

From Classroom to Research: Providing Different Services for Computer Architecture Education

Sandro Neves Soares,
Universidade de Caxias do Sul (Brasil)
snsouares@ucs.br

Flávio Rech Wagner
Universidade Federal do Rio Grande do Sul (Brasil)
flavio@inf.ufrgs.br

Abstract

T&D-Bench (Teaching and Design Workbench) is now a mature framework for processor modeling and simulation. The framework's processor models have been employed by students to execute practical exercises in Computer Architecture (CA) courses and, by the instructors, to illustrate CA concepts in classroom. The framework itself has been employed by seniors and researchers to make the design space exploration of embedded processors, which is accomplished by modeling new processors or by extensions to the existing processor models. The results of the research activities, incorporated to the processor models, may then be employed in classroom, producing a virtuous circle. This work describes all the services developed for T&D-Bench along the last years. For example, a practical use in research to model an architectural feature that reduces energy consumption, as well as a comparison with a well-established Architecture Description Language are presented.

Categories and Subject Descriptors

C.1 [Processor Architectures]: Single Data Stream Architectures

General Terms

Design, Languages

Keywords

Processor Modeling and Simulation, Computer Architecture Education

"Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WCAE '07, June 9, 2007 San Diego, CA
Copyright 2007 ACM 978-1-59593-797-1/07/0006...\$5.00

1. Introduction

Our main objective when we planned the framework T&D-Bench was to offer modeling resources that were available only in professional and research CAD tools, while avoiding the steep learning curves required for understanding and using such tools. This way, the designers (students, instructors and researchers) would have available an easy and rapid modeling process to create processor models from scratch or to explore the design space, altering models or their configurations.

At simulation time, the processor models had to incorporate, automatically, but optionally, graphical user interface resources for tracking and steering the experiments, so as to be a valuable tool for teaching and learning Computer Architecture (CA).

Other requirement for T&D-Bench was an easy access by the users, as well as an easy installation, and utilization process, to simulate the processor models and also to employ the framework's modeling resources.

This set of characteristics, i.e., an easy and rapid modeling process, graphical user interface resources for tracking and steering the simulations, and an easy access by the users, make T&D-Bench an innovative framework to be employed in CA Education.

After some years using the framework, we have closed a virtuous circle, i.e., students that learned CA concepts using the T&D-Bench processor models are now working on research projects investigating real problems. They are helping to extend the processor models and, in consequence, creating new tools and subjects to be presented in CA courses. This paper describes the current stage of the framework and the services it provides for CA Education.

2. T&D-Bench Design Methodology

The kernel of the T&D-Bench design methodology is a component library where the description of processor datapath components is similar to VHDL

behavioral descriptions of entities. A simplified description language (T&D-SDL or simply TDSDL) is employed to select, parameterize, and interconnect these components in order to define the processor micro-architecture; and also to specify component execution sequences (as hardware micro-operations) that compose elementary execution units. These elementary execution units can be reused to form the behavior of instructions. Timing of the processor is also defined by TDSDL. It must be expressed separately from the previous specifications and later associated to individual component execution statements in 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.

TDSDL handles aspects that are found in every processor, independently of its type – RISC, VLIW, microcontroller, stack machine, etc. TDSDL specifications are translated into internal data structures that can be manipulated by a set of specialized methods, called *macros*, which are provided by the environment to model complex or specific architectural mechanisms. The component library, TDSDL and the macros constitute the three layers of modeling resources in T&D-Bench design methodology, as shown in Fig. 1.

MACROS				
<i>Datapath macros</i>	<i>Instructions related macros</i>	<i>Execution Stages related macros</i>	<i>Instruction Queues macros</i>	<i>Other utility macros</i>
T&D-SDL DESCRIPTION LANGUAGE				
<i>Micro-architecture Specifications</i>	<i>Instruction Set Specifications</i>	<i>Timing Specifications</i>		
COMPONENT LIBRARY				
<i>FPU's</i>	<i>ALUs</i>	<i>Memories</i>	<i>Register Banks</i>	...

