Why the RISC-V Extension Architecture is Perfect for ISS Modelling in C++

Simon Southwell

April 2025

Preface

This document is from an article written in April 2025 and published on LinkedIn, covering how RISC-V extensions are modelled in the *rv32* instruction set simulator using inheritance.

Simon Southwell
Cambridge, UK
April 2025

Contents

Introduction 4
RISC-V and Extensions 4
Constructing a Base Class 6
The rv32 RV32I base class *6*
Basic Program execution *7*
A Note on Virtual Methods 9
Instruction Fetches *9*
Instruction Decode *10*
Instruction Execution Methods *12*
Adding *Zicsr* Extensions 14
Adding New CSR Access Instructions *15*
Adding Exception Functionality *16*
Adding Conventional Standard Extensions 17
The rv32c_cpu Class *18*
Configuring Extensions 19
Auto-generating the Extensions Header *21*
Extending the Model Without Inheritance 22
Conclusions 23

Introduction

In this document I want to talk about some specifics of a particular instructions set simulator (ISS) that models the RISC-V instruction set architecture (ISA)—namely the *rv32* ISS. Uniquely, of my ISS models, this ISS has an architecture feature that comes directly from the nature of the RISC-V instruction set architecture that it's modelling. I'm going to assume, from this point on, you have some familiarity of the RISC-V ISA. If not, you can read my article that is an Introduction to RISC-V. I will also assume some rudimentary knowledge of C++ and its classes, with a basic idea of inheritance, though we will look a little more closely at the inheritance within this document in reference to an programming an ISS model.

An advantage of the RISC-V ISA is the use of extensions to expand the base compliant specification in a formal way, whilst allowing a compliant implementation to only include those extensions (if any) that are required for the particular application. When designing the ISS for a RISC-V processor, based on previous iterations of ISS models that I'd constructed (see, for example the mico32 model) it occurred to me that I would need a way to cleanly delineate extension functionality and be able to add new extensions without disrupting the code that is already implemented. In C++, class inheritance is a way to take a base class and extend and/or alter its functionality, in a similar manner to the RISC-V extensions to the base specification.

So in this document I want to look at the *rv32* architecture and use this as a case study on some relevant features of C++ inheritance and how this allows the model to be configured to add just those extensions required, and no more. The model can then be matched up with real-world implementations and respond in the expected manner if an unimplemented instruction is encountered. After all, if all the functionality of the available extensions were included, then it could execute an instruction that the processor it's modelling would not be able to process. The model has an accompanying GUI script to select amongst the available extensions to configure the model code to compile for the selected extensions only. There is also a method to extend the model without writing a derived class, or reference to *rv32*'s source code at all, and we'll look at this as well.

RISC-V and Extensions

I have written about the RISC-V instruction set architecture before in a previous article, but it is worth summarising here the subjects relevant to our discussion. The RISC-V ISA has a base specification for 32-bit and 64-bit (and even 128-bit)

architectures. These are known as RV32I and RV64I (we'll skip 128-bit specs), with the 'I' standing for 'integer'. To be RISC-V compliant the implementation must fully implement one of these specifications, with RV32I instructions included in an RV64I architecture.

Unusually for an ISA, the RISC-V base specifications do not include control and status register functionality, including the exception and interrupt handling. I guess this allows a minimalist core that is meant only to run a single thread that's not event based if resources are scarce. There are a set of standard extensions that have been ratified by the RISC-V standards body, but custom extensions are allowed. The table below lists *some* of the standard extensions, but this is not exhaustive, as new extensions are being ratified all the time and added to the standard extension list.

extension	description
Zicsr	Control and status registers
M	Integer multiply and divide
A	Atomic operations
F	Single precision floating point
D	Double precision floating point
C	Compressed (16-bit instructions)
B	Bit manipulation
E	Embedded reduced state
Zifencei	Load and store fence

The naming conventions for the extensions is somewhat esoteric. Most have a single letter of the alphabet, but 26 letters soon run out, and so there are extensions that use 'Z' followed by some other characters. Take, for example, the Zicsr extension. The 'i' is for integer followed by 'csr' for the control and status registers—you get the idea. When specifying an implementation's compliance to extensions, it is written out as the base plus extension letters. E.g., RV32IMZicsr, meaning a 32-bit architecture with base integer specification with multiply/divide instructions and CSR registers.

Another quirk is that the 'G' "extension" is actually a collection of the base plus a certain set of extensions. So RV32G is the same as RV32IMAFDZicsr_Zifencei. It is a convenient shorthand for a "general purpose" implementation that can be used in a number of applications domains. This includes all the extensions in the table above except the bitwise (B), compressed (C) and embedded (E) extensions. The B extensions are bit-manipulation instructions, whilst the C extension defines compressed 16-bit instructions which are a sub-set of the main architecture instructions (and can be expanded to full size instruction equivalents) used to save code space. They can be freely mixed with full sized instructions, which does mean

that full sized instructions can be aligned to a 16-bit boundary rather than naturally aligned to the architecture word length, which must be handled.

