Experiences with VHDL Models of COTS RISC Processors in Virtual Prototyping for Complex System Synthesis

Thomas Egolf, Paul Kalutkiewicz, Vijay K. Madisetti, Shahram Famorzadeh
Electrical and Computer Engineering, Georgia Tech, Atlanta, GA 30332-0250
†Lockheed Sanders Inc., P.O. Box 868, Nashua, NH 03061.

Proceedings of VHDL International Users’ Forum (VIUF) Spring’95

Abstract—Significant increases in system complexity, shorter time to market requirements, rising life-cycle costs, and requirements for rapid upgrades of embedded signal processor systems together with agile system redesign (adaptability) to multiple mission and threat scenarios, are motivating the development of new methodologies for rapid prototyping of systems (i.e., system synthesis) as part of the US Department of Defense’s RASSP program. One methodology, that is the subject of this paper, involves the use of virtual prototyping to rapidly design hardware and software and facilitate their integration and test. This paper describes various aspects of the virtual prototyping process, starting with the VHDL library development of commonly used COTS processors, then proceeding to the modeling and prototyping of multiprocessor systems, and finally the virtual prototyping of an Infrared Search and Track (IRST) system prototype developed in collaboration with the Lockheed Sanders’, Hughes’, and Motorola RASSP demonstration teams. This paper will attempt to document our experiences with virtual prototyping of complex digital systems, such as radar signal processors, consisting of a few hundred processors with computational complexity rated at 1-100 GFLOPs, processing real time image data at 100-1000s Mbytes per second. Our main result shows that virtual prototyping reduced the in-cycle time required for hardware/software integration and test from 10-24 months typically required for large-scale designs to less than 2 months (assuming that all model libraries were designed and validated off-cycle). It was also observed that virtual prototypes that model a 200-processor signal processor to a high-level of fidelity (e.g., clock-edge behavior) demand very high computational resources (a potential show-stopper), and efficient mechanisms that trade-off fidelity (e.g., mixed-level) for performance need to be investigated.

I. INTRODUCTION

A virtual prototype (VP) may be defined as a “software simulatable” model of a hardware component, board, or system containing sufficient accuracy to guarantee its successful hardware/system-level realization capable of executing an application within a set of real-time, size, area, and power constraints. Virtual prototyping, essentially, eliminates hardware fabrication from the in-cycle design loop, speeding up the design process, allowing concurrent codesign of hardware and software, and also facilitating rapid HW/SW integration. Consequently, virtual prototyping is an important thrust of the Rapid Prototyping of Application Specific Signal Processing (RASSP) program efforts sponsored by the ARPA/US Department of Defense [1-2,11].

Virtual prototyping impacts the system synthesis process in the following positive ways —

- Permits the application of HW/SW codesign methodologies for verification of task partitioning and performance requirements,
- Allow verification of data distribution rates through various hardware elements prior to hardware development,
- Initiates software code development and test on the VP prior to the final system HW/SW integration and concurrently with HW design,
- Test portions of the software that interface to the hardware (i.e., communication protocol configuration) to discover design flaws early in the process that could require expensive software solutions later in the design phase,
- Facilitate hardware/software integration and test,
- Document the detailed system design for future model year upgrades.

The general flow during the initial stages of signal processor design is depicted in Figure 1 and is based on the following elements.

- Early trade-off analysis (i.e., search past design/reuse archives, architecture selection, allocation, scheduling, and HW/SW partitioning) to meet algorithmic, functional and sys-
Fig 1. The Virtual Prototyping Process.

- Commercial off-the-shelf (COTS) versus custom selection ("make" versus "buy" decisions),
- Identification (and reuse) and procurement of requisite VHDL model libraries for COTS and ASIC components and their validation,
- Interface design and verification,
- Detailed validation (test) via simulation of selected hardware and software architectures.

The first step in the design process is the determination of the high-level requirements of the application and includes an iterative algorithm and performance-level modeling stage of system design. This includes:

- Bit-width analysis (if low-power fixed-point front-end implementations are desired) to determine the minimum length required to meet algorithmic-level criteria,
- Performance-level modeling to determine the computational throughput and I/O requirements of the system to assist in the hardware/software architecture selection process.

Once the preliminary architectural trade-off analysis is completed, a library of COTS or custom parts is selected so that these components meet the requirements determined in the previous step. In this step, tradeoffs between an all-COTS solutions, a mixture of custom and COTS, or an all-custom solution are explored (the latter can be motivated by application dependent form constraints). If an all-custom solution is chosen, then the question may be raised as to the types of parts used to design the component. In this case, a custom non-COTS solution or all COTS solution can be used. These issues are addressed within the domain of algorithm design and performance level modeling (with extensive use of past archives and design advisors).

Virtual prototyping begins when this tradeoff analysis is completed and a few architectural choices are made. At the start of this stage all component models selected from a COTS library and those especially designed for a custom application are validated. If the library contains previously validated parts then only the custom parts are tested. Upon component verification sign-off, the behavior of the interface between components is designed/tested, with preference for standardized protocols. If there is a problem with the interfacing then the previous stage must be reanalyzed to determine the component responsible for the problem. After the interface behavior is validated, performance issues must be reexamined and compared with those found in the initial stages of performance modeling. If there is a problem at this stage, then the architecture selection phase is revisited so as to choose a different COTS component that will meet the requirements, or a redesign of a custom component is required to meet the performance criteria. Consequently, the design process is iterative.