Figure 1. Design Methodology Resources Overview

2.1. Component Library

The component library is implemented as a set of classes where each class describes an individual structural component that exists in processor datapaths. The designer must program a new class only if a structural component required by the model is not available in the library. A T&D-Bench component has *ports*, *contents*, and *attributes* and executes a certain *behavior*. This is the logical structure of a component in T&D-Bench. *Ports* are used to connect components (input and output ports) or to provide control signals to

the component (control ports). The component *contents* are employed to store the internal state in case of registers and memories or other sequential logic components. *Attributes* are employed to store property values of a specific component, such as the number of registers in a register file. The *behavior* of a component is defined by operations on the input data that produce, and eventually save, results at the outputs. The default logical structure of the T&D-Bench components simplifies the task of programming new components for the library.

2.2. T&D-Bench Simplified Description language

2.2.1. Micro-architecture specification. The specification of the processor micro-architecture is carried out by the statements *create* and *link*. The *create* statement defines the processor datapath components using the form: *create* <name> <type> <value1> ... <valueN>, where *name* is the component name; *type* is the component type, a register file for example, identified by a number; and *value1* to *valueN* are numbers to set values of component properties, such as size and bit width. Each new component inserted in the component library must provide an online documentation describing its properties. The *link* statement defines connections between components. It has the form: *link* <source> <outport> <target> <inport>, where *source* and *target* are the names of the components to be connected, and *outport* and *inport* are the names of the output port of the source component and of the input port of the target component, respectively. Ports have default names in the form of E0 to EN, for input ports, and S0 to SN, for output ports. Fig. 2 shows a small fragment of the acemMIPS [2] micro-architecture description, using the *create* and *link* statements.

```
// create the integer register file (values in the documentation)
// type=50; nRegisters=32; bitWidth=32; nEntries=4; nOuts=11
create GPRFile          50 32 32 4 11
// create the integer functional units. type=8; notUsed=0, bitWidth=32
create alu1_ex          8 0 32
create alu2_ex          8 0 32
// connect the ports between the register file and the functional units
link GPRFile           S0    alu1_ex    E1
link GPRFile           S2    alu2_ex    E1
```

Figure 2. Micro-architecture Specification

2.2.2. Instruction set specification. *Elementary execution units* are built by the definition of component execution sequences. The statement for the specification of a component execution has the following forms: <name>.<behavior | read | write> and <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*. Each statement that represents an 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>*. *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 processor, such as the fetch process. Fig. 3 shows two elementary execution units, named *readRegisterBank* and *executeAluOperation*, and their use to describe an arithmetic and logic instruction of the acemIPS processor. The timing information is also included and will be explained in the next sub-section.

```
// E.E.U. readRegisterBank: reads two registers of the register bank
// The actual register numbers are set during simulation depending on
// decoded instruction fields
GPRFile.NR0 = 0
GPRFile.NR1 = 1
GPRFile.read

// E.E.U. executeAluOperation: defines the functional unit operand
// source using multiplexer mxGeneral. The actual control port numbers
// are set during simulation
mxGeneral.SEL = 0
mxGeneral.behavior
// defines the operation to be applied by the functional unit. The actual
// control port numbers are set during simulation
aluGeneral.OP = 0
aluGeneral.behavior
// aluGeneral is replaced by alu1_ex or alu2_ex depending on the
// availability of the functional units. The same happens for mxGeneral.

// Arithmetic and Logic Instruction
mop[0] include readRegisterBank
mop[1] include executeAluOperation
```

Figure 3. Arithmetic and Logic Instruction Specification

2.2.3. Timing specification. Fig. 4 shows the timing of one of the five execution paths of the acemIPS processor, related to the second integer functional unit *alu2_ex*. Other paths are related to the first integer functional unit and to the floating point, branch, and Load/Store units. We consider an *execution path* the set of stages necessary to execute an instruction. These stages are later linked to the *micro-operations* defined in the instruction set specification (the definition of stages carries only timing information). The *execution path alu2* includes two execution stages. The first stage is used by arithmetic and logic instructions to read the register file (ALU2READ), and the second one (ALU2EXECUTE) is used to execute the operation in the functional unit. These stages are pipelined and modeled to be activated at consecutive cycles – the first

numbers after each stage identifier are used to specify these cycles: 0 and 1. This behavior can be altered by the second numbers, in the same lines, to activate the stages in the same clock cycle for example, as explained below.

```
NAME=alu2
PIPELINED
ALU2READ, 0, 0
ALU2EXECUTE, 1, 1
```

