Experience in Modeling 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, Brazil
[2] Federal University of Rio Grande do Norte, Natal, RN, Brazil

Abstract. This paper presents a formal model, developed in B, of the Z80 instruction set. The formal model includes different libraries that specify entities commonly used in microcontrolers and microprocessors. Not only this model can be used for documentation purposes, but also to simulate and test the assembly code. It is also a fundamental component to extend the reach of the B method from the specification of the functional requirements of a system to an assembly-level implementation thereof.

1 Introduction

The B method [1] supports the formal development of software from a specification of functional requirements down to an imperative implementation of those requirements, amenable to synthesis to a programming language such as C or Ada. However, the synthesis step is not amenable to proof of correctness, and [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 model of the instruction set of the Z80 microcontroller [17]. This microcontroller was chosen because of several factors: it contains essential and common concepts of microcontrollers and even microprocessors; it has extensive available documentation; and it has been widely used. A first, partial, Z80 model presented in [10] was fully verified, but it used constructs that made it impossible to animate.

The new model supports animation in ProB [9]. Essentially, the new model[3] no longer contains definitions derived from infinite sets. Using B constructs for modularity, auxiliary libraries of basic modules were developed as part of the construction of the microcontroller model. Such a library has many definitions of concepts common to microcontrollers. Besides the new Z80 model, a petroleum production test system was developed using the Z80 assembly language to analyse the code simulation and the verification process.

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

The formal model of an instruction set has several applications. Animation allows to simulate the execution of assembly programs including support for interrupts, input and output instructions. Other possible uses include documentation, the construction of simulators, and can possibly be 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 such formalizations are of particular importance to us: provide a more solid documentation artifact for microcontrollers and build a reference model for a formal compilation approach in B.

It is often the case that the official documentation of an instruction uses textual description, mathematical formulas, examples, etc. Sometimes, the instruction descriptions have a specific notation, some informations are organized in different sections of a document and the textual descriptions are not standardized. The manuals also have semi-formal and informal descriptions, these descriptions may leave errors, inconsistency and ambiguity. For example, in the official manual of the microcontroller Z80 [17], the semantics of an instruction (in particular its effects on the flag register) are described informally, and this description is scattered in different pages. Furthermore, the official manual of the microcontroller Z80 has also several problems that were identified over time [15].

A solution to avoid ambiguity and inconsistencies is to formally specify the assembly instruction set, making it possible to prove properties of the instructions. The formal model also restricts the definitions to correct typing and uses only well-defined expressions. Furthermore, the developer may use the refinement capabilities of the B method to specify at different levels of abstraction and add more specific details or useful restrictions in refined models.

This paper is structured as follows. Section 2 presents the elementary libraries and the modeling of some elements common to microcontrollers. Section 3 presents the B model of the Z80 instruction set. Related work are discussed in section 4. 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 contain definitions to represent: 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 the following:

Power contains the standard definition of exponentiation, as well as some basic properties, e.g. values of different powers of two. Given this definition, these properties have been proved interactively with the Atelier-B prover.

Hardware Library defines bit vectors with size 8 and 16, basic functions to manipulate bit vectors and important assertions. Details are given in sections 2.1, 2.2 and 2.3.

Types Library defines sets of naturals and integers represented with 8 bits and 16 bits, basic conversion functions between those types and bit vectors, and important assertions. It is presented in section 2.4.

Platform defines functions of the arithmetic logic unit, memory unit, registers and assembly instructions. Section 3 provides additional details.

The dependencies between those libraries are depicted in Figure 1.

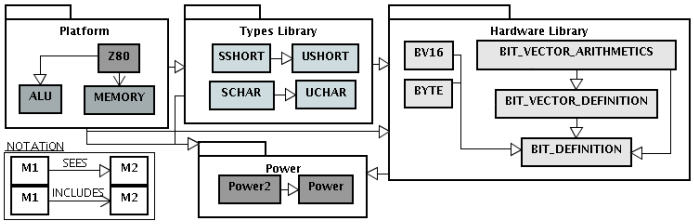

Fig. 1. Dependency diagram of the Z80 model.

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), and a conversion function from Booleans to bits.

