 'automatic' disassembly words will keep track of forward branches, and only stop disassembling when a 'return' type opcode is displayed that does not have a pending conditional branch past it. (?PAUSE is called at the end of each line, though, so you can simply type QUIT to stop disassembling, or space bar to pause and RETURN to continue).

All forms of 'automatic' disassembly will examine two state variables to determine runtime conditions for:

1. whether generic or JForth specific register names will be displayed. (see `DISM-NAMES,` below)
2. whether the timing cycles will be displayed in the 5th column instead of the default ascii information. (see DISM-CYCLES, below)

These state variables are DISM-NAMES and DISM-CYCLES, respectively. For example, to display the timing cycle information, enter DISM-CYCLES ON then disassemble something, observing the 5th column.

Timing cycles are of the form (xx:yy:zz) where:

xx = no. of 68000 bus cycles this instruction yy = no. of 68000 bus cycles since beginning of word (ignores branches & loops)

zz = no. of microsecs since beginning of word (on std. A2000, 7.16 Mhz 68000)

NOTE: *Timing cycles are estimates at best and largely for entertainment value.*

CPU registers are displayed according to the state of DISM-NAMES as follows:

```
DISM-NAMES = 0    DISM-NAMES != 0
           a0        a0
           a1        a1
           a2        a2
           a3       +64K  image-base+64K
           a4        org  origin
           a5        up   user pointer
           a6        dsp  data stack pointer
           a7        rp   return pointer
           d0        d0
           d1        d1
           d2        d2
           d3        d3
           d4        loc  locals
           d5       iloop 83-type loop index 1
           d6       jloop 83-type loop index 2
           d7        tos  top of stack
```

NOTE: the same flag, DISM-NAMES also determines whether the disassembler will attempt to display the names of routines being 'called'.

Disassembling within the JForth Image

For examining code within the JForth image, the two most often used words are DISM and DEF.

```
DISM  ( rel-addr -- )     \ for example,   ' D+ DISM
```

Disassembles from the relative address on the stack in 'automatic' mode (see above). DISM does not recognize strings, and attempts to disassemble them.

```
DEF   (  <wordname> -- )  \ for example:  DEF D+
```

DEF attempts to find the next input word as a dictionary entry. If it is successful, it disassembles from the code-field-address. Note that DEF will recognize a compiled string, displaying its content in ascii form.

Disassembling outside of the JForth Image

Three words are useful for disassembling code that exists outside of the JForth image.

```
ADISM ( abs-addr -- )
```

ADISM disassembles from the absolute address on the stack, with addresses labeled as such.

```
RISM  ( rel-addr -- )
```

RISM disassembles from the rel-address as does DISM, but displays the first line as address 0, with successive addresses being labeled relative to the beginning of the disassembly.

```
RELDISM ( org-addr rel-addr -- )
```

RELDISM changes the output addressing scheme similar to RISM, but accepts an additional parameter specifying the address to consider relative 0, the origin address.

The Disassembler as a MODULE

In JForth V1.3 and higher, the Disassembler may be compiled and saved as a MODULE (as is the default for the release version, in MOD:DISASSEM.MOD). JF:LOADJFORTH, the standard procedure for recompiling the com:JForth image, automatically regenerates the .MOD file anew if MODULE support is included.

When implemented as such, the following words will automatically load the DISASSEM module from MOD: if needed:

```
DEF  DISM  RELDISM  RISM  ADISM  DISM-DONE?  DISM-WORD?
INIT-DISM  DISM-CYCLES    DISM-NAMES DISM-ORIGIN .REGNAMES?
SHOW-CYCLES
```

When the Disassembler is being used in module form, these words also form the entire programmatic interface to the facility.