PAPER Special Section on VLSI Design and CAD Algorithm

Top-down Retargetable Framework with Token-level Design for Accelerating Simulation Speed of Processor Architecture

Jun Kyoung KIM[†a)], ***Regular Member***, **Ho Young KIM**[†b)], ***and*** **Tag Gon KIM**[†c)], ***Nonmembers***

SUMMARY This paper proposes a retargetable framework for rapid evaluation of processor architecture, which represents abstraction levels of architecture in a hierarchical manner. The basis for such framework is a hierarchical architecture description language, called XR^2, which describes architecture at three abstraction levels: instruction set architecture, pipeline architecture and micro-architecture. In addition, a token-level computational model for fast pipeline simulation is proposed, which considers the minimal information required for the given performance measurement of the pipeline. Experimental result shows that token-level simulation is faster than the traditional cycle-accurate one by 50% to 80% in pipeline architecture evaluation.

key words: *retargetable simulator, trace-driven simulation, token-level simulation*

1. Introduction

The development of leading processors has well been characterized by Moore's law[1]. Moore's law says that the amount of device integrated on a given silicon area doubles every year. This law has held until the late 1970s, but the doubling period has increased to eighteen month since late 1970s or early 1980s. Nevertheless, this law can give us conviction that the improvement of processor will continue.

Not only development of process technology but also progress of processor architecture has made an important role. Actually, without the improvement of process technology, architecture innovation can not be reflected due to shortage of area and excessive power consumption. But outstanding growth in process technology such as the deep sub-micron technology has enabled one to implement a very complex architecture on a single die. Therefore, the progress in processor architecture can be made without considering the physical integration ability.

Current needs require very complex architecture. For example, IXP1200, a network processor from Intel, adopts multi-processor architecture with seven processor cores in total, one StrongARM core and six microengines[2]. Each microengine is a multi-threaded processor for a zero-overhead context switching by supplying large register files which can hold up to 4 contexts at the same time.

Architecture innovation results in a large and complex architecture design space. But there are two obstacles which impede exploration of the design space. One is long evaluation time and the other is delayed development time of simulator for the new architecture. In designing a processor with a top-down approach from instruction set architecture to pipeline architecture, two simulations are often necessary: instruction set simulation and cycle accurate simulation. One can evaluate most performance indices necessary to select an architecture only by performing cycle accurate simulation. The amount of operations to be performed at cycle accurate simulation is far more than that at instruction set simulation. Thus, cycle accurate simulation takes much more time than instruction set simulation. In addition, various techniques can be applied to instruction set simulation such as compiled simulation[3]. Similar techniques can be applied to cycle accurate simulation, but no significant improvement in simulation time has been made by such techniques.

In top-down design of processor architecture, a higher level architecture can be synthesized into a set of lower level alternatives. More specifically, an instruction set architecture can be implemented by different pipeline architectures. Thus, an architecture evaluation step is such that for each instruction set architecture, a number of cycle accurate simulation for alternative pipeline architectures should be performed. This implies that improving simulation time for cycle accurate simulation significantly reduces architecture evaluation cost.

The purpose of this paper is to propose a retargetable framework to support top-down design of processor architecture with an emphasis on fast simulation of pipeline architecture. The basis for such framework is a hierarchical architecture description language, called XR^2, which describes architecture at three abstraction levels: instruction set architecture, pipeline architecture and micro-architecture. In addition, a token-level computational model for fast pipeline sim-

Manuscript received March 17, 2003.
Manuscript revised June 13, 2003.
Final manuscript received July 25, 2003.

[†]The authors are with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea.
a) E-mail: jkkim@smslab.kaist.ac.kr
b) E-mail: hykim@smslab.kaist.ac.kr
c) E-mail: tkim@ee.kaist.ac.kr

ulation is proposed, which considers the minimal information required for the given performance measurement of the pipeline.

The next section introduces the related works. Section 3 shows the overall structure of our framework including the background for the structure. The design stage for instruction set architecture will be shown at Section 4. The token-level model and simulation algorithm, which mostly differentiate our framework from others, is given at the Section 5. The effectiveness of our framework will be proved by exemplifying ARM9TDMI at the Section 6. In Section 7, we draw the conclusion and further work.

2. Related Works

Two contexts will be introduced and compared as related works. One is the retargetable framework, most of which adopts automatic simulator synthesis from processor description in ADL(architecture description language). The other is various techniques for fast evaluation of processor architecture.

2.1 Retargetable Framework

The most similar work with ours in terms of retargetable framework is the nML approach[4]. The Target Compiler technology[5] supports a retargetable framework with the name of CHESS/CHECKER which is based on the nML. The Sim-nML project, which is in progress at IIT Kanpur[6], also adopts nML as its ADL. The processor description of nML consists of two rules, mode rule for specifying addressing mode and operation rule for instruction description. Both rules can have syntax component for assembly code syntax, image component for binary encoding and uses component for resource utilization. In addition, operation rule has action component which contains the behavior to be executed for the operation. Retargetable tools such as retargetable assembler generator, retargetable dis-assembler, retargetable functional simulator, retargetable timing simulator, retargetable compiler and retargetable processor synthesizer are under development. It seems that many of them are in their early stages of development[7]. There are two levels of simulator, retargetable functional simulator and retargetable timing simulator. [7] reports that the speed of the functional simulator exceeds 1MIPS on host processor like 233MHz Pentium, but timing simulator is only about 4000 instructions per seconds, which is too slow to be used for large design space exploration.