We model bits as a set of integers: $BIT = 0..1$. The negation is an unary function on bits and it is defined as:

$$bit\_not \in BIT \to BIT \land bit\_not = \{0 \mapsto 1, 1 \mapsto 0\}$$

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:

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

The module provides the following lemmas for commutativity and associativity:

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

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

Finally, the conversion from Boolean to bit is simply defined as:

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

All the lemmas that are provided in this module have been automatically proved by the prover of Atelier-B.

2.2 Representation and manipulation of bit vectors

Sequences are predefined objects in B and are natural candidates to represent bit vectors. In B, sequences are functions whose 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 offset where necessary, so that predefined functions and proof rules may be used. We define bit vectors as non-empty sequences of bits: $BIT\_VECTOR = \mathrm{seq}\,1(BIT)$. As *BIT_VECTOR* is an infinite set, this definition hinders the animation in ProB. As a workaround, we defined specialized types *BYTE* and *BV16* without reference to *BIT_VECTOR*.

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

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

Function *bv_catenate* takes as parameters two bit vectors v and w, and returns 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. The domain of the binary operators is restricted to pairs of bit vectors of the same length:

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

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 Modeling bytes and bit vectors of length 16

Bytes, i.e. bit vectors of length 8, are a common entity in hardware design. We provide the following definitions:

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

BYTE_INDEX is the domain of the functions modeling bytes. It starts at 1 to be compatible with sequences. However, since the hardware community starts indexing from zero, definition *PHYS_BYTE_INDEX* provides functionalities obeying this convention. Type *BYTE* is as a specialized type of *BIT_VECTOR*, with a size constraint. Other specific definitions are provided to facilitate further modeling: the type *BV16* is created for bit vectors 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* maps bit vectors to natural numbers:

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

An important lemma associated to this definition is:

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

2.5 Basic data types

The definition of instruction sets usually have bit vector and integer types as listed in table 1. For each type, a number of functions are defined, including conversion between different types.

Table 1. Details of basic data types

Type Name	*Range*	*Physical Size*	*Num. POs*	*Library*
BIT	0..1	1 bit	118	Hardware
BYTE	–	1 bytes	154	Hardware
BV16	–	2 bytes	75	Hardware
UCHAR	0..255	1 byte	30	Types
SCHAR	-128..127	1 byte	30	Types
USHORT	0..65535	2 byte	62	Types
SSHORT	-32768..32767	2 bytes	62	Types

Usually, each type module just needs to specialize definitions that are given in the hardware modeling 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 be 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 lemmas.

We also created lemmas on pairs of dual conversion functions, such as:

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

Similarly, several other general functions and lemmas were created for all other data types. Some times, we need create arithmetic and logic functions that are more specific for a determined microprocessor, like Z80 that is presented in the section 3.

The definitions, functions and properties from hardware library and types library are difficult to verify, on the other hand these libraries can be reused in several others microprocessors without to verify again. Nearly 50% of all these proof obligations shown in the Table 1 are verified automatically or with few user pass[4] . But many others proof obligations are difficult to verify, because they involve functions that contain arithmetic expressions that are complex to manipulate with the interactive prover of AtelierB. However, all theses 531 proof obligations were verified completely.

2.6 Verification of hardware library and types library

This section presents the verification of the aforementioned modules using AtelierB provers. By default, AtelierB [4] does not generate proof obligations to check the type of functions defined in the $PROPERTIES$ clause of a machine. For example, the definition $inc \in \{0,1\} \rightarrow \{0,1\} \wedge inc = \lambda(x).(x \in \{0,1\} | x+1)$, does not create a proof obligation to check the consistency of the definition. However, this definition is inconsistent since $inc(1) = 2$ and $2 \notin \{0,1\}$.

To avoid such inconsistencies, we state separately the required properties in the *ASSERTIONS* clause: this causes corresponding proof obligations to be generated and verified. However, we have not been able to build the corresponding proofs using the rule base included in the free distribution of AtelierB. So we formalized new proof rules to check these properties in AtelierB and used them in the interactive prover of AtelierB.

The hardware library has many conversion functions with arithmetic, for example, Bit_vector to Natural, Natural to Bit_vector, Integer to Bit_vector

[4] The user pass are steps that aid the prover to find the proof.

and 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.

