The Computer Architecture and Hardware Description Language

A.S. Burlakov, A.E. Khmelnov
Institute for System Dynamics and Control Theory of Siberian Branch of Russian Academy of Sciences,
Irkutsk, Russian Federation
alex_burlakov@bk.ru

The paper introduces CAHDL, the Computer Architecture and Hardware Description Language. CAHDL is aimed to specify processor instructions and computer hardware formally. Although CAHDL supports RTL modeling to some extent, it is does not describe digital circuits. CAHDL is designed first to specify CPU and hardware behavior, rather than their structure. CAHDL can be applied in simulators, compilers, disassemblers, testers and other programs requiring specification of computer architecture. In the end of the paper we observe how CAHDL is applied in computer simulation. The paper brings some CAHDL syntactic constructions as examples and explains their semantics. Parsing methods are also briefly reviewed.

I. INTRODUCTION

There are many hardware description languages like Verilog HDL [1] or VHDL [2] aimed to specify integrated circuits. These languages work on three levels of abstraction: on structural level, on RTL level and on behavioral level. Although CAHDL supports RTL modeling to some extent, it works on behavioral level. The main purpose of the language is to provide a high level specification of a computer architecture which could be used in software.

The CAHDL is an object oriented imperative language with declarative constructions. It operates the high level constructions like processor, memory, keyboard, etc. The syntax of CAHDL bases on C++ ISO/IEC programming language. There are also some syntax constructions taken from Verilog and from technical literature [3-6]. The source [7] was used in the development of the syntax of description of arguments of processor instructions.

The CAHDL was successfully applied in the DCU32INT program (Delphi Compiled Units Disassembler) [8] alongside with FlexT specification language [9]. In the end of the paper we observe the application of CAHDL in a computer simulator. This simulator was used in testing of the CAHDL itself and in debugging specifications written in CAHDL.

II. SYNTAX

Computer Architecture and Hardware Description Language is an object-oriented programming language designed for describing computer architecture. The syntax was influenced by C++ and Verilog and reminds C++ to some extent. However, there are special syntactical constructions, which cannot be easily realized in any existing programming language, even by C macros.

The language supports data types of objects, integers and strings. They may be used in scalar variables, arrays, constants and references. There are special mechanisms of *nesting* and *concatenation* of integer variables (see below) which are implemented by the use of references.

Besides the standard assignment operators (=, +=, -=, *=, /=, &=, |=, %=) and program flow control operators (*if*, *for*, *while*), the language provides operators working with different bits. These operators considerably facilitate the process of describing of computer architecture [10].

The most interesting of the operators are extract, concatenation and branching operators. Extract operators can extract a separate bit <*bit_number*> or a sequence of bits <*higher_bit..lower_bit*>. We don't use the Verilog form of bits extraction with square brackets, because they are reserved for arrays. An operator of concatenation (colon character) concatenates two or more operands. A branching operator *case* works similar to the visual basic *case* operator.

In order to adjust an environment, the specialist has to operate with objects of corresponding types. Memory, processor, monitor are all some types and they are represented in a description as some device classes. The language supports an encapsulation mechanism. A class may encapsulate fields and methods. The language also supports an inheritance mechanism. It allows to access public fields and methods of a parent class. Class fields and methods could be hidden if they are located in a *private* section. Like in C++, if a field is located in *protected* section, it is only available in derived classes. Listing 1 represents a fragment of the description of the most complicated computer device – CPU (Central Processor Unit).

```
class CCpuIx86 : public CCpu
{
private:
  CMemoryIx86 m_memory;
  // general purpose registers
  bits<32> EAX, EBX, ECX, EDX;
  bits<32> ESI, EDI;     // index
  bits<32> ESP, EBP;     // stack
  bits<32> EIP;          // program
  bits<32> EFLAGS;       // flag
  bits<16> CS,SS,DS,ES,FS,GS;
  bits<16>& AX = EAX<15..0>;
  bits<8>& AH = AX<15..8>;
```

```
  bits<8>& AL = AX<7..0>;
  bits<8>& BH = EBX<15..8>;
  bits<8>& BL = EBX<7..0>;
  bits<16>& BX = BH:BL;
  // SP length would be set automatically
  bits& SP = ESP<15..0>;
  bits<1>& CF = EFLAGS<0>;
  bits<1>& ZF = EFLAGS<6>;
public:
  bits& RNs(bits<3> N, bits<1> W){
    if(W)    case(N){
      maskb 0:0:0 then return EAX;
      maskb 0:0:1 then return ECX;
      maskb 0:1:0 then return EDX;
      ...
    }
  }
  void Process(){
    bits<3> Ns, Nd;
    bits<1> W;   // ? dword : word
    unsigned 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));
      maskb 0:1:0:0:Nd:0          then INC(R(Nd));
      maskh 1:e                   then PUSH(DS);
      maskh 1:f                   then POP(DS);
      ...
    }
  }
  ...
};
```

