SimpleScalar Simulation of the PowerPC Instruction Set Architecture

Karthikeyan Sankaralingam Ramadass Nagarajan Stephen W. Keckler Doug Burger
Computer Architecture and Technology Laboratory
Department of Computer Sciences
Tech Report TR2000-04
The University of Texas at Austin
cart@cs.utexas.edu — www.cs.utexas.edu/users/cart

ABSTRACT

In this report, we describe a modification to the SimpleScalar tool set to support the PowerPC ISA. Our work is based on Version 3.0 of the publicly available SimpleScalar tool set. We briefly describe features of the PowerPC ISA relevant to the simulator and provide operating system specific implementation details. We made modifications to the suite of five simulators that model the micro-architecture at different levels of detail. The timing simulator *sim-outorder* simulates PowerPC binaries on the Register Update Unit (RUU) micro-architecture. The five simulators were tested by simulating the SPEC CPU95 benchmarks to completion. The tool set simulates binaries compiled for 32–bit IBM AIX running on PowerPC.

1 Overview

The SimpleScalar tool set (release 3.0) can simulate the Alpha ISA and the PISA ISA [1]. In this work, we extend this tool set to support the PowerPC ISA which is defined in *The PowerPC Architecture Specification* [2]. Currently, only the 32–bit implementation of the PowerPC architecture is supported. Future versions may support the 64–bit architecture. Binaries compiled for 32–bit IBM AIX can be run on one of the several provided simulators on an IBM AIX machine. The target operating system we support in this release is IBM AIX. However, we also provide a minimally tested cross–platform simulator running on Sun Solaris, simulating PowerPC binaries compiled on an IBM AIX machine.

The remainder of this report is organized as follows. Section 2 explains the features of the PowerPC ISA and its differences from the Alpha and PISA ISAs. In Section 3, we explain the machine/Operating System (OS) specific details that should be addressed in a simulator. Section 4 provides an overview of the different simulators in the tool set and briefly describes the modifications we made to each of the simulators. In Section 5, we provide the details on instruction emulation. The functioning and simulation of the loader is explained in Section 6 and in Section 7, we provide the details of executing system calls. Miscellaneous operating system issues handled by the simulator are dealt with in Section 8. The working of the timing simulator (`sim-outorder`) is explained in Section 9. Instructions for building and using the simulator are provided in Section 10.

In the remainder of this document, target will refer to the ISA being simulated (PowerPC) and host will refer to the machine on which the simulator is executed.

2 ISA Description

The PowerPC ISA has some features that make it different from the Alpha and PISA ISAs. For example, the Alpha ISA has 215 instructions with 4 instruction formats and the PISA ISA has 135 instructions with 4 instruction formats. The PowerPC ISA on the other hand has 224 instructions with 15 instruction formats. Not all of these instructions are implemented in the simulator. In this section, we describe features of the ISA that are implemented in the simulator.

2.1 Registers

The PowerPC architecture defines 32 General Purpose Registers (GPR) and 32 Floating Point Registers (FPR). The GPRs are 32 bits wide and the FPRs are 64 bits wide. A 32-bit Condition Register (CR) is logically divided into 8 subfields CR0 to CR7 each subfield being 4 bits long. This register holds condition codes. The 32-bit Link Register (LR) is used for transferring program flow and a 32-bit Count Register (CTR) is used for loops. Certain instructions implicitly compare the CTR to zero to detect loop termination condition. The CTR can also be used for transferring program flow. The status of the floating point unit is saved in a 32-bit wide Floating Point Status Control Register (FPSCR). A 32-bit wide Fixed Point Exception (XER) contains the status and exceptions generated while executing fixed point instructions. This 27 fields of the FPSCR and the 5 fields of the XER are described in pages 137–141 and pages 48–49 of *The PowerPC Architecture Specification* [2]. Figure 1 outlines all of the user registers in the SimpleScalar implementation of the PowerPC ISA. The machine specific registers defined in the PowerPC ISA are not shown and are not handled in the simulator.

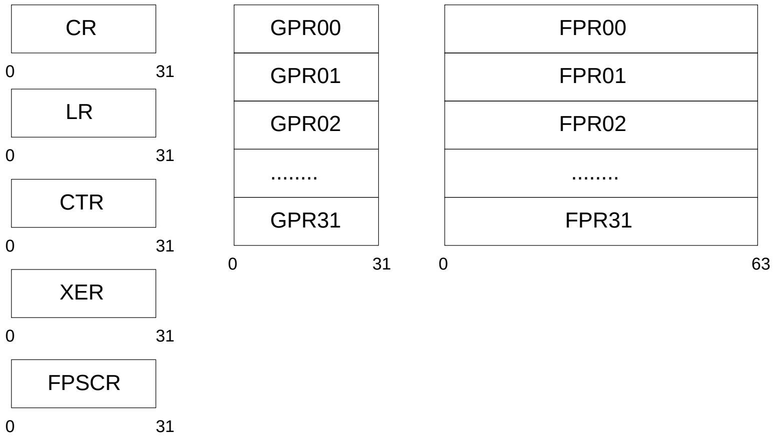

Figure 1: PowerPC user register set

2.2 Instructions

PowerPC instructions are four bytes long and always word aligned. Thus for a given instruction address, the two lower order bits are ignored. Bits 0-5 always provide the opcode. Many instructions also have an extended opcode. Some instructions have reserved fields which must be set to zero. Illegal instructions that are not defined invoke the system illegal instruction handler. In the simulator, a panic call halting the simulator is invoked during instruction decode. Not all of the instructions defined in the ISA are implemented by the simulator. Only the user level instructions allowed on a 32-bit target are implemented.