There are many more extensions that are applying for standardisation, but hopefully you get the idea of how this works. It allows a flexibility in compliant implementations to only include what's necessary for the intended use, but still remain compliant. The RISC-V `gcc` toolchain has command line options to specify both the architecture size (32 or 64) and the extensions it can use. E.g.:

```
riscv-unknown-elf-gcc -mabi=ilp32 -march=rv32imc <other options>
```

Here the `-mabi` (for application binary interface) is set to `ilp32`, which says that the integers, longs and pointers are 32 bits—i.e. compile for 32-bit architecture (there is an `lp64` option for 64-bits as well as variants of both). The `-march` option (for architecture) specifies `rv32imc`, saying the 32-bit compilation can include multiply/divide and compress instructions on top of the base integer instructions.

So, having understood RISC-V extensions, let's switch topics to C++ and inheritance and how we can combine these to construct the RISC-V ISS's architecture.

Constructing a Base Class

Before we can use inheritance to extend functionality there must be a 'base' class from which to inherit. The RISC-V architecture's most basic specification is the RV32I, which specifies a 32-bit core with only the basic integer based instructions implemented. In this specification it does not specify any control and status registers, or even exception handling. One might imply some hard coded exception defaults, such as an exception address vector, but this is not specified. So we can construct an ISS model class and use this RV32I specification as our basis from which we can extend functionality to add support for other RISC-V extensions using C++ class inheritance. It is worth taking some time now to describe the general architecture of base class we are going to use for reference in later sections. I have documented ISS architecture before, in general terms, in a previous article but it will be helpful to summarise the specific *rv32* base class that's is being discussed and which forms the basis of the ISS model.

The *rv32* RV32I base class

The *rv32* ISS defines a base class, `rv32i_cpu`, which, as well as implementing code for all the RV32I instructions, has all the functionality of the ISS mechanism to read instructions, decode these instructions and execute the instruction functionality—

updating internal state as appropriate. It is this functionality that is the focus of this document, but the base class has other features useful in an ISS model. I will list these below for completeness, but then we can put them to one side for the moment:

- Cycle count and real-time clock for timing
- An internal 1Mbyte memory model
- User callback function features for memory accesses and unimplemented instructions
- Ability to load RISC-V ELF binary executable code
- A remote gdb debugging interface
- External API to read and write memory, read registers and reset the CPU state
- A default timing model for instructions, externally updatable to match other processor core implementation timings.

More details of the above features can be found in the *rv32* manual, but we'll concentrate on the fetch-decode-execute functionality but, knowing we will need to deal with exceptions, interrupts and traps at some point, we will also put some hooks in for this.

Basic Program execution

From an external user API point of view there is a single method to start running a program on the ISS, assuming we've already registered any callback functions (e.g. `register_ext_mem_callback()`) and loaded a program (via `read_elf()`):

```
int run(const rv32i_cfg_s &cfg);
```

This `run` method is called with a single argument—a reference to a configuration structure. The details are not important but is used to enable or disable different features such as run-time disassembly and dumping registers or memory when execution completes. It can also set reasons for program execution to stop and return from this method, such as after executing a set number of instructions, reaching a particular address, or encountering a system instruction such as `ebreak` or `ecall`, or on an instruction error. If and when the method returns it returns an error status, such as `SIGTRAP` or `SIGTERM`, with 0 being a no error status.

Under the hood of the run method is a basic loop to do the fetch, decode and execute but with some place holder calls for handling exceptions and interrupts.

```
int rv32i_cpu::run(rv32i_cfg_s &cfg) {

  for (instr_count = 0;
      (cfg.num_instr == 0 || instr_count < cfg.num_instr) && !error &&
     !(cfg.en_brk_on_addr && cfg.brk_addr == state.hart[curr_hart].pc);
       instr_count++) {

        // Firstly, check interrupt status
        if (!process_interrupts()) {
            // Fetch instruction
            curr_instr = fetch_instruction();

            // Decode
            p_entry = primary_decode(curr_instr, decode);

            // Execute, calling an instruction method
            if (p_entry != NULL) {
                error = execute(decode, p_entry);
            }

            // Process any illegal instruction exceptions
            if (error == SIGILL && !halt_rsvd_instr) {
                process_trap(RV32I_ILLEGAL_INSTR);
                error = 0;
            }
        }
    }
    return error;
}
```

The run method, with an abbreviated version shown above, is a basic for loop, keeping count of the number of times around the instruction loop and a set of criteria for exiting the loop. This includes reaching a configured number of times around the loop, an error condition returned by the execution method, or hitting a specified address on the program counter (PC). Within this loop three methods do the processing of instructions—namely, `fetch_instruction()`, `primary_decode()` and `execute()`.

Bracketing these three basic operation are calls to some exception handing methods. The first, `process_interrupts()` is for handling interrupts from external sources, from a timer, or from a software generated interrupt and is called before the current instruction is processed. The second method, `process_trap()`, is called after an instruction is processed and it generates an illegal instruction signal (`SIGILL`). This allows any user registered callback for illegal instructions to be called for adding custom instructions to the basic implementation. It was previously mentioned that the RV32I specification does not specify exception behaviour, so what are these two methods doing in the base class? Well, this `run` method is not going to be overloaded under normal circumstances and will be common to all versions of the

ISS, so some place holder methods are defined in the base class to be overridden by extension derived classes. In the `rv32i_cpu` class, the `process_interrupts()` method simply returns 0, indicating that no interrupt is active. The `process_trap()` method does actually change the PC value to a fixed trap address set at `0x00000004`.

