Running HP 2000 Time-Shared BASIC on SIMH

J. David Bryan, 9-Dec-2017, revised 4-Sep-2020

The HP 2100 simulator for the 21xx and 1000 series of machines supports execution of the HP 2000 family of Time-Shared BASIC operating systems. The 2000A and 2000E products run on a single CPU, whereas the 2000B, 2000C, 2000F, and 2000 Access products use a dual-CPU configuration. The former present no special problem for execution on SIMH, but the latter require some considerations to run reliably.

The Dual-CPU Hardware Implementation

The HP 2000B, C, F, and Access versions split the TSB operating system into two parts running on separate CPUs. The primary CPU, designated as the *System Processor*, runs the BASIC interpreter for up to 32 concurrent users. The secondary CPU, designated as the *I/O Processor*, handles I/O through the terminal multiplexers. An HP 12875A Processor Interconnect kit provides communication between the SP and the IOP. The kit consists of two bidirectional 16-bit parallel interfaces installed in each CPU. One interface in each CPU is designated as the output interface, and the other is designated as the input interface. Cables cross-connect the interfaces between the CPUs.

Attached to the System Processor are the hard drives, the magnetic tape drives, and, for the B/C/F systems, the paper tape reader. The I/O Processor hosts the system line printer and, for the Access system, optional paper tape and card punches and readers. Each processor has a bootstrap loader residing permanently either in core or in ROM.

A consequence of this arrangement is that the device used to load the operating system into the IOP — the paper tape reader for B/C/F systems, the magnetic tape drive for Access — is connected to the SP. Therefore, a cross-load between CPUs must be performed. The IOP tape contains the cross-loader that runs on the SP, followed by the configured IOP operating system that is sent across the Interconnect to the IOP. Each program consists of a sequence of absolute binary records terminated by paper tape trailer or a magnetic tape mark.

The Hardware System Startup Process

To start these TSB systems, the system operator performs the following steps after applying power to both CPUs and all peripherals:

- The IOP bootstrap loader for the Processor Interconnect is run. The loader initializes the Interconnect and then sits in a loop waiting for the first word to arrive from the SP.

- The paper or magnetic tape containing the IOP program is mounted on the corresponding drive on the SP.

- The appropriate SP bootstrap loader is run to read the cross-loader program into SP memory. The loader halts the CPU when loading is complete.

- The cross-loader is run. This program reads the IOP operating system from tape and sends each byte across the Processor Interconnect to the IOP, where it is received by the IOP bootstrap loader and placed into memory. The SP guards against IOP failure by running a handshake timeout counter on each byte sent.

- When the cross-loader receives an end-of-tape or end-of-file indication, it sends a stream of zero-length records to the IOP. This causes the IOP bootstrap loader to halt its CPU after a loader-dependent number of empty records have been received.

- The next zero-length record sent by the SP is not acknowledged by the IOP (because it has halted). When the timeout counter expires, the SP halts.

- The IOP operating system is run. It initializes itself and then waits for a ***Start Timesharing*** command to be sent from the SP across the Processor Interconnect.

- The SP bootstrap loader for the system disc is run. This loads the TSB disc boot extension into memory. When the load is complete, the SP halts.

- The SP boot extension is run. This brings the rest of the SP operating system program into memory. The SP program initializes itself, prompts the system operator for the current date and time, sends a ***Start Timesharing*** command across the Processor Interconnect to the IOP, and then waits for an operator command or user logon. The system is now up and running.

Three aspects of the above sequence are critical to successful system startup:

- The IOP bootstrap loader must complete its initialization of the Processor Interconnect kit before the SP cross-loader sends its first byte across the link.

- The SP cross-loader must receive an acknowledgement within the timeout period for each byte sent to the IOP bootstrap loader.

- The IOP operating system must complete its initialization before the SP operating system sends the ***Start Timesharing*** command.

If the first sequence requirement is violated, initialization will clear the input register containing the first byte, and the IOP operating system will be corrupt. A timeout during the cross-load will cause early termination of the transmission, and again the IOP operating system will be corrupt. If the SP sends its ***Start Timesharing*** command before the IOP is ready, the command will be ignored. TSB will appear to start normally, but the IOP terminal multiplexer will be unresponsive.

