

# A Revised I/O Simulation for the HP 21xx/1000

J. David Bryan, 14-Nov-2008

The HP 2100 simulator for the 21xx and 1000 series of machines originally modeled I/O interface communication with the CPU by dispatching I/O instructions to the interfaces for action. A revised model, based on dispatching I/O backplane signals, has been implemented to solve several problems inherent in the original design.

## The HP I/O Hardware Structure

The structure of the I/O system is compatible across all HP 21xx/1000 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 are interchangeable, and an interface derives its I/O address from the slot into which it is installed. Lower-numbered slots have interrupt priority over higher-numbered ones.

An I/O timing cycle is divided into five periods, designated T2 through T6. On the early 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 various T-periods to control the timing of the interfaces.

The basic structure of a typical HP interface consists of a control and a flag flip-flop. A programmed "set control" instruction asserts the STC signal on the I/O backplane to set the control flip-flop and initiate operation on the I/O device. When an operation completes, the device sets the flag flip-flop, which asserts the FLG signal. The state of the flag can be tested under program control. The CPU asserts either the SFS or SFC signal to test whether the flag flip-flop is set or clear, and the interface then asserts 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.

Because device completion occurs asynchronously with I/O timing, a flag buffer flip-flop is inserted before the flag flip-flop. The flag buffer is set asynchronously by the device, and then the flag is set during the appropriate CPU T period. If DMA is employed, the flag also requests a DMA cycle by asserting SRQ. If the device requires a per-operation start signal, then a command flip-flop is added that is set under program control and cleared by device completion, although this flip-flop is not involved in the backplane interface.

The combination of flag buffer, flag, and control enables generation of an interrupt request signal, IRQ, presuming no higher-priority device is interrupting. Priority is

established through a chain of signals (PRH and PRL) passing from higher-priority interfaces to lower-priority interfaces; the chain is broken by the first interface asserting flag and control. Interrupt acknowledgement asserts IAK, which clears the flag buffer, leaving the flag and control set to continue to hold off lower-priority devices until the interrupt service routine is complete.

A power-on signal (PON) is provided to indicate that the power supply voltages have stabilized after initial turn-on. A front-panel PRESET button sends a signal (POPIO) to allow cards to be reset to known conditions in preparation for running programs. PRESET or a programmed reset via execution of a CLC 0 instruction will assert the control reset (CRS) signal. A typical interface might use PON to enable output drivers to the device, set the flag buffer and flag in response to POPIO, and clear the control flip-flop in response to CRS. The combination of POPIO and CRS turns off the I/O device and places the interface in an idle condition.

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, a few interfaces depart from the standard implementation, either to improve I/O transfer speed, or to meet a particular device control requirement.

## **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 function simulated power-on and PRESET. The DEVICE structure's context value pointed to a device information block (DIB) that contained the interface slot number (select code), values for 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 the 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 simulator 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 execution started and stopped.

## Problems with the Implementation

This implementation worked well with the initial 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 12606 and 12610 fixed disc interfaces added at version 2.9 were not tied into the interrupt request logic, so setting control, flag, and flag buffer did not generate an interrupt. Because interrupt generation was calculated in the CPU simulator, the interface simulator had to use the command flip-flop as the control flip-flop to avoid generating interrupt requests inappropriately.

At version 3.2, CPU interrupt acknowledgement handling added a special case for the 12581 and 12892 memory-protect cards, which cleared both flag and flag buffer in response to IAK. Also at 3.2, the DMA service requests were separated from the flags, as the forthcoming 7905 disc simulator required separate control over these two values, and a new SRQ flip-flop value was added to the DIBs. The new disc 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 device reset functions to differentiate between power-up reset and ordinary reset. The 12578 and 12895 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 devices respond to CRS as CLC, i.e., by clearing the control flip-flop. However, not all do. In particular, the DMA card clears control in response to CLC but control and command in response to CRS. Clearing command stops an in-progress DMA operation, which the original implementation failed to do.

## Impasse

For version 3.8-1, a simulation of the 12936 and 12620 Privileged Interrupt Fences was planned. These devices are required to run the 12920 terminal multiplexer under the DOS and RTE operating systems. These systems run with the interrupt system off when servicing any device, as they are not reentrant. The 12920 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 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 service handler.

The 12936 has a unique behavior. Setting either control or flag denies priority. An interrupt occurs when flag and flag buffer are set and control is clear. The flag and flag buffer are cleared with the CLF instruction but set with the OTA/B instruction. This presented a problem because of the implicit assumptions of the roles of control, flag, and flag buffer 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:

$$\text{CONTROL}' = \text{CONTROL} + \text{FLAG} * \text{FLAGBUF}$$

$$\text{FLAG}' = 1$$

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

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 upon simulation execution.

The choices, then, were to accept that the state display would be wrong and to disallow user changes to the flip-flop values, to add another special-case to the CPU simulator to accommodate the atypical I/O behavior, or to remodel the I/O simulator 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 simulations increased, implementation of an I/O model closer to the actual hardware was selected.

