Automated Generation of Accurate VLSI Behavioral Processor Models for Simulation and Synthesis

Yong-kyu Jung [1], *Vijay K. Madisetti* [1], *John W. Hines* [2]
[1] ECE - Georgia Tech, Atlanta, GA 30332-0250
[2] U.S. Air Force, WL/ELED Wright Patterson AFB, OH 45433-7319

Proceedings of *Second Annual ARPA RASSP Conference*, Crystal City, VA, July 25-27, 1995.

Abstract

A new process for automating the creation of Full-Behavioral (FBM) and Instruction Set Architecture (ISA) models in VHDL for complex processors and components is described, with results from the automation of a PowerPC 601 described in some detail. A number of advantages to this approach are described together with its impact on the hardware/software codesign and system prototyping processes.[1]

1 Introduction

The Rapid Prototyping of Application-Specific Signal Processors (RASSP) project of the US Department of Defense (ARPA and Tri-Services) targets a 4X improvement in the design, prototyping, manufacturing, and support processes (relative to current practice). As per E/F current practice (1993), the prototyping time from system requirements definition to production and deployment, of multiboard signal processors, is between 37 and 73 months [8]. Out of this time, $25-49$ months is devoted to detailed hardware/software (HW/SW) design and integration (with $10-24$ months devoted to the latter task of integration). With the utilization of a promising top-down hardware-less codesign methodology based on full-behavioral VHDL models of HW/SW components, it appears feasible that the HW/SW integration time could be reduced to a few weeks ($1-2$ months) [10]. Potential show-stoppers lie in the limited availability of high quality VHDL behavioral models of components (timing and function). In addition, the time to build a single model of a complex RISC processor (such as i860XP) is approximately a man-year. We describe a mechanism via which full-behavioral models of complex components can be automatically generated in VHDL from published information available in data manuals. This method could also be used in the iterative design synthesis of custom pipelined processors for domain/application-specific applications.

[1] This research was supported by Advanced Research Projects Agency (ARPA), Department of Defense, under the RASSP program, 1994-1997.

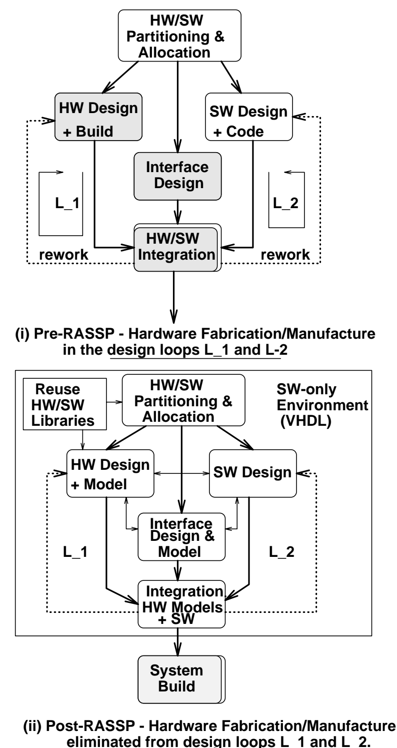

Fig 1. **HW/SW codesign - (i) Current practice (1993-1994) and (ii) True HW/SW codesign. Note elimination of hardware fabrication, assembly and board/system level manufacture from the design loops. Software can be tested on virtual hardware that is also concurrently being designed. Savings in time and cost, capability for customer input, and concurrent life-cycle support and upgrade planning is facilitated. Shaded areas imply hardware (board/MCM level).**

2 HW/SW codesign practice

The Educator/Facilitator current practice (1993) model for signal processor design is presented in detail in [8] in this proceedings. The various stages in a "waterfall"-type process flow are demarcated together with time ranges (min, max) for each stage. The time lines have also been validated via communications with the industrial entities involved in large system design and implementations. In this paper, we focus on the specific tasks of hardware, software, and interface design and their eventual integration.

