NLSIM EXAMPLES

The following are NLSIM examples. Examples consist of hardware platform simulation and software written in MIPS assembly language to exercise that platform.

In these examples, all performance numbers are computed using my Sony laptop model FXA-36 with an AMD CPU running at 1 GHz.

PLATFORM #1 c:\mips32\generic

This is a generic platform that consists of only the MIPS CPU and memory. A platform that is compatible with SPIM.

SPIM or SPIM MIPS simulator is a popular MIPS32 simulator written by James Larus. If you go to www.google.com and enter MIPS simulator, the number one entry is the web page for SPIM MIPS simulator. Here is a quick feature comparison between NLSIM and SPIM.

	NLSIM MIPS simulator	SPIM MIPS simulator
Simulate MIPS32 instructions	yes	yes
Simulate MIPS32 CPU	yes	no
Type of debugging break points	Execute Data read Data write	Execute
Simulate external hardware	yes	no
Cache decoded instruction	yes	yes
Support self-modified code	yes	yes
Simulate exception	yes	yes
Simulator external interrupts	yes	no
Type of address translation	Block mapping	none
Simulate CPU cache	no	no
Keep track of execute cycle	yes	no
Simulate the entire 4 Gigabytes memory space.	yes	no
Other features	Give warning on un-initialized memory read. Stop simulator on un-initialized instruction.	?
Performance	About 20 mips on 1 Ghz PC	About 20 mips on 1 Ghz PC

sort.asm Example #1 Platform #1 directory: c:\mips32\generic

This first example for this platform is to sort 16,384 32-bit integers. This array is populated with numbers in ascending order counting from 0 to 16383. It then sorted in descending order. For NLSIM, it takes 52 seconds to sort this array for a total of over one billion instructions. The performance is roughly 22 mips.

To compile sort.asm, open a DOS window, change directory to

```
c:\>cd c:\mips32\generic
c:\mips32\generic>mips_asm sort
```

