MIT Concurrent VLSI Architecture Memo 63

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

MSIM Users Guide

Steve Keckler, Whay S. Lee, Nick Carter

Abstract

This memo describes the usage of the M-Machine simulator MSIM. A quick tutorial of the simulator is provided, including how to run a sample program and display the state of the MAP processor. We then describe the initial state of the simulator, and how to load and run program. The differences between the current implementation of the simulator (v1.4) and the architecture specification are described. Finally the commands available while running the simulator are listed.

Keywords: M-Machine, msim, massl, mas.

1 MSIM Tutorial

1.1 Introduction

MSIM is an instruction level simulator for the M-Machine. It can operate in interactive command-line mode and include commands from a text file. This section gives a brief overview of how to use msim.

First we introduce some conventions for filename extensions:

- `.m`: Massl[1] assembly programs.
- `.o`: Object files produced the assembler `mas`.
- `.out`: Executable files produced the linker `lnk`. This is the input program file needed by msim.
- `.cmd`: Command files for msim.

All of the files discussed by name in this tutorial reside in `/home/mm/tools/test`.

The msim executable is named `msim`, and can accept a number of optional arguments at invocation:

`msim` [opts] <program file> <cmd file>

Options:

- −asm <filename>: load program into memory of all nodes
- −cmd <filename>: read command file and execute commands
- −x_dim <int>: X dimension of cube. Defaults to 1
- −y_dim <int>: Y dimension of cube. Defaults to 1
- −z_dim <int>: Z dimension of cube. Defaults to 1

If multiple arguments are given to `msim`, without "-", the first is assumed to be the program and the second a command file. If the program file is omitted no program is loaded. If the command file is ommitted, no commands are executed. If no arguments are given, the simulator enters interactive mode directly.

1.2 Command File

The command file is used to automatically execute msim commands without typing them at the command line. It may contain any valid msim command line instructions listed in Section 4. The command line instructions from this file, if specified at the msim invocation, will execute as if they were entered interactively. A command file may also be included using the `include` command. Figure 1 shows a sample command file, *sample.cmd*.

If no command file is specified, or when the end of the command file has been reached, msim enters interactive mode. We will demonstrate the interactive mode in Section 1.5.

[1] Refer to *System Tools for the M-Machine*.

```
; This is a sample command file

; if uncommented, the following line includes another command file here.
; include "foobar.cmd"

; load up the memory with some data.
; it is probably better to use a data segment since it will be relocatable.
init_memory "sample.dat"

step 250

; monitor on map 0, vthread 0, cluster 0, ialu, reg 8
watch_reg 0 0 0 0 8

; take a look at this range of memory
dv $35000 $35010
```

Figure 1: Sample command file, *sample.cmd*

```
ORG 0x35000
133
```

Figure 2: *sample.dat*

1.3 Object File

The `.out` object file is the M-machine program executed by msim. A `.out` file is obtained from assembling a `.m` file using the M-Machine Assembler[2]. Figure 3 shows a sample massl program that will be used in the tutorial example.

1.4 System Software

The MAP requires a number of system threads to be running in order to provide services such as exception handling, address translation, and synchronization event handling. The object files containing this code must be linked into the executable.

The makefile shown in Figure 4 links in the appropriate object files. The system code starts up exception handler threads in V-Thread 5, and event handler threads which can handle synchronization misses, LTLB misses, and GTLB misses, in V-Thread 4. Note the distinction between *exceptions*, which are error conditions caused by incorrect code, and *events*, which are situations, such as LTLB misses, which require software intervention to allow the user program to complete, but do not imply an error in the program.

In addition, stack space is set up and the stack pointer (SP=i2), frame pointer (FP=i3), and data pointer (DStart=i5) are initialized. The data pointer points to the beginning of the user data segment. Finally, a branch to `_main` is taken to jump to user code.

