Cycle-true simulation of the ST10 microcontroller including the core and the peripherals

Lovic Gauthier, Ahmed Amine Jerraya
SLS group
TIMA Laboratory
46, avenue Flix Viallet 38031 Grenoble France
lovic.gauthier@imag.fr, Ahmed-Amine.Jerraya@imag.fr

Abstract

With the rising complexity of electronic systems, containing more and more both hardware and software parts, it becomes necessary to simulate simultaneously hardware and software parts at whatever abstraction level. These simulation techniques, called co-simulation, require fast and flexible simulators. In this paper, we introduce the elaboration of a microcontroller simulator including the core and the peripherals for an accurate hardware/software co-simulation at the clock-cycle level. It is our goal to have a simulator which is fast enough to simulate a few minutes of real time execution within a reasonable laps of time. To be more precise, we deal here with the realization of a simulator for the ST10 microcontroller and its integration into a co-simulation environment. This paper gives our global approach and explains our techniques in detail. It also presents new tracks to obtain improved performances with this kind of simulators.

1. Introduction

It is now possible to integrate programmable components and specific circuits on a single chip. This enabled the current trend of joint development of software and hardware parts. These techniques, called codesign, are introduced in [2, 3]. A major advantage of codesign is to allow system validation early in the design process. This reduces the cost of debugging. One of the major issues of these techniques is the performance of the simulation.

It is our objective to study a working method for the development of microcontroller simulators which are precise at the cycle level and faster than the usual HDL-based simulation. Especially we want to make a cycle-true simulator for the ST10 microcontroller able to simulate millions of cycles within a reasonable laps of time. This correspond to few minutes of real time application.

1.1. State of the Arts

Software simulation consists in making a model of the processor executing the software code. In this paper we will concentrate on a single kind of processor : microcontrollers. There are several classic methods to model a microcontroller [1]. We are going to describe them without going into details.

- Hardware simulations: principally they are obtained by means of emulators that reproduce at gate level the circuit to be simulated. Their advantages are a tiny simulation grain and a very high speed (often they can go as fast as the circuit to be simulated). Their main disadvantage is the very high cost of the material required for the emulation. It may also be very difficult to debug the code at the source level.

- Software simulations based on hardware description languages: here we have for instance VHDL and VERILOG. These languages allow to describe circuits from the behavioral level down to the gate level. These simulations have the advantage of being very precise and offering opportunity of visualizing the behavior details of the circuit. However these precise simulations of complex circuits such as microcontrollers are very slow (typically 5 to 10 instructions per second for a VHDL simulator [6]).

- Software simulations based on instruction-set simulators [7]: Instruction-set simulators (ISS) are programs that simulate the execution of the instructions of a processor. There are instruction-set simulators at different specification levels but most of the time they act at the instruction level. These simulators are much faster than the ones mentioned above (most of the time, they

are compiled programs and no longer interpreted descriptions). However they are less precise than the first two simulations. Moreover, by definition they do not really simulate the circuit (they only simulate the instruction set). Electronic factors are thus not taken into account: input/output gates are not simulated, neither are the microcontroller peripherals. Even some cycle-true behavior such as accesses to external buses may be difficult to handle.

As we can see none of the above mentioned methods provide a cycle true simulation which is both very fast and practical to use.

1.2. Our approach

Our approach consists in choosing a compromise solution between the instruction set simulator and the HDL model: we designed a microcontroller model, that simulates the whole circuit including the peripheral at the clock cycle level. Its characteristics are:

- It takes into account the specificity of microcontroller, namely both parts: the core and the peripherals. In fact the simulation is made of two distinct modules. The first simulates the core and the latter simulates the peripherals. This separation allows a modular design of the simulator. In our case two different approaches were used for the design of the two parts.

- The simulation grain is the I/O cycle for the simulator. The user can communicate with the simulator via variables that contain the values of the registers and the physical ports of the target microcontroller. These values are available after each simulation cycle that corresponds to a clock cycle.

- The model is pin accurate : thanks to the model of the peripherals, the model of the microcontroller may include all the I/O pins of the corresponding chip.

The advantages of this approach are:

- the separation of the core and peripherals gives more flexibility for the development of the simulation model.

- It is easy to integrate the simulator into a codesign environment taking into account the pin connections, and time constraints.