To simulate sort, type

```
c:\mips32\generic>nlsim sort
```

To run, within NLSIM, type

```
>go
```

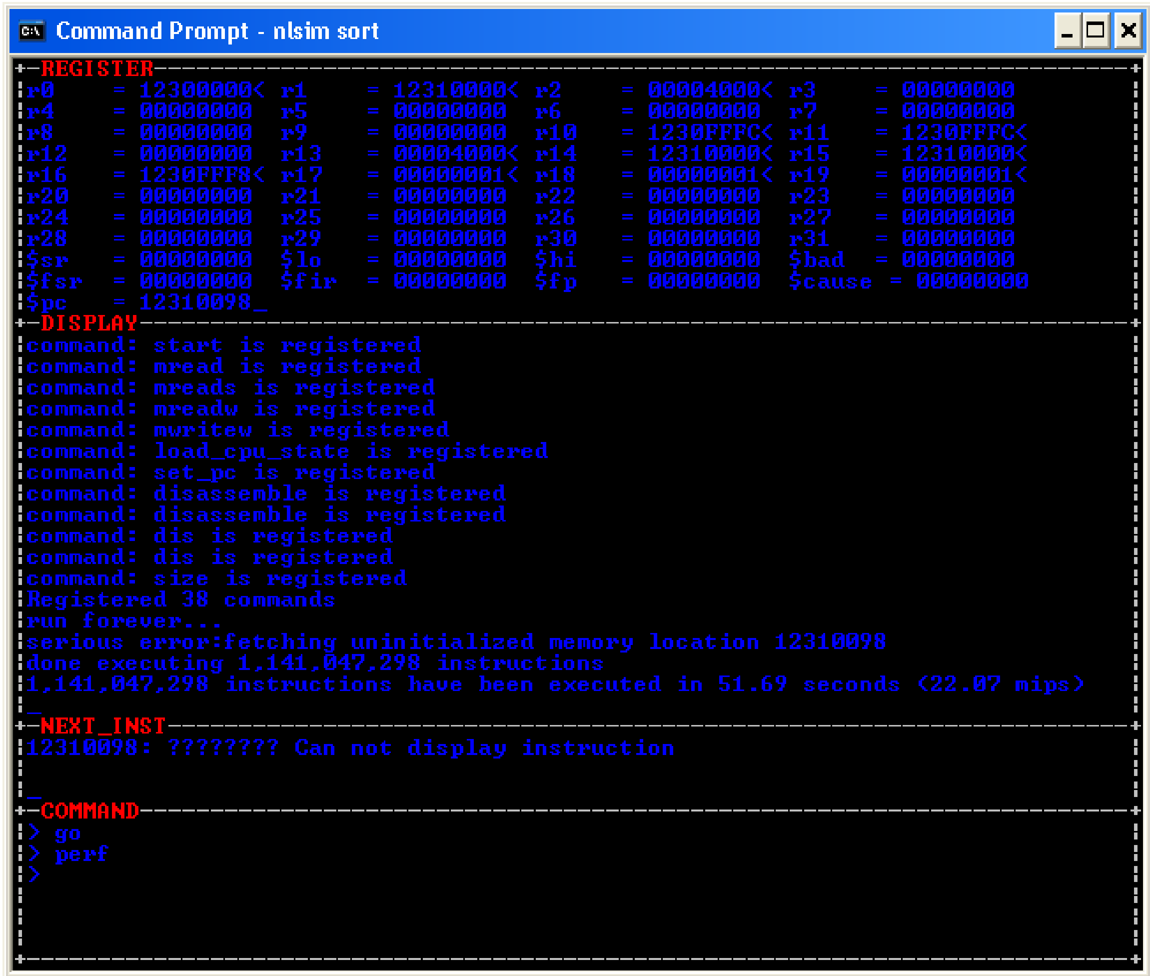
After simulation finishes, in order to get performance figure, type

```
>perf
```

To exit simulation, type

```
>quit
```

Here is the snapshot of the result:



Note that, the simulator starts to execute the first instruction at address 0xbfc0 0000. But for simplicity, if the source code contains symbol “main” or “bat_dau”, then the execution starts at this “main” symbol wherever it is.

For SPIM, it takes the same amount of time to execute this program. That means SPIM execution context is also very efficient.

Playing around with SPIM a bit and figure out that the simulator also supports self-modifying instruction. However, SPIM does not have any data memory state or execution memory state. SPIM will decode all instructions within text segment at loading time. Whenever a location within text segment is modified, SPIM will re-decode instruction immediately.

NLSIM has all the memory states and does not have to decode instruction ahead of time. It will decode instruction “on demand”. That is it only decodes instruction that is about to execute. Writing data to a previously decoded instruction location will not trigger instruction decoding.

This would lead us to a small example.

self.asm Example #2 Platform #1 directory: c:\mips32\generic

Here is the compiled listing of self.asm, self.lst:

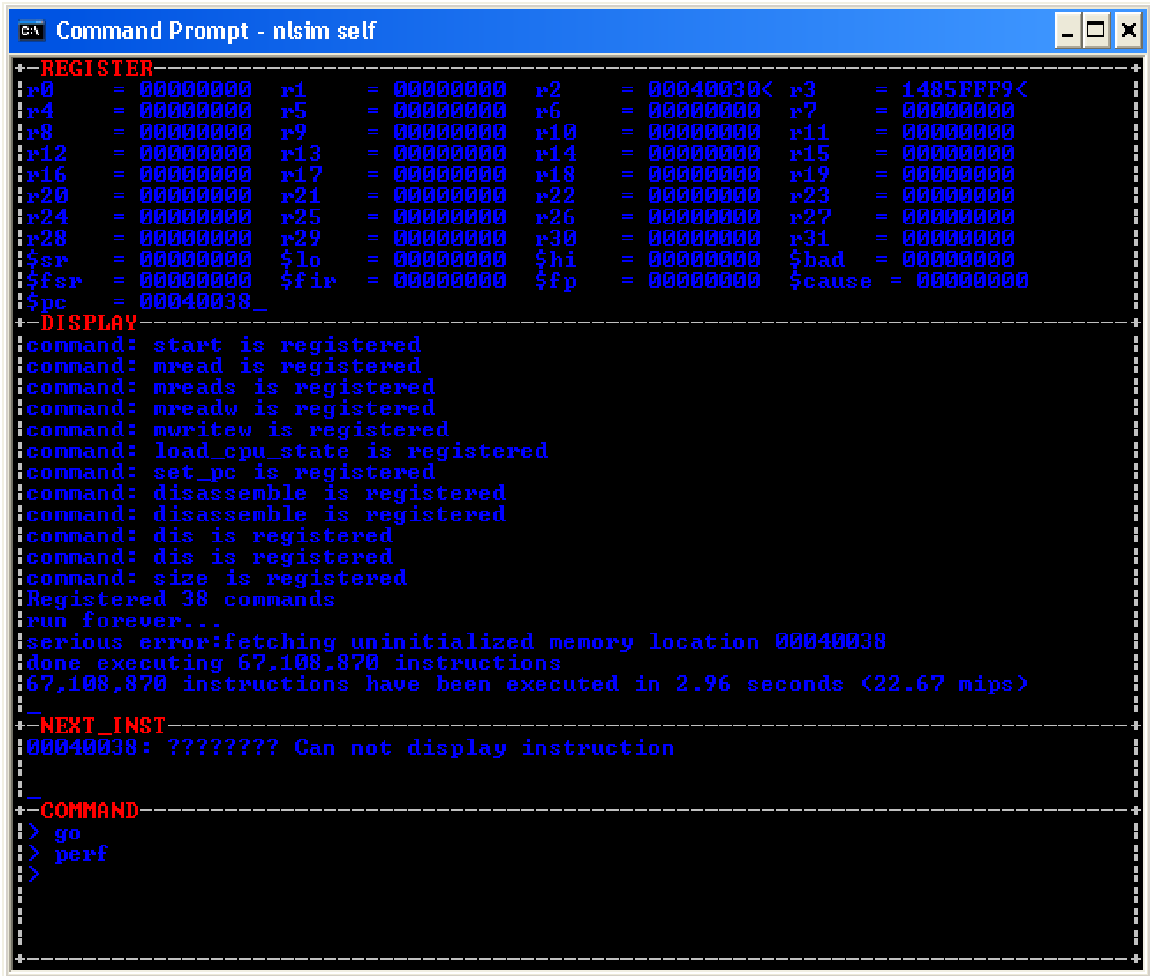
```
                           org 0x00040000
                   bat_dau:
3C020004 00040000          lui      r2,@pounding
34420030 00040004          ori      r2,r2,pounding&0xffff ; r2 = addr of
modifying instruction
8C430000 00040008          lw       r3,0(r2)             ; r3 = opcode to modify
3C0400FF 0004000C          lui      r4,0xff
3484FFFF 00040010          ori      r4,r4,0xffff         ; r4 = loop counter
00A52826 00040014          xor      r5,r5,r5
                   loop:
AC430000 00040018          sw       r3,0(r2)             ; modify "self" opcode
2484FFFF 0004001C          addiu    r4,r4,-1             ; counter--
1485FFFD 00040020          bne      r4,r5,loop
00000000 00040024          nop

                   stop_here:
00000000 00040028          nop
00000000 0004002C          nop
                   pounding:
1485FFF9 00040030          bne      r4,r5,loop
00000000 00040034          nop

```

You can write similar code with SPIM. Do not declare any data segment. Just use text segment.

NLSIM in this case runs about 10 times faster than SPIM.

Note: if you run this program with SPIM for a couple times, Windows will run out of virtual memory.