Figure 4. *alu2_ex* Execution Path

The first statement of a timing specification has the form: *NAME=<name>*, where *name* identifies the *execution path* being defined. The next statement is used to define the *time execution mode*, where the possible values are PIPELINED or NONPIPELINED. They state whether the execution of the instructions overlap or not, respectively. The next statement specifies each execution stage: *<name>*, *<origin>*, *<target>*, where *name* is the stage identifier; *origin* is the stage number, used to associate each *elementary execution unit micro-operation* to a specific *execution stage* of the modeled processor; and *target* is the actual stage in which the corresponding micro-operation will be executed at simulation time. The *origin* stage number range will set the maximum number of execution stages of a specific execution path of the processor. In an extreme case, each *micro-operation* can be identified by a different *origin* stage number. The *target* number can be used to group, during simulation time, micro-operations related to different original execution stages, thus providing an easy way of modifying the processor timing specifications. Some of the stages defined can work as *dummy* stages, i.e., they will be employed only to increment time.

The central idea of the simulation engine is that the *processor* model 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 an op-code fetched from a component (usually the instruction memory). Each of these temporary entities are composed by: a) 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 b) a list of *micro-operations* inherited from the corresponding *elementary execution units* of the matched instruction type.

The statements to define instruction behavior considering timing aspects have the form *mop[origin*

stage number] *micro-operation*, where *mop* means micro-operation, and the number inside the brackets indicates the corresponding *origin* stage number to which the *micro-operation* is associated. During the execution of an instruction in a specific *execution stage*, only its *micro-operations* whose *mop* numbers match the *target stage number* are executed. The use of this statement can be seen in Fig. 3, where the *readRegisterBank* elementary execution unit of an arithmetic and logic instruction is executed during the ALU2READ stage (*origin* stage number 0), and the *executeAluOperation* is executed during the last stage (*origin* stage number 1). If the designer wants to read the register file and execute the *alu2_ex* operation at the same clock cycle, he/she has only to change the *target* numbers in Fig. 4 from 0 and 1 to 0 and 0. No changes in the instruction set description are required.

The specification of the various aspects of the processor requires only seven reserved words: *create*, *link*, *behavior*, *read*, *write*, *include*, *NAME*, and *mop*. All other tokens (strings or numbers) to be provided to complete the language statements are names of structures existing in the processor (components, ports and pipeline stages), or names and numbers created by the designer to identify *execution paths* or *stages*, component properties or control port values.

Because of this reduced number of reserved words, the simple format of the statements in the description language, as explained above, and the clear distinction between the sections to specify the processor micro-architecture, instruction set and timing, a graphical front-end can be easily employed to produce the descriptions. For example, the micro-architecture specifications may be created by dragging and dropping images from a toolbar, representing the structural components in the library, over a draw panel; by connecting its ports using the mouse, and by setting the components properties values using dialog boxes.

2.3. Macros

The use of macros to model more complex or specific architectural mechanisms is carried out by programming the T&D-Bench's framework main class, called *processor*. It contains all the information about the processor, obtained from the TSDSL specifications in the form of data structures and default methods which are called by the simulation engine to simulate the model. The *behavior* method of this class is called at every simulation time. The code inside the method *behavior* must call other four predefined methods. The method *initialize* performs initializations, such as the insertion of the first instruction in the execution stages

to activate the fetching process. It is called only once. The method *fetch* fetches and returns new instructions to be decoded. The method *decode* decodes the instructions, associating them to the *elementary execution units* of a specific instruction type. This pair of methods *fetch-decode* provides enough flexibility to model different types of instructions (CISC, RISC, and VLIW). Finally, the method *execute* executes the instructions based on the time execution mode.