2.3 Storage Model

PowerPC provides for bytes, halfwords and words as its primitive data types. Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte. Storage operands may be bytes, halfwords, words or double words, or for the *Load/Store Multiple* and *Move Assist* instructions, a sequence of bytes or words. The address of a storage operand is the address of its first byte. Misaligned addresses are allowed for data accesses. The PowerPC architecture supports both *little endian* — MSB at bit 32/64 and *big-endian* — MSB at bit 0, byte ordering. Only the *big-endian* byte ordering, is supported in the simulator.

3 AIX Operating System Overview

Two main operating system issues involved in porting SimpleScalar to a new architecture are:

1. Loader
2. System calls

Loader: Since the simulator takes a binary file as input, we need to know the binary file format and the tasks performed by the OS loader before it starts executing the program. The AIX loader loads the program into memory and resolves relocatable references to memory addresses. System calls which are embedded in the binary file as relocatable references are also resolved by the loader.

There are also other minor issues like passing environment variables and program arguments which need to be handled by the loader. These implementation details for our simulator are explained in Section 6.

System calls: Since we implement only user level instructions, system calls are implemented using the host machine as a proxy to execute the system call. When a system call is made by the simulated program, the simulator obtains the arguments passed to the call and makes the call at the source level by calling the corresponding user level function call. The details of detecting and executing the system calls are explained in Section 7.

4 Implementation Overview

The SimpleScalar tool set is modular and can be modified to provide support for new ISAs and micro-architecture features. The different "structures" simulated like the cache, memory, registers, instruction emulation and micro-architecture are placed in separate files. The five simulators — *sim-fast, sim-cache, sim-profile, sim-bpred and sim-outorder* in the tool set share these common files. The objective of the project was to create a functional simulator and a timing simulator using the register update unit(RUU) micro–architecture and executing the PowerPC instruction set.

Getting the functional simulator to work involves changing the register definitions, register file sizes, instruction emulation, the loader and the system call interface. The cache simulator and branch prediction simulator are based on the functional simulator and worked right away when we completed the functional simulator. We needed to make relatively minor changes for getting the full timing simulator (sim-outorder) to work because of several idiosyncrasies in the PowerPC ISA which were incompatible with the RUU micro-architecture capability. The problems we faced and the solutions are explained in Section 9.

5 Instruction Emulation

All of the five simulators share the instruction definitions from the same file called `machine.def`. This file contains the code for instruction emulation (in C or inline assembly) and the register and functional dependencies of the instruction. The correctness of the dependencies in an instruction does not affect its definition. Even if some dependencies are wrong, the functional simulator, cache simulator and branch prediction simulator will work. However, for the correct functioning of the timing simulator, these dependencies must be defined correctly.

The mechanism of defining an instruction's dependencies and its implementation are explained in [1] and we will not dwell on those details. Instruction decoding and the mechanism for supporting extended opcodes for a single primary opcode are also explained in the technical report. We made minor modifications to the tool set to support instruction decode for the PowerPC ISA. These changes are documented in the provided source code.

The instructions defined are listed appendix A. As previously mentioned, a few instructions are defined only in the 64-bit mode, and the simulator halts with an illegal instruction error when any of these instructions are encountered.

The PowerPC architecture implements the IEEE Standard 751-1985 floating point arithmetic specification. The floating point processor raises a number of exceptions and supports four rounding modes. To simulate the floating point processor we adopted a two pronged approach. On an IBM AIX host, all the floating point instructions are executed natively using inline assembly code. On a non-native host, the instructions are emulated at the source level. This emulation is incomplete and does not emulate all of the behavior of the processor being simulated. All the information that is stored in the FPSCR which controls rounding modes and exception status is ignored. The host type is detected when the simulator is compiled and the appropriate implementation is selected.

5.1 Native Floating Point Implementation

Most of the *computational* floating point instructions modify a large number of fields/flags in the FPSCR. Computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. On a native host, a *true emulation* of the floating point processor can be achieved by executing the instruction natively. By *true emulation*, what we mean is the change of state in the simulated machine after the execution of the instruction will be same as the change of state—registers and memory, of a real machine.

In the simulator, *true emulation* is achieved by executing the instruction using inline assembly. The state variables affected by a computational floating point instruction are:

1. One of the Floating Point registers (FPR)

2. Floating point status and control register (FPSCR)

3. Condition Register (CR)

The register file of the simulated machine is saved as a variable in the simulator. The FPSCR and CR are fields in this register file. The following steps are done to execute a computational floating point instruction:

1. Copy the simulated machine's FPSCR (from register file variable) into the host machine's FPSCR.

2. Execute the floating point instruction in the host machine—machine on which the simulator is running. This will affect the state of the FPSCR in the real machine. The output generated by execution is copied to the simulator's register file.

3. Copy the value of the FPSCR from the host machine into the FPSCR field in the target machine's register file data structure.

Figure 2 shows a typical Floating Point instruction emulation. It shows the code fragment for the `FADD` instruction.

Lines 8 to 11 execute the instruction natively. The original FPSCR value is passed using `fpscrin` and the updated value is written to `fpscrout`. This value is copied to the register file variable maintained by the simulator using the macro on line 16. The `mtfsf` and `mffs` instructions copy values into the FPSCR and from the FPSCR respectively. Lines 14 and 15 copy the output register value generated by the execution of this instruction to the register file in simulator.

