Formal Modelling of a Microcontroller Instruction Set in B

Formal Modelling of a 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 functional aspects of the instruction set of microcontroller platforms using the notation of the B method. The paper presents specifically the case of the Z80 platform. This work is a contribution towards the extension of the B method to handle developments up to assembly level code.

1 Introduction

The B method [1] supports the construction of safety systems models by verification of proofs that guarantees its correctness. So, an initial abstract model of the system requirements is defined and then it is refined until the implementation model. Development environments based on the B method also include source code generators for programming languages, but the result of this translation cannot be compared by formal means. The paper [4] presented recently an approach to extend the scope of the B method up to the assembly level language. One key component of this approach is to build, within the framework of the B method, formal models of the instruction set of such assembly languages.

This work gives an overview of the formal modelling of the instruction set of the Z80 microcontroller [6][1]. Using the responsibility division mechanism provided by B, auxiliary libraries of basic modules were developed as part of the construction of microcontroller model. Such library has many definitions about common concepts used in the microcontrollers; besides the Z80 model, it is used by two other microcontrollers models that are under way.

Other possible uses of a formal model of a microcontroller instruction set include documentation, the construction of simulators, and be possibly the starting point of a verification effort for the actual implementation of a Z80 design. Moreover the model of the instruction set could be instrumented with non-functional aspects, such as the number of cycles it takes to execute an instruction, to prove lower and upper bounds on the execution time of a routine. The goal of this project, though, is to provide a basis for the generation of software artifacts at the assembly level that are amenable to refinement verification within the B method.

This paper is focused on the presentation of the Z80 model, including elementary libraries to describe hardware aspects. The paper is structured as follows.

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

Section 2 provides a short introduction to the B method. Section 3 presents the elementary libraries and the modelling of some elements common to microcontrollers. Section 4 presents the B model of the Z80 instruction set. Section 5 provides some information on the proof effort needed to analyze the presented models. Related work is discussed in Section 6. Finally, the last section is devoted to the conclusions.

2 Introduction to the 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 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 its 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. Figure 1 has a very basic example. The clause *MACHINE* has the name of module. The next two clauses respectively reference external modules and create an instance of an external module. The *VARIABLES* clauses declares the name of the variables that compose the state of the machine. Next, the *INVARIANT* clause defines the type and other restrictions on the variables. The *INITIALIZATION* specifies the initial states. Finally, operations correspond to the transitions between states of the machine.

```
MACHINE    micro
SEES    TYPES, ALU
INCLUDES    MEMORY
VARIABLES    pc
INVARIANT    pc ∈ INSTRUCTION

INITIALISATION pc := 0
OPERATIONS
JMP(jump) =
  PRE jump ∈ INSTRUCTION
  THEN pc := jump
  END
END
```

Fig. 1. A very basic B machine.

3 Model structure and basic components

We have been developed a reusable set of basic definitions to model hardware concepts and data types concepts. These definitions are grouped into two separate development projects and are available as libraries. A third project is devoted to the higher-level aspects of the platform. Thus, the workspace is composed of: a hardware library, a types library and a project for the specific platform, in this case the Z80. The corresponding dependency diagram is depicted in Figure 2; information specific to each project is presented in the following.

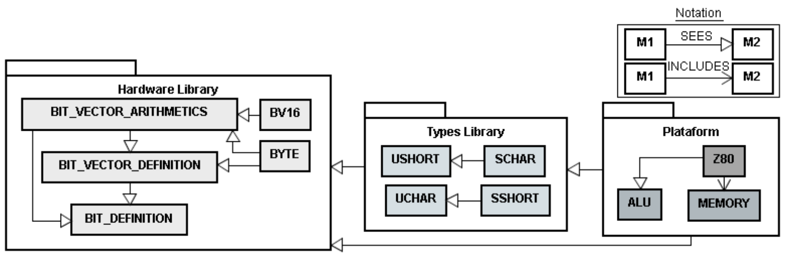

Fig. 2. Dependency diagram of the Z80 model.

3.1 Bit Representation and Manipulation

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 a 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:

$$\begin{array}{l} bit\_and \in BIT \times BIT \rightarrow BIT \wedge \\ \forall(b1, b2).(b1 \in BIT \wedge b2 \in BIT \Rightarrow \\ \quad ((bit\_and(b1, b2) = 1) \Leftrightarrow (b1 = 1) \wedge (b2 = 1))) \end{array}$$

The module provides the following lemmas for conjunction, either:

$$\begin{array}{l} \forall(b1, b2).(b1 \in BIT \wedge b2 \in BIT \Rightarrow \\ \quad (bit\_and(b1, b2) = bit\_and(b2, b1))) \wedge \\ \forall(b1, b2, b3).(b1 \in BIT \wedge b2 \in BIT \wedge b3 \in BIT \Rightarrow \\ \quad (bit\_and(b1, bit\_and(b2, b3)) = bit\_and(bit\_and(b1, b2), b3))) \end{array}$$

