A Cycle-accurate Template Microprocessor Model of a Von Neumann Architecture Based on SystemC

Lubomir Bogdanov, Ratcho Ivanov
Department of Electronics, Faculty of Electronic Engineering and Technologies
Technical University of Sofia
8 Kliment Ohridski blvd., 1000 Sofia, Bulgaria
{lbogdanov, r.ivanov}@tu-sofia.bg

Abstract – **The paper presents different aspects of the creation of a cycle-accurate, pipeline-accurate, single-issue, von Neumann microprocessor model that can be used as a template to model any existing microarchitecture. The goal of this development is simplicity and modularity so that any embedded developer could create a model out of the instruction timings given in the datasheet. The model is created as a separate entity to allow for code reuse and support for many families and sub-families of microcontrollers.**

Keywords – **cycle-accurate simulation; deeply-embedded; instruction set simulator; microprocessor model; systemc.**

I. Introduction

Nowadays virtual prototypes are an integral part of the system development and simulation on each level of abstraction is a common procedure. Many industry-leading companies invest in sophisticated software as the hardware complexity continues to grow. Register transfer level (RTL) and electronic system-level (ESL) design use simulation to verify the operability of new hardware configurations. Different teams develop libraries, hardware and software, at different abstractions to help the final integration of their components into a system by the main architect of the project.

A major effort in the creation of RTL and ESL models is done by the Accellera Systems Initiative that groups together leading companies like ARM, NXP, STMicroelectronics, Intel and others, to create C++ libraries for design and verification of electronic products under the name SystemC. The positive aspect of such an approach is that the libraries are based on standard C++ and can be ported to any operating system. Therefore a simulation environment and models created in SystemC are OS-independent and compiler-independent.

Though many such tools exist already in the industry, most of them are closed-source, or partly-open source. The final goal of this research is to create:

- an entirely open source simulation environment;
- an entirely open source simulation models.

Using SystemC for such a task has one more advantage – recent releases contain the SystemC AMS libraries that can be used for analog and RF functionality which would extend the simulation not only for digital circuits but also for analog devices.

The structure of the future simulation environment is shown in Fig. 1. The aim is to create complete SystemC models of deeply-embedded microcontrollers such as the families of MSP430, ARM Cortex-M, PIC16/18, Xtensa,

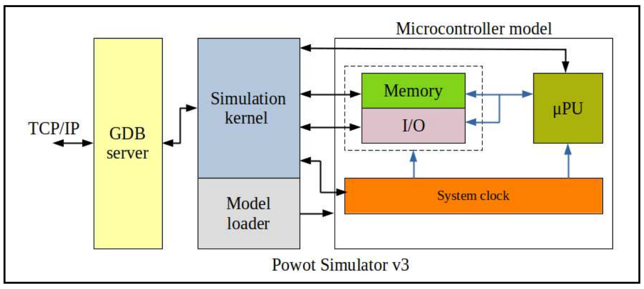

Fig. 1. Internal structure of the proposed cycle-accurate simulation environment (successor of the Powot Simulator).

etc. The tool is a successor to the instruction set simulator (ISS) Powot Simulator version 2 and adds instruction timings and a pipeline of the microprocessor. Energy costs of instructions will be supported. The current paper discusses only the microprocessor from the block diagram shown in Fig. 1.

The level of abstraction of the proposed template model is RTL and logic gates are not simulated. This would increase the simulation speed, as well as provide ease of model creation – a developer could treat the microprocessor as a black box, using only specifications given in the datasheet of the microprocessor or the microcontroller.

II. Literature Overview

To develop a microprocessor model, an insight of the microarchitecture is needed. However modern microprocessors on the market are closed source and they should be treated as a black box. In most of the cases the datasheets contain instruction cycles and details that are enough to create a model. If no such information is given, experimental measurements must be conducted to extract those timings by using a microcontroller with an external clock generator and a pulse counter. Then, the stages of the pipeline have to be modeled using C++ and the SystemC libraries. A good starting point for learning about the instruction execution and the pipeline is the book [1] and a generalized and shortened version of this book is given in [2]. Authors start with a simple example of a single-issue, scalar processor using just a program counter (PC), memory and an arithmetic-logic unit (ALU).

