THUMPSim:One Simulation Framework for Processor Architecture Evaluation

Youhui Zhang
Yu Gu
Dongsheng Wang
Weimin Zheng
Department of Computer Science
Tsinghua University
100084, Beijing, P.R.China
E-mail: zyh02@tsinghua.edu.cn

KEYWORDS
Cycle-accurate simulation, Co-verification, embedded processor, MIPS.

ABSTRACT

Evaluation has become the mainstay of computer architecture design and relies heavily on simulators and simulator infrastructure. To improve the evaluation performance and design efficiency, one process simulation framework, THUMPSim, is implemented. THUMPSim can provide the designer with different evaluation accuracies on several simulation levels. Its core contains two main parts, the architecture-independent control module and the definition of the processor, to provide an infrastructure for pluggable components to improve its flexibility. In addition, an event-driven signal update algorithm is employed to simulate concurrent activities in real systems. Some other techniques, including object-oriented component definition, and code auto-generation, are also employed to predigest the simulation work. THUMPSim is employed by our team to develop THUMP107 embedded CPU and speed up the software/ hardware co-development remarkably.

INTRODUCTION

As ASIC designs explode in size and complexity, evaluation has become the mainstay of processor architecture design before the traditional RTL coding, layout design and verification flow. The tremendous complexity of systems is making it both difficult to reason about and expensive to develop. Detailed software simulations have therefore become essential for evaluating ideas in the architecture field. Industry uses simulation extensively during processor and system design because it is the easiest and least expensive way to explore design options. Simulation is even more important in research to evaluate radical new ideas and characterize the nature of the design space(Skadron et al. 2003).
However, current simulation infrastructure is written in ways that limit code sharing and lacks a satisfying component-based architecture. Current tools tend to dictate the majority of research that gets done, because it is easier to study questions that can be answered with existing tools. This means that there are parts of the design space that end up only lightly explored, because of the difficulty of doing so.
The workshop summary (Adve et al. 2001) suggests that simulation frameworks and simulator construction frameworks were superior to monolithic simulators or simulator. But in fact, most existing simu-lation tools belong to the latter. As we know, examples of some popular simulators include the SPIM simulator (Breach et al. 1999), the simplescalar simulator (Burger and Austin 1998) and the SuperDLX simulators(Moura 1993). Once developed, such simulators are difficult to retarget to a modified micro architecture without a significant amount of effort.
Therefore, we develop a simulation framework and its construction tool, named THUMPSim, to provide users with a method to construct the fast prototype of the target design. In contrast with the most existing simulators, THUMPSim has the following features.

1. A component-based infrastructure is completed. That is, it contains two main parts, the architecture-independent driven engine and the definition of the target design, to provide an infrastructure for pluggable components to improve its flexibility.
2. An event-driven signal update algorithm is employed to simulate all hardware activities. Based on this algorithm, it provides a set of definition rules for the design target to resemble concurrent communication in real systems----not the function-call interface found in sequential programming languages such as C.
3. Based on the previous definitions, designers can describe the interface and function of any component, signal and instruction in the target design, then a pre-compilation-program will covert the description to proper C++ codes automatically to generate the target simulator, which completes the construction framework and ease the development.

Owing to THUMPSim, we implemented one cycle-accurate processor simulator fast when our research team began to design a MIPS 4KC (MIPS Technologies, Inc. 2002) like embedded processor. This simulator is an accuracy C-model of our CPU, which implements the micro-architecture of the processor and some basic peripherals. Based on this simulator, software engineers

Proceedings 19th European Conference on Modelling and Simulation
Yuri Merkuryev, Richard Zobel, Eugène Kerckhoffs © ECMS, 2005
ISBN 1-84233-112-4 (Set) / ISBN 1-84233-113-2 (CD)

began to start work on OS and applications in parallel to the development and manufacturing of the silicon components. In addition, the simulator plays a very important role in our validation and verification work. THUMPSim helps us to speed the validation of the functionality of our processor accurately, and most of the hardware and software development can be executed simultaneously.
The rest of this paper is organized as follows: The next section presents the related projects and shows what aspects are different in THUMPSim. The detailed design and implementation of the framework are described in section 3. Section 4 presents the brief overview of its usage for our CPU design and the relative workflow. Section 5 summarizes this paper.

