IP/SOC 2005

Session : Specific Modeling

A STRATEGY FOR SPECIFYING SYSTEMC MICRO-CONTROLLERS MODELS USING THE ADL ARCHC[1]

Patricia Lira, Vítor Schwambach and Edna Barros, Centro de Informática – UFPE

Recife Brazil
(pfal, vsc, ensb@cin.ufpe.br)

Abstract

Electronic System Level (ESL) design is a highly effective approach for creating complex chips and systems. At this level engineers design and verify systems using abstract models, concentrating their efforts on systems architecture and algorithms rather than on low level RTL design implementations. Additionally, with advances in the semiconductor technology, it has been made affordable the implementation of digital systems as multiple processors SoCs (MPSoCs) or platforms. In order to speed-up an ESL design it is important to have processors models at distinct abstraction levels starting from functional and transaction levels to register transfer level. This paper presents an approach for specifying micro-controllers models using the ADL (Architecture Description Language) ArchC and SystemC. This approach has been used to specify an 8051 micro-controller in SystemC allowing the design of a very efficient model in few weeks.

1. INTRODUCTION

ESL design is a highly effective approach for creating complex chips and systems. ESL design has mainstreamed — it is now an established design methodology at most of the world's leading system-on-chip (SoC) design companies, and is being increasingly used in system design. Additionally, with advances in the semiconductor technology, it has been made affordable the implementation of digital systems as multiple processors SoCs (MPSoCs) or platforms.

Shared memory multi-processor systems-on-chip (MPSoCs) have been widely used in today's high performance embedded systems, such as network processors and parallel media processors (PMP). They combine the advantages of data processing parallelism of multi-processors (MP) and the high level integration of systems-on-chip (SoC) [1]. In order to speed-up an ESL design it is important to have processors models at distinct abstraction levels ranging from functional and transaction levels to register transfer level. This paper presents an approach for specifying micro-controllers models using ADLs and SystemC. This approach has been used to specify an 8051 micro-controller in SystemC allowing the design of a very efficient model in few weeks. The following section presents related works regarding ADLs and tools for SoC specifications. The approach for specifying micro-controllers models using ArchC is described in section 3, including a very briefly introduction into ArchC. The resulting specification of the 8051 micro-controller is described in section 4. Finally, section 5 presents some conclusions and future works.

2. RELATED WORK

In this section we briefly discuss some related works regarding specification of SoCs with ADLs. Most ADLs allow the description of processor architectures and its instruction set, but not all provide a mechanism to integrate the processor model with other peripheral components to create a SoC.

LISA[7][9] is a processor description language which allows for architecture exploration, implementation and system-level simulation. Its focus is the formal description of programmable architectures along with peripherals and also external interfaces.

[1] This work has been supported by CNPq (Brazilian funding agency) under projects Brazil-IP (grant nr. 552078-02/6) and ProPlat (grant nr 480733/04-0)

EXPRESSION[8] is an architecture description language that is suited for the description of different types of architectures: VLIW, ASIP, DSP and conventional processor architectures like RISC. Functional, cycle accurate and compiled simulation simulators can be generated from an EXPRESSION description.

ConvergenSC[10] is a tool for platform based design for generic application domains. It comes with a model library that includes ARM and MIPS processor models and also bus models like AMBA. For processors that are not in the library, designers must create them separately using the LISATek product family [9] and use them as components.

3. DEVELOPMENT METHODOLOGY

Most micro-controllers include a CPU, which is able to execute a defined instruction set, as well as peripherals such as timers, interrupt controllers, serial and parallel interfaces, etc. As mentioned, most ADLs lack the capability to specify peripherals. In this work we describe a strategy for obtaining SystemC specifications of micro-controllers using the ADL ArchC [2][3][5]. The main idea is to describe the instruction set using the ArchC ADL. From the ArchC description a SystemC description of the CPU is generated. This SystemC description is then modified in order to allow an easy integration of SystemC peripheral descriptions. Figure 1 shows the main steps of the proposed strategy.

The instruction set and architecture resources are described using ArchC, from this description, a SystemC description can be generated using the ArchC tool called *ac_sim*. The obtained processor simulator in SystemC is then modified for introducing a binding mechanism, which allows accessing peripheral registers through read and write memory operations. Additionally, an interrupt handling mechanism should be included for handling peripheral interruptions. Finally, the SystemC descriptions of all peripherals can be easily integrated resulting in a SystemC description of the micro-controller.

