The Evolution of the HP 21xx/1000 I/O Simulation

J. David Bryan, 15-Mar-2022

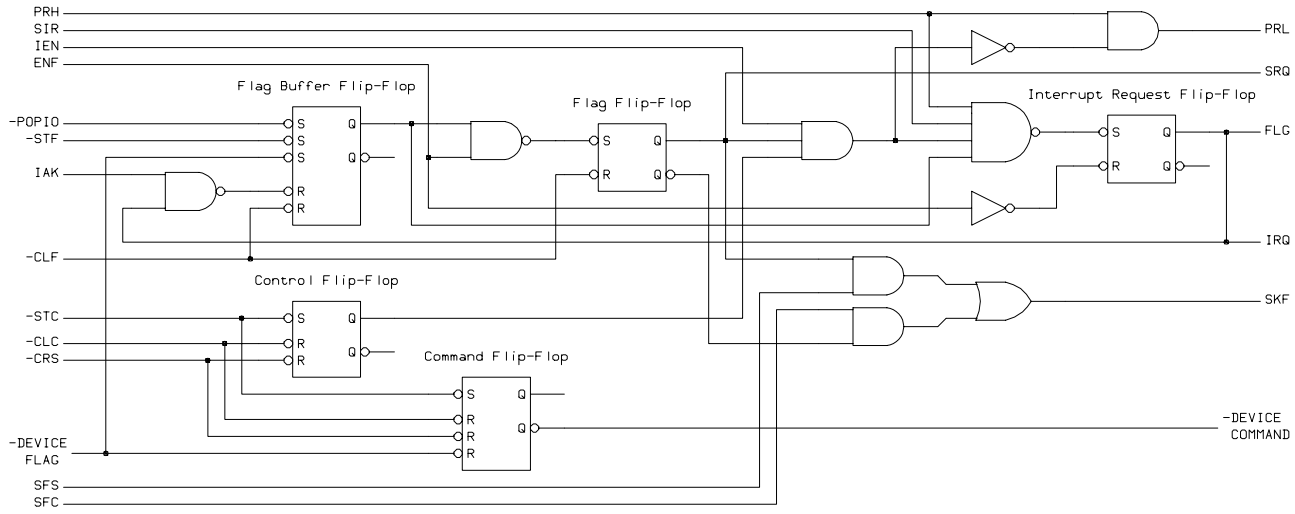
The I/O subsystem design of the simulator for the HP 21xx and 1000 series of machines has undergone several revisions over the past fifteen years. These were driven by limitations in each of the various designs that became apparent as increasingly complex I/O interfaces were added. With each revision, the simulation and the hardware implementation have converged. The current model simulates the hardware signals closely.

The HP I/O Hardware Structure

The design of the I/O system is compatible across all HP 21xx/1000-M/E/F systems. The I/O backplane distributes a 16-bit data output path, a 16-bit data input path, and control and timing signals to the interface cards. All I/O card slots in the backplane are electrically interchangeable, and an interface derives its I/O address (called the *select code*) and its interrupt priority from the slot into which it is installed. Lower-numbered slots have priority over higher-numbered ones.

An I/O timing cycle is divided into five periods designated T2 through T6. On the early hardwired machines (2114-2116), these form a subset of, and are synchronous with, the machine cycle that occupies T0-T7. On the later microprogrammed machines (2100 and 1000), each microcycle occupies one T-period; the micromachine runs asynchronously with the I/O subsystem and synchronizes whenever an I/O micro-order is executed. Backplane signals are asserted during specific T-periods to control the timing of the interfaces.

The control logic of the typical HP I/O interface is shown below in simplified form:



The logical state machine is constructed from five S-R (set-reset) flip-flops, typically implemented as cross-connected NAND gates. The device side of the interface uses three of these: the Control, Command, and Flag Buffer flip-flops. The other two — the Flag and Interrupt Request flip-flops — form the CPU side. In the idle state, the Flag Buffer and Flag flip-flops are set, and the Control, Command, and IRQ flip-flops are clear.

The CPU sends these backplane signals to the interface:

- PRH — priority high
- SIR — set interrupt request (periodic at T5)
- IEN — interrupt enable
- ENF — enable flag (periodic at T2)
- POPIO — power-on preset to I/O
- STF — set the flag flip-flop
- IAK — interrupt acknowledge
- CLF — clear the flag flip-flop
- STC — set the control flip-flop
- CLC — clear the control flip-flop
- CRS — control reset
- SFC — skip if the flag is clear
- SFS — skip if the flag is set

The CPU receives these backplane signals from the interface:

- PRL — priority low
- SRQ — service request
- FLG — flag
- IRQ — interrupt request
- SKF — skip on flag

The interface asserts DEVICE COMMAND to the device, and the device asserts DEVICE FLAG to the interface.

To perform a device operation, the CPU executes a programmed “Set Control and Clear Flag” instruction, which asserts the STC and CLF signals on the I/O backplane to set the Control and Command flip-flops and clear the Flag Buffer and Flag flip-flops. The Command flip-flop output asserts the DEVICE COMMAND signal to the I/O device, which initiates the operation. When the operation completes, the device asserts the DEVICE FLAG signal to set the Flag Buffer flip-flop and clear the Command flip-flop, denying the DEVICE COMMAND signal to the device. In response, the device denies the DEVICE FLAG signal to complete the operation.

Because device completion occurs asynchronously with I/O timing, the Flag Buffer output is qualified with the periodic ENF signal, so that the Flag flip-flop will set only during the T2 I/O period. If DMA is controlling the interface, the SRQ signal is asserted to initiate the next I/O cycle.

The state of the Flag flip-flop may be tested under program control. A programmed “Skip if Flag Set” or “Skip if Flag Clear” instruction asserts either the SFS or SFC signal to test whether the flip-flop is set or clear, and the interface responds by asserting the SKF signal if the Flag is in the state indicated by the request. This advances the CPU program counter,

causing the next instruction to be skipped to indicate that the programmed condition was met.

The Control and Flag flip-flop outputs are qualified with IEN to deny PRL to all lower-priority interfaces. Interrupt priority is established through the chain of PRH and PRL signals passing from interface to interface across the I/O backplane. Each interface receives its PRH from the PRL output of the next-higher-priority (lower select code) I/O slot and supplies its PRL to the PRH input of the next-lower-priority (higher select code) slot. This chain is broken at the first interface that has its Control and Flag flip-flops set when interrupts are enabled.