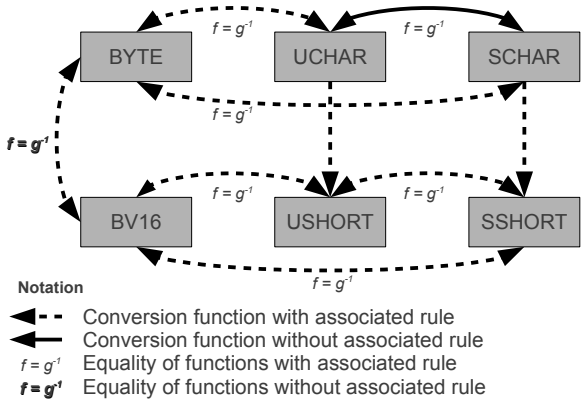

Fig. 2. Conversion functions

The conversion functions need to satisfy two properties: bijectivity and equality to the reverse of the dual conversion function. Three such proof obligations were verified automatically with AtelierB: ($uchar\_schar \in UCHAR \rightarrowtail\!\!\!\!\!\rightarrow SCHAR$), ($schar\_uchar \in SCHAR \rightarrowtail\!\!\!\!\!\rightarrow UCHAR$) and ($byte\_bv16 = bv16\_byte^{-1}$). For the remaining proof obligations, we again had to create new proof rules. There are three kinds of proof rules, that we illustrate by giving one example each:

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

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

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

Eight proof obligations require that conversion functions are finite. These proof obligations were verified using the expression evaluator of ProB. Finally, we have included rules related to binary arithmetic as developed in [3]. These rules are used in the interactive theorem prover with binary arithmetic to prove important properties [7,11].

3 A B model of the Z80 instruction set

The *Z80* is a CISC microcontroller developed by *Zilog* [17]. 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.

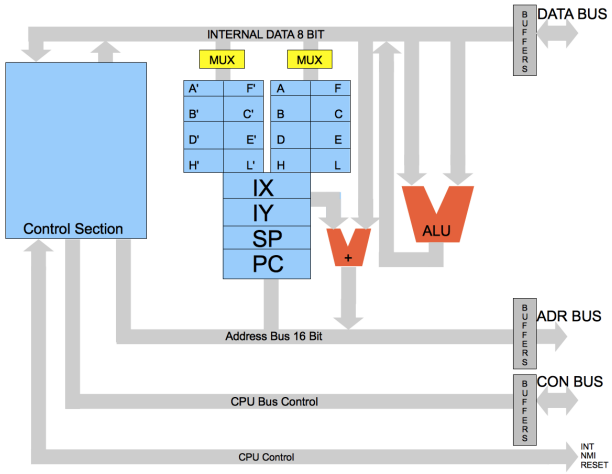

Fig. 3. Simplified internal organization of Z80 CPU.

The Z80 has 158 different instructions, including all the 78 from the 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 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 other 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	**SEES**
Z80	ALU, $BIT\_DEFINITION$, $BIT\_VECTOR\_DEFINITION$,
INCLUDES	$BYTE\_DEFINITION$, $BV16\_DEFINITION$,
MEMORY	$UCHAR\_DEFINITION$, $SCHAR\_DEFINITION$,
	$SSHORT\_DEFINITION$, $USHORT\_DEFINITION$

3.1 Modeling 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*). These definitions are represented by *INVARIANT*.

INVARIANT

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

$iy \in BV16 \wedge i\_ \in BYTE \wedge r\_ \in BYTE \wedge iff1 \in BIT \wedge iff2 \in BIT \wedge$
$im \in (BIT \times BIT) \wedge i\_o\_ports \in BYTE \rightarrow BYTE$

The internal registers contains 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_reg8*) 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*. These sets are represented below.

SETS

id_reg_8 = { *a0*, *f0*, *b0*, *c0*, *d0*, *e0*, *h0*, *l0*, *a_0*, *f_0*, *b_0*, *c_0*, *d_0*, *e_0*, *h_0*, *l_0* };

id_reg_16 = { *AF* , *BC*, *DE*, *HL*, *SP*}

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

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:

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

Another derived function is *bv_(ireg_plus_d)*, which returns the value in the memory address returned by the *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:$update\_flag\_reg \in (BIT \times BIT \times BIT \times BIT \times BIT \times BIT) \rightarrow (\{f0\} \times BYTE)$. This function is defined as:

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

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 memory is very simple: $mem \in BV16 \rightarrow BYTE$.

