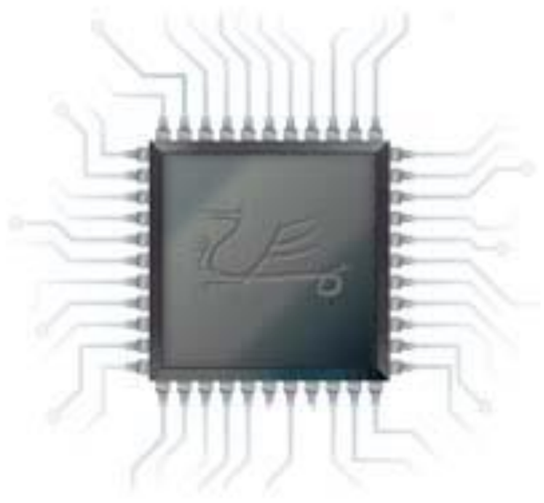
Modelling SoC Systems with C++



Simon Southwell

October 2023

Preface

This document brings together 2 articles written in October 20223 and published on LinkedIn, covering the software modelling of SoC systems in C++.

Simon Southwell
Cambridge, UK
October 2023

Contents

Part 1: Modelling Processing Elements

Introduction

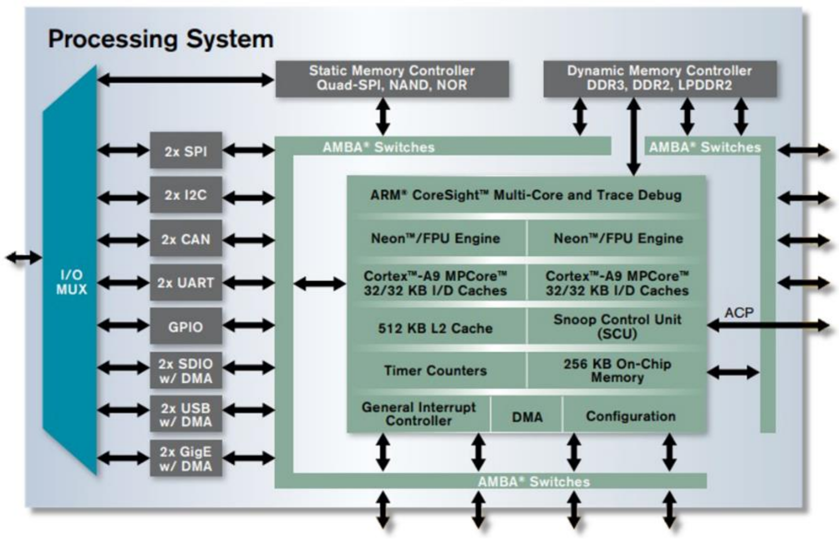
In this two part document I want to discuss the modelling of System-on-Chip (SoC) systems in software. In particular in C++, but other programming languages could be used. The choice will depend on many factors but, as we shall see, there are some advantages in modelling with a language that will also be the 'programming language' of the model. Modelling processor-based systems in software is not uncommon. In my own career alone, I have seen this done, to varying degrees, at Quadrics, Infineon, Blackberry, u-blox and Global Inkjet Systems and have been involved in constructing some of these models and used the models at all of them.

SoCs are generally characterised by having on a single chip many of the functions that might, in the past, have been separate components on a PCB, or set of PCBs. We can define some common characteristics of an SoC that we're likely to find on any device. The SoC will have one or more processor cores, and this immediately implies a memory sub-system, with a combination of one or more devices from ROM, Flash, SRAM, DDR DRAM etc., to facilitate running programs and operating systems. The cores may have caches to varying levels, and may support virtual memory, implying an MMU (memory management unit) or, if not VM support, at least memory protection in the form of an MPU (memory protection unit). Some memory mapped interconnect or bus will be needed for the core(s) to access memory and other devices, so an interconnect/bus system will be present, such as Amba busses (APB, AHB, AXI, CHI), or Intel/Altera busses (Avalon). Almost certainly, the processor will need to support interrupts, and an interrupt controller would then be needed for multiple nested interrupts with, perhaps, support for both level and edge triggered interrupts. If support for a multi-tasking and/or real-time operating system is needed, then a real-time timer that can generate interrupts will be present, along with other counter/timer functionality, including perhaps watchdog timers.

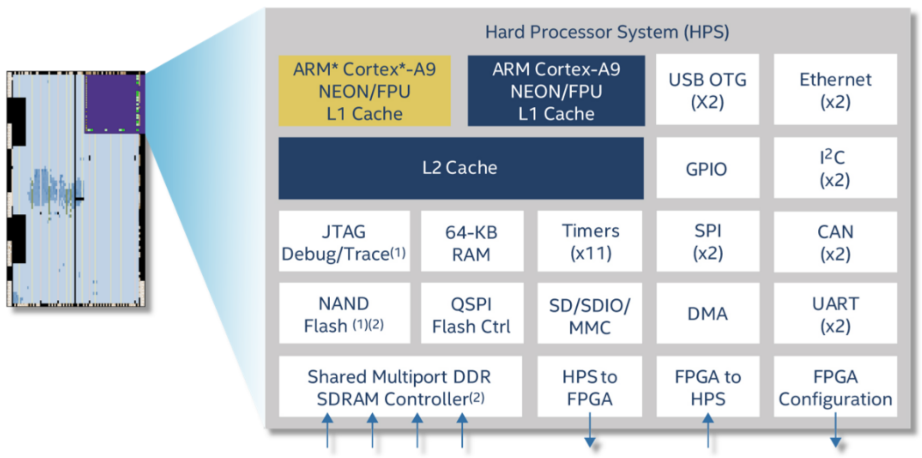
Once we have the processor system, with memory and interconnect, the SoC will need to interact with the real world via peripherals. These might be mapped on the main memory address map, but there may be a separate I/O space. The peripherals might be low bandwidth serial interfaces (UART, I^{2}C, SPI, CAN), or higher bandwidth interfaces, such as SDIO, USB, Gigabit Ethernet or PCIe. Moving data from the interfaces (especially those with high bandwidth) might require direct memory access (DMA) functionality, utilising streaming bus protocols. Encryption and security support may also be required. For control and status of external devices, a certain number of general purpose I/O (GPIO) pins may be supported, as well as analogue-to-digital converters (ADCs) and/or digital-to-analogue convertors (DACs). There may also be peripherals to drive display devices such as LCD displays.

Having a set (or sub-set) of these general-purpose functions, where required, custom functions, specific to the system being developed, can be added. In an FPGA based SoC these might be implemented in the logic part of the device. The diagrams below show two commonly used FPGA devices that have SoC hard macro logic (Custom ASIC type logic implementation): one from AMD and one from Intel.

AMD/Xilinx Zynq 7000



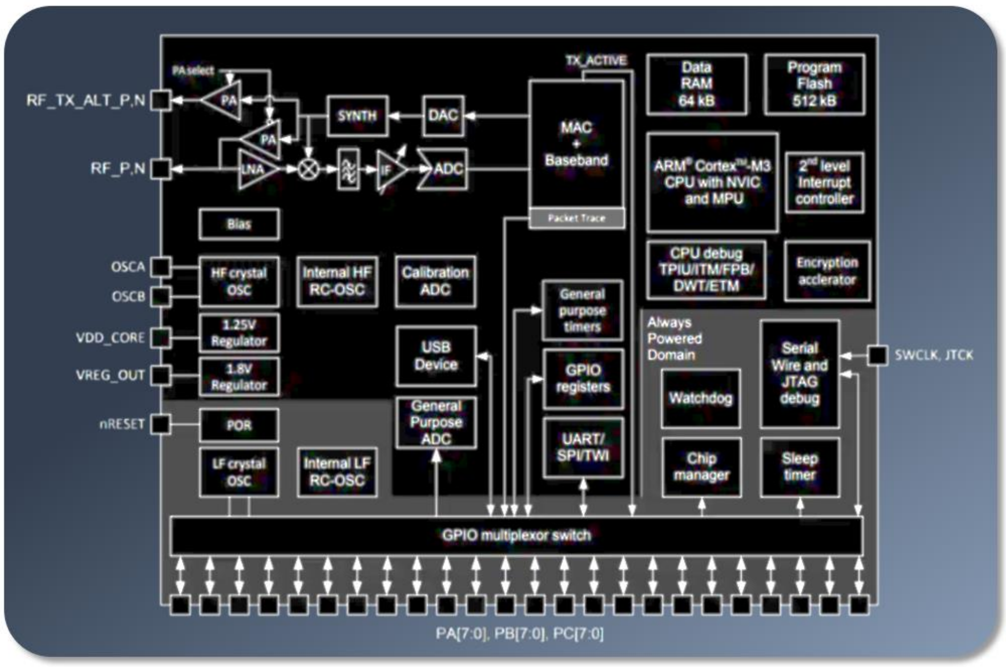
Intel/Altera Cyclone V



These two devices have very similar architectures and sets of SoC components. This is not surprising for two reasons. Firstly, they serve the same market and are competitors in that market, and secondly, they are based around the same processor system, namely the ARM Cortex-A, using the same Amba interconnect. They do, however, give a good reference point to what a generic SoC might look like, and what functionality is present. They contain a lot of options for interfaces and protocols, and a specific implementation may not use all of them, so any modelling of a given system need only model what is going to be present and used in the implementation.

SoCs are not restricted to FPGAs, and many ASICs follow this same pattern. I worked on a Zigbee based wireless ASIC which was also ARM based and had a smaller set of peripherals, but not dissimilar to those above so customers to adapt the chip for their specific application.

Silicon Labs EM358x



Having defined a typical set of functions one might find in an SoC, we find that there are a lot of complex things present, from the processor cores to the peripherals and all the functionality and protocols in between. How can we make a model that covers all this functionality?

If cycle accurate modelling is required then the model is likely to converge in complexity of the logic implementation and we need some strategies to simplify the problem or else the model development will rival the logic in effort and elapsed time. It is possible, if an HDL implementation of the logic is available, to convert this to a programming language. The Verilator simulator can convert SystemVerilog or Verilog to C++ or SystemC (a set of libraries for event driven simulation) which can be interfaced to other C++ based model functions. However, this rather negates some of the advantages of having a C++ model; namely, having a model on which software can be developed *before* logic implementation is available, speed of execution, and ease of system modification for architecture experimentation and exploration. So, is it worth making a software model of an SoC system at all?

In the rest of this part of the document, and the following part, I want to break down each of the functions we looked at for an SoC and look at strategies for putting together software models to quickly and usefully construct a system that can be used to develop software and explore a design space either before committing to a logic development or used in parallel with a development to shorten schedules and mitigate risk. We will begin by looking at ways to have a processing element on which we can run our embedded software.

