Microprocessor Verification:
A Tutorial

Michael L Coe
Phillip J. Windley

Research Report LAL-92-10
March 16, 1994

Abstract

Microprocessor verification has become an area of much interest. The generic interpreter theory provides a model and methodology for the specification and verification of microprocessors. Prior to this model, most microprocessor correctness proofs were ad hoc, and the formalism for one proof would be of little use for another. Recently, we have updated the generic interpreter theory. This report is the verification of a simple microprocessor called Tamarack using this new model. Since Tamarack is simple, this should provide an excellent example of how to apply the generic interpreter theory.

This report details the specification and verification of Tamarack. Sections two through six will highlight the specification, and sections seven and eight will show how the verification was completed.

Contents

1 Introduction **5**

2 Preliminaries **6**
2.1 Generic Interpreter Theory 6
2.1.1 Microprocessor Specification. 6
2.1.2 The Abstract Representation 7
2.1.3 Specification and Verification 8
2.2 Tamarack 9
2.3 Defining the Abstract Representation 12
2.4 Theory Obligations 13

3 The Electronic Block Model **15**
3.1 EBM State 15
3.2 The Environment 16
3.3 EBM Specification 17

4 The Phase Level **19**
4.1 Phase State 19
4.2 Phase Environment 19
4.3 The Phase Level Instruction 19
4.4 The `select` Function 21
4.5 Defining the `instructions` function 21
4.6 The `output` Function 21
4.7 `subout`, `substate`, and `subenv` 21
4.8 The Synchronization Predicate 21
4.9 Defining the Phase Level Interpreter 21

5 The Microprogram Level **24**
5.1 The Micro State 24
5.2 Environment 24
5.3 Microinstructions 24
5.4 The `select` Function 26
5.5 The `instructions` Function 26
5.6 Output 26
5.7 `subout`, `substate`, `subenv` 26
5.8 The Synchronization Predicate 27
5.9 Defining the Micro Level Interpreter 27

6 The Macro Level **28**
6.1 Macro State 28
6.2 Environment 28
6.3 Macroinstructions 28
6.4 The `select` Function 30
6.5 The `instructions` Function 30

6.6 The `output` Function . . . 31
6.7 `subout`, `substate`, `subenv` . . . 31
6.8 The Synchronization Predicate . . . 31
6.9 Defining the Macro Level Interpreter . . . 32

7 Verifying an Interpreter 33

8 The Final Correctness Statement 38

9 Conclusion 42

A EBM Definitions 44

B The Microinstructions 49

C The Macroinstructions 65

1 Introduction

The formal verification of microprocessors is a relatively new area of research. In the last few years, formal techniques been applied to verifying microprocessor behavior. A few notable examples are [Coh88, Hun87, Joy89, Win90]. In [Win90], we presented a formal model or methodology for specifying and verifying microprocessors. In [Win90] the model was used to formally verify a microprocessor called AVM-1. Prior to this most of the proofs for microprocessors were conducted in an *ad hoc* fashion, thus all the formalism built for that processor was of little use in other verifications. An updated model is presented in [Win92b]. The changes to the model include:

- Reducing the number of theory obligations that need to be proven to verify an interpreter from three to two.
- The specification of the output of an interpreter was also added to make the model more complete.
- The instructions function, the function which selects the next state transition function, was changed so that it returns a function instead on selecting it from a list.

This paper presents the specification and verification of a simple microprocessor called Tamarack using the updated generic interpreter theory. We do this as a means of assuring the model is useful[1] as well as providing a tutorial on its use. Because of its simplicity, Tamarack makes an excellent example, and thus has been verified many times [Gor83, Joy86, Joy89]. Our work closely follows that of Joyce in [Joy89]. The results of the verification presented here, that Tamarack is correct, are not new; the purpose of the proof and this paper are to serve as a tutorial on using the generic interpreter theory in microprocessor verification and demonstrate state-of-the-art methods in microprocessor specification and verification.

[1]Indeed, when first used, we found that one of the new theorems in the generic interpreter theory did not perform the task we expected of it. A new, more general, theorem resulted from the effort.

2 Preliminaries

This section provides a brief introduction to the generic interpreter theory and the architecture of Tamarack. For more information see [Joy89, Win92b].

2.1 Generic Interpreter Theory

There have been numerous efforts to formally model microprocessors. The best known of these include Jeff Joyce's Tamarack microprocessor [Joy89], Warren Hunt's FM8501 microprocessor [Hun87], Avra Cohn's VIPER microprocessor [Coh88] and our *AVM–1*. Tamarack is a simple microprocessor with only 8 instructions. FM8501 is larger (roughly the size of a PDP-11), but has not been implemented (a 32–bit version is currently being verified and implemented by Hunt *et al.*). VIPER was designed by Britain's Royal Signals and Radar Establishment (RSRE) at Malvern to provide a formally verified microprocessor for use in safety critical applications and is commercially available. *AVM–1* is a 32–bit microcoded microprocessor with a load–store architecture. *AVM–1* was not implemented, but a pipelined version is currently being specified, designed, and verified for implementation.

2.1.1 Microprocessor Specification.

The specifications for the microprocessors mentioned above appear very different on the surface; in fact, the specification to FM8501 is even in a different language. On closer inspection, however, we find that each of them (as well as many others) use the same implicit behavioral model. In general, the model uses a state transition system to describe the microprocessor. A microprocessor specification has four important parts:

1. A representation of the state, **S**. This representation varies depending on the verification system being used.
2. A set of state transition functions, **J**, denoting the behavior of the individual instructions of the microprocessor. Each of these functions takes the state defined in step (1) as an argument and returns the state updated in some meaningful way.
3. A selection function, **N**, that selects a function from the set **J** according to the current state.
4. A predicate, **I**, relating the state at time $t + 1$ to the state at time t by means of **J** and **N**.

Discovering the similarity in previous efforts provided a common basis for studying microprocessor verification. To make the commonality usable, we have formalized a microprocessor specification model. The formalization provides a step–by–step approach to microprocessor specification by enumerating the important definitions that need to be made for any microprocessor specification. It also explicitly tells us what needs to be proven to verify an interpreter.

The formalization models an interpreter. An interpreter is a computing structure with one control point. One of the many available instructions is chosen at this control point based on the current state and inputs. The state is then processed by this instruction and the cycle begins again. Interpreters differ from standard state machine models only in form, not in power; but, their form is particularly amenable to modeling microprocessors and other similar computing devices.

Table 1: The abstract functions and their types for the generic interpreter model.

Operation	*Type*
`instructions`	`":*key -> (*state->*env->*state)"`
`select`	`":*state->*env->*key"`
`output`	`":*key->(*state->*env->*out)'"`
`subout`	`":*out'->*out"`
`substate`	`":*state'->*state"`
`subenv`	`":*env'->*env"`
`Impl`	`":(time'->*state')->(time'->*env')->(time'->*out') ->bool"`
`sync`	`":*state'->*env'->bool"`

2.1.2 The Abstract Representation

The formalization is done in the HOL theorem proving system [Gor88] using the abstract theory package [Win92a]. An abstract theory is parameterized so that some of the types and constants in the theory are undefined inside the theory except for their syntax and an algebraic specification of their semantics. The abstract representation is the interface to the generic model and crucial to the work reported here since it the abstract representation that structures the specification. We specify the abstract representation of the generic interpreter by defining a list of abstract objects and operations. Table 1 shows the operations and their types.

The following abstract types are used in the representation.

- `:*state` represents the state.
- `:*env` represents the environment. The environment provides inputs to the interpreter.
- `:*key` is type containing all of the keys. Keys are used to select instructions. For example, the opcodes form the keys at the macro–level.
- `:*out` represents the output of an interpreter.

Using the functions from the abstract representation, we can define an interpreter. The form of this definition is essentially the same as the form of specific microprocessor specifications given in [Coh88, Hun87, Joy89].

```
⊢def ∀gi s e p.
    INTERP gi s e p =
    (∀t.
      let k = select gi(s t)(e t)
      in
       ((s(t + 1) = instructions gi k(s t)(e t)) ∧
        (p t = output gi k(s t)(e t))))
```

The specification of an interpreter is a predicate relating the contents of the state stream at time $t + 1$ to the contents of the state stream at time t. The relationship is defined using the functions from the abstract representation.

The generic interpreter theory also contains numerous theorems about the abstract representation. The most important is a correctness proof that shows that `Impl` implements `INTERP` when the theory obligations are satisfied. The theory obligations that have to be proven for an interpreter are `INST_CORRECT` and `OUTPUT_CORRECT`. The definitions for these are given below.

```
INST_CORRECT =
⊢def ∀s' e' p' k gi.
    INST_CORRECT gi s' e' p' k =
    implementation gi s' e' p' ⇒
    (∀t.
      let s t' = substate gi(s' t')
      in
       let e t' = subenv gi(e' t')
       in
        let f t' = sync gi(s' t')(e' t')
        in
         ((select gi(s t)(e t) = k) ∧ f t ⇒
          (∃c.
            Next f(t,t + c) ∧ (instructions gi k(s t)(e t) = s(t + c)))))
```

`INST_CORRECT` states that given a `select` function and a synchronization function (`sync`), there exists a time `c` where the implementation level and the interpreter level are synchronized, and at that time (t+c) the state is equal to the output of the `instructions` function.

```
OUTPUT_CORRECT =
⊢def ∀s' e' p' k gi.
    OUTPUT_CORRECT gi s' e' p' k =
    implementation gi s' e' p' ⇒
    (∀t.
      let s t' = substate gi(s' t')
      in
       let e t' = subenv gi(e' t')
       in
        let p t' = subout gi(p' t')
        in
         let f t' = sync gi(s' t')(e' t')
         in
          ((select gi(s t)(e t) = k) ∧ f t ⇒
           (p t = output gi k(s t)(e t))))
```

`OUTPUT_CORRECT` states that given a `select` function and a synchronization predicate, the output of the implementation level filtered through the function `subout` is equal to the output of the interpreter.

2.1.3 Specification and Verification

Specifying an interpreter is done by creating a concrete definition for each of the functions in Table 1. Then, using these concrete definitions, we instantiate a concrete theorem of the form of INTERP (this process will be shown in more detail in section 4.9). To verify the interpreter, we have to prove the theory obligations for our instantiated interpreter.

For a microprogrammed processor such as Tamarack there are typically three interpreters we specify: the macroprogramming level, the microprogramming level, and the phase level. Each interpreter specification defines behavior as implemented by a lower level. For example the micro level interpreter defines the behavior of the microinstructions implemented by the phase level. Using this idea we can build a hierarchy of levels. The lowest level in our hierarchy is the hardware definition or electronic block model which implements the phase level interpreter. Using this hierarchy we can break down the verification to a series of smaller proofs.

- EBM $\Longrightarrow$ Phase Level
- Phase Level $\Longrightarrow$ Micro Level
- Micro Level $\Longrightarrow$ Macro Level

From these smaller proofs we can derive the final correctness statement: EBM $\Longrightarrow$ Macro Level. It may be possible to obtain a final correctness lemma by proving the hardware properly implements the assembly language instructions. However this proof would likely be very tedious, very unstructured, and very time consuming. The generic interpreter theory places a structure on the verification; the proof is naturally broken down into smaller, more tractable proofs.

2.2 Tamarack

Tamarack-3 is a simple microprocessor that has been verified many times, most recently by Jeff Joyce [Joy89]. A main goal of this verification was not to change any of Joyce's original definitions. Many of them are completely unchanged. Some of them however were changed slightly to fit the GIT. Most of the changes only dealt with changing the parameters of functions. On the other hand, a great deal of the definitions presented here are original and were created to meet the specifications of the generic interpreter theory.

At the assembly language level, Tamarack has just eight instructions, which are summarized in Table 2 [2]. Each instruction is a full-word in length. The first three bits of each instruction are the opcode, and the remaining bits are the address of an operand. This address is the absolute memory address of a data operand.

