Experiences in Modelling of 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. The paper [6,9] has 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 presents a new version of the formal modelling [9] of the instruction set of the Z80 microcontroller [26]. The modelling [9] was verified completely, but it cannot be animated. This modelling was changed to support animation in ProB[13]. Basically, the new modelling[3] changed elements represented by infinite sets and adjusted the implications of this change. 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 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.

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

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

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 artefact 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 [26], if the users want to know the action of an instruction and its effects on the flag register then he needs to reader 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 [24]. 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 [24]. 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 simple example of error is in some official documentation saying that a group of instructions leaves the *CF* flag unaffected and the *NF* flag is not always set, but may also be reset [24]. This example can cause serious problems for programmers. This error and others 7 different errors were previously identified by [24]. So to help the construction of Z80 B model, other references [24,20] were also consulted.

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 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 **??** explains the building and verifying assembly-level refinements. Related works are discussed in section 5. 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) for the formal specification of functional requirements 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 [21].

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 among 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 its initial states are valid, and 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** clause declares the name of the variables that compose the state of the machine. Next, the **INVARIANT** clause defines the valid states, specifying the type and other restrictions on the variables. The **INITIALISATION** specifies the initial states. Finally, operations correspond to the transitions among states of the machine.

```
MACHINE    micro              INITIALISATION pc := 0
SEES       TYPES, ALU         OPERATIONS
INCLUDES   MEMORY             JMP( jump ) =
VARIABLES  pc                   PRE jump ∈ INSTRUCTION
INVARIANT                       THEN pc := jump
  pc ∈ INSTRUCTION              END
                                END
```

Fig. 1. A very basic B machine.

3 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 has constants, that represents all results of power function needed, and the definition 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 3.1, 3.2 and 3.3.
- **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 3.4.
- **Platform**: it has the definition functions of arithmetic logic unit, memory unit, registers and assembly instructions. This is presented in section 4.

The corresponding dependency diagram is depicted in Figure 2; information specific to each project is presented in the following.

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 Boolean to bit.

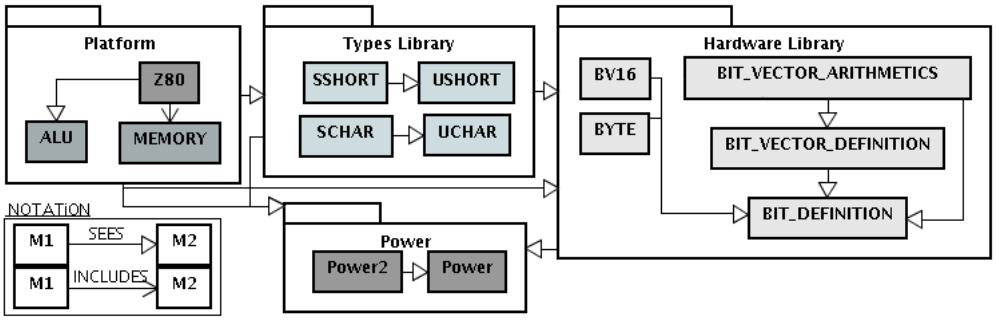

Fig. 2. Dependency diagram of the Z80 model.

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:

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

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 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 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.

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

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:

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

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:

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

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 are a common entity in hardware design. We provide the following definitions:

$BYTE\_WIDTH = 8 \wedge BYTE\_INDEX = 1\ ..\ \mathrm{BYTE\_WIDTH}\ \wedge$
$PHYS\_BYTE\_INDEX = 0\ ..\ (BYTE\_WIDTH\text{-}1) \quad \wedge$
$BYTE = (BYTE\_INDEX \rightarrow BIT) \quad \wedge \mathbf{card}(BYTE) = 256 \wedge$
$\forall\ (b1).(b1 \in BYTE \Rightarrow \mathbf{size}(b1) = BYTE\_WIDTH \wedge b1 \in \mathbf{seq1}(BIT))$

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.

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

3.5 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 related to the type of function and check the type of functions with non-linear arithmetic is a difficult task.

3.6 Verification Process in Hardware Library and Types Library

This section presents the proof process of modules from basic data types. By default, the AtelierB [4] does not verify the type of defined functions in the clause $PROPERTIES$ from Machine. For example, the function *inc*, defined as $inc \in \{0,1\} \rightarrow \{0,1\} \wedge inc = \lambda(x).(x \in \{0,1\} | x+1)$, not creates a false proof obligation in Machine module. 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 function in $PROPERTIES$ clause and its type in $ASSERTIONS$ clause.

