# TAM: An Abstract Machine Specification in Z

*TAM: una especificación de máquina abstracta en Z*

Ignacio Trejos-Zelaya
Computer Science Costa Rica Institute of Technology
Costa Rica
itrejos@tec.ac.cr

Iván A. Salazar-Solano
Computer Science Costa Rica Institute of Technology
Costa Rica
ivan.a.salazar.solano@gmail.com

Jennifer Caballero
Computer Science Costa Rica Institute of Technology
Costa Rica
jennifer.caballero.solis@gmail.com

Francisco J. Torres-Rojas
Computer Science Costa Rica Institute of Technology
Costa Rica
torresrojas@gmail.com

**Fecha de recibido: 01 de marzo de 2020**

**Fecha de aprobado: 08 de abril de 2020**

**Abstract—**Finding bugs in the late stages of hardware design development is expensive. In particular, for a microprocessor architecture, unambiguity is an essential property. Formal methods can help designers to identify inconsistencies in a given system's specifications. This paper presents a formal description of a subset of the instruction set of the Triangle Abstract Machine (TAM) architecture in Z. TAM is an abstract machine suitable for the implementation of block-structured, Algol-like programming languages, such as Pascal, Modula, Oberon, and Triangle. TAM's architecture is stack-based which simplifies the code generation. Z's mathematical notation and its schema's structure help to describe logical and arithmetic instructions and also provide mechanisms suitable for modeling complex instructions that access registers, memories, and the stack. This research proposes a precise—yet abstract—approach that avoids the specification of low-level concepts such as

bits. The work reported here is a case study in formal specification applied to a Computing Science subject.

**Index Terms—**formal specification, instruction set architecture, microprocessor architecture, stack computer architecture, Z notation, TAM (Triangle Abstract Machine).

**Resumen-** Encontrar errores en las últimas etapas del desarrollo del diseño de hardware es costoso. En particular, para una arquitectura de microprocesador, la falta de ambigüedad es una propiedad esencial. Los métodos formales pueden ayudar a los diseñadores a identificar inconsistencias en las especificaciones de un sistema dado. Este artículo presenta una descripción formal de un subconjunto del conjunto de instrucciones de la arquitectura Triangle Abstract Machine (TAM) en Z. TAM es una máquina abstracta adecuada para la implementación de lenguajes de programación tipo Algol estructurados en bloques, como Pascal, Modula , Oberon y Triangle. La arquitectura de TAM se basa en la pila, lo que simplifica la generación de código. La notación matemática de Z y la estructura de su esquema ayudan a describir instrucciones lógicas y aritméticas y también proporcionan mecanismos adecuados para modelar instrucciones complejas que acceden a registros, memorias y la pila. Esta investigación propone un enfoque preciso, aunque abstracto, que evita la especificación de conceptos de bajo nivel, como los bits. El trabajo presentado aquí es un estudio de caso en especificación formal aplicado a una asignatura de Ciencias de la Computación.

**Términos del índice:** especificación formal, arquitectura de conjunto de instrucciones, arquitectura de microprocesador, arquitectura de computadora de pila, notación Z, TAM (Triangle Abstract Machine).

## I. INTRODUCTION

The precise and unambiguous modeling of system properties and behavior is one of the benefits of using formal specification languages [19]. For a microprocessor architecture, unambiguity is an essential property. Formal methods can help designers to identify inconsistencies in a given system's specification, and when used in early development stages, they can help to avoid costly design flaws likely to appear later in the testing stages [30]. Finding bugs the late stages of

hardware design development is very expensive; for instance, the FDIV bug in the Intel Pentium processor had a quantified cost of over $400 million [13].

On the other hand, documentation for microprocessor instruction sets is usually distributed in tables, semi-formal formulae, and informal text [2], whereas a formal language specification of the microprocessor architecture ensures unambiguity of the documentation and enables verification.

The Z-formal specification language is based on set theory and mathematical logic [26], [31]. This paper uses Z to specify a subset of the Triangle Abstract Machine (TAM) instruction set as described in [28]. Arithmetic and Boolean instructions are modeled using common mathematical logic concepts while more complex instructions—that involve memory accesses and stack manipulation—require the definition of a model to access the memory, the stack, and the registers.

Section II provides an overview of part of the required background. The memory and registers of TAM are specified in Section III, while most of the instruction set for TAM are specified in Section IV. Section V deals with the loading of programs and the initial state of the TAM machine. Finally, Section VII presents the conclusions and the sketches' future work.

## II. BACKGROUND

The stack has a long and multi-faceted tradition in Computing: as a mechanism for carrying procedure or call and return function [1], as a natural way for describing syntax analysis methods and program translation [24], [16], and as the basis of most techniques for the implementation of recursion [5], among others.

E.W. Dijkstra and J.A. Zonneveld solved the challenges of implementing recursive procedures and functions—with their corresponding parameter-passing mechanisms—in a block-structured language setting in the first working compiler for Algol 60 [9]. That

work inspired the design of several real computer architectures that would use stacks in support of high-level programming languages [6], more prominently, those by Burroughs [29]. Over the years, register computer architectures with complex instruction sets (CISC) or reduced instruction sets (RISC) have tended to dominate the market, yet stack computer architectures have survived and thrived as abstract machines hat simplify compiler code generation algorithms and ease a programming language's portability [17], [18].

Both in academia and industry, "abstract" and "virtual" machines have been variously proposed. Of special attention are the Pascal ETH P-System and the UCSD Pascal which use variants of Wirth's P-machine suitable for efficient compilation and interpretation of Pascal- like programming languages [22], and the Java Virtual Machine (JVM) [20].

Abstract machines can be implemented in hardware circuits, in software interpreters or translators, or combinations of both. The software implementation of an abstract machine via interpretation can help to provide early feedback on the processor's desired behavior and assist in porting a language's implementation to diverse hardware architectures. The interpreter's code can be instructive to learners of the programming language implementation.