Components in a Virtual Prototype contain the complexity of the unit being modeled to the level of detail that mimics (in part or completely) the documented physical characteristics, however, for purposes of efficient simulation, some component models could be utilized at a higher level of design abstraction. The documented physical characteristics of RISC processors can include the interface timing behavior, the instruction set execution times, the internal register formats, the cache structure and replacement strategies, and the data types of the floating point and integer units. In formation available on a custom-designed ASIC may include the detailed specification of its performance, timing files, and its gate-level implementation. Information not documented must be inferred by the modeler and available for users to modify and update when more information becomes available.

In the following sections, the various stages in virtual prototyping are discussed. The MCV9 Race
architecture from Mercury Computer Systems [3] and the Infrared Search and Track (IRST) system [4] designed in collaboration with the Lockheed Sanders-Hughes-Motorola RASSP teams, were chosen as representative examples of complex systems. The processor modeling description will primarily focus on the Intel i860XP processor contained within the processing element module of the MCV9 architecture. All modeling was done on Vantage Spreadsheet Version 5.011 or 5.015.

Section II of this paper addresses the issues of processor modeling. Included are descriptions for modeling the internal functionality and the interface behavior. The modeling methods for the various processor elements are described and the external timing behavior for a set of instructions that require data from the interface is examined.

Section III discusses the test bench required for a complex processor model. Memory and memory controller models are developed to test both the interface behavior and internal functionality of the i860. The methods employed to build the test routines and write script files for model regression testing will also be described. Clock and reset models were developed for synchronization between components.

Section IV describes the insertion of our VHDL models into the virtual prototype of the IRST system [4]. The various stages of building of VP are discussed, and tests performed at each stage are described. Metrics such as instructions/second executed were collected for various application programs compiled and run on the model. The effects on simulation time upon insertion of the i860 into the MCV9 and the MCV9 into the IRST are shown. Code was developed on the prototype to pass data through the various components for testing the interface protocol and data throughput rates of the system.

II. Processor Modeling

The i860XP component model contains a fully functional (FFM) behavior of the i860 design unit and its corresponding test bench. The fully functional processor model [5] describes the behavior of the documented (complete or partial) characteristics of the processor, and is comprised of an internal behavioral model and an interface wrapper with hooks between the two (Figure 2).

II.A Functional Partitioning

The internal register and data path elements of the i860 are shown in Figure 3. The current FFM of the i860XP is partitioned into seven VHDL pro-
processes as shown in Figure 4. The initial implementation consisted of a single process and represented a model closer to a purely instruction set simulator (ISS). In this case, only the results of the instruction execution were of importance, and no interface behavior was implemented. Instructions and data were read from internal memory arrays and care was not taken to accurately model the execution time of each instruction. As the model progressed and interface capabilities were added, it was no longer possible to use a single process because internal and external events required concurrency. One example occurred when a data load was in progress at the same time a non-pipelined floating point add was passing through the pipeline stages. If a wait statement was inserted inside the process, a stall would occur until the load had finished and the pipeline would not be advanced until the data was returned. Although the simulation speed was significantly higher for the ISS (≈ 2000 instructions/sec. as compared to ≈ 200), the ability to test interface characteristics inside a system environment could not be achieved. (We are currently investigating switching between ISS and fully functional by use of boolean variables to enable or disable each mode).

A breakdown of the functionality contained in each process is listed in the following table. The partition was chosen so as to minimize the use of internal signals (and mitigate their overhead). When data is passed between processes, bits and boolean signals proceed without modification, however, bit vectors are converted to integers or arrays of integers to minimize the signal count. They are converted to bit vectors upon reception by the receiving process. Later releases of our models may combine the single-cycle and burst-mode reads writes into one process. The majority of the functionality is contained inside the decode execute process and is an artifact of the initial ISS model.

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Functional Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>Instruction Cache, Instruction Buffers, Reset Functionality</td>
</tr>
<tr>
<td>MMU</td>
<td>Translation Caches, Address Translation Algorithm, Reset Functionality</td>
</tr>
<tr>
<td>Data Load/Store</td>
<td>Data Cache, Write Buffers, Pipelined Float Load Buffers, Reset Functionality</td>
</tr>
<tr>
<td>Single Cycle</td>
<td>Interface handling for single cycle reads and writes, Reset Functionality</td>
</tr>
<tr>
<td>Two Cycle</td>
<td>Interface handling for burst reads/writes of 2 cycles</td>
</tr>
<tr>
<td>Cache Fill/Write</td>
<td>Interface handling for cache fills and write backs</td>
</tr>
<tr>
<td>Decode/Execute</td>
<td>Core Execution Unit, Floating Point Execution Units, Graphics Unit, FP Pipelines, Register Files, Pipelined Float Load Buffers, Special Registers, Instruction Decode Mechanism, Dual Mode Processing, Instruction Interaction Checks, Dependency Checking, Trigger Mechanisms for Data Load/Store</td>
</tr>
</tbody>
</table>

### II.B Internal Model

The internal model provides for instruction fetch, decode, execution and write back. In addition, it could model internal data storage devices such as caches, registers and pipeline stages. The internal model is primarily contained inside the decode execute process, however, the instruction fetch, MMU, and data load/store processes contain the caches to permit their implementation using variable types. The functionality of each instruction, for example, floating-point add and multiply or integer logical operations, must also be modeled using a method that promotes reuse. Reuse is an important consideration when model year upgrades require the implementation of improved processors. Therefore, the code should be written with this in mind. The left hand side of Figure 2 illus-
Fig 5. Processor Status Register.

