

Reference Manual for the LatticeMico32 soft CPU Instruction Set Simulator



Simon Southwell

August 2016

Copyright

Copyright © 2016 -2017 Simon Southwell (Wyvern Semiconductors)

This document may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from the copyright holder.

Disclaimers

No warranties: the information provided in this document is “as is” without any express or implied warranty of any kind including warranties of accuracy, completeness, merchantability, noninfringement of intellectual property, or fitness for any particular purpose. In no event will the author be liable for any damages whatsoever (whether direct, indirect, special, incidental, or consequential, including, without limitation, damages for loss of profits, business interruption, or loss of information) arising out of the use of or inability to use the information provided in this document, even if the author has been advised of the possibility of such damages.



Introduction

The intent of presenting this model is mostly informative and (hopefully) it tries to strip down fundamental processor system concepts into simple, easily digestible pieces, whilst still ending up with a sufficiently complex system that has all the main concepts contained within it, such that an OS can be booted (μ Clinux), and the model can interface to third party IDEs and debugging systems (such as Eclipse, via GDB). More complex concepts, such as caching, pipelining and memory protection and management (for multi-threading etc.), are skipped for clarity, as these might be considered as optimisations that have been added to processor systems for reasons of speed and ease of use since microprocessors first appeared (which did not have these), and are not the foundational constructs of a processor based system. The more advanced topics will have to wait for another article.

The LatticeMico32 processor was chosen for this project as having a straightforward architecture that is more easily understood compared to, say, a 32 bit ARM processor, but is still a modern 32 bit embedded processor currently used and supported, with a toolchain that is relevant to other current systems developments, and is thus transferable knowledge. The LatticeMico32 already has an ISS built-in to the toolset supplied by Lattice Semiconductors, and this model is not meant to replace it. However, by having source code freely available, with clarity of understanding at its heart, the model is readily modified for non-standard systems, or extended for additional peripherals, or even simply experimented with to aid understanding of the system as it stands.

The package includes the source code for the instruction set simulator, modelling the LatticeMico32 soft-CPU. The core model is coded as a single C++ class (`lm32_cpu`), which can be, and is meant to be, integrated into other system models. It implements all the non-optional features, and most of the optional features of that core. It has both C++ and C compatible APIs and is extensible to include additional modelled functionality. A test platform (`cpumico32`) is bundled in the package, as is a case study of an embedded Linux system (`lnxmico32`), based around the model. An interface for remote debugging via GDB (and, by extension, third party IDEs, such as Eclipse) is also supplied. The source is free-software, released under the terms of the GNU licence (see [LICENCE](#) included in the package).

Features

Included Features:

- All supported core instructions
- All h/w modelled for configurable instructions
 - Multiplier
 - Divider
 - Sign extender
 - Barrel shifter
- Configurable internal memory
- All h/w debug break- and watchpoints modelled
- Cycle count functionality
- Configurable 'hardware', as per the Mico32
- Run-time and static disassembly
- Data and Instruction caches for timing model
- Extensibility via callbacks
 - Intercept memory accesses
 - Regular callback with ability for external interrupt generation
 - JTAG register access callback
- Configurable execution break points

- On a given address
- After a single step, or clock tick
- After a fixed number of cycles
- On 'hardware' debug break point
- Access to internal Memory
- Access to internal state
- Compatible with GNU tool chain ([lm32-elf-xx](#))
- Both C++ and C linkage interfaces available
- Separate GDB remote debug interface code is included

Features Not Included:

- JTAG interface model (but callback interface included)
- User definable instructions

The code is a simple exercise in modelling a RISC based embedded CPU. It comes with absolutely no warranties for accuracy, or fitness for any given purpose, and is provided 'as-is'. Hopefully it is useful for someone, and feel free to extend and enhance the model, and maybe let me know how it's going.

Simon Southwell (simon@anita-simulator.org.uk)
Cambridge, August 2016

Source files

Listed and described here are those source files that make up the Mico32 ISS library (e.g. [libmico32.so](#)). These are the source files needed for integration into other C or C++ environments. The source files for the example executable and test bench program, [cpumico32](#), are not described in this document (i.e. [cpumico32.cpp](#), [cpumico32.c.c](#) and [lm32_get_config.cpp](#)). These are still freely available for use, under the terms of the GNU public license, but do not form part of the core functionality of the simulator, and are not documented.

The main header files comprise those listed below:

- [src/lm32_cpu.h](#)
- [src/lm32_cpu_hdr.h](#)
- [src/lm32_cpu_mico32.h](#)

For integrating the model with external programs only [lm32_cpu.h](#) needs be included in source code that references the API, but this header makes reference to [lm32_cpu_hdr.h](#), which will need to be in the include path when compiling. The [lm32_cpu_mico32.h](#) header is only used by the internal source files, and includes all the definitions and types needed by this code. The [lm32_cpu.h](#) header has the major class definition for the model ([lm32_cpu](#)), and the other header, [lm32_cpu_hdr.h](#), has all the definitions for need by external programs using the API.

The following listed files define the methods that belong to the class [lm32_cpu](#), and headers specific to those methods. The class methods are split over several files, but all belong to the single [lm32_cpu](#) class.

- [src/lm32_cpu.cpp](#)
- [src/lm32_cpu_inst.cpp](#)
- [src/lm32_cpu_elf.h](#)
- [src/lm32_cpu_elf.cpp](#)
- [src/lm32_cpu_disassembler.cpp](#)

The entry point methods and program flow methods are all defined in [lm32_cpu.cpp](#), whereas the instructions themselves have methods defined in [lm32_cpu_inst.cpp](#). There is almost a one-to-one mapping of LM32 instructions and the instruction methods, but a couple of methods double up for multiple instructions. The processing of the ELF program files are handled in methods defined in [lm32_cpu_elf.cpp](#), with its header file as [lm32_cpu_elf.h](#). Code disassembly is handled by methods in defined in [lm32_cpu_disassembler.cpp](#).

A cache is implemented, for timing modelling purposes, and is instantiated in the main class for both the data and instruction caches. It is defined in the following files:

- [src/lm32_cache.h](#)
- [src/lm32_cache.cpp](#)

A C linkage interface is provided, for those requiring to integrate the model into a C environment, and this is defined in the following files:

- [src/lm32_cpu_c.h](#)
- [src/lm32_cpu_c.cpp](#)

For external programs interfacing to the model over the C interface, the `lm32_cpu_c.h` header must be included in source code making reference to that API, in place of `lm32_cpu.h`. The `lm32_cpu_hdr.h` still needs to be in the include path, when using the C interface.

Building code

Included in the package is a makefile to build the code under Linux or Cygwin, and support is also provided for MSVC 2010. Under the UN*X systems, by default (i.e. simply typing 'make') it will build the following:

- `cpumico32`
- `libmico32.a`
- `libmico32.so`

The first is an executable (see [man/cpumico32.1](#)) for running simple programs, particularly the self-test programs provided in the package—see “Testing” section below. The next two are a static and dynamic library respectively, and are the libraries an external program can use to link with the model, choosing the appropriate one depending whether static or dynamic linking was most appropriate for the particular application. The API for the libraries, and its use, is described in the “API” section below.

The makefile also, by default, builds the code with debug information (with `g++` option `-g`) and as position-independent code (with option `-fPIC`—though this option is not needed in Cygwin). These are defined in the make variable ‘`COPTS`’, and can be overridden at the make command line. By default, the model is ‘big endian’, just like the Lattice processor. For variants that have been modified to be ‘little endian’, the model can be compiled with a `COPTS` value that includes the definition option “`-DLM32_LITTLE_ENDIAN`”.

No code coverage information is included by default, but the ‘`COVOPTS`’ make variable can be set at the command line to add, say, ‘`gcov`’ coverage information in the build (e.g. `COVOPTS="-coverage"`). If ‘`lcov`’ is available, the HTML output can be generated with ‘`make coverage`’, after the tests have been executed. The output is placed in a directory `cov_html/`.

Support for MSVC 2010 is provided, with a solution file (`.sln`) in the `msvc/` directory, along with the minimal set of project files to read in to the MSVC 2010 IDE, and compile and run the model, but if `MSBuild.exe` is in the `PATH` under Cygwin, then the `makefile` has support to build from the command line with ‘`make MSVC`’. The `MSBuild.exe` executable is part of `Microsoft.NET`, and thus can normally be found in a directory (for example, under Cygwin) such as:

```
<cdrive_path>/Windows/Microsoft.NET/Framework/v4.0.30319
```

The `<cdrive_path>` is the Cygwin path to the windows disk (most likely `/cygdrive/c`) and the final directory name will depend on the particular version of Microsoft NET installed. For 64 bit machines, a 64 bit version of the executable will be under `Framework64`.

By default, a make build for MSVC builds a ‘Release’ executable, which is placed in the same directory as for the other builds of the `makefile`. If a ‘Debug’ version is required, then the default can be overridden via the `MSVCCONF` make variable—i.e.:

```
make MSVCCONF="Debug" MSVC
```

Like the make for UN*X, the MSVC build produces a `cpumico32.exe` executable, but only a single library, `libmico32.dll`.

API

The API to the model is a C++ interface (though a C interface is provided—see “C Linkage Interface” section below), that consists of a single object (of class `lm32_cpu`, as defined in `lm32_cpu.h`) that has a set of methods for configuring the model, setting control of program flow, and running executable code. Definitions are provided in `lm32_cpu_hdr.h` needed to communicate with some of these methods, and set their parameters. This is all described in the sections to follow. In summary, the methods are:

```
lm32_cpu (int verbose,
          bool disable_reset_break,
          bool disable_lock_break,
          bool disable_hw_break,
          bool disable_int_break,
          bool disassemble_run,
          uint32_t num_mem_bytes,
          uint32_t mem_offset,
          int mem_wait_states,
          uint32_t entry_point_addr,
          uint32_t cfg_word,
          FILE* ofp,
          lm32_cache_config_t* p_dcache_cfg,
          lm32_cache_config_t* p_icache_cfg,
          uint32_t disassemble_start)

int lm32_run_program (const char* elf_fname,
                     int run_cycles,
                     int break_addr,
                     int exec_type,
                     bool load_code)

void lm32_register_int_callback (p_lm32_intcallback_t callback_func)
void lm32_register_ext_mem_callback (p_lm32_memcallback_t callback_func)
void lm32_register_jtag_mem_callback (p_lm32_jtagcallback_t callback_func)

void lm32_reset_cpu (void)

void lm32_set_verbosity_level (int level)
lm32_time_t lm32_get_current_time (void)
void lm32_set_configuration (uint32_t word)
uint32_t lm32_get_configuration (void)
lm32_time_t lm32_get_num_instructions (void)

uint32_t lm32_read_mem (uint32_t byte_addr, int type)
void lm32_write_mem (uint32_t byte_addr, uint32_t data, int type, bool dis_cyc_cnt)

void lm32_dump_registers (void)
lm32_state lm32_get_cpu_state (void)
void lm32_set_cpu_state (lm32_state new_state)
void lm32_set_hw_debug_reg (uint32_t address, int type)
void lm32_set_gp_reg (uint32_t index, uint32_t value)
```

Initialisation

The model object is created by instantiating a variable of type `lm32_cpu` class, or creating via 'new'. The constructor, `lm32_cpu()`, has a set of inputs for the initial configuration of the model.

verbose	
type	int
valid values	LM32_VERBOSITY_OFF, LM32_VERBOSITY_LVL_1

<i>default value</i>	LM32_VERBOSITY_OFF
<i>description</i>	Controls level of verbosity. Currently this is either on or off (i.e. only one level). When on, a disassembled output showing program flow is sent to the output stream (see 'ofp' description below).

disable_reset_break	
<i>type</i>	bool
<i>valid values</i>	true, false
<i>default value</i>	true
<i>description</i>	Controls whether the model will break and return on a reset exception. If, from a callback function, <code>lm32_reset_cpu()</code> is invoked, emulating a pin reset, a reset exception is flagged internally to the model, and the exception is handled if enabled. When this parameter is <code>false</code> , this will cause a break and return from <code>lm32_run_program()</code> with a value of <code>LM32_RESET_BREAK</code> . This break occurs whether or not the internal state of the model means the exception is handled (e.g. if the <code>IE</code> register is set to disable interrupts). This is useful if the calling program wants to reconfigure the model between resets. If the parameter is <code>true</code> , the model does not break on a reset.

disable_lock_break	
<i>type</i>	bool
<i>valid values</i>	true, false
<i>default value</i>	false
<i>description</i>	Controls whether the model will break and return on detection of a program 'lock' condition; i.e. an instruction with a 'jump to self' characteristic that would lock further program flow. This is useful when running a program with a definite termination point (<code>while(1);</code>). However, in an event driven environment, the main thread may have this construct, with the system simply responding to incoming events, and this feature would be disabled, with alternative breaking needed to return from the model.

disable_hw_break	
<i>type</i>	bool
<i>valid values</i>	true, false
<i>default value</i>	true
<i>description</i>	Control whether the model will break and return on detection of a hardware break or watch points (if configured). When enabled, after a hardware breakpoint or watchpoint is reached, the model will return control to the calling program. On re-entry to the model, the program flow will continue from exception point, including calling the exception vector code, as it would have without the external break.

disable_int_break	
<i>type</i>	bool
<i>valid values</i>	true, false
<i>default value</i>	true
<i>description</i>	Control whether the model will break and return on detection of an external

	interrupt event When enabled, after an interrupt is active, the model will return control to the calling program. On re-entry to the model, the program flow will continue from exception point, including calling the exception vector code, as it would have without the external break.
--	--

disassemble_run	
<i>type</i>	bool
<i>valid values</i>	true, false
<i>default value</i>	false
<i>description</i>	When set 'true', 'running' the program does not execute the program code, but simply runs through the code linearly generating disassembled output (as if verbose were set). This turns the model into straight forward disassembler.

disassemble_start	
<i>type</i>	uint32_t (#include <stdint>)
<i>valid values</i>	0x0 to 0xffffffffc
<i>default value</i>	0
<i>description</i>	When verbose mode set, specifies after which cycle the verbose output will start to be displayed.

num_mem_bytes	
<i>type</i>	uint32_t (#include <stdint>)
<i>valid values</i>	0x0 to 0xffffffffc
<i>default value</i>	65536
<i>description</i>	Define the number of bytes of internally modelled memory. This value is rounded up to a 4 byte boundary. The internal memory is contiguous and is read- and writeable. Internal memory does have to be specified, but this places a requirement on having a registered external memory callback to handle all accesses to memory (see "Callbacks" section below). Internal memory will be masked by a callback that intercepts addresses that overlap the internal memory space.

mem_offset	
<i>type</i>	uint32_t (#include <stdint>)
<i>valid values</i>	0x0 to 0xffffffffc
<i>default value</i>	0
<i>description</i>	Define the byte address offset for the internal memory, if used. This value is rounded down to the nearest 4 byte word boundary. Useful if code to be executed is located in a region away from address 0, but it is still desired to have it loaded into internal memory. Usually used with entry_point_addr argument (see below).

mem_wait_states	
<i>type</i>	int
<i>valid values</i>	0 to INT_MAX
<i>default value</i>	0
<i>description</i>	Defines the number of wait states to be associated with read or write accesses to internal memory. This value will be added to the cycle count for

	any access to the internal memory over and above any issue or stall due to the load or store instruction.
--	---

entry_point_addr	
type	uint32_t (#include <stdint>)
valid values	0x0 to 0xffffffffc
default value	0
description	Defines the address for the reset PC value. The model normally starts, or resets to address 0, which is where the default internal memory resides. If code is compiled for a different location, then the internal memory can be relocated with <code>mem_offset</code> , and the reset address specified with this argument, to point to the relocated internal memory (or a region handled by an external memory callback).

