The Verification of AVM–1 *

Phillip J. Windley
Division of Computer Science
University of California, Davis

Research Report CSE-90-21
July, 1990

Abstract

AVM–1 is a microprocessor designed at the University of California, Davis to serve as a testbed for experiments in microprocessor verification. This report is intended to document the HOL code that verifies that the design of *AVM–1* implements its specification. The report begins by briefly describing the architecture and organization of *AVM–1* . The organization of the proof is discussed and several technical details relating to the execution of the proof scripts in HOL are presented. The final section of the report contains all of the HOL code necessary to perform the verification.

*This work was sponsored under Boeing Contract NAS1-18586, Task Assignment No. 3, with NASA-Langley Research Center.

Contents

Table 1: The program status word.

Bit	Meaning when set
0	Last ALU result was zero
1	Last ALU operation caused a carry
2	Last ALU result was negative
3	Last ALU operation caused a overflow
4	Interrupts enabled
5	In supervisory mode

1 Introduction.

This technical report is intended to document the HOL code verifying a microprocessor called *AVM–1* . This section will give a brief introduction to the design of *AVM–1* . The next section will discuss the organization of the proof and present some technical details concerning the execution of the proof scripts in HOL. The last section contains the proof scripts used to verify *AVM–1* .

1.1 *AVM–1* .

We have designed a computer designated *AVM–1* (*A Verified Microprocessor*) to demonstrate the use of generic interpreters in verifying hierarchically decomposed microprocessor specifications. For a more detailed look at the architecture and organization of *AVM–1* , see [Win90a].

Our design is an attempt to build a microprocessor that is at once verifiable, implementable, and usable. We have been influenced by our own experience in verifying microprocessors [Win90b], the experience of others [Joy89, Coh88], and our desire to provide hardware features in support of operating systems; such features include interrupts, memory management, and supervisory modes. *AVM–1* is part of a verified chip set being designed and verified by the Computer Systems Verification Group at the University of California, Davis. Other pieces of the system include a memory management unit, a floating point unit, an interrupt controller, and a direct memory access chip.

1.2 An Architectural View.

A computer's architecture is its programming interface; an architecture describes a language and how that language is interpreted. The language definition contains a specification of the computer's state and the instructions available for manipulating that state. The architecture must also define how instructions are selected.

The instruction set for *AVM–1* was inspired by the RISC I instruction set found in Katevenis [Kat85]. There are a number of differences, but many features in the RISC I instruction set (such as using ALU operations to synthesize a MOVE instruction) were incorporated into the *AVM–1* instruction set. As we will see in the section on organization, however, *AVM–1* cannot be called a RISC architecture since its microcoded implementation is somewhat different than today's RISC chips.

1.2.1 The Registers.

AVM–1 has a load–store architecture based on a large register file. The register file is divided into three portions:

1. Register 0 which is read–only and contains the constant 0.
2. Seven supervisor–mode registers including a distinguished register for use as the supervisor stack pointer (`SSP`). The supervisor–mode registers are read–only unless the CPU is in supervisor–mode (determined by the 6^{th} bit in the program status word).
3. Twenty–four general purpose registers.

Two additional registers are visible at the architectural level: the program counter and the program status word. The program counter (`PC`) is used to sequence the computer—it indicates which instruction to execute next.

The program status word (`PSW`) is used to keep track of the status of the last ALU operation, whether or not interrupts are enabled, and the privilege level of the CPU. Table 1 shows the meaning of the 6 bits in the program status word.

AVM–1 shares a register, `IVEC`, with the interrupt controller. This register contains the interrupt vector and is read–only as far as the CPU is concerned.

1.2.2 The Instruction Set.

The instruction set contains 30 instructions. The opcode space has room for 64; the upper half of the opcode space is reserved for future co–processors. As mentioned above, the instruction set is based on a load–store architecture, meaning that most instructions are not allowed to access memory for their operands.

The instruction formats are simple and regular. Figure 1 shows the four instruction formats. All of the formats use the same opcode field.

In formats 1 and 2, the instruction is divided into four fields. The top 6 bits (31–26) give the opcode of the instructions. The next 5 bits (25–21) denote the destination register in most operations. The third field (bits 20–16) selects the register used as the A operand in most operations. In format 1, the fourth field is comprised of bits 15–11 and is used to select the register used as the B operand. In format 2, the fourth field uses all of the 16 remaining bits to form an immediate number (0 to $(2^{16}-1)$).

Format 3 is identical to formats 1 and 2 except that only the opcode and destination fields are used. Format 4 uses only the opcode field.

There is a trade off between instruction format complexity and verification effort, so in general the instruction format should be kept as simple as possible. A regular instruction format, while not essential to verification, can greatly reduce the amount of detail that has to be dealt with in the proof.

Table 2: The *AVM–1* instruction set.

Mnemonic	*Format*	*Effect*
JMP	2	Jump to new location on condition flags
CALL	2	Call subroutine
INT	2	User interrupt
RTI	4	Return from interrupt
GPSW	3	Get program status word
PPSW	3	Put program status word
LD	1	Load register
ST	1	Store register
LSL	1	Logical shift left
LSR	1	Logical shift right
ASR	1	Arithmetic shift right
RTN	3	Return from subroutine
LDI	2	Load register using immediate value
STI	2	Store register using immediate value
ADD	1	Add
ADDC	1	Add with carry
SUB	1	Subtract
SUBC	1	Subtract with borrow (carry)
BAND	1	Bit–wise conjunction
BOR	1	Bit–wise disjunction
BXOR	1	Bit–wise exclusive disjunction
BNOT	1	Bit–wise negation
ADDI	2	Add using immediate value
ADDCI	2	Add with carry using immediate value
SUBI	2	Subtract using immediate value
SUBCI	2	Subtract with borrow using immediate value
BANDI	2	Bit–wise conjunction using immediate value
BORI	2	Bit–wise disjunction using immediate value
BXORI	2	Bit–wise exclusive disjunction using immediate value
NOOP	4	No operation

Format 1

31	25	20	15	10	0
opcode	dest	A	B	unused	

Format 2

31	25	20	15	0
opcode	dest	A	immediate	

Format 3

31	25	20	0
opcode	dest	unused	

Format 4

31	25	0
opcode	unused	

Figure 1: The instruction formats in *AVM–1* .

The 30 programming level instructions are shown in Table 2. There is a group of 8, 3–argument arithmetic instructions and another group of 8 arithmetic instructions that use a 16–bit immediate value. There are 4 instructions for loading and storing registers. In addition, there are instructions for performing user interrupts, jumps, subroutine calls, and shifts. For a detailed description of the instruction set, see [Win90a].

Synthesizing Addressing Modes. Besides the `CALL` and `INT` instructions which must access a stack, only the load and store instructions can access memory. All of the other instructions only operate on the internal registers. This makes the implementation of the instruction set easier and results in faster operation of most of the instructions.

The addressing mode in the load and store instructions uses the sum of two numbers, a register and either a register or an immediate value, to calculate the address of the memory operation. This is a flexible scheme which allows most popular addressing modes to be synthesized.

Table 3 (adapted from [Kat85]) shows how the memory addressing scheme in *AVM–1* can be used to support common constructs in modern high–level languages.

- In direct mode, the A register holds the base of the data segment and the immediate value allows addressing within $\pm 2^{15}$ of the base.

Table 3: Synthesizing addressing modes using *AVM–1* 's load and store instructions.

Mode	*HLL Usage*	*Synthesizing in AVM–1*
Direct	Global Scalar	M[R[a] + imm]
Indirect	Pointer Dereferencing	M[R[A] + R[0]]
Indexed	Record Field	M[R[a] + imm]
Indexed	Array Element	M[R[a] + R[b]]

- In indirect mode, the A register holds the value of the pointer. `R[0]` holds the constant 0.
- To perform memory operations on records, the A register holds the base address of the record and the immediate field holds the field offsets into the record.
- Array operations are performed by using the A register to hold the base address of the array and the B register hold the index.

1.2.3 Selecting Instructions.

We select instructions in the instruction set using the opcode portion of the word in memory pointed to by the current value of the program counter. We will only use the 5 least significant bits of the opcode field, giving space for 32 instructions.

Table 4 gives a breakdown of the opcodes for *AVM–1* . The instruction set is divided into four groups depending on the value of the first 2 bits in the opcode. The first two groups contain miscellaneous instructions, the third group contains ALU operations and the fourth group contains the immediate version of the instructions in group 3.

1.3 An Organizational View.

This section describes the organization of *AVM–1* –what components are used and to what effect. The implementation of *AVM–1* can be divided into two major parts: the datapath and the control unit. We will discuss each of these.

1.3.1 The *AVM–1* Datapath.

The *AVM–1* datapath is loosely based on the AMD 2903 bit-sliced datapath [Adv83] and is shown in Figure 2. The signals shown at the right-hand side of the figure connect to the control unit. The signals on the left go to or come from the environment. Note that none of the clocking signals are shown.

The datapath has three buses, a register file containing 32 registers, and numerous support registers and latches. Two buses, `A` and `B`, are connected to the output ports on the register file and system

