A Formal Model of Microcontroller Instruction Set in B

Valério Medeiros Jr[1], David Déharbe[1]

Federal University of Rio Grande do Norte, Natal RN 59078-970, Brazil

Abstract. This paper describes an approach to model the microcontroller platforms. More specifically, it shows details about the Z80 model. The model has been developed using the B method; which applies math and logic concepts to describe characteristics from platforms. Therefore, this modelling can be used in platform projects to document, build simulators, verify properties about the model and verify at assembly level software. The verification at assembly is the feature more important, because this allows to develop faithful software to the algorithm language. Thus, the platform model was built for this purpose. Finally, this paper shows some relevant techniques used in models and to describe some details of our approach.

1 Introduction

This work describes briefly the modelling of microcontroller Z80 [6] and some techniques used to build it. This modelling[1] use the B method [1], because it has a great maturity and advanced support. Besides, the modelling of microcontroller can extend the level of verification of B method, as it will be explained.

The B method supports the construction of safety systems model by verification of proofs that guarantees its correctness. So, an initial abstract model of the system is defined and refined until the implementation model. The B tools still have the code generator to programming language, but this feature is not verified. In [4] is presented an approach to extend the verification of B method until the assembler language. In this approach, the construction of systems begins from the abstract level and it is formally refined until the target assembly platform.

Therefore, the first step of this approach is to build the platform modelling, of the microcontroller, in B. Using the responsibility division mechanism provided by B, basic modules was developed to be used as elementary library to help the constructing of microcontroller model. The elementary library has many definitions about common concepts used in the microcontrollers. This library is used by Z80 model and other two model microcontrollers that are under construction.

The microcontroller modelling can be used in platform projects documentation, simulators construction, consistency verification and software verification

[1] The interested reader in more details is invited to visit our repository at: http://code.google.com/p/b2asm

at assembly level. The verification at assembly level is more interesting utility because it allows the development of trustworthy software by construction, that is extremely desirable in embedded critical software.

In this paper we introduced the modelling of Z80 microcontroller, since elementary library until assembly instructions. The paper is structured as follows. Section 2 provides a general introduction to the B method. Section 3 presents the elementary libraries and of microcontrollers basic concepts. Section 4 presents the Z80 B modelling. Section 5 explain a little about the proofs of models. The next section (6) discusses the related works. Finally, the last section has the conclusions.

2 B Method

The B method for software development [1] is based on the B Abstract Machine Notation (AMN) and the use of formally proved refinements up to a specification sufficiently concrete that programming code can be automatically generated from it. Its mathematical basis consists of first order logic, integer arithmetic and set theory, and its corresponding constructs are very similar to those of the Z notation

A B specification is structured in modules. A module defines a set of valid states, including a set of initial states, and operations that may provoke a transition between states. The design process starts with a module with a so-called functional model of the system under development. In this initial modelling stage, the B method requires that the user proves that, in a machine, all the initial states are valid, and that operations do not define transitions from valid states to invalid states.

Essentially, a B module contains two main parts: a header and the available operations, in the figure 1 has a intuitive example. The clause *MACHINE* has the name of module. The following two are used to access an external module and create an instance of an external module. The *VARIABLES* declares the variable names. The following declares its types and the restrictions. The *INITIALIZATION* defines the its initial value. The last define the operations that remember procedures from programming language.

MACHINE *micro*	**INITIALISATION** $pc := 0$
SEES *TYPES*, *ALU*	**OPERATIONS**
INCLUDES *MEMORY*	$JMP(jump) =$
VARIABLES *pc*	**PRE** $jump \in INSTRUCTION$
INVARIANT $pc \in INSTRUCTION$	**THEN** $pc := jump$
	END ...

Fig. 1. A part of B simplified module

3 Overview of Plataform's B Models

We have been developed a reusable set of basic definitions to model hardware concepts and data types concepts. These definitions are grouped in two projects of libraries and other project to model the platform. Thus, our workspace is arranged through three projects: hardware library, types library and the project of specific platform, in this case the Z80.

The dependency graph between these projects is depicted in figure 2 and the next sections will explain each one of them.

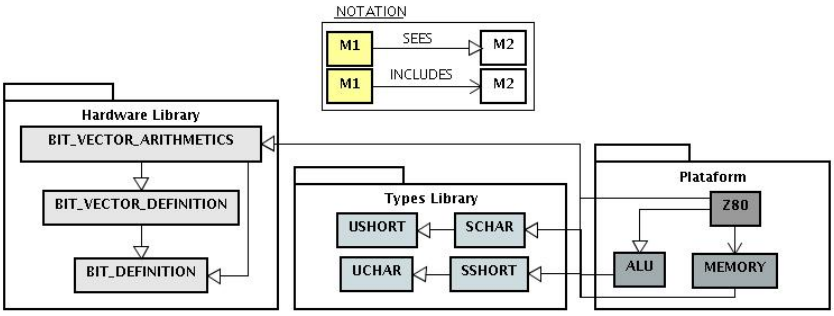

Fig. 2. Dependency graph between the projects that compose the platform model of Z80.

3.1 Definitions to Represent and Manipulate Bits

The entities defined in the module *BIT_DEFINITION* are the type for bits, logical operations on bits (negation, conjunction, disjunction, exclusive disjunction), as well as a conversion function from booleans to bits.

First, bits are modelled as the set of integers: $BIT = 0..1$. The negation is an unary function on bits and it is defined as:

$bit\_not \in BIT \rightarrow BIT \wedge \forall(bb).(bb \in BIT \Rightarrow bit\_not(bb) = 1 - bb)$

The module also provides lemmas on negation that may be useful for the users of the library to develop proofs:

$\forall(bb).(bb \in BIT \Rightarrow bit\_not(bit\_not(bb)) = bb)$

Conjunction is an unary function on bits and it is defined as:

$bit\_and \in BIT \times BIT \rightarrow BIT \wedge$
$\forall(b1, b2).(b1 \in BIT \wedge b2 \in BIT \Rightarrow$
$((bit\_and(b1, b2) = 1) \Leftrightarrow (b1 = 1) \wedge (b2 = 1)))$

The module provides the following lemmas for conjunction, either:

$\forall(b1, b2).(b1 \in BIT \wedge b2 \in BIT \Rightarrow$
$(bit\_and(b1, b2) = bit\_and(b2, b1))) \wedge$
$\forall(b1, b2, b3).(b1 \in BIT \wedge b2 \in BIT \wedge b3 \in BIT \Rightarrow$
$(bit\_and(b1, bit\_and(b2, b3)) = bit\_and(bit\_and(b1, b2), b3)))$

The module provides definitions of *bit_or* (disjunction) and *bit_xor* (exclusive disjunction), as well as lemmas on those operators. These are standard and their expression in B is similar as for *bit_and*, they are thus omitted.

Finally, the conversion from booleans to bits is simply defined as:

$bool\_to\_bit \in \mathbf{BOOL} \rightarrow BIT \wedge bool\_to\_bit = \{\mathbf{TRUE} \mapsto 1, \mathbf{FALSE} \mapsto 0\}$

Observe that all the lemmas that are provided in this module have been mechanically proved by the theorem prover included with our B development environment. None of these proofs requires human insight.

3.2 Representation and Manipulation of Bit Vectors

Sequences are pre-defined in B, as functions whose the domain is an integer range with lower bound 1 (one). Indices in bit vectors usually range from 0 (zero) upwards and the model we propose obeys this convention by making an one-position shift where necessary. This shift is important to use the predefined functions of sequences. We thus define bit vectors as non-empty sequences of bits, and BIT_VECTOR is the set of all such sequences: $BIT\_VECTOR = \mathrm{seq}(BIT)$

The function *bv_size* returns the size of a given bit vector. It is basically a wrapper for the predefined function **size** that applies to sequences.

$bv\_size \in BIT\_VECTOR \rightarrow \mathcal{N}_1 \wedge$
$bv\_size = \lambda\, bv.(bv \in BIT\_VECTOR \mid \mathbf{size}(bv))$

We also define two functions *bv_set* and *bv_clear* that, given a bit vector, and a position of the bit vector, return the bit vector resulting from setting the corresponding position to 0 or to 1, and a function *bv_get* that, given a bit vector, and a valid position, each one returns the value of the bit at that position. Only the first definition is shown here:

$bv\_set \in BIT\_VECTOR \times \mathcal{N} \rightarrow BIT\_VECTOR \wedge bv\_set =$
$\lambda\, v, n.(v \in BIT\_VECTOR \wedge n \in \mathcal{N} \wedge n < bv\_size(v) \mid v \leftarrow\!\!+ \{n+1 \mapsto 1\})$

Additionally, the module provides definitions for the classical logical combinations of bit vectors: *bit_not*, *bit_and*, *bit_or* and *bit_xor*. Only the first two are presented here. Observe that the domain of the binary operators is restricted to pairs of bit vectors of the same length:

$bv\_not \in BIT\_VECTOR \rightarrow BIT\_VECTOR \wedge$
$bv\_not = \lambda\, v.(v \in BIT\_VECTOR \mid \quad \lambda\, i.(1..bv\_size(v)) \mid bit\_not(v(i))) \wedge$
$bv\_and \in BIT\_VECTOR \times BIT\_VECTOR \rightarrow BIT\_VECTOR \wedge$
$bv\_and = \lambda\, v_1, v_2.(v_1 \in BIT\_VECTOR \wedge v_2 \in BIT\_VECTOR \wedge$
$\quad bv\_size(v_1) = bv\_size(v_2) \mid \lambda\, i.(1..bv\_size(v_1)) \mid bit\_and(v_1(i), v_2(i)))$

We provide several lemmas on bit vector operations. These lemmas express properties on the size of the result of the operations as well as classical algebraic properties such as associativity and commutativity.

3.3 Modelling Bytes and Bit Vector of Length 16

Bit vectors of length 8 are bytes. They form a common entity in hardware design. We provide the following definitions:

$BYTE\_WIDTH = 8 \wedge BYTE\_INDEX = 1 \,..\, \mathrm{BYTE\_WIDTH} \wedge$

$PHYS\_BYTE\_INDEX = 0 \,..\, (BYTE\_WIDTH\text{-}1) \quad \wedge$
$BYTE = \{\ bt \mid bt \in BIT\_VECTOR \wedge bv\_size(bt) = BYTE\_WIDTH\} \quad \wedge$
$BYTE\_ZERO \in BYTE \wedge BYTE\_ZERO = BYTE\_INDEX \times \{0\}$

The *BYTE_INDEX* is domain of byte modelled. It starts at 1 to obey a definition of sequences from B method. However, the functions from byte encapsulate the access and these functions use the *PHYS_BYTE_INDEX*. The *BYTE* type is a specialized type from *BIT_VECTOR*, but it has a size limit. Some others more specific definitions are created to help the developer. The type *BV16* is created for bit vector of length 16 in a similar way.

3.4 Bit Vector Arithmetics

Bit vectors are used to represent and combine numbers: integer ranges (signed or unsigned). Then, our library defines functions to manipulate this date, for example, the function *bv_to_nat* that maps bit vectors to natural numbers:

$bv\_to\_nat \in BIT\_VECTOR \rightarrow \mathcal{N} \wedge$
$bv\_to\_nat = \lambda\, v.(v \in BIT\_VECTOR \mid \sum i.(i \in \mathsf{dom}(v).v(i) \times 2^i))$

An associated lemma is: $\forall\, n.(n \in \mathcal{N}_1 \Rightarrow bv\_to\_nat(nat\_to\_bv(n)) = n)$

3.5 Basics Data Types

The instruction set of microcontrollers usually have common data types. These types are placed in the types library. Each type module has functions to manipulate and convert its data. There are six common basics data types represented by modules, see details in table 1.

