The ICSM, an Instruction Set Model for Simulating Embedded Software in Mixed-Technology Systems

Norman J. Elias
Norman J. Elias, Independent Consultant
32 Sturbridge Lane
Trumbull, CT 06611
norm.elias@ieee.org

Abstract

Mixed-signal, Mixed-technology simulation has evolved to meet the challenges of increasing hardware complexity. Embedded software adds a new dimension to this challenge. Meaningful system simulation must accurately model software timing, not just algorithmic behavior. This paper answers that challenge by developing an instruction set modeling approach to integrate software execution into the simulation. The basis of the ICSM is to resolve the model accuracy to the instruction boundaries, i.e., to accurately model the state of the CPU, data memory and I/O at the conclusion of each instruction. In comparison to clock-cycle accuracy, this instruction-cycle accuracy is simpler to model and faster to compute. The instruction-cycle simulation model (ICSM) approach is exemplified by specific ICSM's of Freescale's HCS12 and SGS-Thomson's ST7 Lite.

The paper presents a structural overview of the micro-controller model and then focuses on the CPU. Key elements of the ICSM include a database structure to store the instruction set, signals to represent the CPU registers, and algorithms for decoding the binary software, accessing data via the full set of addressing modes, executing the instructions, synchronizing software timing to the simulation, and storing results. Other features include exception handling for resets and interrupts. The text explains how the model data structures and algorithms are derived from product literature – datasheets and reference manuals. Models are implemented in VHDL-AMS and in a mix of MAST® and C language. The paper presents a validation of the models and illustrates their operation in system simulation applications from the domains of power electronics and automotive electronics.

1. INTRODUCTION

This paper introduces the concept of an instruction-cycle simulation model (ICSM) for modeling embedded software in mixed-technology systems. Precise simulation of the system must account for micro-controller timing, a critical factor that cannot be incorporated into a simple algorithmic model of the software. The ICSM models the software by executing the CPU instructions in sequentially and with timing resolved to the instruction boundaries. It computes the state of the micro-controller at the conclusion of each instruction and interfaces to the system hardware the micro-controller I/O ports. The ICSM is an instruction set simulation model [1] that omits cycle-accurate details within the instruction boundaries. It is linked to the system simulation by coding it in the native language of the simulator. This paper presents examples of an SGS-Thomson ST7 Lite processor that has been has been modeled in MAST and C for Synopsys' Saber [2] and a Freescale HCS12 that has been modeled in VHDL-AMS for Mentor's SystemVision [3].

Section 2 provides a structural overview of the full micro-controller model. Sections 3 and 4 present the key elements of the CPU model generically and with device-specific details for the two examples. The presentation highlights the modeling considerations specific to full mixed-technology system simulation. Section 5 discusses model validation and presents specific simulation applications to electronic power conversion and automotive throttle control. The conclusions in Section 6 discuss directions for

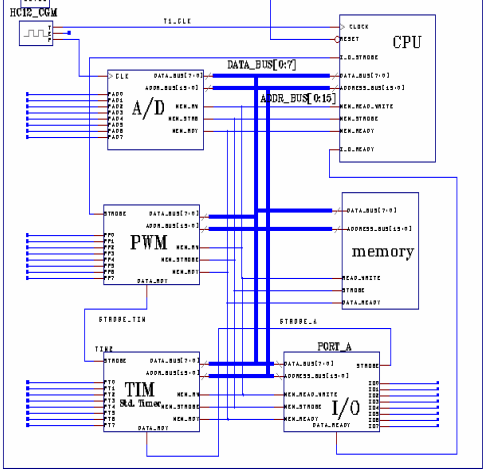


Figure 1 ICSM Structural Overview

future exploration of this modeling approach.

2. OVERVIEW OF THE MICRO-CONTROLLER MODEL

Figure 1 shows a top level schematic of a micro-controller including the CPU, memory, a data-port and a few selected peripherals. The CPU and memory are the core elements. Each micro-controller family offers a range of configurations with variations in the peripherals. For any simulation, it makes sense to include only those peripherals used by the system under study. Therefore, these peripherals are modeled as independent modules interconnected at the schematic level. These modules share memory access with the CPU.