RELATED WORK

Examples of some popular simulators include the SPIM simulators, the simplescalar simulator and the SuperDLX simulators. Once developed, such simulators are difficult to retarget to a modified micro architecture without a significant amount of effort. And these simulators lack the cycle accurate feature so they are unfit for our development.
In fact, a number of researchers met under the aegis of the US National Science Foundation's Computer Systems Architecture program to discuss the experimental and evaluation problems that processor architecture research faces in December 2001. The attendants agreed that simulation frameworks and simulator construction frameworks were superior to monolithic simulators or simulator code libraries written in sequential languages. In addition, modularity, portability and accuracy should be considered seriously (Adve et al. 2001). Some examples frameworks include Asim (Emer et al. 2002) and the Liberty Simulation Environment (Vachharajani et al. 2002).
Therefore, these issues were also evaluated firstly when we began to design THUMPSim. In our sense, it is should be a simulation framework rather than a single simulator. And it should own a component-based infrastructure and provide some construction tools for designers to ease the development. According to this policy, we design and implement the framework, which will be introduced in the following sections.

ARCHITECTURE OF THUMPSIM

Overview

We define accuracy as a measure of the fidelity of the simulated machine to the actual, while flexibility means the ability of the simulator to continue to explore a broad design space. As more and more features are modeled accurately, it becomes increasingly difficult to support design space exploration that strays far from the chosen direction. In contrast, flexibility in turn decreases as a consequence of increasing accuracy. To pursue both flexibility and accuracy, THUMPSim accomplishes a component-based infrastructure, which has the following features.
1. The framework core contains two parts, the architecture independent driven engine and the definition of the processor. So it is suitable for construction of different processors with high flexibility.
2. It gives object oriented component definition syntax. Based on the method, designers can describe some definitions on different levels for the target design. That is, the target can be simulated with different accuracies, including cycle-accuracy and functional accuracy.
As showed in Figure 1, THUMPSim contains two main parts, the simulation framework (gray modules) and the definition of the target design.