In hardware, the inherent delays in the operator's actions ensure that the first and third requirements of the startup sequence are always met. Once started, the IOP bootstrap loader runs unimpeded, so the second requirement is also met.

Simulation of the Dual-CPU Configuration

The HP 2100 simulator supports execution of the dual-CPU TSB versions. Two simulator instances are used — one to run the SP program, and the other to run the IOP program. The *IPL* device simulates the Processor Interconnect kit, with a memory region shared between the two instances serving as the interconnecting cables. The simulator also provides the special IOP firmware that is required to run the 2000 Access version.

Given the complexity of starting a dual-CPU TSB system, it is desirable to use an automated command file that ends with the system ready for user logons. Starting the TSB system under simulation has the same requirements as in hardware. In addition, the use of shared memory to simulate the CPU interconnection imposes an additional requirement: the memory region to be shared between the simulator instances must be established before either the IOP or SP begins program execution. These requirements complicate the use of automated simulator startup command files to bring up a TSB system.

Starting a Time-Shared BASIC System on SIMH

In hardware, the I/O configuration and the interconnections between the processors are established by the physical locations of the I/O cards installed in the CPU card cages and the cabling between the Processor Interconnect cards. In simulation, these aspects are established by configuration commands executed after the two simulator instances are started but before any software is loaded.

Manual configuration of the SP and IOP simulator instances for each execution is tedious and error-prone. A better approach would be to have separate SP and IOP command files that configure the CPU and I/O device simulations, set up the shared memory region for the IPL devices, and mount the required disc and tape images on their respective devices. With such files, a direct simulation of the preceding sequence would require the user to:

- start an instance to process the SP command file, then
- start an instance to process the IOP command file, then
- start the IOP binary loader, then
- start the SP cross-loader, then
- start IOP operating system execution, and then
- start SP operating system execution.

The delays inherent in manually starting SP execution after the IOP is started guarantee that the IOP initializations have completed first.

Starting the system in this way is inconvenient. It is preferable to run a single command file that handles all of the subsidiary actions, including cross-loading the IOP operating software and starting the SP and IOP programs. The problem with such an approach is guaranteeing that the startup sequence requirements are met.

Problems with Automatic Startup Command Files

Given SP and IOP command files as outlined above, two options for TSB startup requiring only one user action are possible. One option is to start each instance with its associated command file from a host operating system shell script. The other is to start one instance with its associated command file and have that instance start the second instance with its command file.

As an example of the first approach, a UNIX Bash startup script might contain:

```
xterm -e hp2100 iop.sim &
hp2100 sp.sim
```

The equivalent Windows CMD script would be:

```
start hp2100 iop.sim
hp2100 sp.sim
```

A problem with this approach is that the two instances execute their command files asynchronously, so the requirement that the IOP starts executing machine instructions first cannot be guaranteed.

For the second approach, the user starts the first simulator instance manually, specifying the SP command file:

```
hp2100 sp.sim
```

...and at some point within the SP file, there is a SIMH spawn command:

```
! start hp2100 iop.sim
```

...that starts a second instance of the simulator to execute the IOP command file. This approach eliminates the need for the host system shell script and guarantees that commands in the SP file preceding the spawn of the IOP instance are executed before the IOP commands. However, this still does not guarantee that the IOP starts first, responds to handshakes within the timeout period, or completes its initializations before the SP program attempts communication. If the host operating system blocks the IOP instance from executing due to higher priority contending processes, the SP command file will continue through SP program loading and execution, which will fail when IOP communication is attempted.

This problem is inherent in having two independent simulator processes running on a multitasking host system. At any point during operation, the host operating system might block or preempt one of the two simulator instances without affecting the other. The implicit hardware assumption, that both CPUs are always free to run, does not hold in simulation. Moreover, there is no way to guarantee that both instances receive the same amount of host CPU time, so the startup sequence might succeed during one simulation session but fail on the next. Even if the instances were operated at the highest priority, they may still block when calling the host OS for terminal output.

