Wildflower Manual

by B. W. Lampson

August 18, 1978 4:44 PM

This document describes the operation of a small processor called Wildflower, from the microprogrammer's point of view. The machine is intended for use in a workstation. It has provision for efficient execution of Mesa code in a manner essentially compatible with other OIS processors, and for controlling a display with 1024x808 visible points refreshed at 36 Hz, a Shugart SA-4000 disk, an Ethernet connection and an 8080 bus. Ordinary Mesa instructions execute slightly faster that 1 Mips with the display running. Main memory can be 128k or 256k words, implemented with 64k RAMs; there is parity but no error correction. A 128k Wildflower is implemented with the equivalent of 200 16 to 20 pin IC packages, including all the input/output controllers.

Warning: this is a draft only. There are errors, inconsistencies and omissions.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

1. Introduction

The purpose of this document is to describe the operation of a small processor called Wildflower, from the microprogrammer's point of view. The machine is intended for use in a workstation. It has provision for efficient execution of Mesa code in a manner essentially compatible with other OIS processors, and for controlling a display with 1024x808 visible points refreshed at 36 Hz, a Shugart SA-4000 disk, an Ethernet connection and an 8080 bus. Ordinary Mesa instructions execute slightly faster that 1 Mips with the display running. Main memory can be 128k or 256k words, implemented with 64k RAMs; there is parity but no error correction. A 128k Wildflower is implemented with the equivalent of 200 16 to 20 pin IC packages, including all the input/output controllers.

1.1 Conventions and notation

Numbers in this document are decimal unless followed by B; thus 10-12B.

Names of registers, busses and microinstruction fields are in small caps (STKP, IOIN, A). Names of functions encoded in the microinstruction's F1 and F2 fields are in bold (**MapRef**, **Y lcy 8**).

Bits in registers are numbered from the most significant bit (0) toward the least significant bit. Fields within registers are given by following the register name with a dot and a pair of numbers: Q.2-4 describes the 3 bit field of the Q register beginning with bit 2 and ending with bit 4 inclusive. Q.2 is short for Q.2-2.

The symbol "_" is used to mean "is replaced by." Thus Q.2-4 _ 2 means that the 3-bit field of Q including bits 2, 3 and 4 is replaced by the bit values 0, 1 and 0 respectively. The symbol "=" is used to indicate an equality test.

Certain prefixes are used with standard meanings: a prefix "p" means "pre", i.e. one cycle early; a prefix "r" means reset.

Uses of "in" and "out" always refer to the processor. Thus IOIN is data coming from an io device, and MEMOUT is data going to the memory.

Memory is by convention divided into 256-word pages. Page *n* thus contains addresses 256**n* to 256**n*+255 inclusive.

2. Control section

Time is divided into units called *clicks*. One click corresponds to one main storage reference. During one click, exactly three *cycles* occur; in each cycle, one microinstruction is executed. A cycle lasts for 100 ns, and a click therefore lasts for 300 ns.

2.1 Tasks

The machine supports five tasks, each with its own micro-PC: emulator, disk, display, ethernet and uBus. Clicks are numbered modulo 8; consecutive clicks numbered 0..7 make up a *round*, which lasts 2.4 us. Each click is a *potential* click for exactly one of the input-output tasks; the choice of task depends on the click number, and is made according to the *schedule* below. If this task is requesting service, its potential click becomes a *service* click, and it executes three instructions during the click; if not, the click becomes an *emulator* click, and the emulator executes three instructions. The schedule is:

Click number	*Task*
0	display
1	disk
2	display
3	disk
4	ethernet/uBus
5	display
6	disk
7	ethernet/uBus

This assignment is for a 40 MHz display; the display gets serviced three times in a round, and gets 32 bits each time, for a bit rate of 32*3/2.4=40 bits/us=25 ns/bit. To paint a line of 1024 bits takes 32 clicks. Adding 7 clicks (5.1 us) for horizontal retrace gives 13 rounds or 31.2 us per scan line. Now multiplying by 875 gives 27.3 ms per frame, which corresponds to a refresh rate of 36.63 Hz. The disk gets serviced at least every 900 ns, which allows the Shugart SA-4000 disk, with a data rate of one byte every 1.1 us, to have one byte data paths. The ethernet gets serviced twice every 2.4 us, which is slightly more than double what is needed for its data rate of one byte every 2.7 us, but avoids latency problems. The uBus gets any clocks the ethernet doesn't want, which allows it a peak bandwidth of 3 Mbits/second when the ether is running, double that when it isn't, and a maximum latency of 13 clicks (since the ether might take two consecutive clicks), or 4 us. All of this is far in excess of its anticipated requirements. Thus all input-output data paths are one byte wide, except for the display which loads directly from the 32-bit memory outputs.

2.2 Microinstruction addressing

The machine has 2048 words of micro-instruction (I) memory, addressed by a 12 bit next instruction address (NIA.0-11); the extra bit allows for each expansion to 3072 or 4096 words of I memory. NIA.3-11 are supplied by an 11 bit field in each instruction called INIA.1-11, with possible modifications discussed below; this means that there is no block or page structure imposed on the I memory.

Branching is done by oring into INIA.11 of the *next* instruction. If INIA of the next instruction is even, this sends control to INIA or INIA+1, depending on the failure or success of the branch respectively. If INIA is odd, the effect of the branch is cancelled. For example:

```
100     ALU=0
101     GOTO[200]
```

will send control, after executing 101, to 200 if ALU#0 in 100, to 201 if ALU=0 in 100.

There are several kinds of multi-way branch. **Ydisp4** ors Y.12-15 into INIA.8-11; it is illegal if Y depends on data from SU or MEMIN. **X12-13Disp** ors X.12-13 into INIA.10-11. **NextMacro** substitutes IB.0-7 for INIA.4-11, which must be 0; it is intended for dispatching on a Mesa bytecode, and must be executed in the second instruction of a click. It is illegal to do any other branch or dispatch together with a **NextMacro** (see section 5.1).

There are also two 4-bit subroutine link registers (LINK1 and LINK2), intended for the emulator's use, which can be loaded from NIA.8-11 (call) and can be ored into INIA.8-11 (return). Both call and return must be done one cycle in advance to work properly for subroutine linkage, and to emphasize this they are called pcall1/2 and preturn1/2; thus:

IA	NIA	instruction
77	100	pcall1
100	200	goto(200)

```
...
200     201     nop
201     202     preturn1
202     101     goto(1)
```

will save 100 in LINK1 and will execute the instructions in the order shown. The next instruction executed after 202 will be 101. Note the responsibility of the programmer to arrange for suitable alignment of the instruction to be returned to, and for oring in enough bits in the final instruction of the subroutine (just a 1 in this case).

