The Unlimited Resource Machine (URM)

David A. Poplawski

February 1995

Abstract

The Unlimited Resource Machine (URM) is a hypothetical instruction set architecture that was designed specifically for study of the architecture of and code generation techniques for machines exhibiting instruction level parallelism. A set of tools have been implemented to support this machine model, including compilers, a (linking) assembler, simulators, and trace analyzers. The architecture and its tools are being used in a variety of research projects at MTU. This report describes the philosophy behind the URM, the architecture and instruction set, the tools and their interfaces, and how to use the tools to model various ILP architectures.

Introduction

The Unlimited Resource Machine[1] (URM) is a hypothetical instruction set architecture that was designed specifically for study of the architecture of and code generation techniques for machines exhibiting instruction level parallelism. A set of tools have been implemented to support this machine model, including compilers, a (linking) assembler, simulators, and trace analyzers. The architecture and its tools are being used in a variety of research projects at MTU.

Modeling Strategy

All modeling of instruction level parallel architectures is based on the generation and use of a trace file of the sequential execution of a URM program. This trace file is then used as input to a program that simulates the action of the instructions in the context of some target parallel architecture.

The separation of the simulation of the semantics of the instructions and the analysis of their action was done for three reasons. First, it simplifies the construction of the execution simulator. The simulator focuses on the semantics of each instruction, one at a time, much the way a simple sequential implementation of a URM computer would. Since all ILP architectures are supposed to preserve the semantics of the program, as defined by the sequential execution of instructions, no aspect of parallelism will affect the results (e.g., outputs of the program, operands used by instructions).

Second, it simplifies the construction of analysis tools. The analyzer does not need to actually simulate the semantics of each instruction (i.e., computing values), so can be focused on simulating the effect of each instruction in the parallel architecture. This simplifies the analyzer and reduces the probability of errors.

Third, only one execution simulator needs to be written and debugged. For pragmatic reasons there are currently two simulators, one a subset of the other. Since we wanted to study speculative execution, the more capable simulator has the ability to execute, then undo the effect of, instructions that wouldn't have executed in a normal sequential execution, and include these "speculatively executed" instructions in the trace of the execution (suitably marked). In the remainder of this document we will, by default, be talking about the more capable simulator. The simpler simulator has all the same capabilities except the ability to do speculative execution.

The Trace Files

The trace of execution of a URM program contains a record for every instruction executed, in the order they would be executed in a sequential machine. The basic trace file contains the following information:

1. A flag for speculative execution, essentially identifying the path of execution past one or more incorrectly exe-

---

1. In previous publications URM stood for "Uniform" Resource Machine.

cuted branch instructions. This field is not present in the trace file produced by the simpler simulator.
2. The address of the instruction.
3. The op-code of the instruction.
4. A list of registers and memory locations accessed, including the number of bytes and whether each access was a read or a write.

Instruction Set Architecture

The philosophy behind the instrcution set architecture is to use a generic collection of instructions that correspond to basic C operations. Instructions exist for manipulating 8 bit characters, 16 and 32 bit integers, and 32 and 64 bit floating point numbers. The number of registers is not bounded, nor is the amount of memory (except as dictated by the available virtual memory on the machine the simulator runs on). Any register can be treated as either 32 bit integers (with 8 bit characters and 16 bit integers occupying the lower bits), as a 32 bit float, or as a 64 bit float. Explicit conversion instructions can be used to convert between types. Support for function calls exists in the form of push, pop, call and return instructions, all of which use an stack pointer register to implement an implicit negative growing stack.

Although the URM instruction set is rather rich, it is assumed that one will choose a subset of the instructions (and addressing modes), pick a fixed number of registers, and target a compiler to this subset. Simple RISC architectures should be easy to define, and most existing architectures can be “approximated”. To assist such approximations, a mechanism for grouping two or more instructions so as to treat them as a single instruction has been included. This allows for the “implementation” of (more complex) instructions that are not present in the URM set. This grouping mechanism could also be used to create VLIW-like instructions. To be useful, the grouping information is included in the trace file created by the simulator.