Single-core host machines have an additional problem. TSB does not have a traditional localized idle loop that is executed while the operating system is otherwise unoccupied. Consequently, the SIMH ***SET CPU IDLE*** command does not idle the simulator when running TSB, and therefore the SP and IOP instances take 100% of their available host CPU times. On a single-core machine, the instances generally will time-share only if they are run at the same priority. However, starting an instance in the background on UNIX with the ***&*** operator automatically drops the execution priority relative to the foreground instance. The same situation pertains on Windows to a different degree: a window displayed on top of another window will receive a priority boost relative to the partially hidden one. As both instances execute continuously, the lower-priority instance will be starved of host CPU time until the higher-priority instance blocks for I/O.

Multi-core PCs do not solve the problem. Even when each instance has its own CPU core and additional cores are available to handle all other system threads, an instance may still block on host disc I/O for, e.g., demand page loading. The situation is worse if other host processes must contend for a limited number of cores — the SP may execute either concurrently with the IOP or while the IOP is blocked by processor contention. A complicating factor is that SIMH automatically decreases its own process priority when simulated execution begins with a ***RUN*** or ***GO*** command. Consequently, an instance executing SCP commands has priority over an instance executing TSB system code. Successful system startup then becomes much more sensitive to the timing of operations within the respective simulator command files.

Even with wholly manual operation of the two simulator instances, the problem remains that the IOP may block while the SP cross-loader is running. Given that the simulator runs about ten times faster than a real HP 2100, the SP will time out should the IOP block for more than about 60 milliseconds, and the load will fail.

In hardware, the IOP takes a deterministic time to respond to a cross-load handshake or execute initialization code. In simulation, this time will vary from some minimum lower limit to essentially an unbounded upper limit, depending on host system load. Constraining the startup sequence within these wide limits is the key to successful operation.

There are five possible approaches to sequencing the two instances, listed here in order of increasing probability of startup success:

- Run asynchronously, and hope for the best.
- Insert timed delays (sleeps) at appropriate points in the command files.
- Insert semaphore-based waits and signals in the command files.
- Rendezvous the two instances periodically to interlock execution.

- Provide two CPUs within a single simulator instance.

With carefully designed command files, the first approach may work on multi-core, lightly loaded host systems. However, success is problematic, and failure may require reloading the system from the last ***HIBERNATE*** tape if the IOP does not respond after the SP has started.

Timed delays help a bit, but the delays must encompass the longest possible IOP preemption, which is impossible to predict. In most cases, the delays will be longer than necessary, yielding slow startup progress. Moreover, delays cannot be inserted within program execution, so the preemption problem during cross-loading still exists with this mechanism.

Having one instance wait on a semaphore until the other instance signals it improves matters substantially over timed delays. The waits at the various critical points of the startup sequence become minimal — if the second instance has already signaled the semaphore, the first instance will proceed directly through the wait without waiting. Preemption of the signaling instance automatically forces the other instance to wait until the first instance resumes. However, because the waits are at the command level, preemption during instruction execution can still cause the startup process to fail.

Execution interlock synchronization allows both instances to execute freely for an agreed number of machine instructions. When that number is reached, each instance attempts to rendezvous with the other instance before proceeding. The first instance to reach the rendezvous point waits for the other instance to arrive, whereupon both instances resume execution for the set number of machine instructions. Rendezvous, implemented by a semaphore-controlled execution gate, ensures that preemption of one instance will cause the other instance to wait automatically at the closed gate. However, the synchronization is only relative, in that the instances can diverge in execution up to the instruction limit. In addition, constant calls to the host operating system to manipulate the semaphore impose an overhead that reduces simulation speed. Another problem is that when one instance halts, such as at completion of the initial cross-load, the other instance suspends execution at the rendezvous and will not resume until either the first instance continues execution or interlocking is canceled.

The only way to guarantee that the SP and IOP instances remain synchronized through preemption by the host operating system is to redesign the simulator to provide both CPUs within a single simulator instance. This is possible; Bob Supnik wrote an unreleased simulator for a six-way symmetric multiprocessor system that interleaved instruction execution among the processors. Because the CPUs execute in lock-step with one instruction executing in each CPU per cycle, blocking the simulator instance blocks all CPUs simultaneously. However, it is not practicable to retrofit this approach to the existing HP2100 simulator, due to the extensive restructuring required. Nor is it necessarily desirable; with both CPUs executing continuously, each would execute machine instructions for one-half of the time and so would run at one-half of the speed of an independent instance.

Of the five methods, combining process synchronization using waits and signals at the command level with rendezvous at the execution level provides the best control with the lowest overhead.