Any form of branch, dispatch or return may be executed in any instruction of a click, except for an IB dispatch. A return may be combined with a branch or dispatch, in which case both modifiers are ored into INIA.

2.3 Microinstruction format and summary

The microinstruction has 44 bits, divided as follows:

4	A (addresses R and C)
4	B (addresses R)
3	MUX control
3	ALU control
3	LOAD control
1	CIN (carry in for arithmetic; also addresses SU)
1	ENU (enables U (or S if ENS) for reading onto X)
1	ENS (enables S for reading onto X, if ENU is set; enables SplitAdder)
1	ENW (enables W for reading onto X)
1	SU_ (if set, writes S if ENS, otherwise U)
4	F1 (addresses W and U; supplies the SC value; see below for other uses)
4	F2 (see below)
11	NIA (next instruction address)
3	unassigned

The F1 and F2 fields specify a variety of miscellaneous functions, not all of which are yet defined. In particular, additional branch conditions will be defined. The current, tentative encoding of these fields is as follows ("*" means that the function is meaningful only for the emulator):

F1

Addresses U (if ENU)
Supplies SC value (enable TBD)
Addresses W (if ENW): **_Y lcy 8** (only if **Y_A**), **_MEMIN**, **_IB***, **_IBL***, **_IBH***,
    _IOIN, **_STKP**
Any of the uses above disables the uses below
STKP operations: **push*** (STKP_STKP+1); **pop*** (STKP_STKP-1); **STKP_***
IB_*
cycle
pEmuU
Disk operations: **KData_**, **KCtl_**, **rKReq**
Ether operations: **EData_**, **ECtl_**, **rEReq**
Display operations: **Display_**, **rDReq**
uBus operations: **UData_, UCtl_, rUReq**

F2

MEMOUT_
Dispatches: **NextMacro***, **AlwaysNextMacro***, **YDisp4**, **X12-13Disp**
Subroutines: **pCall1***, **pRet1***, **pCall2*, pRet2***
Branch conditions: **F<0**, **Y<0**, **X<0**, **Xodd**, **F=0** (logic only), **Carry, PageCarry, pCt=Emu, Refill*, EException**
IBPTR_1*, **IBPTR_2***, **IBPTR_3***, **IBPTR_4***
RH_*?, **MapRef***?
CIN=PC16*
_KData, **_KStatus**
_EData, **_EStatus**
_UData, **_UStatus**

2.4 Initialization

The machine is initialized by an asynchronous external reset signal. While this signal is true, each task in turn is started at 0. The tasks are run in a fixed order during each round: Emulator (4 times), Disk, Ethernet, Display, uBus. They get themselves sorted out by testing the **pCt=Emu** branch condition in the first cycle of the click. If this condition is true, the emulator task is running, and the microcode writes 0 into an R register RT and sends control to the start of the emulator's microcode. Otherwise it dispatches on RT and increments it by 1. As a result, the four non-emulator tasks will dispatch on 0, 1, 2 and 3 respectively (the emulator must shake the dispatch). As long as reset remains true, these dispatches will be ineffective, since each click is forced to start again at 0. When reset finally is removed, however, the dispatches will take effect and initialization will be complete. Suitable microcode to accomplish initialization is:

```
IoVec:   Dispatch table mod 4 [Kinit, Einit, Dinit, Uinit]
Start:   AT[0], pCt=Emu?, UT_RT
         goto[Io, Emu], sink_UT, Xdisp4
Io:      goto[IoVec], RT_RT+1
Emu:     goto[EmuInit], shake[7], RT_0
```

3. Arithmetic section

The arithmetic data paths are shown in figure 1. The processor has 16 fast R registers, and 256 slow SU registers, divided into S registers and U registers as described below. There is also a special-purpose Q register which is intended for the exclusive use of the emulator. The R registers are addressed directly by the 4 bit A and B fields of the instruction; two locations, whether the same or different, may be read simultaneously. The U registers are addressed by F1, CIN, and the task number; thus there are 32 U registers per task. Note that since CIN is part of the U address, care must be taken in writing U using F_A and simultaneously doing arithmetic in the ALU, since CIN will affect the arithmetic. In all other cases of U references, a logic operation is being done and hence the value of CIN is ignored by the ALU. A function **pEmuU** allows any task to address the emulator's U registers instead of its own in the *next* instruction; it is illegal in the last instruction of a click (since it would affect the next task).

The S registers are intended for the Mesa stack, and are addressed from the 4 bit stack pointer (STKP) register; there are functions to increment and decrement STKP, and it can be loaded from Y or read onto W. CIN also contributes to S addressing, and there are in fact two separate sets of 16 S registers, selected by CIN. There is only one STKP, however. This is not intended to be a useful feature. See section 5.2 for more details on S.

The fact that SU is slow means that an SU register cannot be the operand of an arithmetic operation, or be loaded from the results of an arithmetic operation, or from MEMIN.

The ALU has two input multiplexors which select its two operands under control of the MUX field of the instruction, as shown in the figure. It is capable of three arithmetic and five logical operations, under control of the ALU field, and its results can be put on the Y bus, or written directly into Q, or written into R directly or with a possible left or right shift by 1 bit, under control of the LOAD field. Slow sources (SU, MEMIN, and IOIN) may not be the operands of arithmetic operations.

The R register addressed by B may be written from the ALU output F, possibly shifted left or right by 1; this is legal whether or not that register is used as an operand in the same instruction. Q may be written from F, or from the old contents of Q shifted left or right by 1. The SU register addressed may be written from the Y bus, but not with the result of an arithmetic operation or with MEMIN, and it may not be both read and written in the same instruction.

Possible values on the X bus are the addressed SU register (selected by ENU), the contents of the W bus (selected by ENW), an 8-bit constant (specified in a manner TBD) or a four-bit small constant SC taken from F1 (enable TBD). The constant occupies X.12-15, and X.0-11 are zero.

Possible values on the W bus (with 0 in any unmentioned bits) are

Y lcy 8: legal only if the LOAD field specifies B_F & Y_A.

MEMIN: a slow source, i.e. no arithmetic and no writing into SU.

IOIN: data coming from the io device selected by F2 (*it would be nice if we could find a way to avoid the use of F2 here, perhaps by using the task number and some random piece of information like CIN*).

IB: 8 bits coming from the emulator's instruction buffer in bits 8-15; see section 5.1.

