The MIDAS (Microprocessor Instruction and Data Abstraction System) Virtual Computing Machine

Stephen Wright
Department of Computer Science, Bristol University

April 30th 2008
<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/2/2006</td>
<td>1</td>
<td>Original release</td>
</tr>
<tr>
<td>14/11/2006</td>
<td>3</td>
<td>Type qualifier added to Nop. Hlt instruction added. Jmp instruction added. Opcode values re-assigned Psh and Pop conditionality removed.</td>
</tr>
<tr>
<td>29/1/2007</td>
<td>4</td>
<td>Correction to Halt heading. Correction to Cmp description.</td>
</tr>
<tr>
<td>4/2/2008</td>
<td>5</td>
<td>Change to 8-bit format.</td>
</tr>
</tbody>
</table>
Table of Contents

1 Introduction ............................................................................................................... 4
2 Machine Architecture ............................................................................................... 5
   2.1 Protected Memory Spaces ............................................................................. 5
   2.2 Little Endian Data ....................................................................................... 5
   2.3 8-bit Memory Granularity .......................................................................... 5
   2.4 The Operand Stack ....................................................................................... 5
   2.5 Operand Stack Element Structure ............................................................... 5
   2.6 Non-Zero Address Base ............................................................................. 5
   2.7 Condition Flag Decisions ........................................................................... 5
3 Instruction Set Definition ......................................................................................... 6
   3.1 Instruction Binary Format .......................................................................... 6
      3.1.1 Instruction Format .............................................................................. 6
      3.1.2 Argument Format ............................................................................... 6
   3.2 Instruction Types ......................................................................................... 6
      3.2.1 No Operation ...................................................................................... 6
      3.2.2 Stack Push .......................................................................................... 7
      3.2.3 Stack Pop ........................................................................................... 7
      3.2.4 Single Operand Operation .................................................................. 7
      3.2.5 Dual Operand Operation .................................................................... 7
      3.2.6 Comparison Operation ....................................................................... 8
      3.2.7 Jump Operation .................................................................................. 9
      3.2.8 Halt Operation .................................................................................... 9
   3.3 Addressing Modes ....................................................................................... 9
      3.3.1 Program Counter Mode ...................................................................... 9
      3.3.2 Immediate Mode ................................................................................. 9
      3.3.3 Direct Mode ....................................................................................... 9
      3.3.4 Indirect Mode ...................................................................................10
   3.4 Jmp Conditionals ..................................................................................... 10
4 Binary Data Format .................................................................................................. 11
5 Example Assembler Program ................................................................................. 12
6 Current Implementations ......................................................................................... 14
   6.1 Virtual Machine .......................................................................................... 14
   6.2 Support Server ............................................................................................ 14
   6.3 Support Tools ............................................................................................. 15
1 Introduction

MIDAS (Microprocessor Instruction and Data Abstraction System) is an implementation of a virtual computing machine, designed specifically for use in real-time, safety-critical control applications.

The machine’s instruction set is particularly designed to allow proof of any software implementation using formal methods, and allow the inclusion of checking mechanisms to detect data corruption at run time.
2 Machine Architecture

2.1 Protected Memory Spaces
Binary instructions are located in a read-only region of the memory space (a “Modified Harvard” architecture).

[Rationale: prevention of executable corruption by bad data accesses.]

2.2 Little Endian Data
All data and instructions are stored in little endian format.

[Rationale: simplification of integer/short/char conversion.]

2.3 8-bit Memory Granularity
Executable and data memory is conventionally organised as individually addressable 8-bit units.

[Rationale: support for existing toolchains.]

2.4 TheOperand Stack
The machine has no registers: all data manipulation is performed using a 32-bit wide “operand stack”, located outside the executable and data memory spaces.

The operand stack should not be confused with the system stack implemented by most programming languages in the machine’s data memory space.

[Rationale: to avoid non-linear performance due to out-of-register spills.]

2.5 Operand Stack Element Structure
The operand stack is composed of a stack of 32-bit data elements, fetches of 8 and 16 bit data from memory result in data being automatically signed-extended to 32 bit signed form.

[Rationale: to allow support for future large data types.]

2.6 Non-Zero Address Base
The memory space begins from address 0x00000001. An attempt to access memory outside the valid address ranges results in execution being halted with an exception.

[Rationale: to detect errors on accesses via NULL-pointers.]