Formal specification languages offer an abstract, unbiased, and precise alternative for modeling and specifying computational systems, such as computer architectures, using discrete mathematical structures. Their logic-mathematical foundation opens the opportunity for proving properties of the models and specification documents, while it also opens the opportunity for correct-by-construction and provably- correct implementation [3].

Z is a formal specification language developed by Oxford University's Programming Research Group in the early 1980s [21]. It is based on Zermelo Fränkel axiomatic set theory and first-order predicate logic [23]. Using Z, mathematical objects and their properties can be collected together in schemas [31]. A

characteristic feature of Z is the use of types. Every object in the mathematical language has a unique type, represented as a maximal set in the current specification [31]. A tutorial introduction to the Z notation can be found within the Reference Manual written by Spivey [26].

Other works have specified microprocessors architectures using formal specification languages. For instance, in [2] the 8-bit Motorola 6800 microprocessor instruction set was specified using Z. This specification defines low-level concepts such as bits and words. The work described in this paper uses a higher level of abstraction; all addresses, instructions, and data are specified as natural numbers instead. In [14], the design and verification of the FM8501 are presented, where several formulas are used to verify the system. The Intel 8085 microprocessor is specified in [11] using algebra. A higher-order logic language (HOL) is used in [15] to describe the formal specification of a micro-coded computer, and in [4], it is used to aid in the design and verification of the VIPER (verifiable integrated processor for enhanced reliability) microprocessor chip.

The Triangle Abstract Machine (TAM) was designed as a vehicle to explain high-level programming language implementation techniques typically used in compilers and interpreters [27]. TAM's instruction set architecture, memory organization, and addressing modes are explained informally and via interpreters written in Pascal [27] or Java [28]. TAM's architecture is simple, yet powerful enough as a natural target of code-generation algorithms for imperative and object-oriented languages. Although TAM's interpreters are readily understandable, they are concrete representations in particular programming languages after all. A programming-language independent description, precise and abstract, opens opportunities for analysis and design. This research uses Z because it makes the specification readable and formally verifiable [12]. To our knowledge, this is the first attempt to specify the TAM abstract machine in this formal notation.

## III. MEMORY AND REGISTERS

TAM separates code memory from data memory. Code memory holds instruction words of 32 bits and the data memory holds words of 16 bits [28]. Let *CodeAddr* be the set for all addresses in the code memory and *DataAddr* the set of all addresses in the data memory, from address 0 to the maximum possible address depending on the storage size:

$$\begin{array}{|l} MaxCodeAddr : \mathbb{N} \\ MaxDataAddr : \mathbb{N} \end{array}$$

$$CodeAddr == 0 \,..\, MaxCodeAddr$$
$$DataAddr == 0 \,..\, MaxDataAddr$$

Data words can store 16-bit signed integers. The values that can be stored in data memory range from -32767 to 32767

$$MinIntValue == -32767$$
$$MaxIntValue == 32767$$
$$Data == MinIntValue \,..\, MaxIntValue$$

The contents of the code store are instructions which may be viewed as 32 bits unsigned integers.

$$Inst == 0 \,..\, 4294964295$$

We specify the memories as follows:

$$CodeStore \mathrel{\widehat=} [CodeMem : CodeAddr \longrightarrow Inst]$$
$$DataStore \mathrel{\widehat=} [DataMem : DataAddr \longrightarrow Data]$$

TAM has 16 registers that hold address values of either CodeAddr or DataAddr. TAM identifies each register with a number. The CP register points to the next instruction to be executed. We identify the registers as follows:

$$\begin{array}{|l} CB, CT, PB, PT, SB, ST, HB, HT, \\ LB, L1, L2, L3, L4, L5, L6, CP : 0 \,..\, 15 \\ \hline CB = 0 \land CT = 1 \land PB = 2 \land PT = 3 \land \\ SB = 4 \land ST = 5 \land HB = 6 \land HT = 7 \land \\ LB = 8 \land L1 = 9 \land L2 = 10 \land L3 = 11 \land \\ L4 = 12 \land L5 = 13 \land L6 = 14 \land CP = 15 \end{array}$$

Code storage and data storage are separate and can differ in size. The registers associated with each are declared independently; the set of all registers is the union of both sets:

$$CodeRegs == \{CB, CP, CT, PB, PT\}$$
$$DataRegs == \{SB, LB, ST, HT, HB, L1, L2, L3, L4, L5, L6\}$$
$$Regs == CodeRegs \cup DataRegs$$
$$Addr == CodeAddr \cup DataAddr$$

Register contents are modeled via a mapping from a register's name to a valid address:

$$
\begin{array}{|l}
\textit{Registers} \\
\hline
Reg : Regs \longrightarrow Addr \\
\hline
Reg(\!|Regs|\!) \subseteq Addr
\end{array}
$$

Status is then defined as one of the possible execution states of the machine:

$$
\begin{aligned}
Status ::= {} & running \mid halted \mid failedInvalidDataAddr \mid \\
& failedInvalidCodeAddr \mid failedInvalidInstruction \mid \\
& failedOverflow \mid failedUnderflow \mid \\
& failedDataStoreFull \mid \\
& failedArithmeticOverflow \mid failedDivZero
\end{aligned}
$$

$$RunStatus \mathrel{\widehat=} [status : Status]$$

| OpCode | Reg | Size | AddrDisp |
|---|---|---|---|

*Figure 1. TAM instructions format*

TAM's state comprises its registers, code store, data store, and program execution status.

$$State \mathrel{\widehat=} Registers \land CodeStore \land DataStore \land RunStatus$$

## IV. INSTRUCTIONS

According to [28], TAM instructions follow a common format as shown in Figure 1.