Here is the snapshot of this program running NLSIM:



benchm.asm Example #3 Platform #1 directory: c:\mips32\generic

benchm.asm is a special assembly language program and requires co-operation with NLSIM to roughly compute the speed of each instruction.

To run benchm.asm, type:

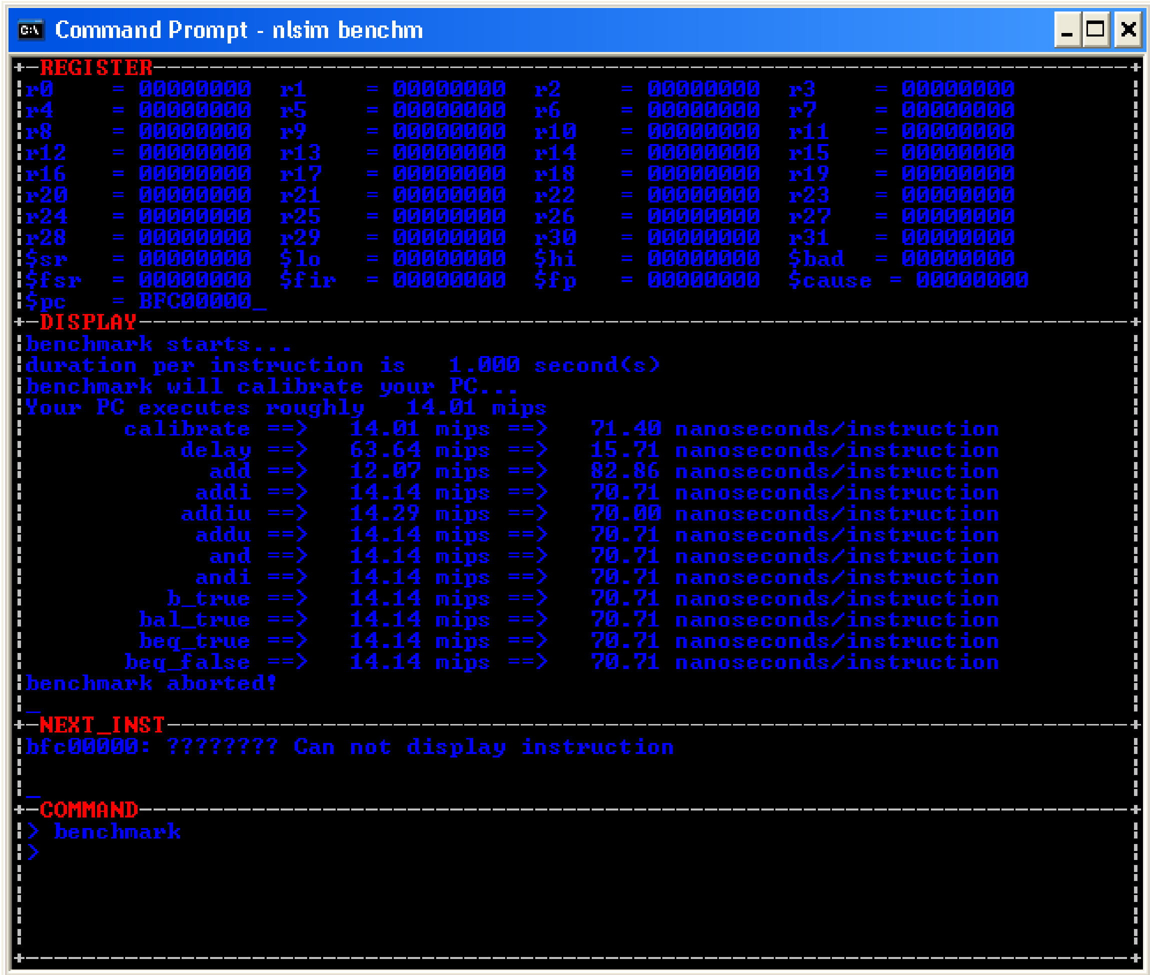
c:\mips32\generic>nlsim benchm

Within NLSIM, type:

>benchmark

Within NLSIM, you can terminate almost any long operation using Control-C.

