2015 The 5th International Workshop on Computer Science and Engineering
doi: 10.18178/wcse.2015.04.024

The Computer Architecture and Hardware Description Language

Aleksey S. Burlakov, Andrey A. Mikhailov

Institute for System Dynamics and Control Theory of Siberian Branch of Russian Academy of Science, Irkutsk, 664033, Russia

Keywords: programming language, parsing, interpreter, computer emulator, computer architecture.

Abstract. The paper introduces CAHDL, a Computer Architecture and Hardware Description Language. The CAHDL is aimed to specify processor instructions and hardware formally. The CAHDL can be used both in documentation and in programs which require information about computer architecture such as emulators, compilers, decompilers, etc. The article considers some syntactic constructions in examples and explains their semantics. The specification parsing methods are also briefly reviewed. The end of the paper observes the software developed to test the language and to debug specifications written in this language.

Introduction

There are many manuals and books describing processor architectures and computer hardware. Intel® 64 and IA-32 Architectures Software Developer's Manual [1] consists of 3439 pages. There is a special site devoted to ARM architecture [2]. Many textbooks and reference books for assembly languages describe particular processor architectures. The software like compilers, disassemblers, computer emulators and device drivers use information about computer architecture.

The paper describes Computer Architecture and Hardware Description Language which could be uses both for writing documentation and for a formal representation of computer architecture in computer programs. Assemblers and disassemblers may use instruction specification for coding and decoding. Computer emulators may also use the information about processor architecture and peripheral devices.

Unlike common programming languages, CAHDL is designed especially to specify computer architecture and peripheral devices. Although CAHDL supports RTL modelling to some extent, it is not aimed to model integrated circuits like Verilog [3] or VHDL [4] does. The main purpose of the language is to provide a high level specification of a computer architecture which could be used in software. Thus, CAHDL operates the high level constructions like processor, memory, keyboard, etc.

CAHDL is an object oriented imperative language with declarative constructions. The syntax of CAHDL bases on C++ ISO/IEC programming language. There are also some syntax constructions taken from Verilog and from technical literature [5-8]. The source [9] was used in the development of the syntax of description of arguments of processor instructions.

Besides the language itself, the paper observes the software especially developed both to test the language and to debug architecture specifications written in CAHLD. This software is an emulator. It interprets CAHDL instructions and emulates the architecture described by them.

The Syntax of CAHDL

CAHDL is an object oriented programming language. It supports encapsulation, polymorphism and single inheritance. The base CAHDL structure is a class. CAHDL program operates different objects such as processor, memory, keyboard, etc. The syntax of CAHDL bases on that of C++ and it has bit manipulation constructions similar to the ones used in Verilog.

Besides from common control flow constructions such as loops, conditional expressions, labels, etc., CAHDL implements declarative structures aimed to specify processor instructions and tabular information in general. It also implements operators both for extracting a bit sequence from a variable and for concatenating variables. CAHDL is an interpreted programming language. It

implements references which may refer not only to a static memory address (like a C++ reference), but also to dynamic constructions like l-value expressions [10].

One of the possible terminal symbols corresponding to an l-value expression is a variable, which might be either scalar variable or an object property. Another possible terminal symbol of an l-value expression is a function, which returns reference. Validation of l-value expressions is carried out during and after parsing of specification. Nonterminal symbols of an l-value expression are operators of bits extraction or variables concatenation.

```
lvalue_expr: nested_id          // Var. or prop.
  | nested_id '(' expr_list ')'  // Method
  // single bit extraction
  | lvalue_expr '<' expr_num '>'
  // multiple bits extraction
  | lvalue_expr '<' expr_num PERIOD expr_num '>'
  | lvalue_expr ':' lvalue_expr  // Concat
;
```

Fig. 1: Syntax of l-value expression in Backus-Naur Form

Verilog form of bits-extraction operator with square brackets is not used because square brackets are reserved for array elements. At the same time colon is reserved for concatenation, thus a period is used as a delimiter in the bits extraction operator. As for the rest, concatenation and bits extraction operators are used exactly as in Verilog. The following example demonstractes the use of l-value expression with the bits concatenation:

```
bits<32> A = 0xffffffff;
bits<1> CF = 0;
CF:A<15..0> = 0x1cdcd; // CF = 1, A = 0xffffcdcd
```

Here A and CF are variables of 32 and 1 bit length respectively. In the third line, A will change its first 16 bits to 0xcdcd and the 17-th bit of 0x1cdcd will be stored in CF. Here is another example with a function used as an l-value:

```
bits<?>& R(bits<3> code, bits<1> W = 1)
{
  int len = W ? 31 : 15;
  case(code){
    maskb    0: 0: 0  then  return EAX<len..0>;
    maskb    0: 0: 1  then  return ECX<len..0>;
    maskb    0: 1: 0  then  return EDX<len..0>;
    maskb    0: 1: 1  then  return EBX<len..0>;
    maskb    1: 0: 0  then  return ESP<len..0>;
    maskb    1: 0: 1  then  return EBP<len..0>;
    maskb    1: 1: 0  then  return ESI<len..0>;
    maskb    1: 1: 1  then  return EDI<len..0>;
  }
};
void MOV(bits<?>& RegD, bits<?>& RegS )
{
      RegD = RegS;
      ...
}
...
```

```
MOV(R(0), R(1));
```

