Verified compilation based on the B method: initial appraisal (extended version)

B. Dantas, D. Déharbe, S. Galvão, V. Medeiros Jr and A. Moreira

Technical Report - UFRN-DIMAp-2008-101-RT - Relatório Técnico
April - 2008 - Abril

The contents of this document are the sole responsibility of the authors.
O conteúdo do presente documento é de única responsabilidade dos autores.

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
www.dimap.ufrn.br

Verified compilation based on the B method: an initial appraisal (extended version)

Bartira Dantas David Déharbe Stephenson Galvão
Valério Medeiros Júnior
Anamaria Martins Moreira
{david, stepgalvao, junior, anamaria, bartira}@consiste.dimap.ufrn.br

Abstract. *This report presents an application of the software engineering process known as the B method beyond the classical algorithmic level provided by the B0 sub-language, and presents refinements of B models in a level of precision equivalent to assembly language. We claim and justify that this extension provides for a more reliable software development process as it bypasses two of the less trustable steps in the application of the B method: code synthesis and compilation. The results presented in the paper have a value as a* proof of concept *and may be used as a basis to establish an agenda for the development of an approach to build* verifying compilers *[4] based on the B method.*

Keywords: Software engineering; formal methods; B method; refinement; compilation.

Resumo. *Esse relatório apresenta a aplicação do método B, um processo de engenharia de software, além do nível algorítmico tradicional provido pela sub-linguagem B0, e apresenta refinamentos de modelos B em um nível de precisão que equivale ao de uma linguagem de assemblagem. Afirmamos, e justificamos, que essa extensão resulte em um processo de desenvolvimento de software mais confiável pois elimina dois passos menos confiáveis na aplicação do método B: a síntese de código em linguagem de programação e a sua compilação. Os resultados apresentados têm valor como* prova de conceito *e são usados como base para estabelecer uma agenda para desenvolver uma abordagem de construção de compiladores verificadores [4] baseados no método B.*

Palavras-Chave: Engenharia de *software*; métodos formais; o método B; refinamento; compilação.

1 Introduction

The reliability of the result of a software development process depends on each step of this process. The B method manages to successfully span the development process from modelling

down to the algorithmic level [2]. The B method consists in building an initial functional model, and then, through incremental refinements, generate a program-like implementation, each step being subject to formal verification. To cover the remaining steps towards a running implementation, one needs to synthesize the algorithmic model to some programming language and then compile the resulting code to the target platform assembly language. These last two steps cannot be verified using the formal verification approach provided by the B method. Indeed, code synthesis maps constructs of languages that do not have common semantic underpinnings. Compilation is even more troublesome, since, in addition to the semantic gap, there is a also usually a deep transformation of the code structure caused by optimisation and other transformations and the effort put in the B-based development may be jeopardized by a bug in the compiler.

The problem thus addressed is difficult and may be viewed as addressing the grand challenge for computing research set forth by Tony Hoare in 2005 [4]: the *verifying compiler*, defined as:

> A verifying compiler uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles. Correctness is specified by types, assertions, specifications, and other redundant annotations that accompany the code of the program.

In our case, the specification is the initial functional model defined as the starting point of every development carried out with the B method, and correctness is established if one can prove that the implementation is a refinement of the specification.

One possible solution to this problem, already addressed in previous work, is to derive test cases from the initial functional model built in the B method [6]. These tests may then be employed to exercise the implementation, and the compliance of the results can be verified against the initial model. However, as with any test-based approach, this solution lacks completeness in practical situations.

This paper proposes an approach that, although based on existing ingredients, innovates in employing them to extend the reach of the B method to the assembly level, thereby eliminating the need for two of the less reliable steps following the application of the B method, namely code synthesis and compilation. Proof obligations may be generated and discharged to check that the resulting assembly program refines (i.e. is a consistent implementation of) the original functional model. Employing this approach, the translation to binary code is as simple and direct as that of an assembler, i.e. does not require modification of the code structure and may be considered as a (trivial) implementation of a one-to-one function from assembly symbolic instructions to their binary correspondents. This work can therefore also be viewed as a first step towards taking on Tony Hoare's grand challenge for building a verifying compiler [4], using as a basis the B method, and can be related to several complementary approaches, such as:

- Source code instrumentation through assertions (e.g. Eiffel, JML, Spec#);
- A formal proof of the compiler correctness [7];
- Certification that the generated binary code does not present some predefined classes of bugs [9].

Plan of the paper. The paper is structured as follows. Section 2 provides a general introduction to the B method. Section 3 motivates and briefly outlines the proposed approach and compares it to a classical application of the B method. The approach is then instantiated using

as computational platform the *random access machine*, a computational model that is conceptually similar to that of a register-based micro-controller. Section 4 presents the main lines of the definition of a B model for the *random access machine* and Section 5 presents the mapping of the main algorithmic constructs of the B method to assembly constructs through a series of simple examples, and introduces thus the structure of an assembly program as a B implementation. In Section 6, the approach is investigated on an industrial computational platform, the PIC16C432, realizing the B development of a simple example up to the assembly level of this platform. Finally, Section 7 draws preliminary conclusions on this work and an agenda for future research in the direction of constructing a production-level verifying compiler based on the B method.

2 Introduction to the B method

The B method for software development [2, 14] is based on the B *Abstract Machine Notation* (AMN) and the use of formally proved refinements up to a specification sufficiently concrete that programming code can be automatically generated from it.

Its mathematical basis consists of first order logic, integer arithmetic and set theory, and its corresponding constructs are very similar to those of the Z notation [13]. Its structuring constructs are however stricter and more closely related to imperative modular programming language constructs, with the intention of being more easily understood and used outside the academic world. Also, its more restrictive constructs simplify the job of support tools. Industrial tools for the development of B based projects have been available for a while now [3, 8], with specification and verification support as well as some project management tasks and support for team work. Its modular structure and characteristics make it adequate for the specification of Application Programming Interfaces (APIs) or other software components.

2.1 An overview

A B specification is structured in modules. A module defines a set of valid states, including a set of initial states, and operations that may provoke a transition between states. The design process starts with a module with a so-called *functional* model of the system under development. In B, such module is called a *machine*. In this initial modelling stage, one may use semi-formal techniques and notations, such as UML, that ease the transition from a requirements document written in natural language to the formal notation of the B method [12, 10]. The B method requires that the user proves that, in a machine, all the initial states are valid, and that operations do not define transitions from valid states to invalid states.

Once an initial functional model has been constructed and verified, the B method provides constructs to define so-called *refinement* modules. A refinement is always associated to another, more abstract, model and specifies a design decision: either about the concrete representation of the state, or about the algorithmic realization of an operation (or both). The B method imposes that the user proves that each refinement conforms to the refined module.

Finally, so-called *implementation* modules form a special case of refinement where the abstraction level is similar to that of a programming language. This paper uses the term *algorithmic model* to qualify such modules. The part of the B notation that may be used to define implementations is called B0; for instance it does not contain non-deterministic constructs. Using as input an implementation module, it is possible to generate source code in a conventional program language such as C or ADA.

Thus, at each step of the B method, illustrated graphically in Figure 1, proofs need to be developed to check the consistency of the model or the conformity of the refinements. Technical details on the proof obligations are provided in Section 2.5.

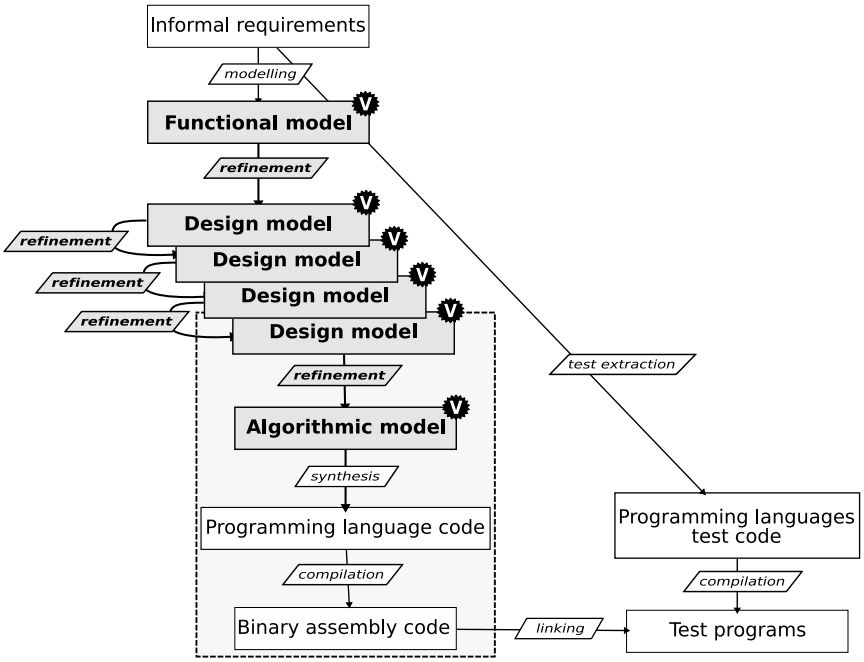

Figure 1: Overview of a software engineering process based on the B method. Rectangle correspond to different artifacts. Slanted rectangles are human or automated activities. Grey rectangles are artifacts produced in the B method. The V labels emphasize that formal verification is applied to the corresponding artifact. The light-grey area is the main focus of this work.

In such process, once the resulting code has been compiled, it may be verified using test data generated from the initial functional model. This verification is generally not exhautive but may detect errors introduced by the code synthesis tool or the compiler.

2.2 The B notation

Essentially, a B module contains two main parts: a state space definition and the available operations. It may additionally contain auxiliary clauses in many forms (parameters, constants, assertions), but those, essentially for practical purposes (i.e. to promote modularity, reuse, etc.), and do not extend the expressive power of the notation. In the remainder, we will restrict our discussion to the core clauses of the module specification.