The code of the methods *initialize*, *fetch*, and *decode* must be provided by the designer. They are very dependent on a specific processor being modeled but, since all the models have these same methods, they could serve as examples to the designer when modeling a new processor. Additionally, in the method *behavior*, the designer must provide the code to insert the instructions created by the method *decode* at the correct execution stages. For example, the macro *ExecPathAlu2.walk (NewInstruction, "ALU2READ")* inserts a new decoded instruction at the execution path containing the integer unit *alu2_ex*. The arguments to macros are identifiers of elements of a library component (ports, contents, attributes) or identifiers of elements of the processor (such as names of components, *execution paths*, *execution stages*, and instruction queues, as well as references to instructions in execution), which have been created by the designer using TSDSL, i.e., they correspond to information provided in previous steps of the modeling process.

The macros are divided into categories, according to the processor aspect they handle. **Datapath macros** are used to access and manipulate the processor micro-architecture. *Datapath.execute*, for instance, executes the behavior of a component or sets/gets the value of a port or attribute. It can be used to test a *hit* or *miss* in the cache memory, for example. **Macros related to instructions** are used to access and manipulate entities representing the instructions in execution. *Instruction.DefineFieldsOfInstructions*, for example, defines the list of attributes (fields) of the processor instructions. *Instruction.set/get* sets/gets an instruction attribute. It can be used to specify the exact functional unit to be employed in the instruction execution (see Fig. 8). **Macros related to execution stages** are used to handle the different *execution paths* of a superscalar processor, as well as the stages in these paths (different cycles or pipeline stages). Example: *ExecStage.walk* inserts an instruction at a specific *execution stage* and advances the other instructions through the next stages (see Fig. 8, later in Section 4). **Macros related to Instruction Queues** are used to handle instruction queues in superscalar processors. Example: *InstructionQueue .add / removeInstruction* inserts or

removes an instruction into/from a specified queue of instructions (see Fig. 8). Other macros are used to execute auxiliary tasks. Examples: *Processor.get/set* gets/sets an attribute of the processor. It can be used, for instance, to specify the utilization or not of code reduction in the rISA framework (explained in Section 4). *SistNum.getBitRange* gets the integer value provided by the bits in the range.

The macros *SistNum.getBitRange* and *Instruction.set* may be employed in the *decode* method to get the value of a bit field of the op-code just fetched and to set an attribute of an instruction object, respectively. The macros *ExecStage.getCurrentInst* and *Instruction.get* may be employed in a method of the class *processor* to detect data hazards. The first is used to get the current instruction in a specified execution, or pipeline stage; and the latter is used to test an instruction field, such as the number of a register to be written.

3. T&D-Bench Processor Models

Various processor models have been developed with the T&D-Bench design methodology, including pipelined and superscalar processors, microcontrollers, and stack machines. They incorporate automatically, but optionally, graphical resources to be employed to track and steer the experiments. This way, all the processor models share the same GUI. The reasons presented in the end of Section 3.2, about a graphical front-end to produce the TDSDL descriptions, are the basis for this GUI generation.

The processor model itself plus the GUI constitute a *didactic simulator*. The various didactic simulators can be employed to present different modules of CA subjects in distinct stages of a course or along a sequence of courses in the curriculum. Since all of them share the same GUI, the student doesn't need to worry about learning a new GUI when he/she initiates the study of a new processor.

A more extensive explanation about the resources available in T&D-Bench *didactic simulators* to track and steer the experiments can be found in [1] – this is the main focus of that work.

4. Results

The next sections describe various use cases of T&D-Bench: from classroom to research, and from the results of this research activities back to classroom, producing a virtuous circle. Additionally, since an objective of T&D-Bench is to offer modeling resources usually available only in professional and research

CAD tools, it is also shown, in Section 4.4, a comparison with a well-established Architecture Description Language.

4.1. Use at Classroom

Around 120 students, enrolled in the last four editions of the Computer Organization and Architecture course of the Information Systems curriculum at UCS, have had a contact with the T&D-Bench didactic simulators. The course provides an overview of introductory Computer Architecture concepts.

