HOW NLSIM IS IMPLEMENTED

1) NLSIM DESIGN GOAL

NLSIM was designed to as an experiment to see how fast a full-featured CPU simulator/debugger can be implemented. The initial requirements are as followed:

- To simulate a 32-bit CPU with an entire address spaces of 4 gigabytes.
- To provide execution and data break points for user to debug their program.
- To simulate hardware attached to the CPU including I/O mapped accesses and interrupts.

2) OPTIMIZATION TECHNIQUES IN NLSIM

TECHNIQUE #1

In a normal simulation cycle, instruction is fetched, e.g. read from memory, decoded to figure out what kind of instruction. Decoded instruction is then executed.

The most widely known technique for speeding up instruction simulation is to remember what we have executed in an instruction location so that it won't be decoded again the next time the instruction executed. This trick works by the fact that instructions are often staying unchanged for a long time. But when instruction changes, decoding process has to be triggered again.

In my implementation, each instruction location, e.g. memory location, has a memory state variable. Besides other states stored in this variable, related states for this case are MEM_STATE_DECODED and MEM_STATE_UNDECODED.

Besides memory array to store instruction code, memory state, there is also an array of **execution context**. Each instruction location has a corresponding **execution context**. Before instruction is decoded, **execution context** has logic to trigger instruction decoding. After instruction is decoded, **execution context** has logic to execute decoded instruction as fast as it can.

In NLSIM, **execution context** is an array of 32 bytes or 8 32-bit integers. The format is followed:

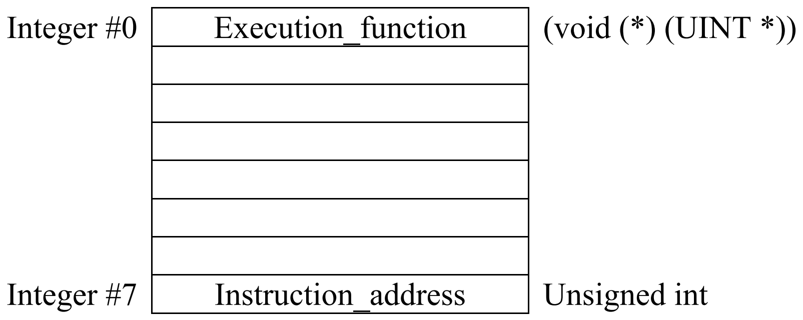

Figure 1 – General layout of execution context.

The first 32-bit integer within **execution context** is used to store pointer to function that handles instruction execution.

For memory state equals MEM_STATE_UNDECODED, this location has pointer to function to decode the instruction located at this location. When execute, instruction is decoded. The first 32-bit integer will be modified to contain pointer to decoded instruction. Memory state will change to MEM_STATE_DECODED.

For memory state equals MEM_STATE_DECODED, execution will execute without decoding phrase.

When reading data at this location, memory state stays unchanged.

When writing data to this instruction location and memory state is MEM_STATE_UNDECODED, memory data is written, memory state stays.

When writing data to this instruction location and memory state is MEM_STATE_DECODED, memory data is written, memory state changes to MEM_STATE_UNDECODED, and the first location of **execution context** will be modified to the function that decodes instruction again.

The last 32-bit integer within **execution context** stores instruction address, just for convenience.

Other locations within **execution context** can be used to store instruction operands, instruction data, next execution pointer, e.g. next execution address. There is no compromise on how execution information is represented for each kind of instruction. The arrangement is different from instruction to instruction.

Here is the example of the 'addu' instruction,

```
addu    r0, r1, r2
```

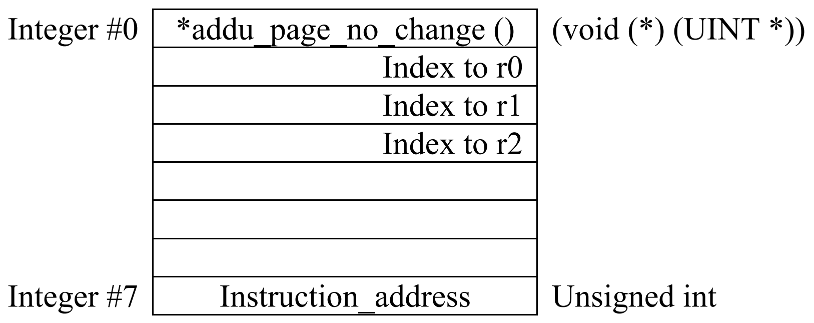

Figure 2 – Example of execution context for 'addu' instruction.

