Functional Semantics of Microprocessors at the Microprogram
Level and Correspondence with the Machine Instruction Level

Hélène COLLAVIZZA
Université de Provence
Case S, 3 Place Victor Hugo
13331 Marseille Cedex 3, FRANCE

Abstract
In order to apply formal verification methods to actual microprocessors, we study the functional semantics of microprocessors of the "Von Neumann Machine" type, at different specification and description levels.

We settle a functional formalism from which the semantic equivalence between adjacent levels will be proved. Here, we propose a general methodology to define the functional semantics of a processor at the Microprogram level. We also present the correspondence between this level and the Machine Instruction level.

1. Introduction
1.1. Motivations
In the last years, many works have been done to apply formal methods to significant microprocessor examples: Gordon [10] with LCF-LSM and then Joyce [15] with HOL proved a simple computer; Cohn [6] modelled and proved the VIPER microprocessor using the HOL system; Hunt verified the FM8501 using the Boyer & Moore theorem prover [14]. For each of these works, a particular formalism or proof methodology is applied to one microprocessor example. Yet, these appear to be verification oriented examples.

We are aiming at a general methodology [2], in order to formalize a large class of microprocessors, starting from their description, as it is given by the designers. The final objective is the automatic building of such formalizations, according to a model suitable to formal verification.

The study presented here concerns the semantics of the microprogram level "level_2", and its correspondence with "level_1", the machine instruction level. Level_2 performs the exchanges between the processor and the main memory. It is not always explicitly specified by the designers but its insertion in the level hierarchy simplifies the proof between "level_1" and "level_3", the micro-instruction level. Indeed, some microprograms are shared by most of the instructions. So, we prove their implementations only once between levels 2 and 3, and verify their compositions between levels 1 and 2. Furthermore, level_2 allows us to smoothly dissociate the main memory from the central processing unit: at level_3, we don't consider the main memory any more, only the internal architecture.

We first present generalities on the functional semantics of microprocessors and then detail the modelling and the correspondences between level_1 and level_2 on a set of main points: the system state, the semantics of the operands and the semantics of the instructions. We also emphasize that our formalism is appropriate for the characteristics of pipelined micro-processors. A detailed formalization of level_1 can be found in [16], a complete formulation for the MTI and examples derived from commercial processors in [17] for level_1 and in [7] for level_2.

1.2. Notations:
- Functional notations:
  - we used lambda expressions as in lambda calculus [5].

The functions are typed as in the ML language [9].

- \lambda \langle x_1, \ldots, x_n \rangle. f indicates that the \lambda symbol covers a n-uple of variables.

- we use standard functional programming constructors [12], such as: let ... for local definitions, the conditional form if ... then ... else ... the selection form case ... is ... - we define a functional form "SUBST" to take into account the assignment of value. It transforms an argument function into a resulting function equal to its argument except on one point where there is a substitution:
  \text{SUBST} : (\text{id} \mapsto \text{val} \mapsto f) \rightarrow (\text{id} \mapsto \text{val} \mapsto f)

- We use the generic list structure for representing tuples denoted \text{<e_1, e_2, \ldots, e_n>}. Standard functions on lists are employed, such as \text{NTH} for accessing elements of a tuple. To make expressions clearer, occurrences of the function NTH are noted FIRST, SECOND, or are noted by particular identifiers referring to the objects of interest, as specialization of polymorphic functions [9].

- To avoid the use of bit-vector/integer conversion functions, we consider that all the objects handled by the processor are bit-vectors, on which arithmetic operations are induced. We denote \text{Num}_n the set of n-bit words.

2. Functional semantics of processors: generalities

The semantics which we define for a processor is inspired from the functional semantics of programming languages [20]. At each level of abstraction, we define the state of the machine constituted by the processor and its addressable memory, and we express each elementary behaviour by means of a function which transforms the state. In general, when going from an upper level to a lower level, more state variables are introduced. At level_1, the specification is the set of the instructions as the programmer sees it. A function "SEM" associates to each instruction a state change function [18]. At level_2, an instruction is implemented by a sequence of microprograms defined as state transition functions. Each microprogram corresponds to a memory-processor information exchange or to a set of internal operations indivisible at level_1. We define a function "implppgm" which associates to each instruction its implementation. Furthermore, we must define all the microprograms used to implement the instructions.