Each TAM instruction has four fields:

$$
\begin{aligned}
opCode &== (0 \,..\, 15) \setminus \{9\} \\
reg &== 0 \,..\, 15 \\
len &== 0 \,..\, 255 \\
operand &== MinIntValue \,..\, MaxIntValue
\end{aligned}
$$

Instruction models the structure of the instruction register (IR), which holds the current instruction to be fetched from code memory, decoded, and executed.

$$
\begin{array}{|l}
\textit{Instruction} \\
\hline
op : opCode \\
r : reg \\
n : len \\
d : operand
\end{array}
$$

Instructions can be encoded as 32-bit integers (Inst).

$$
\begin{array}{|l}
toInst : Instruction \rightarrowtail\!\!\!\!\!\twoheadrightarrow Inst \\
toInstruction : Inst \rightarrowtail\!\!\!\!\!\twoheadrightarrow Instruction \\
\hline
toInst = toInstruction^{\sim}
\end{array}
$$

In addition to proper instructions, TAM has 28 primitive operations that comprise arithmetic, logical, comparison, and I/O actions. They are activated using addresses outside the normal code space.

$$
\begin{aligned}
PrimitiveBase &== 1024 \\
PrimitiveTop &== 1024 + 28
\end{aligned}
$$

1) Fetch stage: When an instruction is fetched from the code store it is first decoded and then executed. The

following schema describes the fetch stage's decode step:

*DecodeStep*

$State$
$Instruction$

$status = running$
$\theta Instruction = toInstruction(CodeMem(Reg(CP)))$

When fetch is successful, the machine should remain running until changed by the execution of the fetched instruction.

*FetchOk*

$DecodeStep$

$op \in opCode$

However, if there is an instruction code not in the TAM's instruction set, the fetch stage will change the status to reflect that.

*FetchCodeError*

$\Delta State$
$\Xi Registers$
$\Xi CodeStore$
$\Xi DataStore$
$DecodeStep$

$op \notin opCode$
$status' = failedInvalidInstruction$

When TAM's status is not running, no attempt should be made to fetch an instruction for subsequent execution.

*NotRunning*

$\Xi State$

$status \neq running$

The Fetch schema specifies the fetch stage.

$$Fetch \mathrel{\widehat=} FetchOk \lor FetchCodeError \lor NotRunning$$

In order to simplify instruction descriptions, the sets of registers appearing in specification schemas are named accordingly. *RegsFixed* are those registers that remain unaltered upon program loading and throughout execution. *DisplayRegs* deal with block structure at run time. *RegsNoCP* are the registers except for *CP*. *RegsNoCPST* are the registers except for *CP* and *ST*. Primitives can only change *CP* and *ST*. *RegsNoCPSTLB* are involved in routine call and return, where adjustments are required for display registers involved in accessing non-local data.

$$RegsFixed == \{CB, CT, PB, PT, SB, HB\}$$
$$DisplayRegs == \{LB, L1, L2, L3, L4, L5, L6\}$$
$$RegsNoCP == \{ST, HT\} \cup DisplayRegs$$
$$RegsNoCPST == \{HT\} \cup DisplayRegs$$
$$RegsUnchangedPrimitive == RegsNoCPST$$
$$RegsNoCPSTLB == RegsNoCPST \setminus DisplayRegs$$

Since neither the instructions nor some of the registers change the code store, the common traits are factored into the Operation schema:

*Operation*

$\Delta State$
$FetchOk$

---

$RegsFixed \lhd Reg' = RegsFixed \lhd Reg$
$CodeMem' = CodeMem$

Additionally, most operations update the CP, so the following schema is defined to avoid repeating that information:

*OperationUpCP*

$Operation$

---

$Reg'(CP) = Reg(CP) + 1$

### A. *Instruction Descriptions*

Not counting primitives, TAM has 15 instructions available. Instruction opcode 9 is not defined in TAM.

$$Instructions == 0 \,..\, 15 \setminus \{9\}$$

These are the opcodes:

$LOAD == 0$
$LOADA == 1$
$LOADI == 2$
$LOADL == 3$
$STORE == 4$
$STOREI == 5$
$CALL == 6$
$CALLI == 7$
$RETURN == 8$
$PUSH == 10$
$POP == 11$
$JUMP == 12$
$JUMPI == 13$
$JUMPIF == 14$
$HALT == 15$

In what follows, instructions are presented in the order of their opcodes:

**LOAD** $(n)$ $d[r]$

Fetch n-words from data address (*d* + register *r*), and push them on top of the stack. Do not affect other data memory words.

*LoadOk*

$OperationUpCP$
$source, dest : DataAddr$
$range : \mathbb{P}\, DataAddr$

---

$op = 0$
$Reg(ST) + n \leq Reg(HT)$
$source = d + Reg(r)$
$dest = Reg(ST)$
$Reg(SB) \leq source$
$source + n \leq dest$
$Reg'(ST) = dest + n$
$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$
$\forall x : \mathbb{N} \mid x < n \bullet$
$\quad DataMem'(dest + x)$
$\quad\quad = DataMem(source + x)$
$range = Reg(ST) \,..\, Reg(ST) + n - 1$
$range \lhd DataMem = range \lhd DataMem'$
$status' = running$

---

**LOADA** $d[r]$

Push the data address (*d* + register *r*) on top of the stack. Do not affect other data memory words.

*LoadAOk*

$OperationUpCP$

---

$op = 1$
$Reg(ST) + 1 \leq Reg(HT)$
$Reg(SB) \leq d + Reg(r) < Reg(ST)$
$Reg'(ST) = Reg(ST) + 1$
$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$
$DataMem'(Reg(ST)) = d + Reg(r)$
$\{Reg(ST)\} \lhd DataMem = \{Reg(ST)\} \lhd DataMem'$
$status' = running$

**LOADI** $(n)$

Pop a data address from the top of the stack, fetch n words starting at that address, and push them on top of the stack. Do not affect other data memory words.