- The use of a programming language for the simulation, and not a hardware description language such as VHDL, increases the chances of getting a fast simulator.

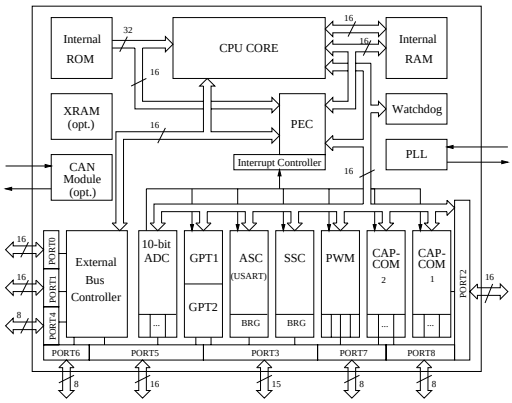

Figure 1. Architecture of the ST10 microcontroller.

The accuracy of this simulator is restricted of course to the cycle level. Certain physical aspects like fine cycle parallelism or timing constraints cannot be handled.

The main disadvantage of this approach is that the time necessary to develop the simulator is longer than the time required for the development of an instruction set simulator or even for HDL model (as far as the circuit is known already).

We are going to show how we realized the ST10 simulator starting from this approach. Section 2 presents the ST10 microcontroller. In section 3, we show how we elaborated the simulator. In section 4, we give an example of application using our simulator. Finally, in section 5, we discuss the possibilities to improve the performances of the ST10 simulator.

2. The ST10 microcontroller

The ST10 microcontroller, manufactured by STMicroelectronics, is composed of a 16-bit microprocessor, set of peripherals (ADC, serial interfaces, etc.) and 9 configurable input/output ports. Although the core has specific ways to communicate with the peripherals, the different parts of the microcontroller run in parallel and autonomously (the core and the peripherals have to run at the same clock speed, but their clocks are independent). The ST10 microcontroller is identical to the C167 of Siemens. We used [12, 11, 9, 10, 8] as sources of information in order to write our model. Figure 1 schematizes the architecture of this microcontroller.

2.1. The core of the system

The core of the system is made of a 16 bit microprocessor. The instruction set contains 236 instructions. The core of the ST10 is 4 stages pipelined and includes, one arith-

metic and logic unit of 16 bits (ALU), dedicated registers called SFRs (Special Function Registers), and a specific additional unit for multiplication and division (the execution of which takes several cycles).

The pipeline [5] allows the processor to execute most of its instructions in a single machine cycle. For the ST10 a machine cycle corresponds to two clock cycles. Actually the complete execution of an instruction is composed of the following four classical steps: Fetch (i.e. the loading of the instruction from the memory), decoding of the instruction and loading of the instruction operands, the actual execution, and the write-back. All these steps are done within one machine cycle, except for the execution step, the delay of which depends on the instruction.

The jump instructions have been optimized. To that end, the destination address of the jump has been calculated beforehand at the decoding phase. Moreover a cache is used to memorize the last jump. Eventually it takes most of the time only one clock cycle. The context switch has also been optimized in particular with the introduction of a system-dedicated stack.

The CPU also offers a large number of additional specific instructions: for the initialization of the microcontroller, for the construction of complex atomic operations, and for the control of its running mode configuration (normal, idle, power down).

2.2. The embedded peripherals

The standard embedded peripherals of the ST10 are: an analog/digital converter of 10 bits (ADC), 2 capture/comparison units (CAPCOM), a pulse width modulator (PWM), 2 general purpose timers (GPT), synchronous and asynchronous serial interfaces, a watchdog timer (WDT), and an external bus controller.

On top of these peripherals, the ST10 can integrate other complex peripherals such as a MAC (Multiplier ACcumulator) or a CAN interface (a communication protocol which is often used in the automotive area). These optional peripherals have not been taken into account in this work. However their modeling is possible using the same approach. The control of the peripherals is done through SFRs (Special Function Register). The feedback from the peripherals to the CPU is given by the same SFRs or by interrupts.

3. The ST10 simulator

The ST10 simulator is composed of 3 parts: a first part which simulates the core, another part which simulates the peripherals, finally a main loop in charge of activating the core and the peripherals. For each step, it makes the core run for a cycle, then the peripherals for a cycle and finally deals with the input/output updates.