cfg_word	
type	uint32_t (#include <stdint>)
valid values	0x0 to 0xffffffff
default value	LM32_DEFAULT_CONFIG
description	<p>At construction, the value of the <code>CFG</code> register can be set with this value to enable or disable hardware features. The value is exactly compatible with the <code>CFG</code> register as defined in the Lattice Mico32 Processor Reference Manual [1], section 'Control and Status Register'. A mask is applied to the value set, so that features not supported by the model cannot be enabled—see “Introduction” section above for unsupported features.</p> <p>Some definitions are defined in <code>lm32_cpu_hdr.h</code> that can be ORed together into the <code>cfg_word</code>, to enable features:</p> <pre> LM32_MULT_ENABLE LM32_DIV_ENABLE LM32_SHIFT_ENABLE LM32_SEXT_ENABLE LM32_COUNT_ENABLE LM32_DCACHE_ENABLE LM32_ICACHE_ENABLE LM32_SWDEBUG_ENABLE LM32_HWDEBUG_ENABLE LM32_JTAG_ENABLE LM32_NUM_BP_[0-4] LM32_NUM_WP_[0-4] </pre> <p>To configure the number of external interrupts, a value (between 0 and 32) can be shifted and ORed into the <code>cfg_word</code>. E.g. <code>(32 << LM32_CFG_INT)</code>.</p>

ofp	
type	FILE* (#include <stdio>)
valid values	NULL, <valid file pointer>
default value	stdout
description	A file pointer to direct verbose and debug output data. By default this is to <code>stdout</code> , but a valid open writeable file pointer can be specified to direct the output.

p_dcache_cfg	
<i>type</i>	lm32_cache_config_t*
<i>valid values</i>	NULL, <valid lm32_cache_config_t pointer>
<i>default value</i>	NULL
<i>description</i>	<p>A pointer to a cache configuration structure. This structure contains values for the configurable parameters of the data cache. This structure is defined, in <code>lm32_cpu_hdr.h</code>, as follows:</p> <pre>typedef struct { uint32_t cache_base_addr; uint32_t cache_limit; int cache_num_sets; int cache_num_ways; int cache_bytes_per_line; } lm32_cache_config_t;</pre> <p>Valid values for these parameters are as per the LatticeMico32 Processor Reference Manual, Chapter 3, table 17. If the parameter is set to <code>NULL</code>, then the data cache will use default values (if a data cache is configured).</p>

p_icache_cfg	
<i>type</i>	lm32_cache_config_t*
<i>valid values</i>	NULL, <valid lm32_cache_config_t pointer>
<i>default value</i>	NULL
<i>description</i>	<p>A pointer to a cache configuration structure. This structure contains values for the configurable parameters of the instruction cache. See <code>p_dcache_cfg</code> for details</p>

Execution and breakpoints

Once a model object is created, a program can be run via the `lm32_run_program()` method. At its simplest, it is called with a program file name to load and execute, and run 'forever'. However, other features are controllable on calling to limit the amount of execution. The call to the method can also return due to other break events that were specified at initialisation (see "Initialisation" section above). A returned value indicates why the `lm32_run_program()` exited. The parameters to the methods are described below:

elf_name	
<i>type</i>	const char*
<i>valid values</i>	string pointer with valid ELF program name, "" (empty string)
<i>default value</i>	"test.elf"
<i>description</i>	<p>The name of the ELF program to load and execute. Can be an empty string if the call is not a load type (e.g. <code>LM32_SINGLE_STEP</code>—see <code>exec_type</code> below).</p>

run_cycle	
<i>type</i>	int
<i>valid values</i>	1 to INT_MAX, LM32_FOREVER or LM32_ONCE
<i>default value</i>	LM32_FOREVER
<i>description</i>	<p>Defines the run cycle count to reach before returning. Any positive value</p>

	between 1 and <code>INT_MAX</code> can be specified, or <code>LM32_FOREVER</code> to not break on a cycle count, or <code>LM32_ONCE</code> . Note that the timing model (see “Timing Model” section below) is such that it can only break on instruction boundaries. As some instructions take multiple cycles, the cycle count on returning may be greater than that specified, up to the amount that the last instruction took to execute. Note also that specifying a <code>run_cycle</code> of a value less than the model's current cycle count is equivalent to running with a value of <code>LM32_ONCE</code> (see below for <code>lm32_get_current_time()</code> method).
--	---

break_addr	
<i>type</i>	int
<i>valid values</i>	0 to 0xffffffffc, <code>LM32_NO_BREAK_ADDR</code>
<i>default value</i>	<code>LM32_NO_BREAK_ADDR</code>
<i>description</i>	Specifies a return point when the <code>PC</code> reaches a particular address, or if no break on address is required (i.e. <code>LM32_NO_BREAK_ADDR</code>). Note that the lower two bits of the specified address are ignored.

exec_type	
<i>type</i>	int
<i>valid values</i>	<code>LM32_RUN_FROM_RESET</code> , <code>LM32_RUN_CONTINUE</code> , <code>LM32_RUN_SINGLE_STEP</code> , <code>LM32_RUN_TICK</code>
<i>default value</i>	<code>LM32_RUN_FROM_RESET</code>
<i>description</i>	Specifies the action on calling the method externally. Normally a value of <code>LM32_RUN_FROM_RESET</code> is specified with an executable file given in ' <code>elf_name</code> ', and any break point settings. The program is loaded into memory, and the CPU reset, starting execution from address 0. For single stepping the program, <code>LM32_RUN_SINGLE_STEP</code> is used. In this case the model returns after executing only one instruction. The <code>run_cycle</code> , <code>break_addr</code> and <code>elf_name</code> parameters are ignored for this type. A similar type is <code>LM32_RUN_TICK</code> . This advances just a single clock, but doesn't necessarily execute an instruction; say if the last instruction takes multiple cycles. It advances time by one clock only, and executes an instruction only when the tick count and the instruction cycle counts agree. It is useful if integration with an environment that has a timing model that advances by single clock ticks. The <code>LM32_RUN_CONTINUE</code> is used to continue execution from the point at which the method last returned. The break parameters are active in this call type, but <code>elf_name</code> is ignored, and no program is loaded.

load_code	
<i>type</i>	bool
<i>valid values</i>	true, false
<i>default value</i>	false
<i>description</i>	Specifies to load the program specified by the <code>elf_name</code> argument into memory. The program will be load to memory, in all events, when <code>exec_type</code> is set to <code>LM32_RUN_FROM_RESET</code> , but this parameter can be used to re-load, or replace a program if the <code>lm32_run_program()</code> call returned on some break point.

Return Value

The `lm32_run_program()` method returns one of several values to indicate why it returned.

- `LM32_USER_BREAK`: returned having reached the user specified break address, set at configuration or passed as a parameter when calling `lm32_run_program()`.
- `LM32_SINGLE_STEP_BREAK`: returned whilst single stepping (i.e. `exec_type` argument was set as `LM32_RUN_SINGLE_STEP` when `lm32_run_program()` called
- `LM32_TICK_BREAK`: returned whilst executing with an `exec_type` of `LM32_RUN_TICK`
- `LM32_RESET_BREAK`: returned if the model was externally reset via `lm32_reset_cpu()` (and reset breaking was not disabled)
- `LM32_LOCK_BREAK`: Reached a program 'lock' condition (and lock breaking was not disabled)
- `LM32_DISASSEMBLE_BREAK`: reached the end of the program during disassemble mode (see “Disassembled Output” section below)
- `LM32_HW_BREAKPOINT_BREAK`[†]: a hardware breakpoint fired, when breaking on hardware debug events were enabled.
- `LM32_HW_WATCHPOINT_BREAK`[†]: a hardware watchpoint fired, when breaking on hardware debug events were enabled.
- `LM32_INT_BREAK`: an external interrupt was active, when breaking on interrupt events were enabled.
- `LM32_BUS_ERROR_BREAK`[†]: an instruction or data bus exception fired.
- `LM32_DIV_ZERO_BREAK`[†]: a divide-by-zero exception fired.

[†] Note: the method will return with these values regardless of whether interrupts are enabled in the IE register. This allows for non-intrusive debugging. The calling function must process the returned value and decide what the appropriate action is. The state of the IE register is available to the calling routine via the `lm32_get_cpu_state()` method, if required. As the external interrupts are level sensitive, this may cause the method to return for each executed instruction that the interrupt(s) are asserted.

Run-time configuration and status

Some methods are provided for inspecting status and setting configuration at run-time, i.e. after `lm32_run_program()` has returned.

<code>lm32_set_verbosity(int level)</code>	
<i>description</i>	Allows the verbosity level to be changed. Its single parameter has valid values as verbose of the <code>lm32_cpu</code> constructor. This is useful for debugging of long programs, which would generate a large output if verbosity specified from time 0. A break can be set up to return at a known point before the area of interest, and verbosity increased before continuing

<code>lm32_get_current_time()</code>	
<i>description</i>	Returns the current internal cycle count of the model. This counts upward from 0 for all execution, and is not the <code>CC</code> value which can be reset. This is useful if wanting a break point on a cycle count relative to current time, rather than an absolute value.

lm32_get_num_instructions()	
<i>description</i>	Returns the current count of instructions executed since time 0. Useful for statistical analysis and performance measurements.

lm32_get_configuration()	
<i>description</i>	Returns the value of the CFG register as a <code>uint32_t</code> . It allows the external inspection of 'implemented' hardware. Since any configuration value is masked (see <code>lm32_cpu()</code> description above, and <code>lm32_set_configuration()</code> description below), then this give a definitive value as being used by the model.

lm32_set_configuration(uint32_t word)	
<i>description</i>	Sets the value of the CFG register, masked to allow only supported features to be enabled. This ability to dynamically update the hardware configuration is not really supported in the Mico32, but it is useful for testing the model. The test suite (see “Testing” section below) run on <code>cpumico32</code> reconfigures the model via the memory callback function to test that removing the features disables them as far as the program is concerned, and the proper response is seen.

Reset event

lm32_reset_cpu()	
<i>description</i>	Method used to model asserting the reset pin of the Mico32. It generates and internal event and is processed accordingly, as for the real processor.

Callbacks

The ISS is a model of a processor core, and its main usage is as a component in a larger system level model. It has an internal memory model for convenience and to aid stand alone testing, but it is via the callbacks that the model can be extended or integrated into a system model of arbitrary complexity. The model supports three types of user defined callbacks that can be registered with the model. One is for calling at each memory access that the CPU performs, the second is called at regular intervals (from once each instruction boundary, to as long as specified in a wait or sleep period). The third is to allow a JTAG interface to be implemented as an add-on, and is invoked whenever the model accesses the `JTX` or `JRX` registers.

The main extension is to map peripherals (including more memory, if desired) into the memory space via the external memory callback function, trapping accesses to addresses with memory mapped peripheral registers and implementing the functionality.

To take a trivial example, to model a system that has a serial output then the external memory callback mechanism would seem to be what you need. It is there to allow modelling of hardware that is external to the `lm32` model itself. By registering a callback function with the method `lm32_register_ext_mem_callback()`, then all accesses to anywhere in the memory space by the model will call this registered function first, with the address and any write data value (assuming a write). Taking an example, if you modelled a serial port with an output register located at, say, `0x80000000` or wherever, then the callback function can test for this address, and print a character to the screen of the value of the lowest byte of the data word passed in, returning a wait-state count for the access (any value of 0 or more). For all other

addresses the callback function returns `LM32_EXT_MEM_NOT_PROCESSED` to allow the model to handle the access. This concept can be extended to model any hardware with a memory mapped register set, accessible by the processor. You can try this mechanism for yourself by registering your own external memory callback function and have it print out all of the memory read and write accesses it sees (returning `LM32_EXT_MEM_NOT_PROCESSED` so that the model can continue). You'll see it accessing all the program locations and any other locations it is trying to access.

For functionality that isn't a memory mapped register, but can generate an interrupt, or requires updating on CPU time, then the interrupt callback can be used. This can be called up to every instruction, or at longer intervals, and it can generate an interrupt upon returning (or not). Taking the serial output port as an example, if it is required that the peripheral generates an interrupt some cycles after a byte is transmitted, then, after a sufficient number of calls to the int callback, from the register write (via the external memory callback), the callback can return a value, indicating an interrupt on one of the interrupt pins.

Note that these two callbacks are shared amongst all modelled peripherals, and the callbacks will have to do initial decoding. For the external memory accesses, this is just decoding incoming addresses, perhaps doing a page decode to identify a particular peripheral, and then calling a model function which does the rest of the decode. For the interrupt callback, then it is envisaged that all peripheral models are called at every cycle, amalgamating any interrupts returned from the peripheral model functions.

A special callback is implemented to handle JTAG accesses, via the `JTX` and `JRX` registers. This is simply a hook for functionality that isn't implemented in the model, should anyone wish to add support for this.

The three callback registration functions are described below:

<code>lm32_register_int_callback(p_lm32_intcallback_t callback_func)</code>	
<i>description</i>	<p>The caller must provide as the parameter input, a pointer to a function of type <code>p_lm32_intcallback_t</code>, e.g.:</p> <pre>uint32_t cb_func(lm32_time_t time, lm32_time_t* wakeup_time);</pre> <p>If a function is registered, the model will call this function at least once at time 0. The user function receives the current time in the '<code>time</code>' parameter. Before returning the function must update the contents of the integer pointed to by '<code>wakeup_time</code>' to indicate the cycle it next wishes to be called. A value of 0 or less than or equal to the '<code>time</code>' parameter will mean it is called after the next instruction. If the callback function wishes to delay being called to a future time, then the <code>wakeup_time</code> is specified for some value greater than '<code>time</code>'. A negative value returned informs the model that a request to terminate is requested, and a user breakpoint is generated.</p> <p>The return value from the callback is the pattern of inputs to the external interrupt pins (up to 32). For configured input pins, the event is set in the <code>IP</code> register of the model, and will generate an interrupt if the model has these enabled (<code>IE</code> and <code>IM</code> settings). If the <code>CFG</code> register is not configured for 32 interrupts, then bits set for all non-configured inputs are ignored, and the <code>IP</code> register bit is not set.</p>

<code>lm32_register_ext_mem_callback(p_lm32_memcallback_t callback_func)</code>	
<i>description</i>	<p>The caller must provide, as the input parameter, a pointer to a function of type <code>p_lm32_memcallback_t</code> e.g.:</p>


```
int cb_func(uint32_t byte_addr, uint32_t *data, int type,
            int cache_hit, lm32_time_t time);
```

If a function is registered the model will call this for every memory access made (i.e. via load and store instructions). The address being called is passed in as 'byte_addr', and data is exchanged via the 32 bit word pointed to by 'data'. On write access types this contains the data to be written. It has meaning to update this on write accesses, as the model will ignore it, but it is safe to do so. On read access types, the data to be returned is placed into the integer pointed to by data. The type parameter takes on one of several values, to indicate the direction and size of access, as shown below.

```
LM32_MEM_WR_ACCESS_BYTE
LM32_MEM_WR_ACCESS_HWORD
LM32_MEM_WR_ACCESS_WORD
LM32_MEM_WR_ACCESS_INSTR
LM32_MEM_RD_ACCESS_BYTE
LM32_MEM_RD_ACCESS_HWORD
LM32_MEM_RD_ACCESS_WORD
LM32_MEM_RD_INSTR
```

The `cache_hit` parameter is a flag which indicates whether the memory access is a true access to memory (when zero), or whether the cache is fetching data on a cache hit (when non-zero). The cache model does not store actual data internally, but only keeps track of which addresses are cached, and still fetches data from memory. This flag allows the external callback functions to differentiate between access types if, say, it is keeping statistics on access rate, location etc.