## A Revised I/O Implementation

Whereas the old model was based on dispatching I/O instructions, the new model is based on dispatching I/O backplane signals. This allows the interface to take whatever action it wants in response. Instead of examining the control, flag, and flag buffer flip-flop values, the CPU monitors these signals from the interface simulators:

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

PRL indicates that interrupt priority to lower-priority devices should be granted. IRQ is set when an interface wants to interrupt the CPU. SRQ is set to initiate a DMA cycle. SKF indicates that the programmed flag test is true and that the next instruction should be skipped.

The interface simulators monitor reception of these signals and dispatch accordingly:

- CLC — clear the control flip-flop
- STC — set the control flip-flop
- CLF — clear the flag flip-flop
- STF — set the flag flip-flop
- SFC — skip if the flag is clear
- SFS — skip if the flag is set
- IOI — I/O data input
- IOO — I/O data output
- ENF — enable flag
- EDT — end of data transfer
- SIR — set interrupt request
- IAK — interrupt acknowledge
- CRS — control reset
- POPIO — power-on preset to I/O

The first eight signals are generated as a result of I/O instructions: the LIA/B and MIA/B instructions generate IOI, the OTA/B instructions generate IOO, and the remaining instructions generate their namesake signals. ENF sets the flag buffer and flag flip-flops. EDT occurs at the end of a DMA transfer. SIR asks the interface to calculate and set the IRQ, PRL, and SRQ values. The CPU sends IAK to acknowledge an interrupt. CRS and POPIO have been discussed previously.

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

- a more consistent structure (only signals are handled, rather than a mixture of signals and I/O instructions)
- elimination of special-cases in the CPU simulator (each interface simulator determines 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 examines flip-flop values, and the bit vectors are set at execution start by sending SIR to all devices; no action is needed at execution stop)
- unified handling of flip-flop values (values exist in one place—as local variables—rather than in the DIB and in the bit vectors, reducing coding error potential)

- simplification of CPU interrupt determination (only IRQ and PRL vectors need to be examined)
- simpler handling of power-on and preset (the simulation reset function simply dispatches POPIO to the signal handler; no duplication of the initialization code)

For efficiency, the simulator does not implement signal generation exactly as in the hardware. Many hardware signals are periodic (e.g., IOI, ENF, SIR) and most signals are common to all interfaces and are qualified by the slot select code on the interface. Under simulation, signals are sent only when action is to be taken and then only to the specific target device. For example, SIR is dispatched only when flip-flops affecting the PRL, IRQ, or SRQ signals are changed, rather than after every instruction. ENF is sent only when the device indicates that the flag buffer and flag are to be set, whereas in hardware, the device asynchronously sets the flag buffer, and then ENF samples the flag buffer at every T2 and sets the flag accordingly. In response to PRESET, POPIO is sent instead of, rather than in addition to, CRS.

## I/O Device Simulator Structure Details

The new I/O implementation requires changes in the CPU simulator and in each device simulator.

The CPU maintains the PRL, IRQ, and SRQ values for all devices in global bit vectors. Each signal requires 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 holds the bits for devices 0-31 (0-37 octal), and element 1 holds the bits for devices 32-63 (40-77 octal). Within each element, the LSB corresponds to the lowest-numbered device. The initial values indicate that all devices are granting priority to lower-priority devices, and no device is requesting an interrupt or DMA service,

A device requests an interrupt by setting its bit in the IRQ array and clearing its bit in the PRL array. The lowest-numbered (highest-priority) request for which an unbroken priority chain exists (all bits below its location are set) is granted. An IAK signal is sent to the device, which must clear its IRQ bit. This removes the interrupt source but maintains a hold-off of lower-priority requests.

When the CPU is called via *sim\_instr()* to begin executing instructions, it first clears the IRQ and SRQ arrays and sets the PRL array. It then sends an SIR signal to every enabled device. In response, each device calculates the values of its IRQ, PRL, and SRQ responses and sets them into the arrays.

The CPU implements the SKF signal as the return value from the device I/O signal dispatcher when it is called to process the SFS and SFC signals.

For clarity, flip-flop variables should be declared as enumeration type *FLIP\_FLOP*:

```
FLIP_FLOP dev_control;  
FLIP_FLOP dev_flag;  
FLIP_FLOP dev_flagbuf;
```

...and assigned values using the enumeration constants *CLEAR* and *SET*. The standard flip-flops for a device should use the device name *dev* as a prefix (e.g., *ptr\_control*, *tty\_control*, etc.) Additional device flip-flops may be declared as desired, even if they do not participate in interrupt requests.