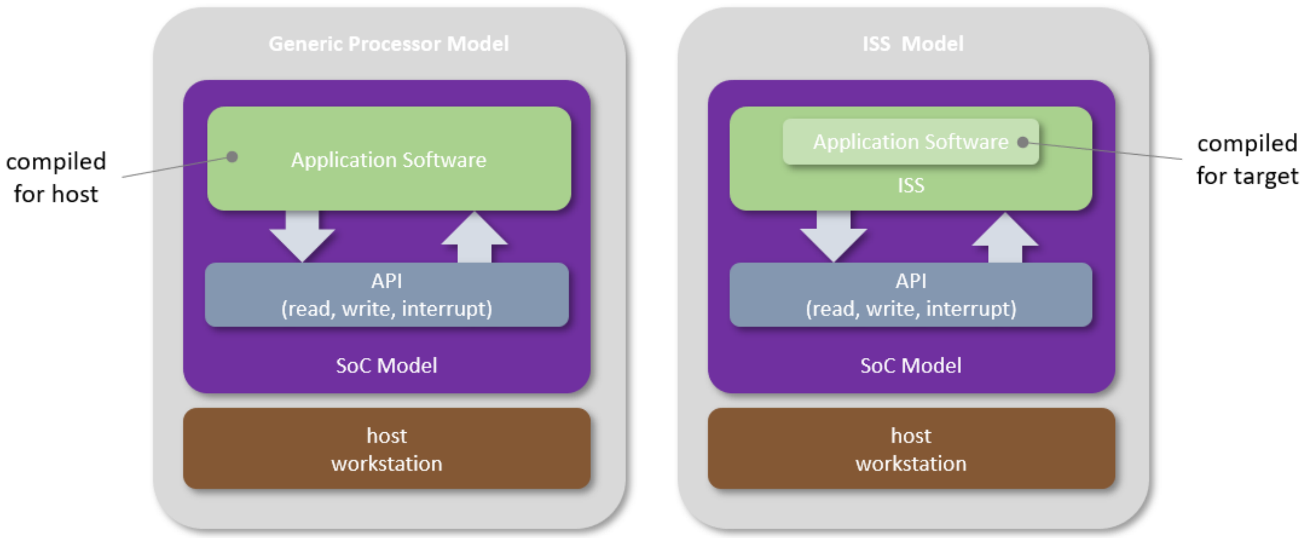
Modelling the Processing Element

The beating heart of an SoC are the processor cores. One of the motivations for building a software model is to execute software that is targeted at the final product. The software will run on one or more cores and, in general, have a memory mapped view of the rest of the SoC hardware. There may be a separation of memory and I/O spaces, but this is just another simple level of decode. Other than the memory and I/O views, the only other external interaction the processor cores usually have is that of interrupts. This may range from a single interrupt (e.g., RISC-V), where an external interrupt controller can handle multiple sources of interrupt, to having a vector of interrupt inputs (e.g., ARM Cortex with built in NVIC). Actually, in either case, the interrupt controller could be modelled as an external peripheral. What remains is what I call the 'processing element', with just this memory and interrupt interfaces. This simplifies what we have to model considerably. The next question is, what processor do we need to model? There are two answers to this, one of which is obvious, and the other not so obvious:

1. Model the processor that is targeted for the product.
2. Don't model the processor.

The first answer is, I hope you'll see, the obvious answer, and we will look at constructing instruction set simulators later, including with timing accurate modelling.

The second option is not so obvious. Whatever we model, we want to run a program that can read and write to memory (or I/O space) and be interrupted, just like a real processor core. If we present an API to that software so that it can do these memory accesses and have interrupt service routines called when an external interrupt event occurs, then we are close to a solution. The model, presumably, is compiled and run on a PC or workstation, likely compiled for an x84-64 processor. Even if the embedded software is targeted for a different processor, such as a RISC-V RV32G processor, then it might still be possible to cross-compile it for the model's host machine—especially if steps are taken to ease this process, as we will discuss shortly. This saves on constructing a specific processor model, which requires a good understanding of the processor's instruction set architecture (ISA), or when no third party model is available. Since an instruction set simulator is, itself, just a program, once we have a generic processing element model, we can simply make the program we run on it an ISS and, voila, we have a model that can run code for an architecture other than the host computer's processor. The diagram below summarises these two cases:



Hopefully it is clear that one is, in general, just an extension of the other and that taking a generic processor route has an 'upgrade path' for more accurate processor modelling as the next logical step.

In the next section I want to look at this generic processing element approach, before looking at methods for constructing instruction set simulators for specific processor instruction set architectures (ISAs).

Generic Processor Models

The question on whether to take a generic processor model approach or use an ISS is really down to the timing accuracy required of the model. With an ISS, instruction execution time is usually well documented and can be built into the model, as we will see when discussing this approach. For a generic solution, we can still have timing models, but these will be more crude estimates based on statistical modelling (or educated guesses). None-the-less, this may still be very useful in constructing the embedded code and running on a model with the desired peripheral functionality.

Memory Accesses

It's fair to say, I think, that most SoC processors memory accesses will be done through the equivalent of load and store instructions of fairly limited functionality, perhaps being able to load or store from bytes to double words etc. From a software viewpoint, this is largely hidden (unless writing in assembly language), and the software manipulates variables, arrays, structures, class members etc. In a generic processor model these memory data structures can just be part of the program and reside on the host. It gets interesting when accessing memory mapped peripherals and their registers.

The simplest API for accessing memory mapped space within the SoC model is perhaps a pair of C like functions, or C++ methods in a class, to read and write such as shown below (assuming ultimately targeting a 32-bit processor):

```
uint32_t read_mem (uint32_t addr, access_type type, bool &fault);
void     write_mem (uint32_t addr, uint32_t data, access_type type, bool &fault);
```

The `type` argument defines the type of access—byte, half-word and so on. Of course, these will be wrapped up as methods in an API class, and there may be I/O equivalents. This isn't an document on C++, but the functions could be overloaded so that the type of data (the return type for `read_mem`, and the `data` argument for `write_mem`) could define the type of access, dropping the `type` argument. Where possible I will avoid obfuscating the points being made with this kind of 'best practice' optimisations. When writing your own models, you should use good coding style (and comment liberally), but I want to keep things simple. You can, of course, write the whole thing in C, and the embedded code to be run on the model may well be in that language in any case.

In many of the embedded systems I have worked on, the software has a virtualising layer between the main code and accessing the registers of the various memory mapped hardware. This is a Hardware Abstraction Layer (HAL) and might consist of a set of classes that define access methods to all the different registers and their sub-fields—perhaps one per peripheral—built into a hierarchy that matches that of the SoC. I.e., a sub-unit may consist of a set of peripherals, each with their own register access class, and even, perhaps, some memory, gathered into a parent class for the sub-unit. The advantage here of having a HAL is that it can be used to hide the access methods we defined above and make compiling the code for both the target and the host running the model that much easier. Ultimately, the HAL will do a load or a store to a memory location. If we arrange things so that, when compiled for the target, the HAL simply makes a memory access (`a = *reg` or `*reg = a`), but when compiled for the model references the methods (`a = read_mem(reg, WORD, fault)` or `write_mem(reg, WORD, fault)`) then the embedded software gets the same view of the SoC registers whether running on the target platform or running on the generic processor as part of the SoC model. Indeed, this was done at one of my employers and the HAL was automatically generated from JSON descriptions, as was the register logic, ensuring that the software and hardware views agreed. Again, avoiding C++ nuances, it is possible (for those interested) that if the register types are not the standard types (e.g., `uint32_t`) but a custom type, accesses such as `a = *reg` or `*reg = a` can be overloaded to call the read and write methods, so retaining pointer access. This is more complicated, and a HAL would virtualise this away anyway, making it unnecessary.

Whether overloading pointers, using a HAL, or just calling a read and write API method directly, from a software view we have an API for reading and writing to a memory mapped bus. We haven't discussed what goes in these methods yet, but we will get to this when we talk about modelling the bus/interconnect.

Interrupts

The other interface to the model of a processing element we identified was for interrupts. Notoriously, on a PC or workstation, when running user privilege programs we don't have access to the computer's interrupts directly. Fortunately, we do not need to.

In a real processor core, at each execution of an instruction, the logic will inspect interrupt inputs, gating them through specific and then master interrupt enables and if one is active, and enabled, will alter the flow of the program in accordance with the processor's architecture. Thus the granularity of an interrupt is at the instruction level. For our generic processor model, we aren't running at the instruction level, but just running a program on a host machine. We do, however, access the SoC model with the read and write API calls. Since the SoC model will be the source of interrupts, this is a good point to inspect the current interrupt state. Glossing over just how that state might get updated for the moment, so long as, at each read and write call, the interrupt state can be inspected, we can implement interrupts and have interrupt service routine functions.

If the read_mem and write_mem methods of the memory access class call a process_int method as the first thing they do, then this can keep interrupt state and make decisions on whether to call an interrupt service routine (ISR) method. The main program is stalled at the memory access call whilst the ISR is running, and so will return to that point when the ISR method exits. The ISRs themselves can access memory and can also be interrupted by higher priority interrupts allowing hierarchical interrupt modelling to be achieved. A sketch for an API class with interrupts is shown below:

```
class ApiWithInterrupts
{
public:
      static const int max_interrupts = 32;

      ApiWithInterrupts () {
          int_active        = 0;
          int_enabled       = 0;
          int_master_enable = false;

          for (int idx = 0; idx < max_interrupts; idx++) {
              isr[idx]      = NULL;
          }
      };

      void  write_mem (uint32_t addr, uint32_t  data, access_type type, bool &fault) {
           process_int();
          /* write access code */
      }

      uint32_t read_mem  (uint32_t addr, uint32_t *data, access_type type, bool &fault) {
           process_int();
          /* read access code */
      }

      void enableMasterInterrupt  (void);
      void disableMasterInterrupt (void);

      void enableIsr    (const int int_num);
      void disableIsr   (const int int_num);

      void updateIntReq (const uint32_t intReq);
      void registerIsr  (const pVUserInt_t isrFunc, const unsigned level);

private:
      void process_int();

      pVUserInt_t isr[max_interrupts];
      uint32_t    int_enabled;
      bool        int_master_enable;
      uint32_t    int_req;
      uint32_t    int_active;
};
```

Here we have a class with the two methods for read and write, and I've shown these with some code to show that an internal process_int method is called before actually processing the access. The class contains some state, with an array of function pointers, set to NULL in the constructor, which can be set to point to external functions via the `registerIsr()` method. A master interrupt variable, `int_master_enable`, can be set or cleared with `enable-` or `disableMasterInterrupt` methods. Similarly, the individual enables can be controlled with `enableIsr` and `disableIsr` methods. To actually interrupt the code, the `updateIntReq` method is called with the new interrupt state, which would set the `int_req` internal bitmap variable, which `process_int` will process. A bitmap `int_active` variable is also used by `process_int` to indicate which interrupts are active (i.e., requested *and* enabled). There can be more than one active, and the highest priority will be the one that is executing.

This type of method is used with the OSVVM co-simulation code and I write about how this is done with more detail in a blog on that website. In this environment there is an `OsvvmCosim` class with, amongst other methods, a `transRead` and `transWrite` methods. This is used as a base class to derive an `OsvvmCosimInt` class, then overloads the `transRead` and `transWrite` methods (and others) to insert a `processInt` method call which models the interrupts. The ISRs don't reside within the class, but external functions can be registered by the user to be called for each of the ISR priority levels. The blog gives more details and the referenced source code can be found on OSVVM's github repository, so I won't repeat the description here, and the details of the `processInt` methods of that code serves to show how this would be done with the sketch class described above, and its `process_int` method.