The callback function can intercept any of the memory accesses to update its own internal state, and thus model memory mapped external blocks. If the callback processed the memory access it must return a cycle count from 0 to `INT_MAX` to indicate the number of wait states that the model must add to its internal cycle count. If the callback is modelling a single cycle access, then 0 would be returned (no wait-states). If the callback is modelling more complex peripherals (e.g. with resource sharing and arbitration) with wait states generated, it would return a positive integer to indicate the number of wait state cycles elapsed. If the supplied address did not access any portion of memory covered by the callback function, or the access was invalid for some reason (misaligned, say), the callback *must* return `LM32_EXT_MEM_NOT_PROCESSED`, to allow the model to attempt an access to its internal memory. Honouring this requirement is important for correct timing operation, and the timing modelling is only as good as the values returned by the callback.

Note that for cache hit accesses, (`cache_hit` non-zero) the returned number can be any value zero or greater (but not `LM32_EXT_MEM_NOT_PROCESSED`, if in a mapped region) as the timing model uses a number based on the timing for a cache access.

lm32_register_jtag_callback (p_lm32_jtagcallback_t callback_func)

<i>description</i>	The caller must provide, as the single input parameter, a pointer to a function of type <code>p_lm32_jtagcallback_t</code> , e.g.:
--------------------	--

	<pre>void cb_func(uint32_t *data, int type, lm32_time_t time);</pre> <p>If a callback function is registered, the model will execute this each time the model accesses the <code>JTX</code> or <code>JRX</code> registers (via the <code>rcsr</code> or <code>wcsr</code> instructions). One of three types is passed in:</p> <pre>LM32_JTX_WR LM32_JTX_RD LM32_JRX_RD</pre> <p>There is no <code>JRX</code> write type as this has no function, and the model ignores the instruction internally. When the type is a write, the pointer <code>*data</code> points to the data byte to be written. On read types, the returned data is placed in to the location pointed to by <code>*data</code>. For both <code>JTX</code> and <code>JRX</code>, bit 8 should contain the 'full' status of the TX or RX register.</p> <p>By default, the model has JTAG as an unimplemented featured. This can be enabled or disabled via <code>lm32_set_configuration()</code>, or set at the model object's construction, but a side-effect of registering a JTAG callback function is that the feature is enabled, and the bit set in the <code>CFG</code> register automatically.</p>
--	---

Internal memory access

The API provides direct access to the model's internal memory, via two methods. Using these methods will also invoke any external memory callback function, and so can be used to peek and poke memory areas implemented externally via the callback.

<code>lm32_read_mem(uint32_t byte_addr, int type)</code>	
<i>description</i>	Returns a 32 bit word with the value at address specified by ' <code>byte_addr</code> '. Valid types are <code>LM32_MEM_RD_xx</code> types as specified for the memory callback (see above). Any other type will cause a fatal error.

<code>lm32_write_mem(uint32_t byte_addr, uint32_t data, int type, bool dbl_cyc_cnt)</code>	
<i>description</i>	<p>Writes the data in '<code>data</code>' to the address specified in '<code>byte_addr</code>'. Valid types are <code>LM32_MEM_WR_xx</code> types as specified for the memory callback (see above). Any other type will cause a fatal error.</p> <p>By default, the cycle count is advanced whenever this function is called, depending on other settings. This can be disabled with the optional third argument for use when, say, loading binary program data using this function. This argument is not available in the C interface function.</p>

There are no safe guards on the calling of these memory access functions, and all internal memory is accessible from them. However, accessing invalid areas of memory will cause fatal exceptions. Use with caution.

Internal state access

Three methods are provided to give access to internal register state of the model, either to the output stream, or returned to the calling program.

lm32_dump_registers()	
<i>description</i>	<p>will print out the complete set of internal registers states to the output stream (as defined by the ofp parameter of the constructor), formatted as shown in the example below:</p> <pre> r00 = 0x00000000 r01 = 0x00003303 r02 = 0x00000000 r03 = 0x00000000 r04 = 0x00000000 r05 = 0x00000000 r06 = 0x00000000 r07 = 0x00000000 r08 = 0x00000000 r09 = 0x00000000 r10 = 0x00000000 r11 = 0x00000000 r12 = 0x00000003 r13 = 0x00000000 r14 = 0x00000000 r15 = 0x00000000 r16 = 0x00000000 r17 = 0x00000000 r18 = 0x00000000 r19 = 0x00000000 r20 = 0x00000000 r21 = 0x00000000 r22 = 0x00000000 r23 = 0x00000000 r24 = 0x0000900d r25 = 0x0000fffc gp = 0x00000000 fp = 0x00000000 sp = 0x0000fff0 ra = 0x00000000 ea = 0x00000000 ba = 0x00000578 pc = 0x000005ac ie = 0x00000005 ip = 0x00000000 im = 0x00000000 icc = 0x00000000 dcc = 0x00000000 cfg = 0x01120037 cfg2 = 0x00000000 cc = 0x000005bb eba = 0x00000000 bp0 = 0x00000241 bp1 = 0x00000251 bp2 = 0x00000261 bp3 = 0x00000271 wp0 = 0x00003000 wp1 = 0x00003101 wp2 = 0x00003202 wp3 = 0x00003303 dc = 0x000000fc deba = 0x00000000 </pre>

lm32_get_cpu_state()	
<i>description</i>	<p>Returns a class <code>lm32_cpu::lm32_state</code> containing a complete set of the register values, plus some other persistent internal state needed by the model. All register fields are of type <code>uint32_t</code>: one for each register. See source file lm32_cpu.h for the details of the class definition.</p>

lm32_set_cpu_state()	
<i>description</i>	<p>Sets the internal model state to that passed into the function, of type <code>lm32_cpu::lm32_state</code>. This contains all the CPU registers, plus some other internal state, needed by the model. This is intended for use in save and restore operations, rather than as a means to update the models internal state externally. See source file lm32_cpu.h for the details of the class definition.</p>

lm32_set_hw_debug_reg(uint32_t addr, int type)	
<i>description</i>	<p>Allows the external setting of the h/w debug registers. The address to be written to the register is passed in on the 'addr' parameter, and the register to access is defined via the 'type'. This can take on one of the following values.</p> <pre> LM32_CSR_ID_BP0 LM32_CSR_ID_BP1 LM32_CSR_ID_BP2 LM32_CSR_ID_BP3 LM32_CSR_ID_WP0 LM32_CSR_ID_WP1 LM32_CSR_ID_WP2 LM32_CSR_ID_WP3 </pre>

	<p>If the model is configured to have less than the full complement of break- or watch points, then attempting to set the registers via this method will have no effect. Note also, that the breakpoint 'addr' value must include the enable bit, exactly as defined for the BPx registers in the reference manual [1], but the watchpoint 'addr' is a pure 32 bit byte address so, in addition to the basic LM32_CSR_ID_WPx value for the 'type', to define which watchpoint is updated, one of four settings must be 'ORed' in with the value to make up the complete watchpoint access type:</p> <pre> LM32_WP_DISABLED LM32_WP_BREAK_ON_READ LM32_WP_BREAK_IN_WRITE LM32_WP_BREAK_ALWAYS </pre> <p>The method updates the relevant Cn fields of the DC register, based on these type values.</p>
--	---

lm32_set_gp_reg(uint32_t idx, uint32_t value)	
<i>description</i>	<p>Allows the external setting of the 32 general purpose registers. The GP register to be written to is passed in on the 'idx' parameter, and the value to set is passed in on the 'value' parameter.</p> <p>This function is meant for debug, and initialisation. Caution should be used if setting the registers externally, whilst code is executing.</p>

C Linkage Interface

A C linkage API is provided as an alternative to the C++ interface, for those who have a C environment that they wish to integrate the model into. The API is purposely as similar to the C++ API as possible, as it has been described above. There is a one-to-one correspondence to the C++ methods, with all the C API functions called **lm32c_<c++ equivalent suffix>**, and each has an additional parameter, except for the initialisation function, as explained below.

The constructor is replaced with a function **lm32c_cpu_init()**. The parameters are the same as for the C++ constructor, only with the Boolean types now defined to be of type '**int**'. Thus their default values are no longer '**true**' or '**false**', but the API defines values **TRUE** and **FALSE**, which replace these. The function returns a handle to a unique object, of type **lm32c_hdl**, which must be saved, as all the other API functions require it to access the initialised model's instantiation. This allows for multiple instantiations of the model.

All the other C++ methods have an **lm32c_xxxx** equivalent, with identical parameters, except that a new first parameter must be given, which is the handle returned by **lm32c_cpu_init()**. For example, if the handle has been saved in a variable **lm32Hdl** of type **lm32c_hdl**, then a reset call is now **lm32c_reset_cpu(lm32Hdl)**, in place of the C++ method **lm32_reset_cpu()**, described in the above sections. The **lm32c_get_cpu_state()** function returns a structure of type **lm32c_state**, rather than the class **lm32_state**, but the field names are identical. The full list of C linkage functions is thus:

- **lm32c_cpu_init()**
- **lm32c_run_program()**
- **lm32c_reset_cpu()**

- `lm32c_register_int_callback()`
- `lm32c_register_ext_mem_callback()`
- `lm32c_register_jtag_callback()`
- `lm32c_set_verbosity_level()`
- `lm32c_get_current_time()`
- `lm32c_set_configuration()`
- `lm32c_get_configuration()`
- `lm32c_get_num_instructions()`
- `lm32c_read_mem()`
- `lm32c_write_mem()`
- `lm32c_dump_registers()`
- `lm32c_set_hw_debug_reg()`
- `lm32c_get_cpu_state()`

The API is defined in the source file header `lm32_cpu_c.h`, which must be included in code using the C API.

Disassembled Output

The model can output (to 'ofp') fully disassembled output in one of two ways. Either the disassembled output can show program flow, during a normal execution of code on the model, or it can simply display a disassembled output of the specified ELF executable file.

Normal execution flow disassembly is instigated either by setting the constructor's 'verbose' parameter, or by calling `lm32_set_verbosity_level()`. When verbosity is enabled the output looks something like the example fragment shown below:

```
0x01ac: (0x2b9f0034)  lw      ba, (sp +00052)      @433
0x01b0: (0x379c0038)  addi    sp, sp, 000056      @434
0x01b4: (0xc3e00000)  b       ba                  @435
*
0x0304: (0x5e8c00a8)  bne     r20, r12, 0000672    @439
0x0308: (0x9a94a000)  xor     r20, r20, r20      @440
0x030c: (0x38013101)  ori     r1, r0, 0x3101      @441
0x0310: (0x30220000)  sb      (r1 +00000), r2     @442
0x0314: (0x5e8000a4)  bne     r20, r0, 0000656    @443
0x0318: (0x38010144)  ori     r1, r0, 0x0144      @444
0x031c: (0xd1010000)  wcsr    DC , r1            @445
0x0320: (0x9a94a000)  xor     r20, r20, r20      @446
0x0324: (0x38013101)  ori     r1, r0, 0x3101      @447
```

The first field displays address of the instruction (i.e. the value of the PC register). When the program flow is disrupted (due to a branch, call, or exception), this field shows '*', and the rest of the line is left blank, to ease finding the jumps when debugging. Field 2 gives the raw instruction value being executed followed by the actual disassembled instruction in field 3. The current cycle count is displayed in the last field. This mostly increases by one, but for instructions that take more cycles to execute, or are stalled etc., the value jumps by a larger amount. In the example above the branch instruction takes 4 cycles to issue, and so the cycle count jumps from 435 to 439.

The pure disassembled output is specified by setting the `disassemble_run` parameter of the constructor, and has nearly identical output to that of the run-time output.

```
0x02f8: (0x9a94a000)  xor     r20, r20, r20
0x02fc: (0x38013101)  ori     r1, r0, 0x3101
0x0300: (0x10220000)  lb      r2, (r1 +00000)
0x0304: (0x5e8c00a8)  bne     r20, r12, 0000672
0x0308: (0x9a94a000)  xor     r20, r20, r20
0x030c: (0x38013101)  ori     r1, r0, 0x3101
0x0310: (0x30220000)  sb      (r1 +00000), r2
0x0314: (0x5e8000a4)  bne     r20, r0, 0000656
0x0318: (0x38010144)  ori     r1, r0, 0x0144
0x031c: (0xd1010000)  wcsr    DC , r1
0x0320: (0x9a94a000)  xor     r20, r20, r20
0x0324: (0x38013101)  ori     r1, r0, 0x3101
```

The main difference here is that there is no cycle count (as this has no meaning in this context), and there will be no breaks in address as the disassembling runs from the lowest to the highest address (of text areas) linearly.

Timing Model

The ISS makes an approximation of time using the issue cycles and result cycles associated with each instruction, as defined in the LatticeMico32 reference manual [1]. Each instruction executed will advance the cycle count by at least its issue cycles, as the next instruction cannot be executed before this time. In addition, if it updates a register, then the result cycles value plus the current cycle count is stored for the target register. This is the earliest time that a future instruction can access this register. When an instruction is executed, its source registers (**RY** and, if applicable, **RZ**) have their availability times checked, and the cycle count is advanced to the time of the latest register's availability. This timing model does not take into account branch prediction, and uses the issue cycle numbers for 'taken' and 'not taken' unmodified, as defined in the reference manual [1]. The internal cycle count is also used as the basis for the **CC** register value. Since this register can be changed by software, but the cycle count needs to run continuously, the **CC** value is emulated by keeping a track of the offset from cycle count and the last programmed value, such that a read of the **CC** register will be correct, whilst still being based on the internal cycle count. This means only a single source is used for all timing.

The model can be advanced by single cycles, as well as single instructions, to allow the model to be called at a regular clock tick count. At each instruction the cycle count is advanced by one or more, depending on the instruction. At each clock 'tick', the clock time is advanced by one clock cycle. Only when the clock tick count matches the instruction cycle count is the next instruction executed. This is also useful when multiple instances of the model are instantiated, as they can be kept in synchronisation, by calling with a clock 'tick' rather than single-stepping, and their internal sense of time will advance at the same rate and remain locked, with just minor differences due to instruction execution granularity.

Caches

The model implements configurable caches for the data and instruction fetches. The cache models are for timing purposes only, and do not actually store data within them, but keep a record of which addresses are cached. If an access to a cached region of memory is made, whether to internal memory or to a region mapped by an external memory access callback function, the data is still accessed as normal, but the reported timing wait states are ignored if a cache hit, and single cycle accesses of the cache are used to update time instead. If a cache miss, however, the memory access wait states (if any) are scaled by the number of words accesses required to fill the cache line, as these would have been fetched by the cache. When no cache is configured, the wait states from memory callbacks or internal memory accesses are used unmodified to update time.

By default, the caches are disabled (i.e. the configuration is for no caches implemented). To enable caching, the configuration register (**CFG**) must be modified at instantiation or via the `lm32_set_configuration()` method (see API section above).

Source Code Architecture

It is not the intention to go into minute detail for the internal architecture of the model here, but a brief overview of the main program flow, internal state, and major structures is in order, to allow anyone wishing to understand or modify the code enough of a handle, that they can explore the details on their own.

Main execution flow

Below is shown some pseudo-code of the main program flow when executing a program. The main `lm32_cpu` class member functions are shown as "`funcname()`", and the phrases between "<" and ">" describe local functionality. The indentation of the pseudo-code shows the calling hierarchy as implemented in the code.

```
lm32_run_program()

    <if running from reset...>
        <load ELF program to memory>

    <while no break point reached...>

        execute_instruction()

            process_exception()
                <process external interrupts>
                <if master interrupts enabled...>
                    interrupt()
                        <if interrupt outstanding...>
                            <generate exception>

            <fetch opcode from PC location in memory>
            <lookup decode_table information using opcode>
            <extract argument fields from opcode>

            <if verbose or disassemble_run...>
                disassemble()

            <if not disassemble_run...>
                <lookup instruction function in tbl_p>
                <execute instruction function using decode_table lookup data>

    <check for break points and flag>
```

The above pseudo-code is a rough outline only, and doesn't show callback handling, memory accesses, disassembling or instruction execution (though this last is described below, in the "Disassembled Output" section).

