Experiences in Modelling a Microcontroller Instruction Set Using B

Valério Medeiros Jr[1], David Déharbe[2]

[1] Federal Institute of Education, Science and Technology of Rio Grande do Norte, Natal RN 59015-000, Brazil
[2] 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 and several details about the representation of elements from microcontrollers. Several models were developed using the notation of the B method. They are used to develop a formally verified software up to the assembly level and allow the simulation of models. This simulation is able to guarantee the consistency between the execution of a software model and a real execution of software, it has wide variety of usages in industry and academia. This paper presents specifically the case of the Z80 platform and quote a theoretic case study important in tanks of the petroleum industry. This work is a contribution towards the extension of the B method to handle developments up to the assembly level code.
Keywords: Embedded Software, Simulation, B method and Verification

1 Introduction

The B method [1] supports the construction of safety systems models by generating proof obligation that must be verified to guarantee its correctness. So, an initial abstract model of the system requirements is defined and then it is refined up to 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. [6] proposed 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 presents a formal modell of the instruction set of the Z80 microcontroller [20]. This microcontroller was chosen because of several factors: it contains essential and common concepts in microcontrollers and until microprocessors; it has extensive documentation available; and it was widely used and is still in use. A first Z80 modell presented in [13] was verified completely, but it uses constructs that are not well supported to animate. This modelling was changed to support animation in ProB[11]. Basically, the new modelling[3] changed elements represented by infinite sets and adjusted the implications of this change. Using the responsibility division mechanisms 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 new Z80 model, a theoretic case study in petroleum production test system was developed using Z80 assembly language to analyse the code simulation and verification process.

The formal model of instruction set provides many benefits. The animation of Z80 model allows to simulate the execution assembly programs including support for interrupts and input

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

and output instructions. Other possible uses of a formal model of a microcontroller instruction set include documentation, the construction of simulators, and can 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. Two aspects of these formalization are of particular importance to us: provide a more solid documentation artifact for microcontrollers, build a reference model for a formal compilation approach in B.

1.1 Problems in formalizing instruction set

In general, the manuals of microcontrollers can be more suited for developers in assembly language. Many times, each instruction of microcontrollers is shown in its official manual using textual description, math description, examples, encoding, number of cycles and other information.Some times, the instruction descriptions have an own notation, some informations are organized in different sections of document and the textual description are not standardized. This does not allow the interpretation by parser. For example in official manual of the microcontroller Z80 [20], if the users want to know the action of an instruction and its effects on the flag register then he needs to read an informal textual description. So, this description can add some ambiguities or even mistakes. The official manual has also several problems that were identified over time [18]. The Z80 is a microcontroller that has been tested and used for many years. This intensive use and test facilitated to find several errors in the description of the instructions in its manual. Several of these problems have been reported in a technical report [18]. Some problems are inaccurate information, for example the description of the action of an instruction on the flag register, partial information and distributed on different pages of manual.

A solution to avoid ambiguity and inconsistencies is to specify formally the assembly instruction set, this creates a pattern of representation and validates properties of instructions. The formal model also restricts the definitions to correct typing and uses only well-defined expressions. Furthermore, the developer can use with B method different levels of abstraction and add more specific details in refined models.

This work is focused on the presentation of modelling of assembly instruction set, including elementary libraries to describe hardware aspects; and the modelling and simulation of a case study of an embedded verified software that can be used in tanks of the petroleum industry.

This paper is structured as it follows. Section 2 presents the elementary libraries and the modelling of some elements common to microcontrollers. Section 3 presents the B model of the Z80 instruction set. Section **??** explains the building and verifying assembly-level refinements. Related works are discussed in section **??**. Finally, the last section is devoted to the conclusions.

2 Model structure and basic components

We have developed a reusable set of basic definitions to model hardware concepts and data types concepts. The models contains definitions to represent the: registers, interruptions, input and output ports, memory and instructions. These definitions are grouped into separated development projects and are available as libraries.

Thus the workspace is composed of modules group and libraries:

- **Power**: it has the basics definitions to help the theorems prover about simple calculus of power, especially power of two. It contains the definition of power function and it has constants, that represent all required results of power function.

– **Hardware Library**: it has the definition of bit vectors with size 8 and 16, basic functions to manipulate bit vectors and important assertions. This is presented in sections 2.1, 2.2 and 2.3.
– **Types Library**: it has the definition of naturals and integers represented with 8 bits and 16 bits, basic conversion functions between the types and bit vectors and important assertions. This is presented in section 2.4.
– **Platform**: it has the definition functions of arithmetic logic unit, memory unit, registers and assembly instructions. This is presented in section 3.

The corresponding dependency diagram is depicted in Figure 1; information specific to each project is presented in the following.

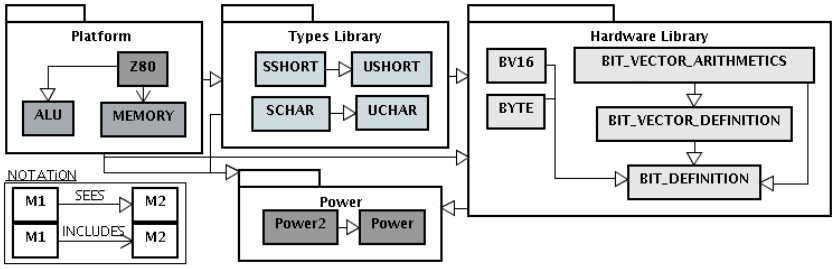

Fig. 1. Dependency diagram of the Z80 model.