Another solution is check with ProB [13], it is able to check 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 this 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 not was possible to build proofs using the base of rules from the free distribution of AtelierB. Furthermore, the AtelierB 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 3 and they have the mean: *BYTE* is a bit vector with eight elements enumerated starting from one; *UCHAR* is a range between 0 and 255; *SCHAR* is a range between -128 and 127; *BV16* is a bit vector with sixteen elements enumerated starting from one; *USHORT* is a range between 0 and 65535; and *SSHORT* is a range between -32768 and 32767.

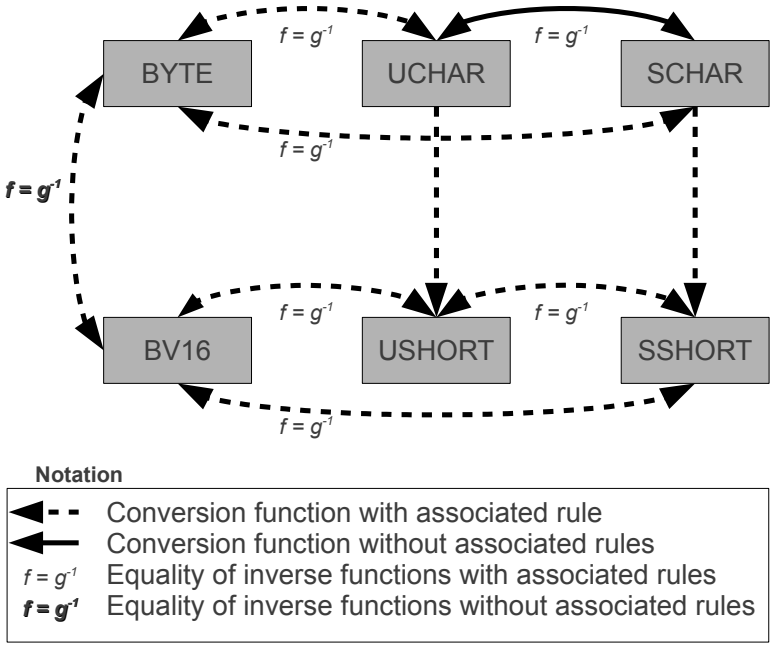

Fig. 3. Conversion functions

We need verify the type of these conversion functions.Basically, these functions must keep two properties: to be bijective and equal to its inverse function. But it is not easy to verify it using the AtelierB. Only these properties more simple 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, rules associated with the last two properties were created, 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 [8,14,18,22]. The others properties with associated rules are very similar, they have just small differences, for example, the size of bit vectors.

4 A B model of the Z80 instruction set

The *Z80* is a CISC microcontroller developed by *Zilog* [26]. The Z80 is composed by different elements and a simplified internal organization is shown in the Figure 4. 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. Groups of registers are represented by the variables of the specification states that appear 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).

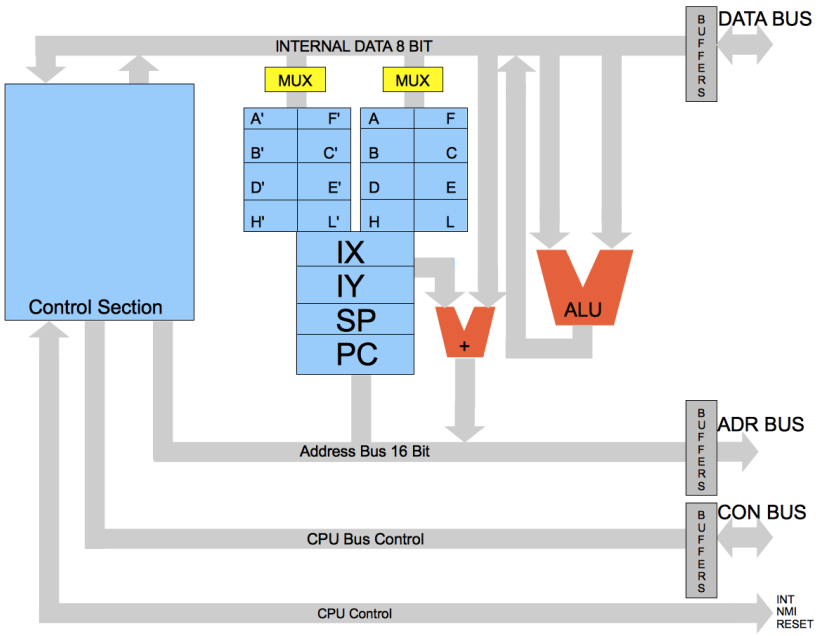

Fig. 4. Internal diagram block of Z80 CPU.

