WinARM - Simulating Advanced RISC Machine Architecture

Shuqiang Zhang

Department of Computer Science
Columbia University
New York, NY
sz184@columbia.edu

Abstract

This paper discusses the design and implementation of the WinARM, a simulator implemented in C for the Advanced RISC Machine (ARM) processor. The intended users of this tool are those individuals interested in learning computer architecture, particularly those with an interest in the Advanced RISC Machine processor family.

WinARM facilitates the learning of computer architecture by offering a hands-on approach to those who have no access to the actual hardware. The core of the simulator is implemented in C with and models a fetch-decode-execute paradigm; a Visual Basic GUI is included to give users an interactive environment to observe different stages of the simulation process.

1. Introduction:

This paper describes how to simulate an ARM processor using the C programming language. In the course of this discussion, the reader is introduced to the details of the ARM processor architecture and discovers how the hardware specifications are simulated in software using execution-driven simulation. Execution driven simulation is also know as instruction-level simulation, register-cycle simulation or cycle-by-cycle simulation [3]. Instruction level simulation consists of fetch, decode and execution phases [4].

ARM processors were first designed and manufactured by Acorn Computer Group in the mid 1980's [1]. Due to its high performance and power efficiency, ARM processors can be found on wide range of electronic devices, such as Sony Playstation, Nintendo Game Boy Advance and Compaq iPAQs. The 32-bit microprocessor was designed using RISC architecture with data processing operations occurring in registers instead of memory. The processor has 16 visible 32 bit registers and a reduced instruction set that is 32-bits wide. The details on the registers and instructions can be obtained from the ARM Architectural Reference Manual [2].

2. Related Works:

This section discusses different types of simulators available today and their different approaches in design and implementation. Most simulation tools can be classified as user level simulators: these simulate the execution of a process and emulate any system calls made on the target computer using the operating system of the host computer [5]. WinARM is an example of this type of simulator; it executes ARM instructions on a host Pentium x86 processor using a fetch-decode-execute paradigm. KScalar Simulator [Moure 6], PPS suite [7], CPU Sim3.1 [8] and OA-Mulator [9] are simulators best suited for educational purposes. They show the basic ideas of computer organization with relatively few details and complexity. They are specifically designed for students who have little or no background in computer architecture and who need a simple introduction [6]. WinARM also belongs in this category because it provides a concise and straightforward introduction to the ARM architecture. On the other extreme of the spectrum is the SPARC V9 Complete Machine Simulator, one of the many well-know complete machine simulators developed to date. These simulate the target computer from the boot stage, including all codes executed by the PROM, the OS that is loaded by the PROM, and any processes subsequently created [5]. Another approach to processor simulation can be seen in the Simx86 simulator. The Simx86 abandons the traditional simulator implementation approach of pre-decoding instructions and cross compilation. Instead, Simx86 favors an object oriented approach to improve extensibility of the simulator at the cost of increased execution time. The Simx86 provides a straightforward way to build a simulator for a processor by allowing each component