Tamarack is a microprogrammed processor with sixteen microinstructions. Joyce did not give a behavioral definition for each microinstruction; he created one definition, `MicroCode`, which defined the control line configurations that the hardware uses to implement the microinstructions. The function is actually a select function which returns the appropriate microinstruction based on the integer value of the microprogram counter. Each assembly language instruction in Table 2 is implemented by a series of these microinstructions.

[2] Table 2 is taken from [Joy89, page 12]

Instruction	Opcode Value	Effect
JZR	0	jump if zero
JMP	1	jump
ADD	2	add accumulator
SUB	3	subtract accumulator
LDA	4	load accumulator
STA	5	store accumulator
RFI	6	return from interrupt
NOP	7	no operation

Table 2: TAMARACK-3 Instruction Set

```
MicroCode =
⊢def  MicroCode n =
      ((n = 0) →  ((‘rpc‘,‘wmar‘,‘none‘),waitidle,jireq,1,2)       |
      ((n = 1) →  ((‘rpc‘,‘wrtn‘,‘zero‘),continue,jump,12,0)       |
      ((n = 2) →  ((‘rmem‘,‘wir‘,‘none‘),waitdack,jump,3,0)        |
      ((n = 3) →  ((‘rir‘,‘wmar‘,‘none‘),waitidle,jopcode,4,0)     |
      ((n = 4) →  ((‘none‘,‘none‘,‘none‘),continue,jzero,5,11)     |
      ((n = 5) →  ((‘rir‘,‘wpc‘,‘none‘),continue,jump,0,0)         |
      ((n = 6) →  ((‘racc‘,‘warg‘,‘none‘),waitidle,jump,13,0)      |
      ((n = 7) →  ((‘racc‘,‘warg‘,‘none‘),waitidle,jump,14,0)      |
      ((n = 8) →  ((‘rmem‘,‘wacc‘,‘none‘),waitdack,jump,11,0)      |
      ((n = 9) →  ((‘racc‘,‘wmem‘,‘none‘),waitdack,jump,11,0)      |
      ((n = 10) →  ((‘rrtn‘,‘wpc‘,‘none‘),continue,jump,0,0)       |
      ((n = 11) →  ((‘rpc‘,‘none‘,‘inc‘),continue,jump,12,0)       |
      ((n = 12) →  ((‘rbuf‘,‘wpc‘,‘none‘),continue,jump,0,0)       |
      ((n = 13) →  ((‘rmem‘,‘none‘,‘add‘),waitdack,jump,15,0)      |
      ((n = 14) →  ((‘rmem‘,‘none‘,‘sub‘),waitdack,jump,15,0)      |
      ((‘rbuf‘,‘wacc‘,‘none‘),continue,jump,11,0)))))))))))))))
```

Each microinstruction listed in the above definition is defined as a tuple of values. Each of these values is the value of a hardware control line. These control lines initiate the hardware actions necessary to perform the microinstruction. This method of defining the microinstructions represents the phase level even though there is not a separate behavioral definition for the phase level.

The architectural layout of Tamarack is given in Figure 1[3]. The important detail to notice about the architecture is that all the registers and control blocks are attached to the system bus with no delay latches. This gives all the components instant access to the bus which means that all RTL actions happen in one phase or cycle. In order for this architecture to work, we need something to control access to the bus. To ensure that only one device is using the bus at any given time, Joyce defined the function `BusOkay` [Joy89, page 39]. `BusOkay` regulates when a value can be read onto the system bus. It does this by analyzing these control lines:

`rmem, rpc, racc, rir, rrtn, and rbuf`

`BusOkay` asserts that the bus is free if only one of the read control lines is set to true.

[3] Figure 1 is taken from [Joy89, page 20]

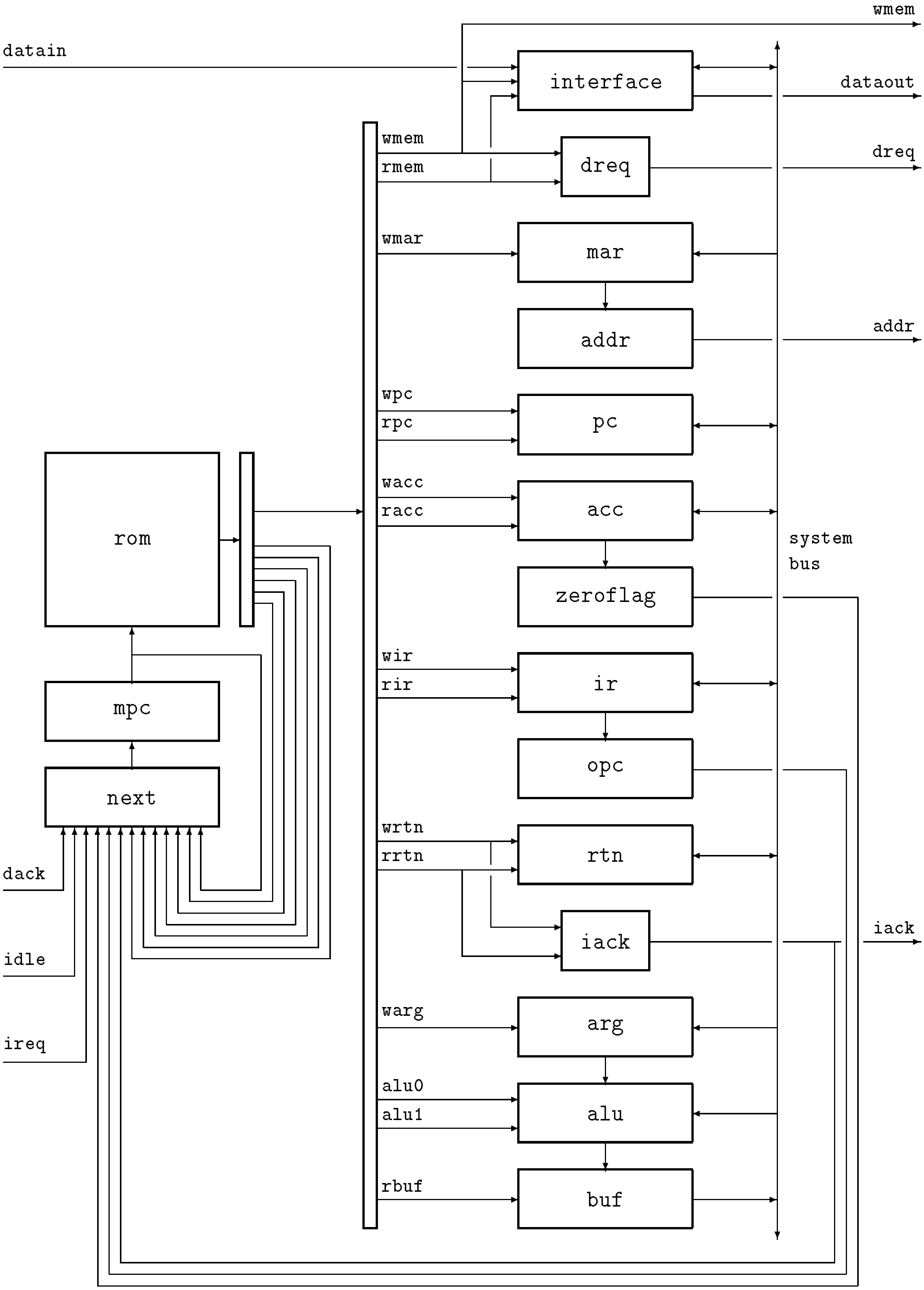

Figure 1: Register-Transfer Level Architecture.

Clearly, Tamarack breaks down nicely into the hierarchical levels we need for the generic interpreter theory: the macroprogramming level, the microprogramming level, the phase level, and the electronic block model. However, we need to fill in several holes to make a complete specification of each level.

2.3 Defining the Abstract Representation

The abstract representation used in this proof is almost exactly the same as the one Joyce used. One element was added: `word3`. `Word3` takes an integer argument and returns a `*word3` (3 bit word).

Using the abstract theory package described in [Win92a], the abstract representation was created with this definition:

```
let tam_rep_LEMMA  = new_abstract_representation  `tam_rep` [
        (`iszero`,      ":(*wordn -> bool)                             ");
        (`inc`,         ":(*wordn -> *wordn)                           ");
        (`add`,         ":(*wordn # *wordn -> *wordn)                  ");
        (`sub`,         ":(*wordn # *wordn -> *wordn)                  ");
        (`wordn`,       ":(num -> *wordn)                              ");
        (`valn`,        ":(*wordn -> num)                              ");
        (`opcode`,      ":(*wordn -> *word3)                           ");
        (`val3`,        ":(*word3 -> num)                              ");
        (`word3`,       ":(num->*word3)                                ");
        (`address`,     ":(*wordn -> *address)                         ");
        (`fetch`,       ":(*memory # *address -> *wordn)               ");
        (`store`,       ":(*memory # *address # *wordn -> *memory)     ");
        (`word4`,       ":(num -> *word4)                              ");
        (`val4`,        ":(*word4 -> num)                              ");
        ];;
```

The abstract representation `tam_rep` is just a type. The type is represented as a 14-tuple. Each element of the tuple is a primitive operation on uninterpreted data types. The uninterpreted types are:

- `*wordn`: An n-bit word.
- `*word3`: A three bit word.
- `*word4`: A four bit word.
- `*memory`: A type representing memory.
- `*address`: A memory address.

The functions in the first column like `iszero` are selector functions. For instance, `address` selects the tenth element of the abstract representation tuple which is the primitive operation taking a n-bit word and returning the address specified in the word.

2.4 Theory Obligations

In Joyce's proof, two assumptions were needed for verification purposes: `Val4Word4_ASM` and `Val3_CASES_ASM` . For this verification, we needed two more assumptions. All four assumptions are listed below.

- `Val3_CASES_ASM` states that for every 3-bit word, the primitive operation `val3` will return a number less than 8.

```
Val3_CASES_ASM = ⊢def  Val3_CASES_ASM rep = (∀w. (val3 rep w) < 8)
```

- `Word4_One2One_ASM` states that if two 4-bit words are equal then the two numbers the words represent are equal.

```
Word4_One2One_ASM =
⊢def  Word4_One2One_ASM rep =
    (∀m n. (word4 rep n = word4 rep m) ⇒  (n = m))
```

- `Val4_CASES_ASM` states that for every 4-bit word the primitive operation `val4` will return a number less than 16.

```
Val4_CASES_ASM = ⊢def  Val4_CASES_ASM rep = (∀w. (val4 rep w) < 16)
```

- `Val4Word4_ASM` states that the primitive operations `val4` and `word4` are inverses of each other for numbers less than sixteen.

```
Val4Word4_ASM =
⊢def  Val4Word4_ASM rep = (∀n. n < 16 ⇒  (val4 rep(word4 rep n) = n))
```

Without these assumptions, there is insufficient information to complete the verification. For example, the function `word4` is uninterpreted, so

`(word4 rep 5) = (word4 rep 7)`

has no meaning without an assumption stating the **word4** is a one-to-one function. The two `cases` assumptions (`Val4_CASES_ASM` and `Val3_CASES_ASM`) are used to provide a basis for case analysis in the verification.

Since we are using the abstract theory package in [Win92a], we shall define these assumptions to be theory obligations. This means that in order for our verification to be valid, these four underlying assumptions must be met. Moreover, since we defined these assumptions as theory obligations, the abstract theory package will automatically include them as assumptions in any of our HOL proofs. The ML function `new_theory_obligations` is used to define our theory obligations.

```
new_theory_obligations
 (`TAMARACK`,
  "∀ (rep:^rep_ty). TAMARACK rep =
     Val4_CASES_ASM rep ∧
     Word4_One2One_ASM rep ∧
     Val3_CASES_ASM rep ∧
     Val4Word4_ASM rep ");;
```

3 The Electronic Block Model

For the most part, the specification of the electronic block model is the same as [Joy89]. All of Joyce's original definitions are used unchanged, except for `TamarackImp` which defines the connection between the `Control Unit` and the `Datapath`. The definitions for the `Control Unit` and `Datapath` and all of their smaller components remain the same. These definitions are listed in appendix A.