The modules are all synchronized to the system clock shown at the output of the clock generation module (CGM). Memory is modeled as an integer array with simple utility functions for converting the data to and from bit vectors. The register map documented in the product literature is incorporated by declaring mnemonic constants for the memory addresses as illustrated in Figure 2. The model does not attempt to accurately reproduce memory transactions as would be required for a cycle-accurate model. Instead the ICSM accesses the memory as needed to compute the MCU state at the instruction boundaries.

```
--          I/O Ports
constant dra_addr:      integer := 16#0000#;    -- Port A DR
constant drb_addr:      integer := 16#0001#;    -- Port B DR
constant ddra_addr:     integer := 16#0002#;    -- Port A DDR
constant ddrb_addr:     integer := 16#0003#;    -- Port B DDR
constant dre_addr:      integer := 16#0008#;    -- Port E DR
constant ddre_addr:     integer := 16#0009#;    -- Port E DDR
```

Figure 2 Portion of Memory Map

The memory is programmed to load the software object code at the start of each simulation. The model reads the standardized S19 format so that it can execute software generated using any commercial compiler or IDE. This also guarantees that the model will execute the exact code that would be burned into a physical prototype.

The clock timing, memory configuration, peripheral functionality, register structure, and instruction set are all well documented in the product literature for each micro-controller. Models of the peripherals are relatively straight forward. The focus of this paper is on the CPU.

3. THE ICSM MODEL OF THE CPU

The essence of the ICSM approach is shown in Figure 3. This is the flow of operations that defines the CPU. The figure depicts a classical von Neumann computer architecture with a step inserted to synchronize timing at the instruction boundaries. Timing can be computed by counting explicit clock pulses or by multiplying the clock period by a pre-stored cycle count. Both approaches have been used successfully. The explicit clock pulse counter is computationally less efficient (in terms of simulator speed) but is more flexible.

The ICSM models a set of signals that constitute a software debugger at the CPU instruction level. The CPU signals include all of the registers listed in Figure 3 along with a text-base reconstruction of the instruction in assembly language format. This is presented in the waveform viewer as a text string that identifies the sequence of instructions executed during the simulation (See Figure 5.4).

3.1. Decode Instruction

The instruction decoder interprets the hex representation of each instruction from program memory. This is always a stream of bytes that can vary in length from one to, usually no more than, six. It's generally convenient to store a

