A formalisation of LEGv8 in Agda

Santiago Arranz Olmos
FAMAF - Universidad Nacional de Córdoba
sarranz233@famaf.unc.edu.ar

Martín Fernández
FAMAF - Universidad Nacional de Córdoba
mfernandez698@famaf.unc.edu.ar

Matías Steinberg
FAMAF - Universidad Nacional de Córdoba
msteinberg565@famaf.unc.edu.ar

Alejandro Gadea
FAMAF - Universidad Nacional de Córdoba
gadea@famaf.unc.edu.ar

Emmanuel Gunther
FAMAF - Universidad Nacional de Córdoba
gunther@famaf.unc.edu.ar

Miguel Pagano
FAMAF - Universidad Nacional de Córdoba
pagano@famaf.unc.edu.ar

ABSTRACT

The LEGv8 architecture is a restricted representation of the ARMv8 architecture. In this paper, we present a formalisation of the LEGv8 architecture in Agda. We have modelled machine words, the processor state, and the semantics of the instruction set; we also include an assembler and disassembler with round-trip correctness. We explain how dependent types allow us to abstract away some repetitive definitions and drive the correctness proof of the assembler.

CCS CONCEPTS

• **Hardware** → *Semi-formal verification*; • **Theory of computation** → *Program verification.*

KEYWORDS

Formalization of Computer Architectures, Mechanization of Semantics, Formalization of Programming Languages

ACM Reference Format:
Santiago Arranz Olmos, Martín Fernández, Matías Steinberg, Alejandro Gadea, Emmanuel Gunther, and Miguel Pagano. 2020. A formalisation of LEGv8 in Agda. In *24th Brazilian Symposium on Programming Languages (SBLP '20), October 19–20, 2020, Natal, Brazil.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3427081.3427086

1 INTRODUCTION

In this short long paper, we present our ongoing advances in the formalisation of the LEGv8 architecture in Agda. The main contribution of the paper is to show that using intrinsically typed constructions is an appealing choice in this context and to highlight the benefits of automating as much as possible using reflection. Our project aims to produce a formalised version of a realistic architecture in Agda in order to assess the feasibility of some techniques to develop correct-by-construction compilers.

There have been various approaches to correct compilers in the literature; Hutton and his co-authors [3, 11] proposed to *calculate* correct compilers starting from a specification. On the other hand, Pardo et al. [9] define a correct-by-construction compiler by using intrinsic typing of the ASTs of the source and target languages. The targets of these works are stack-based abstract machines. Recently, Bahr and Hutton [4] adapted their proposal to a register-based machine, but still far away from a realistic architecture.

In contrast, there are certified compilers for large languages targeting commercially available processors: CompCert [8] is a well-known example of a formally verified compiler from C to real architectures. The compiler developed by Tan et al. [15] is another such example from an ML-like language. These works include or are based on a formalised model of the target architectures, even though the model might be quite abstract. For instance, Barany [5] remarks that in CompCert the memory holds high-level values instead of words of bits and the arithmetic operations are implemented by Coq functions on integers.

Our goal is to have a realistic architecture formalised for experimenting the proof of correctness of compilers in Agda. In particular, one would be able to explore both Hutton's and Pardo's proposal with respect to a realistic architecture. As will be discussed, we made several simplifications of the real architecture, and left out non-essential features; we plan to further expand our formalisation with these in the future. Doing so would enable the development of a correct compiler targeting the LEGv8 architecture, and eventually the ARMv8 architecture, which is our ultimate goal.

Our presentation starts in Sec. 2 by a brief description of the LEGv8 architecture; then we move to our design of machine words in Sec. 3 that is used to model the memory and the registers, as shown in Sec. 4. In Sec. 5 we discuss our encoding of mnemonics and an assembler and disassembler. We proceed by describing in Sec. 6 the semantics of instructions and programs (i.e. lists of instructions). Finally, we conclude by discussing some of our design choices.

2 THE LEGV8 ARCHITECTURE

The LEGv8 architecture [10] is a restricted representation of the ARMv8 architecture. ARMv8 [2] specifies a set of reduced instruction set computing (RISC) architectures that share various general design decisions. A LEGv8 processor has thirty-one 64-bit general-purpose registers (named X0 to X30) [2, Chapter B1] and one 64-bit constant zero register (XZR).[1] It also has a dedicated register for the stack pointer (SP), as well as one for the program counter (PC), though it is not directly accessible as a register. In addition, there are other registers inaccessible for applications, and we leave them out for the moment.

[1] These registers can also be used as 32-bit registers.