3.1 EBM State

Before the specification for the EBM model can be given, the state and environment must be defined.

The state of a microprocessor is traditionally defined as the program counter, the registers, status flags, and anything else that would need to be saved if the state of processor needed to be preserved. Traditionally, memory is also included in the state, since the assembly language programmer sees it as part of the state.

For Tamarack, the state at the EBM level consists of the following:

- `pc` : the program counter. Type: `*wordn`.
- `acc` : The accumulator. Type: `*wordn`.
- `mem` : Memory. Type: `*memory`.
- `ir` : The instruction register. Type: `*wordn`.
- `rtn` : The return address register. Type: `*wordn`.
- `arg` : A register that holds arguments for ALU operations. Type: `*wordn`.
- `buf` : The buffer register. Temporary storage for ALU results. Type: `*wordn`.
- `mar` : The memory address register. Type: `*wordn`.
- `iack` : The interrupt acknowledgment flag. Type: `bool`.
- `mpc` : The microprogram counter. Type: `*word4`.

State is often represented by a tuple, but we find it more conveniant to use an abstract data type. The following definition creates the state data type for the EBM level. `Create_record` takes as its input a list of string–type pairs. The first element in the pair is the name of the selector function for that element of the state. The second element of the pair is the type for the state element. For example, for the state element `pc` the selector function is `Pc` and the type of the program counter is `*wordn`.

The function `create_record` does a tremendous amount of work behind the scenes. It creates the state data type, the selector functions on the state, the update functions for the state, and some useful theorems about the state. For verification purposes the most important of these is the selector functions. Selector functions take a state and return the value of a specific element of the state. For instance, the selector `Pc` takes a state and returns the value of the program counter.

```
create_record `EBM_State`
  [
    (`Pc`, ":*wordn");
    (`Acc`,":*wordn");
    (`Mem`,":*memory");
    (`Ir`,":*wordn");
    (`Rtn`,":*wordn");
    (`Arg`,":*wordn");
    (`Buf`,":*wordn");
    (`Mar`,":*wordn");
    (`Iack`,":bool");
    (`Mpc`,":*word4");
    ];;
```

3.2 The Environment

The environment consists of the control lines that connect Tamarack to the outside world. For this verification, we will verify Tamarack in fully synchronous mode as shown in Figure 2.[4] This means that all reads and writes to memory are completed in one cycle. The processor does not have to wait to read or write data. To achieve this, the `dack` and `idle` pins are wired to high.

For Tamarack, the environment consists of [5]:

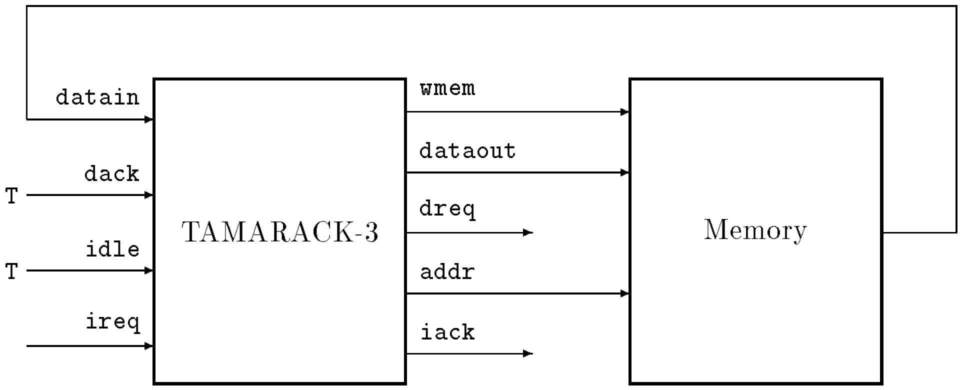

Figure 2: Fully Synchronous Operation.

- `ireq` : The interrupt request flag. Type: `bool`.
- `idle` : The idle flag. This flag and the **dack** flag put the processor into fully synchronous mode when both are wired to high. Type: `bool`.
- `addr` : The address register. This holds the address of the **mar** . Type: `*address`.

[4] Figure 2 was taken from [Joy89, page 16]

[5] The interrupt acknowledgment flag, `iack` is defined in the state so as to be consistent with Joyce's definition of the externally visible state [Joy89, page 11]

- `dack` : Data acknowledgment flag. Type: `bool`.
- `wmem` : Flag denoting if memory is being written to. Type: `bool`.
- `dreq` : The data request flag. Type: `bool`.
- `dataout` : A word of data being written to memory. Type: `*wordn`.
- `datain` : A word of data coming in from memory. Type: `*wordn`.

The following definition shows the definition of the environment.

```
create_record `Env`
  [
    (`Ireq`,":bool");
    (`Idle`,":bool");
    (`Addr`,":*address");
    (`Dack`,":bool");
    (`Wmem`,":bool");
    (`Dreq`,":bool");
    (`Dataout`,":*wordn");
    (`Datain`,":*wordn");
   ];;
```

3.3 The EBM Specification

The Electronic Block model is defined to be the composition of the `Datapath` and `Control Unit`. Existential quantification is used to hide internal lines. Our definition is the same as that of Joyce in spirit but has been altered to fit the generic interpreter theory. In this EBM definition, the `SynMemory` definition is also included. Joyce did not include the memory specification until he proved the macro level, but it is included here just to be consistent among each level of the hierarchy. With some rewriting, Joyce's definition `SynSystem` could be transformed into the definition that follows (if the same state and environment types are used).

```
tamarackEBM =
⊢def tamarackEBM rep s e p =
    (∃zeroflag opc cntls.
      CntlUnit rep (Dack o e,Idle o e,Ireq o e,Iack o s,zeroflag,opc,Mpc o s,cntls)
      ∧
      DataPath rep (cntls,Datain o e,Mar o s,Pc o s,Acc o s,Ir o s,Rtn o s,Iack o s,
       Arg o s,Buf o s,Dataout o e,Wmem o e,Dreq o e,Addr o e,zeroflag, opc)
      ∧
      SynMemory rep(Wmem o e,Addr o e,Dataout o e,Mem o s,Datain o e))
```

To fit the generic interpreter theory the EBM definition must have type:

```
:(time->*state')->(time->*env')->(time->*out')->bool
```

The reason for this is that the EBM definition will be used to instantiate the abstract function `Impl` for the phase level. Notice than s, e, and p (the variables representing state, environment and output respectively) are specified as functions of time. This causes a problem since our selectors for state and environment are not defined on a type of "`:time->state`". We use function composition to solve our problem. Consider the term "`Dack o e`" used in the definition above. We compose the two functions to get a function which will return the `dack` value of an environment function of the type "`:time->env`".

4 The Phase Level

The EBM level is not an interpreter, since it is structural rather than behavioral in nature, so the phase level is our lowest level interpreter. In [Joy89], Joyce did not specify a phase level. He did verify the phase level by proving that that EBM definition, `TamarackImp`, implies the correct execution of each microinstruction. Although this was valid, we believe that the lack of a methodology here made the verification more difficult. That is, a great deal of thought had to be undertaken to determine the correct theorems to prove to insure that the phase level was correct. By specifying the phase level as an interpreter, we know exactly what lemmas to prove to show the correctness of the phase level. What follows are all original definitions.

4.1 Phase State

The state of the microprocessor at the phase level is exactly the same as that of the EBM model presented in section 3.1.

4.2 Phase Environment

The environment for the phase level is the same as that of the EBM presented in section 3.2.

4.3 The Phase Level Instruction

The GIT is based on the idea that an interpreter takes an input of a state and environment, executes a function on the state called a state transition function, and returns a new state. As mentioned earlier, because of the architectural layout of Tamarack, everything happens in one clock cycle or one phase. Because of this, the phase level contains only one instruction. The definition for this instruction is based entirely on the Figure 9 of [Joy89]. The function `phase_level`, defined below, takes a state and environment and updates the state according to the value of the `mpc`. The definition is very long and listed on the next page.

```
phase'level =
⊢def phase'level rep s e =
   (let pc = Pc s
    and acc = Acc s
    and addr = Addr s
    and mem = Mem s
    and ir = Ir s
    and rtn = Rtn s
    and arg = Arg s
    and buf = Buf s
    and mar = Mar s
    and iack = Iack s
    and mpc = Mpc s
    and ireq = Ireq e
    and idle = Idle e
    and dack = Dack e
    and wmem = Wmem e
    and dreq = Dreq e
    and dataout = Dataout e
    and datain = Datain e
    in
     let new'pc = ((val4 rep mpc = 5) → ir |
           ((val4 rep mpc = 10) → rtn |
            ((val4 rep mpc = 12) → buf | pc)))
     in
      let new'acc = ((val4 rep mpc = 8) → fetch rep(mem,address rep mar) | ((val4 rep mpc = 15) → buf | acc))
      in
       let new'mar = ((val4 rep mpc = 0) → pc | ((val4 rep mpc = 3) → ir | mar))
       in
        let new'ir = ((val4 rep mpc = 2) → fetch rep(mem,address rep mar) | ir)
        in
         let new'rtn = ((val4 rep mpc = 1) → pc | rtn)
         in
          let new'arg = ((val4 rep mpc = 6) → acc | ((val4 rep mpc = 7) → acc | arg))
          in
           let new'buf = ((val4 rep mpc = 11) → inc rep pc |
                 ((val4 rep mpc = 13) → add rep(arg,fetch rep(mem,address rep mar)) |
                  ((val4 rep mpc = 14) → sub rep(arg,fetch rep(mem,address rep mar)) | wordn rep 0)))
           in
            let new'mem = ((val4 rep mpc = 9) → store rep(mem,address rep mar,acc) | mem)
            in
                let new'iack = ((val4 rep mpc = 1) → T | ((val4 rep mpc = 10) → F | iack))
             in let new'mpc =
                  ((val4 rep mpc = 0) → ((¬(idle ∨ ¬dack)) → mpc | ((ireq ∧ ¬iack) → word4 rep 1 | word4 rep 2)) |
                   ((val4 rep mpc = 1) → word4 rep 12 |
                    ((val4 rep mpc = 2) → ((¬dack) → mpc | word4 rep 3) |
                     ((val4 rep mpc = 3) → ((¬(idle ∨ ¬dack)) → mpc | word4 rep((val3 rep(opcode rep ir)) + 4)) |
                      ((val4 rep mpc = 4) → (iszero rep acc → word4 rep 5 | word4 rep 11) |
                       ((val4 rep mpc = 5) → word4 rep 0 |
                        ((val4 rep mpc = 6) → ((¬(idle ∨ ¬dack)) → mpc | word4 rep 13) |
                         ((val4 rep mpc = 7) → ((¬(idle ∨ ¬dack)) → mpc | word4 rep 14) |
                          (((val4 rep mpc = 8) ∨ (val4 rep mpc = 9)) → (¬dack) → mpc | word4 rep 11) |
                           ((val4 rep mpc = 10) → word4 rep 0 |
                            ((val4 rep mpc = 11) → word4 rep 12 |
                             ((val4 rep mpc = 12) → word4 rep 0 |
                 (((val4 rep mpc = 13) ∨ (val4 rep mpc = 14)) → ((¬dack) → mpc | word4 rep 15) | word4 rep 11))))))))))))))
              in
             EBM'State new'pc new'acc new'mem new'ir new'rtn new'arg new'buf new'mar new'iack new'mpc)
```

4.4 The `select` Function

The `select` function takes as input a state and an environment and returns a value of type `*key`. Since there is only one instruction in our phase level, the select function is trivial and just returns the value `one`.

```
⊢def Phase_select rep s e = one
```

4.5 Defining the `instructions` function

The `instructions` function takes as input the key returned by `select` and returns a state transition function. `Phase_instr` takes as input a key of type `one` and returns the function `phase_level`.

```
⊢def  phase_instr rep k = phase_level rep
```

4.6 The `output` Function

The `output` function defines the output of the interpreter specified. Tamarack has no output at all, so `phase_output` is defined to return the type `one`. The reason we return the type `one`is that is trivial, and it makes it easier to reason about correctness of the output.

```
⊢def  phase_output rep k s e = one
```

4.7 `subout`, `substate`, and `subenv`

There is no data abstraction between the phase level and the EBM model; The state, environment, and output are all the same for the two levels. So, `subout` , `substate` , and `subenv` are all defined to be the identity function `I`.