So here we have a framework to build an SoC and run a program. We have defined a class with read and write capabilities and the ability to update interrupt state and have the running program interrupted with prioritised and nested interruptable interrupt service routines, provided externally by registering them with our class. We can now write a program and a set of ISRs that uses this class to do memory mapped accesses and support interrupts. I've left off the details of the `read_mem` and `write_mem` methods, for now, as this is how we will talk to the rest of the model which will be dealt with in the next part of the document.

Instruction Set Simulators

With the class defined from the last section we can write arbitrary programs and interact with the rest of the model (when we get that far). Of course, that arbitrary program could just be an instruction set simulator (ISS). One difference is that the granularity of interrupts will be at the instruction level, rather than the read and write memory level, and the ISS model of the processor itself, in some cases, will be contain the interrupt handling code. Thus the API class we defined before simplifies considerably, with the read and write methods no longer requiring a `process_int` call, and all the code associated with interrupts disappears. We still need to inspect interrupt state but, as we shall see, a slightly different method is used. In the OSVVM code, the non-interrupt class (`OsvvmCosim`) is defined as a base class, and then a derived class (`OsvvmCosimInt`) overloads the read and write methods to insert an interrupt processing method at the beginning of each one, and then call the base class's read or write method. If this split was done to the class from the last section, then the base class could be used for an ISS, which wouldn't need the interrupt functionality externally. In the rest of this section I want to outline the architecture of an ISS model which is largely common to modelling any processor's ISA.

Just as for a logic implementation, we have some basic operation we must implement:

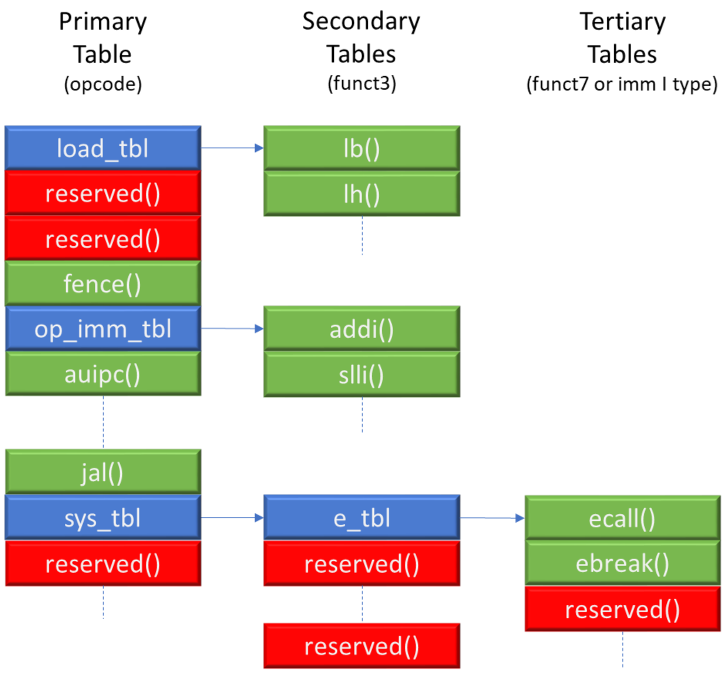
- Reading an instruction
- Decoding the instruction
- Executing the instruction

These three basic functions are repeated, in an execution loop, indefinitely or until some termination condition has been reached, such as having executed a particular number of instructions, executed a particular instruction (like a break for example) or some such state, set up prior to running the processor. In addition to these basic functions, some state also needs to be modelled for things like internal registers and the program counter. These can all be collected into a processor model class.

For reading an instruction we are already set up, as we can use our API with the `read_mem` method to read instructions preloaded into memory though, as we'll see later, this will be done via an indirection. For RISC type processors, only one word is read per instruction such as ARM, RISC-V and LatticeMico32 processors. Therefore the decode process is completely isolated from the other steps. This is the case for my RISC-V and LatticeMico32 ISS models. For non-RISC, usually older, processors, instructions may be variable in length, with a basic instruction opcode followed by zero or more arguments. Therefore a decode, or partial decode needs to be done, and then any further bytes/words read before moving to execution. Thus, reading and decoding can be entwined somewhat, complicating the first two stages, though only mildly so. This is the case for my 6502 processor and 8501 microcontroller models.

Decoding an instruction will involve extracting opcode bits to uniquely identify the instruction for execution, with the other bits being 'arguments' such as source and destination registers, immediate bits and the like. Depending on how many opcode bits the processor's ISA defines will determine how many possible unique instructions there can be, though they might not all decode to a valid instruction. The number of opcode bits might be quite small, such as for the LatticeMico32, which has 6 bits and 64 possible instructions and the 8051 which has 8 bits for 256 possible instructions. For other processors it may be much higher and the RISC-V RV32I processor's R-type instructions have 17 bits (see my article on RISC-V for more details). Many ISS models I have seen use a switch/case statement for the decoding. For the small opcode processors, like the LatticeMico32 with 6 bits, a switch statement with 64 cases to select the execution code is manageable. For the larger opcode spaces, such as the 17 bits of RISC-V RV32I, this then becomes 131072 cases most of which will be invalid instructions. To manage all of the different architectures, I prefer to use a hierarchy of tables which have pointers to instruction execution methods as part of each entry. For the smaller opcode spaces, this table hierarchy can be one deep (i.e., a single flat table), but for the large spaces this is broken down. The RISC-V instruction formats have a common 7-bit opcode, and then have various other funct*X* fields of various sizes, such as a three bit funct3 or a seven bit funct7 fields. We can use this to produce a hierarchy. An initial primary table can be made with the number of entries for the opcode (i.e., 128). Each entry in the table can have a flag saying whether it is an instruction, and then has a pointer to an instruction execution method, or points to another table. A secondary table would have entries for the funct3 field, and a tertiary table would have entries for the funct7 field. This can be repeated for any depth required. Decoding then walks down the table until it finds an instruction entry.

The diagram below, taken from the RV32 ISS Reference Manual, shows this situation.



So what might each table entry look like? Here we define a structure (class) to group all the relevant information and make an array of these structures for the tables. The code snippet below shows a top-level structure for the rv32 ISS.

```
typedef struct rv32i_decode_table_t
{
    // Flag to indicate 'ref' a sub-table reference (and not an instruction entry)
    bool                                                sub_table;

    // Either a reference to an instruction entry or a reference to a sub-table
    union {
        rv32i_table_entry_t                             entry;   // A decoded entry
        rv32i_decode_table_t*                           p_entry; // A pointer to a sub-table
    } ref;

    // Pointer to an instruction method
    pFunc_t                                             p;
} rv32i_decode_table_t;
```

This structure has the sub-table flag, a union of either a pointer to a decoded instruction data structure or to another table and then a pointer to an instruction execution function (which is null if a sub-table). The decoded instruction data structure is all the fields of the instruction extracted out, which is ‘filled in’ by the decode code. Since this will be passed to all the instruction execution functions, it contains all possible fields for all instruction types, so that the instruction execution methods can simply pick the out appropriate values they need.

```
typedef struct {
    uint32_t            instr;          // Raw instruction
    uint32_t            opcode;         // Opcode field
    uint32_t            funct3;         // Sub-function value (R, I, S and B types)
    uint32_t            funct7;         // Sub-function value (R-type)
    uint32_t            rd;             // Destination register
    uint32_t            rs1;            // Source register 1
    uint32_t            rs2;            // Source register 2
    uint32_t            imm_i;          // Sign extended immediate value for I type
    uint32_t            imm_s;          // Sign extended immediate value for S type
    uint32_t            imm_b;          // Sign extended immediate value for B type
    uint32_t            imm_u;          // Sign extended immediate value for U type
    uint32_t            imm_j;          // Sign extended immediate value for J type
    rv32i_table_entry_t entry;          // Copy of instruction table entry
} rv32i_decode_t;
```

The decode table arrays are constructed and filled by the constructor, to link the table hierarchies and point to the instructions’ execution methods. In the execution loop, decoding does the ‘table walk’, indexing down the table with the opcode and funct*X* values as indexes, until it reaches an instruction execution entry. The entry decode table is filled in from the raw instruction value, and then the instruction method, pointed to in the entry, is called with the entry decode table as an argument. If the pointer is pointing to a ‘reserved’ method, then the decoding reached an invalid/unsupported instruction, and an exception can be raised.

The instruction execution methods are now simply a matter of executing the functionality of the instruction. Below is an example of an add instruction method:

```
void rv32i_cpu::addr(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] +
                                         (uint32_t)state.hart[curr_hart].x[d->rs2];
    }

    increment_pc();
}
```

As you can see, this is now fairly straight forward. The branch and jump instructions will update the PC (with the former on a condition), whilst the load and store instructions will do reads and writes. Now these could use the API class that we defined earlier, but a better way (as I hope I will convince you) is to have internal read and write methods which call an external callback function registered with the ISS model. (If you are not familiar with pointers to functions and callback methods, I talk about these in one of my articles on real-time-operating systems, in the *Asymmetric Multi-Processor* section.

The reason I think this is better is because it decouples the ISS completely from the rest of the SoC model, allowing the ISS to drop in to any SoC model, which can register its external memory access callback function and run code on the ISS for that environment. The rv32 ISS read and write methods check for any access errors (such as misaligned addresses) and then executes the external memory callback function if one has been registered. If one hasn't been registered, or if the call to the callback returns indicating it didn't handle the access, the ISS will attempt to make an access to a small 64Kbyte memory model it has internally. If the access is outside of this range, then an error is generated.

Many (but not all) of the instructions can generate exceptions and in the rv32 ISS a `process_trap` method is defined to handle these called, as appropriate, from the instruction execution methods, with the trap type. The `process_trap` method simply updates register state for the exception and sets the PC to the appropriate exception address. Since interrupts are forms of exception, we can also have a `process_interrupts` method. This, though, is not called from the instruction methods, but is in the execution loop so that it is called every instruction. Some processors have a mixture of internal and external interrupt sources. So, for example, the RISC-V processor can generate timer interrupts internally (ironically from timers that are allowed to be external), whilst also have an external interrupt input. In order for the ISS to be able to be interrupted by external code we, once again, use an externally registered callback function. At each call to `process_interrupts` this callback (if one registered) is executed and returns the external interrupt request state. This is then processed against interrupt enable state and, if enabled, a call to the `process_trap` is made, with an interrupt type instead of an internal exception type and the PC will be altered similarly to that for an exception.

