Reference Manual
for the *rv32_cpu* RISC-V soft CPU
Instruction Set Simulator

(version 0.5 draft)

Simon Southwell

July 2021

Copyright

Disclaimers

Simon Southwell (simon@anita-simulators.org.uk)

Cambridge, UK, July 2021

Introduction

The document details the design and use of an open-source RISC-V instructions set simulator, `rvs32_cpu`. It is meant as an exercise in modelling a modern RISC based processor with the aim to demonstrate the operations of such a device, without the complexities of a hardware implementation. The code itself is also designed to be accessible and expandable, as an education tool and a starting point for customisation. As such, this document, as well as usage information, will give some details of the internal design of the source code sufficient to navigate and modify the code, as desired. At this time it's fixed at implementing the RV32 features, with RV64 being a possible future expansion.

The model trades off between performance and clarity. It was designed to be an efficient model running at many Mips on a modern typical performance machine, but where clarity of operation would be unnecessarily obfuscated, a more practical route is taken. In addition, the use of third party libraries (e.g. boost) is avoided, as is some more esoteric features of the latest C++ language specification. Even use of the STL is kept to a minimum. It is aimed to be supported for compilation on both windows and Linux platforms, and support for Visual Studio and Eclipse is provided. The model can be used without the need to understand the code, but if one desires to delve deeper, and make modifications, then the intent is that the code is as easy to understand as possible, consistent with implementing the desired features. A 32 bit RISC ISS for the Lattice Mico32 soft core processor is already published [3], and this implementation borrows much from this, whilst extending and improving on some of the ideas in that architecture.

The project is a prelude to an open source soft core logic implementation targeted at FPGA. This, too, will be aimed at an educational and accessible implementation, rather than a high performance, low footprint and low power solution, whilst still implementing useful embedded soft core processor.

Basic Features

The current model has the following basic features:

- RV32I ISA model
  - Support for RV32E via compile option
- Zicsr extensions, with CSR instructions and registers
- RV32M extensions
- Single HART
- Only Machine (M) privilege currently mode supported
- Trap handling
- Cycle count and real-time clock
- Interrupt handling
  - External interrupts
  - Timer interrupts
  - Software interrupts
- Basic internal memory model (16KBytes)
- External memory callback feature
- External interrupt callback feature
- Disassembler, both run-time and static
- Loading of ELF programs to memory

Class structure

The ISS is written in C++ and is structured around a base class that implements the RISC-V RV32I specification [1]. This base class can be expanded with ever succeeding derived

classes to add the features to the RISC-V expansion specifications, allowing an arbitrary mix of feature sets.

The idea is to implement the minimal RISC-V 32 bit feature set (RV32I) in this base class, with enough hooks to allow a derived class to add additional feature sets that match the expansion specifications (e.g. M, Zcsr, F etc.), each with its own derived class that inherits the features from the previously implemented classes to make up a valid RISC-V implementation model. The diagram below summarises this intended structure, indicating what has been implemented, what is planned and possible future expansions

	rv32i_cpu	Base (I plus implied Zifencei)
Future Expansions	rv32csr_cpu	Zicsr
rv32c_cpu		Example bypass of feature
	rv32m_cpu	M (multiply and divide)
rv32usr_cpu		Current_status
	rv32a_cpu	A (atomic operations)
rv32sup_cpu	rv32f_cpu	F (single precision float)
rv64x_cpu		
	rv32d_cpu	D (double precision float)
	rv32_cpu	Top level implementation class
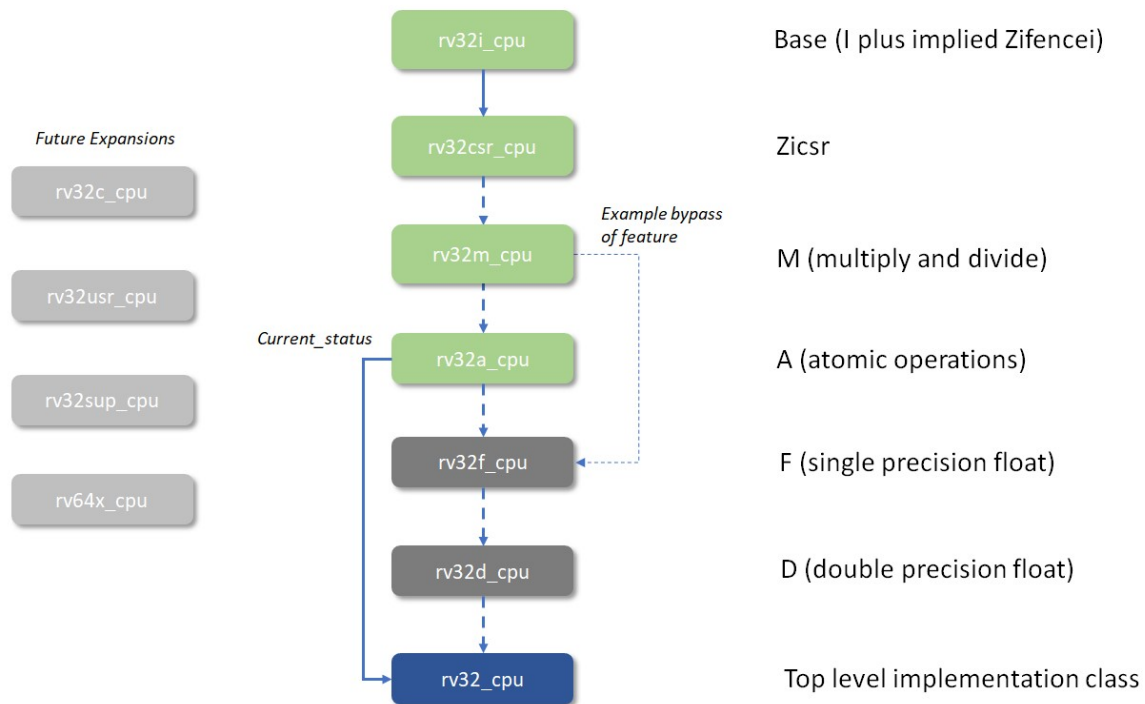
The diagram shows, in green, the currently implemented classes. The rv32i_cpu class is the base class implementing the RV32I specification. In addition it also has the Zifenci features implied, as the single HART implementation and the nature of the ISS operation means all FENCE instructions are nops. Synchronous traps are also implemented in the base class, such as misaligned addresses or system calls. A basic memory model is included in the base class, but an external memory access callback feature allows for expansion to a more complex model by attaching an external memory software model. For instance, the memory model used in a PCI express root complex model, giving a full 64 bit memory availability ([4] Chapter *Internal Memory Structure*).

Also included is a derived class, rv32_csr, which implements the CSR instructions and manages the CSR register updates. The actual CSR address space is modelled in the base class, but is not accessible externally. This will make it easier to add more expanded features that update and use the CSR registers that are not currently implemented. The expansion class uses the, now visible, CSR features to then implement the asynchronous interrupt features of external interrupts, timer interrupt and software interrupt. External interrupts become available by a callback feature which allows external code to set interrupts status, and the possibility of attaching a model of a debug module

Another derived class, rv32m_cpu, implements the RV32M extensions for integer multiply and divide. This is derived from the rv32_csr class, and add 8 more instruction functions to the mix, as updates the decode tables to include them.

As yet unimplemented classes are planned to implement A, F and D expansion features, bringing the model to the RV32G standard. The diagram shows a possible progression of inheritance from the CSR class in a linear sequence. However, these do not have to be added in that order, and features might be skipped. The diagram shows a route where the atomic operations are skipped, giving an RV32MFD,Zcsr,Zfencei implementation. The architecture is expecting a linear inheritance progression though, with the final derived class being that which the top level class inherits.