*LoadIOk*

$OperationUpCP$
$source, dest : DataAddr$
$range : \mathbb{P}\ DataAddr$

$op = 2$
$Reg(SB) < Reg(ST)$
$Reg(ST) + n \leq Reg(HT)$
$source = DataMem(Reg(ST) - 1)$
$dest = Reg(ST) - 1$
$Reg(SB) \leq source$
$Reg'(ST) = Reg(ST) + n$
$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$
$\forall x : \mathbb{N} \mid x < n \bullet$
$\quad DataMem'(dest + x) = DataMem(source + x)$
$range = dest\ ..\ dest + n - 1$
$range \lhd DataMem = range \lhd DataMem'$
$status' = running$

**LOADL** $d$

Push the 1-word literal d on top of the stack. Do not affect other data memory words.

*LoadLOk*

$OperationUpCP$

$op = 3$
$Reg(ST) \leq Reg(HT)$
$Reg'(ST) = Reg(ST) + 1$
$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$
$DataMem'(Reg(ST)) = d$
$\{Reg(ST)\} \lhd DataMem = \{Reg(ST)\} \lhd DataMem'$
$status' = running$

**STORE** $(n)$ $d[r]$

Pop *n*-words from the stack's top, and store them starting at data address (*d* + register *r*). Do not affect other data memory words.

*StoreOk*

$OperationUpCP$
$source, dest : DataAddr$
$range : \mathbb{P}\ DataAddr$

$op = 4$
$source = Reg(ST) - n$
$dest = d + Reg(r)$
$Reg(SB) \leq source$
$Reg(SB) \leq dest$
$dest + n \leq Reg(HT)$
$Reg'(ST) = source$
$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$
$\forall x : \mathbb{N} \mid x < n \bullet$
$\quad DataMem'(dest + x) = DataMem(source + x)$
$range = dest\ ..\ dest + n - 1$
$range \lhd DataMem = range \lhd DataMem'$
$status' = running$

**STOREI** $(n)$

Pop an address dest from the top of the stack, then pop n-words, and store them from data address dest. Do not affect other data memory words.

*StoreIOk*

$OperationUpCP$
$source, dest : DataAddr$
$range : \mathbb{P}\ DataAddr$

$op = 5$
$source = Reg(ST) - 1 - n$
$dest = DataMem(Reg(ST) - 1)$
$Reg(SB) \leq dest$
$dest + n \leq Reg(HT)$
$Reg'(ST) = source$
$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$
$\forall x : \mathbb{N} \mid x < n \bullet$
$\quad DataMem'(dest + x) = DataMem(source + x)$
$range = dest\ ..\ dest + n - 1$
$range \lhd DataMem = range \lhd DataMem'$
$status' = running$

**CALL** $(n)$ $d[r]$

Call the routine at code address (*d* + register *r*). Use the address in register *n* as the static link. This instruction must create a stack frame (composed of static link, dynamic link, and return address) on top of the stack. The static link will be addressed by the content of register *n*. The dynamic link is pointed by *LB* register's content. The return address is that of the instruction after the **CALL**, this is, *CP* + *1*. The *LB* register will become the base of the new topmost frame, which was pointed by the *ST* register before the call operation. Only 3 data words are affected by this instruction.

In TAM, the contents of the registers *L1, L2, L3, L4, L5, L6* are updated relative to *LB*'s value as shown in the *DisplayRegisters* schema. The register to be used depends on the difference of nesting between the caller and the callee. Display registers allow for efficient access to non-local data from nested blocks.

*DisplayRegisters*

$$
\begin{aligned}
&Registers \\
&DataStore \\
&\hline
&Reg(L1) = DataMem(Reg(LB)) \\
&Reg(L2) = DataMem(Reg(L1)) \\
&Reg(L3) = DataMem(Reg(L2)) \\
&Reg(L4) = DataMem(Reg(L3)) \\
&Reg(L5) = DataMem(Reg(L4)) \\
&Reg(L6) = DataMem(Reg(L5))
\end{aligned}
$$

When **CALL** and **CALLI** instructions are executed, code addresses may correspond either to a proper programmer-written routine (procedure or function) or to a primitive operation. Primitives are trapped and are processed on top of the stack without creating a stack frame.

Schema CallRoutineOk describes the call to a programmer-written routine.

*CallRoutineOk*

$$
\begin{aligned}
&Operation \\
&range : \mathbb{P}\, DataAddr \\
&\hline
&op = 6 \\
&Reg(ST) + 3 \leq Reg(HT) \\
&Reg(CB) \leq d + Reg(r) < Reg(PB) \\
&DataMem'(Reg(ST)) = Reg(n) \\
&DataMem'(Reg(ST) + 1) = Reg(LB) \\
&DataMem'(Reg(ST) + 2) = Reg(CP) + 1 \\
&Reg'(LB) = Reg(ST) \\
&Reg'(ST) = Reg(ST) + 3 \\
&Reg'(CP) = d + Reg(r) \\
&RegsNoCPSTLB \lhd Reg' = RegsNoCPSTLB \lhd Reg \\
&range = Reg(ST) \,..\, Reg(ST) + 2 \\
&range \lhd DataMem = range \lhd DataMem' \\
&status' = running
\end{aligned}
$$

Data memory words with addresses comprised between *ST* and *ST + 2* will contain the stack frame. **CALL** updates all display registers, *ST*, and *CP*, but not the *HT* register. No other data memory words will be affected when a routine is called.

Schema *CallPrimitiveOk* partially describes what is involved in calling primitive operations. Calls to primitives do not create stack frames and, thus, do not update display registers. To save space, primitives are not presented in this paper.