2.1 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 Boolean to bit.

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:

$bit\_not \in BIT \rightarrow BIT \wedge bit\_not = \{0 \mapsto 1, 1 \mapsto 0\}$

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 a binary 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 commutativity and associativity:

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

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 from this paper.

Finally, the conversion from Boolean to bit 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.

2.2 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: $BIT\_VECTOR = \mathrm{seq}\,1(BIT)$. The BIT_VECTOR is an infinite set and it hinders the animation in ProB. To avoid this obstruction, the specialized types ($BYTE$ and $BV16$) cannot reference directly BIT_VECTOR.

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:

$bv\_set \in BIT\_VECTOR \times \mathcal{N} \rightarrow BIT\_VECTOR \wedge bv\_set =$

$\lambda\, v, n.(v \in BIT\_VECTOR \wedge n \in \mathcal{N} \wedge n < size(v) \mid v \leftarrow\!\!+ \{n+1 \mapsto 1\})$

The function *bv_catenate* takes as parameters two bit vectors v and w, and returns the result of the concatenation of v and w, such that v constitutes the most significant part of the result.

$bv\_catenate \in BIT\_VECTOR \times BIT\_VECTOR \rightarrow BIT\_VECTOR \wedge$

$bv\_catenate = \lambda\ \mathrm{v}, w\ .\ (v \in BIT\_VECTOR \wedge w \in BIT\_VECTOR \mid v\,\hat{}\,w)$

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:

$bv\_not \in BIT\_VECTOR \rightarrow BIT\_VECTOR \wedge$

$bv\_not = \lambda\, v.(v \in BIT\_VECTOR \mid \lambda\, i.(1..size(v)) \mid bit\_not(v(i))) \wedge$

$bv\_and \in BIT\_VECTOR \times BIT\_VECTOR \rightarrow BIT\_VECTOR \wedge$

$bv\_and = \lambda\, v_1, v_2.(v_1 \in BIT\_VECTOR \wedge v_2 \in BIT\_VECTOR \wedge$

$size(v_1) = size(v_2) \mid \lambda\, i.(1..size(v_1)) \mid bit\_and(v_1(i), v_2(i)))$

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.

2.3 Modelling bytes and bit vectors of length 16

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

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

$PHYS\_BYTE\_INDEX = 0\ ..\ (BYTE\_WIDTH\text{-}1) \wedge$

$BYTE = (BYTE\_INDEX \rightarrow BIT) \wedge \mathbf{card}(BYTE) = 2^8 \wedge$

$\forall\ (b1).(b1 \in BYTE \Rightarrow \mathbf{size}(b1) = BYTE\_WIDTH \wedge b1 \in \mathbf{seq1}(BIT))$

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 as a specialized type of 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.

2.4 Bit vector arithmetic

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:

$bv\_to\_nat \in BIT\_VECTOR \rightarrow \mathcal{N} \wedge$

$bv\_to\_nat = \lambda\, v.(v \in BIT\_VECTOR \mid \sum i.(i \in \mathrm{dom}(v).v(i) \times 2^{i-1}))$

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

2.5 Basics data types

The instruction set architecture usually have common bit vector type and integer type. In table 1, the first three are placed in the hardware library and the last four types are placed in the integer types library. Each type module has functions to manipulate and convert its data. There are seven common basic data types represented by modules.

Table 1. Details of basic data types

Type Name	*Range*	*Physical Size*	*Associated Proof Obligations*
BIT	0..1	1 bit	118
BYTE	–	1 bytes	153
BV16	–	2 bytes	75
UCHAR	0..255	1 byte	30
SCHAR	-128..127	1 byte	26
USHORT	0..65.535	2 byte	62
SSHORT	-32.768..32.767	2 byte	58

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 arithmetic 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.

We also created the following lemmas:

$\forall(val).(val \in UCHAR | byte\_uchar(uchar\_byte(val)) = val) \wedge$

$\forall(by).(by \in BYTE | uchar\_byte(byte\_uchar(by)) = by)$

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

Nearly 50% of all these proof obligations shown in the table 1 are verified almost automatically. Although many others proof obligations are difficult to verify, because these check the type of functions with non-linear arithmetic is a difficult task.

2.6 Verification Process in Hardware Library and Types Library

This section presents the proof process of modules from basic data types. By default, AtelierB [4] does not verify the type of defined functions in the clause $PROPERTIES$ from Machine. For example, the function inc, defined as $inc \in \{0,1\} \rightarrow \{0,1\} \wedge inc = \lambda(x).(x \in \{0,1\}|x+1)$, does not create a proof obligation to check the consistency of the definition. However, this function is not well typed because $inc(1) = 2$ and $2 \notin \{0,1\}$. To guarantee the type of functions, it must be declared separately: the definition of the function in $PROPERTIES$ clause and its type in the $ASSERTIONS$ clause.

Another solution is to check with ProB [11], as it is able to verify the type of functions, but its support is more suitable for finite functions. However, these functions use the concept of sequence, that is an infinite set. So a better solution is to formalize rules to check these properties in AtelierB.