A top level class, `rv32_cpu`, is always the object instantiated by external code. The chain of expansion classes is inherited by this top class, wherever it terminates, with the top level inheriting the final derived class in the chain. This allows the model's specification to be changed without altering any instantiations of the model in external code. It is also expected that in this class any custom features required to be added should be implemented.

The other features being considered, but not yet planned, are shown in the diagram. These include compressed code (RV32C), User and Supervisor privilege levels, and RV64x features. As much as possible, these features are considered in the current model's design, but may require some refactoring to allow expansion whilst maintaining the class expansion hierarchy.

API

This section describes the application programming interface for the model. Different parts of the API are defined in the different classes, and these will be detailed here, along with any methods overridden by higher classes

Base Class

The following table is a list of the API methods available from the base class:

Function	Method prototype
Constructor	`rv32i_cpu(FILE* dbgfp = stdout)`
Program load	`void read_elf(const char* const filename)`
Read memory access	`uint32_t read_mem (const uint32_t byte_addr, const int type)`
Write memory access	`void write_mem (const uint32_t byte_addr, const uint32_t data, const int type)`
Reset	`void reset_cpu (void)`
Register access	`uint32_t regi_val (uint32_t reg_idx)`
PC access	`uint32_t pc_val ()`
Register memory callback function	`void register_ext_mem_callback (p_rv32i_memcallback_t callback_func)`
Program execution	`int run(const rv32i_cfg_s &cfg)`

The aim, as much as is possible, is to be able to run a program with as little configuration as is possible. To work 'out of the box', as it were. To this end the constructor has only one argument—a file pointer for redirecting debug data to a file. This defaults to `stdout`, and so need not be given.

To load a program, the run method needs just the filename of a valid RISC-V ELF executable. By default, this will be loaded into the internal memory of the model. If a memory callback function has been registered, the callback will still be executed to give the function a chance to process this data. It is passed in with a type of `MEM_WR_ACCESS_INSTR` so can be treated differently from a normal write if required. The callback may chose not to process it, and the data will then be processed internally.

Direct access to memory is provided by `read_mem` and `write_mem` methods. Access via these methods will also be passed to any registered memory callback function, but with the passed in type. The CPU can also be reset with the `reset_cpu` method, which sets the PC to a fixed vector of `RV32I_RESET_VECTOR`. Calling this method is equivalent to the reset pin and does not initialise the model's internal state not involved with the CPU itself, such as the disassemble active state. This kind of state is initialised at construction.

External Memory Callback

To register an external memory access callback function the `register_ext_mem_callback` method is called, passing in a pointer to a function of type `p_rv32i_memcallback_t`. The is a function whose prototype is:

```
int func (const uint32_t    byte_addr,
               uint32_t    &data,
         const int          type,
         const rv32i_time_t time);
```

When registered, the function will be called whenever an access to the memory address space is made. The access byte address is passed in, along with a reference to a data word, and a and access type. In addition a time is passed in (the cycle count since reset). If a write access, then the data argument will contain the data to be written. If a read type, then the data should be updated with the read value. On return, the callback is expected to return either a count (in cycles) that it modelled the access to take (which can be 0), or `RV32I_EXT_MEM_NOT_PROCESSED`, to indicate that the access did not match an address space it was modelling, or that that was some other fault with the access. The time argument can be used for calculating any wait states for the returned access. When the callback indicates it did not process the access, the then model will attempt to process it.

Modelling memory is not the only use of the external memory callback, but as a gateway to the entire memory space of a larger system model in which the processor model resides. Models of registers in peripherals (e.g. a UART) can all be mapped via the callback, for instance, thus extending the processor model arbitrarily.

Running a program

The `run` method is used to execute a program (loaded with `read_elf`). It requires a configuration structure to be passed in of type `rv32i_cf_s`. As stated before, in order to work out of the box, a variable of this type can be constructed and passed in without modification, and the default settings will be used. The structure is defined as follows:

```
struct  rv32i_cfg_s {
    const char*     exec_fname;
    uint32_t        start_addr;
    unsigned        num_instr;
    bool            rt_dis;
    bool            dis_en;
    bool            hlt_on_inst_err;
    bool            en_brk_on_addr;
    uint32_t        brk_addr;
    FILE*           dbg_fp;
};
```

The name of an executable file to be loaded is given in `exec_fname`. This would normally be used by the `read_elf` method, but is made available to the `run` method for debug purposes. The default value is `test.exe`. The start address, from where execution will begin, is configured in `start_addr`, and defaults to 0.

Disassembly output is controlled with two flags. Normal, linear, disassembly is controlled with `dis_en`. When `true`, the model will simply run linearly from the start address to the end, disassembling the code. It defaults to `false`. The `rt_dis` flag controls run-time disassembly. When `true` the model will execute the program as normal, but output disassembled instructions as it goes. It defaults to false, and is also overridden by `dis_en` being true. An example output fragment is shown below.

```
                         ⋮
00000128: 0x00100513    addi      a0, zero, 1
0000012c: 0x01f51513    slli      a0, a0, 31
00000130: 0x00054c63    blt       a0, zero, 24
      *
00000148: 0x00001297    auipc     t0, 0x00001000
0000014c: 0x6e42a283    lw        t0, 1764(t0)
00000150: 0x00028a63    beq       t0, zero, 20
      *
00000164: 0x30005073    csrrwi    zero, 0x300, 0
00000168: 0x00001297    auipc     t0, 0x00001000
0000016c: 0x6bc2a283    lw        t0, 1724(t0)
00000170: 0x34129073    csrrw     zero, 0x341, t0
00000174: 0xf1402573    csrrs     a0, 0xf14, zero
00000178: 0x30200073    mret
      *
0000017c: 0x800000b7    lui       ra, 0x80000000
00000180: 0x00000113    addi      sp, zero, 0
                         ⋮
```

A couple of controls are provided to halt execution on certain conditions. The halt_on_instr_err, when true, flag will break execution and return from run if a reserved or unimplemented instruction is executed (the default is false). The en_brk_on_addr flag, when true, will break execution when the PC reaches the address specified in brk_addr (default false).

The dbg_fp is a pointer of type FILE*. By default this is stdout, but can be set to a valid file pointer that's been opened for writing. This will redirect any debug information (e.g. disassembled output) to the specified file. This would normally be used by the rv32i_cpu constructor.

Zicsr Class

The class implementing the Zicsr extensions (rv32csr_cpu) extends the API with a single additional method.

```
void register_int_callback (p_rv32i_intcallback_t callback_func)
```

This is used to register a callback for allowing external interrupts to be generated. If no callback is registered, then no external interrupts are possible. If a function is to be registered it must have the type p_rv32i_intcallback_t. That is it must have a prototype of:

```
uint32_t func (const rv32i_time_t time, rv32i_time_t *wakeup_time);
```

The callback function has a time (in cycle counts) passed in, and a pointer to a wakeup time for returning a scheduled time for the next call to the function. This can be 0 for calling every cycle. If wishing to add a delay, then the passed in time (the current time) is used plus any delay required until the next execution. The function returns the interrupt state. This is either 0 if no interrupt being generated, or a non-zero value for interrupt active. The RISC-V has a single external interrupt input, and uses external interrupt controllers to arbitrate between multiple interrupts. The interrupt callback function feature allows the model to be extended to add models of such controllers to be added.