Here is the snapshot:



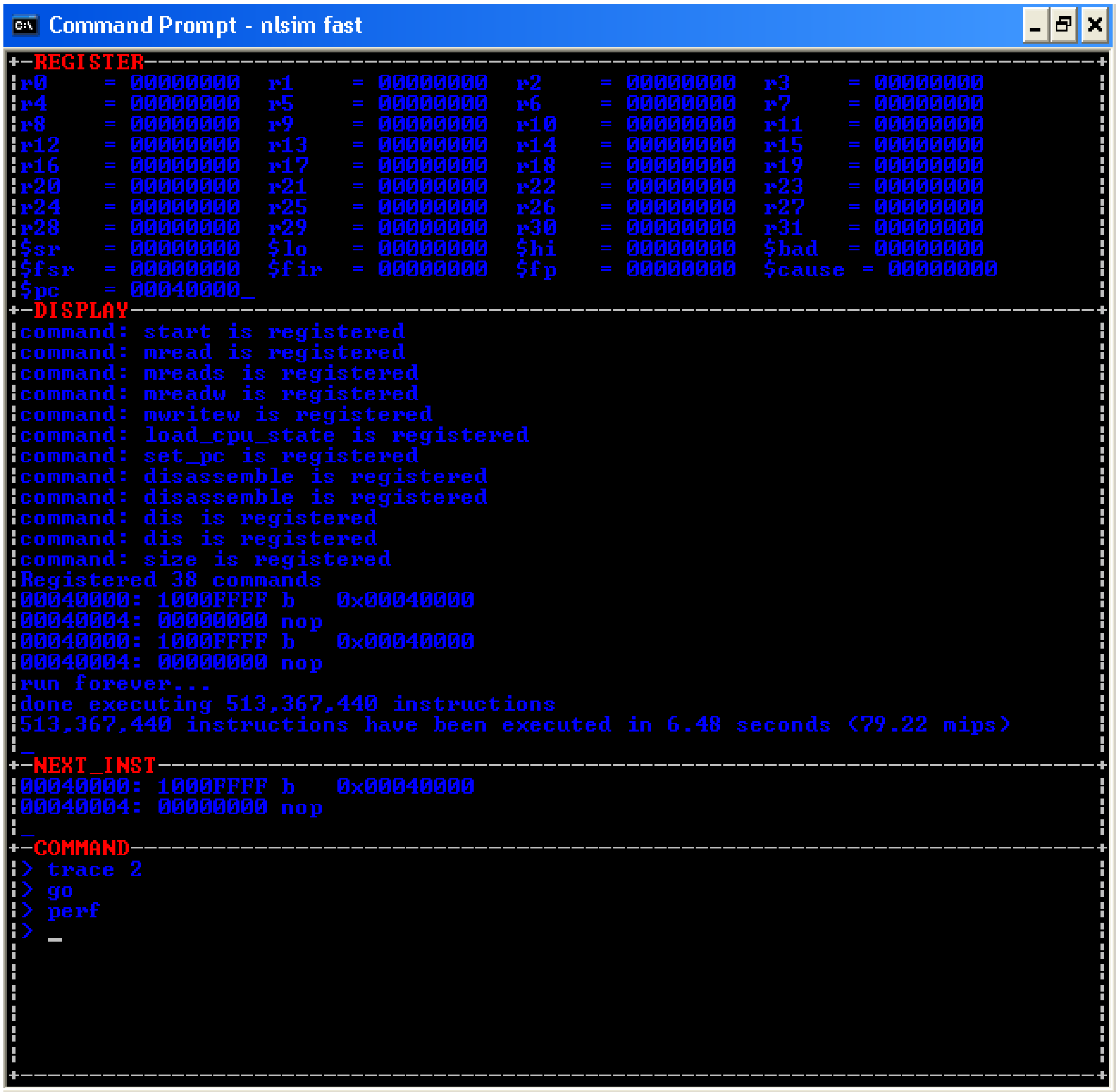
fast.asm **Example #4 Platform #1 directory: c:\mips32\generic**

The previous benchmark example only tries to mimic execution of a typical instruction. It turns out that on my laptop each instruction only runs about 14 mips. This is not true as you notice that the sort.asm program runs about 22 mips.