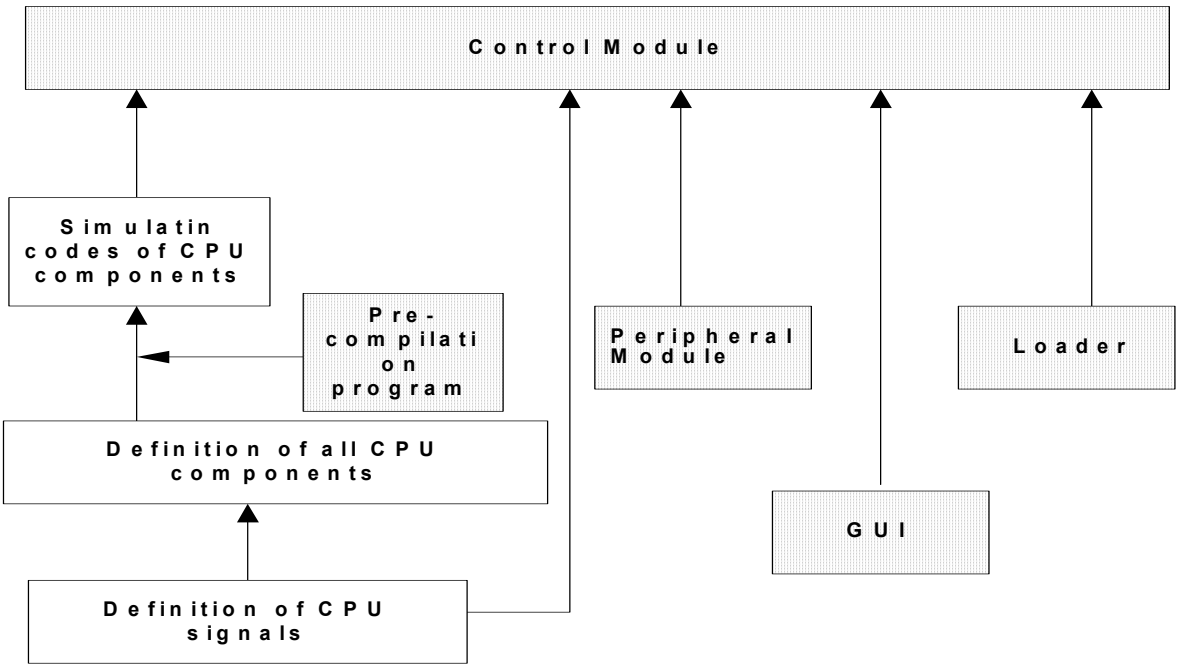

Figure 1: The Internal Modules of THUMPSim

Control module manages the whole system and schedules other modules to implement the driven algorithm. Loader can load OS or verification vectors into the simulator. And the key board, UART, terminal console and memory controller are simulated by peripheral module. It means different simulated designs can share the common modules, which increases the reusability of code. At the same time, to separate definitions of the target design from the framework improves its flexibility. That is, modification of some components, even of the whole architecture, can be implemented just through altering the definition of necessary signals and components. When the elementary architecture design is finished, designers can implement the simulation as the steps below.

1. Signals connecting processor components, inter-pipeline-stage registers and all instructions are declared according to the definition rule.
2. Processor components, including Decoder, ALU, MMU, DCache / ICache, BIU, Shifter, MDU, controller, CP0 and so on, are defined. Based on the predefined virtual interfaces in THUMPSim, the function and signals of one component can been described in a C++ like language.

Based on the description of interconnection signals and components, THUMPSim provides the tool to construct the target architecture, which will be introduced in 3.3. In fact, the original version of our CPU design was 5-stage and we implemented its simulator firstly. And then, we modified it into 7-stage, which only took us less than 3 weeks owing to the high flexibility of THUMPSim.

Definition Rules

One instruction definition sample looks like the following line, which describes the format of "ADD" instruction.

```
INST (I_ADD, "add r%d, r%d, r%d", RD(inst),
                RS(inst), RT(inst))
```

Instruction description is used to disassemble a decoded instruction to show it on the GUI. RD, RS and RS are macros to draw different subfields of one instruction.
For signals, the following format is employed:

```
SIGNAL_DEFINE(Name,Type, Width, Stage)
```

Name: the identification.
Type: wire signal or register signal.
Width: the data width.
Stage: the pipeline stage that the signal belongs to.

Two different types of signal are supported, wire signal and register signal. One type is just the connection between two or more components while the other is used to describe the inter-pipeline-stage registers or the general register file. And the stage info is only meaningful to register signals. Two typical signals are presented as follows.

```
SIGNAL_DEFINE(Alu_Out_EX_MEM_v32, ST_REG,
                32, STAGE_MEM)
SIGNAL_DEFINE(BranchPC, ST_WIRE, 32,
                STAGE_NONE)
```

The first is an inter-pipeline-stage-register, Alu_Out_EX_MEM_v32, that stores the 32-bit computing result of ALU component and the content of this register is used in MEM stage.
The second describes a 32-bit wire signal, BranchPC, which is the target of a branch instruction.
Definition of the logic of every component is stored in a distinct *.def* file, which looks like a C++ object declareation file.
The *.def* file is made up of five fields.

1. *Component* contains three sub fields:
   Name: the identification.
   Input :the set of its input signals
   Output: the set of its output signals
   All input / output signals should be declared before.
2. *Variable* describes the internal local variables.
3. *Function* defines the local functions used by *Execute* field.
4. *Initialize* gives initialization codes executed at the start.
5. *Execute* presents the function of this component, which is called by the driven engine when any input signal is modified.

Based on the previous definition, the whole net list of the target design can be described and one event-driven signal update algorithm is employed to drive all signals and components.
We should note that different levels of simulation can be implemented on the definition rules. For example, any stage of the target processor can be regarded as one single component to complete its function after the inter-pipeline signals have been defined. Of course, to divide the stage component into detailed components, including decoder, ALU, MMU, Cache and so on, can define more accurate descriptions with higher simulateion accuracy. The choice lies on the users' needs. General speaking, a rough simulation is required in the early stage while the more accurate will be needed later.

Code Auto-generation and Debugging Functions

THUMPSim provides a simulator construction framework to enable rapid exploration of design alternatives by automatically weaving architectural component models together as implied by the definitions of the design. When the definitions are completed, one pre-compilation-program coverts them to proper C++ code automatically. And then, the code can be linked to the other modules to create the target simulation executable. For example, the definition of ADDer is presented in Figure 2 and the converted code is listed in Figure 3. We can see that Adder has three input and three output signals. After conversion, the definition becomes a C++ class and all input/output signals are maintained in respective arrays, which will be used by the event-driven signal update algorithm in the next section.
Some useful debugging functions are provided in the executable. First, a round signal trace mechanism is supported. Behaviors of the target design in every cycle are logged, including memory & I/O accessing , register

read & write, cache miss, TLB miss and so on, which can been used to evaluated the performance of the design. Moreover, these traces can be compared with those from RTL-level simulation directly to locate bugs if existing. Second, instruction & data breakpoints are implemented. In addition, most signals and registers can been watched and modified on-line in THUMPSim.

```
[Component]
NAME = Adder
INPUT = Add_OpA_v32
INPUT = Add_OpB_v32
INPUt = Add_Op_Ctrl_DE_EX_v3
OUTPUT = Add_Out_v32
OUTPUT = Add_Flag
OUTPUT = Overflow_Exception
[Execute]
.....
```

Figure 2: The Definition of Adder

Adder.h:

```
class CAdder_COM : public CComponent{
        public:
                void Execute();
                CAdder_COM();
                ~CAdder_COM();
                bool Initialize();
        protected:
        };
```

Adder.cpp:

```
voidCAdder_COM::Execute()
{
……
}
CAdder_COM::CAdder_COM(){
        m_strComponentName =
"Adder_COM";
        m_nInputNum = 3;
        m_nOutputNum = 3;
        m_psgnInput=new
SIGNAL_ENUM[3];
        m_psgnInput[0]=SG_Add_OpA_v32;
        m_psgnInput[1]=SG_Add_OpB_v32;
        m_psgnInput[2]=SG_Add_Op_Ctrl_
DE_EX_v3;
        m_psgnOutput=new
SIGNAL_ENUM[3];
        m_psgnOutput[0]=SG_Add_Out_v32;
        m_psgnOutput[1]=SG_Add_Flag;
        m_psgnOutput[2]=SG_Overflow_Exc
eption;
}
Bool CAdder_COM::Initialize()
{ return TRUE; }
```

Figure 3: The Converted Code

Moreover, the status of the simulated target can be saved to a file. Then the status file will been loaded into THUMPSim to restore the previous state if necessary, which is very useful to debug a long term program or the OS.

Event-driven Signal Update Algorithm

As we know, the definitions of the target design will be converted into proper C++ code. It is a sequential programming language provides the function-call interface and is not suitable to describe the concurrent activities in real systems. To simulation the transformation of all signals updated by components concurrently, one signal update algorithm is implemented to simulate the hardware execution with high performance.

Event-driven mechanism is employed in the algorithm, that is, one component will be executed if and only if any of its input signals is modified in the current cycle. And our algorithm guarantees that no component will be called more than once in a cycle. To perform this algorithm correctly and rapidly, THUMPSim must generate the execution sequence of all components automatically. The following steps are performed just once at the start to make the sequence.

1. Attach all components to Group 0.
2. Attach all register signals to terminated signal list.
3. one variable, CurrentGroup, is set to zero.
4. Look up components that belong to CurrentGroup. If none, go to step 7. Otherwise for every matched component, increase its Group NO by one if no input signal belongs to terminated signal list.
5. Look up components that belong to CurrentGroup. If none, go to step 7. Otherwise attach all output signals of all matched components to terminated signal list.
6. Increase CurrentGroup by one and go to step 4.
7. End.

When the execution sequence is fixed, our event-driven signal update algorithm will be called once in every simulated cycle.

1. Empty the current execution list.
2. Browser all signals. If one register signal is modified in the previous cycle, attach all components that employ this signal as the input to the current execution list.
3. one variable, CurrentGroup, is set to zero.
4. Browser the current execution list. If no componet belongs to CurrentGroup, go to step 6.
5. If any, execute the component and check its output signals. If any has been modified, move all components that employ this signal as the input to the current execution list. Remove the executed component from the current execution list and go to step 4.
6. Increase CurrentGroup by one. If CurrentGroup is not greater than the number of groups, go to step 4.
7. Update all register signals and end!

