Design Space Exploration using T&D-Bench

Sandro Neves Soares
PGCC – UFRGS / UCS – CARVI
snsoares@ucs.br

Flávio Rech Wagner
Institute of Informatics – UFRGS
flavio@inf.ufrgs.br

Abstract

This paper presents T&D-Bench - Teaching and Design Workbench, a software infrastructure for modeling and simulation of state-of-the-art processors. It combines features that simplify and accelerate the processor design process without restricting the designer possibilities, thus representing a good trade-off for educational and research purposes that is not found in other environments. In T&D-Bench, a new model is constructed by the designer using a script language to define micro-architecture, instruction set, and timing aspects of the processor. These scripts can be produced by a graphical front-end, and a Java simulator targeted at the modeled processor is automatically built from the scripts. This approach can fit well the requirements imposed by the educational environment. Fine-tuning adjustments or the description of more complex processor mechanisms can be achieved by means of modifications in selected parts of the software infrastructure.

1. Introduction

Hardware designers are facing enormous challenges at all stages of the design process, due to increasing system diversity and complexity. Nowadays, many classes of processors coexist, like RISCs, VLIWs, DSPs, and microcontrollers. Furthermore, the system-on-chip technology emerged, integrating components such as processors, co-processors, memory systems, and interfaces in a single chip. Thus, the choice of the right component or set of components to satisfy the design requirements becomes a very complex task to designers working in industry. As a consequence of this situation, design space exploration is now an integral part of the design flow of modern computing systems, especially for high performance microprocessors [1]. Design space exploration is achieved by means of powerful software environments employed by designers to model and simulate the design alternatives. Such environments include those based on description languages (Architecture or Hardware Description Languages) and performance simulation tools. Since the use of these languages or software infrastructures requires steep learning curves from the designers, it constitutes a problem when short time frames are mandatory to develop and deliver new products, in industry, or to obtain and deliver new results in research.

ADLs, HDLs, and performance simulation tools do not have educational purposes as their primary goals, but they are all used in educational environments. In this context, the required steep learning curves become much more problematic. The languages and software infrastructures constitute an extra material to be integrated into the already crowded computer architecture courses. It results that many educators choose the very common processor-specific graphical simulators to illustrate computer architecture topics. Although these simulators provide resources that accelerate the students comprehension, they are not extensible. Therefore, they can not be employed to expose the students to the variety of design possibilities available nowadays.

This context indicates that a more simplified and rapid design process is highly desirable, not only for designers working in research and in industry, but also for students (the future designers) and educators. Since a design methodology conciliating the demands from these two related worlds is not available, this paper presents T&D-Bench - Teaching and Design Workbench, a software infrastructure aimed at the rapid development of state-of-the-art processors simulation models. The remaining of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the T&D-Bench design methodology, highlighting the resources for describing more complex processor mechanisms. Section 4 shows obtained results. Section 5 evaluates the design methodology, by means of comparisons. Final conclusions and future work are drawn in Section 6.

2. Related work
Software environments based on description languages (Architecture Description Languages or Hardware Description Languages) and performance simulation tools are employed for processor modeling and simulation in research and in industry. They are usually adapted to be used also in education.

Hardware Description Languages, such as VHDL or Verilog, are largely employed in industry and universities for modeling, simulation, verification, and synthesis of processors or digital systems in general. They were constructed in such a way that, from their descriptions, it is possible to generate automatically the information needed for hardware manufacturing. Because of this, the language constructs, such as commands, data types, and units, have a strict relation to the hardware mechanisms and behavior, such as bit vectors and concurrent execution. This style of description, using a low abstraction level since too much hardware details are required, is not the best option for design space exploration, since detailed descriptions cannot be created and simulated in shorter periods of time. Thus, HDLs are mostly employed in steps that follow the design space exploration in the design flow. In education, although the correspondence between language constructs and the hardware can be important for understanding computer architecture concepts, the skills required by the designer to use these languages can take time that would be better employed in the study of computer architecture topics.