Tools

The basis of the toolset for the URM is the assembler and instruction-level simulator. It is intended that these tools be generic enough to support the simulation of a large range of ILP architectural characteristics without having to be rewritten (or even modified). Given these tools, compilers exhibiting a range of parallel code generation techniques and optimizations can be targetted, and analyzers simulating the effects of various parallel architectural features can be written.

Intermediate File Formats

The interface between the compiler and assembler is the assembly language file. Appendix B describes the format of this file.

The assembler outputs an “executable” (by the simulator) file. This file contains a line for each instruction or data initializing assembler command (e.g., .word). The first line contains two hexadecimal integers - the address of the first unused memory location above the program and data, and the highest register number used by the program. Each successive line contains two hexadecimal integers followed by a string of hexadecimal digits. The first hex number is a code indicating how to interpret what follows. A code of 0 indicates an executable instruction, non-zero indicates data, with the value of the code indicating how many “bytes” of data follow. The second hex number is the starting address of the instruction or data that follows. The string of hex digits after that are the encoding of an instruction or the actual bytes of data. In the case of an instruction the number of bytes can be determined from the addressing mode information in the instruction itself.

The simulator outputs a trace of the execution of the program supplied as input. The output is a series of “records” (in binary), one record per instruction executed. Appendix C describes the format of this file.

Compiler(s)

Compilers that currently generate URM code are based on the ROCKET [1] retargetable compiler.

Assembler

The two-pass assembler reads one or more assembler files and generates an executable object file. Since the

URM system has no linker, the function of linking multiple files and resolving references between files is done by the assembler. Library functions, also in assembly language, are also linked in. The assembler is invoked by the command:

asm *executable-file asm-file-1 asm-file-2* ... [-l*ibrary*]

The files *asm-file-1*, *asm-file-2*, etc will be assembled and linked, and the output written to *executable-file*. One of the assembly language files must contain a function named _main, which must be declared global in that file through the use of the .globl assembler command. The *executable-file* is appropriate for input to the simulator. If no library is specified, a default library is linked in. This library uses a stack-based calling sequence.

Library

A library contains interfaces to printf and scanf functions, as well as a few mathematical functions (see Appendix B). Most library functions consist of a "trap" instruction surrounded by a function entry/exit wrapper. However some are actually implemented directly in URM instructions. In both cases assumptions have been made about the function calling conventions (including parameter passing) as well as which URM instructions to use. Since different URM subsets, with different calling conventions, may be desired, different implementations of the library will be needed.

Simulator

The simulator reads an object file and simulates the execution, instruction by instruction, of the program contained therein. A trace of the execution can be produced upon request, either into a file, or piped into another command (using popen). All information about grouped instructions appears in the trace file.

The simulator is invoked by the command:

sim [-s] [-t *trace-file* | -p *command*] [-m *memsize*] [-w *window-size*] *executable-file*

where *trace-file* is the name of the file in which to put the execution trace, *command* is a command to execute that will read the *trace-file* from stdin, *memsize* is the number of bytes of simulated memory to use (default is 1 Mbyte), and *executable-file* is the output of the assembler. *Window-size* is the number of instructions to execute down any wrong path from a conditional branch instruction. Since there may be other conditional branches after the first, there may be a tree of paths. *Window-size* is the height of that tree. If some instruction along any path down the tree causes an execution error (such as reference out of the address space) or is a halt instruction, that path is (prematurely) terminated. Hence some paths may not be *window-size* instructions long. The -s flag will cause the simulator to collect and print statistics about the executed program.

Analyzer(s)

An analyzer uses a trace file from one of the simulators and produces information about the execution of the program. Most analyzers simulate some architectural feature (or combination of features), such as pipelining, functional units, cache, etc., and many involve some aspect of timing. Since there can be such a wide range of analyzers, no specific description of one of them is included here. However many have already been constructed for various purposes (class projects, thesis work and other research)

Miscellaneous Tools

A disassembler reads an executable file and prints the contents in an assembly language like format. This can be useful in debugging as the address of each instruction and the actual operand values are shown.

A program for printing the contents of a trace file can be used to get a readable version of a trace file. Versions for both forms of trace file exist.

References

1. Sweany, Phillip H. and Beaty, Steven J. Overview of the ROCKET Retargetable C Compiler. Technical Report CS-94-01, Department of Computer Science, Michigan Technological University. January 1994.