5.2 Non-native Floating-Point Implementation

Figure 3 contains the code listing for the non-native implementation of the FADD instruction. As can be seen from the code, the modifications to FPSCR are ignored. On a non-native host, the contents of the FPSCR are ignored and the rounding mode of the compiler which is used to compile

```
#define FADD_IMPL
{
    qword_t _a, _b;
    qword_t *dest;
    double double_a, double_b, double_dest;
    _a = PPC_FPR_DW(RA); /* copy source registers into temporary */
    _b = PPC_FPR_DW(RB); /* register type variables */
    memcpy(&double_a, &_a, sizeof(double) ); /* copy temporary reg. type */
    memcpy(&double_b, &_b, sizeof(double) ); /* variables into doubles */
    asm ("mtfsf 0xFF,%2; fadd %0,%3,%4; mffs %1"
      : "=f" (double_dest), "=f" (fpscrout) /* copy in FPSCR */
      : "f" (fpscrin), "f" (double_a),      /* add */
         "f" (double_b) );                  /* copy out resulting FPSCR */
    fp1 = (int *) (&fpscrout);              /* and output value */
    memcpy(&_fp, (fp1+1), 4);
    dest = (qword_t *) (&double_dest);
    PPC_SET_FPR_DW(FD, *dest);          /* write output value to reg. */
    PPC_SET_FPSCR( *(int *) (fp1+1));   /* write resulting FPSCR */
}
```

Figure 2: FADD implementation on IBM AIX host

the simulator is always active. In our simulation of the SPEC CPU95 benchmarks, we noted that ignoring changes to FPSCR did not affect execution.

A few of the computational floating point instructions modify the Condition Register (CR). According to the result of the instruction - $<, >, = 0$ or overflow, CR1 (second 4 bits of CR) is set to 0, 1, 2 or 3. On the simulator this is done by comparing the result generated after execution. This step does not vary between native IBM AIX and non-native hosts.

5.3 Misaligned Accesses

The PowerPC architecture allows misaligned addresses to access data. To support this in the simulator, the alignment of every memory read and write is checked and for every misaligned read/write, the two consecutive words are read and the correct bytes are stitched together and returned.

Every misaligned memory read-word results in two simulated memory reads and consequent simulated page-faults and cache-misses if any. Every misaligned memory write-word results in two simulated memory reads to read the two words aligned on word boundaries that are affected by the write, two memory writes to write back both the modified words and the consequent simulated page-faults and cache-misses of all these four accesses.

A misaligned memory read/write of a half-word (16 bits) spanning two words, results in two reads for a memory read and two reads and two writes for a memory write. A misaligned memory read/write of a half-word that does not span a word, does not incur any extra reads or writes.

```
#define FADD_IMPL                                                  \
{                                                                  \
  qword_t _a, _b;                                                  \
  qword_t *dest;                                                   \
  double double_a, double_b, double_dest;                          \
  _a = PPC_FPR_DW(RA);                                             \
  _b = PPC_FPR_DW(RB);                                             \
       memcpy(&double_a, &_a, sizeof(double) );                    \
       memcpy(&double_b, &_b, sizeof(double) );                    \
       double_dest = double_a + double_b;                          \
  dest = (qword_t *) (&double_dest);                               \
  PPC_SET_FPR_DW(FD, *dest);                                       \
}
```

Figure 3: FADD implementation on non-AIX host

5.4 Little Endian Hosts

Support for little endian hosts is based on the cross endian memory access macros provided in SimpleScalar 3.0. Little endian hosts are supported by reordering the bytes before they are written to or read from simulated memory. During program execution memory is accessed in four ways.

1. Loading the program: The OS loader copies the program code segment to memory when the program is loaded. In the simulator the code segment is read from the program binary file and written to simulated memory.

2. Data segment, program arguments and environment variables: These values are also written by the loader to memory.

3. System calls: Some system calls read or write data to buffers. The *fread* system call for example reads a block from a file and writes it to a buffer in memory

4. Load/Store instructions: Instructions that read or write register values to memory.

All four types of memory accesses pass through the same memory access macros in the simulator. To provide cross endian support the bytes written to memory are reordered before writing and after reading from simulated memory on little endian hosts. Reordering the bytes in this manner, guarantees that the contents of simulated memory is big endian irrespective of endianness of the host. Reordering the contents using the macros provides the correct values on little endian host when the values are used in computation in the instruction emulation sections of the simulator. The memory access macros are defined in `memory.h`. The functional simulator has been tested only for a few of the integer SPEC benchmarks on X86 Linux.

6 Loader

As previously mentioned, there are two main functions that are performed by the loader:

1. Loading the program into memory, setting up its environment variables and arguments

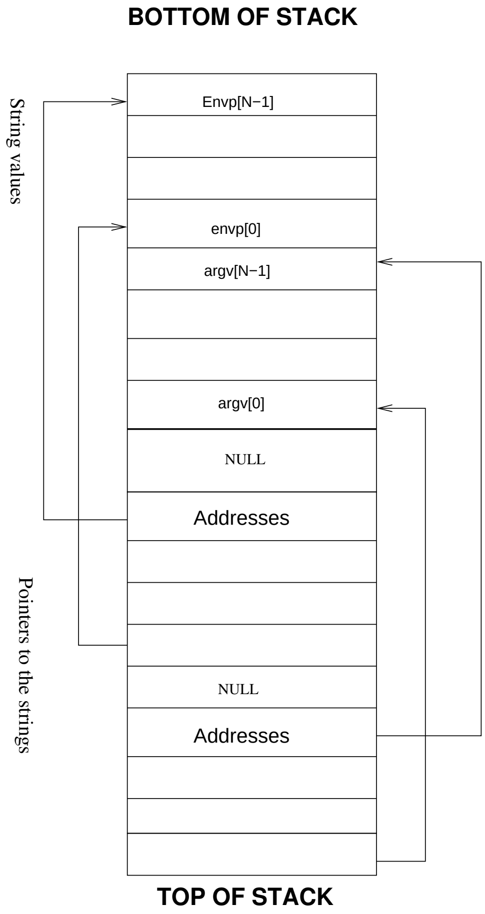

Figure 4: Stack layout with environment variables and program arguments

2. Relocatable references in the loader segment of the program are assigned to locations in memory. Relocatable references are addresses to objects whose memory address is determined and allocated at run time by the loader.

