THEORY OF DESIGN

FOR

FOONLY MODELS F4A AND F4X

9 May 1985

Table of Contents

CHAPTER 1

INTRODUCTION

This document is for use by experts who have already completed the full Foonly course on the design of this system. Therefor no attempt is made herein to cover every detail of the design; instead we present purpose-oriented (rather than implementation-oriented) explanations for those aspects of the system which are least obvious in their design and operation.

The (expert) reader is expected to consult in parallel with this manual all the other sources of information on the system, including data path diagrams, microcode definitions, flow charts, and logic drawings.

Extensive cross-references are provided; refer to the Table of Contents at the end of the document to find the page numbers of the referenced sections. Symbols enclosed in pointy brackets are names of logic drawings.

The F4A and F4X are dexcribed together in this manual; all references to extended addresses are inapplicable to the F4A. Other differences are mentioned explicitly.

CHAPTER 2

MAIN DATA PATHS

These are mostly rather straightforward. Refer to the 2901 data sheets for details of the ALU bit slice.

Note that the DBUS is the path into the ALU from the rest of the machine. The AC's and the Q register are in the ALU.

Following are some notes about the more subtle aspects of the data paths.

2.1. MEM and HOLD

The path called MEM, and referred to by D[MEM], is either the output of the HOLD register or of the MB register, whichever has been most recently loaded. HOLD is loaded only by DEST[HOLD], and MB is loaded only by any CPU read from memory. If both occur together, MB wins (<SQCTL2>).

2.2. AMEM Addressing

The 1K AMEM can be addressed in either of two modes (<SQAMAD>):

i. "Globally", with a 10-bit address

ii. "Locally", in blocks of 16 words, with a 4-bit address.

Global AMEM addresses come from the JADR field or from the 10-bit AMEM PDL PTR. For Local addressing, the highest order bit is 0, and the next five come from the ION (ie the currently selected I/O device number); since FBUS Interrupts (see 5.3) automatically select the interrupting device, a

16 word local AMEM block is thereby made available for use by the interrupt microcode for that device.

The low order 4 bits in the local mode come either from the D field or the DEST field, whichever is in the range 40:57.

2.3. AC Addressing

On <SQACAD> we see the 8 available combinations of AC Address A and AC Address B; see data on the 2901.

2.4. ALU1 - the Other Set of Registers

As is indicated on the main data path diagram, there are 2 ranks of 2901's connected in parallel; the ALU1 bit of the microinstruction selects the alternative rank. This kluge is there to provide some registers for use by microcode; the most often used is the Q register of ALU1.

2.5. ALU Shifting - End Connections

The 2901's one-bit shift functions are available, with the (72 bit) wrap-around connections at the ends being controlled by the low order 2 bits of the MAPF field.

CHAPTER 3

MAIN MEMORY OPERATION

Memory cycles start at cycle boundaries; the address of a memory cycle must therefor be available before the end of the preceding cycle. In particular, the (virtual) address for any CPU cycle must be on the EA near the end of the preceding cycle (how near the end depends on whether mapping is enabled). It is also required that the MA be loaded with this address, because if any WAIT cycles (5.2) occur the MA will be gated to the EA during them (see <SQEACT>).

Write cycles (ie Data Store or DS cycles) are requested during the cycle before they are to happen, by the DEST field. The write data must be loaded into HOLD not later than the end of the requesting cycle. The microinstruction corresponding to the write cycle must contain trap code bits in MAPF (see 5.3.3.1.1).

Read cycles (Data Fetch (DF) or Instruction Fetch (IF)) are requested by the microinstruction which will be executed during the memory cycle; the DFRQ and IFRQ bits do the trick. The data appears on MEM at the beginning of the next cycle.

The read data may also appear late in the read cycle on the EA; this is enabled by SPECIEA-FROM-MEM] or by IDISP (6.1.1). In addition, bits 00:08 of the read data go to the MMA board for use during Principal Idisp cycles.

The first microinstruction to refer to the read data must contain trap code bits in MAPF (see 5.3.3.1.1).

3.1. MAIN MEMORY CONTROL

Control for main memory operations is mostly on <MCCTL1, MCCTL2, MCCTL3>; a bit of left-over timing logic is on <SQMCT> and the refresh interval counter is on <SMCTL1>. Memory cycles may be initiated by the CPU, the refresh counter, or I/O devices (DMA cycles via FBUS).

On <MCCTL1>, near the end of every cycle, any type of request causes START MEM (which goes to <MCCLK3> to become MC MEM GO); what type of cycle we will have is determined by priority gating and recorded in FFs at end of cycle. A complication is caused by the fact that a single microinstruction may request both a Data Fetch (DF) and an Instruction Fetch (IF) by using the DF/WT feature; see 5.2.1.

On <MCCTL2> a separate but consistent set of priority gating sends signals to the MAP part of the Address Engine (3.2.3) informing it about what source of address to use and whether to map it.

Since the IFRQ and DFRQ bits of the microinstruction causes the memory read to happen during that microinstruction, they are examined by the memory control logic not from the MI, but directly from the MM lines; they are the most timing-critical of the micromemory bits.

3.2. THE ADDRESS ENGINE

The address engine is the portion of the hardware primarily concerned with generating, handling, and mapping main memory (ie macro) addresses. It comprises the EA (Effective Address) bus, the PC, the MA, the AA

(Address Adder), the SEGMEM, the EAX MUX, the PPC/JPPC, the IOA, and the MAP (see the MAP data path drawing for the last five items). The control logic for most of this is on <SQAECT, MCCTL2>, but see 3.2.2. The operation of the PPC is described in 6.2.3; the remaining portions of the address engine are discussed below.

3.2.1. BASIC CPU ADDRESS MECHANISM

The EA is the virtual address input to the MAP. Mapping is done during the final portion of the cycle, with the physical address becoming available by cycle end (see 3.2.3), which allows memory references to start at the cycle boundary (3.1). Thus the address to be used by a memory reference must be on the EA during the cycle preceding the reference cycle.