APPENDIX A - URM INSTRUCTION SET

Registers

The URM has an unlimited number of registers, numbered starting at 0. Register 0 is assumed to be the stack pointer, and is used implicitly by several instructions. Any register can contain either a four byte integer, a four byte single precision floating point number (a float), or an eight byte double precision floating point number (a double). An attempt to use an integer value in a register as a float or double is undefined. An attempt to use a float value in a register as an integer or double is undefined. An attempt to use a double value in a register as an integer or float is undefined.

Memory

The URM has an unlimited number of 8-bit bytes of memory (actually it is limited by the implementation of the simulator on a given architecture - practically there will be considerably fewer than 2^{32} bytes). Memory is byte addressible.

The Stack

The URM instruction set supports a negatively growing stack. By convention the stack pointer contains an address four bytes less than the last thing on the stack. The crt0 instruction initializes the stack pointer to point to the 7 bytes less than the last implemented memory location (it also does other things - see the description below).

Addressing Modes

The URM has five addressing modes:

Mode	Assembler Syntax	Examples
Immediate	$value	$78, $0x9, $-88, $x, $x+5, $x-17
Register	**r**num	r9, r100, sp (an alias for r0)
Direct	value	4985, x, x+99, x-0x4c
Register Displacement	value(**r**num)	0(r1), 0x55(sp), -6(r5), x+8(r9)
Memory Displacement	$value_1$(value$_2$)	0(100), 0x55(0x34)

All values are signed, 32-bit numbers. The result, when interpreted as an address, is an unsigned value (hence a negative result will be interpreted as a very large address).

When used as a source:

Immediate	The value itself
Register	The contents of the register
Direct	The contents of memory at the value
Register Displacement	The contents of memory at the address that is the sum of the contents of the register and the displacement value
Memory Displacement	The contents of memory at the address that is the sum of the contents of the index memory word and the displacement value

If the instruction specifies a byte operand, the lower byte of the immediate or register, or a single byte of memory, is used. If the instruction specifies a halfword operand, the lower two bytes of the immediate or register, or two bytes of memory, are used. If the instruction specifies a word operand, the entire immediate or register, or four bytes of memory, are used. If the instruction specifies a float operand, the entire register, or four bytes of memory, are used. If the instruction specifies a double operand, the entire register, or eight bytes of memory, are used.

When used as a destination:

Immediate	Illegal
Register	The register is changed
Direct	The memory location at the value is changed
Register Displacement	The memory location at the address that is the sum of the displacement value and the contents of the register is changed
Memory Displacement	The memory location at the address that is the sum of the contents of the index memory word and the displacement value is changed

If the instruction specifies a byte operand, then only the lower byte of the register, or a single byte of memory, is changed. If the instruction specifies a halfword operand, then only the lower two bytes of the register, or just two bytes of memory, are changed. If the instruction specifies a word or float operand, the entire register, or four bytes of memory are changed. If the instruction specifies a double operand, the entire register, or eight bytes of memory, are changed.

When used as a branch target:

Immediate	Illegal
Register	The contents of the register is the address transfered to
Direct	The value itself is the address transfered to
Register Displacement	The sum of the contents of the register and the displaement value is the address transfered to
Memory Displacement	The sum of the contents of the index memory word and the displacement value is the address transfered to

Instructions

Each instruction consists of an operation code followed from zero to three operands. The type of each operand is determined by the operation code.

In the descriptions below, an operand can be a source (“src”) or destination (“dst”) or a branch “target”. The type of each operand is indicated by the suffix (b - byte, h - halfword, w - word, f - float, d - double). A prefix of “r” indicates that the operand must be a register. A prefix of m indicates that the operand must be a memory location (i.e., a direct, register displacement or memory displacement mode).