4.8 The Synchronization Predicate

The abstract theory function `sync` takes a state and an environment and returns a boolean value. This boolean value denotes if the the two levels are synchronized. Since there is no temporal abstraction between the electronic block model and the phase level interpreter, the concrete function `phase_sync` is trivially defined to return True. That is, the two levels are always synchronized.

```
phase_sync =  phase_sync rep s e = T
```

4.9 Defining the Phase Level Interpreter

The following HOL session shows how to instantiate the phase level interpreter. We start by specifying the representation for the phase level.

```
let GI_REP =
"GI
 (phase_instr rep)
 (Phase_select rep)
 (phase_output rep)
 I
 I
 I
 (tamarackEBM rep)
 (phase_sync rep)";;
```

GI_REP is just a term. Each of the functions listed in the term is a concrete function which we use to instantiate the abstract functions listed in Table 1. For instance, the function Phase_select is the concrete function used to instantiate the abstract function select.

We now define our interpreter. The definition phase_Interp_def states that the phase level interpreter is equal to the abstract function INTERP with the arguments of our defined interpreter representation, state, environment, and output.

```
phase_Interp_def =
⊢def phase_Interp rep s e p =
    INTERP
    (GI
     (phase_instr rep)
     (Phase_select rep)
     (phase_output rep)
     I
     I
     I
     (tamarackEBM rep)
     (phase_sync rep)
    s
    e
    p
```

We now take this definition and instantiate the final theorem describing the phase level interpreter. GI_type is the abstract data type used in abstract theory used to define interpreter model. We use the function instantiate_abstract_definition to instantiate the desired theorem. This function takes four arguments: a theory, a definition in the theory to be instantiated, the concrete definition, and a list of variables and their types to instantiate. Here we are instantiating the definition INTERP from the theory GI_def with the concrete definition phase_Interp_def and the abstract variable gi is instantiated with the representation we defined–GI_REP.

```
let GI_type = abs_type_info (theorem `GI_def` `GI`);;


let Phase_Interp = save_thm
  (`Phase_Interp`,
  instantiate_abstract_definition `GI_def` `INTERP` phase_Interp_def
    [ "gi:^GI_type", GI_REP;
     ]);;
```

The resulting theorem is shown below.

```
⊢ ∀rep s e p.
    phase_Interp rep s e p =
    (∀t.
      let k = Phase_select rep(s t)(e t)
      in
       ((s(t + 1) = phase_instr rep k(s t)(e t)) ∧
        (p t = phase_output rep k(s t)(e t))))
```

5 The Microprogram Level

The next level in our interpreter hierarchy is the microprogram level interpreter. The phase level defined how microinstructions were implemented by the lower level hardware. The microprogram level defines the behavior of the microinstructions.

5.1 The Micro State

The state of the microprocessor at the microprogram level is the same as the state at the phase level and EBM level. However, we define a new data type for the micro level state just to show that the state transition functions for this level update the micro level state.

```
create_record `Micro_State`
  [
    (`uPc`, ":*wordn");
    (`uAcc`,":*wordn");
    (`uMem`,":*memory");
    (`uIr`,":*wordn");
    (`uRtn`,":*wordn");
    (`uArg`,":*wordn");
    (`uBuf`,":*wordn");
    (`uMar`,":*wordn");
    (`uIack`,":bool");
    (`uMpc`,":*word4");
    ];;
```

5.2 Environment

The environment for the microprogram level is the same as the one described in section3.2.

5.3 Microinstructions

In [Joy89], Joyce defined the microcode in the function `Microcode`. However, he did not give a full behavioral definition of each microinstruction. Once again, the definitions for the microinstructions that follow are original and based on `figure 26` of [Joy89]. Only the microinstruction `u0` is listed here. All of the microinstructions are listed and explained in Appendix B.

```
⊢def  u0 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
      let new_mpc = ((¬(idle ∨ ¬dack)) →  mpc |
             ((ireq ∧ ¬iack) →  word4 rep 1 | word4 rep 2)) in
       let new_mar = pc in
         let new_buf = wordn rep 0 in
      Micro_State pc acc mem ir rtn arg new_buf new_mar iack new_mpc)
```

`u0` takes three arguments: the abstract representation, `rep`, the state, `s`, and the environment `e`. The definition needs access to the components of the state (i.e. mem, pc etc.). The `let` statements in the first part of the definition break the elements of the state out for later use.

In microinstruction zero, three things occur: the memory address register is set to the program counter, a new microprogram counter is computed, and the buffer register is set to zero.

To compute the next microprogram counter, the machine checks to see if it is waiting for data to arrive from memory. If so, then the `mpc` does not change. If Tamarack is not idle, then Tamarack checks to see if an interrupt has occurred. If so, then the new `mpc` is 1, otherwise the `mpc` is 2. All of this is done is this line:

```
let new_mpc =
              ((~(idle \/ ~dack)) =>
               mpc |
               ((ireq /\ ~iack) => word4 rep 1 | word4 rep 2)) in
```

Next, the memory address register, `mar`, gets the value of the `pc`. This initiates the instruction fetch. It is done in the line:

```
let new_mar = pc
```

The buffer register receives a value of 0 (`wordn rep 0`) since no ALU operation has taken place. This happens during every microinstruction if there is no ALU operation.

The last line in the definition of `u0`creates the new state and returns it.

5.4 The `select` Function

Once again the `select` function takes a state and environment and returns a value of type `*key`. For the microprogram level, the key is the microprogram counter, `mpc`. So, the function `micro_select` is simply equal to the Micro State selector `uMpc` which returns the current value of the **mpc**.

```
micro_select = ⊢def  micro_select rep s e = uMpc s
```

5.5 The `instructions` Function

The `instructions` function for the microprogram level is not trivial like it's phase level counterpart. There are sixteen microinstructions, and the `key`, returned from `micro_select` is of type `*word4`. The function `micro_instr` takes the key returned from `micro_select`, and returns a microinstruction. The function picks the next microinstruction by comparing the value of `val4 rep k` which is a value between zero and fifteen (by the assumption `Val4Word4_ASM)`.

```
⊢def micro'instr rep k =
  ((val4 rep k = 0) → u0 rep |
  ((val4 rep k = 1) → u1 rep |
   ((val4 rep k = 2) → u2 rep |
    ((val4 rep k = 3) → u3 rep |
     ((val4 rep k = 4) → u4 rep |
      ((val4 rep k = 5) → u5 rep |
       ((val4 rep k = 6) → u6 rep |
        ((val4 rep k = 7) → u7 rep |
         ((val4 rep k = 8) → u8 rep |
          ((val4 rep k = 9) → u9 rep |
           ((val4 rep k = 10) → u10 rep |
            ((val4 rep k = 11) → u11 rep |
             ((val4 rep k = 12) → u12 rep |
              ((val4 rep k = 13) → u13 rep |
               ((val4 rep k = 14) → u14 rep | u15 rep)))))))))))))))
```

5.6 The `output` Function

The `output` function is once again trivially defined.

```
Micro_output = ⊢def  Micro_output rep k s e = one
```

5.7 `subout`, `substate`, `subenv`

There is no data abstraction between the environment of the phase level and that of the microprogram level, so the function `subenv` is defined to be the identity function `I`. This is also the case for `subout`.

Even though there is no data abstraction between the micro state and the EBM state, we have two different abstract data types defining them. So the function `Phase_substate` is used to show the corresponding conversion between the two states.

```
⊢def  Phase_substate
    (EBM_State pc acc addr mem ir rtn arg buf mar iack mpc) =
    Micro_State pc acc mem ir rtn arg buf mar iack mpc
```

5.8 The Synchronization Predicate

Since each microinstruction is takes only one phase or cycle to execute, there is no temporal abstraction between the microprogramming level and the phase level. Once again, our `sync` function is trivially defined to return `true`.

```
micro_sync = ⊢def  micro_sync rep s e = T
```

5.9 Defining the Micro Level Interpreter

The microprogram level interpreter is defined and instantiated in the same manner as that of the phase level. The only difference is that the representation we use to instantiate, `GI_REP`, is defined for the microprogram level. The representation defines the relationship between the implementation, the phase level, and the interpreter, the microprogram level.

```
GI_REP =
"GI
 (micro_instr rep)
 (micro_select rep)
 (Micro_output rep)
 Phase_substate
 I
 I
 (phase_Interp rep)
 (micro_sync rep)"
```

6 The Macro Level

This is the highest level in our hierarchy. It is the level that models what the assembly language programmer sees. The microprogramming level defined the behavior of the microinstructions. The macroprogramming level defines the behavior and semantics of the macroinstructions or assembly language instructions. Once again start by defining the state and environment.

6.1 Macro State

While there was no data abstraction between the lower level interpreters, there is a great deal of data abstraction at the macro level (for our purposes, data abstraction deals only with the hiding of elements of the state). The state of the microprocessor at the macroprogramming level consists of only five elements:

- **mem**
- **pc**
- **acc**
- **rtn**
- **iack**

6.2 Environment

The environment for the macro level is the same as the one defined in section 3.2.

6.3 Macroinstructions

As shown in Table 2, Tamarack has eight users instructions. Tamarack also has one privileged instruction for processing interrupts: IRQ. Since IRQ is not available to the user we refer to it as a pseudo-instruction.

The first step in specifying the macroinstrctions is to define an abstract data type to represent their syntax. The HOL session below defines the `Instruction` type for Tamarack.

```
let Instruction_axiom = define_type `Instruction`
  `Instruction =
   JZR                              |
   JMP                              |
   ADD  *address                    |
   SUB  *address                    |
   LDA  *address                    |
   STA  *address                    |
   RFI                              |
   NOP
    `;;
```

The `Instruction` type is used to define the syntax of the macroinstructions. For instance, the type definition tells us that the ADD instruction has the format:

```
ADD  *address
```

That is, `*address` is the address in memory where the value to be added to the accumulator resides.

Using the `Instruction` type, we specify the behavior of each macroinstruction. Except for the modifications due to state and instruction types, these definitions are the same as Joyce's.

Only one macroinstruction is listed here, but the rest are listed in Appendix C.

```
M_ADD =
⊢def ∀rep adr s e.
    M_ADD rep adr s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      let operand = fetch rep(mem,adr)
      in
       let new_pc = inc rep pc
       in
        let new_acc = add rep(acc,operand)
        in
         State mem new_pc new_acc rtn iack)
```

The definition `M_ADD` (Meaning of ADD) takes four arguments: The abstract representation, an address, a state, and an environment. The `let` statements once again break out the necessary elements from the state by the use of selectors. To start the ADD instruction, we first fetch the operand that is to be added to the accumulator. This is specified by the line:

```
let operand = fetch rep (mem,adr)
```

This is the only significant departure from Joyce's definition. In Joyce's definition, the `adr` is calculated by two fetches to memory [Joy89, page 50]

The next line updates the program counter, `pc`, by incrementing it. Next, the accumulator, `acc`, is updated by performing the add with the specified operand. Finally, the new state is created and returned.