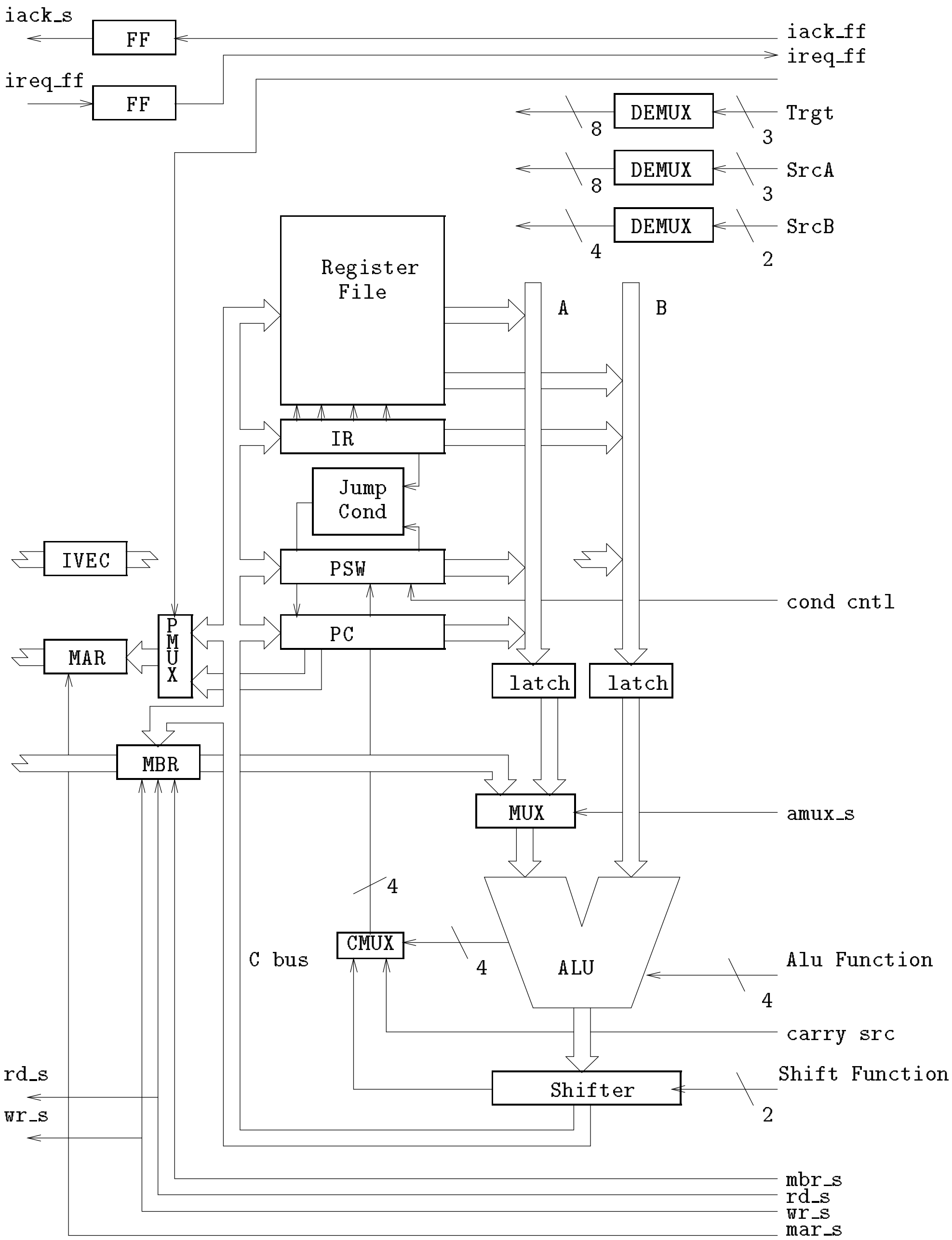

Figure 2: The *AVM-1* Datapath

Table 4: Opcode breakdowns for *AVM–1* 's instruction set.

	00XXX	01XXX	10XXX	11XXX
000	JMP	LSL	ADD	ADDI
001	CALL	LSR	ADDC	ADDCI
010	INT	ASR	SUB	SUBI
011	RTI	RTN	SUBC	SUBCI
100	GPSW	NOOP	BAND	BANDI
101	PPSW	NOOP	BOR	BORI
110	LD	LDI	BXOR	BXORI
111	ST	STI	BNOT	NOOP

registers. The `C` bus is connected to the input port on the register file and the system registers. In addition, the interrupt vector is attached to the `B` bus through a special port to the interrupt controller.

The `A` and `B` buses feed the inputs to the ALU through two latches. The memory buffer register can also serve as the `A` input to the ALU through a multiplexor on the ALU input. The ALU performs simple arithmetic and boolean operations on the values on its `A` and `B` inputs. The results of the ALU operation are fed to the shifter which can perform logical and arithmetic shifts. The result from the shifter is put onto the `C` bus for distribution.

In addition to a result, the ALU produces a set of status bits (negative, zero, carry, and overflow) which can be saved in the program status word directly. If desired, a one–bit multiplexor also allows the bit shifted out of the shifter to be saved in the carry field of the PSW. The control lines to the PSW allow the supervisor and interrupt enable bits to be set and cleared and each of the status bits to be loaded individually.

The status from the PSW and the destination field of the instruction register are fed into the jump code circuitry. This combinatorial circuit calculates the jump conditions shown in Table 5 and supplies a boolean result which is used to determine if the program counter should be loaded from the `C` bus. The program counter can also be loaded unconditionally.

The instruction register can be loaded from the `C` bus, but only the immediate portion of the instruction register can be placed on the `B` bus.

The memory address register can be loaded directly from the program counter or from the `C` bus. This allows the MAR to be loaded quickly for instruction fetches while still allowing calculated addresses for loads and stores.

The datapath has two flipflops for holding the status of interrupt actions and three demultiplexors for decoding register selection signals from the control unit.

1.3.2 The Control Unit.

The control unit for *AVM–1* is shown in Figure 3. The control unit has four major blocks: the microprogram counter, the microinstruction register, the clock, and the microrom.

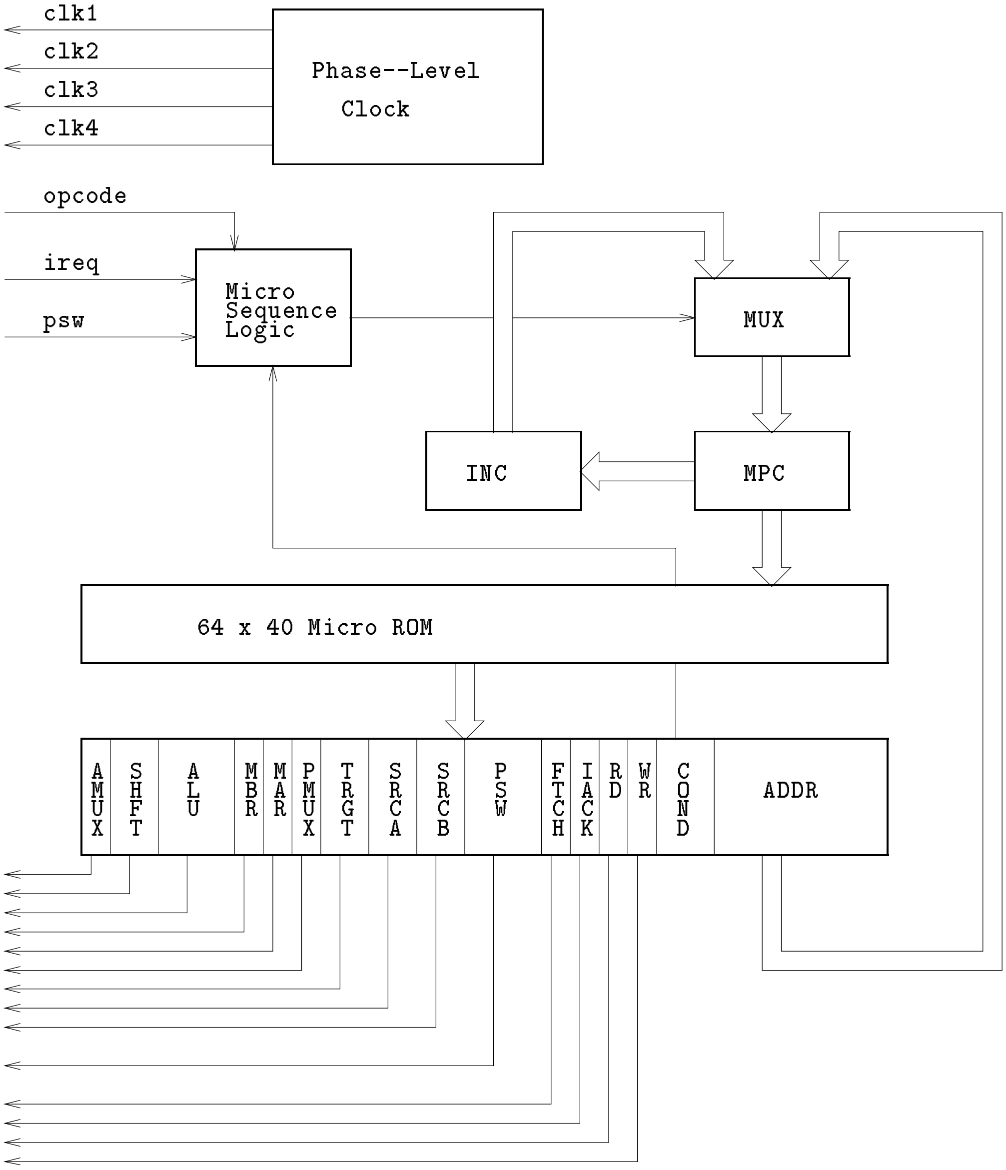

Figure 3: The *AVM-1* Control Unit

Table 5: Implementation of the jump codes for the `JMP` instruction. `cf` is the carry flag in the PSW, `zf` is the zero flag, etc.

Code	*Implementation*
0	cf
1	$\neg$cf
2	vf
3	$\neg$vf
4	nf
5	$\neg$nf
6	zf
7	$\neg$zf
8	($\neg$cf $\vee$zf)
9	$\neg$($\neg$cf $\vee$zf)
10	(nf xor vf)
11	$\neg$(nf $\oplus$vf)
12	$\neg$((nf $\oplus$vf) $\vee$zf)
13	((nf $\oplus$vf) $\vee$zf)
14	true
15	true

The microprogram counter is the most complex of the four. The purpose of the microprogram counter is to compute the next address for the microprogram based on the current system state. The microprogram counter is fed the condition and address (`addr`) fields from the microinstruction register, the opcode from the instruction register, and the supervisory and interrupt enable bits from the program counter. There are 5 jump conditions:

1. No jump; the microprogram counter is incremented. This is the default operation.
2. Jump to `addr` unconditionally
3. Jump to the location given by the `opcode` signal and an offset (4 in this case). This allows us to use a table lookup approach to instruction decoding in the microcode. We only use the 5 least significant bits of the 6–bit opcode; the top half of the instruction set is reserved for a coprocessor.
4. Jump to `addr` if the interrupt signal is true and interrupts are enabled.
5. Jump to `addr` if the supervisory mode signal is true.

The microinstruction register is a 40–bit register that holds the current microinstruction. The only special feature of the register is that each of the fields from the microinstruction are available through separate ports for use elsewhere in the control unit and datapath.

The microinstruction format is shown in Table 6. A microinstruction consists of 40 bits in 24 fields. The fields in a microinstruction can be broken into 4 groups: those affecting the operation of

Table 6: The microinstruction format for *AVM–1* .

Operation Group

Bits	Mnemonic	Description
1	AMUX	Toggle MUX on A-bus
2	SHFT	Shifter function
4	ALU	ALU function
1	MAR	Load MAR from P-Mux
1	MBR	Load MBR from C-bus
1	PMUX	Toggle MUX loading MAR
3	SRCA	A-bus source
2	SRCB	B-bus source
3	TRGT	C-bus target

Program Status Word Group

Bits	Mnemonic	Description
1	S_SM	Set supervisory mode bit in PSW
1	C_SM	Clear supervisory mode bit in PSW
1	S_IE	Set interrupt enable bit in PSW
1	C_IE	Clear interrupt enable bit in PSW
1	LD_C	Load carry bit in PSW
1	LD_V	Load overflow bit in PSW
1	LD_N	Load negative bit in PSW
1	LD_Z	Load zero bit in PSW
1	CSRC	Source of carry (shifter or alu)

External Signals Group

Bits	Mnemonic	Description
1	IACK	Interrupt acknowledge signal
1	FTCH	Fetch signal
1	RD	Read signal
1	WR	Write signal

Microprogram Counter Group

Bits	Mnemonic	Description
3	COND	Microcode jump condition
6	ADDR	Next address

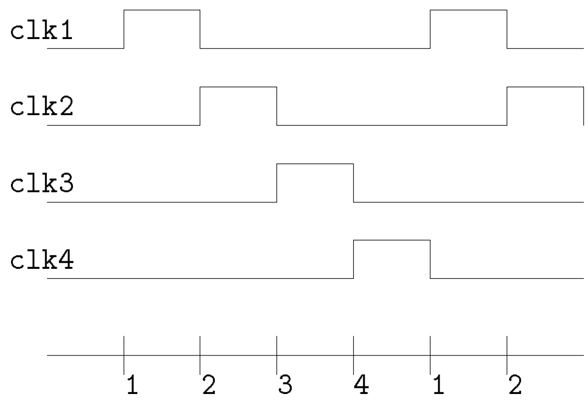

Figure 4: The clock signals in *AVM–1* .

the microprocessor, those affecting the program status word, those dealing with external signals, and those that are used for microinstruction sequencing. For a detailed description of the microinstructions, see [Win90a].