Data Movement

ldb	rdst.b,msrc.b	load byte
ldh	rdst.h,msrc.h	load halfword
ldw	rdst.w,msrc.w	load word
ldf	rdst.f,msrc.f	load float
ldd	rdst.d,msrc.d	load double
stb	rsrc,.b,mdst.b	store byte
sth	rsrc.h,mdst.h	store halfword
stw	rsrc.w,mdst.w	store word
stf	rsrc.f,mdst.f	store float
std	rsrc.d,mdst.d	store double
movb	dst.b,src.b	move byte
movh	dst.h,src.h	move halfword
movw	dst.w,src.w	move word
movf	dst.f,src.f	move float
movd	dst.d,src.d	move double

Arithmetic

addb	dst.b,src1.b,src2.b	add byte
addh	dst.w,src1.h,src2.h	add halfword
addw	dst.h,src1.w,src2.w	add word

addf	dst.f,src1.f,src2.f	add float
addd	dst.d,src1.d,src2.d	add double
subb	dst.b,src1.b,src2.b	subtract byte (dst = src1-src2)
subh	dst.h,src1.h,src2.h	subtract halfword
subw	dst.w,src1.w,src2.w	subtract word
subf	dst.f,src1.f,src2.f	subtract float
subd	dst.d,src1.d,src2.d	subtract double
mulb	dst.b,src1.b,src2.b	multiply byte
mulh	dst.h,src1.h,src2.h	multiply halfword
mulw	dst.w,src1.w,src2.w	multiply word
mulf	dst.f,src1.f,src2.f	multiply float
muld	dst.d,src1.d,src2.d	multiply double
divb	dst.b,src1.b,src2.b	divide byte (dst = src1/src2)
divh	dst.h,src1.h,src2.h	divide halfword
divw	dst.w,src1.w,src2.w	divide word
divf	dst.f,src1.f,src2.f	divide float
divd	dst.d,src1.d,src2.d	divide double
modb	dst.b,src1.b,src2.b	modulo byte (dst = src1%src2)
modh	dst.h,src1.h,src2.h	modulo halfword
modw	dst.w,src1.w,src2.w	modulo word

Logical

bcompb	dst.b,src.b	bitwise complement byte (ones complement)
bcomph	dst.h,src.h	bitwise complement halfword
bcompw	dst.w,src.w	bitwise complement word
bandb	dst.b,src1.b,src2.b	bitwise and byte
bandh	dst.h,src1.h,src2.h	bitwise and halfword
bandw	dst.w,src1.w,src2.w	bitwise and word
borb	dst.b,src1.b,src2.b	bitwise or byte
borh	dst.h,src1.h,src2.h	bitwise or halfword
borw	dst.w,src1.w,src2.x	bitwise or word
bxorb	dst.b,src1.b,src2.b	bitwise exclusive or byte
bxorh	dst.h,src1.h,src2.h	bitwise exclusive or halfword
bxorw	dst.w,src1.w,src2.w	bitwise exclusive or word
blshb	dst.b,src1.b,src2.w	bitwise left shift byte (dst = src1 << src2)
blshh	dst.h,src1.h,src2.w	bitwise left shift halfword
blshw	dst.w,src1.w,src2.w	bitwise left shift word
brshb	dst.b,src1.b,src2.w	bitwise right shift byte (dst = src1 >> src2)
brshh	dst.h,src1.h,src2.w	bitwise right shift halfword
brshw	dst.w,src1.w,src2.w	bitwise right shift word

Note: the shift instructions use C semantics for left and right shifts.

Conversions

cvtbh	dst.h,src.b	convert byte to halfword
cvtbw	dst.w,src.b	convert byte to word
cvtbf	dst.f,src.b	convert byte to float
cvtbd	dst.d,src.b	convert byte to double
cvthb	dst.b,src.h	convert halfword to byte
cvthw	dst.w,src.h	convert halfword to word
cvthf	dst.f,src.h	convert halfword to float
cvthd	dst.d,src.h	convert halfword to double

cvtwb	dst.b,src.w	convert word to byte
cvtwh	dst.h,src.w	convert word to halfword
cvtwf	dst.f,src.w	convert word to float
cvtwd	dst.d,src.w	convert word to double
cvtfb	dst.b,src.f	convert float to byte
cvtfh	dst.h,src.f	convert float to halfword
cvtfw	dst.w,src.f	convert float to word
cvtfd	dst.d,src.f	convert float to double
cvtdb	dst.b,src.d	convert double to byte
cvtdh	dst.h,src.d	convert double to halfword
cvtdw	dst.w,src.d	convert double to word
cvtdf	dst.f,src.d	convert double to float

Note: all values are assumed to be signed when converting to larger types.