The AA is actually only an incrementer, but for speed it is implemented as a carry-lookahead adder. Notice that either the PC or MA can be incremented and the result placed in the PC or on the EA, so the PC+1 function or MA+1 to PC function can be done without involving the EA. The EA can use uncorrected memory data or MEM data, for handling macro instructions and indirect words (see 6.1.1); it can also use data from OBUS for cases where instructions or addresses are calculated in the main ALU (indexing, eg).

It is necessary that the MA contain the address of any memory reference during the reference cycle; in case the cycle is delayed by wait cycles, the logic on <SQAECT> forces MA to EA so the desired reference can be started after the wait.

There are only two original sources of addresses: the EA/SEGMEM/EAX MUX combination generates all CPU addresses, while the IOA supplies addresses from the FBUS for DMA cycles. There are two secondary sources, however: the PPC and JPPC registers store physical addresses from the map output for later use in instruction prefetching (see 6.2). A selection of one of these four address sources is made via tri-state bussing on the MP board (control logic on <MCCTL2>). See also 3.1.

3.2.2. EXTENDED ADDRESS HARDWARE

A (virtual) address is called global if its section part (06:17) is valid; otherwise it is called local. Please note that this useage of "global" differs from that in the DEC description of extended addressing; there an address is called "local" if the last step of its calculation was not allowed to change the section part, even if a valid section part was generated by an earlier step.

Whenever a virtual address is accepted from the EA by the MAP, the section number is taken from the EA if the EA is global; if not, the section number is supplied by the Segment Memory, or SMEM (see 3.2.2.2). The EAX MUX <MXVAO> does the selection.

3.2.2.1. GLOBALITY DETERMINATION

The rules for determining the globality of addresses depend on the section number of the PC ("PC SECT") and, during an effective address calculation, on the section number of the current partial effective address ("E SECT"). Logic on <SQAXCT> generates this state information. The rules

also depend on the current context of instruction execution.

A logic net, partly on the MP board (<MXEXT>) and partly on SQ (<SQAXCT>) determines the globality of the EA (EA GLOBAL) dynamically, in a context-dependent way:

i. FORCE GLOBAL <SQAXCT> is generated by SPEC[GLOBAL], by global index cycles, by the MA being gated to the EA if the MA is global (see below), and during the 1st half of every cycle except jump cycles which have local addresses. (This last condition is really relevant only during PIDISP cycles, when the PC, which is always global, is on the EA for mapping.)

ii. FORCE LOCAL <SQAXCT> is generated by PC SECT=0 (obviously), by PDISP (see 6.1.1.3), by E SECT=0 during an address calculation, and during indirect cycles when bit 0 of the indirect word is on. The last two conditions are are required by the effective address calculation algorithm.

iii. If neither of the above two condidtions obtains, the logic on <MXEXT> recognizes 3 cases:

iv.

a. During a Local Index Cycle, globality must be determined by whether the section number in the index AC (00, 06:17) is greater than zero. Since the index add is adding either 0 or -1 to the left half of the AC (see 6.1.1.3.1), this can be

determined by checking the resulting section number (EA00, 06:17) for the values 0 and -1, plus knowing whether a carry is propagated into bit 17.

b. At other times, globality is determined simply from whether the section number of the EA is greater than 0, except that if EA01 is on, a zero section number will be global.

3.2.2.2. SMEM OPERATION

The SMEM (<MXSEG>) is controlled by logic on <SQAXCT>. It stores two section numbers at all times (in locations 0,1): the current PC section number and the current MA section number. Which section number is in location 0 is determined by the state of the 74S74 in the upper left of <MXSEG>, which is called the SMEM SEL FF.

During the first half of a cycle following the loading of MA LT (except on Jump Cycles with local addresses), the new MA section number is stored into the SMEM (whether or not it is valid, notice). Following any load of PC LT, the SMEM SEL FF is complemented, thus making the most recent MA section number become the new PC section number. This implies the obvious coding rule.

3.2.2.3. MA section number and PC section number

The PC is always interpreted as global. PC LT is loaded along with PC RT whenever we have SPEC[GLOBAL] or a Jump Cycle with a global address; this latter is, of course, the only normal way of changing PC section numbers.

The globality of the MA is determined from EA GLOBAL when MA LT is loaded, namely together with MA RT on all non-IDISP cycles and also IDISP cycles with EA GLOBAL (see 6.1.1.3.3). MA00:01 are not data bits: MA00 indicates MA LOCAL, and MA01 indicates MA GLOBAL.

3.2.2.4. Extended Indexing Hack to DBUS

Local Index Cycles require a half-word sign extend (see 6.1.1.3.3). This is handled on <BOAM1, B1AM0> by forcing the select for the DBUS multiplexor to 3 (CONST) instead of 1 (MEM) for bits 06:17 only, and feeding the corresponding data inputs with a buffered version of MEM18. In addition, the ALU carry into bit 03 is inhibited to prevent a carry from making the sign of the result different from the sign of the AC.

3.2.3. The MAP

The MAP selects an address from one of three sources:

- the EA (or EA plus SMEM),
- the IOA,
- or the PPC.

Each source is treated somewhat differently.

3.2.3.1. EA Addresses

These are virtual addresses, and if the map is on, their page part is translated by table lookup in the Map Memory to produce the physical page number. The Map Memory address is a hash of the current context number and

the (extended) virtual page number (<MXCTXT, MXADV0, MXADV1>).

The physical page number in the Map Memory has most of its bits represented in binary, but the 4 bits which select one pair of memory boards are pre-decoded into a 4 by 4 x/y code (called MS 0:7). See <MXIPAR>. All the physical address bits are loaded by cycle clock into the PMA, which actually drives the memory address lines to the boards.