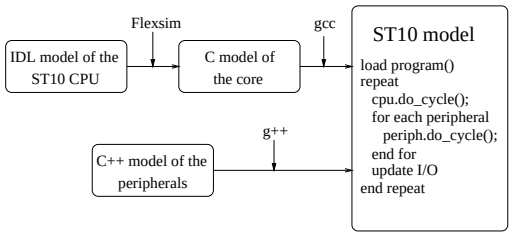

Figure 2. General running scheme of the simulator

Figure 2 shows the structure and the generation flow of the simulator. The model of the core has been generated automatically using an instruction set simulator generator called Flexsim [7] (see 3.1). And the peripherals have been described in C++. They are compiled in order to obtain the library of the peripherals.

Both libraries are linked to the main program making up the complete simulator of the microcontroller.

Now we are going to present the simulation of the core, and then the simulation of the peripherals.

3.1. The model of the core

The model of the core has been designed by means of Flexsim, a generator of an instruction set simulator developed by STMicroelectronics [7]. This program is actually a kind of compiler that takes as input a description of the processor in a language called IDL and produces as output the C code for the simulator. The IDL language is close to a programming language and has a C-like syntax. It is easy to describe a processor in this language thanks to specific structures managing the instructions and the bit processing. It is also possible to describe a basic pipeline. However, general hardware cannot be described in IDL.

3.1.1 The execution model of the ST10

The registers define the state of the core. These registers are of course modeled by variables (in terms of programming language). In order to model the fact that when assigned registers take their new values only at the next cycle, we can use intermediate variables to store the future values of the registers. With regard to the ST10, these registers are memory-mapped registers. They mirror a part of the internal memory which is simulated by an array.

The simulation of the ST10 model core can be decomposed into 5 sequential basic steps. On top of the four step corresponding to the pipeline stage we inserted an extra step for the management of of external events. This step handle interrupts, inputs outputs, etc. Rest of the simulation follows the 4 steps execution scheme described above.

3.1.2 The parallelism

Parallelism is omnipresent in the core of the ST10 as it is a pipelined processor. The simulation of the pipeline with a sequential programming language like IDL can be done by executing sequentially each stage following the order of the pipeline : fetch, decode, execution and write-back.

In order to avoid stalls, the ST10 core integrate by-passes between blocs. This parallelism cannot be simulated sequentially with a one cycle grain. In order to simulate without slowing down the simulation by going to the next simulation grain, we choose to move the execution of the parts that need a by-pass to the next cycle. With this choice, the internal simulation is not cycle-accurate anymore, but as those by-passes do not deal with external event the difference is not perceived.

3.1.3 The interrupts

Interrupts are signals causing breaks in the processor instruction sequence. Because of their high priority, interrupts have to be handled with first at each cycle by the simulator. Those interrupts can be simulated by flags that have to be checked at the beginning of each cycle. If an interrupt occurs, the simulator has to jump to the appropriate interrupt vector, and change the state of the core (for instance masked interrupts).

3.2. Peripheral models

The peripherals of a microcontroller are actually kinds of ASICs which are integrated with the core on the same circuit. Usually the core disposes of specific means to access them: registers and interrupts. Those peripherals may be very different but they often have the same (classical) structure: a control part, which is an automaton, and an operative part which may be: registers, computation, or analog parts.

3.2.1 Simulation of the peripherals

The simulation of the control parts of the peripherals is the same as the simulation of finite states machine (FSM). It is easy to do the simulation of a FSM in a language like C with the use of a switch/case-based structure on a variable that represents the state. Figure 3 gives an example, at each cycle the function do_cycle() is executed with a new state.

The simulation of the operative parts is quite simple for non analog peripherals: if it is a register, the simulation is done through variables and if it is a computation, it is simply simulated through the corresponding operation of the programming language which is used (C++ in our case).

```
Periph::do_cycle()
{
  textbfswitch (state)
  {
    case 0 :
      .........
      state=1;
      break;
    case 1 :
      .........
      state=2;
    break;
      .........
  }
}
```

Figure 3. Simulation of a peripheral's FSM.

3.2.2 The analog parts

The problem here is how to represent the analog values on a software simulator (e.g. digital). We have to make two choices: the choice of the format of representation (floating point, fixed point, integer) and the choice of size and precision of the representation. Actually, these choices completely depend on the application. For the analog/digital converter (ADC) of the ST10, we opted for 16 bits integers, because the ADC converts into 10 bits integers anyway.