The set of microprograms is divided into two classes: microprograms which perform internal operations and read/write microprograms which modify the memory access state variables. The read/write operations are decomposed into a sequence of microprograms according to the read/write flowchart given by the designers. We assume that the main memory behaves as expected by the processor: at the end of a correct read microprogram sequence (i.e. if the processor has sent right values to the memory access pins), the data port receives the value read, and for a write access, the memory receives the value to be written.

We group into parameterized microprograms those which
modify the same state variables but assign different values according to the instruction mnemonics, data size or addressing mode classes. This notion corresponds to a real preoccupation of the designers [1] and furthermore, simplifies the verification process. In fact, composing the microprograms between level_1 and level_2, we keep the formal parameters. On the other hand, between level_2 and level_3, we prove for each microprogram instantiated with an actual parameter, that it is correctly realized by a microinstruction sequence. So, between levels 1 and 3, we cover the whole spanning tree of possible state transitions, thanks to the microprogram composition, but we prove only once a same microprogram which belongs to different paths.

3. Microprocessor state definition:

3.1. State definition at level_1.

At any moment, the state is defined as the content of all the storage elements available to the programmer. Since these elements play differentiated parts, the state is structured into three important storage location categories: the main memory, one or several register banks and specialized registers, among these, at least the Program Counter and the Status Register. Memories and register banks are defined as functions which associate to a memory address a value. Each specialized register is represented by a 0-ary function. These functions are named from the considered state component: for example, MEM for the main memory, SR for the Status Register. The current state of the system is represented by:

\[ \text{STATE}_1 = \langle \text{MEM}, \text{REG}, \text{PC}, \text{SR}, \ldots \rangle \]

3.2 State definition at the microprogram level:

The state includes the set of level_1 state variables plus the set of the main memory access pins and the set of instruction registers. These variables have different kinds: memorizing elements and input/output memory signals. In fact, this distinction between storage elements and signals is modeled only at level_3. Indeed for a microinstruction, the value of a storage element is expressed in terms of the past value of the state while a signal value can depend on current values of others signals. On the other hand, at level_2, the whole set of buffers and signals is not considered; in particular, connections between internal busses and signals or input/output pins are not visible. So there is no contemporary relations between variables and we can consider all access pins as state variables.

Generally, the access pins of the main memory can be represented by the following state variables: ADDRESS (memory access address), DATA (current value read or to be written in the memory), VMA (Valid Memory Address signal) and READ (read/write signal). There may be other memory signals which define for example, the data length (as in the Motorola 68000) or a DTACK "data acknowledge" signal for asynchronous read/write operations when the memory accesses are modeled as handshake sequences.

If the instruction format is not uniform, we include as state variables the maximum set of words used to code an instruction: the Instruction register IR which almost defines the mnemonics, and some virtual variables W1, W2... used to buffer operand specification words (at level_3, these variables correspond to actual registers whose number is at most the number of virtual variables). The notion of virtual variable is analogous to the notion of "integer execution unit".

The current state of the system can be represented by:

\[ \text{STATE}_2 = \langle \text{MEM}, \text{REG}, \text{PC}, \text{SR}, \ldots, \text{IR}, \text{W1}, \text{W2}, \ldots, \text{ADDRESS}, \text{DATA}, \text{VMA}, \text{READ}, \ldots \rangle \]

3.3. Example of a processor with a single instruction format: The MTI processor.

In order to highlight the model being used, we consider the MTI processor, developed at CNET, Meylan (FRANCE) [19], whose instructions have a single format, coded with two 16-bit words. It has an instruction set including 22 basic arithmetic, logical, shift, branch, stack and subprogram instructions. The internal registers available to the programmer are: a general purpose 16-bit register bank REGI, a special purpose 32-bit register bank DEP, a special purpose 16-bit register bank BASEI, a Program Counter and a Status Register. Memory words are referenced by 22-bit based addresses.