IBL, IBH: the 4 lsb or 4 msb of the instruction buffer, in bits 12-15.

STKP: the 4-bit stack pointer, in bits 12-15.

3.1 Shifts and cycles

The LOAD field of the instruction allows single and double shifts of Q and an R register, as shown in figure 1. Normally, zeros are shifted in at the ends, and on a double shift F.15 and Q.0 are connected. The **cycle** function changes a double shift to a double cycle, connecting F.0 and Q.15 as well as F.15 and Q.0. It changes a single left shift to shift PC.16 (see section 5.4) into R.15, and a single right shift to shift Q.15 into R.0. *Need a picture for this.*

4. Memory

A memory reference is always started on the first instruction of a click, using the contents of Y as the 16 lsb of the memory address MAR.0-19. The 4 msb of MAR come from an auxiliary memory called RH.0-3, which is addressed by A. RH can be loaded from X.12-15 using **RH_**. If ALU specifies an arithmetic operation, its carry is added to RH to produce the 4 msb of MAR.

If the memory operation is to be a store, the data to be stored must be supplied on Y, and **MEMOUT_** must be specified, during the second instruction of the click. Otherwise the operation is assumed to be a fetch.

Data from the memory is available on MEMIN during the third instruction of the click unless it was a store. The data can be put onto W and thence onto X, and thence sent to the ALU. Only a logic operation is possible, and if Y_F is specified the Y data may not be sent to SU.

During a fetch the memory actually supplies 32 bits from the even-odd word pair addressed by MAR. The 32 bits can be sent to the display (using **Display_**) or to the instruction buffer (using **IB_**) in the third instruction of the click. This can be done independently of putting MEMIN on W.

If **MapRef** is specified in the first instruction, rather than using RH,,Y.0-15 for MAR, 377B,,RH,,Y.0-7 are used instead. This makes it convenient to reference a *map* stored at the top of memory, on the assumption that the page size is 256 words. The intended use of this facility is illustrated by the following microcode for making a mapped reference from the emulator. The format of a map entry is assumed to be

0-7	8 lsb of real page number
8	unused
9-11	referenced, dirty and write-protected bits (not present=dirty and wp)
12	present and references (i.e. read OK)
13	present, referenced and dirty (i.e., write OK)
14-15	2 msb of real page number

T*i* are R registers:

```
MAR_Q_address, MapRef
T1_Q and 377B
T2_MEMIN, RH[T1]_MEMIN, X12-13Disp

-- now proceed with the reference as usual.
-- the first instruction takes the dispatch on the fault bits supplied in the third cycle of the
previous click. For a read, MapFault should be 1 mod 3, resulting in a 2-way branch on
need for special action; for a write, MapFault should be 2 mod 3 to get the same effect. If
action is required, it is necessary to decode T2.9-11 to figure out what has happened: page
not present, write protect fault, set referenced bit, set dirty bit. Q has the 16 lsb virtual
address; the 4 msb must be retrieved from some know place.
MAR_T1+SplitAdder[T2,0], goto(MapFault)
. . .
```

5. Mesa emulator support

The instruction buffer (IB) and the stack (S, STKP) and their associated functions, as well as the SplitAdder and **Cin=PC16** functions, are provided to speed up the emulation of Mesa bytecodes.

5.1 Instruction buffer

There is a 5-byte instruction buffer (IB) which holds the next few bytes of the Mesa instruction stream. Its operation is somewhat complicated; this paragraph gives a birds-eye view, and the rest of this section supplies precise details; figure 2 contains a detailed example. Bytes 1..4 can be loaded from a storage doubleword with the **IB_** function. Associated with the five bytes is a pointer register IBPTR. Its possible values are 0..6, and for the values 0..4 the corresponding byte of IB appears on the IB bus. If IBPTR>4 the value of IB is undefined. The IB bus can be read onto W or used for a dispatch; in either case IBPTR is advanced to point to the next byte. Either the left or the right 4 bits can also be read onto W.

The five bytes are called IB0, IB1, IB2, IB3 and IB4. The **IB_** function must be executed in the third cycle of a click, immediately after **AlwaysNextMacro**; it does

```
IB1 _ MEMIN.0-7
```

IB2 _ MEMIN.8-15
IB3 _ MEMIN.16-23
IB4 _ MEMIN.24-31
IB0 _ IB if IBPTR=0, otherwise undefined

There are also four functions **IBPTR_*i***, *i* IN [0..3] which set IBPTR as well as loading IB from MEMIN as shown.

If IBPTR=*i* (*i* IN [0..4]), then IB*i* will appear on IB; otherwise the value of IB is undefined. The idea is that IB1-4 are the four bytes of a doubleword, and IB0 is an overflow byte which is used to smooth out the fetching of another doubleword, as described below.

Either IB, or IBH (IB.0-3), or IBL (IB.4-7) can be read onto W right justified. Zeros appear in bits not supplied from IB. When IB (but not IBL or IBH) is read, IBPTR is *advanced*, i.e. incremented by 1, so that if IB*i* was on IB before, IB(*i*+1) will be on IB afterwards.

The **NextMacro** function can be used to dispatch on IB. If IBPTR <=2, then IB replaces NIA.4-11 (which must be 0), and IBPTR is advanced. In this case, at least two bytes remain in IB, so that a two or three-byte instruction can retrieve its operand bytes without having to worry about finding the buffer empty (lines 4, 6, 7, 9, 1, 12, 16 in figure 2). If IBPTR>2, then 0 or 1 is ored into NIA, depending on whether IBPTR<5 or not, and IBPTR is not advanced (lines 2, 3, 10, 11, 15, 18, 19). The **AlwaysNextMacro** function dispatches in any case, and advances IBPTR mod 3; thus 3 is advanced to 0 and 4 to 1. The idea is that control goes to 0 or 1 when IB needs to be refilled. The refill code will fetch the next doubleword in the first cycle, repeat the dispatch with **AlwaysNextMacro** in the second cycle (except when IBPTR was 5; see below), and load IB from MEMIN with **IB_** in the third cycle (lines 5, 8, 17). Since refill may occur with two bytes left in IB, there may still be one byte left after the dispatch; this last byte is saved in IB0 by **IB_** (lines 8, 17).

Since refill occurs unless there are at least three bytes in IB, there will always be at least one byte left when the next macro is started, except in the case of a three-byte instruction which occupies the last three bytes of a doubleword; hence is is always possible to overlap the refill fetch with an instruction dispatch, except in this case, which results in IBPTR=5 at the time of the **NextMacro**, and hence in a dispatch to 1 (line 14). In this case the **AlwaysNextMacro** dispatch is not possible, and the next macro cannot be dispatched until the following click (line 15). The resulting lost click is only paid on 1/4 of the 3-byte instructions, however.