2.7 Condition Flag Decisions
The machine contains a boolean condition flag, which is set or cleared by comparison instructions. The flag controls execution of the conditional jump instruction.

[Rationale: provision of conditional execution branching.]
3 Instruction Set Definition

3.1 Instruction Binary Format
All MIDAS opcodes have a basic 8-bit instruction field, followed by an optional 32-bit argument field:

\[ iixx <aaaaaaaa> \]

Note that the MIDAS assembler language reflects the binary format, with each nibble of the instruction being represented by a pneumonic, separated by a “.”.

3.1.1 Instruction Format
Bits 7-4: Operation type code (see 3.2 Instruction ).
Bit 3-0: Operation-specific field.

3.1.2 Argument Format
Bits 31-0: 32-bit address or data element, depending on the operation (see 3.2 Instruction Types) and addressing mode (see 3.3 Addressing Modes).

Example:
psh.im4 0x12345678 = 23 12345678 (hexadecimal)

As the machine is little endian, the complete opcode appears in memory as follows:

\[
\begin{array}{c}
00000001: 23 \\
00000002: 78 \\
00000003: 56 \\
00000004: 34 \\
00000005: 12 \\
\end{array}
\]

3.2 Instruction Types
The defined operation types are summarised in the following table.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Assembler Pneumonic</th>
<th>Instruction Code</th>
<th>Instruction Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Operation</td>
<td>nop</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Push</td>
<td>psh</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Pop</td>
<td>pop</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Single-op</td>
<td>sop</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Dual-op</td>
<td>dop</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Compare</td>
<td>cmp</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Jump</td>
<td>jmp</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Halt</td>
<td>hlt</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

3.2.1 No Operation
nop
This operation has no effect apart from advancing the program counter.

### 3.2.2 Stack Push

\texttt{psh.<addressing-mode> <argument>}

This operation copies a data element onto the top of the operand stack and increments the stack size counter by 1.

The addressing-mode field is described in 3.3 Addressing Modes.

The Push instruction supports the PC, Immediate, Direct and Indirect addressing modes.

### 3.2.3 Stack Pop

\texttt{pop.<addressing-mode> <argument>}

This operation copies a data element from the top of the operand stack and decrements the stack size counter by 1.

The addressing-mode field is described in 3.3 Addressing Modes.

The Pop instruction only supports the Direct and Indirect addressing modes.

### 3.2.4 Single Operand Operation

\texttt{sop.<operation>}

The operation removes the top entry of the operand stack, performs a transformation on it, and pushes the result to the top of the stack. The size of the stack is therefore unaffected.

The available operations are summarised in the following table.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>C equivalent</th>
<th>Assembler Pneumonic</th>
<th>Instruction Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cast to float</td>
<td>Cast signed integer to float</td>
<td>(float)(x)</td>
<td>i2f</td>
<td>1</td>
</tr>
<tr>
<td>Cast to integer</td>
<td>Cast float to signed integer</td>
<td>(int)(x)</td>
<td>f2i</td>
<td>2</td>
</tr>
</tbody>
</table>

### 3.2.5 Dual Operand Operation

\texttt{sop.<operation>}

This operation removes the top 2 entries of the operand stack, performs a combinational operation on them, and pushes the result to the top of the stack. The size of the stack is therefore reduced by 1. The entry directly beneath the top of operand stack contains the first operand and the top entry contains the second operand.