The MTI state at level_1:

\[ \text{STATE}_1 = \langle \text{MEM, REG, DEP, BASE, PC, SR} \rangle \]

where MEM: Num22->Num16 is the main memory, REG: Num6->Num16 represents 32 16-bit registers, DEP: (mENum4 / s[n][j][1])->Num32, is a bank of 32 32-bit "program status double words", BASE: Num3->Num16 represents 8 16-bit "bases" used for based addressing, PC and SR are 0-ary functions: a->Num16

We define functions for selecting the elements of STATE1: mem, function of type STATE1 -> (Num22 -> Num16), which selects MEM, and likewise reg, dep, b, pc, sr...

The MTI state at level_2:

\[ \text{STATE}_2 = \langle \text{MEM, REG, DEP, BASE, PC, SR, IR, W, ADDRESS, DATA, VMA, WRITE} \rangle \]

where IR: a->Num_1 is the instruction register, W: a->Num_1 is used for the second instruction word, DATA: a->Num_1 is multiplexed and contains either the data or the 16 lower bits of the address, ADDRESS: a->Num_1 contains the 6 higher bits of the address, VMA and WRITE of type a->Num_1 have their usual meaning.

4. Semantics of the operands:

4.1. Instruction syntax:

To represent an instruction, we use the same formalism at the two specification levels. Any machine instruction is decomposed into a number of fields among which one is the mnemonic "MNEM", which symbolizes the kind of instruction, and the others are constituted by a set of parameters "PAR" used to specify the operands.

4.2. Semantics of the operand at level_1:

In order to specify the meaning of the operands, depending on the different instruction fields, we use the "environment" function. This function expresses the semantic interpretation of instruction fields, in the form of one or several operands. An operand can have different types: it can be a digital constant, a digital function of the state, or a location [18]. A location is defined as an ordered pair LOC = <SEL, PL> where SEL is a function selecting a memorizing element, and PL is a digital function of the state used to calculate a pin-point place in the memorizing element. If a location LOC is used as source, its contents is formulated by:

\[ \text{CONT}(\text{LOC}) = \text{SEL}(\text{STATE}) \langle \text{PL}(\text{STATE}) \rangle \]

On the other hand, if LOC is used as destination for a value val, the assignment is expressed by:

\[ \text{SUBST} (\text{SEL}(\text{STATE}), \text{PL}(\text{STATE}), \text{val}) \]

If the number and meaning of parameters change from one instruction to another, the set of instructions is partitioned into mnemonic classes, so that all the instructions in the same class can be expressed by the same abstract syntax. We associate a particular "environment" function to each one of these classes.
Example: MTI instruction syntax and environment function.

- Instruction syntax:
  A MTI instruction is represented by `< MNEM,MODE,R,X,WORD >` where “MNEM” takes value in a set of 17 identifiers for the operation code, “MODE” in a set of 5 identifiers for the addressing mode, R ∈ Numbers is used to define an operand (one of the registers REG), X ∈ Numbers and WORD ∈ Numbers are used with ‘MODE’, to define another operand.

Field selection functions are named: s_mnem, s_mode, ...

- Environment function:
  The MTI environment function, noted “ENV”, associates to each instruction a pair of operands, noted < OP1, OP2 >. OP1 associates either a constant, for branch or interrupt request, or a REG location. OP2 is either the immediate operand or a memory location < REGi, x, word > where AA receives the actual address from the mode and the offset fields of an instruction.

4.3. Semantics of the operands at level-2:

At the microprogram level, we don’t wholly use the Environment function to settle the operands. If the operand is a register location or a digital function of the state, it is computed in the same way as defined by the Environment function. In the other case, it is in the main memory and its computation is decomposed in several read microprograms, depending on the addressing mode. The link between the values computed at the two levels is the “Actual Address” function. If the value given by the Environment function is the location <MEM, AA(inst)> then at level-2, the operand is settled by a read microprogram sequence in which ADDRESS receives AA(inst).

Example: computation of the MTI operands at the two levels

For this processor, the first operand is located in the REG register bank, while the second is in the main memory. So, we compute the first operand in the same way at the two levels. On the other hand, the second operand is computed at level-1 by means of the Environment function while it is computed at level-2 by a read microprogram sequence. We give below the computation of the second operand for a calculation instruction in a direct addressing mode.