Key Model State

The list below show some of the major state used in the model.

- **state**: Contains all the CPU's modelled registers, e.g. `r[32]`, `pc`, `im` etc. There is a field corresponding to each register in the Mico32 CPU, including debug registers. It is of type `lm32_cpu::lm32_state`. It also carries other persistent state, used by the model, that will need saving for save and restore operations

- **state.int_flags**: bitmapped value indicating pending exception. Each of the bits, from bit 0 to bit 7, corresponds to the exception ID as defined in the reference manual [1]. This is part of the `state` structure.
 - **state.cycle_count**: number of executed cycles since time 0. Note that this is not the number of instructions executed. Instructions that take multiple cycles, increment this count by more than 1. This is part of the `state` structure.
- **mem**: pointer to internal memory. This can be `NULL` if none defined, and all memory handled by callback functions.
- **mem_tag**: pointer to internal memory tag that contains debug tag data to mark the access types for internal memory locations. Can be `NULL`—see **mem** above.
- **rt**: CPU general purpose registers' next availability times. See the 'Instruction Functions' section below for details of usage.
- **tbl_p**: pointer to table of instruction function pointers. See the 'Decode Table' section below for more details.
- **decode_table**: table of instruction decode information. See the 'Decode Table' section below for more details.

Decode Table

At the heart of the execution of the model is a decode table used for quick lookup of decode information for a given instruction's opcode. The decode table consists of 64 entries with the following structure type:

```
struct lm32_decode_table_t {
    const char* instr_name;
    unsigned    instr_fmt;
    lm32_time_t result_cycles;
    lm32_time_t issue_cycles;
    lm32_time_t issue_not_takencycles;
    unsigned    signed_imm;}
```

It is a constant table, and held in the global `decode_table` variable, initialised at compilation. The `instr_name` field is a string for disassembly purposes, whilst the `instr_fmt` gives information as to the instruction format for that opcode. There are slightly more formats than that defined in the reference manual [1], as quirks of some instructions need uniquely identifying. The definitions in `lm32_cpu_mico32.h` prefixed “`LM32_FMT_`” give all the possible values. The three time based fields, correspond to the values of cycle taken for each instruction as defined by the reference manual [1], with an issue count a results count and (if a decision branch) a not taken issue count. The `signed_imm` field indicates whether any immediate bits of the instruction are signed or not. For instructions with no immediate value, this is a “don't care”. An example initialisation for a table entry, for the `sxtb` instruction is shown below:

```
{"sxtb    ",    INSTR_FMT_RC,    1,    1,    0, INSTR_SE_DONT_CARE}
```

The table is used during execution of instructions. During decode, a structure is used for constructing decode information, as shown below.

```
struct lm32_decode_t {
    uint32_t    opcode;
    uint32_t    reg0_csr;
    uint32_t    reg1;
    uint32_t    reg2;
    uint32_t    imm;
    const lm32_decode_table_t* decode; }
```

This structure is like a form that is filled in as the instruction is processed. The 'opcode' field is set with the raw fetched instruction value, and then the fields are separated into the `regXX` and `imm` fields, depending on the instruction type. The type is derived from the last field which is a pointer to an entry in the `decode_table`, described above. During decode the opcode is used to fetch the `decode_table` location for the instruction, and the pointer to the entry is stored in the decode field. It is a pointer to this structure that is ultimately passed in to the instruction functions for use in execution the instruction functionality.

The `tbl_p` pointer of the `lm32_cpu` class points to a table of 64 entries, corresponding to the 64 opcodes, and contains pointers to functions that will implement that opcode's function. The table's type is an `lm32_func_table` class, with an array of pointers of type `pFunc_t`. This corresponds to a member function of `lm32_class`, with a form "`void lm32_<instr_name>(p_lm32_decode_t p)`", with the sole argument being a pointer to an object of type `lm32_decode_t`, as shown above. The table is constructed and initialised in the constructor of the `lm32_cpu` class.

Instruction Functions

The actual instruction execution functions are defined in the source file `lm32_cpu_inst.cpp`, and all have a similar basic format. An example is shown below for the byte sign extend instruction (`sxtb`).

```
void lm32_cpu::lm32_sxtb (p_lm32_decode_t p) {
    if (state.cfg & (1 << LM32_CFG_X)) {
        cycle_count += calc_stall(p->reg0_csr, NULL_REG_IDX, cycle_count);

        int32_t ry = SIGN_EXT8(state.r[p->reg0_csr] & BYTE_MASK);

        state.r[p->reg2] = ry;
        rt[p->reg2] = cycle_count + p->decode->result_cycles;

        state.pc += 4;

        cycle_count += p->decode->issue_cycles;
    } else
        lm32_rsrvd(p);
}
```

The function is passed in a pointer to the decode information looked up in `execute()`, and filled in with extracted argument fields (e.g. `rx`, `rz` indices, or immediate values etc.). As sign extension is an optional feature, the `CFG` register state (`state.cfg`) is inspected, and if sign extension is not implemented, then the `lm32_rsrvd()` instruction function is called instead. Not all instructions are optionally implemented, and these instruction's functions don't have a test like this.

If it is implemented, then the first job is to see if any source registers are stalled. In this case there is only one source register (indexed by `p->reg0_csr`), and `calc_stall()` is called that returns a number representing any number of cycles to wait before that source register is available. The 'rt' table has a list of cycle counts indicating when each of the 32 general purpose registers are next available. Any source register for an instruction that has an 'rt' entry in the future (relative to `cycle_count`) generates a wait state count that is the difference between `cycle_count` and the 'rt' entry for the register. In the case of instructions with two source registers, the larger of the two calculated wait cycles is returned. This is added to the current `cycle_count` to effectively delay execution of the instruction.

The value of the register indexed is retrieved from state and signed extended, as required for this instruction's function, into a variable `ry`. The destination register, indexed by `p->reg2`, is updated with the `ry` value, and then the 'rt' table entry for the indexed destination register is updated to contain the cycle time that it will next be available. This is the current `cycle_count` (with stalling already added), plus the `result_cycles` for this particular executed instruction, as defined in the `decode_table` entry passed into the function.

The `PC` is incremented to the next instruction (for branches this might be to a different address), and the `cycle_count` incremented by the value of the `issue_cycles` for the instruction, as defined in the `decode_table` entry, that was passed into the function via the 'p' pointer.

The 64 opcodes all have an entry in the `tbl_p` table, and point to a function like that in the above example.

Testing

Test Platform

As has been mentioned above, an executable environment, `cpumico32`, is constructed that instantiates the `lm32` model, and provides sufficient control and facilities to allow the model to be fully tested. This includes a command line control interface for configuring the model and testing, as well as a set of callback functions to allow testing of such things as interrupts etc.

Detailed discussion of the code is not undertaken here—the code is not complicated, and inspection of the source should be sufficient—but a brief description of the program’s usage is given. The usage message for `cpumico32` is as follows:

```
Usage: cpumico32 [-h] [-g] [-G <COM #>] [-v] [-x] [-d] [-D] [-I]
      [-n <num>] [-b <addr>] [-r <addr>]
      [-R <num>] [-f <filename>] [-m <num>] [-o <addr>] [-e <addr>]
      [-l <filename>] [-c <num>] [-w <wait states>] [-i <filename>] [-T]

-h Display this help message
-g Start up in GDB remote debug mode (default: off)
-G Specify COM port to use for GDB remote debug (default: 6)
-n Specify number of instructions to run (default: run forever)
-b Specify address for breakpoint (default: none)
-f Specify executable ELF file (default: test.elf)
-l Specify log file output (default: stdout)
-m Specify size of internal memory in bytes (default: 65536)
-o Internal memory offset (default 0x00000000)
-e Specify an entry point address (default 0x00000000)
-v Specify verbose output (default: off)
-x Enable disassemble mode (default: disabled)
-d Disable breaking on lock condition (default: enabled)
-r Address to dump value from internal ram after completion (default: no dump)
-R Number of bytes to dump from RAM if -r specified (default 4)
-D Dump registers after completion (default: no dump)
-I Dump number of instructions executed (default: no dump)
-c Set configuration word value to enable/disable features
-w Set the number of wait states for internal memory (default 0)
-i Specify a .ini filename to use for configuration (default none)
-T Enable internal callback functions for test (default disabled)
```

For the most part, the command line options map directly to configuration options of the model’s constructor, or configuration methods. The options `-m`, `-o`, `-e`, `-v`, `-x`, `-d`, `-c`, and `-w` all get mapped to the constructor’s inputs unmodified. The `-l` option specifies an executable filename, which `cpumico32` opens for writing, and passes the file pointer to the constructor. The option `-g` (plus `-G`, in windows version) runs the program in GDB debug mode, which is discussed in the “GDB Interface” section.

The options `-f`, `-n`, `-b` map to the first three arguments of the `lm32_run_program()` method (`elf_name`, `run_cycles` and `break_addr` respectively). The `exec_type` and `load_code` inputs are controlled internally by the program, with the `exec_type` value defaulting to `LM32_RUN_FROM_RESET`, but this can be overwritten by a test program to change its type, via the memory callback function. The program always loads the specified ELF program, but controls loading of the code in case of a break, where `lm32_run_program()` will be re-entered, but the code does not need reloading.

The options `-r`, `-R`, `-D` and `-I` all control the post-run calling of debug data dumping. The `-r` and `-R` has the program dump memory values from internal RAM, via the `lm32_read_mem()` method, specifying the start address and the number of bytes (always rounded to a whole word). The `-D` option causes a call to the `lm32_dump_registers()` method, and `-I` a call to the `lm32_get_num_instructions()` method.

As mentioned before, the `cpumico32` program has internal callback functions for the three callback that can be registered with the model. These are specific to testing the model, and can have side effects if non-test code is run. Therefore, by default, they are not enabled. when testing, a `-T` option is specified to enable them.

All of the above configuration command line options, and additional configurations, can be set by using a `.ini` configuration file. The `-i` option is used to specify the `.ini` file to use. Values specified in this configuration file override the default values but, in turn, can be overridden by the command line option, allowing a mix of methods, and final command line control. The default test `.ini` file, used by model testing, is show below:

```
;
; INI file used for test. DO NOT EDIT!
;
```

```
[program]
filename=test.elf
entry_point_addr=0
```

```
[configuration]
cfg_word=0x11203f7
```

```
[debug]
log_fname=stdout
test_mode=false
verbose=false
ram_dump_addr=-1
ram_dump_bytes=0
dump_registers=false
dump_num_exec_instr=false
disassemble_run=false
```

```
[breakpoints]
user_break_addr=-1
num_run_instructions=-1
disable_reset_break=false
disable_hw_break=false
disable_lock_break=false
```

```
[memory]
mem_size=65536
mem_wait_states=0
mem_offset=0
```

```
[dcache]
cache_base_addr=0
cache_limit=0xffffffff
cache_num_sets=512
cache_num_ways=2
cache_bytes_per_line=4
```

```
[icache]
cache_base_addr=0
cache_limit=0x7fffffff
cache_num_sets=1024
cache_num_ways=2
cache_bytes_per_line=4
```

The first five sections should be fairly self-explanatory, and map to command line options. The two cache sections allow control of the cache configurations that are passed into the model's constructor, which have no command line equivalent, and so can only be modified from default settings with a `.ini` configuration file.

Callback Functionality

The `cpumico32` program implements and registers three callbacks with the model in order to allow full coverage in testing the model. It provides a means to generate external interrupts, with a time to fire them, a means to alter the configuration register to dynamically enable or

disable hardware features, a means to reset the model, changing the execution type as it does so, and to test JTAG accesses.

These controls (except the JTAG) are implemented by memory mapping 'registers' from location `0x20000000`, implemented in the memory callback function, with offsets defined in the source code. An interrupt pattern can be written at offset `COMMS_INT_PATTERN_OFFSET`, along with a time (relative to current time) at `COMMS_TIME_OFFSET`. The interrupt callback function, when called will generate an interrupt if any of the pattern bits are set, after the time set by a write to the `COMMS_TIME_OFFSET` register.

The individual bits of the configuration register (CFG) can be written to via the next set of locally defined offsets (`COMMS_NUM_INT_OFFSET` to `COMMS_WDOG_EN_OFFSET`). Note that when reading these locations returns the whole configuration register value, not the individual bit. At an offset of `COMMS_RESET_OFFSET`, a write will reset the model, as if the reset pin had been activated, and also set the local execution type variable to whatever data value was written, to override the default.

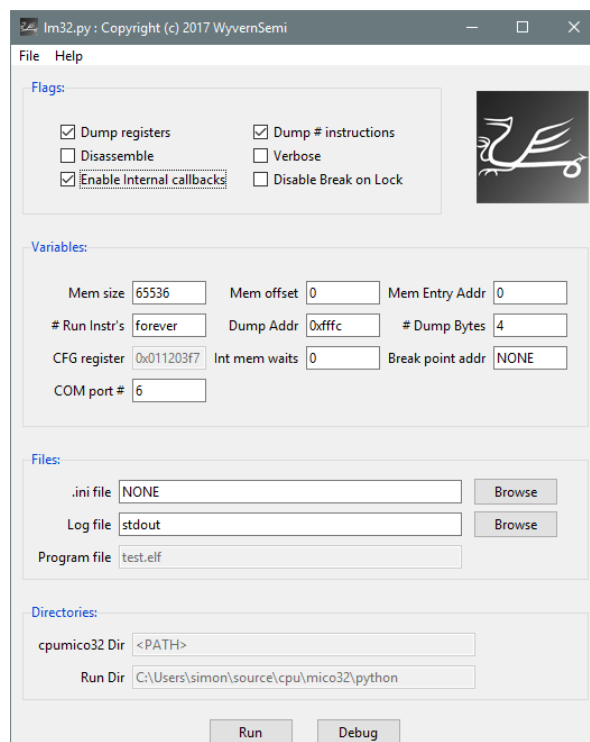
The JTAG callback function implements a simply loopback functionality. JTAG transmit register write loads a value to a local variable, that can be read when the JTAG receive register is read (or the TX register read).

With this functionality and configurability in `cpumico32` all features of the model can be exercised, and a set of assembler code tests have been constructed to do just that, detailed in the Test Code section.

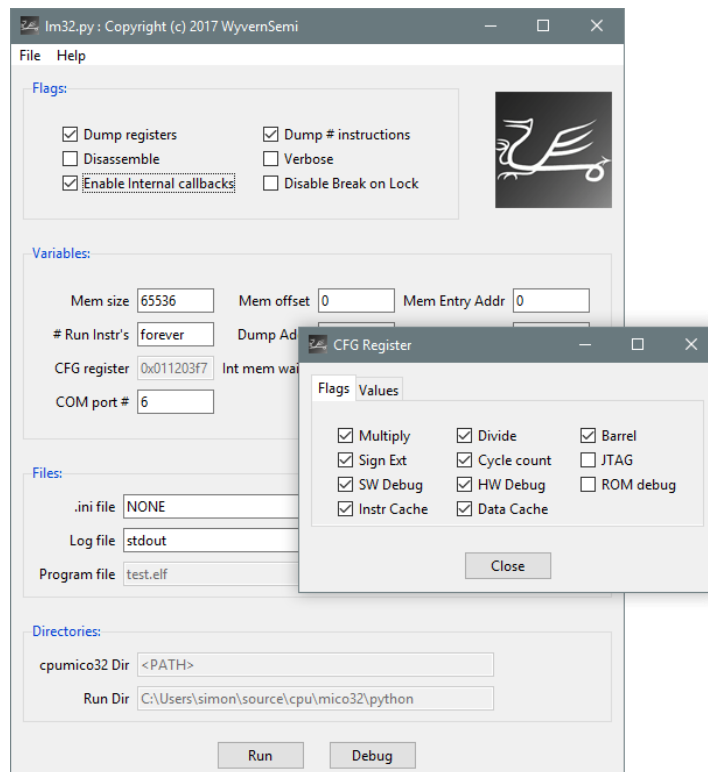
GUI for `cpumico32`

The package comes with a python based GUI for `cpumico32` program, if configuraing the command lines seems to complicated and cumbersome. It is located in the `python/` directory and is called `lm32.py`. The script uses python3, and the `tkinter` module, which is usually bundled with the python package.

When run (e.g., on windows, `python3 python/lm32.py`), a window appears looking like the following:



All the flags and arguments of the command line can be controlled from this GUI. The windows will open with the default values of the `cpumico32` program, and the GUI is used to adjust these. It will check for valid inputs, and raise an error if outside of prescribed limits. The program to be run is selected from the menu under 'File->Open ELF File...', with the selection displayed at the bottom in the 'Utility files' frame. The CFG register setting is shown in the 'Variables' frame, but greyed out. To alter this value, the box may be double-clicked with the mouse, and a new popup appears, like that shown below:



The popup window has two tabs, with one for the binary flags enabling or disabling features, whilst the other has values defining the number of watch- and breakpoints, as well as the number of external interrupts. Updating the flags and values automatically updates the value in the configuration register box.

When configuration is complete, the 'Run' button at the bottom can be pressed, and it will execute a `cpumico32` command with all the appropriate command line arguments. It is assumed that `cpumico32` is on the search path, or is in the directory from whence the script was run.

By default, the program will search for the `cpumico32` executable on the path, as indicated by the 'cpumico32 Dir' box in the Directories area. This can be changed from the file menu ('File->cpumico2 Folder...'), to select a particular folder containing an executable. This is useful to select between a debug or a release development version, say. In addition, the program uses the directory from which it was run as the working directory (as indicated by the 'Run Dir' box), but this can also be changed from the file menu ('File->cpumico2 Folder...'). Any relative references (such as for the program file, log file, etc.) are automatically updated if the run directory is changed.

The output from running the command is sent to a new window, including the contents of any specified log file. The window will look something like the following figure (where, in this example, registers and number of executed instructions are dumped, and the contents of

memory at `0xffffc` are printed). The first line in the window is the command that was executed, with the command line options, as a reference for using in scripting etc.

```

output
cpumico32.exe -DIT -r 0xffffc

r00 = 0x00000000 r01 = 0x00000315 r02 = 0xfffffe43 r03 = 0xfffffe43
r04 = 0xffffffff r05 = 0x776f646e r06 = 0x694b2073 r07 = 0x385c7374
r08 = 0x575c312e r09 = 0x6f646e69 r10 = 0x50207377 r11 = 0x6f667265
r12 = 0x6e616d72 r13 = 0x54206563 r14 = 0x6b6c6f6f r15 = 0x3b5c7469
r16 = 0x505c3a43 r17 = 0x72676f72 r18 = 0x46206d61 r19 = 0x73656c69
r20 = 0x38782820 r21 = 0x4d5c2936 r22 = 0x6f726369 r23 = 0x74666f73
r24 = 0x20535620 r25 = 0x65646f43 gp = 0x6e69625c fp = 0x5c3a433b
sp = 0x6c6f6f54 ra = 0x6c415c73 ea = 0x0000900d ba = 0x0000ffff

pc = 0x000000e8 ie = 0x00000000 ip = 0x00000000 im = 0x00000000
icc = 0x00000000 dcc = 0x00000000 cfg = 0x01120bf7 cfg2 = 0x00000000
cc = 0x0000004e eba = 0x00000000

bp0 = 0xffffffff bp1 = 0xffffffff bp2 = 0xffffffff bp3 = 0xffffffff
wp0 = 0xffffffff wp1 = 0xffffffff wp2 = 0xffffffff wp3 = 0xffffffff
dc = 0x00000000 deba = 0x00000000

Number of executed instructions = 51

RAM 0xffffc = 0x0000900d

```

Test Code

A set of assembler programs were developed and are provided for execution on `cpumico32`, that execute a range of self-tests to verify the model. In order to compile and run these tests it is assumed that the LatticeMico System Development Tools and the Lattice Diamond Design Software (required for the Mico tools) are downloaded and installed. These are freely available from the Lattice Semiconductor website (<http://www.latticesemi.com/>) under the “Support->FPGA Software Home” page.

These tests are all directed tests, but cover nearly all aspects of the model including all instructions and all exceptions. Each program lives in a solitary directory under the directory `test/<category>/` and each sub-directory has a single source file, `test.s`. These tests are self-checking and return a value `0x0000900d` in memory location `0x0000ffffc` if the test passes, or `0x00000bad` if it fails (if the program never terminates cleanly, then this value is undefined—but is unlikely to be the pass value). Below is listed the features covered by the tests, and the test directory that contains the test that covers that feature.

Arithmetic instructions

Instruction	Covering test location	Status
add	instructions/add/	Completed
addi	instructions/add/	Completed
sub	instructions/sub/	Completed
sextb	instructions/sext/	Completed
sextb	instructions/sext/	Completed
mul	instructions/mul/	Completed
muli	instructions/mul/	Completed

div*	instructions/div/	Completed
divu	instructions/div/	Completed
mod*	instructions/div/	Completed
modu	instructions/div/	Completed

Comparative instructions

Instruction	Covering test location	Status
cmpe	instructions/cmp_e_ne	Completed
cmpei	instructions/cmp_e_ne	Completed
cmpne	instructions/cmp_e_ne	Completed
cmpnei	instructions/cmp_e_ne	Completed
cmpg	instructions/cmpg/	Completed
cmpgi	instructions/cmpg/	Completed
cmpgu	instructions/cmpg/	Completed
cmpgui	instructions/cmpg/	Completed
cmpge	instructions/cmpge/	Completed
cmpgei	instructions/cmpge/	Completed
cmpgeu	instructions/cmpge/	Completed
cmpgeui	instructions/cmpge/	Completed

Shift instructions

Instruction	Covering test location	Status
sl	instructions/sl/	Completed
sli	instructions/sl/	Completed
sr	instructions/sr/	Completed
sri	instructions/sr/	Completed
sru	instructions/sr/	Completed
srui	instructions/sr/	Completed

Logical instructions

Instruction	Covering test location	Status
and	instructions/and/	Completed
andi	instructions/and/	Completed
andhi	instructions/and/	Completed
or	instructions/or/	Completed
ori	instructions/or/	Completed
orhi	instructions/or/	Completed
nor	instructions/or/	Completed
nori	instructions/or/	Completed
xor	instructions/xor/	Completed
xori	instructions/xor/	Completed
xnori	instructions/xor/	Completed
xnor	instructions/xor/	Completed

Branch instructions

Instruction	Covering test location	Status
be	instructions/branch_cond/	Completed
bne	instructions/branch_cond/	Completed
bg	instructions/branch_cond/	Completed
bgu	instructions/branch_cond/	Completed
bge	instructions/branch_cond/	Completed
bgeu	instructions/branch_cond/	Completed
b	instructions/branch_uncond/	Completed
bi	instructions/branch_uncond/	Completed

call	instructions/branch_uncond/	Completed
calli	instructions/branch_uncond/	Completed

Memory access instructions

Instruction	Covering test location	Status
lb	instructions/load/	Completed
lbu	instructions/load/	Completed
lh	instructions/load/	Completed
lhu	instructions/load/	Completed
lw	instructions/load/	Completed
sb	instructions/store/	Completed
sh	instructions/store/	Completed
sw	instructions/store/	Completed

Control/Status access instructions

Instruction	Covering test location	Status
rcsr	instructions/csr/	Completed
wcsr	instructions/csr/	Completed

* Note that the `div` and `mod` instructions are listed in the instruction table in the LatticMico32 Processor Reference Manual [1], but are not documented in the instruction descriptions. They are not supported in the GNU assembler either. The implementation in this ISS implementation assumes signed arithmetic and the tests use `'.word <opcode>'` to insert the instruction into the test that the assembler won't recognise and compile.

Exceptions

Exception	Covering test location	Status
reset	exceptions/instruction/	Completed
divide by 0	exceptions/instruction/	Completed
system call	exceptions/instruction/	Completed
break instr	exceptions/instruction/	Completed
DC.RE	exceptions/instruction/	Completed
ext interrupt	exceptions/external/	Completed
rsrwd	exceptions/ibus_errors/	Completed
instr bus err	exceptions/ibus_errors/	Completed
disable instr	exceptions/ibus_errors/	Completed
data bus err	exceptions/dbus_errors/	Completed
hw breakpoint	exceptions/hw_debug/	Completed
hw watchpoint	exceptions/hw_debug/	Completed
Reset Event	exceptions/hw_debug/	Completed

ISS user API testing

Test	Covering test location	Status
Mem callback	exceptions/external/	Completed
Int callback	exceptions/external/	Completed
Instr count	api/num_instr/	Completed
Run-time ctrl	covered in cpumico32.cpp	Completed
HW debug ctrl	covered in cpumico32.cpp	Completed
State access	covered in cpumico32.cpp	Completed
Re-entrance	covered in cpumico32.cpp	Completed
Extnl breaks	covered in cpumico32.cpp	Completed

Executing Tests

The tests are all run via a `runtest.sh` script that lives in the `test/` directory. Changing directory to `test/` and running `runtest.sh` will execute all the tests, giving a pass/fail criteria for each, with a summary at the end. An easier way to execute the tests, especially when doing coverage measurements, is to use the `makefile`. When building code, a command `make test` will get the build up-to-date, and then run the test script. The tail end of the output should be something like that shown below:

```
.
.
Running test  exceptions/dbus_errors
PASS
Running test  exceptions/hw_debug
PASS
Running test  api/num_instr
PASS

Tests run : 24
Tests pass: 24
Tests fail: 0
```

The test script runs `cpumico32` with arguments of `'-T -r 0xffffc'`, but additional arguments can be added by setting the environment variable `CPUMICO32_ARGS`. This must contain a string of valid `cpumico32` arguments but, even when valid, it is not guaranteed that testing will pass for all possible argument combinations, so use with care.

Coverage

Coverage for the self-tests was performed using `gcov` and `lcov`, with support in the `makefile`. Excluded from the coverage was any disassembler or debug output code as, although this can be covered to a level of 100%, it cannot be verified in an automatic self-test, and it does not affect the accuracy of the model. Similarly, the `cpumico32` top level code was not included, as this is a test/example program, and not part of the model. The core files covered were thus:

- `lm32_cpu.cpp`
- `lm32_cpu.h`
- `lm32_cpu_inst.cpp`
- `lm32_cache.cpp`
- `lm32_cache.h`
- `lm32_cpu_elf.cpp`

The diagram below shows the LCOV report generated by executing the following commands:

```
make clean
make COVOPS="-coverage" test
make coverage
```

The report generated is created in the directory `cov_html/src`, and accessed via `index.html`.

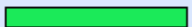
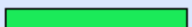
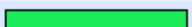
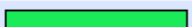
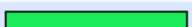
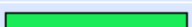
LCOV - code coverage report

Current view: [top level - src](#)

Test: [lm32.info](#)

Date: 2013-07-24

	Hit	Total	Coverage
Lines:	1168	1168	100.0 %
Functions:	89	89	100.0 %

Filename	Line Coverage ↕		Functions ↕	
lm32_cache.cpp		100.0 %	42 / 42	100.0 % 3 / 3
lm32_cache.h		100.0 %	1 / 1	100.0 % 1 / 1
lm32_cpu.cpp		100.0 %	367 / 367	100.0 % 15 / 15
lm32_cpu.h		100.0 %	10 / 10	100.0 % 5 / 5
lm32_cpu_elf.cpp		100.0 %	30 / 30	100.0 % 1 / 1
lm32_cpu_inst.cpp		100.0 %	718 / 718	100.0 % 64 / 64

Generated by: [LCOV version 1.10](#)

In order to obtain a goal of 100% coverage, some waivers on lines of code were needed on unreachable lines of code. The exceptions followed 6 broad categories, detailed below:

Checks on parameters etc. that should never fail, and are meant as debug aids for invalid calls from elsewhere in the code, or to protect against invalid values from the API. The following waivers are of this type.

method	coverage waiver
lm32_set_verbosity_level()	invalid verbosity level
lm32_set_hw_debug_reg()	invalid debug register type
lm32_read_mem()	invalid read access type
lm32_write_mem()	invalid write access type
interrupt()	invalid exception ID
lm32_rcsr()	Invalid CSR register index
lm32_wcsr()	Invalid CSR register index
lm32_cache()	Parameter checks

Memory allocation failures that should never happen. Mostly [malloc\(\)](#) calls, where a failure here would indicate a system level problem. The following were waived on this basis:

method	coverage waiver
lm32_write_mem()	memory allocation, and error handling
lm32_run_program()	memory allocation failure

Code associated with disassembled and debug output, cannot be self-tested. Coverage is possible, but not meaningful. [lm32_cpu_disassembler.cpp](#) was wholly excluded, but calls to disassembler methods from the core functionality still needed waiving:

method	coverage waiver
execute_instruction()	call to disassemble() function and pc update in disassemble mode
lm32_run_program()	break on disassemble run

User defined break address handling actually terminates the program, and so cannot be self-tested. The feature is for debug purposes only, in any case, so a waiver was added:

method	coverage waiver
<code>lm32_run_program()</code>	user break address trap

File exception handling also terminates the program, and has no meaning in terms of model accuracy:

method	coverage waiver
<code>read_elf()</code>	file opening exception handling
<code>read_elf()</code>	unexpected EOF handling
<code>read_elf()</code>	program load overflowing memory

ELF file checks only fire with an invalid ELF executable file, and don't affect the model's accuracy for executing a valid ELF file, and so the checks were waived:

method	coverage waiver
<code>read_elf()</code>	all ELF header checks

Creation of cache without parameters is not done in the testing, as the tests exercise the various settings (including the defaults) by explicitly setting the cache configurations.

method	coverage waiver
<code>lm32_set_configuration</code>	default parameters on cache object creation

With the above waivers in place, the three listed files, containing all the methods, bar the disassembling, have 100% coverage.

Not Covered

Despite the 100% measured coverage metrics, there are some aspects of the model as yet uncovered by formal testing.

- Various internal memory sizes (all tests run with the default 64K RAM)
- Running multiple instances of the model
- Accuracy of the timing model against a known good reference

GDB Interface

A GDB interface is available for connecting the model with a GDB session via a remote target. It does this, when configured, by opening a pseudo-terminal, and advertising the device path, which can then be used by GDB to connect to the model using its command `target remote <device>`. With this arrangement the model then looks like a hardware system connected to the host via a serial interface. One important difference is that the debugging is non-intrusive on the model. Normally for hardware systems connected via a serial port a 'stub' is required to be compiled with the program ([6], section 20.5), along with some user supplied routines that know about the local serial interface protocols etc, that intercept interrupts and communicate with GDB to affect the debugging functionality. With this implementation, with the advantages of visibility within the model, the code being debugged does not need to be modified with an additional stub.

As mentioned elsewhere, it is expected that the Lattice Semiconductor tool chain for the LatticeMico32 processor is available to use the GDB facility. In particular, the GDB program that must be used is `lm32-elf-gdb`.

The GDB interface is supported under both Linux and Windows (not tested under Cygwin), but uses slightly different methods between the two. For Linux it relies on the built-in pseudo-terminal facilities, which are simple to utilise, whereas on Windows third-party code or the implementation of virtual ports and null modems is required. The `com0com` project has been used to test the windows implementation of the interface, requiring the code to only open a COM serial port, and have the GDB session connect to a paired COM port, with the two communicating via a NULL modem model.

Supported Features

The implementation supports only a subset of the possible GDB commands that can be sent over a remote interface, but it supports more than the minimum required ([6], section E.1).