Two values are maintained for every register signal. One saves the original value used by components in the current cycle and the other stores the new value generated. So to update one register signal in step 7 means that the new value will overlay the original at the end of this cycle.

The Workflow

For one instruction, its execution flow in THUMPSim can be described as the following basic steps.

1. It is fetched by the simulator, which is triggered by the modification of the content of PC register. During the fetch stage, instruction cache is accessed first. If cache misses, the memory main will be visited through BIU.
2. Then, decoder is triggered to parse the instruction and stores the result into the pipeline stages between DE and RF stages.
3. Register files are accessed if necessary to get the values of source registers, which will modify the relative signals to trigger execution components. So this instruction will be performed cycle by cycle till WB stage.

Of course, there are often several instructions running currently in the simulator and the potential interlock / hazard among them should be coped with by the pipeline control component.

We implement THUMPSim in MFC 7.0 and its GUI is showed in Figure 4. Window 1 displays the current instructions running in the processor and the pipeline stages they belong to. All internal signals and inter-pipeline-stage registers can be watched in windows 2. Window 3 is the register file view and window 4 is the memory view.

Now THUMPSim can load and run some types of verification vectors directly:

1. Executable files in ECOFF format, including the Linux OS kernel.
2. Other vectors written manually or generated by RTPG.

THE USAGE

THUMPSim is used in the design flow of an embedded processor developed by our team. The processor, THUMP107, is a MIPS 4KC like embedded processor. It was produced employing the TSMC 0.18um technology in October, 2003. And tests show that its highest frequentcy is 500MHz. The simulator played an important role in the design flow that is divided into four stages as showed in Figure 5.

During the design stage, the elementary micro-architecture specification was finished and designers implemented the simulator based on the specification. Then software engineers started to port OS for THUMP107 and develop applications in THUMPSim in parallel to the development and manufacturing of hardware components.

Many jobs, listed as follows, had been done in the simulation environment without any final hardware and prototype in hand.

1. Migrate OS
2. Test Spec2000 Integer benchmarks
3. Develop boot code for the system
4. Write and test most verification vectors
5. Implement the test bench of the whole system

In the co-design and co-verification stage, HDL codes were finished with the guidance of THUMPSim code. Verification vectors were also performed. It is necessary to note that all verification vectors, testing applications and OS suitable for the simulator can run on the RTL-level and the gate-level directly. That is, the c++ core of the simulator is replaced by the RTL core through PLI interface and the remaining parts, including basic peripherals, DRAM module, control module and the GUI are unmodified. Then we can operate the mixed test bench in the way of THUMPSim and the RTL simulator employed is Cadence NC-Verilog.

So testing traces generated by THUMPSim are able to be compared with those from the lower level simulation directly to locate bugs if existing. Its flow is presented