Architecture Description Languages, such as LISA [2], EXPRESSION [3], and ArchC [4], were developed for processor design space exploration in research. They automatically generate simulators, compilers, debuggers, assemblers, and other tools for model experimentation. ADLs have specific resources (in the form of language constructs) to model the processor organization (micro-architecture) and architecture (instruction set). The abstraction levels used by these languages are higher than those employed in HDLs. This characteristic accelerates the design process and the simulation of the processor model, but it hides some hardware details that would be important for the students’ comprehension as well as for the hardware manufacturing. For example, the components of the processor organization explicitly modeled in ADLs are usually those carrying information related to the processor state, such as registers and memories, while the functional units are only implicitly described in the instruction set description. As with HDLs, the use of ADLs requires time to acquire the skills necessary to model even the most simple processor.

Performance simulation tools [1] are also employed for processor design space exploration. Differently from HDLs and ADLs, such environments have their modeling resources based on software modules, available in the form of classes, files, or related sets of functions, that can be parameterized, reused, modified, and extended to be employed in new processor models. Such modules are normally implemented using a general-purpose programming language such as C, C++, or Java. Although these infrastructures can show very distinct modeling and simulation characteristics, it is possible to distinguish two categories of tools. One category is composed by environments that include fixed but highly parameterized processor models. The popular SimpleScalar tool set [5] is an example. The other category includes environments that, as ADLs, provide resources for modeling and simulation of processors and other computing systems, such as multiprocessors. The LSE [6] environment is an example of this category. Simulators from these two categories are largely used for design in research and in industry, because they are flexible and their simulation runs are very fast. They are used in education too, and for this purpose it is very common that they incorporate, during their development, graphical packages suited to this kind of use. But these infrastructures are large and very complex programs whose comprehension requires a proportional effort.

Processor-specific graphical simulators, or teaching simulators, as ESCAPE [7] and SPIECS [8], provide graphical resources to the user for tracking and steering the experiments that are extremely interesting in educational environment. But, since they do not allow the development of new processor models, they have no application in research or in industry.

Differently from the environments described earlier, T&D-Bench was originally planned to take into account the requirements imposed by both educational and research environments. T&D-Bench’s main objective is to provide a simplified and rapid design process, in a way that computer architecture educators and students can create new models using no more than the knowledge from the traditional computer science and engineering curricula, with the same modeling resources that exist in other environments used for design space exploration. To achieve this goal, the environment employs a mixed approach in the design process. As in ADLs, the designer begins a new model using a specialized script language to define aspects from the various processor domains: micro-architecture, instruction set, and timing. As in performance simulation tools, fine-tuning adjustments or the description of more complex processor mechanisms can be achieved by means of
modifications in selected parts of the software infrastructure, using resources available in a traditional object-oriented programming language.

3. T&D-Bench Design Methodology

The component library is the basis for the construction of new processor models in T&D-Bench. Micro-architectural processor aspects are modeled by the selection, parameterization, and interconnection of components available in the library. The definition of component execution sequences (as hardware micro-operations) builds elementary execution units that can be reused to form the instructions' behavior. Timing of the processor can be expressed separately from the previous specifications and later associated to individual component execution statements into elementary execution units. The T&D-Bench simulation procedure can use these timing specifications in different ways to model mono-cycle, multi-cycle, and pipelined microprocessor execution paths. These specifications are produced by the T&D-Bench script language and are translated to internal data structures that can be manipulated by a set of specialized functions, called T&D-Bench macros, which are provided by the environment to model more complex processor mechanisms.

3.1. T&D-Bench components

T&D-Bench components, implemented as classes in the software infrastructure, correspond to the different structural components in the processor datapath, such as multiplexers, registers, functional units, register files, memories, and others. A T&D-Bench component has ports, contents, and attributes and executes a certain behavior. Ports are used to connect components (input and output ports) or to provide control signals to the component (control ports). The component contents is employed to store the component internal state in case of registers and memories or other sequential logic components. Combinational components do not have contents. Attributes are employed to store property values of a specific component, such as the number of registers in a register file, or to carry status information, such as a flag to indicate that a new instruction was just fetched from the instruction memory. The behavior of a component is defined by operations on the input data that produce, and eventually save, results at the outputs. This component composition, in the T&D-Bench design methodology, is enough to describe micro-architectural and instruction set aspects of the modeled processor. Figure 1 shows two pipeline stages of the DLX processor [9], and Figure 2 shows the use of the T&D-Bench script language to model the micro-architectural aspects of these two stages.