Fig 6. Instruction Decoding Tree.

The register files were represented by arrays of integers. This is an artifact of the ISS model and is planned to change to arrays of BIT.32 format to match the format of the data bus. The data bus is 64 bits wide. Data written to the bus can be directly loaded into the register files without conversion. This has the side effect of increasing the simulation kernel since BIT.32 requires more memory that its integer representation, however, the simulation speed should be increased. Both register files are of the same type allowing data to be transferred from one to the other (instructions $\text{iseq}$ and $\text{iseq}$ of the i860) without conversion.

II.B.(ii) Instruction Set Decoding and Execution

The i860 instruction set was partitioned along natural boundaries into four main categories. These include:

- REG-Format Instructions
- Core Escape Instructions
- CTRL-Format Instructions
- Floating-Point Instructions

Figure 6 shows the partition and the tree structure used to decode the opcodes of each instruction. Cuse statement constructs in VHDL are used to implement the decoding tree and obtain the opcode specified in the input instruction. The hierarchy consists of multiple stages where the first stage primarily decodes the REG-Format instructions (needs bits 31 down to 26) or determines if it

```
alias PM : BIT_VECTOR(7 downto 0) is PSR(31 downto 24)
alias PS : BIT_VECTOR(1 downto 0) is PSR(23 downto 22)
alias SC : BIT_VECTOR(4 downto 0) is PSR(21 downto 17)
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th></th>
<th>Variable Name</th>
<th>ADDER PIPELINE VARIABLES</th>
<th>OUTPUT VARIABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfadd.m</td>
<td>14,14,10</td>
<td>Rm3</td>
<td>Rm2</td>
<td>Rm1</td>
</tr>
<tr>
<td>pfadd.m</td>
<td>14,14,10</td>
<td>Rm3</td>
<td>Rm2</td>
<td>Rm1</td>
</tr>
<tr>
<td>shl.r/4,0</td>
<td>14,14,10</td>
<td>Rm3</td>
<td>Rm2</td>
<td>Rm1</td>
</tr>
<tr>
<td>pfadd.m</td>
<td>14,14,10</td>
<td>Rm3</td>
<td>Rm2</td>
<td>Rm1</td>
</tr>
<tr>
<td>pfadd.m</td>
<td>14,14,10</td>
<td>Rm3</td>
<td>Rm2</td>
<td>Rm1</td>
</tr>
</tbody>
</table>

Fig 7. Pipeline Processing Variables.

is another instruction type. If the instruction was not a REG-Format type, then the second stage is decoded and this continues to subsequent stages if required. The following list describes the bits used for decoding by each type of instruction format:

- **REG-Format** uses only bits 31 down to 26
- **Core Escape** uses bits 31 down to 26 and bits 4 down to 0
- **CTRL-Format** uses bits 31 down to 26 and since it is the same as REG-Format, it was chosen to be the **when others => option in the case statement**
- **Floating-Point format** uses bits 31 down to 26 and bits 6 down to 0 with subdivisions for the DPC field (bits 3 down to 0)

After decoding the opcode, the next step was to obtain the operands based on the decoding of the remainder of the instruction (source and destination registers, branch offsets, etc., ...). A call to a function or procedure is then done to implement the functionality of the instruction (i.e., Execution Stage).

**II.B.(iii) Pipeline Organization**

All pipelines were implemented using the same general scheme. Variables were used to improve simulation time efficiency. Pipelines, in general, contain the final result of the operation in their last stage and earlier stages contain partial results. In the current pipeline model, the calculation is done in the first stage since the source operands are known at that time. The result is then propagated through the remaining stages and is present at the final stage on the correct clock cycle. Figure 7 illustrates how variables are assigned on successive clock cycles when a pipelined add operation is in progress. The result is calculated in the first stage, and on each successive clock cycle when a pipelined add is present, the results are passed to the next stage variable. The final result is contained in PFAD_REAL.DOUBLE_RES and is available for use on the fourth operation of the add pipeline. This is possible through the bypass feature of the i860. One notices the instruction shl occurs in the middle of the sequence and has no affect on the pipeline stages. If a scalar add instruction was initiated at the end of this sequence, the results in the pipeline would be lost upon its completion since all scalar instructions must pass through each pipeline stage.

**II.B.(iv) Cache Modeling**

All caches were modeled using the VHDL record type. Figure 8 illustrates the data type named DATA_CACHE_TYPE and the code used to represent it. The caches were placed in different processes based on the required functionality of the specific process. The MMU contains the translation caches, the instruction fetch process contains the instruction cache and the data load/store element contains the data cache. Some handshaking is still required for example, when an instruction cache miss occurs, the MMU process must be initiated to load another instruction from external memory. The MMU may first have to do an address translation and could also have a cache miss. In this case, it would first go to external memory to load one of its page table entries prior to doing the address translation on the instruction address. The same set of events is possible for data cache misses.

**II.C Interface Model**

The interface model (IM) [5] simulates the interaction between the multiple components connected to the i860. To model the interface correctly, external elements such as memory, clock and reset generators and memory controllers must be developed to exercise all functionality. The memory controller must interpret the data or instruction transfer type for all processor operations that interface to the bus. It also sets the policy for specific transfer types. For example, the memory controller can disable cache writes for addresses in a specified range. The controller must also be
Fig 8. Data Cache Model.

capable of generating special signals such as interrupts to completely test the processor's interface behavior.

The timing diagram for instruction cache misses and various types of data load/stores is shown in Figure 9. In this figure, the system starts from reset (held high for at least 10 clock cycles) and begins processing instructions from a known address contained in the program counter (PC). The PC (0xFFFF_F000) is compared to the cache virtual addresses but due to reset, all elements have been invalidated so there was an instruction miss.

A load to external memory is initiated

\[ EXT\_CODE\_READ = 1 \]

and the cache fill occurs in the time frame from about 300 ns to 400 ns. Eight instructions can be seen in the instruction buffer (INSTRLBUFF) array as they are being loaded from external memory. They are transferred to the cache during the loading process.

The memory controller was designed to convert instruction load operations into cache fills by asserting the KEN_N signal during the first instruction fetch. The processor interprets this to be a cache fill and proceeds loading the additional three instructions into the buffer array. The processor releases the address bus after the first load at which point the memory controller takes control and generates the correct addresses for subsequent loads. The diagram also illustrates the processing of various load and store instructions as can be seen by the memory strobe activity (STROBE).

An example of the VHDL code contained in the decode/execute process used to initiate the load/store process is contained below.