The IBM AIX system calls are present as relocatable references in the loader segment. The loader determines the addresses of these system calls and writes those values in memory when the program is loaded.

6.1 Environment Variables and Program Arguments

On a real machine, environment variables are passed as an array of string pointers to the `main` function call (for a C program). The loader decides where to allocate space for the environment variables and creates the array of pointers and passes the first element of the array to `main`. The end of the array is denoted by a NULL value. Every environment variable is a single string with an "=" separating the variable name and its value.

In the simulator, the environment variables and the array of pointers to the variables are saved on the stack. The environment passed to the program being simulated is the environment in which the simulator is running. First, all the environment variables are pushed on the stack one after another. These variables are null-terminated strings (character arrays). Then the program arguments are pushed on the stack one after the other in reverse order, `argv[0]` (the full path of the program being simulated) pushed as the last argument. These arguments are also null-terminated string values.

A zero (NULL) is then pushed on the stack. Then the address of each environment variable is pushed on the stack. The zero pushed earlier is used to determine end of environment variables when the values are popped by the program from the top of the stack. The top of stack at this stage is saved as the pointer to the environment variables. Another zero (NULL) is pushed on the stack to indicate end of array of program argument pointers. Then the address of each argument is pushed on the stack. The top of stack at this stage is saved as the pointer to the program arguments. Figure 4 shows the layout of the stack when the loader has completed storing environment variables and program arguments.

Figure 5 shows the actual contents of the stack for a simulated program. Note that the list of environment variables has been truncated.

```
0x7fff ffff HOME=/home/karu\0         # envp[4]
0x7fff ffef TERM=xterm-color\0        # envp[3]
0x7fff ffde PWD=/home/karu/ss3ppc\0   # envp[2]
0x7fff ffc8 SHELL=/bin/bash\0         # envp[1]
0x7fff ffb8 PS1=\h:\w>\0              # envp[0]
0x7fff ffad ijpeg1\0                  # argv[1]
0x7fff ffa6 ./sim-outorder\0          # argv[0]
0x7fff ff97 \0\0\0                    # 3 zeros for padding

# remaining values on stack
# are all addresses of values above
0x7fff ff94 0x0000 0000               # NULL (4 bytes of zero)
                                      # Denotes end of array to follow
0x7fff ff90 0x7fff ffff
0x7fff ff8c 0x7fff ffef
0x7fff ff88 0x7fff ffde
0x7fff ff84 0x7fff ffc8
0x7fff ff80 0x7fff ffb8
0x7fff ff7c 0x0000 0000               # Denotes end of array to follow
0x7fff ff78 0x7fff ffad
0x7fff ff74 0x7fff ffa6

R4 = 0x7fff ff74
R5 = 0x7fff ff80
Bottom of stack = 0x7fff ffff
Top of stack    = 0x7fff ff74
```

Figure 5: Stack contents at program startup

As specified by the AIX calling conventions, registers 3 onwards are used to pass arguments. The loader sets register 3 to the number of program arguments (`argc`), register 4 to the address of

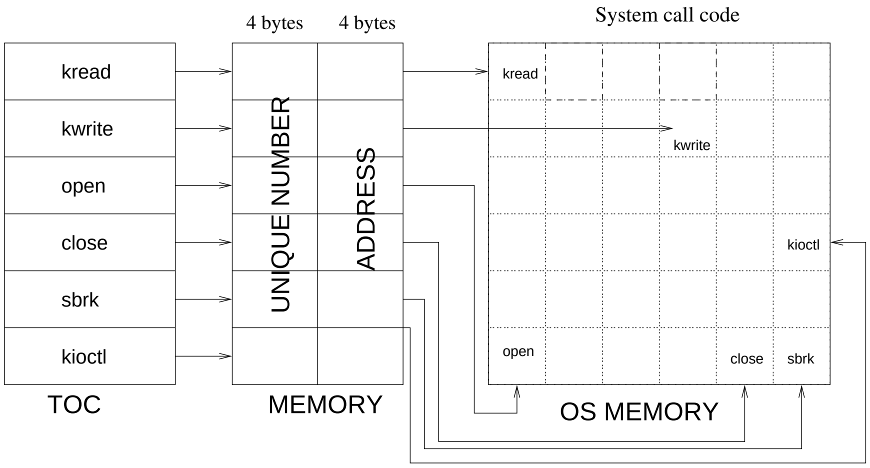

Figure 6: System Call Mechanism. The first memory block is allocated by the loader and each entry is 8 bytes long. The second memory block is the entire system memory and the OS system call code resides there.

```
0x10007f40 <sbrk>:      lwz     r12,188(r2)   # Read an address from TOC+188
0x10007f44 <sbrk+4>:    stw     r2,20(r1)     # Save R2 on stack
0x10007f48 <sbrk+8>:    lwz     r0,0(r12)     # Load first word from address
                                              # pointed to by R12 into R0
0x10007f4c <sbrk+12>:   lwz     r2,4(r12)     # Load second word into R2
0x10007f50 <sbrk+16>:   mtctr   r0            # Copy R0 into CTR
0x10007f54 <sbrk+20>:   bctr                  # Jump to CTR
                                              # The actual system call code
                                              # is at address CTR
```

Figure 7: Instructions for SBRK system call

program arguments (`argv[]`), and register 5 to the address of environment variables (`envp[]`).