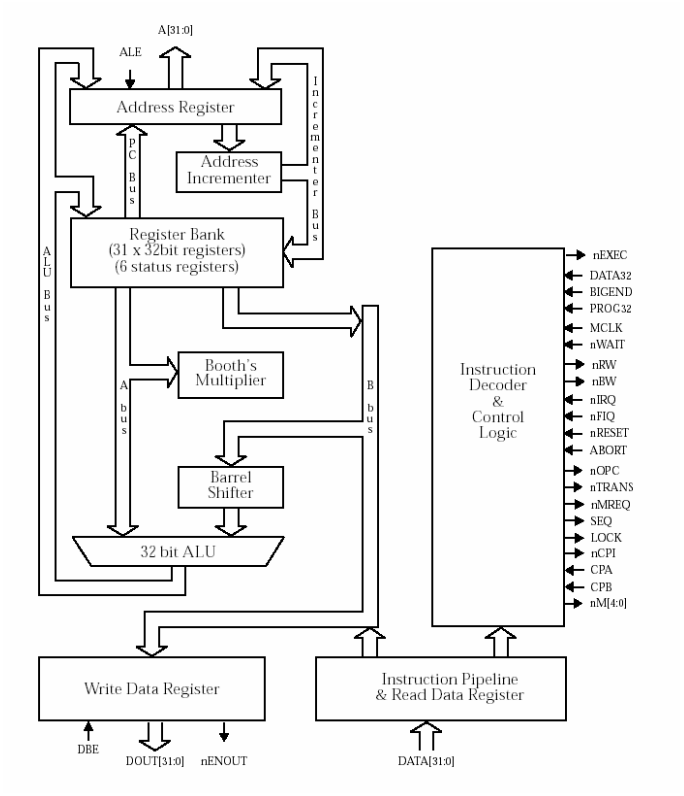


Figure 1. ARM Architecture

of the processor to be represented directly in the simulator by an object. The simulator can easily be extended by adding new classes of instructions without the daunting task of constructing a new simulator [10]. WinARM, on the other hand, retains the traditional approach for building simulators, and is composed of fetch, decoding and execution phases [4]. This approach tends to favor execution speed of the simulators [10].

3. Designing and Building WinARM

The basic approach in designing WinARM is to simulate all the necessary components of the ARM architecture in C code. This includes the register bank, instruction decoder, the 32-bit ALU and all other components. Figure 1 shows a detailed view of the major components in the ARM architecture and how they interact with each other.

The sixteen 32-bit user level registers, stack space, and memory of the ARM architecture are all modeled using arrays of unsigned long types. The fetch-decode-execute paradigm of the ARM instruction set is simulated via several C functions that get passed the 32-bit instructions. The decode function determines the type of the instruction based on its bit pattern and calls the appropriate instruction execution function. See Figure 2 for details on the instruction set of the ARM processor.

The traditional approach of building simulators which focuses on pre-decoding instructions to an intermediate representation and cross compilation [10] is also used for building the WinARM simulator,. Therefore, a cross compiler is required to generate ARM machine code to run on the host Pentium x86 machine.

Finally, a visual basic GUI is included in the simulator to provide users of WinARM an interactive environment to work in.

3.1 Cross Compilation

The idea behind simulation is to be able to ‘execute’ non-native code on a host machine. In order to obtain non-native machine code, a cross compiler is required. Many free versions of the ARM cross compiler are readily available on the

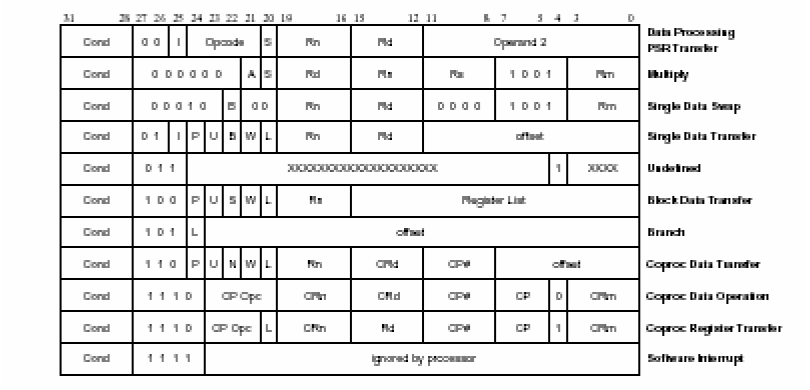

31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 5 4 3 0
Cond | 0 0 | I | Opcode | S | Rn | Rd | Operand 2 — Data Processing PSR Transfer
Cond | 0 0 0 0 0 0 | A | S | Rd | Rn | Rs | 1 0 0 1 | Rm — Multiply
Cond | 0 0 0 1 0 | B | 0 0 | Rn | Rd | 0 0 0 0 | 1 0 0 1 | Rm — Single Data Swap
Cond | 0 1 | I | P | U | B | W | L | Rn | Rd | offset — Single Data Transfer
Cond | 0 1 1 | XXXXXXXXXXXXXXXXXXXX | 1 | XXXX — Undefined
Cond | 1 0 0 | P | U | S | W | L | Rn | Register List — Block Data Transfer
Cond | 1 0 1 | L | offset — Branch
Cond | 1 1 0 | P | U | N | W | L | Rn | CRd | CP# | offset — Coproc Data Transfer
Cond | 1 1 1 0 | CP Opc | CRn | CRd | CP# | CP | 0 | CRm — Coproc Data Operation
Cond | 1 1 1 0 | CP Opc | L | CRn | Rd | CP# | CP | 1 | CRm — Coproc Register Transfer
Cond | 1 1 1 1 | Ignored by processor — Software Interrupt

Figure 2. ARM instruction Set

world wide web. The one used for this simulator was created by Jason Wilkins (fenix@ io.com).