The Flag Buffer, Flag, and Control flip-flop outputs are qualified with PRH, IEN, and SIR (a periodic signal asserted during period T5) to set the Interrupt Request flip-flop and thereby assert the IRQ and FLG signals to the CPU. Only one interface can assert IRQ and FLG at a given time — the highest-priority interface. That interface will deny PRL to all lower-priority interfaces, which will inhibit those interfaces from setting their Interrupt Request flip-flops, even though they may have their Flag Buffer, Flag, and Control flip-flops set.

Asserting IRQ and FLG together causes the CPU to perform an interrupt acknowledgment cycle and execute the instruction residing in the main memory location corresponding to the interface's select code — typically, a jump-to-subroutine call to the interrupt service routine. Interrupt acknowledgement asserts the IAK signal, which clears the Flag Buffer and leaves the Control and Flag flip-flops set to continue holding off lower-priority devices until the service routine completes. The ENF signal clears the Interrupt Request flip-flop at T2 to complete the interrupt acknowledgement cycle.

The IRQ and FLG signals always assert together on HP 21xx/1000 machines. They are physically tied together on all interface cards, and the CPUs depend on their concurrent assertion for correct operation. In simulation, we check only IRQ from the interface, although FLG will be asserted as well.

When power is initially applied to the CPU or the front panel PRESET button is pressed, the POPIO and CRS signals are asserted simultaneously to initialize all of the installed interfaces to the idle state in preparation for running programs. A programmed "Clear Control" instruction directed to select code 0 also asserts the CRS signal. A "Clear Control" instruction directed to the select code of a specific interface clears the Control and Command flip-flops, which turns off the associated I/O device and inhibits interrupt generation.

An important consideration is that while the foregoing structure is common, it is neither required nor universal. An interface card is free to drive the backplane signals in any manner that meets the I/O timing requirements. Indeed, several interfaces depart from the standard implementation, either to improve I/O transfer speed, or to meet particular device control requirements.

The Original SIMH Implementation

At its origin in revision 2.5, the HP simulator embodied the typical interface structure mentioned above. The device's reset routine simulated power-on and PRESET. The

DEVICE structure's context value pointed to a device information block (DIB) that contained variables representing the interface's select code, the Flag Buffer, Flag, Control, and Command flip-flops, and a pointer to the interface simulator's I/O instruction handler. The CPU simulator dispatched I/O instructions to the addressed device, which altered its four flip-flop values accordingly.

The CPU simulator calculated DMA service requests by inspecting the flag values of each interface. Interrupt requests were calculated by ANDing the control, flag, and flag buffer values from each interface, and the priority chain was determined by ANDing the control and flag values. During interrupt acknowledgement, the CPU cleared the flag buffer value of the interrupting interface.

Because these calculations were done after each I/O operation, the values were stored in bit vectors during simulation runs for speed. However, to allow I/O device select codes to be reassigned during simulation stops and to allow user alteration of the flip-flop states, the values had to be stored in the DIBs. To accommodate both requirements, the values were copied between the DIBs and the bit vectors each time simulated execution started and stopped.

Problems with the Initial Implementation

This implementation worked well with the original set of devices supplied with the HP simulator. As new devices were added, though, minor issues arose, due to interfaces that did not follow the standard design.

For example, the Control flip-flops on the 12606B Fixed-Head Disc Memory and 12610B Drum Memory interfaces that were added at version 2.9 are not tied into the interrupt request logic, so setting the Control, Flag, and Flag Buffer flip-flops does not generate an interrupt. Because interrupt generation was calculated in the CPU simulator, the interface simulator had to use the command variable to represent the Control flip-flop to avoid generating interrupt requests inappropriately.

At version 3.2, CPU interrupt acknowledgement handling added a special case for the 12581A and 12892B Memory Protect cards, as they clear both the Flag and Flag Buffer flip-flops in response to IAK. Also at 3.2, the DMA service requests were separated from the flags, and a new SRQ value was added to the DIBs, as the forthcoming 13037D Disc Controller simulator required separate control over these two values. The new disc controller required notification of DMA transfer completion as well, so the EDT (end of data transfer) backplane signal was dispatched to all interfaces as a pseudo-I/O instruction.

At version 3.3, the ***–P*** option to the ***RESET*** command was added to allow the device reset routines to differentiate between power-up reset and ordinary reset. The 12578A and 12895A DMA simulators had been clearing their control words as part of the reset handler. While this is correct for power-up, it is not correct for PRESET, and this error manifested itself in RTE "slow bootstrap" failures.

At version 3.6, the CRS backplane signal was introduced as another pseudo-I/O instruction. The original implementation had sent a CLC instruction to each interface in response to a CLC 0 execution. Most interfaces respond to CRS and CLC identically by clearing their

Control and Command flip-flops. However, not all do. In particular, the DMA cards clear Control in response to CLC but Control and Command in response to CRS. Clearing the Command flip-flop stops an in-progress DMA operation, which the original implementation failed to do.

A Design Impasse

For version 3.8-1, a simulation of the 12936A and 12620A Privileged Interrupt Fences was planned. The PIFs are required to run the 12920A Terminal Multiplexer under the DOS and RTE operating systems, respectively. These systems run with the interrupt system off when servicing any device, as they are not reentrant. The 12920A is not buffered and will lose characters if it requests service while the interrupt system is off. The PIF is used in conjunction with special multiplexer drivers to break the priority chain to all lower-priority devices, allowing the interrupt system to remain on. This allows the higher-priority multiplexer to be serviced immediately, even during execution of a lower-priority device interrupt handler.

The 12936A has a unique behavior. Setting either the Control or Flag flip-flop denies PRL. IRQ asserts when the Flag and Flag Buffer flip-flops are set and the Control flip-flop is clear. The Flag and Flag Buffer flip-flops are cleared by asserting the CLF signal but set by asserting the IOO ("I/O Data Output") signal that is normally used to strobe data from the CPU into the interface's output data register. The STF signal is ignored.