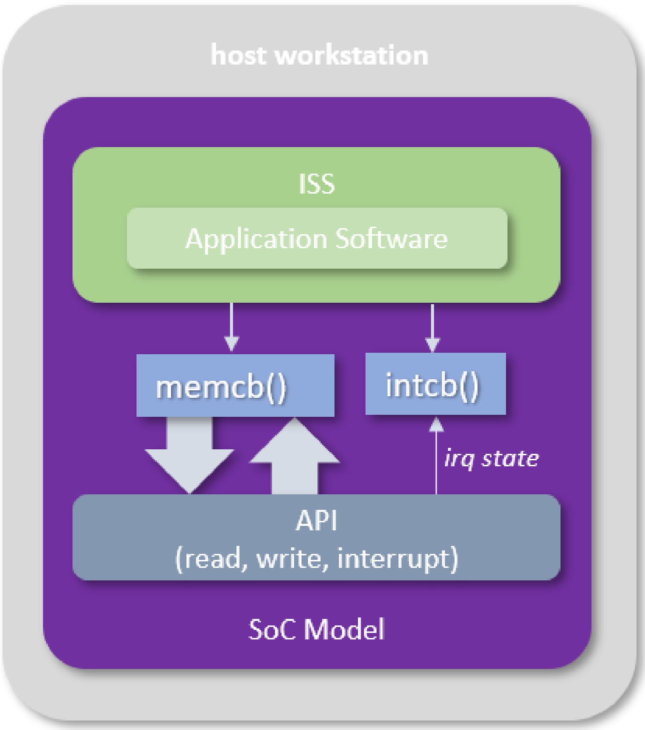
So we now have all the components for the basic functionality, with decode, execution, memory access and exception/interrupt handling. To run an actual program we just need a run loop.

```
    while (!halt) {
        if (!process_interrupts()) {
            // Fetch instruction
            curr_instr = fetch_instruction();

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

            // Execute
            if (p_entry != NULL)
                error = execute(decode, p_entry);
            else
                process_trap(RV32I_ILLEGAL_INSTR);
        }
    }
```

We can now refine our ISS diagram a bit with the callback functions provided by the external model software, called by the ISS for memory accesses and inspecting interrupt state, and then these callbacks making use of the API we defined earlier.



Timing Models

If modelling timing to some degree is important (and it may not be), then what is possible will depend on the choice of whether using a generic model or a processor specific model. With the generic model the software running on the processing element is just a program running on the host machine and only when it interacts with the rest of the model (e.g., do a read or write) is there any concept of the advancement of 'model' time. Of course, if there is some data on the average mix between memory access and non-memory access instructions in a similar real-world system, then an estimate for clock cycles run between calls to the API read and write methods can be made, and some state kept that's updated when calls to the methods made. It will, necessarily, be a crude estimate, but may be a useful approximation of how a system might perform. However, the generic method is not really suitable for more accurate performance measurements.

With an ISS things become a lot better. Processor execution times are usually well understood and documented for a given implementation. For example, from the documentation of the rv32 RISC-V softcore we have:

- 1 cycle for arithmetic and logic instructions
- Jumps take 4 cycles
- Branches take 1 cycle when not taken and 4 cycles when taken
- Loads take 3 cycles plus wait states
- Stores take 1 cycle plus wait states

This is a fairly straight forward specification, and the ISS instruction execution functions can update some count state to keep track of cycle time. The exception is the memory access instructions which also have a wait state element. This wait delay is a black-box as far as the processor is concerned, as it is caused by the external modelling of the memory sub-system. Therefore, the memory callback function prototype specifies a return value which is the additional cycles added by the memory access, and the SoC model memory functionality will calculate this. Thus, the memory access instructions will add their base timing to the cycle count, and the callback return value will then be subsequently added. From a processor point of view, then, we have cycle accurate behaviour. Of course, if the core has complex features, such as out of order execution, dynamic branch prediction, or is superscalar in architecture, accurate cycle counts become harder as the model must take these factors in to account.

Debugging

With my software hat on, the first question I might ask when presented with a software model of a system to program is, how will I debug my code? Here, I'm talking about the code that is running on the model, rather than the code that *is* the model—which can be debugged using the normal tools and methods as it's just an application running on the host machine. In fact, for the generic processor model, the code to be executed is just an extension of that model code, perhaps usefully separated from the model code, but compiled and linked with it. So the same techniques and tools can be used here and the issue is sorted.

For an ISS based model things are a little bit more complicated—but not much. Here the software running on the model (using the ISS) is probably cross-compiled for the particular processor, and the host tools can't be used and one would need to use those supplied with the toolchain for that modelled processor architecture. However, taking `gdb` as a common debug tool, it has a remote mode where it can connect to a processor remotely, via a TCL/IP socket, and then send 'machine' versions of the common commands to load programs, set breakpoints, run code, inspect memory, step and continue the program etc. If the ISS model has some TCP/IP server code that can receive and decode the `gdb` machine commands, and then act appropriately, sending any required responses, a debugging session can be set up using the processor's version of `gdb`. This has, in fact, been done for my RISC-V and LatticeMico32 ISS models, and the source code can be inspected for how this is implemented. The LatticeMico32 ISS documentation has sections on the `gdb` interface, and an appendix on how to set up the Eclipse IDE with `gdb` so that a full IDE can be used for debugging. So, now we have a full debug environment and can debug code.

Multiple Processing Elements

Many embedded systems have multiple processor cores, and we may need to model this. With the generic processor model, we might ask, do we really need to model multiple cores as the code is just host application code? The main motivation here is that the system may be set up to have different functionality on each core, rather than just have them as a pool of processing resources to run processes and threads as allocated by the operating system. For this latter situation, maybe no further modelling is needed for multiple cores. Even if the embedded code is multi-threaded, these can just be threads running on the host. The only issue to solve is that multiple threads accessing the memory read and write API will need to be done in a thread safe way. This might be wrapping the API calls with mutexes to make sure any access is completed atomically by each thread. For the case with each core performing different tasks, the source code is likely to be structured with this split, and so these could be run as separate host threads and the same methods used to ensure thread safe operation. Again, as for the timing models, the generic processor model will stray from accurate and predictable flow of code when modelling multiple cores in this way, and it is mainly for software architecture accuracy, with the aim to be able, as much as possible, to compile and run the same code on the model as for the target platform.

For the ISS based modelling we don't need to rely on threads or mutexes and can maintain a single threaded application. How? It was implied, when discussing debugging, that the ISS model is able to be stepped one instruction at a time. The run loop code snippet shown earlier had a '*while not halt*' as the main loop, where *halt* is doing a lot of heavy lifting. Actual real code will have a whole host of possible reasons to break out of the loop, allowing breaks on reaching a certain address in the instructions (a break point) or after a certain number of instructions have been run, such as 1 (a step). We can use this step feature to advance the processor externally instead of free running. Now a run loop can have calls to step multiple ISS objects and step them in sequence. With access to the ISS objects' concept of time, the execution order can be improved by, at each loop iteration, only step the ISS object that has a smaller cycle count time. This, then, minimises the error in cycle counts between the processor models and keeps them synchronised. This is discussed in more detail in the LatticeMico32 documentation, under the *Multi-processor System Modelling* section, for those wanting to know more about this subject. This method can be done for any number of processor cores required to be modelled and can even be done with different processor models, which is not an uncommon situation in some embedded systems.

Conclusions

We have started our look at constructing software models of SoC systems by looking at ways to allow us to run embedded software on a 'processing element'. This might be a virtual element where a specific processor isn't modelled, or an instruction set simulator, where the target processor is fully modelled. In either case we present an API that does the basic processor external operations—read and write to memory space and get interrupted.

For the generic model, the API can be used directly for reads and writes, and strategies were discussed to model nested interrupts whilst maintaining single threaded code. Timing modelling with the generic processor model was shown to be limited in accuracy but may have some useful application with real-world based estimates.

The ISS modelling was broken down to mimicking the basic steps of a core and we looked at using a table hierarchy for instruction decoding and the use of pointers to instruction execution methods in the table, to be executed when the decoding terminates in the decoded instruction entry within the tables. To de-couple the processor models from the rest of the modelling, the API was not used directly, but callback functions used for memory space accesses and inspecting interrupt request state, which can then use the API, allowing modelling of the bus and interconnect to be external to the processor model, which we'll discuss in the part of the document. Methods were also discussed for accurate timing models and debugging, as well as how to handle the modelling of multiple processor cores.

In this first part, we have just focused on processor modelling, and we still have to look at the bus/interconnect, memory sub-system and all the various peripherals, as well as looking at interfacing to external programs to extend the model's usefulness into such domains as co-simulation or to interface to other external models. I will cover these subjects in part 2 of the document.

Part 2: Infrastructure

Introduction

In the part 1 in this series, we looked at the modelling of the processing element of an SoC. In this second part I want to talk about the rest of the system model. When writing this second part, the size of what I wanted to cover in this subject grew and grew, and I had to start trimming the details down in order to get a document that was of digestible length, knowing that many reading this document are new to the subject, or less familiar with software than they are with logic. So, I have made this second part into an overview of approaches but have tried to compliment this with references to examples that I have made available in the open-source domain, and of other relevant documents and articles that describe how these are constructed so that people can dive deeper to the level of their interest. I have, professionally, been involved in modelling using additional techniques than covered here for which I don't have examples available. I have mentioned a few which extrapolate from the techniques described below but have tried to keep this to a minimum. I hope I've succeeded, though, in laying a foundation from which someone can start constructing their own models.

We will do ourselves a great favour if, when constructing the rest of an SoC model, we separate out certain aspects of the blocks we wish to model:

- Basic functionality
- Specific protocols
- Timing

Usually (not always) the basic functionality of an SoC block is easy to describe. For example, an AXI bus facilitates a read or a write of data between (using ARM terminology) a 'manager' (such as a processor) and a 'subordinate', as indexed with an address. This is also true of the AHB, APB, Avalon bus, or wishbone or any memory mapped bus or interconnect protocol. There is a 'transaction' between the manager and the subordinate in the data exchange. It would be easiest to model at this transactional level. The details of the protocol really only manifest themselves at this transaction level as timing for the transaction and some specific facilities, such as privilege level—which may be common between other similar protocols or may not.

By separating out the different aspects of the functionality like this we can restrict what is modelled to meet the specific needs. If transactional modelling meets requirements, then a model of a block can more constructed more easily and quickly.

Timings would then, of course, be approximations—if timing is of interest at all. To get more accurate timings, the certain parts of the block's detailed functionality may need to be modelled. This might be at the level of "operation *a* takes *m* cycles, operation *b* takes *n* cycles" etc., if the functionality is that linear, all the way to a full blown cycle accurate model with full protocol modelling. This latter is most important if it interfaces with another model that needs the proper protocol, for example your interfacing to a 3rd party model that expects the received bits to be fully fledged ethernet packets, say. There are no short cuts, and the requirements on the model will depend on the intended use from purely functional to allow initial software architecture testing, to the ability to get accurate estimates for software and hardware performance. With separation of these aspects, complexity and features can be added over time to refine the model, whilst still having a system up and running before all aspects are implemented.

In this document it is not possible to go through every single SoC type block and discuss in detail how to model these. I want to mention the most common and reference some of my open-source code that uses these methods by way of working examples. The open-source examples can't be as complete as a whole professional system (I'm only a one man band) such as the systems I worked on at Infineon as a software modelling developer. Here, for instance, the idea was to have a GUI where you could pick components from a library—core, bus, memory, peripherals as required, etc.), and then automatically generate a software model from this, ready to run embedded software. Also, the 8051 model was used at Silicon Labs when I specified a 'soft MAC' for the Zigbee systems and modelled a solution, using the 8051 model to Silicon Lab's product specs, which could run some prototype embedded code that the software team had done some time before. Naturally, the code for these system models is not in my possession or open-source. What I'm hoping is that I have enough examples to show how to approach modelling an SoC, starting with an easy base system which can be quickly constructed, especially if the embedded software architecture is constructed with software modelling in mind, to adding complexity as and when needed.

FPU

In part 1, we left modelling processor's behind. However, we didn't talk about floating point functionality. Many embedded processors don't support FP arithmetic, but many do. From the modelling perspective, though, we can treat floating point support as a separate extension.