The following signal macros are provided in *hp2100\_defs.h* to aid implementation:

```
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 for variable-name prefix N  
setstdPRL(S,N) - set PRL for select code S and variable-name prefix N  
setstdIRQ(S,N) - set IRQ for select code S and variable-name prefix N  
setstdSRQ(S,N) - set SRQ for select code S and variable-name prefix N  
  
PRL(S) - return PRL state for select code S  
IRQ(S) - return IRQ state for select code S  
SRQ(S) - return SRQ state for select code S
```

The *setstdNNN* macros use the standard logic to set the indicated signal values. That is:

```
SKF = SFS * FLAG + SFC * FLAG  
PRL = CONTROL * FLAG  
IRQ = CONTROL * FLAG * FLAGBUF  
SRQ = FLAG
```

The variable-name prefix is the string that indicates the names of the variables to set. For example:

```
setstdSRQ (sc, dev);
```

...will set the SRQ array bit for select code *sc* to the value of variable *dev\_flag*. The macros assume that the flip-flop variables are named (e.g.) *dev\_control*, *dev\_flag*, and *dev\_flagbuf*.

If the standard logic is not applicable, a simulator may use the *setNNN* macros to set the indicated signals as desired.

Each device simulator defines a DIB and places a pointer to it in the *ctxt* field of the associated DEVICE structure. The DIB structure consists of the device select code and a pointer to the I/O signal dispatcher. The simulator must declare the dispatcher as follows:

```
uint32 dev_io (uint32 sc, IOSIG signal, uint32 data);
```

The CPU will call the signal dispatcher and pass the device's select code, the I/O signal, and either the value of the A or B register (for signal IOO) or zero. The function dispatches the signal as follows:

```
const IOSIG base_signal = IOBASE (signal);

switch (base_signal) {
    case ioCLF:
        ...
        break;

    case ioSTF:
        ...
        break;

    [ additional signal handlers... ]

    default:
        ...
        break;
}

if (signal > ioCLF)
    dev_io (sc, ioCLF, 0);
else if (signal > ioSIR)
    dev_io (sc, ioSIR, 0);

return data;
```

In response to an I/O instruction, the CPU will send a single signal or a signal plus the CLF signal. For instance, executing an STC 10,C instruction will send *ioSTC* + *ioCLF*, whereas a STC 10 instruction will send *ioSTC* alone. The *IOBASE* macro extracts the original (base) signal. As all signals have non-zero values, and *ioCLF* has the highest value, a signal value greater than *ioCLF* indicates that base signal execution should be followed by CLF execution. The arrangement of signal values also has all those that affect interrupt requests or priority assigned to values greater than that of the *ioSIR* signal. If such a signal has been executed, it is followed by an SIR execution to update the IRQ, PRL, and SRQ signals. Finally, a combined status and data value is returned to the caller.

A simulator for a device interface with the standard flip-flop logic and I/O buffers will employ the following common signal handlers within the switch statement. For the flag logic:

```

case ioCLF:
    dev_flag = dev_flagbuf = CLEAR;
    break;

case ioSTF:
case ioENF:
    dev_flag = dev_flagbuf = SET;
    break;

case ioSFC:
    setstdSKF (dev);
    break;

case ioSFS:
    setstdSKF (dev);
    break;

```

Interface responses to the ENF and STF signals are usually identical, and therefore the *ioENF* handler may simply fall into the *ioSTF* handler. If the actions are different, however, then *ioENF* must be given its own handler.

Although the standard responses to *ioSFC* and *ioSFS* are the same, separating the cases improves the optimization of the SKF value calculation.

For input and output:

```

case ioIOI:
    data = dev_ibuf;
    break;

case ioIOO:
    dev_obuf = data;
    break;

```

For the control logic:

```

case ioPOPIO:
    dev_flag = dev_flagbuf = SET;

case ioCRS:

case ioCLC:
    dev_control = CLEAR;
    break;

case ioSTC:
    dev_control = SET;
    break;

```

Note that PRESET generates POPIO and CRS signals, but we dispatch only the former, so the POPIO handler always falls into the CRS handler. Moreover, if the CRS action is the same as the CLC action, the CRS handler may simply fall into the CLC handler, as in the example above. Otherwise, the CRS action should be specified, and a `break` inserted to terminate the handler.



```
return SCPE_OK;
}
```

If any power-on actions need to be taken, they are done first. Then POPIO is sent directly to the signal dispatcher, where execution will fall into the CRS handler. Finally, any required simulator-specific operations, e.g., canceling in-progress simulation events or initializing state variables, are performed.

A device simulator generally will set the flag buffer and flag in response to operation completion, typically in the device's unit service routine. This may be done by:

```
dev_io (dev_dib.devno, ioENF, 0);      /* send ENF signal */
```

If special handling is required, e.g., because SRQ is separated from FLG on the interface, then ENF may be used to set the flag and SIR may be used to set SRQ:

```
dev_srq = SET;                        /* set SRQ flip-flop */
dev_io (dev_dib.devno, ioSIR, 0);     /* send SIR signal */
```

SIR must be dispatched if any of the flip-flops that affect generation of CPU signals are changed. For a standard interface, where SRQ follows FLG, these would be the control, flag buffer, or flag flip-flops. If SRQ is independent of FLG, then SIR is required after changing the SRQ flip-flop as well.

## Summary

The original I/O simulation structure 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 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, reimplementing 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.