2.1 Whither true codesign ?

True HW/SW codesign allows both hardware and software to be designed within a common framework, and simulated together before being fabricated. Current practice attempts to automate this process via HW/SW/Interface partitioning followed by three individual paths to HW, SW and Interface design and implementation, respectively (as shown in Figure 1). A drawback with this approach is that software can be designed and tested only if a hardware platform (at board and rack levels) is available. The latter is time- and cost-consuming (even if it utilized FPGA technology or HW modelers). It must be understood that the software is not just application-specific software, but also control, diagnostic and test software. Often, control, diagnostic, and test software requires an order of magnitude larger man-hour effort than does application software [8]. Conventional hardware software co-design methods assign a token interest in the issue of software required for control, diagnostic and test purposes, and attempt to catch all integration issues under the term "interface". The approach shown in Figure 1(ii) represents a "true" HW/SW codesign wherein software models (in a HDL such as VHDL) of HW are provided to the SW developers and the entire software is designed and tested and integrated with the HW models long before any hardware is fabricated or manufactured. Thus, the design loops L_1 and L_2 are quick, and require no hardware fabrication & engineering cost, and in addition provide capability for complete system design using a process known as *virtual prototyping* [1,5,10,11].

Virtual prototyping in top-down VHDL based design involves transition of the design through a number of abstraction levels, each of which represents an executable specification of the signal processor to be designed. At the network performance-level of virtual prototyping very little of the functionality (if any) of the target system is modeled, and the focus is on the utilization and efficiency of the SW implementation on a candidate HW architecture. Since this design takes place at a high level of abstraction, single events can represent functional blocks of a few tens to a few thousands of HOL instructions. The loss of functional detail is compensated by high simulation speeds (100,000-1,000,000 instructions/second) that can assist in the rapid architectural evaluation and selection. When limited functionality is added to performance models, additional estimates can be obtained about HW/SW functionality and timing (at the coarse level of wall-clock seconds) of prototype. Simulation speeds are reduced consequently (10,000-20,000 instructions/second). At an even higher accuracy of system modeling and simulation, with $bit-true$ functional and $clock-cycle$ level timing resolution, the use of ISAs and FBMs is recommended. This increase in accuracy is traded-off with slower simulation speeds (10-1000 instructions/second).

2.2 Showstoppers

The assumption, of course, is that libraries of full-behavioral HW models in SW are available, accurate, and interoperable, and that simulation times can be kept manageable. VHDL can be used with advantage in this true HW/SW codesign philosophy — one that embraces a hardware-less system design. Recent experience with hardware-less HW/SW codesign has shown that it is efficient, often reducing time for HW/SW integration to a matter of weeks, and also allows rapid upgrades, together with savings in cost [9]. Once virtual prototyping is completed, it is expected that that pathway through which a field prototype can be manufactured, supported and upgraded will be straighforward.

3 Modeling for HW/SW codesign

Several classes of models have been found suitable for HW/SW codesign. When emphasizing HW/SW integration, two classes have been found particulary useful - ISAs and FBMs. We will utilize the RASSP taxonomy [6] to define these two classes.
(1) *Instruction Set Architecture (ISA) Models* — An ISA model describes the function of the complete instruction set recognized by a given programamble processor, along with (and as operating on) externally known register set and memory/input-output space. An ISA model will execute any machine program for that processor and give exactly the same results as that processor (e.g., bit-true) as long as the initial states are the same for both simulation and the real system. Port registers, if modeled, are also bit-true. Instructions span multiple clock cycle, and ISA models need not contain any internal structural implementation information.
(2) *Full-Behavioral Models (FBMs)* — A FBM (also known as full-functional model [10]) is a processor model that exhibits all documented timing and functionality of the modeled component, without specifying internal structural implementation details. Thus, the full-behavioral model is more detailed than the ISA model in that it includes clock-edge timing information in addition to functionality. A number of full-behavioral processor models are available from Georgia Tech's RASSP Techbase effort [5]. The issue of creating ISA and FBMs will be examined next.

3.1 Populating VHDL model libraries

While complete or incomplete gate-level VHDL models are sometimes available from vendors, and are

accurate for use as ISAs and FBMs, a number of limitations exist — (1) gate-level models are very slow in terms of simulation times, (2) reveal confidential component design (intellectual property) information, and (3) HW/SW codesign assumes that the hardware component is continuously being designed (e.g., changing instruction set, optimized behavior, etc), and thus the gate-level description does not exist. Thus, the focus is on creation of behavioral models of complex parts. Commercial Instruction-set simulators (ISS), which can provide debug information for processors, have limited applicability within a VHDL-based environment (without wrappers and loss in efficiency) where multiple models at varying levels of detail are co-simulated during the top-down design process. In addition, they do not allow redesign of the hardware component, a trend that is increasing finding favor in application-specific markets (e.g., use of core-based functional design of DSP ASICs [2]).