The specification of the state components appears in the VARIABLES and INVARIANT clauses. The former enumerates the state components, and the latter defines restrictions on the possible values they can take. If V denotes the state variables of a machine, the invariant is a predicate on V. Let us denote INV such invariant predicate. Verifications carried out throughout the development process have the intention of checking that no invalid state will ever be reached as long as the operations of the machine are used as specified.

For the specification of the initialisation as well as the operations, B offers a set of so-called *substitutions*. These are "imperative-like" constructions with translation rules that define their semantics as the effect they have on the values of any (global or local) variables to which they are applied. The semantics of the substitutions is defined by the *substitution calculus*, a set

MACHINE *traffic_light*
SETS $COLOR = \{green, yellow, red\}$
VARIABLES *color*
INVARIANT $color \in COLOR$
INITIALISATION $color :\in COLOR$
OPERATIONS
$advance =$
CASE *color* **OF**
EITHER *green* **THEN** $color := yellow$
OR *yellow* **THEN** $color := red$
OR *red* **THEN** $color := green$
END
END
END

Figure 2: An example of a functional model in B

of rules stating how the different substitution forms rewrite to formulas in first-order logic. Let S denote a substitution, E an expression, then $[S]E$ denotes the result of applying S to E.

For instance, an operation that would increment a counter variable v can be specified as $v := v + 1$. Indeed, the basic substitution is very similar to the side-effect free assignment construct found in imperative programming languages. Applying such substitution to an expression consists in substituting the target variable v with the source expression. For instance, $[v := v+1]v \geq 0$ simplifies to $v+1 \geq 0$. Besides the basic substitution, the B method provides more elaborate substitution constructions, such as:

- Non deterministic substitution $\mathbf{ANY}\, v\, \mathbf{WHERE}\, C\, \mathbf{THEN}\, S\, \mathbf{END}$ applies substitution S with variable v having any value that satisfies predicate C.

  Substitution $v :\in V$, where V is a set, is equivalent to $\mathbf{ANY}\, x\, \mathbf{WHERE}\, x \in V\, \mathbf{THEN}\, v := x\, \mathbf{END}$.

- Parallel substitution $[S \parallel S']$ applies both substitutions S and S' simultaneously.

- The substitution with pre-condition $\mathbf{PRE}\, C\, \mathbf{THEN}\, S\, \mathbf{END}$ is used to specify a partial operation, defined only when condition C holds. For instance, the operation that increments v only when it is smaller than value *top* may be specified as $\mathbf{PRE}\, v < \mathit{top}\, \mathbf{THEN}\, v := v + 1\, \mathbf{END}$.

2.3 Example of a functional model

The example shown in Figure 2 illustrates the most basic clauses in the functional model of a traffic light. The model is named *traffic_light*, as defined in clause MACHINE. A set named $COLOR$ is then defined; it contains three elements corresponding to the possible light colors. The state is composed of a single variable, named *color*. The value of this variable must belong to $COLOR$ and is non-deterministically initialized with one element of this set. Follows then the specification of the operation $advance$, modelling a transition of the traffic light.

2.4 Example of a refinement

The previous model may be refined to a module where the state is a single integer value, as illustrated in Figure 3. The refinement is named *traffic_light_data_refinement* and the refined models is referenced in the REFINES clause. The CONSTANTS clause declares two functional constants, and their definitions are given in the PROPERTIES clause. The VARIABLES clause declares the unique component state in the refinement, and its relationship with the functional

REFINEMENT *traffic_light_data_refinement*
REFINES *traffic_light*
CONSTANTS $color\_refine, color\_step$
PROPERTIES
$color\_refine \in COLOR \rightarrow \mathbb{N} \wedge$
$color\_refine = \{green \mapsto 0, yellow \mapsto 1, red \mapsto 2\} \wedge$
$color\_step \in 0..2 \rightarrow 0..2 \wedge$
$color\_step = \{0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0\}$
VARIABLES $count$
INVARIANT $count \in \mathbb{N} \wedge count \in 0..2 \wedge count = color\_refine(color)$
INITIALISATION $count := 0$
OPERATIONS
$advance = count := color\_step(count)$
END

Figure 3: A B refinement module for the example of Figure 2

model state is established in the INVARIANT clause: the value of $count$ must be equal to the application of function $color\_refine$ to the abstract variable $color$. The refinement of the initial state is specified in the INITIALISATION clause: the variable $count$ is assigned the value 0. The refined version of the $advance$ operation is specified in the OPERATIONS clause.

2.5 Proof obligations

To guarantee the correctness of a full design with the B method, one must check that functional models are consistent and that each refinement is compatible with the module it refines. To this end, proof obligations need to be generated and verified. Such proof obligations are described in the remainder of this section.

A functional model is considered consistent when:

1. The initialization actions take the machine into a valid state. Thus the initialization substitution S must establish the invariant: $[S]INV$.

2. No operation may take the machine from a valid state to an invalid state, as long as the user provided parameters and the machine variables are such that the pre-condition PRE for application of the substitution S corresponding to this operation evaluates to true: $PRE \wedge INV \Rightarrow [S]INV$.

For instance, the proof obligation for the initialisation clause of the model of Figure 2 is:

$$\begin{array}{ll} & [color :\in COLOR]color \in COLOR \\ \equiv & \forall x \bullet (x \in COLOR \Rightarrow [color := x]color \in COLOR) \\ \equiv & \forall x \bullet (x \in COLOR \Rightarrow x \in COLOR) \\ \equiv & true \end{array}$$

Refinements are also subject to verification. The B method generates proof obligations establishing that the result of a refinement is compatible with the original specification: pre-conditions cannot be weakened, post-conditions cannot be strengthened, and the invariant must be preserved.

In the case of a refinement, let INV_R and INV_M be the respective invariant of the refinement and machine, the consistency is verified when:

- The refinement initialisation, denoted $INIT_R$, shall establish that every concrete initial state refines some abstract initial state. If $INIT_M$ denotes the initialisation of the abstract model, this property is expressed in the substitution calculus as:

$$[INIT_R]\neg[INIT_M]\neg INV_R.$$

- For the operations, three properties need to be checked. First, the refinement operation OP_R must be aplicable whenever the corresponding abstract operation OP_M is. Therefore its pre-condition PRE_R shall be weaker than the the abstract operation pre-condition PRE_M. Second, any transition of the concrete model shall correspond to, and be compatible with, a transition of the abstract model. Finally, the operation outputs o shall be the same when the inputs are equal. These conditions are formalized in the following fashion:

$$INV_M \wedge INV_R \wedge PRE_M \Rightarrow PRE_R \wedge [[o' := o]OP_R]\neg[OP_M]\neg(INV_R \wedge o = o'),$$

  where o' is a (set of) fresh variable(s).

In addition to this set of proof obligations, several other proof obligations need to be verified. For instance, whenever an operation is applied, one must verify that its precondition is entailed by the context where the operation is used. Also, for each module, one needs to verify the feasability of the system, that is the different constraints on the module parameters, constants and variables are satisfiable.

The B method requires two classes of tools to support verification:

- The *generation* of proof obligations from a B module, that may be carried out in a totally automatic fashion.

- The *verification* of proof obligations, that is semi-automatic. Generally, a large part of the proof obligations is simple enough to be proved without human intervention. For the remainding proof obligations, the user may interact with the verifier to add rules, or to select relevant hypothesis, instantiate quantified formulas, realize simplifications, and other such operations that allow either to discharge the proof obligation, or to discover that the proof obligation cannot be verified because it is not valid, as there is an error in the module that remains to be corrected.

3 Overview of the approach

The *weakest link* in a software production process based on the B method is the synthesis of software in a programming language and its compilation towards the target platform assembly language. As the B0 language is close to programming constructs, code synthesis is usually considered safe; however if the target language uses constructs unsupported by the B0 language (e.g. object orientation), this transformation may not be as straightforward as it seems. In industrial practice, a redundant tool chain (two code synthesis tools and two compilers) is used to produce two versions of the program. Both versions are executed and their results should agree. This approach requires twice the number of computational resources that would be necessary if only one instance of the generated program was executed.

This paper proposes a new approach to address these issues, by applying the concepts of the B method to generate software artifacts down to assembly level. The approach is divided

Figure 4: Extending the B method down to the assembly level: ideal (left) and actual (right).

into: (1) modelling the target computational platform, and (2) refining the algorithmic model to an implementation solely based on the platform model.

The target platform may be modeled with the B abstract machine notation: the state of the machine represents the state of the platform (i.e. registers and memory), and each operation represents an assembly instruction. This only needs to be performed once for a given target platform. Further details are provided in Section 4, with a model of the Random Access Machine model of computation, and in Section 6.1, with a model of an industrial micro-controller.

The algorithmic model has to be further refined into an assembly-level model. The latter model is defined on top of the target platform model discussed previously. A general strategy of this refinement is to map the state variables of the algorithmic model to different addresses of the platform memory, and to translate the algorithmic-level operations to combinations of operations defined in the platform model corresponding to the assembly language instructions. The resulting assembly-level refinement needs to be proved compilant with the corresponding functional model. We then obtain a software artifact at the assembly level that is provably compliant with the initial functional model.

This approach provides an extension to the B method as sketched in the left of Figure 4. However, the classic B method has some limitations that prevent us from applying this ideal "strategy", namely that an algorithmic model may typically employ loop constructs, for which B does not provide conditions to establish that a refinement is correct.

Fortunately, it is possible to devise another solution to circumvent this limitation without modifying or extending the B method. This solution is illustrated at the right of Figure 4: instead of establishing a refinement relation between the assembly and the algorithmic models, consider a refinement of the design model directly preceding the algorithmic model in the refinement process. The construction of the assembly implementation from the algorithmic implementation should be formalized by a set of rules that can be implemented to form a B-based formal compiler, but the verification would be carried out as usually in a B refinement, with respect to the design model. Section 5 provides a series of small examples, representative of the different kinds of algoritmic constructs provided by the B method, and shows the correspondence between algorithmic and assembly implementations. They all use the assembly-like platform presented in the following.