Figure 1. DLX decode and execute stages

Figure 2. Defining a micro-architecture

The create statements define the processor datapath components. They have the form:

\[ \text{create } <\text{name}> <\text{type}> <\text{value1}> ... <\text{valueN}> \]

where: name is the component name. The DLX register bank is named registers, for example. type is the component type, a register for example, identified by a number. value1 to valueN are numbers to set component specific properties, such as size and bit width. The TDSim program, presented later in this paper, contains a help describing the component types available in the library and their related properties.

The link statements define connections between components. They have the form:

\[ \text{link } <\text{source}> <\text{outport}> <\text{target}> <\text{inport}> \]

where: source and target are the names of the components to be connected. output port and input port are the names of the output port of the source component and the input port of the target component, respectively. Ports have default names in the form of E1 to EN.
indicating input ports and S1 to SN indicating output ports, but the designer can alter these names.

As stated earlier, the definition of component execution sequences builds elementary execution units that can be reused to form the behavior of instructions. Figure 3 shows two elementary execution units, named DecodeEEU and ExecuteEEU, that will be later associated to the decode and execute pipeline stages of a DLX arithmetic instruction execution, respectively. The first is described in lines 2 to 4, and the second is described in lines 7 to 12.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>// DecodeEEU</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>registers.NR1 = 0</td>
<td>Set control ports SR1</td>
</tr>
<tr>
<td>3</td>
<td>registers.NR2 = 0 and NR2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>registers.read</td>
<td>Read the two registers from</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>the bank</td>
</tr>
<tr>
<td>6</td>
<td>// ExecuteEEU</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>muxa.SEL = 0</td>
<td>Set selection control port of multiplexer muxa</td>
</tr>
<tr>
<td>8</td>
<td>muxa.behavior</td>
<td>Execute the multiplexer behavior</td>
</tr>
<tr>
<td>9</td>
<td>muxb.SEL = 0</td>
<td>The same for muxb</td>
</tr>
<tr>
<td>10</td>
<td>muxb.behavior</td>
<td>The same for muxb</td>
</tr>
<tr>
<td>11</td>
<td>alu.OP = 0</td>
<td>Set alu operation</td>
</tr>
<tr>
<td>12</td>
<td>alu.behavior</td>
<td>Execute the ALU behavior</td>
</tr>
</tbody>
</table>

Figure 3. Defining instruction behavior

The first form is used to execute the component behavior. Read and write are used for reading and writing components with contents, respectively. Behavior is used for components without contents. The second form indicates the control ports set by the elementary execution unit. The value in the script of second form indicates the control ports set by the behavior.

The processor timing is described separately from the previous micro-architecture and instruction behavior specifications and includes: the specification of the time execution mode to be employed for the execution of instructions at simulation time; and the specification of the various execution stages (cycles or pipeline stages) of a instruction of the modeled processor. Figure 4 uses the DLX timing specification to illustrate this aspect of the T&D-Bench design methodology.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PIPELINED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>FETCH, 0, 0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DECODE, 1, 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>EXECUTE, 2, 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>MEMORY, 3, 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>WRITEBACK, 4, 4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Defining timing

3.2. Modeling processor timing aspects

The processor timing is described separately from the previous micro-architecture and instruction behavior specifications and includes: the specification of the time execution mode to be employed for the execution of instructions at simulation time; and the specification of the various execution stages (cycles or pipeline stages) of a instruction of the modeled processor. Figure 4 uses the DLX timing specification to illustrate this aspect of the T&D-Bench design methodology.