The current approach to model development is best described in [3,4,10]. All these approaches model the internal and external microarchitecture of the component behaviorally from manufactured-supplied data (or via abstraction to higher levels of functional and timing information from gate-level descriptions). This is a manual, time consuming (in man-years), and error-prone (i.e., verification) operation, and often equivalent to designing the component all over again. While we have used this approach, and continue to use this approach in developing ISAs and FBM models, an investigation into automated generation of these models was long overdue.

3.2 A new approach - autogeneration

An alternative approach to developing ISAs and FBMs that is automated is described in Figure 2. The processor being modeled (or designed) is described by parametrized generalized time-stationary [2] pipelines (single or multi-), associated memories/registers, and a generalized controller. The user-defined or vendor-supplied information on the instruction set, architectural constraints (hazards, timing), are captured in terms of processor-specific input data files. These parametric input data files then are automatically converted to lookup tables (LUTs). The LUTs are utilized by the AMG to generate the control (timing) and functional information from the input application instruction stream. We have used this approach to synthesize behavioral models of the PowerPC 601 RISC processor and an implementation will be described in the next section.

3.3 A new approach - iterative synthesis

The approach described in Figure 3 describes the process flow for automating the iterative synthesis of application-specific processors. Here the instruction-set of a programmable processor can itself be customized and iteratively designed during the HW/SW codesign process. The application drives the iterative instruction-set and architecture codesign (which are captured from input data files as LUTs) by the AMG. The controller, pipeline, and associated logic of the AMG are then simulated to measure performance on the target application. After optimization of the instruction set and timing, the AMG may be synthesized using commercial RTL-level or behavioral synthesis tools. Application-specific functional libraries can also be used with advantage when combined with VHDL and the emerging VITAL standards for sign-off quality timing simulation. Future papers will discuss and document the approach of Figure 3.

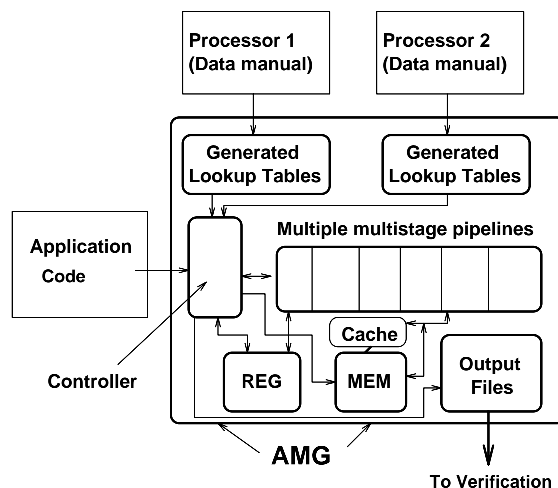

Fig 2. **Automated Processor Model Generation (AMG) for Simulation**

4 Automated Model Generator - AMG

The automated model generator (AMG) is an ISA or FBM model that accepts the application instruction stream and processor-specific data in the form of input tables, that are processed internally to provide all documented functional and timing characteristics as output files. The same AMG can be reused for creating models of multiple versions of the same chip, or independent families of processors.

4.1 Structure of the AMG

The automated ISA model generator consists of six major "blocks", as described below (See Figure 4).

1. **Pipeline:** A single pipeline for a RISC processor consists of the following six stages — (1) Instruction Fetch (IF), (2) Instruction Dispatch (IDP), (3) Instruction Decode (ID), (4) Instruction Execute (IE), (5) Cache Access (CA), and (6) Write back (WB). These stages were implemented as procedures within a VHDL process description of the pipeline.
2. **Memory Block (MB):** The MB consists of an *instruction queue (IQ)*, *instruction and data memories (IM & DM)*, and a *cache (CACHE)*.