The clock is a simple four–phase counter with a strobe line for each phase. Figure 4 shows the output timing for the clock. The `clk1` line, for example, is only true during phase 1, the `clk2` line is true during phase 2, and so on.

The microrom holds the microcode and is made from a read–only memory that is 40–bits wide and 64 words long.

1.3.3 Timing.

The timing of *AVM–1* is based on a four phase clock (see Figure 5). During the four phases, the machine performs the following state transitions:

1. In phase 1, the microinstruction register is loaded from the microrom.
2. In phase 2, the latches feeding ALU are loaded from the register file and system registers.
3. In phase 3, the results from the ALU and shifter are calculated. In addition, the MAR can be loaded from the PC in this phase.
4. In phase 4, the result calculated in phase 3 is stored back into the register file and system registers.

Every microinstruction is executed by the phase sequence described above. Since microinstructions are used to implement the macroinstructions, the timing for a macroinstruction is dependent on the number of microinstruction in its implementation. In most cases this number is 4.

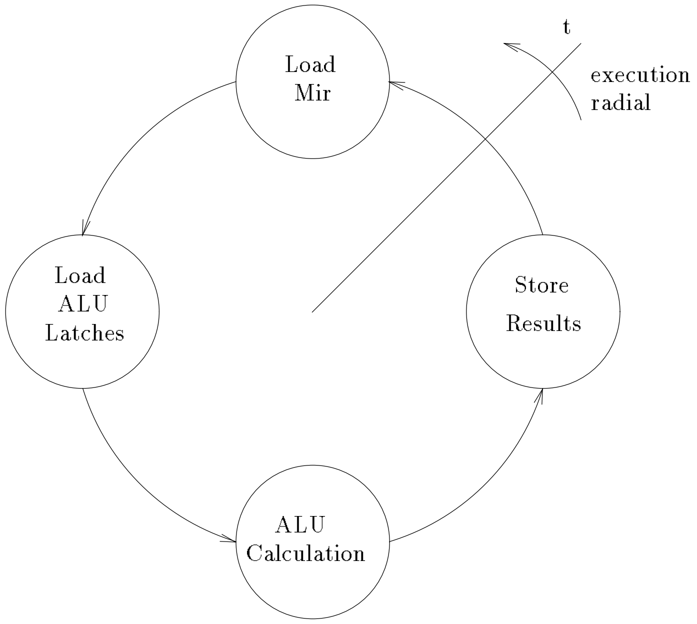

Figure 5: A PERT phase diagram for *AVM–1* .

2 The Organization of the Proof

This section presents the organization of the proof of *AVM–1* in HOL. The section discusses the overall proof organization, gives a description of the theories making up the proof and gives some measurements of the complexity of the proof.

2.1 Proof organization

The proof for *AVM–1* contains more than 25 theories. This section presents the general proof organization (the hierarchy of theories) and briefly describes the contents of each theory.

Figure 6 shows how the major theories of the proof of *AVM–1* are related. This hierarchy shows `avm.th` as the child theory of a long ancestry that follows the hierarchical decomposition discussed in [Win90a]. The picture is not complete; there are many theories not shown. For example, `aux_def.th` is the ancestor of almost every theory in the proof.

The rest of this section gives a taxonomy of the major theories in the proof of *AVM–1* .

Generic Interpreters. The generic interpreter theories include the synchronous model, the temporal abstraction theory, and the asynchronous model.

- **gen_I_sync.th** — Defines and verifies a synchronous version of the generic interpreter theory.

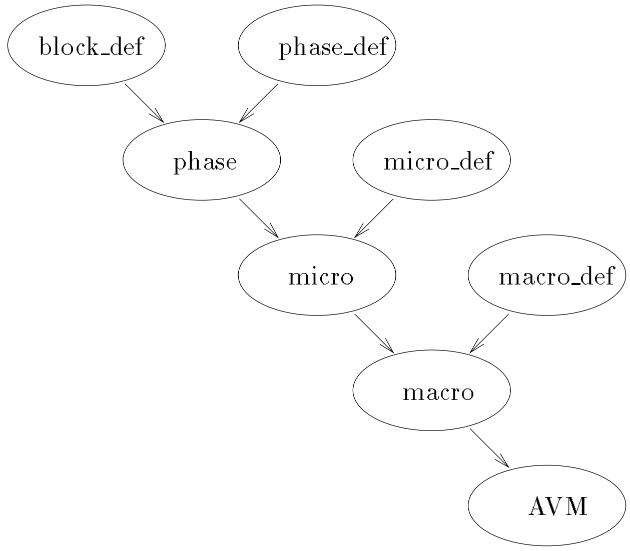

Figure 6: The theory hierarchy for the proof of *AVM–1* .

- **time_abs.th** — Defines a temporal abstraction function and proves several useful lemmas concerning it.
- **gen_I.th** — Contains the generic definition of an interpreter used in the definition and proof of the various levels in *AVM–1* .

Auxiliary Theories. There are a number of auxiliary theories that are used throughout the proof of *AVM–1* .

- **aux_defs.th** — Contains the abstract definition for n–bit words. The definition is accomplished using the functions in `abstract.ml`, the ML code for producing abstract theories.
- **aux_thms.th** — Contains auxiliary definitions and theorems. The theory is an ancestor of many of the main theories in the proof.
- **jump_def.th** — Contains the definition of the jump condition logic that is used at every level.
- **regs_def.th** — Contains the definition of the register file. Several distinguished registers are defined and the function for updating the register file is given.

The Electronic Block Model. The electronic block model description depends on a number of theories. The definition makes use of a generic ALU that is subsequently instantiated to define the ALU used in *AVM–1* . The shifter and microprogram counter are also defined separately.

- **mux16_def.th** — Contains the definition of a 16 input multiplexor that is used in the definition of the generic ALU theory.

- **gen_alu.th** — Contains the abstract definition and verification of a 16 function ALU.
- **alu_def.th** — Contains the instantiation of the generic ALU theory presented in the last section for a specific set of functions. The correctness result is meaningless since the modules used to implement the functions are null modules. This does not affect the validity of the proof presented here since only the definition is used in subsequent theories. A number of theorems about the ALU's output are proven here and are used in subsequent proofs.
- **shifter_def.th** — Contains the definition of a 4 function shifter that is used in defining the electronic block model. A number of theorems about the shifter's output are proven here and are used in subsequent proofs.
- **mpc_def.th** — Contains the definition of the microprogram counter unit that is used in the definition of the electronic block model and the phase–level.
- **mpc_def.th** — Contains the definition of the state selectors for the electronic block model.
- **block_def.th** — This theory contains the definition of the electronic block model. The theory contains the definition of most of the blocks used to construct the electronic block model.

The Phase–Level. This section presents the theories that define the phase–level interpreter. Also presented is the theory that verifies the phase–level interpreter with respect to the electronic block model.

- **ucode_aux.ml** — Contains the ML code that defines the microcode assembler. No theory is created; the assembler is an ML program that creates the appropriate terms for a given program statement.
- **ucode_def.th** — Defines the type for the microcode as well as a number of selector functions that return the various fields that make up a microinstruction.
- **phase_def.th** — Defines the abstract behavior of the 4 phase–level instructions and gives several auxiliary definitions used in instantiating the abstract interpreter theory.
- **phase.th** — Contains the correctness result for the phase–level. The result is obtained by instantiating the generic interpreter theory contained in `gen_I.th`.

The Micro–Level. This section presents the theories that define the micro–level interpreter. Also presented is the theory that verifies the micro–level interpreter with respect to the phase–level interpreter.

- **micro_def.th** — Defines the abstract behavior of the 64 micro–level instructions and gives several auxiliary definitions used in instantiating the abstract interpreter theory.
- **uinst_def.th** — Defines the microinstructions and combines them together into the microrom.
- **micro.th** — Contains the correctness result for the micro–level. The result is obtained by instantiating the generic theory `gen_I.th`.

The Macro–Level. This section presents the theories that define the macro–level interpreter. Also presented is the theory that verifies the macro–level interpreter with respect to the micro–level interpreter.

- **macro_def.th** — Defines the abstract behavior of the 32 macro–level instructions and gives several auxiliary definitions used in instantiating the abstract interpreter theory.
- **macro.th** — Contains the correctness result for the macro–level. The result is obtained by instantiating the generic theory `gen_I.th`.

The Final Result. This section presents the theory that prove *AVM–1* correct. The theory is the descendant of all of the theories presented earlier.

- **avm.th** — Contains the correctness result for the microprocessor. The final result is obtained by combining the correctness results from `phase.th`, `micro.th` and `macro.th`.

2.2 Proof Metrics.

Table 7 presents the run–times for the various theories in the proof on a SPARCStation with 16 Mbytes of memory. The times are CPU seconds. The table also gives the number of primitive inferences required to run the corresponding ML script in HOL. We were using version 1.10 of HOL built using the Austin Kyoto Common Lisp compiler.

The total time to run the proof was 208029.1 CPU seconds, or nearly 58 CPU hours. The proof took almost a week of elapsed time because the core images were quite large (as high as 29 Mbytes) and caused the operating system to thrash when garbage collecting.

There are several files in the table that were not discussed in the last section. Due to size limitations of main memory, the files `mk_mic_x1.ml` and `mk_mic_x2.ml` were broken out of `mk_micro.ml` and `mk_mac_I.ml`, `mk_mac_1.ml`, and `mk_mac_2.ml` were broken out of `mk_macro.ml`.

Table 7: Script run–times on a SPARCStation with 16M of memory.