In addition to the physical page number, the Map Memory produces a TAG field which is used to determine whether the entry is valid. Parts of the TAG are compared (<MXVA2>) to the USER mode bit, the virtual section number, and part of the context number. The 3 bits of the context not compared, together with the virtual page number, are transformed one-to-one by the hasher, and therefor are implicitly included in the compare by virtue of the Map Memory location addressed. As a special hack, TAG bit 03 must be on to enable the context compare in EXEC mode; this allows for gluing down parts of the resident monitor.

The low-order bits of the EA are gated directly onto the PADR bus (<MXPAIO>). The PMA MUX at the input of the PMA combines the tranlated page number with the unchanged low-order bits to form the final physical address.

If the map is off, the page and section parts of the EA are gated to the PA bus (<MXPAIO>), except the 4 memory select bits, which go to the PD. The PD is decoded into the 8 PA MS bits; then the PMA MUX passes the whole mess through to the PADR and hence the PMA.

3.2.3.2. PPC Addresses

Addresses in the PPC are already mapped (see 6.2.3), and thus already have the MS bits decoded. Therefor the PPC can be, and is, gated directly onto the PA bus.

3.2.3.3. IOA Addresses

These addresses are treated like EA addresses with the map off.

CHAPTER 4

FBUS OPERATION

FBUS operation is exactly as documented for the model F3.

The control logic is on <MCFBCT>, and is straightforward. Note, however, the interlock with DMA useage of the IOD; see 5.2.

CHAPTER 5

MICROCODE SEQUENCING

Ordinary squencing of microinstructions is handled by the data paths shown on the <MICRO SEQUENCER> data path diagram. The logic is on <SQJMP0, SQJMP1, SQMAD0, SQMAD1, SQMAD

5.1. Basic Address Generation

SQ MMAD 00:13 are the address lines to the micromemory. Based on the MI BRANCHING signal (a function of the condition generator and the JCODE field -- see <SQJMP0, SQJMP1>), these lines get the MIC (which always contains one plus the location of the current microinstruction) or the Branch Address (MI BA).

The MI BA can come from several sources:

- the JADR field of the microinstruction (for jumps)
- the IDISP address (see 6.1.1.1)
- the SDISP address, which is an OR of the OBUS and the JADR field; this is for calculated dispatches
- the JMEM; this allows dispatching through tables in the JMEM, with SDISP address used to address the JMEM
- the POPJ register; see 5.2

5.2. PUSHJ/POPJ

The MI PDL register is 12 bits, but only 8 bits count, so the stack is some 256 word segment of the JMEM. The only tricky aspect of PUSHJ and POPJ is that when PUSHing the value being stored is duplicated in the POPJ REG, and during a POPJ is taken from there while the JMEM accesses the next thing down on the stack, which is then loaded into POPJ REG at end of cycle. Thus POPJ never has to suffer the access time of the JMEM.

section(Wait Cycles)

Sometimes the CPU must pause in its execution of microinstructions to wait for completion of a CPU memory cycle, which may have been delayed by either a refresh or a DMA cycle (3.1). Also, the CPU may need to wait for a DMA cycle to complete so that the CPU can use the FBUS data lines for an IOB operation. Occasionally, too, a cycle must be aborted (ie its effects must be prevented); see 5.3.3. Finally, the CC may have set the CPU RUN switch to 0. In all of these cases, the CPU takes Wait Cycles.

The logic for generating WAITING is on <MCCTL3>, and is simple. The effects of WAITING are also simple: the distribution of register clocks to the machine is inhibited, including those to the MIR (<MCCLK3, SQCTL3>). Thus the machines state is frozen for a cycle, and the microinstruction in the MIR gets another chance the following cycle.

Of course, some clocks must still go out so that whatever needs to be completed will finish; these are called Ungated Clocks.

5.2.1. DF/WT Cycles

A microinstruction which does a DFRQ may request, via the WAIT bit, that all other functions of that microinstruction be delayed until the read operation is complete. Thus the microinstruction spends at least 2 cycles in the MIR, all but the last one being Wait cycles, during which nothing happens execpt the DF. During the final, non-Wait cycle, everything else specified by the microinstruction (which may include an IFRQ) is allowed to happen; since this is after the DF read, a D[MEM] will reference the data fetched by that same microinstruction.

5.3. Traps

The usual sequence of microinstruction addresses can be altered by traps, which are of two types, Trap0 and Trap1. Trap0 is associated solely with IDISP operations (see 6.1.1.2, 6.1.1.3), while Trap1 is more general. Both use the same mechanism, however.

5.3.1. Common Trap Mechanism

Traps access the trap microinstruction by selecting one of the two Trap Ram banks of the micromemory: bank 0 is selected for Trap0, and bank 1 for Trap1 (<MMASEL>). Since this selection occurs at the output of the micromemory, it is fast, and the RAM's in the trap banks are also very fast.

The address input to the trap RAM's comes from logic on <SQTRP0, SQTRP1>. SQ TRAP AD 00:03 comes either from either the Trap1 encoder, the Trap0 encoder, or from SQ MM AD 09:12; this last source allows the trap RAM to be

addressed in the usual way, with bank 0 (Trap0) addresses appearing as even locations and bank 1 (Trap1) addresses as odd locations.

5.3.1.1. The Missing Bits in the Trap RAM

An oddity of the two trap RAM banks is that they are missing some of the bits of a complete microinstruction word. This was forced by space limitations on the MMA board. The missing bits are:

- MAPF field
- SPEC field
- MASK field
- ROT field
- JCOND field
- ALU DST field, [4:2]
- LD AR
- ALU1 SEL
- DFRQ
- IFRQ

These bits were chosen as being not likely to be used in trap microinstructions.