4 A B model for an assembly-like language

In this section, we present the functional model of a minimal processor. Its actual definition in the B notation is interspersed with an informal description of the model.

The essence of current micro-processors and micro-controllers is captured by the Random Access Machine computation model, which is Turing complete [5]. We model this machine in B so that it can be used as a computational platform with a level of abstraction equivalent to that of many assembly languages.

MACHINE ram

The variables composing the state of the machine are mem, an unbounded memory storing natural numbers, pc, the program counter, and end, that is used to detect the end of the computation.

CONCRETE_VARIABLES mem, pc, end
INVARIANT
$mem \in (\mathbb{N} \rightarrow \mathbb{N}) \wedge pc \in \mathbb{N} \wedge end \in \mathbb{N}$

When executing a program on the ram processor, we always assume that the value of the program counter is less or equal than that of the end marker. When it is equal, then the execution stops.

The machine can be initialized in any state.

INITIALISATION
$mem :\in (\mathbb{N} \rightarrow \mathbb{N}) \parallel pc :\in \mathbb{N} \parallel end :\in \mathbb{N}$

Each instruction of the ram processor is modelled as a B operation. However, before starting the execution of a program, one needs to reset the program counter and the end of program marker. This functionality is provided by the operation $init$, that takes as input the size of the program, i.e. the number of instructions. This is the necessary condition for the ram processor to be able to start executing instructions.

OPERATIONS
$init(sz) =$
PRE $sz \in \mathbb{N}$ **THEN**
$pc := 0 \parallel end := sz$
END;

The first modelled instruction is nop, the so-called no-operation. Its only effect is to increment the program counter.

$nop =$ **PRE** $pc + 1 \leq end$ **THEN** $pc := pc + 1$ **END**;

The instruction set is responsible for storing a constant value val into a memory location a. Observe that the execution of an instruction has a side effect on the program counter.

$set(a, val) =$
PRE $a \in \mathbb{N} \wedge val \in \mathbb{N} \wedge pc + 1 \leq end$ **THEN**
$mem(a) := val \parallel pc := pc + 1$
END;

Following is the inc instruction. It is the only data modifying instruction in our machine. It is responsible for incrementing the value stored at the location a of the memory.

$inc(a) =$
  PRE $a \in \mathbb{N} \wedge pc + 1 \leq end$ **THEN**
    $mem(a) := mem(a) + 1 \parallel pc := pc + 1$
  END;

The $copy$ instruction copies the value stored in address src to the address $dest$.

$copy(src, dst) =$
  PRE $src \in \mathbb{N} \wedge dst \in \mathbb{N} \wedge pc + 1 \leq end$ **THEN**
    $mem(dst) := mem(src) \parallel pc := pc + 1$
  END;

The following two instructions are testing instructions and are very similar. Instruction $testgt$ checks whether the value stored in location $a1$ is strictly greater than the value stored in location $a2$. If this is the case, the program counter is incremented once, otherwise twice. Instruction $testeq$ checks for equality of the stored values and applies the same modifications to the control flow.

$testgt(a1, a2) =$
  PRE $a1 \in \mathbb{N} \wedge a2 \in \mathbb{N} \wedge$
    $(mem(a1) > mem(a2) \Rightarrow (pc + 1 <= end)) \wedge$
    $(mem(a1) <= mem(a2) \Rightarrow (pc + 2 <= end))$
  THEN IF $mem(a1) > mem(a2)$ **THEN** $pc := pc + 1$ **ELSE** $pc := pc + 2$ **END**
  END;
$testeq(a1, a2) =$
  PRE $a1 \in \mathbb{N} \wedge a2 \in \mathbb{N} \wedge$
    $(mem(a1) = mem(a2) \Rightarrow (pc + 1 <= end)) \wedge$
    $(mem(a1) \neq mem(a2) \Rightarrow (pc + 2 <= end))$
  THEN IF $mem(a1) = mem(a2)$ **THEN** $pc := pc + 1$ **ELSE** $pc := pc + 2$ **END**
END;

The $goto$ instruction directly alters the value of the program counter. It is an unconditional branch, and is used in combination with the testing instructions to implement conditional and iteration constructions.

$goto(val) =$ **PRE** $val \in \mathbb{N} \wedge val \leq end$ **THEN** $pc := val$ **END**;

5 Compilation of B0 constructs: examples

The modelling language of the B method has a sub-language, named B0, for algorithmic models. Variable assignment as well as sequential, conditional and iterative composition form the basic constructs of B0. This section shows, through a series of increasingly complex (yet still basic) examples, how these constructs can be refined as combinations of assembly instructions of the ram processor presented in Section 4.

5.1 Sequence

The design model in the machine $sequence$ has two variables and a single operation that swaps the values of these variables.

MACHINE $sequence$
VARIABLES a, b
INVARIANT $a \in \mathbb{N}\, b \in \mathbb{N}$
INITIALISATION $a :\in \mathbb{N} \parallel b :\in \mathbb{N}$
OPERATIONS
$run =$
BEGIN
$a := b \parallel b := a$
END
END

The salient parts of a possible algorithmic model are given below. The variables are implemented in a machine nat that provides setter and getter functionalities for a variable holding a value of the set $\mathbb{N}$.

IMPORTS $ai.nat, bi.nat$
INVARIANT $a = ai.value \wedge b = bi.value$
OPERATIONS
$run =$
VARIANT tmp **VARIANT**
$tmp := bi.value;$
$bi.set(ai.value);$
$ai.set(tmp)$
END
END

Compilation to assembly includes mapping the variables to memory locations and algorithmic-level instructions to sequences of assembly instructions. The static memory allocation is formalized in the invariant of the assembly-level model. In the presented examples, the types of the model variable and of the memory locations match, so *the assembly invariant is obtained by substitution of the implementation variables by their corresponding memory location in the algorithmic invariant*. For this example, we have:

IMPORTS $uc.ram$
INVARIANT $a = uc.mem(0) \wedge b = uc.mem(1)$

The compiled assembly program is a sequence of three $copy$ instructions that perform the swap between the memory locations corresponding to the model variables. The assembly-level operation is thus:

OPERATIONS
$run =$
BEGIN
$uc.init(3);$
$uc.copy(1, 2);$
$uc.copy(0, 1);$
$uc.copy(2, 0)$
END

Although this model is a valid refinement of the initial machine, we will show, in the next example, that its construction pattern cannot be used as a general solution to build assembly-level refinements.

5.2 Conditional statement

This new example system has two operations: one to register values and one to return the largest registered value. The state is the set of all the values registered so far.

MACHINE *conditional*
VARIABLES st
INVARIANT $st \in \mathbb{P}(\mathbb{N})$
INITIALISATION $st := \emptyset$
OPERATIONS
  $add(v) =$
  PRE $v \in \mathbb{N} \wedge v \notin st$ **THEN**
    $st := st \cup \{v\}$
  END;
  $res \longleftarrow largest =$ **PRE** $st \neq \emptyset$ **THEN** $res := \max(st)$ **END**
END

This is a classical example where an abstract variable may be refined to a variable of a simpler data type. We focus our attention on the refinement of the first operation:

IMPORTS $mi.nat$
INVARIANT $\max(st) = mi.value$
OPERATIONS
  $add(v) =$
  IF $mi.value < v$ **THEN**
    $mi.set(v)$
  END

The assembly model may no longer be a sequence of assembly instructions as in the previous example. The reason is that the execution flow is not linear, due to the conditional construct. To cope with this case, the assembly model is built as the execution model of the ram machine: based on the current value of the program counter, an instruction is fetched and executed. This action is repeated until the program counter reaches the end of the program:

IMPORTS $uc.ram$
INVARIANT $max(st) = uc.mem(0)$
OPERATIONS
  $add(v) =$
  BEGIN
    $uc.init(3)$;
    WHILE $uc.pc < uc.end$ **DO**
    BEGIN
      CASE $uc.pc$ **OF**
        EITHER 0 **THEN** $uc.set(1, v)$
        OR 1 **THEN** $uc.testgt(1, 0)$
        OR 2 **THEN** $uc.copy(1, 0)$
        END
      END
    END
    INVARIANT ...(see below)
    VARIANT ...(see below)
    END
  END;
  $res \longleftarrow largest = res \longleftarrow uc.mem(0)$
END

This strategy needs a loop construct that halts when the program counter reaches the end marker. We call this the *fetch loop*: it associates each possible value of the program counter with the corresponding assembly instruction. The local variable pc maintains the value of the program counter of the ram machine. The local variable end stores the end marker of the program and remains constant. Both variables are employed in the formulation of the variant and invariant of the loop. As the algorithmic model does not jump backwards, the variant can be expressed as the distance between the end of the program and the value of the program counter.