These rules are assumed true hypotheses for the conversion functions and they are used in interactive prover of AtelierB. They were created because it was not possible to build proofs using the base of rules from the free distribution of AtelierB. Furthermore, AtelierB's prover cannot simplify easily arithmetic expressions.

The hardware library has many conversion functions with non-linear arithmetic, for example, Bit_vector to Natural, Natural to Bit_vector, Integer to Bit_vector, Bit_vector to Integer. These functions are organized in modules and each module contains auxiliary functions for a data type. The modules are represented by gray rectangles in Figure 2.

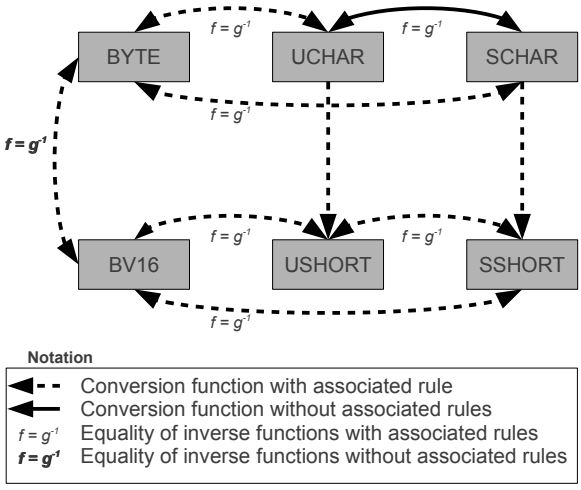

Fig. 2. Conversion functions

We needed to verify the type of these conversion functions.Basically, these functions must keep two properties: to be bijective and equal to the inverse of the dual function. This is not easy to verify it using AtelierB. Only these more simple proof obligations shown below were verified automatically with the AtelierB:

1. $(uchar\_schar \in UCHAR \rightarrowtail\!\!\!\!\!\rightarrow SCHAR)$
2. $(schar\_uchar \in SCHAR \rightarrowtail\!\!\!\!\!\rightarrow UCHAR)$

3. $(byte\_bv16 = bv16\_byte^{-1})$

However, there are similar proof obligations that needed to create associated rules, in general, they are almost identical then only three are illustrated here:

1. $(byte\_uchar \in BYTE \rightarrowtail\!\!\!\!\!\!\rightarrow UCHAR)$
   $\lambda\ v0.(v0 \in 1\ldots 8 \rightarrow \{0,1\} \mid 128 \times v0(8)+64 \times v0(7)+32 \times v0(6)+16 \times v0(5)+8 \times v0(4)+4 \times v0(3)+2 \times v0(2)+1 \times v0(1)) \in (1\ldots 8 \rightarrow \{0,1\}) \rightarrowtail\!\!\!\!\!\!\rightarrow 0\ldots 255$

2. $(uchar\_byte \in UCHAR \rightarrowtail\!\!\!\!\!\!\rightarrow BYTE)$
   $\lambda\ v0.(v0 \in 0\ldots 255 \mid [v0\ \mathbf{mod}\ 2\ /\ 1\ ,v0\ \mathbf{mod}\ 4\ /\ 2,v0\ \mathbf{mod}\ 8\ /\ 4,v0\ \mathbf{mod}\ 16\ /\ 8,v0\ \mathbf{mod}\ 32\ /\ 16,v0\ \mathbf{mod}\ 64\ /\ 32,v0\ \mathbf{mod}\ 128\ /\ 64,v0\ \mathbf{mod}\ 256\ /\ 128]) \in 0\ldots 255 \rightarrowtail\!\!\!\!\!\!\rightarrow (1\ldots 8 \rightarrow \{0,1\})$

3. $(\ byte\_uchar = uchar\_byte^{-1}\ )$
   $\lambda\ v0.(v0 \in 1\ldots 8 \rightarrow \{0,1\} \mid 128 \times v0(8)+64 \times v0(7)+32 \times v0(6)+16 \times v0(5)+8 \times v0(4)+4 \times v0(3)+2 \times v0(2)+1 \times v0(1)) =$
   $(\ \lambda\ v0.(v0 \in 0\ldots 255 \mid [v0\ \mathbf{mod}\ 2\ /\ 1,v0\ \mathbf{mod}\ 4\ /\ 2,v0\ \mathbf{mod}\ 8\ /\ 4,v0\ \mathbf{mod}\ 16\ /\ 8,v0\ \mathbf{mod}\ 32\ /\ 16,v0\ \mathbf{mod}\ 64\ /\ 32,v0\ \mathbf{mod}\ 128\ /\ 64,v0\ \mathbf{mod}\ 256\ /\ 128]))^{-1}$

Other four rules were used to demonstrated that the functions are finite: $byte\_uchar \in \mathcal{F}(byte\_uchar)$; $uchar\_byte \in \mathcal{F}(uchar\_byte)$; $ushort\_bv16 \in \mathcal{F}(ushort\_bv16)$; $bv16\_ushort \in \mathcal{F}(bv16\_ushort)$. The expansion of the first rule is:

1. $(\ byte\_uchar \in \mathcal{F}\ (byte\_uchar)\ )$
   $\lambda\ v0.(v0 \in 1\ldots 8 \rightarrow \{0,1\} \mid 128 \times v0(8)+64 \times v0(7)+32 \times v0(6)+16 \times v0(5)+8 \times v0(4)+4 \times v0(3)+2 \times v0(2)+1 \times v0(1)) \in \mathcal{F}\ (\ \lambda\ v0.(v0 \in 1\ldots 8 \rightarrow \{0,1\} \mid 128 \times v0(8)+64 \times v0(7)+32 \times v0(6)+16 \times v0(5)+8 \times v0(4)+4 \times v0(3)+2 \times v0(2)+1 \times v0(1)))$

So there are in total twenty four rules related to binary arithmetic.These rules are used to facilitate the manipulation of theorem prover with binary arithmetic developed by [3]. They guarantee important properties that are demonstrated in [8,14]. The others properties with associated rules are very similar, they have just small differences, for example, the size of bit vectors.