Some of the missing bits are forced to one state or the other during a trap cycle of any kind, in order to disable the functions they control. For example, the ROT and MASK fields have enough bits forced on to bypass the rotator/masker. DFRQ and IFRQ are supplied by special logic on <SQCTL1>. When Trap RAM is addressed as ordinary micromemory (during a non-trap cycle), all the missing bits appear to be 1's.

5.3.2. Trap0

For the use of Trap0, see 6.1.1.1 and 6.1.1.3. Trap0 conditions are detected on <SQTRP0> during IDISP cycles, and the appropriate trap addresses encoded by logic at the right of <SQTRP1>.

The main thing to notice about Trap0 cycles is that they do not generate WAITING; ie the trap cycle (the IDISP during which the Trap0 is generated) does not get aborted. All of its usual effects occur except that the next microinstruction comes from the trap location.

5.3.3. Trap1

Trap1 conditions are generated in various parts of the machine, then collected and enabled on <SQTRP0, SQTRP1>, after which a priority encoder produces the TRAP1 AD 00:03 bits.

The most important thing to remember about Trap1 is that Trap1 cycles are aborted; ie they generate WAITING, so that none of their effects occur.

5.3.3.1. MAP FAULT and ECC ERROR Traps

These types of Trap1 share a rather complicated generation mechanism. MAP FAULT is detected during the mapping operation before a memory cycle begins, while ECC errors can only be detected after the completion of a read operation. Furthermore, it is desired not to take traps for any reason on a read cycle until and unless the data is actually referenced. These considerations require a means of keeping track of the status associated with a memory operation from before the start to possibly well after the finish of the actual memory cycle.

The logic for this is on <MCMPST, MCMBST, MCSTCT>. When the PPC is loaded, the associated map status bits are stored in the PPC STAT register on <MCMPST>. When any memory cycle is started, its map status bits are recorded (at T100) in the PMA STAT register, either from the PPC STAT or directly from the map. During a write cycle, the PMA STAT is tested to generate a MAP FAULT trap <MCSTCT> if needed.

After the end of a read cycle (at T100), MB STAT is loaded with the map status from PMA STAT as well as the ECC status just computed. MB STAT is then checked whenever the CPU uses the data on MEM, and a trap generated at that time if needed. Useage of the data is detected <MCSTCT> either when a D[MEM] occurs (and MEM has memory data on it) or during the cycle following the gating of memory data onto the EA bus (see <SQEACT>).

5.3.3.1.1. MAP Trap Action by Microcode

When a MAP FAULT trap (MAPTRP) happens, the microcode examines the status preserved in MB STATUS to determine just what kind of cycle failed, and therefor where to find its address. In addition to the MAP FAULT and ECC status bits in MB STAT (described above), MB STATUS contains the bits from the MAPF field of the trapped microinstruction, and possibly 2 of the bits from the MAPF field of the memory cycle microinstruction; these bits are called "trap code" bits. They are documented in the big comment at the beginning of F4KLDF.SLO (on the F4A, F4DEF.SLO). They help specify how to find the failing address, and also how to return to the trapped microinstruction.

The page tables are then consulted to compute a valid mapping for the failing address. If no valid mapping is found, the MAPTRP microcode generates a (macro) Page Fault Trap. If one is found, it is loaded into the Map Memory (3.2.3), and execution is resumed by returning to the cycle that was trapped, after the memory operation has been performed.

5.3.3.1.2. ECC Error Action by Microcode -- ECC Correcction

The Machine normally performs all CPU memory read operations with ECC correction disabled (<SMECC1>); furthermore, memory data is used during some IDISP cycles (6.1.1) via special paths that bypass the corrector. ECC correction is handled in the following sneaky manner:

An ECC trap procedes like a MAPTRP (5.3.3.1.1) to find the failing address. Then, if the error is correctable, ECC correction is enabled and

the failing read re-done. Then correction is disabled again and the trapped cycle re-entered, with the corrected data safely ensconced on MEM.

5.3.3.2. Other Types of Trap1

All the following are allowed to happen only during the first cycle of a new macro instruction (<SMION>).

FBUS Interrupts	I/O devices cause these - see 6.4
PC FLAG Traps	See 6.3
PC=AC	See 6.5
IMPURITY Traps	These occur whenever the preceding instruction has done a store with PC=AC; see <SQTRPO>

CHAPTER 6

MACRO INSTRUCTION SEQUENCING

The main source of the F4's speed is the efficiency of its mechanism for fetching and dispatching on macro instructions. Fetching and dispatching occurr entirely during the execution of the preceding instruction, so that there are no overhead cycles between instructions.

6.1. INSTRUCTION DISPATCHING

The microcode specific to a particular instruction is entered by an Instruction Dispatch (IDISP) cycle, which also provides the mechanism for handling Effective address calculation and certain trap conditions. There are several varieties of IDISP. An IDISP of any type is signaled by two things in the uinst: the IDISP RQ bit is on, and the JCODE is 6. The various types of IDISP are distinguished by bits in the JADR field; the logic is mostly on <SQTRPO, SQEACT>.

The last cycle of every instruction is, by definition, a Principal Idisp or PIDISP cycle; the cycle after a PIDISP is the first cycle of the next instruction. A variation on PIDISP is JDSP, which is used at the end of a jump instruction, or wherever else the instruction sequence is branching.

Other types of dispatch cycles are used to begin execution of an instruction's microcode following special preparatory actions such as address calculation (6.1.1.3), Map Traps (5.3.3.1.1), or Previous Context Execute setup (PXCT).

An IDISP may get its subject instruction from one of several sources -- incoming memory data (UNCORR MEM), the MEM bus, the OBUS, the IR, or even a combination of the IR and one of the others.

We will first discuss the aspects common IDISPs in general (6.1.1), and then we will consider the peculiarities of the main types of IDISP.