SBLP '20, October 19–20, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8943-3/20/09…$15.00
https://doi.org/10.1145/3427081.3427086

The main ARMv8 instruction set is called A64, and both this and the LEGv8 instruction set are divided into four functional groups [2, Chapter C1]:

- Branch instructions, exception generating instructions, and system instructions;[2]
- Data-processing instructions;
- Load and store instructions; and
- SIMD and scalar floating-point data-processing instructions.[3]

The LEGv8 instruction set contains a reasonable number of instructions of each group; these are convenient to work with, in the sense that they are sufficient for most purposes, while not including complex instructions with multiple effects, for example. We therefore believe it to be a representative subset of A64.

Other possible "subsets" of A64 are the A32 and T32 instruction sets [2, Chapter A1].[4] They are designed to improve efficiency and to overcome certain hardware constraints, though, so they are not convenient for our purposes in the sense discussed previously.

Additionally, ARMv8 defines a weakly ordered memory architecture [2, Chapter B2], which allows the observation of memory accesses in a different order from the program one. This order must preserve the observations, i.e., if a memory effect M_1 appears in the program before another memory effect M_2, both to the same location, then M_1 must occur before M_2. Some instructions make concurrent programs feasible with the weak order, such as barriers and the pair load-reserve and store-conditional. A formalisation of the memory model is proposed in [1, Section 2]. Since our focus at this stage of development is the processor, we will assume a naive memory model for now, in which the memory effects occur immediately after their appearance in the program, without any reordering or delay.

3 MACHINE WORDS

We model bits as Booleans and n-bit words as vectors of n Bits. We render true and false as I and O, respectively.

```
Word : ℕ → Set
Word = Vec Bit
```

We did consider other alternatives for the machine word implementation, namely Fin n and ℕ mod n, but these were discarded mainly because of the difficulties they present when doing bitwise operations. There are other reasons as well: ℕ mod n requires the computation of extremely large numbers (e.g. 2^{64} in the case of 64-bit words) when normalising words, for example.

Since words are merely vectors of bits some operations on words are simply the pointwise extension of the underlying operation on bits:

```
_&_ : {n : ℕ} → Word n → Word n → Word n
_&_ = zipWith _∧_
```

Other operations such as addition, subtraction and bit shifts can be implemented easily as well. For instance, addition can be defined as the projection of addition with carry.

```
addWithCarry : ∀ {n} → Word n → Word n → Bit × Word n
addWithCarry [] [] = O , []
addWithCarry (x :: xs) (y :: ys) =
  let (c₀ , bs) = addWithCarry xs ys
      (c₁ , b)  = bitAdd c₀ x y
  in (c₁ , b :: bs)

_+_ : {n : ℕ} → Word n → Word n → Word n
v + w = proj₂ (addWithCarry v w)
```

Having defined addition, we can define inc to increment a value of type Word n in one; and then define the additive inverse (in two's complement) -_ and subsequently subtraction:

```
- : {n : ℕ} → Word n → Word n
- = inc ∘ map not

_-_ : {n : ℕ} → Word n → Word n → Word n
v - w = v + (- w)
```

We have projections from ℕ in Word n (and also with ℤ) and embeddings in the other directions:

```
toℕ   : {n : ℕ} → Word n → ℕ
fromℕ : {n : ℕ} → ℕ → Word n
toℤ   : {n : ℕ} → Word n → ℤ
fromℤ : {n : ℕ} → ℤ → Word n
```

as well as signed and unsigned casting between words of different lengths:

```
castUnsigned : {n m : ℕ} → Word n → Word m
castSigned   : {n m : ℕ} → Word n → Word (suc m)
```

Moreover, we prove the correctness of addition _+_ on machine words with respect to addition on natural numbers _+n_ and then of incrementing (inc), notice the use of leading zeros to avoid overflow:

```
add-correctness : ∀ n (w w' : Word n) → (x y : ℕ) →
                  toℕ w ≡ x → toℕ w' ≡ y →
                  toℕ ((O :: w) + (O :: w')) ≡ x +n y

inc-correctness : ∀ n (w : Word n) x → toℕ w ≡ x →
                  toℕ (inc (O :: w)) ≡ suc x
```

These properties will be useful to reason about compilers, but we do not use them in the mechanisation of the architecture itself. On the other hand, to manipulate register names as words of bits we need to prove the following property:

```
toℕ<2^n : ∀ {n} (w : Word n) → toℕ w < 2 ^ n
```

Since there are 32 registers it seems sensible to represent them with Fin 32 (we define patterns for referring to them by their usual names); then 5 bits are enough to encode a register name. To ensure that a word w : Word 5 can be converted to a Fin 32 we need a proof of toℕ w < 32.