A Note on Virtual Methods

Both these methods in the base class are defined as "virtual", as are some other methods that are meant to be overridden. So, what's the difference between overriding a virtual method versus overriding a normal method? Well, both will still work in general, but there is a difference when using pointers for a base class to access methods of an object of a derived class. If a base class method is not virtual, then a pointer of base class type pointing to a derived class object will call the base class method when accessed. If the method was defined as virtual, however, then the base class pointer dereference will get the derived class method. This is all to do with polymorphism, but does it matter for an ISS model? Well, probably not, as I can't see any reason why pointers of base class type would be mixed with pointers of derived class type would co-exist. I have defined them as virtual mainly to indicate that they are *intended* to be overridden, and the overriding methods would then be the ones to use under all circumstances.

One other way of defining a virtual method in a base class by declaring the prototype and then making equal to 0: e.g. `virtual void func (void) = 0;`. This means that before the class can be used to generate an object, that function must be overridden to provide the functionality. A class containing such a virtual method is called an abstract class. In the *rv32* ISS there are no abstract classes and the base class can be used stand-alone.

Other methods are defined as virtual that will be useful when deriving extension classes:

- `reset()`: sets privilege mode and sets PC to reset vector address
- `increment_pc()`: Defaults to incrementing by 4, but can be overridden for compressed instruction handling, where incremented by 2
- `fetch_instruction()`: Defaults to reading 32-bit instructions, but overridable for compressed instructions

Instruction Fetches

Fetching instructions is the simplest of the operations in processing code It is a wrapper for calling an internal method called `read_mem()` and it is this method that

does the heavy lifting. Firstly it does a load of checking on the address for correct alignment, raising a fault if in error and calling a `process_trap()` method. Since the *rv32* model has an internal timer it does also some decoding of the address to see if it is accessing the memory mapped timer registers and makes calls to the appropriate internal timer methods to fetch or set timer state. Otherwise, the method checks if the user registered a callback for memory accesses and calls this if it did, passing in the address, a reference to a 32-bit data variable, the type of access (read, for instructions) and the current cycle count. The callback can decide that it can't handle the memory access and can return with a value indicating so, or it can handle the access, updating the data on reads and returning with a cycle count for the operation which gets added to the tally kept by the *rv32* model (in `cycle_count`). If the callback didn't process the access, then `read_mem()` will access its internal memory if the address is in range. If the address is out of range then `process_trap()` is called with a load access fault. Just for interest an equivalent `write_mem()` method is defined and, along with `read_mem()`, are public API methods. They are also used for loading programs to memory and thus will load to external locations if a callback is registered, and they are used by the remote gdbserver code as well.

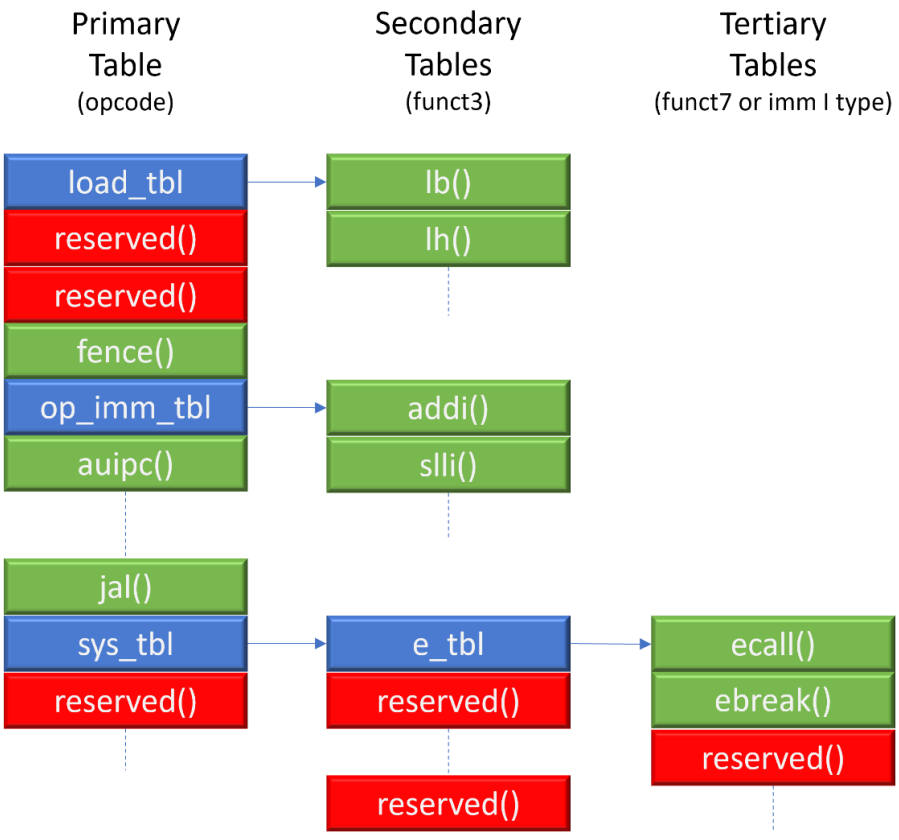
Instruction Decode

For some processors' instruction architectures there are a limited number of instructions defined. For example, the LatticeMico32 softcore processor from Lattice Semiconductors, comparable in many ways to the RISC-V RV32I specification, has just six bits of opcode for a maximum of 64 instructions. In a case like this, decoding can be just a matter of a switch statement on the opcode bits with calls to methods for executing the instruction or even, if the instruction functionality is simple enough, to implement the instruction code at each case clause. This is certainly very fast in terms of execution of the model.