The invariant of the fetch loop establishes the relationship between the variables of the ram machine and the local variables of the operation and the state of the ram memory for each possible valuation of the program counter:

INVARIANT
$0 \leq uc.pc \wedge uc.pc <= uc.end \wedge$
$(uc.pc = 0 \Rightarrow uc.mem(0) = max(st)) \wedge$
$(uc.pc = 1 \Rightarrow uc.mem(0) = max(st) \wedge uc.mem(1) = v) \wedge$
$(uc.pc = 2 \Rightarrow uc.mem(0) = max(st) \wedge uc.mem(1) = v \wedge max(st) < v) \wedge$
$(uc.pc = 3 \Rightarrow uc.mem(0) = max(v, max(st)))$
VARIANT $uc.end - uc.pc$

5.3 Loop

The last example implements the addition of two integers as iterated increments (excerpts):

MACHINE $iteration$
VARIABLES a, b, s
INVARIANT $a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge s \in \mathbb{N} \wedge (a + b) \in \mathbb{N}$
OPERATIONS
$run = s := a + b$
END

Recall that the ram machine only has an increment operation, and addition needs to be implemented iteratively, as in the following algorithmic model (excerpts):

IMPORTS $ai.nat, bi.nat, si.nat$
INVARIANT $a = ai.value \wedge b = bi.value \wedge s = si.value$
OPERATIONS
$run =$
VARIANT $partial, i$ **VARIANT**
$partial := ai.value;$
$i := 0;$
WHILE $i \neq bi.value$ **DO**
BEGIN
$partial := partial + 1;$
$i := i + 1$
END
INVARIANT $i \in \mathbb{N} \wedge i \leq bi.value \wedge partial = ai.value + i$
VARIANT$(bi.value - i)$
END;
$si.set(partial)$
END
END

Based on this algorithm, we devise the following assembly-level model of the addition algorithm by iterated increments (excerpts):

INVARIANT $a = uc.mem(0) \wedge b = uc.mem(1) \wedge s = uc.mem(2)$
OPERATIONS
$run =$
BEGIN
$uc.init(7)$;
$i := 0$;
WHILE $uc.pc < uc.end$ **DO**
BEGIN
CASE $uc.pc$ **OF**
EITHER 0 **THEN** $uc.copy(0, 2)$
OR 1 **THEN** $uc.set(3, 0)$
OR 2 **THEN** $uc.testeq(1, 3)$
OR 3 **THEN** $uc.goto(7)$
OR 4 **THEN** $uc.inc(2)$
OR 5 **THEN** $uc.inc(3)$
OR 6 **THEN**
BEGIN
$uc.goto(2); i := i + 1$
END
END
END
END
INVARIANT
$0 \leq uc.pc \wedge uc.pc \leq uc.end \wedge i \in \mathbb{N} \wedge$
$uc.mem : (\mathbb{N} --> \mathbb{N}) \wedge uc.mem(0) = a \wedge uc.mem(1) = b \wedge$
$(uc.pc = 0 \Rightarrow i = 0 \wedge uc.mem(2) = s) \wedge$
$(uc.pc = 1 \Rightarrow i = 0 \wedge uc.mem(2) = a) \wedge$
$(uc.pc = 2 \Rightarrow (uc.mem(2) = a + uc.mem(3) \wedge uc.mem(3) = i \wedge i \leq b)) \wedge$
$(uc.pc = 3 \Rightarrow (uc.mem(2) = a + uc.mem(3) \wedge uc.mem(3) = i \wedge i = b)) \wedge$
$(uc.pc = 4 \Rightarrow (uc.mem(2) = a + uc.mem(3) \wedge uc.mem(3) = i \wedge i < b)) \wedge$
$(uc.pc = 5 \Rightarrow (uc.mem(2) = a + uc.mem(3) + 1 \wedge uc.mem(3) = i \wedge i < b)) \wedge$
$(uc.pc = 6 \Rightarrow (uc.mem(2) = a + uc.mem(3) \wedge uc.mem(3) = i + 1 \wedge i < b)) \wedge$
$(uc.pc = 7 \Rightarrow (uc.mem(2) = a + uc.mem(3) \wedge uc.mem(3) = i \wedge i = b))$
VARIANT $pgvar(uc.pc, uc.mem(1), i)$
END
END
END

This assembly model follows the same pattern as the previous example. Instructions 0 and 1 codify the preamble, 2 to 6 the fetch loop, and 7 is the end of the program execution. Note that we need a variable, here called i, to keep track of the number of times the increment loop has been executed. Its value is initially zero and it is incremented whenever the program counter is 6, i.e. when the algorithm jumps back to the evaluation of the loop condition. The role of this variable is to help show the correctness of the refinement (it is only accessed in the invariant); it will not be represented in the assembly program. Also, observe that the invariant of the fetch loop states that when the value of the program counter is two, the algorithmic loop condition shall hold.

The variant is the value of a function application to the three program elements pc, the program counter, $uc.mem(1)$, the value of b, and i, the number of times the increment iteration

of the algorithm has been executed. The function is called $pgvar$ (program variant) and is defined in the following machine:

MACHINE $iterationasmvariant$
CONSTANTS $pgvar, pgsz, la, lsz$
PROPERTIES
$pgsz \in \mathbb{N} \wedge la \in \mathbb{N} \wedge lsz \in \mathbb{N} \wedge$
$pgvar \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} +- > \mathbb{N} \wedge$
$pgsz = 7 \wedge lsz = 5 \wedge la = 2 \wedge$
$\forall pc, b, i \bullet$
$(pc \in \mathbb{N} \wedge b \in \mathbb{N} \wedge i \in \mathbb{N} \Rightarrow$
$(pc = 7 \Rightarrow pgvar(pc, b, i) = pgsz - pc) \wedge$
$((pc = 0 or pc = 1) \Rightarrow$
$pgvar(pc, b, i) = pgsz - pc + lsz * b) \wedge$
$((pc = 2 or pc = 3 or pc = 4 or pc = 5 or pc = 6) \Rightarrow$
$pgvar(pc, b, i) = pgsz - la + lsz * (b - i) - (pc - la)))$
END

The expression of the function body is quite complex and, for the sake of clarity, uses three numeric constants: $pgsz$ the total number of instructions in the program, lsz the size of the block codifying the loop and la the address of its first instruction. The value returned by $pgvar$ is greater or equal to the number of instructions that remain to be executed. The corresponding expression needs to be derived from the loop variant of the algorithmic level model.

5.4 Lessons learnt

The three presented examples form a basic, yet fully expressive, language. These assembly models have been built manually and the refinement relation between these models and the initial design models has been checked using the B4free tool through the Click'n'Prove interface [1]. Note that, in general, it is not possible, due to the rules of the B method, to establish a B refinement of a so-called algorithmic model into an assembly model. Our intuition is that it would be easier, as the semantic gap is narrower.

In the general case, deriving the assembly program from the algorithmic models involves applying classical compilation techniques to map algorithmic variables and instructions to memory locations and a sequence of assembly instructions respectively. We have seen that the elements of a verifiable assembly level refinement are:

- The model invariant formalizes the mapping of variables to memory locations, and is obtained by applying, in the algorithmic invariant, substitution of the algorithmic variables by the corresponding memory locations,
- Each model operation is a so-called fetch loop that associates the possible values of the program counter with the corresponding assembly instructions,
- Fetch loop invariants specify the state of the assembly machine for each possible value of the program counter. It may be derived from the model invariant, the semantics of assembly instructions, and the possible loop invariants of the algorithmic model.
- Fetch loop variants express an upper bound of the number of instructions that remain to be executed, for each possible value of the program counter. The expression of such functions may be derived using static code analysis such as worst-case execution time techniques [11].

6 Case study on an industrial platform

6.1 Modelling the PIC16C432 with B

The principles developped in the previous section are now applied to a real computing platform: the PIC16C432 micro-controller. This is a simple, low-cost solution to run embedded software in a variety of applications.

The B model of the PIC platform is structured in several modules:

- Module PIC contains the specification of the state of the micro-controller and its intruction set. Details are given in Section 6.1.1.
- Module ALU provides the definitions of the different arithmetic and logic functions used in the specification the PIC instruction set.
- Module $TYPES$ holds the definitions of the different data types that are employed in the specification of the PIC, such as data words, memory addresses, etc. This module includes itself two other auxiliary modules containing definitions to represent and manipulate bits and bit vectors of size eight (the width of the PIC).

6.1.1 The module PIC

The model starts with the name of the main module and references to the modules ALU and $TYPES$ that contain the referenced definitions.

MACHINE PIC
SEES $ALU, TYPES$

The semantics of the instruction set is given based on the update of the following state components of the PIC:

- w, the working register;
- z, a zero status bit;
- c, a carry status bit;
- pc, the program counter;
- $stack$, the execution stack;
- mem, the data memory.

The model of the PIC state is thus:

VARIABLES $w, z, c, pc, sp, stack, mem$
INVARIANT
$w \in WORD \ \wedge \ z \in BOOL \ \wedge \ c \in BOOL \wedge$
$mem \in REGISTER \rightarrow WORD \wedge$
$pc \in INSTRUCTION \wedge$
$stack \in \mathbb{N} \rightarrowtail INSTRUCTION \wedge$
$sp \in \mathbb{N} \ \wedge \ \mathrm{dom}(stack) = 0..(sp - 1)$

In this definition, $WORD$, $REGISTER$ and $INSTRUCTION$ are respectively the names of the type for data words, memory addresses and instruction memory addresses. Their definitions are given in the module $TYPES$. Note that the model does not contain the instruction memory itself, but only the value of the program counter.

Each assembly instruction is modeled as an operation in the B module. These instructions are classified as: data copy, logic and arithmetic operations, bit-level operations, and control flow. Only a representative subset of the instructions are presented here. The full model is given in appendix A.

The next two operations model data copies instructions:

- Operation $MOVLW$ models the assignment of a given value k to the working register w.

$MOVLW(k) =$
PRE $k \in WORD$ **THEN**
$w := k \parallel pc := INSTRUCTION\_NEXT(pc)$
END;

- Operation $MOVWF$ models the instruction to copy a word stored in the working register to a given data memory address f.

$MOVWF(f) =$
PRE $f \in REGISTER$ **THEN**
$mem(f) := w \parallel pc := INSTRUCTION\_NEXT(pc)$
END;

Note that both instructions increment the program counter unconditionally.

The instruction set of the PIC16C432 includes sum, subtraction, conjunction, disjunction and exclusive disjunction. They are all binary operations and are modelled in a similar fashion. We restrict our presentation to the model of the summation instructions. Each such operation has two versions:

- The first version models an instruction with a single argument, which is a constant value k. The instruction combines k with the value stored in the working register and stores the result in this register.

