**SETIT 2007**
4th International Conference: **S**ciences of **E**lectronic, **T**echnologies of **I**nformation and **T**elecommunications
March 25-29, 2007 – TUNISIA

# A SystemC Transaction Level Model for the MIPS R3000 Processor

Yon Jun Shin and Sofiène Tahar

*Concordia University*

*1455 de Maisonneuve West, Montreal, Quebec H3G 1M8, Canada*

**{yon shin,tahar}@ece.concordia.ca**

Ali Habibi*

*MIPS Technologies*

*1225 Charleston Rd., Mountain View, CA, 94041, USA*

**habibi@mips.com**

**Abstract:** Processor cores in embedded applications build today the cornerstone of System-on-Chip designs. Among the most successful RISC (Reduced Instruction Set Computer) cores are the MIPS processors used in applications such as DVD, automotive, broadband access, networking, etc. In this paper, we design and verify a Transaction Level Modeling (TLM) architecture of the MIPS R3000 in SystemC. The TLM in SystemC is adopted so that abstract data types can be used for higher (abstract) level modeling and faster simulation. The processor is implemented such that the instruction and data caches contain all the necessary instructions and data to eliminate complex memory access management, respectively. To simulate the processor, we provide a script that automatically generates the machine instruction code from assembly language.

**Key words:** SystemC, MIPS Processor, Transaction Level Modeling.

## INTRODUCTION

System-on-Chip (SoC) is a popular technology used in embedded applications these days. Electronic components that are laid out on a PCB (Printed Circuit Board) to make systems not too long ago can be now integrated in a single piece of silicon to be produced in a chip. As a system is developed on a single chip, it not only requires hardware components but also software components that have to be developed. Hence hardware description languages such as VHDL or Verilog show their limitations when used in SoC development. Alternative languages for integrating both hardware and software development were sought after and several languages have been introduced such as SystemC [Gro02] and SystemVerilog [Sys06]. But recent acceptance of SystemC as an IEEE standard [IEE05], it became the premier choice for SoC development language.

SystemC was developed by Open SystemC Initiative (OSCI) on top of C++ which is a mature and one of the widely used software development languages. With the modeling of hardware behavior as library in C++, both software and hardware can now be modeled in a single language, making it easy to simulate and test a system in early stages of the design cycle. Due to rapidly changing product cycles and required time to introduce in the market, SystemC, as an IEEE standard, becomes even more attractive language when it comes to system-level design due to reuse of intellectual property (IP) blocks.

The basic idea of Transaction Level Modeling (TLM) is to establish communication through function calls that represent transactions rather than signals as at the Register Transfer Level (RTL). In this paper, we adopt the OSCI interpretation of TLM including Programmers View (PV), which contains no timing; Programmers View with Timing (PVT), which adds timed protocols and can analyze latency or throughput; and Cycle Accurate, which is accurate to the clock edge but does not model internal registers [OSI06].

MIPS processors are very popular processor cores in SoC applications due to their effectiveness in terms of simplicity, processing power, and low power consumption. In this paper, we design and verify a TLM architecture in SystemC of the MIPS R3000 processor. As a high level abstract model, TLM enables both hardware and software to be developed

* This work was done while the first author was at Concordia University

concurrently at an early design stage hence making it an effective way for new product development.

There can be found in the open literature two related work on SystemC modeling of MIPS processors. For instance, the company CoWare has developed a library of SystemC based models for Electronic System Level (ESL) design [CoW06]. The library contains several processors from the MIPS and ARM families. The second is a work of Madsen et al. [Jor03], who developed a MIPS R2000 processor in SystemC and implemented it in Xilinx Spartan II FPGA. Their SystemC model was at the Register Transfer Level (RTL). In contract to the above, in this paper, we are interested in a TLM design for the processor that can be used either as a testbench or a model to guide the implementation of a synthesizable core MIPS R3000.