This presented a problem because of the implicit assumptions of the roles of the control, flag, and flag buffer variables by the CPU simulator. The only way that those assumptions could be maintained was if the PIF simulator made these translations between its internal flip-flop values and those maintained by the CPU:

$$CONTROL' = CONTROL + FLAG * FLAGBUF$$

$$FLAG' = 1$$

$$FLAGBUF' = \overline{CONTROL} * FLAG$$

The prime values would be presented to the CPU as the device flip-flop values, while the original values would be presented to the user when the device state was examined. While this would coerce the CPU into generating the correct interrupt request and priority chain behavior, interrupt acknowledgement would clear the flag buffer, which would have to be reset to the indicated value for proper operation. Fortunately, the correct value could be restored during processing of the STF instruction that would be sent to the card by the OS interrupt handler. Unfortunately, the visible state presented to the user would be wrong between these two events. Equally unfortunately, user alteration of the visible values would not be reflected in the translated values, because the CPU simulator simply copied the DIB values to the bit vectors when simulated execution began.

The choices, then, were to accept that the state display would be wrong and disallow user changes to the flip-flop values, to add more special cases to the CPU simulator to accommodate the atypical I/O behavior, or to remodel the I/O simulation to allow interfaces to set the interrupt request and priority chain values directly. Given that the existing implementation embodied assumptions that were not valid across all I/O interfaces, and

given that the number of special cases was increasing as the breadth of the HP device simulations increased, implementation of an I/O model closer to the actual hardware was selected.

The First Revised I/O Implementation

Whereas the old model was based on dispatching I/O instructions, the revised model was based on dispatching I/O backplane signals. This allowed the interface to take whatever action it wanted in response. Instead of examining the Control, Flag, and Flag Buffer flip-flop values, the CPU monitored vectors representing the PRL, IRQ, SRQ, and SKF output signals from the interface simulators. The CPU sent the aforementioned list of input signals to the interface, along with these additional HP I/O backplane signals:

- IOI — I/O data input
- IOO — I/O data output
- EDT — end of data transfer
- PON — power on normal

The first two signals are generated as a result of I/O instructions: the LIA/B and MIA/B instructions generate IOI, and the OTA/B instructions generate IOO. EDT occurs at the end of a DMA transfer, and PON is sent as part of the power-on processing.

In addition to allowing more flexibility in interface design, the new implementation had a few other advantages:

- A more consistent structure (only signals were handled, rather than a mixture of signals and I/O instructions).
- Elimination of special cases in the CPU simulator (each interface simulator determined its own responses).
- Elimination of the flip-flop values from the DIB and of copying values between the DIB and the bit vectors (the CPU no longer examined flip-flop values, and the bit vectors were set at simulated execution start by sending SIR to all devices; no action was needed at execution stop).
- Unified handling of flip-flop values (values existed in one place — as local variables in the individual device simulators — rather than in the DIB and in the bit vectors, reducing coding error potential).
- Simplification of CPU interrupt determination (only the IRQ and PRL vectors needed to be examined, rather than combinations of the three flip-flop vectors).
- Simpler handling of power-on and preset conditions (the device reset function simply dispatched PON and/or POPIO and CRS to the signal handler; no duplication of the initialization code).

For efficiency, the simulator did not implement signal generation exactly as in the hardware, where ENF and SIR are periodic, PON is asserted continuously, and most signals are

common to all interfaces and are qualified at the interface by the select code. Under simulation, signals were sent only when actions were to be taken and then only to the specific target device. For instance, PON was dispatched only once during power-on reset, rather than being included in every I/O cycle. SIR was dispatched only when flip-flops affecting the PRL, IRQ, or SRQ signals were changed, rather than after every instruction. ENF was sent only when the device indicated that the Flag Buffer and Flag flip-flops were to be set, whereas in hardware, ENF samples the Flag Buffer value at every T2 and sets the Flag accordingly.

A Problem with the Revised Implementation

The initial revised I/O implementation modeled the parallel hardware backplane as a sequence of individual signal dispatches, with each signal assigned an enumeration value. For programmed I/O instructions, the CPU simulator sent single signals (e.g., STC) or a signal pair (e.g., STC + CLF) to the target device's signal handler. The CLF enumeration value was chosen so that the handler could separate the two signals from the sum.

While the CPU asserts at most two signals concurrently, DMA may assert up to five. A normal DMA I/O cycle consists of an IOI or IOO signal to transfer the data, a CLF signal to clear the Flag Buffer and Flag flip-flops to complete the prior I/O request, and an optional STC signal to set the Command flip-flop to begin the next request. In addition to these three signals, the last DMA cycle adds an EDT signal to indicate the end of the DMA transfer and an optional CLC signal to idle the device. These signals are asserted in specific T-periods, as follows:

	Input		Output	
Signal	**Normal Cycle**	**Last Cycle**	**Normal Cycle**	**Last Cycle**
IOI	T2-T3	T2-T3		
IOO			T3-T4	T3-T4
STC *	T3		T3	T3
CLC *		T3-T4		T3-T4
CLF	T3		T3	T3
EDT		T4		T4
* if enabled by DMA Control Word 1				

The initial implementation simulated a DMA cycle by dispatching the required signals sequentially. For example, a normal output cycle might send IOO and then STC + CLF, and a final output cycle might send IOO, then STC + CLF, then CLC, and then EDT.

A problem arose, however, when the forthcoming 12821A Disc Interface simulator was being written. This card takes certain actions when IOO and CLF or EDT are asserted concurrently. Because the DMA signals were dispatched sequentially, detection would fail.

A review of the existing device simulators showed that two other cards also acted upon multiple signals:

Interface	Device	Condition	Action
12566B	LPS	STC + CLC	Flag does not set in diagnostic mode
12821A	DI	CLC + CLF	Master reset
12821A	DI	IOO + CLF	Inhibit setting of end-of-transfer flag
12821A	DI	IOO + EDT	Sets last-byte-out flag
12875A	IPL	IOO + EDT	Delay DMA completion interrupt for TSB

The problem of sequential signal dispatching had been worked around in the LPS and IPL simulators, but there was no easy solution to the DI issues, and it was clear that a better I/O implementation was needed.

The Second Revised I/O Implementation

To address these issues, and to accommodate future I/O card simulators more generally, a fully parallel I/O signal dispatch was implemented. Each I/O cycle, whether originated by the CPU or DMA, resulted in a single call to the device's signal handler. The signal parameter passed to the handler was replaced by a set of enumerated signals, and the DMA cycle simulator was rewritten to supply all of the signals required for a given I/O cycle concurrently. The signal handler still processed the signals sequentially, but a device simulator could now detect whether signals were issued together.