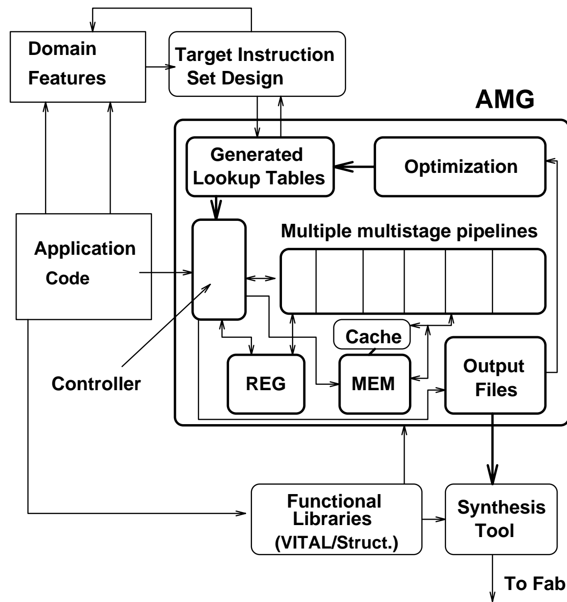

Fig 3. **Automated Processor Model Generation (AMG) for Iterative Synthesis**

3. **Data Register Block (DRB):** The DRB consists of a number of register arrays (DER, ECR, CWR), including a general purpose register (GPR) to allow storage for resolution of pipeline data hazards. A number of 32-by-32 bit data register arrays are also reserved for the user.
4. **Control Register Block (CRB):** The CRB consists of register arrays (CR, HIR (hazard information registers), SCR (system control registers), HDR (hazard destination registers)) to control various stages of the pipeline.
5. **System Generating Logic (SGL) Block:** The SGL converts specific input data (i.e., in form of tables.dat) from manufacturer or instruction-set designer into Lookup Tables (LUTs) that can be used by the AMG. Thus, information about differing processors can be converted to a standardized internal representation that can then be utilized by the AMG in generating instructional function and timing. The six automatically generated internal LUTs are opcode lookup table (OPLUT) containing opcodes and extended opcodes for user-defined instructions, a decode lookup table (DCLUT) containing information on the bit length of the opcode and other instruction fields, an execute lookup table (EXLUT) that stores information for the execution latencies and the identification of every instruction to map into an executable location in the IE, a hazard lookup table (HLUT) containing information on data hazards of registers and memory, an extended opcode lookup table (EOLUT) consisting of data related to extended opcodes, and a system generation lookup table (SGLUT)