6.1.1. General Goals for IDISP Cycles; Instruction Sources

Any type of IDISP must achieve certain goals:

i. In the simplest case (see next two items for the exceptions), the IDISP must branch to a microcode location appropriate for the opcode of the instruction; this is called "dispatching on the opcode".

ii. If the IDISP is the last cycle of a jump instruction that is not branching, or of a skip instruction that is branching, and also in certain special cases, the microcode location branched to must be a trap location appropriate to the circumstances.

iii. Since the new instruction may need to fetch a memory operand during its first cycle, the IDISP must generate the physical address to be used in that case. If this requires an Effective Address Calculation, the IDISP must branch to the microcode for indexing or indirecting. See 6.1.1.3.

iv. The IDISP may need to load all or part of the IR.

v. The PC must end up set to one plus the location of the new

instruction.

vi. Certain contingencies, such as interrupt requests from I/O devices or special conditions in the CPU, are allowed to interrupt the normal sequencing of instructions by means of traps (5.3). IDISP cycles control the times at which some types of these interruption traps are allowed to occur.

An IDISP always operates on four data fields: an opcode, an indirect bit, an index field, and an address field. The last three are collectively refered to as the address part of the data. The four data fields may not all come from the same source during any particular IDISP cycle, but in the simplest case (ordinary PIDISP) they all come from an instruction word.

- If the data is being fetched during the IDISP, the UNCORR MEM bus is the data source. This can happen during PIDISP (6.1.2) or indirect cycles (6.1.1.3).

- If an instruction has been prefetched before a PIDISP, it is normally available on the MEM bus.

- The data may be the result of a calculation in the main data paths, in which case it is taken from the OBUS. This occurs during index cycles (6.1.1.3), when executing from the AC's (6.5), and in other special situations.

- Opcode, indirect bit, and index fields may come directly from the IR (6.1.1.1, 6.1.1.3).

- The indirect bit may even come from the MA (6.1.1.3) !

Whatever the source of the instruction data, in all cases the address field of this data (i.e. the current Partial Effective Address) must appear on the EA during the IDISP; see 6.1.1.3 for details. The logic for this gating depends on the normal microcode fields for controlling the EA, on certain bits in JADR, and also on whether or not a CPU read operation is in progress (logic is on <SQEACT>).

In the case of a PIDISP, the opcode for the instruction must also appear on EA 00:08; for all other types of IDISP the opcode must already be in IR 00:08.

6.1.1.1. Goal i. Dispatching on the Instruction Opcode

If there are no trap conditions (see 6.1.1.2) and no address calculation is required (see 6.1.1.3), any IDISP branches to a microcode location determined as follows: bits 00:03 come from the IDISP REG (<SQCTL2>), bits 04:12 come from bits 00:08 of the opcode, and bit 13 comes from the condition EA=AC. The IDISP REG normally contains binary 0011, so the "dispatch address" for opcode X is 6000 + 2X + A, where A=0 if -EA=AC and A=1 if EA=AC; thus the microcode for each instruction has two entry points, one for the case where its effective address is a memory address and one for the case where its effective address is an AC address.

The dispatch address is generated by the Microsequencer; its fields are assembled onto the MI BA, with the opcode being taken from the EA (on PIDISP) or LCL IR (on all other IDISPs) (<SQCTL2, SQMAD2>), and BRANCHING

is forced true (<SQJMP0>), which gates MI BA to SQ MM AD (see 5), which of course, goes to the MM boards.

Since this path is rather long and slow, however, a sneak path is provided for the case where the instruction is being fetched during a PIDISP cycle. In this case the MMA board looks directly at the incoming instruction on the MEM R DAT lines, bits 00:08, to get bits 04:12 of the MM AD, and also gets MM AD 13 directly from the EA=AC condition; MM AD 00:03 are taken from SQ MM AD 00:03 as usual (see <MMAMAD>).

Another sneak path is provided for the opcode bits in the case where the instruction is coming from MEM during a PIDISP (see 6.1.1); in this situation the BO board gates MEM 00:08 onto the EOBUS, and the MMA takes the opcode from there. However, this logic is currently disabled, as it has proven not to allow any shortening of the cycle (because there is nearly always a memory store in progress in this case).

6.1.1.2. Goal ii. IDISP Trapping

Mainly in order to save time during branch instructions (i.e. macro Jumps and Skips), IDISP's are conditional; if no address calculation is required (see 6.1.1.3), but COND (the microcode branch condition; see <SQJMP0, SQJMP1>) is false, the dispatch on the opcode is overridden by a TRAP0 (see 5.3). The trap location is determined by 3 bits of the JADR field.

6.1.1.2.1. IDISP Traps on JUMPS

The microcode assumes that every JUMP instruction will branch; the last cycle of the microcode is a dispatch (a JDSP - see 6.1.2.2) to the target instruction. However, the dispatch cycle is conditional on the appropriate things for the particular JUMP, and therefore a TRAPO happens if the JUMP did not branch. The trap microcode then fetches the and dispatches on the instruction following the JUMP.

6.1.1.2.2. IDISP Traps on SKIPS

The microcode assumes that SKIPs will fail to skip; the last cycle of the microcode is a dispatch on the following instruction. This cycle is made conditional, however, and a TRAPO occurs if the SKIP did skip. The Trap uinst just does a normal instruction fetch and dispatch to get to the next instruction.

6.1.1.2.3. Other IDISP Traps

A few instances of JDISP in the microcode are conditioned on -EA=AC, and thus trap if branching into the AC's. The trap microcode then takes the usual steps (see 6.5).

6.1.1.3. Goal iii. Effective Address Calculation

The first uinst of each instruction expects to find its effective address in the MA, and the physical version in the PMA. Therfore, every IDISP places on the EA the best approximation to the effective address available at the time (which is called the current partial effective address). For the PIDISP which begins each instruction, the partial effective address is

just the right half of the instruction; if no address caculation is called for, this is the effective address, and the PIDISP dispatches on the opcode.