The available combinational operations are summarised in the following table.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>C equivalent</th>
<th>Assembler</th>
<th>Instruction Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>32-bit integer addition</td>
<td>x + y</td>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>Subtract</td>
<td>32-bit integer subtraction</td>
<td>x - y</td>
<td>sub</td>
<td>2</td>
</tr>
<tr>
<td>Multiple</td>
<td>32-bit integer multiplication</td>
<td>x * y</td>
<td>mul</td>
<td>3</td>
</tr>
<tr>
<td>Divide</td>
<td>32-bit integer division</td>
<td>x / y</td>
<td>div</td>
<td>4</td>
</tr>
<tr>
<td>Modulo Divide</td>
<td>32-bit integer modulo division</td>
<td>x % y</td>
<td>mod</td>
<td>5</td>
</tr>
<tr>
<td>AND</td>
<td>Bit-wise logical AND</td>
<td>x &amp; y</td>
<td>and</td>
<td>6</td>
</tr>
<tr>
<td>OR</td>
<td>Bit-wise logical OR</td>
<td>x</td>
<td>y</td>
<td>oor</td>
</tr>
<tr>
<td>XOR</td>
<td>Bit-wise logical exclusive OR</td>
<td>x ^ y</td>
<td>xor</td>
<td>8</td>
</tr>
<tr>
<td>Shift Left</td>
<td>Bit shift left</td>
<td>x &lt;&lt; y</td>
<td>shl</td>
<td>9</td>
</tr>
<tr>
<td>Shift Right</td>
<td>Bit shift right</td>
<td>x &gt;&gt; y</td>
<td>shr</td>
<td>10</td>
</tr>
<tr>
<td>Add</td>
<td>32-bit floating point addition</td>
<td>x + y</td>
<td>fadd</td>
<td>11</td>
</tr>
<tr>
<td>Subtract</td>
<td>32-bit floating point subtraction</td>
<td>x - y</td>
<td>fsub</td>
<td>12</td>
</tr>
<tr>
<td>Multiple</td>
<td>32-bit floating point multiplication</td>
<td>x * y</td>
<td>fmul</td>
<td>13</td>
</tr>
<tr>
<td>Divide</td>
<td>32-bit floating point division</td>
<td>x / y</td>
<td>fdiv</td>
<td>14</td>
</tr>
</tbody>
</table>

Example:

psh.im1 2 # Load 2
psh.im1 3 # Load 3
dop.sub    # Perform 3 – 2 = 1

3.2.6 Comparison Operation

cmp.<comparision>

This operation compares the top 2 entries of the operand stack and sets the machine’s conditional flag appropriately. The data on the operand is assumed to be 32-bit signed integer data. The size of the stack is therefore reduced by 2. The entry directly beneath the top of operand stack contains the first operand and the top entry contains the second operand.

The available comparison operations are summarised in the following table.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>C equivalent</th>
<th>Assembler</th>
<th>Instruction Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>Test stack entry 0 equals entry 1</td>
<td>==</td>
<td>eq</td>
<td>1</td>
</tr>
<tr>
<td>Not equal</td>
<td>Test stack entry 0 does not equal entry 1</td>
<td>!=</td>
<td>ne</td>
<td>2</td>
</tr>
<tr>
<td>Greater than</td>
<td>Test stack entry 0 greater than entry 1</td>
<td>&gt;</td>
<td>gt</td>
<td>3</td>
</tr>
<tr>
<td>Less than</td>
<td>Test stack entry 0 less than entry 1</td>
<td>&lt;</td>
<td>lt</td>
<td>4</td>
</tr>
<tr>
<td>Greater or equal</td>
<td>Test stack entry 0 greater than or equals entry 1</td>
<td>&gt;=</td>
<td>ge</td>
<td>5</td>
</tr>
<tr>
<td>Less or equal</td>
<td>Test stack entry 0 less than or equals entry 1</td>
<td>&lt;=</td>
<td>le</td>
<td>6</td>
</tr>
</tbody>
</table>

Example:
psh.im1 1 # Load 1  
psh.im1 2 # Load 2  
cmp.gt  # Perform 2 > 1, sets condition flag true

3.2.7 Jump Operation
jmp.<condition-decision>

If the condition-decision is met, the operation copies the top element of the operand stack to the program counter, and decrements the stack size counter by 1. If the condition-decision is not met, the stack is unaffected and the program counter is incremented by the instruction size.

The condition-decision field is described in 3.4 Jmp Conditionals.

3.2.8 Halt Operation
hlt

This operation causes a shutdown of the virtual machine without an exception being raised.

3.3 Addressing Modes
The “push” and “pop” instructions can access data from the machine via a number of addressing modes. Each addressing mode exists in 3 forms: 8, 16 and 32 bit data sizes.

3.3.1 Program Counter Mode
The current value of the program counter is copied to the top of the stack. The 32-bit instruction argument is ignored.

Example:
   psh.pc 0x00000000 # Load PC onto the stack

3.3.2 Immediate Mode
A data element given as the instruction argument is transferred to the top of the stack. For 8 and 16 bit immediate mode, a 32-bit argument is still used, but the redundant bits are ignored.