The user's data space begins at the location specified by the user space pointer and grows upward. All subsequent memory references should be to addresses derived from this pointer, so that code, page tables,

[2]Refer to *System Tools for the M-Machine*.

```
/**************************************************************
*  sample.m:
*    Divides a number in memory by z, through iterative subtraction.
*
**************************************************************/

text;                                /* this is needed by massl */

#include "regdef.h"                  /* system headers */
#include "std.h"
#include "stack.h"

x := 18;                             /* some constants */
y := 9;
z := (x+y);

/* some CPP definitions */

#define nreg i6                      /* nreg = N % z */
#define zreg i7
#define divreg i8      /* divreg = N/z */
#define DataPtr DStart               /* i2 contains pointer to data segment */

/* always begin with the _main label. This is assumed by the startup code. */

_main::                              /* this is a label */

  instr ialu        mov #z, zreg
   memu        lea DataPtr, _val, nreg;
  instr ialu        mov IntZero, divreg /* IntZero aliased to i0 in regdefs.h */
        memu        ld nreg, nreg;      /* load 133 into nreg */

  /* The system startup code leaves a user data pointer in i2.  We use the
     label _val in the data segment to reference the value 133 indirectly. */

  instr ialu        ige zreg, nreg, cc2;
  instr ialu ct cc2 br _done;
  instr;                      /* delay slots */
  instr;
  instr;

_loop:      /* user labels prefixed with underscore by convention */
  instr ialu        igt zreg, nreg, cc2;
  instr ialu cf cc2 br _loop;
  instr ialu cf cc2 add divreg, #1, divreg
        memu cf cc2 sub nreg, zreg, nreg;
  instr;
  instr;
      /* divreg = N/z, nreg = N % z */
_done:
  instr ialu        br pass;         /* user pass routine provided in system */
  instr;                             /* there is also a "fail" routine. */
  instr;
  instr;

data USER;
_val: u64 133;      /* initialize data segment with 133 */

end;                                 /* this is needed by massl */
```

Figure 3: Sample Massl file, *sample.m*

```
# Makefile for sample.m
# Marco Fillo and Steve Keckler
# May 11, 1994

MASX= /home/mm/tools/bin/mas
LNKX = /home/mm/tools/bin/lnk
HDIR=/home/mm/apps/stdhead
SYSDIR=/home/mm/apps/sys/obj

# System object files
SYSOBJ=  $(SYSDIR)/systoff.o  $(SYSDIR)/init.o $(SYSDIR)/malloc.o \
  $(SYSDIR)/physical.o $(SYSDIR)/boot.o $(SYSDIR)/ltlb_event.o \
  $(SYSDIR)/sync_event.o $(SYSDIR)/gtlb_event.o \
  $(SYSDIR)/passfail.o


STDHDS=$(HDIR)/regdef.h $(HDIR)/std.h $(HDIR)/stack.h

sample.out:     $(SYSOBJ) sample.o $(STDHDS) Makefile
 $(LNKX) -o sample.out $(SYSOBJ) sample.o


sample.o: sample.m $(STDHDS)
 $(MASX) -I$(HDIR) -c -o sample.o sample.m

clean:
 /bin/rm *.o
```

Figure 4: Sample Makefile

and synchronization queues are not overwritten.

To create the executable, type the following:

```
csh> /usr/bin/make sample.out
```

Message handlers are not currently provided in the system code. This feature will be provided in subsequent releases of the boot code. If internode communication is desired, message handlers must be forked by the user program into the message handling H-Thread slots.

A generic exception handler is provided that signals errors but does not perform error recovery.

1.5 Running Msim

We now go through a sample execution of msim using the sample files given above. As a convention, the `typewrite` font is used for outputs from msim, the **boldface** font is used to indicate user input, and the *italic* font is used for extra explanations. The following shows how to assemble the program using `/home/mm/tools/bin/mas` and running it using `/home/mm/tools/bin/msim`.

`csh>` **/usr/bin/make sample.out**
`csh>` **msim sample.out sample.cmd**

```
M-Machine simulator:  msim v1.4

Creating 1 Map node.

[Cycle 0] 0
```

(msim prints out a few lines of "[cycle 0] 0" as it executes the instructions in sample.cmd.)

```
[Cycle 30]
[Node 0, Slot 0] Register c0.i5
being monitored.  [Current value:  uint:3735928559]
```

This is due to watch_reg in sample.cmd

```
[Cycle 250]
$35000 0:int:144
$35008 0:int:0
```

These are printed by the "dv" instruction in sample.cmd. The 144 was put there by init_memory.