Internal Architecture

The internal architecture of the ISS follows a conventional pattern. Within the run method (the entry point from external code) it will loop performing a fetch decode and execute stage, using internal methods `fetch_instruction`, `primary_decode` and `execute` respectively. What is, perhaps, less conventional is that the decoding to the functionality modelling the instructions is done by indexing into a set of tables that (ultimately) lead to a pointer to a function (a method of the class) which is then called with decoded information. Whether this is a 'better' solution than a more traditional switch statement is open to debate, but the code, I believe, is less verbose and more understandable, with decoding a simple indexing into tables, and execution doing the same things for every instruction—i.e. calling the function using the pointer recovered. This method was successfully deployed in the lm32_cpu instruction set simulator [3].

Base Class

The base class (`rv32i_cpu`) contains all of the fetch, decode and execution code. In addition it has individual methods for each of the RV32I instructions to perform the function of these instructions. It contains the CPU state in a single `state` structure (`rv32i_state` class), which contains a single hart (indexed by `curr_hart` member variable) expandable to multiple harts, and a 4096 entry CSR table, along with the current privilege level—though this is always at machine privilege for this class.

The hart member of `state`, itself, is a class of type `rv32i_hart_state`, which contains an array of words for the register file (`x`) and the program counter (`pc`).

Even though the CSR and privilege level state is not used in the base class, since CSR instructions not supported and only machine level privilege, by defining the entire state here, adding save and restore functionality becomes that much easier in the future, and expansion classes need only manipulate the base class state, and not add to it, which would make arbitrary expansion inclusion more complicated in dependencies.

Instruction Fetch

The base class has a virtual method, `fetch_instruction`, which simply uses an internal method (`read_mem`) to read a 32 bit word from memory. Although this is a single line method, it is encapsulated as a virtual method to allow it to be overridden by a child class. In particular, if wanting to add support for compressed instructions (RV32C), the method can be replaced to handle the mix of 16 bit and 32 bit instruction fetches.

A companion virtual method is the `increment_pc` method which, in the base class, simply in increments the PC by 4. Again, for RV32C support, this may be overridden to manage the PC increments depending on the instruction type. The `read_mem` method checks for alignment and access faults, and this, too, is a virtual function that can be upgraded to check alignments errors for 16 bit instruction accesses.

Decode

Decoding is done with the `primary_decode` method and a set of tables to return a pointer to a function, which is the method to implement the instruction.

The first part of decoding is extracting the sub-fields of the instruction. The philosophy taken here is to decode all possible fields of the instruction upfront, populating a `rv32i_decode_t` structure, a reference of which is passed into `primary_decode()`. This has entries for all the immediate types, expanded to 32 bits, sign extended as required, as well as the `rs1`, `rs2`, and `rd` indexes. The `funct3` and `funct7` values are also extracted, and a copy of the instruction

value itself (to aid debug). It also has a field for a copy of the decode table entry—again to aid debug.

The reason for decoding all possible fields, without discerning which are needed for a particular instruction is two-fold. First it means the decode structure is common to all instruction methods, allowing the table structure and function point method to be implemented, and because this is how the first state of a logic decode implementation is likely to be constructed. This model is a prelude to a logic implementation, and will be used to verify and debug the hardware. Constructing it in such a way as to cross-check with logic will aid in its speedy delivery.

The `primary_decode` method returns a point to a decode table entry of type `rv32_decode_table_t`. This structure has three fields:

- `sub_table`: a boolean flag to indicate the entry is either a pointer to another decode table or an instruction entry
- `p`: a pointer to an instruction function, which is NULL if `sub_table` is true, else a pointer to the instruction method.
- `ref`: a union between a decoded table entry, when `sub_table` false, and a pointer to another table, when `sub_table` true.

The `sub_table` flag is invariant in its function and allows a decode table structure to be implemented with varying levels, mixing pointers and instruction entries. The RISC-V instruction set is decoded with various fields depending on the instruction type ([1] Ch. 2, sec 2.3). For U-type and J-type only the 7 bit opcode field is used, and the bottom two bits are always set for 32 bit instructions. For the I-, S- and B-type instructions a 3 bit field (`funct3`) sub-divides the decode further. The R-type instructions decode even further with a 7 bit field (`funct7`). The system instructions, potentially expand the `funct7` bits to 12 (though only a few bits are used). With this in mind the decode structure has primary, secondary and tertiary decode tables. An example structure is shown below.
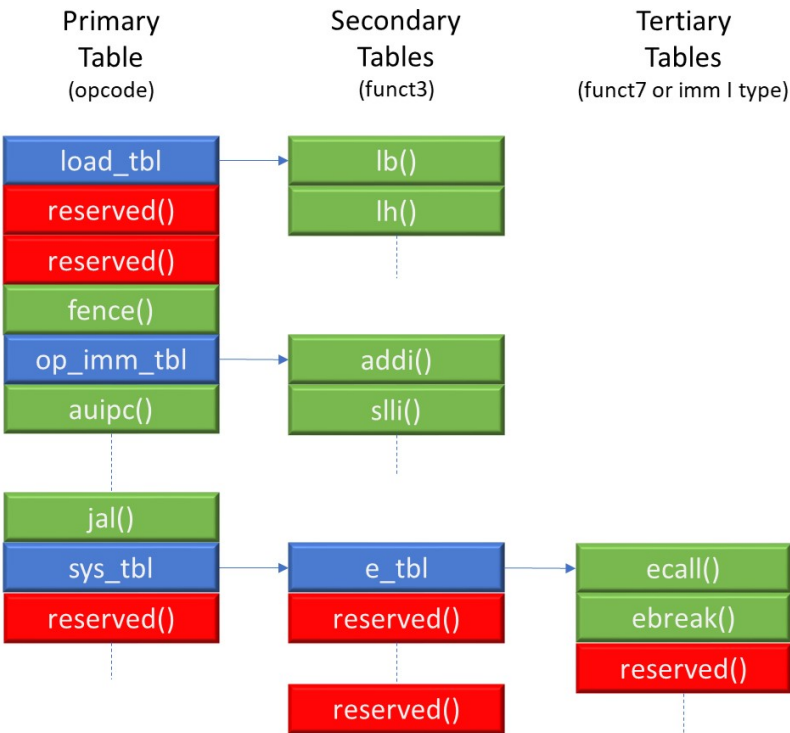
The primary decode table (`primary_tbl`) entries hold the decode of the instruction opcode field. This table is 32 entries long in the base class, as the bottom two bits are always 11b for 32 bit instructions, leaving 5 bits to decode. Where no further decoding is required, such as for the JAL and AUIPC instruction, the table entry holds an instruction entry (shown in green), with the instruction information and the pointer to the instruction's method. If no valid instruction exists at that decode point, it still has an entry but points to a reserved instruction method to handle that situation (shown in red), but the decoding is unaware of this. If further decoding is required, then the entry is a pointer to another decode table (shown in blue), with `sub_table` set true. From the primary table, a secondary table exists for each entry that requires a `funct3` decode. These tables have exactly the same format as the primary table, and so can be a mix of instruction entries, or pointers to tertiary tables. The tertiary tables decode on `funct7` (or the top 12 bits, depending), and should have no sub-tables (which is checked for in `primary_decode()`).

When an instruction entry is reached, the `primary_decode` method returns the instruction table entry for passing to `execute()`.

The decode tables are initialised in the base class's constructor, filling in the tertiary, secondary and finally primary tables. Where instructions are not implemented in the base class, the tables are filled with pointers to the `reserved` instruction method. Child class' constructors can then override this adding new instruction methods for those implemented by that class, allowing expansion of the base functionality. The base class constructor will be called first, so the derived classes will override the reserved instruction entries correctly.