The function R returns a reference to one of the general registers depending on the code value. The reference is similar to wire type used in Verilog. If W flag is set, then an extended (32 bits) register reference will be returned, otherwise a 16 bits register reference will be returned. The length of the reference returned is unknown in advance; hence we use a question mark as an argument for bits type. In fact the question mark is syntactic sugar and can be omitted with the angled brackets, thus bits<?> is the same as bits. If we execute the MOV function with R(0) and R(1) passed as arguments, then the ECX value returned by R(1) will be set into the EAX variable returned by R(0).

The case operator reminds the Transact SQL operator of the same name. Unlike C++ switch operator, case works not only with constants, but also with variables, expressions and bit masks. In the latter case, case works as a predicate of arity N, where N is a number of entries of variables in the concatenation sequence. This feature of case operator is used for representation of tabular information, such as processor instruction specification. In the following CAHDL code

```
bits<3> Ns, Nd; // Src and dst register number
bits<1> W; // ? dword : word
short opcode = m_memory.ReadByte(EIP);
case(opcode)
{
 maskb  1:0:0:0:1:0:1:W:1:1:Nd:Ns  then
   MOV(R(Ns,W),R(Nd,W));
   break;

 maskb  0:1:0:0:Nd:0              then
   INC(R(Nd));
   break;
}
```

the case statement represents the table with two rows specifying MOV and INC instructions of IA-32 architecture. The binary concatenation located between maskb and then keywords acts like a set of predicate arguments. Each entry of a variable is an argument to the predicate. If there is a possible set of values of the variables in the concatenation sequence which make the sequence equal to the case operand, then the variables will be assigned with these values and the instruction sequence following the then keyword will be executed. The length of the variables in the sequence must be static to make it possible to find a predicate solution. In order to use octal or hexadecimal concatenation, maskb should be replaced with masko or maskh respectively and the digit prefixes (‘0’ or ‘0x’) should be omitted. In the next version of CAHDL it is planned to extend the case statement capabilities to make them working not only with concatenation sequences, but also with Boolean expressions. In this way it will remind Prolog rules.

Implementation

The emulator implemented to test the CAHDL has the following structure.

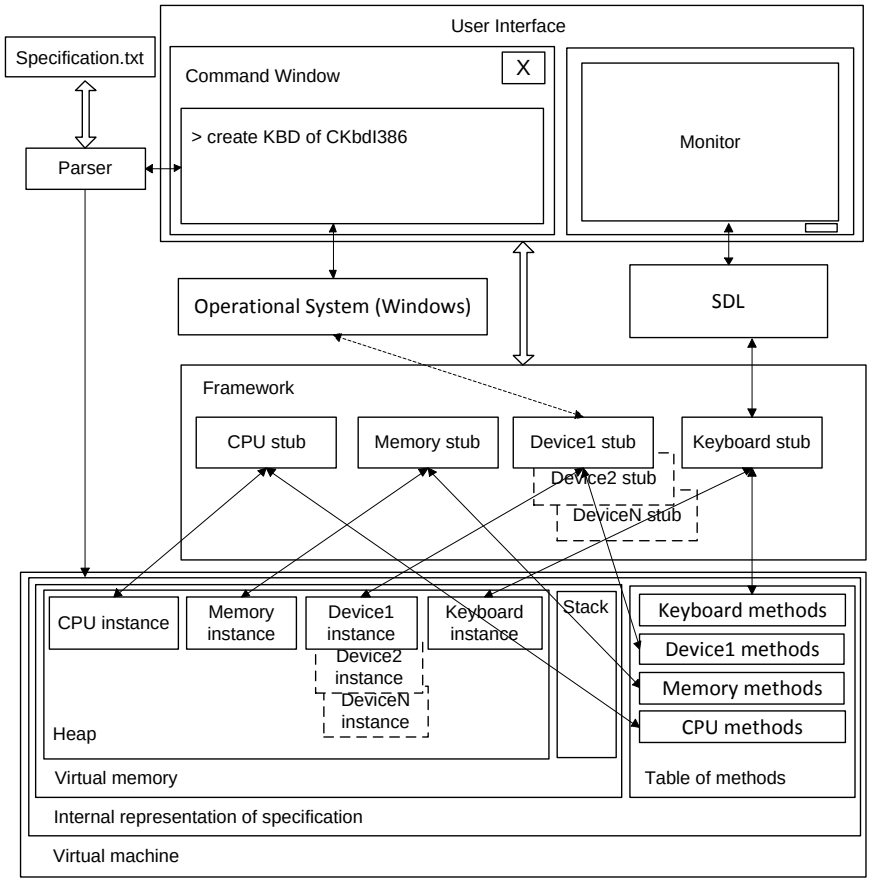

Fig. 2: Emulator's structure

It consists of user interface, framework, virtual machine and parser. The text of architecture specification is located in an outer plaintext file. The emulator works as follows: when user starts the application and selects the hardware to create, the specifications of the hardware are parsed into internal representation then the specified memory image is loaded into memory and the emulation starts.

