Modeling I²C Communication Between SoCs with SystemC-AMS

Mohamad Alassir, Julien Denoulet, Olivier Romain, Patrick Garda
Université Pierre et Marie Curie – Paris 6 – EA2385
3 Rue Galilée – Boîte Courrier 252
94200 – Ivry sur Seine, France
Email: mdibassir@hotmail.com, julien.denoulet@upmc.fr, olivier.romain@gmail.com, patrick.garda@upmc.fr

***Abstract*— In this paper, we show how to model the I²C bus communications between the nodes of an embedded system. For this purpose, we use the SystemC-AMS model of an I²C bus controller IP based on the PFC8584. Then we show how this IP can be included into two kinds of embedded systems nodes: on the one hand a 8051 micro-controller node, on the other hand a MIPS based SOC node. Finally the simulation results show the successful operation of the multi-master I²C bus communications between these two nodes in SystemC-AMS. Moreover the SystemC-AMS simulation introduces a small 10% overhead over the digital SystemC simulation for the two nodes multi-master I²C communication.**

I. Introduction

Nowadays, the design of embedded systems is a key issue for the electronics industry [1]. Many ongoing research projects address this issue by increasing the design abstraction level from the circuit to the system level [2]. However, most embedded systems gather analog and digital electronics with embedded processor cores running large amounts of embedded software. Therefore, neither VHDL-AMS nor SystemC are well-suited to model the architecture of these embedded systems. On the one hand, VHDL-AMS is perfect to model analog and digital electronics, but the simulation speed of processor cores is too slow in VHDL for the simulation of embedded software [3]. On the other hand, SystemC is perfect to model digital electronics and processor cores executing embedded software, but it cannot model analog electronics [4]. These observations led to the proposal of SystemC-AMS, for which beta releases of the language and the simulation engine are now available [5].

We focus in this paper on field bus communication modeling, as most embedded systems are built out of a set of nodes connected through field busses such as I²C or CAN. These nodes are typically mixed analog and digital SOC, including one (or several in the near future) processor core(s) and an array of peripheral interfaces. In its most usual form, the node is simply a micro-controller, connected to various sensors or actuators. Automotive electronics provide a number of such embedded systems.

In this paper, we show how to model the field bus communications between the nodes of an embedded system. Whereas our methodology is generic, we will present it in the specific case of the I²C bus for didactical purposes. Thus, we first present the key features of the I²C bus and we describe the SystemC-AMS model of an I²C bus controller IP we already introduced in [11]. Then we show how this IP can be included into two kinds of embedded systems nodes: on the one hand a micro-controller, on the other hand a SOC. Finally we give the simulation results and performances for the I²C bus communications between these two kinds of nodes in SystemC-AMS.

II. I²C Protocol

Historically, the first I²C (Inter-Integrated Circuit) controllers and the I²C protocol were designed by Philips at the beginning of the eighties for television applications [7]. The I²C bus was invented to provide communication on a two-wire bi-directional bus – a serial data (SDA) and clock (SCL) – between a small number of devices (sensors, LCD, micro-controller,…). The main technical characteristics of an I²C bus controller IP are as follows:

- Compatible with Philips I²C standard
- The parallel interface must be compatible with standard micro-controllers / micro-processors
- Transfer frequency is up to 100 kbits/s for standard transmission mode
- 7 bits addressing mode
- Start / Stop / Acknowledge generation and detection
- Bus busy detection

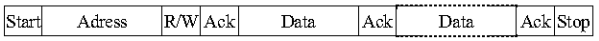

Fig. 1. I²C data frame

Data frame for the standard mode is made of a start bit, a 7 bits address, a Read/Write bit, an acknowledge bit (ACK) and a sequence of data bytes. Each data byte is followed by an acknowledge bit issued by the target device. A stop bit finalizes the transmission (Figure 1). Each bit is transmitted on SDA in conjunction with the SCL clock.

- Transfers are initiated by a START condition. It happens when a falling transition occurs on the SDA line while SCL is high (Figure 2).

1-4244-0755-9/07/$20.00 ©2007 IEEE

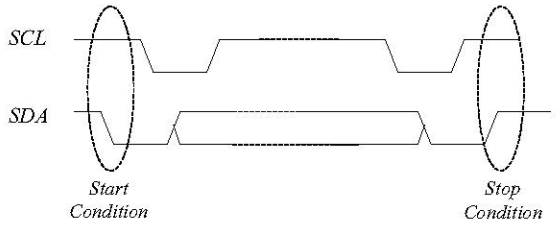

Fig. 2. Start and Stop conditions

- Transfers end with a STOP condition. It happens when a rising transition occurs on the SDA line while SCL is high (Figure 2).
- Data is considered valid during the high state of SCL. Therefore, SDA signal must remain stable during this half period (Figure 3).