EXPRESSION[8] is a lisp-style ADL for automatic generation of a retargetable toolkit. It supports mixed behavioral/structural representation of a target processor and parameterization for hierarchical memory subsystem. The structural information includes functional and storage units connected by pipeline and data transfer paths. The data transfer path is a path between functional unit and storage; pipeline path is a path from the very first stage to the last stage consisting of functional units and pipeline registers. It also supports a set of retargetable compiler and simulator. There are two kinds of simulators, an exploration simulator for fast design space exploration and a cycle-accurate simulator. In spite of the two kinds of simulator, much emphasis is on the cycle-accurate simulator that requires mixed behavioral/structural representation.

LISA[9] outperforms others in terms of simulation performance by introducing various techniques such as compiled simulation, dynamic scheduling and static translation. We will review this method later.

ISDL[10] is a machine description language designed to support automatic generation of an assembler, disassembler, code generator and instruction level simulator, especially for VLIW processors. A designer has only to describe operations and specify what operations can't be executed at the same time, to obtain the toolkit. But the method of specifying constraint is complex, therefore being easy to make errors. A timing simulator is not available. But in many cases, VLIW has very simple pipeline architecture, whose stages consume only one cycle. In addition, the compiler performs every scheduling for VLIW processors. Therefore, designer would be able to predict the performance metrics easily without timing simulation.

MDes[11] and Tensilica[12] attack the optimal architecture-finding problem by assuming a basic processor template, HPL-PD and Xtensa respectively, with parameterization and tuning it for application's sake. Such methods can predict the performance-related metrics such as power and area accurately, but the design space in which the processor architecture can vary is small. That is, the degree of retargetability is low.

2.2 Improving Architecture Evaluation Speed

Many approaches have been attempted to improve the evaluation cost of processor architecture. There is a tradeoff between evaluation cost and correctness of the evaluated result. A typical approach for accelerating the simulation of pipelined architecture is cycle-based simulation[13]. In contrast to event-driven simulation, this method fixes an execution sequence of entities that comprises the overall architecture, and invokes all the entities in that sequence. By removing the overhead of event scheduling, it can achieve fast evaluation. The more entities are active for each control step, the lower the evaluation cost is. LISA framework[9] tries to improve the simulation performance by moving instruction decoding, operation sequencing and operation instantiation to compile time.

Unlike simulation, analytic methods seek to achieve fast evaluation time with sacrificing evaluation correctness. [14] introduced a systematic sampling

Architecture	Design Stage	Information	Implementation/Technique	Objectives
ISA	HiXR^2	• Instruction behavior • Addressing mode behavior	• Retargetable compiler • Retargetable simulator with compiled simulation technique	• instruction/addressing mode correctness • (unoptimized) code size
Pipeline Architecture	Token-level LowXR^2	• Pipeline arch. : sequence of stages • Stage description : latency & resource connectivity (No behavior description of instructions and addressing modes)	• Retargetable simulator with - compiled simulation technique - trace-driven simulation - cycle-based simulation	• total cycle count • resource utilization • (rough)power estimation • (optimized) code size
	Cycle-true LowXR^2	• Cycle-accurate behavior of instruction and addressing modes	• Retargetable simulator with - compiled simulation technique - cycle-based simulation	• cycle true snapshot for debugging and implementation
Micro-Architecture	MicroXR^2	• HDL model for data- and control-path	• Synthesizer from XR^2 model to HDL model (not yet implemented)	• Implementation correctness •Power/area estimation

Fig. 1 XR^2 : Design Stages

method that utilized contiguous traces of data. The result showed about 15 times faster evaluation time than that of simulation by sacrificing the correctness by 13 percents. However there is a problem: how faithfully the sampled trace represents original data. [15] proposed an iterative sampling-verification-resampling technique, which measures the representative of partial data by instruction frequency, basic-block densities and cache statistics. With initial clusters selected at random intervals, iterative checks are performed if they represent the original benchmark. If not, new traces are inserted. This method could guarantee accuracy, but the sample is inclined to grow to the full trace. A few more works are reported in this context. Nevertheless, how closely a sample trace represents the original benchmark remains as a problem.

The proposed XR^2(eXtensible, Reconfigurable and Retargetable) framework provides top-down, hierarchical, seamless design flow for processor development. In addition, we introduced a new abstraction level for pipelined architecture with which one can obtain the performance metrics in a rapid and correct way.

3. XR^2 : Overview and Backgrounds

Considering the nature of processor design flow, we have concluded that the following properties are helpful to support a rapid exploration on a large design space.

- Hierarchical design: reducing design space at a high level of abstraction leads to great reduction of design space at low level of abstraction
- ADL(Architecture Description Language) based on sound semantics: sound semantics enables us to investigate and specify relationships between design parameters precisely. Furthermore, automatic parameter change can be performed more conveniently and in a systematic way, which enables automatic design space exploration
- Retargetable framework: delayed development for simulator for the architecture change is eliminated

We have concluded that the objective-driven modeling and simulation is the best choice for processor design[16]. Fig.1 shows the design stages which constitute our top-down design framework, the information required at each design stage and the performance parameter obtainable by simulation of the stage. The information necessary to implement each stage is enclosed to the same box with the corresponding stage. For example, we only need behaviors of instructions and addressing modes for $HiXR^2$, while we need extra data such as pipeline architecture and stage description for Token-level $LowXR^2$.