LISTING 1. EXAMPLE OF IA-32 PROCESSOR ARCHITECTURE DESCRIPTION

The listing 1 is an incomplete description of IA-32 processor architecture. This description is represented as a *CCpuIx86* class. In the body of the class are registers and a memory type object (see Listing 5). General purpose registers, index registers and stack pointer registers are represented as integer variables 32 bits length. Segment registers are 16 bits integers.

The reference type realizes mechanisms of nesting and concatenation. These mechanisms make it possible to declare AX register as a nested variable of EAX register. AH and AL registers are nested in AX register. At the same time, a variable containing other variables may be declared as a concatenation (colon character) of these variables. Thus, in the example, BX is declared as a concatenation of AH and AL registers.

It is notable, that if the size of the variable is not defined, then it would be set automatically, as it is done for SP register. In that case, the length of SP register would be 16 bits pointing to the first 16 bits (from 0 to 15) of ESP register.

The carry and zero flags are defined as a single bit variables pointing to zero and sixth bits of EFLAGS respectively.

It is worthy to discuss the *case* brunching operator. The keyword *case* was chosen to designate for branching operator because of its similarity to the Visual Basic *case* operator. It works not only with constants as C++ *switch* operator does, but also with expressions and bitmasks. An example of the *case* operator could be found in listing 1, in the *Process* and *RNs* functions. In the row

```
maskb 1:0:0:0:1:0:1:W:1:1:Nd:Ns
```

within the *Process* function, the *case* statement works as a predicate of arity 3, where 3 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. The syntax of *case* operator is as following:

```
CASE '(' expr ')' '{'
  case_body_list
'}' ';'

case_body_list: case_body_row
| case_body_list case_body_row
;

case_body_row:
  WHEN expr1 THEN stmt_list1
  | MASKB binary_mask_conc      THEN stmt_list2
  | MASKO octal_mask_conc       THEN stmt_list3
  | MASKH hexadecimal_mask_conc THEN stmt_list4
;
```

LISTING 2. CASE SYNTAX

The keyword *when* is followed by an *expr1*, the result of which will be compared to the result of *expr*. In case of matching, the *stmt_list1* will be executed. After the keywords *maskb*, *masko*, *maskh* follows a concatenation of numbers written in binary, octal or hexadecimal format, but without prefixes '0' or '0x' for octal and hexadecimal numbers respectively. If there is a variable in a concatenation sequence, it would be interpreted as a number, rather than a variable. For instance, in a row

```
maskb 1:0:0:0:1:0:1:W:1:1:Nd:Ns
    then MOV(R(Ns,W),R(Nd,W));
```

LISTING 3. IA-32 MOV COMMAND IN CAHDL

after the keyword *maskb* follows a sequence of binary formatted bits and variables, joined by the concatenation operator. When this concatenation is compared to a number, these variables may be interpreted as any value, which does not exceed the length of a corresponding variable. The length of *Ns* and *Nd* is 3 bits. It is known from their declarations (see listing 1).

A concatenation may be considered as a bitmask. If a mask matches some required number, then the variables in a concatenation sequence would be assigned with values so that the result of concatenation would be equal to the value required. Instead of using of C++ statements like in example below

```
// expr – is a case argument
// look listing 2
Ns = expr & 7;
Nd = (expr >> 3) & 7;
if((expr >> 6) & 3!= 3)
  return;
W = (expr >> 8) & 1;
if((expr >> 9) & 127!= 69)
  return;
MOV(R(Ns,W), R(Nd,W));
```

LISTING 4. FRAGMENT OF IA-32 MOV COMMAND IN C

we write this instruction in two lines of code (see listing 3).