3.2.3 The communication between the core and the peripherals

General case

Two methods are commonly used to perform such communication:

- the use of specific registers (the SFRs for the ST10) to simulate it, this case, one can use variables representing the registers that are shared by the core simulator and the peripheral simulators.
- The use of specific interrupts, in this case, one has to set up a communication system working with flags between the core simulator and the peripheral simulators. For instance these signals may be variables which will be read by the core at each cycle to know whether there is an interrupt or not.

Most of the time access by register and access by interrupts are used together: registers are used for the configuration of the peripherals and for the storing of the results, and interrupts allows the peripheral to warn the core in case of an event.

The case of the external bus controller of the ST10

The external bus controller is an exception for two reasons: it is a complex circuit capable of supporting various communication protocols and its links with the core are transparent to the user. For the simulation of the external bus behavior we had to program an artificial stall of the pipeline occurring at various moments (instruction or operand fetch, write-back). To obtain a correct simulation, we had to list all possible cases (instruction fetch of 2 bytes followed by external operand fetch, instruction fetch of 2 bytes followed by write-back, etc.) and handle them one by one.

4. Results and applications

4.1. The model of the ST10

The complete design of the ST10 model took 5 months: 3 months and 2 persons to code the core and the peripherals, and 2 months and only 1 person to adjust the cycle-true aspect. The final simulator takes approximately 90,000 code lines, the bulk of which concerns the core (that was automatically generated by Flexsim [7]). The various performance tests showed that the simulator can run at a speed between 30,000 and 50,000 instructions/second on an Ultra Sparc station of 167 MHz having it run without trace. This constitute a huge improvement of performance when compared to existing ST10 simulators based on VHDL [6] that provides less than 10 instruction per second. Those tests also showed that the peripherals use few resources in comparison to the core (approximately 20% of the core use when they all run).

4.2. Integration into a co-simulation environment

We used the co-simulation [4] tools developed by Le Marrec [6]. This is a distributed co-simulation environment base on a backplane. The environment allows an easy integration of new simulator engines thanks to an open API. The co-simulation environment supports several languages including Matlab, C and VHDL. It interacts with simulators using two generic functions : EXIN for data input and EXOUT for data output. The backplane also allows to interconnect several instances of the same simulator. This possibility implies the possibility to have several copies of the same simulator running in parallel.

4.3. Simulation of a multi-processor architecture

We investigated in particular simulations involving several processors. Therefore we opted for architecture with several blocks hardware (ASIC) and software (ST10). Thanks to the ST10 simulator we have been able to perform cycle-true co-simulations.

5. Evaluation and Future work

During the design and the test of the simulator, we noticed that some operations computed by the simulator were often useless. Moreover, it might be overdoing it a little bit when reproducing a given architecture just to simulate it. The present simulator may then be improved. There are the two main sources for optimization: the pipeline simulation and the flag management. We believe that with these solutions it will be possible to obtain much faster simulators (e.g. 5 times faster).

5.1 New management scheme of the pipeline

As seen before, it is difficult to design the simulation of the pipeline (in particular due to dependency problems) and it is very costly in terms of computation resources. In the present version we include the structure of the pipeline in the simulator in order to make that the external communications occur at the correct date. Since we need to simulate only the temporal aspects of the pipeline, we may avoid to keep the pipeline structure. The basic principle is then to simulate instruction by instruction - without dealing with the pipeline - using a local memory. All the external accesses are fetched, and they are dealt with independently thanks to a scheduler that simulate the timing constraints. This solution may keep the cycle true accuracy of the simulation.

5.2. Flag management

Flags are bits indicating the result of an instruction. Processors usually have the following flags: C bit (carry indicator), Z bit (set when the outcome is equal to 0), N bit (set when the most significant bit of the outcome is equal to 1) and V bit (set when there is an overflow).

Most of the time there are still other bits. Their electronic implementation is not very expensive (a few gates are enough) but it is not the same story for their simulation. As a matter of fact, every bit has to be calculated which gives each time an additional sequential operation. Eventually the simulator would spend more time calculating the bits than executing the instruction itself, whereas in fact those flags are seldom used in the programs. They are only used by the conditional jumps and a few other kinds of instructions (such as carried additions). However almost all the instructions modify them.