The work presented in [3] gives a classification of modern processor microarchitectures that deviate from the standard von Neumann model. This should be kept in mind while developing models because the code of the model should be structured in such a way that adding new features is made easily and without rewriting the core functionality.

Some knowledge about superscalar processors is needed because such microarchitectures are starting to enter the market. One example is the ARM Cortex-M7 that is integrated in the deeply-embedded microcontroller STM32F769. A description of such an architecture is given in [4] where an example C program and its Assembler equivalent are analyzed. Data hazards involving registers are also explained.

In [5] a comparison between a RISC and a CISC architecture is given. Two specific processors are analyzed – the Alpha 21164 and the Intel Pentium Pro. Implementation details of the pipeline are shown that may help for modeling. For example, a superscalar processor may have more functional units in its core than issue queues.

The paper [6] presents details about vector processors. Vectored processing moved from supercomputers to embedded systems. An example is the RISC-V architecture and the RISC-V-based microcontroller ESP32-S2. Because RISC-V is an open source instruction set, it is very likely that in the future more of these implementations will be seen. A template model should contain such an option also.

Further architectural details are given in [7] and [8] for the R4300i and UltraSPARC-I processors. They contain cache memories (which are also common for the modern microprocessors) and on the block diagrams the connection between them and the processor could be seen. The paper [8] even shows the width of the internal buses. The paper [7] shows one fundamental detail about pipelined processors – the internal logic works on both edges of the clock signal and this should be reflected in the SystemC model. Otherwise the model wouldn't be cycle-accurate at all.

III. Basic Blocks of a Cycle-Accurate SystemC Microprocessor Model

Implementing a microprocessor model is a task that cannot be accomplished on its own – additional external (to the microprocessor) modules are required. Figure 2 shows the minimum setup that is needed to start the development.

The **microcontroller block** is a top-level entity that holds all of the modules and the buses. This block has a specific name corresponding to a particular part number. The rest of the modules can be reused but the microcontroller – cannot.

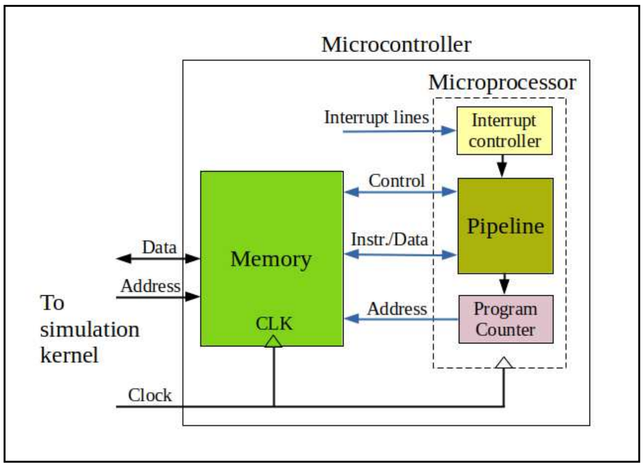

Fig. 2. Minimum modules required to start the microprocessor model development.

The **microprocessor block** contains a program counter (PC), an interrupt controller, and components of the pipeline – stages, interconnect buffers, functional units, etc.

The **pipeline block** extracts information about instruction execution timings from a library and controls the PC during program jumps and pipeline stalls. The pipeline is controlled not only by the program instructions but also by an interrupt controller.

The **program counter** is a parallel register from the register file and is connected directly to the address bus of the microprocessor. The address bus is an output-only bus and no other outputs can be connected to it, or the SystemC simulation kernel will report an error during runtime. The PC is used to stop the instruction fetching during a pipeline stall. This event can be detected and displayed to the user.

The **interrupt controller** block breaks the fetch-decode-execute cycle and inserts a vector table address in the program counter. Before this happens, a stack frame is pushed onto the stack and some cycles are stolen to simulate interrupt latency. This block is asynchronous and is not connected to the clock signal.