The rest of the paper is organized as follows. Section 1 introduces the MIPS R3000 architecture. Section 2 describes details of the MIPS R3000 SystemC TLM design steps. Section 3 presents experimental results we conducted for evaluating the model. Finally, Section 4 concludes the paper.

## 1. MIPS R3000 Processor

MIPS R3000 processor is a 5 stage pipelined processor that implements 32-bit MIPS instruction set architecture (ISA). It was introduced in 1988 and contained 0.11 million transistors on 66.12mm2 die size with 1.2 µm process technology [HP02]. It also contained instruction cache and data cache with sizes of 64k byte each and was running between 20 to 40MHz consuming 4W of power. Due to small power consumption and heat characteristics of embedded MIPS implementations, low cost, widely available development tools, and simple architecture, etc., this processor is a good candidate to be used in SoC.

| Type | 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|---|
| R | opcode (6) | rs (5) | rt (5) | rd (5) | shamt (5) | function (6) |
| I | opcode (6) | rs (5) | rt (5) | immediate (16) | | |
| J | opcode (6) | address (26) | | | | |

**Figure 1. MIPS R3000 Instruction Format**

### 1.1. MIPS Instruction Format

MIPS instructions have 3 different formats, namely R, I, and J type instructions. Figure 1 shows the format of each 32 bit instruction type [HP02]. R-type instructions access two general purpose registers rs, rt and use them as operands of ALU (Arithmetic and Logic Unit) operations and save the result back to rd. These instructions require not only opcode but also function field to further specify the operations. Most of the R-type ALU instructions have “000000” as opcode field deferring the operation decision to the function field.

I-type instructions take immediate constant value as one of the operands. The result of the ALU operation is saved back to rt as well.

J-type instructions, which is Jump instruction, uses 26 bit address field to get the target address.

### 1.2. MIPS Pipeline Stages

Each instruction is stored in instruction cache in the order of execution and fetched at every clock cycle. It goes through 5 pipeline stages to be completed. The 5 stages that each instruction goes through are: instruction fetch (IF), instructions decode (ID), execute (EX), data memory (MEM), and write-back (WB).

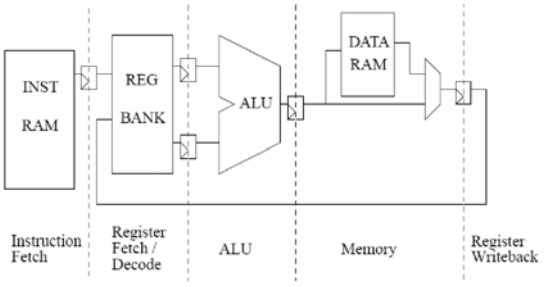


**Figure 2. MIPS Pipeline Structure [Bre02]**

The Instruction Fetch stage fetches the instruction from instruction memory (instruction cache) in the address given by the program counter (PC) and updates its value for next instruction. Fetched instructions are then passed on to the ID stage.

The Instruction Decode stage decodes the instruction and accesses the General Purpose Register (GPR) for the operands (two for ALU operations) in the EXE stage. Also immediate values are sign-extended to be 32 bit vector and Jump target address and Branch condition as well as target addresses are determined in this stage. If the branch condition is met, then the program counter will be updated with the target address and the new PC will be used as next instruction address to be fetched.

The Execute stage "executes" the instruction. In fact, all ALU operations are done in this stage. The ALU is the Arithmetic and Logic Unit and performs operations such as addition, subtraction, logical AND, logical OR, etc. In the EXE stage, the address for memory access for load or store instructions is also calculated.

The Memory Access stage performs any memory access required by the current instruction. So, for loads, it would access a memory location and load the value onto GPR. For stores, it would store an operand into memory address specified in the instruction.

The Write-Back stage writes the result of instruction back to register file. Careful attention is needed to write the register file before it is read by another instruction.