6.2 System calls

System calls are listed as relocatable references in the loader segment of the binary file. Every such entry in the loader segment has a name, address and various other fields. The address field points to an entry in the Table of Contents (TOC), which contains a unique entry for every system call. On a real machine 8 bytes (2 words) of memory are allocated by the loader and the start address of these 8 bytes is written into the TOC entry for that system call. The loader also fills in the values of the two words that it has allocated. The first word is a unique number that identifies the system call and the second word is the address where the actual system call code resides in memory. Figure 6 explains the system call mechanism.

For each system call a sequence of user level instructions are executed. Figure 7 contains all of the user level code executed for the `sbrk` system call. Every system call contains the same six user level instructions except for the offset in the first instruction. Adding this offset to the start of the TOC gives the address of the system call in the TOC. This is the address that is saved in the loader segment of the binary file.

In the simulator, a predecode is done before the simulation starts. In this predecode step, the entire instruction stream is scanned word by word and when this sequence of 6 instructions is detected, the last of these instructions - `bctr` is replaced with a new instruction called `sc`. This `sc` instruction is the System Call instruction that the PowerPC defines. A user level program is guaranteed to not have this in its instruction sequence. So it is safe to use this opcode to indicate a system call.

The loader in the simulator does things a bit differently compared to a real loader. Every element in the loader segment is examined. Whenever a relocatable entry is detected the name of the field is compared with the names of system calls emulated by the simulator. If this system call is implemented in the simulator, eight bytes are allocated on the stack. The first word is set to a unique number identifying the system call, the second word is ignored. The unique numbers for the system calls are chosen arbitrarily and are listed in `syscalls.h`. The address of the first word is written to the TOC address present as a field in the loader segment entry. If a system call is encountered that is not supported, the unique number stored for the system call is -1.

When the `sc` instruction is encountered, the simulator is in exactly the same state as a real machine would have been except for the values in CTR, R0 and R2. While a real machine would have had a valid memory address pointing to the system call code in CTR and R0, the simulated machine has a unique number identifying the system call in R0. We compare R0 with the known unique values and appropriate system call code is "simulated". A value of -1 in R0 indicates that the simulated program is making an unsupported system call. When this happens, the return value from the system call is set to zero and a warning is printed to `stderr`.

7 Executing System Calls

A system call is exactly like a function call, except that it is OS code and not visible to the user. On the simulator, a system call results in the `sc` instruction being emulated as explained in the previous section. We first examine R0 to determine what system call has been made. System calls are passed arguments like any user level function, in the registers R3-R31. The arguments are read into variables in the simulator and the user level function call corresponding to the system call is called from the simulator with the arguments. Return values if any, are passed back by setting R3. Changes if any, that are made to the buffers are simulated by copying changes to the simulated memory. Figure 8 contains the code executed by `sc` when a `kread` system call is encountered.

System calls flags contain implicit meaning based on their values which vary across operating systems. Hence, on a non-AIX host, the system flags if any, have to be translated from the AIX values to the host OS values before the system call is made and back from host values to the corresponding AIX values. On Solaris for example, the second argument to the *fseek* system call is one of 0,1 or 2 meaning beginning, current or end of file respectively. On AIX the same argument contains the macro SEEK_SET, SEEK_END or SEEK_CUR to indicate the whence argument. For each system call, its system flags should be translated. This procedure is documented for SimpleScalar 3.0 and is explained in [1].

```
char *buf;
int retval;

buf = (char *) malloc(regs->regs_R[5]+1);
assert (buf != NULL);
retval = read(regs->regs_R[3], buf, regs->regs_R[5]);
/* write back output to simulated memory */
mem_bcopy(mem_access, mem, Write, regs->regs_R[4], buf, retval);
regs->regs_R[3] = retval;
free(buf);
```

Figure 8: Emulated source code for read system call

Name	Function	Address
divss	a = a % b return remainder	0x3200
divus	a = a % b (unsigned) return remainder	0x3280
quoss	a = a / b return quotient	0x3380
quous	a = a / b (unsigned) return quotient	0x3300
mulh	a = a * b (return high 32 bits)	0x3100
mull	a = a * b (return low 32 bits)	0x3200

Table 1: Millicode instructions

8 Other OS specific details

8.1 Millicode

A few operations in PowerPC are implemented using millicode. These are like function calls and the meaning of the arguments is implicit. There are 6 millicode instructions whose functions are defined in Table 1. On a real machine their location is fixed in memory and they are called by branching to their address. Program flow is resumed by saving the next Program Counter in the Link Register (LR) before branching and transferring program flow to the LR at the end of the millicode routine.

This behavior is faithfully simulated including the address where the millicode is located. The millicode is written to memory by the loader by calling the `writemillicode` function in `loader.c`.

8.2 System Configuration

AIX maintains a data structure called `system-configuration` which contains a number of fields describing the configuration of the system. The definition of the struct can be found at `/usr/include/sys/systemcfg.h` on an IBM AIX system. In the simulator we do not define all the fields of this struct. Only the architecture and implementation fields are set. Architecture is set to 0x02 and implementation is set to 0x10 corresponding to POWER-604.

```
lwzx rd,ra,rb
Input Dependencies: DNA,PPC_DGPR(RA),PPC_DGPR(RB),DNA,DNA
Output Dependencies: PPC_DGPR(RD),DNA,DNA,DNA,DNA

stwx rs,ra,rb
Input Dependencies:
PPC_DGPR(RS),PPC_DGPR(RA),PPC_DGPR(RB),DNA,DNA
Output Dependencies: DNA,DNA,DNA,DNA,DNA

DNA means no dependency.
PPC_DGPR is a macro that refers to the register file data
structure in the simulator.
RA, RB, RS are implicit arguments whose values are
detemined by decoding the instruction.
```

Figure 9: Example to illustrate input and output dependencies for the PowerPC ISA

9 Full Timing Simulation

`sim-outorder` is the detailed out-of-order pipeline simulator of the SimpleScalar's suite of simulators. Existing versions of SimpleScalar support the PISA and the Alpha ISA. We describe a port of `sim-outorder` to support the PowerPC architecture. The complexities of the PowerPC ISA as opposed to the simple PISA and Alpha ISA's present implementation challenges. This section describes the problems faced and consequent changes that were made to `sim-outorder` in order to port it to the PowerPC architecture. We made modifications in the timing simulator to handle the increased number of dependences an instruction is allowed to have in the PowerPC ISA. We also made several modifications to support misaligned accesses, complex memory instructions which write to memory and modify registers and a few complex floating point instructions that perform more than one simple floating point operation.