3.2 Instruction Fetching

Once ARM machine code has been obtained using a cross compiler, the instruction fetching process begins. The object file generated by the assembler is read 32 bits at a time since all ARM instructions are 32-bits wide. The header and footer sections of the object file, which contains ARM system information, are discarded because they are not used by the simulator. All the fetched instructions are then stored in the Program Memory space, which is modeled by a fixed size array of unsigned long type. After all the instructions are fetched and stored, the Decoding stage begins.

3.3 Instruction Decoding

All of the ARM instructions are 32-bits wide, with a predefined and distinct bit pattern. WinARM uses these bit patterns to determine the type of the instruction being decoded, and calls appropriate execution functions to execute each specific instruction. See figure 2 for the bit patterns of each type of ARM instruction. The decoding process isolates the bit pattern of each incoming instruction and compares it to the set of predefined bit pattern, if there is a match, the instruction is sent to the target Execution function to be executed.

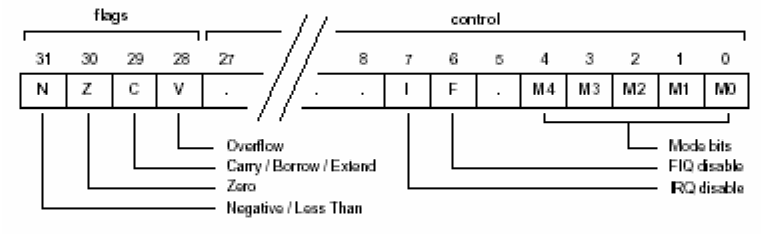


Figure 3. CPSR Register

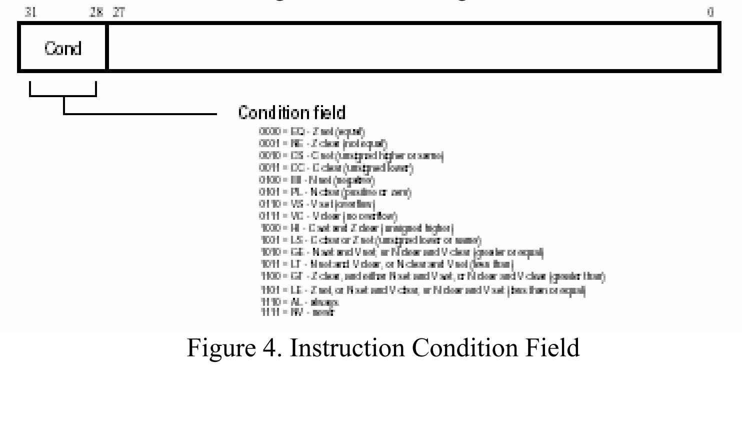
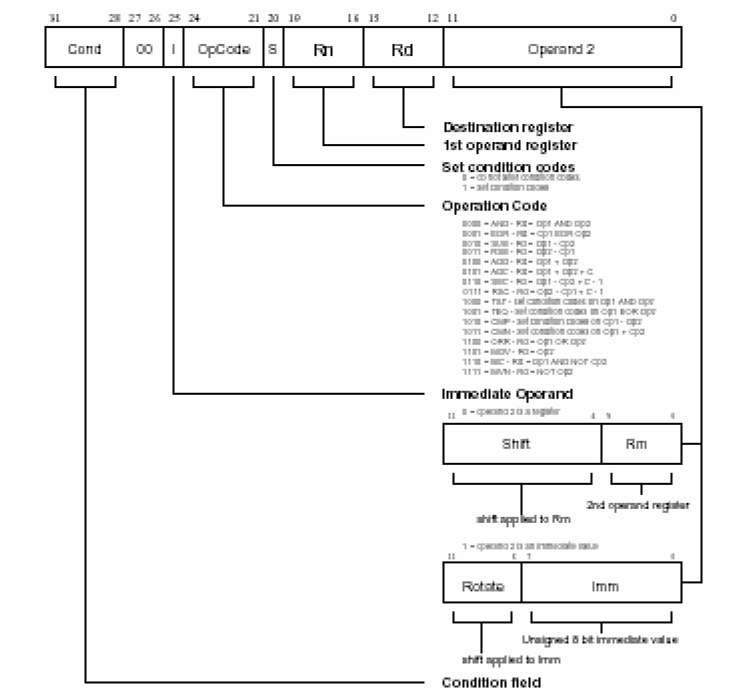

