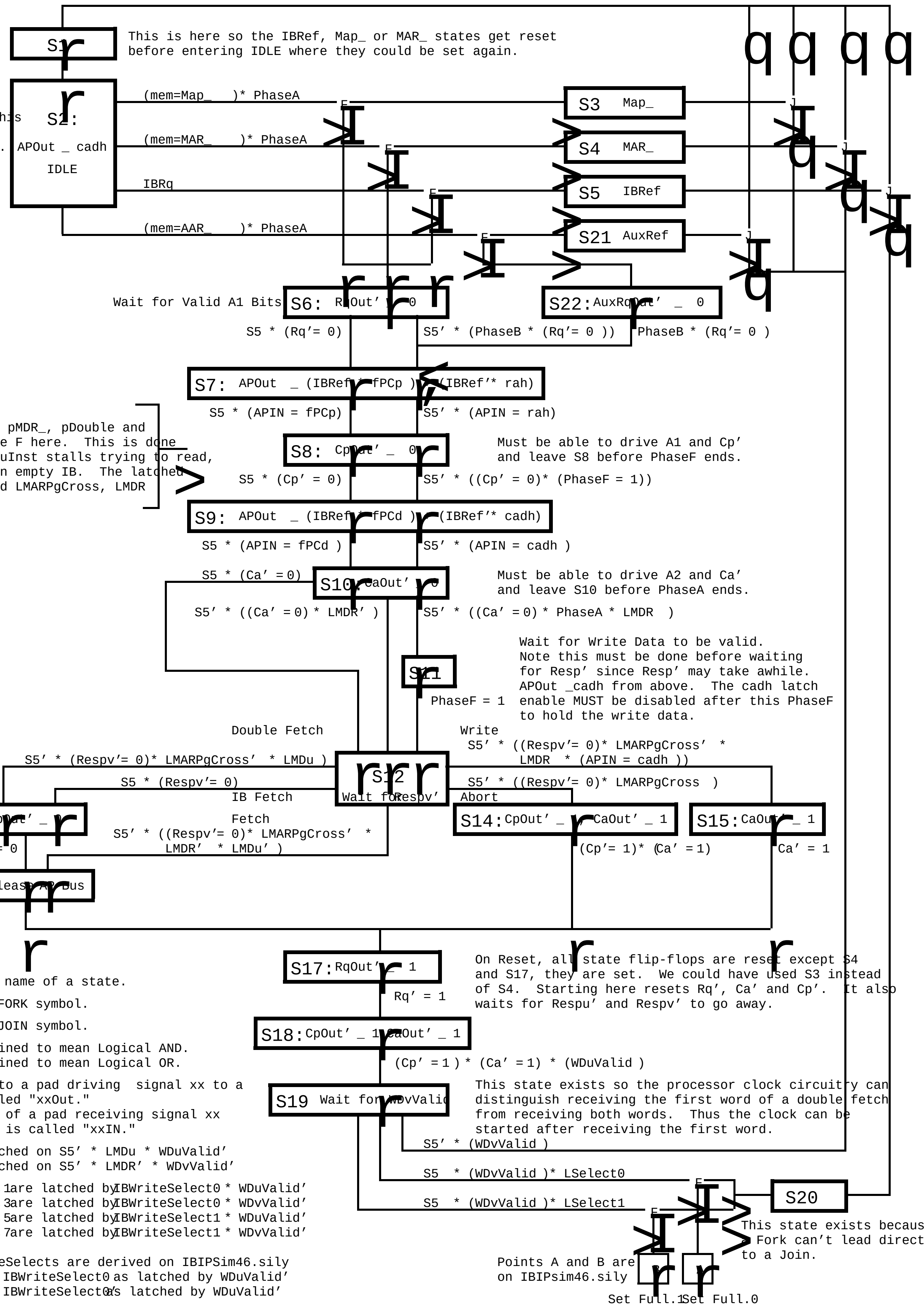

S1
This is here so the IBRef, Map_ or MAR_ states get reset before entering IDLE where they could be set again.
S2:
APOut _ cadh
IDLE
(mem=Map_)* PhaseA
(mem=MAR_)* PhaseA
IBRq
(mem=AAR_)* PhaseA
F
J
S3 Map_
S4 MAR_
S5 IBRef
S21 AuxRef
S6: RqOut' _ 0
S22: AuxRqOut' _ 0
Wait for Valid A1 Bits
S5 * (Rq'= 0)
S5' * (PhaseB * (Rq'= 0))
PhaseB * (Rq'= 0)
S7: APOut _ (IBRef * fPCp) + (IBRef'* rah)
S5 * (APIN = fPCp)
S5' * (APIN = rah)
S8: CpOut' _ 0
S5 * (Cp' = 0)
S5' * ((Cp' = 0)* (PhaseF = 1))
S9: APOut _ (IBRef * fPCd) + (IBRef'* cadh)
S5 * (APIN = fPCd)
S5' * (APIN = cadh)
S5 * (Ca' = 0)
S10: CaOut' _ 0
S5' * ((Ca' = 0) * LMDR')
S5' * ((Ca' = 0) * PhaseA * LMDR)
S11
PhaseF = 1
Double Fetch
Write
S5' * ((Respv'= 0)* LMARPgCross' *
LMDR * (APIN = cadh))
S5' * (Respv'= 0)* LMARPgCross' * LMDu)
S5 * (Respv'= 0)
S12
Wait forRespv'
S5' * ((Respv'= 0)* LMARPgCross)
IB Fetch
Fetch
S5' * ((Respv'= 0)* LMARPgCross' *
LMDR' * LMDu')
Abort
S13: CpOut' _ 0
Cp' = 0
S14: CpOut' _ 1, CaOut' _ 1
(Cp'= 1)* (Ca' = 1)
S15: CaOut' _ 1
Ca' = 1
S16: Release AP Bus
S17: RqOut' _ 1
Rq' = 1
S18: CpOut' _ 1, CaOut' _ 1
(Cp' = 1) * (Ca' = 1) * (WDuValid)
S19 Wait for WDvValid
S5' * (WDvValid)
S5 * (WDvValid)* LSelect0
S5 * (WDvValid)* LSelect1
S20
This state exists because a Fork can't lead directly to a Join.
Points A and B are on IBIPsim46.sily
Set Full.1
Set Full.0

cadh reflects the F Bus. This is broadcast for debugging.

Latch MARPgCross, pMDR_, pDouble and p_MDu during Phase F here. This is done in case the next uInst stalls trying to read, or dispatch on, an empty IB. The latched signals are called LMARPgCross, LMDR and LMDu.

Must be able to drive A1 and Cp' and leave S8 before PhaseF ends.

Must be able to drive A2 and Ca' and leave S10 before PhaseA ends.

Wait for Write Data to be valid. Note this must be done before waiting for Resp' since Resp' may take awhile. APOut _cadh from above. The cadh latch enable MUST be disabled after this PhaseF to hold the write data.

On Reset, all state flip-flops are reset except S4 and S17, they are set. We could have used S3 instead of S4. Starting here resets Rq', Ca' and Cp'. It also waits for Respu' and Respv' to go away.

This state exists so the processor clock circuitry can distinguish receiving the first word of a double fetch from receiving both words. Thus the clock can be started after receiving the first word.

Sx is the name of a state.

F is the FORK symbol.

J is the JOIN symbol.

"*" is defined to mean Logical AND.

"+" is defined to mean Logical OR.

The input to a pad driving signal xx to a pin is called "xxOut."
The output of a pad receiving signal xx from a pin is called "xxIN."