*CallPrimitiveOk*

*Operation*

$$op = 6$$
$$Reg(PB) \leq d + Reg(r) \leq Reg(PT)$$
$$Reg'(CP) = Reg(CP) + 1$$
$$RegsUnchangedPrimitive \lhd Reg' = RegsUnchangedPrimitive \lhd Reg$$

**CALLI**

This instruction supports calling functions or procedures passed as actual parameters. A closure representing the routine is already in the stack, occupying 2 words below the stack top (*Reg(ST)).* The jump address will be in *Reg(*ST) - 1. The static link remains in the stack (where it is, at *Reg(ST)* - 2), and the stack frame will be completed by pushing the dynamic link and the return address onto the stack. When calling programmer- written routines, **CALLI** updates all display registers, ST, and *CP*, but not the *HT* register.

*CallIRoutineOk*

*Operation*

$$range : \mathbb{P}\, DataAddr$$

$$op = 7$$
$$Reg(ST) + 1 \leq Reg(HT)$$
$$Reg(CB) \leq DataMem(Reg(ST) - 1) < Reg(PB)$$
$$DataMem'(Reg(ST) - 2) = DataMem(Reg(ST) - 2)$$
$$DataMem'(Reg(ST) - 1) = Reg(LB)$$
$$DataMem'(Reg(ST)) = Reg(CP) + 1$$
$$Reg'(LB) = Reg(ST) - 2$$
$$Reg'(ST) = Reg(ST) + 1$$
$$Reg'(CP) = DataMem(Reg(ST) - 1)$$
$$RegsNoCPSTLB \lhd Reg' = RegsNoCPSTLB \lhd Reg$$
$$range = Reg(ST) - 1\,..\,Reg(ST)$$
$$range \lhd DataMem = range \lhd DataMem'$$
$$status' = running$$

Schema *CallIPrimitiveOk* partially describes what is involved in indirectly calling primitive operations. In this case, **CALLI** does not create a stack frame. Primitives are not presented in this paper.

*CallIPrimitiveOk*

*Operation*

$$op = 7$$
$$Reg(PB) \leq DataMem(Reg(ST) - 1) \leq Reg(PT)$$
$$Reg'(CP) = Reg(CP) + 1$$
$$RegsUnchangedPrimitive \lhd Reg' = RegsUnchangedPrimitive \lhd Reg$$

**RETURN** $(n)$ $d$

Data memory words between those pointed by *LB* and *ST* accommodate the stack frame, local data, and n words for function results (*n* = 0 for procedures). The *n*-word results should be moved to the data words whose addresses start at *Reg(LB)-d.*

*ReturnOk*

*Operation*

$source, dest : DataAddr$

$range : \mathbb{P}\ DataAddr$

---

$op = 8$

$Reg(LB) - d + n \leq Reg(HT)$

$source = Reg(ST) - n$

$dest = Reg(LB) - d$

$\forall x : \mathbb{N} \mid x < n \bullet$

$DataMem'(dest + x) = DataMem(source + x)$

$Reg'(ST) = Reg(LB) - d + n$

$Reg'(LB) = DataMem(Reg(LB) + 1)$

$Reg'(CP) = DataMem(Reg(LB) + 2)$

$RegsNoCPSTLB \lhd Reg' = RegsNoCPSTLB \lhd Reg$

$range = Reg(LB) - d \,..\, Reg(LB) - 1$

$range \lhd DataMem = range \lhd DataMem'$

$status' = running$

**PUSH** *d*

**PUSH** moves up the *ST* register, opening space for d data memory words. Only the *ST* and *CP* registers change. The contents of these data memory words are unaffected.

*PushOk*

*OperationUpCP*

---

$op = 10$

$d + Reg(ST) \leq Reg(HT)$

$Reg'(ST) = d + Reg(ST)$

$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$

$DataMem' = DataMem$

$status' = running$

**POP** (*n*) *d*

The **POP** operation moves the n words below *ST* to occupy data memory words starting at memory address *ST - n - d.*

Only the *CP* and *ST* change.

*PopOk*

*OperationUpCP*

$source, dest : DataAddr$

$range : \mathbb{P}\ DataAddr$

---

$op = 11$

$Reg(SB) \leq Reg(ST) - n - d$

$source = Reg(ST) - n$

$dest = Reg(ST) - n - d$

$\forall x : \mathbb{N} \mid x < n \bullet$

$DataMem'(dest + x) = DataMem(source + x)$

$Reg'(ST) = Reg(ST) - d$

$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$

$range = Reg(ST) - n - d \,..\, Reg(ST) - n - 1$

$range \lhd DataMem = range \lhd DataMem'$

$status' = running$

**JUMP** *d*[*r*]

Unconditionally, jump to the address *d* + *Reg(r).* Data memory is not changed and only the *CP* will change. *r* can be any register, though it is normally *CB*.

*JumpOk*

*Operation*

---

$op = 12$

$Reg(CB) \leq d + Reg(r) < Reg(PB)$

$Reg'(CP) = d + Reg(r)$

$RegsNoCP \lhd Reg' = RegsNoCP \lhd Reg$

$DataMem' = DataMem$

$status' = running$

**JUMPI**

Pop the address stored on the top of the stack and then jump to it. Only *CP* and *ST* change. **JUMPI** is an indirect jump.

*JumpIOk*

*Operation*

$op = 13$
$Reg(CB) \leq DataMem(Reg(ST) - 1) < Reg(PB)$
$Reg(SB) < Reg(ST)$
$Reg'(CP) = DataMem(Reg(ST) - 1)$
$Reg'(ST) = Reg(ST) - 1$
$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$
$DataMem' = DataMem$
$status' = running$

**JUMPIF** $(n)$ $d[r]$

It pops 1 word from the stack and compares it to n. If they are equal, control is transferred to code address *(d + Reg(r)).* Otherwise, the execution continues with the next instruction. Only *CP* and *ST* change. **JUMPIF** is a conditional jump.