After the semantics of each instruction are specified, a definition combining them is needed. The function `M_INST` listed below defines the semantics of any instruction. If it wasn't for the fact that we also have the interrupt request instruction, IRQ, this function could serve as our `instructions` function. It takes a key of the type `Instruction` and returns the proper state transition function to execute. However, IRQ is a legal instruction, so `M_INST` only specifies the behavior of the `instructions` function when a user instruction is to be executed. We will address this problem some more in the next two sections.

```
M_INST =
⊢def (∀rep. M_INST rep JZR = M_JZR rep) ∧
   (∀rep. M_INST rep JMP = M_JMP rep) ∧
   (∀rep adr. M_INST rep(ADD adr) = M_ADD rep adr) ∧
   (∀rep adr. M_INST rep(SUB adr) = M_SUB rep adr) ∧
   (∀rep adr. M_INST rep(LDA adr) = M_LDA rep adr) ∧
   (∀rep adr. M_INST rep(STA adr) = M_STA rep adr) ∧
   (∀rep. M_INST rep RFI = M_RFI rep) ∧
   (∀rep. M_INST rep NOP = M_NOP rep)
```

6.4 The `select` Function

For the macro level interpreter, the `select` function takes a state and an environment and returns a key which is used by the `instructions` function to select the appropriate state transition function. The `select` function decodes the assembly language opcode. The function below implements the decoding of assembly language instructions.

```
decw =
⊢def decw rep w = ((val3 rep(opcode rep w) = 0) →  JZR |
     ((val3 rep(opcode rep w) = 1) →  JMP |
      ((val3 rep(opcode rep w) = 2) →  ADD(address rep w) |
       ((val3 rep(opcode rep w) = 3) → SUB(address rep w) |
        ((val3 rep(opcode rep w) = 4) →  LDA(address rep w) |
         ((val3 rep(opcode rep w) = 5) →  STA(address rep w) |
          ((val3 rep(opcode rep w) = 6) →  RFI | NOP)))))))
```

Since the instruction IRQ is a pseudo machine instruction, our `select` function must also determine if there is an interrupt. If so, then the function must return something that tells the `instructions` function to return IRQ. The function `Opcode` specifies this behavior. `Opcode` returns a key of coproduct type `Instruction + one`. A left injection represents a normal user instruction. A right injection represents an interrupt.

```
Opcode =
⊢def ∀rep decrep s e.
    Opcode rep decrep s e =
    (let pc = mPc s
     and mem = mMem s
     and ireq = Ireq e
     and iack = mIack s
     in
      ((ireq ∧ ¬iack) →
       INR one |
       INL(decw rep(fetch rep(mem,address rep pc)))))
```

6.5 The `instructions` Function

The `instructions` function for the macro level takes a key of type `Instruction + one` . The reason for the injection is that our opcode is only represented by three bits (`*word3`), but we have nine possible instructions that could execute. The IRQ instruction is not a user instruction, but it is a valid machine instruction. Since we don't want to increase the size of the opcode, the left injection is used to get around this problem. The function `macro_instr` implements the `instructions` function for the macro level. If the key is a left injection, the function `M_INST` defined earlier is called with the correct instruction syntax. If the key is a right injection, the interrupt request function `M_IRQ` is returned.

```
macro_instr =
⊢def  macro_instr rep key = (ISL key →  M_INST rep(OUTL key) | M_IRQ rep)
```

6.6 The `output` Function

As before, the output function is trivially defined.

```
macro_output = ⊢def macro_output rep k s e = one
```

6.7 `subout`, `substate`, `subenv`

The environment does not change between the micro level and the macro level, so `subenv` is defined to be `I` .

`Subout` is also once again defined to be `I` .

As mentioned earlier, there is a significant amount of data abstraction between the micro level and macro level in the state of the microprocessor. This is modeled by the definition `micro_substate`.

```
micro_substate =
⊢def micro_substate(Micro_State pc acc mem ir rtn arg buf mar iack mpc) =
    State mem pc acc rtn iack
```

6.8 The Synchronization Predicate

The macroprogramming level is the first level in which we have temporal abstraction. Each macroinstruction is implemented by a number of microinstructions. This number ranges from three to eight. The first microinstruction executed for each macroinstruction is microinstruction zero (i.e. the `mpc` is equal to zero). The microprogram level and the macroprogram level are synchronized (i.e. their states are the same modulo abstraction) when the value of the `mpc` is equal to zero. Following this idea, we define our synchronization predicate for the macro level, **macro_sync**.

```
macro_sync = ⊢def macro_sycn rep s e = (uMpc s = word4 rep 0)
```

The definition uses the micro state selector function `uMpc` to check the value of the `mpc`.

6.9 Defining The Macro Level Interpreter

The macroprogramming level interpreter is defined and instantiated in the same manner as the previous interpreters. The representation, `GI_REP`, is given below.

```
GI_REP =
"GI
 (macro_instr rep)
 (Opcode rep decrep)
 (macro_output rep)
 micro_substate
 I
 I
 (micro_Interp rep)
 (macro_sync rep)"
```

The final instantiated theorem for the macro level interpreter is given below.

```
Macro_Interp =
⊢ macro_Interp rep decrep s e p =
    (∀t.
      let k = Opcode rep decrep(s t)(e t)
      in
       ((s(t + 1) = macro_instr rep k(s t)(e t)) ∧
        (p t = macro_output rep k(s t)(e t))))
```

7 Verifying an Interpreter

The generic interpreter theory is an abstract theory. To verify one of our interpreters (phase, micro, macro), we must prove the theory obligations for the abstract theory that defines the GIT. There are two theory obligations that we must prove in order to show that a lower level implements a higher level interpreter. These two obligations are: INST_CORRECT and OUTPUT_CORRECT [Win92b].

For this section, we will verify the theory obligations for the microprogram level interpreter. We start our proof by setting our goal. Our goal is to prove the theory obligations for our interpreter representation. The following HOL session shows how to do this.

```
g  "GI_THOBS ^GI_REP";;

"IS_GI
 (GI
  (micro_instr rep)
  (micro_select rep)
  (Micro_output rep)
  Phase_substate
  I
  I
  (phase_Interp rep)
  (micro_sync rep))"
    [ "Val4Word4_ASM rep" ]
    [ "Val3_CASES_ASM rep" ]
    [ "Word4_One2One_ASM rep" ]
    [ "Val4_CASES_ASM rep" ]
```

Notice that the theory obligations we defined in section 2.4 are automatically included on the assumption list.

Now we break apart the goal into the two respective subgoals.

```
OK..
2 subgoals
"!s' e' p' k.
  phase_Interp rep s' e' p' ==> (!t.
    let s t' = Phase_substate(s' t') in
     let e t' = I(e' t')  in
      let p t' = I(p' t') in
       let f t' = micro_sync rep(s' t')(e' t')  in
        ((micro_select rep(s t)(e t) = k) /\ f t ==> (p t = Micro_output rep k(s t)(e t))))"
    [ "Val4Word4_ASM rep" ]
    [ "Val3_CASES_ASM rep" ]
    [ "Word4_One2One_ASM rep" ]
    [ "Val4_CASES_ASM rep" ]

"!s' e' p' k.
  phase_Interp rep s' e' p' ==> (!t.
    let s t' = Phase_substate(s' t') in
     let e t' = I(e' t') in
      let f t' = micro_sync rep(s' t')(e' t') in
       ((micro_select rep(s t)(e t) = k) /\ f t ==>
        (?c.
          Next f(t,t + c) /\ (micro_instr rep k(s t)(e t) = s(t + c)))))"
    [ "Val4Word4_ASM rep" ]
    [ "Val3_CASES_ASM rep" ]
    [ "Word4_One2One_ASM rep" ]
    [ "Val4_CASES_ASM rep" ]
```

We now have two goals: one to prove theory obligation `OUTPUT_CORRECT` and one to prove the theory obligation `INST_CORRECT`.

The top goal is the one for `INST_CORRECT`. The object of the goal is to show that the phase level interpreter correctly implements each of the sixteen microinstructions. To prove this we will do a case analysis on the value of microprogram counter, `mpc`. The theorem `Val4_Values_LEMMA`, which follows naturally from the assumption `Val4_CASES_ASM`, is our basis for the case analysis.

```
Val4_Values_LEMMA=
⊢ Val4_CASES_ASM rep ⇒
   (val4 rep w = 15) ∨   (val4 rep w = 14) ∨  (val4 rep w = 13) ∨
   (val4 rep w = 12) ∨   (val4 rep w = 11) ∨  (val4 rep w = 10) ∨
   (val4 rep w = 9)  ∨   (val4 rep w = 8)  ∨  (val4 rep w = 7) ∨
   (val4 rep w = 6)  ∨   (val4 rep w = 5)  ∨  (val4 rep w = 4) ∨
   (val4 rep w = 3)  ∨   (val4 rep w = 2)  ∨  (val4 rep w = 1)  ∨   (val4 rep w = 0)
```

So our next step in the proof is to do some rewriting and add the conclusion of the theorem `Val4_Values_LEMMA` to our assumption list by using `IMP_RES_TAC`.

```
OK..
"(t < (t + 1) /\ (!t''. ~(t < t'' /\ t'' < (t + 1)))) /\
 (((val4 rep(Mpc(s' t)) = 0) =>  u0 rep |
  ((val4 rep(Mpc(s' t)) = 1) =>  u1 rep |
    ((val4 rep(Mpc(s' t)) = 2) =>  u2 rep |
     ((val4 rep(Mpc(s' t)) = 3) => u3 rep |
      ((val4 rep(Mpc(s' t)) = 4) => u4 rep |
       ((val4 rep(Mpc(s' t)) = 5) => u5 rep |
        ((val4 rep(Mpc(s' t)) = 6) => u6 rep |
         ((val4 rep(Mpc(s' t)) = 7) => u7 rep |
          ((val4 rep(Mpc(s' t)) = 8) =>  u8 rep |
           ((val4 rep(Mpc(s' t)) = 9) => u9 rep |
            ((val4 rep(Mpc(s' t)) = 10) => u10 rep |
             ((val4 rep(Mpc(s' t)) = 11) => u11 rep |
              ((val4 rep(Mpc(s' t)) = 12) => u12 rep |
               ((val4 rep(Mpc(s' t)) = 13) => u13 rep |
                ((val4 rep(Mpc(s' t)) = 14) => u14 rep | u15 rep)))))))))))))))
  (Micro_State (Pc(s' t)) (Acc(s' t)) (Mem(s' t)) (Ir(s' t)) (Rtn(s' t)) (Arg(s' t))
   (Buf(s' t))
   (Mar(s' t))
   (Iack(s' t))
   (Mpc(s' t)))
  (e' t) =
  Phase_substate
  (phase_level
   rep
   (EBM_State
    (Pc(s' t))
    (Acc(s' t))
    (Mem(s' t))
    (Ir(s' t))
    (Rtn(s' t))
    (Arg(s' t))
    (Buf(s' t))
    (Mar(s' t))
    (Iack(s' t))
    (Mpc(s' t)))
   (e' t)))"
    [ "Val4Word4_ASM rep" ]
    [ "Val3_CASES_ASM rep" ]
    [ "Word4_One2One_ASM rep" ]
    [ "Val4_CASES_ASM rep" ]
    [ "!t. s'(t + 1) = phase_level rep(s' t)(e' t)" ]
    [ "!w.
        (val4 rep w = 15) \/
        (val4 rep w = 14) \/
        (val4 rep w = 13) \/
        (val4 rep w = 12) \/
        (val4 rep w = 11) \/
        (val4 rep w = 10) \/
        (val4 rep w = 9) \/
        (val4 rep w = 8) \/
        (val4 rep w = 7) \/
        (val4 rep w = 6) \/
        (val4 rep w = 5) \/
        (val4 rep w = 4) \/
        (val4 rep w = 3) \/
        (val4 rep w = 2) \/
        (val4 rep w = 1) \/
        (val4 rep w = 0)" ]

() : void
```

Now we do a `STRUCT_CASES_TAC` on the top assumption which results in sixteen goals. Each of these goals is just a matter of rewriting with the appropriate microinstruction definition and simplifying.