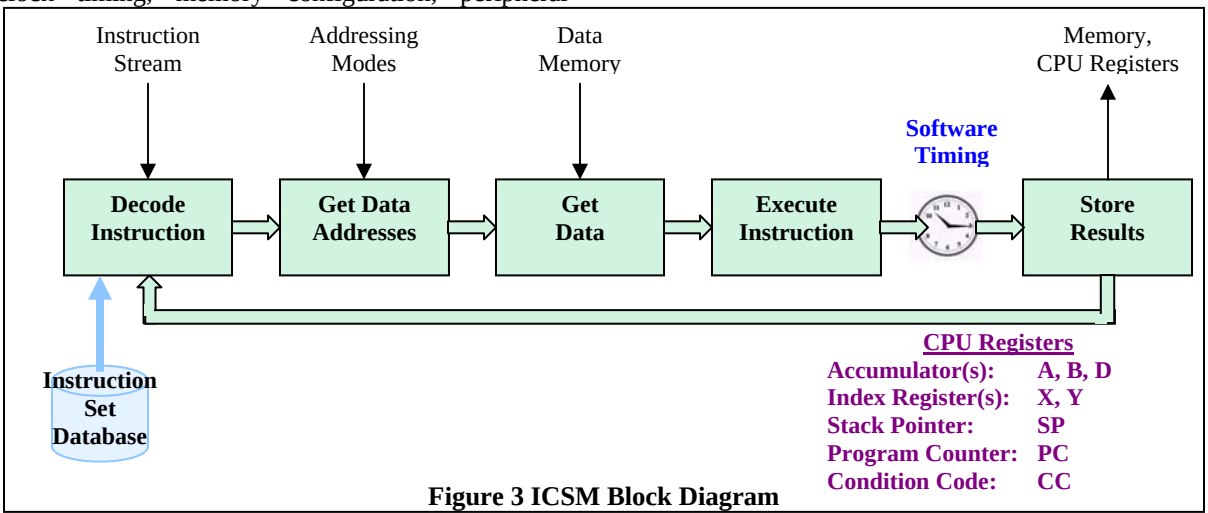


Figure 3 ICSM Block Diagram

limited sequence of bytes in a local array and to refill that buffer as needed from memory. The instruction decoder keeps count of the number of bytes read for each instruction not only to refill the buffer but to set a pointer to the next instruction. This pointer is the preliminary update to the program counter. The instruction decoder algorithm accounts for the device-specific structure of the instruction stream which is generically in the form of:

[pre-byte,] **opcode**, [post-byte, [post-byte,…]]

where the square brackets indicate optional bytes. The bytes identify the instruction to be executed and the mechanism to be used for accessing data. The decoder uses these bytes to index an internal database of instruction descriptors derived from the product documentation. The instruction descriptors provide all the information the model needs to execute the instructions. At startup, the program entry point is saved in the program counter. The CPU extracts the instruction stream starting at that address and decodes the bytes to determine all the information needed to execute the first instruction. At each succeeding instruction, the instruction stream is read starting at the updated program counter value.

3.2. Get Data Addresses, Get Data

Micro-controllers support a menu of mechanisms or modes for addressing memory. Inherent instructions require no data from memory but others may access (read or write) one or two bytes. Immediate addressing includes the data in the instruction stream. Direct modes provide the memory addresses in the instruction stream. Indirect modes provide pointers. The data addresses are themselves stored as data. Finally, there is typically a menu of indexed addressing modes which generate the addresses by adding or subtracting integer offsets to the contents of a specified CPU register. The offsets are programmed into the instruction streams. The classes of addressing modes are generic but the mechanisms for storing addressing information, especially indexing offsets are specific to each micro-controller. They must, therefore, be programmed explicitly for each MCU. Once an address is computed, it is a simple matter to get data from memory before execution or store results afterwards.

3.3. Execute Instruction

By separating the execution of the instruction from getting the data, storing the data or synchronizing timing, the ICSM focuses the execution on the computation of results. This computation is complex even by itself since it includes both primary and secondary effects. The primary effect is to update a CPU register or a data value as in computing the sum in an ADD instruction. Secondary effects include condition codes and program counter revisions as in the carry bit generated by an ADD or a jump generated by a branching instruction. Certain addressing modes call for post-incrementation of index registers which must also be computed when the instruction is executed.

The execute instruction module is readily implemented as a VHDL case construct (or switch in C) keyed to the instruction type. To keep the code compact, condition bits can be updated using separate functions called at the conclusion of the module using the results of the instruction execution to guide the condition code generation.

3.4. Store Results

The model can execute the instruction at any time after it is decoded so long as these results are not stored until the specified number of clock cycles have elapsed. Therefore, the ICSM separates the task of storing results from that of executing the instructions. Storing results is a straight forward process except for the added complication of asynchronous resets or high priority interrupts. If such an event takes place before an instruction is scheduled to complete, the MCU never completes the instruction. The ICSM models this behavior by inserting a simple test before storing the results. If a reset or similar event is detected (e.g., by a separate VHDL process), the model skips the store results step and responds to that reset event.

3.5. Program Counter, Stack Operations, And Interrupts

Program flow is dictated by the sequence of addresses written to the program counter. Sequential execution of instructions is guaranteed by keeping track of bytes read by the instruction decoder. The execute instructions module updates the program counter to account for branching, subroutine entry and return instructions. Stack operations are handled by the get data (POP) and store results (PUSH) modules.