*JumpIfOk*

*Operation*

$op = 14$
$Reg(CB) \leq d + Reg(r) < Reg(PB)$
$Reg(SB) < Reg(ST)$
$Reg'(ST) = Reg(ST) - 1$
$RegsNoCPST \lhd Reg' = RegsNoCPST \lhd Reg$
$DataMem' = DataMem$
$status' = running$

If the value on top of the stack equals *n*, the jump is made to code address *(d + Reg(r)).*

*JumpIfTrueOk*

*JumpIfOk*

$DataMem(Reg(ST) - 1) = n$
$Reg'(CP) = d + Reg(r)$

If *n* differs from the value on top of the stack, execution continues with the next instruction.

*JumpIfTrueOk*

*JumpIfOk*

$DataMem(Reg(ST) - 1) = n$
$Reg'(CP) = d + Reg(r)$

**HALT**

Stop program execution, changing the status to halted. Contents in data memory and registers are preserved.

*HaltOk*

*Operation*

$op = 15$
$Reg' = Reg$
$DataMem' = DataMem$
$status' = halted$

B. Error situations

Several schemas are defined to check anomalous situations. First, an error base schema is:

*ErrorBase*

*OperationUpCP*

$num : \mathbb{N}$

$DataMem' = DataMem$
$RegsNoCP \lhd Reg' = RegsNoCP \lhd Reg$

Stack overflows should be prevented:

*StackOverflow*
*ErrorBase*

$$Reg(ST) + num > Reg(HT)$$
$$status' = failedOverflow$$

The following schemas check for possible stack overflows when *n, 1, 3* or *d* words are to be pushed onto the stack:

$$StackOverflowN \mathrel{\widehat=} [StackOverflow \mid num = n]$$
$$StackOverflow1 \mathrel{\widehat=} [StackOverflow \mid num = 1]$$
$$StackOverflow3 \mathrel{\widehat=} [StackOverflow \mid num = 3]$$
$$StackOverflowD \mathrel{\widehat=} [StackOverflow \mid num = d]$$

There might be underflows when an operation tries to pop below the stack base. The schemas to check those situations are defined below:

*StackUnderflow*
*ErrorBase*

$$Reg(ST) - num < Reg(SB)$$
$$status' = failedUnderflow$$

The following schemas check for possible stack underflows when *n, 1, n+1, n+d* or *2* words are to be popped from the stack:

$$StackUnderflowN \mathrel{\widehat=} [StackUnderflow \mid num = n]$$
$$StackUnderflow1 \mathrel{\widehat=} [StackUnderflow \mid num = 1]$$
$$StackUnderflowN1 \mathrel{\widehat=} [StackUnderflow \mid num = n + 1]$$
$$StackUnderflowND \mathrel{\widehat=} [StackUnderflow \mid num = n + d]$$
$$StackUnderflow2 \mathrel{\widehat=} [StackUnderflow \mid num = 2]$$

Incorrectly generated routine returns may cause stack underflows when attempting to pop d words from the stack, plus the topmost stack frame:

*StackUnderflowR*
*ErrorBase*

$$Reg(LB) - d < Reg(SB)$$
$$status' = failedUnderflow$$

When memory addresses are to be accessed, each operation must ensure that those addresses actually exist.

*InvalidAddrBase*
*ErrorBase*
$base : \mathbb{N}$

$$\exists x : \mathbb{N}_1 \mid x < num \bullet$$
$$base + x < Reg(SB) \lor base + x > Reg(HB)$$
$$status' = failedInvalidDataAddr$$

Addressing in TAM can be direct or indirect. Direct addressing is relative to a register's content *(r)*, plus a displacement *(d)*.

$$InvalidAddrD \mathrel{\widehat=} [InvalidAddrBase \mid base = d + Reg(r)]$$

*InvalidAddrND* and *InvalidAddr1D* correspond to the cases when *n* or *1* words are to be read using direct addressing.

$$InvalidAddrND \mathrel{\widehat=} [InvalidAddrD \mid num = n]$$
$$InvalidAddr1D \mathrel{\widehat=} [InvalidAddrD \mid num = 1]$$

Invalid data addressing might happen when reading n words indirectly, using an address stored on the stack's top:

*InvalidAddrNI*

*InvalidAddrBase*

$$base = DataMem(Reg(ST) - 1)$$
$$num = n$$

Code memory may also be addressed incorrectly.

*InvalidCodeAddr*

*ErrorBase*

$$num < Reg(CB) \lor num > Reg(PT)$$
$$status' = failedInvalidCodeAddr$$

Incorrect code memory accesses may occur using direct *(InvalidCodeAddrD)* or indirect *(InvalidCodeAddrI)* addressing modes.

C. Total Descriptions

Given the partial specification of each TAM instruction and possible error situations that they may undergo, we can specify instructions totally - that is, in all circumstances.

Load instructions move data to the stack's top. They may overflow but do not underflow. The total operations for **LOAD**, **LOADA**, **LOADI**, and **LOADL** are defined as follows:

$$LoadOp \mathrel{\widehat=} LoadOk \setminus (source, dest, range) \lor StackOverflowN \lor InvalidAddrND$$
$$LoadAOp \mathrel{\widehat=} LoadAOk \lor StackOverflow1 \lor InvalidAddr1D$$
$$LoadIOp \mathrel{\widehat=} LoadIOk \setminus (source, dest, range) \lor StackOverflowN \lor StackUnderflow1 \lor InvalidAddrNI$$
$$LoadLOp \mathrel{\widehat=} LoadLOk \lor StackOverflow1$$