When a jump instruction is executed, the microcode must figure out which byte is being jumped to, and execute the proper **IBPTR_*i*** function in the third cycle of the click in which the new instruction doubleword is fetched. It can then dispatch with **AlwaysNextMacro** in the next click, and this dispatch can be overlapped with a refill if the jump was to byte 3 or 4, since in this case it is not necessary to wait for the **NextMacro** to detect the need for a refill.

5.2 Stack

A special set of 16 registers (S) is provided to hold the Mesa evaluation stack, together with a 4-bit register (STKP) to address them. There are functions to increment and decrement STKP (**push** and **pop**), and it can be read onto W.12-15 and loaded from Y.12-15. An S register behaves just like a U register. Note that the operations on STKP act at the *end* of the instruction, so that S is addressed by the *old* value of STKP during the instruction.

There are actually two sets of S registers; the set to use in a particular cycle is selected by the CIN bit. It is expected that CIN will always be 0 when S is referenced.

The intended use of S registers is that the top of the stack will be kept in an R register (called TOS), and that STKP will be kept pointing to the S register containing the second word on the stack. This means that the top two words are immediately available for a (non-arithmetic) binary operation.

These are simply programming conventions, however.

5.3 Split adder

The split adder feature ors a bit into the MUX field of the microinstruction for bits 8-15 only. The effect is to change the selection of G and H inputs to the ALU for bits 8-15, as follows (* is the operation specified by the ALU field):

A * Q	becomes	0 * A
A * B	becomes	X * A
0 * Q	becomes	X * Q
0 * B	becomes	X * 0

The motivation for this feature is to allow an operation of the form TOS + (MDS + IB), by choosing MUX=A*B, A=TOS, B=MDS and X=IB, and taking advantage of the fact that MDS has 0 in the 8 lsb, while IB has 0 in the 8 msb.

SplitAdder is invoked by the ENS bit (unless this kludge proves intolerable). If ENU is off, ENS has no other function, so this encoding causes no problem. If ENU is on, selecting S automatically invokes SplitAdder, with two consequences:

> SplitAdder cannot be used when reading or writing U. If it is only wanted with arithmetic to Y, this is not a problem, since U cannot participate in arithmetic.
>
> SplitAdder is always invoked when S is selected. This isn't a problem when S is being read, since MUX will never have one of the values which is transformed by SplitAdder. Care is needed when S is written, however, since only the righthand MUX values in the above table can be used.

5.4 PC.16

The machine contains a single-bit register PC.16 which is intended to be used for the least significant bit of the Mesa byte program counter; an R register called PC will be used for the remainder of the program counter. There is a function **Cin=PC.16** which makes PC.16 the carry into the ALU, and also complements PC.16 (at the end of the instruction, so that the old value is the one used for the carry). The effect, when combined with PC_PC+0, is to add 1 to the 17-bit PC. PC.16 can also be left-shifted into an R register; see section 3.1.

6. Input-output

Wildflower does not have a general-purpose high performance input-output system. Instead, it has specialized arrangements for handling three high-bandwidth devices: disk, ethernet and display. It also has an microcomputer bus, on which low-bandwidth devices are supposed to live. The advantage of this scheme is that the disk, ethernet and display can be handled with a very small amount of hardware. All three devices depend heavily on receiving a known amount of service from the processor with latencies of about 1 us. The assignment of clicks to input-output tasks is detailed in section 2.1.

There is an input bus IOIN which allows data from the devices to be read onto X; this is a slow source, which does not permit arithmetic or writing into SU. There is also an output bus IOOUT for sending data to the devices. *It is not yet decided whether this will be Y or H. If it is Y,* This is a slow sink, which cannot accept data from arithmetic, SU, or MEMIN. Functions for each device read input data and status onto IOIN and load output data and control from IOOUT. Each device also has a function to reset its request signal.

In general, io works in the following way (see figure 3). Each device has a *request* signal which indicates that the device wants service from the processor. This signal is the output of a flip-flop which is clocked by the processor clock; thus any necessary synchronization is done before this flip-flop (figure 3a). During the second cycle of the click preceding a potential click for the device, its request signal is examined. If it is true, the device's task runs during the next click, which thus becomes a service click for the device (figure 3b).

Normally, the task will execute a function which clears the request signal during the service click, and therefore the task will not run again until the device sets request again. Care must be taken to avoid a race in which the request is set again before it is cleared; this would cause one request to be lost. Alternatively, the task may refrain from clearing the request, in which case every potential click for that task will be a service click. Among other things, this provides a simple way for a task to time short intervals, by simply leaving its request set, counting clicks, and multiplying by the frequency of potential clicks for the task in the schedule. Also, properly designed synchronous devices, such as the display, may operate in this mode. It would be possible to operate an asynchronous device like the ethernet in this mode, and test in each click to determine whether data should be transferred; this would result in some wasted cycles.

6.1 Latency and bandwidth

If a device request is latched at time 0, the earliest time that the first microinstruction executed by the task serving that device can complete is 300 (ns, or .3 us). The latest time is 300d+500, where d is the maximum number of clicks between successive potential clicks for that task (2 for display and disk, 4 for ethernet); the delays are illustrated in figure 3b. Of course, if the request is not removed, then the maximum delay between service clicks is 300d+100; this is the time between completing the last instruction of one service click and completing the first instruction of the next one.

A device which receives service n times per round (8 clicks), and transfers b bytes in every service click, can sustain a peak transfer rate rp of nb/2.4 megabytes/second, or 3.33nb megabits/second. Adequate buffering is of course required to sustain this rate. The amount of buffering requires depends on the amount of synchronization required between the device and the processor. Note that we do not include any parallel-serial conversion register in the amount of buffering we count. At one extreme, the display has n=3 and b=4, for rp=40 Mbit. This is also the actual rate r, because the display is run synchronously with the processor, and has just enough extra buffering beyond the minimum of b bytes to make use of every service click. At the other extreme, the ethernet has r=3 Mbit, which means that n=b=1 provides enough bandwidth.

The reader is invited to find flaws in the analysis which follows.