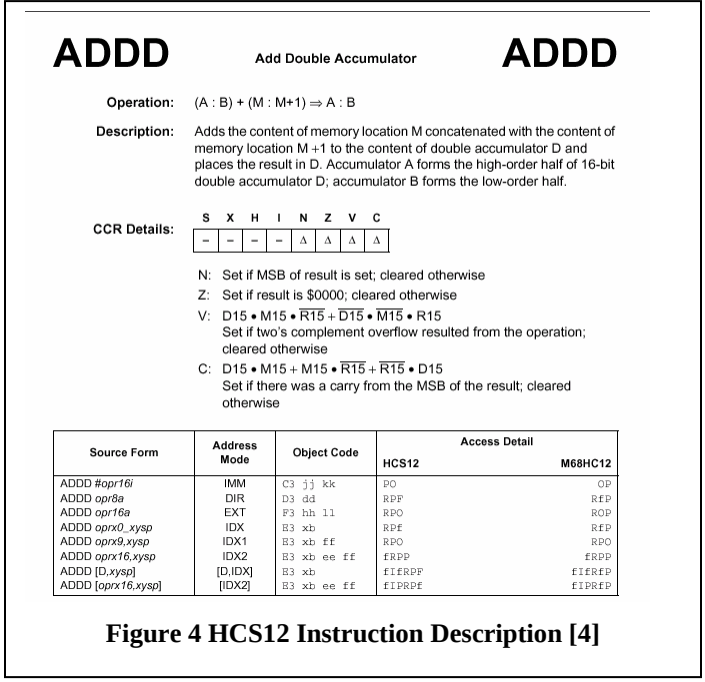
Interrupts are processed by a separate module, not shown in Figure 3. This module detects interrupts when they occur, sets a flag at that time and processes the interrupts after results are stored. In case of multiple interrupts, the module is programmed to process the highest priority as documented in the MCU datasheet. The interrupt module pushes the CPU state onto the stack and revises the program counter to force execution from the start of the ISR. The previous instruction sequence is resumed when

the return from interrupt is executed, at which time the CPU state is popped from the stack.

4. THE INSTRUCTION SET DATABASE

Figure 4 shows the description of an add instruction as published in the HCS12 reference manual. The instruction set database is a tabular representation of this information. The database stored as a disk file or coded directly into the model. A directly coded database saves time during the simulation and is easily implemented as an array of descriptors indexed to the opcode. The instruction decoder indexes the database (using pre-byte and opcode values) and then uses the stored description to guide further decoding of the post-bytes.

Figure 5 shows a sample entry for the ADDD instruction. The data is stored internally in order to avoid extra disk accesses during the simulation. Each entry provides information taken from the data sheet. The vector of cycle counts correspond to the distinct indexing modes indicated in Figure 4.1. Specific cycle counts are determined by counting the "Access Details" characters used in the HC12 reference manual to classify the individual clock cycles. The reference manual lists a variety of add instructions referenced to different CPU registers and distinct in terms of whether or not a prior carry bit is included. To avoid redundancy in the execute instruction module, the model classifies all of these different opnames as "ADD" optypes.