$ADDLW(k) =$
PRE $k \in WORD$ **THEN**
ANY $result, carry, zero$ **WHERE**
$result \in WORD \wedge carry \in BOOL \wedge zero \in BOOL \wedge$
$result, carry, zero = add(k, w)$
THEN
$w, c, z := result, carry, zero$
END $\parallel$
$pc := INSTRUCTION\_NEXT(pc)$
END

- The second version has two arguments: f, an address in data memory, and a bit d. The value in f is combined with the value in the working register. If d is set, then the result is stored in the working register, otherwise it is stored in the memory at the address f.

$ADDWF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
ANY $result, carry, zero$ **WHERE**
$result \in WORD \wedge carry \in BOOL \wedge zero \in BOOL \wedge$
$result, carry, zero = add(mem(f), w)$
THEN
IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** $\|$
$c := carry \| z := zero$
END $\|$
$pc := INSTRUCTION\_NEXT(pc)$
END

In both versions, the status z and c are assigned to indicate respectively if the result of the operation was zero, and if there has been an overflow. In the presented modelling, the combination is performed with a function named add, the definition of which is given in module ALU (details are provided in Section 6.1.2).

Next are presented the operations that model control flow instructions.

- Operation $GOTO$ models the unconditional jump and is simply defined as:

$GOTO(k) =$
PRE $k \in INSTRUCTION$ **THEN**
$pc := k$
END;

- Operations $CALL$ and $RETURN$ model instructions to respectively call and return from a procedure. The corresponding B operations specify how the state of the stack and the program counter are altered by these instructions:

$CALL(k) =$
PRE $k \in INSTRUCTION$ **THEN**
$stack(sp) := INSTRUCTION\_NEXT(pc) \| sp := sp + 1 \| pc := k$
END;
$RETURN =$
PRE $sp > 0$ **THEN**
$pc := stack(sp - 1) \| stack := 0..(sp - 2) \lhd stack \| sp := sp - 1$
END;

- The instruction set of the PIC16C432 has different test instructions that provide conditional jumps of the program counter. For instance, the operation $BTFSC$ models an instruction with two parameters: a memory address f and a position b. If the b-th bit of the word stored in address f is not set, then the next instruction is not executed (the program counter is incremented twice), otherwise, the next execution is executed.:

$BTFSC(f, b) =$
PRE $f \in REGISTER \wedge b \in WORD\_POSITION$ **THEN**
IF $bitget(mem(f), b) = 0$ **THEN**
$pc := INSTRUCTION\_NEXT(INSTRUCTION\_NEXT(pc))$
ELSE
$pc := INSTRUCTION\_NEXT(pc)$
END
END;

Note that the definition of this operation uses an auxiliary function, named $bitget$, which is specified in module ALU, presented in the following.

6.1.2 The module ALU

The ALU module defines several mathematic functions that are used to define the model of the PIC16C432 micro-controller. The definitions are based on the $TYPES$ module that contains the definitions for the types of the domain and the range of such functions. For instance, the ALU module defines the constants add and $bitget$, that have both functional type and value. They are thus defined in the $PROPERTIES$ clause of the module:

$$
\begin{array}{l}
add \in (WORD \times WORD) \longrightarrow (WORD \times BOOL \times BOOL) \wedge \\
\forall w_1, w_2, s \bullet \\
\quad (w_1 \in WORD \wedge w_2 \in WORD \wedge s \in \mathbb{N} \wedge s = w_1 + w_2 \Rightarrow \\
\qquad ((s \leq 255 \Rightarrow add(w_1, w_2) = (s, FALSE, bool(s = 0))) \\
\qquad \wedge (256 \geq s \Rightarrow add(w_1, w_2) = (s - 256, TRUE, bool(s = 256))))) \wedge \\
bitget \in WORD \times WORD\_POSITION \longrightarrow BIT \wedge \\
\forall w, i \bullet (w \in WORD \wedge i \in WORD\_POSITION \Rightarrow bitget(w, i) = WORD\_TO\_BV(w)(i))
\end{array}
$$

6.2 Case study: traffic light

The functional model of the traffic light (Section 2.3) and its refinement (Section 2.4), are used as a case study to derive two refinements: the first is at an algorithmic level of detail and corresponds to a traditional B implementation (see Figure 1), and the second is an assembly level refinement towards the PIC16C432 micro-controller, modelled in Section 6.1.1.

6.2.1 Refinement at the algorithmic level

This refinement implements the traffic light model using the usual concepts from the B method. The refinement variable $count$ is implemented as an instance, named $state$, of the machine nat, that models the store of a natural number.

IMPLEMENTATION *traffic_light_alg*
REFINES *traffic_light_data_refinement*
IMPORTS $state.nat$
INVARIANT $state.value \in 0..2 \wedge state.value = count$
INITIALISATION $state.set(0)$
OPERATIONS
$advance =$
  IF $state.value = 0$ **THEN** $state.set(1)$
  ELSIF $state.value = 1$ **THEN** $state.set(2)$
  ELSE $state.set(0)$ **END**
END

6.2.2 Refinement at the assembly level

The assembly-level refinement is longer than the algorithmic-level and is presented incrementally. The state is an instance of the PIC machine named m. The invariant establishes the relationship with the refined model (see Figure 3): the value of $count$ is stored at address 0 of the data memory. Once the algorithmic level variables have been mapped to the micro-controller's memory, we obtain the invariant of the refinement by substitution of the algorithmic

variables with their corresponding memory locations. Thus variable $state.value$ from the module *traffic_light_alg* is substituted with $m.mem(0)$.

IMPLEMENTATION *traffic_light_asm_pic*
REFINES *traffic_light_data_refinement*
IMPORTS $m.PIC$
INVARIANT $m.mem(0) \in 0..2 \wedge m.mem(0) = count$

Initialization shall set the value of the memory location 0 to the value 0. This is done by a sequence of two assembly instructions: first the value 0 is stored into the working register w, then the value in w is copied to address 0. Here the refinement proof is fully automatic:

INITIALISATION
  BEGIN
    $m.MOVLW(0)$;
    $m.MOVWF(0)$
  END

The operation $advance$ may be implemented on the PIC16C432 by the following assembly program (for the sake of clarity, instructions have been numbered and commented):

```
0: BTFSC 0,1     ; tests if location 0 is 2
1: GOTO 8        ; jumps to instruction 8
2: BTFSC 0,0     ; tests if location 0 is 1
3: GOTO 6        ; jumps to instruction 6
4: MOVLW 1       ; sets w to 1
5: GOTO 9        ; jumps to instruction 9
6: MOVLW 2       ; sets w to 2
7: GOTO 9        ; jumps to instruction 9
8: MOVLW 0       ; sets w to 0
9: MOVWF 0       ; copies w to location 0
```

Such program has been derived from the algorithmic model manually, but could be automated using conventional compilation techniques. Note that the control flow of the assembly program is not linear as it includes conditional and unconditional jumps. The assembly-level refinement of the $advance$ operation[1] is then built using the ideas presented in Section 5:

$advance =$
  BEGIN
    $\underline{m.CALL(0)}$;
    WHILE $m.pc < 10$ **DO**
    BEGIN
      CASE $m.pc$ **OF**
        EITHER 0 **THEN** $\underline{m.BTFSC(0,1)}$
        OR 1 **THEN** $\underline{m.GOTO(8)}$
        OR 2 **THEN** $\underline{m.BTFSC(0,0)}$
        OR 3 **THEN** $\underline{m.GOTO(6)}$
        OR 4 **THEN** $\underline{m.MOVLW(1)}$
        OR 5 **THEN** $\underline{m.GOTO(9)}$
        OR 6 **THEN** $\underline{m.MOVLW(2)}$
        OR 7 **THEN** $\underline{m.GOTO(9)}$
        OR 8 **THEN** $\underline{m.MOVLW(0)}$
        OR 9 **THEN** $\underline{m.MOVWF(0)}$
        END
      END
    END
    INVARIANT ... (see below)
    VARIANT ... (see below)
    END;
    $\underline{m.RETURN}$
  END
END

[1] Instances of the PIC instructions have been underlined.

The invariant of the fetch loop follows. For each possible value of the program counter, the invariant states the relationship between the state of the refinement and the state of the refined module. When the program counter is zero, then this relationship coincides with the module invariant. Subsequent relationships were derived manually using the semantics of the corresponding assembly instructions.

INVARIANT
$0 \leq m.pc \wedge m.pc \leq 10 \wedge m.sp > 0 \wedge$
$m.mem(0) \in 0..2 \wedge$
$(m.pc = 0 \Rightarrow (m.mem(0) \in 0..2 \wedge m.mem(0) = m.count)) \wedge$
$(m.pc = 1 \Rightarrow m.mem(0) = 2 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 2 \Rightarrow m.mem(0) \neq 2 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 3 \Rightarrow m.mem(0) = 1 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 4 \Rightarrow m.mem(0) = 0 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 5 \Rightarrow m.mem(0) = 0 \wedge m.mem(0) = m.count \wedge m.w = 1) \wedge$
$(m.pc = 6 \Rightarrow m.mem(0) = 1 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 7 \Rightarrow m.mem(0) = 1 \wedge m.mem(0) = m.count \wedge m.w = 2) \wedge$
$(m.pc = 8 \Rightarrow m.mem(0) = 2 \wedge m.mem(0) = m.count) \wedge$
$(m.pc = 9 \Rightarrow (m.mem(0) \in 0..2 \wedge m.mem(0) = m.count \wedge m.w = color\_step(m.count))) \wedge$
$(m.pc = 10 \Rightarrow (m.mem(0) \in 0..2 \wedge m.mem(0) = color\_step(m.count)))$

There is no back loop in this assembly program and it is very easy to derive a loop variant from the distance between the current instruction and the last instruction of the program:

VARIANT$(10 - m.pc)$

The correctness of the refinement was verified with B4free's provers through the Click'n'Prove interface. Most proof obligations were verified automatically, without user intervention. For the remainding proof obligations, the necessary interaction was generally reduced to a minimum (one or two steps of quantifier instantiation and hypothesis selection). A single proof obligation, involving arithmetic, required introducing basic rules to define exponentiation, which seemed to be unknown from the prover, as well as considerably longer interactions to establish the required lemmas.