```vhdl
INT_TO_BITS(REAL_ADDR, TEMP_32);
DATA_ADDR_SIG <= BITS_TO_INT(TEMP_32);
OP_SIZE := 8;
OP_SIZE_SIG <= OP_SIZE;
RYTE_NUM_SIG <= RYTE_NUM;
LOAD_STORE_START <= '1', '0';
after DEFAULT_TIME;
TRANSACTION_TYPE <= 4;
DATA_LOAD_IN_PROGRESS <= '1';
wait for 0 ns;
```

This code starts the load/store process by setting the LOAD_STORE.START signal to a '1'. The address, byte-enable information, operand size, and transaction type are required for decoding in the load/store process. The "DATA LOAD IN PROGRESS" variable is set for possible external read/write conflicts at a later time. The load/store process decodes the bus operation required based on the transaction type. The following code segment is used for illustration.

```vhdl
wait until (LOAD_STORE_START'EVENT and LOAD_STORE_START = '1') or (RESET'EVTENENT and RESET = '1');
```

case TRANSACTION_TYPE is
-- LD_8/I, LD_16_32/I,
-- FLD_32_64/I, PFLD_32_64/I
-- instructions
when 4 =>
  LEN <= '0';
  CACHE_N <= '1';
  MIU_N <= '1';
  DC_N <= '1';
  WR_N <= '0';
  PCYC <= '0';
  CTYP <= '0';
  case BYTE_NUM_SIG is
  when 0 =>
    case OP_SIZE_SIG is
    when 0 =>
      BE_N <= "11111111";
```
BE_W <= "11111100";
when 32 =>
BE_W <= "11100000";
when 64 =>
BE_W <= "00000000";
when others =>
assert FALSE
report "Illegal Op size in ldint, scyc, ldio, ld_8, fld"
severity warning;
end case;
INT_TO_BITS(DATA_ADDR_SIG,TEMP_32);
ADDRESS <=
To_StdLogicVector(TEMP_32(31 downto 3));
-- PCYC and CTYP are only
-- defined for memory
-- reads/writes
ADS_N <= '0','1' after CLK_PERIOD;
-- Trigger the external read process
EXTERNAL_BUS_OP <= '1';
wait until RISING_EDGE(CLK);
wait until BRDY_N'EVEN'T and BRDY_N = '0';
end case;
end case;

III. Testbench Modeling
Test and verification are important for obtaining a measure of the prototype's correctness and accuracy. A test bench must be devised to evaluate both the internal and external behavior of the unit. Also, a complete test plan must address the accuracy as well as modifications to the model.
An automated regression test suite is utilized to quickly evaluate the effect of changes to the model during development. This test suite is run using script files, and any deviations from expected responses are returned to help localize the search for problems.