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

4.1 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
$rgs8 \in id\_reg\_8 \rightarrow BYTE \wedge$
$pc \in INSTRUCTION \wedge sp \in BV16 \wedge ix \in BV16 \wedge iy \in BV16 \wedge$
$i\_ \in BYTE \wedge r\_ \in BYTE \wedge$
$iff1 \in BIT \wedge \quad iff2 \in BIT \wedge$
$im \in (BIT \times BIT) \wedge$

$i\_o\_ports \in BYTE \rightarrow BYTE$

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 register 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
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 $\wedge$ bv_get(rgs8(f0),2) $\neq$ 1) in the invariant. When a overflow happen, it can be dangerous. So the developer may restrict its use. However, this restriction can also become more difficult to verify the model.*

4.2 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 a displacement from this base. This displacement is specified as a two's complement signed integer. This function is defined as:

$bv\_ireg\_plus\_d : (BV16 \times SCHAR \rightarrow BV16) \wedge$
$bv\_ireg\_plus\_d = \lambda\ (\ ix\_iy\ ,\ disloc\ )\ .\ (\ ix\_iy \in BV16 \wedge disloc \in SCHAR$
$\mid ushort\_bv16\ (\ (bv16\_ushort\ (\ ix\_iy\ ) + disloc\ )\ \textbf{mod}\ 65536\ )\ )$

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:$update\_flag\_reg \in (BIT \times BIT \times BIT \times BIT \times BIT \times BIT) \rightarrow (\{f0\} \times BYTE)$.

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

4.3 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 represents a byte. Thus, the data from the memory module is very simple, as shown below, but additional care must be taken to preserve the memory consistency.

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).*

4.4 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. 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*.

$half8UCHAR \in UCHAR \rightarrow UCHAR \wedge$

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

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:

$add8UCHAR = \lambda\ (carry,w1w2).$

$(carry \in BIT \wedge w1 \in UCHAR \wedge w2 \in UCHAR \mid$

$(((carry + w1 + w2\ )\ \textbf{mod}\ 2^8\ ),$

$bool\_bit(\ carry + uchar\_schar(w1) + uchar\_schar\ (w2) < 0),$

$bool\_bit(\ carry + w1 + w2 > UCHAR\_MAX),$

$bool\_bit(\ carry + half8UCHAR(w1) + half8UCHAR(\ w2) \geq 2^4),$

$bool\_bit(\ (\ (carry + w1 + w2\ )\ \textbf{mod}\ 2^8\ ) = 0))\ \ )$

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, but only some simplest functions also are explained below:

- *instruction_next* $\in$ *USHORT* $\rightarrow$ *USHORT* - It receives the actual value from program counter register (*pc*) and returns its increment.
- *is_negative* $\in$ *BYTE* $\rightarrow$ *BIT* - It returns the most significant bit, in other words, the signal bit.

The logic functions that are defined in the *BYTE* and *BV16* module are included in the *ALU* module. These functions can be seen and used directly in the *ALU* and *Z80* modules.

4.5 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 4.6) and the last category represents the external actions (shown in 4.7). 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* function 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_*).

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;
The other instructions have a similar structure.

4.6 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 is from I/O group, then it receives an identifier of register "*rr*" and, in this place, it stores the found value 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*

[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.

WHERE
$negative \in BIT \wedge zero \in BIT \wedge half\_carry \in BIT \wedge pv \in BIT \wedge$
$add\_sub \in BIT \wedge carry \in BIT \wedge$
$negative = is\_negative\ (\ io\_ports\ (\ rgs8\ (\ c0\ )\ )\ ) \wedge$
$zero = is\_zero\ (\ io\_ports\ (\ rgs8\ (\ c0\ )\ )\ ) \wedge$
$half\_carry = 0 \wedge$
$pv = parity\_even\ (\ io\_ports\ (\ rgs8\ (\ c0\ )\ )\ ) \wedge$
$add\_sub = \ 0 \wedge$
$carry = z\_c$
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
The main I/O instructions work similarly, for example: *OUT(n),A* or *OUT (C), r*.

