Volume 3, Issue 11, November 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Formal Processor Description for OS Modules Generation

Alexander Yu. Drozdov, Yuriy N. Fonin, Victor E. Vladislavlev
Faculty of Radio Engineering and Cybernetics,
MIPT, Russia

Abstract – This article provides a new high-level formal description of the processor cores architecture. Possibilities to use the description for generating the system software are shown on the example of automatic generation of the OS microkernel context-switching module.

Keywords – architecture, formal description, operating system, microkernel, ADL.

I. Introduction

The complexity of new microprocessor cores and micro architecture development is determined not only and not so much by the complexity of hardware development itself, but by the complexity of the necessary software component development as well. This equally applies to the creation of new architectures "from scratch", as well as to enhancement and development of the existing architectures. Changes in the processor instruction set cause the necessity to modify at least assembler, disassembler and debugger, and in some cases compiler and linker. If, in addition to new instructions, the register file expansion or new flag or interrupt adding is required, then it will be necessary to change the operating system microkernel. The ever-increasing complexity of created chip opens new opportunities. This, in turn, dictates the need to find new alternative approaches to microprocessor core designing, which would obviate the problems described above.

One of the alternatives is using high-level Architecture Description Languages (ADL). In addition to standard hardware description languages, such as VHDL or Verilog, the unified architecture description in ADL makes it possible to automatically generate system software, for example assembler, linker, compiler or a set of tests. By using this approach, one can make all the changes in a single specification, and all the components are generated automatically, which significantly reduces the cost of architecture changes. However, it should be noted that modern high-level architecture description languages are suitable only for microprocessor cores and co-processors development that are modules which require software presence. To design and simulate cache memory communication devices (e.g., USB-controller) as well as hardware accelerators implementing the required functionality (e.g., Fourier transform) it is advisable to use classic architecture description languages – VHDL or Verilog. SystemC – C++ class library is also actively used for model development of digital integrated circuits.

There are three types of high-level ADLs: structural, behavioral and mixed. Structural ADLs, such as MIMOLA [1], are similar to traditional hardware description languages, such as VHDL or Verilog, and are used mainly for hardware synthesis out of ADL description. However, structured ADLs include constructs that make it possible to specify additional element properties. For instance, MIMOLA includes language constructs for program counter or program memory bus definition. Behavioral ADLs, such as NML [2] or Sim-NML [3], are mainly used for system software modeling and development. Each command functionality in behavioral ADLs is described in an explicit form, which greatly simplifies system software generation algorithms. Mixed ADLs, such as LISA [4] or EXPRESSION [5], are similar to behavioral languages. However, these languages include constructs that make it possible to generate descriptions suitable for hardware synthesis out of processor description. In mixed languages, as well as in behavioral ones, instructions are described in an explicit form, which makes it possible, in addition to hardware generation, to automatically generate system software: assembler, compiler and debugger.

Each ADL has its own Intermediate Representation (IR) being used for subsequent generation of the required components. Theoretically, this IR can be used to develop tools and utilities that analyze the architecture and automatically generate different hardware or software components. But in practice, most ADL users have no access to IRs and have no opportunities to create their own extensions for architecture analysis or to generate other software or hardware components.

This article provides a formal architecture description. On the one hand, this description can be derived from processor ADL description translation or created on the basis of the basic ADL IR. On the other hand, this description enables to implement user-defined algorithms of architecture analysis and component generation, such as compiler, operating system modules or sets of tests. The algorithm of automatic generation of the OS microkernel context-switching module is described as an example.

This work was supported by the Ministry of Education and Science of Russian Federation under contract No02.G25.31.0061 12/02/2013 (Government Regulation № 218 from 09/04/2010)

II. Basic Elements Used for Processor Description

Core description can be divided into two components: structural and behavioral. Structural core description implies a set of elements which define the processor state, such as registers, flags, input-output ports, interrupt signals, etc. The behavioral description component is defined by a set of functions performed by the processor: instructions, hardware interrupt handlers and other actions, such as internal counters changing.

A. Sets

The set theory is suitable to describe and analyze the architecture. Members of sets will be the constituent processor parts: registers, flags, ports, etc. Members of set can also be micro-operations (see below), expressions or constants. An index is assigned for every member of such a set $G = \{g_0, g_1, g_2, ...g_{N-1}\}$.

For each set we shall introduce function $G^{-1}(g) = \begin{Bmatrix} idx, g \in G \\ -1, g \notin G \end{Bmatrix}$, which expresses:

- member index (0 – is also an index) if the element belongs to the set, and
- -1, if the element does not belong to the set.