If indexing or indirecting is indicated, however, a TRAPO (5.3) takes us to microcode for performing that operation (usually a single uinst); this microcode ends with another IDISP cycle (not a PIDISP), which once again dispatches on the opcode if no further address calculation is needed, but takes a TRAPO if some is needed. Thus the address calculation is sequenced by a chain of IDISP's which generate TRAPO's, terminated by an IDISP which finally dispatches on the opcode. (It is important to remember that a TRAPO does not abort the cycle trapped out of.) At the end of every cycle of this process, of course, the map places the physical version of the partial effective address in the PMA (3.2.3).

The details of an address calculation can be rather complex, especially in the case of extended addressing. General comments follow, but for a detailed understanding the reader must study

- the DEC hardware manual's description of effective address calculation
- the Foonly "Address Calculation" flow charts
- the trap microcode at the beginning of F4KLDF.SLO (F4BASE.SLO for the F4A)
- the Address Engine description (3.2)

- the logic on <SQTRP0, SQTRP1, SQEACT, SQACSL, SQAXCT, MXEXT, MXSEG>

To summarize, then. The TRAP0's for indexing and indirecting are individually enabled by bits in the JADR field of the IDISP; this permits the avoidance of endless redundant traps. There are multiple trap locations for each kind of trap, to distinguish things like global versus local addressing and memory versus AC addresses. Index traps have higher priority than indirect traps.

6.1.1.3.1. Indexing

An index trap occurs if the current index field is non-zero. Except during Global Indirect Cycles (see 6.1.1.3.2), the index field is EA 14:17 (see <SQACAD>), and the trap initiates a Local Index Cycle. During Global Indirect Cycles the index field is EA 02:05, and the trap takes us to a Global Index Cycle. An index cycle always adds the indicated AC to MEM to produce its address field.

During a Global Index Cycle (and always on the F4A) this add uses 36 bits of both operands and loads the result into the whole MA. During a Local Index Cycle (on the F4X) bit 18 of MEM is extended through the left half as a sign bit, and the resulting quantity is added to the full AC(see <B0AM1, B1AM0); the right half of the result goes to the MA, but the left half of the MA is not loaded unless the result is Global (see 6.1.1.3.3).

6.1.1.3.2. Indirecting

An indirect trap takes place if no index trap is being requested and the indirect bit is 1; the indirect bit is MA 13 during a Local Index Cycle, MA 1 during a Global Index Cycle, EA 1 during a Global Indirect Cycle, and EA 13 otherwise.

If -EA=AC, the indirect trap goes directly (sorry) to an Indirect Cycle, which fetches the pointer word (placing it in MEM and IR 13:35), then dispatches according to the new index and indirect fields. If EA=AC, the indirect trap goes instead to a preliminary cycle which gets the pointer out of the AC into MEM and then branches to the Indirect Cycle itself; this preliminary cycle is needed because the pointer must end up in both IR 13:35 and MEM, and there is no way of loading both of those from the OBUS in one cycle.

6.1.1.3.3. How Extended (30 Bit) Addresses Are Generated

At the end of each address calculation there must be a 30 bit virtual effective address, the leftmost 12 bits of which are called the E SECT. Refer to the section on the Address Engine (3.2) for details of the hardware functions mentioned in the following description of how the E SECT is generated.

EA GLOBAL is forced false during a Principal IDISP, thus causing MA GLOBAL to be off at the beginning of the address calculation. During indexing and indirect cycles, the left half of the MA (MA LT) is loaded exactly whenever EA GLOBAL is true, i.e. whenever the current cycle is

producing a 30 bit result. MA GLOBAL therefore stays off until an extended address is generated, after which it remains on, while MA LT thereafter contains the section of the most recently generated global address.

As long as MA GLOBAL is false, E SECT is taken to be equal to PC SECT, and the MAP will use the PC SECT entry in SEGMEM for all references which have -EA GLOBAL; see 3.2.2.2. Once MA GLOBAL is on, MA LT contains E SECT, and the MA SECT entry in SEGMEM will be used during -EA GLOBAL cycles. In all cases, of course, EA GLOBAL causes the MAP to take the section from the EA.

6.1.1.4. Goal iv. IR Loading

During every IDISP, the partial effective address must be gated to the EA long enough before the end of the cycle for the map to translate it to a physical address. Since the IR is fed by the EA, appropriate portions of the IR can be loaded at the end of the cycle. In particular, a PIDISP loads the whole IR, since the entire new instruction is on the EA (6.1.2). Indirect cycles and index cycles (6.1.1.3) load IR13:35 and IR RT respectively, thus keeping the IR updated to the latest partial effective address.

6.1.1.5. Goal v. Updating the PC

Normally a PIDISP needs to increment the PC by one (the exception is JDSP, 6.1.2.2). This is easily accomplished by gating PC to AA input and loading PC from AA at end of cycle (3.2); logic is on <SCEACT>. Other types of IDISP usually need to suppress incrementing the PC (as during an

XCT dispatch), and a bit in the JADR allows this.

6.1.1.6. Goal vi. Controlling Interrupt Conditions

PI's (6.4), PC FLAG traps (6.3), Impurity Traps, and PC<20 traps (6.5) are all allowed to happen only at end of instruction, and are therefore sampled by PIDISP. They all generate TRAP1 (5.3) during the next cycle (the PIDISP+1 cycle).

6.1.2. The Principal Instruction Dispatch (PIDISP or JDSP Cycle)

A PIDISP is the last cycle of every instruction, and therefore marks the beginning of each new instruction. Since no effective address calculation has yet been done (6.1.1.3), the partial effective address is just the right half of the instruction. Since the whole instruction shows up on the EA, all of the IR is loaded. The PC is always set to one plus the location of the new instruction (6.1.1.5).