The approach used in classroom includes the use of the simulators by the instructor to illustrate the following topics: (1) the von Neumann model; (2) sequential and combinational circuits; and (3) an introduction to the instruction set and micro-architecture of a didactic micro-programmed processor called Neander. After that, programming exercises are proposed to the students using the assembly language of the Neander processor. Finally, the students choose one of the exercises and present their solution to the other students in the classroom. In the 2005 edition, some of the enrolled students also developed programs for the MIPS processor, even though this processor was not a subject of the course.

The MIPS simulator allows the students to visualize, for example, the temporal evolution of the instructions in execution, their superposition in time, and the actions to handle data and resource conflicts, as well as the exact behavior executed by a datapath component, during the execution of an instruction in a specific clock cycle, by means of a color scheme. Various possibilities to steer the simulations are also available

Some questions about the use of the simulators are proposed to the students in the last week of the courses. The answers report a better comprehension of the CA concepts, mainly related on how to program using the assembly language and also to the understanding of the relationship between architecture and organization.

4.2. A practical use in research

The T&D-Bench MIPS processor model (see Figure 5) was extended to support a popular architectural feature called rISA [3], as part of an ongoing research work being developed with our framework.

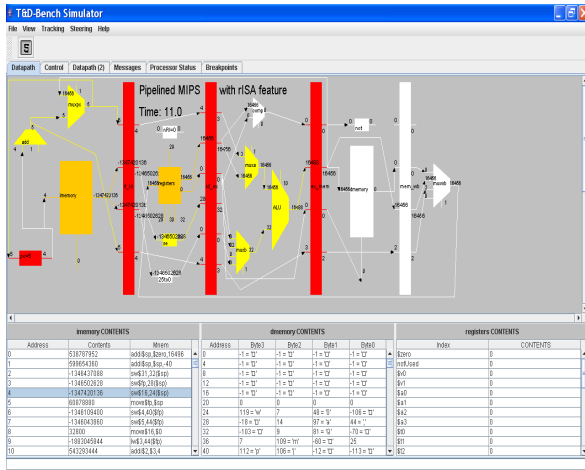


Figure 5. MIPS with rISA Processor Model GUI

rISA [3] is used to reduce instruction memory size of programs. rISA processors execute instructions from two different Instruction Sets (IS): the “normal” IS, and the “reduced bit-width” one. The reduced bit-width instruction set encodes the most frequently occurring instructions in fewer bits. ARM7TDMI, MIPS32/16 bit TinyRISC, STMicro's ST100 and the ARC Tangent processor are examples of rISA processors.

In addition to the code size reduction benefits, a program with rISA instructions requires less fetches to the memory subsystem when it executes. This, in consequence, decreases the energy consumption.

Four MiBench programs [4], namely *CRC32*, *bitcount*, *qsort*, and *stringsearch*, were employed in the experiments with the MIPS extended model. The C code of these programs was compiled to assembly using the *gnu-gcc cross-compiler*. Most of the simulations were executed in batch mode, but the graphical resources are available and were a valuable resource to debug the rISA framework (see Fig. 6).

Figure 7 shows the percentage of originally marked instructions (candidates to be reduced) that were not reduced for the CRC32 MiBench program, due to one of the 3 following causes: overflow of operands, branch and jump handling (branches or jumps, between normal and reduced blocks of instructions, must be handled), or small size of the reduced block. Four different reduced IS's, each one containing 16 opcodes, were used in these experiments. These results are being employed to direct our research work.