Table 1. Descriptions of basic data types

Type Name	UCHAR	SCHAR	USHORTINT	SSHORTINT	BYTE	BV16
Range	0..255	-128..127	0..65.535	-32.768..32.767	–	–
Physical Size	1 byte	1 byte	2 bytes	2 bytes	1 bytes	2 bytes

Usually, for each type module just needs to detail or redefines the pre-defined concepts. For example, the function *bv_to_nat* from bit vector arithmetics that is specialized to *byte_uchar*. The set *BYTE* is a subset of the *BIT_VECTOR*, then this function can defined as follows:

$byte\_uchar \in BYTE \rightarrow \mathcal{N} \wedge$
$byte\_uchar = \lambda(v).(v \in BYTE | bv\_to\_nat(v))$

The definitions of the library types reuse the basic definitions from the hardware library. This provides greater confidence and facilitates the prove process, because the prover can reuse the previous defined lemma.

The inverse function is easily defined as *uchar_byte*.

$uchar\_byte \in UCHAR \rightarrow BYTE \wedge$
$uchar\_byte = (byte\_uchar)^{-1}$

Similarly, several other functions and lemmas were created for all other data types.

4 Description of the Z80 B model

The *Z80* is an important CISC microcontroller developed by *Zilog* [6]. It supports 158 different instructions. These instructions are classified into these categories: load and exchange; block transfer and search; arithmetic and logical; rotate and shift; bit manipulation (set, reset, test); jump, call and return; input/output; and basic cpu control.

The main module includes an instance of memory module and accesses the definitions from: basic data types modules and *ALU*.

MACHINE
Z80
INCLUDES
MEMORY
SEES
ALU, BIT_DEFINITION, BIT_VECTOR_DEFINITION,
BYTE_DEFINITION, BV16_DEFINITION,
UCHAR_DEFINITION, SCHAR_DEFINITION,
SSHORT_DEFINITION ,USHORT_DEFINITION

Each instruction is represented by B operations in the module Z80. By default, all parameters from operations are or predefined elements in the model or integers values in the decimal representation. The internal registers contain 208 bits of reading/writing memory. It includes two sets of six general purpose registers which may be used individually as 8-bits registers or as 16-bits register pairs. The work registers are represented by variable *rgs8*. The domain of *rgs8* (*id_regs8*) is a set formed by identifiers of registers of 8 bits. These registers can be accessed in pairs, forming 16-bits, resulting in other set of identifiers of 16-bits registers, named *id_reg16*. The main work register of Z80 is the accumulator (*rgs8*(*a0*)) used for arithmetic, logic, input/output and loading/storing operations.

4.1 Modelling registers, input and output ports and instructions

The Z80 has many register of diferrents types and many instructions. It includes alternative set of accumulator, flag and general registers. The CPU contains a stack pointer (*sp*), program counter (*pc*), two index registers (*ix* and *iy*), an interrupt register (*i_*), a refresh register (*r_*), two bits (*iff1*, *iff2*) used to control the interruptions, a pair of bits to define the interruption mode (*im*) and the input and output ports (*i_o_ports*). Below, its definitions are specified in the *INVARIANT*.

INVARIANT

$rgs8 \in id\_reg\_8 \rightarrow BYTE \wedge pc \in INSTRUCTION \wedge$
$sp \in BV16 \wedge ix \in BV16 \wedge iy \in BV16 \wedge$
$i\_ \in BYTE \wedge r\_ \in BYTE \wedge iff1 \in BIT \wedge iff2 \in BIT \wedge$
$im : (BIT \times BIT) \wedge$ i_o_ports $\in BYTE \rightarrow BYTE$

A simple example of instruction is a *LD_n_A*, as shown below. Many times, to model an instruction is necessary to use the predefined functions, these help the construction of model. This instruction use the *updateAddressMem* function from *Memory* module and it receives an address memory and its new memory value. Finally it increments the program counter (*pc*) and update the refresh register (*r_*).