Let “inst” be a calculation instruction, mod=s_mode(inst), x=s_x(inst) and word=s_word(inst); then:

- At the machine instruction level, the operand value is defined by the contents of the second operand:
  \[ \text{op2_val}_{\text{level1}} = \text{SECOND}(<\text{ENV}(\text{inst})>) ] (\text{state1}) \]

- At the microprogram level:
  For a correct read access, on account of the multiplexing of DATA, the memory access pins take the successive values:
  1. ADDRESS and DATA receive the address, WRITE<0, VMA<1
  2. VMA<0 to point out that a read can be done
  3. DATA receives the read value
  This corresponds to a sequence of three microprograms. 2) and 3) are always realized respectively by the microprograms "read_req" and "reading":

  \[
  \begin{align*}
  \text{read}_\text{req} &= \lambda \text{st2}.
  \\
  &<\text{mem}(\text{st2}), \text{reg}(\text{st2}), \text{dep}(\text{st2}), \text{base}(\text{st2}), \text{pc}(\text{st2}), \text{ar}(\text{st2}), \text{lr}(\text{st2}), \\
  &\text{address}(\text{st2}), \text{data}(\text{st2}), 0, 0 >
  \\
  \text{reading} &= \lambda \text{st2}.
  \\
  &<\text{mem}(\text{st2}), \text{reg}(\text{st2}), \text{dep}(\text{st2}), \text{base}(\text{st2}), \text{pc}(\text{st2}), \text{ar}(\text{st2}), \text{lr}(\text{st2}), \\
  &\text{address}(\text{st2}), \text{CONT}(<\text{mem}\text{address}\text{data}->)(\text{st2}), \\
  &\text{vm}(\text{st2}), \text{write}(\text{st2})>
  \end{align*}
  \]

Different microprograms are used to compute the address from different sources; for example from DATA when an instruction occurs, or from the instruction fields:

\[
\begin{align*}
\text{read}_\text{ad}(\text{mod}, \text{word}) &= \lambda \text{st2}.
  \\
&<\text{mem}(\text{st2}), \text{reg}(\text{st2}), \text{dep}(\text{st2}), \text{base}(\text{st2}), \text{pc}(\text{st2}), \\
&\text{ar}(\text{st2}), \text{lr}(\text{st2}), \text{w}(\text{st2}), \\
&\text{AD}_{\text{PRIM}}(\text{mod}, \text{word})[0:5]>(\text{st2})>
  \end{align*}
\]

AD_PRIM(mod, word)[6:21](st2), 1.0-

Note that for a write access, we use on a same microprogram two anterior values of DATA: the address and the value to be written. This is modelled using the "Past" functional[16],[7].

The second operand is thus computed by the sequence "reading o read_req o read_ad(mod, word)" where o denotes the functional composition. Its value, given by "reading" and carried by DATA is:

\[
\begin{align*}
\text{CONT}(<\text{mem}, \text{AD}_{\text{PRIM}}(\text{mod}, \text{word})[0:5]),
&\text{AD}_{\text{PRIM}}(\text{mod}, \text{word})[6:21]) > (\text{st2})
  \end{align*}
\]

The function AA used in the level_1 expression is defined from AD_PRIM. If the mode is direct then AA is equivalent to AD_PRIM else AA computes the address from CONT.<mem,AD_PRIM(mod,word)[0:5]>

5. Semantics of machine instructions

5.1. Semantics of machine instructions at level-1: the "SEM" function

The function "SEM" associates to each instruction I" its functional semantics: the function SEM(I), of type State1 -> State1.

In the general case, where the instructions have several coding formats, the function "SEM" is split up into several subfunctions, on account of the different format classes. An environment function "ENV" is associated to the class #i, and each subfunction "SEM" is in turn split into other subfunctions, according to the different operative kinds of instructions: calculation, transfer, branch, ... .

Example: Semantics of M1T1 instructions

The MTI semantic function is defined by:

SEM = λ inst st1.

\[
\begin{align*}
&\text{let op1 = FIRST}(\text{ENV}(\text{inst})), \text{op2 = SECOND}(\text{ENV}(\text{inst})), \text{mnem = s_mnem}(\text{inst})
  \end{align*}
\]

<table>
<thead>
<tr>
<th>Case</th>
<th>mnem</th>
<th>SEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD, STORE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH, POP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCON, STCON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITREQ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where SEM-CALC, SEM-TRANSF... define the semantics for each operative category of instructions. Calc is the set of mnemonics for the calculation instructions.