MDu is latched on S5' * LMDu * WDuValid'
MDv is latched on S5' * LMDR' * WDvValid'

IB bytes 0,,1 are latched by IBWriteSelect0 * WDuValid'
IB bytes 2,,3 are latched by IBWriteSelect0 * WDvValid'
IB bytes 4,,5 are latched by IBWriteSelect1 * WDuValid'
IB bytes 6,,7 are latched by IBWriteSelect1 * WDvValid'

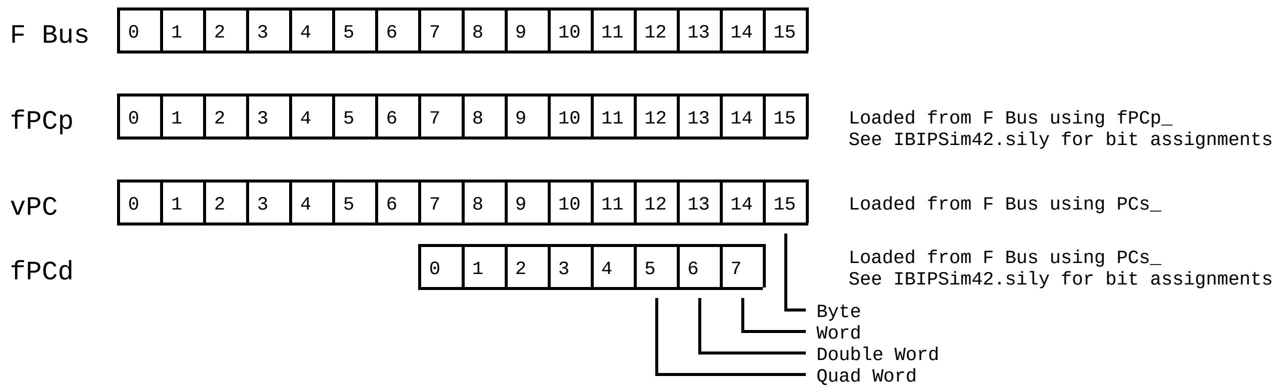
The IBWriteSelects are derived on IBIPSim46.sily
LSelect0 _ IBWriteSelect0 as latched by WDuValid'
LSelect1_ IBWriteSelect0' as latched by WDuValid'

XEROX SDD	Project	Reference	File	Designer	Rev	Date	Page
	Daisy	Bus Interface - State Diagram	IBIPSim43.sily	Garner, Davies	A	8/19/83	43

1. To increase the performance of the Daisy P chip, an autonomous instruction fetch unit has been incorporated. This circuitry can fetch ahead in the instruction stream when the Emulator would otherwise not be using memory. The bytes fetched are held in an 8-byte Instruction Buffer. Use of the Buffer requires two pointers. One, vPC, holds the virtual memory address of the next byte to be used by the Emulator. A copy of this byte will be in the Instruction Buffer, so the Emulator need not wait to access it. The other pointer, fPCp,,fPCd, is used to point to the next real memory location from which bytes will be fetched. One may think of the fPCp,,fPCd concatenation as forming a program counter which is always ahead of, or equal to, the vPC. The difference between them equals the number of bytes in the Instruction Buffer.

fPCd.7 is always sent to memory as 0. Its actual value can be read from the X bus.

2.



3. The Instruction Fetch Unit consists of an Instruction Buffer and fetch logic. The Instruction Buffer is written by the fetch logic and read by the Mesa Emulator. There are a number of exception conditions encountered in normal operation. The vPC is the lower 16 bits of Mesa's virtual program counter (actually PC + CodeBase). It is incremented by hardware each time a byte is removed from the IB (successful IBDisp, AwIBDisp, _ib, etc)). The fPCd is the least significant 8 bits of the real address of the double word to be fetched into the IB. The vPC register points to bytes, fPCd points to words. fPCd.[0..7] corresponds to vPC.[7..14]. The difference between these two quantities equals the number of words in the Instruction Buffer. fPCd.[0..7] >= vPC.[7..14]. Because Mesa pages have 256 words (512 bytes), the least significant 8 bits of a given virtual address and its corresponding real address are the same. They give the displacement of the word in a page.

Definition:

An"IBAccess" is any operation which reads a byte in the Instruction Buffer. The byte may or may not be removed from the IB. IBAccess is true either if the microinstruction contains one of {_ib, _ibSE, _ibHigh, _ibLow, AwIBDisp} or if fPCAtEndOfPage is false while a microinstruction has an IBDisp.

Definition:

An "IBConsume" is an IBAccess that actually removes the byte read from the IB. All IBAccesses are IBConsumes except for _ibHigh and _ibLow.

Rule1:

An IBDisp is an IBAccess (and an IBConsume).

Exception to Rule 1:

If the last bytes put into the IB came from the end of a page, fPCAtEndOfPage is set. If a microinstruction attempts an IBDisp while fPCAtEndOfPage is set, the IBDisp is cancelled and an IBDispTrap is generated instead. Thus there is no IBAccess or IBConsume.
Special microcode then trys to find a real memory address for the next virtual page. If the next virtual page is resident in real memory, fPCp is loaded, fPCAtEndOfPage is turned off and both the Emulator and the IFU can proceed. If the page is not resident, the Emulator saves whatever data is necessary in case the Mesa instruction must be restarted and proceeds with the Mesa instruction in hopes that data from the next page won't be needed.
If it is, we arrive at the Exception to the Exception to Rule 2.

Rule 2:

An IBAccess causes a word to be read from the IB.

Exception to Rule 2:

If the IB is empty, the IBIP clock will be stopped to delay the IBAccess while the Instruction Fetch Unit puts some Mesa bytes into the IB. The IBAccess will be allowed to proceed only when the data arrives.

Exception to the Exception to Rule 2:

If the IB is empty and filling it would require the Instruction Fetch Unit to access a new page, the access will be cancelled, the clock will not be stopped and an IBEmptyTrap will be generated. This results in a page fault being taken since the new page cannot be resident. Any Mesa Instruction emptying the IB must have encountered an IBDispTrap that would have reset fPCAtEndOfPage if the next page were resident.
Even if the Emulator jumps to the end of a virtual page, the IBDisp in the jump microcode will generate an IBDispTrap because fPCAtEndOfPage will be true by PhaseB of the IBDisp microinstruction.

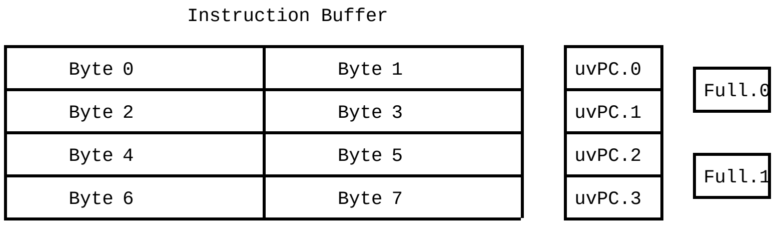
4. The Instruction Buffer can hold 8 bytes or 2 double words. The Instruction Fetch Unit uses double word fetches when accessing memory. fPCd.7 is always sent to the memory as 0. Its actual value can be read from the X bus.

5. The method used to synchronize the Emulator with events originating outside the P chip is to stop the P Chip clock. The clock may also be stopped when the Emulator requres a resource (Instruction Buffer, AP Bus) which is not available. See IBIPSim49.sily for details of the clock holding curcuitry. Because the clock may be stopped waiting for an Instruction Buffer fetch to complete, the Instruction Fetch machine itself may not depend on the clock. Primarily for this reason, the AP Bus State Machine and the Instruction Fetch State machine have been integrated into a single asynchronous state machine (see IBIPSim43.sily).

6. The Instruction Buffer itself must be addressed so that the correct bytes may be read and written. However, one must be very careful when building an asynchronous machine that its inputs will not have unwanted pulses or edges at the wrong times. After much investigation, it was decided that circuits using the least significant bits of vPC and fPCd as Instruction Buffer addresses always produced such glitches. Because of this, the Instruction Buffer is addressed by a rather strange circuit (at least if you are used to standard TTL stuff) shown on the next page.