Store instructions move data from the top of the stack into (lower parts of the) data memory. The total operations for **STORE** and **STOREI** are defined as:

$$StoreOp \mathrel{\widehat=} StoreOk \setminus (source, dest, range) \lor StackUnderflowN \lor InvalidAddrND$$
$$StoreIOp \mathrel{\widehat=} StoreIOk \setminus (source, dest, range) \lor StackUnderflowN1 \lor InvalidAddrNI$$

Call instructions may call either programmer-defined or primitive routines. Programmer-defined routines create a stack frame at the stack's top. **CALL** and **CALLI** total operations are defined as follows:

$$CallOp \mathrel{\widehat=} CallRoutineOk \setminus (range) \lor CallPrimitiveOk \lor StackOverflow3 \lor InvalidCodeAddrD$$
$$CallIOp \mathrel{\widehat=} CallIRoutineOk \setminus (range) \lor CallIPrimitiveOk \lor StackOverflow1 \lor InvalidCodeAddrI$$

**RETURN** may cause underflow if code generation was incorrect, but it cannot overflow the stack.

$$ReturnOp \mathrel{\widehat{=}} ReturnOk \setminus (source, dest, range) \\ \lor StackUnderflowR$$

**PUSH** and **POP** deal with the storage needs of local blocks. **PUSH** expands the size of the stack and may cause it to overflow.

$$PushOp \mathrel{\widehat{=}} PushOk \lor StackOverflowD$$

Stack underflow should be prevented when attempting to **POP** *n* + *d* words from the stack.

$$PopOp \mathrel{\widehat{=}} PopOk \setminus (source, dest, range) \lor \\ StackUnderflowND$$

Direct (**JUMP**), indirect (**JUMPI**) and conditional jumps (**JUMPIF**) can only be made to valid code addresses.

$$JumpOp \mathrel{\widehat{=}} JumpOk \lor InvalidCodeAddrD$$
$$JumpIOp \mathrel{\widehat{=}} JumpIOk \lor InvalidCodeAddrI \\ \lor StackUnderflow1$$
$$JumpIfOp \mathrel{\widehat{=}} JumpIfTrueOk \lor JumpIfFalseOk \\ \lor InvalidCodeAddrD \lor StackUnderflow1$$

$$HaltOp \mathrel{\widehat{=}} HaltOk$$

**HALT** always succeeds in stopping execution.

D. Instruction execution

Executing instructions amounts to select them after the fetch and decode stages.

$$Execute \mathrel{\widehat{=}} LoadOp \lor LoadAOp \lor LoadIOp \lor LoadLOp \\ \lor StoreOp \lor StoreIOp \\ \lor CallOp \lor CallIOp \lor ReturnOp \\ \lor PushOp \lor PopOp \\ \lor JumpOp \lor JumpIOp \lor JumpIfOp \\ \lor HaltOp$$

Fetching and execution is the composition of successful fetch & decode, followed by execution proper - modulo fetch errors.

$$FetchExecute \mathrel{\widehat{=}} (FetchOk \land Execute) \\ \lor FetchCodeError \\ \lor NotRunning$$

## V. PROGRAM LOAD AND INITIAL STATE

After loading the program into code memory, and prior to program execution, TAM enters its initial state. The first state of the machine is specified. After this, TAM will start the fetch-execute cycle at code address 0 (pointed by *CP*).

$$
\begin{array}{|l}
\textit{Init} \\
\hline
State' \\
code? : \operatorname{seq} Instruction \\
Instruction' \\
\hline
DataMem' = (\lambda\ a : DataAddr \bullet 0) \\
\forall a : 0 \mathinner{..} (\#code? - 1) \bullet \\
\quad CodeMem'\ a = toInst(code?(a + 1)) \\
Reg'(CT) = \#code? - 1 \\
Reg'(CB) = Reg'(CP) = 0 \\
Reg'(PB) = PrimitiveBase \\
Reg'(PT) = PrimitiveTop \\
Reg'(SB) = Reg'(ST) = Reg'(LB) = 0 \\
Reg'(HB) = Reg'(HT) = MaxDataAddr \\
DisplayRegisters' \\
\theta Instruction' = toInstruction(CodeMem'(Reg'(CP)))
\end{array}
$$

*CB, CT, PB, PT, SB,* and *HB* will remain fixed during program execution. *LB* points to the global block upon program start, as do all other display registers. *ST* points to the first word in data memory. CP points to the first instruction in code memory and the Instruction register is pre-fetched from that code address.

## VI. DISCUSSION

This article aimed to provide a formal specification of the TAM instruction set, using the Z-notation. In addition to the instructions, TAM's initial state has also been specified.

Most of the instructions specified in this document are very similar to other architectures such as the Freescale HCS12 [10], with its load and store instructions as well as flow control instructions, call and jump. As a result, it can be used as a starting point to create formal specifications for other microcontrollers and microprocessors.

Moreover, direct access to data storage, coupled with a stack discipline provides convenient support to the usual parameter-passing mechanisms required for implementing high-level languages that align to the imperative, functional or object-oriented paradigms. Call- by-value and call-by-reference are the more common parameter-passing mechanisms, but call-by-result and call-by-value-result can be supported with no changes to TAM's instruction set nor to its stack discipline.

The arrangement offers a natural fit with a nested block structure (lexical scoping) and static binding. The compliance of the specification presented in sections III, IV and V with the Z syntax, scope and type rules has been validated using the Fuzz type checker created by Mike Spivey [25].

## VII. CONCLUSIONS AND FUTURE WORK

The Z formal notation was used in this research to specify a subset of TAM's instruction set. The use of Z for describing the instruction set architecture aids hardware designers to catch bugs in early stages and provides an unambiguous source of documentation for the hardware system as well as an interface for a compiler's code generator. TAM has been widely used in teaching how a compiler works in universities [8], [7]. A precise description of the instruction set's semantics helps compiler writers devise appropriate data representations, code generation patterns, and algorithms, as well as protocols for procedure/function/ method call and return, or exception handling.