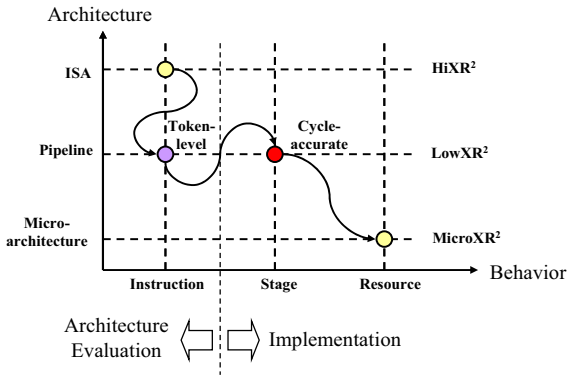


Fig. 2 Taxonomy for Dividing the Design Space

In devising our framework, we enumerated the simulation objectives. Information required to achieve each simulation objective determines the design stage. We divided the required information, which should be provided by a designer, into four design stages according to the following criteria.

- simulation speed issue: a design stage should be as high as possible for the objectives of modeling and simulation
- Seamless design issue: a design stage should not be too far from the both higher/lower neighbor design stage

- Information exchange between design stages should be well defined

The Resulting XR^2 framework has four design stages as in Fig.1. Three of the design stages, $HiXR^2$, cycle accurate $LowXR^2$ and MicroXR^2, exist in many other frameworks[4]. First, $HiXR^2$ is a design stage for instruction set architecture, which consists of instruction set and addressing modes. Fig.2 shows that behavior is described at per-instruction basis and architecture is instruction set architecture. This design stage enables us to determine what instructions and addressing modes are useful for a specific application. Cycle accurate $LowXR^2$ is a design stage with which we can model the cycle accurate behavior including all the operations on storage elements. In this design stage, behavior is defined for stages and architecture is pipelined architecture as in Fig.2. This model helps us determine a pipeline architecture including data path and control path. A disadvantage of this model is poor simulation performance. The last one, MicroXR^2, is an implementation stage. Generally, this consists of HDL models. Fig.3 shows the overall flow of our framework. The trace will be explained in Chapter 5.

In this work, we added an extra design stage called Token-level $LowXR^2$ which addresses the slow simulation speed with cycle accurate $LowXR^2$. Token-level $LowXR^2$, although it does not render a cycle accurate snapshot of storage elements, helps evaluating a candidate design in a fast way. We call this extra stage of simulation as token-level because this stage treats each instruction as a token which is passed between pipeline stages based on the following architecture data: latency, resource availability, data dependency and in-order execution guarantee. These data are often called static since the simulator can extract them from the pipeline architecture description unlike real values of storages which can be determined only at run-time. Clearly we observed that it is unnecessarily too inefficient to simulate these statically extractable data at cycle accurate simulation stage. Therefore, in our framework, the token-level simulator extracts these data by performing the control path cycle accurate simulation with the description of the static architecture and instruction set simulation. As shown later, we observed in our experiments that this reuses of static data at cycle accurate simulation stage dramatically reduce the overall simulation time.

We should make it sure that to rapidly evaluate processor architecture in terms of total cycle count, area and power estimation for large design space exploration, one needs not perform traditional slow cycle-accurate simulation. Instead, he(or she) has only to perform fast token-level simulation for that purpose.

The next two sections describe design stages, $HiXR^2$ and Token-level $LowXR^2$ respectively, in more detail. We will focus on the Token-level $LowXR^2$ because this is the major contribution of our framework. We will not explain the cycle accurate $LowXR^2$ in this paper because it is similar to cycle-accurate model in other frameworks.

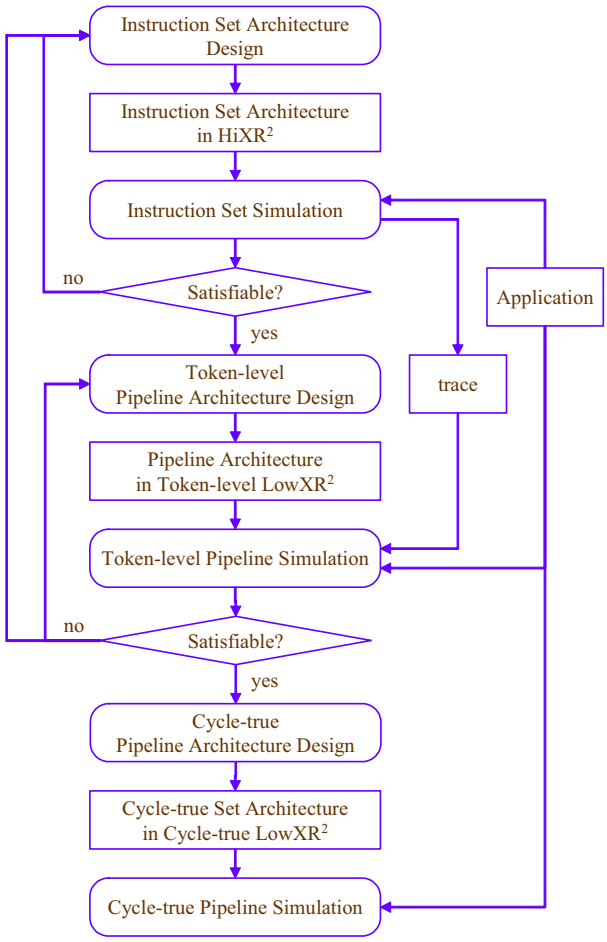


Fig. 3 Overall Flow of Modeling and Simulation in XR^2 Framework

4. $HiXR^2$: Overview and Backgrounds

4.1 Semantics : HiISA

This design stage defines the target processor with instruction set, addressing mode, storage set and two relations. The semantics of $HiXR^2$, HiISA, is defined as follows.

HiISA $= < IS, AM, ST, R_{IA}, R_{AS} >$, where

- IS: a structured set of instructions
- AM: a set of addressing modes
- ST: a set of storages of target architecture
- $R_{IA} \subseteq IS \times AM$: A relation between the instruction set and addressing modes
- $R_{AS} \subseteq AM \times \{ST \cup IMM\}$: A relation between

the addressing modes and associated storages and IMM.

- IMM is defined for the immediate addressing mode. This is handled separately because this has no relationship with storage elements

4.2 Semantics : Syntax and Example

Syntax is the textual representation of HiISA. It consists of three sections, each of which corresponds to the respective set of HiISA

```
Storage arm9x_Storage {
    Memory dmem {
        nummemcells 10000
    }
    RegisterFile GR {
        type INTEGER NumOfRegs 17 Naming r
    }
}
```

Fig. 4 Storage Specification

Fig.4 shows the storage description exemplified by the ARM9 processor. It consists of memory element and register files. Each register declaration has its type and naming. In addition, register file has the number of register slots.

```
AddrMode ASR_IMM : arm9x_AddrMode {
    operands {
        2
        GR Rm
        IMMEDIATE imm
    }
    syntax {   Rm ";" "ASR " imm  }
    action {
        temp_operand = arshift(Rm, imm);
        return temp_operand;
    }
}
```

Fig. 5 Addressing Mode Specification

Fig.5 shows the addressing mode description for ARM9. It has operand, syntax and action components. Operands section declares the number of operand and what storage is used including immediate. Action section is a C/C++ description of the addressing mode, which returns the reference of storage implied by the addressing mode.

Finally, the instruction description of $HiXR^2$ is shown at Fig.6. It consists of mnemonic section, operands section and action section. The body of action section is also a C/C++ description. Functions used within this section are implemented at external

```
Instruction add : ArithmeticOPs {
    mnemonic "ADD"
    operands {
        3
        REG Rd : DST
        REG Rn : SRC
        Oprnd2 Oprnd2 : SRC
    }
    action {
        Rd = add(Rn, Oprnd2);
        SetBit(CPSR, 29, carry) ;
        SetBit(CPSR, 28, overflow);
        if (Rd == 0)    SetBit(CPSR,30,1);
        else            SetBit(CPSR,30,0);
        SetBit(CPSR, 31, GetBit(Rd,31));
    }
}
```

Fig. 6 Instruction Specification

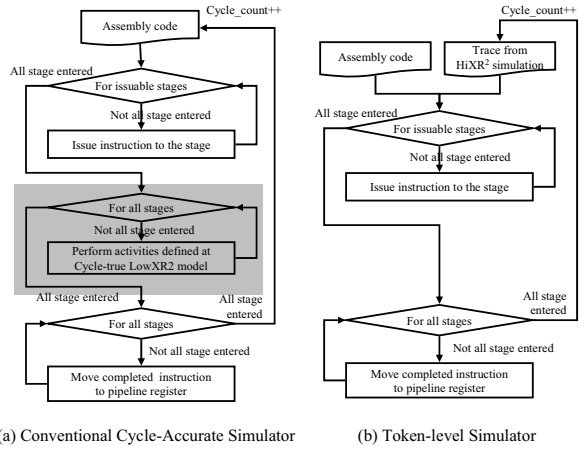
file. Encoding information is optional because we support simulation with assembly code. We leave how the instruction and addressing modes are encoded as a further work because adequate algorithm for it will help power and area optimization.

5. Token-level $LowXR^2$: Pipeline Architecture

5.1 Brief Overview

Fig.7(a) shows the flow chart of traditional cycle-accurate simulator. We profiled a cycle-accurate simulator for g.721 benchmark from MediaBench. The exemplified architecture is ARM9TDMI processor. Table1 illustrates the relative time spent for important processes in cycle-accurate simulation.

The most time-consuming task is the data-path behavior executed at a stage by an instruction for each phase(the shaded portion of Fig7.(a)). Token-



(a) Conventional Cycle-Accurate Simulator (b) Token-level Simulator

Fig. 7 Difference between Cycle-accurate Simulation and Token-level Simulation

Table 1 Profiling the Cycle-accurate Simulation for G.721

	sec. spent	% spent
Condition check if instruction can enter a stage	0.05656	5.013
Initialization performed when instruction is issued to a stage	0.06019	5.335
Data-path behavior for instruction & addressing mode	0.54497	48.301
clean-up(instruction in stage → pipeline register)	0.07688	6.814
OS overhead(API and function call)	0.38968	34.538
Total	1.12828	100

level $LowXR^2$ enables a rapid evaluation of processor architecture by eliminating this overhead of data-path processing. The resulting simulator has the form of Fig.7(b). As the figure describes, the token-level simulator does not perform any datapath operation which is implied by an instruction such as addition, subtraction, multiplication, load, store etc. This kind of operation occupies much portion of simulation time because every instruction has its operation to execute.

If we assume that scheduling of an instruction within pipeline consists of (i) issuing instruction to the first stage, (ii) passing instruction in one stage to the next stage, (iii) determining stall on data hazard/resource conflict and (iv) flushing the pipeline for branching instruction, the following information is sufficient to perform scheduling.

- Pipeline architecture, defined as a sequence of stages
- Data transfer path between stages
- Latency information for stages
- Stage-resource relationship
- The target addresses resolved for branch instructions

With the Token-level $LowXR^2$, one can model and parameterize the above architectural choices. We will see the semantics, syntax and simulator algorithm in more detail.

5.2 Semantics: B-PASS

B-PASS $= < HiISA, STAGES, RES, pipe, resource, lat >$ where

- $STAGES$: set of pipeline stages
- RES : set of resources
- $pipe : IS \times AM \rightarrow STAGES^n$ for positive integer n
  - The pipeline architecture is defined as sequence of elements of set STAGES and a function of instruction set and addressing modes.
- $resource : IS \times AM \times STAGES \rightarrow RES$
- $lat : STAGES \times IS \times AM \rightarrow I \times I$ where I is a set of non-negative integer.
  - Stage has latency information: issue/result latency.

B-PASS(Basic Pipeline Architecture System Specification) formalism defines pipeline architecture in a hierarchical way from HiISA. STAGES is a set of stages which constitute pipeline architecture. RES is a set of resources. pipe is a function that maps instruction set and addressing mode to a pipeline architecture which is a sequence of stages. We can model what resources are used at what stage, depending on an instruction and addressing mode with resource function. Latency information is specified with lat function. The first integer n is an issue latency, which implies that the stage can receive a new instruction every nth clock cycle. The second integer n is a result latency, which implies that n clock cycles are necessary for an instruction to finish its job at the stage.

5.3 Syntax

The syntax of Token-level $LowXR^2$ consists of five sections: general section, instruction aliasing section, pipeline section, resource section and stage section. Fig.8(a) shows the pipeline section. This defines the pipeline architecture of all the instruction with the sequence of stages. This implies the data-path implicitly. Resource section defines the type and the number of resources used at the architecture as Fig.8(b).

```
<PIPELINE>
ALL_INSTS
{ (IF, ID, EX, MEM, WB); }
</PIPELINE>
```

(a) Pipeline specification

```
<RESOURCE>
ALU;
RPORT : 2;
...
</RESOURCE>
```

(b) Resource specification

Fig. 8 Pipeline and Resource Specification

Fig.9 shows the stage section description exemplified by ARM9 processor. One can model and parameterize various architectural choices at stage section easily as in the figure. There are five stages, each of which has issue latency and result latency. This example shows only the single-cycle stages. Multi-cycle stage can be modeled by assigning adequate integers to issue latency and result latency parameters. Most of the current off-the-shelf processors use the two phase activities. The phase keyword specifies at which phase the described activities would be performed. For example, at the first phase, which is defined as the time

```
<STAGE>
IF {
    issue latency = 1; result latency = 1;
    phase(0){ use PC; }
    switch(inst){
        case(BRANCH_INSTS):
            phase(0){ lock PC; }
    }
}
ID{
    issue latency = 1; result latency = 1;
    phase(0){ fetch_opds; lock_dest; }
}
EX{
    issue latency = 1; result latency = 1;
    use ALU;
    switch(inst){
        case(BRANCH_INSTS):
            phase(0){ unlock PC;}
        case(ALU_INSTS):
            phase(1){ bypass_to EX; }
    }
}
MEM{
    issue latency = 1; result latency = 1;
}
WB{
    issue latency = 1; result latency = 1;
    phase(0){ unlock_dest; }
}
</STAGE>
```

Fig. 9 Stage Specification

the clock goes high, PC is unlocked for the branch instruction. At the second phase, which is the time the clock does down, the results are bypassed to the EX stage in case the instruction is a kind of ALU instruction. switch statement is introduced for modeling convenience. use, lock, unlock keywords are for modeling resource connectivity to stages. The keyword use specifies resource utilization for single cycle. lock and unlock keywords specifies a resource utilization encompassing the boundary of stages.

In addition, some syntactic sugars specialized to processor architecture are supported. For example, with keywords fetch_opds, lock_dest, unlock_dest and bypass_to can we model the data path of processor. When keyword lock_dest is specified at a stage, it implies that the destination register should be locked at the stage to declare this register is unavailable. Keyword unlock_dest performs the unlocking of the destination register. Keyword fetch_opds implies that operand fetching is done at the stage. If the registers which contain the source operands are locked by other instruction, it should wait till the registers become available. Keyword bypass_to specifies the forwarding logic. This keyword has a stage argument to which the result, obtainable by decoding the assembly code, is directly forwarded to. With these keywords and simple scoreboarding algorithm[17] can we model and simulate the data hazard easily. There are a few keywords for modeling convenience such as switch or case, but all the keywords for modeling data-path of a processor are described above.

The behavior of an instruction is not specified because the simulation at this abstraction level does not perform the evaluation of register value at all. In other words, only information about control-path is specified. This is helpful in simplifying not only the modeling but also retargeting a processor.

5.4 Token-level $LowXR^2$ Simulation

The simulation algorithm of Token-level $LowXR^2$ consists of the following three steps:

for every stage

- (step1) to check the condition if an instruction can enter a stage
- (step2) to initialize simulation-related parameters if condition of (step1) is evaluated to be true
- (step3) to cleanup the parameter value when a stage has completed instruction

One cycle is consumed between (step3) and (step1).

5.4.1 Conditions to passing instructions between stages

Each stage model has issue latency count and result latency count for simulation. They are initialized with their corresponding latencies when an instruction enters the stage(step2). The count values decrement by one to zero each time a clock cycle passes.

The conditions for an instruction to enter a stage are depicted in Fig.10(a) and Fig.10(b). Fig.10(a) shows the issueing condition of an instruction to the first stage of a pipeline. First, issue latency count should be zero, which implies that at lease the amount of issue latency cycles has passed after the previous instruction enters the stage. The second condition is that the resource should be free. This resource handling is used not only for checking resource conflicts but also power estimation. Power estimator exploits the number of accesses to resources. Fig.10(b) shows the conditions of an instruction to proceed from a stage to its next stage.

In addition to the two conditions of the previous case, two more conditions should be considered. For a stage where operands are fetched, the source operand registers should be available; that is, no previous instruction has declared the register to be its result register and has not written to the register yet. When an instruction goes to a stage where operand fetch should be performed, it first checks if the source operand is available by looking into the scoreboard bit. If available, it can proceed to the next stage when its result

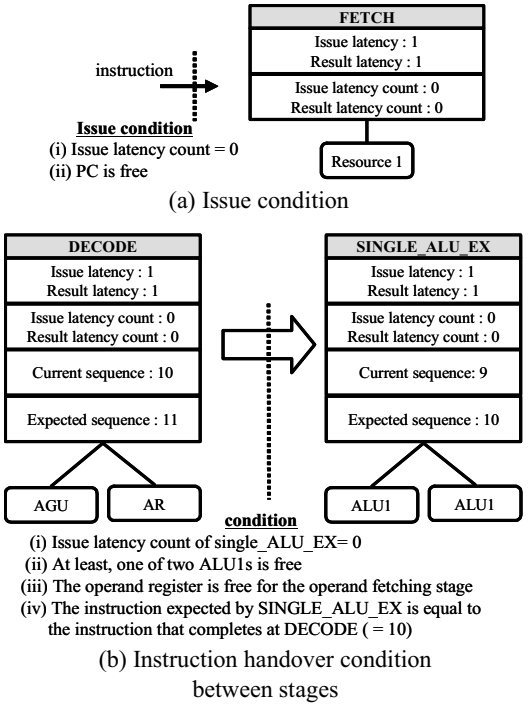


Fig. 10 Conditions for Instruction Proceeding

latency is zero. If not, it should wait for the instruction that locked the register to unlock it. Dependency checking of implicitly used registers such as status register is also handled. When an instruction enters a stage where it is specified to lock the destination, it should lock the destination.

The fourth condition is for guaranteing in-order execution as in Fig.11. Two variables are managed by the token-level simulator, current sequence and the expected sequence. Current sequence implies the sequence number of instruction which resides in the stage. Expected sequence of a stage implies what instruction the stage expects to come. When the stage s3 has two ancestor stages s1 and s2, this condition decides which instructions at s1 and s2 will be issued to the stage s3(Fig.11). By setting the expected sequence of s3 with minimum execution sequence of s1 and s2, the simulator can decide which instruction can proceed to s3 next time. Earlier issued instruction enters a stage earlier in case of conflict.

5.4.2 Resolving ambiguous target address:trace-driven simulation

We have shown how our Token-level $LowXR^2$ simulator handles the model without evaluating the true values of storage elements. However, branching instruction, which modifies the program counter directly based on result of comparison instruction, causes ambiguous target problem as in Fig.12.

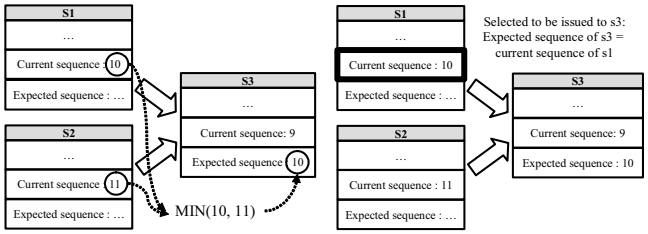


Fig. 11 In-order Execution Guarantee

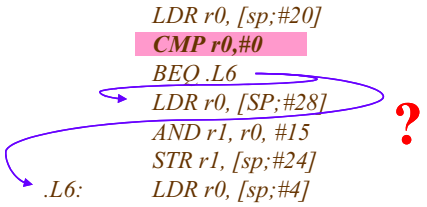


Fig. 12 Inability to Predict Target Address

Because our simulator does not evaluate the real value for r0 register in Fig.12, it can not determine the target address. We solved this problem by introducing the trace-driven simulation. A trace is a real execution sequence of a program. We can obtain it with a fast instruction set simulation as in Fig.3. In other words, we can exactly know all the target addresses of branch instructions by performing fast instruction set simulation. To reduce the trace file size, the following format is introduced.
(Program counter of branch instruction, target address)