However, if, like the RISC-V ISA, there are more decode bits, the size of the switch becomes unmanageably large. As well as seven bits of opcode, some instructions can have a *funct3* field of three bits for further sub-decoding and even (for R-type instructions) another seven *funct7* bits of sub-decoding. This gives a total of a possible 131072 instructions just for the R-types, though reduced somewhat by other types that don't have as many sub-decode bits. None-the-less, this is a large space, most of which will not decode to a valid instructions. Because of this the *rv32* ISS uses tables, decoded on each of the three decode fields: opcode, *funct3* and *funct7*. A primary decode table decodes on the 7 bits of opcode, and is thus 128 entries. Each entry in the table is a structure (`rv32i_decode_table_t`) that has a union (`ref`)

which is either a pointer to another table (of the same `rv32i_decode_table_t` type) or a decoded entry structure containing information for disassembly of an instruction. In addition, a flag indicates whether the entry is for a sub-table or not. When true, the `ref` union is a pointer to another table for decoding on the sub-decode bits—from the primary table this is a secondary table decode on *funct3*—otherwise `ref` is instruction decoded information for use for printing when disassembling and then a pointer to an execution function is valid in the table entry structure. It is this function that is the method for the particular instruction implementation.

Thus the primary table entries may consist of instruction method pointers or of pointers to *funct3* sub-tables. A special instruction method, `reserved()`, is set for any instruction that is not implemented which will call `process_trap()` if executed. There will only be one primary table, whereas there will be as many secondary sub-tables as required for instructions with a *funct3* field, though each will only be 8 entries big. Similarly, the secondary tables may contain pointers to instruction methods, or pointers to the `reserved()` method, or pointers to tertiary tables, decoded on *funct7* (or sometimes on `imm` fields for a few immediate instructions). Thus a linked list of tables is produces of up to a depth of three and the decode functionality 'walks' through the tables until it finds an instruction entry (including `reserved()`). The diagram below summarises this situation.



The tables are filled in within the `rv32i_cpu` class's constructor. The prototype for all instruction execution methods is as shown below:

```
void (rv32i_cpu::* pFunc_t) (const p_rv32i_decode_t);
```

The `p_rv32i_decode_t` is a structure that is filled in by the `primary_decode()` method and contains the raw instruction value and all the possible fields separated out into individual entries in the structure, including immediate values (appropriately sign extended), the *rd* and *rs* index values, the decode field values and a copy of the decoded entry for disassembly, containing a pointer to an instruction name string.

Instruction Execution Methods

For the *rv32* ISS, all the instruction execution methods have the same prototype and more or less do the same thing—update state based on the particular instruction to be executed. But, within the commonality, there are three subtle sub-categories:

- Linear instructions. E.g. arithmetic, logic bitwise
- PC Altering instructions: E.g. branches and jumps
- Memory accesses: loads and store.

The linear instructions always have the PC incremented to the next instruction in memory. For the `rv32i_cpu` class this means incrementing it by 4. In addition, they can't generate any exceptions and always update state into one of the 32-bit processor registers `x0` to `x31` (though `x0` always remains at 0) using either the value from another register or an immediate value encoded into the instruction. An example method of this type is shown below:

```
// Add immediate instruction
void rv32i_cpu::addi(const p_rv32i_decode_t d)  {
  if (d->rd) {
    state.hart[curr_hart].x[d->rd] =
       (uint32_t)state.hart[curr_hart].x[d->rs1] + (int32_t)d->imm_i;
  }
  increment_pc(); // Compressed instruction aware PC increment
}
```

The PC updating jump and branch instructions update the special `pc` register and, for most branch instructions, is conditional on some criteria. If the criteria are met the PC is updated based on an immediate value added to the current PC value, otherwise it is incremented as for linear instructions. Jumps may also include a value relative to a processor register rather than the current PC value, and all update a nominated processor register to store the address of the next linear instruction. What is different

about this category is that the instructions can generate exceptions. The instruction may generate an instruction address that is invalid—for example an instruction address that is aligned to an odd byte boundary. This is checked for and the `process_trap()` method called if failed. An example is shown below:

```
// Branch if equal instruction
void rv32i_cpu::beq(const p_rv32i_decode_t d) {
  if (state.hart[curr_hart].x[d->rs1] == state.hart[curr_hart].x[d->rs2]) {
    // Check for misalignment on target address
    if (access_addr & RV32_IADDR_ALIGN_MASK) {
      process_trap(RV32I_IADDR_MISALIGNED);
    else
      state.hart[curr_hart].pc = state.hart[curr_hart].pc + d->imm_b;
  else
    increment_pc();
}
```

The memory access instructions are the only ones to access outside of the processor registers. In the *rv32* ISS, they all make use of the internal `read_mem()` and `write_mem()` methods, both of which will offer an access to any externally registered user callback function and can also generate exceptions (calling `process_trap()` from within the read and write methods). Other than that they update processor register state if a load type instruction or store a processor register's value to the addressed memory location. An example is shown below.

```
// Load word instruction
void rv32i_cpu::lw(const p_rv32i_decode_t d) {
  bool access_fault = false;
  access_addr = state.hart[curr_hart].x[d->rs1] + d->imm_i;

  // read_mem will generate misalignment traps
  uint32_t rd_val = read_mem(access_addr, MEM_RD_ACCESS_WORD, access_fault);

  if (!access_fault) {
    state.hart[curr_hart].x[d->rd] = rd_val;
    increment_pc();
  }
}
```

We now have methods for the three major steps in the instruction loop that can implement to the RV32I specification, as well as having hooks for implementing interrupts and exceptions—and this will be our first extension.