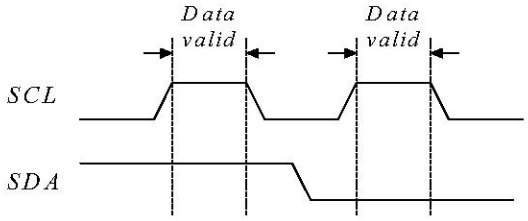

Fig. 3. Data validity period

- I²C allows multi-master communications and features an arbitration management protocol for such transmissions. Arbitration takes place on the SDA line, while the SCL line is at the high level. Bus control is given to the master which transmits a low level while the others transmit a high level. A master which loses arbitration switches to high impedance its data output stage (Figure 4).

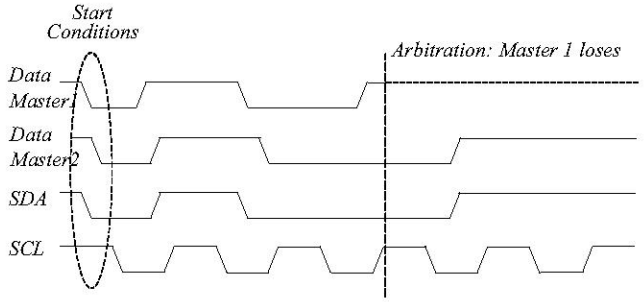

Fig. 4. Arbitration procedure with two masters.

As a specification for our I²C controller model, we chose the PFC8584 designed by Philips [8].

III. I²C Bus Controller Architecture

This section presents our SystemC and SystemC-AMS mixed model for an I²C bus controller. This was already introduced in [11].

A. General Architecture and Modeling Language

We concentrated our work on an I²C bus controller IP model. One of the reasons for such a choice is that I²C provides a simple example of a mixed-signal interface, unlike other busses such as USB or IEEE1394. Also, I²C is still often used in embedded systems. Still, the following model of architecture could be applied to other field busses, such as CAN.

The basic structure for our controller IP features two distinct blocks (Figure 5): a digital block, interfaced with a master microprocessor, manages the protocol, timing and control of specific sequences, while an analog block ensures the access to the external bus and models its behavior.

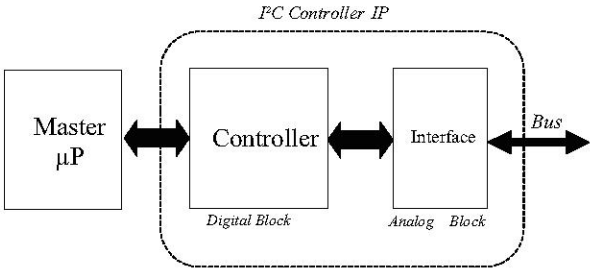

Fig. 5. Block architecture for the I²C bus controller IP

One of the issues raised by modeling mixed designs (either hardware/software or analog/digital) is the diversity of environments and languages involved [6]. The emergence of SystemC has provided a unifying solution which is now completed by the development of a SystemC-AMS library. Since both are C++ libraries, they can be interfaced to describe a mixed design. Thus, it allows to model at a system-level, and in the same simulation environment, the execution of a microprocessor code and the response of the analog parts of a system. It also provides faster, while still accurate, simulation results than with a VHDL-AMS environment. As a result we used SystemC 2.1 to model the digital block and SystemC-AMS 0.15 to model the analog block of our IP.

B. Digital Block Architecture

The architecture of the digital block is divided into three blocks (Figure 6).

