NLSIM User Guide

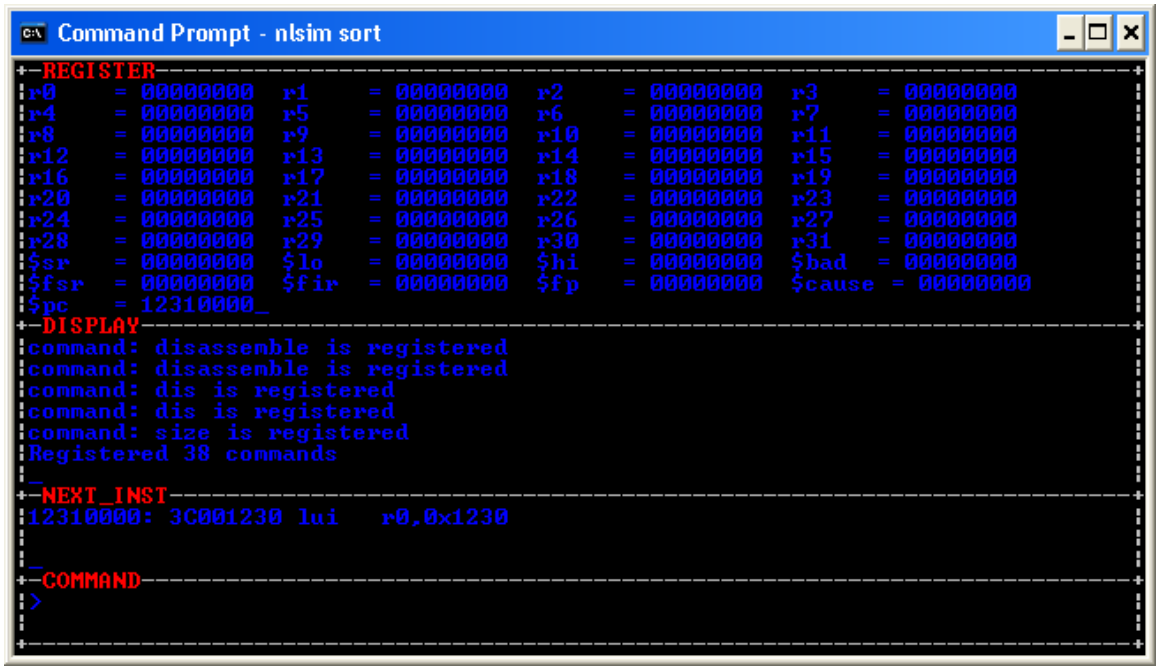
1) INVOKING NLSIM

NLSIM program is intended to run under DOS or LINUX.
To simulate a program named 'sort' compiled using mips_asm:

- Open a DOS or LINUX windows
- Resize the window as large as you can
- Type: **c:\mips32\generic> nlsim sort**

NLSIM with automatically configures sub-windows depends on the size of opened window.

Here is a snap shot under DOS



2) LAYOUT OF SUB_WINDOWS UNDER NLSIM
NLSIM has four windows.

- The window on top is used to display MIPS CPU registers
- The next window is used to display simulator messages. Any message either as a result of a command or as a result of simulation is displayed here.
- The next thin window is used to display the next assembly language instruction to execute.
- The bottom window is used to enter NLSIM commands.

3) NLSIM COMMANDS

‘quit’ or ‘q’	To exit simulator.
‘reg’ or ‘r’	To display register value. This command is now obsolete since we have separate window to display registers.
‘go’	To run simulator forever or until it hits a break point or until we stop using control-c.
‘step’ or ‘s’ <cnt>	To execute a specified number of instructions. Step without argument will step 1 instruction.
‘trace’ or ‘t’	Like ‘step’ but also print out executed instruction.
‘perf’	To display the performance of the simulator based on the last execution.
‘mread’ <addr> <cnt>	To dump 8-bit memory values.
‘mreads’ <addr> <cnt>	To dump 16-bit memory values.
‘mreadw’ <addr> <cnt>	To dump 32-bit memory values.
‘mwritew’ <addr> <data> <data> …..	To write 32-bit memory values.
‘set_pc’ <value>	To set instruction address.
‘disassemble’ or ‘dis’ <addr> [<cnt>]	To disassemble instructions.
‘list_break’	To list all the break points. Break point can be set by user or can be set by simulator to simulate external hardware.
‘list_symbol’	To list all symbols within an assembly language program. Symbols start with ‘_’ will not be listed.
‘benchmark’ or ‘b’	To display performance of each instruction. The simulator must be loaded with file ‘benchm’.
‘break_exec’ <start_addr> <end_address>	To set execution break point(s) for a range of addresses.