ADDD Add Double Accumulator **ADDD**

Operation: (A : B) + (M : M+1) ⇒ A : B

Description: Adds the content of memory location M concatenated with the content of memory location M +1 to the content of double accumulator D and places the result in D. Accumulator A forms the high-order half of 16-bit double accumulator D; accumulator B forms the low-order half.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $0000; cleared otherwise
V: $D15 \bullet M15 \bullet \overline{R15} + \overline{D15} \bullet \overline{M15} \bullet R15$
Set if two's complement overflow resulted from the operation; cleared otherwise
C: $D15 \bullet M15 + M15 \bullet \overline{R15} + \overline{R15} \bullet D15$
Set if there was a carry from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
ADDD *#opr16i*	IMM	C3 jj kk	PO	OP
ADDD *opr8a*	DIR	D3 dd	RPf	RfP
ADDD *opr16a*	EXT	F3 hh ll	RPO	ROP
ADDD *oprx0_xysp*	IDX	E3 xb	RPf	RfP
ADDD *oprx9,xysp*	IDX1	E3 xb ff	RPO	RPO
ADDD *oprx16,xysp*	IDX2	E3 xb ee ff	fRPP	fRPP
ADDD [D,*xysp*]	[D,IDX]	E3 xb	fIfRPP	fIfRfP
ADDD [*oprx16,xysp*]	[IDX2]	E3 xb ee ff	fIPRPf	fIPRfP

Figure 4 HCS12 Instruction Description [4]

5. MODEL VALIDATION AND APPLICATIONS

The ICSM is easily validated by comparing simulation results to either an existing software debugger or to a hardware evaluation board. A thorough test of each instruction must account for all addressing modes and all possible condition bit transitions. A single test can exercise a number of similar instructions as in the example shown in Figure 6.

```
-- ------------------------------------------------------------- -- instr_tbl
-- ADDD: Add to D (INDEXED)                          opcode E3     -- instr_tbl
b := 16#E3#*instr_descriptor'length;                               -- instr_tbl
c := 16#E3#*ncycle_descriptor'length;                              -- instr_tbl
                                                                   -- instr_tbl
next_instruction :=                                                -- instr_tbl
   (optypes'pos(ADD),       opnames'pos(ADDD),                     -- instr_tbl
    NONE,                                    -- postbyte type      -- instr_tbl
    of_DREG, NA,                    -- CPU register identifiers (reg_id1, reg_id2)
    INDEXED,    NO_ADDRESS,                  -- mem1 mem2 adressing modes
    NA,                                      -- memorder           -- instr_tbl
    NA,                                  -- push vs. pull stack    -- instr_tbl
    2,                              -- number of memory bytes accessed -- instr_tbl
    retain_bit, retain_bit, retain_bit, retain_bit,      -- s x h i  bit flags
    eval_bit,   eval_bit,   eval_bit,   eval_bit);       -- n z v c  bit flags
          -- IDX   IDX1  IDX2  [IDX2]  [D, IDX]                    -- instr_tbl
next_ncycles := (3,    3,     4,     6,      6);                   -- instr_tbl
                                                                   -- instr_tbl
for j in 1 to instr_descriptor'length loop tbl(b+j) := next_instruction(j); end loo
for j in 1 to undefined_ncycle'length loop cyctbl(c+j) := next_ncycles(j); end loop
                                                                   -- instr_tbl
-- ------------------------------------------------------------- -- instr_tbl
```

Figure 5 ICSM Instruction Descriptor

Applications of the ICSM have been published separately and demonstrate its role in detecting and correcting timing bottlenecks as well as other hardware/software interactions.

```
; IMM addressing
   ADDB #$1F   ; B <- $1F, SXHINZVC = 11010000
   ADCB #$E5   ; B <- $04, SXHINZVC = 11110001
   ADCB #$1F   ; B <- $24, SXHINZVC = 11110000

 ; DIR
   ADDB 3      ; B <- $24+mem($03) = $23,
               ; SXHINZVC = 11110001
   ADCB 3      ; B <- $23+C+mem($03) = $23,
               ; SXHINZVC = 11110001

 ; EXT
   ADDB $0800  ; B <- $23+mem($0800) = $93,
               ; SXHINZVC = 11011010
   ADCB $0800  ; B <- $93+C+mem($0800) = $03,
               ; SXHINZVC = 11010001

 ; IDX 5 bit offset (0 <= |offset| < 16=$10)
   ADDB 5, X   ; B <- $03+mem(X+5=$0800) = $73,
               ; SXHINZVC = 11010000
   ADCB 5, X   ; B <- $73+C+mem(X+5=$0800) = $E3,
               ;  SXHINZVC = 11011010
```