File Name	*Time (CPU sec.)*	*Inferences*
def_aux.ml	3070.7	88
mk_aux.ml	1117.5	33852
def_regs.ml	41.0	14
def_jump.ml	50.7	4
def_macro.ml	2373.5	84
mk_time.ml	126.8	7256
mk_I.ml	229.9	11727
def_micro.ml	7063.6	48460
def_mpc.ml	6.4	4
def_ucode	115.6	50
def_phase.ml	915.2	32
def_mux16.ml	344.2	29211
mk_gen_alu.ml	8038.4	101155
def_alu.ml	2325.3	70815
def_shift.ml	129.0	2891
def_select.ml	1969.0	43903
def_block.ml	1316.0	14738
mk_phase.ml	12818.4	355161
def_uinst	568.5	107
mk_mic_x1.ml	54846.2	1589683
mk_mic_x2.ml	51300.6	1500604
mk_micro.ml	13505.3	295744
mk_mac_I.ml	688.3	3985
mk_mac_1.ml	16774.1	389738
mk_mac_2.ml	20256.1	457606
mk_macro.ml	7247.9	200120
mk_avm.ml	790.9	10031
	208029.1	5167063

3 The Proof

This section documents the HOL theories that make up the proof discussed in [Win90a]. The HOL source code for each theory is presented. The proof organization is presented in Section 2.

3.1 The Generic Interpreters

3.1.1 Synchronous Interpreters

This section presents the ML code that creates the theory `gen_I_sync.th`.

```
%----------------------------------------------------------------

   File:         mk_I.ml

   Author:       (c) P. J. Windley 1990

   Date:         09 JAN 90

   Modified:     14 FEB 90

   Description:

   Defines a generic interpreter used in subsequent specifications.
   The interpreter is proven to be correct under certain obligations.
   The interpreter in this file is synchronous.

   2/13/90 -- Modified to take external lines into account.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                  `/muztag/home/windley/hol/Library/assoc/`;
                                 ]);;

system `/bin/rm gen_I_sync.th`;;

new_theory `gen_I_sync`;;

map loadf [`abstract`];;

new_type_abbrev(`time`,":num");;

new_type_abbrev(`time'`,":num");;

%----------------------------------------------------------------
 Generic specification
----------------------------------------------------------------%

let cpu_abs = new_abstract_representation
```

```
    [
     (`inst_list`,":(*key#(*state->*env->*state))list")
      ;
     (`key`,":*key->num")
      ;
     (`select`,":*state->*env->*key")
      ;
     (`cycles`,":*key->num")
      ;
     (`substate`,":*state'->*state")
      ;
     (`subenv`,":*env'->*env")
      ;
     (`Impl`,":(time'->*state')->(time'->*env')->bool")
      ;
     (`count`,":*state'->*env'->*key'")
      ;
     (`start`,":*key'")
      ;
    ];;

make_inst_thms cpu_abs;;

let I_rep_ty = abstract_type `gen_I_sync` `key`;;

let INTERP_def = new_definition
   (`INTERP`,
    "! (rep:^I_rep_ty) (s:time->*state) (e:time->*env) .
     INTERP rep s e =
       !t:time.
        let n = (key rep (select rep (s t) (e t))) in (
        s(t+1) = (SND (EL n (inst_list rep))) (s t) (e t))"
   );;

let INTERP_DEF_EXPANDED = EXPAND_LET_RULE INTERP_def;;


let inst_correct_def = new_definition
   (`INST_CORRECT`,
    "! inst:(*key#(*state->*env->*state))
       (s':time'->*state')
       (e':time'->*env') .
     INST_CORRECT rep s' e' inst =
        (Impl (rep:^I_rep_ty) s' e') ==>
         (!t:time'.
            let s = (\t. (substate rep (s' t))) in
            let e = (\t. (subenv rep (e' t))) in
            let c = (cycles rep (select rep (s t) (e t))) in (
            (select rep (s t) (e t) = (FST inst)) /\
            (count rep (s' t) (e' t) = (start rep)) ==>
              ((SND inst) (s t) (e t) = (s (t + c))) /\
              (count rep (s' (t + c)) (e' (t + c)) = (start rep))))"
   );;

let INST_CORRECT_EXPANDED = BETA_RULE(EXPAND_LET_RULE inst_correct_def);;
```

```
new_theory_obligations
   [
    "EVERY (INST_CORRECT (rep:^I_rep_ty)
                         (s':time'->*state')
                         (e':time'->*env'))
           (inst_list rep)"
    ;
    "!k:*key. (key (rep:^I_rep_ty) k) < (LENGTH (inst_list rep))"
    ;
    "!k:*key . k = (FST (EL (key (rep:^I_rep_ty) k) (inst_list rep)))"
    ;
   ];;


let IMPL_NEXTSTATE_LEMMA = TAC_PROOF
   (([],
    "let s = (\t:time .(substate rep (s' t))) and
         e = (\t:time .(subenv rep (e' t))) in (
     (Impl (rep:^I_rep_ty)) s' e' ==>
       (!t:time'.
        (count rep (s' t) (e' t) = (start rep)) ==>
          ((substate rep (s' (t+(cycles rep (select rep (s t) (e t)))))) =
           (SND (EL (key rep (select rep (s t) (e t)))
                    (inst_list rep))) (s t) (e t))))"),
    EXPAND_LET_TAC
    THEN BETA_TAC
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM_LIST (\asl .
        let asl' =
             map (PURE_REWRITE_RULE [EVERY_EL;INST_CORRECT_EXPANDED]) asl in
        MAP_EVERY ASSUME_TAC
         (map
           (\thm.
            (SPEC "(key (rep:^I_rep_ty)
                    (select rep
                     (substate rep(s' t))
                      (subenv rep (e' t))))" thm) ?
            (SPEC "(select (rep:^I_rep_ty)
                    (substate rep(s' t))
                     (subenv rep (e' t)))" thm) ?
            thm) asl'))
    THEN RES_TAC
    THEN POP_ASSUM (\thm. ASSUME_TAC (REWRITE_RULE [] (SPEC "t:time'" thm)))
    THEN RES_TAC
    THEN FIRST_ASSUM (ACCEPT_TAC o SYM_RULE)
   );;

let IMPL_NEXTSTATE_LEMMA_EXPANDED =
    BETA_RULE (
    EXPAND_LET_RULE IMPL_NEXTSTATE_LEMMA);;

let time_shift = new_prim_rec_definition
   (`time_shift`,
    "(time_shift f (s:time->*state) (e:time->*env) 0 = 0) /\
```

```
     (time_shift f s e (SUC n) = (
          let t = (time_shift f s e n) in
          t + (f (s t) (e t))))"
   );;

let I_CLOCK_LEMMA = TAC_PROOF
   (([],
    "let s = (\t:time .(substate rep (s' t))) and
         e = (\t:time. (subenv rep (e' t))) in (
     (Impl rep) s' e' /\
     ((count rep) (s' 0) (e' 0) = (start rep)) ==>
     !t. let t_impl =
             (time_shift (\st env. (cycles rep (select rep st env))) s e t) in
        (count (rep:^I_rep_ty)) (s' t_impl) (e' t_impl) = (start rep))"),
    EXPAND_LET_TAC
    THEN BETA_TAC
    THEN REPEAT GEN_TAC
    THEN STRIP_TAC
    THEN INDUCT_TAC
    THEN REWRITE_TAC [time_shift; o_DEF;LET_DEF]
    THEN (FIRST_ASSUM ACCEPT_TAC ORELSE ALL_TAC)
    THEN POP_ASSUM (\thm. ASSUME_TAC
          (CONV_RULE (TOP_DEPTH_CONV BETA_CONV)
           (ONCE_REWRITE_RULE [o_DEF] thm)))
    THEN BETA_TAC
    THEN POP_ASSUM_LIST (\asl .
        let asl' =
             map (PURE_REWRITE_RULE [EVERY_EL;INST_CORRECT_EXPANDED]) asl in
        MAP_EVERY ASSUME_TAC
         (map
           (\thm.
             (SPEC "(key (rep:^I_rep_ty)
                      (select rep
                       (substate rep
                        (s'
                         (time_shift
                          (\st env. cycles rep(select rep st env))
                          (\t'. substate rep(s' t'))
                          (\t'. subenv rep (e' t')) t)))
                       (subenv rep
                        (e'
                         (time_shift
                          (\st env. cycles rep(select rep st env))
                          (\t'. substate rep(s' t'))
                          (\t'. subenv rep (e' t')) t)))))" thm) ?
             (SPEC "(select (rep:^I_rep_ty)
                     (substate rep
                      (s'
                       (time_shift
                        (\st env. cycles rep(select rep st env))
                        (\t'. substate rep(s' t'))
                        (\t'. subenv rep (e' t')) t)))
                     (subenv rep
                      (e'
                       (time_shift
```

```
                      (\st env. cycles rep(select rep st env))
                      (\t'. substate rep(s' t'))
                      (\t'. subenv rep (e' t')) t))))" thm) ?
           thm) asl'))
    THEN RES_TAC
    THEN POP_ASSUM (\thm. ASSUME_TAC (REWRITE_RULE []
           (SPEC "(time_shift
                   (\st env. cycles (rep:^I_rep_ty) (select rep st env))
                   (\t'. substate rep(s' t'))
                   (\t'. subenv rep (e' t')) t):time'" thm)))
    THEN RES_TAC
   );;

let I_CLOCK_LEMMA_EXPANDED =
    BETA_RULE (
    EXPAND_LET_RULE I_CLOCK_LEMMA);;

let IMPL_I_CORRECT = prove_thm
   (`IMPL_I_CORRECT`,
    "let s = (\t:time .(substate rep (s' t))) and
        e = (\t:time .(subenv rep (e' t))) in (
     (Impl rep) s' e' /\
     ((count (rep:^I_rep_ty)) (s' 0) (e' 0) = (start rep)) ==>
     let f = time_shift (\st env. (cycles rep (select rep st env))) s e in
     (INTERP rep) (s o f) (e o f))",
    EXPAND_LET_TAC
    THEN BETA_TAC
    THEN REPEAT GEN_TAC
    THEN PURE_REWRITE_TAC [INTERP_DEF_EXPANDED;o_DEF]
    THEN STRIP_TAC
    THEN IMP_RES_TAC (PURE_ONCE_REWRITE_RULE [o_DEF] I_CLOCK_LEMMA_EXPANDED)
    THEN GEN_TAC
    THEN BETA_TAC
    THEN PURE_ONCE_REWRITE_TAC
          [EXPAND_LET_RULE (REWRITE_RULE [ADD1] time_shift)]
    THEN BETA_TAC
    THEN POP_ASSUM (\x. ASSUME_TAC (SPEC "t:time'" x))
    THEN IMP_RES_TAC IMPL_NEXTSTATE_LEMMA_EXPANDED
   );;



close_theory();;
```