LD_n_A (*nn*) =
  PRE $nn \in USHORT$
  THEN
  updateAddressMem (*ushort_to_bv16* (*nn*) , *rgs8* (*a0*)) ||
  $pc := instruction\_next$ (*pc*) || $r\_ := update\_refresh\_reg(r\_)$
END

The microcontroller model can specify several security properties. For example, the last operation could have a restriction to write only in a defined region of memory.

5 Proofs

The proof obligations allow to verify the data types, important system properties and if the expressions are well-defined (WD)[2]. The properties provide additional guarantees, because it can set many safety rules. However, the model can be very difficult to prove.

The building of model was a hard and difficult task, because we needed to build, prove, fix and re-prove several predicates of the libraries.

However, few proof commands[3] can be useful to prove the majority of the proof obligations. Because, there are many similar assembler instructions. A good example is a set of 17 proof commands that quickly aided the verification of 99% (2295) of proofs WD. Finally, the techniques mentioned help the verification of all the 2926 proof obligations.

6 Related Works

There are in the literature of computer science some approaches [2, 3] to model hardware and the virtual machines using the B method. Then, the B method has been used successfully to model the operational semantic. However the cost of modelling was still expensive and this paper quoted some techniques to lower

[2] An expression is called "well-defined" (or unambiguous) if its definition assigns it a unique interpretation or value.

[3] The proof commands are step that just indicate a walk to prover make the proof, thus theses commands cannot introduce false hypotheses.

the cost of modelling. Although, the modelling of the Z80 do not have modelled some small aspects.

In general, the researchers [3] are concentrated at model the Java Virtual Machines (JVM), but it is not easy to specify, because it is very big and complex. Then, this work specify a model more simple to get better results, at the same time, the model stay near to model of actual microcontrollers. The main motivation of our research is the verification of assembler level and consequently the verification of hardware model. Thus, the some aspects are not modelled. For example, the time of execution of instruction, the pipeline and others questions not directly related with the semantic software. However, there are many other specialized techniques to verify these questions.

7 Conclusions

This work has shown an approach to the formal modelling of microcontrollers using the B method. This approach provide interesting results and had a great automatization in the prove process. For instance, it is interesting to say that: during the building process of the Z80 model we found some errors and ambiguities in the official manual[6]. The platform model can replace (or improve) the documentation used by assembler programmers. Because the formal model represents the instructions effects and the B method has an easy notation, similar to pascal language. Besides, the formal model restricts the definitions to correct typing, uses expressions well-defined and allows verifying properties of the model.

Future works comprise the development of a real study case in the oil area and the project of a compiler to aid the approach.

Acknowledges: This work received supports from ANP (National Agency of Oil) and CNPq.

References

1. Abrial, J. R. The B Book: Assigning Programs to Meanings. Cambridge University Press, United States of America, 1 edition, 1996.
2. Aljer, P. Devienne, S. Tison and J-L. Boulanger andG. Mariano. Bhdl: Circuit Design in B. A. In ACSD, Third International Conference on Application of Concurrency to System Design, pages 241-242, 2003.
3. Casset L.; Lanet J. L. A Formal Specification of the Java Bytecode Semantics using the B method.Technical Report, Gemplus. 1999.
4. Dantas, B; Déharbe, D; Galvão, S; Et al. Applying the B Method to Take on the Grand Challenge of Verified Compilation. In: SBMF, Savaldor, 2008. SBC.
5. Hoare, C. A. R. The verifying compiler, a grand challenge for computing research. In: VMCAI, p. 78-78, 2005.
6. Zilog. Z80 Family CPU User Manual. www.zilog.com/docs/z80/um0080.pdf