Figure 2 provides a general description of how the 5 stage pipeline is structured. Figure 3 shows how the sequence of instructions is executed through the pipeline in each cycle. The main advantage of pipeline architecture is that theoretically at each clock cycle, each pipeline block is processing an instruction. Therefore the throughput of the processor becomes 1 at each clock cycle ignoring the initial latency to fill up the pipeline for 5 clock cycles.

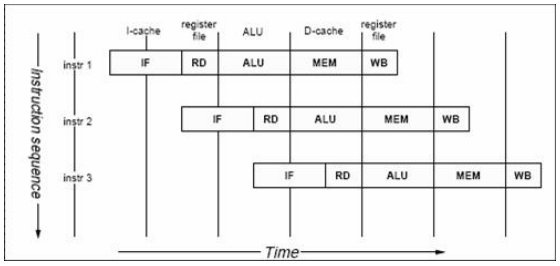


**Figure 3. MIPS Instruction Execution Sequence in Pipeline [HP02]**

## 2. TLM SystemC Design

### 2.1. SystemC Transaction Level Modeling

TLM in SystemC is motivated to provide early system model platform for software development. Without such platform for software in the early stage of system development, software and hardware have to be developed side by side and they can only be put together for testing/verification after the prototype system has been produced. By using higher abstract TLM, which functionality is presumably correctly modeled, system verification can be started early in the development stage. TLM also emphasizes on functionality rather than actual implementation, it enables faster simulation speed than pin-based model [Ghe05].

First, two classes of SystemC modules (*sc_module*), *moduleA* and *moduleB* are generated. Each module can pass transactions through an interface, and it has to be defined for communication between two modules. Once the interface class is defined with transaction function that can be used in it, a port of type *interface* is declared in one of the module and the interface member functions are implemented in the other module. The SystemC code in Figure 4 is an example of how to implement Transaction Level Model using interface.

First, two classes of *sc_module*, *moduleA* and *moduleB* are created. Each module has its own process therefore can work concurrently. Once the modules are created, class of interface, *moduleA_moduleB_if* is created with member function(s) declared in it. Interface can be directly connecting two modules or can have intermediate channel between two modules. Interface is directly connected between the two modules in the above example. *moduleA* declares a *sc_port* of type *moduleA_moduleB_if* and the member function *moduleA_moduleB_function* will be called through *moduleA_moduleB_port* to pass parameters to *moduleB*. Parameters can be any number and any types defined by a specific member function. This abstracts any communication/protocol needed for data to move from one module to another in the real implementation and it saves the declaration of many ports and simulation time, since the actual communication may take a number of clock cycles to establish connection before sending/receiving data. The virtually declared member function *moduleA_moduleB_function ()* is implemented in the body of *moduleB*.

Depending on the implementation of member function, *moduleA* can pass parameters to *moduleB* or it can read parameters from *moduleB*. Figure 5 describes the structure of above implementation. The modules and processes inside them can be synchronized with an external clock signal just like an RTL model. But in TLM, modules can be synchronized without the use of clock signal at all. When parameters are passed from *moduleA* to *moduleB* through *moduleA_moduleB_function*, as soon as the function is called, *moduleB* can be notified by SystemC built in *notify ( )* function. This will reduce latency for modules waiting for clock to be asserted, resulting in reduced simulation time.

```
/*********************************************
Interface declaration
*********************************************/
template <class T> class modlueA_moduleB_if :
virtual public sc_interface
{ public:
virtual void
moduleA_moduleB_function (T& parameterA,
T& parameterB) = 0;
};
/*********************************************
Module A
*********************************************/
template <class T> class moduleA : public sc_module
,
public moduleA_moduleB_if<T>
{ public:
sc_port <modlueA_moduleB_if <type> >
moduleA_moduleB_port;
SC_HAS_PROCESS(moduleA);
moduleA(sc_module_name name):sc_module(name)
{
SC_METHOD(entry);
}
private:
private variables declaration
};
template <class T> void moduleA<T>::entry() {
// body of main function of moduleA;
// calling interface function to pass
// parameters to moduleB
moduleA_moduleB_port->moduleA_moduleB_function(
parameterA, parameterB);
}
/*********************************************
Module B
*********************************************/
template <class T> class moduleB : public sc_module
{ public:
SC_HAS_PROCESS(moduleB);
moduleB(sc_module_name name):sc_module(name)
{
SC_METHOD(entry);
}
private:
private variables declaration
};
template <class T> void moduleB<T>::entry() {
// body of main function of moduleB;
}
template <class T> void moduleB <T> ::
moduleA_moduleB_function (T& parameterA,
T& parameterB)
{
// body of member function if interface;
}
```