It is notable, that different formats of a digit representation cannot be mixed in a single row, i.e. *maskb* cannot be followed by digits in a hexadecimal format. Moreover, if we write some digits after *maskh* keyword, these digits would be interpreted as hexadecimal ones. For example, an instruction of concatenation *maskh 1:1* would be interpreted as number 17 rather than number 3.

Abstract classes are used as an interface of basic devices. Like in other programming languages, in CAHDL we cannot create objects of abstract classes. In order to create a device we need to define its behavior in a class inherited from the base abstract class. In the example above (see listing 1) the class *CCpuIx86* inherits fields and methods of a virtual abstract *CCpu* class, hence it must implement all *CCpu* abstract methods, one of which is the *Process* method.

Other computer devices like random access memory or hard disk drive may be described in a similar way. There we need to define an inherited class, which will implement all the abstract methods. Let's see how it would look like for a memory class.

```
class CMemoryIx86 : public CMemory
{
private:
  // the size of memory in bytes
  const RAM_SIZE = 0x80000000;//2Gb

public:
  CMemoryIx86(){…};
  unsigned char ReadByte(unsigned long address){…};
  void WriteByte(unsigned long address
                , unsigned char value) {…};
  …
}
```

LISTING 5. MEMORY CLASS IN CAHDL

The code in listing 5 is self-explanatory. There we define memory class as one derived from *CMemory* class. In order to set a memory size, we just change the value of *RAM_SIZE* constant.

If we need to track memory changes, we would have to edit code for *WriteByte* method. It is also possible to change CPU class methods in order to track register values. (Tracking values changes method is widely used in debugging technics.) Besides memory tracking we may even change the logic of some methods, accordingly to our demands.

III. PARSING

The description of a computer's architecture is stored in a simple text file. When the simulator is started, the function *yyparse* begins to parse the description text. The body of *yyparse* function is generated automatically by Bison and Yacc programs, according to the language grammar description[11]. The CAHDL has some syntactical constructions, 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 [11,12]. In the body of *yyparse* function we build a parsing tree. In the nodes of the tree are objects of different classes. However, all these classes have the same base class.

The grammar of the language developed is context-sensitive. The aim of context-sensitivity is to simplify code understanding. Context-sensitivity deals only with the representation of numbers. In general, hexadecimal numbers must follow '0x' prefix. If we use a hexadecimal number in the *case* construction within *maskh* section, then the prefix must be omitted. In case of ambiguity, when an identifier may be interpreted as both a number and a variable of the same name, the number would be preferred due to the fact that it is easier to rename the variable rather than represent a number in another form.

For example (see listing 1), *PUSH* operation in *Process* method would be executed, if only the *opcode* variable matches mask 0x1e. In this case, the symbol *e* would be interpreted as a number 14, rather than an "e" named variable. This means that *opcode* must be equal to 0x1e.

The language supports modality at a level of files. This modality is similar to one used in C++. In order to include another file of description, we have to place it in quotation marks following an *#include* keyword.

Bison and Yacc generate a parser. This parser is not an executive file, but a bunch of source files. This source code is included into the simulator's project. When the embedded parser founds a grammar mistake in a description file, it alerts an error outputting some debug information and then it terminates the program's execution. Now, the debugging information is quite simple and reveals only position of a mistake, but does not tells, what kind of mistake there was. It is supposed to improve this description debugging technics in the later releases of CAHDL.

IV. APPLICATION

The CAHDL was used in Delphi Compiled Units Disassembler [8]. (This tool is aimed to parse *.dcu files and represent their structure in terms of Delphi statements and functionality in terms of assembler code.) The CAHDL was also applied in computer simulator to specify computer architectures. This simulator is observed below.