3.1.2 Temporal Abstraction

This section presents the ML code that creates the theory `time_abs.th`.

```
%----------------------------------------------------------------

   File:         mk_time.ml

   Author:       (c) P. J. Windley 1990

   Date:         19 FEB 90

   Modified:

   Description:

   Creates a theory of temporal abstractions as defined in [1,2].
   The theory defines several temporal operators and a temporal
   projection function that can be used to relate time at
   different levels of abstraction.

   [1] Melham, Thomas F., "Abstraction Mechanisms for Hardware
   Verification"

   [2] Joyce, Jeffrey J., "Multi-Level Verification of
   Microprocessor-Based Systems"


----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                  `/muztag/home/windley/hol/Library/assoc/`;
                                 ]);;

system `/bin/rm time_abs.th`;;

new_theory `time_abs`;;

new_type_abbrev(`time`,":num");;

let First = new_definition
   (`First`,
    "! g t. First g t =
        (!p:time. p < t ==> ~(g p)) /\
        (g t)"
   );;

let Next = new_definition
   (`Next`,
    "! g t1 t2. Next g (t1,t2) =
        (t1 < t2) /\
        (!t:time . t1 < t /\ t < t2 ==> ~(g t)) /\
        (g t2)"
```

```
   );;

let Temp_Abs = new_prim_rec_definition
   (`Temp_Abs`,
    "(Temp_Abs g 0 = @ t:time . First g t) /\
     (Temp_Abs g (SUC n) = @ t:time . Next g (Temp_Abs g n,t))"
   );;


let LEMMA1 =
    GEN_ALL (
    DISCH_ALL (
    CONJUNCT2 (
    SELECT_RULE (
    ASSUME "?t:time. P t /\ Q t"))));;

let FIRST_LEMMA1 = TAC_PROOF
   (([],
    "! f .
     (? t:time . f t) /\
     (!t. f t ==> (?n. Next f(t,t + n) /\ r(t,t + n))) ==>
     ? t. First f t"),
    REPEAT GEN_TAC
    THEN STRIP_GOAL_THEN ((MAP_EVERY ASSUME_TAC) o CONJUNCTS)
    THEN IMP_RES_TAC WOP
    THEN PURE_ONCE_REWRITE_TAC [First]
    THEN PURE_ONCE_REWRITE_TAC [CONJ_SYM]
    THEN ASM_REWRITE_TAC []
   );;

let FIRST_LEMMA2 =
    GEN_ALL (
    REWRITE_RULE [SYM_RULE First] (
    IMP_TRANS
        (SPEC_ALL (
         REWRITE_RULE [First] FIRST_LEMMA1))
        (BETA_RULE (
         SPECL ["\t. (!p. p < t ==> ~(f p))";
                "\t. (f t):bool"] LEMMA1))));;

let NEXT_LEMMA1 = TAC_PROOF
   (([],
    "! f m .
     (? t:time . f t) /\
     (!t. f t ==> (?n. Next f(t,t + n) /\ r(t,t + n))) /\
     (f (Temp_Abs f m)) ==>
     ? t. Next f (Temp_Abs f m, t)"),
    REPEAT STRIP_TAC
    THEN RES_TAC
    THEN ASSUM_LIST (\asl . MAP_EVERY STRIP_ASSUME_TAC asl)
    THEN EXISTS_TAC "(Temp_Abs f m) + n'"
    THEN ASSUM_LIST (\asl. FIRST (map ACCEPT_TAC asl))
   );;

let NEXT_LEMMA2 =
```

```
    GEN_ALL (
    REWRITE_RULE [SYM_RULE Next] (
    IMP_TRANS
        (SPEC_ALL (
         REWRITE_RULE [Next] NEXT_LEMMA1))
        (PURE_ONCE_REWRITE_RULE [SYM_RULE CONJ_ASSOC] (
         BETA_RULE (
         SPECL ["\t. ((Temp_Abs f m) < t) /\
                     (!t'. (Temp_Abs f m) < t' /\ t' < t ==> ~f t')";
                "\t. (f t):bool"] LEMMA1)))));;

let ALL_F_Temp_Abs = TAC_PROOF
   (([],
    "! f .
     (? t:time . f t) /\
     (!t. f t ==> (?n. Next f(t,t + n) /\ r(t,t + n))) ==>
     !m . f (Temp_Abs f m)"),
    REPEAT GEN_TAC
    THEN STRIP_GOAL_THEN ((MAP_EVERY ASSUME_TAC) o CONJUNCTS)
    THEN INDUCT_TAC
    THEN REWRITE_TAC [Temp_Abs]
    THENL [
        IMP_RES_TAC FIRST_LEMMA2;
        IMP_RES_TAC NEXT_LEMMA2
        ]
   );;

let ONE_OR_THE_OTHER = TAC_PROOF
   (([],
    "! a b . ~(a = b) ==> (a < b \/ b < a)"),
    INDUCT_TAC
    THEN INDUCT_TAC
    THEN ASM_REWRITE_TAC [SYM_RULE NOT_SUC;LESS_0;
                          INV_SUC_EQ;LESS_MONO_EQ]
   );;

let First_UNIQUE = prove_thm
   (`First_UNIQUE`,
    "! g t1 t2 .
      (First g t1 /\ First g t2) ==> (t1 = t2)",
    PURE_ONCE_REWRITE_TAC [First]
    THEN REPEAT STRIP_TAC
    THEN ASM_CASES_TAC "t1 = t2"
    THEN ASM_REWRITE_TAC []
    THEN IMP_RES_TAC ONE_OR_THE_OTHER
    THENL [ % 1 %
        ASSUM_LIST (\asl. ASSUME_TAC (
            SPEC "t1:time" (el 4 asl)))
    ; % 2 %
        ASSUM_LIST (\asl. ASSUME_TAC (
            SPEC "t2:time" (el 4 asl)))
    ]
    THEN RES_TAC
   );;
```

```
let Next_UNIQUE = prove_thm
   (`Next_UNIQUE`,
    "!t t1 t2 f .
      (Next f (t,t1) /\ Next f (t,t2)) ==> (t1 = t2)",
    PURE_ONCE_REWRITE_TAC [Next]
    THEN REPEAT STRIP_TAC
    THEN ASM_CASES_TAC "t1 = t2"
    THEN ASM_REWRITE_TAC []
    THEN IMP_RES_TAC ONE_OR_THE_OTHER
    THENL [ % 1 %
        ASSUM_LIST (\asl. ASSUME_TAC (
            SPEC "t1:time" (el 4 asl)))
    ; % 2 %
        ASSUM_LIST (\asl. ASSUME_TAC (
            SPEC "t2:time" (el 7 asl)))
    ]
    THEN RES_TAC
   );;



let NEXT_CHOOSE_LEMMA = TAC_PROOF
   (([],
    "!f u n .
      (?t. f t) /\
      (!t. f t ==> (?n. Next f(t,t + n) /\ r(t,t + n))) /\
      Next f(Temp_Abs f u,(Temp_Abs f u) + n) ==>
      ((@t. Next f(Temp_Abs f u,t)) = ((Temp_Abs f u) + n))"),
    REPEAT GEN_TAC
    THEN STRIP_GOAL_THEN ((MAP_EVERY ASSUME_TAC) o CONJUNCTS)
    THEN MATCH_MP_TAC
        (SPECL ["Temp_Abs f u";
                "(@t. Next f(Temp_Abs f u,t))";
                "(Temp_Abs f u) + n";
                "f:num->bool"] Next_UNIQUE)
    THEN ASM_REWRITE_TAC []
    THEN CONV_TAC SELECT_CONV
    THEN IMP_RES_TAC ALL_F_Temp_Abs
    THEN IMP_RES_TAC
        (SPECL ["f:num->bool";
                "u:time"] NEXT_LEMMA1)
    THEN ASSUM_LIST (\asl. ACCEPT_TAC
       (REWRITE_RULE [el 3 asl] (el 1 asl)))
   );;


let INF_Temp_Abs = prove_thm
   (`INF_Temp_Abs`,
    "! f r.
     (? t:time . f t) /\
     (! t:time . f t ==> ? n . Next f (t,t+n) /\ r(t,t+n)) ==>
     ! u . r (Temp_Abs f u, Temp_Abs f (u+1))",
    REPEAT GEN_TAC
    THEN STRIP_GOAL_THEN ((MAP_EVERY ASSUME_TAC) o CONJUNCTS)
    THEN REPEAT GEN_TAC
```

```
    THEN IMP_RES_TAC ALL_F_Temp_Abs
    THEN PURE_ONCE_REWRITE_TAC [SYM_RULE ADD1]
    THEN ASM_REWRITE_TAC [Temp_Abs]
    THEN ASSUM_LIST (\asl . STRIP_ASSUME_TAC (
        REWRITE_RULE [el 1 asl]
          (SPEC "Temp_Abs f u" (el 3 asl))))
    THEN IMP_RES_TAC NEXT_CHOOSE_LEMMA
    THEN ASM_REWRITE_TAC []
   );;


let Temp_Abs_DEGENERATE = prove_thm
   (`Temp_Abs_DEGENERATE`,
    "Temp_Abs (\t:time.T) = I",
    CONV_TAC (DEPTH_CONV FUN_EQ_CONV)
    THEN INDUCT_TAC
    THEN ASM_REWRITE_TAC [Temp_Abs;I_THM]
    THENL [ % 1 %
        MATCH_MP_TAC
                (SPECL ["\t':time.T";
                        "@t. First (\t':time.T) t";
                        "0";
                        ] First_UNIQUE)
        THEN CONV_TAC (DEPTH_CONV SELECT_CONV)
        THEN REWRITE_TAC [First;NOT_LESS_0]
        THEN EXISTS_TAC "0"
    ; % 2 %
        MATCH_MP_TAC
                (SPECL ["n:time";
                        "@t. Next (\t:time.T)(n,t)";
                        "SUC n";
                        "\t':time.T"] Next_UNIQUE)
        THEN CONV_TAC (DEPTH_CONV SELECT_CONV)
        THEN REWRITE_TAC [Next;LESS_SUC_REFL]
        THEN CONJ_TAC
        THENL [ % 2.1 %
            EXISTS_TAC "SUC n"
        ; % 2.2 %
            ALL_TAC
        ]
    ]
    THEN REWRITE_TAC [LESS_SUC_REFL;LESS_LESS_SUC;
                      NOT_LESS_0]
   );;
```