**Figure 4. SystemC TLM Template**

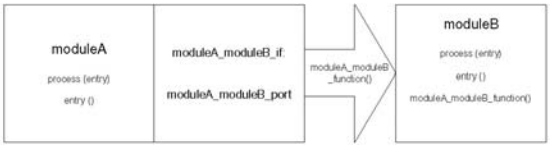


**Figure 5. SystemC TLM Structure**

### 2.2. MIPS R3000 Processor TLM Design

We designed the MIPS R3000 processor in SystemC Transaction Level Model. Even though the abstract model may deviate from the actual hierarchical structure, it is designed such that the structure remains very close to the pipeline structure shown Figure 2. Each block is defined as *sc_module* and as explained in Section 3, each module is connected through pre-defined interfaces if transactions are needed between particular modules. Figure 6 below depicts the modules used in the design.

The main functional modules are:

- ICACHE: instruction cache
- FETCH: instruction fetch module
- DECODE: instruction decode module
- EXE: execution/ALU module
- DCACHE: data cache module 4
- FPU (optional): floating point unit module

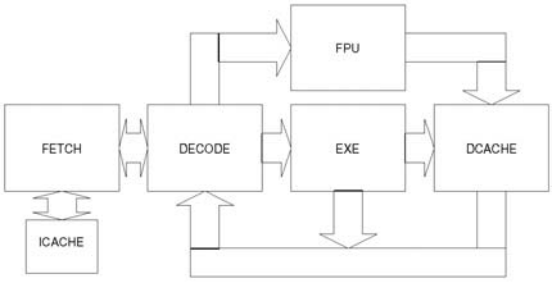


**Figure 6. MIPS R3000 Model Structure**

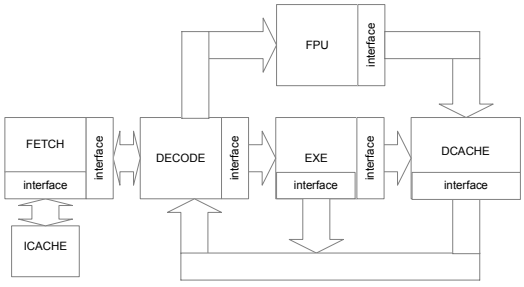


**Figure 7. MIPS R3000 TLM Structure**

Although each module has to work in synchronization, the notion of clock signal and the data movement from one module to another at the clock edge is abstracted to transactions. Hence the "WRITE BACK" stage from MIPS 5 stage pipeline is not a separate module whereas each of EXE, FPU and DCACHE writes results back to the GPR as soon as they are available. Each module passes data to necessary modules via interfaces.

The new architecture block diagram that includes interfaces is shown in Figure 7. The SystemC implementation of R3000 processor is separated in 10 files: 6 header files corresponding to each module from the block diagram, 1 header file for all the interface declarations, 1 main module that connects all the sub modules, 1 directive header file that contains some global constants and the last is the main for simulation. All the names of modules/files and their detailed function/implementation are explained below.