3 A B model of the Z80 instruction set

The *Z80* is a CISC microcontroller developed by *Zilog* [20]. The Z80 is composed by different elements and a simplified internal organization is shown in the Figure 3. This figure has some important elements of Z80 CPU: ALU, registers of 8 and 16 bits and input/output ports.

The Z80 has 158 different instructions, including all the 78 from Intel 8080 microprocessor, and all of them were formally specified in B. 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. Each category of instruction has different elements of specification.

This section shows the elements that make up the different types of instructions. The main elements are specified in the microcontroller module Z80 and parts of it are presented below. Basically, the state of microcontroller is formed by data from program counter, ports, registers, etc. The transitions between the states are actioned by execution of instruction or an external action. Groups of registers are represented by the variables of the specification states that appear

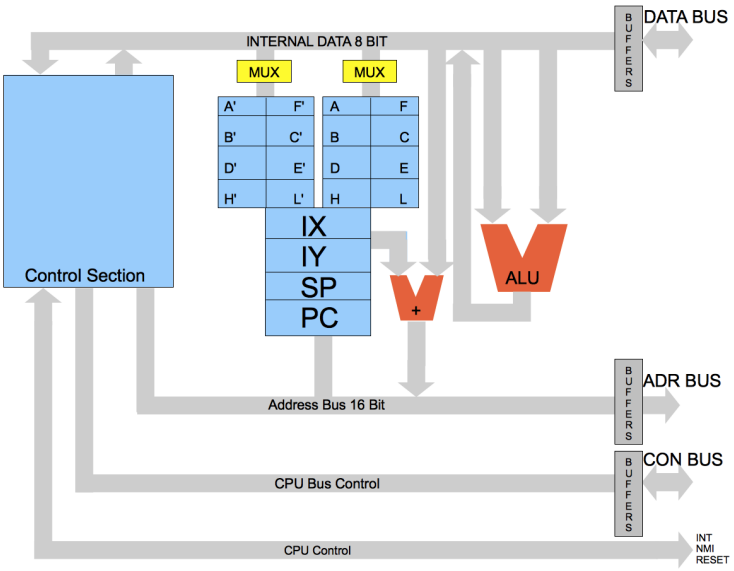

Fig. 3. Simplified internal organization of Z80 CPU.

in clause *VARIABLES*. The declaration of valid states of variables is in the *INVARIANT* clause and the initial state is defined in the *INITIALISATION* clause. The assembly instructions are defined by the clause *OPERATIONS* and, in general, each instruction has three actions: update the program counter, update the flag register and its main effect. General functions that can be used in others microcontroller models are defined in the modules of data types. Specific functions of the microcontroller are defined in the ALU (arithmetic logic unit) module.

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

3.1 Modelling registers

The Z80 CPU 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 represented by *INVARIANT*.

INVARIANT
$rgs8 \in id\_reg\_8 \rightarrow BYTE \wedge$

$pc \in INSTRUCTION \land sp \in BV16 \land ix \in BV16 \land iy \in BV16 \land$
$i\_ \in BYTE \land r\_ \in BYTE \land$
$iff1 \in BIT \land \quad iff2 \in BIT \land$
$im \in (BIT \times BIT) \land$
$i\_o\_ports \in BYTE \rightarrow BYTE$

The internal registers contain 176 bits of read/write memory that are represented by identifiers used as parameters in the instructions . It includes two sets of six general purpose registers which may be used individually as 8-bit registers or as 16-bit 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 are called main register set (*a0,b0,c0,d0,e0,f0,h0,l0*) and alternate register set (*a_0,b_0,c_0,d_0,e_0,f_0,h_0,l_0*). These registers can be accessed in pairs, forming 16-bits, resulting in another set of identifiers of 16-bits registers, named *id_reg16*.

SETS

$id\_reg\_8 = \{\ a0\ ,\ f0\ ,\ f\_0\ ,\ a\_0\ ,$
$b0\ ,\ c0\ ,\ b\_0\ ,\ c\_0,$
$d0\ ,\ e0\ ,\ d\_0\ ,\ e\_0,$
$h0\ ,\ l0\ ,\ h\_0\ ,\ l\_0\};$
$id\_reg\_16 = \{\ BC\ ,\ DE\ ,\ HL\ ,\ SP\ ,\ AF\ \}$

The main working register of Z80 is the accumulator (*rgs8(a0)*) used for arithmetic/logic, input/output and loading/storing operations.