In general, the instructions can access all memory addresses, 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 in a refinement model 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.

Assuring the absence of overlapping of address regions: *To assure that address regions are well defined, then the designer must verify the expression:* $PROGRAM\_R\_ADR \cap DATA\_R\_ADR \cap = \{\} \wedge STACK\_R\_ADR \cap DATA\_R\_ADR \cap = \{\} \wedge PROGRAM\_R\_ADR \cap STACK\_R\_ADR = \{\}$

Preserving the memory consistency: *In general, the access to some address regions is dangerous. Then, each instruction has a specific pre-condition that verifies if the new address memory, that will be updated, is a member of its region. For example, the PUSH program instruction permits 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*.

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

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:

$$\begin{aligned}
&add8UCHAR = \lambda\ (carry, w1, w2).\\
&(carry \in BIT \wedge w1 \in UCHAR \wedge w2 \in UCHAR \mid\\
&\quad (((carry + w1 + w2\ )\ \textbf{mod}\ 2^8\ ),\\
&\quad bool_bit(\ carry + uchar_schar(w1) + uchar_schar\ (w2) < 0),\\
&\quad bool_bit(\ carry + w1 + w2 > UCHAR_MAX),\\
&\quad bool_bit(\ carry + half8UCHAR(w1) + half8UCHAR(w2) \geq 2^4),\\
&\quad bool_bit(\ (\ (carry + w1 + w2\ )\ \textbf{mod}\ 2^8\ ) = 0))\ \ )
\end{aligned}$$

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 Modeling the actions and instructions

Each instruction is represented by a B operation in Z80 module. 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 is $LD\_(nn)\_A$ [5] 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 $nn \in USHORT \quad \wedge \quad nn \quad \in DATA\_R\_ADR$ **THEN**
  updateAddressMem ($ushort\_bv16$ (nn) , $rgs8$ ($a0$)) ||
  $pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END

3.6 Modeling 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 $IN\_r(C)$ [5] instruction is represented by the following B operation. It receives a 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 ($rgs8$ ($c0$))) $\wedge$ $half\_carry = 0 \wedge$

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

$pv = parity\_even\ (\ io\_ports\ (\ rgs8\ (\ c0\ )\ )\ ) \wedge add\_sub = \ 0 \wedge carry = z\_c$

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

END

3.7 Modeling the external actions

The external actions change the state of the microcontroller, for example, refreshing the I/O ports and interruption request. The external actions are also modeled 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 instruction *ext_update_io_ports* just changes 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. Interruptions allow devices to 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 modeled by B operations and its main effects are presented here[6].

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({ $(sp\_minus\_two \mapsto pc\_low),(sp\_minus\_one \mapsto pc\_high\ )$ })||
$sp := sp\_minus\_two\ ||\ pc := 102\ ||\ iff1{:=}0\ ||\ iff2{:=}\ iff1\ ||$

[6] 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.

$r\_ := update\_refresh\_reg(r\_)$

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*.
- 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 $byte\_bus = opcode\_\ldots\_instruction$
    ...
  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.
$iff1 := 0 \parallel iff2 := 0 \parallel im := (0 \mapsto 0) \parallel pc := 0 \parallel i\_ := [0,0,0,0,0,0,0,0] \parallel$
$rgs8 := rgs8 \leftarrow \{ (a0 \mapsto [0,0,0,0,0,0,0,0] , (f0 \mapsto [0,0,0,0,0,0,0,0] \} \parallel$
$r\_ := [0,0,0,0,0,0,0,0] \parallel sp := [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$

3.8 Verification and validation

The model presented in this paper is suitable both for validation through animation with ProB, and for verification through the standard proof obligations generated by the application of the B method.