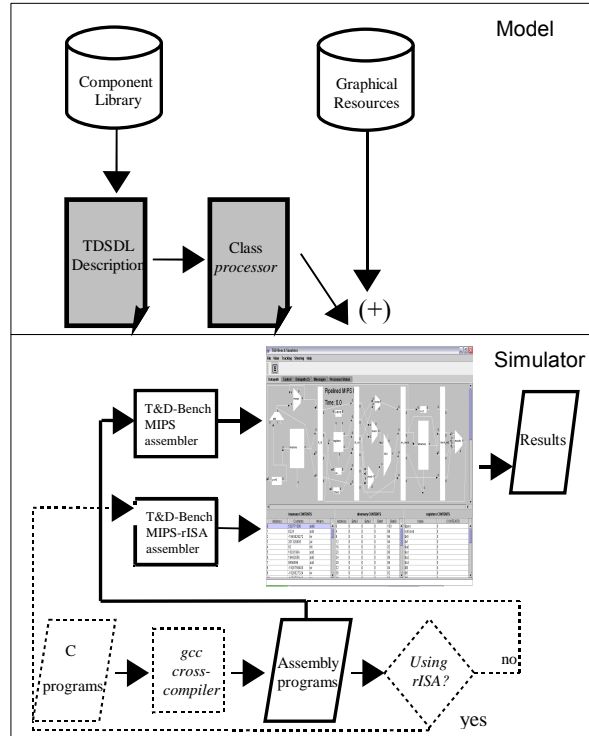


Figure 6. T&D-Bench rISA Framework

4.3. Involving students with research

The importance of the research work described in the last section to Computer Architecture education is threefold: (1) it is an opportunity to let students know and work on a real problem; (2) while developing new software modules for the framework, they can improve their Computer Architecture and programming skills; and (3) the extended processor model can be used in classroom to introduce beginners to real problems in embedded systems design: it does close the virtuous circle.

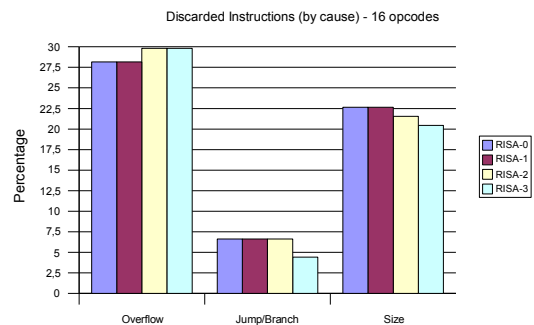


Figure 7. Discarded Instructions from Reduction (by cause)

In 2006, two senior students helped in the T&D-Bench rISA framework implementation. One of them worked in the process of compiling the MiBench C benchmarks, using the *gnu-gcc cross-compiler*, and the

other developed a debug module to insert breakpoints to stop the execution of the benchmarks on the processor model. This year, a senior student is improving the graphical resources of the rISA framework to make easier the configuration of the experiments and the visualization of the results (it will be important in the classroom).

In the past years, other tasks were also executed. A senior developed and explored a component cache memory, and a graduate student worked in the graphical resources module of the framework (those automatically generated to the processor models). All of them reported a better understanding of Computer Architecture concepts and a greater interest in the area.

4.4. A comparison with an ADL

In order to demonstrate that T&D-Bench modeling resources are comparable to that of research CAD tools, we compare it with an Architecture Description Language. The ADL EXPRESSION [2] was our choice since it is a well-established Architecture Description Language and available as an open source release in the public domain.

A series of nine modifications in a model of the acesMIPS processor (a VLIW MIPS R4000-based architecture), described with EXPRESSION, is proposed in [2], together with the required design steps to implement them. The comparison was performed by first modeling the same processor with the T&D-Bench resources and then applying the same modifications to this model. The modifications are: 1) add new complex instructions, 2) change register accessibility, 3) add a single/multi-cycle functional unit, 4) add a new pipelined functional unit, 5) delete a pipeline path, 6) modify access times of caches/memories, 7) modify associativity of caches, 8) change sizes of caches/memories, and 9) add new memory components. They are classified into three categories: those related to ISA exploration (items 1 and 2), those related to pipeline exploration (items 3 to 5), and those related to memory subsystem exploration (items 6 to 9).