9.1 machine.def

`machine.def` contains the input/output dependencies and functional unit requirements for every supported PowerPC instruction. These specifications are crucial to ensure a correct and deadlock free timing simulation. These specifications are read by the timing simulator to enforce dependences and simulate out-of-order execution. Integer instructions are allowed up to have 5 input and 5 output dependence. For memory operations, a particular order was enforced in the specification of input dependencies. The first input dependence is the register value to be written to the memory (only for a store, no dependences for a load) and the second and the third input dependencies specify the input operands for effective address computation as shown in Figure 9 for an example load and store instruction.

9.2 Register and Memory Access Functions

The floating point and condition register access functions were rewritten for PowerPC as they were different from PISA and Alpha. Memory access functions were modified to ignore certain type of faults such as mis-alignment faults, since PowerPC allows addresses to be misaligned unlike PISA and Alpha. The PowerPC ISA supports a few complex floating point instructions that perform more than one simple floating point operations. We made a few modifications to account for the multiple cycles these instruction would require to execute.

9.3 Register Dependencies

PowerPC instructions may have up to five input and up to five output dependencies. For example, the `fnmsubsd rd,ra,rc,rb` instruction(Floating Negative Multiply-Subtract Single) uses all the five input dependencies (three source operands, `FPSCR` and `CR`). The `ruu_dispatch` and `ruu_issue` modules were augmented to check for these extra dependencies before firing the instruction execution.

9.4 Stores with Updates

Previous versions of SimpleScalar required that a store instruction does not modify the architected register file. When a store instruction is issued, it has all the information required from the architectural state (a register value) and the writeback pipeline stage is bypassed. In PowerPC, store instructions could modify the register file. For example, the `stwu` instruction stores a word in the memory and writes the effective address into a specified register. (`stwu rs,4(ra)` writes `4+(ra)` back into `ra`). To account for these register updates, in our implementation, all stores were made to go through the writeback stage.

9.5 Millicode

The Program Counter (PC) nearly always points to an address within the text segment. However, on a mis-speculated path, the PC can point to an address that lies outside the text boundary. `sim-outorder` puts in a semantic check to recognize these invalid addresses, and when these are encountered a `NOP (ori r0,r0,r0)`instruction is passed down the pipeline, instead of the invalid instruction. This behavior prevents invalid instructions from crashing the simulator. This check is done for every fetched instruction.

As described in section 8.1, PowerPC uses millicode to execute some arithmetic operations. Millicode resides in the lower memory which is outside the text area. During the execution of a millicode instruction, say the `mull` instruction, the PC contains the address corresponding to this millicode. This address must be interpreted as legal, even though it does not fall in the text segment. Hence, the semantic check described earlier should be augmented to recognize addresses that fall in the millicode area. There are a total of six millicode instructions and the code for these do not lie in one contiguous block. Instead of checking for each millicode address, only the boundaries are checked. This is an optimization to save simulation time, as this check needs to be performed every cycle.

9.6 Predecode

In SimpleScalar 3.0 predecode of the instruction stream is not done in `sim-outorder`. However, for reasons described in the previous sections we require a predecode for each of the simulators to make minor code modifications to handle system calls.

9.7 Load and Store Multiple Words

PowerPC has two fixed-point load and store multiple instructions (`LMW` and `STMW`) and 4 fixed-point move assist instructions (`LWSI,LSWX,STSWI, STSWX`). On PowerPC systems operating in little-endian byte order, executing these instructions causes the system alignment error handler to be invoked. On systems operating in big-endian byte order, they fetch/store one or more words from/to storage. Since these instructions access one or more words and hence one or more registers in a single instruction, they could cause a lot of register dependencies, potentially up to 32.

These instructions have been implemented as blocking instructions in the simulator and follow the big-endian behavior. We do not implement little-endian mode. Before an instruction of this class is dispatched, the pipeline is drained so that all the previous instructions are committed. No other instruction that follows this instruction is dispatched until this instruction has committed. Such an implementation makes sure that all dependencies with respect to previous and later instructions are satisfied correctly.

In order to correctly account for the memory stalls that may be caused by these instructions, the following has been done.

1. Each of the addresses that an instruction of this form accesses, is presented to the memory system one by one to check for tlb and cache hits.
2. The access latency for each address is computed and the total access latency for this instruction is found.

However, there is one problem with this implementation. All stores go to the Load/Store queue(LSQ) and loads first check this queue before going to the memory system. But according to the current implementation, for a `STMW` instruction, only the first word in the sequence of accesses is stored in the LSQ, subsequent words are not stored. Hence, the memory access penalty may not be captured accurately by the simulator. Most PowerPC hardware implementations use microcode to perform the LMW and STMW instructions and our implementation is a close approximation to what happens in reality.