[2] We leave out exception generating instructions and system instructions for the moment.
[3] We leave out floating-point and SIMD instructions for the moment.
[4] These are not subsets strictly speaking, but in the sense that for every A32 or T32 instruction there is an equivalent A64 instruction, apart from instructions that only make sense in the context of the A32 or T32 instruction sets.

4 PROCESSOR STATE

The processor state consists of thirty-two 64-bit registers, an address space of 64 bits, and a 64-bit program counter.

Register is a simple enumerated type, and the register field and memory are modelled as functions:

```
RegField : Set
RegField = Register → Word 64

Memory : Set
Memory = Word 64 → Word 64
```

that is, a register field is a function that maps Registers to Word 64s.

ARMv8 has a von Neumann architecture, yet we assume a dedicated memory space for code, that is, a Harvard architecture [6]. The difference between the two is that a Harvard machine has a dedicated instruction memory, separated from data memory, while a von Neumann machine does not. This was significant in the past since CPUs could use data and instruction memory pathways more efficiently, but it ceased to be relevant when cache systems were introduced. Our simplification allows us to disregard instruction fetching, decoding, and similar concerns, as well as to reason more abstractly about the correctness of programs. On the other hand, it excludes the possibility of self-modifying code. The machine state is, therefore, a record indexed by the length of the instruction memory, so as to guarantee that branch instructions (also known as jump instructions) target valid positions in instruction memory.

```
record State (n : ℕ) : Set where
  field
    pc      : Fin n
    regfield : RegField
    memory  : Memory
```

Hence, the program counter acts as an index of the instruction vector, which we discuss in the next section.

5 INSTRUCTIONS

The functional groups of instructions as presented in Sec. 1 are quite heterogeneous, comprising instructions with different number or types of arguments; for example, the R format includes (among others) LSL, AND and ADD, each with the following syntax and semantics:

```
ADD X1, X2, X3        ; X3 := X1 + X2
AND X1, X2, X3        ; X3 := X1 & X2
LSL X5, X4, #3        ; X5 := X4 * 8
```

Notice that the third argument of ADD and AND is a register but is an immediate value for LSL. The semantics of ADD and AND are quite similar: apply some function on the contents of the registers X1 and X2 and put the result in X3. We say that the *argument type* of AND and ADD is "three registers", while that of LSL is "two registers and an immediate value".

In order to give a more consistent and regular structure to the instruction set, we make use of instruction *formats*, inspired by formats in [10]. Instruction *formats* describe the number and type of their parameters; for instance, the format Rarith corresponds to arithmetic instructions, which take three registers as parameters. We explicitly introduce a data type Format where each constructor represents a format; in fact, we have a more fine-grained grouping than that in [10].

We depart slightly from the set of formats introduced by Patterson and Hennessy: the formats we call Rarith, Rshift and Rbranch are only one format, R, even though they have different signatures. Such a design is convenient for the hardware implementation, as it makes the decoding logic simpler, but rather troublesome when formalising the architecture.

Our formalisation includes the six instruction formats of LEGv8. The format R has three distinct uses, which we model separately, as discussed previously: arithmetic, logical, and branching.

```
data Format : Set where
  Rarith Rshift : Format
  I             : Format
  D             : Format
  B Rbranch     : Format
  CB            : Format
  IM            : Format
```

The formats B (branching to an absolute address), CB (conditionally branching to an absolute address) and Rbranch (branching to an address stored in a register) are for branching instructions, while Rarith (arithmetic and logical instructions involving only registers), Rshift (bit shifting instructions involving a register and the number of positions to shift), I (arithmetic and logical instructions involving a register and a 12-bit number) and IM (storing a 16-bit value to a register) are for data-processing instructions, and D for load and store instructions.

Inspired by van Geest and Swierstra [18] we use a *description* of the argument type associated to each format:

```
args : Format → ArgType
args Rarith  = reg ⊗ word 6 ⊗ reg ⊗ reg
args Rshift  = reg ⊗ reg ⊗ word 6 ⊗ word 5
args I       = reg ⊗ reg ⊗ word 12
args D       = reg ⊗ reg ⊗ word 9 ⊗ word 2
args B       = word 26
args Rbranch = reg ⊗ word 6 ⊗ word 5 ⊗ word 5
args CB      = reg ⊗ word 19
args IM      = reg ⊗ word 16 ⊗ word 2
```