In this example, I will try to run some simple instructions to show that the simulator can run even faster. Here is the fast.asm program:

```
                          org 0x00040000
                   main:
1000FFFF 00040000         b       main
00000000 00040004         nop

```

And here is the snapshot of execution. It runs at 79 mips.



span.asm **Example #5** **Platform #1** **directory: c:\mips32\generic**

This span.asm program demonstrates NLSIM ability to run program with large amount of code and data. The program first copies branch instruction to the beginning of each 64 Kbytes page for a total of one gigabyte. It then executes branch instructions it creates. That is to jump from page to page. The execution is extremely slow since each branch generates two page swaps. To cache one gigabyte of data, it requires roughly 1.25 gigabytes of hard disk spaces. You may not be able to execute a similar program with any other simulator.

exception.asm **Example #6** **Platform #1** **directory: c:\mips32\generic**

exception.asm is a program that will execute an unaligned data read instruction. This will generate an exception. Within the exception handler at address 0xbfc0 0380, this bad instruction will be skipped. Upon returning from this exception, the next instruction will be execute normally.

This example demonstrates the ability to generate internal exception by NLSIM.

The following is the snapshot of tracing this program:

```
Command Prompt - nlsim exception
+-REGISTER---------------------------------------------------------------------+
|r0     = 00001000  r1     = 10400000  r2     = FFFFFFFB  r3     = 00000001<
|r4     = 00000000  r5     = 00000000  r6     = 00000000  r7     = 00000000
|r8     = 00000000  r9     = 00000000  r10    = 00000000  r11    = 00000000
|r12    = 00000000  r13    = 00000000  r14    = 00000000  r15    = 00000000
|r16    = 00000000  r17    = 00000000  r18    = 00000000  r19    = 00000000
|r20    = 0000101C  r21    = 0000000E  r22    = 00000000  r23    = 00000000
|r24    = 00000000  r25    = 00000000  r26    = 00000000  r27    = 00000000
|r28    = 00000000  r29    = 00000000  r30    = 00000000  r31    = 00000000
|$sr    = 00000000  $lo    = 00000000  $hi    = 00000000  $bad   = 00000000
|$fsr   = 00000000  $fir   = 00000000  $fp    = 00000000  $cause = 00000000
|$pc    = 00001020_
+-DISPLAY----------------------------------------------------------------------+
|BFC00000: 3C000000 lui    r0,0x0000
|BFC00004: 3400000C ori    r0,r0,12
|BFC00008: 40010000 mfc0    r1,r0,0
|BFC0000C: 3C02FFFF lui    r2,0xFFFF
|BFC00010: 3442FFFB ori    r2,r2,-5
|BFC00014: 00220824 and    r1,r1,r2
|BFC00018: 40810000 mtc0    r1,r0,0
|BFC0001C: 3C000000 lui    r0,0x0000
|BFC00020: 34001010 ori    r0,r0,4112
|BFC00024: 00000008 jr    r0
|00001010: 3C000000 lui    r0,0x0000
|00001014: 34001000 ori    r0,r0,4096
|00001018: 8C020005 lw    r2,5(r0)
|BFC00380: 3C150000 lui    r21,0x0000
|BFC00384: 36B5000E ori    r21,r21,14
|BFC00388: 4014A800 mfc0    r20,r21,0
|BFC0038C: 26940004 addiu    r20,r20,4
|BFC00390: 4094A800 mtc0    r20,r21,0
|BFC00394: 42000018 eret
|0000101C: 8C030004 lw    r3,4(r0)
|_
+-NEXT_INST--------------------------------------------------------------------+
|00001020: 1000FFFB b    0x00001010
|00001024: 00000000 nop
|_
+-COMMAND----------------------------------------------------------------------+
|> trace 20
|> _
+------------------------------------------------------------------------------+
```