Since the output of our interpreter is trivially defined, the second goal is simple. It is also just a matter of rewriting.

After proving these subgoals, we have the following theorem:

```
Micro_GI_THOBS_THM =
. ⊢ IS_GI
     (GI
      (micro_instr rep)
      (micro_select rep)
      (Micro_output rep)
      Phase_substate
      I
      I
      (phase_Interp rep)
      (micro_sync rep))
```

This theorem states that we have met the theory obligations for the micro level interpreter. Following this format, the theory obligations for the phase level and the macro level were also verified.

The theorem stating our phase level representation meets the theory obligations is given below. The proof of the phase level was very similar to that of the micro level; even though the phase level has only one instruction, it updates the state according to the value of the microprogram counter. So once again we did a case analysis on value of the `mpc`.

```
PHASE_GI_THOBS_THM =
.... ⊢ IS_GI
        (GI
         (phase_instr rep)
         (Phase_select rep)
         (phase_output rep)
         I
         I
         I
         (tamarackEBM rep)
         (phase_sync rep))
```

The macroprogramming level proof was the most difficult of the three. There were nine subgoals: one for each user instruction and one for the interrupt service instruction, `IRQ`. We list the theorem stating the theory obligations have been proven for the macro level below.

```
Macro_GI_THOBS_THM =
..... ⊢ ∀decrep.
          IS_GI
          (GI
           (macro_instr rep)
           (Opcode rep decrep)
           (macro_output rep)
           micro_substate
           I
           I
           (micro_Interp rep)
           (macro_sync rep))
```

8 The Final Correctness Statement

Now that we have proven the theory obligations for each interpreter in our hierarchy, our goal is to instantiate a final correctness theorem of the form:

$EBM \Longrightarrow$ Macro Level Interpreter

We have a lemma for each level stating that the theory obligations have been proven for that level. To reach our final goal, we need to combine these lemmas in a meaningful way. We do this by instantiating the abstract theorem `GI_VERT_COMPOSITION`.

```
GI_VERT_COMPOSITION =
.... ⊢ (INTERP gi1 = implementation gi2) ⇒
        IS_GI
        (GI
         (instructions gi2)
         (select gi2)
         (output gi2)
         ((substate gi2) o (substate gi1))
         ((subenv gi2) o (subenv gi1))
         ((subout gi2) o (subout gi1))
         (implementation gi1)
        ( λ s e. sync gi1 s e ∧ sync gi2(substate gi1 s)(subenv gi1 e)))
```

This theorem states that if we have two interpreters where one implements the other (such as the phase level interpreter implements the micro level interpreter) the composition of the two interpreters is an interpreter.

To instantiate this theorem for the micro and macro level interpreters use the ML function `instantiate_abstract_theorem` from the abstract theory package. This function takes as input two strings and two lists. The strings specify which theory and which theorem to instantiate. The first list specifies which interpreter representation to specialize `gi1` and `gi2` with in the abstract theorem. The second list contains the theorems showing that the theory obligations for the interpreters have been proven.

```
let MICRO_MACRO_CORRECT =
  instantiate_abstract_theorem `GI_def` `GI_VERT_COMPOSITION`
  [ "gi1:^GI_type'", MICRO_GI_REP;
    "gi2:^GI_type", MACRO_GI_REP; ]
  [ MICRO_GI_THOBS_THM;
    MACRO_GI_THOBS_THM;
  ];;
```

```
MICRO_MACRO_CORRECT =
...... ⊢ (INTERP
           (GI (micro_instr rep) (micro_select rep) (Micro_output rep) Phase_substate  I I
           (phase_Interp rep) (micro_sync rep)) = micro_Interp rep) ⇒
             (∀s' e' p' k.  INST_CORRECT
            (GI
            (macro_instr rep) (Opcode rep decrep) (macro_output rep)
            (micro_substate o Phase_substate) (I o I) (I o I)  (phase_Interp rep)
            (λ s e. micro_sync rep s e ∧ macro_sync rep(Phase_substate)(I e))) s' e' p' k)
          ∧
          (∀s' e' p' k. OUTPUT_CORRECT
           (GI  (macro_instr rep) (Opcode rep decrep) (macro_output rep)
            (micro_substate o Phase_substate) (I o I) (I o I) (phase_Interp rep)
            (λ s e. micro_sync rep s e ∧ macro_sync rep(Phase_substate s)(I e))) s' e' p' k)
```

The result has not been simplified. After some rewriting and conversions, we get the following theorem:

```
MICRO_MACRO_CORRECT_AUX=
...... ⊢ IS_GI
          (GI
           (macro_instr rep)
           (Opcode rep decrep)
           (macro_output rep)
           (micro_substate o Phase_substate)
           (I)
           (I)
           (phase_Interp rep)
           (λ s e. micro_sync rep s e ∧ macro_sync rep(Phase_substate s)(I e)))
```

This theorem states that the composition of the two interpreters is itself an interpreter. This interpreter describes the behavior of the macro programming level but is implemented by the phase level. Notice that the **substate**, **subenv**, and **subout** functions for the new interpreter are compositions of the two levels respective functions. The two synchronization predicates are conjoined showing that both predicates must hold for the macro level to be synchronized with the phase level.

To reach our final goal, we want an interpreter that defines the macro level but is implemented by the electronic block model. To achieve this, we simply repeat the process with the theorem we just generated and the phase level lemma for the theory obligations.

```
let CORRECT =
  instantiate_abstract_theorem `GI_def` `GI_VERT_COMPOSITION`
  [ "gi1:^GI_type'", PHASE_GI_REP;
    "gi2:^GI_type", (rand (concl th4)) ]
  [ PHASE_GI_THOBS_THM;
    MICRO_MACRO_CORRECT_AUX;
  ];;
```

Once again we do some rewriting to get the desired theorem.

```
whole_gi =
...... ⊢ IS_GI
          (GI
           (macro_instr rep)
           (Opcode rep decrep)
           (macro_output rep)
           (micro_substate o Phase_substate)
           I
           I
           (tamarackEBM rep)
           (λ s'' e''.
             phase_sync rep s'' e'' ∧
             micro_sync rep s'' e'' ∧
             macro_sync rep(Phase_substate s'')e''))
```

This theorem tells us that the composition of the previous interpreter and the phase level interpreter is another interpreter. This interpreter describes the macro level and is implemented by the electronic block model. Notice that the theorem is in the same format as that of the theorems showing the correctness of the theory obligations for a given level. Using this theorem we can now instantiate the final correctness theorem for Tamarack.

```
let TAMARACK_CORRECT_aux =
  instantiate_abstract_theorem `GI_def` `IMPL_IS_CORRECT`
  [ "gi:^GI_type", ( rand (concl whole_gi))]
  [ whole_gi];;




TAMARACK_CORRECT =
...... ⊢ ∀s' e' p'.
           let s t = (micro_substate o Phase_substate)(s' t)
           and e t = e' t
           and p t = p' t
           and f t =
                 phase_sync rep(s' t)(e' t) ∧
                 micro_sync rep(s' t)(e' t) ∧
                 macro_sync rep(Phase_substate(s' t))(e' t)
           in
            let abs = Temp_Abs f
            in
             (tamarackEBM rep s' e' p' ∧ (∃t. f t) ⇒
              macro_Interp rep decrep(s o abs)(e o abs)(p o abs))
```

The theorem `TAMARACK_CORRECT_aux` (not shown) is the actual instantiated correctness lemma. However, for cosmetic purposes, we do some rewriting and conversions to produce `TAMARACK_CORRECT`.

Notice the dots in the final theorem above. These dots are the assumptions that are necessary for this theorem to hold. These are the theory obligations we defined in section 2.4. There are two more assumptions in this list. In section 3.2, we stated that the verification would be done with the processor in fully synchronous mode. To put Tamarack in fully synchronous mode, the `dack` and the `idle` lines must be wired high. These two assumptions were added for the proof of the macro programming level.

9 Conclusion

We have presented the specification and verification of a simple microprocessor called Tamarack. The verification was done using the generic interpreter theory. The generic interpreter theory provided a structural methodology for the verification. The specification was completed by creating some behavioral definitions for each level. These definitions were then used as concrete functions to instantiate the abstract functions of the generic interpreter to create an interpreter model of the level.

The overall proof was broken down into three smaller proofs: verifying the phase level, micro level, and macro level. Each of these proofs was completed by proving that each level satisfied the theory obligations for an interpreter. After each level was verified, the results from those verifications were put together to create an overall correctness statement stating that the electronic block model correctly implements the assembly language level of the microprocessor. This method of partitioning the proof into a series of smaller proofs make the verification more tractable. By breaking the proof down into smaller proofs, we avoided the probable combinatorial explosion of cases in the proof that the hardware model correctly implements macroprogramming level.

Even though this was a simple example, the generic interpreter theory can be used to verify any microprocessor that can be specified as a hierarchy of interpreters. We are currently working on the verification of a RISC-like, pipelined microprocessor.

References

[Coh88] Avra Cohn. Correctness properties of the VIPER block model: The second level. Technical Report 134, University of Cambridge Computer Laboratory, May 1988.

[Gor83] Michael J.C. Gordon. Proving a Computer Correct using the LCF-LSM Hardware Verification System. Report No. 42, Computer Laboratory, Cambridge University, 1983.

[Gor88] Michael J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Press, 1988.

[Hun87] Warren A. Hunt. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. Elsevier Scientific Publishers, 1987.

[Joy86] J. Joyce, G. Birtwistle and M. Gordon. Proving a Computer Correct in Higher Order Logic. Report No. 100, Computer Laboratory, Cambridge University, 1986.

[Joy89] Jeffrey J. Joyce. *Multi–Level Verification of Microprocessor–Based Systems*. PhD thesis, Cambridge University, December 1989.

[Win90] Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, Division of Computer Science, June 1990.

[Win92a] Phillip J. Windley. Abstract theories in HOL. Technical Report LAL-92-07, University of Idaho, Laboratory for Applied Logic, June 1992.

[Win92b] Phillip J. Windley. A theory of generic interpreters. Technical Report LAL-92-06, University of Idaho, Laboratory for Applied Logic, April 1992.