Figure 6 Portion of an ADD Instruction Test

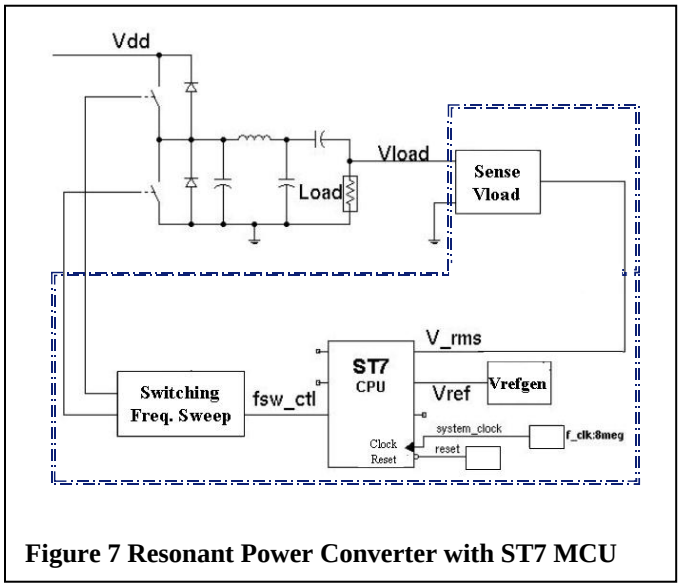


Figure 7 Resonant Power Converter with ST7 MCU

Figure 7 shows a resonant power converter schematic. The load voltage is regulated by modulating the switching frequency following a simple algorithm in the form.

```
while (true) {
    if (V_rms > Vref)  fsw_ctl = 1
    else               fsw_ctl = 0
}
```

Figure 8 shows the application of simulation results to an MCU evaluation. Clock speeds below 3.5 MHz are shown to degrade performance by introducing excess ringing in the load voltage amplitude. Performance can be improved at the lower clock speeds by software revisions to complete calculations within a single switching cycle. Satisfactory operation at a lower clock frequency translate into product cost reductions that can be critically important to profitability.

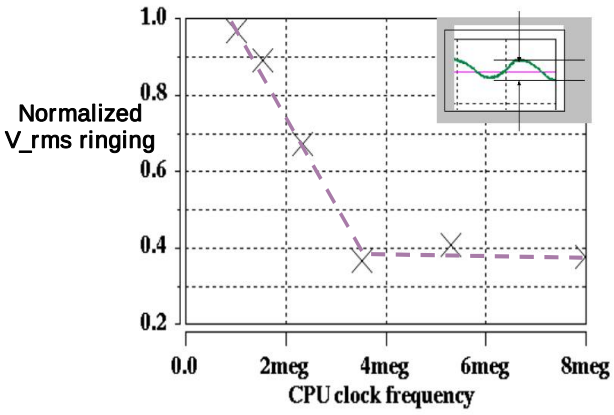


Figure 8 MCU Clock Speed Evaluation

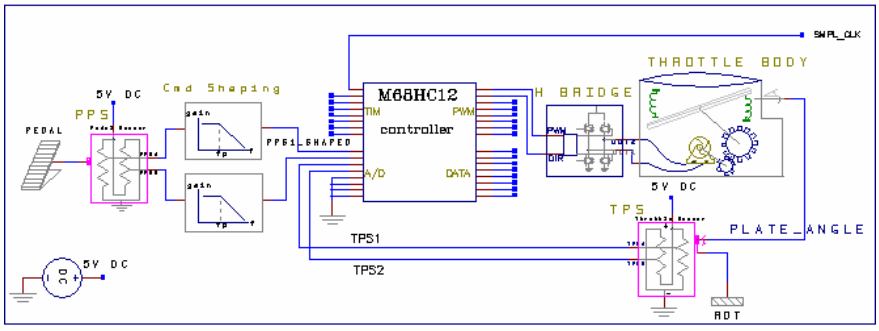


Figure 9 Electronic Throttle Schematic

Figure 9 shows the schematic of an electronic throttle controlled by an HC12 MCU. The step response of the