31 28 27 0
Cond
Condition field
0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = CS - C set (unsigned higher or same)
0011 = CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or Z set (unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (greater or equal)
1011 = LT - N set and V clear, or N clear and V set (less than)
1100 = GT - Z clear, and either N set and V set, or N clear and V clear (greater than)
1101 = LE - Z set, or N set and V clear, or N clear and V set (less than or equal)
1110 = AL - always
1111 = NV - never

Figure 4. Instruction Condition Field



31
28
27
26
25
24
21
20
19
16
15
12
11
0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd = Op1 AND Op2
0001 = EOR - Rd = Op1 EOR Op2
0010 = SUB - Rd = Op1 - Op2
0011 = RSB - Rd = Op2 - Op1
0100 = ADD - Rd = Op1 + Op2
0101 = ADC - Rd = Op1 + Op2 + C
0110 = SBC - Rd = Op1 - Op2 + C - 1
0111 = RSC - Rd = Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd = Op1 OR Op2
1101 = MOV - Rd = Op2
1110 = BIC - Rd = Op1 AND NOT Op2
1111 = MVN - Rd = NOT Op2
Immediate Operand
0 = operand 2 is a register
Shift
Rm
2nd operand register
shift applied to Rm
1 = operand 2 is an immediate value
Rotate
Imm
Unsigned 8 bit immediate value
shift applied to Imm
Condition field

Figure 5. Data Processing Instruction

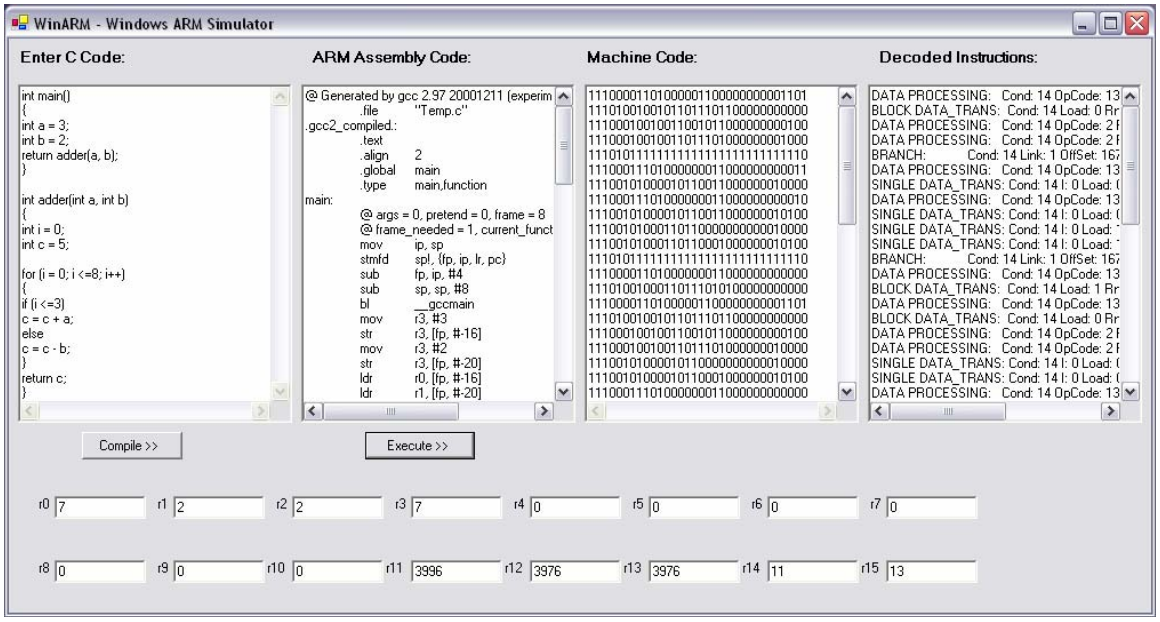
3.4 Instruction Execution

The main working unit of the simulator consists of the execution functions for each different type of ARM instruction. These functions do the actual arithmetic computations, move data between simulated registers and memory, update the CPSR register, and keep track of the program counter and stack pointer.