A EBM Definitions

Here we present the Electronic Block Model definitions. All of these are the same as those of Joyce [Joy89].

```
SynMemory =
⊢def ∀rep w addr dataout mem datain.
    SynMemory rep(w,addr,dataout,mem,datain) =
    (∀t.
      (¬w t ⇒  (datain t = fetch rep(mem t,addr t))) ∧
      (mem(t + 1) = (w t →  store rep(mem t,addr t,dataout t) | mem t)))


CntlUnit =
⊢def ∀rep dack idle ireq iack zeroflag opc mpc cntls.
    CntlUnit rep(dack,idle,ireq,iack,zeroflag,opc,mpc,cntls) =
    (∃f0 f1 f2 f3 addr1 addr2 next rom.
      NextMPC
      rep
      (f0,f1,f2,f3,dack,idle,ireq,iack,zeroflag,opc,addr1,addr2,mpc,next) ∧
      MPC(next,mpc) ∧
      ROM rep(mpc,rom) ∧
      DecodeROM(rom,f0,f1,f2,f3,addr1,addr2,cntls))



NextMPC =
⊢def ∀rep f0 f1 f2 f3 dack idle ireq iack zeroflag opc addr1 addr2 mpc
    next.
    NextMPC
    rep
    (f0,f1,f2,f3,dack,idle,ireq,iack,zeroflag,opc,addr1,addr2,mpc,next) =
    (∀t.
      let waitcond =
            (f0 t,f1 t = waitdack) ∧ ¬dack t ∨
            (f0 t,f1 t = waitidle) ∧ ¬(idle t ∨ ¬dack t)
      in
       (next t =
        (waitcond →
         mpc t |
         ((f2 t,f3 t = jump) →
          addr1 t |
          ((f2 t,f3 t = jireq) →
           ((ireq t ∧ ¬iack t) →  addr1 t | addr2 t) |
           ((f2 t,f3 t = jzero) →
            (zeroflag t →  addr1 t | addr2 t) |
            word4 rep((val3 rep(opc t)) + (val4 rep(addr1 t))))))))
```

```
DecodeROM =
⊢def ∀rom f0 f1 f2 f3 addr1 addr2 cntls.
    DecodeROM(rom,f0,f1,f2,f3,addr1,addr2,cntls) =
    (∀t. cntls t,(f0 t,f1 t),(f2 t,f3 t),addr1 t,addr2 t = rom t)


ROM =
⊢def ∀rep inp out.
    ROM rep(inp,out) =
    (∀t. out t = CompileMicroCode rep(MicroCode(val4 rep(inp t))))


MPC = ⊢def ∀inp out. MPC(inp,out) = (∀t. out(t + 1) = inp t)


MicroCode =
⊢def  MicroCode n =
        ((n = 0) →  ((`rpc`,`wmar`,`none`),waitidle,jireq,1,2)             |
        ((n = 1) →  ((`rpc`,`wrtn`,`zero`),continue,jump,12,0)             |
        ((n = 2) →  ((`rmem`,`wir`,`none`),waitdack,jump,3,0)              |
        ((n = 3) →  ((`rir`,`wmar`,`none`),waitidle,jopcode,4,0)           |
        ((n = 4) →  ((`none`,`none`,`none`),continue,jzero,5,11)           |
        ((n = 5) →  ((`rir`,`wpc`,`none`),continue,jump,0,0)               |
        ((n = 6) →  ((`racc`,`warg`,`none`),waitidle,jump,13,0)            |
        ((n = 7) →  ((`racc`,`warg`,`none`),waitidle,jump,14,0)            |
        ((n = 8) →  ((`rmem`,`wacc`,`none`),waitdack,jump,11,0)            |
        ((n = 9) →  ((`racc`,`wmem`,`none`),waitdack,jump,11,0)            |
        ((n = 10) →  ((`rrtn`,`wpc`,`none`),continue,jump,0,0)             |
        ((n = 11) →  ((`rpc`,`none`,`inc`),continue,jump,12,0)             |
        ((n = 12) →  ((`rbuf`,`wpc`,`none`),continue,jump,0,0)             |
        ((n = 13) →  ((`rmem`,`none`,`add`),waitdack,jump,15,0)            |
        ((n = 14) →  ((`rmem`,`none`,`sub`),waitdack,jump,15,0)            |
        ((`rbuf`,`wacc`,`none`),continue,jump,11,0)))))))))))))))))


CompileMicroCode =
⊢def ∀rep cntl wcode jcode n m.
    CompileMicroCode rep(cntl,wcode,jcode,n,m) =
    Cntls cntl,wcode,jcode,word4 rep n,word4 rep m


Cntls =
⊢def ∀tok1 tok2 tok3.
    Cntls(tok1,tok2,tok3) =
    (tok2 = `wmem`),(tok1 = `rmem`),(tok2 = `wmar`),(tok2 = `wpc`),
    (tok1 = `rpc`),(tok2 = `wacc`),(tok1 = `racc`),(tok2 = `wir`),
    (tok1 = `rir`),(tok2 = `wrtn`),(tok1 = `rrtn`),(tok2 = `warg`),
    ((tok3 = `inc`) ∨ (tok3 = `add`)),
    ((tok3 = `inc`) ∨ (tok3 = `sub`)),(tok1 = `rbuf`)
```

```
jopcode = ⊢def jopcode = F,F


jzero = ⊢def jzero = F,T


jireq = ⊢def jireq = T,F


jump = ⊢def jump = T,T


continue = ⊢def continue = F,F


waitidle = ⊢def waitidle = T,F


waitdack = ⊢def waitdack = T,T


DataPath =
⊢def ∀rep cntls datain mar pc acc ir rtn iack arg buf dataout wmem dreq
    addr zeroflag opc.
    DataPath
    rep
    (cntls,datain,mar,pc,acc,ir,rtn,iack,arg,buf,dataout,wmem,dreq,addr,
     zeroflag,opc) =
    (∃bus busokay alu pwr gnd rmem wmar wpc rpc wacc racc wir rir wrtn
      rrtn warg alu0 alu1 rbuf.
      DecodeCntls
      (cntls,wmem,rmem,wmar,wpc,rpc,wacc,racc,wir,rir,wrtn,rrtn,warg,
       alu0,alu1,rbuf) ∧
      BusOkay(rmem,rpc,racc,rir,rrtn,rbuf,busokay) ∧
      Interface rep(busokay,wmem,rmem,bus,datain,dataout) ∧
      OR(wmem,rmem,dreq) ∧
      Register(busokay,wmar,gnd,bus,bus,mar) ∧
      AddrField rep(mar,addr) ∧
      Register(busokay,wpc,rpc,bus,bus,pc) ∧
      Register(busokay,wacc,racc,bus,bus,acc) ∧
      TestZero rep(acc,zeroflag) ∧
      Register(busokay,wir,rir,bus,bus,ir) ∧
      OpcField rep(ir,opc) ∧
      Register(busokay,wrtn,rrtn,bus,bus,rtn) ∧
      JKFF(wrtn,rrtn,iack) ∧
      Register(busokay,warg,gnd,bus,bus,arg) ∧
      ALU rep(alu0,alu1,arg,bus,alu) ∧
      Register(busokay,pwr,rbuf,alu,bus,buf) ∧
      PWR pwr ∧
      GND gnd)
```

```
DecodeCntls =
⊢def ∀cntls wmem rmem wmar wpc rpc wacc racc wir rir wrtn rrtn warg alu0
    alu1 rbuf.
    DecodeCntls
    (cntls,wmem,rmem,wmar,wpc,rpc,wacc,racc,wir,rir,wrtn,rrtn,warg,alu0,
     alu1,rbuf) =
    (∀t.
      wmem t,rmem t,wmar t,wpc t,rpc t,wacc t,racc t,wir t,rir t,wrtn t,
      rrtn t,warg t,alu0 t,alu1 t,rbuf t =
      cntls t)


Register =
⊢def ∀busokay w r inp bus out.
    Register(busokay,w,r,inp,bus,out) =
    (∀t.
      (busokay t ⇒  r t ⇒  (bus t = out t)) ∧
      (out(t + 1) = (w t →  inp t | out t)))



Interface =
⊢def ∀rep busokay w r bus datain dataout.
    Interface rep(busokay,w,r,bus,datain,dataout) =
    (∀t.
      (busokay t ⇒  r t ⇒  (bus t = datain t)) ∧
      (dataout t = (w t →  bus t | wordn rep 0)))



BusOkay =
⊢def ∀rmem rpc racc rir rrtn rbuf busokay.
    BusOkay(rmem,rpc,racc,rir,rrtn,rbuf,busokay) =
    (∀t.
      busokay t =
      (rmem t →
       ¬(rpc t ∨ racc t ∨ rir t ∨ rrtn t ∨ rbuf t) |
       (rpc t →
        ¬(racc t ∨ rir t ∨ rrtn t ∨ rbuf t) |
        (racc t →
         ¬(rir t ∨ rrtn t ∨ rbuf t) |
         (rir t →  ¬(rrtn t ∨ rbuf t) | (rrtn t →  ¬rbuf t | T))))))
```

```
TestZero =
⊢def ∀rep inp out. TestZero rep(inp,out) = (∀t. out t = iszero rep(inp t))



AddrField =
⊢def ∀rep inp out.
    AddrField rep(inp,out) = (∀t. out t = address rep(inp t))



OpcField =
⊢def ∀rep inp out. OpcField rep(inp,out) = (∀t. out t = opcode rep(inp t))



ALU =
⊢def ∀rep f0 f1 inp1 inp2 out.
    ALU rep(f0,f1,inp1,inp2,out) =
    (∀t.
      out t =
      ((f0 t,f1 t = T,T) →
       inc rep(inp2 t) |
       ((f0 t,f1 t = T,F) →
        add rep(inp1 t,inp2 t) |
        ((f0 t,f1 t = F,T) →  sub rep(inp1 t,inp2 t) | wordn rep 0))))



JKFF =
⊢def ∀j k out.
    JKFF(j,k,out) = (∀t. out(t + 1) = j t ∧ ¬out t ∨ ¬k t ∧ out t)



OR =
⊢def ∀inp1 inp2 out. OR(inp1,inp2,out) = (∀t. out t = inp1 t ∨ inp2 t)



PWR = ⊢def ∀out. PWR out = (∀t. out t = T)



GND = ⊢def ∀out. GND out = (∀t. out t = F)
```

B The Microinstructions

In this section, we describe in more detail the microinstructions for Tamarack. There are sixteen microinstructions. For each of them, we will list the HOL source definition and give a brief description of the code.

Microinstruction 0

Microinstruction 0 serves two purposes. First, the instruction fetch is initiated. Second, `u0` determines if an interrupt has occurred. The value of the program counter is loaded into the memory address register to begin the instruction fetch. A value of zero is loaded into the buffer register since no ALU operation has taken place. Finally, the microprogram counter for the next microinstruction is computed. If Tamarack is waiting for data from memory, then the `mpc` does not change. If Tamarack is not idle, we check for an interrupt. If the `ireq` line is high and the `iack` is low, then an interrupt has occurred and it has not been serviced. If we have an interrupt, the `mpc` is set to one. If Tamarack is not idle and an interrupt has not occurred, then the `mpc` is set to two.

```
⊢def ∀rep s e.
    u0 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
      let new_mpc =
            ((¬(idle ∨ ¬dack)) →
             mpc |
             ((ireq ∧ ¬iack) →  word4 rep 1 | word4 rep 2)) in
       let new_mar = pc in
         let new_buf = wordn rep 0 in
      Micro_State pc acc mem ir rtn arg new_buf new_mar iack new_mpc)
```

Microinstruction 1

Microinstruction 1 prepares the microprocessor for servicing an interrupt. First the return address register is loaded with the current value of the program counter so we may resume processing after the interrupt is serviced. Next, the buffer register is loaded with the value of zero. This value will be loaded into the program counter in microinstruction 12. This is the address where the interrupt service routine is located. Next, the interrupt acknowledgment flag is set to true denoting that we are currently servicing an interrupt. Finally, the `mpc` is set to twelve.

```
⊢def ∀rep s e.
    u1 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e  in
      let new_rtn = pc  in
       let new_buf = wordn rep 0 in
        let new_iack = T in
         let new_mpc = word4 rep 12 in
       Micro_State pc acc mem ir new_rtn arg new_buf mar new_iack new_mpc)
```

Microinstruction 2

Microinstruction 2 continues the instruction fetch process. First, the instruction register is loaded with the contents of memory at the location specified by the memory address register. If Tamarack has to wait for the data to arrive from memory, then the new `mpc` is set to two once again. Otherwise, the `mpc` is set to three. The buffer register is loaded with zero once again since no ALU operation has taken place. This will occur in every microinstruction where there is ALU operation.

```
⊢def ∀rep s e.
    u2 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
      let new_ir = fetch rep(mem,address rep mar) in
       let new_mpc = ((¬dack) → mpc | word4 rep 3) in
        let new_buf = wordn rep 0 in
         Micro_State pc acc mem new_ir rtn arg new_buf mar iack new_mpc)
```