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:

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

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 Vectors of Length 16

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

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

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

The $BYTE\_INDEX$ is the domain of the functions modelling bytes. It starts at 1 to obey a definition of sequences from B. However, it is common in hardware architectures to start indexing from zero. The definition $PHYS\_BYTE\_INDEX$ is used to provide functionalities obeying this convention. The $BYTE$ type is a specialized type from BIT_VECTOR, but it has a size limit. Other specific definitions are provided to facilitate further modelling: 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). Therefore, our library includes functions to manipulate such data, 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, each type module just needs to instantiate concepts that were already defined in the hardware modelling library. For example, the function bv_to_nat from bit vector arithmetics is specialized to $byte\_uchar$. As the set $BYTE$ is a subset of the BIT_VECTOR, 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 proof process, because the prover can reuse the previously defined lemma.

The inverse function $uchar\_byte$ is easily defined:

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

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

4 Description of the Z80 B model

The *Z80* is a CISC microcontroller developed by *Zilog* [6]. It supports 158 different instructions and all of them were specified. These instructions are classified into these categories: load and exchange; block transfer and search; arithmetic and logical; rotate and shift; bit manipulation; jump, call and return; input/output; and basic cpu control.

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

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 a B operation in the module Z80. By default, all parameters from operations are either 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 working 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 another set of identifiers of 16-bits registers, named *id_reg16*. The main working 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 different types of registers and instructions. The CPU contains general-purpose registers (*id_reg_8*), 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, part of the corresponding definitions are replicated from the **INVARIANT**:

$rgs8 \in id\_reg\_8 \rightarrow BYTE \land pc \in INSTRUCTION \land$
$sp \in BV16 \land ix \in BV16 \land iy \in BV16 \land$

$i\_ \in BYTE \wedge r\_ \in BYTE \wedge iff1 \in BIT \wedge iff2 \in BIT \wedge$
$im : (BIT \times BIT) \wedge \text{i\_o\_ports} \in BYTE \to BYTE$

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* ∈ *USHORT*
  THEN
  updateAddressMem (*ushort_to_bv16* (*nn*) , *rgs8* (*a0*)) ||
  pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END

The microcontroller model can specify 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 they can set many safety rules. However, the model can be very difficult to prove.

Several iterations were needed to provide the good library definitions as well as to fine-tune the model of the microcontroller instructions by factoring common functionalities into auxiliary definitions.

However, few proof commands[3] need to be used to prove most proof obligations. As there are many similar assembly instructions, some human-directed proofs, when replayed, could discharge other proof obligations. A good example is a set of 17 proof commands that quickly aided the verification of 99% (2295) of WD proofs. We also set up a proving environment consisting of networked computers to take advantage of the distribution facilities now provided in the B development environment. Finally, all of the 2926 proof obligations were proved using the tool support of the development environment.

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, in both works the B method has been used successfully to model the operational semantic.

[2] An expression is called "well-defined" (or unambiguous) if its definition assigns it a unique interpretation or value.
[3] The proof commands are steps that direct the prover to find the proof, and cannot introduce false hypothesis.

However the cost of modelling was still expensive and this paper quoted some techniques to lower the cost of modelling.

In general, the researchers employing the B method have focused on more abstract level of description of software. Considering low-level aspect, there has been previous work on modelling the Java Virtual Machine [3].

The main motivation of our research is the development of verified software up to the assembly level, which requires specifying the semantics of the underlying hardware. Thus, some aspects were not modelled in our work such as the execution time of the instructions. Also we did not consider the microarchitecture of the hardware as the scope of our work does not include hardware verification. However, there are many other specialized techniques to verify these questions.

7 Conclusions

This work has shown an approach to the formal modelling of the instruction set of microcontrollers using the B method. During the construction of this model, some ambiguities and errors were encountered in the official reference for Z80 microcontroller [6]. As the B notation has a syntax that is not too distant from that of imperative programming languages, such model could be used to improve the documentation used by assembler programmers. Besides, the formal notation used is analyzed by software that guarantees the correctness of typing, the well-definedness of expressions, in addition to safety properties of the microcontroller state.

Future works comprise the development of software with the B method from functional specification to assembly level, using the Z80 model presented in this paper. The mechanic compilation from B algorithmic constructs to assembly platform is also envisioned.

Acknowledges: This work received support from ANP (*Agência Nacional do Petróleo, Gás Natural e Biocombustíveis*) and CNPq (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*).

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 J-L. Boulanger and G. 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. L.; Moreira, A. M. and Medeiros Jr, V. G.. 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. http://www.zilog.com/docs/z80/um0080.pdf