And here is how to invoke the execution:

```
((void (*) (UINT *)) (*(UINT *) current_execute_handle))
                                ((UINT *) current_execute_handle);
```

current_execute_handle points to the beginning of **execution context**.
And here is function implementation:

```
void addu_page_no_change (UINT *operand)
{
  reg [operand [1]] = reg [operand [2]] + reg [operand [3]];
  current_execute_handle =
        (void (*) (UINT *)) ((UINT) current_execute_handle + 8 * sizeof (UINT));
}
```

This technique, technique #1, is responsible for a ten-fold increase in performance.

<u>TECHNIQUE #2</u>

The last technique and the arrangement of **execution context** help to produce a very tight execution.

```
((void (*) (UINT *)) (*(UINT *) current_execute_handle))
                                ((UINT *) current_execute_handle);

void addu_page_no_change (UINT *operand)
{
  reg [operand [1]] = reg [operand [2]] + reg [operand [3]];
  current_execute_handle =
        (void (*) (UINT *)) ((UINT) current_execute_handle + 8 * sizeof (UINT));
}
```

The first statement is within a while loop to execute a specified number of instructions. And the second piece of code is a typical function execution.

Let look at the original while loop before technique #2 was introduced.

```
while (instruction_count < target_instruction_count) {
  ((void (*) (UINT *)) (*(UINT *) current_execute_handle))
                                  ((UINT *) current_execute_handle);
  instruction_count++;
}
```

In this loop, execution of instruction is carried out instruction by instruction. Note that, adding counter and checking if-statement happen for every instruction cycle.

If we can execute more than one instruction at a time, and still handle things correctly, that would be great. The final code as of this release looks like this:

```
while (instruction_count + 20 <= target_instruction_count)
{
  ((void (*) (UINT *)) (*(UINT *) current_execute_handle))
                                  ((UINT *) current_execute_handle);
     -
     -
     // execute these statement 20 times
     -
     -
  ((void (*) (UINT *)) (*(UINT *) current_execute_handle))
                                  ((UINT *) current_execute_handle);

  instruction_count += 20;
}
while (instruction_count < target_instruction_count) {
  ((void (*) (UINT *)) (*(UINT *) current_execute_handle))
                                  ((UINT *) current_execute_handle);
  instruction_count++;
}
```

That is on average, on a long run, we reduce the loop overhead by 95 percent. This trick, technique #2, improves the performance by about 20 percent.

The catch is that within while loop #1, if something happens and we need to stop the simulator in the middle of 20 statements, we have to do something to preserve the stopping states.

The solution is to save 'current_execute_handle', then assign 'current_execute_handle' to a dummy **execution context** that does nothing but counting number of instructions that should not be executed.

After getting out of the while loop(s), we restore the saved 'current_execute_handle', and adjust executed instruction counter by subtracting instruction that should not be executed!

3) TECHNIQUES TO SOLVE OTHER ISSUES IN NLSIM

TECHNIQUE #3

NLSIM uses hard disk space to cache **execution context** and other data structure so that simulation of the entire 32-bit address space is possible.

The 32-bit address space is divided in pages. The upper 16-bit of address space is used to select page number. Thus there are a total of 65536 pages. The lower 16-bit of address space is used as index into a specific page. Thus each page represents 64K bytes of memory space.

Page information is defined as C structure as followed:

```
memory_info page_handle [NUM_PAGE];
```

where NUM_PAGE is 65536.

Each page_handle [upper 16 bits of address space] has all the information about a page of size 64K bytes. Among other variables, variables that are worth to mention include:

`UINT page_state;` **/* page state */**

This is the state of the page which tells whether the page is in memory, not in memory, swapped out to hard disk, not allocated, or illegal to use.

`void *memory;` **/* memory */**

This is the pointer to 64K bytes of memory space. This space will be allocated dynamically when page is loaded into PC memory.

`UCHAR (*memory_state) [16384];` **/* data memory state */**

This is an array of data memory state for each 32-bit location. Data memory state and related state table will be useful for many tasks which include deciding whether an instruction is decoded or not, whether a memory location has user data access break points or hardware data access break points. Please refer to TECHNIQUE #4 for detailed description.

`UCHAR (*exec_state) [16384];` **/* execution memory state */**

This is the array of execution memory state for each 32-bit execution location. Execution memory state and related state tables are useful for setting up execution break points. Please refer to TECHNIQUE #4 for detailed description.