The test bench for the 1860 FFM consists of the following components as shown in the right half of Figure 2:

- Memory
- Memory Control
- Clock and Reset Generators

### III.A The Memory Model

The memory model is implemented using a dynamic allocation scheme as shown in Figure 10. The VHDL `record` construct is used to define each memory segment. This structure implements a doubly-linked list where the first and last memory segments point in only one direction. The record contains fields for segment low/high addresses, an array containing the data and a set of pointers to the previous and next segments. The data array contains 4096 elements (stored as integers). When an address is presented to memory outside the range of the current segments, a new segment is allocated and the data, if required, is loaded from a file using TEXTTO. Routines were written to create new segments, reset memory and search

the memory segments for data. The naming convention used for the memory files is illustrated in Figure 11. The location placed on the address pins is used to form the file name required for loading.

#### III.B Memory Controller

The memory controller performs the majority of the bus interface test bench activities. It categorizes the outputs of the 1860 into three main groups: address strobe initiated events, bus arbitration activities and cache snooping protocol. There are also a few miscellaneous signals such as lock, reset and interrupt that must be exercised.

As an example of the address strobe initiated interactions, a nesting of case statements first determines the cycle type to be performed. If the type is a memory read or write, it then determines the cycle length of the transaction. The behavioral description takes the form of a "controller" as shown by the following code segment used for its decoding.

```vhdl
if (ADS_N = '0') then
  CYCLE_DEFINITION :=
    std2bit_vector(MIO_N & DC_N & WR_N, '0');
  case CYCLE_DEFINITION is
    when "000" => -- Interrupt acknowledge
    when "001" => -- Special cycle
    when "010" => -- I/U Read
    when "011" => -- I/O Write
    when "100" => -- Code Read
    when "101" => -- Reserved
```

---

Fig 11. Memory File Naming Convention.

7.19
when "110" => -- Memory Read
MEMORY_TRANSFER_TYPE :=
std2bit_vector(PCYC & CTYP & WR_N,'0');
case MEMORY_TRANSFER_TYPE is
when "000" => -- Normal read
when "010" => -- Pipelined load (PFLD)
when "100" => -- Page directory read
when "110" => -- Page table read
when "001" => -- Write-through (S-stage)
when "011" => -- Store miss or write-back
when "101" => -- Page directory update
when "111" => -- Page table update
end case;
when "111" => -- Memory Write
MEMORY_TRANSFER_TYPE :=
std2bit_vector(PCYC & CTYP & WR_N,'0');
case MEMORY_TRANSFER_TYPE is
when "000" => -- Normal read
when "010" => -- Pipelined load (PFLD)
when "100" => -- Page directory read
when "110" => -- Page table read
when "001" => -- Write-through (S-stage)
when "011" => -- Store miss or write-back
when "101" => -- Page directory update
when "111" => -- Page table update
end case;
end if;

III.C Clock and Reset Generators

The clock generator provides the synchronization mechanism for the various components and the reset initializes the system to a known state. Reset also causes a trap to occur in the i860 and the PC begins processing instructions at the address 0xFFEFF0 as seen in Figure 9. This loads the highest memory file MEM262143.

III.D Test Procedure

A test suite was developed to verify both internal and interface behavior of the processor. At least one test was developed for each instruction of the i860 by writing hand coded assembly and carefully reviewing the correctness of the simulation results before adding it to the test suite. Also a set of three application programs generated from compiled C code was added to the suite. Simulation results stored for later comparison included the internal register contents and a database dump of all the interactions that occurred on the interface pins. As the model was developed, changes to a segment of code required a complete regression run on the test suite. To save time, the entire procedure was automated using script files and the suite required 3 to 5 minutes to complete. The output report indicates which tests passed and failed.

Figure 12 outlines the test plan for doing this regression testing. Through the course of integration with the MCV9, various bug reports were submitted by external users of the model. These were inserted as shown in Figure 12, and additional tests were added to the suite to help detect this fault.

IV. VP IN SYSTEM SYNTHESIS

This section describes the procedure taken to integrate the various components of the MCV9 and IRST systems. The components were developed across multiple organizational boundaries (Lockheed Sanders, Hughes, Georgia Tech, and Motorola) and integration problems were mitigated with proper test bench development at each stage. A list of some of the lessons learned from this prototyping effort can be found in [6]. The tests implemented throughout the process and their resulting simulation times are presented in the following sections.

The i860XP, memory and VME models were developed at Georgia Institute of Technology’s DSP Laboratory through ARPA RASSP Techbase support, while the remainder of the MCV9 subsystem was created at Lockheed Sanders. Models for the IRST system were developed at Hughes Aerospace & Electronics Company, Lockheed/Sanders and Motorola. System hardware integration was un-
dertaken by Hughes.

A significant amount of man-hours may be required to develop these prototypes when existing VHDL models of the components or similar components are not available. Current ARPA efforts towards populating libraries with a set of COTS models should decrease development time for future prototypes. Processor models, for example, can take on the order of a man-year to create. This time should decrease for subsequent models due to reuse of common code and VHDL packages. It is hoped the chip vendors will, in the near future, provide models for their products to assist in virtual prototyping. We are also looking at mechanisms to automate model development.