Microinstruction 3

Microinstruction 3 loads the source operand into the memory address register. Then `u3` sets the `mpc` to the opcode of the instruction in the instruction register, `ir`, plus four. This will select one of the microinstructions between four and ten for the next one to execute. The reason the four is added is because we have completed three states in our finite state machine and four is the offset to add to the opcode to select the next state. For example, suppose our instruction is a JZR. The opcode for JZR is 0. `u3` will set the next `mpc` to $0 + 4 = 4$, and microinstruction four is the microinstruction that starts the actions for a JZR instruction. To see this more clearly, please refer to the state machine for Tamarack in [Joy89, page 25].

```
⊢def ∀rep s e.
    u3 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e  in
       let new_mar = ir in
        let new_mpc =
             ((¬(idle ∨ ¬dack)) →
              mpc |
              word4 rep((val3 rep(opcode rep ir)) + 4)) in
         let new_buf = wordn rep 0 in
        Micro_State pc acc mem ir rtn arg new_buf new_mar iack new_mpc)
```

Microinstruction 4

Microinstruction 4 implements the macroinstuction JZR. `u4` selects the next microinstruction according to the value in the accumulator. If the value in the accumulator is equal to zero, then the `mpc` is set to five which implements a jump. If not, then the `mpc` is set to eleven and no jump will occur.

```
⊢def ∀rep s e.
    u4 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
      let new_mpc = (iszero rep acc →  word4 rep 5 | word4 rep 11) in
       let new_buf = wordn rep 0 in
        Micro_State pc acc mem ir rtn arg new_buf mar iack new_mpc)
```

Microinstruction 5

Microinstruction 5 implements the JMP instruction. This is done by setting the new program counter equal to the word in the instruction register (remember that the program counter is a full word register). The `mpc` is reset to zero to start executing the next macroinstruction.

```
⊢def ∀rep s e.
    u5 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e  in
      let new_pc = ir in
       let new_mpc = word4 rep 0 in
        let new_buf = wordn rep 0 in
         Micro_State new_pc acc mem ir rtn arg new_buf mar iack new_mpc)
```

Microinstruction 6

Microinstruction 6 begins the execution of an ADD instruction. The value in the accumulator is loaded into argument buffer so that it may be used by the ALU later. The `mpc` is then set to thirteen if Tamarack is not idle.

```
⊢def ∀rep s e.
    u6 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
      let new_arg = acc in
       let new_mpc = ((¬(idle ∨ ¬dack)) →  mpc | word4 rep 13) in
        let new_buf = wordn rep 0 in
         Micro_State pc acc mem ir rtn new_arg new_buf mar iack new_mpc)
```

Microinstruction 7

Microinstruction 7 does the same thing as `u6`. The value in the accumulator is loaded into the argument buffer for use by the ALU (but this time for a SUB instruction).

```
⊢def ∀rep s e.
    u7 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
      let new_arg = acc in
       let new_mpc = ((¬(idle ∨ ¬dack)) →  mpc | word4 rep 14) in
        let new_buf = wordn rep 0 in
         Micro_State pc acc mem ir rtn new_arg new_buf mar iack new_mpc)
```

Microinstruction 8

Microinstruction 8 implements the LDA instruction. The accumulator is loaded with the word in memory located at the address in the memory address register. The `mpc` will remain at eight until Tamarack receives the data (when the `dack` is true). After Tamarack receives the word from memory, the `mpc` is set to eleven.

```
⊢def ∀rep s e.
    u8 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e  in
      let new_acc = fetch rep(mem,address rep mar) in
       let new_mpc = ((¬dack) →  mpc | word4 rep 11) in
        let new_buf = wordn rep 0 in
         Micro_State pc new_acc mem ir rtn arg new_buf mar iack new_mpc)
```

Microinstruction 9

Microinstruction 9 implements the actions of the STA instruction. STA stores the value of the accumulator to a specified location in memory. `u9` stores the contents of the accumulator to memory at the address in the memory address register.

```
⊢def ∀rep s e.
    u9 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e  in
      let new_mem = store rep(mem,address rep mar,acc) in
       let new_mpc = ((¬dack) →  mpc | word4 rep 11) in
        let new_buf = wordn rep 0 in
         Micro_State pc acc new_mem ir rtn arg new_buf mar iack new_mpc)
```

Microinstruction 10

Microinstruction 10 implements the RFI (return from interrupt) instruction. The program counter is loaded with the value in the return register. The interrupt acknowledgment flag is set to false denoting that we are not currently processing an interrupt. The `mpc` is then set to zero to begin processing the next macroinstruction.

```
⊢def ∀rep s e.
    u10 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
      let new_pc = rtn in
       let new_iack = F in
         let new_mpc = word4 rep 0 in
          let new_buf = wordn rep 0 in
       Micro_State new_pc acc mem ir rtn arg new_buf mar new_iack new_mpc)
```

Microinstruction 11

Microinstruction 11 computes the value of the new program counter (if no jump is taken). The value in program counter is simply incremented and stored in the buffer register.

```
⊢def ∀rep s e.
    u11 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
      let new_buf = inc rep pc in
       let new_mpc = word4 rep 12 in
        Micro_State pc acc mem ir rtn arg new_buf mar iack new_mpc)
```

Microinstruction 12

Microinstruction 12 assigns to the program counter the value calculated in `u11`. The value in the buffer register is loaded into the program counter, and the `mpc` is set to zero to start processing the next macroinstruction.

```
⊢def ∀rep s e.
    u12 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e  in
      let new_pc = buf in
       let new_mpc = word4 rep 0 in
         let new_buf = wordn rep 0  in
          Micro_State new_pc acc mem ir rtn arg new_buf mar iack new_mpc)
```

Microinstruction 13

Microinstruction 13 continues the ADD instruction started in `u6`. The buffer register receives the value of adding the contents in the argument buffer (the value in the accumulator) and the value in memory at the location specified by the memory address register.

```
⊢def ∀rep s e.
    u13 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
      let new_buf = add rep(arg,fetch rep(mem,address rep mar)) in
        let new_mpc = ((¬dack) → mpc | word4 rep 15) in
       Micro_State pc acc mem ir rtn arg new_buf mar iack new_mpc)
```

Microinstruction 14

Microinstruction 14 continues the SUB instruction initiated by `u7`. The buffer register is assigned the value of subtracting the the word in memory, specified by the memory address register, from the contents of the argument buffer. The `mpc` is then set to fifteen to complete the SUB instruction.

```
⊢def ∀rep s e.
    u14 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e  in
      let new_buf = sub rep(arg,fetch rep(mem,address rep mar)) in
        let new_mpc = ((¬dack) →  mpc | word4 rep 15) in
        Micro_State pc acc mem ir rtn arg new_buf mar iack new_mpc)
```

Microinstruction 15

Microinstruction 15 completes both the ADD and SUB instructions. The contents of the buffer register (the result of either the add or sub) are copied into the accumulator. The `mpc` is then set to eleven to compute the new program counter.

```
⊢def ∀rep s e.
    u15 rep s e =
    (let pc = uPc s
     and acc = uAcc s
     and mem = uMem s
     and ir = uIr s
     and rtn = uRtn s
     and arg = uArg s
     and buf = uBuf s
     and mar = uMar s
     and iack = uIack s
     and mpc = uMpc s
     and ireq = Ireq e
     and idle = Idle e
     and dack = Dack e
     and wmem = Wmem e
     and dreq = Dreq e
     and dataout = Dataout e
     and datain = Datain e in
        let new_acc = buf in
         let new_mpc = word4 rep 11 in
            let new_buf = wordn rep 0  in
         Micro_State pc new_acc mem ir rtn arg new_buf mar iack new_mpc)
```

C The Macroinstructions

Here we present the definitions for the macroinstructions. These are almost identical to Joyce's [Joy89]. The only differences are the addition of the state and environment types, and because of the Instruction data type we defined, some of the definitions were changed a little to show the semantics of the instruction. For example, The ADD instruction now takes an extra argument of the type `*address` which is the memory address for the value to add to the accumulator.

JZR

The JZR instruction performs a jump if the value in the accumulator is equal to zero. To perform this operation the memory contents at the location specified by the program counter are loaded and assigned to the variable `inst`. If the value in the accumulator is zero, then the program counter is assigned the value of `inst`, otherwise the program counter is simply incremented.

```
M_JZR =
⊢def ∀rep s e.
    M_JZR rep s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      let inst = fetch rep(mem,address rep pc)
      in
       let nextpc = (iszero rep acc →  inst | inc rep pc)
       in
        State mem nextpc acc rtn iack)
```

JMP

The JMP performs an unconditional jump. The instruction word `inst` is loaded into the program counter unconditionally.

```
M_JMP =
⊢def ∀rep s e.
    M_JMP rep s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      let inst = fetch rep(mem,address rep pc)
      in
       State mem inst acc rtn iack)
```

ADD

The ADD instruction adds a value located in memory to the accumulator. The M_ADD definition takes the argument `adr` with the state and environment. `adr` is the address in memory that holds the value to add to the accumulator. This `adr` is the main difference between Joyce's definitions and the next four definitions listed here. The `adr` effectively eliminates the first fetch instruction in the definition of ADD [Joy89, page 50] since `adr` is already the address representation of the program counter passed in by the definitions `M_INST` and `Opcode`. To implement the ADD, the operand is loaded from memory at the location specified by `adr`. Next, the program counter is incremented. Then, the accumulator is updated with the result of adding the operand and the current value of the accumulator. Finally, the new state is created and returned.

```
M_ADD =
⊢def ∀rep adr s e.
    M_ADD rep adr s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      let operand = fetch rep(mem,adr)
      in
       let new_pc = inc rep pc
       in
        let new_acc = add rep(acc,operand)
        in
         State mem new_pc new_acc rtn iack)
```

SUB

The SUB instruction subtracts a value located in memory from the accumulator. It executes in the same manner as the ADD instruction except that the accumulator is updated with subtract operation instead of an add.

```
M_SUB =
⊢def ∀rep adr s e.
    M_SUB rep adr s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      let operand = fetch rep(mem,adr)
      in
       let new_pc = inc rep pc
       in
        let new_acc = sub rep(acc,operand)
        in
         State mem new_pc new_acc rtn iack)
```

LDA

The LDA instruction loads a value from memory into the accumulator. The operand to load is fetched from memory at the location specified by `adr`. The program counter is incremented, and the new state is created with the accumulator now holding the value fetched from memory.

```
M_LDA =
⊢def ∀rep adr s e.
    M_LDA rep adr s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      let operand = fetch rep(mem,adr)
      in
       let new_pc = inc rep pc in State mem new_pc operand rtn iack)
```

STA

The STA instruction stores the value in the accumulator to memory. The memory is updated by storing the value of the accumulator to memory at the address `adr`.

```
M_STA =
⊢def ∀rep adr s e.
    M_STA rep adr s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      let newmem = store rep(mem,adr,acc)
      in
       let new_pc = inc rep pc in State newmem new_pc acc rtn iack)
```

RFI

The RFI instruction updates the state of the microprocessor after an interrupt has been serviced. The program counter is assigned the value that is in the return register, and the interrupt acknowledgment flag is set to false.

```
M_RFI =
⊢def ∀rep s e.
    M_RFI rep s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      State mem rtn acc rtn F)
```

NOP

The NOP instruction makes no changes to the state except to increment the program counter.

```
M_NOP =
⊢def ∀rep s e.
    M_NOP rep s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      let new_pc = inc rep pc in State mem new_pc acc rtn iack)
```

IRQ

The IRQ instruction initiates the interrupt service routine. The program counter is set to zero and the interrupt acknowledgment flag is set to true.

```
M_IRQ =
⊢def ∀rep s e.
    M_IRQ rep s e =
    (let mem = mMem s
     and pc = mPc s
     and acc = mAcc s
     and rtn = mRtn s
     and iack = mIack s
     in
      let new_pc = wordn rep 0 in State mem new_pc acc pc T)
```