The **memory block** is an abstraction of the entire address map of the microprocessor. Instruction and data memory, as well as peripheral registers must be mapped into this block in the future. For now, only instruction/data memory with single cycle access times is needed, a vector table and a stack region. This block must be exposed to the simulation kernel, so that the user could write some instructions and data before performing power-on-reset. No memory controllers are modeled so far. The development effort is put entirely on the microprocessor.

A **clock signal** is shared between the microprocessor and the memory block. Simulation events take place on both edges – this will ease the writing of the C++ code and will make the model cycle-accurate. For simplicity – debug messages of instruction execution are shown on both edges. This way the core state is visible to the user.

A. The Microprocessor and the Pipeline

In Fig. 3 the internal structure of the pipeline is shown. The template model contains three stages but from a programming point of view they are identical and the user can add as many stages as needed. The fetch stage differs from the others because it contains the logic for the PC.

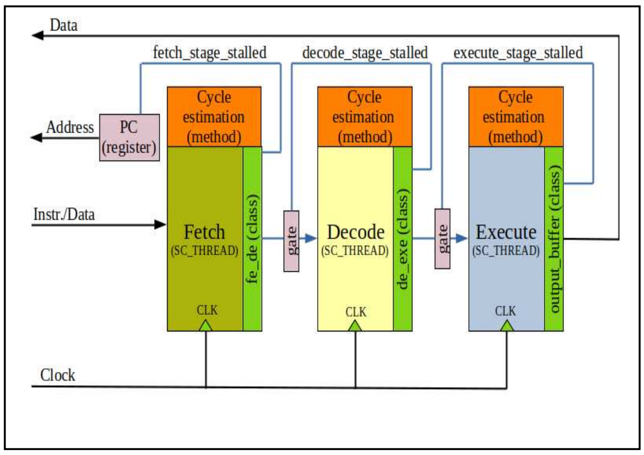

Fig. 3. Internal structure of the modeled pipeline.

Instruction fetching begins with the PC. The PC is a standard 32-bit variable of the type "uint32_t". This variable models a parallel register. Its size can be changed to 8-, 16- or 64-bit. If the modeled microprocessor contains a PC with a different resolution, the nearest bigger value should be chosen, e.g. for a 10-bit PC choose a 16-bit variable, for a 20-bit PC choose a 32-bit, and so on.

Next, the memory or peripheral device will output a value on the instruction/data bus. Because many peripherals and memories could be connected to it, this bus must be implemented as a logic vector – a special type of signal that can have logic high, logic low and high-impedance values. The bus is declared in SystemC as:

```
sc_inout<sc_lv<16>>
instr_data_bus_fetch_in
{"cpu_instr_data_bus_fetch_in"};
```

where sc_inout declares a bidirectional wire, sc_lv<16> declares a 16-bit bus width whose wires can have logic 1, 0 or high impedance states, and the string in the curly braces sets the name of the bus. The latter is an important aspect of the SystemC simulation because if an error occurs, the SystemC kernel outputs a message related to this string. If the bus is not given a name, SystemC will use an automatically generated one which would make the debugging obscure. A more sophisticated bus structures exist, such as shown in [9], and is up to the user whether they will be implemented or not.

Once the instruction/data bus holds a valid value, it will be directed to the fetch module. The module is modeled as a method of a special type – a synchronous thread in SystemC. This is needed because the SystemC simulation kernel can treat it as a digital hardware and can route signals concurrently. If it is implemented as a normal C++ method, each hardware signal will enter each module sequentially, which will lead to a non-functioning device. For example, if an Enable signal is connected to three modules, this signal must reach simultaneously each module in the same clock cycle, if asserted. But a C++ program is a sequential program, so a problem arises. To deal with this problem, SystemC contains a special macro that declares the method as a thread. In the current template this is done the following way:

```
SC_MODULE(cpu){
  sc_in_clk clock {"cpu_clock"};
...
  void fetch();
  void decode();
  void execute();
...
  SC_CTOR(cpu){
  ...
       SC_CTHREAD(fetch, clock.pos());
       SC_CTHREAD(decode, clock.pos());
       SC_CTHREAD(execute, clock.pos());
  ...
  }
};
```

where the macro SC_CTHREAD( ) must be invoked in the constructor of the module, SC_CTOR( ), and the clock edge for the activation of the method is declared as rising. It is worth noting that this template can be used for microprocessors with synchronous gates only. Some designs try to lower the consumed power by using asynchronous gates, like in [10] and [11]. Asynchronous modeling would require a more detailed knowledge of the hardware and the models would be bigger in code size. For simplicity, authors consider only synchronous microarchitectures in this paper.