3.1.3 Asynchronous Interpreters

This section presents the ML code that creates the theory gen_I.th.

```
%----------------------------------------------------------------

   File:         mk_I.ml

   Author:       (c) P. J. Windley 1990

   Date:         09 JAN 90

   Modified:     19 FEB 90

   Description:

   Defines a generic interpreter used in subsequent specifications.
   The interpreter is proven to be correct under certain obligations.
   The interpreter in this file is synchronous.

   2/13/90 -- Modified to take external lines into account.

   2/19/90 -- Modified to make asynchronous.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                  `/muztag/home/windley/hol/Library/assoc/`;
                                 ]);;

system `/bin/rm gen_I.th`;;

new_theory `gen_I`;;

map loadf [`abstract`];;

map load_parent [`time_abs`];;

new_type_abbrev(`time`,":num");;

new_type_abbrev(`time'`,":num");;

%----------------------------------------------------------------
 Generic specification
----------------------------------------------------------------%

let cpu_abs = new_abstract_representation
    [
     (`inst_list`,":(*key#(*state->*env->*state))list")
      ;
     (`key`,":*key->num")
      ;
     (`select`,":*state->*env->*key")
```

```
     ;
    (`substate`,":*state'->*state")
     ;
    (`subenv`,":*env'->*env")
     ;
    (`Impl`,":(time'->*state')->(time'->*env')->bool")
     ;
    (`count`,":*state'->*env'->*key'")
     ;
    (`start`,":*key'")
     ;
   ];;

make_inst_thms cpu_abs;;

let I_rep_ty = abstract_type `gen_I` `key`;;

let INTERP_def = new_definition
   (`INTERP`,
    "! (rep:^I_rep_ty) (s:time->*state) (e:time->*env) .
     INTERP rep s e =
       !t:time.
        let n = (key rep (select rep (s t) (e t))) in (
        s(t+1) = (SND (EL n (inst_list rep))) (s t) (e t))"
   );;

let INTERP_DEF_EXPANDED =
    BETA_RULE (
    EXPAND_LET_RULE INTERP_def);;

let inst_correct_def = new_definition
   (`INST_CORRECT`,
    "! inst:(*key#(*state->*env->*state))
       (s':time'->*state')
       (e':time'->*env') .
     INST_CORRECT rep s' e' inst =
        (Impl (rep:^I_rep_ty) s' e') ==>
         (!t:time'.
            let s = (\t. (substate rep (s' t))) in
            let e = (\t. (subenv rep (e' t))) in
            let f = (\t. (count rep (s' t) (e' t) = (start rep))) in (
            (select rep (s t) (e t) = (FST inst)) /\
            (count rep (s' t) (e' t) = (start rep)) ==>
              ? c.
              Next f (t,t+c) /\
              ((SND inst) (s t) (e t) = (s (t + c)))))"
   );;

let INST_CORRECT_EXPANDED =
    BETA_RULE (
    EXPAND_LET_RULE inst_correct_def);;

new_theory_obligations
   [
    "EVERY (INST_CORRECT (rep:^I_rep_ty)
```

```
                        (s':time'->*state')
                        (e':time'->*env'))
         (inst_list rep)"
    ;
    "!k:*key. (key (rep:^I_rep_ty) k) < (LENGTH (inst_list rep))"
    ;
    "!k:*key . k = (FST (EL (key (rep:^I_rep_ty) k) (inst_list rep)))"
    ;
   ];;

let IMPL_NEXTSTATE_LEMMA = TAC_PROOF
   (([],
    "let s = (\t:time .(substate rep (s' t))) and
         e = (\t:time .(subenv rep (e' t))) and
         f = (\t. (count rep (s' t) (e' t) = (start rep))) in (
     (Impl (rep:^I_rep_ty)) s' e' ==>
       (!t:time'.
        (count rep (s' t) (e' t) = (start rep)) ==>
          ? c .
          Next f (t,t+c) /\
          ((substate rep (s' (t + c))) =
           (SND (EL (key rep (select rep (s t) (e t)))
                    (inst_list rep))) (s t) (e t))))"),
    EXPAND_LET_TAC
    THEN BETA_TAC
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM_LIST (\asl .
        let asl' =
             map (PURE_REWRITE_RULE [EVERY_EL;INST_CORRECT_EXPANDED]) asl in
        MAP_EVERY ASSUME_TAC
         (map
           (\thm.
            (SPEC "(key (rep:^I_rep_ty)
                    (select rep
                     (substate rep(s' t))
                      (subenv rep (e' t))))" thm) ?
            (SPEC "(select (rep:^I_rep_ty)
                    (substate rep(s' t))
                     (subenv rep (e' t)))" thm) ?
            thm) asl'))
    THEN RES_TAC
    THEN POP_ASSUM (\thm. ASSUME_TAC (REWRITE_RULE [] (SPEC "t:time'" thm)))
    THEN RES_TAC
    THEN FIRST_ASSUM (
          MATCH_ACCEPT_TAC o
          (CONV_RULE (ONCE_DEPTH_CONV (RAND_CONV SYM_CONV))))
   );;

let IMPL_NEXTSTATE_LEMMA_EXPANDED =
    BETA_RULE (
    EXPAND_LET_RULE IMPL_NEXTSTATE_LEMMA);;

let IMPL_I_CORRECT = prove_thm
   (`IMPL_I_CORRECT`,
    "let s = (\t:time .(substate rep (s' t))) and
```

```
        e = (\t:time .(subenv rep (e' t))) and
        f = (\t:time .(count rep (s' t) (e' t) = (start rep))) in
    let abs = (Temp_Abs f) in (
    (Impl (rep:^I_rep_ty)) s' e' /\
    (?t. f t) ==>
    (INTERP rep) (s o abs) (e o abs))",
   EXPAND_LET_TAC
   THEN BETA_TAC
   THEN REPEAT GEN_TAC
   THEN PURE_REWRITE_TAC [INTERP_DEF_EXPANDED;o_DEF]
   THEN STRIP_GOAL_THEN ((MAP_EVERY ASSUME_TAC) o CONJUNCTS)
   THEN BETA_TAC
   THEN MATCH_MP_TAC (
       BETA_RULE (
       REWRITE_RULE [UNCURRY_DEF] (
       SPECL ["(\t:time . (count rep (s' t) (e' t) =
                          (start (rep:^I_rep_ty))))";
              "(\(t1:time,t2:time) .
               substate rep (s' t2) =
               (SND
                (EL (key (rep:^I_rep_ty)
                     (select rep
                      (substate rep (s' t1))
                       (subenv rep (e' t1))))
                    (inst_list rep)))
               (substate rep (s' t1))
               (subenv rep (e' t1)))"
       ] INF_Temp_Abs)))
   THEN CONJ_TAC
   THEN (FIRST_ASSUM MATCH_ACCEPT_TAC ORELSE ALL_TAC)
   THEN REPEAT STRIP_TAC
   THEN IMP_RES_TAC IMPL_NEXTSTATE_LEMMA_EXPANDED
  );;


close_theory();;
```

3.2 The Word Representation

This section presents the ML code that creates the theory `aux_def.th`.

```
%----------------------------------------------------------------

   File:        def_aux.ml

   Description:

   Defines generic functions used in subsequent specifications.

   Author:      (c) P. J. Windley 1989
   Date:        29 DEC 89

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

system `/bin/rm aux_def.th`;;

new_theory `aux_def`;;

loadf `abstract`;;

new_type_abbrev(`time`,":num");;

let abs_rep = new_abstract_representation [
% ALU functions %
       % addition without carry %
       (`add`, ":(*wordn # *wordn -> *wordn)                  ")
        ;
       % addition with carry %
       (`addc`, ":(*wordn # *wordn # bool -> *wordn)         ")
        ;
       % carry predicate for add %
       (`addp`, ":(*wordn # *wordn # *wordn) -> bool         ")
        ;
       % predicate carry for addc %
       (`addcp`, ":(*wordn # *wordn # *wordn) -> bool        ")
        ;
       % overflow predicate for add %
       (`aovfl`, ":(*wordn # *wordn # *wordn) -> bool        ")
        ;
       % increment %
       (`inc`, ":(*wordn -> *wordn)                          ")
        ;
       % subtract without carry %
       (`sub`, ":(*wordn # *wordn -> *wordn)                  ")
        ;
       % subtract with carry %
       (`subc`, ":(*wordn # *wordn # bool) -> *wordn         ")
```

```
          ;
        % carry predicate for sub %
        (`subp`, ":(*wordn # *wordn # *wordn) -> bool         ")
          ;
        % overflow predicate for sub %
        (`sovfl`, ":(*wordn # *wordn # *wordn) -> bool        ")
          ;
        % decrement %
        (`dec`, ":(*wordn -> *wordn)                          ")
          ;
        % bitwise and %
        (`band`, ":(*wordn # *wordn -> *wordn)                ")
          ;
        % bitwise xor %
        (`bxor`, ":(*wordn # *wordn -> *wordn)                ")
          ;
        % bitwise or %
        (`bor`, ":(*wordn # *wordn -> *wordn)                 ")
          ;
        % bitwise not %
        (`bnot`, ":(*wordn -> *wordn)                         ")
          ;
% Test functions %
        % negative? %
        (`negp`, ":(*wordn -> bool)                           ")
          ;
        % zero? %
        (`zerop`, ":(*wordn -> bool)                          ")
          ;
% SHIFTER functions %
        % shift left %
        (`shl`, ":(*wordn -> *wordn)                          ")
          ;
        % shift right %
        (`shr`, ":(*wordn -> *wordn)                          ")
          ;
        % arithmetic shift right %
        (`asr`, ":(*wordn -> *wordn)                          ")
          ;
% Bit functions %
        % most significant bit %
        (`msb`, ":(*wordn -> bool)                            ")
          ;
        % least significant bit %
        (`lsb`, ":(*wordn -> bool)                            ")
          ;
% Coercion functions %
        % numeric vaule of n-bit word %
        (`val`, ":(*wordn -> num)                             ")
          ;
        % wordn representation of number %
        (`wordn`, ":(num -> *wordn)                           ")
          ;
        % address representation of a word %
        (`address`, ":(*wordn -> *address)                    ")
```

```
        ;
% Subranging functions %
       % opcode portion of word %
       (`opcode`, ":(*wordn->(bool#bool#bool#bool#bool#bool))")
        ;
       % destination portion of word %
       (`dest`, ":(*wordn -> *reg_len)                        ")
        ;
       % source A portion of word %
       (`srca`, ":(*wordn -> *reg_len)                        ")
        ;
       % source B portion of word %
       (`srcb`, ":(*wordn -> *reg_len)                        ")
        ;
       % value of reg_len  %
       (`reg_len`, ":(*reg_len -> num)                        ")
        ;
       % immediate portion of word %
       (`imm`, ":(*wordn -> *wordn)                           ")
        ;
% Subranging functions for the Program Status Word %
       % interrupt enable bit in word %
       (`get_ie`, ":(*wordn -> bool)                          ")
        ;
       % supervisory mode bit in word %
       (`get_sm`, ":(*wordn -> bool)                          ")
        ;
       % carry bit in word %
       (`get_cf`, ":(*wordn -> bool)                          ")
        ;
       % overflow bit in word %
       (`get_vf`, ":(*wordn -> bool)                          ")
        ;
       % zero bit in word %
       (`get_zf`, ":(*wordn -> bool)                          ")
        ;
       % neg bit in word %
       (`get_nf`, ":(*wordn -> bool)                          ")
        ;
       % create psw %
       (`mk_psw`, ":((bool#bool#bool#bool#bool#bool) -> *wordn)")
        ;
% Memory functions %
       % fetch a word from memory %
       (`fetch`, ":(*memory # *address) -> *wordn             ")
        ;
       % store a word in memory %
       (`store`, ":(*memory # *address # *wordn) -> *memory  ")
        ;
       % transmute memory %
       (`trans`, ":*memory -> *memory                         ")
        ;
% Interrupt instructions %
       (`int_trans`, ":*wordn -> *wordn                       ")
        ;
```

```
        (`int_fetch`, ":*wordn -> *wordn                        ")
         ;
        ];;

let rep_ty = abstract_type `aux_def` `opcode`;;

close_theory();;
```

3.3 Auxiliary Files

The section presents several auxiliary theories that are used throughout the specification and verification of *AVM-1* .