To work with subsets we shall use a standard set of operations:

1) Intersection: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$;
2) Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$;
3) Complement: $A \setminus B = \{x \mid x \in A \text{ and } x \notin B\}$.

B. Micro-operations

Basic elements for the behavioral microprocessor model description are micro-operations. In general, the processor functionality can be described only by means of Boolean algebra operations and access operations to registers, memory, ports, etc. However, to provide more efficient architecture analysis and, in particular, OS component generation it is reasonable to use mathematical and shift operations.

The list of micro-operations for the behavior microprocessor model description is as follows:

1) Resource access operations:
   a. MEM(mem_space, addr) means memory access, where mem_space is a memory port index, addr is an expression that specifies the access address;
   b. REG(reg_id) means register access, where reg_id is a register index in the list of processor registers;
   c. FLAG(flag_id) means flag access;
   d. PORT(port_id) means input/output port access;
   e. SET(set_id, exp) means accessing an element of the processor resource set by index, where index set_id is a set index in the list of processor sets, exp is an expression that specifies a set member index;
   f. ARG(arg_num) means instruction argument numbered arg_num.
2) Arithmetical operations:
   a. ADD(exp1, exp2) means sum of exp1 and exp2;
   b. SUB(exp1, exp2) means subtracting of exp1 and exp2;
   c. MUL(exp1, exp2) means product of exp1 and exp2;
   d. DIV(exp1, exp2) means division of exp1 by exp2;
   e. NEG(exp1) means multiplying exp1 by -1.
3) Logical operations:
   a. AND(exp1, exp2) means logical “AND”;
   b. OR(exp1, exp2) means logical “OR”;
   c. XOR(exp1, exp2) means “exclusive OR”;
   d. NOT(exp1) means inversion.
4) Shift operations:
   a. LSL(exp1, exp2) means logical left shift, exp1 value is shifted to the left for exp2 bits;
   b. LSR(exp1, exp2) means logical right shift, exp1 value is shifted to the right for exp2 bits;
   c. ASL(exp1, exp2) means arithmetical left shift, exp1 value is shifted to the left for exp2 bits;
   d. ASR(exp1, exp2) means arithmetical right shift, exp1 value is shifted to the right for exp2 bits;
5) BITS means access to individual bits of the resource, for example:
   a. BITS((2:5), REG(r1)) – accessing from the second to the fifth bits of the r1 register;
   b. BITS((1:17), ADD(REG(r1), REG(r2))) – taking from the 1st to the 17th bits of the sum of r1 and r2 registers.
6) Assignment operations:
   a. ASG(res, exp) means assign of exp value to the register, flag, location or port.
7) Control transfer operations:
   a. JZ (cond, sub_tree) means performance of expressions subset sub_tree, if cond is equal to zero;
   b. JNZ (cond, sub_tree) means performance of expressions subset sub_tree, if cond is nonzero.
8) Software interrupt call operation IRQ(irq_num), where irq_num is an interrupt number.

C. Functions

Function is represented by a graph which nodes are expressions. Expressions, in turn, are described as a tree whose nodes are micro-operations. The root of the tree can be either assignment operator (ASG), or JZ or JNZ operator.

III. Processor Core Description

A. Processor Structure

Processor structure is described by the following set of elements:

1) A set of registers;
2) A set of flags;
3) A set of input/output ports;
4) A set of memory ports;
5) Interrupt subsystem.

B. Registers

Registers or register arrays have the following set of properties:

1) Register size in bits;
2) Register name in assembly language – optional.

Registers can be united into sets. A set of registers can be used as a micro-operation argument.

C. Flags

Flags are described by the following set of properties:

1) Flag container – register, port or interrupt signal which contains the flag;
2) bit number in the container;
3) flag name in assembly language – optional.

D. Input/Output Ports

Input/output ports are described by the following set of properties:

1) Port type – input, output or bidirectional;
2) Port size in bits;
3) Port name in assembly language – optional.

Input port allows only reading port status. Output port allows only writing data in the port, while read operation is not allowed. Bidirectional port allows both reading of the port status and writing to the port. PORT (port_obj) micro-operation as well as SET (ports_set, index_exp) micro-operation, where ports_set is a set of input-output ports, is used to access an input/output port.

E. Memory Ports

Memory port is described by the following set of properties:

1) Address bus size in bits;
2) Data bus size in bits.

F. Interrupt Subsystem

Processor interrupt subsystem is described by the following set of properties:

1) Number of interrupts;
2) List of interrupt priority;
3) List of interrupt types;
4) List of interrupt handlers (one handler can belong to several interrupts);
5) Size of external interrupt vector port in bits – optional.