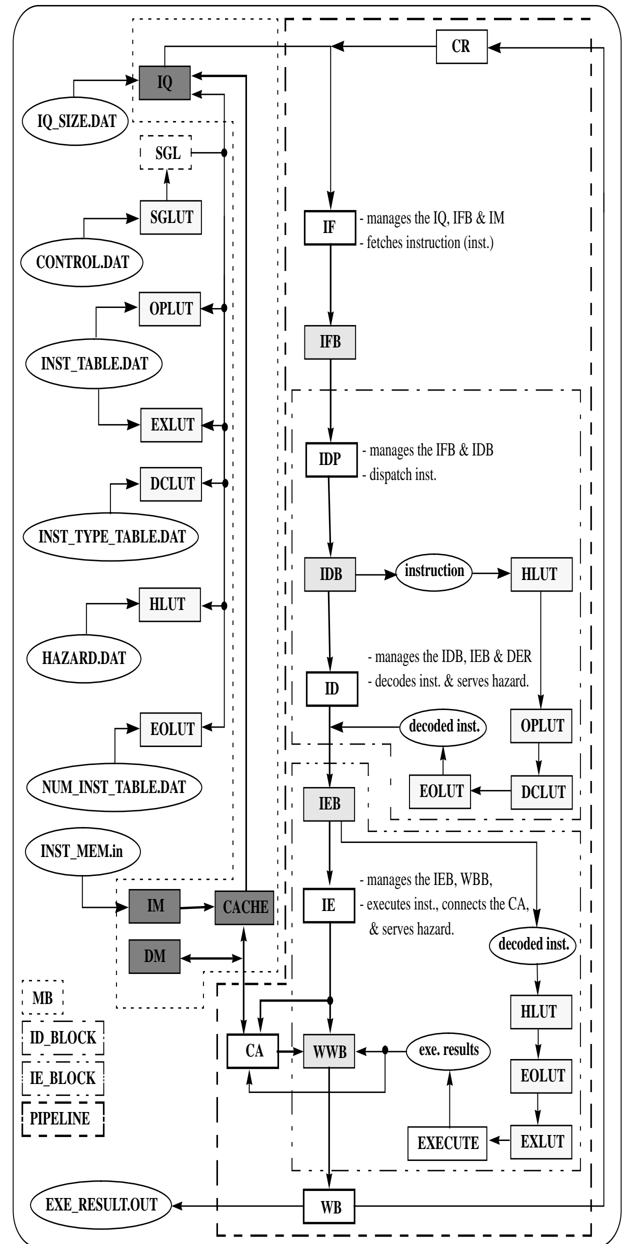

Fig 4. **The Anatomy of Georgia Tech's Automated VHDL Model Generator (AMG). Hardware vendor or designer-supplied data is obtained as *.DAT, which are then automatically converted to lookup tables (LUTs) by the AMG. The output file is the behavior of the hardware executing the software (functional values and timing).**

that is used by the SGL. It may be iterated that the SGL automatically creates these LUTs based on manufacturer or designer-supplied processor or instruction-set information.

6. **Stage Buffer Block (SBB):** The SBB consists of buffers for stages of the pipeline (e.g., IFB, IDB, WBB, IWB, etc).

4.2 Operation of the AMG

We now discuss the operation of the AMG as follows:

1. **Step 1 (Fetch and Dispatch):** An instruction if fetched from the IM and brought to the IQ and CA in the pipeline. The IF fetches the instruction from the IQ and stores it in the IFB. The IDP then initiates the dispatch of the instruction from the IFB and translates it into the IDB. In order to decode this instruction, the opcode or the extended opcode is first extracted from the instruction. The instruction type is then docoded from the information available in the lookup table OPLUT.

2. **Step 2 (Decode):** The ID then obtains the extended opcode information from the EOLUT and the instruction format from the DCLUT using the decoded instruction type information as a key. In the final step at the ID, the instruction is dissassembled, the information disseminated, and valid instruction fields are stored in the IEB. After completing the decode operation, the ID checks for data dependency on the current instruction. The information stored in DER is utilized for this check, and the information is propagated to the hazard registers, HIR and HDR, with operate in conjunction with the HLUT. Operands for the operation are put in the IE buffer (IEB)

3. **Step 3 (Execute):** The IE begins operation if the IEB is nonempty. The IE updates the GPR and the DER, and picks out appropriate information from the EXLUT — i.e., instruction execution latencies, location of procedures, requirements for cache access for executing the function or process, and then executes the procedure (the AMG currently supports upto 1024 user-defined operations). The result is then stored in the WBB or sent to the CA (if cache access is needed). The HIR and HDR are then updated to allow hazard resolution for the subsequent instructions in the pipeline.

4. **Step 4 (Cache Access and Write Back)**: The CA then reads/writes data from/to the cache in case of a cache hit, or the DM in case of a cache miss. The WB updates the CWR through the DER or ECR before writeback. If the instruction is processed by CA, the CWR is updated by the ECR, else, it is updated by the DER (resolving hazards between IE and CA). The result is then written to the GPR and all hazard conditions caused by the current instructions are void. The WB also generates the *output file* with the necessary user-specified information on execution times and functional results required from the model.