9.8 Misaligned Accesses

Previous versions of SimpleScalar required memory addresses to be aligned on a word boundary and exited with a fault when a misaligned address was encountered. PowerPC however, allows memory addresses to be misaligned. A misaligned word access essentially translates to two consecutive accesses followed by a selection and combination of the correct set of bytes. The memory access functions of `sim-outorder` were modified to allow misaligned addresses. To correctly account for memory system latencies, the same solution, as described previously for load/store multiple instructions is adopted. The memory system is always presented with the correct number of accesses in case of a misaligned access. We have assumed here that a misaligned access can involve at the most one more memory access. Two consecutive addresses need to be presented only in the following cases.

- Misaligned word access
- Half-word access that spans two aligned words

The `ruu_dispatch` module was modified to detect misaligned accesses. Since the instructions are effectively executed in this stage, the memory address being accessed is known. Using this address and the type of memory operation (`LMW/LWZ etc.`), the required number of accesses needed to complete this memory instruction is computed. This involves checking for the type of memory instruction and checking if the address is aligned (i.e whether the next word needs to be accessed).

9.9 Floating Point Instructions

PowerPC has, as part of its floating point instructions, a set of instructions that perform a floating point multiply, add and possibly negate, all in one instruction. An example is the `fmadd rd,ra,rc,rb` instruction (Floating Point multiply add). We assumed that the multiplier unit has an add and negate block at the end and hence the functional unit latency for these instructions are assumed to be the same as that of a floating point multiply instruction.

10 Using SimpleScalar-PPC

The simulator is built according to the directions specified in [1] for SimpleScalar 2.0. Refer to that document for installation and usage of the tool set. Currently there are two separate `machine.def` files - one for native and another for non-native floating point implementations. For building the simulator do the following:

```
make config-ppc
make sim-fast
make sim-outorder
```

If you are building the simulator on a non-native host, you must use a different `machine.def` file. To do this, issue the following commands.

```
rm machine.def
ln -s target-ppc/powerpc-nonnative.def machine.def
make clean
make sim-fast
make sim-outorder
```

10.1 Compiler switches

A few of the instructions are defined only on some PowerPC implementations. These are the class of Floating Point rounding and conversion instructions. To enable the simulation of these instructions, the `FP_ROUND_CONVERSION_INST` macro should be defined. If your host machine does not implement these instructions, you will not be able to build the simulator. The problem does not arise for non-native builds of the simulator where all the floating point instructions are implemented in software. Since, GCC can compile for several PowerPC targets, the appropriate target should be chosen. To cover the entire PowerPC ISA, use the compiler switch `-mpowerpc`. The target identification flag is required to provide native support for floating point instructions on IBM AIX hosts.

The `FP_ROUND_CONVERSION_INST` macro and the `-mpowerpc` switch are by default defined in the Makefile.

10.2 Compiling Application Programs

Only programs that are statically linked can be simulated. To create static binary files using gcc, use the command:

```
gcc -static file.c -o file.out
```

If you are using the IBM AIX compiler, use the command:

```
cc -bnso -bI:/usr/lib/syscalls.exp file.c -o file.out
```

10.3 Bug Reports

Please send bug reports to *karu@cs.utexas.edu.*

11 Acknowledgments

We would like to thank Pat Bohrer, Tom Keller and Rick Simpson for their help in providing us with details about AIX system behavior.

References

[1] D. Burger and T. M. Austin, "The simplescalar tool set version 2.0," *University of Wisconsin-Madison Computer Sciences Department Technical Report*, June 1997.

[2] C. May, E. Silha, R. Simpson, and H. Warren, *The PowerPC Architecture: A Specification for a new family of RISC processors.* Morgan Kaufmann Publishers, May 1994.

APPENDIX A

List of instructions implemented. When instructions other than these are encountered, the simulator will come to a halt.

Instruction Name	Function
sc	Syscall
subf[o], subf[o].	Subtract From
subfic	Subtract From immediate carrying
subfc[o], subfc[o].	Subtract From carrying
subfe[o], subfe[o].	Subtract From extended
subfme[o], subfe[o].	Subtract From minus one extended
subfze[o], subfe[o].	Subtract From zero extended
add[o], add[o].	Add
addc[o], addc[o].	Add carrying
adde[o], adde[o].	Add extended
addi, addi.	Add immediate
addic, addic.	Add immediate carrying
addis	Add immediate shifted
addme[o], addme[o].	Add to minus one extended
addze[o], addze[o].	Add to zero extended
mulhd, mulhd.	Multiply high doubleword
mulhdu, mulhdu.	Multiply high doubleword unsigned
mulhw, mulhw.	Multiply high word
mulhwu, mulhwu.	Multiply high word unsigned
mulld[o], mulld[o].	Multiply low doubleword
mulli	Multiply low immediate
mullw[o], mullw[o].	Multiply low word
divw[o], divw[o].	Divide word
divwu[o], divwu[o].	Divide word unsigned
slw, slw.	Shift left word
srw, srw.	Shift right word
sraw, sraw.	Shift right algebraic word
srawi, srawi.	Shift right algebraic word immediate
cntlzw, ctlnzw.	Count leading zeros word
extsb, exstb.	Extend sign byte
extsh, exsth.	Extend sign halfword
extsw, exstw.	Extend sign word
cmp	Compare
cmpi	Compare immediate
cmpl	Compare logical
cmpli	Compare logical immediate
addi	Add immediate
addis	Add immediate shifted
xori	XOR immediate
xoris	XOR immediate shifted
and, and.	AND
andc, andc.	AND with complement