The animation in ProB is not as fast and practical as a simulation with a dedicated tool (e.g. [13]). ProB is a general-purpose animator of formal specification, and addresses computationally more difficult problems. Anyway, it is capable of animating, and also possibly model checking, a formal model of the instruction set of the Z80, which was our initial motivation in constructing that model. One may simulate step-by-step sequences of Z80 instructions and visualize the visited states of ports, memory and registers. It is a useful tool to debug by creating a trace of the visited states, offering to analyse constants, functions and expressions, and search possible violations.

The construction of the model with B generates a large number of proof obligations and the verification is a time-consuming task. We describe here some details on this effort and some solutions we have devised to reduce the effort. In B, the generated proof obligations allow to verify the data satisfy invariant properties, check important system properties and that the expressions are well-defined (WD)[7].

Table 2 displays statistics of the verification process. Note that the figures in it include only the number of *non obvious* proof obligations[8]. The titles of columns use acronyms: **PO** - number of proof obligations; **UPO** - number of (yet) unproved proof obligations; **WD PO** - number of proof obligations related to well definition lemmas; **UWD PO** - number of (yet) unproved proof obligations related to well definition lemmas.

There is still a large number of open proof obligations, i.e. we have not yet found their proof, or they are not valid and our model is not correct. Each such proof obligation requires interaction with a proof assistant. We see that for a project of that size, a large effort needs to be dedicated to find those proofs. One interesting feature of the proof assistant is, once a proof has been found, it can automatically apply it to other proof obligations, as it is often the case that the same proof script discharges several proof obligations.

We had a previous experience with a similar effot, where we were able to discharge all the proof obligations: that of the model of the Z80 instruction set designed without restricting ourselves to constructs that may be animated with

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

[8] Obvious proof obligations are simple formulas automatically eliminated by the tool.

Table 2. Verification statistics

Component	PO	UPO	WD PO	UWD PO	Total PO	Total UPO	Total Percentage
BIT	49	0	69	0	118	0	100%
BV16	6	0	69	0	75	0	100%
BYTE	18	0	136	0	154	0	100%
POWER	3	0	4	0	7	0	100%
POWER2	18	0	0	0	18	0	100%
SCHAR	4	0	26	0	30	0	100%
SSHORT	4	0	58	0	62	0	100%
TYPES	71	0	0	0	71	0	100%
UCHAR	4	0	26	0	30	0	100%
USHORT	4	0	58	0	62	0	100%
ALU	51	22	117	47	168	69	59%
MEMORY	13	0	0	0	13	0	100%
Z80 - Initialization, Properties, Assertions	66	9	150	11	216	20	91%
Z80 - Input and Output	46	11	281	16	327	27	92%
Z80 - Logic Arithmetic 1	88	33	343	84	431	117	73%
Z80 - Logic Arithmetic 2	40	20	178	27	218	47	79%
Z80 - Bit Control	111	34	763	157	874	191	78%
Z80 - External Actions	31	0	60	38	91	38	59%
Z80 - General 1	81	19	420	20	501	39	92%
Z80 - General 2	26	10	139	14	165	24	86%
Total	**734**	**158**	**2897**	**364**	**3631**	**569**	**84%**

ProB, and without modeling external actions (as described in section 3.7). In that first experience, the verification took us about two months. In that second project, we have been able to verify 3062 (e.g. roughly 80 %) of 3631 proof obligations in one week.

In order to be able to verify such a large number of proof obligations in such a short term, we employed two strategies: parallelism and *user pass*. AtelierB has an option to run any number of threads in parallel and also to send proof obligations to other, possibly remote, processes running AtelierB. With those options we had the possibility to have eight concurrent instances of the prover. We thus split the instruction set of the Z80 in eight sub-modules and checked them concurrently (each instance of AtelierB may only verify one module at a time).

The second strategy, called *user pass*, takes advantage of another feature of the AtelierB prover. We defined a number of proof scripts which are then applied to all remaining proof obligations. Few generic user passes are enough to prove many proof obligations. Also, as the models of many instructionsa are similar, proofs found in interactive mode may similarly be replayed and discharge

other proof obligations. We have defined a set of 17 user passes that are able to discharge more than 2500 WD proof obliagations.

3.9 Considerations about the Z80 model

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 complement the existing documentation for users and assembly programmers.