4.7 Modelling the external actions

The External actions change the state of 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 interruptions, 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 the 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($\{\ (sp\_minus\_two \mapsto pc\_low),(sp\_minus\_one \mapsto pc\_high\ )\ \}$)||
$sp := sp\_minus\_two\ ||\ pc := 102\ ||\ iff1{:=}0\ ||\ iff2{:=}\ iff1\ ||$
$r\_ := update\_refresh\_reg(r\_)$

[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.

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 interruption 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* .
- 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 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.

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 [19] and its B assembly model can be executed manually in ProB. The modelling of instruction set and the B assembly models allow to simulate the execution and check several important properties.

5 Building and verifying assembly-level refinements

This section shows a case study that carries an out experimental evaluation of a pilot project with B refinements until the assembly level. The object of the pilot project was developed to analyse a petroleum production test system.

5.1 Petroleum production test system

The production test is "the process used to monitor the production of an oil field, this process is executed in each oil field in a predefined frequency" according to [19]. In this process, some characteristics are analysed (concentration of water, volume and temperature) from oil, gas and water.

This production test system has an important purpose in the process of oil extraction. It is required to analyse the production capacity of wells and calculate the government taxes, special participations and participations for landowners. However, based on the registers of the information system test, 10% of production tests are failed tests [19]. So, we have developed an embedded software verified until the assembly level to help the production test. Actually, the developed software is simplified, because its main objective is to evaluate and to improve the verification up to the assembly level.

The production test system is used to control and to evaluate the oil production. It has a system supervisor that calculates the oil production and controls the valves based on the state of system and information from sensor interface and radar. The radar informs the total level of fluid in the tank and the interface sensor informs the concentration of water in oil that helps to identify the level of interface oil and water.

The production test has basically three stages according to [19]. First, the conditioning stages begin, the control system of the tank is updated with data collected from the last test, the system keeps the well aligned[6] and the drain valve is closed in the end of this stage.

Then the process of filling and decanting begins. After filling and waiting, which can wait between 8 and 20 hours of decanting, the draining stage begins. The draining stage is on that last one stage, the system must control the drain valve not to allow the extracted oil to flow and to estimate final quantity oil. Only the calculus used to estimate oil quantity in this last stage is important to the case study verified up to assembly level.

The three stages are shown, in the same sequence, in Figure **??**. In this Figure, it is also possible to see the used devices in the production test.

[6] A well is aligned when the valves direct the flow pipe from the well to the tank.

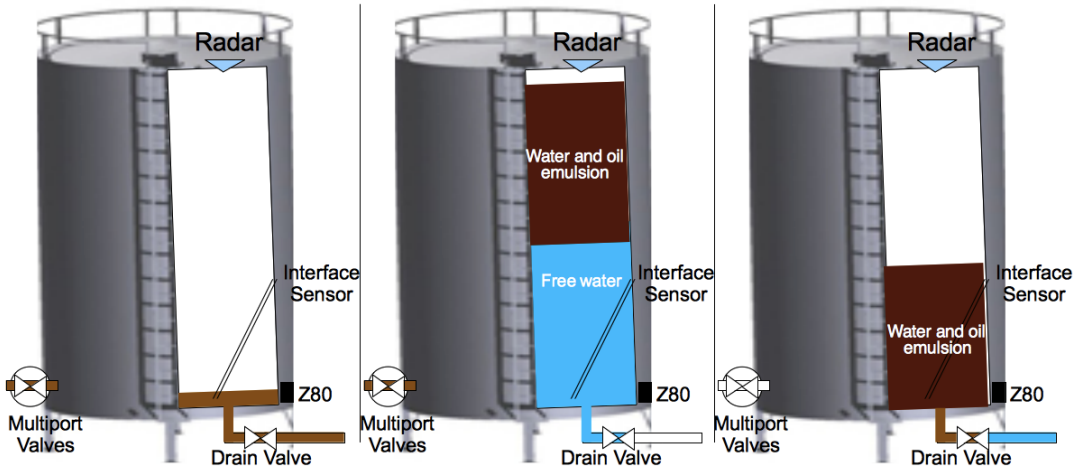

Fig. 5. Stages of the production test.