### **2.2.1.** *Icache Module*

The icache *sc_module* reads an instruction file called "icache" and stores in the array of length defined in "derective.h" as MAX CODE LENGTH. As it reads instructions from the file, it prints out at initialization. The Icache module does not have a process, since once it is initialized it only provides instruction cache structure in the form of an array. The fetch module will read the corresponding location of array depending on the program counter. For the fetch module to have access to the array in the icache module, an *icache_read()* member function is defined.

### **2.2.2.** *Fetch Module*

The fetch module is connected to both the icache and decode modules, hence it needs an interface for each transaction. *fetch_icache_if* is used for transaction from the icache module and fetch decode if is used for transaction to the decode module. *fetch_icache_if* provides *icache_read (current PC)* for the transaction of reading instructions from the instruction cache array, whereas *fetch_decode_if* provides a *fetch_to_decode (instruction, next PC)* function for transaction to the decode module passing instructions and the next program counter. Each of the transactions has to go through a predefined port of corresponding interface type. *icache_read ( )* uses port *fetch_icache_port* of type *fetch_icache_if* and *fetch_to_decode ( )* uses port *fetch_decod_port* of type fetch decode if. Once all the structure is well defined, the main body is implemented such that the fetch module receives the program counter as *address_if_instruction_cache* and read an instruction from instruction cache. Read instruction is then passed to the decode module along with the next program counter, which is the current PC + 1 (assuming the instruction cache is 32 bit registers). After executing one cycle of its duty, the fetch module notifies its main process with event that the process is sensitive to so that next cycle can be started as well.

### **2.2.3.** *Decode Module*

The decode module is the busiest module of all in the MIPS processor model. It decodes instructions from the fetch module, passes the decoded parameters to the EXE or FPU module, determines if the branch condition is met, updates the program counter to branch target address if the branch condition is met and sends a no-op instruction if the branch instruction is not met, etc. At initialization, the constructor reads a file called "register", which the user writes as initial register file. When the processor is started, it receives an instruction and PC from the fetch module. The unsigned instruction value is treated as 32 bit binary number and each of the necessary bit fields, such as opcode, rs, rt, rd, immediate, etc, are extracted by shifting the binary string. The extracted bit fields are assigned to corresponding variables for them to be passed to the EXE module according to the decoded result. If an instruction is found to be branch or jump, then the condition is checked and the next program counter is updated with the branch/jump target address according to the outcome of the condition. The original program counter or the updated one, according to branch, is then transferred to the fetch module. This module uses three interfaces: *decode_fetch_if*, *decode_exe_if*, and *decode_fpu_if*. Also due to interaction with number of other modules, it defines many interface member functions: *fetch_to_decode ( ), alu_result_to_GPR (), mult_div_result ( ),* and *dcache_to_GPR ( )*.

### **2.2.4.** *Exe Module*

The Exe module receives operands and function code from the decode module and the process starts from there. Upon process start, it executes one of the following ALU operations: add, sub, or, and, xor, nor, address calculation (addition), mult, or div. Unless the instruction is load/store which requires dcache access, the results of ALU operations are written to the register file in the decode module. If an instruction is load/store then the calculated address and corresponding data or destination register address are passed onto the dcache module for memory access. Exe module has three interface member functions all from the decode module. The decode module uses one of the three functions to pass correct parameters to the Exe module depending if the instruction is R-type, I-type, or Load/Store. When one of the member functions is called in the decode module, the Exe module process is notified and started its cycle.

### **2.2.5.** *Dcache Module*

The dcache module contains the dcache array which is data cache. It initializes an array *dcache[ ]* which size is determined by the user from directive.h. Then a memory file called "dcache" is read and updates the value. The process of this module is

sensitive only to load/store instructions from the Exe module so that it is activated only when needed. It only reads from the dcache array or writes to it depending whether it is a load or store instruction.