In addition to the predefined bit pattern of each instruction type, there are many variable bits in each instruction. Depending on the state of these variable bits, each instruction could be executed in many different ways. For example, a Data Processing Instruction has the 'I' bit which determines if the second operand is a register or an immediate value. If the second operand is a register, the instruction also contains a shift field, which can specify different types of shifts. The shift amount can either be a register or an immediate value. The OpCode field may specify sixteen different kinds of operations the Data Processing instruction may perform. See Figure 5 for details on Data Processing Instructions.

Another important aspect of WinARM is the simulation of CPSR (Program Status Register) and the condition flag of each ARM instruction. These two, in combination, determine whether the current decoded instruction should be executed or ignored, which is of utmost importance when running branch instructions. The CPSR register has a four bit Condition Code that consists of N, Z, C and V flags (see Figure 3). These flags are updated and latched each time any instruction is executed with the S bit field set to 1 [2]. Every ARM instruction contains a Condition field of four bits, but each individual bit of the instruction condition field has no direct correlation with each of the condition flags in the CPSR register. Instead, there are sixteen possible instruction conditions; each condition requires the system to check the state of one or more CPSR condition flags. See Figure 4 for details on each of the condition field requirements. If the requirements were met, the current instruction gets executed, otherwise, it is ignored and the PC (program counter) gets incremented by one and points to the next instruction to be executed.

The PC (Program Counter), which is register r15, points to the next ARM instruction to be executed. The value of the PC can be updated or moved just like any other ARM register, for instance, a branch instruction would update the PC with its offset so the execution path can jump to somewhere else in the Program Space. In conjunction with register r14, the link register, the PC can simulate a return from a called function: the old PC value is copied into r14 before branching occurs, upon returning from the called function, the value in r14 is copied back into the PC, so the



Enter C Code:

```
int main()
{
int a = 3;
int b = 2;
return adder(a, b);
}

int adder(int a, int b)
{
int i = 0;
int c = 5;

for (i = 0; i <=8; i++)
{
if (i <=3)
c = c + a;
else
c = c - b;
}
return c;
}
```

ARM Assembly Code:

```
@ Generated by gcc 2.97 20001211 (experim
        .file       "Temp.c"
.gcc2_compiled.:
        .text
        .align      2
        .global     main
        .type       main,function
main:
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, current_funct
        mov         ip, sp
        stmfd       sp!, {fp, ip, lr, pc}
        sub         fp, ip, #4
        sub         sp, sp, #8
        bl          __gccmain
        mov         r3, #3
        str         r3, [fp, #-16]
        mov         r3, #2
        str         r3, [fp, #-20]
        ldr         r0, [fp, #-16]
        ldr         r1, [fp, #-20]
```

Machine Code:

```
11100001101000001100000000001101
11101001001011011101100000000000
11100010010011001011000000000100
11100010010011011101000000001000
11101011111111111111111111111110
11100011101000000011000000000011
11100101000010110011000000010000
11100011101000000011000000000010
11100101000010110011000000010100
11100101000110110000000000010000
11100101000110110001000000010100
11101011111111111111111111111110
11100001101000000011000000000000
11101001000110111010100000000000
11100001101000001100000000001101
11101001001011011101100000000000
11100010010011001011000000000100
11100010010011011101000000010000
11100101000010110000000000010000
11100101000010110001000000010100
11100011101000000011000000000000
```

Decoded Instructions:

```
DATA PROCESSING:   Cond: 14 OpCode: 13
BLOCK DATA_TRANS:  Cond: 14 Load: 0 R
DATA PROCESSING:   Cond: 14 OpCode: 2
DATA PROCESSING:   Cond: 14 OpCode: 2
BRANCH:            Cond: 14 Link: 1 OffSet: 16
DATA PROCESSING:   Cond: 14 OpCode: 13
SINGLE DATA_TRANS: Cond: 14 I: 0 Load:
DATA PROCESSING:   Cond: 14 OpCode: 13
SINGLE DATA_TRANS: Cond: 14 I: 0 Load:
SINGLE DATA_TRANS: Cond: 14 I: 0 Load:
SINGLE DATA_TRANS: Cond: 14 I: 0 Load:
BRANCH:            Cond: 14 Link: 1 OffSet: 16
DATA PROCESSING:   Cond: 14 OpCode: 13
BLOCK DATA_TRANS:  Cond: 14 Load: 1 R
DATA PROCESSING:   Cond: 14 OpCode: 13
BLOCK DATA_TRANS:  Cond: 14 Load: 0 R
DATA PROCESSING:   Cond: 14 OpCode: 2
DATA PROCESSING:   Cond: 14 OpCode: 2
SINGLE DATA_TRANS: Cond: 14 I: 0 Load:
SINGLE DATA_TRANS: Cond: 14 I: 0 Load:
DATA PROCESSING:   Cond: 14 OpCode: 13
```

r0	7	r1	2	r2	2	r3	7	r4	0	r5	0	r6	0	r7	0
r8	0	r9	0	r10	0	r11	3996	r12	3976	r13	3976	r14	11	r15	13