Execution

The `execute` method takes both the decoded instruction data, with the instruction field information, and the decode data entry, containing the pointer to an instruction method, as inputs.

Before acting upon the decoded data, it checks for any interrupt status by calling the `process_interrupts` method. For the base class, this does nothing, but expansion classes can add interrupt functionality and the method returns non-zero to indicate an interrupt is active, in which case `execute` returns without executing the decoded function, though not with an error status.

If no interrupts are outstanding, then, if enabled, a check is made whether an unimplemented instruction was executed and `execute` returns with an error status. This allows external control of execution, breaking on an unimplemented instruction.

Finally, if no interrupts and no breaking on unimplemented instructions, the instruction function pointer is used to call the method for the decoded instruction, passing in the instruction decode information structure.

Instructions

The instruction methods all have the same prototype:

```
void rv32i_cpu::<instruction>(const p_rv32i_decode_t d)
```

Just the instruction decoded information is passed in, allowing the method to extract what values it needs, without further decoding. The method all follow a basic execution pattern as well, with a disassembly macro call, appropriate to the instruction format type, followed by the actual instruction execution code, and a call to `increment_pc()`—except for jump and branch instructions. For branch, the PC increment is only called if the branch not taken. For the jump instructions it is not called. Similarly the system instructions do not increment the PC.

The instruction decode methods are simple and only a few lines of code. The bulk of the work was done in the decode, and the actual instructions are easily executed with few lines—after all, we are running on a computer with very similar operations implemented in its CPU. A couple of example instruction methods, for BEQ (branch if equal) and R-type XOR instructions, are shown below:

```cpp
void rv32i_cpu::beq(const p_rv32i_decode_t d)
{
    RV32I_DISASSEM_B_TYPE(d->instr, d->entry.instr_name, d->rs1, d->rs2, d->imm_b);

    if (!disassemble && state.hart[curr_hart].x[d->rs1] ==
                        state.hart[curr_hart].x[d->rs2])
    {
        state.hart[curr_hart].pc += d->imm_b;
        RV32I_DISASSEM_PC_JUMP;
    }
    else
    {
        increment_pc();
    }
}
```

```cpp
void rv32i_cpu::xorr(const p_rv32i_decode_t d)
{
    RV32I_DISASSEM_R_TYPE(d->instr, d->entry.instr_name, d->rd, d->rs1, d->rs2);

    if (d->rd)
    {
        state.hart[curr_hart].x[d->rd] = state.hart[curr_hart].x[d->rs1] ^
                                         state.hart[curr_hart].x[d->rs2];
    }

    increment_pc();
}
```

Note that the branch instruction only executes the instruction code if not disassembling. This is because, when doing linear disassembling, we don't want to take the jumps. Normally the functions are allowed to continue regardless of disassembly mode, to avoid adding unnecessary code, but this can't be done for branch, jump or system calls.

Traps

In the base class implementation, traps are caused by only a limited set of conditions, all of which are classed as synchronous.

- System instructions
    - `ecall`
    - `ebreak`
- Memory access faults
    - Instruction fetch misalignments
    - Load misalignments
    - Store misalignments

- Access faults
    - Load access faults
    - Store Access faults

Since the base class does not implement the CSR registers, and thus has no `mtvec` to redirect to a configured address on a trap, a trap will cause a jump to a fixed address at defined as `RV32I_FIXED_MTVEC_ADDR`, specified in `rv32i_cpu_hdr.h`.

The trap detection is distributed amongst the internal memory access methods (`readmem()`, `writemem()`) and the implemented system instruction methods (`ecall()` and `ebreak()`). All call the `process_trap()` method, with the relevant trap type passed in. As mentioned before, the process trap method is a virtual function, and can be overloaded to implement the more sophisticated processing using the CSR registers for exception handling.

Internal Memory

The base class has a simple internal memory. It is a byte array, large enough to hold `RV32I_INT_MEM_WORDS` words (or four times as many bytes). Its main purpose is to allow basic testing of the model without the need to implement an external memory system, hooked in via the external memory access callback function. It is located from a base address of 0.

It may still be used as memory even if a callback is defined, as if the callback returns an access as unprocessed it is 'offered' to the memory. At this time, the internal memory can't be relocated in the address map.

Other Important Internal State

Of the remaining state held in the base class a couple are worth mentioning. Although the use of a cycle count is not visible to running software until the Zicsr functionality is added, a `cycle_count` member variable is defined, and is maintained. Firstly it is incremented the `execute` method after each call to an instruction method. Secondly, if a memory access callback is defined and is called with the cycle count passed in, and any returned delay value (i.e. wait states) is added to this value.

The real time clock value (`mtime`) and the time compare register (`mtimecmp`) are *not* in CSR registers (see [2] sec 3.1.10), but memory mapped to the address space. So the base class defines an internal member variables, `mtimecmp`. If an access to memory access either of the two words mapped from `RV32I_RTCLOCK_CMP_ADDRESS`, then it will read or update the 32 bit half of the 64 bit value as appropriate. Similarly at `RV32I_RTCLOCK_ADDRESS`, and the next word, if read, then the actual real time is returned in units of 1µs. Writing to these locations have no effect. Within the base class, no interrupt is generated on the time compare, this feature being added by the `rvcsr_cpu` derived class (see next section). The locations of the time values is fixed in memory, and if a memory callback processes these location then it will effectively mask the functionality.

Zicsr Class

The rv32_csr class is the first extension class, implementing the Zicsr extension specification, with the CSR access instructions and implementation of CSR register functionality ([1] Ch. 9, [2] Ch. 2, 3). This class has the following features (added to the `rv32i_cpu` class features):

- Adds functionality to the CSR instructions
    - Implementation of methods `csrrw`, `csrrs`, `csrrc`, `csrwi`, `csrrsi` and `csrrci`
    - Constructor updates base class's secondary decode table, `sys_tbl`, with new CSR instructions.
- Extends system instructions
    - `access_scr` method added to add updates to CSR registers on trap

- o Adds `mret` instruction method
- o Constructor updates base class's tertiary decode table, `e_tbl`, with new system instructions, and adds to `sys_tbl`.
- Adds CSR registers:
  - o `mvendor`: implemented, fixed at 0
  - o `marchid`: implemented, fixed at 0
  - o `mimpid`: implemented, fixed at 0
  - o `mstatus`: implemented
  - o `mhartid`: implemented, fixed at 0 (only one HART supported)
  - o `misa`: implemented: set correctly, not writable
  - o `medeleg`: not implemented
  - o `mideleg`: not implemented
  - o `mie`: implemented
  - o `mtvec`: implemented
  - o `mcounteren`: implemented
  - o `mscratch`: implemented
  - o `mepc`: implemented
  - o `mcause`: implemented
  - o `mtval`: implemented
  - o `mip`: implemented
  - o `mpmpcfgx`: not implemented
  - o `mpmpaddrx`: not implemented
  - o `mcycle, mcycleh`: implemented
  - o `minstr, minstrh`: Register implemented, but counter not implemented
  - o `mhpmcounterx, mhpmcounterxh`: Register implemented, but counter not implemented
  - o `mcountinhibit`: implemented, fixed at 0
  - o `mhpmeventx`: not implemented
- Overloads the base class `reset()` method to update CSR registers on a reset.
- Overloads the base class `process_trap()` method to update CSR registers

CSR Functionality Implementation

The actual CSR register space is implemented in the base class, but the functionality is added in this class. The way this works is to access the base class register space via an `access_csr` method, and a `csr_wr_mask` method. The latter of these is a simple switch statement on a passed in address, returning a write mask for an implemented register, with an `unimp` flag returned as false, or 0 with the `unimp` flag true, if the register is not implemented.