The first statement defines the time execution mode. It has the form:

\(<name>,<behavior | read | write>\)

\(<name>,<controlport> = value\)

The first form is used to execute the component behavior. Read and write are used for reading and writing components with contents, respectively. Behavior is used for components without contents. The second form indicates the control ports set by the elementary execution unit. The value in the script of Figure 3 is arbitrary and will be changed by the actual value at simulation time, when the instruction is decoded. Comments in Figure 3 show more details of these two statements.

Each statement that represents a individual component execution, or control port configuration, is named a micro-operation. Scripts describing elementary execution units can be saved in files and reused in other scripts by means of the statement include <file>. The elementary execution units can be rearranged not only for the description of a specific instruction type behavior but also for the description of other behavioral aspects of the modeled processor, such as the fetch process for example.

It is worth to notice that only six reserved words are introduced by the T&D-Bench script language: create, link, behavior, read, write, and include. All other tokens are names of structures existing in the modeled processor (components and ports) or numbers related to component properties or control ports.
The statements to define instruction behavior considering timing aspects have the following forms:

\[
\text{mop[origin stage number]} \text{ micro-operation}
\]

where: \(\text{mop}\) means micro-operation; and the number inside the brackets indicates the corresponding origin stage number to which the micro-operation is associated.

In the example of Figure 5, the micro-operations of the two elementary execution units are associated to the DECODE and EXECUTE stages respectively. Since the target stage number in Figure 4 is the same as the origin stage number, the micro-operations of the two elementary execution units are actually executed in two successive time units, overlapped to the execution of other instructions, since the time execution mode is \textit{PIPELINED}. Figure 6 shows alternative timing specifications for the DLX processor, to illustrate the use of the target stage number.

<table>
<thead>
<tr>
<th>PIPELINED</th>
<th>NONPIPELINED</th>
</tr>
</thead>
<tbody>
<tr>
<td>FETCH, 0, 0</td>
<td>FETCH, 0, 0</td>
</tr>
<tr>
<td>DECODE, 1, 0</td>
<td>DECODE, 1, 1</td>
</tr>
<tr>
<td>EXECUTE, 2, 0</td>
<td>EXECUTE, 2, 2</td>
</tr>
<tr>
<td>MEMORY, 3, 1</td>
<td>MEMORY, 3, 3</td>
</tr>
<tr>
<td>WRITEBACK, 4, 1</td>
<td>WRITEBACK, 4, 4</td>
</tr>
</tbody>
</table>

The description on the left groups micro-operations to simulate a DLX processor with two pipeline stages. Micro-operations related to the original stages FETCH, DECODE, and EXECUTE are executed in only one time unit (target stage number = 0), as well as those related to stages MEMORY and WRITEBACK (target stage number = 1). The description on the right models a multi-cycle DLX (the instructions do not overlap), with each instruction executing in five time units. In this non-pipelined specification, if all the target stages numbers were changed to 0, the model would correspond to a mono-cycle DLX.

The specification of timing aspects requires only one more reserved word: \(\text{map}\). Due to the reduced number of reserved words and the simple format of the statements in the script language, as explained above, a graphical front-end can be easily employed to produce such descriptions.

### 3.3. Simulation procedure

The kernel of the T&D-Bench simulation procedure is the entity \textit{processor}, implemented as a class in the software infrastructure. It contains all the information about the modeled processor, obtained from the script language specifications in the form of data structures (micro-architecture, instruction set, and timing), and default methods called by the simulation engine to simulate the model. The central idea of the simulation engine is that the processor entity is activated at every simulation time to create and execute new instructions by making them go through the target execution stages previously defined. These instructions are temporary entities that are created based on a op-code fetched from a component (the instruction memory normally). Each of these temporary entities are composed by: a set of attributes, defined by the designer, such as op-code, type, and control port identifiers whose associated values are provided to the component control ports during the instruction execution; and a list of micro-operations inherited from the corresponding elementary execution units of the matched instruction type. During the execution of an instruction in a specific execution stage, only its micro-operations whose \(\text{map}\) number matches the target stage number are executed.