Given a set of populated libraries, virtual prototypes can utilize existing models to rapidly test possible HW/SW tradeoffs in the initial design phases, and help determine a proposed architecture for a given application. After selection of an architecture, software design can be initiated on the virtual prototype of the chosen processor, even before any hardware is fabricated leading to concurrency in the design process. The code can be run to verify system performance as well as detect any design flaws caused by misinterpretations of interface requirements between components. Customer feedback can also be effectively incorporated into the virtual prototyping process ensuring final success.

IV.A The i860 Prototype

The i860 was tested as described in section III. Prior to integration with the MCV0, the entire test suite was run without any problems. The application code contained in the test suite was created by compiling C programs. Statistics were collected for the processor model running the application code. These statistics were used as baselines to study the effects on instruction processing speed as additional components were added to the model. A total of three different application programs were tested. The programs contained varying complexity and consisted of code for converting Fahrenheit to Celsius, FIR filtering, and Sobel edge detection on an image. The amount of wall clock run time for each application is listed in the following table along with instructions/cycle executed.

<table>
<thead>
<tr>
<th>Appl.</th>
<th>Wall Clock Run Time</th>
<th>Int. Simuln Time</th>
<th>Inst./sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail2Ccl</td>
<td>5.1 sec</td>
<td>35 μs</td>
<td>216</td>
</tr>
<tr>
<td>FIR</td>
<td>35 sec</td>
<td>280 μs</td>
<td>211</td>
</tr>
<tr>
<td>Sobel</td>
<td>70 min</td>
<td>31.8 ms</td>
<td>234</td>
</tr>
</tbody>
</table>

For the first test, the time was based on calculating the conversion values starting at 0 degrees and ending at 300 degrees in steps of 20. For the FIR filter with nine taps, an input vector of 64 values is filtered. The Sobel operator processed a 128x128 image with eight bits/pixel. The numbers were obtained using a Sun Sparc10 workstation with 128 Mbytes of memory. The times are subject to system loading and processor memory resources but were checked at various times of the day with minimal variation. There was also very little deviation in the amount of instructions executed per second, the average being 220. The fluctuations are dependent on the type of code being executed. For example, loads and stores to the external busses require more signal manipulations and hence take longer, however internal core instructions require minimal decoding and processing and simulate much faster.
IV.B The MCV9 Virtual Prototype

Upon verification of the i860XP model, the processor and memory model were ready to be combined with the remaining components of the MCV9. The i860XP model replaced the existing processor interface model provided by Mercury Computer Systems Inc. The remaining components included the CE-ASIC (Compute Element), crossbar (XBAR) switch and interlink model. The ASICs were represented as a combination of behavioral and gate level models, including LSI 100K primitives for particular boundary cells and macros. To enhance simulation performance these primitive calls were removed and replaced with their behavioral equivalent resulting in an increased simulation time by a factor of 10. All components of the final subsystem contained behavioral level descriptions. The shaded boxes in Figure 13 were connected as part of the MCV9 virtual prototype used for test purposes.

The MCV9 subsystem integration time required less than a week. This illustrates the importance of thorough testbench modeling. Packages were created to help convert BIT types to Std_logic and visa-versa to account for the multi-value logic system used at the module level. An entire structural representation of the MCV9 was created, but only components of interest were used in the design. All other components were left “open”. Most of the problems encountered consisted of understanding the Mercury architecture and fixing errors in the VHDL syntax where translators were used. The initial integration phase was to ensure that the all models reset to proper conditions, and interface from the i860 model to the CE-ASIC and local memory were correct. C programs were written to program key setup registers within the CE-ASIC and initialize certain module parameters. The following list of tests were run at this stage of the prototyping.

- Test 1: Reset module and initialize the MISCON register.
  Purpose: Set key ASIC and module parameters for MCV9 configuration.
- Test 2: Reading and writing to the CE-ASIC.
  Purpose: Test CE-ASIC handshake logic to i860XP interface logic. Ensure that the i860 model provides the proper signals to the CE-ASIC.
- Test 3: Memory Test
  Purpose: Test to make sure that the i860 model could access memory on the MCV9.

Once it was established that C programs could be written that successfully programmed the CE-ASIC, a XBAR (crossbar) was added to the simulation and new tests were developed. Following successful integration and testing of multiple XBARs, more complex programs were created to verify communications at the boundaries of the MCV9. The two major interfaces of the MCV9 module consist of the VME Bus and the RaceWay. A VME Bus model and Interlink model were added to the module level test bench to verify I/O connectivity and proper interface protocol.

To further test the capabilities of the MCV9 model multiple instantiations of the i860 processor model were created to pass data from one processing element to another. Figure 13 shows a diagram of testbench.

A summary of the tests performed at this stage are listed below:

- Test 1: i860 ⇒ XBARs ⇒ RaceWay ⇒ Interlink
  Purpose: This test demonstrated the ability to write data to the RaceWay from the i860.
- Test 2: i860 ⇒ XBARs ⇒ VME ⇒ VME Driver
  Purpose: This test demonstrated the ability to write data to the VME from the i860.
- Test 3: i860 ⇒ XBARs ⇒ i860
  Purpose: In this test 16 processors were instantiated to simulate an entire MCV9 module. The test was used to validate the passage of data from one processor to another.