Factorising the arguments into a description and an interpretation is a flexible and convenient approach since it permits to define function by analysing the description. ArgType has three constructors, for registers, words of a certain length, and the pairing of two ArgTypes.

```
data ArgType : Set where
  reg   : ArgType
  word : ℕ → ArgType
  _⊗_  : ArgType → ArgType → ArgType
```

The intended meaning of each constructor is obvious:

```
⟦_⟧ᵗ : ArgType → Set
⟦ reg     ⟧ᵗ = Register
⟦ word n ⟧ᵗ = Word n
⟦ a ⊗ b   ⟧ᵗ = ⟦ a ⟧ᵗ × ⟦ b ⟧ᵗ
```

We will use ArgType for having a uniform definition of the semantics of each instruction (see Sect. 6) and also for the encoding/decoding of instructions. We also include in args bit-fields that are used only in the encoding of instructions. The format Rarith serves as an illustration of this: the second parameter given by args, word 6 , is disregarded by all mnemonics of that format.

As discussed, our formalisation includes arithmetic and logical operations, memory access, and branching (both absolute and conditional) instructions. To refer to such operations, we use their *mnemonics*. These are modelled as a family indexed by formats:

```
data Mnemonic : Format → Set where
  ADD AND EOR ORR SUB : Mnemonic Rarith
  LSL LSR : Mnemonic Rshift
  ADDI ANDI EORI ORRI SUBI : Mnemonic I
  LDUR LDURB LDURH LDURSW : Mnemonic D
  STUR STURB STURH STURW : Mnemonic D
  B BL : Mnemonic B
  BR : Mnemonic Rbranch
  CBZ CBNZ : Mnemonic CB
  MOVK MOVZ : Mnemonic IM
```

Instructions are modelled as mnemonics together with the arguments described by the semantics of their format.

```
Ins : Format → Set
Ins f = Mnemonic f × ⟦ args f ⟧ᵗ
```

Let us revisit the instructions shown before, now written in Agda; here X*i* and pad have type Register and Word 6, respectively.

```
addEx andEx : Ins Rarith
addEx = ADD , X1 , pad , X2 , X3
andEx = AND , X1 , pad , X2 , X3
lslEx : Ins Rshift
lslEx = LSL , X5 , X4 , pad , # 3
```

Some clarification as to the semantics of instructions: Rarith mnemonics are arithmetic and logical operations on register values, LSL and LSR are left and right bit shifting instructions, and I mnemonics are similar to the Rarith ones but on a register and a 12-bit number. The LDUR and STUR mnemonics involve loading from or storing to memory, and their variants load or store a single byte, a half-word, or a sign-extended word. The B and BL mnemonics are branch instructions that jump to a specified 26-bit address, with the latter storing the value of the PC in X30 before doing so; BR is also a branching instruction but to the address stored in a register. Both CBZ and CBNZ are conditional branch instructions, the first jumping to a specified 19-bit address if the value of the register argument is zero, and the second doing so if it is not. Lastly, the MOVK and MOVZ mnemonics store a 16-bit value in a register, keeping or zeroing the rest of the bits.

Finally, we model programs as sequences of instructions; in order to ensure that jumps are to defined positions, we use vectors instead of lists. As the format of an instruction is part of its type, different instructions may have different types, thus we cannot use Vecs, as they are homogeneous. To overcome this difficulty, we use *Heterogeneous Vectors* instead

```
HVec : {I : Set} → (I → Set) → List I → Set
```

which allow us to construct lists of elements of an indexed family.

```
Code : List Format → Set
Code = HVec Ins
```

Let us recall at this point that we are assuming a Harvard architecture; in order to model a von Neumann architecture, it is necessary to encode instructions as binary words to store them in memory and to decode words as instructions (an evidently partial operation since not every word will encode a valid instruction).

Encoding instructions as words

The encoding/decoding of instructions can be interpreted as an instance of parsing/pretty-printing: to encode an instruction we must produce a word by mapping each mnemonic to its opcode and concatenating the latter to the binary representation of its arguments. Encoding is clearly a total operation, whereas the decoding of a word may fail if no opcode can be recognised; moreover, even if an opcode is recognised, the rest of the word might not contain an appropriate encoding of arguments corresponding to this mnemonic. In any case, these operations should satisfy both of the following:

- Decoding an encoded instruction should be successful, and the result should be the original instruction; and
- Encoding a successfully decoded machine word should yield the same word.

The simplistic encoding of instructions as the concatenation of the encodings of each component (mnemonic and arguments) does not coincide with the actual LEG or ARMv8 one; thus our encoding is not suitable to interact with code generated by other assemblers, but it is still of use, allowing us to explore the von Neumann model discussed in the next section.