As stated earlier, a software infrastructure class \textit{processor} implements the T&D-Bench design methodology entity \textit{processor}. The \textit{behavior} method of this class is called at every simulation time. The code inside the \textit{behavior} method must call other four predefined methods. The method \textit{initialize} performs initializations, such as the insertion of the first fetch in the execution stages. It is called only once. The method \textit{fetch} fetches and returns new instructions to be decoded. The method \textit{decode} decodes the instructions, associating them to the elementary execution units of a specific instruction type. Finally, the method \textit{execute} executes the instructions based on the time execution mode. In the current implementation of T&D-Bench, the code of the methods \textit{initialize}, \textit{fetch} and \textit{decode} must be provided by the designer. Additionally, in the method \textit{behavior}, the designer must provide the code to insert the instruction temporary entities, created by the method \textit{decode}, at the correct execution stage. A set of specialized functions, named T&D-Bench \textit{macros}, are provided for this task.

### 3.4. Advanced resources

T&D-Bench macros can be used for fine-tuning adjustments in the model or for the description of more complex processor mechanisms. Table 1 shows existing macros whose calls can be inserted in the code of existing processor methods or in new methods created from scratch. It is desirable that calls to these new methods are inserted in the \textit{behavior} method. Thus, the entry point to employ the T&D-Bench...
advanced resources is the method behavior of the class processor.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor.get</td>
<td>Set/get an attribute of the processor entity</td>
</tr>
<tr>
<td>Processor.set</td>
<td></td>
</tr>
<tr>
<td>Datapath.execute</td>
<td>Execute a method from a component: read, write, behavior, set or get (to port or attribute values)</td>
</tr>
<tr>
<td>Datapath.link/delink</td>
<td>Create or remove a connection between two components</td>
</tr>
<tr>
<td>Instruction.</td>
<td>Define the attributes (fields) of the modeled processor instructions</td>
</tr>
<tr>
<td>Instruction.get Instruction.set</td>
<td>Set/get an instruction attribute</td>
</tr>
<tr>
<td>Instruction.</td>
<td>Deactivate or replace a component in the instruction micro-operation list. The component will not be executed or a substitute will be executed</td>
</tr>
<tr>
<td>Instruction.</td>
<td>Change the micro-operation list of a instruction, assigning a new value to a control port</td>
</tr>
<tr>
<td>Instruction.</td>
<td>Insert a new micro-operation into the micro-operation list. It will execute the method of the component at the specified execution stage</td>
</tr>
<tr>
<td>ExecStage.getCurrentInst</td>
<td>Return the instruction executing at a specific execution stage</td>
</tr>
<tr>
<td>ExecStage.insert</td>
<td>Insert a instruction at a specific execution stage and advance the other instructions through the next stages</td>
</tr>
<tr>
<td>ExecStage.freeze</td>
<td>Interrupt the execution of micro-operations related to the specific execution stage</td>
</tr>
<tr>
<td>ExecStage.insertBubble</td>
<td>Insert a bubble (nop) at a specific execution stage</td>
</tr>
<tr>
<td>ExecStagediscard</td>
<td>Discard the instructions between two execution stages</td>
</tr>
<tr>
<td>Pipeline.</td>
<td></td>
</tr>
<tr>
<td>testInstruction</td>
<td>Test if the attribute of the instruction at a specified stage has a specific value</td>
</tr>
</tbody>
</table>

The next figures illustrate the use of T&D-Bench macros using examples extracted from the processor models already available in the environment. Figure 7 shows the method fetch of the DLX processor model. The component instruction memory imem has an attribute that is internally set to 1 to indicate that a new instruction was just read. In the example code, this attribute is tested (line 2) and, if its value is true, the value in the output port of imem (the op-code) is read (line 3) and returned by the method (line 5). If there is no new instruction at this simulation time (in the case of a non-pipelined version), a zero is returned (line 6).

```java
01)private long fetch ( ) {
2)if(dtp.execute("imem",GET,"FETCH",STATUS)==1L) {
3) long lOp = dtp.execute("imem",GET,"\01",OUT);
4) dtp.execute("imem",SET,"FETCH",STATUS,0);
5) return ( lOp );
6} else return ( 0L);
```