The signal handler processed the set of signals in ascending enumeration value order. The order of execution generally followed the order of T-period assertion. One complication was that the assigned T-periods for certain signals differ between CPU I/O and DMA I/O cycles:

Signal	CPU I/O Cycle	DMA I/O Cycle
IOI	T4-T5	T2-T3
STC	T4	T3
CLC	T4	T3-T4
CLF	T4	T3

The period shift allows sufficient time for SRQ assertion to steal consecutive I/O cycles from the CPU. This is not relevant to simulation, so a single signal processing order was used for both CPU and DMA cycles.

Lessons Applied to the HP 3000 Simulator

This revised implementation was introduced in version 3.9-0 and has worked well through release 28. In the interim, the author wrote and released a simulation of the HP 3000 Series III system. The 3000 I/O hardware structure, while substantially different from that of the 21xx/1000 machines, is still built around an I/O backplane that distributes signals to interface cards addressed by select codes (called *device numbers* in HP 3000 parlance). The I/O simulation was designed using the lessons detailed above.

In particular, the simulation extended the concept of signal dispatch to a close representation of the hardware. The signals sent and received over the simulated backplane are almost exactly those present on the real backplane. This was done for two reasons. First, the resulting software modules could serve as a reference to the hardware behavior. A person interested in understanding how the HP 3000 I/O hardware works but without knowledge of electronics may read the simulator code instead of the schematic diagrams. Second, the experience with HP 2100 simulator failures due to subtle hardware interactions, coupled with the author's general unfamiliarity with the HP 3000 hardware and the significantly more complex 3000 operating system, meant that following the hardware closely would lead to the best chance of operational success.

The decision has worked well — no changes to the HP 3000 I/O simulation structure have been needed from initial development through the current version.

Aligning the 3000 I/O simulation closely with the hardware provided two benefits that were not obvious at the start of the project. First, meaningful I/O bus tracing could be provided. Being able to see the CPU-to-interface interaction on a cycle-by-cycle basic proved immensely helpful in developing and debugging the various device interfaces, especially when coupled with CPU instruction tracing during I/O diagnostic execution. The detailed Theories of Operation sections in the HP hardware manuals could be applied almost directly to the simulated operations. Second, polling for interrupts after each CPU instruction would not be required; instead, the device interface simulation would assert the INTREQ signal to initiate the interrupt acknowledge cycle.

These two advantages could be applied to the HP 2100 simulator if the I/O simulators were modified to follow the hardware more closely. Beginning with release 29, this has been implemented.

Limitations of the Second Implementation

The second I/O implementation changed the simulator from dispatching I/O instructions to dispatching I/O signals. However, signals were not returned from the device interfaces to the CPU. Instead, three internal CPU variables were manipulated directly by each interface simulator. Because the CPU was unaware of the manipulations, device interrupts and DMA service requests had to be discovered by polling at various places in the instruction execution loop: after resuming from a simulator stop, after every event service call, after each DMA cycle, and after every I/O instruction execution. This was inefficient, as only a fraction of the polls executed actually resulted in an interrupt or DMA request. Moreover, I/O debug tracing added at release 27 provided an incomplete picture of the I/O backplane, as some of the input signals were missing, and no output signals were shown.

In this second model, the CPU maintained the PRL, IRQ, and SRQ values for all devices in global bit vectors. Each vector required a two-element array of unsigned 32-bit integers:

```
uint32 dev_prl [2] = { ~0, ~0 };
uint32 dev_irq [2] = {  0,  0 };
uint32 dev_srq [2] = {  0,  0 };
```

Element 0 held the bits for devices with select codes 0-31 (0-37 octal), and element 1 held the bits for devices with select codes 32-63 (40-77 octal). Within each element, the LSB corresponded to the lowest-numbered device. The initial values indicated that all devices were granting priority to lower-priority devices, and no devices were requesting interrupts or DMA service,

A device requested an interrupt by setting its bit in the IRQ vector and clearing its bit in the PRL vector. The lowest-numbered (highest-priority) request for which an unbroken priority chain existed (that is, all bits below its location were set) was granted. An IAK signal was sent to the device, which cleared its IRQ bit. This removed the interrupt source but maintained the hold-off of lower-priority requests.

Macros were provided in *hp2100_defs.h* to perform the required vector manipulations:

```
setSKF(B)   — set SKF to boolean value B
setPRL(S,B) — set PRL for select code S to boolean value B
setIRQ(S,B) — set IRQ for select code S to boolean value B
setSRQ(S,B) — set SRQ for select code S to boolean value B

setstdSKF(N) — set SKF from fields in structure N
setstdPRL(N) — set PRL from fields in structure N
setstdIRQ(N) — set IRQ from fields in structure N
setstdSRQ(N) — set SRQ from fields in structure N

PRL(S) — return boolean PRL state for select code S
IRQ(S) — return boolean IRQ state for select code S
SRQ(S) — return boolean SRQ state for select code S
```

The *setstdNNN* macros used the standard logic to set the indicated signal values, i.e.:

```
SKF = SFS * FLAG + SFC * FLAG

      _______________
PRL = CONTROL * FLAG

IRQ = CONTROL * FLAGBUF * FLAG

SRQ = FLAG
```

These macros typically were used in the signal handler routine's **switch** statement that dispatched the signals received from the CPU:

```
switch (signal) {
   case ioSFC:
      setstdSKF (dev);
      break;

   case ioSFS:
      setstdSKF (dev);
      break;
```

```
case ioSIR:
   setstdPRL (dev);
   setstdIRQ (dev);
   setstdSRQ (dev);
   break;

[ additional signal handlers... ]

)
```

The macros worked as intended, but they hid the underlying actions from view. From reading the code, for instance, one could not tell that *setstdIRQ* requested an interrupt if the Control, Flag Buffer, and Flag flip-flops were all set. The macros also imposed restrictions on variable names; in this case, *setstdIRQ* required that the DIB pointer be named *dibptr*, that *dev* was a structure that contained fields named *control*, *flag*, and *flagbuf*, and that the CPU module exported the bit vectors named *dev_prl*, *dev_irq*, and *dev_srq*. This made the operation of the signal handler unnecessarily opaque.

After a device interface requested an interrupt by invoking *setstdIRQ* to set the appropriate bit in the *dev_irq* vector, the CPU had to poll to recognize the interrupt. This was because the priorities of the various interrupt requests were resolved at poll time, rather than at interrupt request time. The *calc_int* routine in the CPU module evaluated all pending interrupt requests and selected the highest-priority request for action.

Unfortunately, for the majority of the *calc_int* routine calls, no interrupt was pending (consider a disc read requiring thousands of DMA cycles to transfer data but only one interrupt at command completion).