The user interface of the emulator has two windows. The first window is console. It is a main application window aimed to read user configuration commands. Another one is a graphical window created by means of the SDL library [11]. It is aimed to emulate the computer monitor.

The emulator framework is a set of stub computer devices. It complies with the von Neumann architecture. There are processor, memory, I/O devices, etc. Since all the devices are stubs, they must be overridden by child classes in the text of specification. The interaction between the stub objects and their child class instances is based upon the event driven programming. The stub objects call their overridden methods. If a method it absent, then the corresponding stub method is called. The following code defines a class describing keyboard architecture:

```
class CKbdI386 : public CKbd
{
    ...
    void KeyDown(bits<32> code)
    {
        ...
    }
    void Compile(CComputer* pc){};
}
```

When the stub keyboard receives a message that a key was pressed it calls then KeyDown method of the keyboard object. When the method is finished, the control returns back to the stub object.

The state of execution of the emulator is stored in the memory of the virtual machine. (We should distinguish the memory of the virtual machine from the RAM object). The memory of the virtual machine consists of heap and stack. Global variables are allocated in the heap and local variables are allocated on the stack. The instances of inherited from stubs classes are global variables. The framework uses a virtual machine to execute the code of methods.

The virtual machine is an abstract stack machine. Before calling a subroutine the caller pushes the arguments from right to left onto the stack, so that the this pointer is always located in the same shift. Then the control goes to the subroutine. It saves the stack frame, executes the commands, deletes the arguments, pushes the result of corresponding length and returns control to the caller.

The set of devices to use is configured manually. A script of configuration may look as follows:

```
create PC of CComputerI386
create CPU of CCpuI386
create RAM of CMemI386
create KBD of CMemI386
create VID of CVga
create FDD of CFloppy
create PIT of CPit
compile
load freedos.img into FDD
start
```

The identifiers CComputerI386, CCpuI386, CMemI386, CMemI386, CVga, CFloppy, CPic are successors of the stubs for computer, processor, memory, keyboard, video adapter, floppy and timer respectively. The compile command combines the entities of the devices. The load command loads the specified image into floppy and the start command starts the emulation.

To parse the specification a parser generated by Bison and Flex is used [12]. The CAHDL has some syntax structures, which could not be reduced by a single lookahead symbol, thus the GLR algorithm was preferred to LALR(1). The main difference between LALR(1) and GLR is that GLR can backtrack more than one parsing node if the production is not matched [12-14].

Conclusions

The paper describes the Computer Architecture and Hardware Description Language (CAHDL). The syntax of the language is based upon the C++ and Verilog syntax. Some examples of usage of the language were given. The language developed can be used for development of the programs, which require information about computer's architecture, such as compilers, decompilers, emulators, etc.

The CAHDL was tested on two computer architectures I8080 and PDP11. They were specified by 600 and 2000 lines of code respectively. At this time a specification of the real mode of I386 architecture is being developed. The estimated number of lines of the specification code is 5000. Considering that the language is still being improved, the number of lines of code may decrease significantly.

References

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, http://www.intel.ru/content/www/ru/ru/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html

[2] ARM Information Center, http://infocenter.arm.com

[3] IEEE Standard Verilog® Hardware Description Language, http://inst.eecs.berkeley.edu/~cs150/fa06/Labs/verilog-ieee.pdf

[4] IEEE Standard VHDL Language Reference Manual, http://edg.uchicago.edu/~tang/VHDLref.pdf

[5] Brusnetzov N.P. Microcomputers. – Moskow. Nauka. Home editing of physical and mathematical literature. 1985, pp. 34-64.

[6] Myers G. Advances in computer architecture. 2nd edition. Willey – interscience publication; 1982.

[7] Brumm P. 80386 A Programming and Design Handbook. TAB Professional and Reference Books. 1987.

[8] Tanenbaum A. Structured computer organization. 5th edition. PH PTR; 2006.

[9] Ramsey N., Fernandez M. "Specifying Representation of Machine Instructions", ACM Trans. Program. Lang. Syst. Vol 19, No 3. 1997, pp. 492-524.

[10] A.A. Aho, M.S. Lam, R. Sethi, J.D. Ullman. Compilers, principles, techniques, and tools. Addison-Wesley Publishing Company. 1986. P. 26.

[11] Simple DirectMedia Layer – Homepage, http://www.libsdl.org/index.php

[12] Bison – GNU parser generator, http://www.gnu.org/software/bison/

[13] Donald E. Knuth. "On the Translation of Languages from Left to Right", Information and control 1965, Vol 8, pp. 608-639.

[14] Friedl J. Mastering regular expressions. 3rd edition: O'Reilly; 2006.