After performing each modification in the T&D-Bench processor model, we count the number of required *design steps* and also the number of involved T&D-Bench *sections*. Then we compare these values to the corresponding numbers required for design space exploration in the EXPRESSION model. We consider as a *design step* the insertion, deletion, or modification of a reduced number of lines (less than 15 lines) in a same *section* of the description. This assumption is more general than that adopted in [2], where the

number of design steps is larger than the number considered in this comparative analysis. We consider as *sections* of the description in EXPRESSION the *Operations*, *Instruction*, and *Operation Mappings* sections of the processor behavior specification and also the sections *Components*, *Pipeline and Data-transfer paths*, and *Memory subsystem* of the structure specification. Additionally, due to their extension, the *Var_groups*, *Op_groups*, and *Operand mappings* subsections of the section *Operations* were considered as different sections in this comparison. We consider *sections* in the T&D-Bench design methodology the TDSL's three distinct spaces to describe the processor micro-architecture, instruction set, and timing, as well as each method of the class *processor* and each class in the component library.

As an example, Fig. 5 shows the design steps in the T&D-Bench design methodology to make the functional unit *alu2_ex* execute only MULT instructions (no other type of instructions), in two clock cycles, while the functional unit *alu1_ex* executes all other arithmetic and logic instructions, except MULTs, in one cycle (this is the modification 3 above, i.e. add a single/multi-cycle functional unit). In the original situation, both integer functional units could execute all arithmetic and logic instructions in one clock cycle. Besides the three design steps (*DS-1*, *DS-2*, and *DS-3*), three sections are involved in this design space exploration in T&D-Bench, namely methods *decode*, *dispatch* (an extension of the method *behavior* in the acesMIPS model), and the timing specification of the *alu2_ex* execution path. In EXPRESSION, two design steps are required and two sections are involved.

```
// DS-1: associates MULT to alu2_ex during decoding. "it" is an
// object instruction, describing a MULT in this case
it.set ( "unit", FIELD, ALU2_EX); // "it"-object instruction
// DS-2: the code below dispatches MULT instructions to
// alu2_ex (using the attribute unit) - it is in the dispatch
// method. Additionally, 8 lines were deleted (not shown)
} else if ( it.get( "unit", FIELD) == ALU2_EX) {
    if ( ExecPathAlu2 != null) {
        if ( ExecPathAlu2.walk( it, "ALU2READ") == false) {
            notDispatchedQueue.add ( it); // if the unit is busy
        }
        currentInstructionQueue.removeInstruction ( );
        NotUsedAlu2 = false; // do not allow the insertion of other
        ExecPathAlu2 = null; // instructions during this cycle
    } else {
        notDispatchedQueue.add ( it);
        currentInstructionQueue.removeInstruction ( );
    }
}
// DS-3: a new execution stage is inserted in the execution
// path related to alu2_ex - ALU2ZERO (a dummy stage)
ALU2ZERO,1,1
ALU2EXECUTE,2,2
```

Figure 8. Pipeline Exploration (item 3)

As another example, a single design step is required in the T&D-Bench design methodology to transform *alu2_ex* into a pipelined functional unit (modification 4 - add a new pipelined functional unit), executing only MULT instructions in two clock cycles (this modification extends the previous modification number 3). This design step requires only the alteration of the time execution mode from NONPIPELINED to PIPELINED. Besides the single design step, only one section is involved in this exploration in T&D-Bench. In EXPRESSION, two design steps are required, but only one section is involved as well.

The results of the comparison are summarized in Table 2 (by category of modifications and totals). Table 1 shows the results in terms of lines of code involved in the whole DSE process.

TABLE I. AVERAGE NUMBERS OF LINES INVOLVED

<i>Average number of inserted lines in EXPRESSION:</i>	6,25
<i>Average number of inserted lines in T&D-Bench:</i>	4,62
<i>Average number of deleted lines in EXPRESSION:</i>	5,6
<i>Average number of deleted lines in T&D-Bench:</i>	3,5
<i>Average number of modified lines in EXPRESSION:</i>	2
<i>Average number of modified lines in T&D-Bench:</i>	0,88