Flag register: another important element is the "f" register (*rgs8(f0)*), that is used as a flag register. This register uses only six bits to represent the execution result status of each instruction. According to the official manual the bits 3 and 5 are not used and the others bits have the follow meaning: *bv_get(rgs8(f0),0)* - the carry bit; *bv_get(rgs8(f0),1)* - the add/subtract bit; *bv_get(rgs8(f0),2)* - the parity or overflow bit; *bv_get(rgs8(f0),4)* - the half carry bit; *bv_get(rgs8(f0),6)* - the zero bit; and *bv_get(rgs8(f0),7)* - the sign bit. These bits also can be used to specify security properties for microcontroller programs.

Assuring the absence of overflow: *To assure that an overflow does not happen, the developer can add this expression (bv_get(rgs8(f0),0) $\neq$ 1 $\land$ bv_get(rgs8(f0),2) $\neq$ 1) in the invariant.*

3.2 Manipulation data functions from Z80

There are some specific functions from Z80 to manipulate the data. In addressing mode, the function *bv_ireg_plus_d* receives the value of a register (*ix* or *iy*) used as a base to point to a region in memory from which data is to be stored or retrieved. An additional byte is included in indexed instructions to specify an offset from this base. This displacement is specified as a two's complement signed integer. This function is defined as:

$bv\_ireg\_plus\_d : (BV16 \times SCHAR \rightarrow BV16) \land$
$bv\_ireg\_plus\_d = \lambda\ (\ ix\_iy\ ,\ offset\ )\ .\ (\ ix\_iy \in BV16 \land offset \in SCHAR$
$\quad \mid ushort\_bv16\ (\ (bv16\_ushort\ (\ ix\_iy\ ) + offset\ )\ \mathbf{mod}\ 2^{16}\ )\ )$

Another derived function is *bv_(ireg_plus_d)*, this returns the value in the memory address returned by *bv_ireg_plus_d* function and its definition is similar.

There is a specific function to refresh the flag register, it is named *update_reg_flag*. It is typed and defined as follow:$update\_flag\_reg \in (BIT \times BIT \times BIT \times BIT \times BIT \times BIT) \rightarrow (\{f0\} \times BYTE)$.

$update\_flag\_reg = \lambda\ (s7\ ,\ z6\ ,h4\ ,pv2\ ,n1\ ,c0\ )\ .\ (\ s7 \in BIT \land z6 \in BIT \land h4 \in BIT \land pv2 \in BIT \land n1 \in BIT \land c0 \in BIT \quad \mid (\ f0 \mapsto [c0,\ n1,\ pv2,\ 1,\ h4,\ 1,\ z6,\ s7])\ )$

3.3 Program, stack and data memory

The Z80 uses a unique memory for storing program instructions, data stack and general-purpose data. The memory has 16-bit addresses and each address holds a byte. Thus, the data from the memory module is very simple:

INVARIANT

$mem \in BV16 \rightarrow BYTE$

In general, the instructions can access all memory address, but it is dangerous. The user may mistakenly access and change data in memory. For added security, it is important that the program instructions has limited access by region. Thus the designer can specify address regions to restrict the access from instructions. The address regions can be specified using constants ($PROGRAM\_R\_ADR$,$DATA\_R\_ADR$,$STACK\_R\_ADR$), these define respectively a range of restricted address for: programs instructions, general purpose data and data stack.

$PROGRAM\_R\_ADR = 0..16384 \wedge$
$DATA\_R\_ADR = 16385..49151 \wedge$
$STACK\_R\_ADR = 49152..65535$

Assuring the absence of overlapping of address regions: *To assure that address regions are well defined, then the designer must to verify the expression:*

$PROGRAM\_R\_ADR \cap DATA\_R\_ADR \cap STACK\_R\_ADR = \{\}$

Preserving the consistency of the memory: *In general, the access to some address regions is dangerous. Then, each instruction has a specific pre-condition that verify if the new address memory, that will be updated, is member of its region. For example, the PUSH program instruction allows write only in the region of stack (STACK_R_ADR).*

3.4 Arithmetic logic unit

There are many functions in the module ALU. In general, the definition of these functions use basic definitions or previously defined functions. For example, the function *half8UCHAR* is used to get the half part of $UCHAR$ value. It is important to know the half carry and it is used in the function *add8UCHAR*.

$half8UCHAR \in UCHAR \rightarrow UCHAR \wedge$
$half8UCHAR = \lambda\ (ww).(ww \in UCHAR \mid ww\ \mathbf{mod}\ 2^4)$

The function *add8UCHAR* receives a bit carry and two $UCHAR$ values and returns respectively the sum, the sign bit, the carry bit, the half carry bit and the zero bit. It is typed as follows: $add8UCHAR : (BIT \times UCHAR \times UCHAR) \rightarrow (UCHAR \times BIT \times BIT \times BIT \times BIT)$ and its definition is:

$add8UCHAR = \lambda\ (carry, w1, w2).$
$(carry \in BIT \wedge w1 \in UCHAR \wedge w2 \in UCHAR \mid$
$\quad (((carry + w1 + w2\ )\ \mathbf{mod}\ 2^8\ ),$
$\quad bool\_bit(\ carry + uchar\_schar(w1) + uchar\_schar\ (w2) < 0),$
$\quad bool\_bit(\ carry + w1 + w2 > UCHAR\_MAX),$
$\quad bool\_bit(\ carry + half8UCHAR(w1) + half8UCHAR(\ w2) \geq 2^4),$
$\quad bool\_bit(\ (\ (carry + w1 + w2\ )\ \mathbf{mod}\ 2^8\ ) = 0))\ \ )$