The first value to be fetched (after power on) on the instruction/data bus must always be an instruction. When the instruction enters the fetch stage, it is passed to a method that must evaluate its opcode and extract cycle penalty for the current stage. This method should be implemented by the user. Currently, the method simply returns 1 cycle per phase, to keep the pipeline full. Here a problem arises – most manufacturers give instruction cycle cost as a single value. This means that the user must either try to figure out the cycle cost per each stage alone, or make some kind of cycle-accurate measurements on a real hardware. Both tasks are daunting and future research will be done to address this problem. This method can also be used for energy consumption estimation and some novel ideas how to do this is given in [12], [13] and [14].

The cycle estimation method holds its respective stage stalled and counts the number of cycles that have passed since the entry of the instruction. When the cycles have been wasted, the instruction is ready to advance into the next stage. But before that, the next stage must have completed the processing of the previous instruction. If there is an instruction in the next stage, the previous stage must stall. If there is no instruction in the next stage, the previous stage can transfer its instruction and load a new one. This behavior can be done easily with an inter-stage buffer. The buffer must implement a blocking read/write FIFO scheme, i.e. if the buffer is full, no new data can be pushed, and if the buffer is empty, the stage must stall on reading it. On Fig. 3 six variables are used for this purpose:

```
bool fetch_stage_stalled;
bool decode_stage_stalled;
bool execute_stage_stalled;

fifo_buffer *fe_de;
fifo_buffer *de_exe;
fifo_buffer *output_buffer;
```

the first three are simple boolean flags, the second three are classes of type "fifo_buffer". The FIFOs are complex variables that must check for data availability on each access, that is why they are implemented as classes. The size of the FIFO is configurable, but in the current model it is set to 1. The methods that the user can use are:

```
  bool push(sc_lv<16> new_element);
  bool pop(sc_lv<16> &next_element);
  bool peek(sc_lv<16> &next_element);
  bool is_empty(void);
```

where each value is declared as a logic vector of 16-bit size (sc_lv<16>) because the instruction/data bus uses the same type. Each method returns 0 on success and 1, if the FIFO is full/empty.

If all of the stages are stalled, the pipeline stalls and no further instructions should be fetched. That is why the flag of the fetch stage blocks the PC from incrementing. This event can be tracked by the user, as it is one of the most important contributors for reducing the instructions per cycle (IPC) parameter of a microprocessor.

B. The Microcontroller

All of the modules described so far are part of a bigger entity, the model of the microcontroller. Such a hierarchy allows that submodules can be reused just like in real life hardware – Texas Instruments' Timer_A and Timer_B modules are one and the same throughout all of the devices, ST Microelectronics' LPTIM and TIM2 also, etc.

That is why the microcontroller instantiates all of the needed submodules (implemented as SystemC classes) and connects them with buses. Here is how this can be done:

```
SC_MODULE(mcu){
  sc_in_clk clock {"..."};
  sc_signal<bool> read_write {"..."};
  sc_signal<sc_lv<16>,  SC_MANY_WRITERS>
instr_data_bus {"..."};
  sc_signal<sc_uint<20>>      address_bus
{"..."};
  sc_signal<sc_uint<16>>      control_bus
{"..."};

  cpu *cpu_;
  flash_memory *flash;

  mcu(::sc_core::sc_module_name);
  ~mcu();
};
```

and then the submodules are connected together in the constructor of the microcontroller:

```
mcu::mcu(::sc_core::sc_module_name){
cpu_ = new cpu("cpu");

cpu_->address_bus_fetch_in(address_bus);
cpu_-
>instr_data_bus_fetch_in(instr_data_bus)
;
cpu_->clock(clock);
cpu_->read_write(read_write);

flash = new flash_memory("flash");

flash->address_bus(address_bus);
flash->data_bus(instr_data_bus);
flash->clock(clock);
flash->read_write(read_write);
}
```

Once each submodule is tested and verified, it can be used in many different microcontrollers throughout different families and subfamilies.

C. The Simulation Environment

Once the microcontroller has been set up with a minimum of a CPU and a memory block, it must be attached to a program that drives the clock line. This has to be done in the simulating program that will invoke the models. Currently this is a very simple program that initializes the microcontroller's memory with some instructions and a single loop that drives the design with some clock cycles:

```
int sc_main(int argc, char* argv[]){
  sc_signal<bool,         SC_MANY_WRITERS>
clock;
  int i;
  mcu mcu_("mcu");

  mcu_.clock(clock);
  mcu_.flash->program(0x4400, 0x1111);
  mcu_.flash->program(0x4402, 0x2222);
  mcu_.flash->program(0x4404, 0x3333);
  mcu_.flash->program(0x4406, 0x4444);
  mcu_.flash->program(0x4408, 0x5555);
  mcu_.flash->program(0x440a, 0xaaaa);

  for(i = 0; i < 20; i++){
       clock = 0;
       sc_start(1, SC_MS);
       clock = 1;
       sc_start(1, SC_MS);
  }
  return 0;
}
```

The simulation program has to be developed further to support device selection, time quants selection, binary loading, GDB debug support, etc.

IV. Simulation Results

In order to test the microprocessor pipeline, two events have to be induced: a pipeline stall and a pipeline bubble.

The first experiment runs a simulation where each instruction spends a single cycle in each stage. Here are the results:

```
execute 0
decode 0
fetch 1111
============== [3] 0 ==============
============== [3] 1 ==============
execute 0
decode 1111
fetch 2222
============== [4] 0 ==============
============== [4] 1 ==============
execute 1111
decode 2222
fetch 3333
============== [5] 0 ==============
============== [5] 1 ==============
execute 2222
decode 3333
fetch 4444
```

To induce a pipeline bubble, instruction 0x3333 must spend 2 cycles in the fetch stage. Here are the results:

```
execute 0
decode 0
fetch 1111
============== [3] 0 ==============
============== [3] 1 ==============
execute 0
decode 1111
fetch 2222
============== [4] 0 ==============
============== [4] 1 ==============
execute 1111
decode 2222
fetch 3333 stall[1]
============== [5] 0 ==============
============== [5] 1 ==============
execute 2222
decode 0        //BUBBLE!
fetch 3333
============== [6] 0 ==============
============== [6] 1 ==============
execute 0       //BUBBLE!
decode 3333
fetch 4444
============== [7] 0 ==============
============== [7] 1 ==============
execute 3333
decode 4444
fetch 5555
```

In clock cycle #4 the fetching stage stalls for one additional cycle which produces a bubble in the decode stage. In iteration #7 the bubble is cleared and execution continues at full rate.

To induce a pipeline stall one of the instructions must block for a time long enough so that the blockage is propagated to the fetch stage (and respectively to the PC). Instruction 0x1111 could stall in the execute stage for 3 cycles, then here are the results:

```
execute 0
decode 0
fetch 1111
============== [3] 0 ==============
============== [3] 1 ==============
execute 0
decode 1111
fetch 2222
============== [4] 0 ==============
============== [4] 1 ==============
execute 1111 stall[2]
decode 2222
fetch 3333
============== [5] 0 ==============
============== [5] 1 ==============
execute 1111 stall[1]
decode 3333 stall[1]
fetch 4444
============== [6] 0 ==============
============== [6] 1 ==============
execute 1111
decode 3333 stall[0]
fetch 5555 stall[1]
============== [7] 0 ==============
============== [7] 1 ==============
execute 2222
decode 3333
fetch 5555 stall[0]
============== [8] 0 ==============
============== [8] 1 ==============
execute 3333
decode 4444
fetch 5555
============== [9] 0 ==============
============== [9] 1 ==============
execute 4444
decode 5555
fetch aaaa
```

Notice how instruction 0x2222 disappears in clock cycle #5. This is because its decoding phase has been completed and the instruction is pushed into the inter-stage buffer. The same event happens with 0x4444 in cycle #6. The stalling of the pipeline happens in cycle #6 where instruction 0x5555 is forced to stall in the fetch stage up to cycle #8 (inclusive). So a 3-cycle stall in the execute stage forced a 3-cycle stall in the fetch stage, which is the expected behavior.