Comparisons

cmpeqb	dst.w,src1.b,src2.b	compare equal byte (dst = (src1==src2))
cmpeqh	dst.w,src1.h,src2.h	compare equal halfword
cmpeqw	dst.w,src1.w,src2.w	compare equal word
cmpeqf	dst.w,src1.f,src2.f	compare equal float
cmpeqd	dst.w,src1.d,src2.d	compare equal double
cmpneb	dst.w,src1.b,src2.b	compare not equal byte (dst = (src1 != src2))
cmpneh	dst.w,src1.h,src2.h	compare not equal halfword
cmpnew	dst.w,src1.w,src2.w	compare not equal word
cmpnef	dst.w,src1.f,src2.f	compare not equal float
cmpned	dst.w,src1.d,src2.d	compare not equal double
cmpltb	dst.w,src1.b,src2.b	compare less than byte (dst = (src1 < src2))
cmplth	dst.w,src1.h,src2.h	compare less than halfword
cmpltw	dst.w,src1.w,src2.w	compare less than word
cmpltf	dst.w,src1.f,src2.f	compare less than float
cmpltd	dst.w,src1.d,src2.d	compare less than double
cmpleb	dst.w,src1.b,src2.b	compare less than or equal byte (dst = (src1 <= src2))
cmpleh	dst.w,src1.h,src2.h	compare less than or equal halfword
cmplew	dst.w,src1.w,src2.w	compare less than or equal word
cmplef	dst.w,src1.f,src2.f	compare less than or equal float
cmpled	dst.w,src1.d,src2.d	compare less than or equal double
cmpgtb	dst.w,src1.b,src2.b	compare greater than byte (dst = (src1 > src2))
cmpgth	dst.w,src1.h,src2.h	compare greater than halfword
cmpgtw	dst.w,src1.w,src2.w	compare greater than word
cmpgtf	dst.w,src1.f,src2.f	compare greater than float
cmpgtd	dst.w,src1.d,src2.d	compare greater than double
cmpgeb	dst.w,src1.b,src2.b	compare greater than or equal byte (dst = (src1 >= src2))
cmpgeh	dst.w,src1.h,src2.h	compare greater than or equal halfword
cmpgew	dst.w,src1.w,src2.w	compare greater than or equal word
cmpgef	dst.w,src1.f,src2.f	compare greater than or equal float
cmpged	dst.w,src1.d,src2.d	compare greater than or equal double

Transfers of Control

beqb	src1.b,src2.b,target	branch equal byte (if src1 == src2)
beqh	src1.h,src2.h,target	branch equal halfword
beqw	src1.w,src2.w,target	branch equal word
beqf	src1.f,src2.f,target	branch equal float
beqd	src1.d,src2.d,target	branch equal double

bneb	src1.b,src2.b,target	branch not equal byte (if src1 != src2)
bneh	src1.h,src2.h,target	branch not equal halfword
bnew	src1.w,src2.w,target	branch not equal word
bnef	src1.f,src2.f,target	branch not equal float
bned	src1.d,src2.d,target	branch not equal double
bltb	src1.b,src2.b,target	branch less than byte (if src1 < src2)
blth	src1.h,src2.h,target	branch less than halfword
bltw	src1.w,src2.w,target	branch less than word
bltf	src1.f,src2.f,target	branch less than float
bltd	src1.d,src2.d,target	branch less than double
bleb	src1.b,src2.b,target	branch less than or equal byte (if src1 <= sec2)
bleh	src1.h,src2.h,target	branch less than or equal halfword
blew	src1.w,src2.w,target	branch less than or equal word
blef	src1.f,src2.f,target	branch less than or equal float
bled	src1.d,src2.d,target	branch less than or equal double
bgtb	src1.b,src2.b,target	branch greater than byte (if src1 > src2)
bgth	src1.h,src2.h,target	branch greater than halfword
bgtw	src1.w,src2.w,target	branch greater than word
bgtf	src1.f,src2.f,target	branch greater than float
bgtd	src1.d,src2.d,target	branch greater than double
bgeb	src1.b,src2.b,target	branch greater than or equal byte (if (src1 >= src2)
bgeh	src1.h,src2.h,target	branch greater than or equal halfword
bgew	src1.w,src2.w,target	branch greater than or equal word
bgef	src1.f,src2.f,target	branch greater than or equal float
bged	src1.d,src2.d,target	branch greater than or equal double
jmp	target	jump (always)