The `access_csr` method takes several arguments—`funct3` to do some access type decode, the CSR address, the destination register index and the source register index/immediate value (depending on access type). The method makes some initial checks for sufficient privilege level (this implementation is always privileged, set at machine mode), before calling `csr_wr_mask`. If the destination register isn't `x0`, then it will load this register with the contents of the CSR register being accessed (if implemented).

If the destination is writable and implemented, then it will update the CSR register, masking with the mask returned by `csr_wr_mask`. For the case of set and clear instructions, the source register/immediate value must be non-zero to be a valid write access ([1], sec. 9.1), and this is checked before the access. If any of the access criteria fail, then `process_trap()` is called with an illegal instruction type

The `access_csr` methods is called from all the CSR access instruction methods, and implements the instructions' functionality in one place, with the actual instruction methods simply accessing the disassembly macro for CRS instructions, calling the `access_csr` method, and then calling `increment_pc()`.

The `rv32csr_cpu` class extends the system instruction functionality with the `mret` instruction method. This updates the MSTATUS register's MIE and MPIE fields ([2] Sec. 3.3). It then updates the PC with the value in the MEPC CSR register.

Although the base class implements `ecall` and `ebreak` instructions, these are not overloaded by the `rv32csr_cpu` class. The base class methods make a call to its own `process_trap` method, and it is this function that is overloaded by the CSR class to implement updating the CSR registers required by the system call trap---that is the MEPC, MCAUSE and the program counter, based on MTVEC and trap cause.

Interrupts

Interrupts, that is asynchronous traps, have one of three possible sources

- External interrupt
- Timer
- Software

As mentioned before, the CSR class add the ability to have external interrupts by allowing a callback function to be registered that is called at regular intervals up to once per instruction cycle. The implemented access to a real time clock in the base class can now be compared with the `mtimecmp` memory mapped register and raise an interrupt. Finally the MSIP field of the `mip` register being written can raise a software interrupt. All these are dependent on the global interrupt enable being set (MIE in `mstatus`) and the individual enables for the three types being enabled in `mie` (MEIE, MTIE and MSIE).

All this functionality is handled in the `process_interrupts` method, which overloads the base class method, called each time the `execute` method is invoked.

RV32M Extension Class

The multiply and divide functionality is added by the `rv32m_cpu` class. Compared to the previously documented classes, this is relatively straight forward.

It has no additional API functions and just overloads the constructor. It is in the class's constructor that the decode tables are updated with the additional instructions, each of which has its own method in this class for the multiply, divide and remainder instructions defined in the specification ([1] Sec 7).

The class has no other methods or internal member variables and is, in this sense, a pure extension derived class. Since there are no traps associated with these instructions (such as divide by 0—[1] Sec 7.2, table 7.1), no overloading of the trap method is required.

RV32A Extension Class

The atomic operation functionality is added by the `rv32a_cpu` class. Since the model has (currently) only one hart, atomic operations are not strictly necessary as synchronisation of memory accesses across harts is not an issue. In addition, even if multiple harts were implemented, this would only be necessary if a memory model that allowed out of order memory accesses to occur. However, the instructions associated with the extensions alter internal state, and this is model correctly.

The AMO*xxx*.W instructions are modelled simply as read-modify-write operations, and the *aq* and *rl* bits are ignore, as they would not change the behaviour of this model. The LR.W and SC.W instructions reserve and release an area of memory, the size of which is not specified in

the RISC-V instruction set manuals [1]. The model specifies a 32 byte granularity as being typical for a cache line from a 32 bit embedded processor.

Since the *aq* and *rl* bits form part of the funct7 field of the R-type instructions, there are effectively four entries in the tertiary decode table (`amow_tbl`) for the AMO.W instructions. The `rv32a_cpu` constructor initialises the table so that for each of the funct7's four entry positions for a given instruction, the same instruction method is loaded. Since the bits are not processed by the instruction methods, then the same behaviour is required for the four possible combination of these bits.

Top Level Class

It is expected that a program wishing to use the model does not instantiate the base class or one of the extension classes directly, but a top level class, called `rv32_cpu`, provided for this purpose. This allows the supported functionality to be changed without the need to alter instantiating code.

Normally the `rv32_cpu` class will inherit the last 'ancestor' of the base class, giving it all the functionality of the chain of derived classes (see Introduction). If custom functionality is to be added, then this could be done in this class, or by inserting a custom child class in the inheritance chain.

Defining Inheritance Chain

Accompanying the `rv32_cpu` class is a header file, `rv32_extensions.h`, that define which extensions are to be added and the inheritance hierarchy of the model. An example contents of the extensions header is shown below:

```
#define RV32_I_INHERITANCE_CLASS
#define RV32_ZIFENCEI_INHERITANCE_CLASS
#define RV32_ZICSR_INHERITANCE_CLASS      rv32i_cpu
#define RV32_M_INHERITANCE_CLASS          rv32csr_cpu
#define RV32_A_INHERITANCE_CLASS          rv32m_cpu
#define RV32_F_INHERITANCE_CLASS          rv32a_cpu
#define RV32_D_INHERITANCE_CLASS          rv32f_cpu

#define RV32_G_INHERITANCE_CLASS          rv32d_cpu

//#define RV32E_EXTENSION

#define RV32_TARGET_INHERITANCE_CLASS     rv32csr_cpu

// Define the class include file definitions used here. I.e. those needed
// for the target spec. Each one defines its predecessor, as including
// headers for later derived classes causes a compile error---even when
// using forward references (needs a completed class reference).

#define  RV32CSR_INCLUDE             "rv32i_cpu.h"
#define  RV32M_INCLUDE               "rv32csr_cpu.h"
#define  RV32A_INCLUDE               "rv32m_cpu.h"
#define  RV32F_INCLUDE               "rv32a_cpu.h"
#define  RV32D_INCLUDE               "rv32d_cpu.h"

#define  RV32_TARGET_INCLUDE         "rv32m_cpu.h"
```

In this example, the first set of definitions specify, for each extension, its immediate parent class. For the base class, with implied Zifenci features, there are no parent classes, and this is

the base class. For each of the rest of the features, from Zicsr to RV32D, these is a linear progression of hierarchy. A value for the RV32G is defined to inherit the double floating point features (and assumes all the other inherited, as shown).

The target class definition, then, picks out the point in this progression that is to be implemented---in this case the RV32I and RV32Zicsr functionality (as that is all that's implemented to date). Were it to choose, say, the RV32D point, and wished to drop RV32A extensions, then the F inheritance would be changed to `rv32m_cpu`, thus skipping this extension class.

Once all the desired class inheritances are defined, the headers required for inclusion by each extension class are defined, being the class definition header of its immediate parent class. These must follow the same inheritance path of the previous definitions. In each extension class definition file, the relevant definition is used (e.g. `#include RV32CSR_INCLUDE`) to pick up the parent class definition without needing to make changes if the hierarchy is changed in `rv32_extensions.h`. This is necessary to ensure only parent classes are seen when compiling an extension, without referencing a derived class, not yet fully defined, which causes a compile error.

One final definition is for the RV32E functionality. This extension is simply to reduce the number of the registers defined for each hart, which defaults to 32, but is reduced to 16. If the `RV32_E_EXTENSION` definition is uncommented. This than compile the base class with only a set of 16 registers, and will trap if others are accessed.