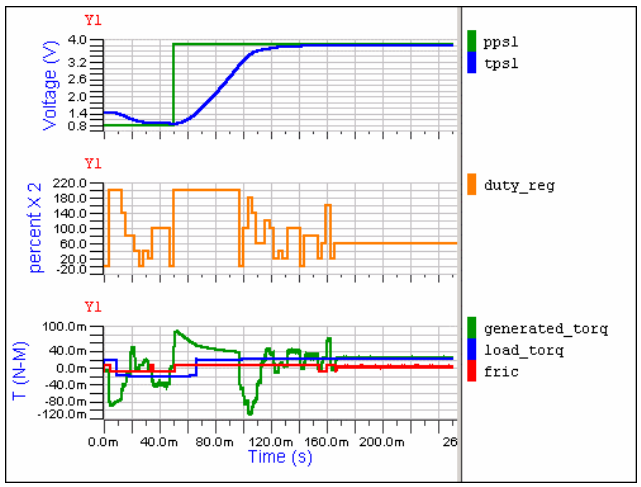


Figure 10 Throttle Waveforms

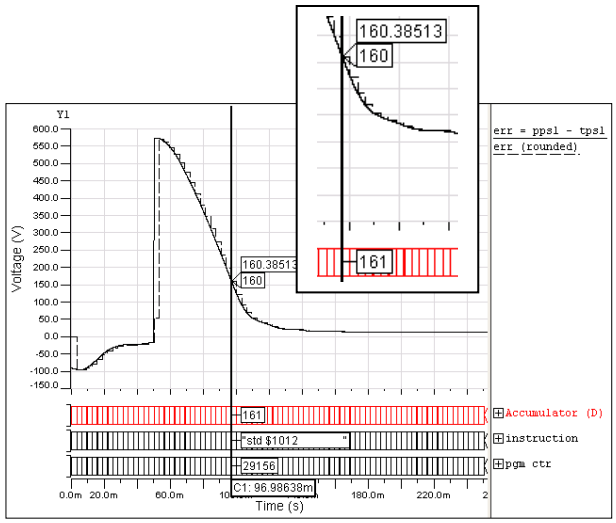


Figure 11 Debugging the Software

throttle for an embedded PID controller is plotted in Figure 10. Figure 11 shows an example in which the simulation is used to detect rounding errors due to the use of fixed point arithmetic. Embedded software relies on fixed-point computation to simultaneously reduce delays and program size.

6. CONCLUSIONS

This paper has introduced the ICSM approach to accurately model embedded software. The ICSM is an instruction set model of a micro-controller that eliminates non-critical of details required for cycle accurate modeling while retaining all of the details needed to simulate mixed-technology system behavior. The ICSM is coded in the native modeling language of the system simulator and synchronized to simulation time at the MCU instruction boundaries. Specific MCU models have been independently generated in MAST/C and in VHDL-AMS. Future development of the ICSM approach can take advantage of commonalities among MCU's to produce a generic template that can more readily be customized to specific processors.

REFERENCES

[1] Norman J. Elias, "Instruction Set Modeling Of Micro-Controllers For Power Converter Simulation," 2003 Applied Power Electronics Conference Proceedings, pp 996-1001, February 2003.

[2] Thomas R. Egel and Norman J. Elias "Using VHDL-AMS as a Unifying Technology for HW/SW Co-verification of Embedded Mechatronic Systems," 2004 SAE World Congress, paper no. 2004-01-0718, March 2004

[3] B. Bailey, R. Klein and S. Leef, "Hardware/Software Co-Simulation Strategies for the Future," System Design/Verification & Test Technical Publication, see http://www.mentor.com/soc/fulfillment/cosim_strategies_658.pdf

[4]Freescale Semiconductors, **HCS12 Microcontrollers, S12CPUV2Reference Manual,** S12CPUV2/D Rev. 0, 7/2003