3.3.1 Auxiliary Theorems

The section presents the ML code that creates the theory `aux_thms.th`.

```
%----------------------------------------------------------------

   File:         mk_aux.ml

   Author:       (c) P. J. Windley 1990

   Date:         15 JAN 90

   Modified:

   Description:

   Prove auxilliary theorems used in subsequent proofs.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`;`assoc/`]));;


system `/bin/rm aux_thms.th`;;

new_theory `aux_thms`;;

loadf `tuple`;;

%----------------------------------------------------------------
 Auxilliary list definitions and theorems
----------------------------------------------------------------%

let SET_EL_DEF = new_prim_rec_definition
   (`SET_EL_DEF`,
    "(SET_EL 0 (lst:(*)list) x = (CONS x (TL lst))) /\
     (SET_EL (SUC n) lst x = (CONS (HD lst) (SET_EL n (TL lst) x)))"
   );;
```

```
let SET_EL = prove_thm
   (`SET_EL`,
    "! h t x .
     (SET_EL 0 (CONS h t) x = (CONS x t)) /\
     (SET_EL (SUC n) (CONS h t) x = (CONS h (SET_EL n t x)))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC [SET_EL_DEF;HD;TL]
   );;

let EL_SET_EL = prove_thm
   (`EL_SET_EL`,
    "! x n lst . EL n (SET_EL n lst x) = x",
    GEN_TAC
    THEN INDUCT_TAC
    THEN REWRITE_TAC [SET_EL_DEF; EL;CONS;TL;HD]
    THEN LIST_INDUCT_TAC
    THENL [
        POP_ASSUM (\x. ASSUME_TAC (SPEC "TL[]:(*)list" x))
        ;
        ALL_TAC
       ]
    THEN ASM_REWRITE_TAC [TL]
   );;

%-----------------------------------------------------------------
 Auxilliary boolean definitions and theorems
-----------------------------------------------------------------%
let xor = new_infix_definition
   (`xor`,
    "! a b . xor$ a b = (a /\ ~b) \/ (~a /\ b)"
   );;


%-----------------------------------------------------------------
 Define addition of a number with a bt6 value
-----------------------------------------------------------------%
let add_bt6 = new_definition
   (`add_bt6`,
    "! x y .
     add_bt6 x y =
        bt6_ival ((bt6_val x) + y)"
   );;

let OFFSET = "4";;

let PLUS_4_LEMMA = TAC_PROOF
   (([],
    "!x.x+^OFFSET = (SUC (SUC (SUC (SUC x))))"),
    CONV_TAC (TOP_DEPTH_CONV num_CONV)
    THEN REWRITE_TAC [ADD_CLAUSES]
   );;

%-----------------------------------------------------------------
 Some other nice conversions
```

```
----------------------------------------------------------------%
let is_SND_term t =
  if is_comb t then
    fst(dest_const(fst(strip_comb t))) = `SND`
  else
    false;;

let SND_CONV t =
   if is_SND_term t then
      let op,pr = dest_comb t in
      let op,[t1;t2] = strip_comb pr in
      SPECL [t1;t2] (
         INST_TYPE [((type_of t1),":*");
                    ((type_of t2),":**")] SND)
   else
      failwith `SND_CONV`;;

let inv_num_CONV n = (
   let x,y = dest_comb n in
   let y_inc = int_to_term ((term_to_int y) + 1) in
   if not(x = "SUC") then fail else
   SYM_RULE (num_CONV y_inc))
   ? failwith `inv_num_CONV`;;


%----------------------------------------------------------------
 Prove that the table lookup doesn't end up at the beginning of ROM
    OFFSET_NOT_BEGINNING = |- !b. ~(add_bt6(F,SND b)4 = F,F,F,F,F,F)
    Run time: 1110.9s
    Intermediate theorems generated: 32451
----------------------------------------------------------------%
let OFFSET_NOT_BEGINNING = TAC_PROOF
   (([],
    "! b:bt6.~(add_bt6(F,SND b)^OFFSET = F,F,F,F,F,F)"),
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC_ALL SIX_TUPLE_VALUE_LEMMA)
    THEN PURE_ONCE_REWRITE_TAC [add_bt6]
    THEN CONV_TAC (ONCE_DEPTH_CONV SND_CONV)
    THEN CONV_TAC (ONCE_DEPTH_CONV bt6_val_CONV)
    THEN PURE_ONCE_REWRITE_TAC [PLUS_4_LEMMA]
    THEN CONV_TAC (TOP_DEPTH_CONV inv_num_CONV)
    THEN CONV_TAC (ONCE_DEPTH_CONV bt6_ival_CONV)
    THEN REWRITE_TAC [PAIR_EQ]
);;

save_thm(`OFFSET_NOT_BEGINNING`,OFFSET_NOT_BEGINNING);;


close_theory();;
```

3.3.2 The Jump Condition

The section presents the ML code that creates the theory `jump_def.th`.

```
%---------------------------------------------------------------

   File:         def_jump.ml

   Author:       (c) P. J. Windley 1990

   Date:         9 APR 90

   Modified:

   Description:

   Defines the function used to describe the jump unit in the
   EBM and to describe jump condition selection in the
   other levels.

---------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`]));;

loadf `abstract`;;

system `/bin/rm jump_def.th`;;

new_theory `jump_def`;;

map new_parent [`aux_def`; `aux_thms`];;

let rep_ty = abstract_type `aux_def` `get_sm`;;


%---------------------------------------------------------------
This definition is used in the jump instruction.
---------------------------------------------------------------%
let JUMP_COND = new_definition
   (`JUMP_COND`,
    "! d . JUMP_COND (rep:^rep_ty) d psw =
        let cf = (get_cf rep psw) and
            vf = (get_vf rep psw) and
            nf = (get_nf rep psw) and
            zf = (get_zf rep psw) in (
```

```
      (d =  0) => cf                   | % carry %
                                         % higher or same (unsigned) %
      (d =  1) => ~ cf                 | % no carry %
                                         % lower (unsigned) %
      (d =  2) => vf                   | % overflow %
      (d =  3) => ~ vf                 | % no overflow %
      (d =  4) => nf                   | % negative %
      (d =  5) => ~ nf                 | % positive %
      (d =  6) => zf                   | % equal %
      (d =  7) => ~ zf                 | % not equal %
      (d =  8) => (~cf \/ zf)          | % lower or same (unsigned) %
      (d =  9) => ~(~cf \/ zf)         | % higher (unsigned) %
      (d = 10) => (nf xor vf)          | % less than (signed) %
      (d = 11) => ~(nf xor vf)         | % greater or equal (signed) %
      (d = 12) => ~((nf xor vf) \/ zf) | % greater than (signed) %
      (d = 13) => ((nf xor vf) \/ zf)    | % greater or equal (signed) %
                  T                   )" % always %
   );;

close_theory();;
```

3.3.3 The Register File

The section presents the ML code that creates the theory `regs_def.th`.

```
%----------------------------------------------------------------

   File:         def_regs.ml

   Author:       (c) P. J. Windley 1990

   Date:         18 JAN 90

   Modified:     10 FEB 90

   Description:

   Defines functions for selecting registers in the register file.
   These functions are used in many of the specifications.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`]));;


loadf `abstract`;;

system `/bin/rm regs_def.th`;;

new_theory `regs_def`;;

map new_parent [`aux_def`;`aux_thms`];;

let rep_ty = abstract_type `aux_def` `get_sm`;;

%----------------------------------------------------------------
 Special names for some of the registers in register file.
 No magic numbers here!
----------------------------------------------------------------%

let zero_reg = new_definition (`zero_reg`,"zero_reg = 0");;

let ZERO_REG = new_definition
   (`ZERO_REG`,
    "! reg_list:(*wordn)list . ZERO_REG reg_list = (EL zero_reg reg_list)"
   );;
```

```
%---------------------------------------------------------------
 Supervisor registers are from 1-7
---------------------------------------------------------------%
let IS_SUP_REG = new_definition
   (`IS_SUP_REG`,
    "!n. IS_SUP_REG n = (0 < n) /\ (n < 8)"
   );;

let ssp_reg = new_definition (`ssp_reg`,"ssp_reg = 1");;

let SSP_REG = new_definition
   (`SSP_REG`,
    "! reg_list:(*wordn)list . SSP_REG reg_list = (EL ssp_reg reg_list)"
   );;



%---------------------------------------------------------------
 UPDATE REGISTER LIST
---------------------------------------------------------------%

let UPDATE_REG = new_definition
   (`UPDATE_REG`,
    "! (rep:^rep_ty) psw n (reg_list:(*wordn)list) value .
     UPDATE_REG rep psw n reg_list value =
         let sm   = (get_sm rep psw) in
         (n = zero_reg)         => reg_list |
         (IS_SUP_REG n /\ ~sm)  => reg_list |
                                   (SET_EL n reg_list value)"
   );;

close_theory();;
```

3.4 The Electronic Block Model

This section presents the theories that define the electronic block model.

3.4.1 A 16 Input Multiplexor

The section presents the ML code that creates the theory `mux16_def.th`.

```
%----------------------------------------------------------------

   File:         def_mux16.ml

   Author:       (c) P. J. Windley 1989, 1990

   Date:         29 DEC 89

   Modified:     13 JAN 90

   Description:

   Defines a 16 input MUX used in subsequent specifications.


----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                 `/muztag/home/windley/hol/ml/`;
                                ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             []));;


system `/bin/rm mux16_def.th`;;

new_theory `mux16_def`;;

let mux_16_def = new_definition
   (`MUX_16_DEF`,
    "! (b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15:*)
       select result .
     MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
            select result =
     ((result =
      (select = (F,F,F,F)) =>  b0 |
      (select = (F,F,F,T)) =>  b1 |
      (select = (F,F,T,F)) =>  b2 |
      (select = (F,F,T,T)) =>  b3 |
```

```
      (select = (F,T,F,F)) =>  b4 |
      (select = (F,T,F,T)) =>  b5 |
      (select = (F,T,T,F)) =>  b6 |
      (select = (F,T,T,T)) =>  b7 |
      (select = (T,F,F,F)) =>  b8 |
      (select = (T,F,F,T)) =>  b9 |
      (select = (T,F,T,F)) => b10 |
      (select = (T,F,T,T)) => b11 |
      (select = (T,T,F,F)) => b12 |
      (select = (T,T,F,T)) => b13 |
      (select = (T,T,T,F)) => b14 |
                              b15 ))"
   );;

let mux_16_application = prove_thm
   (`MUX_16`,
    "! (b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 r:*) .
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (F,F,F,F) r) = (r = b0)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (F,F,F,T) r) = (r = b1)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (F,F,T,F) r) = (r = b2)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (F,F,T,T) r) = (r = b3)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (F,T,F,F) r) = (r = b4)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (F,T,F,T) r) = (r = b5)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (F,T,T,F) r) = (r = b6)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (F,T,T,T) r) = (r = b7)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (T,F,F,F) r) = (r = b8)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (T,F,F,T) r) = (r = b9)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (T,F,T,F) r) = (r = b10)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (T,F,T,T) r) = (r = b11)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (T,T,F,F) r) = (r = b12)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (T,T,F,T) r) = (r = b13)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (T,T,T,F) r) = (r = b14)) /\
       ((MUX_16 (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                (T,T,T,T) r) = (r = b15))",
    REWRITE_TAC [mux_16_def;PAIR_EQ]
   );;

close_theory();;
```