Development Environment

Various development tools have been used in the construction of this model. Some are essential parts of the process, and have support as part of the package, whilst others are used as convenience utilities (of varying usefulness). This section list the tools used and any dependencies and assumptions that may have been made.

The actual model source code has be written to be, as largely as possible, to be stand alone and not to have a complex set of external dependencies. A couple of development environments have been used (see below) and supported in the package, but the code should compile with any modern c++ compiler, of that is more convenient. Support for these, though, must come from the user.

RISC-V Toolchain

In order to compile test code, a RISC-V toolchain is required. Minimally we need an assembler, though the riscv-tests use a C pre-processor as well. The Gnu Compiler Collection (gcc) supports RISC-V, and this was used in the testing of the model. In particular the riscv64-unknown-elf, with gcc version 10.1.0 toolchain. Tat the time of writing, pre-built binaries for windows exist at the following link:

> sysprogs.com/getfile/1107/risc-v-gcc10.1.0.exe

These can be installed anywhere that's convenient, though the scripts assume that the `bin` directory has been added to `PATH`.

Compiling Code

To compile code one can use either the assembler or gcc. The riscv-tests must use gcc to get the pre-processing features but, if writing one's own tests that do not use this, then the assembler is all that's required.

Using assembler

To compile an assembler program with the toolchain, to the RV32 ABI, something like the following commands can be used:

```
riscv64-unknown-elf-as –fpic –march=rv32i –aghlms=<testname>.list \
                   –o <testname>.o <testname>.S
riscv64-unknown-elf-ld <testname>.o –Ttext 0x0 –melf32lriscv \
                   –o <testname>.exe
```

With these two commands, an ELF executable is produced which is suitable for loading into the model. It specifies that the `_start` symbol is address 0, the object file is made relocatable and it uses RV32 ABI standards (rather than RV64). The assembler produces a listing, and the command line options basically turn just about everything on.

Using gcc

Compiling with gcc instead of the assembler is fairly similar, requiring options to specify the correct ABI standard

```
riscv64-unknown-elf-gcc -c -fpic -march=rv32i -mabi=ilp32 \
                   <include path options> -o <testname>.o
riscv64-unknown-elf-ld <testname>.o –Ttext 0x0 –melf32lriscv \
               -o <testname>.exe
```

The gcc command still compiles a relocatable object file, but will also need a set of –I include path options. When compiling of the riscv-tests, this might be:

```
-I.-I<TESTENVDIR>\p -I<TESTDIR>\isa\macros\scalar
```

The `<TESTENVDIR>` and `<TESTDIR>` paths are for the locations of the `riscv-test-env` and `riscv-tests` repositories.

Development Tools

It has always been the intent to target the model for both a Windows and a Linux platform. As such two development environments have been setup, and the setup files checked-in as part of the package. For Windows 10 and Ubuntu 20.4LTS were the two development environments.

Visual Studio Community 2019 (Windows)

Visual studio 2019, community edition is freely available for non-commercial use, and Version 16.10.2 was used in the model's development. As of writing, this is available from the following URL:

https://visualstudio.microsoft.com/vs/features/cplusplus/

In the `iss/` directory of the package is a folder `visualstudio/`, which contains the top level solution file and project files for the rv32 executable. The solution compiles the model into a library, and a separate folder, in the `iss/` directory, (`librv32`) contains the project files for this.

In general, if Visual Studio is installed correctly one need only open the solution file in `visualstudio/` and the IDE will find the rest of the project files. A screen like that shown below should be apparent.
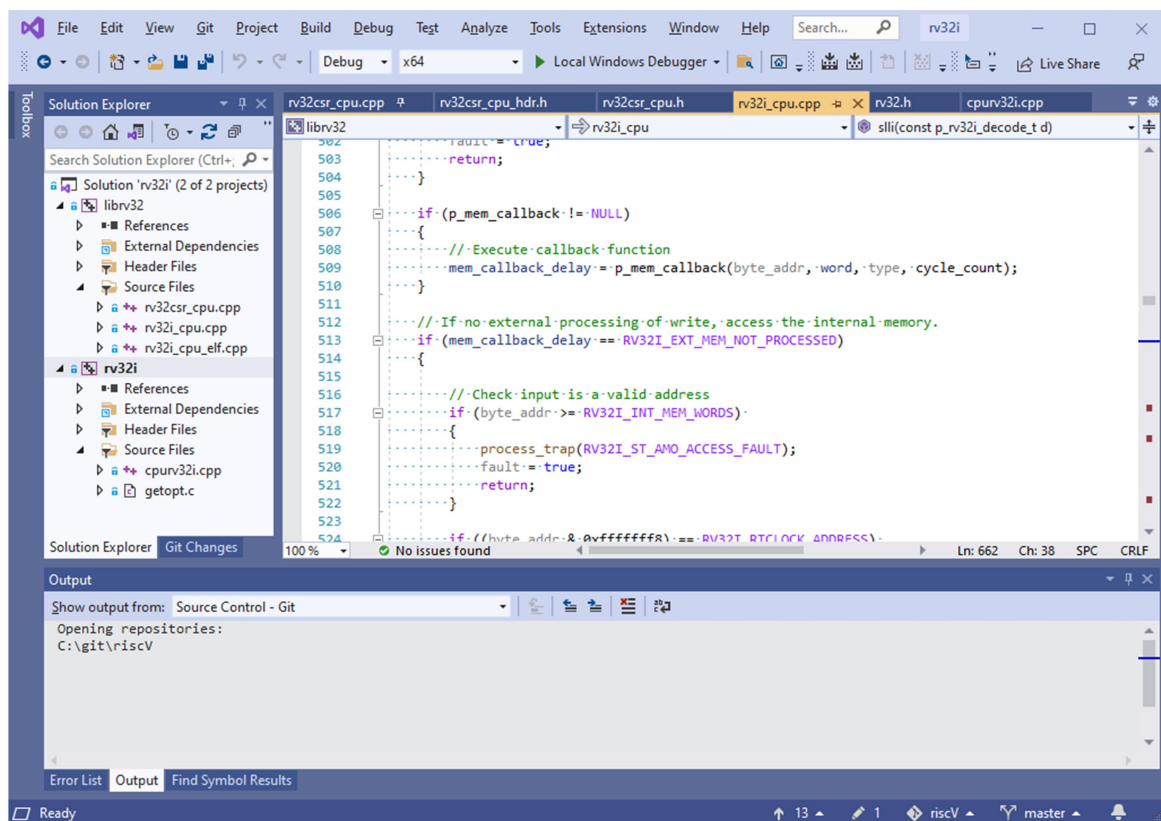
When the solution is built, the library, `librv32.a` will be located in `librv32\x64\Debug`. If the active configuration is `Release`, this replaces the `Debug` in the path. If win32 was selected (not recommended), this deletes the x64 part of the path. All combinations should work.

Eclipse (Linux)

For Linux development the Eclipse IDE was used—in particular version Oxygen.3a Release (4.7.3a, March 2018). This, like visual studio, is feely available and, as of writing, is available at the following URL:

https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/oxygen/3a/eclipse-cpp-oxygen-3a-linux-gtk-x86_64.tar.gz

In the `iss/` directory of the package is a folder `eclipse/`, which contains the top level solution project files for the rv32 executable. There are also project files to compile the model into a library, and a separate folder, in the `iss/` directory, (`librv32/eclipse`) contains the project files for this.