B models were created to represent partially the production test system. This report shows, in **??**, the main part of specification system is organized in three different B models: *TestCalc*, *TestCalc_i* and *TestCalc_basm*.

5.2 B modelling

This section shows the B modelling and the embedded software that calculates the proportion of water and oil emulsion produced.

The proportion of factors of water and oil emulsion are estimated by collected information in the draining stage. The information used are the levels of the tank: initial level (full tank) and final level (tank with only water and oil emulsion) from draining stage. So, the subtraction of two values determines a proportional factor of free water proportion and the final level of tank determines a proportional factor of produced water and oil emulsion (crude oil). To determine exactly the quantity of oil emulsion and water free, it is needed to multiply the levels by the correction factor, which represents the format of the tank and its deformations, because the tank has not a perfectly cylindrical shape. Thus, the final embedded software becomes generic for any tank shapes.

The modelling is started in the functional model *TestCalc*. This model has two important variables *oil_factor* and *free_water_factor* that represent, respectively, the final oil factor and the free water factor. The *Invariant* and the operation of functional model. The *Invariant* declares the types of variables as *UCHAR*, a positive integer value with 8 bits, in other words, it represents the positive integers between 0 up to 255. The operation receives as a parameter the initial value and final value of the tank, then, the software calculates the oil factor and free water factor. A precondition of this operation is that the initial level of full tank is bigger or equal than the level after the draining stage. This precondition is a simplification and it avoids to create exceptions up to the assembly level.

INVARIANT

$oil\_factor \in UCHAR \land free\_water\_factor \in UCHAR$

OPERATIONS

update_factor(*initial_level*, *final_level*) =

PRE

initial_level $\in$ *UCHAR* $\wedge$ *final_level* $\in$ *UCHAR* $\wedge$
final_level $\leq$ *initial_level*

THEN

free_water_factor := *initial_level* - *final_level* ||
oil_factor := *final_level*

END

The operation *update_factor* of functional model *TestCalc* is too much similar to its algorithmic modell, then the algorithmic modell was developed, verified and its presentation is omitted here.

Part of B modelling in assembly level is shown as it follows. It is specified in the model *TestCalc_basm* and it has the same semantic of manipulation of variables that the abstract model (*TestCalc*), however it uses operations that represent the assembly instructions of an instance of Z80 model to manipulate its memory.

The *Invariant* clause determines the relation of variables of initial model (*free_water_factor* and *oil_factor*) to the values from address 2 and 3 from I/O port; to determine the relation are used functions that convert values between binary representation and integer representation.

IMPORTS

Z80

INVARIANT

byte_uchar(*io_ports*(*uchar_byte*(2))) = (*free_water_factor*) $\wedge$
byte_uchar(*io_ports*(*uchar_byte*(3))) = (*oil_factor*)

Below is presented the operation *update_factor* using the instructions in the assembly level. This operation has the same signature as the most abstract model and the parameters received are converted to binary representation and transfered to the updated operation of the ports of the microcontroller (*ext_update_io_ports*).

Briefly, the sequence of instructions illustrated performs the following procedures. The first three instructions are representing just copying the data from external input and output ports to memory registers “A” and “C”. The following instruction performs a subtraction, then copies the other factors proportion of free water and crude oil, respectively, to ports 2 and 3. The reader may consult the site repository of this work to know more details about the specification of each statement and the complete *update_factor* operation.

```
update_factor(initial_level, final_level) =
    ASSERT
        initial_level ∈ UCHAR ∧ final_level   ∈ UCHAR ∧ final_level ≤ initial_level
    THEN
        VAR local_pc IN      local_pc := 0;      set_pc(local_pc);
            WHILE local_pc < 9 DO
                CASE local_pc OF
                    EITHER 0 THEN ext_update_io_ports(0,uchar_schar(initial_level));
                                    IN_A_9n0(0)
                    OR 1 THEN          LD_r_r_(b0,a0)
                    OR 2 THEN          ext_update_io_ports(1,uchar_schar(final_level));
                                    IN_A_9n0(1)
                    OR 3 THEN          LD_r_r_(c0,a0)
                    OR 4 THEN          LD_r_r_(a0,b0)
                    OR 5 THEN          SUB_A_r(c0)
                    OR 6 THEN          OUT_9n0_A(2)
                    OR 7 THEN          LD_r_r_(a0,c0)
                    OR 8 THEN          OUT_9n0_A(3)
                    END     END;     local_pc ← get_pc
                INVARIANT
                  local_pc ∈ 0 .. 9 ∧ rgs8 ∈ id_reg_8 → BYTE
                   ∧ r_   ∈ BYTE ∧ io_ports ∈ BYTE → BYTE
                   ∧ pc ∈ 0 .. 9 ∧ free_water_factor ∈ UCHAR ∧
                  (local_pc = 0 ⇒ ( pc = 0 ∧ instruction_next(pc) = 1 ∧
                        byte_uchar(io_ports(uchar_byte(2))) =  free_water_factor ∧
                        byte_uchar(io_ports(uchar_byte(3))) =  oil_factor ) ∧
                        ...
                  (local_pc = 9   ⇒ ( pc = 9 ∧
                        byte_uchar(io_ports(uchar_byte(0))) =  initial_level ∧
                        byte_uchar(io_ports(uchar_byte(1))) =  final_level ∧
                        byte_uchar( rgs8(a0) ) = ( final_level) ∧
                        byte_uchar( rgs8(c0) ) = ( final_level ) ∧
                        byte_uchar( rgs8(b0) ) = ( initial_level ) ∧
                        byte_uchar(io_ports(uchar_byte(2))) =  ((initial_level - final_level) mod 256)
                         ∧
                        byte_uchar(io_ports(uchar_byte(3))) =  final_level ))
                VARIANT (9 - local_pc) END
                END     END
        END
```