### 2.2.6. *FPU Module*

The FPU module can be started only when an instruction is found to be a floating point instruction in the decode module. FPU has its own 32 bit floating point register file and it reads the "float register" file to initialize the register file. In a real MIPS processor, the floating point unit is a separate processor that can execute IEEE754 arithmetic. Once a floating point instruction is decoded in the decode module, it passes all instruction information to the FPU module then the FPU process is started. Received floating point instructions are executed according to the specification of the instruction using its own float register. If the instruction needs memory access, then the parameters are passed to the dcache module.

### 2.2.7. *MIPS CPU Module*

This is the top module that contains all the sub-modules to make a complete processor. It declares all the units from corresponding modules and connects all the ports between the modules. This MIPS CPU module can be used as one of the processors in a multi-processor platform later on.

### 2.2.8. *MIPS CPU.cpp*

This is the *main* for the MIPS TLM to simulate the processor. It defines time intervals and the maximum time for simulation.

### 2.2.9. *Machine Code Instruction Generator*

To run the MIPS R3000 processor model, instructions in binary machine code need to be stored in the instruction cache. Normally, it is the compiler that generates the machine code from assembler program. To test various instructions in timely manner, a simple Perl script code from SystemC 2.0.1 *risc_cpu* example library has been modified to be fully compatible to MIPS instructions.

```
sub dec2bin {
my $str = unpack("B32", pack("N", shift));
return $str;
}
sub bin2dec {
return unpack("N",
pack("B32",
substr("0" x 32 . shift, -32)));
}
```

**Figure 8. Perl Routines: bin2dec, dec2bin**

We define two sub routines, "bin2dec" and "dec2bin" (see Figure 8), which would allow the program to read assembly code written in text file. "bin2dec" converts a 32 bit binary string to integer, while "dec2bin" converts an integer argument to a 32 bit binary string.

Using a while loop, the program reads one assembly code at a time converting it to corresponding machine code. From a single assembly code, it extracts necessary information such as opcode, two operands, destination, immediate, and address, etc. The opcode is taken as character string whereas all others are taken as integers (see Figure 9).

```
chop($_);
$_ =~ s/\s+/ /g;
($opcode, $arg_1, $arg_2, $arg_3) = split(/ /,$_);
$arg_1 =~ (s/(R||F||r||f)//g);
$arg_2 =~ (s/(R||F||r||f)//g);
$arg_3 =~ (s/(R||F||r||f)//g);
```

**Figure 9. Extraction of Assembly Arguments**

Once all the necessary register fields are extracted in integer, they are converted to 32 bit binary strings and again necessary binary digits are extracted (see Figure 10). For example, operands or destination register fields are 5 bit binary number. Hence from 32-bit binary string corresponding to the register number, 27 leading zeros are deleted to make them 5 bit binary string.

```
$rd = dec2bin($arg_1);
$rs = dec2bin($arg_2);
$rt = dec2bin($arg_3);
$imm = dec2bin($arg_3);
$target = dec2bin($arg_1);
$imm_branch = dec2bin($arg_2);
$rt =~ s/00000000000000000000000000000//g;
$rs =~ s/00000000000000000000000000000//g;
$rd =~ s/00000000000000000000000000000//g;
$rzero = "00000";
$fp_op = "010001";
$single = "10000";
$double = "10001";
```

**Figure 10. Variable Assignment**

Using if statements, the opcodes are decoded to construct corresponding machine code strings. From the opcode, 32 bit binary string is constructed by concatenating necessary fields in order. Then using the *bin2dec* sub-routine, it is converted to a decimal integer. Then again the decimal integer is converted to 8 digit hexadecimal output and printed (Figure 11).

```
if ($opcode =~ /\badd\b/)
{
$funct = "100000";
$bin_code = $R_type.$rs.$rt.$rd.$rzero.$funct;
$dec_code = bin2dec($bin_code);
printf ("0x");
printf ("%08x",$dec_code);
}
```

**Figure 11. Machine Code Construction**

After the while loop goes through every assembly instruction and printing each instruction in a new line, an additional instruction 0xffffffff is printed indicating that there are no more instructions to be executed while running the simulation.