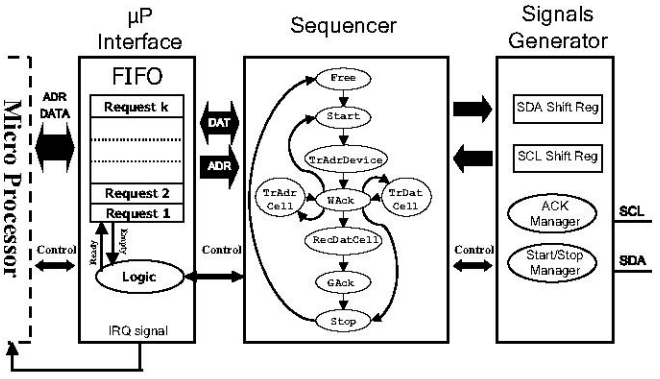

Fig. 6. Controller digital block architecture

- A micro-processor interface handles communication with the microprocessor core. It is built around a FIFO that stores the successive requests coming from the microprocessor bus (8051 bus, VCI, AMBA, etc…). When the controller has finished a transaction on the bus, and if a request is stored in the FIFO, the FIFO is read and the interface extracts the information needed by the sequencer (type of operation, address, data) to perform the new communication. The interface also includes an interrupt line connected to the microprocessor core so that it can read a data received from the I²C bus.
- The core of the controller is a sequencer which translates the request from the master into a detailed sequence respecting the I²C protocol (frame generation, byte transmission or reception, etc…).
- Finally, a signal generator module manages or drives the SCL and SDA bus lines (Figure 7).

This block manages the Acknowledge generation/detection depending on the operating mode (transmitter or receiver). A shift register either serializes data to be sent to the bus line in transmitting mode or collects information from the bus in reception mode. It also sets the transmission frequency by dividing the system clock with a user defined constant.

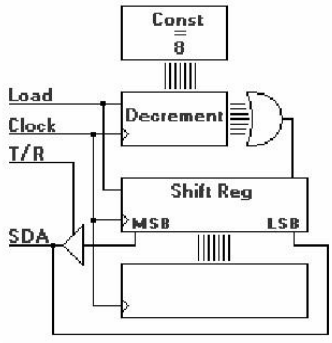

Fig. 7. SDA line manager

C. *Analog Block*

The specifications of the I²C protocol indicate that the the devices must have open-drain or open-collector outputs (depending on technology) in order to perform a wired-AND function to manage multi-master mode. Both lines also feature a pull-up resistor to VCC (Figure 8.a).

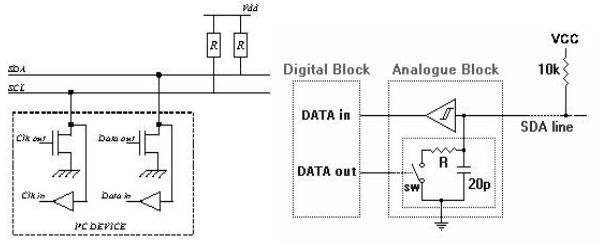

Fig. 8. (a) I²C electrical scheme (b) SystemC-AMS model.

We modeled this analog interface with the SystemC-AMS 0.15 library, by instantiating linear elements (resistors, capacitors, voltages sources, etc…). Figure 8.b represents the interface model for the SDA line. It translates the logic levels sent by the digital block to voltages across the bus lines. The SCL model is similar.

In this model, the transistor is represented with an interrupter and a resistor as no transistor model is available in SystemC-AMS 0.15. A capacitor is used to manage the rising and falling times of the SDA signal, according to the PFC8584 component specifications. A change in its value will modify the duration of the rising and falling times. Finally, a pull-up resistor sets the line at a high state when no command is applied on the bus. To read a value coming from the bus, a threshold detection is applied to the SDA line and it provides a logical value to the digital block. Figure 9 gives a SystemC-AMS sample code of the analog interface for either line of the bus.

```
SCA_SDF_MODULE(ADC){   // Threshold Detector
   sca_sdf_in<double> in;
   sc_out<bool> out;

   void init(){}
   void sig_proc(){
        if(in.read() > 2.5) out = 1;
        else out = 0;}

   SCA_CTOR(ADC){}
};

SCA_SDF_MODULE(data_conv){
   sca_sdf_in<double> portin;
   sc_out<double> portout;

   double portout_;
   void init(){ portout = 1.0e3;}
   void sig_proc(){
        portout_ = portin.read();
        portout = portout_ * 3.0e2;}
   SCA_CTOR(data_conv){}
};

SC_MODULE(SDA_Mgr){
   sc_in<bool> sda_in;    // From digital block
   sc_out<bool> sda_out;  // To digital block
   sca_sdf_signal<double> sda_sdf; // To I2C Bus

   sc_signal<double> r_value;
   sc_signal<bool> sig_;
   sca_elec_port out;          // Output port
   sca_elec_port gnd;          // Ground port
   sca_elec_node y;            // Internal node

   sca_sc2r *r1;               sca_c *c1;
   sca_rswitch *sw1;           sca_vd2sdf *conv1;
   data_conv *data_conv1;      can *can1;

   SC_CTOR(SDA_Mgr){

    ADC1=new ADC("ADC1");      // Threshold
       ADC1->in(sda_sdf); ADC1->out(sda_out);

         sw1=new sca_rswitch("sw1");  // Switch
       sw1->off_val = true; sw1->p(y);
       sw1->n(gnd); sw1->ctrl(sda_in);

    r1=new sca_sc2r("r1");     // Resistor
       r1->ctrl(r_value); r1->p(out); r1->n(y);
```

```
    data conv1=new data conv("data conv1");
       data conv1->portin(sda sdf);
       data conv1->portout(r value);

    c1=new sca_c("c1");        // Capacitor
       c1->value = 20.0e-12;
       c1->p(out); c1->n(gnd);

     conv1=new sca_vd2sdf("conv1"); // Output voltage
        conv1->p(out); conv1->n(gnd);
        conv1->sdf voltage(sda sdf);
     }
};
```