A related function to subtract operation is *subtract8UCHAR*. There are the same functions for the $SCHAR$ type, they are respectively *add8SCHAR* and *subtract8SCHAR*, all these functions are of 8 bits ($BYTE$) and defined similarly. In the same way, the arithmetic functions for 16 bits

(*BV16*) are defined. The module ALU has several others functions, for example:*instruction_next* $\in$ *USHORT* $\rightarrow$ *USHORT* - It receives the actual value from program counter register (*pc*) and returns its increment; and *is_negative* $\in$ *BYTE* $\rightarrow$ *BIT* - It returns the most significant bit, in other words, the signal bit. As the logic functions that are defined in the *BYTE* and *BV16* module are included in the *ALU* module, they can be seen and used directly in the *ALU* and *Z80* modules.

3.5 Modelling the actions and instructions

Each instruction is represented by a B operation in the module Z80. The main module (*Z80*) has three categories of operations: a category represents the instructions of microcontrollers, a second category represents the input and output of data (shown in 3.6) and the last category represents the external actions (shown in 3.7). A simple example of instruction category is *LD_(nn)_A* [4] shown below. The pre-defined functions are necessary many times to model the instructions, these functions facilitate the construction of instruction set model. By default, all parameters in operations are either predefined elements in the model or integers values in decimal representation. This instruction use the *updateAddressMem* operation from *Memory* module and it receives a address memory and its new memory value. It also increments the program counter (*pc*) and update the refresh register (*r_*). The other instructions have a similar structure.

LD_9nn0_A (*nn*) =
  PRE *nn* $\in$ *USHORT* $\wedge$ *nn* $\in$ *DATA_R_ADR*
  THEN
  updateAddressMem (*ushort_bv16* (*nn*) , *rgs8* (*a0*)) ||
  pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END

3.6 Modelling the input/output instructions

The Z80 has an extensive set of input and output (I/O) instructions and 256 ports for devices. This model can transfer data blocks between the I/O devices and any of the internal registers or memory address.

The *IN_r(C)* [4] instruction is represented by the following B operation. It receives an register identifier "*rr*" and, it stores the value from "rr" in the *C* port address. Besides, it increments the program counter and updates the flag registers.

IN_r_9C0 (*rr*) =
PRE *rr* $\in$ *id_reg_8* $\wedge$ *rr* $\neq$ *f0* **THEN**
  ANY
    negative , *zero* , *half_carry* , *pv* , *add_sub* , *carry*
  WHERE
    negative $\in$ *BIT* $\wedge$ *zero* $\in$ *BIT* $\wedge$ *half_carry* $\in$ *BIT* $\wedge$ *pv* $\in$ *BIT* $\wedge$
    add_sub $\in$ *BIT* $\wedge$ *carry* $\in$ *BIT* $\wedge$
    negative = *is_negative* (*io_ports* (*rgs8* (*c0*))) $\wedge$
    zero = *is_zero* (*io_ports* (*rgs8* (*c0*))) $\wedge$
    half_carry = 0 $\wedge$
    pv = *parity_even* (*io_ports* (*rgs8* (*c0*))) $\wedge$
    add_sub = 0 $\wedge$

[4] The tools B does not allow to use parentheses in identifiers, and the characters {"(",")"} are replaced respectively by {"9","0"} in the actual specification.

$carry = z\_c$
THEN
$rgs8 := rgs8 \Leftarrow \{ (rr \mapsto io\_ports (rgs8 (c0))) ,$
$update\_flag\_reg(negative,zero,half\_carry,pv,add\_sub,carry)\}||$
$pc := instruction\_next (pc) || r\_ := update\_refresh\_reg(r\_)$
END

3.7 Modelling the external actions

The external actions change the state of the microcontroller, for example, refreshing the I/O ports and interruptions request. The external actions are also modelled by operations and they are named with the prefix "*ext_*" and followed by the name of action. There are four external actions: *ext_update_io_ports*, ext_NMI and ext_INT, *ext_Reset*. The *ext_update_io_ports* just change the state of I/O port, see.

ext_update_io_ports(*address*,*value*)=
PRE $address \in UCHAR \wedge \quad value \in SCHAR$ **THEN**
$io\_ports (uchar\_byte (address)) := schar\_byte (value)$
END

The other external actions are related to interruptions. The interruptions allow that the devices suspend a routine from CPU and start another service routine. This service routine can exchange data or signals between CPU and external devices. When a routine is finished, then the CPU comes back to the last routine that was interrupted.

For the interrupts, the following elements are important: the interrupt flip-flops (*iff1* and *iff2*), the types of interrupts (maskable and non-maskable), the interrupt mode (set with the *IM 0*, *IM 1*, *IM 2* instructions) and the $i\_$ register.

Flip-flops *iff1* and *iff2* control the maskable interrupts (*INT*). When *iff1* is set, the interrupt is enabled, otherwise it is disabled; *iff2* is used only as backup for *iff1*. The instructions *EI* and *DI* respectively enable and disable the maskable interrupts, setting *iff1* to 1 and 0.

The interruptions and the *reset* action can change the state of program counter. Theses actions are modelled by B operations and its main effects are shown below[5].