7 Conclusion and future work

In this paper, we identified a potential way to apply the B method to develop provably correct assembly level implementations and, therefore to address in part the challenge of the *verifying compiler* [4]. We have presented the results of an initial appraisal of the feasability of this approach.

We designed, in B, a formal model inspired from the Random Access Machine, a computational model that is similar to that of mainstream micro-controllers and micro-processors. Using as a target the model of this machine, we developed three assembly level B implementations of increasing complexity that employ four basic, yet computationally complete, algorithmic constructs: assignment, sequence, conditional and loop. We obtained thus a template for assembly level models for the B method. This template shall be instantiated by a mapping between algorithm variables and the machine memory addresses, the compilation of the algorithm in a sequence of assembly instructions, the construction of the assembly program variant and invariant, based on the memory layout mapping, the algorithmic model invariants

and variants, and the assembly instruction semantics. These ideas have been applied to develop a provably correct assembly-level refinement on an industrial platform for a small case study.

This initial appraisal shows the feasibility of the B-based approach to build a verifying compiler, thereby taking advantage of a large body of techniques and tools to address this grand challenge.

Several directions of work need to be developed to meet the goals of the Tony Hoare's challenge. First, we shall develop B models of actual computing platforms[2]. Second, we shall define formal rules for the construction of assembly level models from algorithmic models, and implement these rules in a prototype tool. The most challenging aspects include the construction of variant and invariant annotations. Another line of work would be to build modules at an intermediate levels of abstraction, corresponding to internal representations employed in compilers, that would make it easier to target different computing platforms within a single development. Finally, to get actual and useful tools that answer T. Hoare's challenge, this work must be integrated with other approaches, such as [9].

References

[1] ABRIAL, J.-R; CANSELL, D. **Click'n prove: Interactive proofs within set theory**. In: TPHOLS, p. 1–24, 2003.

[2] ABRIAL, R. **The B-Book: Assigning programs to meanings**. Cambridge University Press, 1996.

[3] CLEARSY. **Atelier B**. http://www.atelierb.eu.

[4] HOARE, C. A. R. **The verifying compiler, a grand challenge for computing research**. In: VMCAI, p. 78–78, 2005.

[5] HOPCROFT, J. E; ULLMAN, J. D. **Introduction to Automata Theory, Languages and Computation**. Addison Wesley, 1979.

[6] JAFFUEL, E; LEGEARD, B. **LEIRIOS test generator: Automated test generation from B models.** In: THE 7TH INTERNATIONAL B CONFERENCE, p. 277–280, 2007.

[7] LEROY, X. **Formal certification of a compiler back-end or: programming a compiler with a proof assistant**. SIGPLAN Not., 41(1):42–54, 2006.

[8] LTD, B.-C. **The b-toolkit**. http://www.b-core.com/btoolkit.html.

[9] NECULA, G. C; LEE, P. **The design and implementation of a certifying compiler**. SIGPLAN Not., 33(5):333–344, 1998.

[10] OSSAMI, D. D; JACQUOT, J.-P; SOUQUIÈRES, J. **Consistency in uml and b multi-view specifications**. In: IFM, p. 386–405, 2005.

[11] PARK, C. **Predicting program execution times by analyzing static and dynamic program paths**. Real-Time Systems, 5(1):31–61, 1993.

[2] A model of a large part of a PIC micro-controller is available through the public site `http://code.google.com/p/b2asm/`. Complete versions and project files of the examples presented in this paper are also available through this site.

[12] SNOOK, C; BUTLER, M. **Uml-b: Formal modelling and design aided by uml**. ACM Transactions on Software Engineering and Methodology, 15(1):92–122, 2006.

[13] SPIVEY, J. **The Z Notation: a Reference Manual**. Prentice-Hall International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[14] WORDSWORTH, J. B. **Software Engineering with B**. Addison Wesley, Boston, 1996.

A Full B model of the PIC16C432

A.1 The $TYPE\_BIT$ module

MACHINE $TYPE\_BIT$
CONSTANTS $BIT, BIT\_FLIP, BIT\_AND, BIT\_IOR, BIT\_XOR$
PROPERTIES
$BIT = 0..1$
$\wedge\ BIT\_FLIP \in BIT \rightarrow BIT \wedge \forall b \bullet (b \in BIT \Rightarrow BIT\_FLIP(b) = 1 - b)$
$\wedge\ BIT\_AND \in BIT \times BIT \rightarrow BIT$
$\wedge\ \forall b1, b2 \bullet (b1 \in BIT \wedge b2 \in BIT \Rightarrow ((BIT\_AND(b1, b2) = 1) \Leftrightarrow (b1 = 1) \wedge (b2 = 1)))$
$\wedge\ BIT\_IOR \in BIT \times BIT \rightarrow BIT$
$\wedge\ \forall b1, b2 \bullet (b1 \in BIT \wedge b2 \in BIT \Rightarrow ((BIT\_IOR(b1, b2) = 1) \Leftrightarrow (b1 = 1) \vee (b2 = 1)))$
$\wedge\ BIT\_XOR \in BIT \times BIT \rightarrow BIT$
$\wedge\ \forall b1, b2 \bullet (b1 \in BIT \wedge b2 \in BIT \Rightarrow$
$((BIT\_XOR(b1, b2) = 1) \Leftrightarrow (((b1 = 1) \wedge (b2 = 0)) \vee ((b1 = 0) \wedge (b2 = 1)))))$
ASSERTIONS
$BIT\_FLIP(0) = 1 \wedge BIT\_FLIP(1) = 0$
$\wedge\ BIT\_AND(0, 0) = 0 \wedge BIT\_AND(0, 1) = 0 \wedge BIT\_AND(1, 0) = 0 \wedge BIT\_AND(1, 1) = 1$
$\wedge\ BIT\_IOR(0, 0) = 0 \wedge BIT\_IOR(0, 1) = 0 \wedge BIT\_IOR(1, 0) = 0 \wedge BIT\_IOR(1, 1) = 1$
$\wedge\ BIT\_XOR(0, 0) = 0 \wedge BIT\_XOR(0, 1) = 1 \wedge BIT\_XOR(1, 0) = 1 \wedge BIT\_XOR(1, 1) = 0$
END

A.2 The $TYPE\_BV8$ module

MACHINE $TYPE\_BV8$
SEES $TYPE\_BIT$
CONSTANTS
$BV8\_INDEX, BV8, BV8\_SET\_BIT, BV8\_COMPLEMENT, BV8\_ALL\_ZEROES,$
$BV8\_AND, BV8\_IOR, BV8\_XOR$
PROPERTIES $BV8\_INDEX = 0..7$
$\wedge\ BV8 = BV8\_INDEX \rightarrow BIT$
$\wedge\ BV8\_SET\_BIT \in (BV8 \times BV8\_INDEX \times BIT) \rightarrow BV8$
$\wedge\ \forall v, i, j, b \bullet (v \in BV8 \wedge i \in BV8\_INDEX \wedge j \in BV8\_INDEX \wedge b \in BIT \Rightarrow$
$(i \neq j \Rightarrow BV8\_SET\_BIT(v, i, b)(j) = v(j)))$
$\wedge\ \forall v, i, b \bullet (v \in BV8 \wedge i \in BV8\_INDEX \wedge b \in BIT \Rightarrow BV8\_SET\_BIT(v, i, b)(i) = b)$
$\wedge\ BV8\_COMPLEMENT \in BV8 \rightarrow BV8$
$\wedge\ \forall v, i \bullet (v \in BV8 \wedge i \in BV8\_INDEX \Rightarrow BV8\_COMPLEMENT(v)(i) = BIT\_FLIP(v(i)))$
$\wedge\ BV8\_ALL\_ZEROES \in BV8$
$\wedge\ \forall i \bullet (i \in BV8\_INDEX \Rightarrow BV8\_ALL\_ZEROES(i) = 0)$
$\wedge\ BV8\_AND \in BV8 \times BV8 \rightarrow BV8$
$\wedge\ \forall v1, v2, i \bullet (v1 \in BV8 \wedge v2 \in BV8 \wedge i \in BV8\_INDEX \Rightarrow$
$BV8\_AND(v1, v2)(i) = BIT\_AND(v1(i), v2(i)))$
$\wedge\ BV8\_IOR \in BV8 \times BV8 \rightarrow BV8$
$\wedge\ \forall v1, v2, i \bullet (v1 \in BV8 \wedge v2 \in BV8 \wedge i \in BV8\_INDEX \Rightarrow$
$BV8\_IOR(v1, v2)(i) = BIT\_IOR(v1(i), v2(i)))$
$\wedge\ BV8\_XOR \in BV8 \times BV8 \rightarrow BV8$
$\wedge\ \forall v1, v2, i \bullet (v1 \in BV8 \wedge v2 \in BV8 \wedge i \in BV8\_INDEX \Rightarrow$
$BV8\_XOR(v1, v2)(i) = BIT\_XOR(v1(i), v2(i)))$
END