But our goal is to use the model of the Z80 instruction set to extend the scope of the B method up to the assembly level [6]. Indeed, we realized a case study where an abstract B model had been refined up to the algorithmic model using the classic development strategy. The algorithmic model was manually compiled to a Z80 assembly program. We then constructed a B module composed with the instructions of that program, and importing the formal model of the Z80 instruction set. We then showed that this assembly-level module was indeed a refinement of the original abstract model.

This case study is a component of a pilot project to analyse a petroleum production test system. Its modeling was developed according to [12] and its B assembly model could also be animated with ProB.

4 Related work

There exists several approaches to model hardware and instruction set using B a. The paper [16] reports a method to specify, design and construct sound and complete instruction set model by stepwise refinement and formal proof using Event-B. It also discusses desirable properties of the instruction set model. However, our work [10] and [14] use the B method, that seems quite appropriated to software development, because it has an implementable language defined, called B0, a model-driven approach [6] to develop software from the functional specification level down to assembly, and tools to convert the models to a programming language.

A related experience on using B in the design of secure micro-controllers is present in [2]. It tries to show the feasibility of such a technique for high confidence trustful devices. The work [14] describes a similar approach with MIPS CPU architecture and it shows how the CPU is formally specified and implemented using the B method.

5 Conclusions

This work has shown an approach to the formal modeling 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 [17]. We have consulted different sources [13,15,17] to avoid such problems in modeling and have developed a case study: a verified embedded software. This case study was interesting to as a feasibility analysis of the approach to derive formally assembly code from a specification of functional requirements in the B method.

The following works are directly related to the objective this paper. The approach to develop verified software down to the assembly level using B was described in [5]. A first case study for this approach was reported in [6], presenting more details as well as a small software that developed up to assembly level in three different platforms. A general view of a previous used in verification process were presented in [10]. The model presented in the current paper added support for animation in ProB and the representation of external actions (interruptions). This set of papers present an important step to software verification up to assembly language, showing the current difficulties and suggesting improvements in tools supporting the B method.

In the future, we plan to improve the support for the development of software with the B method from functional specification to assembly level, using the Z80 model presented in this work. For instance, the mechanic compilation from B algorithmic constructs to assembly platform is also envisioned. Finally, another important activity is to develop a formal model of a platform used in LLVM Compiler [8], which would enable us to integrate different compilation techniques into our B-based framework.

Acknowledgements: The animation of the model would not have been possible without the help of Michael Leuschel who kindly provided feedback and developed several improvements to ProB to meet our needs.

References

1. J. R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1 edition, 1996.
2. Marc 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).
3. Gottfried Wilhelm Leibniz By Karl Immanuel Gerhardt. Leibnizens mathematische schriften. *G. H. Pertz*, 1859. Available at: http://www.archive.org/details/leibnizensmathe07leibgoog. Accessed on January 10, 2012.
4. Clearsy. Atelier B web site. Available at: http://www.atelierb.eu, 2009. Accessed on January 10, 2012.
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. Emeritus James L. Hein. *Discrete Structures, Logic, and Computability*. Portland State University, 2010.
8. Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.
9. 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.
10. 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.
11. Umesh Vazirani Sanjoy Dasgupta, Christos Papadimitriou. *Algorithms*. Mc Graw Hill, 2006. Available at http://www.cs.berkeley.edu/ vazirani/algorithms/all.pdf. Accessed on January 10, 2012.
12. Paulo Sérgio Silva. Automação da drenagem no teste de produção convencional em tanque cilíndrico. Master's thesis, UFRN-PPgEEC, 2008.
13. Vladimir Soso. Z80 simulator ide. on-line, 2002. Available at: http://www.oshonsoft.com. Accessed on January 10, 2012.
14. Paul Subotic and Ramunas Gutkovas. Provably correct software project: Behaviourly correct mips simulator. Technical report, Uppsala university, 2010.
15. Sean Young. *The Undocumented Z80 Documented*, November 2003. Available at: http://www.myquest.nl/z80undocumented/z80-documented.pdf. Accessed on January 10, 2012.
16. 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.
17. Zilog. *Z80 Family CPU User Manual*. ZiLOG Worldwide Headquarters, 910 E. Hamilton Avenue, 2001. Available at:http://www.zilog.com/docs/z80/um0080.pdf. Accessed on January 10, 2012.