Adding *Zicsr* Extensions

It is unique for RISC-V (as far as I know) that the control and status registers are not included as part of the basic specification and have their own extension. For RISC-V this is the *Zicsr* extension. For any practical general purpose processor implementation there will need to be some control and status register functionality, but it is not completely out of the question, on a bare-metal implementation that is not event driven, that the *Zicsr* functionality is not required with the savings in implementation resource usage that would otherwise be needed. Thus, from a modelling point of view it makes sense to treat this extension like any other.

The *Zicsr* extension defines not only a set of new 32-bit registers, separate from the processor registers, but in a large 4096 register space. Accessing the registers in the space requires a new set of instructions (`csr<xx>` and *x*`ret`), but also implementations for the rules for handling interrupts and exceptions. Fortunately, the whole 4096 register space is not used and, even more fortunately, not all possible registers are obligatory. The table below shows the registers used, as an example, for the NIOS V/m Softcore Processor from Intel/Altera.

offset	type	name	description
12'h300	read-write	**mstatus**	Machine status register
12'h301	read-write	**misa**	ISA register (indicating which extensions are present—can be 0)
12'h304	read-write	**mie**	Interrupt enable register
12'h305	read-write	**mtvec**	Interrupt/exception vector base address register
12'h341	read-write	**mepc**	Exception program counter (address of interrupted instruction)
12'h342	read-write	**mcause**	Code indicating interrupt/exception that caused a trap
12'h343	read-write	**mtval**	Register holding exception specific information
12'h344	read-write	**mip**	Interrupt pending register
12'hf11	read-only	**mvendorid**	Registered ID of vendor of implementation (can be 0)
12'hf12	read-only	**marchid**	Registered base architecture of the processor hart (can be 0)
12'hf13	read-only	**mimpid**	Implementation ID (revision number—can be 0)
12'hf14	read-only	**mhartid**	HART (hardware thread) ID. Must implement at least 1 HART with ID 0

As can be seen, this is a small set of registers and even the green coloured instructions can be hardwired to 0. Note that there are also some 64-bit timer and timer compare register defined (`mtime` and `mtimecmp`) that are memory mapped rather than in the CSR space. This matches closely what the *rv32* ISS implements with the addition of an `mscratch` register and `mcycle` and `minstret` registers to make available clock cycle count and number of completed (retired) instructions. The registers in the table are mostly to do with interrupt and exception handling.

So, we need to add to the base `rv32i_cpu` class functionality to implement these CSR registers, the instructions to access them, and the functionality to handle the interrupts and exceptions. A new class, `rv32csr_cpu`, is defined which is derived from the `rv32i_cpu` base class—i.e. it inherits the base class, making all the previous functionality available. It will, however, override the virtual methods defined in the base class and update the decode tables to include its new instructions, which were previously 'reserved'.

Adding New CSR Access Instructions

There are just two simple steps to adding new instructions. Firstly, defining the execution methods, exactly as we did for the `rv32i_cpu` class execution methods. The new `csrxx` instructions are like the linear instructions except that state reading and updating now includes the new CSR registers. Secondly, we need to add the new instructions to the appropriate places in the decode tables. In the base class we did this in its constructor. We could completely overload the constructor and initialise the tables from scratch in the `rv32csr_cpu` class's constructor, but it has no knowledge of these instructions and their execution methods. Fortunately, we can call the base class's constructor from within the derived class's constructor, and so we can rely on the fact that all the tables exist and are populated with the RV32I instructions, and all other initialisations are completed. The code below shows the syntax used for the *rv32* ISS:

```
#include "rv32i_cpu.h"

class rv32csr_cpu : public rv32i_cpu
{
public:
    // Constructor declaration
    rv32csr_cpu      (FILE* dbgfp = stdout);

    /**** Rest of class definition ****/
};

// Constructor definition
rv32csr_cpu::rv32csr_cpu(FILE* dbgfp) : rv32i_cpu(dbgfp)
{
    /**** Rest of constructor definition ****/
}
```

Here we see that the `rv32csr_cpu` class declares `rv32i_cpu` to be inherited and has a constructor method prototype, as normal. In the constructor's definition the base class constructor is called explicitly in the derived class's initialisation list, passing in

the parameter from its own argument list. The rest of the constructor code can now fill in the relevant table entries with the new `csrxx` instructions and do its own state initialisation.

Adding Exception Functionality

It has been mentioned above that the `rv32i_cpu` base class had place holder methods that needed overriding, and the `rv32csr_class` overrides the following methods:

- **`reset()`**
- **`process_trap()`**
- **`process_interrupts()`**

The reset method for `rv32csr_cpu` needs to initialise its CSR registers to disable all interrupts and set load/store privilege mode to normal. Once again, the derived class knows nothing of the base class's initialisation requirements or even has access to all the state that needs initialising. However, we can call the base class's reset method explicitly as shown in the code below:

```
void rv32csr_cpu::reset()
{
    rv32i_cpu::reset();

    state.hart[curr_hart].csr[RV32CSR_ADDR_MSTATUS] =
                  ~(RV32CSR_MIE_BITMASK | RV32CSR_MPRV_BITMASK);

    state.hart[curr_hart].csr[RV32CSR_ADDR_MCAUSE]  = 0;
}
```

For the `process_trap()` method, there is no need to call the base class method, and it simply adds the functionality to update state based on exception conditions. This involves updating CSR registers to reflect the nature of the exception, store exception address locations and update the PC value to an exception address to start running trap code or interrupt service routines.