Controlling TSB Startup with Process Synchronization

While the execution order of SP and IOP command files is strictly sequential, the order in which the commands interleave in the overall sequence varies with system load. Constraints may be imposed on the order by synchronizing the processes at certain critical points. Essentially, this involves suspending one process until the other process "catches up" to the same point in the startup sequence.

Release 28 of the HP 2100 simulator added two SCP commands for this purpose. The ***SET IPL WAIT*** command suspends a simulator process until a matching ***SET IPL SIGNAL*** command is issued by the other process. If the signal command is issued before the wait command, the latter will not suspend but instead will proceed immediately with the next command. These commands control command-file execution but not CPU execution.

Release 30 extended synchronization to the execution of machine instructions. The ***SET IPL INTERLOCK=<count>*** command sets up a rendezvous point with a closed gate and directs each instance to rendezvous after executing the specified number of instructions. The first instance that arrives at the gate waits until the other instance arrives to open it. After both instances resume, the gate closes, awaiting another rendezvous. The effect is that neither instance can get more than ***<count>*** instructions out of step with the other.

As an example, consider the 2000F cross-loading sequence that loads the IOP operating system. From the foregoing discussion, the critical requirement is that the IOP bootstrap loader must be running and have completed its initialization before the SP cross-loader is started. An SP command file that provides the required startup sequence contains these lines (in part):

```
; Load the cross-loader from the paper tape reader.

attach -E PTR IOP.tape
boot PTR

; Verify that the load was successful.

assert T=102077

; Start the IOP simulator instance to receive the IOP program.

! start hp2100 iop.sim

; Start the cross-loader to transfer the IOP program.

deposit P 000002
go
```

```
; Verify that the cross-load was successful.

assert T=102077
```

...where the ***start*** command above is used to spawn an asynchronous HP 2100 instance to execute the ***iop.sim*** command file, and the ***assert*** commands are used to ensure that the bootstrap and cross-load transfers completed successfully with HLT 77 instructions. The 2000F cross-loader is a simple assembly language program:

```
TWP   STC PR,C      START PHOTOREADER
      SFS PR        WAIT FOR A CHARACTER
      JMP *-1
      LIA PR        GET CHARACTER FROM PHOTOREADER
      CLB           RESET COUNTER
TWP10 SFC C2        WAIT FOR IOP TO ACKNOWLEDGE
      JMP TWP30
      INB,SZB       INCREMENT TIME-OUT COUNTER
      JMP TWP10
      CLC PR        TIMED OUT - TURN OFF
      HLT 77B         PHOTO READER AND HALT
      JMP TWP
TWP30 OTA C2        OUTPUT CHARACTER
      STC C2,C      SET FLAG
      JMP TWP       GET NEXT CHARACTER
```

The critical point is the OTA C2 instruction at label TWP30. This outputs the first byte to the Processor Interconnect. Initialization of the Interconnect residing in the IOP must have occurred by this point in the SP's execution, or the byte will be lost, and the cross-load will fail.

The corresponding IOP command file contains:

```
; Start the IOP binary loader to receive the program.

deposit S 000000
boot IPL

; Verify that the load was successful

assert T=102077
```

The ***boot IPL*** command loads the Processor Interconnect bootstrap into memory and executes it. The simulator provides a special Basic Binary Loader bootstrap that begins with this assembly language sequence:

```
BOOT  LDA DPISC,I   GET INPUT CARD SELECT CODE
      JMP CONFG     CONFIGURE I/O INSTRUCTIONS
START CLC 0,C       INITIALIZE THE INTERCONNECT
      [...]

CONFG ADA SFS       CONFIGURE THE
      STA SFS.C       SKIP-IF-FLAG-SET INSTRUCTION
      ADA STC       CONFIGURE THE
      STA STC.C       SET-CONTROL INSTRUCTION
```

```
          ADA MIB        CONFIGURE THE
          STA MIB.C        MERGE-INTO-B INSTRUCTION
          JMP START      START THE TRANSFER
```

The critical point here is the CLC 0,C instruction at label START. This clears the Interconnect card, including its input data register. The IOP must execute this instruction before the SP gets to label TWP30 in its loader, or data will be lost.