From the 3000 work, it was clear that if priority was communicated to the signal handler via the unused PRH signal, then an interrupting device could determine unequivocally that its request would be honored, and so polling could be avoided.

The Third I/O Implementation

Beginning with release 29, the I/O system simulation has been rewritten to more closely model the actual hardware. This has no user-visible impact, except that IOBUS tracing now accurately reflects the hardware I/O bus signals. For example, the PRH and PRL signals now appear in IOBUS traces and show the I/O priority settings. Any user-written device interfaces will have to be changed to use the new I/O structure; the Microcircuit Interface in the *hp2100_mc.c* source file is a simple example that may be used as a model for the changes needed.

The new structure better segregates CPU and device operations. The CPU module exports only two I/O routines for use by the device simulators. The *io_assert_ENF* routine is provided to assert the ENF and SIR signals (periodic in hardware but asserted here when the Flag Buffer, Flag, or Control flip-flop is changed). The *io_assert_SIR* routine is provided to assert the periodic SIR signal when the device asserts the SRQ signal independently of the FLG signal. All other communication is via calls to the device interface routine that models the control logic hardware.

I/O System Declarations

The declarations needed for the device simulators are now contained in the new *hp2100_io.h* source file.

A simulator for a given device interface defines a DIB structure:

```
typedef struct dib DIB;

struct dib {
   IO_INTERFACE    *io_interface;
   IO_INITIALIZER  *io_initializer;
   IO_POWER        *io_power;
   uint32          select_code;
   uint32          card_index;
   const char      *card_description;
   const char      *rom_description;
   };
```

...and places a pointer to it in the *ctxt* field of the associated *DEVICE* structure. The DIB contains a pointer to the interface routine that will interact with the I/O backplane, a pointer to an optional routine that handles execution-time initialization, a pointer to an optional routine that handles power supply state changes, the select code of the interface, a card index associated with the device, a description of the card that is printed with the ***SHOW CPU IOCAGE*** command, and, if an HP 1000 boot loader ROM is provided for the device, a description of the ROM that is printed with the ***SHOW CPU ROMS*** command. For interface routines that serve only one I/O card, the card index value is set to 0. If several I/O cards are served, the card index values of the corresponding DIBs are set to 0, 1, 2, etc.

In general, there is a one-to-one correspondence between a hardware I/O interface card and a *DEVICE* structure. A good example is a two-card disc interface, such as the 13210A Disc Controller supporting one to four HP 7900 disc drives. The associated DP simulator defines separate *DEVICE* structures (DPD and DPC) for the data and command cards, as well as separate interface routines. This is appropriate, as the cards are of different designs and respond differently to the I/O signals.

In some cases, though, a peripheral may use two cards with identical or nearly identical behaviors. In such cases, duplicating the interface routines is unnecessary, is more difficult to maintain, and may result in significant code bloat if the card operation is complex.

For example, the 12897B Dual Channel Port Controller consists of two channels that are identical except for priority. The 12875A Processor Interconnect Kit uses two identical 12566B Microcircuit Interface cards — one used as an input device, and the other used as an output device. In both simulators, a common interface routine (or routines, in the case of the DCPC, which has primary and secondary select codes for each channel) is employed, and the cards' state variables, such as the Control and Flag flip-flops, are kept in arrays indexed by the DIB card index. For the DCPC, the card index corresponds to the last bit of the select code addressed (2 or 6 $\rightarrow$ 0, 3 or 7 $\rightarrow$ 1). For the PIK, the card index corresponds to the communication direction (input card $\rightarrow$ 0, output card $\rightarrow$ 1).

The I/O backplane signals are defined as disjoint enumeration values, so that sets of signals sent to and received from the interface may be defined. The declarations are of this form:

```
typedef enum {
   ioPON   = 000000000001,
   ioIOI   = 000000000002,
   ioIOO   = 000000000004,
   ioSFS   = 000000000010,

   [ additional signals... ]

   } INBOUND_SIGNAL;

typedef enum {
   ioSKF   = 000000000001,
   ioPRL   = 000000000002,
   ioFLG   = 000000000004,
   ioIRQ   = 000000000010,
   ioSRQ   = 000000000020,

   cnIRQ   = 000000000040,
   cnPRL   = 000000000100,
   cnVALID = 000000000200
   } OUTBOUND_SIGNAL;

typedef INBOUND_SIGNAL  INBOUND_SET;

typedef OUTBOUND_SIGNAL OUTBOUND_SET;
```

The *INBOUND_SIGNAL* and *OUTBOUND_SIGNAL* declarations mirror the hardware signals that are received and asserted, respectively, by the interfaces on the I/O backplane. A set of one or more signals forms an *INBOUND_SET* or *OUTBOUND_SET* that is sent to or returned from the device interface. Under simulation, the CPU and DMA dispatch one *INBOUND_SET* to the target device interface per I/O cycle. The interface returns an *OUTBOUND_SET* and a data value combined into a *SIGNALS_VALUE* structure to the caller.

Three outbound pseudo-signals are not derived from the hardware. The *conditional IRQ* signal, *cnIRQ*, is asserted by the interface routine under the same conditions as the *ioIRQ* signal, except that it is not conditional on PRH and IEN. That is:

```
ioIRQ = CONTROL * FLAG * FLAG_BUFFER * SIR * IEN * PRH

cnIRQ = CONTROL * FLAG * FLAG_BUFFER * SIR
```

Similarly, the *conditional PRL* signal, *cnPRL*, is the same as *ioPRL*, except that it is not conditional on either PRH or IEN:

$$ioPRL = \overline{CONTROL * FLAG * IEN * PRH}$$

$$cnPRL = \overline{CONTROL * FLAG}$$

The *conditionals valid* signal, *cnVALID*, is asserted if *cnIRQ* and *cnPRL* have been evaluated. Typically, this occurs in response to SIR, but certain interface designs may require evaluation in response to some other signal.

These pseudo-signals permit the CPU to avoid polling all lower-priority interfaces when PRL changes. A later section in this paper describing hardware deviations explains their use.