(msim has finished executing sample.cmd. It has entered interactive mode now.)

```
[Cycle 250] help
Type ``help all'' to see the description of all commands
Type ``help <category>'' to list commands in a category
Type ``help <command>'' for information on given command
Categories:
    Builtins
    Select
    Network
    EventQueue
    Trace&Track
```

Summon "help".

```
[Cycle 250] msim>  run
```

Let it run till finished.

```
[Cycle 253]
[Node 0, Slot 0] Wrote c0.i8
        => int:0
[Cycle 275]
[Node 0, Slot 0] Wrote c0.i8
        => int:1
```

watch_reg tracks modification to the register being watched.

```
[Cycle 280]
[Node 0, Slot 0] Wrote c0.i8
        => int:2
[Cycle 285]
[Node 0, Slot 0] Wrote c0.i8
        => int:3
[Cycle 290]
[Node 0, Slot 0] Wrote c0.i8
        => int:4
```

133 / 27 = 4

```
[Cycle 328]
Program terminated and passed
```

msim prints this whenever it branches to the "pass" routine in boot.m

```
[Cycle 328] msim>   quit
Msim terminated at cycle 328
csh>
```

We're done.

2 Simulator Parameters

2.1 Simulator Defaults

The simulator allows certain internal parameters to be changed to allow flexibility in the where and how programs are run and to allow experimentation with the machine's architecture. Table 1 shows the parameters

Parameter	Default	Meaning
x_dim	1	number of nodes in X dimension
y_dim	1	number of nodes in Y dimension
z_dim	1	number of nodes in Z dimension
init_ip	ptr:P.0x15.0x0	location where code is loaded and started
init_cp	ptr:P.0x15.<data start>	pointer loaded into i2 of startup thread boot.m initializes i2 to 0x35000
log_page_size	9	physical page size = $2^{log\_page\_size}$
offchip_memory_size	0x50000	memory per node in bytes
log_num_banks	2	number of memory banks = $2^{log\_num\_banks}$
log_block_length	3	cache block length = $2^{log\_block\_length}$ words
log_line_length	3	cache line length = $2^{log\_line\_length}$ words
associativity	1	set associativity of cache bank
log_num_sets	5	number of cache lines per set
offchip_latency	2	SDRAM latency for first word
offchip_word_time	1	SDRAM latency per successive word
ltlb_associativity	2	set associativity of ltlb
log_ltlb_sets	6	number of ltlb entries per set
read-base	10	base for reading integers from command line
print-base	10	base for printing integers in the simulator

Table 1: Simulator Defaults

and their defaults.

These default may be changed by using the `setq` command in the simulator. However, in order for them to take affect, the simulated machine must be created using `make_map` after changing the parameters. The initial instruction pointer `init_ip` defaults to a physical pointer that does not require translation, even if the paging mechanism is enabled. For now `init_ip` should be left alone as the simulator will not handle translation for IPs and current coding conventions require that code reside aligned to the segment base. These conventions will change, so stay tuned.

The initial data pointer `init_cp` is set by the simulator to point to the beginning of the table of contents in the first data segment. This pointer is used by the system startup code to initialize stack space. The initial pointers provided to user code are described in *A Function Call Convention for M-Machine Assembler Programming*. The system may provide virtual pointers if the paging system is enabled, and it may be adjusted depending on the amount of space required for the program. If additional data space is required, `offchip_memory_size` can be increased.

2.2 Startup

The simulator is initialized with H-Thread 0 running in User V-Thread A (which is thread slot 0). All other V-Threads are inactive, and the other H-Threads in slot 0 are vacant. If multiple map nodes are created, the program is downloaded into the memory of all of the nodes and all of the nodes begin executing identically. In order to have different nodes execute different code, a test against the node identifier (nnr) can be made. The nnr may be accessed through configuration space as described in the architecture specification.