The `process_interrupts()` method also fully overrides the base class method and is really a wrapper for the `process_trap()` method but processes the incoming signals, masking them against global and specific masks in the CSR registers and only calling `process_trap()` if an interrupt is active and enabled. However, pending status is also mapped to a CSR register (specifically `mip`) and this is updated in all events within this method. External interrupt state signalling is done via a user

registered callback function which returns any active interrupt state. The *rv32* ISS model has an internal timer which can also generate an interrupt signal, as well as a memory mapped software interrupt, flagged externally from the user callback.

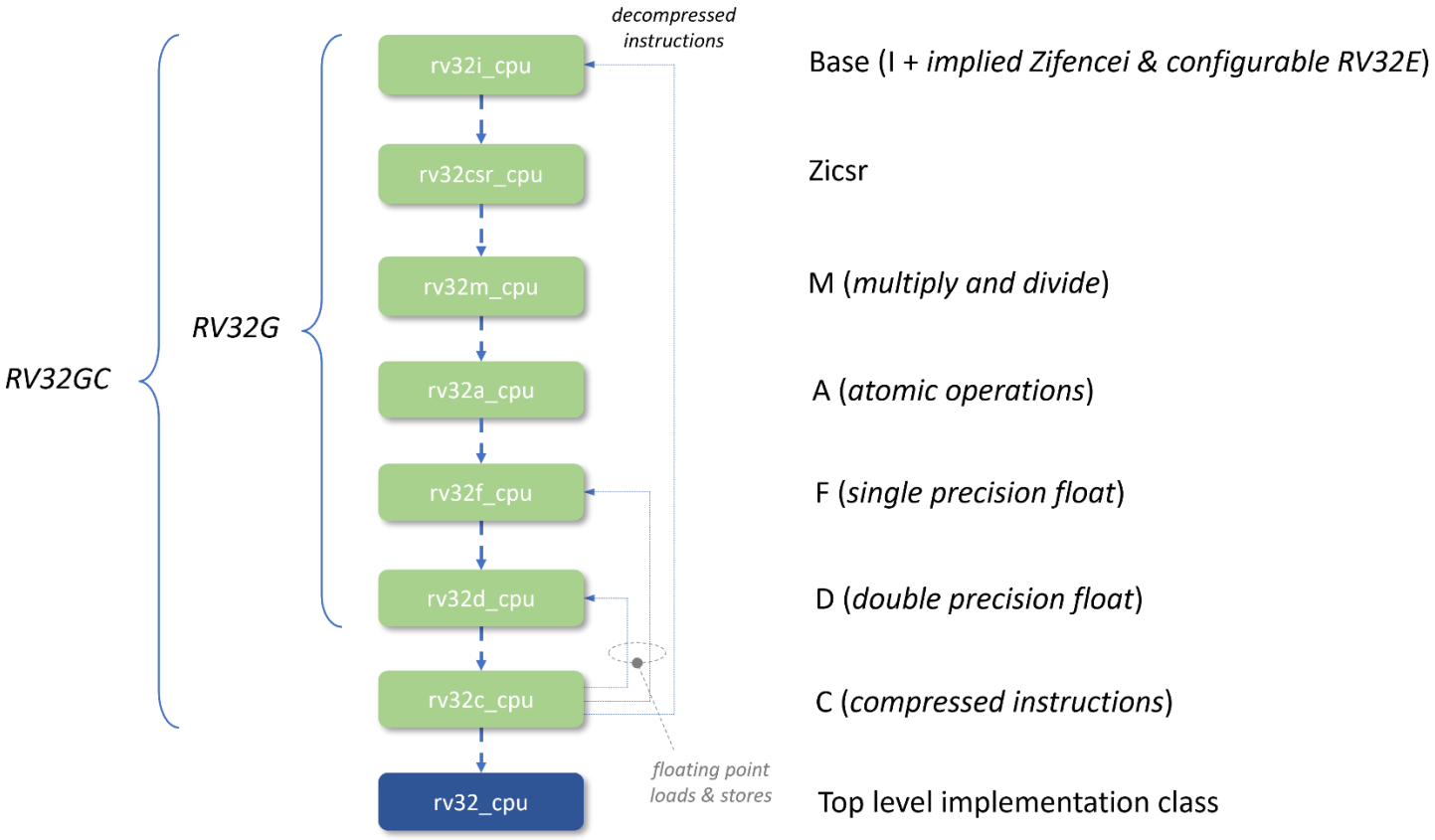
One last method that is overridden is the `run()` method. This is simply to call an internal method, prior to exiting, to update CSR register instruction count values to latest internal counts so that they display correctly if dumping of CSR registers is enabled. This is really an optional override for convenience, and the base class method was not declared as virtual.