V. Conclusion

A simple template SystemC model of a microprocessor is proposed in this paper. The model will be open-sourced and could be used by research groups to model different microarchitectures as a black box, without detailed knowledge of the underlying hardware. By simply using instruction cycle numbers from a datasheet, a fast model development will be possible. The model is independent of memory and peripherals, and can be used in many microcontrollers, just like HDL code is reused to create different families and sub-families of microcontrollers. A development effort is needed at the beginning to create the modules and later on a system-level approach can be used to simply drag-and-drop different components to form a bigger model.

The simulation environment is yet rudimental, but will be enhanced in the future to allow users to perform more complex simulations. The most important part is a GDB connection that will enable the simulator to be connected to a graphical development environment such as Eclipse.

Future improvements must include the development of a new template for asynchronous microprocessors, as they are starting to enter the market for low-power applications. Energy estimation must be included also, along with the cycle-accurate timings.

Acknowledgment

The authors would like to thank the Research and Development Sector at the Technical University of Sofia for the financial support.

REFERENCES

[1] D. Patterson, J. Hennessy, "Computer Organization and Design: The hardware/software interface", 5th edition, ISBN: 978-0-12-407726-3, Elsevier, 2014.

[2] P. Machanick, "Computer Architecture: a qualitative overview of Hennessy and Patterson", unpublished, online, 2001.

[3] J. Silc, T. Ungerer, B. Robic, "A survey of new research directions in microprocessors", Microprocessors and Microsystems 24, pp.175-190, 1999.

[4] J. Smith, G. Sohi, "The Microarchitecture of Superscalar Processors", Proceedings of the IEEE, Vol. 83, Issue 12, DOI: 10.1109/5.476078, 1995.

[5] D. Bhandarkar, "RISC versus CISC: A Tale of Two Chips", ACM SIGARCH Computer Architecture News, Vol. 25, Issue 1, pp.1-12, 1997.

[6] R. Espasa, M. Valero, J. Smith, "Vector Architectures: Past, Present and Future", ICS '98: Proceedings of the 12th international conference on Supercomputing, pp.425-432, 1998.

[7] "R4300i Microprocessor", Open RISC Technology, datasheet, rev. 0.3, 1997.

[8] M. Tremblay, D. Greenlay, K. Normoyle, "The Design of the Microarchitecture of UltraSPARC-I", Proceedings of the IEEE, Vol. 83, Issue 12, DOI: 10.1109/5.476081, 1995.

[9] J. Parcerisa, J. Sahuquillo, A. Gonzalez, J. Duato, "On-Chip Interconnects and Instruction Steering Schemes for Clustered Microarchitectures", IEEE Transactions on Parallel and Distributed Systems, Vol. 16, Issue 2, DOI: 10.1109/TPDS.2005.23, 2005.

[10] A. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, T. Lee, "The Design of an Asynchronous MIPS R3000 Microprocessor", Proc. Seventeenth Conference on Advanced Research in VLSI, DOI: 10.1109/ARVLSI.1997.634853, 1997.

[11] A. Lines, "The Vortex: A Superscalar Asynchronous Processor", 13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'07), DOI: 10.1109/ASYNC.2007.28, 2007.

[12] V. Kulkarni, G. Udupi, "A Simplified Software Energy Consumption Estimation for Embedded System", Journal of Embedded Systems, Vol. 4, No. 1, 7-12, DOI:10.12691/jes-4-1-2, 2017.

[13] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, P. Cook, "Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors", IEEE Micro, Vol. 20, Issue 6, DOI: 10.1109/40.888701, 2000.

[14] I. Delgado-Lozano, M. Martínez-Rodríguez, A. Bakas, B. Brumley, A. Michalas, "Attestation Waves: Platform Trust via Remote Power Analysis", CANS 2021: Cryptology and Network Security, pp.460–482, 2021.