XEROX SDD	Project	Reference	File	Designer	Rev	Date	Page
	Daisy	Instruction Fetch Unit: Documentation	IBIPSim44.sily	Garner, Davies	A	8/19/83	44

7. ## Conceptual Representation



The uvPC bits are a Unary Virtual Program Counter. They form a program counter that counts by setting exactly one of the uvPC bits to 1. For example, if byte 0 were to be read next, uvPC.0 would be set and uvPC.1, uvPC.2 and uvPC.3 would all be reset. The byte within the word to be read is indicated by vPC.15. It is 0 for reading even bytes and 1 for reading odd bytes.

The Instruction Fetch Unit uses double word reads to fill the Instruction Buffer. If Full.0 is set, at least one of the bytes in Byte0..Byte3 has been loaded from memory but not yet read. The same relationship holds for Full.1 and Byte4..Byte7. In this sense, Full.x really means "NotEmpty.x." All four combinations of Full.[0..1] (00, 01, 10, 11) are legal and occur as part of normal operation.
Two interesting signals are derived from the Full bits. One, IBRoom, indicates there is room for another double word in the IB. The other, IBEmpty, indicates the IB is empty.
The equations are:

IBRoom = (Full.0’+ Full.1)’
IBEmpty = (Full.0’* Full.1)’

8. Each time the Mesa Emulator executes a jump instruction, the Instruction Buffer must be "cleared." As part of the Emulator routine for the jump instruction, a "PCs _" clause appears in a microinstruction (before or after possibly re-mapping the new virtual PC page number into fPCp). This

a) causes vPC.[0..15] _ F.[0..15],

b) causes fPCd.[0..7] _ F.[7..14],

c) causes uvPC.0 _ F.14’,

d) causes uvPC.1 _ F.14,

e) causes uvPC.2 _ uvPC.3 _ 0,

f) causes Full.0 _ Full.1 _ 0.

The next time a microinstruction attempts to read or dispatch on a byte from the Instruction Buffer (an IBAccess), that microinstruction will be held until the Instruction Fetch Unit fetches a double word. The IBAccess, almost certainly an IBDisp, may occur in the microinstruction immediately following the PCs_. It may also occur later.

9. As a result of an IBReference, fPCd is incremented (by 2) and one of the Full bits is set. To prevent a race between Emulator jump code and this action, any microinstruction containing a PCs_ clause is held up before it enters the MicroInstruction Register (MIR). As usual, this is done by stopping the clock. It may proceed only when the IBReference completes.

10. The clock is also stopped whenever the next microinstruction contains an IBAccess (IBDisp, AwIBDisp, _ib, _ibSE, etc) and the IB is empty. The stopping of the clock before the first IBDisp after a PCs_ clause is a special case of this. This holding of the microinstruction prevents the Emulator from reading data that hasn’t arrived yet.

11. In normal operation, Full.0 is set after the upper double word has been loaded with data. It is reset when uvPC.2 goes to 1 indicating all bytes in the upper double word have been read. Full.1 is set when the lower double word has been written and is reset when uvPC.0 is set. It is possible that the Emulator will do an IBAccess just as an IBReference is loading a double word into the IB. This will not cause a problem since the byte being read cannot lie in the double word being written. Care must be taken to ensure the IB WriteEnable signals have no glitches at the wrong time.

12. Generation of the Instruction Buffer write enables requires both select terms (which latch) and timing terms (when data is valid). This note describes the select terms.

the Instruction Buffer write selects are formed as follows (Note IBReferences are only started when there is room in the IB):

a) Write the lower double word of the IB if either the upper double word is not empty (Full.1=1) or both double words are empty and the next byte will be read from the lower double word (uvPC.0 = 1 or uvPC.1 = 1). Note the only way uvPC.1 can be set while both Full bits are reset is that a PCs_ clause has just been executed.

b) Write the upper double word of the IB if either the lower double word is not empty (Full.0 = 1) or both double words are empty and the next byte will be read from the upper double word (uvPC.2 = 1).

c) Within a double word, the word with an even IB (and memory) address is written with data latched by Respu’. The odd addressed word is written with Respv’ data. Note there are two sets of latches on the pads, one for Respu’ data and one for Respv’ data. Even words in the IB are wired to the Respu’ latch and odd words to the Respv’ latch.

The Select Equations are:

IBWriteSelect0 = Full.1 + (Full.0’ * Full.1’ * (uvPC.0 + uvPC.1))

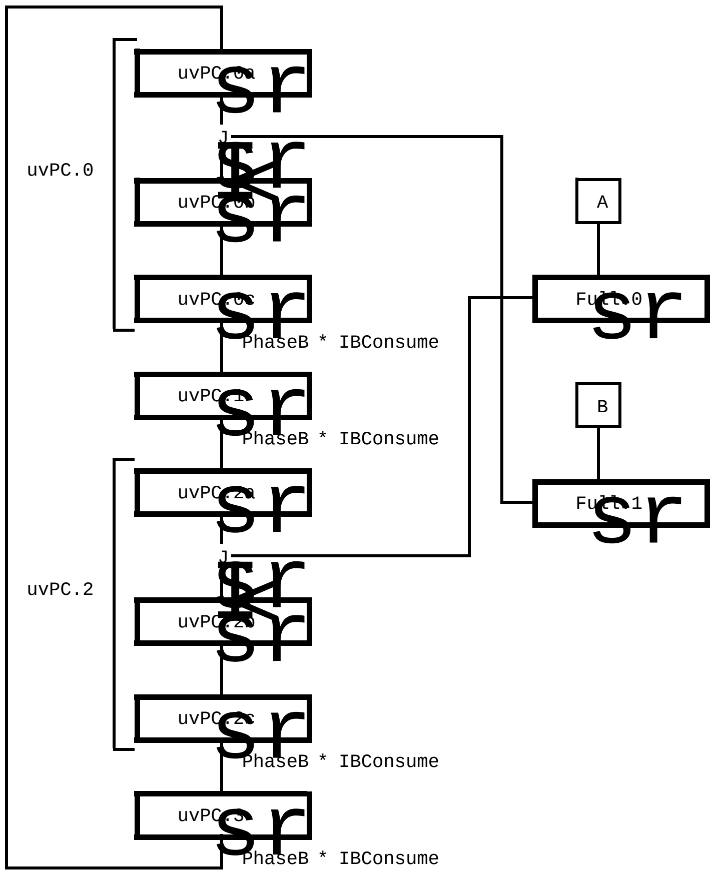
IBWriteSelect1 = Full.0 + (Full.0’ * Full.1’ * (uvPC.2)

Note that if the Emulator reads IB Byte 7 just as one of the low double words is being written, Full.1 will go to 0 and uvPC.0 will go to 1. This could cause a glitch in the IBWriteSelect0 term. To help avoid this, we re-write the equations as follows:

IBWriteSelect0 = (Full.1 + Full.0)’ * (Full.1 + uvPC.0 + uvPC.1)

IBWriteSelect1 = (Full.0 + Full.1)’ * (Full.0 + uvPC.2)

If the circuit is implemented in this way, we must only ensure that the (Full.1 + uvPC.0 + uvPC.1) term does not glitch. This is equivalent to making sure uvPC.0 is set before Full.1 is reset. This is incorporated into the state machine shown below (we go to uvPC.0a, then to a state which resets Full.1). Note uvPC.0 = uvPC.0a + uvPC.0b + uvPC.0c. uvPC.2 is similarly constructed.