Latency considerations, however, make this inadequate with single-byte buffering. The amount of buffering required is given by a=(.3d+s+.5)r if the task takes or delivers a byte in its first cycle. Here s is the maximum delay between the device taking or delivering the data, and the system clock on which the request is clocked, including worst-case synchronization. If the data handling is synchronized to the clock (as is the case for the ethernet), there may still be one cycle of maximum synchronization delay to allow for the fact that r is not synchronized to the processor clock. This delay s might be negative, provided that s_{min}+230 ns>0, (where s_{min} is the *minimum* delay, and the 230 ns is the minimum service time, 300 ns, minus the setup time for the IOIN bus, 70 ns). For the ethernet, which is synchronized when the wire is sampled on input, s=100, d=7 if n=1, and this results in a=8.1 bits of buffering required. If we want to have only 8 bits, we must decrease d, which can only be done by serving the device more than once per round. On output the ethernet operates completely synchronously with the processor, so that s=0 and 8 bits would be enough.

It is also necessary to ensure that an io task doesn't clear its request after a second one comes in (thus losing the second request). Assume that the clear is done in the first cycle of the service click. The interval between requests is b/r. This, rounded down to the nearest cycle, must be greater than the time from request to completion of first instruction, with a one cycle margin to allow for synchronization variations. Thus floor(b/r)-1 > 3d+5. If the request flipflop is gated so that the set overrides the clear, then equality is also safe.

6.2 Disk

This section describes the disk controller design and microprogramming details. For a complete understanding it would be prudent to read the interface manual for the SA-4000, especially sections 4 and 5.

The Shugart SA-4000 disk has a bit time of 140 ns, a byte time of 1.12 us, and 18000 bytes/rotation; hence the rotation time is 20.16 ms. The parameters of the interface are b=1, d=2, n=3, r=7.14 Mbits, s=-120. The value for s is computed as follows. The minimum synchronization delay is one cycle or 100 ns. The microcode takes or delivers data on the *second* cycle of the service click; and we include this extra delay into s for convenience. If the request raised on the disk bit clock is raised three bit times before the data is available, we get s_{min}=200-3*140=-220, which is legal. The maximum synchronization time is two cycles or 200 ns, so s=300-420=-120. This yields a=(.6-.12+.5)7.14=1.02*7.14=6.96, so we get by with one byte of buffering.

For request clearing, we have 11-1 >= 3*2+5, or 11>=11. This means that the request might be set and cleared on the same clock.

The intention is to format the disk into *sectors*, each containing three records: a 2-word *header*, a 10-word *label*, and a 256-word *data* record. Each record also has a *preamble*, described below, and a 1-word *checksum* appended to it. Possible operations on a sector are to read any number of complete records, and then write the remainder of the sector. There are 30 sectors on one track, and hence 600 bytes per sector. The information above (not counting the preambles) amounts to 542 bytes, which leaves 58 bytes, or 19 in each record, for the preamble. This amount of space must accommodate the 8 us acquisition time of the disk's read clock circuit, as well as all the time required for synchronization with the microcode.

Each interaction with the microcode can occur as little as 400 ns after the disk-related hardware generates the signal, or as much as 1300 ns. Thus there is a disk-to-microcode synchronization error of almost one byte time. The preamble must be extended by twice this time, since the error is in one direction if reading is fast and writing slow, and in the other direction if the other way around.

The preamble must have the property that the disk has 8 us between the arrival of Read Gate and the occurrence of read data. The idea is to write 8 us worth of zeros, followed by a 1 bit, followed immediately by data. The interface looks for the 1 bit in the serial stream coming from the disk, and thus synchronizes itself to the data stream. So the needed length of preamble is 8+2=10 bytes, all zero except for a 1 at the end of the last byte; if time is measured in service clicks, there will also be some breakage error. Since there is room for 19 bytes, we are in good shape. Details of the timing are presented below.

The disk controller interprets five functions:

KData_ takes a byte from IOOUT for transmission to the disk. **_KData** reads a byte from the disk onto IOIN.8-15. These must be done within 11 cycles of the request, which is just the maximum delay with d=2.

rKReq resets the disk request, unless it is being set in the same cycle.

KCtl_ writes the disk control register from IOOUT. The bits of this register are sent directly to the disk (except for CanLoadKSR, which is internal to the controller, and Write Gate, which is gated with OKtoRun). The bits are interpreted as follows:

04	Kinit: used for read initialization, as described below.
05	KReset': when reset, the disk is reset, and will neither generate nor remember requests.
06	WantKByteReq: when set, allows a request to occur every 8 disk bit clock times. When a transfer is not in progress, this signal should normally be reset.
07	Direction Select: determines the direction of head motion on a seek. 1 is "out" or away from the center of the disk.
08	Step: instructs the disk to move the heads one track in the direction specified by Direction Select. The 0-1 transition is the active one. Direction Select must not be changed in the same click. Step must spend at least 2 cycles in each state, which means that only one step/click is possible.
09-11	Head Select: selects one of the 8 possible heads.
12	Fault Clea: Any write fault is reset on the 1-0 transition of this signal.
13	Read Gate: turns on the phase locked loop in the disk which controls data recovery. It should be turned on only in the preamble of a disk record. There is good data and clock 8 us after Read Gate is set.
14	Write Gate: turns on the write amplifier in the disk. This signal is gated with an internal signal called OKtoRun, which ensures that the disk will not see Write Gate during power-up.
15	CanLoadKSR: allows the shift register in the disk controller which does serial-parallel conversion to be loaded from the output buffer. This signal must be 0 during reading and 1 during writing. It also participates in read initialization, as described below.

_KStatus reads the disk status information onto IOIN. This information is frozen whenever KReq is set, to ensure that it will not change between the occurrence of an interesting event and the time it is read by the task. Hence it is *not* possible to leave KReq set and monitor the status for some interesting event.

08	Track00: set when the disk is on track 0. This information is used to get the microcode and the disk to agree about where the heads are.
09	Write Fault: set when the disk has decided that an improper attempt has been made to write. The disk latches this condition until it is reset by the Fault Clear control signal.
10	Seek Complete: cleared immediately after a Step, and set when the head completes stepping. After Seek Complete sets, it is still necessary to wait 20 ms for the head to settle before reading or writing.
11	Sector Mark: true for 1.1 us at the beginning of each sector (sector boundaries are set by jumpers in the disk). This signal causes a request, and hence the occurrence of Sector Mark is frozen in KSTATUS until the request is cleared. If the request is cleared within 1.1 us, it may be set again, since the disk's Secotr Mark may not have gone away.
12	Index Mark: true synchronously with Sector Mark once per rotation of the disk.
13	Ready: normally true continuously within 3 minutes after power up. If Ready disappears, there will be a request, and hence the state of Ready will be frozen in KSTATUS until the request is cleared.