The previous *IOHANDLER* routine has been redefined as the *IO_INTERFACE* routine and is now declared as:

```
typedef SIGNALS_VALUE IO_INTERFACE
   (const DIB    *dibptr,
    INBOUND_SET  inbound_signals,
    HP_WORD      inbound_value);
```

The DIB pointer points at the DIB associated with the interface, providing the select code and card index of the interface, if needed. The set of inbound signals contains the enumeration values of the signals asserted for this I/O cycle, and the inbound value is a 16-bit data value supplied to the interface by the CPU (representing the I/O Bus Output data lines). The return value is this structure:

```
typedef struct {
   OUTBOUND_SET  signals;
   HP_WORD       value;
   } SIGNALS_VALUE;
```

The outbound signal set contains the enumeration values of the signals asserted by the interface, and the outbound value is a 16-bit data value supplied by the interface to the CPU (representing the I/O Bus Input data lines). The inbound and outbound data values are valid only when the IOO and IOI bus signals, respectively, are present in the inbound signal set.

The state variables that represent the Command, Control, Flag, and Flag Buffer flip-flops used in the interface routine, along with the Input Data Register and Output Data Register (if applicable) are typically declared in a structure:

```
typedef struct {
   HP_WORD      output_word;          /* output data register */
   HP_WORD      input_word;           /* input data register */
   FLIP_FLOP    command;              /* command flip-flop */
   FLIP_FLOP    control;              /* control flip-flop */
   FLIP_FLOP    flag;                 /* flag flip-flop */
   FLIP_FLOP    flag_buffer;          /* flag buffer flip-flop */
   } CARD_STATE;

static CARD_STATE state;              /* per-card state */
```

No Interrupt Request flip-flop is needed. In hardware, it serves only to synchronize the IRQ and FLG signals with the SIR signal; in simulation, IRQ and FLG are asserted by the interface routine only when SIR is received.

The flip-flops are assigned values using the enumeration constants *CLEAR* and *SET*. Any additional per-card state variables should be declared in the structure as well.

If an interface routine is to serve multiple cards, an array of structures may be used:

```
static CARD_STATE state [2];          /* state for two cards */
```

...and state variables may be accessed as:

```
state [card].flag_buffer = SET;
```

...where *card* is the current card number (identified by *dibptr→card_index*).

The Interface Routine

The interface routine dispatches the inbound signal set as follows:

```
SIGNALS_VALUE dev_interface (const DIB *dibptr, INBOUND_SET inbound_signals,
   HP_WORD inbound_value)
{
INBOUND_SIGNAL signal;
INBOUND_SET    working_set = inbound_signals;
SIGNALS_VALUE  outbound    = { ioNONE, 0 };
t_bool         irq_enabled = FALSE;

while (working_set) {
   signal = IONEXTSIG (working_set);

   switch (signal) {

      case ioCLF:
         ...
         break;

      case ioSTF:
         ...
         break;

      [ additional signal handlers... ]

      )

   IOCLEARSIG (working_set, signal);
   }

return outbound;
}
```

The *IONEXTSIG* macro isolates the next signal in the execution order to be processed. After that signal is processed, the *IOCLEARSIG* macro removes it from the working set, and signal dispatching continues until the set is exhausted. The original inbound signal set remains available to enable detection of concurrent signal assertions, if needed.

For example, executing an STC 10B,C instruction calls the interface routine with an *inbound_signals* value of *ioSTC* | *ioCLF* | *ioSIR*. The *IONEXTSIG* macro extracts the *ioSTC*, *ioCLF*, and *ioSIR* signals in sequence. Finally, a combined outbound signals set and data value are returned to the caller.

A simulator for an interface card with the standard flip-flop logic and data registers employs the following common signal handlers within the **switch** statement. For the flag logic, the actions are derived directly from the hardware logic:

```
case ioCLF:
   state.flag_buffer = CLEAR;
   state.flag        = CLEAR;
   break;

case ioSTF:
   state.flag_buffer = SET;
   break;

case ioENF:
   if (state.flag_buffer == SET)
      state.flag = SET;
   break;

case ioSFC:
   if (state.flag == CLEAR)
      outbound.signals |= ioSKF;
   break;

case ioSFS:
   if (state.flag == SET)
      outbound.signals |= ioSKF;
   break;
```

The *ioSTF* signal is always accompanied by an *ioENF* signal, so the handler for the former only sets the flag buffer variable.

For input and output:

```
case ioIOI:
   outbound.value = state.input_word;
   break;

case ioIOO:
   state.output_word = inbound_value;
   break;
```

The *input_word* and *output_word* typically are set or used, respectively, by other device operations. For example, the input value might be set by a “request status” operation. The output value might be used by the device event routine to write the values to a simulated paper tape. The outbound data value is significant only for the IOI signal, where the value will be stored in a register or memory. For all other signals, the returned data value is ignored.

For the control logic:

```
case ioPOPIO:
   state.flag_buffer = SET;
   state.output_word = 0;
   break;

case ioCRS:
   state.control = CLEAR;
   state.command = CLEAR;
   break;
```

```
case ioCLC:
   state.control = CLEAR;
   state.command = CLEAR;
   break;

case ioSTC:
   state.control = SET;
   state.command = SET;
   break;
```

Application of main power asserts the hardware signals POPIO and CRS. PON, POPIO, and CRS are asserted when the front-panel PRESET button is pressed. Most cards do not respond to PON, so the corresponding handler is usually omitted; however, its presence or absence can be used to distinguish between a power-on preset and an operator preset. The CLC 0 instruction generates CRS only. The CRS action may or may not be the same as the CLC action, depending on the specific interface design. All three signals are asserted to all interface cards in the system.

Asserting STC typically starts the device operation. The *ioSTC* handler usually schedules the event service to represent the hardware I/O delay by calling the *sim_activate* routine.

For the interrupt logic, the actions again are derived directly from the hardware logic:

```
case ioSIR:
   if (state.control & state.flag)
      outbound.signals |= cnVALID;
   else
      outbound.signals |= cnPRL | cnVALID;

   if (state.control & state.flag & state.flag_buffer)
      outbound.signals |= cnIRQ;

   if (state.flag == SET)
      outbound.signals |= ioSRQ;
   break;

case ioIAK:
   state.flag_buffer = CLEAR;
   break;

case ioIEN:
   irq_enabled = TRUE;
   break;

case ioPRH:
   if (irq_enabled && outbound.signals & cnIRQ)
      outbound.signals |= ioIRQ | ioFLG;

   if (!irq_enabled || outbound.signals & cnPRL)
      outbound.signals |= ioPRL;
   break;
```

In hardware, SIR is asserted during every T5 I/O period; it is used to qualify the setting of the Interrupt Request flip-flop. In simulation, SIR is asserted in any I/O cycle that may affect the flip-flop.