For economy of space, not all of the TAM architecture's 28 primitives are presented in this paper. Only the logical, arithmetic, and comparison operations are specified herein.

Future work will include sequential I/O and dynamic data memory management. In addition, given that many of the instructions specified for TAM are shared by other architectures such as the HCS12 microcontroller [10], this research can build on TAM's specification to create formal specifications in Z for microprocessors and microcontrollers widely used in industrial and academic applications.

The work reported here is a case study in the specification of an instruction set processor architecture suitable as a target for compiling imperative, functional or object-oriented programming languages. The formal specification can serve as a basis for describing other, more realistic, processor architectures. A formal specification opens avenues for performing-machine- assisted formal reasoning (on the specification itself), verification of compiling algorithms, testing of processor prototypes, validation of processor implementations, synthesis of descriptions into circuits, or systematic formal refinement into programs or

hardware via hierarchically-structured formal design notations.

## REFERENCES

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. “Compilers: Principles, Techniques, and Tools, 2nd. ed.”. Addison- Wesley, 2006.

[2] J. P. Bowen. “Formal Specification and Documentation of Microprocessor Instruction Sets”. Microprocessing and Microprogramming, Volume 21, Issues 1-–5, pp. 223–230, August 1987.

[3] J. P. Bowen. “Provably Correct Systems: Community, Connections, and Citations”. Provably Correct Systems 2017, NASA Monographs in Systems and Software Engineering. Springer-Verlag, pp. 313-328, March 2017.

[4] B. Brock and W. A. Hunt. “Report on the Formal Specification and Partial Verification of the VIPER Microprocessor”. Proceedings of the Sixth Annual Conference on Computer Assurance, June 1991.

[5] W. H. Burge. “Recursive programming techniques”. Addison-Wesley, 1975.

[6] Y. Chu (ed.). “High-Level Language Computer Architecture”. Academic Press, 1975.

[7] “COMP3012/G53CMP Compilers 2018/19” http:// www.cs.nott.ac.uk/_psznhn/G53CMP/

[8] “CSCE 531 Spring 2008: The Triangle Language Processor” https://cse.sc.edu/_mgv/csce531sp08 /pr/ TriangleReadme.html

[9] E. W. Dijkstra. “Recursive programming”. Numerische Mathematik, 2(1), pp. 312–318, May 1960.

[10] Freescale Semiconductor “CPU12Reference Manual”. https://www.nxp.com/docs/en/r eference- manual/CPU12RM.pdf

[11] A. Geser. “A Specification of the Intel 8085 Microprocessor: A Case Study”. Conference on Algebraic methods: theory, tools and applications. LNCS 394, Springer-Verlag, June 1987.

[12] I. Hayes. “Specification Case Studies, 2nd. ed”. Prentice Hall International, 1992.

[13] D. L. Hill and J. Rushby. “Acceptance of Formal Methods: Lessons from Hardware Design”. IEEE Computer, April 1996.

[14] W. A. Hunt. “FM8501: A Verified Microprocessor”. Lecture Notes in Artificial Intelligence, LNCS 795, Springer-Verlag, 1994.

[15] J. J. Joyce. “Formal Verification and implementation of a Microprocessor”. In: Birtwistle G., Subrahmanyam P.A. (eds) VLSI Specification, Verification and Synthesis. The Kluwer International Series in Engineering and Computer Science (VLSI, Computer Architecture and Digital Signal Processing), vol 35. Springer-Verlag. pp 129–157, 1988.

[16] D. E. Knuth. “On the translation of languages from left to right”. Information and Control, 8 (6), pp. 607–639, December 1965.

[17] P. Koopman. “Stack computers. The new wave”. Ellis Horwood, 1989.

[18] C. E. Laforest. “Second-Generation Stack Computer Architecture”. Thesis, University of Waterloo, 2007.

[19] C. Matthews and P. A. Swatman. “Fuzzy Concepts and Formal Methods: A Fuzzy Logic Toolkit for Z”. Proceedings of the First International Conference of B and Z Users. LNCS 1878, Springer-Verlag, pp. 491– 510, September 2000.

[20] Oracle Corp. “Java Language and Virtual Machine Specifications”. Oracle Corporation, 2019. https://docs. oracle.com/javase/specs/

[21] G. O’Reagan. “Z Formal Specification Language. In: Concise Guide to Formal Methods.”. Springer, 2017.

[22] S. Pemberton and M. Daniels. “Pascal Implementation: The P4 Compiler and Interpreter”. Ellis Horwood, 1986.

[23] V. Ruhela, “Z Formal Specification Language - An Overview”. International Journal of Engineering Research and Technology, Vol. 1 Issue 6, 2012.

[24] K. Samelson and F.L. Bauer. “Sequential formula translation”. Communications of the ACM, 3(2), pp. 76– 83, Feb. 1960.

[25] J. M. Spivey. “Fuzz typechecker for Z.” https://spivey. oriel.ox.ac.uk/corner/Fuzz typechecker for Z

[26] J. M. Spivey. “The Z Notation. A Reference Manual, 2nd ed.”. Prentice-Hall International, 1992.

[27] D. A. Watt. “Programing Language Processors”. Prentice Hall International, 1993.

[28] D. A. Watt and D. F. Brown. “Programing Language Processors in Java”. Pearson Education, 2000.

[29] W. T. Wilner. “Design of the Burroughs B1700”. AFIPS ’72 Proceedings of the 1972 Fall Joint Computer Conference, part I, pp. 489–497, December 5-7, 1972.

[30] J. Wing. “What is a Formal Method?”. Carnegie Mellon University Research Showcase, November 1989.

[31] J. Woodcock and J. Davies. “Using Z: Specification, Refinement, and Proof”. Prentice-Hall International, 1996.