Example:
   psh.im4 0x00000000 # Load zero onto the stack

3.3.3 Direct Mode
A data element is transferred to/from the top of the stack from a location in the data memory. The instruction argument states an absolute address in the data memory. If the supplied address is outside the machine memory range an exception is raised.

Example:
   psh.di .L3 # Fetch data at label .L3 from data memory
3.3.4 Indirect Mode
A data element is transferred to/from the top of the stack from a location in the data memory. The instruction argument states an absolute address in the data memory containing an entry containing the address of the data entry to be transferred. If the either address is outside the machine memory range an exception is raised.

Example:
psh.im4 .DataPtr # Load location PtrStore with address DataPtr
pop.di .PtrStore # "
psh.in .PtrStore # Fetch data at label .DataPtr via .PtrStore

The addressing modes are summarised in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Target Data</th>
<th>Assembler Pneumonic</th>
<th>Instruction Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>VM PC</td>
<td>pc</td>
<td>1</td>
</tr>
<tr>
<td>Immediate 8-bit</td>
<td>Instruction argument</td>
<td>im1</td>
<td>2</td>
</tr>
<tr>
<td>Immediate 16-bit</td>
<td>Instruction argument</td>
<td>im2</td>
<td>3</td>
</tr>
<tr>
<td>Immediate 32-bit</td>
<td>Instruction argument</td>
<td>im4</td>
<td>4</td>
</tr>
<tr>
<td>Direct 8-bit</td>
<td>Data memory pointed by argument</td>
<td>di1</td>
<td>5</td>
</tr>
<tr>
<td>Direct 16-bit</td>
<td>Data memory pointed by argument</td>
<td>di2</td>
<td>6</td>
</tr>
<tr>
<td>Direct 32-bit</td>
<td>Data memory pointed by argument</td>
<td>di4</td>
<td>7</td>
</tr>
<tr>
<td>Indirect 8-bit</td>
<td>Data memory pointed by pointer pointed by argument</td>
<td>in1</td>
<td>8</td>
</tr>
<tr>
<td>Indirect 16-bit</td>
<td>Data memory pointed by pointer pointed by argument</td>
<td>in2</td>
<td>9</td>
</tr>
<tr>
<td>Indirect 32-bit</td>
<td>Data memory pointed by pointer pointed by argument</td>
<td>in4</td>
<td>10</td>
</tr>
</tbody>
</table>

3.4 Jmp Conditionals
The “jmp” instruction is controlled by a conditional execution mechanism based on the state of the machine’s Condition Flag. The Condition Flag is set and cleared by Comparison Operations (see 3.2.6 Comparison Operation).

The available condition decisions are summarised in the following table.

<table>
<thead>
<tr>
<th>Decision</th>
<th>Logic</th>
<th>Assembler Pneumonic</th>
<th>Instruction Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>Jump if condition flag set true</td>
<td>co</td>
<td>1</td>
</tr>
<tr>
<td>Always</td>
<td>Always jump regardless of condition flag</td>
<td>al</td>
<td>2</td>
</tr>
</tbody>
</table>
4 Binary Data Format