Simulation runtimes are presented in the following table for the first two tests.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Instr/ sec.</th>
<th>Int1 Simul Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCV9 2 Interlink</td>
<td>7.5</td>
<td>1200 ns</td>
</tr>
<tr>
<td>MCV9 2 VME Driver</td>
<td>5.5</td>
<td>3000 ns</td>
</tr>
</tbody>
</table>

These numbers give an indication of current capabilities of simulating large virtual prototypes running application code. Diagnostic and test code provide a viable candidate since they are typically less compute-intensive to verify and test in a VP. Some methods for improving the simulation speeds are suggested in [7-9, 12], but our runtimes reflect commercially available tools. The following test code was used to configure the CE-ASIC transfer protocol and send data generated by the i860 software, through the three cross-bars (XBARs) and...
Fig 14. Timing Diagram for Raceway Write Cycles.

across to the interlink model.

```
typedef long* PT;
typedef long* MA;
#define CE_BASE 0xfffff000
#define MISC_REG (PT)(CE_BASE + 0xC10)
#define BASE_MEM (PT)(0x1FFFFFF0)
#define EM_PAGE_1_EXT_ROUTE (PT)(CE_BASE + 0xD14)

main ()
{
    *MISC_REG = 0x00000103;
    *EM_PAGE_1_EXT_ROUTE = 0x4A600000;
    *BASE_MEM = 0x0AAAAAAAA;
}
```

A listing of the corresponding assembly code for this application is shown below.

```
//75| *MISC_REG = 0x00000103;
    or 259,r0,r16 // 259 = 0x103
    st.l r16,-1008(r0)
//76| *EM_PAGE_1_EXT_ROUTE = 0x4A600000;
    orh 19024,r0,r19 // 19024 = 0x4A60
    st.l r19,-740(r0)
    //77| *BASE_MEM = 0xAAAAAAAA;
    or 43600,r0,r18 // -21846=43600u=0xaaaa
    orh 43690,r18,r18//-21846=43690u=0xaaaa
    or 65520,r0,r17 // -16 = 65520u = 0xffff
    orh 8191,r17,r17 // 8191 = 0xffff
    st.l r18,0(r17)
```

Figure 14 is the resulting plot of the output stored in a results database. The code first configures the CE-ASIC by writing the particular value to the address of the MISC_REG register. All writes use the assembly store long integer instruction. A route word is then sent to configure the cross-bars. Once the first cross-bar is configured, the data is output to the appropriate address corresponding to page one of the external memory map. The data can then be seen to propagate through each of the additional XBARs after being configured by the route word. This test verifies that the configuration hardware of the CE-ASIC and XBAR are properly routing the data. It also gives an accurate data transfer time for passing

7.23
information over the network to external devices which can later be back annotated into improved performance level models.

**IV.C The IRST Prototype**

The complete IRST system is shown in Figure 15. Upon completion of the MCV9 tests, the remaining components required for system level simulation were added. This included the data distribution front end card created by Hughes Aerospace and the video processor card created by Motorola [4]. Data could then be routed to the processors memory through the data distribution card and RaceWay network, processed by the MCV9 elements and finally passed again through RaceWay network to the video card. Control signals were sent over the VME to the video card.

The Virtual Prototype consisted of the shaded modules in Figure 15. The goals of the virtual prototyping at this level were two-fold. The first was to check for continuity and bus protocols across the major interfaces. Second, to develop a set of low level diagnostic tests that could be run and compared against the real hardware. All tests were developed and run on the MCV9 module. All code was written in Ada and compiled to the format that the i860 model required as described in section III(A).

Because the Sparc card was not modeled, simple programs were created for the MCV9 to control the data distribution and video output cards. Basic functions that were tested include, software reset, setting up configuration registers, and programming video sub-windows. The following list indicates the complete set of tests that were run at the system-level and some of the problems that were found with each test.

- **Test 1: Software RESET**
  Purpose: In most systems the importance of a RESET is underestimated, and even the most experienced designers overlook the possibility of potential errors. The first test was to ensure that all boards could be reset via software. Reset was applied to the boards by setting and clearing a bit in a control register via the VME bus. After the reset was applied, all registers were checked to ensure they were configured to the default value. Problems were encountered with programming the correct "route word" through the XBARs to the VME interface. Once this was corrected the tests executed with no errors.

- **Test 2: VME register test**

Purpose: Software was written to write and read all of the registers on the data distribution and video output cards via the VME bus (modeled at Georgia Tech). If all tests passed, software would write a known pattern (A5A5A5A5) to the same location in local memory repeatedly. If the tests failed then software would spin executing NOP instructions. These basic loops made it easy to determine quickly if the test passed or failed. Errors were found with the VME handshake for A32 D16 mode. These errors were corrected and the test executed with no errors.