Instruction Name	**Function**
andi.	AND immediate
andis.	AND immediate shifted
or, or.	OR
orc, orc.	OR with complement
ori	OR immediate
oris	OR immediate shifted
nor, nor.	NOR
nand, nand.	NAND
xor, xor.	XOR
xori	XOR immediate
xoris	XOR immediate shifted
eqv, eqv.	Equivalent
neg[o], neg[o].	Negate
lbz	Load byte and zero
lbzu	Load byte and zero with update
lbzux	Load byte and zero with update indexed
lbzx	Load byte and zero indexed
lfd	Load Floating Point double
lfdu	Load Floating Point double with update
lfdux	Load Floating Point double with update indexed
lfdx	Load Floating Point double indexed
lfs	Load Floating Point single
lfsu	Load Floating Point single with update
lfsux	Load Floating Point single with update indexed
lfsx	Load Floating Point single indexed
lha	Load halfword algebraic
lhau	Load halfword algebraic
lhaux	Load halfword algebraic with update indexed
lhax	Load halfword algebraic indexed
lhbrx	Load halfword byte-reverse indexed
lhz	Load halfword and zero
lhzu	Load halfword and zero with update
lhzux	Load halfword and zero with update indexed
lhzx	Load halfword and zero indexed
lmw	Load multiple word
lswi	Load string word immediate
lswx	Load string word indexed
lwa	Load word algebraic
lwarx	Load word and reserve indexed
lwaux	Load word algebraic with update indexed
lwax	Load word algebraic indexed
lwbrx	Load word byte-reverse indexed
lwz	Load word and zero
lwzu	Load word and zero with update
lwzux	Load word and zero with update indexed
lwzx	Load word and zero indexed
stb	Store byte

Instruction Name	Function
stbu	Store byte with update
stbux	Store byte with update indexed
stbx	Store byte indexed
stfd	Store Floating Point double
stfdu	Store Floating Point double with update
stfdux	Store Floating Point double with update indexed
stfdx	Store Floating Point double indexed
stfiwx	Store Floating Point as integer word indexed
sts	Store Floating Point single
stfsu	Store Floating Point single with update
stfsux	Store Floating Point single with update indexed
stfsx	Store Floating Point single indexed
sth	Store halfword
sthbrx	Store halfword byte-reverse indexed
sthu	Store halfword with update
sthux	Store halfword with update indexed
sthx	Store halfword indexed
stw	Store word
stwbrx	Store word byte-reverse indexed
stwu	Store word with update
stwux	Store word with update indexed
stwx	Store word indexed
stmw	Store multiple word
stswi	Store string word immediate
stswx	Store string word indexed
b[[l][a]	Branch
bc[l][a]	Branch conditional
bclr[l]	Branch conditional to Link register
bcctr[l]	Branch conditional to Count register
crand	Condition register AND
crandc	Condition register AND with complement
cror	Condition register OR
crorc	Condition register OR with complement
crxor	Condition register XOR
crnor	Condition register NOR
crnand	Condition register NAND
creqv	Condition register Equivalent
rlwimi, rlwimi.	Rotate Left Word Immediate then Mask Insert
rlwinm, rlwinm.	Rotate Left Word Immediate then AND with Mask
rlwnm, rlwnm.	Rotate Left Word then AND with Mask
mcrf	Move Condition register field
mcrfs	Move to Condition register from FPSCR
mtfsb1, mtfsb1.	Move to FPSCR bit 1
mtfsb0, mtfsb0.	Move to FPSCR bit 0
mtfsfi, mtfsfi.	Move to FPSCR field immediate
mtfsf, mtfsf.	Move to FPSCR fields
mffs, mffs.	Move from FPSCR

Instruction Name	Function
fdiv, fdiv.	Floating Point Divide
fdivs, fdivs.	Floating Point Divide single
fsub, fsub.	Floating Point Subtract
fsubs, fsubs.	Floating Point Subtract single
fadd, fadd.	Floating Point Add
fadds, fadds.	Floating Point Add single
fmul, fmul.	Floating Point Multiply
fmuls, fmuls.	Floating Point Multiply single
fres, fres.	Floating Point Reciprocal
fneg, fneg.	Floating Point Negate
fabs, fabs.	Floating Point Absolute value
fnabs, fnabs.	Floating Point Negative Absolute value
fmsub, fmsub.	Floating Point Multiply-Subtract
fmsubs, fmsubs.	Floating Point Multiply-Subtract single
fnmsub, fnmsub.	Floating Point Negate Multiply-Subtract
fnmsubs, fnmsubs.	Floating Negate Multiply-Subtract Single
fmadd, fmadd.	Floating Point Multiply-Add
fmadds, fmadds.	Floating Point Multiply-Add single
fnmadd, fnmadd.	Floating Point Negate Multiply-Add
fnmadds, fnmadds.	Floating Point Negate Multiply-Add Single
fsqrt, fsqrt.	Floating Point Square Root
fsqrts, fsqrts.	Floating Point Square Root single
frsqrte, frsqrte.	Floating Point Reciprocal Square Root Estimate
fcmpo	Floating Point Compare ordered
fcmpu	Floating Point Compare unordered
frsp, frsp.	Floating Point round to Single-Precision
fctiw, fctiw.	Floating Point convert to integer word
fctiwz, fctiwz.	Floating Point convert to integer word with round toward zero
fmr, fmr.	Floating Point Move Register
fsel, fsel.	Floating Point Select

Notes:

- For all instructions, a dot suffix indicates that the result of the instruction is compared with zero and CR0 bit 0, 1 or 2 is set depending on whether the result is less than, greater, or equal to zero. Further, CR0 bit 3 is set to the Summary Overflow (SO) of the XER register after completion of the instruction execution.

- A dot suffix for a floating point instruction indicates that, CR1 is set to 0, 1 or 2 is set depending on whether the result is less than, greater, or equal to zero.

- Fixed point instructions which have an optional “o” suffix update the XER register.

- For the branch processor instructions, a suffix of “l” indicates that the Link Register is updated by the instruction. A suffix of “a” denotes that the branch target address is calculated by adding a computed value to the address of the current instruction. The corresponding instruction without the suffix “a”, would simply compute a branch target address and return it.