V. THE SIMULATOR'S STRUCTURE

The simulator has the 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 structure of the simulator can be represented as follows:

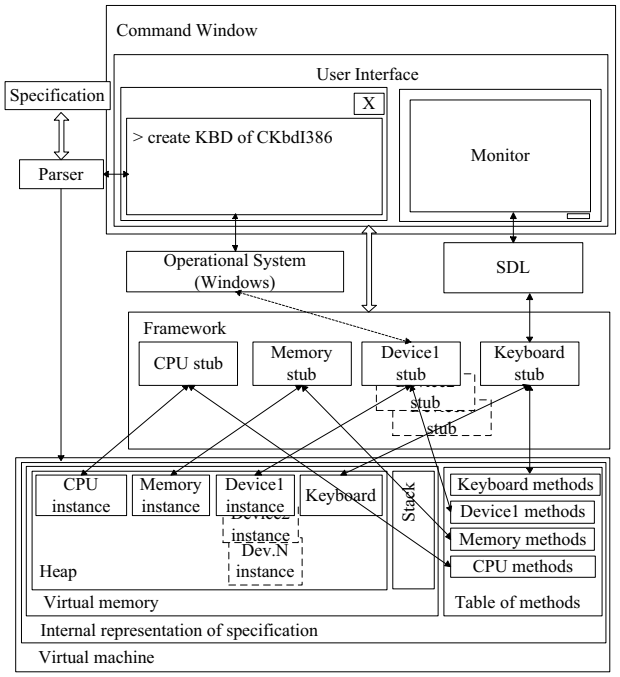

FIGURE 1. SIMULATOR'S CLASS HIERARCHY

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 is absent, then the corresponding stub method is called.

The simulator consists of two windows. The main window is console another is graphical.

A. Console window

Console window is used to adjust the environment. There the user inputs a path to an image file to upload, a path to a file with specification of architecture and a command that starts simulation.

B. Graphical window

Graphical window is created by SDL library [13][13] only when the simulator starts, because the resolution of the screen and other settings are set only when the architecture description is already parsed and interpreted. This window is aimed to simulate the computer's screen. Graphical window may also work in a textual mode. In this mode, all the symbols are represented as small bitmaps. Graphical window handles keyboard buttons messages, mouse movements and mouse button clicks. In order to synchronize two windows an infinite cycle is used rather than a threading mechanism, because it makes the debugging of simulator more convenient.

C. Periphery devices

Simulated periphery devices interact with a user by the standard Windows API functions. For example, when a mouse button is clicked within bounds of SDL simulated window i.e. the simulated screen, a message loop handles the message and dispatches it to a mouse device object. Mouse object interrupts processing of the simulated operational system. When the interruption is handled, the processor goes on processing the operational system, where the cursor's position is already shifted.

VI. INITIALIZATION

The simulator's environment is being initialized only when a text of a computer's architecture description is parsed into a set of objects of different classes. Every syntax construction has its own class. For example, there are classes storing objects of type of class, function, variable, integer, operation, etc. All the objects of these classes are stored hierarchically. For instance, a variable may belong whether to a function or to a class. A function may only belong to a class. A class of an upper level contains all the lexical classes.

The initialization proceeds in two steps. Firstly, the class and function headers are parsed. Secondly function bodies are traversed. Two traversals are needed to be able to refer to a method which is declared in the text below.

VII. SIMULATION

When the simulator's environment is properly configures, there may start the application to do simulation of the required architecture. In order to start simulation, the specialist is provided with the console interface where he has to input a path to an image-file, which would be loaded into the simulator's memory.

When the simulation stars, the program creates a screen object by the use of SDL library (see Simulators structure). Then the main loop is started to handle device messages (mouse movements and clicks, keyboard buttons pressed, etc.).

VIII. CONLUSION

The paper describes the Computer Architecture and Hardware Description Language. The syntax of the language is based upon the C++ and Verilog syntax. Some examples of usage of the language were given. The CAHDL 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] IEEE Standard Verilog® Hardware Description Language, http://inst.eecs.berkeley.edu/~cs150/fa06/Labs/verilog-ieee.pdf

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

[3] Brusnetzov N.P. Microcomputers. – Moskow: "Nauka", home editing of physical and mathematical literature, 1985. UDC 519.6

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

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

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

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

[8] DCU32INT - Delphi Compiled Units Disassembler, http://hmelnov.icc.ru/DCU/index.ru.html

[9] Data description language FlexT: Flexible types for description static data, http://hmelnov.icc.ru/FlexT/FLEXT_CSCC99.htm

[10] A.S. Burlakov. "Specification Language of Computer Simulator Virtual Hardvare", Vestnik of Irkutsk State Technical University. Vol. 12, 2014, pp. 12-17

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

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

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