Comments:

1. Points A and B are from IBIPSim43.sily. They specify the points at which the Full bits are set.

2. Dividing uvPC.0 and uvPC.2 into three states satisfies 2 conditions:

   a) uvPC.0 (or 2) must be set before Full.1 (or 0) is reset (see 12 above),

   b) Being in uvPC.0 (or 2) must not prevent Full.0 (or 1) from being set as would be done if the IB were filled up.

   As noted above, we must arrive in uvPC.0 before resetting Full.1. This is why uvPC.0a (and uvPC.2a) exist. Entering state uvPC.xa both sets the uvPC.x term in the IBWriteSelect equations and signals that it is time to reset the corresponding Full bit. State uvPC.xb resets uvPC.xa and the Full bit. We enter uvPC.xc to release the reset on Full. We assume the Full bit was already set on entering uvPC.xa. If it were not we would have just been reading from an empty section of IB. We also assume that the state machine enters each successor state before leaving each predecessor state so (uvPC.xa + uvPC.xb + uvPC.xc) does not glitch.

3. The effect of a PCs_ clause is:

   a) uvPC.0c _ F.14’,

   b) uvPC.1 _ F.14,

   c) all other uvPC and Full states are reset.

4. There are two interesting signals generated from the Full bits. One indicates that there is room in the IB for another double word. The other indicates that the IB is empty. Note that only microcode (by accessing the IB, advancing the uvPC and clearing the Full bits) can set these signals. The equations are:

   IBRoom = (Full.0’ + Full.1)’

   IBEmpty = (Full.0’ * Full.1)’

5. The uvPC bits change in PhaseB because the IB is actually read in Phase A.

6. An IBConsume is an operation which reads and removes a byte from the IB. All IBAccess operations are also IBConsumes except _ibHigh and _ibLow (see IBIPSim44.sily).

13. The IB memory circuits are interesting (or strange, depending on your point of view) latches. The pads on the edges of the chip encode each data bit on two lines. 01 = bit is 0, 10 = bit is 1, 11 = bit isn't valid yet. The latches require that both the write enables be set and the input bits reach the 01 or 10 states before they will be written. We assume they are written very soon after the 01 or 10 state is reached.

14. There are two sets of data bits latched from the AP Bus. The first set is latched with when Respu' rises. The second set is latched whenever Respv' rises. The first set goes to the even words of the Instruction Buffer array and to the MDu latch. The second goes to the odd words of the IB array and the MDv latch.

15. The MDu latch is activated only on double fetch operations. The MDv latch is activated for all MAR_ or Map_ operations. The IB is written only in IBReferences.

16. Generation of the write enable term for any particular latch involves both a select term (given above or in IBIP46.sily) and a timing term. Latches receiving Respu' data are enabled when they are selected and Respu' drops causing all the data inputs to go to the 11 state. The write enables are reset when all data bits reach either the 10 or 01 state. We assume the delays involved in both detecting the data bits have reached valid states and disabling the write enables covers the data setup time to the end of the write enable pulse. Thus:

```
IBWriteEnable01 = IBWriteSelect0 * WDuValid'
IBWriteEnable23 = IBWriteSelect0 * WDvValid'
IBWriteEnable45 = IBWriteSelect1 * WDuValid'
IBWriteEnable67 = IBWriteSelect1 * WDvValid'
```

17. We are now ready to discuss the conditions that intiate an IBReference. The general philosophy is that the Instruction Fetch logic should stay out of the way as much as possible while attempting to maintain an adequate number of bytes in the Buffer.

Following this line of reasoning, we derive two conditions under which the Instruction Fetch unit should initiate a Fetch:

a) Begin a fetch whenever the next microinstruction contains an IBAccess (See IBIPSim44.sily) and the IB is empty, indicated by IBEmpty (IBIPSim45.sily).

b) Begin a fetch if there is room in the IB for another double word (IBRoom=1) and the current microinstruction contains an IBWindow clause. This gives the microcoder explicit control of the times at which the IB will fetch. The microcoder should use IBWindow when the Emulator won't be using memory for awhile and the bytes fetched will probably be used. It is expected that after the basic structure of the microcode and the chip set timings are set, microcoders will tune the performance by placing IBWindow clauses in the code.

18. An IBReference will not start if the fPCd has reached the end of a memory page. This is because it doesn't know where the next virtual page is in real space. The fPCd is incremented by 2 during each IBReference. If there is a carry out of bit 0, the fPCAtEndOfPage flag is set. When this flag is set, IBWindow clauses are ignored. When it is set, each IBDisp executed causes a microcode trap. In this IBDispTrap code, the new virtual page number is examined. If the new virtual page is already in real memory, the corresponding real page number is put in fPCp and the code dispatches on the next byte in the IB. This can be done since the trap occured when the Emulator expected to dispatch on a new bytecode. If the new virtual page was not present in memory, the machine state is saved and the Emulator attempts to dispatch on the next bytecode anyway. It is not legal to cause a page fault until we are certain the new page will be needed. Thus, if there is an IBAccess, the IB is empty and fPCAtEndOfPage is true, we know a new page must be brought in. When this condition is detected, another type of microcode trap is generated. This trap code recovers the state of the machine at the beginning of the bytecode, saves this, and puts this process to sleep while scheduling the page fault process. When the page arrives, this process may be restarted at the beginning of the bytecode that caused the fault.

19. The following discussion assumes fPCAtEndOfPage is not set.

If the next microinstruction contains an IBAccess and the IB is empty, the Emulator clock will be stopped. If an IBReference is already in progress, it will complete, a Full bit will be set, IBEmpty will be reset and the Emulator clock will start. If the Instruction Fetch Unit was idle, it will wa for any P chip bus activity to complete and begin an IBReference.

20. This act of waiting for bus activity to complete implies a restriction on the type of bus activity allowed when using the IB. In particular, the microcoder may not include an MDR_ clause and an IBAccess in the same microinstruction. Referring to the bus machine state diagram in IBIPSim43.sily, one can see that the microinstruction immediately following a MAR_ or Map_ defines the type of memory operation to be performed. If the operation is a write (has an MDR_ clause), the bus machine must wait until the end of this second microinstruction for the write data. If, however, that same microinstruction contains an IBAccess, it would be held before being executed. This would result in a deadlock in which the clock holding circuitry would be waiting for the IFU to fill the IB, the IFU would be waiting for the bus machine to release the bus so it could start and the bus machine would be waiting for the clock holding circuitry to release the second microinstruction.

21. It is interesting to note that because an IBAccess can stop the clock, an ordinary memory read can be "interrupted" by an IBReference. Consider a standard short Mesa routine containing a MAR_, an IBDisp and a _MD in successive microinstructions. The clock could be stopped just before the IBDisp microinstruction. The bus state machine would see that the microinstruction after the MAR_ didn't have an MDR_ or _MDu clause, so it would assume a single word fetch. This fetch would be completed by the asynchronous bus state machine. At the end of the fetch, the data would be latched in the MDv register. Upon entering State S2 (IDLE), the bus state machine would see IBEmpty and IBAccess, so it would start an IBReference. When it completed, a Full bit would be set, causing IBEmpty to go false and the clock would start. The Emulator would then execute the IBDisp and _MD microinstructions normally.

22. One might (almost correctly) think the IBRequest is:

IBRq = (IBEmpty * pIBAccess * PhaseF) +

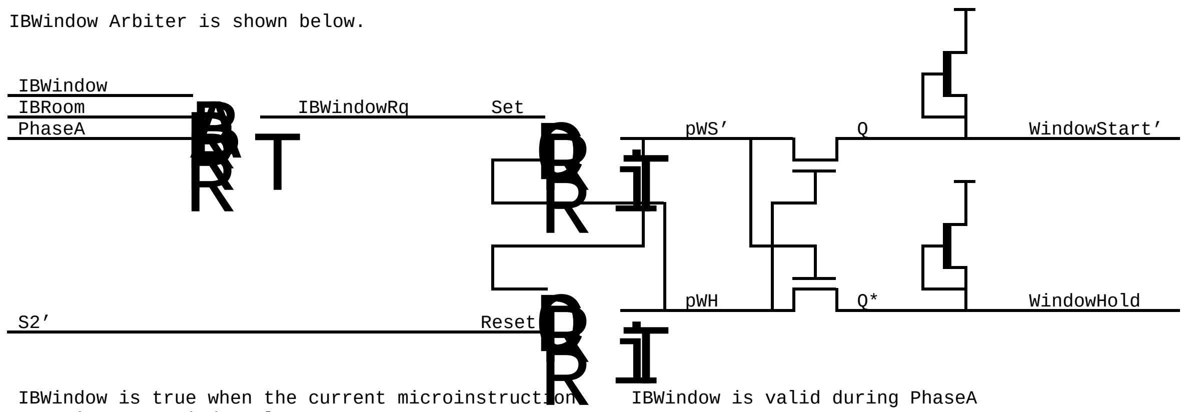
(IBRoom * IBWindow * PhaseA)

pIBAccess is the indication that the next microinstruction contains an IBAccess. This indication is valid in PhaseF. An IBAccess of an empty IB will cause the clock to stop and wait for the Instruction Fetch logic to figure out what is happenning. So long as the IBEmpty flag is correctly set by the time the bus state machine lands in state S2 (IDLE), the Instruction Fetch unit will see no logic hazards in this term.

The IBWindow term is another story. The whole purpose of providing an IBWindow clause is to allow the Emulator to proceed in parallel with an IBReference. Hence, we cannot stop the Emulator clock on every IBWindow just to let the IFU see it. In fact, it is most reasonable to let the IFU ignore the IBWindow if bus is already busy doing something else. The problem then may arise if the bus state machine arrives in state S2 (IDLE) just as PhaseA is ending, turning off that term. This could cause a hazard and great confusion in the bus state machine. It could conceivably decide to go to no state at all, effectively disabling bus activity forever.

23. To remedy this problem, we install an arbiter made of scary (to the old-fashioned TTL designer) gates and transistors. it is shown on the next page.

24. The IBWindow Arbiter is shown below.



IBWindow is true when the current microinstruction contains an IBWindow clause.

IBRoom = (Full.0' Full.1)'

IBWindow is valid during PhaseA

S2' is a 1 when the bus state machine is NOT in state S2 (IDLE).

This is a strange circuit. Note that because of the MOS transistors, WindowStart',,WindowHold may be in only the 01, 10 or 11 states. If both NOR gates produce 00, both transistors are turned off and the pullups force both WindowStart' and WindowHold to the 1 state. In fact, it is claimed that whenever the NOR gate outputs are equal, WindowStart' and WindowHold are both set to 1. We need to know what WindowStart' and WindowHold do in order to understand the circuit. The basic idea is that when WindowStart' is 0, we will start an IBReference. When WindowHold is a 1, we hold the Emulator clock. While the arbiter is deciding what to do (NOR gate outputs tracking each other, and no longer in PhaseA), we hold the clock AND don't start an IBReference. When it decides, the circuit will either start an IBReference, leave S2, and release the clock or just release the clock. If the microinstruction being held contains a MAR_ or Map_, some gate delay conditions must be met to ensure the clock stays held if we leave S2 and this circuitry releases the clock. There is an explicit list of critical delay assumptions on IBIPSim50.sily - IBIPSim52.sily.

Normally, the bus machine is either in the S2 (IDLE) state when IBWindowRq is set or it stays outside S2 for the duration of the IBWindowRq pulse. Explore these two cases:

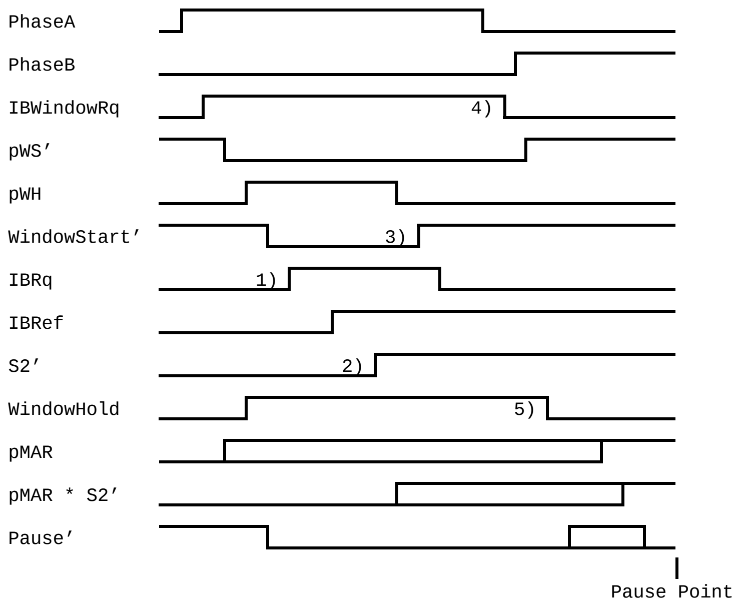
a) The bus state machine was already in the S2 state.

In this case, the action proceeds as follows:

1) The appearance of IBWindowRq causes an IBRq
2) The IBRq causes the bus state machine to leave the S2 state, setting S2'.
3) Assuming this happens before PhaseA completes, both the set and reset inputs are active, so both outputs are set to 1.
4) PhaseA ends so only the reset input is active. Note that if PhaseA ends before the machine exits S2, we simply skip 3) above.
5) With the reset active and the set inactive WindowStart' stays at 1 and WindowHold goes to 0. This allows the clocks to proceed if there were no other conditions holding them.

Note that if things are slow so WindowHold stays up past the Pause Point, the clock is just held and the machine slows down a bit until WindowHold finally falls.