There are three conditions which cause a disk request: next byte (only if WantKByteReq is set), not Ready, and Sector Mark. The request will appear between 100 ns and 200 ns after the condition becomes true. If KReset' is reset (0), there will be no disk requests.

The proper sequence for writing a record on the disk with *m*+1 bytes of preamble and *n* bytes of data after a sector mark is as follows:

We assume that the idle state of the interface is with 0 in KDATA and CanLoadKSR set. Turn on Write Gate and WankKByteReq, and block. The delay is .4-1.3 us from the sector mark. The next request will be caused by a byte clock.

At the next service click, delay from sector mark is another .4-2.4 us, or a total of .8-3.7 us.

FOR *i* IN [1..*m*-1] DO KDATA_0; Block ENDLOOP. Delay is 1.12*m*.

Now *m* zero bytes have been written (actually the *m*th one is just being written). Do KDATA_1; this writes the last byte of preamble. Block.

FOR *i* IN [1..*n*] DO KDATA_*Data*[*i*]; Block ENDLOOP

KDATA_*ChecksumHi*; Block; KDATA_*CheckSumLo*; Block; KDATA_0.

The time available for the preamble is 19 byte times. Taking off 1 for safety, and 4 (3.7/1.12=3.30) for the initial delay results in *m*=13. Hence the 1 bit at the end of the preamble is 1.12*14-.14=15.54 us from the beginning of the preamble, or 16.34-19.24 from the sector mark.

The proper sequence for reading a record from the disk after a sector mark is as follows:

Block (delays 1.0-2.2 us from the request). This ensures that the heads are into the region where 0's have been written. In the first cycle of the third click, turn on Read Gate.

FOR *i* IN [1..11] DO Block ENDLOOP (delays 8.7-9.0 us).

Now the disk has acquired its read clock and is supplying good data, which is zeros. Turn on KInit, CanLoadKSR and WankKByteReq, and block. The next request will be caused by a byte clock. There will be a 0 in the shift register. Delays .4-2.2 us.

Now the counter in the controller is locked up at 9, and we are 10.1-13.4 us from the sector mark. Turn off KInit and CanLoadKSR, and block. The controller is now waiting for the 1 bit at the end of the preamble, and the next request will occur when the first data byte is ready to be read with _KDATA.

FOR *i* IN [1..*n*] DO *Data*[*i*]_KDATA; Block ENDLOOP

*ChecksumHi*_KDATA; Block; *CheckSumLo*_KDATA; Set CanLoadKSR.

Slight variations are appropriate for reading and writing after a previously read or written sector.

6.3 Ethernet

This section to be written by Mike Schroeder.

6.4 Display

Unlike the other io devices on the machine, the display gets its timing entirely from the execution of microinstructions. The interface contains a 32-bit main shift register DMAIN which can be loaded from MEMIN.0-31, an auxiliary 16-bit shift register DAUX which holds the bits which may remain to be sent out when new memory data arrives, a counter which controls switching between the main and auxiliary registers, some fast logic convert the 20 MHz rate at which the shift registers operate to the 40 MHz at which the display itself operates, and a control register which supplies horizontal and vertical sync signals, and synchronizes the shift registers to the microcode as described below.

The display needs 4 bits/cycle, or 12 bits/click. Hence in 3 clicks it needs 36 bits, which is 4 more than the memory can supply. This deficiency is remedied by servicing the display at intervals of 2, 3 and 3 clicks within a round of 8 clicks; thus the service pattern is A x B x x C x x, where the capital letters name the display service clicks. DMAIN shifts out its last bit at the end of the third cycle of A (A.3), and is loaded from MEMIN on that clock. At B.3 24 bits have been consumed, and 8 remain. The 8 bits are saved in DAUX, and 32 new bits are loaded into DMAIN, so that 40 bits are now buffered up. By C.3 36 bits have been consumed, 8 from DAUX and 28 from DMAIN. The 4 remaining bits are again saved in DAUX, and 32 new bits again arrive. By the next A.3, all 36 bits have been consumed, and we are back where we started.

To deliver a total of 1024 bits requires 32 clicks, or $10^2/_3$ rounds. Things are arranged so that the 1st data click is a B click, and the 32nd data click is a C click. On the next A click the task sets horizontal blanking into the control register. During the next B, C, A, B, C and A clicks the task idles (or samples the keyboard, or whatever). This 5.0 us is for horizontal retrace. On the final A click, the task fetches 0. On the following B click the task fetches the first doubleword of data, and clears horizontal blanking. The 8 zeros in DAUX are supplied as data, followed by the first 32 bits of real data in DMAIN, and we go around the loop again. Thus a scan line is a total of 13 rounds long.

The display has a request signal which is set at the end of the first cycle of the first click of every *n*th round (i.e. every 2.4*n* us), where *n* is part of the display control register. Normally the display task doesn't clear its request signal. During vertical retrace, however, and whenever the display is painting scan lines not covered by bitmap, the task does clear the request, and thus takes only one click in 8*n*, rather than 3*n* clicks.. It is also possible to have blank left and right borders in multiples of 3*32=96 bits, or about 1 inch, and take only 1 click/round instead of 3, using this mechanism with *n*=1.

The display is controlled by two functions:

Display_ loads DMAIN from MEMIN and simultaneously loads an assortment of control information from IOOUT, as follows:

08-10	the request setting interval *n* described above.
11-12	4 times the number of bits to be taken from DAUX before switching to DMAIN. This should be 0 on an A click, 2 on a B click, and 1 on a C click.
13	if 1, the display is turned off (black).
14	if 1, horizontal blanking is turned on.
15	if 1, vertical blanking is turned on.

rDReq clears the display request signal, which is set at the start of every *n*th round, as described above.

6.5 Microcomputer bus

Nothing has been decided yet about how this will work.