Figure 6. WinARM GUI

PC would point to the instruction before the branch occurred. The WinARM stack, similar to the Program Space and memory space, is simulated with an array of type unsigned long. There are no pop and push methods implemented explicitly for the stack. Instead, block data transfer instructions stm and ldm are used to simulate stack pop and push, which updates register r13, the stack pointer.

3.5 Simulator GUI

A WinARM GUI was built using Microsoft Visual Basic, which gives WinARM users a more interactive environment to learn about simulators. Users can type in C code in the Enter C Code text area.Upon clicking the execute button, the simulator would compile the C code and display the resulting ARM assembly code in the ARM Assembly Code text area. The machine code generated and the decoded version of the machine code would be displayed in the Machine Code text area and the Decoded Instructions text area respectively. The sixteen user registers would also be updated with their final state. See Figure 6.

4. Future Work

The current version of WinARM has only simulated the mostly commonly used instruction types, such as Data Processing, Multiply, Single and Block Data Transfer and Branch. More work will be done to simulate the remaining instruction types. Instruction pipelining also need to be implemented along with clock cycles to make the simulator more complete. Currently, WinARM can only handle integer arithmetic, future versions would also to incorporate floating point arithmetic.

5. Conclusion

The WinARM simulator was developed to target audiences interested in learning the inner workings of a processor simulator and to gain some insight into the ARM architecture. WinARM uses a traditional fetch-decode-execute paradigm to execute non-native machine code on a host processor in favor of execution speed [4]. The fetch-decode-execute phases of the simulator were built using C for efficiency reasons, and a GUI built in Microsoft Visual Basic was supplied so users can see the different steps taken in the simulation process and the final state of each register.

References:

[1]funkysh, “Into my ARMs” www.phrack.org/show.php?p=58&a=10

[2] ARM Architectural Reference Manual – Issue D, 2000 Advanced RISC Machines LTD

[3] D. A. Sykes, B.A. Malloy, The Design of an Efficient Simulator for the Pentium Pro Processor, In *Proceedings of the 1996 Winter*

Simulation Conference, pp. 840-847, 1996.

[4] I. Barbieri, M. Bariani, M. Raggio, A VLIW Architecture Simulator Innovative Approach for HW-SW Co-Design, *2000 IEEE international Conference on Multimedia and Expo*, Vol. 3. pp1375-1378, 2000.

[5] B. Clarke, A. Czezowski, P. Strazdins, Implementation Aspects of a SPARC V9 Complete Machine Simulator, In *Conferences in Research in Information Technology, Vol. 4.* Australian Computer Society, pp. 23-32, 2001.

[6] J. C. Moure, D. I. Rexachs, E. Luque, The KScalar Simulator, *ACM Journal of Educational Resources in Computing*, Vol. 2, No. 1, pp. 73-116 March, 2002.

[7] B. K. Gunther, Facilitating Learning in Advanced Computer Architecture through Appropriate Simulation, *ACSC 23rd Australasian Computer Science Conference, 2000.* pp. 104-112, 1999.

[8] D. Skrien, CPU Sim3.1: A Tool for Simulating Computer Architectures for Computer Organization Classes, *ACM Journal of Educational Resources in Computing*, Vol. 1, No. 4 , pp. 46-59, December, 2001.

[9] F. Menczer, A. M. Segre, OAMulator: A Teaching Resource to Introduce Computer Architecture Concepts, *Journal of Educational Resources in Computing,* Vol. 1, No. 4, pp18-30, December, 2001.

[10] A. R. Shealy, B. A. Malloy, Simx86: An Extensible Simulator for the Intel 80x86 Processor Family, In *Proceedings of the 30th Annual Simulation Symposium*, pp. 157-166, 1997.