Function Calls

pushb	src.b	push byte (then decrement stack pointer by 1)
pushh	src.h	push halfword (then decrement stack pointer by 2)
pushw	src.w	push word (then decrement stack pointer by 4)
pushf	src.f	push float (then decrement stack pointer by 4)
pushd	src.d	push double (then decrement stack pointer by 8)
popb	dst.b	pop byte (increment stack pointer by 1 first)
poph	dst.h	pop halfword (increment stack pointer by 2 first)
popw	dst.w	pop word (increment stack pointer by 4 first)
popf	dst.f	pop float (increment stack pointer by 4 first)
popd	dst.d	pop double (increment stack pointer by 8 first)
call	target	“pushw” address of next instruction, jump to target
callr	rdst.w,target	put address of next instruction into rdst.w, jump to target
ret		“popw” value from stack, jump to that value

Miscellaneous

crt0	direct	initialize stack pointer, pushw return address, jump to direct
nop		do nothing
trap	imm, msrc.w	invoke simulator function (code = “imm”, params at “src.w”)
halt		terminate execution

Note: the crt0 instruction requires a “direct” addressing mode, and the trap instruction requires an “immediate” addressing mode.

Debugging

regdmp	dump all registers used by program

Trap Functions

The “trap” instruction is used to provide access to what would normally be operating system or complex library functions. The execution of a trap instruction causes the simulator to take zero or more parameters starting at the address indicated by the second operand of the trap instruction, call the function indicated by the first operand, and return the result at the address indicated by the second operand. The stack are all registers are left unchanged. The functions currently implemented are:

_atan	expects a double, returns a double
_cos	expects a double, returns a double
_exit	no parameters expected, never returns (executes a halt)
_exp	expects a double, returns a double
_fabs	expects a double, returns a double
_log	expects a double, returns a double
_printf	expects an address of a format string and a value
_scanf	expects an address of a format string and an address for the value, returns an integer
_sin	expects a double, returns a double
_sqrt	expects a double, returns a double

APPENDIX B - ASSEMBLER SYNTAX AND OPERATIONS

Syntax

Any line of input may contain a comment, which begins with a pound sign (#) and ends at the end of the line. Everything from the pound sign to the end of the line is ignored by the assembler, except the end of the line itself, which, as described below, terminates a statement.

Input is free form - there are no indentation or column requirements. White space (spaces and tabs) is optional, except as stated below.

A program is a sequence of statements. Each statement can contain zero or more labels, which need not be on the same line as the statement, but must preceed it. Each label is followed by a colon. Blank lines can appear between labels and the statment they label. A label consists of a sequence of lower/upper case letters, digit, and the underscore, and must not begin with a digit.

The value associated with a label is equal to the address of the first byte of the statement it labels, including any alignment that is required by the statement. The scope of a label is the file in which it occurs, unless it is made global to all files being linked via the .globl assembler command.

A statement is either a list of executable instructions or assembler commands separated by semicolons. It terminates with the end of the line the statement is on. Statements cannot span more than one line. All instructions or assembler commands consist of an operation followed by zero or more operands. The operation and its operands must be separated by at least one space or tab. Operands are separated by commas.

Examples:

```
            instruction                 # an unlabeled statement containing one instruction
L1:         command                     # a statement with one label (L1) containing one command
_main:
            instruction1; instruction 2 # another statement with one label (_main) containing
                                        # two instructions
L2:
L3: L4:
L5:
            instruction                 # a statement with 4 labels (L2-L5) containing one instruction
```

Assembler Commands

Assembler commands are used to give the assembler information about the scope of a name (.globl), reserve uninitialized memory (.space), or initialize memory locations to a value (.byte, .halfword, .word, .float, .double, .string).