The opcode corresponding to each mnemonic is given by the following relation (we only show a few cases)

```
data OpCode : ∀ {f} → Mnemonic f → Word ⟦ f ⟧n → Set where
  OpADD : OpCode ADD (I::O::O::O::I::O::I::I::O::O::O::[])
  OpAND : OpCode AND (I::O::O::O::I::O::I::O::O::O::O::[])
  OpEOR : OpCode EOR (I::I::O::O::I::O::I::O::O::O::O::[])
```

For instance, the first line associates the word 10001011000 (0x458) to the instruction ADD of LEGv8, exactly as in the book [10].[5] The above relation induces a bijection between mnemonics and opcodes:

```
OpCode-functional : ∀ {f} → IsFunctional $ OpCode {f}
OpCode⁻¹-functional : ∀ {f} → IsFunctional $ OpCode⁻¹ {f}
OpCode-functional = auto
```

These lemmas are obvious but their proofs require a tedious analysis with n^2 cases, where n is the number of mnemonics; it is an excellent opportunity to use the reflection capabilities of Agda to deal with them automatically. We refer the interested reader to [16, 17] for more information about reflection in Agda. We also use reflection for the pretty-printer of mnemonics.

```
pp : {f : Format} (i : Mnemonic f) → ∃ (OpCode i)
```

[5] LEGv8 has different mnemonics for variants of ADD (immediate) and ADD (shifted registry) of ARMv8; the ADD instruction of LEGv8 corresponds to the 64-bit and the LSL variant of the ADD (shifted registry) instruction in ARMv8.

The decidability[6] of OpCode follows from the decidability of equality of words; in the following snippet ⟦ f ⟧n corresponds to the length of the opcodes for mnemonics of the format *f*:

```
OpCode-dec : ∀ {f} → (s : Word ⟦ f ⟧n) → (m : Mnemonic f) →
             Dec (OpCode m s)
OpCode-dec s m with pp m
... | (s' , opCodeRule) with s ≟ s'
... | no neq  = no $ neq ∘ flip OpCode-functional opCodeRule
... | yes refl = yes opCodeRule
```

From this we can deduce the decidability of the existence of some mnemonic of some fixed format *f* corresponding to the word *s*:

```
OpCode⁻¹-dec : ∀ f → (s : Word ⟦ f ⟧n) → Dec (∃ (OpCode⁻¹ s))
OpCode⁻¹-dec f s =
  ex-mnem-dec f (OpCode⁻¹ s) (OpCode-dec s)
```

In the above definition ex-mnem-dec is a function that proves the decidability of the existence of some mnemonic for any predicate that is decidable individually:

```
ex-mnem-dec : (f : Format) → (P : Mnemonic f → Set) →
              (∀ m → Dec (P m)) → Dec (∃ P)
ex-mnem-dec Rarith = ex-arith-dec
ex-mnem-dec Rshift  = ex-shift-dec
```

Each of the functions ex-fmt-dec above boils down to a simple application of the fold ins-fmt-fold corresponding to Mnemonic fmt. Both ins-fmt-fold and ex-fmt-dec are generated by reflection; again, this allows us to add new formats and mnemonics with minimal effort.

Given a word *s* of length ⟦ f ⟧n , parsing a mnemonic of format *f* amounts to using the decidability of the converse of OpCode:

```
parse-mnem-strict : ∀ {f} → Word ⟦ f ⟧n → Maybe (Mnemonic f)
parse-mnem-strict {f} s with OpCode⁻¹-dec f s
... | no _                = nothing
... | yes (mnemonic , _) = just mnemonic
```

The correctness of parse-mnem-strict means that the round-trip of parsing after pretty-printing a mnemonic should be successful and return that mnemonic.

```
rt-parse-mnem : ∀ {f} (m : Mnemonic f) →
                parse-mnem-strict (proj₁ $ pp m) ≡ just m
```

With these functions, we can define a parser that consumes only the opcode of a mnemonic and leaves the rest of the word intact. We have a correctness result for this as well:

```
parse-mnem : {f : Format} →
             (s : Word (⟦ f ⟧n + (32 − ⟦ f ⟧n))) →
             Maybe $ Σ[ m ∈ Mnemonic f ] Word (32 − ⟦ f ⟧n)

rt-mnem : ∀ {f} (m : Mnemonic f) → (r : Word (32 − ⟦ f ⟧n)) →
          parse-mnem (proj₁ (pp m) ++ r) ≡ just (m , r)
```