`UINT (*exec) [131072];` **/* execution context */**

This is the array of **execution context** mentioned previously. There are 16384 execution locations in a page. Each location has a corresponding **execution context**. Each **execution context** is 8 32-bit integers. Remember the first integer location within each **execution context** is **execution pointer**, a pointer to an action function that governs the execution engine. Please refer to TECHNIQUE #4 for detailed description.

```
UINT (*exec_save) [16384];
```
/* saved execution pointer */

This is the array of saved **execution pointer**. When execution break point function takes place of **execution pointer**, original **execution pointer** is stored here. Please refer to TECHNIQUE #4 for detailed description.

<u>TECHNIQUE #4</u>

To simulate data break points, we add more states into **data memory state** variable. Remember, each memory location has a corresponding **data memory state**.

To simulate hardware I/O mapped access, we do the same thing, e.g. adding more states to **data memory state** variable.

There is a **data memory state** variable for each 32-bit memory location. In this program, it is stored within the variable page_handle [].memory_state [].

The following is definition of memory states:

The least 3 significant bits, e.g. bit 0 to 2, have the following non-breakpoint states:

```
#define MEM_STATE_CHECK       0
#define MEM_STATE_BUS_ERROR   1
#define MEM_STATE_UNALLOC     2
#define MEM_STATE_UNINIT      3
#define MEM_STATE_SWAP_OUT    4
#define MEM_STATE_UNDECODED   5
#define MEM_STATE_DECODED     6
```

The next 4 bits, e.g. bit 3 to 6, define states for data and I/O break points:

```
/* hardware state break, bit 3-4 */
#define MEM_STATE_READ_HW     8
#define MEM_STATE_WRITE_HW    0x10


/* user state break, bit 5-6 */
#define MEM_STATE_READ_STOP   0x20
#define MEM_STATE_WRITE_STOP  0x40
```

For every memory access whether it is 8-bit, 16-bit or 32-bit, accessing function we are using is based on the state of the data memory.

For user data break points, e.g. `MEM_STATE_READ_STOP` or `MEM_STATE_WRITE_STOP`, after fulfilling required access, the simulator will be stopped. A flag is also set to indicate the reason for stopping the simulator.

For hardware data break points, e.g. `MEM_STATE_READ_HW` or `MEM_STATE_WRITE_HW`, callback functions will be called to fulfill access requirement. Callback functions are supplied by hardware simulation logic at the time of registering hardware data break points. Hardware break point will not cause simulator to stop. It is used to simulate hardware external to the CPU.

Believe it or not, none of the four types of breakpoint we describe so far uses any if-statement for checking whether the simulator hits any break point or not. All are implemented using finite states and finite state machines.

<u>TECHNIQUE #5</u>

To simulate user execution break point and hardware execution break point, the following variables for each execution location are used:

exec_state	**execution memory state**
exec (0)	**execution pointer**
exec_save	**saved execution pointer**

execution memory state has the following states:

```
#define MEM_EXEC_HW           1
#define MEM_EXEC_FETCH        2
#define MEM_EXEC_USER         4
```

And it has the following meanings:

For `MEM_EXEC_HW`, every time an instruction at this location is executed, a callback function will be called. Callback function is registered by external simulation logic by using function register_exec_break(). After this action callback function is carried out, the simulator will continue to execute instruction at this current location. This type of action is used to simulate hardware external to the CPU.

For `MEM_EXEC_USER`, every time an instruction at this location is about to be executed, the simulator will be stopped. This state is set whenever user issued a **break_exec** command.

For `MEM_EXEC_FETCH`, when an instruction at this location is executed, instruction decoding is performed every time. That is we can not remember decoded instruction at this location. This state is set as a result of setting hardware memory read break point by external simulation logic using function register_word_read_break().

This **execution memory state** will decide what state functions will be stored within **execution pointer**, e.g. exec (0), and **saved execution pointer**, e.g. exec_save.

Anytime when **execution memory state** changes, **execution pointer** will be loaded with:

exec_lookup_exec [**execution memory state**]

And **saved execution pointer** will be loaded with:

exec_lookup_save [**execution memory state**]

When **execution memory state** has state MEM_USER_EXEC, the running simulator will be stopped by executing execute_user_stop() function. For this case, in order to re-start the simulator again, **execution pointer** will be loaded with:

exec_lookup_dynamic [**execution memory state**]

That is all! This logic with states and state functions are setup in a way that there is no if-statement within execution engine per instruction cycle!