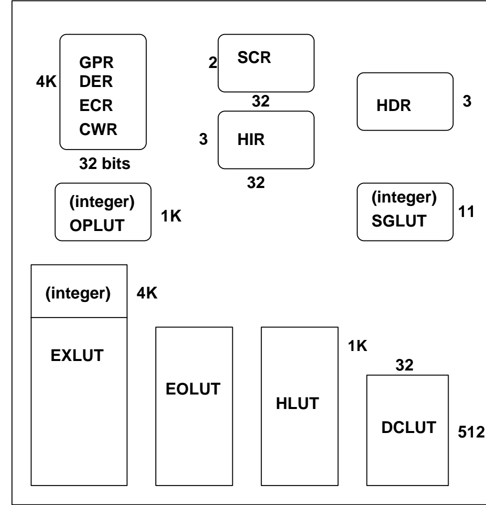

Fig 5. **Register and Memory Data Structures**

4.3 Implementation of the AMG

To test the AMG we first implemented the AMG in VHDL, and successfully modelled a subset of the ISA of the PowerPC 601 with a single pipeline. More recently, the AMG has been generalized to model multiple concurrent pipelines and other processors (e.g., i860 and ADSP 21060).

In one of our PowerPC 601 variations of the AMG, that is fully operational, each memory within the MB was implemented as a 32 bit-vector array (same as the instruction length). The IQ IM, and DM are 64-by-32, 8K-by-32, and 20K-by-32 arrays, respectively. The SBB was implemented as four buffers, one of which is the IEB that is a 256-integer variable buffer for maintaining latency and executing function information in the IE stage, the others maintain bit-vector and one integer type variable for maintaining the latencies of other pipeline stages. Figure 5 summarizes the sizes of the other register and memory arrays utilized in our implementation. Note that the user of the AMG can tailor the pipeline to suit his/her implementation specifications, and can also utilize more than one pipeline within the AMG (i.e., the PowerPC 601 has three pipelines — integer, floating, and branch). The AMG currently has been implemented in about 5K lines of uncommented VHDL source code.

4.4 Performance of the AMG - PowerPC 601

Figure 6 describes the performance of a PowerPC 601 model generated by the AMG. The input source code is described in Test Bench A, and was input to the AMG. The AMG then generates the function and timing behavior via output files (shown also in Figure 6), and via VHDL *signals* (that are displayed on a VHDL

simulator spreadsheet in Figure 7). Tables 1 and 2 in Figure 6 describe the detailed clock-cycle resolved operations of the pipeline for the PowerPC 601. The exact timing for the completion of each instructions are also shown. In Figure 7, for instance, the multiply is described in the *decode* buffer as $7c4118d6$, and has a latency of 5 clock cycles, which are successively decremented as shown on the signal INST.EXE.CYC.1. Typical instructions executed per second on the virtual model generated by an unoptimized AMG were in the order of $500-1000$ for single pipelines, and less for multiple pipelines $(10-200)$. For a 1000-instruction test bench, the execution times on a SPARC-10 workstation were; multiple pipeline AMG (242.95 sec), PowerPC with multiple pipelines (235.55 sec), Single pipeline AMG (18.0 sec), PowerPC with single pipeline (1.45 sec). The time required to generate a model is limited only by the time it required to enter the input.DAT tables from the manufacturer's data sheets (or in the case of iterative synthesis, from the designer), and took about a man-month for the PowerPC. The AMG consists of about 5K lines of VHDL source code and utilized the Vantage VHDL Spreadsheet at Georgia Tech's DSP Laboratory.

5 Summary and Conclusions

Models have been shown to very useful in the system prototyping process, often reducing HW/SW design and integrations costs by a factor of four or more. The contributions of this paper are as follows -

1. A new method for automated generation of full-behavioral and ISA models for complex pipelined processors has been proposed. We believe that this is the first such proposal and its implementation.

2. A new method for iterative synthesis, where the instruction-set of a processor can be customized to the application software, utilizing true hardware/software codesign is proposed.

3. Successful demonstration of the proposed method for automated generation, using the PowerPC 601 as an example. Our results show that the speeds in instructions per second range between $500-100$ for auto-generated single pipelines and $5-100$ for multiple pipelines, and compare well to manually generated behavioral models. The time required for model development is, however, much shorter, requiring $2-3$ man-months for an ISA model (without interface timing), as opposed to $1-2$ man-years for the manual method of model generation.

   Further optimization of the automated model generation process is an ongoing investigation.

Acknowledgements

Thanks to M. Rubeiz of Wright Patterson Labs (USAF) for carefully reviewing the manuscript.

References

[1] M. Richards, "The Rapid Prototyping of Application-Specific Signal Processors Program," Proc. of *First Annual RASSP Conference*, August 1994.