The printing and parsing of arguments is driven by ArgType; notice that the parsing of arguments is a total operation because we take as many bits (⟦ f ⟧ˡ) as needed to ensure that the arguments fit in the corresponding type:

```
pp-args : {f : ArgType} → ⟦ f ⟧ᵗ → Word ⟦ f ⟧ˡ
parse-args : {f : ArgType} → Word ⟦ f ⟧ˡ → ⟦ f ⟧ᵗ
```

By combining the parser and pretty-printer for mnemonics and arguments we can easily define these operations for instructions:

```
pp-ins : {f : Format} → Ins f → Word (⟦ f ⟧n + (32 − ⟦ f ⟧n))
```

It is important to note that since Word is a dependent type, one cannot transparently equate words (w : Word n) and (w' : Word m) unless m≡n definitionally; when this is not the case, one needs to use heterogeneous equalities or explicitly *cast* the type of either *w* or *w'*.

```
parse-ins : ∀ {f} → Word (⟦ f ⟧n + (32 − ⟦ f ⟧n)) → Maybe (Ins f)
parse-ins {f} w = do
  (i , rest) ← parse-mnem {f} w
  let args = parse-args' rest (sym ⟦ f ⟧args)
  just (i , args)
```

We can prove that pretty-printing and parsing an instruction is successful and the result is the appropriate instruction:

```
rt-ins : ∀ {f} → (i : Ins f) → parse-ins {f} (pp-ins i) ≡ just i
```

Finally we can define the top-level parser as attempting to parse under each format:

```
parse-ins' : ∀ f → Word 32 → Maybe (Ins f)
parse-ins' f w rewrite sym (m+[n−m]≡n ⟦ f ⟧n⩽32) =
  parse-ins {f} w

parser : Word 32 → Maybe $ Σ[ f ∈ Format ] (Ins f)
parser w = foldr (λ f → g f <|>_) nothing formats
  where
  g : (f : Format) → Maybe $ Σ[ f ∈ Format ] (Ins f)
  g f = parse-ins' f w >>= λ i → just (f , i)
```

Let us note that our parser is useful to decode instructions from the memory; in order to parse a string with a complete program it would be wise to use parsing combinators [7].

As an example of the encoding of an instruction, we show in detail that of ADD , X1 , pad , X2 , X3 as a binary word of length 32:

$$\underbrace{10001011000}_{\text{ADD}}\ \underbrace{00001}_{\text{X1}}\ \underbrace{000000}_{\text{pad}}\ \underbrace{00010}_{\text{X2}}\ \underbrace{000011}_{\text{X3}}$$

6 SEMANTICS

Since we assume a mono-core setting and a naive memory model, and do not currently model pipelined excecution, the semantics of each instruction is deterministic and can be modelled as a function. Executing an instruction involves some combination of the following:

- updating the register field;
- updating a memory entry; or
- updating the program counter.

For convenience, we introduce a function, setReg, to update the value of a register and increase the program counter.

[6] A relation R : A → C → Set is decidable if there is a decision procedure for R, that is there exists a function decR : (a : A) → (c : C) → (R a c) ⊎ ¬ (R a c) that given any elements *a* and *c* decides if either there is a proof for R a c or a proof of ¬ (R a c).

```
setReg : ∀ {ls} → Register → Word 64 → State ls → State ls
setReg rd val st =
  record st { pc = nextPC (pc st)
            ; regfield = setRegfield (regfield st) rd val
            }
```

The most naive definition of the semantics function consists of an equation per mnemonic; for example, let us focus on two Rarith mnemonics:

```
⟦_⟧i : ∀ {f} → (i : Ins f) → {n : ℕ} → State n → State n
⟦ ADD , rd , _ , rn , rm ⟧i st = setReg rd val st
  where val = regfield st rn + regfield st rm
⟦ AND , rd , _ , rn , rm ⟧i st = setReg rd val st
  where val = regfield st rn & regfield st rm
```

As mentioned previously, the rest of the Rarith mnemonics also entail binary operations on registers. It is clear that the use of formats lends itself to the abstraction of patterns like the one above, so as to avoid repetition. Each of the preceding equations computes a value given by a binary operation on the values of two registers, *rn* and *rm*, and stores said value in the *rd* register. To model this we introduce a new function on formats ⟦_⟧op : Format → Set specifying the type of the operations involved in their semantics; e.g. ⟦ Rarith ⟧op = Word 64 → Word 64 → Word 64.