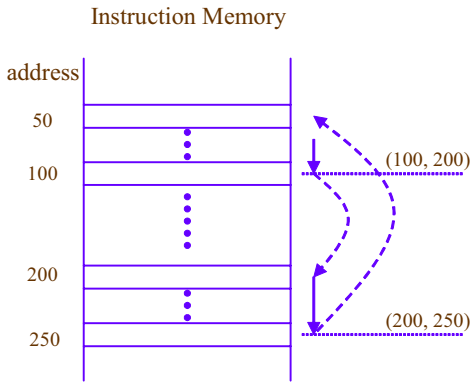


Fig. 13 Treatment of the Trace Information

Fig.13 shows how our simulator treats the trace. $(100, 200)$ implies that when simulator fetches instruction located at memory address 100, the instruction is a branch instruction and the simulator should fetch the instruction located at the memory address 200. If the next trace is $(250, 50)$, the simulator jumps to memory

address 50 when the simulator meets the instruction of location 250

Simulations for different pipeline architectures, which have the same instruction set architecture, can share only one trace, if they can be abstracted to the same instruction set architecture. This is an advantage of top-down framework. In case that an instruction set architecture does not meet constraints, one should introduce new instructions or delete useless instructions. To investigate the correctness, code density and instruction mix with the new instruction set architecture, one should perform instruction set simulation. With the trace obtained by the simulation can one evaluate different pipeline architectures, too.

6. Experiment

We experimented with ARM9TDMI to show how the token-level simulation reduces the exploration time. ARM9TDMI is a high performance Thumb compatible processor, to provide a performance upgrade path from the ARM7TDMI. We used ADPCM, IDCT and g.721 benchmarks which are members of MediaBench. The experiment is done using AMD AthlonXP 2100+, whose operating system is Windows XP.

We compared the performance of instruction set simulator, token-level pipeline simulator and cycle accurate simulator as shown in Table 2. The simulation results are presented in terms of MIPS(million instructions for second) and MCPS (million cycles per second).

We applied compiled simulation technique to our $HiXR^2$ simulator, which results in simulation performance above 20MIPS. Compared to that, the performance of pipeline simulator is inferior because of considerable amount of processing for synchronizing all the stages.

The difference between cycle-accurate simulation and token-level simulation results from additional data path operations as depicted in Figure 7. The additional data path operations include (i) calculating memory address, (ii) propagating data values between stages via pipeline registers and (iii) performing operation defined for each instruction. For fair comparison, same techniques are applied to each simulator.

Then, we will see how the time required for design space exploration can be reduced under the following assumptions:

- Instruction set architecture is fixed
  - 1 instruction set simulation is necessary
- After n pipeline simulations, we can find satisfiable design
  - n pipeline simulations should be performed
  - Average x number of cycles are executed
  - Average y number of instruction are executed

Then the time required to explore a design stage using traditional cycle accurate simulation is as follows:

$$t_1 = n \times \frac{x}{MCPS_{cycle\_true}} (\text{in } 10^{-6} second)$$

The exploration time proposed in this paper requires only one instruction set simulation and n token-level simulations. This is formalized as follows:

$$t_2 = \frac{y}{MIPS_{ISS}} + n \times \frac{x}{MCPS_{token\_level}} (\text{in } 10^{-6} second)$$

Then the speedup is as follows.

$$\begin{aligned} speedup = \frac{t_2}{t_1} \times 100 &= \frac{\frac{y}{MIPS_{ISS}} + n \times \frac{x}{MCPS_{token\_level}}}{n \times \frac{x}{MCPS_{cycle\_true}}} \times 100 \\ &= \left(\frac{y \times MCPS_{cycle\_true}}{n \times x \times MIPS_{ISS}} + \frac{MCPS_{cycle\_true}}{MCPS_{token\_level}}\right) \times 100 \\ &\approx \frac{MCPS_{cycle\_true}}{MCPS_{token\_level}} \times 100 \end{aligned}$$

For example, we expect 76.5% speedup in exploring the design stage with adpcm benchmark. We expect 68.7% and 57.1% speedup for idct and g.721 respectively.

7. Conclusion

In this paper, we have shown that the time required to explore a large design space can greatly be reduced due to the two factors. One is retargetable framework which renders us simulator fast by reconfiguring the machine descriptions. The other is the token-level simulation which enables us to evaluate each candidate design of pipeline architecture in a rapid way. The hierarchical framework is found to be a good solution for the objective-driven modeling and evaluation of a processor.

Formal specification of design space is necessary for design space construction and exploration. To be a useful environment for a design space exploration, adequate domain knowledge should be reflected in a formal way. More flexible debugging technique is possible such as debugging-from-the-middle. To investigate the behavior of a processor at the nth cycle, there is no other way except performing slow cycle accurate simulation to the nth cycle. But by combining fast instruction set simulator and fast token-level simulator, we can jump to the nth cycle very fast. From the (n - (number of stages of longest pipeline))th cycle, the cycle accurate simulator starts to run with memory managed by instruction set simulator. These are reserved as further work.

References

[1] Moore, G., "Progress in Digital Integrated Electronics," *IEEE International Electronic Devices Meeting*, 1975.

[2] Tom R. Halfhill, "Intel Network Processor Targets Router", *Microprocessor Report*, vol.13, Sep., 1999.