Starting at label BOOT, the IOP executes 10 instructions to reach the critical point. Starting at label TWP in the cross-loader, the SP executes 6 + 2 * TIME instructions to reach the critical point, where TIME is the photoreader fast read time. With the default time (100 event ticks), the SP reaches the critical point after 206 instructions. Therefore, the sequence can be guaranteed by placing a ***SET IPL INTERLOCK*** command in the SP command file immediately before the spawn command that starts the IOP instance and specifying an interlock value greater than 10. If the IOP instance start is delayed or is blocked after starting, the SP will pause for a rendezvous while executing the SFS PR / JMP *-1 loop and will not resume until the IOP has executed past the CLC 0,C initialization.

Another critical point occurs after cross-loading when the IOP and SP operating systems are run. Using the CPU instruction tracing capability (***SET CPU DEBUG=INSTR***), the IOP is seen to complete its initialization after approximately 260,000 instructions, and the SP sends the ***Start Timesharing*** command after about 1,000,000 instructions. So specifying any reasonable interlock value will ensure that the IOP is ready for the first SP communication.

As a second example, consider the 2000 Access IOP operating system generator. This program runs on the SP to generate a new IOP configuration. When generation is complete, the SP cross-loads the new system to the IOP. Then, if a copy is to be saved to magnetic tape, the program is cross-loaded back to the SP in memory-image format. In hardware, the system operator follows this sequence:

- After answering the generation questions, the SP prints START IOP PROTECTED LOADER. PRESS RETURN.
- The IOP bootstrap loader for the Processor Interconnect is run.
- Pressing the RETURN key on the system console transfers the new system to the IOP.
- When the transfer completes, the IOP bootstrap loader halts.
- When the SP times out, it prints MOUNT IOP COPY TAPE. PRESS RETURN.
- The generator tape is removed from the magnetic tape drive and a blank tape is mounted in its place.
- After pressing the RETURN key on the system console, the SP prints START IOP AT LOCATION 2002.

- The P register on the IOP is set to 2002, and the RUN button is pressed.

- The IOP program is transferred to the SP and written to tape. When the transfer completes, the IOP halts.

- The SP prints the system generation memory and entry point map. When the map is complete, the SP halts.

Execution interlocking is used as above to transfer the new operating system from the SP to the IOP. However, interlocking must be disabled to restore asynchronous operation after the IOP bootstrap halts. Otherwise, the SP cross-loader would not time out; instead, it would pause in its timeout loop to wait for the IOP to open the execution gate, which cannot occur if the IOP is halted.

But there is a problem: with interlocking disabled, the IOP instance would be free to start execution at location 2002 before the SP had printed the prompt message and readied itself to receive the program from the IOP. The HP 2000 Access Operator's Manual cautions, "Never attempt to start the IOP at location 2002 until you receive this message. If you do, you will have to configure the IOP again."

Therefore, we use the ***WAIT*** and ***SIGNAL*** commands to synchronize the instances. In the IOP command file after the bootstrap loader halts, we have:

```
; Verify that we had a successful cross-load.

assert T=102077

; Resume asynchronous operation to allow the SP to run freely.

set IPL INTERLOCK=0

; Wait for the operator to mount the new tape.

set IPL WAIT

; The SP is now ready.  Start the IOP to dump memory across the IPL.

deposit P 002002
go

; Verify that the dump succeeded.

assert T=102077
```

In the corresponding place in the SP command file, we have:

```
; Continue until the IOP halts and then mount the new tape.

go until "MOUNT IOP COPY TAPE.  PRESS RETURN"

detach MSC0
attach -N MSC0 IOPCOPY.tape
```

```
reply "\r"

; Continue until the SP is ready to receive the IOP copy.

go until "START IOP AT LOCATION 2002"

; Re-enable execution synchronization to ensure that the IOP is ready
; to dump when we are ready to receive.

set IPL INTERLOCK=50

; Signal the IOP that the SP is now ready to receive the program copy.

set IPL SIGNAL

; Receive the IOP program and write it to the mounted tape.

go

; Verify that we had a successful copy.

assert T=102077
```

Some host platforms do not support the underlying system library routines necessary to provide process synchronization. In this case, the ***SET IPL WAIT*** command falls back to a simple two-second timed wait, which may provide enough host processor time for the other process to reach its ***SET IPL SIGNAL*** command (which, in fallback mode, has no effect). If this is insufficient, a ***DEPOSIT IPL WAIT <n>*** command may be used to lengthen the pause.