We give the expression of the function SEM-CALC:

SEM-CALC = λ (mnem, st1, op1, op2).

\[
\begin{align*}
&\text{let arg1 = CONT}(\text{op1})(\text{st1}), \text{arg2 = CONT}(\text{op2})(\text{st1}), \text{carry} = \text{NTH}(\text{sr}(\text{st1}), 11), \text{dest_pl} = \text{SECOND}(\text{op1}),
  \end{align*}
\]

\[
\begin{align*}
&\text{let res = CASE mnem is}
  \end{align*}
\]

\[
\begin{align*}
&\text{RRC, RLO: ALO (mnem, arg2, carry)} \quad \text{ADC, SBC: } \text{ALO (mnem, arg1, arg2, carry)}
  \end{align*}
\]

\[
\begin{align*}
&\text{ASR: ALO (mnem, arg1)} \quad \text{otherwise: ALO (mnem, arg1, arg2)}
  \end{align*}
\]

\[
\begin{align*}
&\text{<mem(st1), SUBST(reg(st1), dest_pl(st1), res), dep(st1), base(st1), NEXT_PC(pc(st1)), NEW_SR (mnem, res, st1)>}
  \end{align*}
\]

where "ALO" interprets the mnemonics of calculation instructions into an arithmetic or logical function.

The value "res" issued from the operation is assigned to a register "REG", by means of the function "SUBST"; the main memory and registers "DEP" and "BASE" are not modified.

5.2. Functional semantics of the Instructions at level-2: the "Impl<mpgmprenm>gm function"

We define the "Impl<mpgmprenm>gm function" to express the decomposition of an instruction into a sequence of microprograms. As for the SEM function, we split "Impl<mpgmprenm>gm in several subfunctions according to the different instruction formats and operative kinds. Since the memory accesses depend on the addressing mode, each subfunction is in addition defined by
cases from the modes.

The first microprograms of each implementation function realize the instruction fetch. They almost include reading the instruction words and incrementing the PC. For microprocessors with several instruction formats, we parameterized the fetch sequence with the number of words to read. So we prove only once the fetch sequence required for all the instructions which have the same format.

For pipelined architectures, we must decompose the fetch and the execution of each instruction into a microprogram sequence which reflects the several pipeline stages. For example, for the CLIPPER processor [13], we decompose each operation executed in a "integer execution unit" into three microprograms which fetch the operand (I stage), perform the arithmetic operation (A stage) and write the result (O stage). To describe the pipeline mechanism, we must specify that these microprograms can be performed at the same time. We have to verify, between levels 2 and 3, that the corresponding microinstruction sequences allow their simultaneous execution.

In addition to the instructions, we must define the microprogram sequences which implement exception cases (reset, external interrupts). As for the instructions, we define an interrupt syntax to represent the several kinds of interrupt. This includes the interrupt level and some information to compute the interrupt vector. These attributes correspond at level 3 to boolean values computed from the interrupt pins. We define a function "impl_interrupt" which associates a microprogram sequence to an interruption. For example, for the 80800, we associate to the interrupt number a microprogram sequence composed of a read sequence to store the value of PC and SR in the supervisor stack, and a read sequence to read the interrupt vector and load it in PC and SR.

Example: the MTI "implpimpmt" function.

The implementation function of the MTI is defined by:

\[
\text{IMPL}_{\text{MTI}} = \lambda \text{inst}. \text{state2}.
\]

\[
\text{case mne}(\text{inst}) \text{ is}
\]

- \[\text{CALC: IMPL}_{\text{CALC}}(\text{inst},\text{state2}) \]

Notice that at this level, we give in parameter the whole instruction and not only the operand functions, while the second operand is computed by a read microprogram sequence whose structure depends on the instruction fields.