.globl	name	declares name to be known outside this file
.space	bytes,alignment	reserves bytes bytes, aligned according to alignment
.byte	value	stores value as a fixed point value in 1 byte value may be an integer or a C character constant
.halfword	value	stores value as a fixed point value in 2 bytes, aligned on a 2-byte boundary value must be an integer
.word	value	stores value as a fixed point value in 4 bytes, aligned on a 4-byte boundary value can be an integer or a name
.float	value	stores value as a single precision value in 4 bytes, aligned on a 4-byte boundary value must be C float constant
.double	value	stores value as a double precision value in 8 bytes, aligned on an 8-byte boundary value must be C double constant
.string	value	stores value as a null terminated string value must be C string

Examples:

```
        .globl      _main       # declares _main to be global to all files being linked
stuff:  .space      16, 4       # reserves 16 bytes aligned on a multiple of 4 boundary
        .byte       7           # initializes a single byte to the value 7
        .halfword   28          # initializes two bytes to the value 28
        .word       1951        # initializes four bytes to the integer value 1951
        .word       abc         # initializes four bytes to the address of abc
myval:  .float      -8.43       # initializes four bytes to the float value -8.43
        .double     8.43E4      # initializes eight bytes to the float value 8.43x10^4
```

Miscellaneous

The assembler automatically puts a “crt0 _main” instruction at memory location 0. This instrucution causes the the stack pointer to be initialized to the highest usable memory locations and then “call”s the function _main.

The assembler automatically assembles a library of “system call”s at the end of the program. This library contains:

_atan
_cos
_exit
_exp
_fab
_free
_log
_malloc
_printf
_scanf
_sin
_sqrt

To call any of the mathematical functions, push a double onto the stack and call the function. The result is returned in r2 as a double.

To call malloc, push an integer onto the stack and call the function. A result is returned in r2 as an integer. To call free, push the address of the block onto the stack and call the function. (Currently malloc allocates space but free does

not free it. Therefore use these functions with care).

To call _scanf, push the address of a word on the stack, then push the address of the format string, then “call _scanf”, and upon return, add 8 to the stack pointer. The return value of from _scanf is returned in r2 as an integer. To call _printf, push a word value on the stack, then push the address of the format string, then “call _printf”, and upon return, add 8 to the stack pointer. In both cases, the format string can contain only a single format descriptor.

APPENDIX C- TRACE FILE FORMAT

The trace file consists of a series of records, one per instruction executed by the simulator. Each record contains the following:

A four byte speculative execution flag (not present in the trace file from the simpler, non-speculative simulator). For all normally executed instructions except conditional branches this flag is 0. For normally executed conditional branches it is either 0 (if the branch was taken) or 0x80000000 (if the branch was not taken). For instructions that should not have been executed (down wrong paths of the program), the value of the flag indicates how prior branch instructions were executed (correctly or incorrectly) to get to that instruction. Consider a flag value $1b_ib_{i-1}$ ... b_2b_1. This value implies that i conditional branch instructions were executed after the first branch that started the program down the wrong path. If b_i is a 1 then ith previous conditional branch was correctly executed to get to this instruction. If b_i is a 0 then the ith previous conditional branch was incorrectly executed to get to this instruction. For example, the flag value 1101 imples that the previous conditional branch instruction was correctly executed, the one before that incorrectly executed, and the one before that correctly executed. Of course the branch that started the program down the incorrect path was incorrectly executed. The 1 bit to the left of b_i is a marker to indicate how many branch instructions were executed down the wrong path. It serves no other purpose. If a condition branch instruction in the trace down an incorrect path has a negative flag, then that specific branch instruction was taken (a positive flag means not taken). Simply negate the flag value (i.e., make it positive) to get the bits indicating the preceeding branch instruction behavior.

The next byte of the record is the opcode of the instruction executed.

The next three bytes contain 6 four bit fields, one for each register or memory operand referenced by the instruction when it executed. Each four byte field consists of a two bit indicator of the size of the operand (0 -> 1 byte, 1 -> 2 bytes, 2 -> 4 bytes, 3 -> 8 bytes) and a two bit indicator of whether the operand was read or written (1 -> read, 2 -> written). If fewer than 6 operands were accessed, then only the first few four bit fields are set - the remainder are 0.

For each operand describe by a four bit field there will be four bytes following the three bytes of fields. If the reference was to a register then the four byte field will be the negative of the register number (e.g., -5 means register 5, 0 means register 0). If the reference was to a memory location, the four byte field will be the memory address (memory address 0 cannot be expressed since 0 means register 0).