- Register reads and write (`P`, `p`, `G` and `g` packets)
- Memory reads and write (`X`, `M` and `m` packets)
- Single thread control (`s` and `c` packets)
- Session termination (`k` and `D` packets)
- Hardware breakpoints (`Z1` and `z1` packets)
- Hardware watchpoints (`Z[2-4]` and `z[2-4]` packets)

In addition, soft breakpoints are implicitly supported via the memory read and writes, as GDB substitutes the instruction at the break point by reading (and storing away) the original instruction, and writing a `break` instruction in its place. The breakpoint interrupt is raised when the break instruction is reached, just as for hardware breakpoints, and the model returns control to the GDB interface. Before resumption, the original instruction is restored with a further write to memory. Soft breakpoints, in reality, can only work in code that is in volatile memory, where the instructions may be overwritten. In the model all code resides in 'memory' that can be overwritten, and so this technique can be used on code, even if it is destined for ROM or Flash.

Currently only support for single threaded code is implemented, though this may change in the future.

Usage

The GDB interface is activated using the command line option `-g`, and is available on both the `cpumico32` and `lnxmico32` compiled programs. Not that for windows the `-G` option is available to run in debug mode *and* specify COM port to use, if different from the default (otherwise `-g` can still be used). The interface itself is not part of the `lm32_cpu` class, as it it needs to sit on top of the model to control it. It is implemented in the files `lm32_gdb.cpp` and `lm32_gdb.h`, and any new projects must include the header and compile in the source code in order to use it. A single function constitutes the API:

```
int process_gdb (lm32_cpu* cpu, int port_num = LM32_DEFAULT_PORT_NUM);
```

The function expects (at least) a single argument which is a pointer to an `lm32_cpu` object, which has been created and configured prior to calling the function. The configuration can include the loading of a program in to memory, though this would normally be done from within the GDB session. In the `lnxmico32` implementation, for instance, the loading and configuring of the μ Clinux code is skipped when in GDB mode. A second argument, for Windows only (it is ignored in Linux), specifies a COM port number to use. It can, and should, be omitted for Linux, and can be omitted from Windows if the default port number is that required.

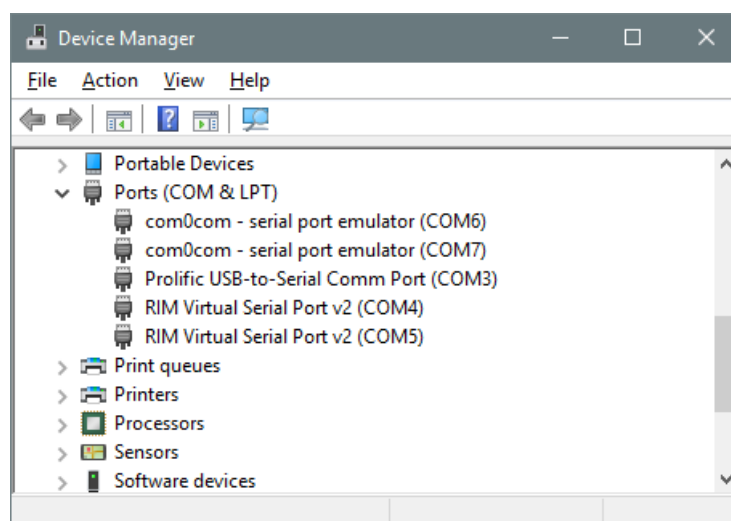
When run (e.g. `cpumico32 -g`, or `cpumico32.exe -G6`) the function will not return until it either losses attachment to a GDB session, or some error has occurred. The function returns 0 for a normal termination, otherwise an error occurred. Once called the function opens a pseudo-terminal (in Linux) and prints out the device created. E.g.:

LM32GDB: Using pseudo-terminal /dev/pts/14

Note that this terminal can change between sessions, so it must be noted for the GDB session that must connect to it. Something similar happens for Windows, but the terminal is fixed, as specified with `-G`, or the default value. The Windows message looks something like the following:

LM32GDB: Using serial port /dev/ttyS6

Despite opening a COM port, the value needed by GDB still needs to be of the form of a Linux style device file, as the program uses Cygwin under the hood. With `com0com`, the device needs to be the paired COM port with that being used by the GDB interface. The `ttySn` numbering, however is one less than the windows COM port numberings, so `ttyS6` refers to COM7. With `com0com` installed and a COM port pair added, the device manager on windows shows the port pairs similar to that shown below:



In the example above, the two ports are COM6 and COM7. The GDB interface must use the lower number of the paired port (e.g. COM6), as it advertises the tty equivalent to the COM port that is one value above it (e.g. COM7 mapping to `/dev/ttyS6`).

A typical session might look like the following. In this example one of the test programs is used as the code to debug, and compiled with symbols include

```
lm32-elf-as -g test.s -o test.o
lm32-elf-ld test.o -o test.elf
```

Assuming that the `cpumico32` is run with the `-g` option (and any other appropriate options, such as `-TDIV`, say) and has displayed the pseudo-terminal path, as above, then the GDB session, in separate terminal, can be run thus:

```
lm32-elf-gdb test.elf
GNU gdb 6.8
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=lm32-elf"...
(gdb) target remote /dev/pts/1
Remote debugging using /dev/pts/1
main () at test.s:37
37          xor      r0, r0, r0
Current language: auto; currently asm
(gdb) load
Loading section .text, size 0xec lma 0x0
Start address 0x0, load size 236
Transfer rate: 1888 bits in <1 sec, 236 bytes/write.
(gdb) hb _ok7
Hardware assisted breakpoint 1 at 0xc8: file test.s, line 109.
(gdb) b _finish
Breakpoint 2 at 0xe4: file test.s, line 122.
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
_ok7 () at test.s:111
111          sexth    r2, r2                # Sign extend
(gdb) s
_ok7 () at test.s:112
112          addi     r3, r1, TEST_VAL10
(gdb) c
Continuing.

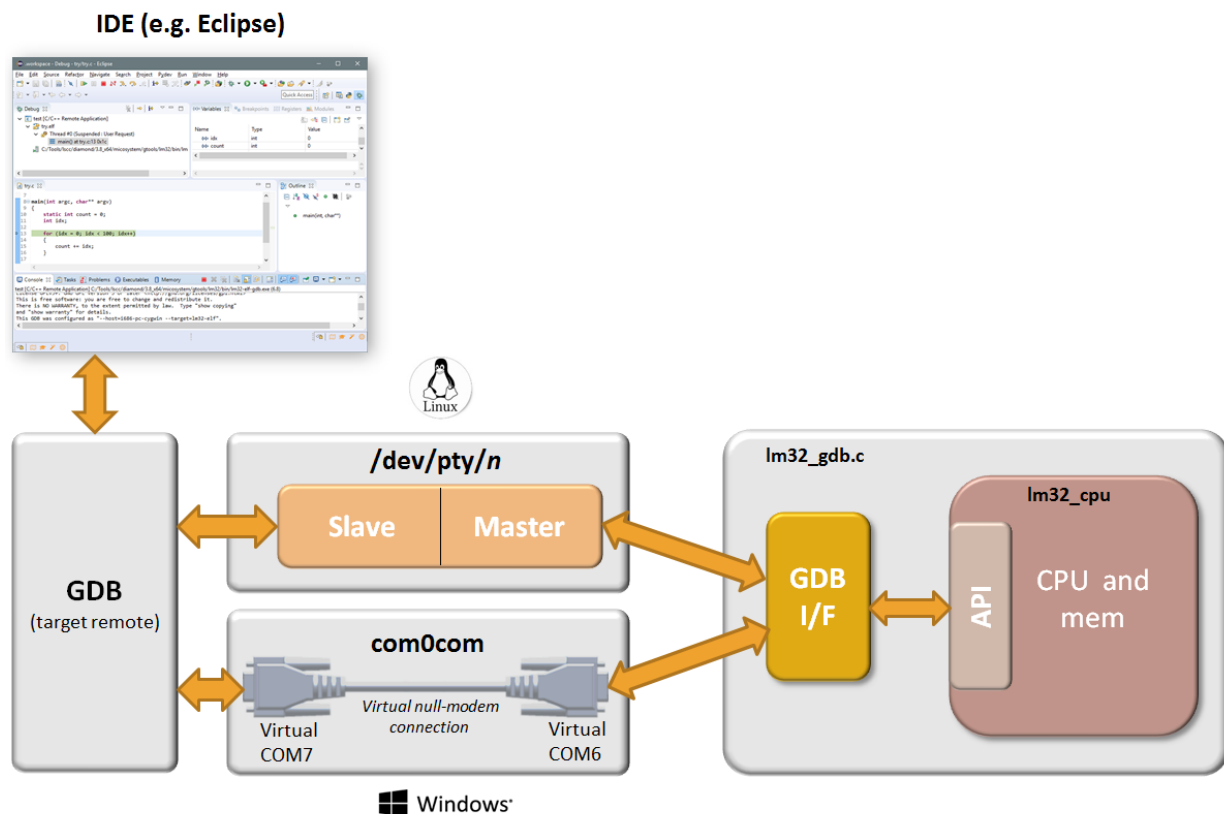
Breakpoint 2, _finish () at test.s:123
123          sw       (r31+0), r30
(gdb) s
_end () at test.s:125
125          be      r0, r0, _end
(gdb) x /wx 0xffffc
0xffffc: 0x0000900d
(gdb) detach
Ending remote debugging.
(gdb) quit
```

In the above run, the compiled code (`test.elf`) was debugged, as specified on the command line to the GDB program. Connection was established to the already running `cpumico32` program in GDB debug mode with the `target remote /dev/pts/14` command. The program is loaded into the `cpumico32` memory with the load command. We could have loaded the program when running `cpumico32`, but there is a chance that the program loaded by `cpumico32` and that for the gdb session, from whence it gets its symbol information, might

mismatch. So loading from gdb ensures that the running program and the symbol tables come from the same source.

Two breakpoints were set; one at `_ok7` with a hardware breakpoint (`hb _ok7`) and another at `_finish` using a soft breakpoint (`b _finish`). Execution is started using the 'continue' command (`c`)—as this is a remote debug session, gdb assumes the target is running already, paused waiting for commands. When the target reaches the first breakpoint GDB halts and displays the source line of the breakpoint. Next the program is stepped (`s`) to the next instruction, before another 'continue'. The program breaks again at `_finish`, and another single step taken to complete the program. Memory is inspected at location `0xffffc` (which contains the pass fail code) using `x /xw 0xffffc`. Since `0x0000900d` is displayed, the test passed. Detaching from the remote system (`detach`) ensures a clean exit by `cpumico32`, which terminates.

The above example shows the main points in using GDB to control and debug the model and a program running on it but, of course, there are many more features that can be used by GDB that are not shown in the above simple session. Any IDE using the GNU relevant toolchain, including GDB, can be used as a front end to the interface, giving full development capabilities to the model. The LatticeMico System Development Tools use Eclipse, and have remote target capabilities, normally used to connect to a development system. Details of setting this up are beyond the scope of this document, but the diagram below summarises the connections between an IDE (such as Eclipse), the GDB debugger application (for lm32) and the model's GDB interface.



Support for running in debug mode from the python GUI for `cpumico32` is available, when run under Linux. An additional 'Debug' button, next to the 'Run' button is added. Using this in place of 'Run' executes the model in debug mode, and the pseudo-terminal information is printed on `stderr` on the terminal from which the python script was run. A GDB session can connect this as described above.

Compile Options

By default, when `cpumico32` is compiled, it has the behaviour as described in the previous sections. However, it can be compiled with various definition in order to modify its behaviour. There are, presently, two conditional compile definitions that can be set:

- `LM32_FAST_COMPILE` : Removes compiling of much code that is not strictly necessary for execution to improve execution speed, but removes many features. (Used in files `lm32_cpu.cpp`, `lm32_cpu_inst.cpp`, `lm32_get_config.cpp`.)
- `LNXMICO32` : Removes some command line options and features not supported by the `lnxmico32` program (see the case study in the next section), and a test for instruction type when disassembly, since `lnxmico32` loads code differently. (Used in files `lm32_cpu.cpp`, `lm32_get_config.cpp`.)

The `LM32_FAST_COMPILE` definition is used to remove as much code from the model, whilst still retaining a viable simulation, in order to maximise the execution speed. To this end, the following features are removed:

- Break point specification removed
 - Removes command line options `-n`, `-b`, `-d`
 - Can still break from interrupt callback
- Memory wait state specification
 - Removes command line option `-w`
- Disassemble mode
 - Removes command line option `-x`
- Removes timing accuracy
 - No cache modelling of timing
 - Callback timing information is ignored
 - Pipeline stalling not calculated
- Memory access alignment checks are disabled
- Memory stats information is not logged (memory tagging removed)
- No watchpoint support
- No hardware breakpoint support

With the compile definition, an additional memory restriction applies—memory size must be a power of 2, as the range masking requires this to avoid using a division.

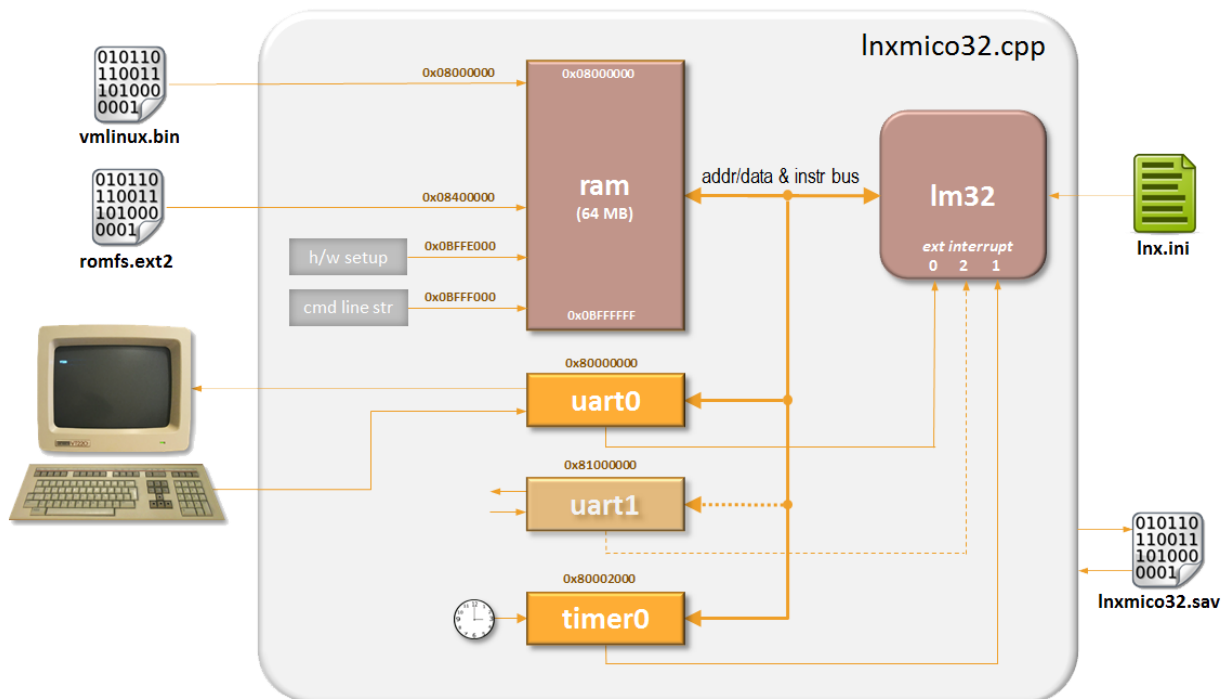
The `LNXMICO32` definition simply removes some further command line options and functionality not needed by the `lnxmico32` case study program described in detail in the next section. The configuration code (`lm32_get_config.cpp`) is shared between the `lnxmico32` and `cpumico32` programs, and the majority of the uses of the definition are found in this file. With the definition active, the `-T` option is not used, as the test callbacks are not implemented in `lnxmico32`, which has its own callback functions, used to model peripherals. In addition a `-V` option is added, but only if `LM32_FAST_COMPILE` is not defined as well, which acts like `-v`, but allows specification of a cycle time when verbosity is activated. The `LNXMICO32` program loads binary images directly to memory, rather than load an ELF file, so the `-f` option is also removed.