This leads to a neat definition of the semantics: for each format, we identify the type of the associated operations and assign each mnemonic an operation of that type; then we define a generic function that does the rest of the work. Continuing with Rarith, we have:

```
semRarith : ∀ {n} → ⟦ Rarith ⟧op → ⟦ Rarith ⟧f →
            State n → State n
semRarith _⊙_ (rd , _ , rn , rm) st = setReg rd val st
  where val = regfield st rn ⊙ regfield st rm
```

Consequently, the semantics function can be defined in the following way (we now show one instruction per format):

```
⟦_⟧i : ∀ {f} → (i : Ins f) → {n : ℕ} → State n → State n
⟦ ADD  , args ⟧i          = semRarith _+_ args
⟦ ADDI , args ⟧i          = semI _+_ args
⟦ LDUR , args ⟧i          = semDread id args
⟦ STUR , args ⟧i          = semDwrite const args
⟦ B    , args ⟧i {suc m}  = semB {m} args
⟦ CBZ  , args ⟧i {suc m}  = semCB {m} isZero args
```

The functions semDread and semDwrite take a function as their second parameter because instructions different from LDUR and STUR transform the loaded/stored word when loading/storing.

Note that the type of *args* in the above definition depends on the format of the accompanying mnemonic; it is only because of Agda's dependent type system that we are allowed to have such uniform and abstract definitions.

It is possible to abstract even further, as for each format the same function call is being made per mnemonic. We can do away with this repetition by collecting all the relevant information of a mnemonic in a record and then simplifying the definition to a single line per format. This information is, at the moment, only the opcode and its semantics; yet we can envision other uses, for instance, the number of clock cycles a mnemonic takes to execute.

Having defined the semantics of each instruction, let us turn our attention to the semantics of a program. Given a code memory (recall that Code is a list of instructions), we define a small-step semantics as the reflexive transitive closure of the relation between states induced by the single step transition given by changing the state according to the instruction indicated by the pc register

```
data _,_⇝_ {ls} (code : Code ls) : Rel (State (length ls)) 0ℓ where
  step : ∀ st → code , st ⇝ ⟦ code !!v pc st ⟧i st

_,_⇝⁺_ : ∀ {ls} (code : Code ls) → Rel (State (length ls)) 0ℓ
_,_⇝⁺_ code = Star (code ,_⇝_)
```

Alternatively we define a functional semantics taking an extra-argument indicating how many steps the processor might perform:

```
exec : ∀ {ls} → ℕ → Code ls → State (length ls) →
       State (length ls)
exec 0 code st = st
exec (suc n) code st = exec n code $ ⟦ code !!v pc st ⟧i st
```

The equivalence between both alternatives is straightforward:

```
exec⇒⇝⁺ : ∀ {ls} (code : Code ls) (st : State (length ls)) →
          ∀ n → code , st ⇝⁺ exec n code st
exec⇒⇝⁺ code st 0 = ε
exec⇒⇝⁺ code st (suc n) = let st' = ⟦ code !!v pc st ⟧i st
                          in step st ▹ exec⇒⇝⁺ code st' n

⇝⁺⇒exec : ∀ {ls} (code : Code ls) (st st' : State (length ls)) →
          code , st ⇝⁺ st' → ∃[ n ] (exec n code st ≡ st')
⇝⁺⇒exec code st .st ε = 0 , refl
⇝⁺⇒exec code st st''(step .st ▹ ps) =
  let st' = ⟦ code !!v pc st ⟧i st
      (n , p) = ⇝⁺⇒exec code st' st'' ps
  in (suc n , p)
```

A more realistic memory. Adapting our model to a von Neumann architecture is not particularly difficult. The record State would no longer be indexed by a natural number, and the field pc would have type Word 64 and increase in steps of 4. The main difference resides in that there is the possibility of an error when parsing an instruction. Therefore, the relation ⇝ and the function exec must take this into consideration:

```
fetch : State → Word 32
fetch st = castUnsigned $ memory st $ pc st

data _⇝_ : State → State → Set where
  step : ∀ {f} {ins} st → parser (fetch st) ≡ just (f , ins) →
         st ⇝ ⟦ ins ⟧i st

exec : ℕ → State → Maybe State
exec 0 st = just st
exec (suc n) st = do
  _ , ins ← parser $ fetch st
  exec n $ ⟦ ins ⟧i st
```

7 CONCLUSION

We formalised in Agda the core fragment of the LEGv8 architecture. We plan to cover the full architecture set, by adding floating-point instructions for example, and then to extend our formalisation to the ARMv8 architecture.

We find that using dependent types not only to specify properties but also to refine some types involved (Code and Mnemonic) allows for a better modularisation and abstraction of the mechanisation of a computer architecture; this can be seen for example in the semantics function, being defined per format rather than per mnemonic, and some parts of the assembler/disassembler.