Adding Conventional Standard Extensions

Adding the *Zicsr* extension demonstrated means to add to the base model both new instructions and new functionality in the CSR registers and exception handling. The other standard instructions, though, tend only to add new instructions and, since we already have a simple means to do this we can add as many extensions as we want. We can do this in a linear hierarchical form with each new extension inheriting from the previously derived class in the chain. The *rv32* ISS currently supports the following standard extensions:

- RV32M: Integer multiply/divide
- RV32A: Atomic operations
- RV32F: Single precision floating point arithmetic
- RV32D: Double precision floating point arithmetic
- RV32C: Compressed (16-bit) instructions

It also supports RV32E (reduced registers for embedded applications) as a compile time option. The RV32MAFD group of extensions is also referred to as RV32G, for general purpose processor extensions. Each of the extensions are defined as separate classes, adding instructions from each, and we can create an inheritance chain like that shown below:



At the end of the chain of extension inheritance is a top level `rv32` class which gives a common class view to the user even if new extensions are added. All the extensions simply add instructions to the tables, as described, with the exception of the RV32C compressed instruction class, `rv32c_class`.

The rv32c_cpu Class

The compressed instruction class doesn't have any instructions of its own as such, but the RV32C specification defines 16-bit instructions that map on to 32-bit equivalents. I.e. They encode a subset of the RV32I instructions. So, to implement this, all that needs to happen is that the compressed instructions are converted to their 32-bit equivalents and then decoded as normal. The RV32C specification allows 16-bit compressed instructions to be freely intermixed with 32-bit instructions which has the side effect that 32-bit instructions might be aligned on an odd 16-bit boundary, and this must be taken into account. However, compressed instructions can be uniquely identified in their opcode bottom two bits of instruction, so partial decoding can be done to determine whether it's a compressed instruction or not.

The `rv32c_cpu` class overloads two inherited functions: namely `increment_pc()` and `fetch_instruction()`. Whereas the base class fetch instruction simply did a word read via the internal `read_mem()` method, the overloaded function must

determine whether an instruction is compressed and expand it if it is. It makes use of the base class's `read_mem()` method (with all the checking that's done there) but must, potentially, hold over half-words to get correct alignment for 16-bit aligned full instructions. It also flags if the current instruction is compressed or not, and this is used by an overloaded `increment_pc()` to increment either by 2 or 4, depending on the setting of the flag.

Configuring Extensions

So now we have an implementation with a useful set of extensions and a means to continue to add new extension functionality. But what if we don't want all the extensions? What if we want a model of a hardware implementation that only has a sub-set of the extensions? If an extension in the ISS model is present but not the core that it's supposed to be modelling, it will behave differently when encountering an instruction that's in the ISS but not the hardware. We need a convenient way of selecting which extensions are included in our model.

Previously, in the inheritance diagram, we had a linear parent-child derivation of a particular order, so that a given extension inherits from a particular class in the chain. It makes sense (and is required) to have the `rv32i_cpu` class as the base class and there is an argument that the `rv32csr_cpu` should be the first derived class. The other extension classes, though, could be inherited in any order and still function correctly. In this case, though, a given extension class would not know which class to inherit from. If we could configure this, we could not only construct the derived classes in any order, but we could even skip one or more extension inheritance steps, thus removing that extension.

The method used by the *rv32* ISS may not be an optimal solution, but the goal was to be able to configure the model without changing the main source code files and using just a header file. The header file was also to be auto-generated to avoid configuration errors possible with a manual construction of the header. Once I describe how the model works, I am open to suggestions on any better method of achieving exactly the same thing.

The first step to achieving the configurability was to replace a specific class for inheritance with a definition, to be defined in a header externally. Referencing the diagram earlier of the `rv32csr_cpu` class, this is now modified to look like the following:

```
#include "rv32_extensions.h"
#include RV32CSR_INCLUDE

class rv32csr_cpu : public RV32_ZICSR_INHERITANCE_CLASS
{
public:
    // Constructor declaration
    rv32csr_cpu      (FILE* dbgfp = stdout);

    /**** Rest of class definition ****/
};

// Constructor definition
rv32csr_cpu::rv32csr_cpu(FILE* dbgfp) : RV32_ZICSR_INHERITANCE_CLASS(dbgfp)
{
    /**** Rest of constructor definition ****/
}
```

We have now pulled in an `rv32_extensions.h` header file and anywhere that an explicit reference to a base class was made this is now replaced with a definition specific to our derived class (`RV32_ZICSR_INHERITANCE_CLASS`, in this case) that has been defined in the extensions header file. This means the parent class is now determined externally in the header file and can be any of the classes available. The include for the original parent class is also updated to use a specific definition to pick up the correct configured parent class's header. All the extension classes are updated like this with their own definitions to use, as defined in the `rv32_extensions.h` header.