2.3 Data Types

The simulator uses four different data types:

Integer: Integers are 64 bits in the simulator, and may be read from the command line or data file. The base used to interpret the integer is determined by `read-base`.

Unsigned Integer: 64 bit unsigned integers. Integers read in the form 0x<number> are hexadecimal and interpreted as unsigned integers.

Floating Point: These are double precision floating point numbers and may be read and written with exponential notation (<mantissa>`e`<exp>) or with a decimal point (<whole>.<frac>).

Pointer: A pointer is a 64-bit quantity with 4 bits of permission state, 6 bits of length field, and 54 bits of address. These may be read and written in the following format: <perm>.<length>.<address>, where length and address are hexadecimal numerals. The permission fields are shown in the following Table 2.

Permission	Meaning
R	Read
RW	Read/Write
XU	Execute User
XP	Execute Privileged
XM	Execute Message
EU	Enter User
EP	Enter Privileged
C	Configuration Space
K	Key
P	Physical
ER	Error Value

Table 2: Permission fields

3 Simulator Inconsistencies

This M-Machine simulator (msim v1.4) is not completely consistent with the architecture described in instruction set manual and the architecture documents. This section lists those inconsistencies.

- The following operations are not yet implemented: `ult`, `ule`, `ugt`, `uge`, `hmul`, `send0pt`, `send1pt`, `lbsu`, `sbsu`.
- Network latency is not modeled.
- Configuration space is not completely implemented. Access to the following locations **is** available:
  - Reading/writing general/cc registers
  - Vacant flag,
  - Hthread IPs
  - Vthread init, thread CPs
  - Pending exception counters, Catastrophic exception handler IPs
  - Node ID, machine dimensions
  - Outgoing message buffer count, increment/decrement outgoing message buffer count.

Event	Event ID	Description
FAULT_SYNC_MISS	2	synchronization miss. These will be broken into more specific event types.
FAULT_BLOCK_STATUS	3	Memory block status miss. These will be broken into more specific event types.
FAULT_LTLB_MISS	6	LTLB miss.
FAULT_GTLB_MISS	7	GTLB miss.

Table 3: Implemented Events

The event subsystem implemented in msim v1.4 is not consistent with that documented in the M-Machine Architecture manual. Table 3 lists the events that are implemented in the simulator. The simulator will be changed to reflect the architecture specification. In general, *boot.m* provides the necessarily handling for the implemented events.

3.1 Caveats

The following are some caveats to bear in mind when using version v1.4 of msim.

- The OutgoingMessageBufferCounter is initialized to 128. It can be reset via the configuration space. Refer to the M-Machine Architecture document.

- Message handlers are not included in this release of *boot.m*. Messages can be received by forking message handlers into H-Threads 2 and 3 of V-Thread 4. Subsequent releases of *boot.m* will provide these facilities.

The exception subsystem is implemented as specified in the *M-Machine Architecture* specification, although not all exceptions are caught. The specified *errval* encoding is implemented, but underflow/overflow fault conditions are not detected by msim.

4 MSIM Commands

This section lists the commands available at the msim command line and describes their use. Note that whenever we refer to an *.out* file, we mean an assembled msim program file.

Builtins

`help` &opt <category or command>

Provide help on a category of commands or a command.

`load_config` <filename>

Load msim configuration file from <filename> into memory.

`include` <filename>

Includes the content of <filename> as command line input.

`load` <filename>

Load a program (an *.out* file) from <filename> into memory. This is not necessary if a *.out* file was specified when invoking *msim*.

`make_map` &opt <x-dim> <y-dim> <z-dim>

Explicitly create a new M-Machine in a 3D mesh with the given dimensions. This is not necessary if a *.out* file was specified when invoking *msim*.

`init_memory` <filename>

Write value to memory from file. Entries in file are either:

$$\text{ORG} < offset > < data >$$
$$\text{or}$$
$$< data >$$

`run`

Run system until <ctrl-C> or program termination.

`step` &opt <num_steps>

Step entire system <num_steps> times. Defaults to single stepping.