‘unbreak_exec’ <break_id> To remove break point(s) set by ‘break_exec’. ‘break_id’ is provided using ‘list_break’.

‘break_read’ <start_addr> <end_addr>
To set 32-bit memory read break point(s) for a range of addresses.

‘unbreak_read’ <break_id> To remove break point(s) set by ‘break_read’. ‘break_id’ is provided using ‘list_break’.

‘break_write’ <start_addr> <end_addr>
To set 32-bit memory write break point(s) for a range of addresses.

‘unbreak_write’ <break_id> To remove break point(s) set by ‘break_write’. ‘break_id’ is provided using ‘list_break’.

4) EXTENDING NLSIM

Besides numerous break point features that can be used by user to debug a program, NLSIM can be extended to simulate external hardware. A collection of functions is written only for this purpose. These functions are used in a separate ‘C’ file that can be linked with the core simulator to make a special purpose simulator. There will be plenty of examples within the distribution on how to simulate hardware.

The template ‘C’ file, external.c, documents all interface functions. Here are the descriptions of what each function does.

```
void register_illegal_page (USHORT start_page, USHORT end_page);
```

This function declares a section of CPU address spaces that is illegal to use by execution program. Each page has size of 64K bytes. Therefore, in a 32-bit address space, there are 65536 pages numbered from 0 to 65535.

```
int register_event (long long expire_time, void (*action) (void));
```

This function is to schedule a call back function “action” when simulation time has exceeded ‘expire_time’. With MIPS CPU, execution time of each instruction is equal. Therefore, simulation time is also counted as instruction cycle.

```
void request_interrupt (UINT int_num);
```

After calling this function, the simulator will enter into appropriate interrupt location. ‘int_num’ is valid between 0 and 5.

```
SHORT register_byte_read_break (address_struct begin, address_struct end,
       void (*action) (address_struct addr, UCHAR *data, UCHAR *state));
```

```
SHORT register_short_read_break (address_struct begin, address_struct end,
       void (*action) (address_struct addr, USHORT *data, UCHAR *state));

SHORT register_word_read_break (address_struct begin, address_struct end,
       void (*action) (address_struct addr, UINT *data, UCHAR *state));
```

These function reserve a section of 8-bit, 16-bit or 32-bit memory locations as I/O reads and register callback functions to supply 'data' to the simulator. Each of these functions returns a short integer as break identification. If the return value is -1, the registration process fails. Break identification can be used to un-register the break point.

```
void unregister_byte_read_break (SHORT break_id);
void unregister_short_read_break (SHORT break_id);
void unregister_word_read_break (SHORT break_id);
```

These functions un-register the break points set by `register_byte_read_break`, `register_short_read_break`, `register_word_read_break`. Registration or un-registration of a break point can be done at any time during simulation. Thus an I/O location can be assigned with multiple uses, a necessary feature of some hardware design.

```
SHORT register_byte_write_break (address_struct begin, address_struct end,
       void (*action) (address_struct addr, UCHAR data, UCHAR *state));

SHORT register_short_write_break (address_struct begin, address_struct end,
       void (*action) (address_struct addr, USHORT data, UCHAR *state));

SHORT register_word_write_break (address_struct begin, address_struct end,
       void (*action) (address_struct addr, UINT data, UCHAR *state));
```

These function reserve a section of 8-bit, 16-bit or 32-bit memory location as I/O writes and register callback functions to output 'data' from the simulator to external simulated environment. Each of these functions returns a short integer as break identification. If the return value is -1, the registration process fails. Break identification can be used to un-register the break point.