A more detailed description of how to create the processor model and then add the binding mechanism and interrupt handling capability to the model is described in the upcoming sub-sections.

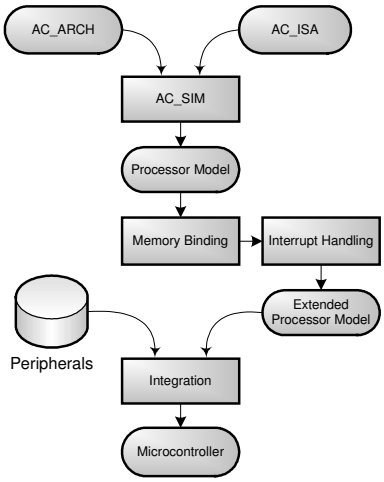

Figure 1 - Development Process

3.1 DESCRIBING THE PROCESSOR MODEL

As mentioned, ArchC is an architecture description language, initially conceived for architecture description, which aims to facilitate and accelerate processor description. In ArchC, a processor architecture description is divided in two parts: the Instruction Set Architecture (ac_isa) description and the Architecture Resource (ac_arch) description.

The ac_isa description includes details about instruction formats, size and names, as well as all necessary information to automatically generate an instruction decoder. The ac_arch description specifies the storage devices, pipeline stages and memory hierarchy of the processor. Figure 2 shows part of the *ac_arch* description for the 8051 micro-controller [4], which is an 8-bit CPU with a set of 255 instructions, 256 bytes of internal RAM (registers), 64Kbytes of external RAM and 64K bytes of ROM.

```
AC_ARCH(i8051_ca){
  ac_cache    IRAM:255;
  ac_cache    IRAMX:64K;
  ac_cache    IRAME:128;
  ac_icache   IROM:64k;

  ac_wordsize 8;

  ARCH_CTOR(i8051_ca) {
    ac_isa("i8051_ca_isa.ac");
  };
};
```

Figure 2 - ArchC architecture description

In this example four memories have been declared with the associated length and word size. The internal RAM includes a RAM with 256 bytes

representing registers, the extended RAM with 64 Kbytes, and a ROM memory with 64 Kbytes.

The ISA description is defined in two files, the first one containing the instructions and format declarations and the other one including instructions behavior.

Figure 3 shows the *ac_isa* description for the *add_ar* instruction, an instruction for adding a register to the accumulator. Each instruction declaration includes its format, its assembler syntax, the instruction identification and the number of cycles to execute it. The *ac_format* keyword defines a specific instruction format, which can be associated with more than one instruction. It allows the designer to access each instruction field individually. ArchC provides mechanisms to describe a common behavior of a set of instructions only once as the behavior of a instruction type. In the example, the format *Type_OP_R* is defined, which is associated with the instruction *add_ar* through the keyword *ac_instr*. Additionally, the *set_asm* specifies the assembler instruction syntax and operand encoding, *set_decoder* indicates the instruction opcode and *set_cycles* specifies the number of instruction cycles in a cycle accurate model

```
AC_ISA(i8051_ca){
  ac_format Type_OP_R = "%op1:5 %reg:3";
  ...
  ac_instr<Type_OP_R> add_ar;
  ...

  ISA_CTOR(i8051_ca){
    add_ar.set_asm("add A, %reg");
    add_ar.set_decoder(op1=0x05);
    add_ar.set_cycles(12);
    ...
  };
};
```

Figure 3 - ArchC instruction set architecture description

Figure 4 shows an example of an *ac_isa* description, which describes the behavior of the *add_ar* instruction. The switch statement is used to describe the instructions cycles. Each case represents one state of the CPU control unit when executing an *add_ar* instruction. In the second cycle, the accumulator value is read and in the following cycle the content of the register to be added is also read. In the cycle number 6 the sum takes place and finally in the cycle 12 the results are written back to the accumulator. The specification is the same as specified in the 8051 datasheet. The cycles omitted updated the flags register. A more detailed discussion on the ArchC architecture description language can be found in [5].

```
//Behavior of instruction
//ADD A, <register>
void ac_behavior(add_ar){
  switch(cycle) {
  ...
  case 2:
    //Reading accumulator
    acc = IRAM.read(ACC);
    break;
  case 3:
    //Reading value from register
    tmpA = IRAM.read(reg_indx);
    break;
  ...
  case 6:
    //Execute sum
    sum = acc + tmpA;
    break;
  ...
  case 12:
    //Store results in accumulator
    IRAM.write(ACC,sum.range(7,0));
    break;
  }
  //Advance one ArchC cycle
  ac_cycle++;
}
```

Figure 4 - ArchC description of ADD A, <register> instruction from 8051

3.2 BINDING CPU AND PERIPHERALS

After the CPU has been described in ArchC, the *ac_sim* tool generates a SystemC processor simulator, which is able to execute any instruction of code compiled for that processor.