Data is held on the stack as an uncommitted 32-bit bit field, which may be interpreted as a signed integer (for integer arithmetic functions and jump instructions), IEEE floating-point (for floating point arithmetic functions) or bit-field (for bitwise combination instructions). Interpretation of bit-fields for a given manipulating instruction is given in the following table.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>psh.*</td>
<td>Bit-field</td>
<td>N/A</td>
<td>Bit-field</td>
</tr>
<tr>
<td>pop.*</td>
<td>Bit-field</td>
<td>N/A</td>
<td>Bit-field</td>
</tr>
<tr>
<td>sop.i2f</td>
<td>Signed integer</td>
<td>N/A</td>
<td>Floating point</td>
</tr>
<tr>
<td>sop.f2i</td>
<td>Floating point</td>
<td>N/A</td>
<td>Signed integer</td>
</tr>
<tr>
<td>dop.add</td>
<td>Signed integer</td>
<td>Signed integer</td>
<td>Signed integer</td>
</tr>
<tr>
<td>dop.sub</td>
<td>Signed integer</td>
<td>Signed integer</td>
<td>Signed integer</td>
</tr>
<tr>
<td>dop.mul</td>
<td>Signed integer</td>
<td>Signed integer</td>
<td>Signed integer</td>
</tr>
<tr>
<td>dop.div</td>
<td>Signed integer</td>
<td>Signed integer</td>
<td>Signed integer</td>
</tr>
<tr>
<td>dop.and</td>
<td>Bit-field</td>
<td>Bit-field</td>
<td>Bit-field</td>
</tr>
<tr>
<td>dop.oor</td>
<td>Bit-field</td>
<td>Bit-field</td>
<td>Bit-field</td>
</tr>
<tr>
<td>dop.xor</td>
<td>Bit-field</td>
<td>Bit-field</td>
<td>Bit-field</td>
</tr>
<tr>
<td>dop.shl</td>
<td>Bit-field</td>
<td>Signed integer</td>
<td>Bit-field</td>
</tr>
<tr>
<td>dop.shr</td>
<td>Bit-field</td>
<td>Signed integer</td>
<td>Bit-field</td>
</tr>
<tr>
<td>dop.fadd</td>
<td>Floating point</td>
<td>Floating point</td>
<td>Floating point</td>
</tr>
<tr>
<td>dop.fsub</td>
<td>Floating point</td>
<td>Floating point</td>
<td>Floating point</td>
</tr>
<tr>
<td>dop.fmul</td>
<td>Floating point</td>
<td>Floating point</td>
<td>Floating point</td>
</tr>
<tr>
<td>dop.fdiv</td>
<td>Floating point</td>
<td>Floating point</td>
<td>Floating point</td>
</tr>
<tr>
<td>cmp.*</td>
<td>Signed integer</td>
<td>Signed integer</td>
<td>Signed integer</td>
</tr>
<tr>
<td>jmp.*</td>
<td>Signed integer</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>hlt</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
5 Example Assembler Program

# Midas test

.section .text
.global start
.align 1

start:

# No hardware setup needed
nop
nop
nop
nop
nop

# Initialise increment to create "int incr = 1;"
psh.im4 1
pop.di4 increment

# Initialise count to create "int count = 0;"
psh.im4 0
pop.di4 count

# Initialise endval to create "int endval = 3;"
psh.im4 3
pop.di4 endval

# Dummies to satisfy in-loop clear-down
psh.im4 0
psh.im4 0

loop:

# Clear result of comparision at end of loop
pop.di4 void
pop.di4 void

# Add count + increment
psh.di4 increment
psh.di4 count
dop.add

# Pop result back into count
pop.di4 count

# Pop operands to void
pop.di4 void
pop.di4 void
# Do compare of count against end
psh.di4 endval
psh.di4 count

# Check for end
cmp.lt

# Loop if condition met
psh.im4 loop
jmp.co

# Exited loop - clear operands and loop vector anyway
pop.di4 void
pop.di4 void
pop.di4 void

# Marker
nop

# Terminate virtual machine gracefully with HLT instruction
HLT

.section .rodata
.align 2
count:
   .long 0x00000000
increment:
   .long 0x00000001
endval:
   .long 0x00000010

# Define FP 10.0
.type endval_f, @object
.size endval_f, 4
endval_f:
   .word 1092616192

hello:
   .string "Hello world"

[section .data
.align 2
var1:
   .long 0x00000000
void:
   .long 0x00000000
6 Current Implementations

6.1 Virtual Machine
2 implementations of the MIDAS VM are currently available:

- A hand-coded C prototype, compiled to PC/Windows using Microsoft Visual C++ 5.0, compiled to PC/Linux using Eclipse CDT 3.1.2.

- An EventB implementation, comprising a formal specification and refinement to implementation, using the open-source RODIN tool (http://sourceforge.net/projects/rodin-b-sharp).

6.2 Support Server
The support server provides the following services:

- Executable image downloads into executable memory.
- Text output via a virtual console.

The server uses Berkley sockets to transport a simple packet protocol. The server is implemented using conventional C programming techniques. In the case of the C prototype, client communication functions are implemented in C: in the case of the B-Method implementation the same C functions are used but are packaged into a B-Method Library machine.

Figure 1: Support Server Architecture
The server has been compiled to PC/Windows using Microsoft Visual C++ 5.0, and compiled to PC/Linux using Eclipse CDT 3.1.2.

6.3 Support Tools
A C-Compiler/Assembler/Linker tool-chain has been implemented using the Gnu Compiler Collection (GCC).

The tool-chain has been compiled to PC/Windows and PC/Linux using the native GCC tool-chain.