We give below the expression of the \(\text{IMPL}_{\text{CALC}}\) function:

\[
\text{IMPL}_{\text{CALC}} = \lambda \text{inst}. \text{state2}.
\]

- let \(mne = \text{mne}(\text{inst}), \text{mod} = \text{mod}(\text{inst}), \text{op1} = \text{OP1}(\text{inst}), \text{imop} = \text{IMOP}(\text{inst})\)

\[
\text{in} \{ \text{case mode is}
\]

- \(\text{IN}: \text{assign_res}_\text{im}(\text{mne},\text{op1},\text{imop})\)

\[
\text{in} \{ \text{SD, XD}: \text{assign_res}(\text{mne},\text{op1}) \text{ o reading o read_reg o read_ad(mod,x,word)}\}
\]

\[
\text{in} \{ \text{SI, X}: \text{assign_res}(\text{mne},\text{op1}) \text{ o reading o read_reg o read_ad_data o reading o read_reg o read_ad(mod,x,word)}\}
\]

- \(\text{fetch(1) state2}\)

where

- \(\text{OP1} \text{ and IMOP compute respectively the first operand and the immediate operand}\)

- \(\text{assign_res}_\text{im} (\text{resp. assign_res}) : \text{the register defined by op1 is assigned the value computed by ALO(mne) from CONT(op1)(state2) and imop(state2) (resp. data(state2))}\).

In fact, "assign_res" (resp. "assign_res_im"), computes the value "res" defined at level 1 (CI chap. 5.1.3), when the second operand is "imop" (resp. "data"). The status register is also modified by these two microprograms.

- \(\text{read_ad(mod,x,word)}: \text{"address" and "data" are assigned the address computed by the AD_PRIM function}\)

- \(\text{read_ad_data: used for indirect addressing, "address" and "data" are assigned the based address computed from the value of "data" returned by the "reading" microprogram}\)

6. Correctness proof between level 1 and level 2:

6.1. Generalitys

To prove that the level 2 specification is consistent with the level 1 specification, we have to verify that:

\[
\text{rest} \circ \text{implppgm}(\text{inst})(\text{st2}) = \text{SEM}(\text{inst}) \circ \text{rest}(\text{st2})
\]

"rest" is the restriction function of level 2 state variables to level 1 state variables.

- \(\ast\) denotes the equality of the two values under some hypothesis: the correct behaviour of the memory.

If the level 1 function is itself a composition of state subfunctions, then we have only to prove the correspondence between each subfunction and some level 2 function composition. Furthermore, since the state change is rather pin-point at the two levels, we have only to prove the equivalence for the variables which are modified. We have also to verify that the others remain unchanged.

6.2. Example: the MTI arithmetic instructions

We give below the composition steps for the arithmetic instructions of the MT I for a direct addressing mode. These instructions modify only the registers bank REG, the SR and the PC. Here we compute only the value of REG.

Let \(\text{st2} \in \text{State2}, \text{inst} \in \{\text{SBC,ADC}\} \text{ with s_mode(\text{inst})} \in \{\text{SD, XD}\}\).

Let \(\text{st1} = \text{rest}(\text{st2})\).

In the following we denote: \(\text{mne} = \text{s_mnem}(\text{inst}), \text{mod} = \text{s_mode}(\text{inst}), \text{x} = \text{s_x}(\text{inst}), \text{word} = \text{s_word}(\text{inst})\) and \(\text{op1} = \text{OP1}(\text{inst})\).

- At level 1, the value of REG is given by:

\[
\text{reg} = \text{SEM}(\text{inst})(\text{st1}) = \text{SUBST}(\text{reg}(\text{st1}), \text{dest_place}(\text{st1}), \text{res})
\]

- At level 2, the value is \(\text{reg} = \text{implppgm}(\text{inst})(\text{st2})\), where implppgm(st2) is "assign_res(mne,op1) o reading o read_reg o read_ad(mod,x,word)"

The read microprograms are defined as in part 4.3 and "assign_res" is:

\[
\text{assign_res}(\text{mne}, \text{op1}) = \lambda \text{st2}.
\]

- \(\text{let carry} = \text{NTH}(\text{st2}(11), \text{arg1} = \text{CONT}(\text{op1})(\text{st2}), \text{dest_place} = \text{SECOND}(\text{op1})\)

- \(\text{let } \text{reg} = \text{case mne is}
\]