TABLE II. AVERAGE NUMBERS OF DESIGN STEPS AND SECTIONS

ISA exploration	
Average number of design steps in EXPRESSION	4
Average number of design steps in T&D-Bench	5
Average number of sections in EXPRESSION	4
Average number of sections in T&D-Bench	5
Pipeline exploration	
Average number of design steps in EXPRESSION	2,3
Average number of design steps in T&D-Bench	1,7
Average number of sections in EXPRESSION	2
Average number of sections in T&D-Bench	1,7
Memory subsystem exploration	
Average number of design steps in EXPRESSION	1,75
Average number of design steps in T&D-Bench	1,5
Average number of sections in EXPRESSION	1,25
Average number of sections in T&D-Bench	1,5
Design space exploration (totals)	
<i>Average number of design steps in EXPRESSION</i>	2,25
<i>Average number of design steps in T&D-Bench</i>	2
<i>Average number of sections in EXPRESSION</i>	1,88
<i>Average number of sections in T&D-Bench</i>	2

The information in Tables 1 and 2 shows reduced numbers of design steps and also of lines of code required to implement the architectural modifications in T&D-Bench. This situation indicates not only that T&D-Bench modeling resources are comparable to that of research CAD tools, but also that they show a high expressiveness and a more simplified and consequently a more rapid modeling process. The number of sections involved, in turn, is slightly superior in T&D-Bench, but it confirms the principles of our design

methodology related to the organization of the modeling resources in three layers and the availability, in TDSL, of specific and orthogonal resources to model the various aspects of a processor. It is worth to emphasize that the proposals of modifications to be applied to the base model were extracted from the documentation that describes the EXPRESSION Architecture Description Language.

5. Related Work

Didactic simulators are simple programs, specialized to be employed in education, with graphical user interface resources for experimenting with some processor models (one to three models are very common). A more extensive presentation of didactic simulators can be found in [1]. CAD tools are complex environments, with a considerable number of resources, largely employed by the industry and also in education. CAD tools provide a contact with professional tools, but the learning of the associated design resources is not a trivial task and requires a considerable time. CAD tools include those based on Hardware Description Languages, such as VHDL or Verilog, those based on Architecture Description Languages, such as LISA [5] and EXPRESSION [2], and performance simulation tools [6], such as SimpleScalar and LSE.

Our motivation to design T&D-Bench, with a more detailed comparison with didactic simulators and CAD tools, can be found in [1].

6. Conclusions and Future Work

T&D-Bench is a framework for processor modeling and simulation that offers an easy and rapid design process, together with graphical user interface resources for tracking and steering the experiments with the processor models. T&D-Bench is available as an open source and platform-independent framework on the Internet (<http://www.tdbench.org>).

This work presented the framework and the virtuous circle of services it provided, over the last years, for Computer Architecture Education. This virtuous circle is composed by students that learned Computer Architecture concepts using the T&D-Bench processor models and are now working on research projects investigating real problems, whose results will be presented in the next Computer Architecture courses. Future work with T&D-Bench includes the modeling of complete embedded systems and the continuity of the framework's utilization in research to explore the design space of embedded systems.

References

- [1] S.N. Soares and F.R. Wagner. "Design Space Exploration of Embedded Processors in Computer Architecture Education using T&D-Bench", 36th Frontiers in Education Conference, San Diego, California, 2006, pp. 19-24.
- [2] P. Biswas, S. Pasricha, P. Mishra, A. Shrivastava, N. Dutt and A. Nicolau, EXPRESSION. Users Manual. Version 1.0, Department of Information and Computer Science, University of California, Irvine, 2003. Available at: <<http://www.ics.uci.edu/~express/>>.
- [3] A. Shrivastava, P. Biswas, A. Halambi and N. Dutt, "Energy Efficient Code Generation using rISA", Proceedings of the Asia and South Pacific Design Automation Conference, 2004, p.475-477.
- [4] M.R.Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", 4th Workshop on Workload Characterization, Dec. 2001.
- [5] S. Pees, A. Hoffmann, V. Zivojnovic and H. Meyr, "LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures", Proceedings of 36th Design Automation Conference, New Orleans, 1999.
- [6] S.S. Mukherjee, S.V. Adve, T. Austin, J. Emer and P.S. Magnusson, "Performance Simulation Tools", Computer, v.35, n.2, Feb. 2002, p. 38-39.