Naturally, when modelling a processor in C++ running on a host, access to floating point functionality is built into the language, so the FP instruction code can just do floating point arithmetic using the `float` and `double` types as appropriate. For many cases this is all that's required. However, not in all cases.

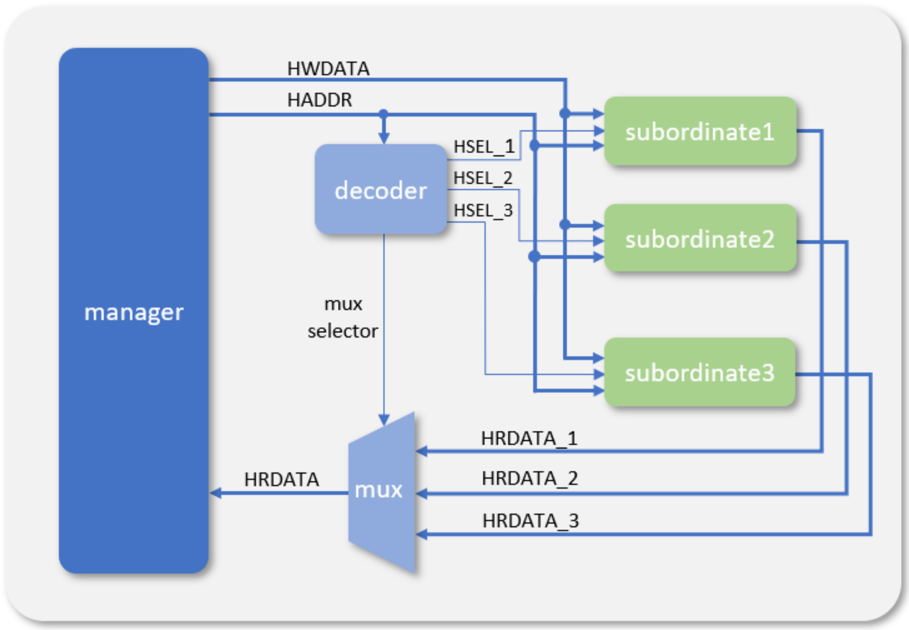
The RISC-V F and D extensions specify different rounding modes. I won't go into details (see the *The RISC-V Instruction Set Manual Volume I* sections 8 and 9), but rounding modes can be set for things like rounding towards zero, rounding down etc. By default, your host will do one of these, but to get accurate results we need to round as appropriate to the configuration. Fortunately, the `fenv` library provides a means to program which mode is used by the processor running our program via the `fsetround` function. The rv32 RISC-V ISS uses this library to set the rounding mode to match that programmed for the processor (see the `update_rm` method in the file `iss/src/rv32f_cpu.cpp`).

Another option which I've used before, professionally, is the softfloat library. This is a standalone library to implement IEEE standard for floating point arithmetic. The advantage of this is that it is agnostic to the features of the host computer that the SoC model is being run on. The disadvantage is that it is slower than using the host's floating point capabilities, if that is an issue. The choice will depend on the priorities of the model performance and features.

Bus and Interconnect

As was outlined in the introduction, the functionality of a memory mapped bus or interconnect is really quite simple, it does a transaction between a source requestor and, ultimately, a completer. There may be layers and hierarchies between these two points, but the destination is uniquely identified by an address. In an AHB system, for example, a decoder will activate one of several chip-select lines, based on some bits of the address.

Simplified AHB configuration



The subordinate may be a peripheral but can also be a device for further decoding, such as a bridge, between AHB and APB for example. So, the modelling of this can be as simple as a switch-case statement to select between a set of appropriate methods to access the subordinate. If the subordinate has further decode selections to make, then this is repeated to reflect the hierarchy. Even with a point-to-point interconnect protocol, such as AXI, the interconnect is usually a crossbar type architecture, but is still connecting between a subordinate and a manager by decoding the address bits. Ultimately, the peripheral model will have a set of registers (or maybe a memory) and the final decode to individual registers and bitfields is done there. If an interconnect has multiple input ports for multiple cores and other manager devices, then this is just the situation of multiple processors discussed in part 1 of the document.

Thus, modelling the bus and interconnect functionality is fairly straight forward. Layering on timing and protocol to this model will involve calculations due to the nature of the transactions, type of operation and any arbitration resolution. The accuracy of this is dependent on the effort for accurately modelling the protocol used and the device doing the decoding or routing. With all the models I have encountered in my career, none have actually claimed a 100% cycle accuracy with the real bus system. This is because this would involve having every event from external sources happening precisely as it would in the real system. Because of arbitration may go a different way if an access request is just one cycle different, and these errors accumulate. What is achievable though, is a system that behaves consistently with the exact same inputs over a model's run (repeatability) and is stochastically representative (precise) over many transactions, so it is worth putting good timing modelling in the bus models if performance measurements are required. Again, precision is a function of effort.

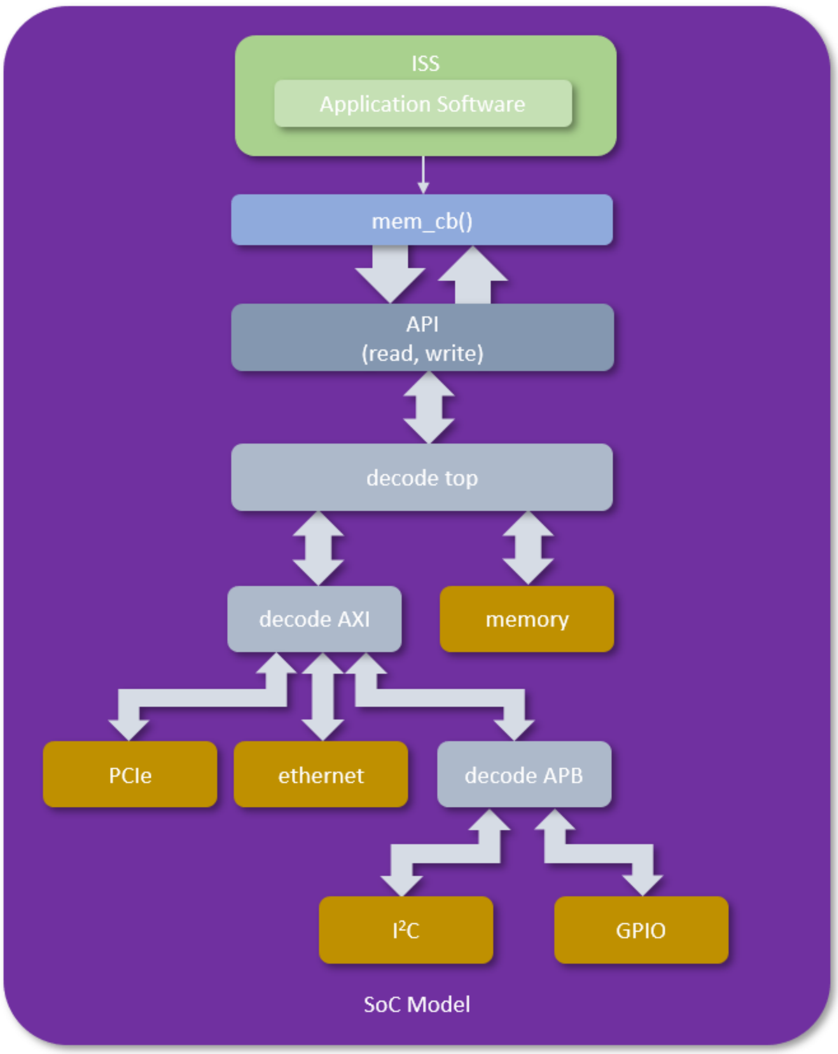
Connection to the Processing Element

In the first part of this document I mentioned the use of callback function from the processing element model which is called whenever it does a load or store (read or write) operation, but I did not elaborate on what that might look like in any detail. Taking the RISC-V model, for example, the callback type is defined as:

```
typedef int (*p_rv32i_memcallback_t) (const uint32_t     byte_addr,
                                      uint32_t           &data,
                                      const int          type,
                                      const rv32i_time_t time);
```

When a callback, of this type, is registered with the model (using the `register_ext_mem_callback` method) it will be called at each memory access by the processor with a byte address, a data value (on write type accesses), an access type and also the processor model's current time value. The `type` value takes one of some defined values in the form of either `MEM_WR_ACCESS_XX` or `MEM_RD_ACCESS_XX`, where `XX` is one of `BYTE`, `HWORD`, `WORD`, or `INSTR`. On read accesses, the value is returned in the `data` argument. The callback has a return value which is normally a cycle count (which can be 0) returning the number of *additional* cycles the access took over and above the processor instruction cycles, allowing wait state counts to be inserted, and these are added by the model to include in its elapsed time reckoning. If the callback returns `RV32I_EXT_MEM_NOT_PROCESSED`, then this lets the model know that the external call did not decode to any active block and get processed by it. The model will then try to see if it decodes to its own internal memory, otherwise it raises an exception. The 6502, 8501 and the LatticeMico32 models all have very similar mechanisms.

This callback, then, is the entry point for the rest of the model and where accesses to any read and write API functions can be made. For simple systems, the callback function itself could do the top level decode and call the read and write API calls for each modelled sub-block or, for more complex systems, a top level API (such as that discussed in part 1) could virtualize away the detail, and a hierarchical call structure made to reflect the system structure. The diagram below shows a simple example of this.



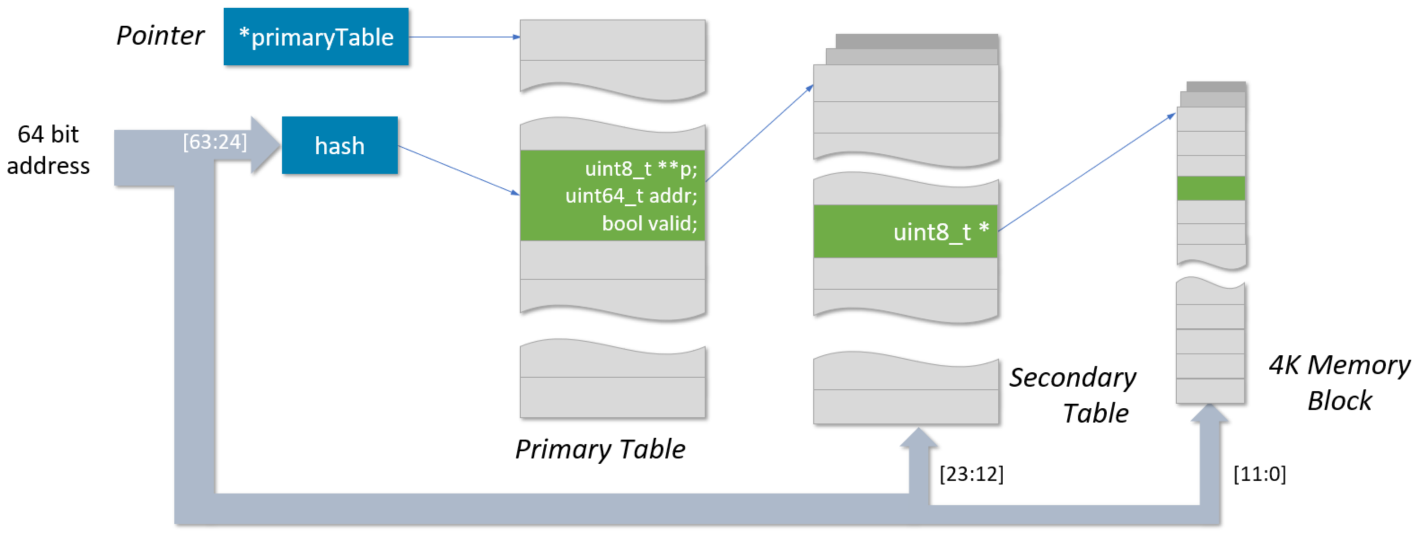
For a virtual processing element, the only difference from the above diagram is that the code modelling the embedded software calls the API directly, and has no callback.