- **Test 3: RAM test**

Purpose: This test was used to write and read certain portions of the RAM on the Video Output card. First, a block of 128 locations was written and then read back. Again, simple loops were used to determine if the tests passed or failed. This test failed and uncovered a significant design flaw with the VME logic on both the data distribution card and the video output card. A design engineer misinterpreted the VME specification and did not use A1 and A2 as part of the address decode. Therefore, only locations capable of access by the MCV9 were 32-bit locations, and this precluded any 16-bit locations. To fix this error a hardware design change was mandated, including a relocation of the address of all the VME control registers. Finding this problem
before the boards were to go to fabrication not only saved debug and rework time, but also allowed the software to remain unchanged. If this error had been discovered on fabricated hardware, software code would have been re-worked to "fix the hardware error". This results in the so-called "spaghetti code" that is endemic to large software/hardware integration projects, where hardware and the software teams operate with little knowledge of each other's efforts. In our IRST system, this error was fixed and re-run with no errors. Also the VME register test had to be re-run to account for the fact the address map changed.

• Test 4: Floating Point RAM test
  Purpose: This is the same as the RAM test mentioned above. It only tested a different section of RAM. No errors were encountered running this test.

• Test 5: Interrupt tests
  Purpose: Various tests were run to force interrupt conditions either by hardware or software to make sure that the hardware and software respond correctly. These tests included:
  - FIFO overflow interrupt,
  - Data overflow interrupt,
  - Beginning of frame interrupt,
  - End of frame interrupt,

Figure 16 indicates metrics collected for the above tests. They were used to quantify the amount of resources required for the various tests on the Virtual Prototype. When running in batch mode with the \(-s t s\) option, the Vantage Spreadsheet simulator records statistics, such as the number of components, signals, processes, drivers, transactions, events, and deltas for a simulation run. It also measures elaboration and simulation CPU time, transactions per CPU second, and events per CPU second. The real time and percent CPU utilization were gathered using the UNIX \textit{time} command. The memory required for the simulation and disk space utilized for the specified simulation times and signals archived were also recorded. As shown, these tests required significant CPU hours to complete and were run on a system consisting of 13,257 components, 23,611 processes, 75,336 signals, and 46,591 drivers. There were 1,048 signals archived for most of the simulations. The i860 model was executing approximately 4.1 instructions per CPU second in this system environment. The design iteration time required on the order of a two days to complete, one day for simulation and one day for verification, and if necessary, redesign. Methods for improving performance of event-driven simulators are also of interest, and preliminary approaches can be found in [7-9, 12].

In the SW/virtual HW integration phase, 5 ADA programs (about 18,718 lines of source code) were ported onto the VP and 46 errors were found and corrected. Of these, 3 were hardware errors, 15 were integration errors, 5 were software errors, and 28 errors resulted from improper VHDL model integration. The turn-around time for correction (including resynthesis of HW from VHDL) of each error was approximately 19 hours. Since no hardware was fabricated, the detection and correction of these errors resulted in considerable savings in cost and resources that would otherwise have been incurred. It may be reiterated that the IRST Virtual Prototype took \textit{less than 3 weeks} to integrate and test.

V. SUMMARY

This paper documents a very promising new approach to system synthesis called virtual prototyping. Various stages of the virtual prototyping process were discussed for the development of the IRST signal processing system. The positive features of virtual prototyping in system synthesis include the following:

1. The system was successfully integrated and tested on the first attempt. Final hardware
integration took only three weeks. This number may be compared with the typical hardware/software integration times of 10-24 months required for large scale projects in current practice (circa 1993).

2. All functions that were tested during simulation worked the first time. Errors were found in areas that were not simulated, such as the voltage regulator and analog circuits.

3. Numerous software system level tests were implemented on the Virtual Prototype.

4. The software learning curve was shortened because code had been running on the virtual prototype weeks prior to actual hardware integration.

5. Software code was much "cleaner" and more portable since there were less changes to account for hardware design errors.

6. A documented description now exists for the system, so future upgrades can be done more rapidly by replacing only those elements that are modified.

There remains room for some improvements:

1. The simulation time can be exceedingly long for application code or operating system code running on the prototype. As a result, an entire 192-processor simulation running application code was not possible due to resource constraints, and the system had to be partitioned into smaller manageable chunks. Mixed-level of simulation models combining higher-level (token-based) performance models with some lower-level FDMs may be useful.

2. There is always the question of "how much simulation is enough" before the hardware can be fabricated with an acceptable degree of confidence. At the present time, it is up to the design engineer to decide (empirically) if the simulations have produced enough results to reduce risks of fabrication.

3. There is a question as to whether fully functional processor models or partially functional interface models are more suitable for virtual prototyping. The latter, clearly, have a performance advantage at the loss of some accuracy. Very long simulation times can be a show stopper.

With advances in the understanding of virtual prototyping for system synthesis, and with increased availability of validated (and interoperable) VHDL models of components, further gains in rapid prototyping will be achieved. In addition, with advances in multi-threaded simulation and hardware accelerators, system level simulation speeds will increase. Research projects funded by ARPA as part of the RASSP project, in addition to those at Georgia Tech and Lockheed Sanders, plan to continue to explore these issues in VHDL based system synthesis. It is expected the semiconductor industry will also assist in the progress in system synthesis by making available detailed VHDL models for their COTS components. Standards for ensuring model interoperability and reuse are also essential for RASSP's success [5].

References


[11]. The RASSP Information Server
URL http://rassp.scra.org


Questions on this article may be addressed
to Prof. V. Madisetti at vkm@ece.gatech.edu, fax (404) 883-9171.