6.1.2.1. Ordinary PIDISP

At the end of non-JUMP ixs, the PIDISP gates the PC to AA, thus getting it incremented (6.1.1.5). The dispatch data may have been prefetched and be on MEM (6.2), or the PIDISP may do an IFRQ (3.1), in which case the data is taken from UNCORR MEM.

6.1.2.2. JDSP - JUMP Dispatch

The IDISP at the end of a JUMP uses a bit of JADR to cause SQ JUMP CY, which defines the dispatch as a JDSP (<SQTRPO>). During the last cycle of a JUMP type instruction, the location of the target instruction is in the MA. The JDSP accordingly gates the MA to AA, thus loading the PC with MA+1 (6.1.1.5). However, there is an important special case; if COND is false (ie the JUMP is failing), logic on <SQAECT> suppresses loading the PC, which therefore continues to point to the location after the JUMP; see 6.1.1.2.1).

The JDSP does a DFRQ to fetch the target instruction, but logic on <MCCTL1> causes IF REQUESTED, since it really is an instruction fetch.

6.1.3. Other IDISPs

Non-principal IDISPs are used during address calculation (6.1.1.3), XCT instructions, and in resuming an instruction after a map trap (5.3.3.1.1). In all cases, the opcode data for the dispatch comes from IR 00:08, although the address part of the dispatch data may come from any source.

6.2. INSTRUCTION PREFETCHING

Every instruction is fetched during execution of the previous instruction, usually during a cycle when the previous instruction is doing something useful which does not involve a memory rx. For example, ADD prefetches its successor during its second cycle, when it is storing its result into an AC, while MOVEM prefetches during its first cycle, when it is getting the store data out of an AC. Jump instructions are special, of course; they have two potential succssors, the explicitly addressed Target Address, used if the jump branches, and the usual normal successor in the location following the jump.

Since the prefetch of a normal (non-branch) successor may occur during the first cycle of an instruction, the normal successor address must be generated and mapped before the instruction begins; this is done during the first half of the PIDISP cycle [6.1.2] which ends the preceeding instruction, and the resulting (physical) address is stored in the PPC until it is needed.

6.2.1. Ucoding Rules for Prefetches

No register is provided just to hold prefetched instructions. If the prefetch cycle coincides with the PIDISP cycle, the incoming instruction does not need to be stored, as it is routed directly from the UNCORR MEM bus to its various destinations (see 6.1.2). Otherwise, the prefetched instruction must be available on the MEM bus when the PIDISP occurrs. This implies that the prefetch must follow all data fetches of the current instruction, and must follow or coincide with the last loading of HOLD (see

rules for MEM bus in 2.1).

6.2.2. Prefetch Address Generation

During the first half of the PIDISP that starts an instruction, the physical address of that instruction's successor is generated and placed in the PPC. This entails, first of all, generating the virtual address of the successor on the EA, where the map can use it. The control for this is on <SQEACT>. There are two cases, depending on whether the PIDISP is a Jump Cycle [6.1.2.2] .

6.2.2.1. Prefetch Address During Ordinary PIDISP

If we are not jumping during the PIDISP (6.1.2) , then the PC contains the address of the instruction to which we are dispatching (which is, a fortiori, one plus the address of the preceding instruction). The normal successor address to this instruction is thus one plus the contents of the PC. Therefore, during the first half of the PIDISP, the PC is sent to the AA, incremented, and put on the EA.

6.2.2.2. Prefetch Address During JDSP

During a JDSP, the MA contains the address of the instruction to which we are dispatching (see 6.1.2.2). The normal successor address to this instruction is thus one plus the contents of the MA. Therefore, during the first half of the PIDISP, the MA is sent to the AA, incremented, and put on the EA.

6.2.3. Prefetch Address Mapping

At about T150 of each PIDISP there is a rising edge of MC C3 MAP CLK <MCCLK2, MCCTL2> which clocks the map output registers as usual (see 3.2.3). At the same time MC PPC CLK loads the MAP MA output, which is the physical prefetch address, into the PPC, from which it is used when the next IF cycle is started (3.1). Note that this IF cycle may start at the end of the PIDISP, in which case the PPC has only stored the prefetch address for part of one cycle, or it may be hundreds of cycles until the next IF (as when one or more Map Traps intervene -- 5.3.3.1.1).

6.3. PC FLAG TRAPS

Instruction sequencing may be interrupted by traps caused by one of the two PC Trap Flags. These bits are set by microcode as a result of either explicit checks (in the case of PDLOV) or TRAP1s (in the case of AROVs). They cause a TRAP1 at end of instruction, and the microcode at <PCTRAP:> processes the trap condition and generates a macro trap.

On the F4A, instead of a KL type macro trap, the microcode generates a PI (see 6.4).

6.4. PIs

The PDP10 Priority Interrupt (or PI) system is entirely emulated by microcode on the F4; the microcode is in F4KLAP.SLO. The emulation is straightforward, and involves the following:

- A routine <PIGEN:> which generates an interrupt request on a specified

PI channel

- A routine <PI-CHECK-RQS:>, called whenever the status of the PI system changes, which causes a macro trap for the highest priority existing request

- A routine <PI-DISMISS:> which leaves a PI

I/O PIs originate as FBUS interrupts, which cause a TRAP1 at end-of-instruction. The hardware automatically places the address of the interrupting device on ION 0:5, thus selecting the device and selecting the block of AMEM assigned to that device. The trap code just dispatches via location 0 of the selected AMEM, which contains, by convention, the interrupt vector address for that device. If the trap code for the device decides to cause a PI, it goes to <PIGEN:>.

6.5. RUNNING IN THE ACs

The F4 must use heroic measures to implement this piece of compatibility with the PDP-6. The main trick is that whenever we are running with PC<20, we set both PC trap flags; this causes a TRAP1 at each end-of-instruction, which takes us to microcode that fetches the next instruction from the AC and dispatches it. This microcode also checks to see that PC is really still <20. The overhead for this is about 11 cycles per instruction.