Fig. 9. I²C controller analog block code sample

IV. I²C Controller Simulation

The I²C controller has been simulated in two different contexts: first in association with a micro-controller used as a field bus node, and then in a System-on-Chip node, using a SoC virtual prototyping platform. Finally both nodes have been connected via the I²C bus.

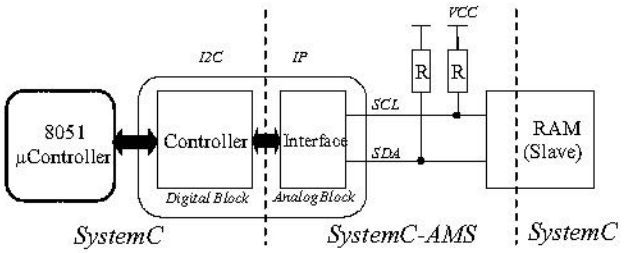

Fig. 10. 8051 Simulation with slave peripheral

A. Simulation of a Micro-controller Node

In this first simulation, we interfaced our controller IP with a 8051 micro-controller[9] SystemC model. The I²C bus featured in Figure 10 connects the "8051 + controller" node to a slave RAM device. The model of this RAM peripheral includes an analog interface in SystemC-AMS, similar to the one introduced in section III-C, which receives or sends data to the I²C bus and a SystemC part modeling the memory cells.

The following assembler code (Fig.11) is compiled and stored in the internal ROM of the micro-controller. When it runs, the program performs an I²C write operation: the data byte 4Eh is written on the B4h cell address of the slave RAM device.

```
        //      Write in RAM(slave address = 0x50)
        //      the value 0x4E in cell 0xF1

main:   MOV     R0, #A0 // Put slave address in R0 register
        MOV     A, #B4  // Put cell address in A register
        MOVX    @R0, A  // Send R0+A to I2C IP
        MOV     R0, #A0 // Put slave address in R0 register
        MOV     A, #4E  // Put data byte in A register
        MOVX    @R0, A  // Send R0+A to I2C IP
```

Fig. 11. 8051 Assembler code sample Byte writing operation.

This write takes place in two stages: first the IP sends the cell address to the slave device (the A0h value is made of the 50h 7-bit slave address and the R/W bit set to 0), then it writes the data byte to this same slave device.

The chronogram in Figure 12 represents the analog behavior of the I²C bus lines during this transmission. We recognize the addresses and data sent by the 8051 master and the start/stop/ack bits inserted by the controller.