However, dependent types impose seemingly spurious casts (in the guise of subst or by rewriting on the left-hand side) because of some equality failing to hold definitionally; for instance in the type of parse-mnem. Moreover, we needed to prove some equalities between machine word lengths to define the logical shifts operations on Words. Some of these issues may well be attributed to our having somewhat contradicting goals: to have a faithful representation of the hardware, as described in the book, and also to have a relatively abstract model such that some invariances hold by virtue of typing (e.g. branch instructions are restricted to a code memory, rather than the whole memory space).

As a result, it might be convenient to decouple these goals and have separate representations of a processor: the more abstract one would correspond to the Harvard architecture we currently have, while the lower-level one is the von Neumann version already discussed. The former could also feature a more abstract memory and register model holding integers (or even tagged data that fits in 64 bits); then, one might consider that Word operations are given by the native Agda operations. As we already mentioned, in the von Neumann architecture, decoding may fail so execution is no longer total. Additionally, the semantics of branch instructions would be more straightforward, with no guarantee as to whether the next word to be decoded is a valid instruction.

Let us acknowledge that there is an abundance of literature on verification of the ARM architecture [12–14]; furthermore, CakeML [15] includes formalisations of the ARMv7 and ARMv8 architectures in HOL-light. In comparison, our work is rather modest, but still is the first publicly available formalisation of an ARM-like architecture in Agda.

In conclusion, to reason about low-level programs we ought to provide proofs of the correctness of low-level operations; this, in turn, enables us to certify the correctness of a compiler targeting our formalised processor. The work by Pardo et al. [9] testifies to this: it presents a correct-by-construction compiler and proposes to study the possibility of using a more realistic processor.

REFERENCES

[1] Jade Alglave, Anthony C. J. Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The semantics of power and ARM multiprocessor machine code. In *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*, Leaf Petersen and Manuel M. T. Chakravarty (Eds.). ACM, 13–24. https://doi.org/10.1145/1481839.1481842

[2] ARM Ltd. 2017. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile) (DDI0487)*. ARM Ltd. https://developer.arm.com/docs/ddi0487/a/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile

[3] Patrick Bahr and Graham Hutton. 2015. Calculating Correct Compilers. *Journal of Functional Programming* 25 (Sept. 2015).

[4] Patrick Bahr and Graham Hutton. 2020. Calculating Correct Compilers II: Return of the Register Machines. *Journal of Functional Programming* (2020). To appear.

[5] Gergö Barany. 2018. A more precise, more correct stack and register model for CompCert. In *LOLA 2018 - Syntax and Semantics of Low-Level Languages 2018*. Oxford, United Kingdom. https://hal.inria.fr/hal-01799629

[6] H.G. Cragon. 1980. The Elements of Single-Chip Microcomputer Architecture. *Computer* 13 (1980), 27–41. Issue 10. https://doi.org/10.1109/mc.1980.1653373

[7] Nils Anders Danielsson. 2010. Total parser combinators. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 285–296. https://doi.org/10.1145/1863543.1863585

[8] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[9] Alberto Pardo, Emmanuel Gunther, Marcos Viera, and Miguel Pagano. 2018. An Internalist Approach to Correct-by-Construction Compilers. In *PPDP*. ACM.

[10] David A. Patterson and John L. Hennessy. 2016. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann.

[11] Mitchell Pickard and Graham Hutton. 2020. Dependently-typed compilers don't go wrong. (2020). http://www.cs.nott.ac.uk/~pszgmh/well-typed.pdf In preparation.

[12] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[13] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 161–168. https://doi.org/10.1109/FMCAD.2016.7886675

[14] Alastair Reid. 2017. Who guards the guards? formal validation of the Arm v8-m architecture specification. *PACMPL* 1, OOPSLA (2017), 88:1–88:24. https://doi.org/10.1145/3133912

[15] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In ***ICFP** '16: Proceedings of the 21th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 60–73. https://doi.org/10.1145/2951913.2951924

[16] Agda Development Team. 2020. Agda. https://agda.readthedocs.io/en/latest/

[17] Paul van der Walt and Wouter Swierstra. 2012. Engineering Proof by Reflection in Agda. In *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Ralf Hinze (Ed.), Vol. 8241. Springer, 157–173. https://doi.org/10.1007/978-3-642-41582-1_10

[18] Marcell van Geest and Wouter Swierstra. 2017. Generic packet descriptions: verified parsing and pretty printing of low-level data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, Sam Lindley and Brent A. Yorgey (Eds.). ACM, 30–40. https://doi.org/10.1145/3122975.3122979