[2] V. K. Madisetti, *VLSI Digital Signal Processors,* IEEE Press, Piscataway, NJ, May 1995.

[3] Z. Navabi, "Using VHDL for Modeling and Design of Processing Units," Proc. of *5th Annual IEEE ASIC Conference and Exhibit*, pp. 315-326, 1992.

[4] L. Maliniak, "Process Builds Accurate VLSI Behavioral Models," Electronic Design, pp. 63-70, May 3, 1993.

[5] V. Madisetti, T. Egolf, S. Famorzadeh, L-R. Dung, "Virtual Prototyping of Embedded DSP Systems," Proc. of *IEEE ICASSP 95.*

[6] C. Hein, T. Carpenter, P. Kalutkiewicz, V. Madisetti, "RASSP VHDL Modeling Terminology and Taxononomy - Revision 1.0," Proc. of *Second Annual ARPA RASSP Conference*, July 1995.

[7] C. Myers, R. Dreiling, "VHDL Modeling for Signal Processor Development," Proc. of *IEEE ICASSP 95.*

[8] V. Madisetti, J. Corley, G. Shaw "RASSP: Current Practice (1993) E/F Model and Challenges," Proc. of *ARPA Second RASSP Conference*, July 1995.

[9] The RASSP Information Server - WWW URL http://rassp.scra.org.

[10] T. Egolf, V. Madisetti, S. Famorzadeh, P. Kalutkiewicz, "Experiences with VHDL Models of COTS RISC Processors in Virtual Prototyping for Complex System Synthesis," Proceedings *VHDL International Users' Forum (VIUF), Spring 1995.*

[11] C. Hein, D. Nasoff, 'VHDL-Based Performance Modeling and Virtual Prototyping," Proc. of *Second Annual ARPA RASSP Conference*, July 1995.

Description of the Instruction Timing & Test Bench

TABLE 1. Instruction Timings for the Test Bench 'A'

Instruction Cycles: 1 2 3 **4** 5 6 7 8 **9** 10 **11** **12** 13 ~ 46 47 **48** 49 **50** **51** 52 **53** 54 ~ 87 88 **89** 90 **91** **92**

Addr. of Inst.	Inst.Name	rD/BI	rS/rA/rB	Instruction Cycles
0	"add"	rD=1	0 / 2 / 3	f d e **w**
1	"mulx"	rD=2	0 / 1 / 3	f d e e e e e **w**
2	"ldx"	rD=3	2 / 0 / 0	f . . . . d e c **w**
3	"add"	rD=2	0 / 3 / 3	f . d e **w**
4	"divx"	rD=1	0 / 2 / 3	f d e e ~ e e **w**
5	"strx"		3 / 1 / 2	f . . ~ . d e c **w**
6	"brx"	BI=3		~ f d . e **w**
7	"addi"	rD=2	0 / 2 / 0	~ f . >
3	"add"	rD=1	0 / 3 / 3	f d e **w**
4	"divx"	rD=1	0 / 2 / 3	~ f d e ~ e e **w**
5	"strx"		3 / 1 / 2	~ f . ~ . d e c **w**
6	"brx"	BI=3		~ ~ f d . e **w**

"**f**" : Instruction fetch & dispatch stage
"**d**": // decode stage
"**e**" : // execute stage
"**c**": // cache access stage
"**w**": // writeback stage
">": purge Instruction on the pipeline
"**.**" : stall on the pipeline

TABLE 2. Characteristic of the instructions

Instruction name	Opcode	Extended opcode	Instruction Type	Latencies
add	31	248	1	1
addi	14	0	2	1
ldx	34	0	3	1
strx	31	215	4	1
mulx	31	107	1	5
divx	31	331	1	36
brx	18	0	5	1

TABLE 3. Description of the instruction fields

Instruction Type	rD	rS	r1	r2	BI	C1	Eop	SIM	UIM	RC	AA	C2	RSV
1	5	0	5	5	0	1	9	0	0	1	0	0	0
2	5	0	5	0	0	0	0	16	0	0	0	0	0
3	5	0	5	0	0	0	0	0	16	0	0	0	0
4	0	5	5	6	0	0	9	0	0	0	0	0	1
5	0	0	0	0	24	0	0	0	0	0	1	1	0

rD : Destination Register
rS : Source Register
r1 : working Register 1
r2 : working Register 2
BI : immediate field for branch
C1 : control bit 1
Eop : extended opcode
SIM : signed immediate field
UIM : unsigned immediate field
RC : record bit
AA : absolute address bit
C2 : control bit 2
RSV : reserved bits