In the case of micro-controllers peripheral models should be included in the processor model. The 8051 micro-controller, for example, includes as peripherals timers, serial interface, interrupt handler, etc. Most micro-controllers implement communication to and from peripherals through read and write operations CPU registers. These registers store data (from and to) peripherals as well as the peripherals status.

Using ArchC the designer is able to specify registers in the CPU, but a problem arises when registers assigned to peripherals must be accessed by the CPU and by the peripheral as well. How data consistency can be guaranteed when accessing such registers?

This work proposes a technique to cope with this problem by ensuring consistency when accessing peripheral registers. Only a single exemplar of peripheral registers is defined, which is located at the peripheral side. Whenever the CPU does a read or write operation in such registers, the registers accessed are the ones on the peripheral side.

ArchC implements all storage elements (registers, memory, caches) using the *ac_storage* class, which includes the methods for reading and writing. For implementing the proposed strategy, a binding mechanism has been developed. This mechanism performs a mapping among registers addresses in the CPU and the corresponding peripheral register.

In order to implement this mapping mechanism, the read and write methods have been changed to deal with the address bindings. For every call to a read or write method in the memory it is first checked if a bound address is being accessed. If the given address is bound to a peripheral then the pointer to that peripheral is retrieved and the method call is redirected to that pointer, otherwise the method executes normally. This causes the processor model to access the appropriate registers in case the registers are located in the peripherals. For redirecting read and write calls to peripherals, the behavioral description of peripherals must implement the *ac_storage* interface. This interface comprises all read and write methods available in the *ac_storage* class, so by implementing this interface the peripheral itself becomes a memory element, making it possible to redirect the read and write calls.

Figure 5 shows the binding of registers in the processor model and the corresponding registers in the peripherals. This way, the processor model can access registers assigned to peripherals in a transparent way, and no further modifications are necessary.

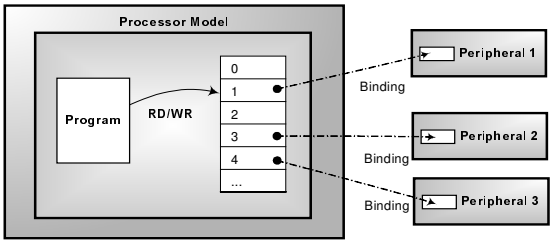

Figure 5 - Peripheral Mapping

3.3 INTERRUPT HANDLING

Some peripherals of a micro-controller need to interrupt the CPU in order to have some kind of data processing or simply to activate another peripheral. To implement this kind of functionality, some mechanism to interrupt the CPU must be provided. The CPU behavior when an interruption occurs must also be specified. To support this feature, an interrupt handler has been developed. decoding of each instruction has been specified.

A device can generate a new interruption by invoking the *requestTrap* method (shown in Figure 6) with a device identifier given as argument. Whenever the interrupt handler receives an interruption request it puts this new interruption on a priority queue. Before decoding a new instruction, the instruction decoder checks if there are any interruptions to be served by calling a method of the interrupt handler, the *getPendingRequests*, which returns the interruption to be served, if any. If there are none, the execution resumes. In the case an interruption is returned, the instruction decoder executes the behavior described in the interrupt handler for that particular interruption.

```
//Called by a device to generate an interruption
void requestTrap(int trap_id);

//Called by the interrupt handler to deactivate
//an interruption
void disableTrap(int trap_id);

//Interrupt Controller function to check if a
//a given interruption is pending.
bool isFree(int trap_id);

//Get pending requests
//Checks if there is an interruption to be
//executed and returns the corresponding
//identifier of the next interrupt to be
//executed. If there are no pending interrupts
//returns -1.
int getPendingRequest();

//Called by the CPU to indicate the interrupt
//handler that the current interruption has
//been treated (RETI instruction executed).
void trapDone();
```