Figure 8 shows the method decode of the FemtoJava [10] processor model. Some comments to help the comprehension are listed below:

- the argument to the method is the op-code returned by method fetch (line 1);
- the op-code of type long is translated to a binary string to get the instruction bit fields (line 2), and a temporary entity instruction is created (line 3);
- the bit field that identifies the FemtoJava instruction type is extracted (line 4) and tested against the op-code of an arithmetic instruction ADD (line 5);
- if the op-code really identifies an instruction ADD, the instruction attributes DESCRIPTION and OP are set. The first attribute is used by the simulation engine to select the instruction behavior description that provides the list of micro-operations to this instruction temporary entity, while the second one is used to set the operation control port of the component ALU.

T&D-Bench provides some auxiliary macros too, as those for numerical system manipulation in Figure 8. But they were not improved for performance till now, as one can see by the use of strings to manipulate bit fields. Figure 9 shows a method created by the designer to detect RAW hazards in the DLX processor model

```java
01)public boolean dataHazard (Instruction parInst) { 02) Instruction itcTransfer, itcTransfer; 03) int iNwm = 1, iNwe = 1, iNw1 = 1, iNw2; 04) if ( parInst == null ) return ( false); 05) itcTransfer = pPipe.getCurrentInst ( "DECODE" ); 06) if ( itcTransfer.getInfo( "W1", FIELD ) ) itcTransfer = pPipe.getCurrentInst ( "EXECUTE" ); 07) if ( itcTransfer.getInfo( "W2", FIELD ) ) itcTransfer = pPipe.getCurrentInst ( "MEMORY" ); 08) if ( iNw1 = iNw2 ) iNw2 = iNw1; 09) if ( iNw1 = iNw2 ) iNw2 = iNw1; 10) iNw1 = pPipe.getData( "NR1", FIELD ); 11) iNw2 = parInst.getData( "NR2", FIELD ); 12) iNw1 = pPipe.getData( "W1", FIELD ); 13) if ( iNw2 == iNw1 ) iNw2 = iNw1; 14) return ( false ); 15) if ( iNw2 == iNw1 ) return ( true ); 16) else if ( iNw2 == iNw1 ) return ( true ); 17) else if ( iNw2 == iNw1 ) return ( true ); 18) return ( false );
```

Figure 8. Decode method of the femtoJava

Figure 9. Hazard detection in DLX

The argument to the method is the last fetched instruction (line 1). The numbers of the registers to be written by the instructions in the stages DECODE,
EXECUTE, and MEMORY are read from the corresponding temporary entities attributes (lines 5 to 10). The numbers of the registers to be read by the current instruction is read (lines 11 and 12). Finally, some tests are executed to detect the RAW hazard (lines 13 to 18). The code in lines 300 to 303, inside the method *behavior*, shows the hazard treatment by the insertion of a bubble in the pipeline stage DECODE and by the temporary interruption of the fetch process.

Figure 10 shows a piece of code representing the dispatching of a instruction to an alternative execution stage (STAGEB), since other instruction is being executed at STAGEA. The functional units *fu_a* or *fu_b* are activated by micro-operations (*fu_a.behavior* or *fu_b.behavior*) of these instructions at STAGEA or STAGEB, respectively. The default is the use of *fu_a* because it is originally described in the instruction type behavior. Since *fu_a* is unavailable at this simulation time (line 1), because the previous instruction is at STAGEA, the new instruction must be dispatched to the alternative execution stage STAGEB (line 3). Before this, its micro-operation list must be updated (line 2) in such a way that *fu_b* is used during the instruction execution instead of *fu_a* (*fu_a.behavior* is changed to *fu_b.behavior*).

```
1) if(pPipe.walk(Instr,"STAGEA")=='false'){
2)   Instruction.substitute("fu_a","fu_b");
3) if(pPipe.walk(Instr,"STAGEB")=='false') {
4)   ... }
```

Figure 10. Dispatch of instructions

For modeling a superscalar processor using T&D-Bench, more complex control mechanisms, such as the Tomasulo algorithm or reorder buffers, are better modeled by new methods, or even new classes, called by the method *behavior* of the class *processor*. Otherwise, the alternative execution datapaths of a superscalar processor can be modeled almost entirely using the script language.