One other use, modifying `lm32_cpu.cpp` code, is to remove the test to see if a memory location is an instruction before disassembling it. The `lnxmico32` program loads code as binary data via the `lm32_write_mem()` function, and thus does not label it as instruction data. In order to debug the code, the check is suspended for this compilation option.

Case Study: An Embedded Linux System

A case study in the usage of the model is given here, in order to demonstrate the features and extensibility of the ISS. A basic, non-mmio Linux system is put together, using the u-boot and uClinux ports to Im32. A minimal system is put together in order to be able to boot the Linux OS, with a light weight Unix environment provided by BusyBox, targeted at embedded platforms.

The diagram below shows the general system layout. It consists of the mico32 model, with two UARTs (UART0 and UART1) and a timer (TIMER0). These are modelled as part of the `lnxmico32` environment (see `lnxuart.cpp` and `lnxtimer.cpp`), and model the Lattice IP implementations for these functions.



The top level for the system is called `lnxmico32`, and has a top level source file `lnxmico32.cpp`. This instantiates mico32 model, shown as the `Im32` and `RAM` boxes in the diagram. It registers its own callback functions for both the external memory accesses (using the model's API method `lm32_register_ext_mem_callback()`) and interrupts (using `lm32_register_int_callback()`). The callbacks handle the register accesses to the peripherals, along with the 'ticking' and passing back interrupt status.

When run, the two binary files, `vmlinux.bin` and `romfs.ext2`, are expected to be in the directory from which the program is executed—as is, by default, a configuration file, `lnx.ini` (see next section).

The binary images for the u-boot/uClinux and RAM filesystem are loaded to memory (at `0x08000000` and `0x08400000`, respectively), and then memory is updated for hardware setup configuration, and an initial boot command string. The simulation can then be started, and the system boots, sending output characters, via UART0, and, once booted, accepting keyboard input to allow logging in and issuing of commands in the shell (`msh`—minimal shell). After boot the screen will look something like the following:

```
~/src/cpu/mico32/test
io scheduler deadline registered
io scheduler cfq registered (default)
lm32uart: Lattice Mico 32 UART driver
ttyS0 at I/O 0x80000000 (irq = 0) is a LM32UART
lm32uart: added port 0 with irq 0 at 0x80000000
ttyS1 at I/O 0x81000000 (irq = 2) is a LM32UART
lm32uart: added port 1 with irq 2 at 0x81000000
RAMDISK driver initialized: 16 RAM disks of 16384K size 1024 blocksize
PPP generic driver version 2.4.2
PPP Deflate Compression module registered
PPP BSD Compression module registered
SLIP: version 0.8.4-NET3.019-NEWTTY (dynamic channels, max=256).
IPv4 over IPv4 tunneling driver
TCP bic registered
TCP cubic registered
TCP westwood registered
TCP htcp registered
NET: Registered protocol family 1
NET: Registered protocol family 17
RAMDISK: ext2 filesystem found at block 0
RAMDISK: Loading 2048KiB [1 disk] into ram disk... done.
VFS: Mounted root (ext2 filesystem) readonly.
mico32 login: █
```

To login to the system, login as `root`, with a password of `lattice`. To exit from the program, from anywhere, type `#!exit!` and press enter.

Configuration

The lm32 model

The mico32 model must be configured correctly for the system to boot properly and, by default, the program, will look for a configuration file `lnx.ini` in the directory from which it is run. This can, of course, be overridden with the `-i` command line option. The `lnxmico32` program shares a number of command line options of the `cpumico32` program, (indeed, it shares common configuration code). The full usage message for the `lnxmico32` program is as follows:

```
Usage: lnxmico32 [-D] [-I] [-r <addr>] [-R <num>]
              [-l <filename>] [-c <num>] [-i <filename>] [-s <filename>] [-S] [-L]

-l Specify log file output (default: stdout)
-r Address to dump value from internal ram after completion (default: no dump)
-R Number of bytes to dump from RAM if -r specified (default 4)
-D Dump registers after completion (default: no dump)
-I Dump number of instructions executed (default: no dump)
-c Set configuration word value to enable/disable features
-i Specify a .ini filename to use for configuration (default none)
-s Specify .sav filename (default lnxmico32.sav)
-S Save state on exit (default no save)
-L Load saved state before execution (default no load)
```

The provided `lnx.ini` options file, for the most part specifies default, but does set the configuration word to a specific value that represents the minimum configuration for `lnxmico32` functionality. The file looks like the following:

```
;
; INI file used for lnxmico32. DO NOT EDIT!
;
[configuration]
cfg_word=0x00003017

[debug]
log_fname=stdout
```

```

ram_dump_addr=-1
ram_dump_bytes=0
dump_registers=false
dump_num_exec_instr=false

[state]
save_file_name=lnxmico32.sav
save_state=false
load_state=false

; When LM32_FAST_COMPILE not defined
; verbose=false
; disassemble_run=false

; [breakpoints]
; user_break_addr=-1
; num_run_instructions=-1
; disable_reset_break=false
; disable_hw_break=false
; disable_lock_break=false

; [memory]
; mem_wait_states=0

; [dcache]
; cache_base_addr=0
; cache_limit=0xffffffff
; cache_num_sets=512
; cache_num_ways=2
; cache_bytes_per_line=4

; [icache]
; cache_base_addr=0
; cache_limit=0x7fffffff
; cache_num_sets=1024
; cache_num_ways=2
; cache_bytes_per_line=4

```

Some of the options (those commented out) are only available if `lnxmico32` is not compiled with `LM32_FAST_COMPILE` defined, which disables disassembling, breakpoints, memory wait states and cache timing simulation. These can be reinstated when compiled without the definition, but will cause a warning if uncommented when compiled with it defined.

The system software

To configure boot and OS software before running, three things need to happen:

- A hardware setup table must be constructed at `0x0BFFE000`
- A boot command line string placed at `0x0BFFF000`
- GP registers 1 to 4 pre-charged with addresses for the above two entries, plus the addresses of the `romfs.ext2` image start (`0x84000000`) and end (`0x84000000 + file length`)

The hardware setup table consists of a consecutive list of structures, with one for the CPU, the memory, the two UARTs and the timer. This is followed by a termination structure. Each structure has a similar format

```

struct {
    uint32_t length;
    uint32_t id;
    .
    .
    <specific payload>
    .
    .
}

```

The length gives the size of the entry (including the length bytes), and the ID is a unique number. The payload for each of the entries also have similar structures (except the terminator), with a 32 byte string array containing the name of the instance (which can be shorter, but not longer, than 32 bytes), followed by parameters for the particular device. The terminator is just a length (8) followed by an ID of 0, with no payload.

For the CPU (“LM32”) the payload is simply a 32 bit number for the clock frequency, in Hz. The memory (“ddr_sdram”), has a parameter for the base address, followed by the size in bytes. The timer (“timer0”) has a 32 bit word for the base address, then four bytes for a write tick count, read tick count, start/stop/control and counter width. A following 32 bit word specifies the number of reload ticks, and the a byte giving the interrupt number (i.e. which of the 32 bit external interrupt pins it is connected to). The structure is then padded to a 32 bit boundary with bytes of 0 value.

The UARTs (“uart0” and “uart1”) have a base address and baud rate parameters (both 32 bits), followed by 8 bytes for number of data bits, number of stop bits, interrupt enable, block on transmit, block on receive, RX buffer size, TX buffer size and its interrupt number. More information can be found in Appendix A of the “Linux Port to LatticeMico32 System Reference Guide” [3].

These hardware setup structures are written consecutively to memory, starting at `0x0BFFF000`, in the order, CPU, memory, timer0, uart0, uart1 and the terminator.

The command line string is used as the u-boot command arguments when starting the system. For `lnxmico32`, this is

```
root=/dev/ram0 console=ttyS0,115200 ramdisk_size=16384
```

Finally, the general purpose registers GP1 to GP4 are pre-charged with four addresses, based on the above configurations. If these were invariant, then a small assembler program could be added in memory that set these values and then jumped to the system entry point, with the initial entry point being this initialisation program. However, to ease modification to system parameters, these are written directly, using the lm32 model’s `lm32_set_gp_reg()` method. GP1 to GP4 are set to have the following addresses: The h/w setup base address, the command string base address, the RAMFS load start address, and the RAMFS load end address + 1.

Having configured the system, the execution of the code can begin.

Use of Callbacks

The `lnxmico32` system registers two callbacks with the lm32 model; one for memory accesses (`ext_mem_access()`) and one for ticking/interrupt generation (`ext_interrupt()`).

The first of these (`ext_mem_access()`) intercepts all memory accesses to the peripherals—the timer and the two UARTs. It separates out the address passed in by the lm32 model into a page address (in this case a 4KB page), and offset within that page. If the page address matches the base address of one of the peripherals, it processes the address, otherwise it simply returns with a `LM32_EXT_MEM_NOT_PROCESSED` status, informing the model it must handle this access.

All the peripheral models for the `lnxmico32` system provide three functions: a read function, a write function and a tick function (see next section). The memory callback function is called with an address and an access type. If the access type is `LM32_MEM_WR_ACCESS_WORD`, then

the selected peripheral's write function is called with the offset address and data value. If not, the read method is called with the offset address and a pointer to the data variable in which to return the read value.

When the address is matched by the callback, a processing time is returned, so that the lm32 model can advance time accordingly with the delay of the access. In `lnxmico32`, this defaults to 1 cycle for all peripheral register accesses.

The tick/interrupt callback function (`ext_interrupt()`) is called regularly by the lm32 model, with a timestamp. The callback returns an interrupts status in a 32 bit word, with each bit representing an external interrupt pin on the mico32 processor, of which there are up to 32. Each time the function is called, it calls each of the tick functions for the peripherals. For the timer this is a function that simply takes the time as an argument, and returns true or false to indicate whether it is interrupting or not. For the UARTs, they take additional parameters to return termination request status, indicate whether it is the keyboard UART and its context. The context is needed, as the function's code is common to all UART instantiations, but supports up to 4 different contexts, and identifies whether UART0 or UART1 calls. Like the timer tick function, the UART tick function returns a Boolean, indicating interrupt request status.

The callback function ORs together all the interrupt statuses of the peripherals, which is returned when the function exits. The termination request statuses of the UARTs are also combined, and if a UART is requesting termination, the value returned in the `wakeup_time` pointer is set as `LM32_EXT_TERMINATE_REQ`, indicating to the lm32 model that an external termination is active. If no termination, a wakeup time of the current time plus `LM32_INTERRUPT_GRANULARITY` (1000 cycles in this case), is returned. This means that the tick function will not be called for at least 1000 cycles (plus a bit to, say, complete an instruction). This *might* be set to 1, so it is called after every instruction, but the rate of activity for these peripherals is not that high, and would produce unnecessary processing overhead. A granularity of even higher might be possible, but 1000 seems to give little noticeable overhead in processing speed.

Peripheral Models

There are two peripheral models implemented for the lnxmico32 system: a timer and a UART. Both provide three interface functions:

```
void lm32_<peripheral>_write (const uint32_t address,
                             const uint32_t data,
                             const int      cntx = 0);

void lm32_<peripheral>_read  (const uint32_t address,
                             uint32_t* data,
                             const int      cntx = 0);

bool lm32_<peripheral>_tick (lm32_time_t time,
                             [
                               bool &terminate,
                               const bool kbd_connected = false,
                             ]
                             const int cntx = 0);
```

The two register access functions are fairly self explanatory, with an address, and either a data value passed in for writes, or a pointer passed in for returning the read value. All accesses are for 32 bit words, as all the registers for the peripherals are aligned to 32 bit word boundaries.

The tick functions all have a `time` parameter input, and a context value (`cntx`), for selecting the particular instance of the peripheral (up to 4 of each), with a default context of 0, allowing this parameter to be omitted if only one peripheral instantiated, as for the timer in `lnxmico32`. The

UART model also has a `terminate` parameter (passed by reference) and a `kbd_connected` input flag. The `terminate` parameter allows the UART to request termination of execution by setting this parameter to `true`. This done if the user types a certain sequence of characters as input to the UART, flagging indication of termination, and allowing the program to exit cleanly with any post processing requirements, such as dumping registers etc. The keyboard flag input enables internal processing of keystrokes as RX data to the UART. As only one UART can process the single keyboard, this flag enables the nominated UART to process key stroke inputs, whilst the others ignore them.

The timer model simulates the behaviour of the Lattice timer IP [4], implementing the four registers to control operation, the counter and the generation of an interrupt, when the counter reaches zero. The counter can continue or stop, depending on the register control settings.

The UART model simulates the behaviour of the Lattice UART [5], implementing the 8 registers. Transmission modelling is the much simpler functionality. When the processor writes a byte to the transmit holding register, the byte is passed straight on to `stdout`, via a `putchar()` call. However, timing is simulated, and the status bits are cleared to indicate that there is no space for another byte. The timer model's tick function notes the time when transmission status goes active, and counts until the transmission time has passed, before setting the status back to allow further transmission. The time calculated is a function of the configured BAUD rate and the clock frequency, and assumes a start, parity and stop bit, on top of 8 bits of data (i.e. 11 bits). When the status bits indicate that the TX buffer is empty once more, if enabled, the model generates an interrupt, which is returned as a `true` status when the tick function exits.

For keyboard input, each time the tick function is called (and the `kbd_connected` flag is set), the model checks if a key has been pressed, and fetches the byte value if it has. This is placed in the RBR register and the "data received" status set. If interrupts are enabled for a data reception, a interrupt status of `true` is returned by the tick function. As well as simulating keyboard input for the system, the model also monitors the input to detect for a specific sequence of key strokes. If the input sequence matches a particular string ("`#!exit!<enter>`"), it sets, to `true`, the `terminate` parameter passed in (by reference) to the tick function call. The system model can choose to ignore this, but in `lnxmico32`, this request is passed back to the `lm32` model to terminate its execution as a user breakpoint.

Save and Restore

The `lnxmico32` program has the ability to save the state of the system on exit, and to reload that state on re-running to restore the running system to exactly where it was upon exit. In order to do this, the state of the CPU, memory and of all the peripherals must be saved and restored. The `lm32` model already has two methods to support this, as documented previously:

- `lm32_get_cpu_state()`
- `lm32_set_cpu_state()`

These two methods transfer state within a single object of type `lm32_cpu::lm32_state`. The CPU model has been designed so that it keeps all relevant internal state within a structure of this type, and it is returned or set as a single structure, so that the calling program need not know what state to save, or even its details. The returned state can be pointed to by a pointer to bytes, and saved as a byte stream to the size of the type. The CPU state does not contain the contents of the memory, as this can be large and needs to be handled externally, using the read/write methods of the CPU model, for optimal handling (more below).

The UART and timer peripherals of `lnxmico32` have followed the same practice as the CPU model, and have single structures containing all the relevant state (with defined, and exported,

types `lm32_uart_state_t` and `lm32_utimer_state_t`), and methods for retrieving them and restoring them:

- `lm32_get_timer_state()`
- `lm32_set_timer_state()`
- `lm32_get_uart_state()`
- `lm32_set_uart_state()`

The state returned by the peripherals contains the state for all the contexts that the peripheral code supports, and not just a single context whether the context has been used or not. This simplifies the save and restore, and makes the returned data size fixed in all cases. The amount of data is small (compared to the memory image), and so has little overhead. Since the data is fixed size, it is saved completely raw, with no additional information, such as size or target peripheral. The format of the `.sav` file has a fixed order of data (see below), so that on restore it is known what data is expected next, and its size inferred.