The IPLO device writes a trace line when an ***ATTACH*** command is issued and the host system does not support process synchronization. For example:

```
set debug stdout
set IPLO DEBUG=STATE

attach -S IPL 1
>>IPLO state: Synchronization is unsupported on this system; using
   fallback
```

The Effect of Throttling on System Startup

As noted earlier, HP 2000 Time-Shared BASIC does not have a traditional idle loop that is executed while the operating system is otherwise unoccupied. Consequently, the SP and IOP instances take 100% of their available host CPU times.

The SP and IOP command files may contain ***SET THROTTLE*** commands to reduce the loads on the host CPU. However, because throttling periodically preempts the SIMH process, use of instruction interlocking is imperative to achieve a successful system startup. Throttling intentionally reduces simulation speed, so it is best employed only after IOP initialization is complete.

The Race Conditions in 2000 Access

The 2000 Access version has two race conditions: one that manifests itself by an apparently normal boot and operational system console but no ***PLEASE LOG IN*** response to terminals connected to the multiplexer, and another that causes a user program's printer or paper tape punch output to stop for no apparent reason.

The first race occurs during SP system loading, and the cause is this code in the SP disc loader (source files S2883, S7900, S79X0, S79X3, and S79XX on the Access source tape):

```
LDA SDVTR      REQUEST
JSB IOPMA,I      DEVICE TABLE
[...]
STC DMAHS,C    TURN ON DMA
SFS DMAHS      WAIT FOR
JMP *-1          DEVICE TABLE
STC CH2,C      SET CORRECT
CLC CH2          FLAG DIRECTION
```

DMA completion causes the SFS instruction to skip the JMP. The STC/CLC pair at the end normally would cause a Processor Interconnect interrupt and a second ***Request Device Table*** command to be recognized by the IOP, except that the IOP DMA setup routine *DMAXF* (in source file SD61) specifies an end-of-block CLC that holds off the interconnect interrupt, and the DMA interrupt completion routine *DMCMP* ends with a STC,C that clears the interconnect flag.

The SP program executes four instructions between DMA completion and the CLC. The IOP program executes 34 instructions between the DMA completion interrupt and the STC,C that resets the Processor Interconnect. In hardware, the two CPUs are essentially interlocked by the DMA transfer, and DMA completion occurs almost simultaneously in each machine. Therefore, the STC/CLC in the SP is guaranteed to occur before the STC,C in the IOP, and the Processor Interconnect interrupt never occurs. Under simulation, and especially on multi-core hosts, that guarantee does not hold. If host load preemption causes the STC/CLC to occur after the STC,C, then the IOP starts a second device table DMA transfer, which the SP is not expecting. Consequently, the IOP never processes the subsequent ***Start Timesharing*** command, and the multiplexer does not respond to user logon requests.

The ***Request Device Table*** command is sent between the ***LOAD OR DUMP COMMANDS?*** and ***DATE?*** prompts that appear on the system console. Enabling instruction interlocking across this code avoids the race. The interlock value is critical; it cannot be more than 16 instructions to allow for the worst-case preemption scenario. That occurs when the SP instance does not receive the last DMA input word during a poll that occurs at the last instruction of the SP's interlock quantum, and then the IOP outputs the last DMA word with the first instruction of its quantum. The IOP will then execute a full quantum (16 instructions) and rendezvous with the SP. If the SP blocks immediately after rendezvous, the IOP can then execute a second full quantum (another 16 instructions) before the SP is able to pick up the last word and execute its CLC. To avoid this, two interlock times must be less than the critical instruction path length of 34 instructions.

Synchronization must remain active at least until the IOP has completed its initialization. If synchronization events are not supported on the host platform, the simulator employs a workaround that decreases the incidence of the problem: the DMA output completion interrupt is delayed to allow the other SIMH instance a chance to process its own DMA input completion interrupt first. This improves the race condition by delaying the IOP until the SP has a chance to receive the last word, recognize its own DMA input completion, drop out of the SFS loop, and execute the STC/CLC. The delay is initially set to one millisecond but is exposed via a hidden IPLI register, *EDTDELAY*, that allows the user to lengthen the delay if necessary.