NMI - Non-maskable interrupts cannot be disabled by the programmer. Then, when a device makes a request, *sp* is pushed, *pc* receives $66H$ (102 in decimal), *iff1* is reset , *iff2* stores *iff1* and the refresh register is updated.

updateStack({ $(sp\_minus\_two \mapsto pc\_low)$,$(sp\_minus\_one \mapsto pc\_high)$ })||
$sp := sp\_minus\_two || pc := 102 ||$ *iff1*:=0 || *iff2*:= *iff1* ||
$r\_ := update\_refresh\_reg(r\_)$

INT - Maskable Interrupt is usually reserved for important functions that can be enabled and disabled by the programmer. When a maskable interrupt action happens, both *iff1* and *iff2* are cleared, disabling the interrupts, *sp* is pushed, the refresh register is updated and the other effects depend on the interrupt mode register (*im*).

- The mode 0 is compatible with 8080 and this mode is selected when $im = (0 \mapsto 0)$. When a non-maskable interrupt happens, the CPU fetches an instruction of one byte from an external device, usually an RST instruction, and the CPU executes it. The instruction code is received from an external device by data bus and it is represented by integer parameter called *byte_bus* .

[5] Some definitions of constants: *sp_minus_two* holds the value of stack pointer minus 2, *sp_minus_one* is the value of stack pointer minus 1, *pc_high* holds the most significant 8 bits and *pc_low* holds the least significant 8 bits.

- The mode 1 is the simplest and this mode is selected when $im = (0 \mapsto 1)$. Simply, when a non-maskable interruption happens, the program counter receives $38H$ (56 in decimal).
- The mode 2 is the most flexible and this mode is selected when $im = (1 \mapsto 1)$. When a non-maskable interruption happens, an indirect call can be made to any address memory. The program counter receives a bit vector of size 16 composed with two parts: the most significant part of the $i\_$ register and the least significant part of the *byte_bus* with the last bit cleared.

The essential part of maskable interrupt is shown below, where *byte_bus* is a parameter of the *INT* operation:

IF $im = (0 \mapsto 0)$ **THEN**
  IF *byte_bus* $\in$ *opcodes_RST_instruction*
  THEN
    pc := *byte_bus* - 199 ||
    updateStack({ *stack*(*sp_minus_one*) $\mapsto$ *pc_low*,
    stack(*sp_minus_two*) $\mapsto$ *pc_high* }) ||
    sp := *sp_minus_two* || $r\_$:= *update_refresh_reg*($r\_$)
  ELSIF *byte_bus* = *opcode_..._instruction*
    ...
  END
ELSIF $im = (0 \mapsto 1)$ **THEN**
  pc := 56 ||
  updateStack({ *stack*(*sp_minus_one*) $\mapsto$ *pc_low*,
  stack(*sp_minus_two*) $\mapsto$ *pc_high* }) ||
  sp := *sp_minus_two* || $r\_$:= *update_refresh_reg*($r\_$)
ELSIF $im = (1 \mapsto 1)$ **THEN**
  pc := *bv16_ushort*(*byte_bv16*($i\_$,*bv_clear*(*rotateleft*(*uchar_byte*(*byte_bus*)),0)))||
  updateStack({ *stack*(*sp_minus_one*) $\mapsto$ *pc_low*,
  stack(*sp_minus_two*) $\mapsto$ *pc_high* }) ||
  sp := *sp_minus_two* || $r\_$:= *update_refresh_reg*($r\_$)
END

RESET - This just resets the registers related to the interruptions.
iff1 :=0 || *iff2*:=0 || *im*:= (0 $\mapsto$ 0) || *pc*:=0 || $i\_$:= [0,0,0,0,0,0,0,0] ||
rgs8 := *rgs8* $\leftarrow$ { (*a0* $\mapsto$ [0,0,0,0,0,0,0,0] , (*f0* $\mapsto$ [0,0,0,0,0,0,0,0] } ||
$r\_$:= [0,0,0,0,0,0,0,0] || *sp* := [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

The modelling of instruction set, interruptions and input/output ports become the animation interesting. The animation in ProB version is not so fast and practical as a simulation, for example in [16], because ProB works manipulating logic and math concepts that can have infinite sets and non determinism, then ProB requires more processing, specially when it works analysing complex expressions or very large expressions. However, a user of ProB with this modelling can: execute manually a sequence of Z80's instructions that represent a program and view the visited states of ports, memory and registers, it is important to debug the effect action on the state space computed; create a tracing of the states; analyse constants, functions and expressions; and search possible violations.

The Z80 formal model provides many benefits, because of the verification of generated proof obligations guarantees: the correct use of data types, the developed security properties and the well-defined of all the expressions. Furthermore, the designer has a big flexibility to create new

and specific security properties, this is very useful to adjust the verification in accordance with the requirements. Moreover, this example of model could replace or improve the used documentation for users and assembly programmers. Z80 model is also useful to develop verified software up to the assembly level. There are many benefits when a formal model is developed, but to verify the model is not an easy task, the first version of Z80 model have already verified completely,but the new Z80 model with support to animation in ProB has not been verified completely, because the changes in Z80 model are recent and the verification time is long.

A case study using the first version of Z80 model was also created and verified since the first abstract B model up to B assembly model. This case study is an object of the pilot project developed to analyse a petroleum production test system in each oil field. Its modelling was developed in according to [15] and its B assembly model can be executed manually in ProB. The modelling of instruction set and the B assembly model allow to simulate the execution of microcontroller and analyse several important properties.