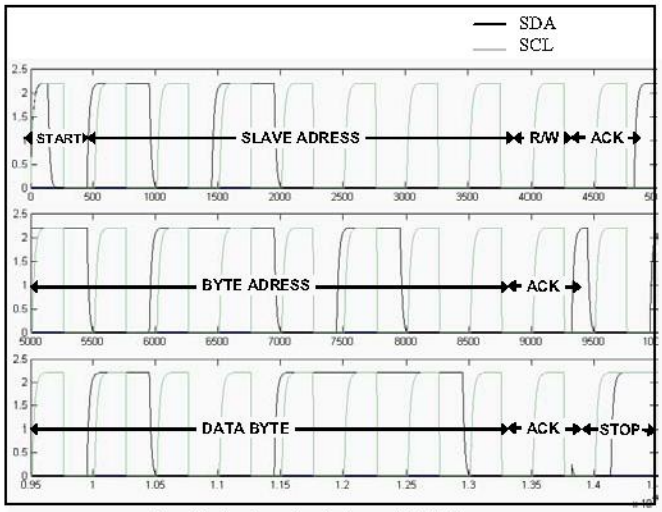

Fig. 12. Analog simulation with 8051 master

B. Simulation of a SoC Node

To simulate our model in a SoC node, we used a virtual prototyping platform for System-on-Chips called SoCLib [10]. SoCLib provides a number of IP models (including processor cores, VCI interconnect, memory devices, I/O controllers, timers, etc…), to easily simulate an application on a user-assembled SoC or MPSoC architecture built around a VCI bus.

We added our controller model to the SoCLib library and we used it to connect a MIPS processor core to an external I²C bus. Then we created a simulation platform (Figure 13) where the MIPS is able to dialog with two external I²C RAMs and the "8051+I²C controller" node introduced in section IV-A.

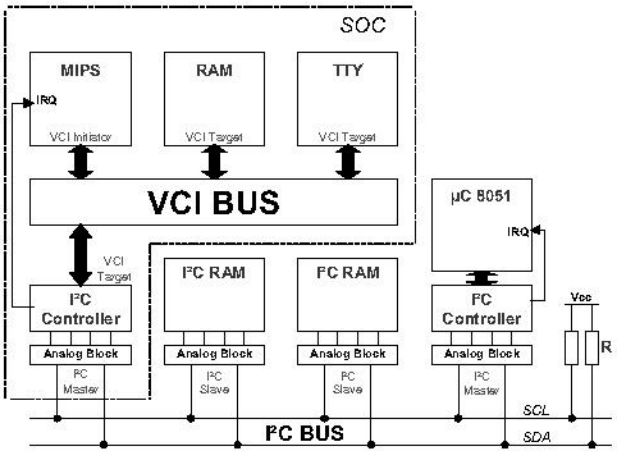

Fig. 13. SoCLib simulation platform

On the SoC part of the platform, a SystemC wrapper has been developed to interface our controller to the VCI bus provided by SoCLib. The platform also includes a RAM component used by the MIPS to store its code and data and a TTY terminal for debug purposes. All the SoCLib components are modeled in SystemC, while the analog interfaces and the I²C bus are described in SystemC-AMS with the models presented in section III-C.

Figure 14 gives a sample of the MIPS code, featuring the assembler functions used to request the controller IP a read or a write on the I²C bus, as well as the interrupt subroutine which is called when a read data has been received from the I²C bus and is available for the MIPS master.

```
// Interrupt Subroutine
void SwitchOnIt(int it) {

    int i;

// Identify the active interrupt of highest priority
    for (i=0; i<8; i++) if (it&(1<<i)) break;

    switch (i) {
       case 0: break;          // SwIt 0
       case 1: break;          // SwIt 1
       case 2: read interface(); break; // It 0
       case 3: break;          // It 1
       case 4: break;          // It 2
       case 5: break;          // It 3
       case 6: break;          // It 4
       case 7: break;          // It 5
       default: break;
    }
}

// Main Program
int main(void)
{
    write byte(0x50,0x57,0x69);
    // Parameters = Slave Adr  (stored in $4 register)
    //              Cell Adr   (stored in $5 register)
    //              Data Byte  (stored in $6 register)


    read byte(0x51, 0xf1);
    // Parameters = Slave Adr  (stored in $4 register)
    //              Cell Adr   (stored in $5 register)

    while (1);
    return 0;
}

// Byte Writing ASM Code Sample
write byte:

    la  $3,0xc0000000  // Load VCI target (I2C IP)
                       // address to $3 register
    sll $4, $4, 9      // Slave adr in high part of register
    or  $4, $4, $5     // Cell adr in low part of register
    sh  $4, 0 ($3)     // (save half-word)
                       // send target+cell adr to IP
    sb  $6, 1 ($3)     // Send data byte to IP
    nop
    j   $31            // return from subroutine
.end write byte

// Byte Reading ASM Code Sample
read byte:

    la  $3,0xc0000000  // Load VCI target (I2C IP)
                       // address to $3 register
    sll $4, $4, 9      // Slave adr in high part of register
    ori $4, $4, 0x100  // Set R/W Bit to Read
    or  $4, $4, $5     // Cell adr in low part of register
    sh  $4, 0 ($3)     // save half-word ->
    nop                // send target+cell adr to IP
    j   $31            // return from subroutine
.end read byte

// IP Interface Reading ASM Code Sample
read interface:

    la  $3, 0xc0000000 // Load VCI target (I2C IP)
                       // address to $3 register
    lbu $2, 0 ($3)     // Load data byte from I2C IP
    nop
    j   $31            // return from subroutine

.end read interface
```

Fig. 14. MIPS code sample.