A.3 The $TYPES$ module

MACHINE $TYPES$
INCLUDES $TYPE\_BIT$, $TYPE\_BV8$
CONSTANTS $WORD\_LENGTH$, $WORD$, $WORD\_POSITION$, NB_WORDS,
$INST\_SZ$, $INSTRUCTION$, $NB\_INSTRUCTIONS$,
$INSTRUCTION\_MAX$, $INSTRUCTION\_NEXT$,
BV_TO_WORD, $WORD\_TO\_BV$, $REGISTER$
PROPERTIES
$WORD\_LENGTH \in \mathbb{N} \wedge WORD\_LENGTH = 8 \wedge$
$NB\_WORDS \in \mathbb{N} \wedge NB\_WORDS = 2^{WORD\_LENGTH} \wedge$
$WORD = 0..(NB\_WORDS - 1) \wedge$
$WORD\_POSITION = 0..(WORD\_LENGTH - 1) \wedge$
$INST\_SZ \in \mathbb{N} \wedge INST\_SZ = 8 \wedge$
$NB\_INSTRUCTIONS \in \mathbb{N} \wedge NB\_INSTRUCTIONS = 2^{INST\_SZ} \wedge$
$INSTRUCTION\_MAX = NB\_INSTRUCTIONS - 1 \wedge$
$INSTRUCTION = 0..INSTRUCTION\_MAX \wedge$
$INSTRUCTION\_NEXT \in INSTRUCTION \rightarrow INSTRUCTION \wedge$
$\forall i \bullet (i \in INSTRUCTION \Rightarrow ((i = 255 \Rightarrow INSTRUCTION\_NEXT(i) = 0) \wedge$
$(i < 255 \Rightarrow INSTRUCTION\_NEXT(i) = i + 1))) \wedge$
$BV\_TO\_WORD \in BV8 \rightarrow WORD \wedge$
$WORD\_TO\_BV \in WORD \rightarrow BV8 \wedge$
$\forall w, v \bullet (w \in WORD \wedge v \in BV8 \Rightarrow$
$((v = WORD\_TO\_BV(w)) \Leftrightarrow$
$(w = 128 \times v(7) + 64 \times v(6) + 32 \times v(5) + 16 \times v(4) +$
$8 \times v(3) + 4 \times v(2) + 2 \times v(1) + v(0)))) \quad \wedge$
$BV\_TO\_WORD = WORD\_TO\_BV^{-1} \wedge$
$\forall n \bullet (n \in \mathbb{N} \wedge n > 0 \Rightarrow 2^n = 2 \times (2^{n-1})) \wedge$
$REGISTER \in \mathbb{P}(\mathbb{Z}) \wedge REGISTER = 0..127$
ASSERTIONS
$2^8 = 256 \wedge NB\_WORDS = 256 \wedge$
$\forall n \bullet (n \in WORD \Rightarrow 0 \leq n) \wedge \forall n \bullet (n \in WORD \Rightarrow n \leq 255) \wedge$
$WORD\_POSITION = BV8\_INDEX$
END

A.4 The ALU module

MACHINE ALU
SEES $TYPES$
CONSTANTS
$add, substract, and, ior, xor,$
$bitclear, bitset, bitget,$
$complement, swap, rotateleft, rotateright$
PROPERTIES
$add \in (WORD \times WORD) \rightarrow (WORD \times BOOL \times BOOL)$
$\wedge \forall (w1, w2, sum) \bullet$
$(w1 \in WORD \wedge w2 \in WORD \wedge sum \in \mathbb{N} \wedge sum = w1 + w2 \Rightarrow$
$((sum \leq 255 \Rightarrow add(w1, w2) = (sum, FALSE, bool(sum = 0))) \wedge$
$(256 \leq sum \Rightarrow add(w1, w2) = (sum - 256, TRUE, bool(sum = 256)))))$
$\wedge substract \in WORD \times WORD \rightarrow WORD \times BOOL \times BOOL$
$\wedge \forall (w1, w2, diff) \bullet$
$(w1 \in WORD \wedge w2 \in WORD \wedge diff \in \mathbb{Z} \wedge diff = w1 - w2 \Rightarrow$
$((diff < 0 \Rightarrow substract(w1, w2) = (diff + 256, FALSE, TRUE)) \wedge$
$(diff \geq 0 \Rightarrow substract(w1, w2) = (diff, bool(diff = 0), FALSE))))$
$\wedge and \in (WORD \times WORD) \rightarrow (WORD \times BOOL)$
$\wedge \forall (w1, w2, w) \bullet$
$(w1 \in WORD \wedge w2 \in WORD \wedge w \in WORD \wedge$
$w = BV\_TO\_WORD(BV8\_AND(WORD\_TO\_BV(w1), WORD\_TO\_BV(w2))) \Rightarrow$
$and(w1, w2) = (w, bool(w = 0)))$
$\wedge ior \in (WORD \times WORD) \rightarrow (WORD \times BOOL)$
$\wedge \forall (w1, w2, w) \bullet$
$(w1 \in WORD \wedge w2 \in WORD \wedge w \in WORD \wedge$
$w = BV\_TO\_WORD(BV8_IOR(WORD\_TO\_BV(w1), WORD\_TO\_BV(w2))) \Rightarrow$
$ior(w1, w2) = (w, bool(w = 0)))$
$\wedge xor \in (WORD \times WORD) \rightarrow (WORD \times BOOL)$
$\wedge \forall (w1, w2, w) \bullet$
$(w1 \in WORD \wedge w2 \in WORD \wedge w \in WORD \wedge$
$w = BV\_TO\_WORD(BV8\_XOR(WORD\_TO\_BV(w1), WORD\_TO\_BV(w2))) \Rightarrow$
$xor(w1, w2) = (w, bool(w = 0)))$
$\wedge bitget \in WORD \times WORD\_POSITION \rightarrow BIT$
$\wedge \forall (w, i) \bullet (w \in WORD \wedge i \in WORD\_POSITION \Rightarrow$
$bitget(w, i) = WORD\_TO\_BV(w)(i))$
$\wedge bitset \in WORD \times WORD\_POSITION \rightarrow WORD$
$\wedge \forall (w, i) \bullet (w \in WORD \wedge i \in WORD\_POSITION \Rightarrow$
$bitset(w, i) = BV\_TO\_WORD(BV8\_SET\_BIT(WORD\_TO\_BV(w), i, 1)))$
$\wedge bitclear \in WORD \times WORD\_POSITION \rightarrow WORD$
$\wedge \forall (w, i, b) \bullet (w \in WORD \wedge i \in WORD\_POSITION \wedge b \in BIT$
$\Rightarrow bitclear(w, i) = BV\_TO\_WORD(BV8\_SET\_BIT(WORD\_TO\_BV(w), i, 0)))$
$\wedge complement \in WORD \rightarrow WORD$
$\wedge \forall (w) \bullet (w \in WORD \Rightarrow$
$complement(w) = BV\_TO\_WORD(BV8\_COMPLEMENT(WORD\_TO\_BV(w))))$
$\wedge swap \in WORD \rightarrow WORD$
$\wedge \forall (w, v) \bullet (w \in WORD \wedge v \in BV8 \Rightarrow$
$(v = WORD\_TO\_BV(w) \Rightarrow$
$swap(w) = BV\_TO\_WORD(\{\ 0 \mapsto v(4), 1 \mapsto v(5), 2 \mapsto v(6), 3 \mapsto v(7),$
$4 \mapsto v(0), 5 \mapsto v(1), 6 \mapsto v(2), 7 \mapsto v(3)\})))$

$\wedge\, rotateleft \in WORD \rightarrow WORD \times BOOL$
$\wedge\, \forall(w,v) \bullet (w \in WORD \wedge v \in BV8 \Rightarrow$
$(v = WORD\_TO\_BV(w) \Rightarrow$
$rotateleft(w) = BV\_TO\_WORD(\{\ 0 \mapsto v(7), 1 \mapsto v(0), 2 \mapsto v(1), 3 \mapsto v(2),$
$4 \mapsto v(3), 5 \mapsto v(4), 6 \mapsto v(5), 7 \mapsto v(6)\}),$
$bool(v(7) = 1)))$
$\wedge\, rotateright \in WORD \rightarrow WORD \times BOOL$
$\wedge\, \forall(w,v) \bullet (w \in WORD \wedge v \in BV8 \Rightarrow$
$(v = WORD\_TO\_BV(w) \Rightarrow$
$rotateright(w) = BV\_TO\_WORD(\{\ 0 \mapsto v(1), 1 \mapsto v(2), 2 \mapsto v(3), 3 \mapsto v(4),$
$4 \mapsto v(5), 5 \mapsto v(6), 6 \mapsto v(7), 7 \mapsto v(0)\}),$
$bool(v(0) = 1)))$

ASSERTIONS
dom$(add) = WORD \times WORD \wedge$
ran$(add) \subseteq WORD \times BOOL \times BOOL \wedge$
dom$(substract) = WORD \times WORD \wedge$
ran$(substract) \subseteq WORD \times BOOL \times BOOL \wedge$
dom$(and) = WORD \times WORD \wedge$
ran$(and) \subseteq WORD \times BOOL \wedge$
dom$(ior) = WORD \times WORD \wedge$
ran$(ior) \subseteq WORD \times BOOL \wedge$
dom$(xor) = WORD \times WORD \wedge$
ran$(xor) \subseteq WORD \times BOOL \wedge$
dom$(bitclear) = WORD \times WORD_POSITION \wedge$
ran$(bitclear) \subseteq WORD \wedge$
dom$(bitset) = WORD \times WORD_POSITION \wedge$
ran$(bitset) \subseteq WORD \wedge$
dom$(bitget) = WORD \times WORD_POSITION \wedge$
ran$(bitget) \subseteq BIT \wedge$
dom$(complement) = WORD \wedge$
ran$(complement) \subseteq WORD \wedge$
dom$(swap) = WORD \wedge$
ran$(swap) \subseteq WORD \wedge$
ran$(rotateleft) \subseteq WORD \times BOOL \wedge$
dom$(rotateleft) = WORD \wedge$
dom$(rotateright) = WORD \wedge$
ran$(rotateright) \subseteq WORD \times BOOL$
END