This isn't an document on code structure, and I don't want to dictate how one might definitively go about putting together software models, and good judgment should prevail for each implementation. However, bear in mind that you could very well, in the future, want to re-use components from your model in another model that has a different mix of components, so keep re-use in mind when structuring your code.

Memory Model

The first major 'peripheral' we will definitely need is memory. A system may have more than one type of memory, such as DDR3 main memory, eeprom, flash, ROM etc. These, of course, could all be modelled as different components, each with its own memory model. From a processor's point of view, though, there is just one flat memory space (excluding any I/O space). Depending on how much detail of the different characteristics for each memory type are required, they could all be modelled as a single structure. This brings us to our first problem. A 32-bit processor has a 4Gbyte address space, and we can't simply create a byte array of that size (well, we could but perhaps shouldn't). Things only get worse when modelling a 64-bit architecture. An SoC system is unlikely to have populated the entire address space, but it may have main memory of 512Mbytes or more, and memory may be distributed within the memory map.

The solution is to have a 'sparse' memory model. This is where memory is allocated on demand as the memory is accessed, usually in blocks (or 'pages'). At start up, the model has no memory allocated, and the first access, with whatever address is provided, will break the address down to access a series of tables that eventually point to a block of memory, dynamically allocated for that access. Any subsequent access to the address space covered by that block will not cause another allocation but will simply access that block of memory. If a new access doesn't land on an allocated block a new one is created and so on. The diagram below, is adapted from the documentation of my memory model.



Here a primary table pointer is initially set to NULL. On the first access a new primary table is allocated with 4K entries. The top bits of the address are 'hashed' to generate a 12 bit number to index an entry in the primary table. The primary table entry has the address for the base of the space—that is, the provided address with only the bits (63 down to 24) used, and the others set at zero. A valid flag marks the entry as having been set, and then there is a pointer to a secondary table. At first access, there are no secondary tables, so memory is allocated for a table with 4K entries. The secondary table is just a set of pointers to byte memory. The entry is indexed by bits

23 down to 12 of the access address, and then that entry points to a dynamically allocated 4Kbyte space, with the bottom 12 bits used to index into there and this is the actual storage space where data is written and read back. As the model is accessed further, secondary tables and 4Kbyte spaces are created as needed, or previously created tables and memory blocks are accessed by walking down the primary and secondary tables, and then writing or reading the allocated memory block. Thus, we have a memory model that spans an entire 64-bit address space, but only uses the amount of memory actually needed, without having to configure it. More details can be found in the model's documentation. Any memory protection or invalid regions would be modelled externally to this model by, say, an MMU or MPU model or, at it's simplest a wrapper class which maps actively present memory and generates an error outside these configured regions.

The memory model was originally targeted for co-simulation with logic, but the core source code (in files `mem.h` and `mem.c`) is purely software and provides an API that allows for both processor type accesses (bytes, half-words, words, double words) but also allows for block accesses. This is important because most SoC main memory systems provide ports for burst accesses and DMA. The model also allows big- or little-endian storage and for multiple 'nodes' allowing more than one memory space to be defined. I would expect that the model (written in C) would be wrapped up in a class so that these different types of APIs could have exposure to the reset of the model, with checks and validations. This might be in the form of a 'singleton', so only one memory model is accessed via this singleton's API object that can be constructed where needed, effectively modelling different ports to the memory. No additional API functionality is provided for loading data, such as the code that might be in a ROM, but this would be loaded using the same API (with an 'instruction' type, say) prior to the model being executed.

Cache

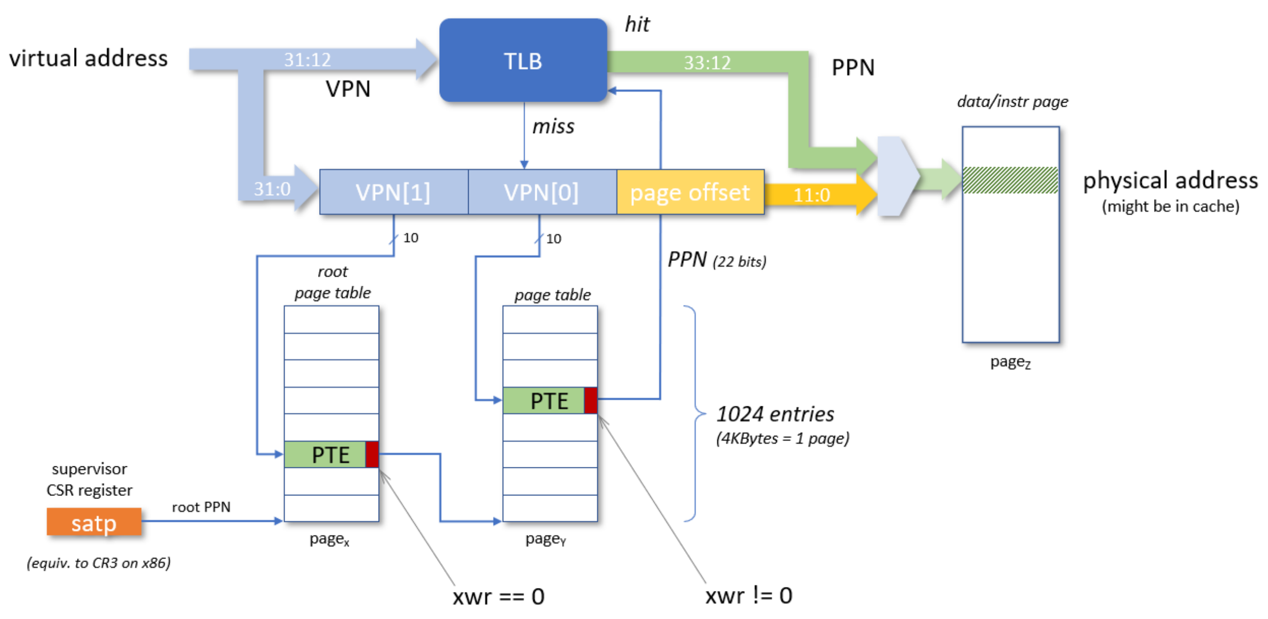
From a program's viewpoint, a cache is 'invisible', in that reading and writing data, whether it is in a cache or not does not make a difference in terms of the values stored and retrieved (and it shouldn't, otherwise something has gone horribly wrong). The main difference is that the access time is much faster when the memory being accessed is in the cache (see my article on caches for more information on how they work). Therefore, the first level that one might model a cache is to adjust wait states that are generated by an access. If accurate timing is not important, then one can skip modelling the cache and, perhaps, use an estimated average for the timing (based on measured data) which incorporates probabilities of when data access in main memory, versus when it is in the cache.

Even with just modelling for more accurate timing, the cache operation must be modelled to know when cache lines are loaded and flushed, and when accesses hit or miss, and then the appropriate timings reported. The LatticeMico32 model has a model of a cache for just this purpose (`lm32_cache.cpp`).

Caches also have different behaviours, depending on configuration, with a 'write-thru' or 'write-back' policy. The former means that all writes to the cache also has that cache-line written to main memory (the benefit of caching only seen on reads), whilst the latter means that the cache-line is only written to memory when the cache-line is evicted to re-use the entry. It might be important to model this if concerned with cache-coherency issues being detected with this model. The cache model then will need to hold the cache-line in internal memory, and only write as per the configured write policy. It may also have to provide an API to allow 'snooping' for cache coherent accesses, but this is all getting beyond the scope of this introduction summary document. The point I want to make is that the level of detail is a function of the expectations from the model, and the effort increases as more detailed modelling is required.

MMU

If the model needs to host an operating system which uses virtual memory (e.g. Linux) then a memory management unit (MMU) will be present and may need modelling. I discuss how VM and MMUs work in my article on the subject, and use RISC-V as an example. The functionality is not so complex (but it's all relative) as the complexity mainly resides in the software. There is no real way around this if the model is to boot a VM OS. The diagram below, from my article, summarises the basic components of a virtual to physical page number lookup for a RISC-V SV32 VM spec., that would need to be modelled for a RISC-V RV32 based system, with the TLB (translation lookaside buffer) being a cache of page translations.



The LatticeMico32 model does have some basic MMU modelling, based on that added to the LatticeMico32 logic for the MilkyMist project by M-LABs. The LatticeMico32 model does, in fact, boot a version of Linux though, ironically, it is an 'MMU-less' version of Linux call µClinux, and doesn't use the MMU features. See the ISS documentation for details.

So we're at the point where we have processing elements (from the first part of the document) with connection to the rest of the model, address decoding and bus/interconnect infrastructure, and a memory sub-system with optional caching and virtual memory. So we can run programs, but it's not really an SoC unless we have some peripherals and connections to the outside world.

Interfaces

Modelling the Software View

As we saw in part 1 of the document, a couple of examples of ARM based FPGA based SoC devices were shown, and a significant number of the bundled peripherals were interfaces to the outside world, with a mix of low bandwidth serial interfaces (e.g., UART, SPI, CAN) and higher bandwidth interfaces (e.g., USB, GbE). What they all have in common is that they communicate data between the SoC device and an external component. As we've seen, the software view of these interface peripherals is usually a set of memory mapped registers, and these can be modelled sufficiently to allow the code to interact with a software model of the interface, initiating control and reading status without necessarily modelling all the internal functionality. The interfaces usually come in one of two flavours—that of addressed accesses (memory mapped) and streaming accesses. For the former, an example might be an I^{2}C interface, which sends read or write commands with an address (though not an address in the memory map as seen from the processor). For the latter case, a UART sends and receives bytes on a point to point connection without an address. Even ethernet, though having network addressing in the packet protocol, can be thought of as a streaming device at the interface.

The functionality of the model for the interfaces, once a register view is established, need not model the protocol used by the interface. There is also the question of what will be connected to the other end of the interface and need modelling. An example of a register accurate, but non-protocol accurate model is that of the LatticeMico32 ISS UART (`lnxuart.cpp`). This is modelled on the Lattice 16550 UART IP core, and models registers that match this specification. The internal behaviour, however, is to emulate a terminal connected to the UART, printing characters to where the program was run from, and receiving keyboard inputs to return from the UART, using the C library functions available to our model code. The UART model has a `lm32_uart_write`, `lm32_uart_read` and `lm32_uart_tick` function as way of an API to the rest of the model, called when the peripheral is addresses. The tick function returns the interrupt status of the interface, allowing connection to the interrupt structure (more later).

As another example, the 8051 ISS model exports its GPIO pins (amongst other things) by allowing the registering of an external callback function that's called on every special function register (SFR) access, which includes the port registers mapped to the GPIO pins. The external functional can decode the SFR access and respond to port 0 to 3 accesses but return a 'not handled' status for any SFRs that are not modelled, and the ISS will handle these internally. If no callback is registered the GPIO port registers are handled by the ISS with registers that can be read or written but have no external connection.