In general, if Eclipse is installed correctly one need only import thee existing project files as librv32 and rv32, and they should compile. Once imported, a screen like that shown below should be apparent.
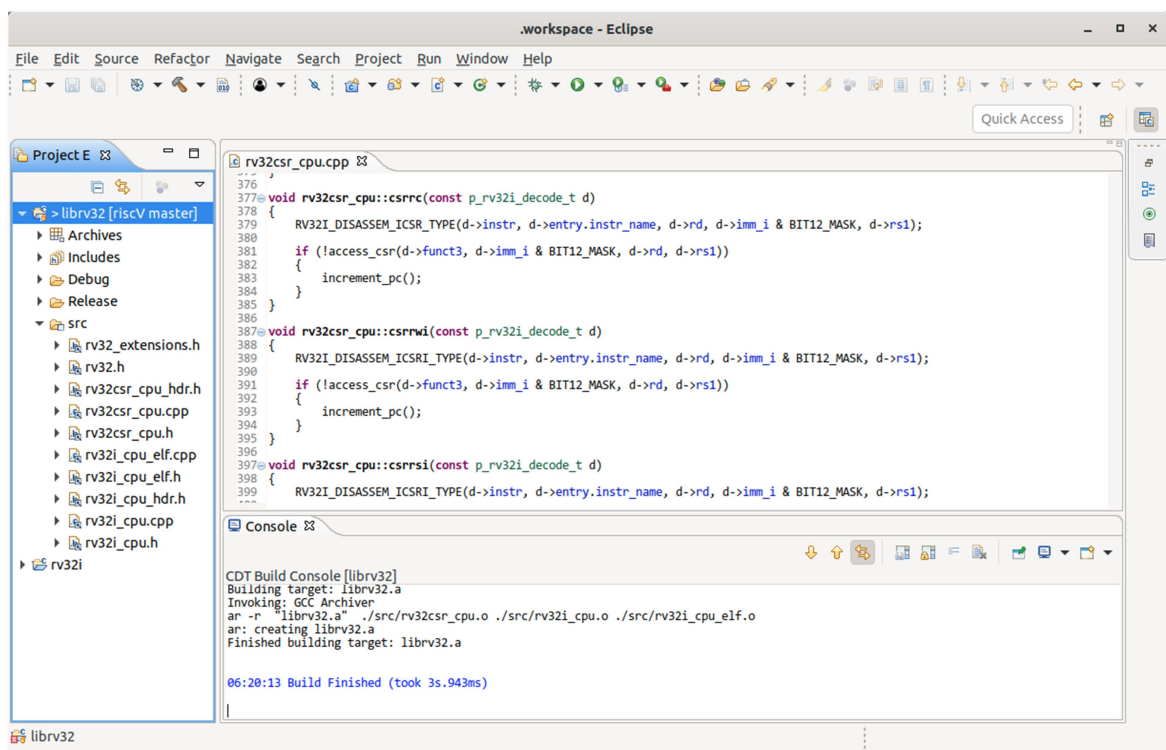
Other Utility Tools

Other tools have been used in this development that need not be utilised but have proven useful, and some of which are assumed in the accompanying scripts. This section lists these and their uses within the project.

MSYS (Windows)

In Linux, if gcc is installed (which it usually is), certain utilities come as standard, one of which is `make`. On Windows this is not the case, but the compilation of RISC-V tests using the provided scripts (e.g. `run32i_tests.bat`) rely on `make` being available. It so happened that MSYS 1.0 was already installed on the original development system, but MSYS 2.0 would probably work just fine as well. As of writing MYS 1.0 can be found at the following URL:

http://downloads.sourceforge.net/mingw/MSYS-1.0.11.exe

This has many Linux shell utilities, including `make`. The scripts expect the MSYS binary directory to be in the path. E.g. `<msys install dir>/1.0/bin`.

Git and Tortoise Git

Git is not required to compile and run the model, but if one wishes to develop and even contribute to the project, then it should be made available, particularly in pulling and pushing from github.com. Github is available for Windows and Linux at the following two URLs:

https://git-scm.com/download/win

https://git-scm.com/download/linux

On windows, an excellent accompanying tool for git is TortoiseGit, which provide a front end to git, making it much more easy to use. It is not exactly a GUI, but integrates itself into windows explorer, showing information about the modifications state, and extending the menus to allow the various operations. This is not essential in using git, but that tool can have some esoteric usage, and TortoiseGit manages this admirably. It is freely available at the following URL:

https://tortoisegit.org/

An example window from rv32 is shown below:
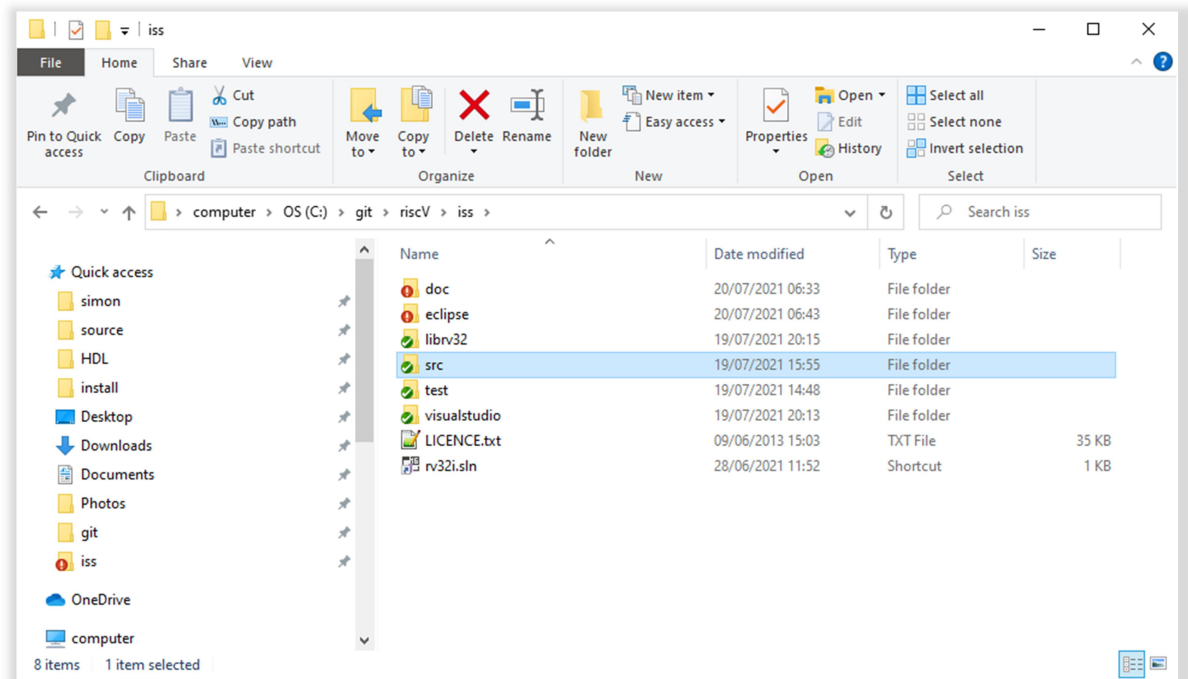
Verification

Top Level Program

As has been mentioned above, the model is compiled as a static library (`librv32.lib` or `librv32.a`). In addition to this there is a top level project (`rv32`) which wraps the library into a compilation for an executable that can load programs and run them on the model. The main purpose of the program is to run the self-checking verification tests. The program provides various command line options to control its operation.