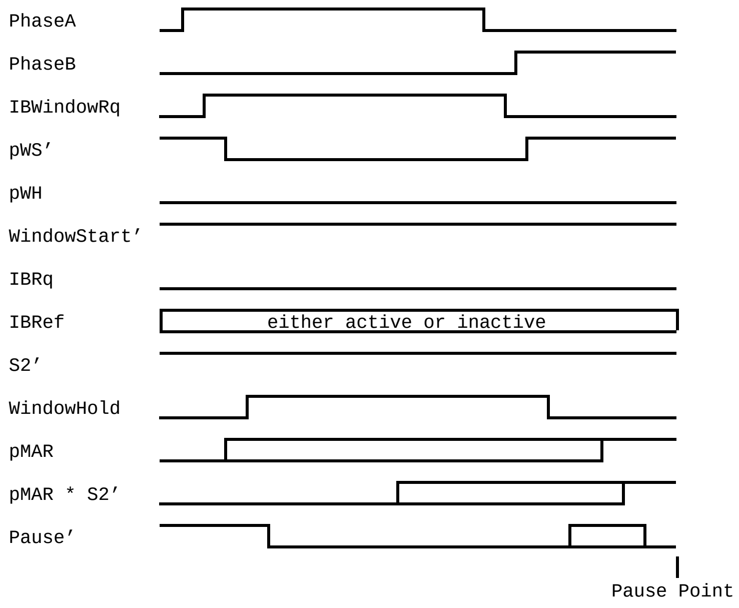
Case a



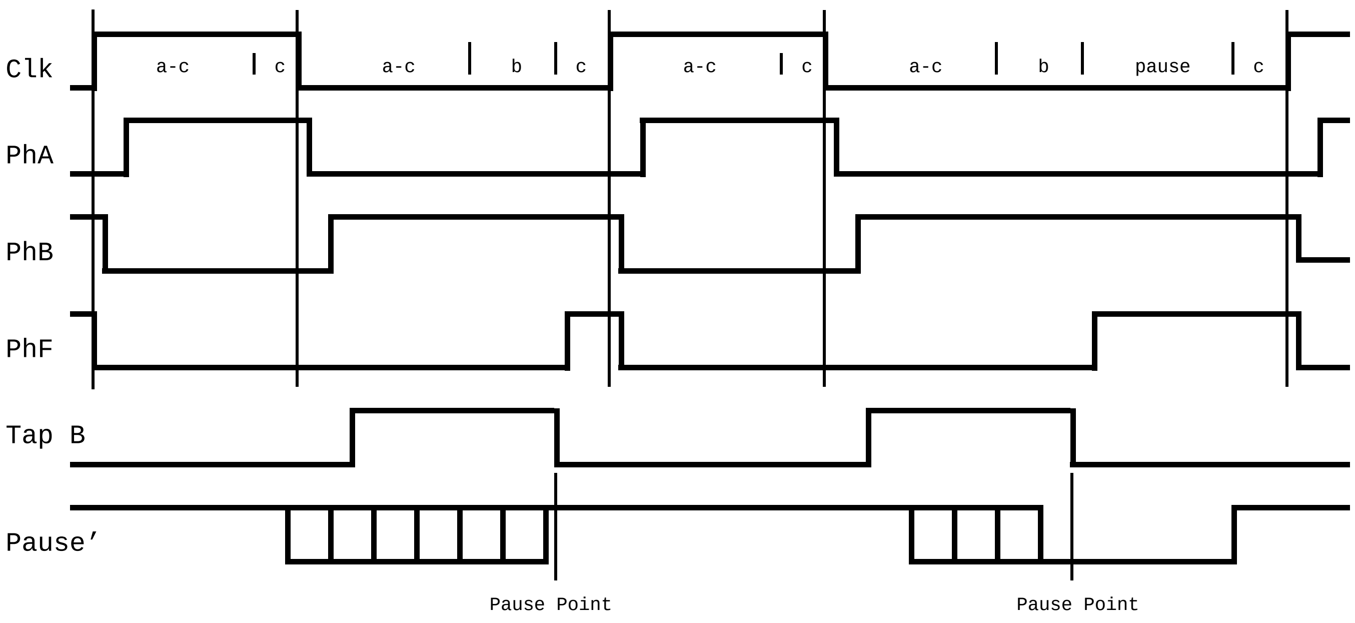
Note that it is fine for Pause' to go HI so long as it is LO before the Pause Point. It cannot be stable before the pMAR flag is stable.

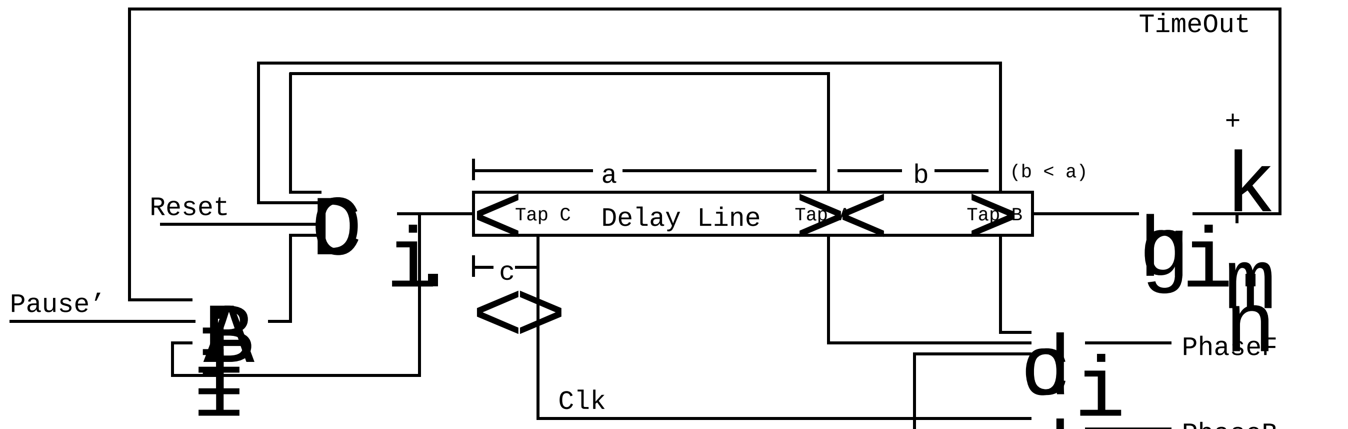
The case of S2' going LO while IBWindowRq is HI is shown in the critical delay documentation (IBIPSim50.sily).

b) The bus state machine stayed outside S2 while IBWindowRq was true. See timing diagram below.



The only action is to set WindowHold while IBWindowRq is set. This only lasts to the end of PhaseA, well before the Pause Point, so there is no net effect on the machine.





PhaseA (=a-c+c = a)

- Read/Write R/RH registers
- Read/Write U registers
- X bus valid
- Precharge F bus
- Compute ALU logical functions
- Compute F15
- Give CA, WriteData to AChip

PhaseB (=a-c+b+pause+c = a+b+pause)

- Compute ALU carries
- F bus valid
- Precharge X bus
- Update/Write registers (Q, Fh, Stkp, vPC, uvPC, fPCd)
- Precharge U/R/RH regs
- Give RA bits to AChip
- Next Microinstruction fields can be decoded

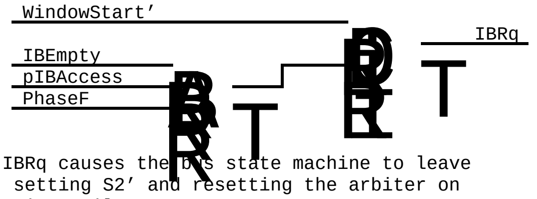
PhaseF (=pause+c)

- F Bus Valid
- c = Time for Fbus_MD, CSA_INIA or Cbus

IBIP Clocking

16 August 83

25. The IBRequest is generated as follows:



An IBRq causes the bus state machine to leave S2, setting S2’ and resetting the arbiter on IBIPSim48.sily

26. This drawing describes the clock holding circuitry. The clock itself is shown in Garner’s drawing IBIPClocking.sil or IBIPClocking.press on [Indigo]<Daffodil>IBIP>Doc and [Indigo]<Daffodil>IBIP>Doc>Press respectively.

27. A couple of subtle points about that clocking scheme should be noted before going on. First, the longest run of 1’s in the delay line is of length a. In order to ensure that the clock operates properly, b must be less than a. If it weren’t, that run of 1’s could lie entirely between taps a and b, both taps would signal 0, and a new run of 1’s would begin erroneously.

Second, notice that the basic mode of operation is for the run of 1’s to fall off the end before Pause’ is examined. When Pause’ goes HI, another run of 1’s begins. The TimeOut line is supposed to restart the clock if Pause’ refuses to go HI. This also causes a microcode trap. The time at which Pause’ is examined is called the "Pause Point."

This feature of ignoring Pause’ until the delay line empties allows one to be fairly sloppy in generating Pause’. The in fact, one need not worry about generating a rising edge on Pause’ at any time. If it occurs when Pause’ is being ignored, it will be ignored. if it occurs when the clock is stopped, it will simply start the clock. One need only worry about falling edges on Pause’ just when the delay line runs out. If Pause’ falls just at the right time, a runt clock pulse could be generated.

28. Another subtle point that escaped me for a couple of weeks is that the delay line empties and the decision about whether to hold the clock or not (the Pause Point in IBIPClocking.sil) is made just BEFORE PhaseF, not at the end of PhaseF. PhaseF starts with the pause and ends c time units after the clock holding circuitry has decided to start the clock. The main consequence of this is that one cannot decide whether or not to stop the clock by looking at signals that are only valid in PhaseF.

29. The main conditions that cause the Emulator clock to stop are:

a) Waiting to use the AP bus,

b) Having started a memory cycle, waiting for the data to return,

c) Waiting for the Instruction Fetch Unit to put bytes into the Instruction Buffer so the Emulator may read or dispatch on them,

d) Waiting for the Instruction Buffer to complete a reference so the Mesa program counters may be loaded. This occurs in the jump routines,

e) Waiting for the IBWindow arbiter to decide whether not to do an IBReference.

Notice that, in general, we don’t stop the clock if an attempted IBAccess will result in an IBDispTrap or an IBEmptyTrap. The only exception is when we jump to the end of a page. There, we stop the clock while the IB data arrives, then allow the IBDispTrap to take place. It’s too messy and unnecessary to wait for the AwIBDisp in the IBDispTrap code to stop the clock and run the IBReference.

30. To enhance performance, a microinstruction needing memory data is allowed to proceed to the point at which the data is stored. When the data arrives, the clock is started. The chip has c time units until PhaseF ends. During this time, the data must be stored and, if the microinstruction specifies a branch based on the F bus contents, the new microinstruction address must be calculated. If we made the entire microinstruction wait until the data arrived, as is done with all of the other cases, we would waste the time needed to get through PhaseA and PhaseB to PhaseF on many memory references.

31. The IBIP uses a two level pipeline in which the fetching and decoding of one microinstruction is overlapped with the execution of the next. We call the microinstruction being executed the "current" microinstruction and the one being decoded the "next" microinstruction. In the equations below, we add the prefix "p" to denote a field from the next microinstruction. For example, "mem" is the field of the current microinstruction indicating what sort of memory operation should take place. The corresponding field in the next microinstruction is called "pmem."

```
Pause’ =  (EnableAPBus *
            (

      a)     ((pmem  = MAR_  )+ (pmem  = Map_  ))* S2’
                                                        +
      b)     (mem  = _MDu  ) * (S5’ * S2’ * S1’ * S19’ )     {Not an IBReference and waiting for 1st word of double fetch.}
             (mem  = _MDv  ) * (S5’ * S2’ * S1’ )  +     +   {Not an IBReference and waiting for 2nd word of double
                                                              fetch or only word of single fetch.  See IBIPSim43.sily}

      c)     IBEmpty * fPCAtEndOfPage’   * ( ((pfS.1= 0 )* (pfZ.[0..1]= 0)) + ((pfS # 15 )* (pfY = 15 ))  ) +
                                                {_ib, _ibSE, _ibLow, _IbHigh}     {AwIBDisp, IBDisp}
      d)     ((pfX = PCs_ ) + (pfX = fPCp_ )) * S5 +

      e)     WindowHold
            )
          )’
```

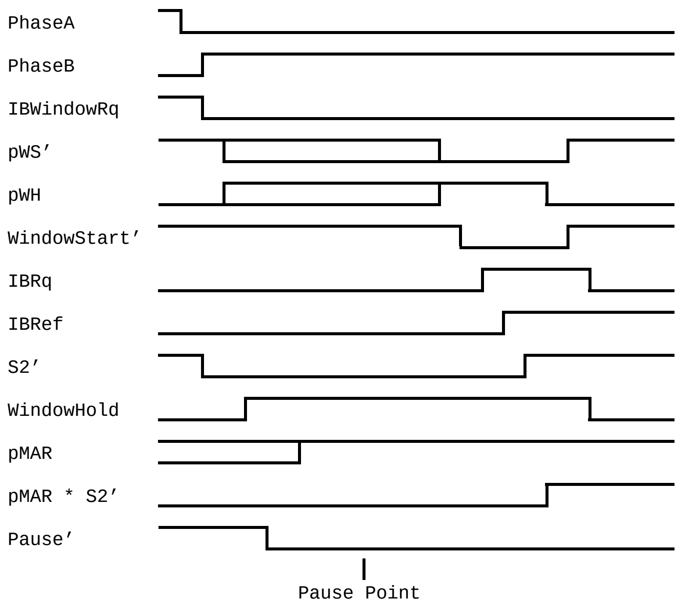
Critical Delay Assumptions

1. After an IBReference, the new value of IBRoom must be valid by the time state S2 (IDLE) is entered.

2. If the clocks are near the Pause Point (IBIPClocking.sil), then leaving S2 must set the (pMAR_ + pMap_) * S2′ term of the Pause′ circuitry before resetting the WindowHold term.

   This condition should only be reached when the bus state machine entered S2 just as IBWindowRq (IBIPSim48.sily) goes LO.

Not to Scale, the Pause Point is too soon.



See IBIPSim48.sily for circuit.

2a. IBWindowRq should set WindowHold before the Pause Point.

Comments

1. One of the two forks at the bottom of IBIPSim43.sily will be taken at the end of an IBReference. This will cause one of the Full bits to be set (see IBIPSim46.sily). The IBRoom calculation is

   IBRoom = Full.0′+ Full.1′,

   so IBRoom should go to 0 soon after this fork unless the IB was completely empty. The other branch of the fork leads to a join with the S5 state and the join leads to S1. Note we cnnot leave S1 for S2 until the Full bit has been set. The designer must additionally guarantee that IBRoom is valid by the time S2 is entered. If it falsely indicated that there was room in the IB and an IBWindow clause was present, another IBReference could begin with disasterous results. Another consequence of IBRoom staying too long could be that IBRq might disappear just as the state machine was leaving S2. This could cause confusion, possibly leaving all the states turned off.

2. We must guarantee that if

   the bus state machine enters S2 just as IBWindowRq drops
   and
   the arbiter (IBIPSim48.sily) waits until after the Pause Point to decide what to do
   and
   it decides to start an IBReference
   and
   the next microinstruction contains a MAR_ or Map_
   THEN
   Pause′ is held LO without a glitch.

   When both IBWindowRq and S2′ drop, pWS′ and pWH can go to an undecided state. It is claimed that while in this state they are equal to each other so both MOS transistors are turned off and both WindowStart′ and WindowHold are HI. WindowHold holds Pause′ LO while a decision is made. If the arbiter had decided to forget the IBReference, WindowHold would simply rise and the clock would start. We assume the arbiter decides to start an IBReference and pulls WindowStart′ LO. This eventually causes S2′ to be raised. This starts two events into motion. First, the (pMAR_ + pMap_) * S2′ term in the pause circuitry starts HI. Second, the S2 transition propagates through the arbiter, passing through both NOR gates and the lower pass transistor, pulling WindowHold LO. To guarantee that the clock circuitry won′t generate a runt pulse, we must guarantee that (pMAR_ + pMap_) * S2′ gets HI before WindowHold goes LO.

If it were possible to violate condition 2a and it took a very long time to set S2′, it would be possible to have a microinstruction containing a MAR_ or Map_ start just as the bus state machine was embarking on an IBReference.

Critical Delay Assumptions

3. Starting an IBReference must set the ((pfX=PCs_) + (pfX=fPCp)) * S5 Pause’ terms before resetting the WindowHold term.

Comments

3. Note that putting an IBWindow clause in a microinstruction which immediately preceeds a MAR_ or Map_ microinstruction is generally a bad idea. The result will be that either nothing happens (if there was no room in the IB) or the memory reference will be delayed by the IBReference. It is generally a better idea to put the IBWindow clauses where they cannot interfere with normal Emulator memory references. Note that it would make sense to have IBWindow and MAR_ in adjacent microinstructions if the IBWindow microinstruction were at the end of a loop that would be executed a couple of times before getting to the MAR_.

   Putting IBWindow and PCs_ clauses in successive microinstructions is a genuinely bad idea. Either there will be no IBReference (if the IB had no room) or the PCs_ clause will be delayed while the Instruction Fetch Unit fetches 4 bytes that will be immediately thrown away. This could only make sense if the IBWindow microinstruction had a branch and the path leading to the PCs_ was seldom executed. Nonetheless, the hardware must perform correctly if a PCs_ microinstruction immediately follows an IBWindow microinstruction. To guarantee this, we must guarantee that if the arbiter decides to execute an IBReference, the clock will be held. The following sequence will be executed:

   a) WindowStart’ will go LO, setting IBRq’

   b) IBRq will set S5 (see IBIPSim43.sily)

   c) In parallel: the ((pfX=PCs_) + (pfX=fPCp_)) * S5 will start towards HI while the machine begins to leave the S2 state.

   d) Leaving S2 will set S2’, resetting the WindowHold term of the clock hold circuitry.

   If the clock is to be held properly, ((pfX=PCs_) + (pfX=fPCp)) * S5 must go HI before WindowHold goes LO.

Critical Delay Assumptions

4. The Instruction Buffer write enables must not glitch.

   a) The WDuValid and WDvValid signals should go to 0 between the time the P chip senses that Respv’ has gone to 0 and it has set Rq’, Ca’ and Cp’ to 1.

   b) The delay between when the WDu or WDv lines actually have valid data and when this is reflected in WDuValid or WDvValid should be longer that the setup times of the destination latches.

   c) It should not be possible for a WDu (or WDv) data line to go from valid to invalid after Respu’ (or Respv’) has gone HI. It should similarly be impossible for a WDu (or WDv) data line to take on, even momentarily, the incorrect value (10 or 01) on the way to the correct value.

   d) The IB state machine must set the uvPC.2 state (see IBIPSim46.sily) before clearing the Full.0 state. Similarly, it must reach uvPC.0 before clearing Full.1.

Comments

4. There are three undesirable consequences of incorrect behavior on the WDuValid, WDvValid and IB write enable signals.

   a) If WDuValid or WDvValid falsely indicate that the data from memory is available before it actually arrives and the bus state machine looks at them while they are in this state, the state machine will proceed as though the data had actually arrived. This will cause both bad data to be used and it will allow the state machine to start Emulator or IBReference before the A chip is ready.

   b) If the IB write signals glitch as they are being sampled to decide which IB Full bit to set, the wrong one could be set. This could cause the Instruction Fetch Unit to lose track of what is really in the IB.

   c) The IB input data is only valid briefly before the IB write enable signals are disabled. If the write enable signals glitch at this time, the IB may be written incorrectly.

   Meeting conditions 4a - 4c guarantees that the WDuValid and WDvValid signals will themselves be valid when the state machine looks at them.

   As discussed in IBIPSim46.sily, the IB write enable signals are composed of select and timing terms. The timing terms are either WDuValid’ or WdvValid’ depending on which word in the double word is being written. If the Select terms are glitch-free, conditions 4b and 4c guarantee that the correct data will be written in the IB or MD destination latches.

   From IBIPSim46.sily, the write select terms are:

   IBWriteSelect0 = (Full.1 + Full.0)’ * (Full.1 + uvPC.0 + uvPC.1)

   IBWriteSelect1 = (Full.0 + Full.1)’ * (Full.0 + uvPC.2)

   Condition 4d guarantees these will be glitch-free even if the Emulator reads the IB just as the IFU is writing to it.

Critical Delay Assumptions	Comments
5. The delay from when a PCs_ clause causes the Full bits to be cleared until the Pause’ signal can be raised must be less than the time from the beginning of PhaseA to the Pause Point.	5. A Mesa jump instruction results in a change in the Mesa PC. This is implemented by having a microinstruction in the Emulator’s jump code execute a PCs_ clause. This, in turn, sets the least significant 16 bits of the virtual program counter (vPC) and the least significant 8 bits of the fetch Program counter (see IBIPSim44.sily - IBIPSim46.sily). It also causes the Instruction Buffer’s Full bits to be reset, indicating that the IB is empty and should be refilled with instruction stream data referenced by the new fetch PC. The decoded PCs_ signal is latched by PhaseA, hence it becomes valid soon after the beginning of PhaseA. It will start clearing the Full bits at this time. If the next microinstruction contains an IBAccess, likely an IBDisp, it must be possible to stop the clock while the Instruction Fetch unit retrives instruction stream data. In order to stop the clock, the Pause’ signal must be valid by the Pause Point. Thus, the propagation delay of the circuits that gate the PCs_ signal, clear the Full bits, set IBEmpty and can activate the Pause’ signal must be less than the interval between the beginning of PhaseA and the Pause Point.
6. The delay from when an IBConsume causes the Full bits to be cleared until the Pause’ signal can be raised must be less than the time from the beginning of PhaseB to the Pause Point.	6. This is similar to the case above except that the uvPC can’t be changed until PhaseB starts. This is because it is used to address the IB in PhaseA. This is potentially a much more serious constraint than #5 above because the circuit can’t start until the beginning of PhaseB.
7. The delay from when the Instruction Fetch machine causes fPCAtEndOfPage to be set until it can be used to cause an IBDispTrap must be less than the delay from the beginning of PhaseF to the beginning of the next PhaseB.	7. If a Mesa jump lands on the last byte of a page (actually any of the last 4 bytes) we must have an IBDispTrap before starting that first bytecode. The Emulator is written assuming an IBDispTrap will occur whenever the IB can’t fetch any more without crossing a page boundry. It is quite likely that the jump microcode will execute a PCs_ followed by an IBDisp. The IBDisp will not trap then because fPCAtEndOfPage should be false. Instead, the IBDisp will stall as the IFU fetches the last double word of the page. At the end of the fetch, the IFU will increment the new fPCd, which starts to set fPCAtEndOfPage. It will also set one of the Full bits, which resets IBEmpty and allows PhaseF of the microinstruction BEFORE the IBDisp to proceed. The actual IBDispTrap takes place in the next PhaseB. There, the vPC, uvPC and Full counters are held and the proper trap address is generated in NIA. While we have until the end of PhaseB to generate the trap address, we cannot begin to increment the counters in PhaseB, then go back. Hence, if the fPCd counter is incremented when the Full bit is set which is when IBEmpty is reset which is when Pause’ goes HI which is when the fixed delay section of PhaseF begins, fPCAtEndOfPage*IBDisp must inable the trap logic which disables the counter updates by the beginning of the next PhaseB. Note that if we attempt to fix the problem shown in #6 above by incrementing the IB counters in PhaseA, this condition becomes more restrictive. We would require the trap logic to be active by the beginning of PhaseA.

32. As seen on IBIPSim41.sily, the least significant bit of the A1 address group is F.15. This is a problem because the A1 bits should be ready by the end of PhaseA and the F bus bits aren't normally ready until the beginning of PhaseF. To fix this, a special circuit has been added that computes F.15 in PhaseA. To make the circuit fast, it only performs arithmetic operations (addition and subtraction). Thus, all microinstructions containing a MAR_ must form the address using some sort of arithmetic operation. Adding 0 is a legal arithmetic operation. The Dandelion allowed one to also send addresses via A-Bypass and one could use an OR operation. The A-Bypass has already disappeared, there is simply no data path from the A side of the R registers to the F Bus that avoids the ALU or LRot box. The OR operation was fairly rare anyway.

33. If the clocks are held too long, a time out signal starts them and causes a trap (IBIPClocking.sil, IBIPControl.sil). This time out can be disabled by some external pin. This is viewed as a debugging aid. Its a little difficult to see what the IBIP will do in the TimeOut trap since its only communication port (the AP Bus) is busted and the bus state is undefined.