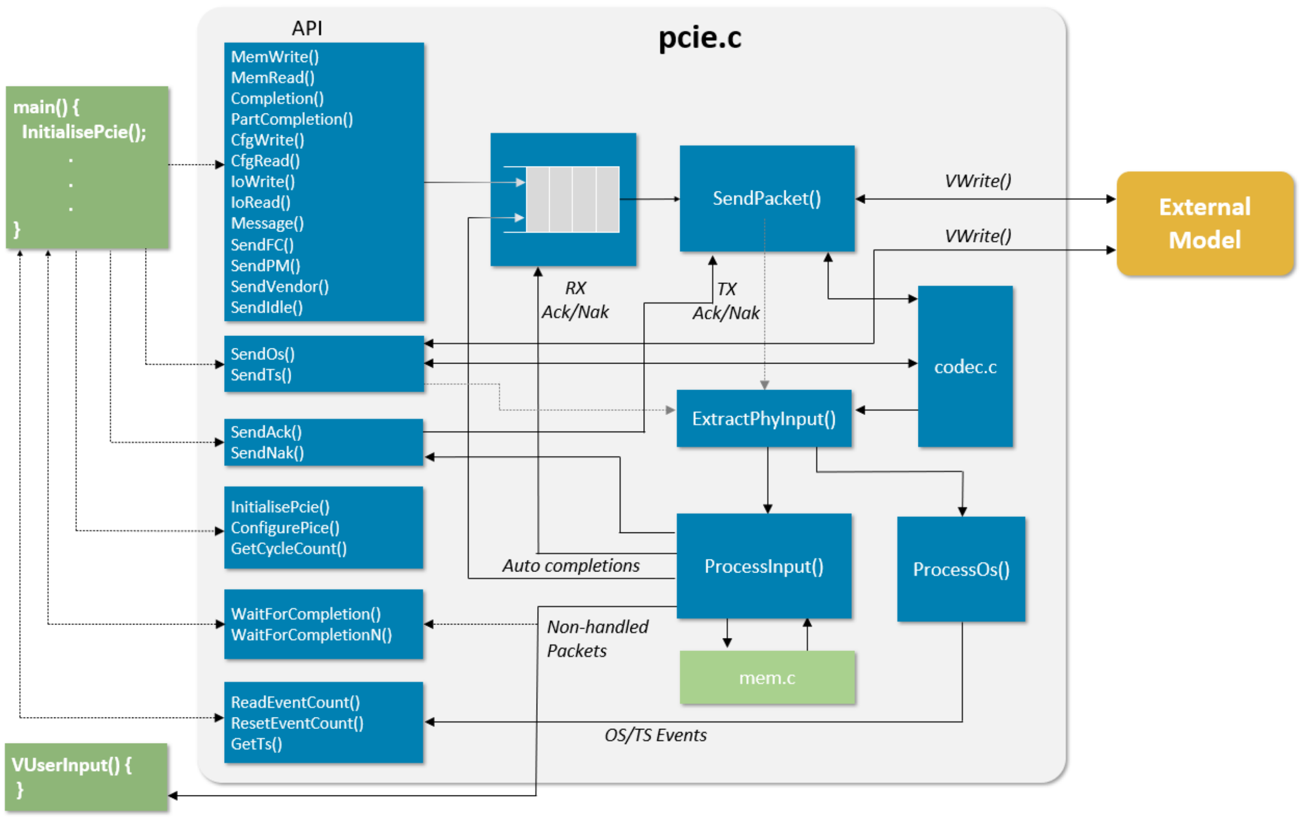
This same basic structure can be used for any of the interfaces mentioned (and others, I'm sure). The actual modelling is restricted to emulating what would actually be connected to the interface. With ethernet, though, the libraries available can allow a TCP/IP client or server (say) to be set up in software that can actually be connected from an external program, and even be a remote device. In the OSVVM co-simulation environment an example of a server TCP/IP socket class (`OsvvmCosimSkt.cpp`) is provided that allows the sending of read and write commands to the co-simulation software from an external connected device. In a demonstration test case, the external client is a python program (both with a GUI and for running as a batch program), but the OSVVM software is agnostic to who is sending the commands, and so the model can be connected to any external client. The co-simulation documentation has more details. Now this example is for co-simulation with a logic simulator, but the connection over the socket is to software, which then does the actual communication with the simulator, so method this works for purely software modelling as well. So, we have a means of actually connecting one of the interfaces to the outside world. In projects I have worked on, a TCP/IP socket connection between the embedded system and an external control program running on a remote host machine is the main data link with the SoC. If our *model* can communicate with the actual control software in this same manner, we have increased the potential software test coverage to include this application code.

Modelling Protocols

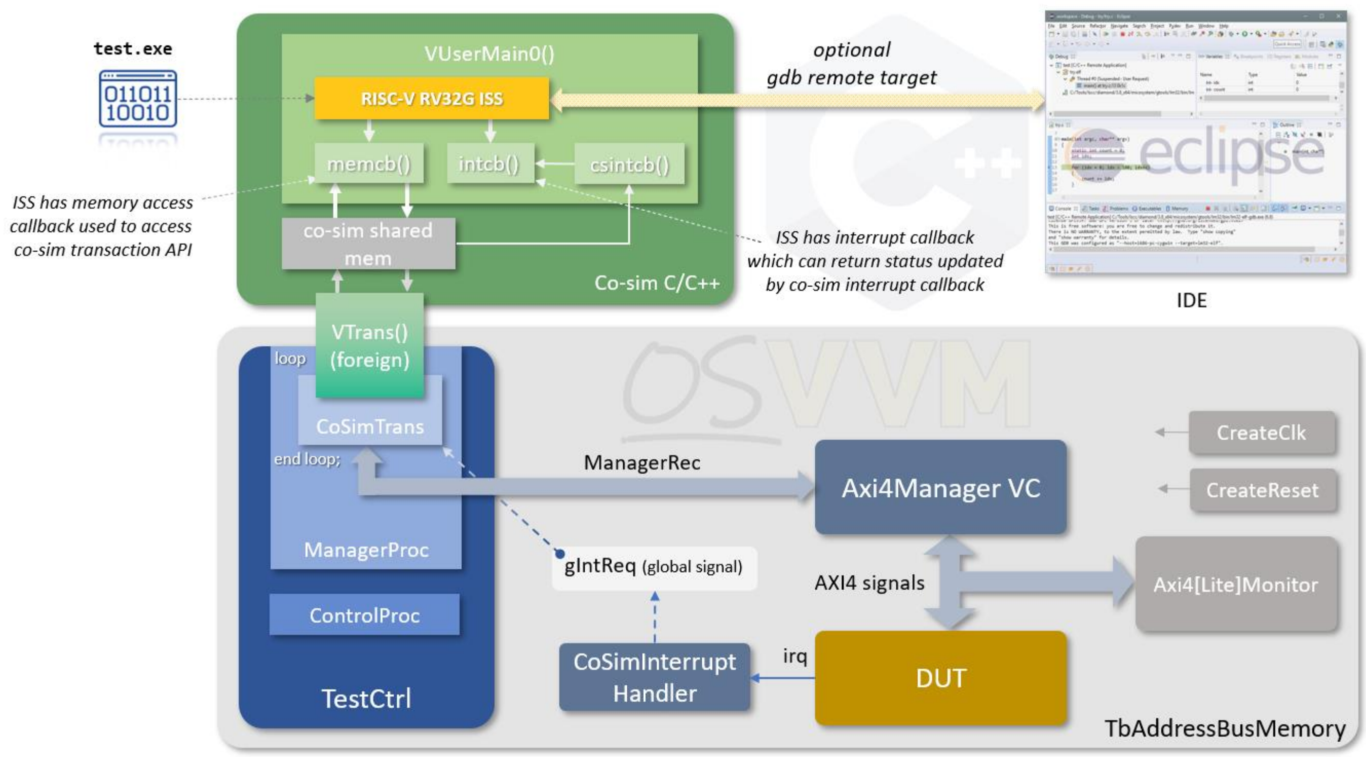
Sometime, having a behavioural model of a peripheral isn't quite enough and a more detailed model is required. We've seen before that this might be because accurate timing estimates are desirable, but it might be that we need a valid protocol to interface with a model that is expecting it. My own experience of this has always centred around co-simulation and driving logic IP from a software model, though this isn't restricted to doing so. However, the advantage here is that, having created a model and used it to develop the initial embedded software, once the logic that connects to the interface becomes available, with a software model of the protocol the logic IP can be connected into the model over a co-simulation link to a simulator and the model can drive the real IP, replacing the emulation code that was originally there. This gives us an upgrade path for the model and a basis of a test platform for logic *and* the driving software.

I've written about the logic simulators and their programming interfaces for co-simulation in more detail before in a set of articles (see parts 1, 2 and 3 for more details), but it's worth looking at an example in summary of what is required. As well as a software modelled TCP/IP traffic generator, I have also created a PCIe model, both of which have been used to drive logic IP for test purposes. A slightly modified diagram from the PCIe model's documentation is shown further below.

The model provides an API (left of the diagram) to generate PCIe gen 1 or gen 2 data for all three layers of the protocol, the ultimate output and input being 8b10b encoded data (on the right of the diagram). The model is aimed at co-simulation, where the "External Model" block is an interface to my Virtual Processor co-simulation element, but this has a simple interface of `VWrite`, and `VRead` functions which mostly index the lanes to update or read the current 8b10b code, so this can easily be used with other software models. All the functionality for the PCIe protocol is done in the software model and only (optional) serialisation was ever done in the logic HDL domain.



If co-simulating, there are other ways of approaching this. In OSVVM co-simulation, for example, the software side uses a generic interface to do reads, writes, bursts and streams etc., without regard to the underlying protocol. In the logic simulation, OSVVM provides 'Verification Components' to translate from the generic transactions to protocol specific signalling, such as for ethernet. If a VC isn't available for a given protocol, then there are instructions on how to construct this for oneself. Which approach is best, a pure software model or an HDL translation from a generic transaction, will depend very much on circumstances and the requirements, and I think both approaches have their advantages. The diagram below shows an OSVVM co-simulation setup running the rv32 ISS model.



Timers

Timers are essential in SoC systems that run a multi-tasking operating system or have real-time capabilities (or both). The simplest way to model this is to use the clock count from the processing element, such as discussed in the part 1 of the document, if such a timing model is implemented. In many SoC systems the clock is running continuously and at a constant frequency, and so can be relied upon to indicate real-time as clock-count divided by clock-frequency. However, many SoCs have variable clock rates, or clock gating, for power saving measures, and thus no reliable clock count that matches real-time. Often these systems have either internal, always powered, logic that is clocked from a separate real-time clock (often 32.768KHz) or relies on an external device. Either way, we need strategies to model these situations.