Our first simulation displays a sequence of I²C operations performed by the two masters, including one write and one read operation for each master device.

In particular, we can see in Figure 15 the 8051 writing the 4Eh byte on the F1h address of an I²C RAM, and then the MIPS reading this same RAM address to get the value just written by the 8051. This SystemC chronogram shows the digital behavior of the SDA bus line (featured on the last row), as well as the SDA commands sent by each component, indicating which device has control of the bus at any given time. We can also notice that an IRQ signal is generated when the 4Eh value is available in the MIPS I²C controller.

Our platform also allows us to test the multi-master arbitration of the I²C bus, as it is implemented in our model. In the chronogram shown in Figure 16, both masters send a start bit at the same moment. The SDA bus line follows the two commands as long as they're identical, and when they differ, the master who sends a 0 bit (in this case the MIPS) takes control of the bus and the 8051 stops transmitting on SCL and SDA. Also of interest is the wired-AND function performed on the SDA and SCL lines of the bus. This shows that our mixed controller model successfully performs electrical multi-master arbitration.

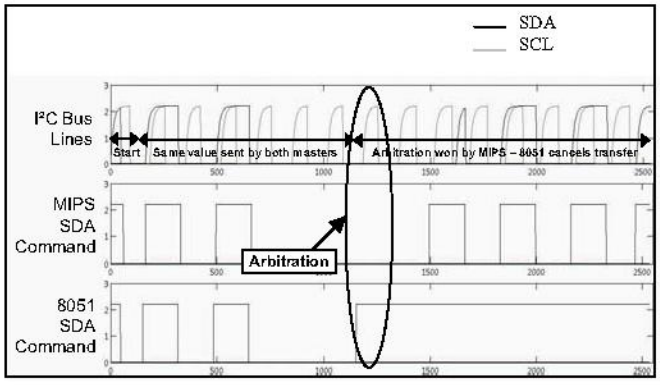

Fig. 16. Multi-master mode arbitration

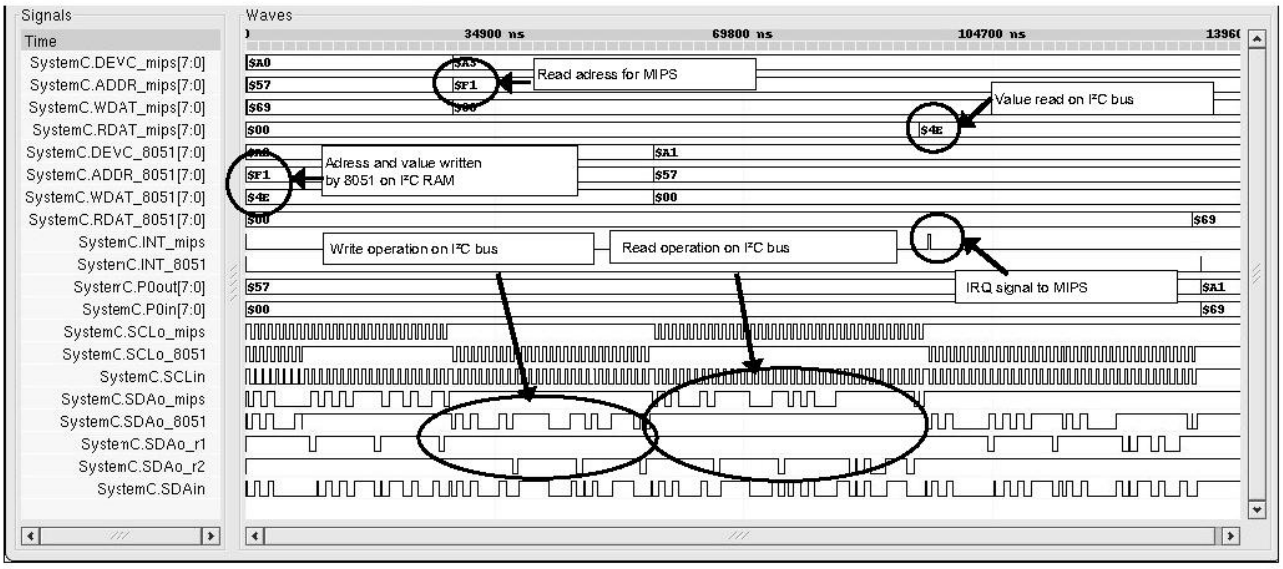

Fig. 15. Simulation with 8051 and MIPS master

C. *Simulation Performance*

Table I presents simulation speeds for the two platforms presented in section IV. The measurements were performed on a Linux workstation equipped with a Pentium M processor running at 1.73 GHz, a L2 cache of 2 Mb and 1 Gb of SDRAM.

Results show that for the 8051 platform the overhead due to the AMS parts of the model is relatively important, though simulation is still very fast. The impact is less prominent with the SoCLib environment, mainly because the proportion of purely SystemC components is much more important. In the latter configuration, analog simulation of the I²C bus is performed only 10% slower than a digital-only simulation.

TABLE I
SIMULATION PERFORMANCE FOR BOTH PLATFORMS IN CYCLES/S

Application	SystemC blocks only	Sys.C+Sys.C-AMS complete model	Overhead
IP+8051 (IV-A)	**149300**	**106 400**	**40%**
IP+MIPS and IP+8051 (IV-B)	**25800**	**23 500**	**10%**

NB: Sampling period: 5 ns _ I²C transmission frequency: 400 kHz
→ 500 points per period

V. CONCLUSION