The component library contains various classes representing structural components found in state-of-the-art processor datapaths. But the programming of new classes is also possible, starting from scratch or by specialization of other available non-abstract classes, using the facilities of the object-oriented Java language.

### 4. Results

A program called TDSim was developed to validate the T&D-Bench design methodology. It constitutes the software infrastructure front-end. Two execution modes are available in the program: interactive and batch modes. In the interactive mode, more adequate to use in introductory computer architecture courses, the user may instantiate and test individual components or set of interconnected components. It is a good start point for the designer to familiarize with the T&D-Bench script language. In the batch mode, the environment reads a complete processor description from the files and generate the cycle-accurate simulator. There are some processor models already available in TDSim. They include the mono-cycle, the multi-cycle, and the pipelined versions of the DLX processor and a model of the FemtoJava microcontroller, used in research (FemtoJava is a stack machine). As stated earlier, TDSim provides a help module that describes all the components, and its associated attributes, available in the component library.

The TDSim program was used in introductory computer architecture courses and received a good evaluation from the students. The use of the script language to model parts of a processor improves the students’ understanding of the whole processor. We are now working in a integration with JhotDraw [11], an open source Java GUI framework developed for technical and structured graphics elaboration, to provide graphical resources during simulation. A simple microprogrammed processor model, also available in TDSim, is already being simulated using graphical resources.

### 5. Evaluation

We chose one environment of each category, used for design space exploration as described in Section 2, to compare to T&D-Bench. VHDL was chosen since it is largely used in industry and in education, ArchC because it has a good documentation available, and SimpleScalar by its large popularity.

T&D-Bench allows similar micro-architecture descriptions as VHDL. But, additionally, it provides resources to model the other processor aspects, i.e., instruction set and timing. The approach of ArchC is the most similar in all aspects to the T&D-Bench design methodology. But the more detailed description of the micro-architecture in T&D-Bench is advantageous to be employed in education and to an eventual translation into a HDL description for a hardware synthesis. SimpleScalar is not reconfigurable as T&D-Bench. Besides, since its software organization does not reflect the processor composition (as the T&D-Bench component library and class *processor*), modifications are more difficult, since the designer must understand this organization, which is derived from particular software engineer skills in the adopted programming language. Furthermore, to the best of our
knowledge, we did not find other environment that requires the utilization of so few elements in the model construction (other than those related to the processor and a general purpose programming language domain):

- only seven reserved words are introduced by the T&D-Bench script language: create, link, behavior, read, write, include and mop;
- the entry point to apply the T&D-Bench advanced resources is the method behavior of the class processor; and
- a set of macros to modify various aspects of the modeled processor.

Again, to the best of our knowledge, we did not find in other environments a functionality that is similar to that provided by the macros, aggregated in a specific resource of the software infrastructure (the set of macros). Furthermore, the use of macros requires arguments that are, almost entirely, identifiers defined by the designer in the previous steps of the design process, as component, port, and execution stages names. This characteristic provides the user with a high degree of freedom to change aspects of the processor at simulation time, since the execution of macros can be fired by the user using the identifiers described earlier.

6. Conclusions and future work

This paper described T&D-Bench, a software infrastructure that combines features that simplify and accelerate the processor design process without restricting the designer possibilities. It thus represents a good trade-off for educational and research purposes that is not found in other environments. The T&D-Bench design methodology employs a mixed approach in the design process, based on a specialized language and on facilities to change the software infrastructure itself. T&D-Bench characteristics are ideal to fulfill the requirements of the current hardware design process. In research, results can be produced and delivered in short time periods. In education, it can be used to motivate and prepare more qualified designers for industry and research.

We are planning the following activities for the near future:
- the automatic generation of the method decode for instruction decoding;
- the construction of the project web page, where the software will be publicly available;
- a more exhausting use of the infrastructure to model new mechanisms of state-of-the-art processors; and
- a full integration with the JhotDraw framework both for modeling and simulation.

References