Figure 6 - Interrupt Handler interface

The instruction decoder executes the interruption service routine, which behavior is described in the interrupt handler. As it can be seen in Figure 7, the current value of the program counter is pushed onto the stack, in order to allow the return to the normal execution flow, once the interrupt service routine has finished.

```
//OCP-IP external interrupt behavior
void ac_behavior(EX0){
  sc_uint<9> aux;

  //Stores current program counter on stack
  aux =  IRAM.read(SP_SFRADDRESS) + 1;
  IRAM.write(aux.range(7,0),pc.range(7,0));
  IRAM.write(SP_SFRADDRESS,aux.range(7,0));
  aux = IRAM.read(SP_SFRADDRESS)+1;
  IRAM.write(aux.range(7,0),pc.range(15,8));
  IRAM.write(SP_SFRADDRESS,aux.range(7,0));

  //Branches to interrupt service routine
  pc.range(7,0) = 0x03;
  pc.range(15,8) = 0x00;
  ac_pc = pc;
}
```

Figure 7 - OCP-IP external interface interruption behavior

Whenever the instruction decoder identifies the instruction of interrupt return, it must inform this event to the interrupt handler, invoking the *trapDone* method shown in
Figure 6, so that it can remove that particular interruption from the queue.

Furthermore, peripherals that can interrupt the processor must receive a pointer to the interrupt handler in their constructor. Thus, when the peripheral wants to interrupt the processor, it calls a method of the interrupt handler to request the interruption. The interrupt handler will then process the request.

3.4 PERIPHERAL INTEGRATION

Once the SystemC processor model has been obtained from an ArchC description and has been modified to include a binding mechanism to peripherals registers and an interruption handler, a complete model of the micro-controller can be obtained by declaring and instantiating each individual peripheral specification.

Figure 8 shows how to integrate peripherals in the SystemC processor description[2]. In this case, the i8051.h file has been modified for integrating an OCP-IP [6] interface peripheral. An advantage of having an 8051 IP-core with OCP interface is the fact that this IP can be connected to any OCP complaint IP-core.

```
//Architecture Module header file
class i8051_ca_arch: public sc_module,
                     public ac_resources {
public:
   ...
 //OCP-IP external interface declaration
 i8051_ocp *ocp;
   ...
 //Constructor
 SC_CTOR(i8051_ca_arch) {
   ...
  //OCP-IP external interface instantiation
  ocp = new i8051_ocp("i8051_ocp",trap_controller);
  //Memory binding of OCP-IP peripheral
  IRAMX.bindAddStorage(OCP_FLAG_REGISTER_ADDR,ocp);
  IRAMX.bindAddStorage(OCP_SLAVE_ADDR,ocp);
  IRAMX.bindAddStorage(OCP_MASTER_ADDR,ocp);
    ...
 }
};
```

Figure 8 - OCP-IP peripheral integration

The integration specification includes peripheral declaration followed by its instantiation. Notice that a pointer to the interrupt handler called *trap _controller* is passed as argument to the constructor of the OCP-IP interface peripheral. Through this pointer the OCP-IP interface can access the interrupt handler and is able to generate an interruption to the processor as explained previously. The code that follows the instantiation binds some addresses in the external memory of 8051 (IRAMX) to the OCP-IP interface peripheral.

The method *bindAddStorage* has been proposed to perform the address binding. The first argument of this method is the memory address, which is going to be bound to the peripheral[3], and the second argument is a pointer to the peripheral itself. Each time this method is invoked, a entry is created in an internal addresses table, which relates the IRAMX address and the OCP-IP interface peripheral. From this point on, whenever read or write methods of IRAMX are called with an address that has been bound to the OCP-IP interface it will relay the call to the OCP-IP peripheral. Thus, the processor-peripheral communication is established.

After all peripherals have been integrated to the processor model and their interruption priorities have been established, the next step is to update the interrupt handler. It must be adapted to correctly handle the peripherals interruptions considering their respective priorities.

4. RESULTS

The proposed strategy has been used to design a cycle accurate model of an 8051 micro-controller, which includes a serial interface with 4 operating modes, 2 timers, 4 parallel input/output ports, 256 bytes of internal RAM (registers), an interrupt handler with four levels of priority, as well as an OCP-IP compliant external interface. The number of SystemC code lines for the peripherals is summarized in Table 1.

The processor has been described at transaction level but with accuracy of cycle. The table also displays the number of SystemC code lines for a SystemC RTL description of the same processor. The efficiency of the obtained SystemC model at transaction level has been evaluated by running the DALTON benchmark [4], a set of programs to test the instruction set. A set of programs to validate the peripherals has also been executed. The simulation time is depicted in Table 2. The benchmarks have been executed using a computer with a Pentium 4 CPU running at 1.8 GHz with 512 MB of RAM. The reader can see that the TL cycle accurate model runs approximately 29 times faster than the RTL description in average.

[2] Obtained after processing of ArchC files by the tool ac_sim

[3] We have used pre-defined constants for each address

Table 1. Comparison of SystemC code lines

	RTL*	TL*	RTL/TL
Processor	21097	11080	1.90
OCP	2595	513	5.06
Port	1650	245	6.73
Timer	1169	581	2.01
USART	2037	477	4.27
Total	28548	12896	2.21

*Unit: Code lines.

Table 2. Comparison of simulation time

	RTL*	TL*	RTL/TL
Divmul	34.98	1.16	30.15
Fib	36.86	1.24	29.75
Int2bin	9.27	0.49	18.91
Sort	47.39	1.43	33.14
OCP	72.75	4.93	14.75
Timer	137.57	7.27	18.92
USART	200.34	3.56	56.27

*Unit: Seconds.

5. CONCLUSION AND FUTURE WORKS

In this paper a strategy for obtaining a micro-controllers model at cycle-accurate transaction level has been proposed. The instruction set is described initially by using the ADL ArchC, which provides a mechanism for describing the syntax and semantic of processor instructions at a very high abstraction level. Therefore, ArchC lacks a mechanism for describing peripheral behavior, included in the most micro-controllers. For describing peripherals, SystemC has been used. The integration of the generated SystemC description of the CPU with the peripherals description has been achieved by introducing a binding mechanism, which allows the mapping of the CPU registers into registers in the peripherals. With this technique, the CPU can access peripheral registers in a transparent way. A mechanism for interrupt handling has also been developed. The occurrence of an interruption is tested after instruction decode. The interruption handler includes the interruption behavior, as well as the interruption priority for the peripherals. A SystemC transaction level cycle accurate description of an 8051 micro-controller has been obtained according to this strategy. The obtained model has less code lines and is 29 times faster then the corresponding RTL SystemC description. The obtained TL cycle accurate model can be used at an early design phase, reducing the design time and allowing a best quality design.

As a future work we can mention the integration of the proposed technique in the ArchC tools, in order to generate a processor model including facilities for peripheral integration. A mechanism for interruption handling is already available in the ArchC language, but must be adapted to allow the peripheral interruptions. Furthermore, mechanisms for facilitating the peripherals integration are under development. This way the micro-controller model could be generated without ever modifying the ArchC generated code, thus reducing even further the effort and time required to model micro-controllers.

7. REFERENCES

[1] Chang, H., et al.: Surviving the SoC revolution: a guide to platform-based design. Kluwer Academic Publishers, Massachusetts (1999)

[2] Rigo, S., Azevedo, R., Araújo, G., Araújo, C. and Barros, E.: The ArchC Architecture Description Language and Tools. In International Journal of Parallel Programming. Kluwer Academic Publishers, 2005, Pages: 453 - 484

[3] Viana, P., Barros, E., Rigo, S., Azevedo, R., and Araújo, G.: "Modeling and Simulating Memory Hierarchies in a Platform-Based Design Methodology". In Proc. of the DATE, 2004

[4] http://www.cs.ucr.edu/~dalton/i8051/i8051syn/

[5] http://www.archc.org

[6] OCP-IP: Open Core Protocol International Partnership, Available at: http://www.ocpip

[7] http://servus.ert.rwth-aachen.de/lisa

[8] Halambi A., Grun P., Ganesh V., Khare A., Dutt N., and Nicolau A.: Expression - a language for architecture exploration through compiler simulator retargetability. In Proceedings of DATE, 199, ACM Press, page 100.

[9] Homann A., Kogel T., Noah A., Braun G., Schliebush O., Wahlen O., Wieferink A., and Meyer H.: A novel methodology for the design of application specific instruction set processors (ASIP) using a machine description language. In IEEE Transactions on Computer-Aided-Design, pages 1338-1354, November 2001.

[10] Coware company. Available at http://www.coware.com, Nov 2005.