This program does not have any “main” symbol. Therefore, it starts to execute at location 0xbfc0 0000. It then jumps to “main1” at address 0x0000 1010. Main1 will generate an unaligned data read exception using instruction “lw r2,5(r0)” at address 0x0000 1018. This will get to the exception handler at address 0xbfc0 0380. This handler will retrieve

EPC and increase EPC by one instruction. Upon returning from the exception handler, execution will resume at the next instruction.

<u>PLATFORM #2 c:\mips32\external_interrupt</u>

ext_int.asm Example #1 Platform #2 directory: c:\mips32\external_interrupt

This is a custom platform that will generate external interrupts. This demonstrates that NLSIM has the ability to simulate external interrupt. The platform also simulates IO read.

External.c is part of the simulator. And ext_int.asm is the simulated MIPS program. These two have to co-operate with each other to carry out this simulation environment.

Explain of ext_int.asm:

This program has two IO read locations. "data" IO read location is at address 0x1111 0000. "done" IO read location is at address 0x1112 0000. Both of these locations are trapped by external simulation logic within the file "external.c".

Every time the external interrupt number 5 is generated, the interrupt handler "int5" will read a number from "data" location, and accumulate into a sum within register "r2". This process will continue as long as "done" is not 1. Checking for done equals to 1 is done within the background loop. When "done" is equal to 1, interrupt is disabled, and the program enters an infinite loop, one of the termination methods.

Explain of external.c.

When simulator finishes loading a MIPS program, phase_for_register_break_point() function will be called. In this case, the following hardware registrations are created:

```
address_struct addr1;
long long sim_time;
int i;

sim_time = 0ll;
for (i = 0; i < 50; i++) {
  sim_time += 10000000ll;
  register_event (sim_time, gen_interrupt5, (void *) 0);
}

addr1.addr = 0x11110000;
register_word_read_break (addr1, addr1, fetch_data);
addr1.addr = 0x11120000;
register_word_read_break (addr1, addr1, fetch_done);
done = 0;
```

The first C code section schedules 50 events ahead. Each event is 10 million cycles apart. When scheduled event cycle is reached, the action function gen_interrupt5() will be called.

The second C code section registers two IO read breakpoints.

The first IO read breakpoint is at address 0x1111 0000. When MIPS program, ext_int.asm, reads this location, function fetch_data() will be called. Fetch_data() will return a value to the simulated program.

The second IO read breakpoint is at address 0x1112 0000. When ext_int.asm reads this location, function fetch_done() will be called. Fetch_done() will return a value to the simulated program.

Fetch_done() returns the current “done” value to ext_int.asm. While fetch_data(), returns the current “current_data” value to ext_int.asm.

```
void fetch_done (address_struct addr, UINT *data, UCHAR *state)
{
  *data = done;
}

void fetch_data (address_struct addr, UINT *data, UCHAR *state)
{
  *data = current_data;
}
```

“done” and “current_data” is setup by gen_interrupt5() as followed:

```
static int num_ints = 50;
static unsigned int data_array [10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
static unsigned int current_data = 0xffffffff;
static unsigned int done;

void gen_interrupt5 (void *data)
{
  static int i = 0;

  current_data = data_array [i++ % 10];
  request_interrupt (5);
  num_ints--;
  if (num_ints == 0) {
      done = 1;
  }
}
```

Every time gen_interrupt5() is called, “current_data” will be assigned with an value within “data_array []”. The next time, “current_data” will be assigned with the next value within “data_array []”.

Every time, gen_interrupt5() is called, interrupt number 5 is requested using function request_interrupt().