4 Related works

There are in the literature several approaches [2,7,10,17] to model hardware and the virtual machines using B. In these works, the B has been used successfully to model the operational semantic. However the cost of modelling is still expensive.

These recent works [12,19] have a similar approach using Event B. However, our work use the B method, that seems more appropriated to software development, because it has implementable language defined, called B0, and tools to convert the models to a programming language.

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 micro architecture of the hardware as the scope of our work does not include hardware verification.

5 Conclusions

This work has shown an approach to the formal modelling of the instruction set of microcontroller using the B method. During the construction of this model, some problems were found in the official reference for Z80 microcontroller [20]. The designer fixed the found problems in documentation and also developed a case study: a verified embedded software. This case study was interesting to analyse the technique of formal verification up to assembly level.

The next works quoted are directly related to the objective this paper. The first work [5] shows the developed metodology to verify software up to assembly level using B. A second work[6] presents more details about the verification approach and a small example software verified up to assembly level in three different platforms. The last work [13] shows a general view of the first version of Z80 model and techniques used in verification process.

These works created a important step to software verification up to assembly language, they show the actuals difficulties and suggest improvements in the B tools.

Future works comprise the development of software with the B method from functional specification to assembly level, using the Z80 model presented in this work. The mechanic compilation from B algorithmic constructs to assembly platform is also envisioned. Finally, another important activity is develop a formal model of a platform used in LLVM Compiler [9].

References

1. J. R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1 edition, 1996.
2. Ammar Aljer, Philippe Devienne, Sophie Tison, Jean-Louis Boulanger, and Georges Mariano. Bhdl: Circuit design in b. In *ACSD*, pages 241–242. IEEE Computer Society, 2003.
3. Gottfried Wilhelm Leibniz By Karl Immanuel Gerhardt. Leibnizens mathematische schriften. *G. H. Pertz*, 1859. Also available as `http://www.archive.org/details/leibnizensmathe06leibgoogl`.
4. Clearsy. Atelier B web site.
   Disponível em: http://www.atelierb.eu, 2009. Acesso em: 04 abril 2009.
5. B. P. Dantas, D. Déharbe, S.S.L. Galvão, A. Martins, and V. G. Medeiros. Proposta e avaliação de uma abordagem de desenvolvimento de software fidedigno por construção com o método B. In *Seminário Integrado de Software e Hardware*, Belém - PA, 2008. XXXV SEMISH.
6. B. P. Dantas, David Déharbe, S. L. Galvão, A. M. Moreira, and V. G. Medeiros Jr. Applying the B method to take on the grand challenge of verified compilation. In *Brazilian Symposium on Formal Methods*, Salvador - BA, 2008. SBMF.
7. Neil Evans and Neil Grant. Towards the formal verification of a java processor in event-b. *Electronic Notes in Theoretical Computer Science.*, 201:45–67, 2008.
8. Emeritus James L. Hein. *Discrete Structures, Logic, and Computability.* Portland State University, 2010.
9. Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.
10. Michael Leuschel. Towards demonstrably correct compilation of java byte code. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelain, editors, *FMCO*, volume 5751 of *Lecture Notes in Computer Science*, pages 119–138. Springer, 2008.
11. Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874, Pisa, Italy, 2003. Springer.
12. Marc and Benveniste. On using b in the design of secure micro-controllers: An experience report. *Electronic Notes in Theoretical Computer Science*, 280(0):3 – 22, 2011. Proceedings of the B 2011 Workshop, a satellite event of the 17th International Symposium on Formal Methods (FM 2011).
13. V. G. Medeiros Jr. and David Déharbe. Formal construction of a microcontroller instruction set model using b. In *Brazilian Symposium on Formal Methods*, Gramado - RS, 2009. SBMF.
14. Umesh Vazirani Sanjoy Dasgupta, Christos Papadimitriou. *Algorithms.* Mc Graw Hill, 2006. Also available as `http://www.cs.berkeley.edu/~vazirani/algorithms/all.pdf`.
15. Paulo Sérgio Silva. Automação da drenagem no teste de produção convencional em tanque cilíndrico. Master's thesis, UFRN-PPgEEC, 2008.
16. Vladimir Soso. Z80 simulator ide. on-line, 2002. Available at:http://www.oshonsoft.com Acessado em:18 abr 2009.
17. Stephen Wright. Using event B to create a virtual machine instruction set architecture. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2008.
18. Sean Young. *The Undocumented Z80 Documented*, November 2003. Available at: http://www.myquest.nl/z80undocumented/z80-documented.pdf. Acessed in:April 18, 2007.
19. Fangfang Yuan, Stephen Wright, Kerstin Eder, and David May. Managing complexity through abstraction: A refinement-based approach to formalize instruction set architectures. In Shengchao Qin and Zongyan Qiu, editors, *ICFEM*, volume 6991 of *Lecture Notes in Computer Science*, pages 585–600. Springer, 2011.
20. Zilog. *Z80 Family CPU User Manual.* ZiLOG Worldwide Headquarters, 910 E. Hamilton Avenue, 2001. Available at:http://www.zilog.com/docs/z80/um0080.pdf. Acessado em:18 abr 2007.