"MIPS assembler.pl" can decode and convert standard MIPS assembly instructions that are compatible to MIPS R3000. It contains 39 integer instructions and 55 floating point instructions (single precision and double precision). However, there are two special attentions required to use MIPS assembler.pl effectively. First, when an instruction requires to calculate a memory address using base and immediate, standard MIPS instructions write "instruction $destination, (offset) $base". The convention in the MIPS assembler is "instruction $destination, $base, offset" in memory accessing instructions. Second, floating point instructions are required to write "f" in front of every floating instruction to indicate that this instruction is a floating point instruction. And also the use of "." is not permitted. For example, a standard MIPS floating point instruction writes "add.s", meaning single precision floating point addition.

```
lw $1, (100)$2 -> lw $1, $2, 100
div.d $f2, $f4, $f6 -> fdivd $f2, $f4, $f6
c.lt.s $f2, $f4 -> fclts $f2, $f4
```

**Figure 12. Special Assembly Instructions for Machine Code Generator**

```
addi R1, R0, 15 0x2001000f
addi R2, R0, 16 0x20020010
addi R3, R0, 1 0x20030001
addi R4, R0, 3 0x20040003
sub R5, R2, R1 0x00223020
add R6, R1, R2 0x00412822
add R7, R3, R4 0x00643820
add R8, R8, R1 0x01014020
subi R1, R1, 1 0x28210001
beq R1, R0, 1 0x10200001
j 4 0x08000004
add R10, R9, R0 0x01205020
addi R11, R0, 15 0x200b000f
sw R11, R1, 0 0xac2b0000
lw R12, R0, 10 0x8c0c000a
addi R2, R0, 16 0x20020010
```

**Figure 13. Test Assembly Code and Corresponding Machine Code**

It is required to write this particular instruction as "fadds", writing "f" in front and deleting the dot. Figures 12 and 13 give some examples of above mentioned cases.

## 3. Experimental Results

### 3.1. Description

To test if the processor is executing the instructions correctly, an assembly test code is written and the corresponding machine code is generated using MIPS assembler.pl. This simple test code first updates several registers with immediate values then sets up a branch instruction for conditional branch. If this branch condition is met, then it will skip the next instruction, if not it will execute the next instruction. The instruction in Figure 14 is found to be Jump and it forces a branch to the instruction at address #4. This forms a loop and the loop terminates when branch condition is met. It is a very simple program but useful to see every register updates and the movement of data through data path. For every cycle, each module will print out what it is processing in terms of instruction number. So it can be easily seen how the instruction moves through each cycle in a pipelined fashion.

Figure 14 is the portion of the output of running the test code. It goes through 114 instruction cycles till it breaks out of loop and does store, load, and addi instructions. Looking at the output diagonally from left low to right high indicates at one cycle, each module has a different instruction being processed. For example, when instruction #112 is fetched, the decode module is decoding instruction #111, Exe module is executing instruction #110. On the other hand, looking diagonally from left high to right low will show how a particular instruction goes through the pipeline. For example, it can be seen that instruction #111 is fetched, decoded, executed through 3 cycles.

Using the register dump option in directive.h, the output can have register file at every cycle of simulation so its content can be easily seen. Figure 15 is the output of the same test code with register dump option.

### 3.2. Discussion

The design of a simple pipelined RISC processor, R3000 in TLM has several advantages over RTL.

- Transaction of arguments by function calls through interfaces eliminates number of ports needed in RTL.
- Abstract data type such as integer, unsigned, byte, word, etc. can be used for arguments not

just bit or bit vector and this makes it easier to interact with software.

- Because of the flexibility of TLM, it can be used as IP block easily.
- The simulation time can be reduced thanks to abstracting actual implementation.
- Since the details of implementation can be abstracted, the design time can be reduced.