The CPU sends the IAK signal when the device's interrupt request is granted. It clears the Flag Buffer flip-flop, which inhibits generation of the IRQ signal while continuing to hold off lower priority interrupts by denying PRL.

The CPU asserts IEN whenever the interrupt system is on. Its function is to qualify generation of the IRQ and FLG signals when an interrupt condition is pending, and no higher-priority interrupt is active.

Finally, signals that are not used by the specific interface routine should have empty handlers. For example:

```
case ioEDT:                              /* not used by this interface */
   break;
```

This is preferable to using a **default** label, as an omitted handler typically will be flagged by the compiler.

The Event Service Routine

A device simulator generally clears its Command flip-flop and sets its Flag Buffer flip-flop in response to operation completion, typically in the event service routine. This corresponds in hardware to asserting the DEVICE FLAG signal. After setting the Flag Buffer, the service routine must assert ENF to set the Flag flip-flop and enable interrupt and DMA service request generation:

```
state.command     = CLEAR;
state.flag_buffer = SET;
io_assert_ENF (&device_dib);
```

A few devices set their Flag Buffer flip-flops outside of the interface or event service routines. An example is a disc drive subsystem that requests an interrupt when the heads load and the drive becomes ready. This is simulated by a ***SET <unit> LOAD*** command, issued by the user, that is processed in an *MTAB* (modification table) validation routine. Because the command is issued when the simulator is stopped, the routine need only set the Flag Buffer, as the ENF and SIR signals are automatically asserted to all device interfaces prior to execution resumption.

The standard method of driving the SRQ signal from the Flag flip-flop output allows DMA to run at only half of the rated bandwidth, i.e., to steal only every other I/O cycle from the CPU. Some interfaces drive SRQ separately from its own flip-flop. Such interfaces are simulated by adding an *srq* variable of type *FLIP_FLOP* to the state variable structure and modifying the *ioSIR* handler to use the SRQ flip-flop instead of the Flag flip-flop:

```
case ioSIR:
   [...]

   if (state.srq == SET)
      outbound.signals |= ioSRQ;
   break;
```

If the event service routine sets the SRQ flip-flop, it must then assert SIR to generate SRQ and initiate another DMA cycle:

```
state.srq = SET;
io_assert_SIR (&device_dib);
```

The Execution-Time Initializer Routine

If a device initializer routine is defined, it is called just before CPU instruction execution begins. The initializer may be used to set up the device state prior to receiving calls through the interface routine. A typical use is a device simulator that must perform configuration that depends on other devices, such as the CPU type. For example, the event service delay may be set to one value if the CPU is configured as a 2100 and a different value if the CPU is configured as a 1000. Such detection cannot reliably occur in the device reset routine, as there is no guarantee that the user will perform a RESET after changing the CPU type. Another potential use is to check for conflicts among unit assignments, for instance that two units have not been assigned to the same bus address by the user.

The initializer returns TRUE if the device initialization is successful and FALSE otherwise. Returning FALSE prevents CPU execution and reports *Simulation stopped*. The initializer should report the cause of the failure to the simulation console before returning FALSE.

The Power State Routine

If a power state routine is defined, it is called in response to a ***POWER*** command that affects the device, as follows:

```
t_stat dev_power (UNIT *uptr, POWER_STATE new_state)
{
t_stat status = SCPE_OK;

switch (new_state) {
   case power_failing:
      ...
      break;

   case power_off:
      ...
      break;

   case power_restoring:
      ...
      break;

   case power_on:
      ...
      break;
   }

return status;
}
```

The unit pointer *uptr* will be NULL; it is provided for possible future expansion to controlling power on individual peripheral units, such as disc drives connected to a common disc controller. The *new_state* will be one of four *POWER_STATE* values, depending on the specific command issued, as follows:

Command	State Value
POWER FAIL	power_failing
POWER OFF	power_off
POWER RESTORE	power_restoring
POWER ON	power_on

The routine takes whatever actions are appropriate for the various power supply states. Unless the device performs some specific action in response to power failing or restoring, the handler can combine the *power_failing* and *power_off* cases into a single power-off action (and similarly with the power-on action). The handler will be called only once per issued command; it is up to the handler to schedule a second call if separate actions are taken during power transitions.

The Reset Routine

Each device may define a *reset* routine that is called in five cases:

- when the simulator is initially started
- in response to a ***POWER RESTORE*** or ***POWER ON*** command, if the device does not provide a separate power state handler
- in response to a ***RESET***, ***RESET ALL***, or ***RESET <device>*** command
- in response to a ***BOOT*** or ***RUN*** command before simulated execution begins
- in response to a ***SET <device> DISABLED*** or ***SET <device> ENABLED*** command.

The first two cases automatically set the *P* value of the *sim_switches* global variable before calling the reset routine; for the third case, ***–P*** may be specified explicitly by the user.

The behavior of the routine is different, depending on whether the CPU or a peripheral device is being reset.

Resetting the CPU performs either a power-on preset (if the *P* switch is set) or an operator preset (if the *P* switch is clear). Both assert the POPIO and CRS signals to all device interfaces as described earlier; an operator preset additionally asserts the PON signal. These actions correspond in hardware to applying power and pressing the front-panel PRESET button, respectively.

Resetting a device performs the same two actions. However, while all devices perform some sort of power-up initialization, few have a separate method of allowing the operator to reset the device. In such cases, the difference between a power-up reset and an operator reset is arbitrary.

Except where explicitly provided by hardware, device reset routines do not alter the state of their associated interface cards, which are initialized by resetting the CPU. Instead, they initialize only the connected peripherals.

Deviations from the Hardware to Improve Performance

The current design follows the hardware logic closely. Only three deviations are employed to improve performance:

- The interface does not decode the select code address.

In hardware, all control signals asserted by the CPU except PRH are common to all I/O slots. Each interface qualifies its control signals with three select-code-addressing signals: IOG (I/O Group enable), LSCM (Lower Select Code Most-significant digit), and LSCL (Lower Select Code Least-significant digit). IOG asserts whenever an I/O Group instruction or DMA cycle is executing. LSCM and LSCL assert when a select code address corresponds to the slot in which the interface is installed. For example, the LSCM signal for slot 13 asserts for any I/O instruction addressing select codes 10-17, while the LSCL signal asserts for any instruction addressing select codes ending in 3 (13, 23, 33, etc.). These three signals are ANDed together and used to enable all other control signals on the interface.