The invariant from **WHILE** must formalize every state of execution and mapping between the variables from the abstract model and values from memory address related. So, for each interaction of **WHILE** , which is associated with a value from the program counter (*pc*). The **VARIANT** clause must express the superior number of instructions to be executed for every possible value of the program counter.

An interesting detail of this model is that instructions of Z80 receive integer values, but the model of Z80 represents the data in binary notation. This difference could become the verification process more difficult. However, the types, functions and lemmas developed to support and to convert the data types facilitate the verification process.

Before starting the verification process, we recommend a review of the assembly code. So, this can be done using the assembly code simulation.

5.3 Assembly code simulation

The assembly code simulation is important to analyse its behavior, because the simulator allows to evaluate and to manipulate the state data (memory, registers and I/O ports) and execution flow. It also allows to remove some doubts that appear during the development.

The process simulation is simple. Basically, the Z80 receives the level of tank from radar, when the process begins the draining stage. When the draining stage finishes, the Z80 receives in another port the value of tank final level. After that, Z80 calculates and returns in other two ports the factor of the extracted free water and extracted crude oil.

This code was simulated using the software [20]. The Figure **??** shows the final state of code execution and the values from elements of microcontroller. The values from the simulator are represented in hexadecimal and binary representation. The window in the left superior side contains the register states of the microcontroller and the options to control the program execution. The right superior window contains the assembly code in execution and a yellow arrow, which informs the actual position of the program counter. The left inferior window contains a memory editor. The right inferior window contains the I/O port states. The address I/O ports *00H* contains the initial value/level of tank (10); the address *01H*, the final value/level of tank (2); the address *02H*, the factor of produced free water (8); and the address *03H*, the factor of produced crude oil (2).