In this paper, we showed how to model the I²C bus communications between the nodes of an embedded system. For this purpose, we used the SystemC-AMS model of an I²C bus controller IP we previously introduced. Then we showed how this IP can be included into two kinds of embedded systems nodes: on the one hand a 8051 micro-controller node, on the other hand a MIPS based SOC node. The simulation results showed the successful operation of the multi-master I²C bus communications between these two nodes in SystemC-AMS. The SystemC-AMS simulation times showed different overheads over digital SystemC: 40% for the 8051 to RAM I²C communication and 10% for the two nodes multi-master I²C communication. This overhead is acceptable for a mixed-signal simulation. Finally, observe that our methodology is generic and we will apply it to other field buses such as the CAN bus.

REFERENCES

[1] M.Chiodo, D.Engels, P.Giusto, H.Hsieh, A.Jurecska, L.Lavagno, K.Suzuki, A. Sangiovanni-Vincentelli: A case study in computer-aided co-design of embedded controllers in Design Automation for Embedded Systems, Vol.1, n°1, 1996, pp.51-67, Springer

[2] J.Liu, X.Liu, and E. Lee: Modeling Distributed Hybrid Systems in Ptolemy II, in Proceedings of the American Control Conference, June 25-27, 2001, pp.4984-4985.

[3] J. Oudinot, J. Ravatin and S. Scotti: Full transceiver circuit simulation using VHDl-AMS, in proc. FDL'02, Esim, Marseille, France, Sept. 24-27, 2002

[4] T.Grčotker, S.Liao, G.Martin, S.Swan: System Design With SystemC, Kluwer Academics, May 2002.

[5] C. Grimm, K. Einwich, A. Vachoux: Analog and Mixed- Signal System Design with SystemC, FDL'04, Lille, France, Sept. 13-17, 2004

[6] International Technology Roadmap for Semiconductors Design, 2005 Edition, http://public.itrs.net

[7] Philips Semiconductors: The I²C-Bus Protocol Specification, Document Order Number: 9398 393 40011, January 2000, http://www.nxp.com/acrobat/literature/9398/39340011_21.pdf

[8] Philips Semiconductors: PCF 8584, I2C bus controller, http://www.nxp.com/acrobat/datasheets/PCF8584_4.pdf

[9] Intel, 80C51 Single-Chip 8-bit Microcontroller datasheet, ftp://download.intel.com/MCS51/datashts/27050108.pdf

[10] SoCLIB. A modelisation & simulation plat-form for system on chip, 2003. http://soclib.lip6.fr

[11] M.Alassir, J.Denoulet, O.Romain, P.Garda : Modelling and Simulation of an I2C Bus Controller in SystemC-AMS, in proceedings of FDL'2006, pp.121-127, Darmstadt, Allemagne, 19-22 Septmbre 2006