`quit`

Exit msim.

`setq` <sym> <value>

Assign the indicated symbol to the given value.

`status` <map_id>

Print status of MAP chip identified by <map_id>.

`slot_status` &opt <slot_id>

Print status of all slots on selected MAP chip. If <slot_id> is selected, then only that slot information is printed.

`print_config`

Print configuration of the map chip.

`print_slot` <slot_id> &opt <dont_print_regfile>

Print status of slot <slot_id> on selected MAP chip. If any non-null <dont_print_regfile> is specified, a shorter printout without the regfile s produced.

`print_cluster` <cluster_id> &opt <slot_id>

Print status of cluster <cluster_id> on selected MAP chip. If <slot_id> is included, only that slot's information will be printed.

`print_unit` <cluster_id> <unit_id> &opt <slot_id>

Print status of unit <unit_id> in cluster <cluster_id> on selected MAP chip. If <slot_id> is included, only that slot's information will be printed.

`print_reg` <slot_id> <cluster_id> <unit_id> <reg_id>

Print contents of the specified register.

`dv` <from> &opt <to>

Display a range of memory verbosely. <to> defaults to <from> + 16.

`sm` <address> <value>

Set memory: set memory contents of <address> to <value>.

`dcache`

Display the entire contents of the cache.

`d-icache` <cluster> <unit>

Display the entire contents of the icache.

`gtlb` &opt <map_id>

Display the contents of the GTLB.

`dops` <from> &opt <to>

Unscramble memory and print operations. <from> and <to> must be aligned on 32 word boundaries.

Select

`select_map` <map_id>

Select MAP chip identified by <map_id>.

`select_slot` <slot_id>

Select slot <map_id> from selected MAP chip.

`select_unit` <cluster_id> <unit_id>

Select unit identified by <cluster_id> <unit_id> from selected MAP chip.

Network

`netitfstat` <node_id> &opt <mesg_prio>

Network interface status. If mesg_prio is specified, the contents of the specified message buffers will be displayed.

`netstat` &opt <node_id>

Print out message statistics. (not yet properly implemented).

EventQueue

`eventq` <node_id>

Print out event queue status.

Trace&Track

`track` <activity> &opt <map_id>

Reports every occurrence of <activity> during execution. <activity> should be a quoted string, and can be one of the following:

- "send ok": SEND operations.
- "message arrival": Message arrival.
- "gtlb miss": Gtlb miss.
- "dcache miss": D-Cache miss.
- "icache miss": D-Cache miss.
- "sync fault": Memory synchronization fault.
- "branch": branches taken.
- "hfork": hfork executed.
- "vacquire": vacquire execute.
- "send stall": SEND requests that are stalled. (not yet implemented).
- "message consumed": Message words being consumed by message handlers (not yet implemented).
- "config space": Accesses to the configuration space (not yet implemented).
- "events": Events. (not fully implemented).
- "exceptions": Exceptions (not yet implemented).
- "halt": Monitors changes to the haltbits. (not yet implemented).

`notrack` <activity> &opt <map_id>

Cancels tracking on <activity>. <activity> could be "all", in addition to those listed above.

`track_switches` &opt <map_id>

Reports which activities are being tracked.

`watch_reg` <map_id> <slot_id> <cluster_id> <unit_id> <reg_id>

Watch any modification to the specified register. Use unit_id = 0 for INT regfile, 2 for FLT regfile.

`nowatch_reg` <map_id> <slot_id> <cluster_id> <unit_id> <reg_id>

Cancel watch on the specified register.

`watch_bbreg` <map_id> <slot_id> <cluster_id> <reg_id>

Watch any modification to the specified branch bank (CC) register. (Not fully implemented).

nowatch_bbreg <map_id> <slot_id> <cluster_id> <reg_id>

Cancel watch on the specified branch bank (CC) register. (Not fully implemented).

watch_mem <map_id> <from> &opt <to>

Report whenever the specified locations are written to. The range defaults to one word.

nowatch_mem <map_id> <from> &opt <to>

Cancels monitoring on the specified memory locations. The range defaults to one word.