Memory read and memory write break points can be simultaneously applied to the same address location to simulator I/O that acts as both read and write.

```
void unregister_byte_write_break (SHORT break_id);
void unregister_short_write_break (SHORT break_id);
void unregister_word_write_break (SHORT break_id);
```

These functions un-register the break points set by `register_byte_write_break`, `register_short_write_break`, `register_word_write_break`. Registration or un-registration of a break point can be done at any time during simulation. Thus an I/O location can be assigned with multiple uses, a necessary feature of some hardware design.

```
SHORT register_exec_break (address_struct begin, address_struct end,
       void (*action) (address_struct addr));
```

This function is used to set an execution break point over a range of executable addresses. When any instruction within this range executed, the 'action' function will be called. This registration function returns a short integer as break identification. If the return value is -1, the registration process fails. Break identification can be used to un-register the break point.

```
void unregister_exec_break (SHORT break_id);
```

This function unregisters the execution break point set by `register_exec_break`. Registration or un-registration of a break point can be done at any time during simulation.

```
byte_write_for_byte_read_cache (address_struct address, UCHAR value);

short_write_for_short_read_cache (address_struct address, USHORT value);

word_write_for_word_read_cache (address_struct address, UINT value);
```

These functions are intended to speed up simulation of I/O reads. If an I/O value has not changed during simulation for a long time, it is a waste of time to invoke I/O read action function to supply read value. In this case, I/O read break point will not be set. I/O read will use the value stored in memory by these functions. When it is time for I/O read value to change, these function will write to appropriate memory location the value used by "I/O read".

```
SHORT symbol_address (char *symbol, address_struct *address);
```

This function will retrieve the address of any label defined within the simulated program. On success this function returns 0. It returns -1 on failure.

```
UINT register_read (int register_number);
```

This function returns the value of a register. Valid register number is from 0 to 31.

```
void register_write (int register_number, UINT value);
```

This function writes a value into a register. Valid register number is from 0 to 31.

```
void non_destructive_word_read (UINT *data, address_struct start, UINT count);
```

This function does a non-destructive read of a memory section into 'data' array starting at memory address 'start' for 'count' locations. This function always reads from memory and will not cause any break point to trigger.

```
void non_destructive_word_write (UINT *data, address_struct start, UINT count);
```

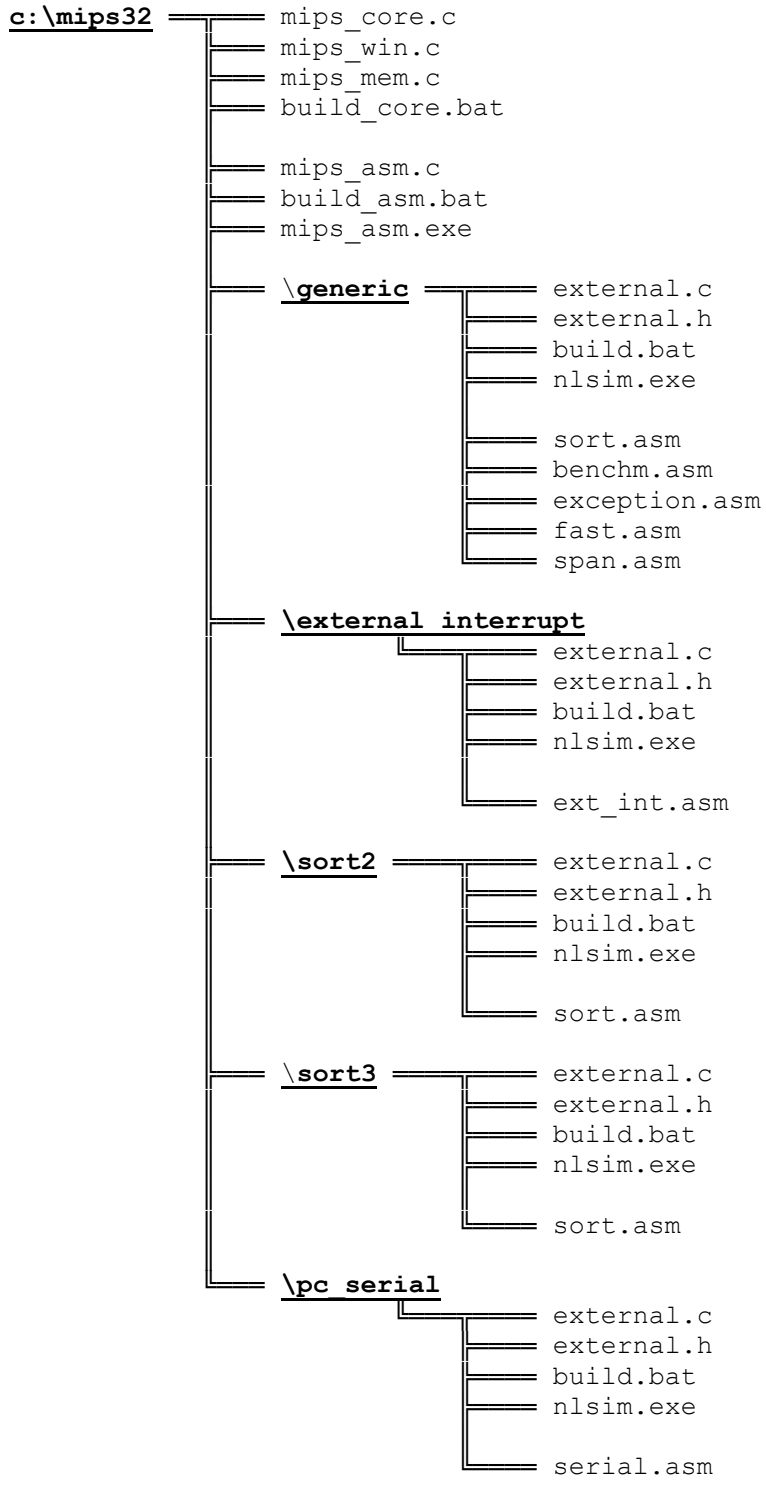
This function does a non-destructive write to a memory section starting at memory address 'start' for 'count' locations using an array of supplied 'data'. This function always writes to memory and will not cause any break point to trigger.

```
void request_time (void (*callback) (long long));
```