A major optimization would indeed be to calculate them only when it is necessary. A possible algorithm is shown in figure 4. Afterwards, it has to be adapted to the processors. For instance in the case of the ST10, there are two kinds of instructions that modify the flags: instructions that modify all of them (like ADD, SUB, ROL, etc.) and instructions

```
if (I[n+1] does not compute the flags)
 or (I[n+1] need the flags)
then
  compute the flags for I[n]
end if
```

Figure 4. Basic decision algorithm for the computing of the flags

```
if (I[n] is a MOV)
then
  if (I[n+1] does not compute the flags)
   or (I[n+1] need the flags)
  then
    compute the flags for I[n]
  else
    if (I[n+1] does not compute the flags)
     or (I[n+1] is a MOV)
     or (I[n+1] need the flags)
    then
      compute the flags for I[n]
    end if
  end if
end if
```

Figure 5. Decision algorithm for the computation of the flags in the case of the ST10

that modify only some of them (like MOV). The algorithm has to be extended to both cases (cf. figure 5). It is possible to refine these algorithms even more in order to calculate the flags less often. However this refinement should not be done at the cost of the detection speed. We have to make a trade-off between the number of flags needlessly calculated and the complexity (and consequently lack of speed) of the detecting algorithm. The debugging possibilities offered by the simulator brings about another problem: a user may be interested in knowing the value of the flags at each cycle which will not be possible if they are only calculated when necessary. Therefore, the presented algorithms may well be a reasonable compromise solution: indeed it is only when the flag modification is positively without effect that it is not calculated.

6. Conclusion

The design of the current systems containing both software and hardware parts requires co-simulation techniques. To that end we need simulators that are easy to integrate into a co-simulation environment and that are fast enough to simulate a rational laps of time of real running. Above all, there are speed problems in regard to low-level simulators. It is with this background in mind that we designed the cycle-true simulator for the ST10 microcontroller. It is able to execute up to 50,000 instructions per second on an Ultra Sparc 167 MHz. Its flexibility allowed us to integrate it easily into a co-simulation environment and to check it on a few co-simulation examples.

This paper presented the global approach and detailed the techniques which allowed us to design the simulator. These techniques are general enough to be applied to other kinds of microcontrollers and cover following aspects: general simulation of a processor core, simulation of a simple pipeline, simulation of a pipeline that is able to avoid stalls, simulation of simple peripherals and simulation of the communications between the core and peripherals through interrupts and registers. The analysis of the present version of the ST10 simulator has shown that it is still possible to gain some speed by working on the pipeline and the flags.

Acknowledgments

The Authors would like to thank all those thanks to whom this project would never have been accomplished: Philippe Le Marrec of TIMA laboratory, Olivier Haller, STMicroelectronics and more specifically Pierre Paulin Michel Favres and Christophe Echwald, and PSA. This work was supported by the projects MEDEA SMT(A403) and CIME(A452).

References

[1] J. A.Rowson. Hardware/software co-simulation. *Automation Conference (DAC '94)*, 1994.

[2] G. DeMicheli M. Sami. Hardware/software codesign. *NATO Advanced Study Institute Workshop on Hardware/Software Codesign*, June 18-30 1995.

[3] D. Gajski F. Vahid S. Narayan J. Gong. Specification and design of embedded systems. *P T R Prentice Hall*.

[4] K. Hagen H.Meyer. Timed and untimed hardware / software cosimulation : Application and efficient implementation. *Internationaal Workshop on Hardware-Software Codesign, Cambridge*, October 1993.

[5] K. Hwang F.A. Briggs. *Computer Architecture and Parallel Processing (chap 3)*. McGRAW-HILL international editions, 1985.

[6] P. Le Marrec C. Valderama F. Hessel A.A. Jerraya. Hardware, software and mechanical cosimulation for automotive applications. *RSP'98*, June 1998.

[7] Y. Mroth. Writing a model of a processor with flexsim. idl language tutorial 1.0. *STMicroelectronics*.

[8] SIEMENS. C166 - 16 bit architecture & instruction set manual - version (09.95).

[9] SIEMENS. C167 / c167sr - product overview.

[10] SIEMENS. C167 users manual. version (03.96).

[11] STMicroelectronics. Family preliminary user manual (st10 family).

[12] STMicroelectronics. St10166 16-bit mcu user manual.