The extension header file itself simply defines these macros in such a way as to create a linked list of inheritance with the required extensions in the list. The code below shows an `rv32_extensions.h` header file when all extensions are included.

```
#ifndef _RV32_EXTENSIONS_H_
#define _RV32_EXTENSIONS_H_

#define RV32_I_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_C_INHERITANCE_CLASS         rv32d_cpu

// Uncomment the following to compile for RV32E base class,
// or define it when compiling rv32i_cpu.cpp
//#define RV32E_EXTENSION

#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                    "rv32f_cpu.h"
#define RV32C_INCLUDE                    "rv32d_cpu.h"

// Define the extension spec for the target model. Chose the
// highest order class that's needed.
#define RV32_TARGET_INHERITANCE_CLASS    rv32c_cpu

// Define target include: must match include of RV32_TARGET_INHERITANCE_CLASS
#define RV32_TARGET_INCLUDE             "rv32c_cpu.h"

#endif
```

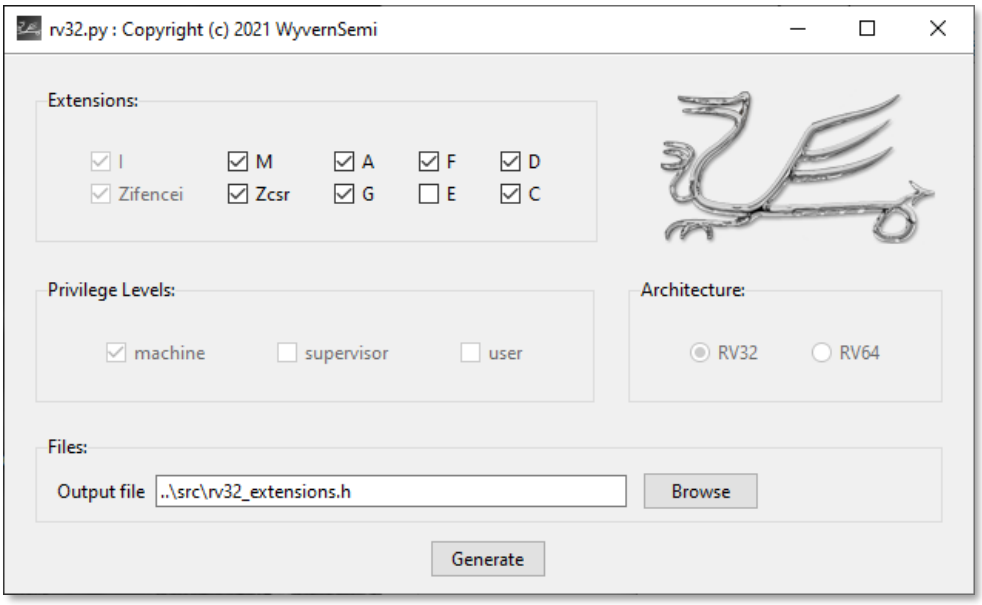
Here the inheritance macros are defined to create a chain like that in the earlier diagram so that, for example, `rv32a_cpu` will have `rv32m_cpu` as its parent class. The header file to include for a given extension is also defined and matches the inheritance class definition. Finally, the top level `rv32` class has its parent class and header file defined, being for the last link in the chain of extensions.

If one or more extensions are to be removed, then it is a matter of an extension further along the chain skipping its nominal parent to make a previous extension its parent. For example, suppose the RV32A atomic extension were not required then the `RV32_F_INHERITANCE_CLASS` definition would be changed to be `rv32m_cpu`, and `RV32F_INCLUDE` would be changed to `"rv32m_cpu.h"`. The values for the two RV32A extension then become “don’t care” as they won’t be used. Now, compiling the model will generate an ISS without the RV32A implemented and an atomic instruction executed will cause an illegal instruction trap.

Auto-generating the Extensions Header

You may have noticed that the extensions header is somewhat awkward to manage if taking out extensions and it sort of defines the same information in two places,

which could be a cause of issues if they mismatch. So, in order to avoid this, we want to auto-generate the extension configuration header. A Python script is provided with the *rv32* ISS model to do just this, and it has a GUI front end, as shown below:



From the GUI we can select which extensions to include. Note, that selecting 'G' will automatically enable MAFD extensions. The output file is specified in a box at the bottom and then the 'Generate' button will generate the header file. There are some future proofing unmodifiable boxes as well, for things like RV64 and different privilege modes, which are not currently supported.

Extending the Model Without Inheritance

The main thrust of this document has been mapping the RISC-V instruction set architecture and the standard extensions to using the inheritance features of the C++ programming language and how adding further extensions proves easy with further derivative class definitions. But what if we want to add new instructions without modifying the main source code to, perhaps, experiment with custom instructions?

The *rv32* ISS has another user callback feature where a user function can be called whenever the model encounters an unknown instruction. The callback function is invoked with the decode structure mentioned earlier and with another structure containing the register and PC values, whether the instruction is compressed and the raw instruction value. Upon return, the callback can indicate whether it processed the instruction or not, whether it did process but generated a trap or whether it successfully executed the instruction. If successful, it will pass back any updated PC

value (with a flag to say so) and similarly if the registers were updated so that the ISS internal state can be updated with these new settings.

This, then, gives a means of extending the model's ISA externally without recourse to adding new source code to the model itself, using the model as a library function, and adding external code to implement these new instructions.

Conclusions

In this document we have looked at modelling the RISC-V instruction set architecture and how it marries, extremely well, with the C++ class based programming language to make modelling an instruction set simulator, that can be extended with further RISC-V standard extensions, very easy. The *rv32* ISS was explored as an example of this and used as a case study to look at some of the features in class inheritance in C++.

Once we had an ISS architecture, a method described how we could make this configurable to select which extensions we wanted in our compiled model and how this could be auto-generated with GUI based python script.

This document is not meant to be definitive on the use of C++ inheritance features or of how to architect a processor instruction set simulator model, but it's hoped that it serves as a more extensive example of both than is usually available as references. The source code for the model is fully available in the github repository for further study.