A.5 The $PIC16C432$ module

MACHINE PIC
SEES $TYPES, ALU$
CONCRETE_VARIABLES
$mem, w, z, c, pc, stack, sp$
INVARIANT
$mem \in REGISTER \rightarrow WORD \wedge w \in WORD \wedge z \in BOOL \wedge c \in BOOL \wedge$
$pc \in INSTRUCTION \wedge sp \in \mathbb{N} \wedge$
$stack \in \mathbb{N} \rightarrow INSTRUCTION \wedge \mathrm{dom}(stack) = 0..(sp-1)$
ASSERTIONS
$\mathrm{ran}(mem) \subseteq WORD \wedge \mathrm{dom}(mem) = REGISTER$
INITIALISATION
$mem :\in REGISTER \rightarrow WORD \parallel w :\in WORD \parallel z :\in BOOL \parallel c :\in BOOL \parallel$
$pc :\in INSTRUCTION \parallel stack := \emptyset \parallel sp := 0$
OPERATIONS
$CALL(k) =$
PRE $k : INSTRUCTION$ **THEN**
$stack(sp) := INSTRUCTION\_NEXT(pc) \parallel sp := sp + 1 \parallel pc := k$
END;
$RETURN =$
PRE $sp > 0$ **THEN**
$pc := stack(sp-1) \parallel stack := 0..(sp-2) \lhd stack \parallel sp := sp - 1$
END;
$RETLW(k) =$
PRE $k \in WORD \wedge sp > 0$ **THEN**
$pc := stack(sp-1) \parallel 0..(sp-2) \lhd stack \parallel$
$sp := sp - 1 \parallel w := k$
END;
$ADDLW(k) =$
PRE $k \in WORD$ **THEN**
ANY $result, carry, zero$ **WHERE**
$result \in WORD \wedge carry \in BOOL \wedge zero \in BOOL \wedge$
$result, carry, zero = add(k, w)$
THEN
$w, c, z := result, carry, zero$
END $\parallel$
$pc := INSTRUCTION\_NEXT(pc)$
END;
$ADDWF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
ANY $result, carry, zero$ **WHERE**
$result \in WORD \wedge carry \in BOOL \wedge zero \in BOOL \wedge$
$result, carry, zero = add(mem(f), w)$
THEN
IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** $\parallel$
$c := carry \parallel z := zero$
END $\parallel$
$pc := INSTRUCTION\_NEXT(pc)$
END;

$SUBLW(k) =$
PRE $k \in WORD$ **THEN**
  ANY $result, borrow, zero$ **WHERE**
    $result \in WORD \wedge borrow \in BOOL \wedge zero \in BOOL \wedge$
    $result, borrow, zero = substract(k, w)$
  THEN
    $w, c, z := result, borrow, zero$
  END ||
  $pc := INSTRUCTION\_NEXT(pc)$
END;
$SUBWF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
  ANY $result, borrow, zero$ **WHERE**
    $result \in WORD \wedge borrow \in BOOL \wedge zero \in BOOL \wedge$
    $result, borrow, zero = substract(mem(f), w)$
  THEN
    IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** ||
    $c := borrow$ || $z := zero$
  END ||
  $pc := INSTRUCTION\_NEXT(pc)$
END;
$ANDLW(k) =$
PRE $k \in WORD$ **THEN**
  ANY $result, zero$ **WHERE**
    $result \in WORD \wedge zero \in BOOL \wedge result, zero = and(k, w)$
  THEN
    $w, z := result, zero$
  END ||
  $pc := INSTRUCTION\_NEXT(pc)$
END;
$ANDLWF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
  ANY $result, zero$ **WHERE**
    $result \in WORD \wedge zero \in BOOL \wedge result, zero = and(mem(f), w)$
  THEN
    IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END**
  END ||
  $pc := INSTRUCTION\_NEXT(pc)$
END;
$IORLW(k) =$
PRE $k \in WORD$ **THEN**
  ANY $result, zero$ **WHERE**
  $result \in WORD \wedge zero \in BOOL \wedge result, zero = ior(k, w)$
  THEN
    $w, z := result, zero$
  END ||
  $pc := INSTRUCTION\_NEXT(pc)$
END;

$IORLWF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
  ANY $result, zero$ **WHERE**
  $result \in WORD \wedge zero \in BOOL \wedge result, zero = ior(mem(f), w)$
  THEN
    IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END**
  END ||
  $pc := INSTRUCTION\_NEXT(pc)$
END;
$XORLW(k) =$
PRE $k \in WORD$ **THEN**
  ANY $result, zero$ **WHERE**
    $result \in WORD \wedge zero \in BOOL \wedge result, zero = xor(k, w)$
  THEN
    $w, z := result, zero$
  END ||
    $pc := INSTRUCTION\_NEXT(pc)$
END;
$XORLWF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
  ANY $result, zero$ **WHERE**
    $result \in WORD \wedge zero \in BOOL \wedge result, zero = xor(mem(f), w)$
  THEN
    IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END**
  END ||
  $pc := INSTRUCTION\_NEXT(pc)$
END;
$BCF(f, b) =$
PRE $f \in REGISTER \wedge b \in WORD\_POSITION$ **THEN**
  $mem(f) := bitclear(mem(f), b) \parallel pc := INSTRUCTION\_NEXT(pc)$
END;
$BSF(f, b) =$
PRE $f \in REGISTER \wedge b \in WORD\_POSITION$ **THEN**
  $mem(f) := bitset(mem(f), b) \parallel pc := INSTRUCTION\_NEXT(pc)$
END;
$BTFSC(f, b) =$
PRE $f \in REGISTER \wedge b \in WORD\_POSITION$ **THEN**
  IF $bitget(mem(f), b) = 0$ **THEN**
    $pc := INSTRUCTION\_NEXT(INSTRUCTION\_NEXT(pc))$
  ELSE
    $pc := INSTRUCTION\_NEXT(pc)$
  END
END;

$BTFSS(f, b) =$
PRE $f \in REGISTER \wedge b \in WORD\_POSITION$ **THEN**
  IF $bitget(mem(f), b) = 1$ **THEN**
    $pc := INSTRUCTION\_NEXT(INSTRUCTION\_NEXT(pc))$
  ELSE
    $pc := INSTRUCTION\_NEXT(pc)$
  END
END;
$CLRF(f) =$
PRE $f \in REGISTER$ **THEN**
  $mem(f) := 0 \parallel z := TRUE \parallel pc := INSTRUCTION\_NEXT(pc)$
END;
$CLRW =$
BEGIN
  $w := 0 \parallel z := TRUE \parallel pc := INSTRUCTION\_NEXT(pc)$
END;
$COMF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
  ANY $result$ **WHERE**
    $result = complement(mem(f))$
  THEN
    IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** $\parallel$
    $z := bool(result = 0)$
  END $\parallel$
  $pc := INSTRUCTION\_NEXT(pc)$
END;
$DECF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
  ANY $result, borrow, zero$ **WHERE**
    $result \in WORD \wedge borrow \in BOOL \wedge zero \in BOOL \wedge$
    $result, borrow, zero = substract(mem(f), 1)$
  THEN
    IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** $\parallel$
    $z := zero$
  END $\parallel$
  $pc := INSTRUCTION\_NEXT(pc)$
END;

$DECFSZ(f,d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
  ANY $result, borrow, zero$ **WHERE**
    $result \in WORD \wedge borrow \in BOOL \wedge zero \in BOOL \wedge$
    $result, borrow, zero = substract(mem(f), 1)$
  THEN
    IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** ||
    $z := zero$ ||
    IF $result = 0$ **THEN**
      $pc := INSTRUCTION\_NEXT(INSTRUCTION\_NEXT(pc))$
    ELSE
      $pc := INSTRUCTION\_NEXT(pc)$
    END
  END
END;
$INCF(f,d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
  ANY $result, carry, zero$ **WHERE**
    $result \in WORD \wedge carry \in BOOL \wedge zero \in BOOL \wedge$
    $result, carry, zero = add(mem(f), 1)$
  THEN
    IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** ||
    $z := bool(result = 0)$
  END ||
  $pc := INSTRUCTION\_NEXT(pc)$
END;
$INCFSZ(f,d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
  ANY $result, carry, zero$ **WHERE**
    $result \in WORD \wedge carry \in BOOL \wedge zero \in BOOL \wedge$
    $result, carry, zero = add(mem(f), 1)$
  THEN
    IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** ||
    IF $result = 0$ **THEN**
      $pc := INSTRUCTION\_NEXT(INSTRUCTION\_NEXT(pc))$
    ELSE
      $pc := INSTRUCTION\_NEXT(pc)$
    END
  END
END;
$GOTO(k) =$
PRE $k \in INSTRUCTION$ **THEN**
  $pc := k$
END;

$MOVLW(k) =$
PRE $k \in WORD$ **THEN**
$w := k \parallel pc := INSTRUCTION\_NEXT(pc)$
END;
$MOVF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
IF $d = 0$ **THEN** $w := mem(f)$ **END** $\parallel$
$z := bool(mem(f) = 0) \parallel pc := INSTRUCTION\_NEXT(pc)$
END;
$MOVWF(f) =$
PRE $f \in REGISTER$ **THEN**
$mem(f) := w \parallel pc := INSTRUCTION\_NEXT(pc)$
END;
$NOP = pc := INSTRUCTION\_NEXT(pc)$;
$ROLF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
ANY $result, carry$ **WHERE**
$result \in WORD \wedge carry \in BOOL \wedge$
$result, carry = rotateleft(mem(f))$
THEN
IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** $\parallel$
$c := carry$
END $\parallel$
$pc := INSTRUCTION\_NEXT(pc)$
END;
$RORF(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
ANY $result, carry$ **WHERE**
$result \in WORD \wedge carry \in BOOL \wedge$
$result, carry = rotateright(mem(f))$
THEN
IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END** $\parallel$
$c := carry$
END $\parallel$
$pc := INSTRUCTION\_NEXT(pc)$
END;
$SWAP(f, d) =$
PRE $f \in REGISTER \wedge d \in BIT$ **THEN**
ANY $result$ **WHERE** $result = swap(mem(f))$
THEN
IF $d = 0$ **THEN** $w := result$ **ELSE** $mem(f) := result$ **END**
END $\parallel$
$pc := INSTRUCTION\_NEXT(pc)$
END
END