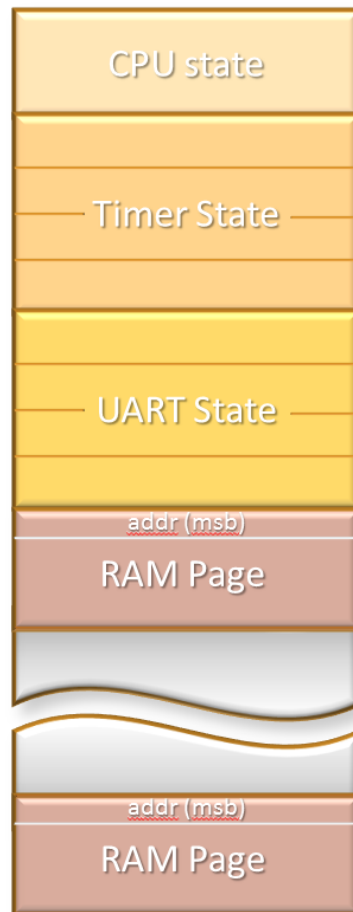
Saving of memory

The memory of the `lnxmico32` program is fairly large at 64MB. A large proportion of this contains the invariant boot and OS programs, and the filesystem ROM image. As these can be reloaded on restore runs, the data in these locations need not to be saved. To make saving of data more efficient, the memory callback function, which is called for all CPU memory accesses, monitors for any write access to the RAM. The RAM, for this purpose, is divided up into 1K pages and a tag kept on each page, setting a flag to true if a write access is made on that page. The size of the page is a compromise between granularity of save data and the size of the tag array. This can be altered without change the function's integrity.

When a save operation is performed, only the pages in RAM that have had a write access are saved. Since the tag array is cleared after that OS and FS are loaded, only subsequent writes are recorded and the pages saved. Since the page sizes are fixed, only the address of the page needs to be pre-pended to the data. This is saved first as four bytes, with MSB format. By forcing this format, the data is independent of the host, and its byte ordering, on which it is run. The data, then, is saved as a set of consecutive addresses and 1K binary data images.

The File Format

As mentioned above, the peripheral data is fixed size. Also, the CPU state comes as a fixed size object. The RAM data is a dynamic set of pages, which will vary from save to save. The file format was chosen to place all the fixed sized data first, and then followed by the RAM data, avoiding the need to delimit the fixed sized data, though the order becomes fixed. The format of the `.sav` file is as shown below:



Control

By default the state is saved to a file `lnxmico32.sav`. This can be changed to a different file name by using the `-s` command line option, or the `save_file_name` parameter in the `[state]` section of the `.ini` file. Saving of state is enabled with the `-S` command line option or `save_state`, and loading of previously saved state is enabled with the `-L` option or `load_state`.

When both saving and restoring are enabled, the affects are accumulative. That is, each load will re-mark the RAM pages that are restored to, with new pages accessed added on top of this, and so on. This is needed as there is no guarantee that all previous pages will be accessed on each new run.

Performance

Performance measures were made using the Linux system model, as this is sufficiently complicated, and representative of a real system, as to yield meaningful results. The system was tested for all supported platforms (as documented previously), with the addition of MSVC Community 2015.

The platform used was an Intel[®] i7 920 CPU, running at 2.67GHz, with a system having 6GB RAM on an ASUS P6T SE Motherboard. The test was to run `lnxmico32 -I`, boot Linux, login as root and exit, which yielded a test of > 400 Million instructions.

Compilation for the Microsoft Visual C environment was all done with the 'Release' mode. For the `gcc` compilations, optimisation options '`-Ofast -fomit-frame-pointer -march=native`' were used.

The results are summarised in the table below:

OS	Compiler	LM32_FAST_COMPILE	Unmodified
Windows 10	MSVC Express 2010	34.2 MIPS	22.6 MIPS
	MSVC Community 2015 x86	31.4 MIPS	21.2 MIPS
	MSVC Community 2015 x64	38.8 MIPS	26.3 MIPS
Cygwin 2.5.2 32 Bit	gcc v5.4.0 (-m32)	45.4 MIPS	28.8 MIPS
Ubuntu 16.04 LTS	gcc v5.4.0 (-m64)	43.8 MIPS	30.2 MIPS
	gcc v5.4.0 (-m32)	30.5 MIPS	19.8 MIPS

The surprising result here is that the Cygwin 32 bit compilation comes out on top, rather than the native 64 bit Linux system (Ubuntu)—though this situation is reversed when not compiled with `LM32_FAST_COMPILE`. Since Cygwin is a 32 bit compiler, and Ubuntu is 64 bit, additional differences may arise from this, though the 32 bit Ubuntu compile was the worst of all. However, compiling for 64 bits in MSVC improved the situation over 32 bits, and the same might be expected for GCC. Unfortunately 64 bit Cygwin was not available at the time of testing, and so a complete analysis has not been done for these differences.

So, focussing on the best result, the system runs at around 45 MIPS, when compiled with `LM32_FAST_COMPILE`, which yields a 57.6% improvement over the fully featured model. The speed of the model will very much depend on the nature of the code being executed, and the profile of the particular instructions, and so the results documented here are only a rough guide of 'best' performances for a limited, though not contrived, test.

Multi-processor System Modelling

In this section is discussed the method for constructing a system model with multiple instantiations of the CPU model, to create a multi-processor system. Note that, to date, the model has not been tested extensively in this manner, but the model is designed to be able to support this, and the recommended method is described here.

Running Models Concurrently

In the case study described in the previous section, a single model is instantiated, and is run by calling the model's `lm32_run_program()` method with an `exec_type` argument set to a value of `LM32_RUN_FROM_RESET`. This means that the CPU model will loop internally, executing instructions indefinitely, until such time as the registered external interrupt callback functions signal for a termination (returning a negative wakeup time), when the model will return from the function call. Using this method when requiring multiple CPU models to be running concurrently would require having each model running in a separate thread, with all the complexities that that would entail. Fortunately this is not required.

The model has two other execution types that can be used to enable concurrency: `LM32_RUN_SINGLE_STEP` and `LM32_RUN_TICK`. The differences between these two types is explained in the 'Execution and Breakpoints' section but, in summary, the first steps one instruction, whilst the second advances the clock by one tick (which may, or may not execute an instruction, if waiting for the last to complete). The single stepping is the simplest and quickest way to advance the model but requires time synchronisation between the models (more below), whereas the ticking advances a known time (one cycle), but the models will need calling more frequently. Note that ticking assumes that the timing model is enabled within the model. When compiled with `LM32_FAST_COMPILE`, for instance, the timing model is disabled, and a tick call and an instruction call are one and the same thing. In this case `LM32_RUN_SINGLE_STEP` should be used, as the clock state is not updated.

When calling the run methods with either the step or tick execution type, the model will return after just one instruction or clock cycle. When multiple models are instantiated, these can be called within an external loop, one after the other, with either a step or tick execution type (don't mix the type between models though). Termination of this external loop is up to the implementer, but the status returned by the calls to the model can be inspected, and breaking the loop could be based on one or all of them requesting termination.

Synchronising Time

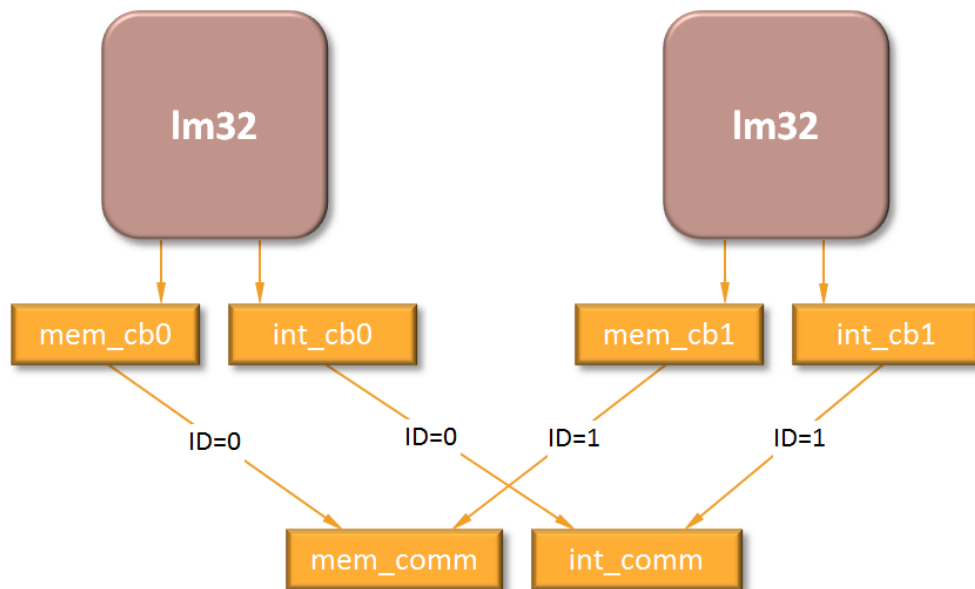
Running the models with an active timing model, but using stepped execution type (for speed), can cause drift in time between models if this is not managed. This is because instructions take different times to execute, and unless the models are all running identical programs, the state of their clocks will advance differently. In this case, the external program's loop needs to inspect time, and run the models appropriately.

The CPU model has a method to inspect time: `lm32_get_current_time()`. On the first loop, all model are stepped, and their time inspected. On the next iteration, the CPU with the earliest time is the one to be run, and any others that have this same earliest time. Any with a time in advance of this are not run. This continues on all subsequent iterations, until termination. This ensures that the CPUs are never more than a few cycles adrift, and keeps them in sync, whilst allowing reduction in the number of calls to the models from a pure ticking model. Whether a ticking model is better than a stepping and synchronising one is a matter of circumstances and preference.

Shared Callbacks

Any system model using the Im32 model will have to have memory and interrupt callbacks registered in order to model external peripherals etc., as for the Linux case study described earlier. If a multi-processor system is to have completely different functionality implemented in each of the callback methods, then different functions can be registered for each of the instantiated CPUs. However, if the CPUs have shared functionality, such as modelling connection to a shared bus, with shared access to memory and peripherals in a common address space, then something else needs to be done.

The model does not return an ID when calling its registered callback functions, and so a method is needed to identify which CPU is making the call, if the callback code is to be shared. The idea here is to have the common code in a separate function, not registered with the models, but with an addition of an ID parameter. Individual 'wrapper' callback functions are registered with each separate CPU which simply calls the common code, with the addition of the ID for the particular processor. It is then up to the common code to keep separate contexts for each processor, where necessary, or access common state. The diagram below illustrates this concept:



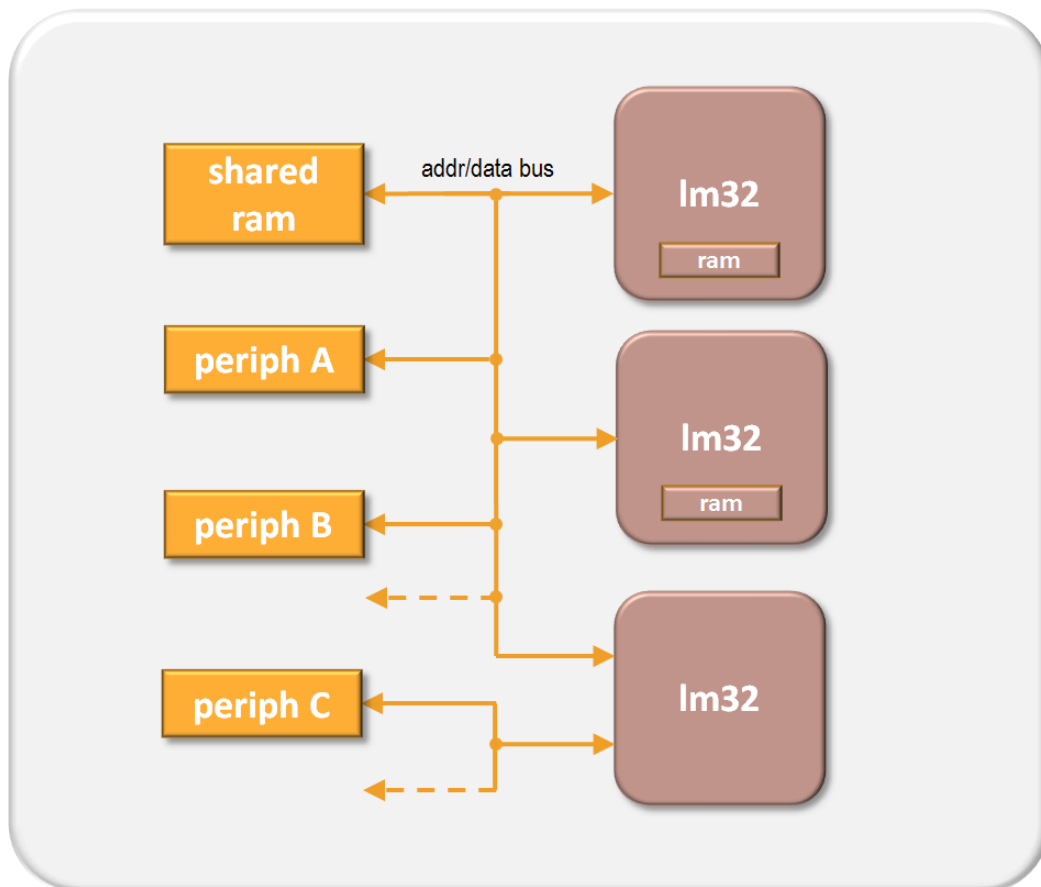
Shared Memory Space

As mentioned in previous sections, the model can have an internal memory, with controllable size, and memory space offset. This memory is separate for each CPU instantiation. The internal memory can be configured to be all, or partly, removed, and the memory callback functions trap and process memory addresses across all or part of the modelled space. To share an address space, whether mapped to memory, or to peripheral registers, the external memory callback functionality processes these addresses, and models the functionality.

For shared space, the external model common callback code must access the common state, regardless of the ID passed in. If modelling state unique to each processor, it must switch context based on the ID. One can also envisage state that is common between a subset of processors, but separate to another subset.

A similar sharing of interrupt sources can be envisaged for the interrupt callback as well, where some external interrupts raise an interrupt pin on all the CPUs (for a broadcast mailbox function, say), or be CPU specific (for a private peripheral, for example).

In this manner, it is possible to set up any arbitrary system of shared or particular memory and peripheral set between multiple CPU instantiations. Internal RAM can be used for private memory (modelling data and/or instruction TCM, say), with external callbacks mapping shared memory and peripherals to the appropriate CPUs. The CPU's program code can be located either privately in internal memory, or be common to multiple instantiations, as required. Such an example system is illustrated in the diagram below:



In this example three processors have access to shared memory and peripherals A and B (and any other on this bus). The third processor also has access to peripheral C, but has no internal memory. Thus, access to shared memory and peripherals A and B are handled by the common callback code, without regard to ID. All memory accesses from the third processor must be handled by the memory callback (as it has no internal memory), and peripheral C is only accessed when receiving an ID for the third processor, otherwise the access is marked as unprocessed. Not shown, but similarly for interrupts, it can be envisaged that one or more of the peripherals on the common bus could interrupt one or more processors as necessary. Peripheral C, however, would only interrupt the bottom processor. So two memory busses, and two interrupt busses are modelled, and can be extended at will.

Further Reading

- [1] *LatticeMico32 Processor Reference Manual*, Lattice Semiconductors, June 2012
- [2] *Using as, the GNU Assembler*, version 2.19.51, Free Software Foundation, 2009
- [3] *Linux Port to LatticeMico32 System Reference Guide*, 2008, Lattice Semiconductors
- [4] *LatticeMico Timer*, version 3.1, 2012, Lattice Semiconductors
- [5] *LatticeMico UART*, version 3.8, 2012, Lattice Semiconductors
- [6] Stallman et. al., *Debugging with GDB*, 10th Edition, Free Software Foundation, 2012