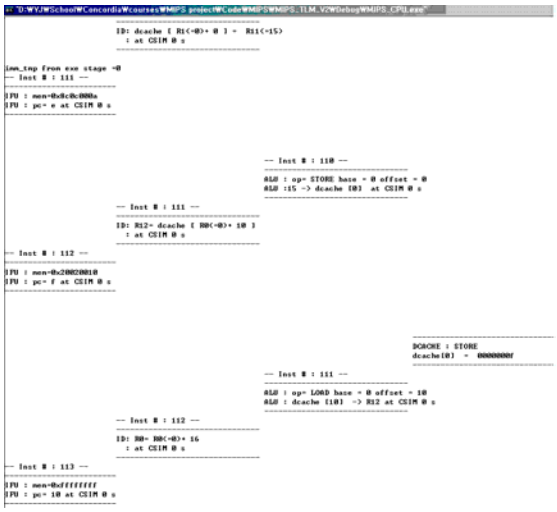

**Figure 14. MIPS TLM Simulation Output**

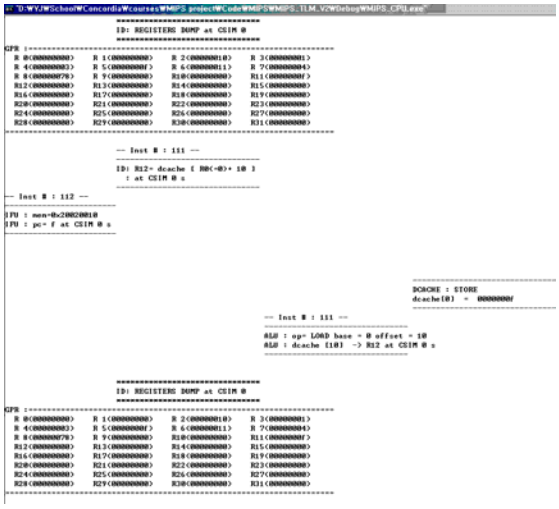

**Figure 15. MIPS TLM Simulation Output with Register Dump**

## 4. Conclusion

In this paper, we modeled a MIPS R3000 processor at the Transaction Level (TLM) in SystemC. The processor has sub-modules defined as pipeline stages and at a high level of abstraction, data moves from one sub-module to another by function calls through interfaces. The use of function calls eliminates the notion of clock cycles. Hence some modules that require multiple clock cycles to process in the real implementation can be simplified in terms of simulation time. Also TLM does not implement pins other than *sc_ports*, it can easily be used as an IP block in other designs. The concept of TLM as well as SystemC is a relatively new practice but as the need for system level simulation rapidly increases, it will be an important part in early development stages of an SoC design flow.

## 5. References


[Bre02] C. Brej. A MIPS R3000 Microprocessor on a FPGA. Technical report, The University of Manchester, Computer Science Department, 2002.

[CoW06] CoWare Inc. CoWare Model Library. Website,2006. `http://coware.com/products/modellibrary_datasheet.php`.

[Ghe05] F. Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Kluwer Academic Publishers, 2005.

[Gro02] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[HP02] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach,* Series in Computer Architecture and Design. Morgan Kaufmann, 2002.

[IEE05] IEEE Standards Association. *IEEE Std 1666TM-2005 Open SystemC Language Reference Manual*. Website 2005. `http://standards.ieee.org/getieee/1666/download/1666-005.pdf`.

[Jor03] N. A. Jorgensen. *MIPS R2000 Core model in SystemC*. Website, 2003. `http://www2.imm.dtu.dk/~jan/socmobinet/courseware/elements/mips_core.htm`.

[OSI06] Open SystemC Initiative. *The SystemC Library*, Website, 2006. `http://www.systemc.org/`.

[Sys06] SystemVerilog. IEEE 1800TM SystemVerilog Standard. Website, 2006. `http://www.systemverilog.org`.