In simulation, the CPU's *io_dispatch* routine calls only the interface routine corresponding to the addressed select code. A direct hardware simulation would call every interface routine for every I/O instruction, even though only one of the routines would respond for each set of calls.

- The ENF and SIR signals are not periodic, and PON is not continuous.

The hardware ENF and SIR signals assert for periods T2 and T5, respectively, of every I/O cycle. I/O cycles run continuously, even when I/O operations do not take place, i.e., when IOG is not asserted. A device asserting DEVICE FLAG will set the Flag Buffer flip-flop on the associated interface asynchronously. The Flag flip-flop will set at T2 of the next cycle, and then, if conditions permit, the Interrupt Request flip-flop will set at T5, generating a CPU interrupt.

A direct simulation of this hardware behavior would call all interface routines after every CPU instruction to assert the ENF and SIR signals that only a device asserting DEVICE FLAG would use. To avoid this, interface simulations must call *io_assert_ENF* or *io_assert_SIR* to request ENF or SIR assertion explicitly after setting their Flag Buffer or SRQ flip-flops.

In hardware, PON is asserted continuously while power is applied to the machine. A direct simulation would add the PON signal to every interface routine call, which would require execution of the *ioPON* signal handler. PON is only useful when differentiating a power-on preset from an operator preset. To avoid this inefficiency, PON is asserted only for an operator preset.

- The interface asserts the pseudo-signals *cnIRQ*, *cnPRL*, and *cnVALID*.

In hardware, the IEN and PRH signals are combinatorial and propagated via PRL, so changing either potentially affects all interfaces in the system. For an interface to generate an interrupt, PRH must be asserted by the next-higher-priority interface. When an interface

generates an interrupt, it denies PRL to the next-lower-priority interface; otherwise, it passes PRH to PRL unaltered. Therefore, a given interface can interrupt only if all higher-priority devices are receiving PRH asserted and are asserting PRL. Clearing that interrupt and reasserting PRL may affect all lower-priority interfaces.

Consider a situation where the interface at select code 10 ("SC 10") has its Flag Buffer, Flag, and Control flip-flops set, IEN and PRH are asserted, and no other interfaces have their Flag flip-flops set. SC 10 will assert IRQ and deny PRL. SC 11 sees its PRH low (because PRL 10 is connected to PRH 11) and so denies its PRL, and this action ripples through all of the lower-priority interfaces. Then, while the interrupt for SC 10 is being serviced, the interface at select code 17 sets its Flag Buffer, Flag, and Control flip-flops. SC 17 is inhibited from interrupting by its PRH being denied.

The interrupt service routine for SC 10 completes by clearing the Flag flip-flop, reasserting PRL to SC 11. That signal ripples through the interfaces at select codes 12-16, arriving at SC 17 as PRH assertion. With PRH asserted, SC 17 generates an interrupt and denies PRL to SC 20 and above.

A direct simulation of this hardware behavior would require calling all lower-priority interface simulators in ascending select code (and therefore decreasing priority) order with the PRH signal and checking for an asserted IRQ signal or a denied PRL signal. This is grossly inefficient.

The difficulty is that when an interface reasserts its PRL signal after servicing an interrupt, the system does not know if a lower-priority interface wants to interrupt. When the priority chain is broken, all lower-priority interfaces will deny PRL and IRQ, even though one or more may otherwise be ready to interrupt. Without propagating the newly reasserted PRL by calling all affected interface routines in descending priority order, the simulator cannot determine if another interrupt is pending.

To avoid making this potentially long sequence of calls, each interface routine returns *conditional* IRQ and PRL signals in addition to the standard IRQ and PRL signals. The conditional signals are those that would result if PRH is asserted and the interrupt system is on. So an interface simulator with its Flag Buffer, Flag, and Control flip-flops set will assert *cnIRQ* and deny *cnPRL*. If PRH and IEN are asserted, then the standard IRQ and PRL signals will also be asserted and denied respectively.

After every interface routine call, the CPU's *io_dispatch* routine saves the returned conditional signals in two bit vectors. Each vector is represented as a two-element array of 32-bit unsigned integers:

```
static uint32 interrupt_request_set [2] = { 0, 0 };
static uint32 priority_holdoff_set  [2] = { 0, 0 };
```

...forming a 64-bit vector in which bits 0-31 of the first element correspond to select codes 00-37 octal, and bits 0-31 of the second element correspond to select codes 40-77 octal. The *interrupt_request_set* array holds conditional IRQ states, and the *priority_holdoff_set* array holds the complement of the conditional PRL states (the complement is used to simplify the priority calculation). Within each element, the LSB corresponds to the lowest

select code. When the CPU is called via *sim_instr* to begin executing instructions, it asserts ENF and SIR to every enabled device. Each device returns the values of its conditional IRQ and PRL signals, which are set into the vectors.

When the *io_dispatch* routine sees that a returned PRL signal has reasserted, it checks the bit vectors to see if a lower-priority interface is now enabled to interrupt. If so, an interrupt-acknowledge cycle is performed.

An interface will assert *cnVALID* if the conditional PRL and IRQ were determined. If *cnVALID* is not asserted by the interface, then the states of the *cnPRL* and *cnIRQ* signals cannot be inferred from their absence in the outbound signal set. The *cnVALID* signal is required because although most interfaces determine the PRL and IRQ states in response to SIR assertion, not all do. In particular, the 12936A Privileged Interrupt Fence determines PRL in response to an IOO signal.

Summary

The original I/O simulation design was based on devices handling I/O instructions from the execution stream. This was a good match to the typical HP interface card, as embodied in the set of devices provided with the initial release of the HP simulator. However, as more complex and higher-performance interfaces were added, special cases had to be included to allow for atypical behavior. Eventually, reimplementation based on a model of I/O backplane signals became attractive to alleviate restrictions of the original design. This new model also removed the special cases and allowed for easier future expansion of the simulated device repertoire.

The advantages of closely following the hardware design were demonstrated clearly in the development of the HP 3000 simulator. Although the author had a very limited understanding of, and no practical experience with, the 3000 I/O structure, and although the 3000 operating system is an order of magnitude more complex than those for the 1000, the I/O simulation required only minimal debugging and has worked flawlessly since.

Close alignment with the hardware also allows the simulator to serve as a model to understanding the original design, as well as to allow debugging to rely on the theories of operation present in the HP hardware manuals.