The remaining part of the magic is to detect entry into the ACs, so that we can set the trap flags. There are several cases:

i. JUMP instructions, if they find they have an E<20, go to <JUMPAC:> which sets the flags and then re-dispatches the JUMP while forcing EA to be greater than 17, but leaving the true value of E in the MA. The JUMP then executes normally, but we get the desired TRAP1 at the end.

ii. POPJ is special, since its target is not E. It does a conditional IDISP , taking a TRAP0 if EA<20, which gets it to code for setting the trap flags (see 6.1.1.2).

iii. IDISPs which return from map traps or I/O traps are also conditional on EA<20.

iv. Context changing JRSTs check explicitly for PC less than 20.

6.6. IMPURITY

To save writers of degenerate code which plays with itself, a comparator always checks during a store cycle to see if MA=PC. If so, a ff is set (see <SQTRP1>) which causes a TRAP1 after the end of the instruction. The trap code just decrements the PC and resumes execution, wasting as much time as possible in the process.

CHAPTER 7

MICROCODING

Writing correct and efficient microcode requires thorough knowledge of the operation of the hardware. The microcoder should therefor study and understand all the other chapters of this manual, at the level of data path diagrams and flow charts. Reference to logic drawings should not be necessary.

At all times when studying this chapter the reader should refer to the file containing the definitions of the microcode word's fields and filed values. This is F4KLDF.SLO for the F4X and F4DEF.SLO for the F4A.

7.1. GENERAL

Most fields of the 88 bit ux word are independent of each other, but some fields have multiple uses.

7.1.1. Notable Restrictions

Multiple useage of JADR implies that one cannot do any kind of jump while:

- globally addressing AMEM; this rules out even POPJ (since MI BA is selecting JADR, and therefor cannot select JMEM).
- reading from JMEM, as in a SPEC[POP

(since MI BA is selecting JMEM)

using D[LIT]

doing any IDISP]

Since IR and PC are loaded from the EA, they cannot be loaded during the cycle before a memory cycle unless they are loaded with the address of the memory operation.

7.1.2. Fields of the Microinstruction

ENB-EA-LEFT	Obsolete. Must be 1 on F4A; undefined on F4X
MEM-WAIT	Signals a wait-for-DF ; (5.2.1)
AA-FROM-MA	(3.2)
ALU1-SEL	Select alternate 2901's (2.4)
IF-RQ	Instruction Fetch RQ
DF-RQ	Data Fetch RQ
CYLEN	Controls length of cycle
COND	Selects condition for Micro and Macro branching
DEST	Selects one of many destinations for OBUS data. 40:57 stores (local) AMEM (2.2).
JADR	Jump address; also global amem addresses (2.2) and control bits for IDISP (6.1.1)
J-CODE	determines what type of jump (if any) --
EA-FROM-OBUS	see 7.1.3
MA-STB-FIELD	load the MA
AR-STB-FIELD	load the AR
D	Select data for DBUS. 40:57 address the (local) AMEM (2.2). Addresses less than 40 address the EOBUS.
MAPF	A multiple useage field. Recorded on MAP FAULTs and ECC errors (5.3.3.1.1) to provide information the trap microcode (no direct hardware effect). Controls end

connections for ALU shifting (2.5). Provides the device subselect for I/O operations (4).

ACSEL — Selects the A and B register addresses for the 2901 (2.3). In the definitions, "MA" means MA32:35, "AC" means IR09:12, "XR" means IR14:17.

source — This is the 2901 "source" field.

ALU-DST — This is the 2901 "destination" field.

carry-in — Inserts a carry into bit 35

alu — Not a physically separate field -- just an umbrella for definitions which specify both sources and operations.

OP — The 2901's opcode field.

SPEC — Special Functions -- see definitions for details.

7.1.3. Funny Features

Multiple Destinations
Notice that one can simultaneously load MA, AR, Q, and any DEST field destination.

EA-FROM-OBUS
This is seldom mentioned explicity, as it is implied by such things as DEST[PC

. One should be aware of it, however. Its default state is 0, which causes EA from AA.

MA+1

The conjunction of LD MA and SEL AA FROM MA (during non-IDISP cycles) causes the AA to increment; this provides an MA INCREMENT without using the SPEC field or main data paths. (Note that the PC can also be loaded with the result.)

Traps force the Last Condition ff to be on. This is needed in order to branch out of a trap location, since the COND field is missing (5.3); the trick is always to use an "LC" type jump (7.1.4).

]

7.1.4. JUMP Types

The CONT value of J-CODE forces .+1 independent of the state of COND.

Jump types whose names start with "LB" (ie Low order Bit) always jump. They must have an even target address; if COND is false, they go to the target address, otherwise they go to the following (odd) location.

Jump types whose names end with "LC" make their branching decision based entirely on the Last Condition ff, which records the state of COND from the previous cycle.

CHAPTER 8

SOME DATA FORMATS

8.1. MAP Write and Read Data

	MP-2	MP-4,5,6
WRITE:		
MAP-TAG		
MAP TAG 03 (1 = Force Hit)	03	03
Section #	none	06:17
Context (low order bits)	04:17(complemented)	18:26
Page #	18:26(complemented)	none
MAP-MA		
Physical addr.	13:26	13:26
MAP ACCESS 0:3	09:12	09:12
CONTEXT	04:19	15:26
READ:		
MAP-TAG		
MAP TAG 03 (1 = Force Hit)	03	03
Section #	none	06:17
Context (low order bits)	04:17(complemented)	18:26
Page #	18:26(complemented)	none

MAP ADR 04:11	N/A	27:34
MAP-MA		
Physical addr.	19:26	15:26
-MEM SEL 0:7	28:35	03:10
MAP ACCESS 0:3	07:10	32:35
MAP MA PARITY	11	11
MAP TAG PARITY	12	12
MAP ADR 00:03	N/A	27:30