7. Debugging hardware

xxx

LOAD

Nop
Q_F
B_F
B_F & Y_A

B_F lsh 1
B_F rsh 1
B,,Q_F,,Q lsh 1
B,,Q_F,,Q rsh 1
An F1 makes shifts into cycles

MUX

Illegal combinations

G	H
A	0
A	A
0	0
X	B

all others are legal

ALU

G+H
G-H
H-G

GvH
G&H
G'&H
G xor H
G' xor H

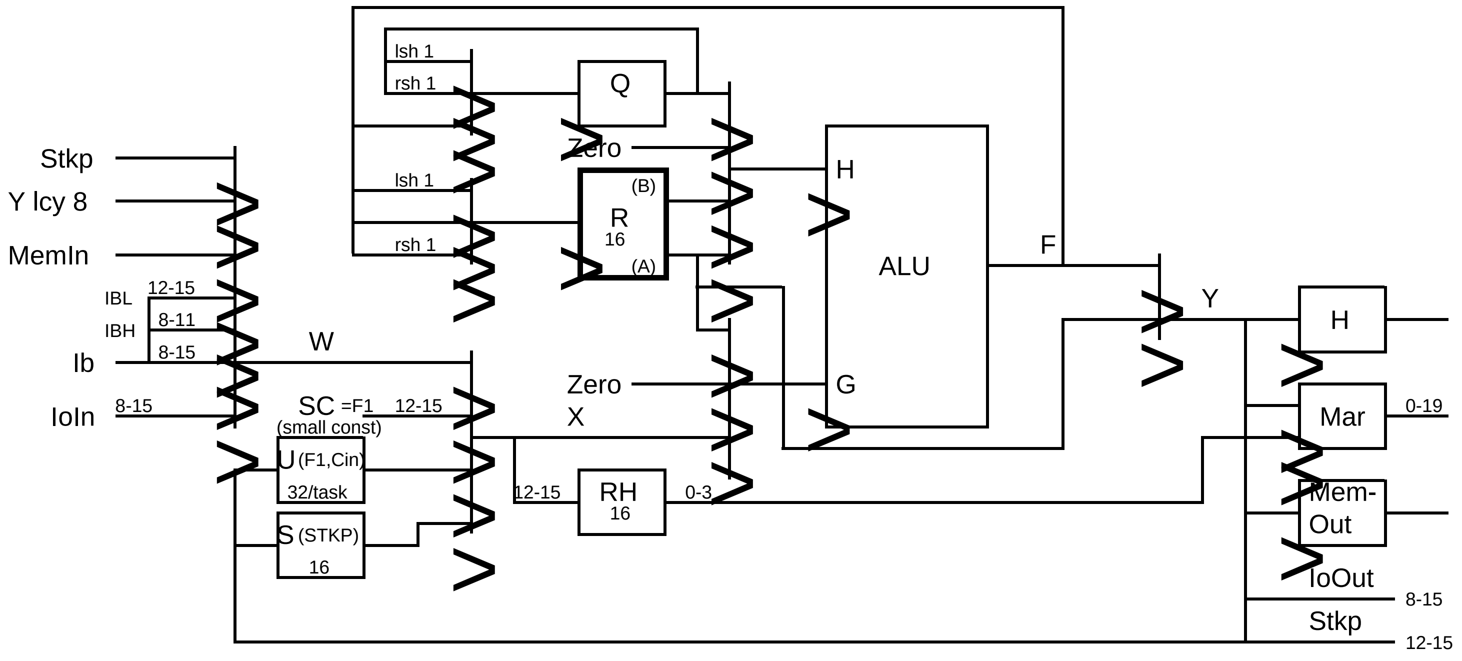


Figure 1: Arithmetic data paths

All paths are 16 bits (0-15) except as noted