For the case of a reliable system clock rate, a model of the timer block just needs to know the time. When we discussed the memory callback of the processor models, I indicated that the processor's view of the time is passed into the callback. This might be passed down to the timer using the memory access hierarchy, but that would mean it would only get an update if a register was accessed in the timer block. The callback function could save the time off to some accessible means, which is better, but the granularity would only be at the load/store instruction execution rate—which might be acceptable. If a clock tick granularity is needed, then the timer will need to be called every clock cycle to inspect the current time so that it can generate any interrupts that may have been set up. This requires that the processor models' step function has clock based stepping as well as instruction based stepping, and the execution loop can then step the processors and call a timer function at a clock granularity. My experience is that this level of granularity isn't really necessary, down to the low nanosecond precision, as the interfaces to the timer are the memory mapped registers, which require a load/store instruction to access (and thus can update the available time) or the interrupts generated by the timer block with, often, a variable latency before the processor responds, making clock cycle precision just part of the 'noise'.

So, these approaches work for running a simulation, where executing the model behaves correctly for its own sense of time but is not synchronised with real time and runs at whatever rate as dictated by the host performance capabilities that the model is running on. Its advantage is repeatability, since its timing is not reliant on outside events. What about running in real-time?

Real-time Clock

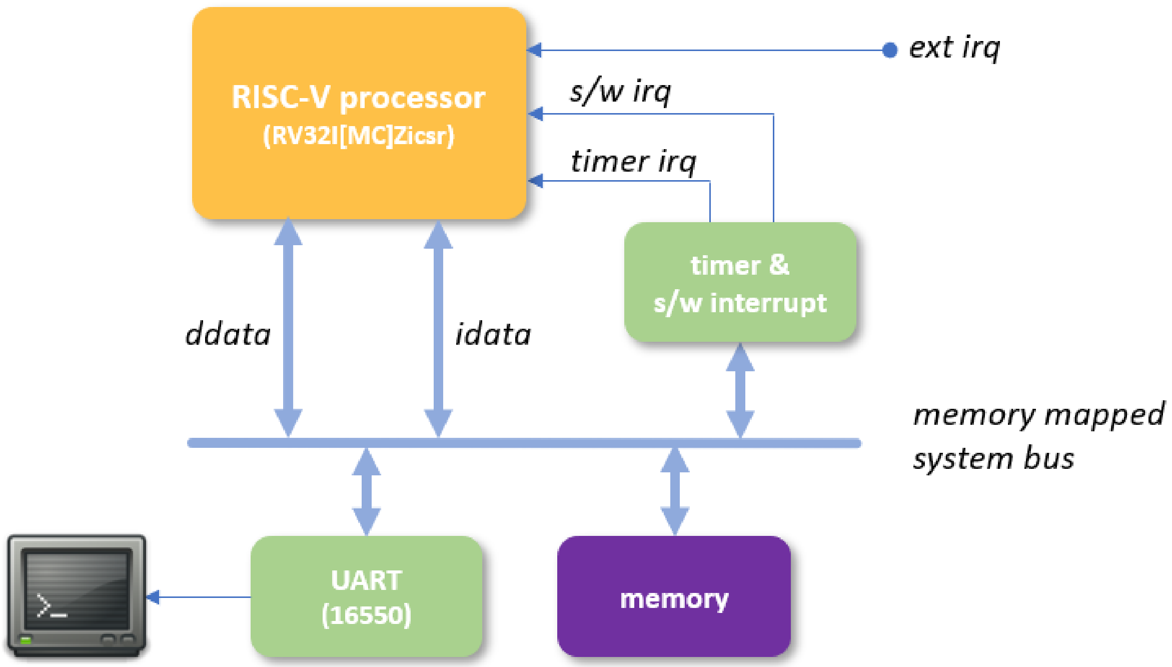
As we shall see later, basic SoC system models can run in the mega-instructions per second range. This lends itself to running in real time. It may be that the real SoC will run at a faster rate, but the model is sufficiently fast still to run at a human level interaction, such as keyboard input, or even a blinking LED at a once-per-second blink rate, and this can be very useful.

For this, then, we need a real-time clock model for our timer block. The host machine will have a real-time clock which we can access through library functions. The RISC-V rv32 ISS is configurable to use its internal cycle count state or a real time clock as the clock value for its `mtime` Zicsr register. A method is defined as show below:

```
#include <chrono>

uint64_t real_time_us() {
     if (use_cycles_for_mtime)
     {
         return cycle_count;
     }
     else
     {
         using namespace std::chrono;
         return time_point_cast<microseconds>(system_clock::now()).time_since_epoch().count();
     }
 }
```

Here the chrono library is used, and the system clock accessed and cast to microseconds since the 'epoch' and returned by the method. This is used by the RISC-V model in a small SoC model which can run the FreeRTOS multi-tasking operating system, scheduling task with time-slicing, and a demonstration is provided. I write about this in my series of articles on real-time operating systems (see parts 1, 2, 3 and 4). In part 4 I discuss porting this to the RISC-V rv32 based SoC model. The SoC system modelled to run FreeRTOS is shown in the diagram below, taken from the 4th article, which uses the UART and memory model we've already discussed, and the real-time library functionality for the timer, as described above. With this it's possible to set up concurrent tasks that execute at real-time delays of, say, 1 second and have available all the other features of the real-time OS.



This is a simple SoC model, but already it is sufficiently able to run an OS supported by a large commercial company (Amazon Web Services) with real-time capabilities and uses many of the techniques discussed already. With components already available—the memory taken from the co-simulation memory model, the UART adapted from the mico32 model, and the rv32 ISS was already implemented—only the timer functionality needed to be implemented (as described above) and the components put together. Thus, a working system is constructed that can run embedded software in a short space of time that is then the starting point for adding more functionality to build a more complex model.

Speed of SoC Models for Real-time

I mentioned in the last sub-section that real-time operation is possible because models can run the mega-instruction per second range. But what kind of range? I will briefly describe what I measure with both the mico32 model running µClinux and the rv32 SoC model running the FreeRTOS demo.

The LatticeMico32 model is much older than the RISC-V ISS, and I went through an exercise, in the past, of putting in a configurability when allowed all 'nonessential' operations to be compiled out. That is various checks and optional functionality were no longer included in the executable. This is the system that the µClinux demonstration is run on, with compilation optimisations set to 'fast'. As I write, my host PC is an Intel i5-8400 CPU running at 2.81 GHz, with 32GBytes of memory. On a virtual Linux machine on this host, running CentOS 7 I have measured between 99 and 101Mips.

The RISC-V system, on this same machine runs at between 10 and 12Mips. Why the disparity? Well, firstly I have not, as yet, gone through the same exercise as I have for the LatticeMico32 model to strip down to barebones code. I've run a profiler when executing the code and, as one would expect, the functions in the execution loop take up most of the time, with the decode method and the processing of interrupts method taking up most of this time. Compared to the LatticeMico32, these two functions are much more complicated. The mico32 decoding is only one table deep with a single 6-bit opcode for indexing, whilst the RISC-V model has depth up to four tables. As for the interrupts, the RISC-V architecture has many more sources of exceptions and more complex enabling and status constructions (with the Zicsr extension). Thus, the model runs more slowly, though still at a fast enough pace to allow real-time operation.

None-the-less, more complex models with, say, multiple processors and more sophisticated peripheral models will run slower still, and coding techniques matching in complexity may be needed to maintain a real-time operation, if this is really required. This might be in the form of utilizing multiple cores in the host's processor, running model components, such as the processor models, in different threads, or using the virtual processing techniques discussed in part 1, which will run much faster than an ISS. When writing models professionally, speed of execution is always a constraint, and coding is always done with this in mind.

Algorithmic Modelling

So far, we have talked about interface modelling and the processors. Some peripherals are not interfaces, however, but process data in an algorithmic way instead. Such things might include ECC, data compression or encryption. How much of the algorithms is modelled depends on what the requirements are.

When I was at Hewlett Packard, I was putting together an FPGA based emulator for a tape storage drive data channel and mechanism. It had to look, to the main controller logic and embedded software, like the tape mechanism and be able to read and write a section of 'tape'. In order to keep things simple, the hardware and FPGA logic just implemented a large buffer to store a partial tape image, with some logic to produce emulated signalling, such as the head spinning signal, and to 'move' up and down the tape. *All* of the tape image was going to be generated in software, and one of the first reasons I went down that path is that I knew some of the software I needed already existed. The engineer developing an ECC solution had already constructed a C model of the algorithm to test and then match against the logic, when it had been developed. I was developing a data compression solution and had done a similar exercise for this. There were some missing bits in the DSP part of the channel, though, and no available model (the design spec. was not even fully complete). Part of this did have to be on the hardware and so a block was constructed that did basically nothing (an effective delta-function for the channel) but looked like the component to the rest of the system. This example demonstrates the kind of modelling decisions that need to be made, making use of existing models, or simplifying the modelling to implement enough for a valid algorithm case, even if this is 'don't alter the data'. I don't have access to these models, but I did re-implement a simplified data compression model, based on the LZW algorithm used at HP (DCLZ), but stripped of unnecessary fluff, and this is an example of an algorithmic model.

Sources of Interrupts

In part 1 of the document, I discussed modelling interrupts for a processing element, with techniques for a virtual processor without resorting to complex coding, and ISS models that use an interrupt callback to check the status of interrupt requests. For both these cases it is likely that an interrupt controller block is modelled. In the SoC model running FreeRTOS mentioned above, for example, the interrupt functionality models the RISC-V CLIC specification. This interrupt controller model, from the processor side, can either call a method to update interrupt state with the virtual processor, or update system state that the ISS interrupt callback can inspect when called.

The sources of interrupts can come from various places, and, in an SoC, most peripherals will likely be a source of interrupts, including things closer to the processor such as MMUs. The interrupt controller block just needs to provide methods to allow each peripheral to update its interrupt request state, both set and clear. If edge triggered interrupts need to be modelled (perhaps configurably), then the interrupt controller model will need to hold request state after the input is returns to inactive, until cleared explicitly by a write to the controller, as dictated by the controller functionality being modelled, such as the ARM NVIC, for example.

Conclusions

In this, the final part of the document on modelling SoC systems in C++, we've concentrated on the other devices other than the processors. This, and part 1, has only been an introduction to this subject, but there have been plenty of references to example code and documentation to allow further investigation. There has not been room to mention all the features a model might have. For example, the mico32 ISS models hardware debug registers, which can be utilized by the `gdb` debugger, and also has save and restore features.

I have covered only in summary the subject of connection to external programs and co-simulation with a logic simulator, but I detail this in a previous set of articles on PLI and on the co-simulation features of OSVVM, including articles on interrupt modelling, and more on modelling event driven, multi-threaded programs on a virtual processor as part of a document on OSVVM co-simulation nodes, both of which are relevant to this discussion.

I hope I've given some insight into modelling SoC systems in C++ but had to trim this to a sensible length. If people would like more detailed discussions on a particular topic, or clarification on the points I've raised, let me know by dropping me a message. Perhaps smaller, targeted, and more detailed documents can be done in the future, if there is interest.

I have rarely had a job where software modelling wasn't used in some capacity, even if I wasn't directly involved (which usually I was). It bridges the gap between embedded software and the logic and hardware at the earliest possible stage, from architecture exploration, to allowing early integration, and then feeds forward as a platform for testing IP, either as means to co-simulate logic with the model, or as a model for comparative verification with the logic with a simulation test bench. Either way, it is useful for embedded software engineers, logic design engineers and design verification engineers to understand what is possible with such models, and for early cooperation and early engagement between these disciplines.