TEST_BENCH 'A' (INPUT)

```
8                                   :end addr.+1
                                      of inst. stream
0                                   :start addr.
                                      of inst. stream
01111100001000100001100111110000    :"add",  rD=1
01111100010000010001100011010110    :"mulx", rD=2
10001000011000100000000000001111    :"ldx",  rD=3
01111100010000110001100111110000    :"add",  rD=2
01111100001000100001101010010110    :"divx", rD=1
01111100011000010001000110101110    :"strx", rD=1
01001000000000000000000000001100    :"brx",  BI=3
00111000010000110001100110011110    "addi",  rD=2
```

TEST_BENCH 'A' (OUTPUT)

```
 4  :end of cycle time   "add",  rD=1
 9  :      //            "mulx"  rD=2
11  :      //            "ldx",  rD=3
12  :      //            "add",  rD=2
48  :      //            "divx", rD=1
50  :      //            "strx", rD=1
51  :      //            "brx",  BI=3
53  :      //            "add",  rD=2
89  :      //            "divx", rD=1
91  :      //            "strx", rD=1
92  :      //            "brx",  BI=3
```

Fig 6. **Results from the AMG-generated PowerPC 601 model**

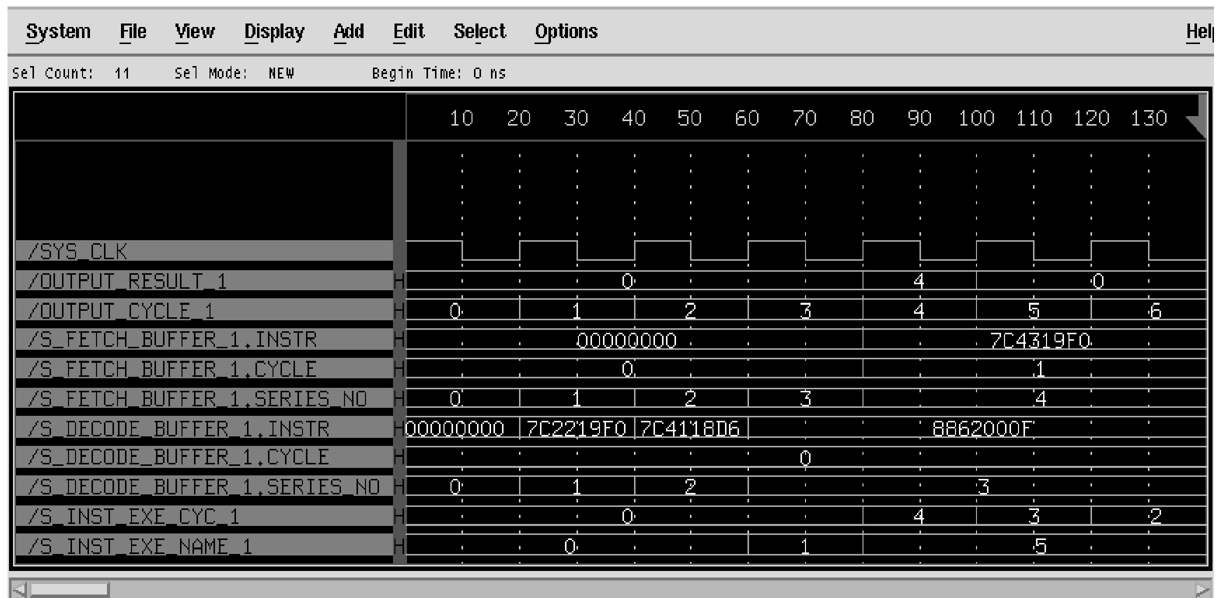

Fig 7. **VHDL simulations of the AMG-generated PowerPC 601 model using testbench of Figure 6 confirm the behavior of the processor.**