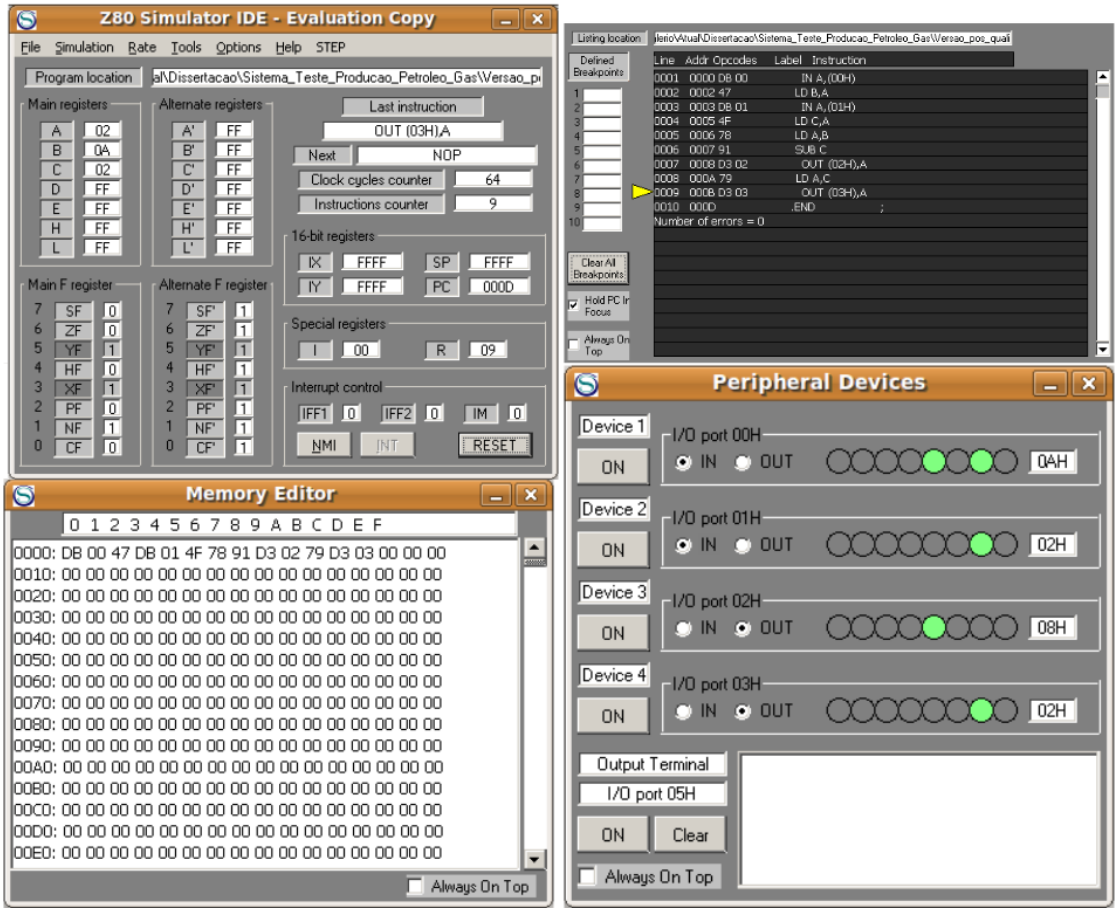

Fig. 6. Window of Z80 simulator [20].

6 Related works

There are in the literature several approaches [2,7,10,12,23] 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 [15,25] 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.

7 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 [26]. 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 work [17] describe the developed library and a Z80 initial model. The last work [16] 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 [11].

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. Valério Medeiros Júnior and David Déharbe. Formal modelling of a microcontroller instruction set in B. In *SBMF*, pages 282–289, 2009.
10. L. Casset; J. L. Lanet. A formal specification of the java bytecode semantics using the B method. Technical report, Gemplus, 1999.
11. Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.
12. 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.
13. 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.
14. M. Morris Mano. *Digital logic and computer design.* Prentice-Hall College, 1979.
15. 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).
16. Valério G. Jr Medeiros and David Déharbe. Formal Modelling of a Microcontroller Instruction Set in B. In *Brazilian Symposium on Formal Methods - Student Paper*, Gramado, RS, 2009. SBMF.
17. Valério G. Jr Medeiros, Stephenson S. L. Galvão, and David Déharbe. Modelagem de microcontroladores em B. In *Anais do VIII ERMAC*, Natal - RN, 2008. Encontro Regional de Matemática Aplicada e Computacional.
18. Umesh Vazirani Sanjoy Dasgupta, Christos Papadimitriou. *Algorithms.* Mc Graw Hill, 2006. Also available as `http://www.cs.berkeley.edu/~vazirani/algorithms/all.pdf`.
19. Paulo Sérgio Silva. Automação da drenagem no teste de produção convencional em tanque cilíndrico. Master's thesis, UFRN-PPgEEC, 2008.
20. Vladimir Soso. Z80 simulator ide. on-line, 2002. Available at:http://www.oshonsoft.com Acessado em:18 abr 2009.
21. John Michael Spivey. *The Z notation - a reference manual.* Prentice Hall International Series in Computer Science. Prentice Hall, 1989.
22. Daniel S. Wilkerson. Why the '+1' in two's complement arithmetic? Also available as `http://daniel-wilkerson.appspot.com/twos_complement.html`.
23. 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.
24. Sean Young. *The Undocumented Z80 Documented*, November 2003. Available at: http://www.myquest.nl/z80undocumented/z80-documented.pdf. Acessed in:April 18, 2007.
25. 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.
26. 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.