After calling this function, the simulator will flush execution pipeline, stop execution, compute current time, and supply time back to simulation environment using the provided callback function.

5) NLSIM DIRECTORY STRUCTURE

The following directory chart is the content of this distribution:


c:\mips32
mips_core.c
mips_win.c
mips_mem.c
build_core.bat
mips_asm.c
build_asm.bat
mips_asm.exe
\generic
external.c
external.h
build.bat
nlsim.exe
sort.asm
benchm.asm
exception.asm
fast.asm
span.asm
\external_interrupt
external.c
external.h
build.bat
nlsim.exe
ext_int.asm
\sort2
external.c
external.h
build.bat
nlsim.exe
sort.asm
\sort3
external.c
external.h
build.bat
nlsim.exe
sort.asm
\pc_serial
external.c
external.h
build.bat
nlsim.exe
serial.asm

To build mips_asm.exe, execute the build_asm.bat bat file.

To build nlsim.exe, we have to compile mips_core.c, mips_win.c, mips_mem.c, and external.c.

Except external.c, other files require no modification. And external.c is used to simulate external hardware.

I have organized them in a way that mips_core.c, mips_win.c and mips_mem.c stay in the root directory.

Each sub-directory will have its own external.c. Therefore, each sub-directory simulates a specific platform. Each sub-directory also has one or more ".asm" file. Each ".asm" file under a specific directory is coded to run under that platform.

6) STEPS FOR INSTALLING AND RUNNING NLSIM EXAMPLES

Before you can compile the assembler and simulator, you need to install a DOS compiler named DJGPP on your machine. DJGPP is free. Under Linux, you need gcc and ncurses.

Unzip nlsim.zip into a directory, for example directory 'c:\mips32'. Add DOS path to have 'c:\mips32', so you will be able to execute mips_asm.exe from any directory.

Open a DOS window, go to 'c:\mips32', and execute 'build_asm.bat', e.g. type 'build_asm' at the DOS prompt. This would build 'mips_asm.exe'.

c:\mips32> build_asm

Within this directory, execute 'build_core.bat' to build the fixed part of 'nlsim.exe'

c:\mips32> build_core

Go to any sub-directory and execute 'build.bat' to build 'nlsim.exe' for that specific platform.

Within the current directory, you can write any mips assembly language program. Then compile the program using 'mips_asm program_name'.

To simulate the compiled program, type 'nlsim program_name'.

For example, under 'c:\mips\generic', to build 'nlsim.exe', type

c:\mips32\generic> build

To assemble sort.asm, type

c:\mips32\generic> mips_asm sort

To simulate ‘sort’, type

c:\mips32\generic> nlsim sort