Command Line Options

Below is shown the usage message for the rv32 executable.

```
Usage: rv32.exe -t <test executable> [-hHbdrv][-n <num instructions>]
     [-a <start addr>][-A <brk addr>][-D <debug o/p filename>]
   -t specify test executable (default test.exe)
   -n specify number of instructions to run (default 0, i.e. run until unimp)
   -a specify  address to start executing (default 0x00000000)
   -d Enable disassemble mode (default off)
   -r Enable run-time disassemble mode (default off. Overridden by -d)
   -H Halt on unimplemented instructions (default trap)
   -b Halt at a specific address (default off)
   -A Specify halt address if -b active (default 0x00000040)
   -D Specify file for debug output (default stdout)
   -h display this help message
```

The main command line options specifies which ELF test program is to be loaded and executed (-t)—with a default of `test.exe`. The a various set of options control when the program is to be terminated (if at all) This might be after a number of instructions have been executed (-n), if an unimplemented instruction is encountered (-H), or at a specific address (-b and -A). Finally, a set options enable disassembled output, either as a static list (-d) or during run-time execution (-r). This debug output can be redirected from `stdout` to a specified file with the –D option.

When running tests from the `riscv-tests` suite (see next section), particularly from within a debugging session, the following command line options are recommended.

        rv32.exe –r –b –D debug.txt –t <testname>.exe

With these option, the code will run, sending disassembled output to debug.txt and break on address 0x00000040 (the default), which is the `<write_tohost>` location, which loops forever. At this point, the code is expecting 0 to be in register `x10`, indicating no failures and 93 to be in `x17` to verify it went through the pass or fail parts of the test program and finished. If both conditions match when the program, is halted a "Pass" message is printed, else a "Fail" message and the two codes in the registers are output.

Test Suite

The model is uses the RISC-V unit test suite (`riscv-tests`) to do a large part of the verification. These tests rely on the RISC-V test environment (`riscv-test-env`), and these two repositories are on git-hub at:

The model is constructed in such a way that these test may be run without the need for modifications, with the potential of introducing issues and false positives that this could bring. All of the tests run have come from the `isa/` folder, and the table below shows the status of the model against the tests.

sub-test folder	test	status
rv32ui	simple.S	Passed
	add.S	Passed
	addi.S	Passed
	and.S	Passed
	andi.S	Passed
	auipc.S	Passed
	beq.S	Passed
	bge.S	Passed
	bgeu.S	Passed
	blt.S	Passed
	bltu.S	Passed
	bne.S	Passed
	jal.S	Passed
	jalr.S	Passed
	lb.S	Passed
	lbu.S	Passed
	lh.S	Passed
	lhu.S	Passed
	lui.S	Passed
	lw.S	Passed
	or.S	Passed
	ori.S	Passed
	sb.S	Passed
	sh.S	Passed
	sll.S	Passed
	slli.S	Passed
	slt.S	Passed
	slti.S	Passed
	sltiu.S	Passed
	sltu.S	Passed
	sra.S	Passed
	srai.S	Passed
	srl.S	Passed
	srli.S	Passed
	sub.S	Passed
	sw.S	Passed
	xor.S	Passed
	xori.S	Passed
rv32mi	breakpoint.S	Not run[†]
	csr.S	Passed
	illegal.S	Passed
	ma_addr.S	Passed
	ma_fetch.S	Passed
	mcsr.S	Passed
	sbreak.S	Passed
	scall.S	Partially run[‡]

		shamt.S	Passed
rv32um		mul.S	Passed
		mulh.S	Passed
		mulhsu.S	Passed
		mulhu.S	Passed
		div.S	Passed
		divu.S	Passed
		rem.S	Passed
		remu.S	Passed
rv32ua		amoadd_w.S	Passed
		amoand_w.S	Passed
		amomax_w.S	Passed
		amomaxu_w.S	Passed
		amomin_w.S	Passed
		amominu_w.S	Passed
		amoor_w.S	Passed
		amoswap_w.S	Passed
		amoxor_w.S	Passed
		lrsc_w.S	Passed
rv32uf		*all*	Waiting for `rv32f_cpu` implementation[††]
rv32uzfh		*all*	Half-precision not supported[‡‡]
rv32d		*all*	Waiting for `rv32d_cpu` implementation[††]
rv32c		*all*	Waiting for `rv32c_cpu` implementation[††]
rv32si		*all*	Supervisor level not supported[‡‡]
rv64*x*		*all*	64 bits not supported[‡‡]

NOTES:
[†] Test relies on debug CSR registers to be implemented, which they currently aren't in this model.
[‡] Test does pass, but hits the `<write_tohost>` infinite loop without going through the `<pass>` section of the code, and the model is set up to break on this. This is not unexpected—see comment in `scall.S` at line 61 onwards.
[††] Tests will be run when functionality added.
[‡‡] Feature not planned for implementation at this time.

A batch file is provided (`run32i_tests.bat`), in the `iss/test` folder, to compile and run all the above tests that currently pass. The `scall.S` test is excluded as it does not meet the proper exit criteria, even though it tests without error. This provides a useful, though not complete, regression test for any changes to the model's source code.

Compiling Tests

A `makefile` is provided in the test folder to compile the tests form the test suit. It assumes that the cross-compiler tool chain is installed and paths setup (see Development Tools section). It also assumes that the test suit and environment are checked out at `c:\git`, though this location can be changed by updating `TESTSRCROOT`. To compile a test from the suite—say `sra.S`—the file must be specified along with the subdirectory in which it is located. There are defaults, but it is useful to specify these explicitly on the command line using the `FNAME` and `SUBDIR` variables. So, for example, to compile `sra.S` the command might look like the following:

```
make FNAME=sra.S SUBDIR=rv32i
```

The command will build this test, placing the executable in the test folder as `sra.exe`, which is suitable for loading and running by `rv32.exe`. The `run32i_tests.bat` script uses the `makefile` to compile, afresh, its listed tests when running through the regression.

Planned Enhancements

Extension Support

As mentioned right at the beginning, in the introduction, the basic model implements the RV32I and Zicsr functionality, with Zifencei implied. RV32E is also available via a compile definition. The model is constructed to be easily extensible for the other specifications, and there are definite plans for A, F and D extensions, to bring the model up to RV32G specification ([1] Ch. 24).

Beyond this, the RV32C extensions are not yet slated for implementation, but the model was designed with this in mind, with virtual functions allowing compressed instructions to be managed with overloaded methods. No plans beyond 32 bit are made. This model is a prelude to the construction of an open source soft-core, aimed at FPGAs, and the performance from 64 bit implementations doesn't fit well with that kind of target platform. This is more like a counterpart for, say, the Lattice Semiconductor mico32 processor, an ISS model of which has already been constructed [3].

Connection to gdb and Eclipse

The mico32 ISS referred to ([3]) also has an interface to allow gdb to connect via a remote gdb session, via either an IP socket or a virtual COM link (windows only). This, then, allows an IDE, such as Eclipse, to be hooked into the model via gdb ([3] Appendix A). It is planned to add this functionality to the rv32 ISS, but it is desirous to do this in such a way as to use the RISC-V debug protocols, and this needs further study.

References

[1] Andrew Waterman, Krste Asanović, et. al., editors. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA: Document Version 20191213*, EECS Department, University of California, Berkeley, December 2019.

[2] Andrew Waterman, Krste Asanović, et. al., editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture: Document Version 20190608*, EECS Department, University of California, Berkeley, June 2019

[3] Simon Southwell, *Reference Manual for the LatticeMico32 soft CPU Instruction Set Simulator*, https://github.com/wyvernSemi/mico32/blob/master/doc/README.pdf, August 2016

[4] Simon Southwell, *PCIe Virtual Host Model Test Component*, https://github.com/wyvernSemi/pcievhost/blob/master/doc/pcieVHost.pdf, March 2017.