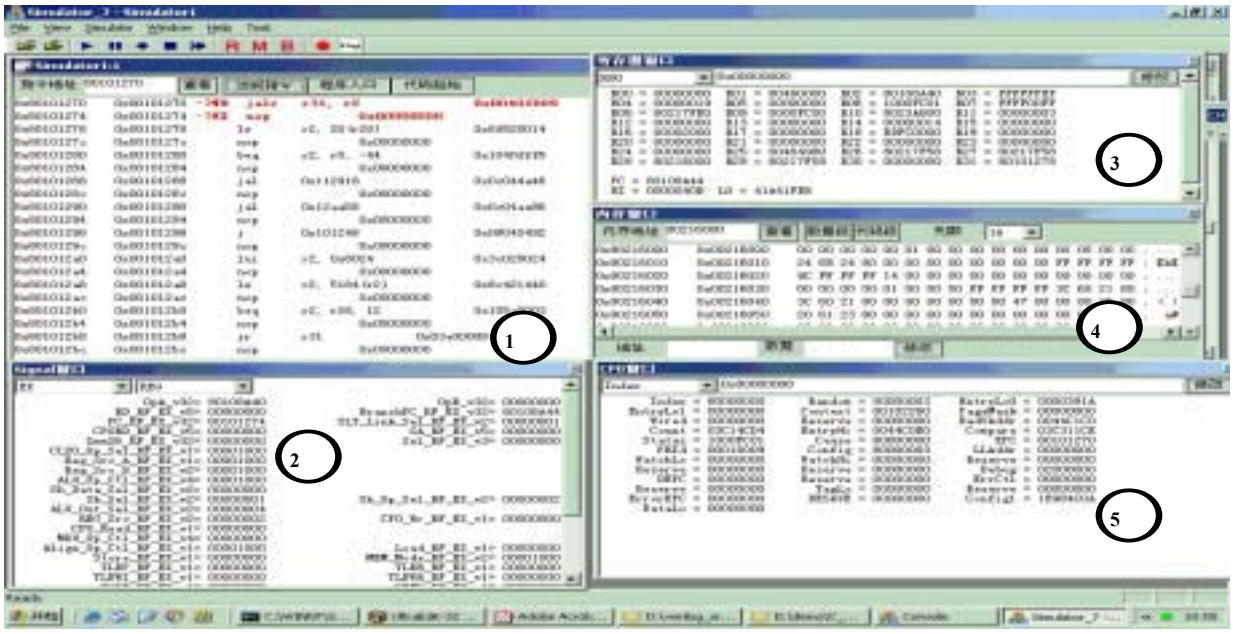

Figure 4. GUI of THUMPSim

briefly in Figure 6.
At last the post-synthesis prototype was tested under the same test bench. In addition to co-design and co-verification, software engineers are also optimizing the compiler for our CPU based on THUMPSim.

CONCLUSION

This paper presents one process simulation framework written in C++. It can provide different evaluation accuracies for the designer based on several simulation levels. This framework contains two main parts, the architecture-independent driven engine and the definition of the processor, to provide an infrastructure for pluggable components to improve its flexibility. In addition, an event-driven signal update algorithm is employed to simulate all hardware activities, which resembles concurrent communication in real systems.
It is employed by our team to develop THUMP107 embedded CPU and speed up the software/ hardware co-development remarkably. That is, before writing HDL codes, it has been developed for OS migration, writing verification vectors and validating the architecture design. In addition, we implemented a co-verification environment that contains the simulator. When testing vectors are produced, both of the c-simulator and the design simulator of the target CPU in different levels can execute them. So testing traces generated can be compared automatically to locate bugs if existing, which simplify the verification process.

REFERENCES

Adve, K et al. 2001. “NSF Computer Performance Evaluation Workshop: Summary and Action Items.” (December), http://www.princeton.edu/~mrm/nsf_sim_final.pdf.

Breach, S. E. 1999. “Design and Evaluation of a Multiscalar Processor.” PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, (Feb).

Burger, Doug and Austin, Todd M. 1998. “The SimpleScalar Architectural Research Tool Set, Version 2.0.” Technical report, UW-Madison Computer Architecture Group, University of Wisconsin-Madison. http://www.cs.wisc.edu / ~mscalar/simplescalar.html.

Emer, J. et al. 2002. “Asim: A Performance Model Framework.” *IEEE Computer*, (Feb), 68-76.

MIPS Technologies, Inc. 2002. “MIPS32 4kc Processor Core Software User's Manual.” http://www.mips.com.

Moura, C. 1993. “SuperDLX a generic superscalar simulator.” Technical Report 64, School of Computer Science, McGill University.

Skadron, K et al. 2003. “Challenges in Computer Architecture Evaluation.” *IEEE Computer*, Volume 36, No. 8 (Aug), 30-36.

Vachharajani, M. et al. 2002. “Microarchitectural Exploration with Liberty.” In the *35th International Symposium on Microarchitecture* (Istanbul, Turkey. Nov. 18-22). IEEE, 271-282.

AUTHOR BIOGRAPHY

YOUHUI ZHANG was born in Zehjiang, China and went to Tsinghua University in 1993. Now he is an Assistant Professor in the Department of Computer Science at the University of Tsinghua where he also received his PhD degree. His research interests are cluster fault tolerance, network storage/ computing, and microprocessor design. He is currently working on chip-multiprocessor project supported by the National High-Tech Program in China. Zhang can be reached at zyh02@tsinghua.edu.cn.