- \(\text{in} \{ \text{RRC,RLC,}: \text{ALO}(\text{mne})(\text{data}(\text{st2}), \text{carry})\}
\]

- \(\text{ADC, SBC}: \text{ALO}(\text{mne})(\text{data}(\text{st2}), \text{carry})\)

- \(\text{ASR}: \text{ALO}(\text{mne})(\text{arg1})\)

- \(\text{otherwise: \text{ALO}(mne)(arg1, data(st2))}\)

- \(\text{in} \{ \text{mem(st2)}, \text{SUBST}(\text{reg}(\text{st2}), \text{dest_place}(\text{st2}), \text{res}), \text{dep(st2), base(st2), pc(st2), NEW_SR(mne, res, st2)}\}
\]

We give below the values of the state variables which are modified during the composition process. We denote "var@i" the value of the state variable "VAR" at step "i" (i.e. after the microprogram \#i); the state value at the beginning of the sequence is "st@0" (i.e. st@0 = st2).

\[
\text{reg@4} = \text{reg}(\text{assign_res}(\text{op1})(\text{st3}))
\]

- \(\text{SUBST}(\text{reg}@3, \text{SECOND}(\text{op1})(\text{st3}), \text{ALO}(\text{mne})(\text{CONT}(\text{op1})(\text{st3}), \text{data}(\text{st3}), \text{NTH}(\text{st3}(3,11)))\)

\[
55
\]
variable set. Thus we can substitute in the expression of "reg@4",

The functions AD-PRIM, ALQ and OP1 are the same at the two
values of others state variables. "address" variables are eliminated since they depend on past
we have only to prove that:

Notice that by composing the microprograms, the "data" or
reg(st1) for reg@0, st1 for st@0, and sr(st1) for sr@0. Therefore,
we have only to prove that:

The expressions obtained at these two specification levels
are quite close. In fact, this is not surprising because level-2 is
introduced only to decompose the instructions according to the
microprogram is proved only once; for example, the sequencing of the microprograms which
allow to decompose the verification process into successive
partial proofs have been obtained for the MTI processor, between
level_1 and level_2, with the BOYER & MOORE inductive proof
system and the OBJ Term Rewriting System [8], and benchmark tests are in progress, using the Tautology Checker TACHE [3].

Acknowledgements.
I would like to thank Mrs Dominique BORRIONE, who has
supervised this work with much care and kindness and Mr
Jean-Luc PAILLET, for his fruitful discussions. The work reported
here was partly supported by CNET (MEYLAN), under contract
n°838091007909245CNS.

References
Functional approach to formal hardware verification: the MTI
processor", Proc.IEEE Intern. Conf. on Computer Design
ICCD'88, Port Chester, New-York, October 1988.
P-Calcule", Tec Rep 89-06, Univ. de Provence. Jul 89
Mathematics Studies, N°6, Princeton Univ. Press, 1941.
the first level", In: VLSI Specification Verification and Synthesis,
Edited by G.BIRTWISTLE and P.A. SUBRAHMANYAM, Kluwer
microprocesseurs: le niveau Microprogramme", Tec Rep n°89-08,
Univ. de Provence.
using OBJ3 and the Boyer-Moore Theorem Prover: a practical
Comparison", Tec Rep n°88-04, Univ. de Provence, Marseille,
Oct. 88.
INRIA, 88.
Univ. of Cambridge.1983.
Scientific Publication,87.
Design", IFIP WG 10.2 Workshop, "From HDL descriptions to
guaranteed correct circuit designs", ed. by D.BORRIONE,
[14] JJOYCE: "Formal verification and implementation of a
microprocessor" in VLSI Specification Verification and Synthesis,
Ed. by G.BIRTWISTLE and P.A. SUBRAHMANYAM, Kluwer
machines pour les processeurs du type Von Neumann" R.R.
MAUP n°88-05, Univ. de Provence, Marseille, Nov 88
[16] J.L.PAILLET: "Functional Semantics of Microprocessors at the
Machine Instruction Level", CHDL 89, Washington, June 89.
[17] PULOU, RAINARD, THOREL: "Microprocesseur a test intégré
Approach to Programming Language Theory",MIT Press 77.