Interrupt priority list elements can be constants, registers or memory cells. In the former case it refers to the fixed-priority system, in the latter case interrupt priorities can be programmed.

Let us consider the following set of types as an example:

1) DEV means external interrupt generated by the peripheral device;
2) CNTR means interrupt generated by the external interrupt controller. It is expected that an interrupt vector value is set on external interrupt vector port along with the interrupt signal;
3) SWI means software interrupt. This interrupt is generated inside the special instruction and is described by IRQ micro-operation;
4) ERR means interruption which occurred in case of error while executing the instruction.

IV. Behavioral Model of Processor

In our situation the behavioral model of processor is described by a set of its instructions and by the interrupt hardware processing functions.

A. Set of Instructions

Each instruction is described by the following set of properties:

1) Instruction mnemonics in assembly language;
2) List of arguments;
3) Function (set of functions) describing instruction semantics.

Instruction argument is described by the following set of properties:

1) Argument type – signed integer, unsigned integer or floating-point number;
2) Argument size in bits;
3) Argument location in the instruction code – number of the first argument bit;
4) Set of admissible values of the instruction – optional.

A set of admissible values of the argument can be defined by a set of constants or by a set of processor elements, such as registers, flags, or input-output ports. In the latter case, all processor elements belonging to the set of admissible values of the argument should have assembler names.

Most modern processors have groups of instructions with identical sets of arguments and with identical sets of admissible values of these arguments. Therefore, while describing the CPU it is advisable to have a separate list of sets of admissible values, and while describing an argument it is advisable to specify a set index according to this list.

Let us consider the description of several ARM7 [7] core architecture instructions:

1) ADDRd, Rn #<imm12> means addition of Rn register value to imm12 12-bit unsigned constant, the result is saved in Rd;
2) ADDRd, Rn, Rm means addition of Rn register value to Rm register value, the result is saved in Rd;
3) SUBRd, Rn #<imm12> means subtracting Rn register value and imm12 12-bit unsigned constant, the result is saved in Rd;
4) SUBRd, Rn, Rm means subtracting Rn register value and Rm register value, the result is saved in Rd.

It is possible to set registers out of R0, R1, R2, …, R15 list as Rd, Rn, Rm arguments for all instructions, e.g.:

- ADDR0, R1, R3 // saving sum of R1 and R3 in R0 register;
- SUBR4, R2, #20 // saving subtracting R2 and 20 in R4 register.

A set of R registers {R0, R1, R2, ... R15} will be created to describe this set of instructions. “Set of admissible value” argument property value for Rn, Rd, Rm arguments of ADD and SUM instructions will be equal to R set index from the list of all processor sets.

Instruction behavior is described by one or several functions. Most modern processors have pipelined architecture. Instruction execution in such processors can be divided according to pipeline stages. In this case, the instruction behavior at each pipeline stage shall be described by an individual function. On the other hand, if the information about instruction dissection according to pipeline stages is of insignificant importance for architecture analysis, availability of one function describing instruction semantics is sufficient, which is obviously easier to analyze.

B. Hardware Interrupt Handlers

Hardware interrupt handler is described by a function which is executed when a request signal for the interrupt with the highest priority is received. As a rule, the handler saves the current program counter value in the memory or in a special register and after that saves the interrupt handler vector value to the program counter. In addition, the handler can execute other operations, such as storing certain registers in the memory, changing flag values, etc. The analysis of hardware interrupt handlers makes it possible to automatically generate interrupt subsystem initialization code as well as the correct software interrupt handler code.

V. Description of Context-Switching Module Generation

The following example describes the algorithm for automatic generation of context-switching procedures. According to the definition, «a context switch is the process for storing and restoring the state (context) of a process so that execution can be resumed from the same point at a later time» [6].

In general, the context-switching algorithm may be divided into three steps:

1) saving the context in the memory;
2) Selection of the next process and initializing of the context pointer for the selected process;
3) Restoring previously saved context from the memory.

A. Context Definition

A formal definition of the context is necessary to generate algorithms storing and restoring the context. In most architectures context is a set of registers which are used when performing applications.

On the basis of different architecture context analysis it was determined that in most cases the context can be defined as a set of registers that:

1) Are the arguments of arithmetic, logic or shift instructions;
2) Are involved in address calculation when accessing memory;
3) Contain flags which save attributes of arithmetic operations (such as “overflow”, “vanishing”, etc.);
4) Are used to save the return address in subroutine call instructions.

Furthermore, in some cases, so-called unnamed registers which are not named in the assembler language can be excluded from the set of registers, satisfying the conditions mentioned above. In general, these registers contain only intermediary instructions results.