[3] R. Luepers, J. Elste, and B. Landwehr. "Generation of interpretive and compiled instruction set simulators." *in Pro-*

Table 2 Simulation Performance

	ISS	Token-level Simulation		Cycle-accurate Simulation	
	MIPS	MIPS	MCPS	MIPS	MCPS
adpcm	23.45	2.117	4.947	1.210	2.803
idct	29.045	3.192	4.753	1.924	2.818
g.721	21.761	2.897	4.559	1.847	2.901

ceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC) Jan. 1999.

[4] Fauth, A., Van Praet, J., Freericks, M. "Describing instruction set processors using nML." *the Proceedings of the European Design and Test Conference*, Mar. 1995.

[5] http://www.retarget.com/

[6] http://www.cse.iitk.ac.in/sim-nml

[7] Moona, R., "Processor Models for Retargetable Tools," *in Proceedings of Rapid System Prototyping*, 2000.

[8] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt and Alex Nicolau,"EXPRESSION: A language for architecture exploration through compiler/simulator retargetability". *in the Proceedings of Design Automation and Test in Europe*. 1999.

[9] A. Hoffmann, A.Nohl, G.Braun, O.Schliebusch, T.Kogel and H.Meyr, "A novel methodology for the design of application specific instruction set processors using a machine description language" *IEEE Transactions of Computer-aided Design and Integrated Circuits and Systems*, November 2001

[10] George Hadjiyiannis, "ISDL: instruction set description language, Version 1.0" *SPAM, MIT RLE*, November, 1998

[11] Bhuvan Middha et.al., "A Trimaran based Framework for Exploring the Design Space of VLIW ASIPs with Coarse Grain Functional Units," *International Symposium on System Synthesis*, October 2002, Kyoto, Japan.

[12] Ricardo E. Gonzalez, "XTENSA: A Configurable and Extensible Processor" *IEEE Micro* March-April, 2000.

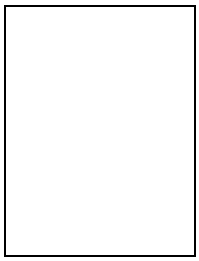
[13] Z.Barzilai, J.L. Carter, B.K.Rosen and J.D.Rutledge, "HSS : A High-Speed Simulator", *IEEE Transactions on CAD/ICAS*, pp.601-617, July, 1987

[14] T.M.Conte, "Systematic Computer Architecture Prototyping". Ph.D thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1992.

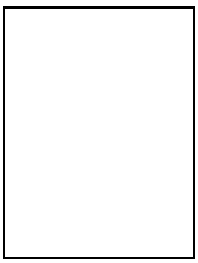
[15] G.Lauterbach. "Accelerating Architectural Simulation by Parallel Execution" *in Proc. of 27th Hawaii International Conference on System Science* Maui, HI, Jan. 1994

[16] J.K.Kim and T.G.Kim, "Trace-driven Rapid Pipeline Architecture Evaluation Scheme for ASIP Design" *in Proc. of Asia South-Pacific Design Automation Conference*,Kitakyushu, Japan, pp.129-134, Jan, 2003

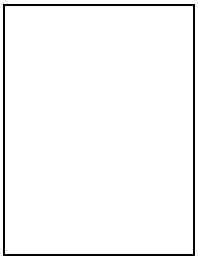
[17] Deszo Sima, Terence Fountain and Peter Kacsuk. Advanced Computer Architectures - A Design Space Approach, ADDISON-WESLEY, 1997



Jun Kyoung Kim received the B.S. degree in Electronic Engineering from Yonsei University in 1997 and M.S. degree in Electrical Engineering from Korea Advanced Institute of Science and Technology in 1999, respectively. From 1999 till now, he stays at Systems Modeling Simulation Laboratory (SMSLAB), in KAIST. His research interests include discrete event systems modeling/ simulation, processor design, co-design and processor description language. His e-mail address is jkkim@smslab.kaist.ac.kr



Ho Young Kim received the B.S. degree in Electronic Engineering from Korea Advanced Institute of Science and Technology in 1999 and M.S. degree in Electrical Engineering from Korea Advanced Institute of Science and Technology in 2001, respectively. From 1999 till now, he stays at Systems Modeling Simulation Laboratory (SMSLAB), in KAIST. His research interests include discrete event systems modeling/ simulation, processor design, cycle/power estimation and processor description language. His e-mail address is hykim@smslab.kaist.ac.kr



Tag Gon Kim received his Ph.D. in computer engineering with specialization in systems modeling/simulation from University of Arizona, Tucson, AZ, 1988. He was a Full-time Instructor at Communication Engineering Department of Bookyung National University, Pusan, Korea between 1980 and 1983, and an Assistant Professor at Electrical and Computer Engineering at University of Kansas, Lawrence, Kansas, U.S.A. from 1989 to 1991. He joined at Electrical Engineering Department of KAIST, Tajeon, Korea in Fall, 1991 as an Assistant Professor and has been a Full Professor at EECS Department since Fall, 1998. His research interests include methodological aspects of systems modeling simulation, analysis of computer/communication networks, and development of simulation environments. He has published more than 100 papers on systems modeling, simulation and analysis in international journals/conference proceedings. He is a co-author (with B.P. Zeigler and H. Praehofer) of Theory of Modeling and Simulation (2nd ed.), Academic Press, 2000. He was the Editor-in-Chief of SIMULATION:Trans of SCS published by Society for Computer Simulation International(SCS). He is a senior member of IEEE and SCS and a member of ACM and Eta Kappa Nu. Dr. Kim is a Certified Modeling and Simulation Professional by US National Training Systems Association