Using this fallback mechanism instead of CPU synchronization only improves the condition. It does not solve it because delaying the IOP does not guarantee that the SP will actually execute. It is possible that a higher-priority host process will preempt the SP, and that at the delay expiration, the SP still has not executed the STC/CLC. Still, in testing, the incidence dropped dramatically, so the problem is much less intrusive.

The second race occurs when a user program writes to a system line printer or paper tape punch. The incidence is higher when a large amount of output is generated quickly. The cause is this code in subroutine #IPAL in the SP main program source (STSB) that is used to send output data to a non-shareable device controlled by the IOP:

```
LDA ERTMP     SEND
IOR ALB         REQUEST
JSB SDVRP,I     CODE
LDB #IPAL     RETRIEVE
INB             BUFFER
LDA B,I           LENGTH
JSB SDVRP,I   SEND TO IOP
SFS CH2       WAIT FOR
JMP *-1         ACKNOWLEDGEMENT
CLF 0         INHIBIT INTERRUPTS
LIA CH2       RETRIEVE RESPONSE
```

If the ***Allocate Buffer*** ("ALB") request is refused by the IOP because all output buffers are in use, the resulting "No Buffer Available" response will cause the SP to issue a ***Release Buffers*** command to the IOP and then suspend the user's program. When the line printer completes its operation on the current output buffer, the IOP releases it and then indicates buffer availability by sending a ***Wake Up User*** command to the SP to retry the allocation request.

The problem occurs when the line printer finishes a line just as the ***Allocate Buffer*** request is made. That request sends two words: the ALB request code and the buffer length. After receiving the second word, the IOP finds that all buffers are in use and denies the request. If the line printer completion then arrives, the IOP immediately sends a ***Wake Up User*** command to indicate that a buffer is now available. If the command arrives between the time the SP sends the buffer length word and the time it retrieves the response, the user program will hang. This occurs because the "No Buffer Available" response subsequently causes the SP to set a flag to indicate that the user has been suspended for buffer availability. If the ***Wake Up User*** command arrives after the response is received but before that flag is set, the command is ignored. So the SP is waiting for the IOP to issue the command, but the IOP has already issued it. This leaves the suspension in force until the user aborts the program by pressing the BREAK key.

Nominally, there are only about nine instructions executed between the STC CH2 (within the SDVRP subroutine) that causes an IOP to accept the buffer length word and the SFS CH2 (above) that detects that the IOP's acknowledgement has arrived. Once detected, the CLF 0 instruction turns the interrupt system off so that commands received from the IOP are deferred until after the user suspension flag is set.

However, the SP's interrupt system is on during the above SFS CH2/JMP *-1 loop, and the Processor Interconnect has the highest I/O interrupt priority. So if, say, a time-base generator interrupt occurs during the SFS loop, several dozen instructions may be executed before control returns to the loop. If, during that time, the IOP command arrives, the resulting higher-priority interrupt is handled before the loop return, and the command is ignored because the "user is suspended" flag has not been set.

Simulation works around this problem by arranging the card service routine to ensure that the IOP status response is picked up before an IOP command if they both are seen during the same input poll. However, there is no general way to ensure that the response is processed by the SP program before a pending IOP command is recognized. That is because the SP does not read the responses of some commands it sends, so simply holding off input card commands until the output card data register is read will not work.

The best we can do to reduce the frequency of the race condition is to delay IOP command recognition after a status response arrives. The IPL simulation does this by rescheduling the poll using a delay of ten times the normal maximum poll delay to give any intervening interrupt handlers time to complete. The next STC directed to either card clears the delay and reschedules the poll for immediate entry, providing rapid response when block data is being transferred in either direction across the Processor Interconnect.

Summary

The HP 2000B, C, F, and Access dual-CPU Time-Shared BASIC operating systems will run on the HP 2100 simulator, but several internal aspects must be considered for successful startup and operation. These arise from the differences in behavior of dedicated hardware versus two simulator instances running on a multi-core host machine. Arbitrary host OS preemption of the separate SP and IOP instances means that the deterministic behavior of the original HP hardware cannot be guaranteed. However, the use of process synchronization commands during system startup will improve the probability of successful system execution.