Following is an algorithm for context definition, see Fig. 1.

We shall define the following set of helping functions to describe the algorithm:

- Mop(exp) – function returns micro-operation type;
- FirstArg(exp) – function returns the first argument of exp operation;
- SecondArg(exp) – function returns the second argument of exp operation;
- GetSet(exp) – for micro-operations SET function returns a set of elements, belonging to the set, which index is determined in the first argument of SET;
- GetInstExpList(inst) – function returns a list of expressions, compounding the function;
- SelectRegs(Set) – function forms a set of registers out of set of random processor elements.

In addition, let us determine Oplist as a set containing all arithmetical, logical and shift operations:
OpList = {ADD, SUB, MUL, DIV, AND, OR, XOR, NOT, LSL, RSR, ASL, ASR}.
The result of the algorithm will be Context set containing all registers that make up the process context.

```
Context := Ø
for each inst ϵ Instructions
    Cont := Cont U FindInstArgs(GetExpList(inst))
end for

// pseudo-code for FindInstArgs
for each exp ϵ InstExpressionList(inst)
    if Mop(exp) = ASG then
        Cont := Cont U FindInstArgs(exp)
    else if Mop(exp) = JZ and Mop(exp) = JNZ then
        // for branch operations it is necessary to analyze micro-operation sequence
        // which is referred by the second argument of the micro-operation
        Cont := Cont U FindInstArgs(SecondArg(inst))
end for

// pseudo-code for FindArgs(exp) which analyzes micro-operation arguments
LocalContext := Ø
if Mop(exp) ϵ OpList then
    LocalContext = LocalContext U FindArgs(FirstArg(exp))
if Mop(exp) ≠ NEG and Mop(exp) ≠ NOT then
    LocalContext = LocalContext UFindArgs(SecondArg(exp))
else if OP(exp) = REG then
    //  register accessing; adding  the register to the context
    LocalContext := LocalContext U GetReg(exp)
else if OP(exp) = SET then
    // micro-operation is a set; adding all the registers belonging to the set in the context
    LocalContext := LocalContext U SelectRegs(GetSet(exp))
return LocalContext
```

Fig. 1 Pseudo-code of the algorithm forming Context set.

B. Context Pointer Definition

Context pointer is a register which contains the address according to which the context should be saved in the memory or read from the memory. As a rule, a context pointer is determined by hardware or compiler developers. For the most architectures, a stack pointer poses a context pointer.

C. Context-Switching Module Generation

When the context is formed, it is necessary to generate code for saving it to the memory and restoring it from the memory. To generate a subroutine storing the context it is necessary to form the following expression for each register belonging to the context:

ASG(MEM(data_bus, ADD(context_ptr, offset)), REG(Cont, i)), where:

- data_bus means processor data bus;
- context_ptr means register, context pointer;
- offset means address offset relative to the beginning of the memory allocated to save the context. ADD (context_ptr, offset) expression defines the address according to which the i-th register will be saved;
- i – means current register index.

Similarly, a subroutine for context restoring can be defined in a sequence of the form:
ASG(REG(Cont, i), MEM(data_bus, ADD(context_ptr, offset)))
Generated microcode is translated into a sequence of instructions. Generation algorithm is similar to code-generating algorithms in high-level language compilers.
Issues of automatic code generation for switching to another process are associated not only with hardware platform features but also with operating system specifics. Taking notice of OS specifics it is possible to form a code which saves and initializes context pointer out of micro-operations.

VI. Conclusion

This article proposes the method of formal microprocessor architecture description, which makes it possible to develop algorithms for automatic architecture analysis and system software generation. The example of automatically generating algorithm of functions saving and restoring process context shows the way how one can use the formal architecture description. Further, it is intended to use formal description for automatic generation of other platform-specific OS components, for example, generation of interrupt handlers prologue or epilogue. In addition, it is intended to

explore the possibility of using the provided formal description to verify microprocessor cores and to automatically generate software development tools for new processors.

References

[1]. "The integrated design of computer system with MIMOLA" Peter Marwedel, University of Kiel.
[2]. "What is nML?" http://www.retarget.com/products/whatisnml.php
[3]. "High level Synthesis from Sim-nML Processor Models" Souvik Basu, Rajat Moona.http://www.cse.iitk.ac.in/users/moona/papers/basu.pdf/
[4]. LISA – Language for Instruction Set Architecture - http://en.wikipedia.org/wiki/LISA_(Language_for_Instruction_Set_Architecture)
[5]. EXPRESSION web page - http://www.ics.uci.edu/~express/
[6]. http://en.wikipedia.org/wiki/Context_switch
[7]. http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf

