Experiences in Testing and Debugging the i960 MX VHDL Model

George S. Powley Jr.  Dr. Joanne E. DeGroat

Department of Electrical Engineering
The Ohio State University
Columbus, OH 43210

Abstract

We describe design decisions made while testing and debugging our behavioral VHDL model of the Intel i960 MX processor. The model provides human readable representations of processor information, to simplify debugging and documentation. An interactive simulation, using an X window interface, aids in debugging the VHDL model and application software. We developed an automated result checking entity to verify instruction operation tests and a flexible method to generate chebyshev approximations of transcendental functions and check their results.

1.1: Structure of the i960 MX VHDL model

The VIHDL model of the Intel i960 MX processor is a behavioral model of the i960 MX architecture that will be used to execute i960 machine code in a hardware/software codesign environment. The i960 MX system consists of the VHDL hardware models of the processor and memory, and the software that will be executed on the i960 MX. Currently, the application software can be written in C and i960 assembly code. The VHDL hardware models included in the system are the bus interface model, memory model, and the i960 MX model (see Figure 1).

Figure 1: i960 MX board block diagram

The purpose of this model is to provide a fast simulation model of the i960 MX processor. Therefore, the model has been written using non-standard signal types to reduce the total number of signals. The bus interface model is required to convert these internal signals to IEEE std_logic format. The dynamically allocated memory model provides access to the full, four gigabyte memory space, and supports an initial memory image input file. The i960 MX model provides an
### 2: VHDL modeling techniques

The VHDL model of the i960 MX processor includes several features which aid in debugging and testing its operation. Descriptive enumerated types are used to provide a meaningful representation of internal processor information, such as opcodes. Internal debugging signals are used to report state information not present on the pins of the processor, such as the instruction issue pipeline. These modeling techniques are discussed in the following sections.

#### 2.1: Descriptive enumerated types

Using descriptive enumerated types cases the development and testing of a VHDL model because they provide immediately recognizable information to the designer. An excellent example of this is the use of an enumerated type to represent the opcode in an i960 MX instruction. An opcode type is defined, which consists of the mnemonics for all 247 instructions in the i960 MX architecture. At the start of the simulation, a file containing the opcode values is loaded into a table. This technique provides three advantages: the opcode values are not hardcoded into the model, the VHDL code is self-documenting, and the simulation results contain instruction mnemonics, instead of opcodes. The VHDL code is self-documenting in all cases where the opcode of an instruction is tested (see Figure 2). Simulation results that are represented by enumerated types provide immediate feedback to the designer, which will reduce development time.

To convert a binary instruction word into a record of enumerated types, a table lookup method is used. The memory model provides the i960 MX model with a binary representation of the instruction word. The i960 MX determines the format of the instruction, and uses table lookup to determine the instruction, operands, and destination.

```vhdl
CASE opcode IS
  WHEN ADDR =>
    -- perform Add Real
  WHEN ADDRL =>
    -- perform Add Long Real
...
```

### Figure 2: Self-documenting VHDL code segment

### 2.2: Internal debugging signals

When debugging the VHDL model of a complex system, it is useful to have information about the state of the system that is not present on the entity’s output ports. We have used internal debugging signals to provide this information. These signals are used to carry actual internal values, such as register contents, as well as to represent higher level debugging information, such as an instruction pipeline. Signals that carry actual internal values can be thought of as test probes. These probes...
allow the designer to examine the contents of a variable. In the i960 MX model, the register file is represented as a variable. Using the internal debugging signals allows the designer to examine the contents of the register file after the simulation has completed. Signals that carry higher level debugging information can be designed to provide almost any type of information. By writing extra VHDL code in the model, the designer can test any number of signals and/or variables, and return a higher level representation of that information.

An example of this is demonstrated by the i960 MX instruction pipeline signals. Extra signals are added to the i960 MX model. These signals report the state of all instructions active in the i960 MX. An instruction can be in one of three states: decoded, issued, or executing. At the beginning of every clock cycle, the state of all decoded instructions is determined, and the instruction pipeline signals are updated. This method provides the designer with immediate feedback about the operation of the instruction scheduler, and is used to verify the operation of the i960 MX’s superscalar architecture.

3: Interactive simulation with X window interface

To simplify the development of the i960 MX model, an interactive interface to the VHDL simulation was created. The interface provides services available from VHDL debuggers, but at a higher level of abstraction, thus reducing the amount of interaction between the designer and the simulation. This interface consists of a C program that calls X Window routines. The X window provides a disassembled listing of the program, the contents of the registers, and a 1 Kbyte window of memory (see Figure 2). The interactive interface allows the designer to watch the contents of register and memory change as the program is executed in single step mode. Although the same goals might be accomplished by reading the results of a batch simulation, the X window approach provides the designer with a familiar interface and organizes the information in a logical way. The X window consists of the program display, register display, and the memory display, which are described below.

3.1: Assembler program display

The program display contains a disassembled listing of the executing program. Disassembled instructions are provided by gdump960, a program from the GNU/960 toolset that reports the address, contents, and disassembled instruction for each word in an i960 object file. The C program reads the gdump960 output file, and places each line in an array. To access this array, Styx is used to pass a signal containing the current Program Counter (PC) register value from the i960 MX VHDL model to the C interface program. When the C interface program receives a new PC value, the program display window is updated.

3.2: Register display

The register display gives the designer immediate feedback about the results of an executed instruction. By using the single step mode, the designer can examine the source operands and results of an instruction, as the program is being executed. This type of interface should be familiar to the designer, and it organizes the information in a compact, logical form. A floating point representation of the floating point registers is also displayed in the register display.

The register display receives all information from the i960 MX VHDL model, through Styx signals. Each time a register is changed, the VHDL model updates the Styx register signals with the new register values. When the C interface program detects this change, the register display window is updated. The conversion from a hex value to floating point values is also performed by the C interface program.

3.3: Memory display

The memory display is useful for debugging programs that use data structures in memory, such as a queue. The designer can set the memory display to show the address range of the queue, and use the single step execution mode to watch as the queue is updated. The combination of the program and register displays provides the designer with information about the memory addresses to be changed, and the memory display gives the results of the memory operation.

The memory display receives all of its information from the memory VHDL model, through Styx signals. When the memory model performs a write operation in the address range of the memory display, the Styx memory signals are updated. When the C interface program detects a change on these memory signals, the memory display window is updated.

3.4: Lessons learned

The idea of using Styx and an X window interface was straightforward, but the implementation was not trivial. The first problem encountered was how to link the library of X routines to the simulation code. At the
time of development, the documentation for Styx was not very clear on this subject, so a rather messy solution was used. Since then, a clean solution has been developed.

Two other problems encountered were how much control should be provided through the X interface, and what control mechanisms should be used. Ideally, this interface could be developed into an application that resembles xdbx (a debugger with GUI), providing breakpoints and other software debugging features. Since the purpose of this interface was to speed development of hardware, the interface remains simple, providing single instruction and free running execution modes.

The choices for control mechanisms were limited by the nature of the VHDL simulation and of X callback procedures. Using widgets to provide menus and buttons is the preferred method of implementing control mechanisms for X windows, but this method does not work with the VHDL simulation. To use widgets, a program registers callback functions with the X server and gives control of the program to the X server, resulting in the C procedure losing control of the program execution. The VHDL simulation requires a Styx C procedure to complete execution in order for the simulation to continue. Therefore, widgets with callbacks are not used by the Styx C procedure. The Styx C procedure could contain code to implement buttons and menus, but our interface uses mouse clicks as an input to minimize programming overhead.

4: Automated results checking

When testing the execution of all i960 MX instructions, a method for automatically checking the results of each test case is desired. The approach we have taken is to include both test code and expected results in an i960 assembly program, thereby simplifying test creation and modification. A result checking entity compares the expected results to the actual results, and generates a report of all tests performed. The test program format and results checking entity are described in the following sections.

4.1: Test program format

The assembly test program contains all information required to perform an instruction test and check the results, namely: test name, expected results, pre-test instructions, instruction under test, and test report instructions. An example test is shown in Figure 4.

```c
#Test addc_1 0_00010000 0_00000000
lda 0xfffffff,g0
modac g0,0,g0         #clear AC
lda 0xfffff,g0
addc g0,1,g4          #test addc
modac 0,0,q5
stml g5,0             #check results
```

**Figure 4: Automatic test code example**

The test name appears in the report generated by the result checking entity. The expected results are represented by one, two, or four hex words. Pre-test instructions are used to set the state of the processor before executing the instruction under test. These instructions could, for example, change the Arithmetic Controls (AC) register and modify register contents that will be used as operands in the test instruction. The instruction under test performs an action, and generates results, which are stored in memory, registers, or both. It is the responsibility of the test report instructions to send the results to the results checking entity. This is accomplished by using the store mixed instructions; stm, stml, and stmq, which store one, two, and four words, respectively. The destination for all store instructions is physical address zero. The results checking entity is activated when a write to memory location zero is detected, as described in the next section.

4.2: Result checking entity

The purpose of the results checking entity is to read the expected results from a file, compare the expected results to the generated results, and produce a report of all tests. The expected results file contains the first line of every instruction test, as shown in Figure 4. The result checking entity interface consists of an address bus, data bus, and read/write line, which are connected to the i960 local bus. When a write operation to physical address zero is detected, the result checking operation is performed. A line is read from the expected results file, and the values are compared to the data written by the i960. For each word in a particular test, a report line is generated. Each report line contains the name of the test, and the result of the test. If the test failed, the expected and generated results are also reported.

5: Floating-point routines and tests

Three problems encountered while writing VHDL functions to implement and test series approximations of trigonometric and logarithmic instructions were: how to determine the series coefficients for extended precision floating-point numbers, how to convert these coefficients
into a binary representation, and how to check the accuracy of the results. Maple, a symbolic math program, is used to solve all of these problems.

5.1: Chebyshev floating-point routines

Maple provides unlimited floating-point precision and a chebyshev approximation routine, two useful features for determining the coefficients of a series approximation. The chebyshev routine returns an approximation equation given the following parameters: function to be approximated, convergence range, and precision required. Maple determines the number of terms needed to meet the precision requirements and returns the approximation equation. An example Maple file that will return an approximation of \( \sin(x) \) is shown in Figure 5.

```
Digits := 30;
with(orthopoly, T);
readlib(write);
read fpout;
eq := simplify(chebyshev(sin(x), x=-1..1, 1e-20));
open('sin.cof');
for i from 0 to degree(eq) do
  fpout(coefl(eq, x, i));
od;
close();
```

Figure 5: Maple code to generate chebyshev coefficients for \( \sin(x) \)

The coefficients of the equation \( eq \) are precise to 30 decimal places, which will meet the requirements of computing an extended precision series. Using Maple to determine a chebyshev approximation is very powerful since the function to be approximated, the convergence range, and the precision of the result can be easily changed.

Once an infinite precision floating-point number is obtained, it must be converted to a binary form so that it can be used by a VHDL program. Maple provides a routine that converts a floating-point number to a binary number with infinite precision. A small routine is used to break the floating-point number into an exponent and a mantissa, then the mantissa is converted to a binary number and the results are written to a file. A C program reads the exponent/binary file and produces a constant table for a VHDL program or a list of hex values for a i960 assembly test program.

5.2: Automatic test generation

After the chebyshev approximation is written as a VHDL subprogram, the results are checked using Maple. A Maple program is used to generate the test values to be computed and the expected results. A C program reads the test values and the expected results, and generates an assembly language test program, as described in Section 4.1. Some results of the extended precision \( \sin \) routine are shown in Figure 6.

<table>
<thead>
<tr>
<th>Test Value</th>
<th>Maple Result (80-bit hex)</th>
<th>VHDL Result (80-bit hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3fff</td>
<td>3fff</td>
</tr>
<tr>
<td></td>
<td>0000000000</td>
<td>000000000</td>
</tr>
<tr>
<td>( \pi/6 )</td>
<td>3ffe</td>
<td>3ffe</td>
</tr>
<tr>
<td></td>
<td>8000000000</td>
<td>ffffffffff</td>
</tr>
<tr>
<td></td>
<td>0000000000</td>
<td>ffffffffff</td>
</tr>
<tr>
<td>( \pi/4 )</td>
<td>3ffe</td>
<td>3ffe</td>
</tr>
<tr>
<td></td>
<td>8000000000</td>
<td>b504f333</td>
</tr>
<tr>
<td></td>
<td>0000000000</td>
<td>f9de6484</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f9de6483</td>
</tr>
<tr>
<td>( \pi/3 )</td>
<td>3ffe</td>
<td>3ffe</td>
</tr>
<tr>
<td></td>
<td>8000000000</td>
<td>ddb3d742</td>
</tr>
<tr>
<td></td>
<td>0000000000</td>
<td>c265539d</td>
</tr>
</tbody>
</table>

Figure 6: Extended precision \( \sin \) results

6: The i960 MX VHDL model test set

To test the i960 MX system model, a number of tests are being developed. The tests fall into three categories: instruction operation tests, instruction scheduling tests, and input/output tests.

The operation tests are used to test all instructions of the i960 MX processor. These tests are implemented with assembly language programs, as discussed in section 4. Different combinations of operands are applied to the operations, in order to test all lines of the VHDL code.

The instruction scheduling tests are used to determine if the instruction scheduler model handles data hazards and structural hazards. These tests are also implemented using assembly language programs. The results of these tests can also be observed using the internal debugging signals, as discussed in section 2.2.

Input/Output tests are required to test the timing of the i960 MX bus signals. These tests will determine the accuracy of the i960 MX bus model compared to an actual i960 MX processor.
7: Conclusions

We have shown the methods used to speed debugging and testing our VHDL model of the Intel i960 MX processor. The goal of the modeling techniques has been to provide the designer with information that might not be available directly from the simulation results. This information is organized in a logical fashion to make the results easier to read. The automated result checking method provides a uniform test program format, which simplifies test writing and maintenance. Finally, Maple has been used to speed development of extended precision floating point routines and their tests.

References