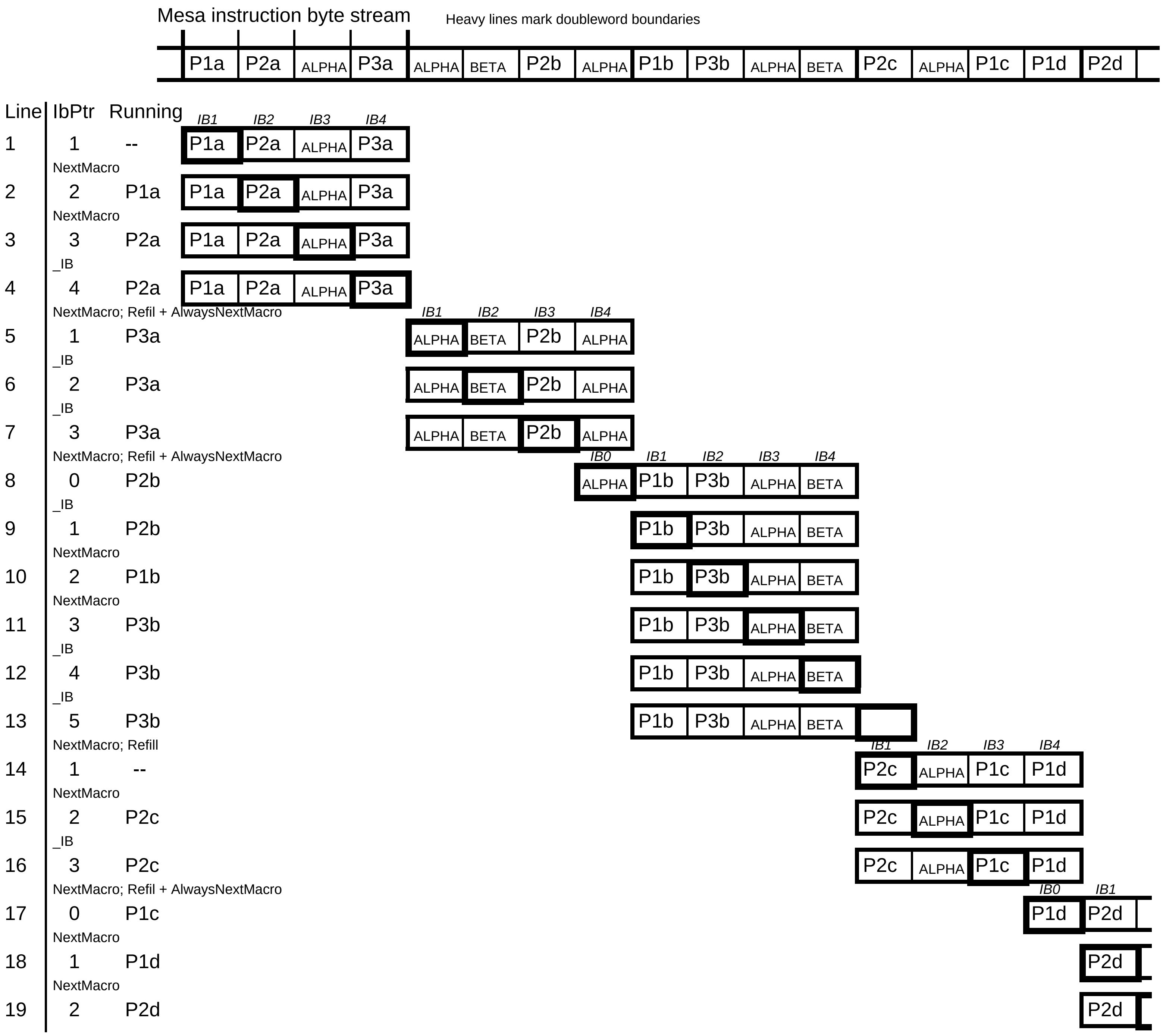
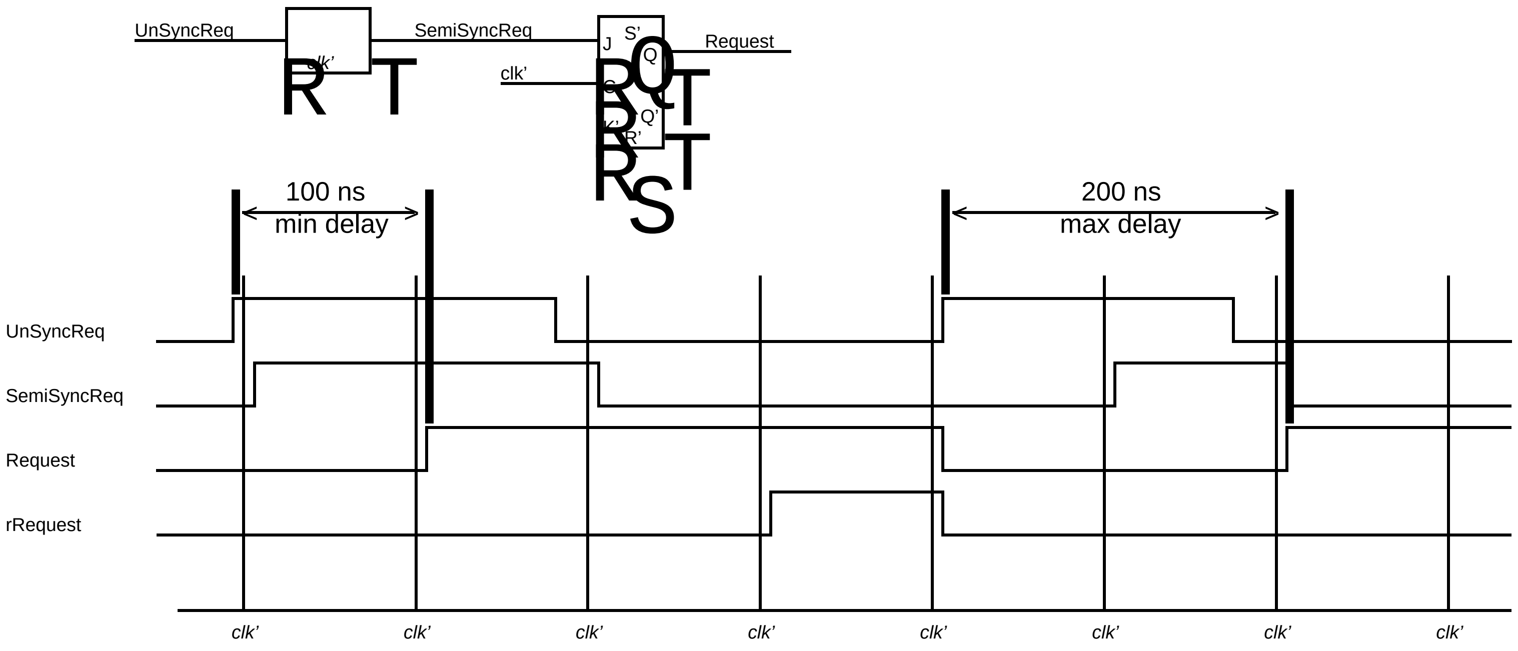
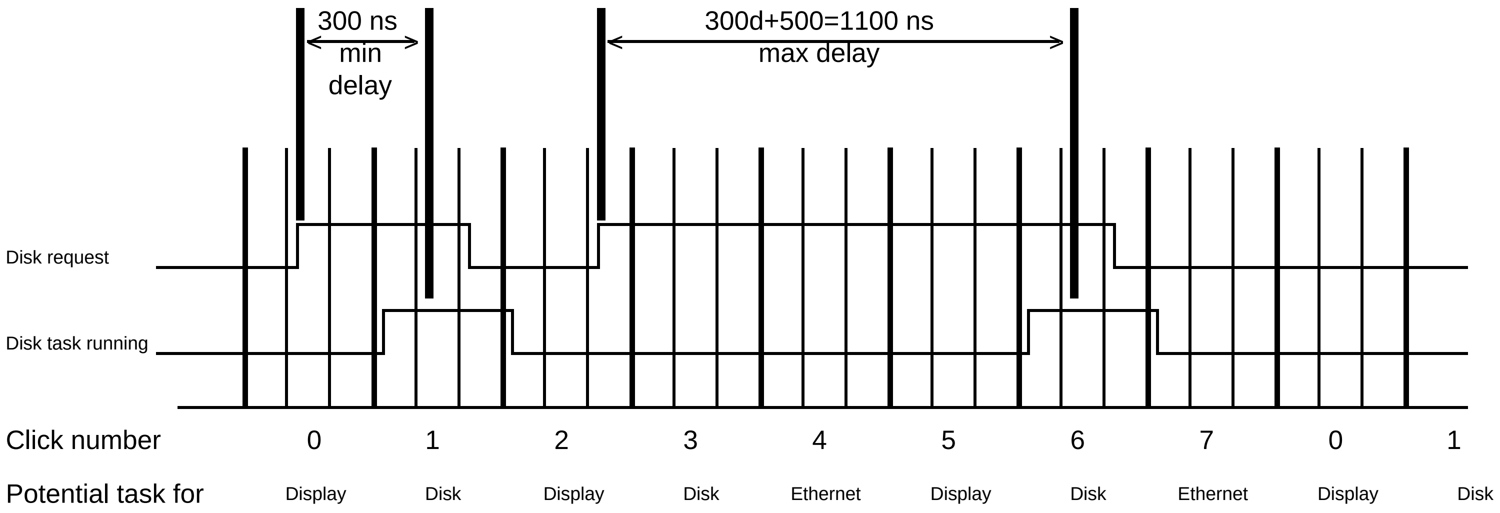


Figure 2: Operation of the instruction buffer

The heavy box encloses the byte which is on IB



(a) Request synchronization circuit and timing



(b) Delay from (synchronized) request to end of first service instruction

Figure 3: Input-output task latency