The SIMH Breakpoint Subsystem

Bob Supnik, 26-Jul-2003 (revised 18-Jan-2006)

COPYRIGHT NOTICE

The following copyright notice applies to the SIMH source, binary, and documentation:

Summary

SIMH provides a highly flexible and extensible breakpoint subsystem to assist in debugging simulated code. Its features include:

- Up to 26 different kinds of breakpoints
- Unlimited numbers of breakpoints
- Proceed counts for each breakpoint
- Automatic execution of commands when a breakpoint is taken

If debugging is going to be a major activity on a simulator, implementation of a full-featured breakpoint facility will be of immense help to users.

Breakpoint Basics

SIMH breakpoints are characterized by a type, an address, a class, a proceed count, and an action string. Breakpoint types are arbitrary and are defined by the virtual machine. Each breakpoint type is assigned a unique letter. All simulators to date provide execution (“E”) breakpoints. A useful extension would be to provide breakpoints on read (“R”) and write (“W”) data access. Even finer gradations are possible, e.g., physical versus virtual addressing, DMA versus CPU access, and so on.

Breakpoints can be assigned to devices other than the CPU, but breakpoints don’t contain a device pointer. Thus, each device must have its own unique set of breakpoint types. For example, if a simulator contained a programmable graphics processor, it would need a separate instruction breakpoint type (e.g., type G rather than E).

The virtual machine defines the valid breakpoint types to SIMH through two variables:

> **sim_brk_types** – initialized by the VM (usually in the CPU reset routine) to a mask of all supported breakpoints; bit 0 (low order bit) corresponds to type ‘A’, bit 1 to type ‘B’, etc.
>
> **sim_brk_dflt** – initialized by the VM to the mask for the default breakpoint type.

SIMH in turn provides the virtual machine with a summary of all the breakpoint types that currently have active breakpoints:

> **sim_brk_summ** – maintained by SIMH; provides a bit mask summary of whether any breakpoints of a particular type have been defined.

When the virtual machine reaches the point in its execution cycle corresponding to a breakpoint type, it tests to see if any breakpoints of that type are active. If so, it calls **sim_brk_test** to see if a breakpoint of a specified type (or types) is set at the current address. Here is an example from the fetch phase, testing for an execution breakpoint:

```
/* Test for breakpoint before fetching next instruction */

if ((sim_brk_sum & SWMASK (‘E’)) &&
     sim_brk_test (PC, SWMASK (‘E’))) <execution break>
```

If the virtual machine implements only one kind of breakpoint, then testing sim_brk_summ for non-zero suffices. Even if there are multiple breakpoint types, a simple non-zero test distinguishes the no-breakpoints case (normal run mode) from debugging mode and provides sufficient efficiency.

Testing For Breakpoints

Breakpoint testing must be done at every point in the instruction decode and execution cycle where an event relating to a breakpoint type occurs. If a virtual machine implements data breakpoints, it simplifies implementation if data reads and writes are centralized in subroutines, rather than scattered throughout the code. For this reason (among others), it is good practice to perform memory access through subroutines, rather than by direct access to the memory array.

As an example, consider a virtual machine with a central memory read subroutine. This routine takes an additional parameter, the type of read (often required for memory protection):

```
#define IF        0           /* fetch */
#define ID        1           /* indirect */
#define RD        2           /* data read */
#define WR        3           /* data write */

t_stat Read (uint32 addr, uint32 *dat, uint32 acctyp)
{
static uint32 bkpt_type[4] = {
      SWMASK (‘E’), SWMASK (‘N’),
      SWMASK (‘R’), SWMASK (‘W’) };

If (sim_brk_summ &&
    sim_brk_test (addr, bkpt_type[acctyp]))
      return STOP_BKPT;
else *dat = M[addr];
return SCPE_OK;
}
```

This routine provides differentiated breakpoints for execution, indirect addresses, and data reads, with a single test.

The Replay Problem and Breakpoint Classes

When a breakpoint is taken, control returns to the SimH control package. When execution resumes, the same breakpoint will be reached and taken. This could result in an endless loop, with the simulator never progressing beyond a breakpoint.

When a breakpoint is taken, SimH remembers the address of the breakpoint and sets an internal flag. If the next breakpoint test is to the same address, SimH suppresses the breakpoint and clears the internal flag. Thus, the simulator can make progress past the breakpoint but will take the breakpoint again if control returns to the same address.

Unfortunately, this simple scheme won't work if a simulator implements multiple breakpoint types, and multiple breakpoints occur within the same instruction. Suppose a simulator implements E, R, and W breakpoints, and an instruction triggers an E breakpoint when fetched and an R breakpoint when executed. SimH will loop endlessly:

- Fetch instruction, take E breakpoint (saved address is instruction)
- Continue
- Fetch instruction, suppress E breakpoint, take R breakpoint (saved address is data address)
- Continue
- Fetch instruction, take E breakpoint
- Continue
- Fetch instruction
- Take R breakpoint, etc

To help with these loops, SimH implements up to 64 breakpoint classes. Each breakpoint class has its own saved address and valid flag. Thus, if the E, R, and W breakpoints are assigned to separate classes, each will be suppressed in turn until the next breakpoint test on that class that fails or that uses a different address.

Breakpoint classes are arbitrary identifiers and can be assigned by the simulator writer as desired. The class is specified as part of the breakpoint type in the call to **sim_brk_test**:

<31:26> = class number (0 by default)
<25:0> = bit mask of breakpoint types

Note that breakpoint classes and breakpoint types are orthogonal. Thus, classes can be used to distinguish different cases of the same breakpoint type. For example, in an SMP system with 'n' processors, classes 0..n-1 could be used for

E-breakpoints for processors 0..n-1. Or in a VAX, classes 1..6 could be used for data breakpoints on operands 1..6, with 0 reserved for the CPU’s E-breakpoints.

SimH’s capability to remember (and suppress) breakpoints on a class basis requires the simulator’s cooperation in some cases. Consider the following loop from an 18b PDP system:

```
LAC FLAG
SNA
JMP .-2
```

If a data breakpoint is set on FLAG, it will not be sprung on alternate instances, because SimH remembers the address of flag and the prior breakpoint. The simulator must inform SimH that instruction execution has completed and that any remembered breakpoint addresses for the data breakpoint class should be forgotten. SimH supplies an API call to do this:

sim_bkpt_clrspc (uint32 class)

Routine **sim_bkpt_npc** (void) will clear all remembered addresses, but this call incurs substantial overhead and should not be used routinely.