If interrupt is generated, e.g. requested, 50 times, “done” value will be changed from 0 to 1. “ext_int.asm” will detect “done” value of 1 within the background loop, disable interrupt, and enter an endless loop.

PLATFORM #3 c:\mips32\sort2

sort.asm Example #1 Platform #3 directory: c:\mips32\sort2

This platform will run the same sort.asm program created in **c:\mips32\generic**. This platform will trap execution of sort.asm at two execution points, before and after sorting. For case before sorting, it will populate values within the sorting array. For case after sorting, it will retrieve the result to verify that the values are actually sorted. The number of instructions required to sort is also computed.

This technique is extremely useful for automated testing.

PLATFORM #4 c:\mips32\sort3

sort.asm Example #1 Platform #4 directory: c:\mips32\sort3

This platform will run the same sort.asm program created in **c:\mips32\generic**. This platform will trap execution of sort.asm at two execution points, before and after sorting, in order to compute the number of cycles executed.

This platform will not populate data within the simulator memory as platform# 3. Instead, it sets external IO access for the entire "data" array, forcing sorting values to be fetched and stored externally. Using this technique, the simulator also can verify the correctness of the algorithm. In addition, external simulation logic also computes the number of reads and writes access to the "data" array.

PLATFORM #5 c:\mips32\pc_serial

serial.asm Example #1 Platform #5 directory: c:\mips32\pc_serial

So far all platform and examples are interacted within the simulated environment. This final example will connect the simulator to an actual hardware, a PC serial port. This example will only run on Windows DOS environment.

Within DJGPP, there is a function call _bios_serialcom(). This function will interact with the PC serial port. It allows reading and writing characters to the serial port. Please go to www.delorie.com for documentation.

In this example, the external simulation logic, external.c, will act as a middleman between the MIPS assembly language program running within the core simulator and the PC serial port.

Here is how it was simulated:

_bios_serialcom() has three input arguments and one return value:

```
serialcom = _bios_serialcom (cmd, port, data);
```

The MIPS program, serial.asm, will write values into cmd, port, and data. These are IO write locations. The external simulation logic will remember these values when they are written. When the MIPS program reads serialcom, an IO read location, the external simulation logic, external.c, will invoke the _bios_serialcom() function call using remembered values for cmd, port, and data. It then returns the return value to serialcom.

Here is how it is implemented in external.c

We have four IO locations to register:

```
symbol_address ("cmd", &addr1);
register_word_write_break (addr1, addr1, receive_write_cmd);

symbol_address ("port", &addr1);
register_word_write_break (addr1, addr1, receive_write_port);

symbol_address ("data", &addr1);
register_word_write_break (addr1, addr1, receive_write_data);

symbol_address ("serialcom", &addr1);
register_word_read_break (addr1, addr1, bios_serialcom_value);
```

And here are their corresponding action functions:

```
unsigned int cmd, port, data, serialcom;

void bios_serialcom_value (address_struct addr, UINT *data1, UCHAR *state)
{
        if (cmd == _COM_INIT)
                data = data_lookup_table [data];
        *data1 = _bios_serialcom (cmd, port, data);
}

void receive_write_cmd (address_struct addr, UINT data, UCHAR *state)
{
        cmd = cmd_array [data];
}

void receive_write_port (address_struct addr, UINT data, UCHAR *state)
{
        port = data;
}

void receive_write_data (address_struct addr, UINT data1, UCHAR *state)
{
data = data1;
}
```

That is it for external.c. With this platform, serial.asm will be able to access the external PC serial port. Serial.asm will read a string from a character terminal, then convert the all characters within this string to upper case, and display the converted string. Serial.asm will not be explained.

The following is the snapshot of the character terminal as it interacts with serial.asm running within the NLSIM simulator. This terminal is minicom running under Linux on a separate machine. This machine and the machine that run NLSIM are connected via a null modem serial cable. Settings for minicom are 9600 bauds, 8 bits, no parity, 1 stop bit, and local echo is on.