3.4.2 A Generic ALU

The section presents the ML code that creates the theory `gen_alu.th`.

```
%-----------------------------------------------------------------

   File:         mk_gen_alu.ml

   Author:       (c) P. J. Windley 1989, 1990

   Date:         29 DEC 89

   Modified:     13 JAN 90

   Description:

   Defines a generic ALU used in subsequent specifications. The
   theory contains a generic proof.

-----------------------------------------------------------------%


set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                   `/muztag/home/windley/hol/ml/`;
                                  ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`]));;


loadf `abstract`;;

system `/bin/rm gen_alu.th`;;

new_theory `gen_alu`;;

map load_parent [`tuple`;`mux16_def`];;

%-----------------------------------------------------------------
 Generic specification
-----------------------------------------------------------------%

let alu_abs = new_abstract_representation
    [
     (`func0`,":*inputs->*output->*flags->bool")
      ;
     (`func1`,":*inputs->*output->*flags->bool")
      ;
     (`func2`,":*inputs->*output->*flags->bool")
      ;
```

```
(‘func3‘,":*inputs->*output->*flags->bool")
 ;
(‘func4‘,":*inputs->*output->*flags->bool")
 ;
(‘func5‘,":*inputs->*output->*flags->bool")
 ;
(‘func6‘,":*inputs->*output->*flags->bool")
 ;
(‘func7‘,":*inputs->*output->*flags->bool")
 ;
(‘func8‘,":*inputs->*output->*flags->bool")
 ;
(‘func9‘,":*inputs->*output->*flags->bool")
 ;
(‘func10‘,":*inputs->*output->*flags->bool")
 ;
(‘func11‘,":*inputs->*output->*flags->bool")
 ;
(‘func12‘,":*inputs->*output->*flags->bool")
 ;
(‘func13‘,":*inputs->*output->*flags->bool")
 ;
(‘func14‘,":*inputs->*output->*flags->bool")
 ;
(‘func15‘,":*inputs->*output->*flags->bool")
 ;
(‘module0‘,":*inputs->*output->*flags->bool")
 ;
(‘module1‘,":*inputs->*output->*flags->bool")
 ;
(‘module2‘,":*inputs->*output->*flags->bool")
 ;
(‘module3‘,":*inputs->*output->*flags->bool")
 ;
(‘module4‘,":*inputs->*output->*flags->bool")
 ;
(‘module5‘,":*inputs->*output->*flags->bool")
 ;
(‘module6‘,":*inputs->*output->*flags->bool")
 ;
(‘module7‘,":*inputs->*output->*flags->bool")
 ;
(‘module8‘,":*inputs->*output->*flags->bool")
 ;
(‘module9‘,":*inputs->*output->*flags->bool")
 ;
(‘module10‘,":*inputs->*output->*flags->bool")
 ;
(‘module11‘,":*inputs->*output->*flags->bool")
 ;
(‘module12‘,":*inputs->*output->*flags->bool")
 ;
(‘module13‘,":*inputs->*output->*flags->bool")
 ;
(‘module14‘,":*inputs->*output->*flags->bool")
```

```
      ;
     (`module15`,":*inputs->*output->*flags->bool")
    ];;

make_inst_thms alu_abs;;

let alu_rep_ty = abstract_type `gen_alu` `func0`;;

let dummy_op_def = new_definition
   (`DUMMY_OP`,
    "DUMMY_OP = @x:one.F"
   );;

let alu_spec_def = new_definition
   (`ALU_SPEC_DEF`,
    "! (rep:^alu_rep_ty) switch inputs output flags .
     ALU_SPEC rep switch inputs output flags =
     ((switch = (F,F,F,F)) => (
         (func0 rep) inputs output flags) |
      (switch = (F,F,F,T)) => (
         (func1 rep) inputs output flags) |
      (switch = (F,F,T,F)) => (
         (func2 rep) inputs output flags) |
      (switch = (F,F,T,T)) => (
         (func3 rep) inputs output flags) |
      (switch = (F,T,F,F)) => (
         (func4 rep) inputs output flags) |
      (switch = (F,T,F,T)) => (
         (func5 rep) inputs output flags) |
      (switch = (F,T,T,F)) => (
         (func6 rep) inputs output flags) |
      (switch = (F,T,T,T)) => (
         (func7 rep) inputs output flags) |
      (switch = (T,F,F,F)) => (
         (func8 rep) inputs output flags) |
      (switch = (T,F,F,T)) => (
         (func9 rep) inputs output flags) |
      (switch = (T,F,T,F)) => (
         (func10 rep) inputs output flags) |
      (switch = (T,F,T,T)) => (
         (func11 rep) inputs output flags) |
      (switch = (T,T,F,F)) => (
         (func12 rep) inputs output flags) |
      (switch = (T,T,F,T)) => (
         (func13 rep) inputs output flags) |
      (switch = (T,T,T,F)) => (
         (func14 rep) inputs output flags) |
      % default %
         (func15 rep) inputs output flags)"
   );;

let ALU_SPEC = prove_thm
   (`ALU_SPEC`,
    "! (rep:^alu_rep_ty) inputs output flags .
     (ALU_SPEC rep (F,F,F,F) inputs output flags =
```

```
        (func0 rep) inputs output flags) /\
    (ALU_SPEC rep (F,F,F,T) inputs output flags =
        (func1 rep) inputs output flags) /\
    (ALU_SPEC rep (F,F,T,F) inputs output flags =
        (func2 rep) inputs output flags) /\
    (ALU_SPEC rep (F,F,T,T) inputs output flags =
        (func3 rep) inputs output flags) /\
    (ALU_SPEC rep (F,T,F,F) inputs output flags =
        (func4 rep) inputs output flags) /\
    (ALU_SPEC rep (F,T,F,T) inputs output flags =
        (func5 rep) inputs output flags) /\
    (ALU_SPEC rep (F,T,T,F) inputs output flags =
        (func6 rep) inputs output flags) /\
    (ALU_SPEC rep (F,T,T,T) inputs output flags =
        (func7 rep) inputs output flags) /\
    (ALU_SPEC rep (T,F,F,F) inputs output flags =
        (func8 rep) inputs output flags) /\
    (ALU_SPEC rep (T,F,F,T) inputs output flags =
        (func9 rep) inputs output flags) /\
    (ALU_SPEC rep (T,F,T,F) inputs output flags =
        (func10 rep) inputs output flags) /\
    (ALU_SPEC rep (T,F,T,T) inputs output flags =
        (func11 rep) inputs output flags) /\
    (ALU_SPEC rep (T,T,F,F) inputs output flags =
        (func12 rep) inputs output flags) /\
    (ALU_SPEC rep (T,T,F,T) inputs output flags =
        (func13 rep) inputs output flags) /\
    (ALU_SPEC rep (T,T,T,F) inputs output flags =
        (func14 rep) inputs output flags) /\
    (ALU_SPEC rep (T,T,T,T) inputs output flags =
        (func15 rep) inputs output flags)",
   REWRITE_TAC [alu_spec_def;PAIR_EQ]
  );;

%----------------------------------------------------------------
 Generic implementation
----------------------------------------------------------------%

let alu_imp_def = new_definition
   (`ALU_IMP`,
    "! (rep:^alu_rep_ty) switch inputs output flags .
     ALU_IMP rep switch inputs output flags =
     ? r0 f0 r1 f1 r2 f2 r3 f3 r4 f4 r5 f5 r6 f6 r7 f7
       r8 f8 r9 f9 r10 f10 r11 f11 r12 f12 r13 f13 r14 f14 r15 f15 .
     (((module0 rep) inputs r0 f0) /\
      ((module1 rep) inputs r1 f1) /\
      ((module2 rep) inputs r2 f2) /\
      ((module3 rep) inputs r3 f3) /\
      ((module4 rep) inputs r4 f4) /\
      ((module5 rep) inputs r5 f5) /\
      ((module6 rep) inputs r6 f6) /\
      ((module7 rep) inputs r7 f7) /\
      ((module8 rep) inputs r8 f8) /\
      ((module9 rep) inputs r9 f9) /\
      ((module10 rep) inputs r10 f10) /\
```

```
      ((module11 rep) inputs r11 f11) /\
      ((module12 rep) inputs r12 f12) /\
      ((module13 rep) inputs r13 f13) /\
      ((module14 rep) inputs r14 f14) /\
      ((module15 rep) inputs r15 f15) /\
      (MUX_16 (r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15)
             switch output) /\
      (MUX_16 (f0,f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15)
             switch flags))"
   );;

new_theory_obligations
   [
    "!inputs output flags.
      (module0 (rep:^alu_rep_ty) inputs output flags)==>
      (func0 rep inputs output flags)";
    "!inputs output flags.
      (module1 (rep:^alu_rep_ty) inputs output flags)==>
      (func1 rep inputs output flags)";
    "!inputs output flags.
      (module2 (rep:^alu_rep_ty) inputs output flags)==>
      (func2 rep inputs output flags)";
    "!inputs output flags.
      (module3 (rep:^alu_rep_ty) inputs output flags)==>
      (func3 rep inputs output flags)";
    "!inputs output flags.
      (module4 (rep:^alu_rep_ty) inputs output flags)==>
      (func4 rep inputs output flags)";
    "!inputs output flags.
      (module5 (rep:^alu_rep_ty) inputs output flags)==>
      (func5 rep inputs output flags)";
    "!inputs output flags.
      (module6 (rep:^alu_rep_ty) inputs output flags)==>
      (func6 rep inputs output flags)";
    "!inputs output flags.
      (module7 (rep:^alu_rep_ty) inputs output flags)==>
      (func7 rep inputs output flags)";
    "!inputs output flags.
      (module8 (rep:^alu_rep_ty) inputs output flags)==>
      (func8 rep inputs output flags)";
    "!inputs output flags.
      (module9 (rep:^alu_rep_ty) inputs output flags)==>
      (func9 rep inputs output flags)";
    "!inputs output flags.
      (module10 (rep:^alu_rep_ty) inputs output flags)==>
      (func10 rep inputs output flags)";
    "!inputs output flags.
      (module11 (rep:^alu_rep_ty) inputs output flags)==>
      (func11 rep inputs output flags)";
    "!inputs output flags.
      (module12 (rep:^alu_rep_ty) inputs output flags)==>
      (func12 rep inputs output flags)";
    "!inputs output flags.
      (module13 (rep:^alu_rep_ty) inputs output flags)==>
      (func13 rep inputs output flags)";
```

```
    "!inputs output flags.
      (module14 (rep:^alu_rep_ty) inputs output flags)==>
      (func14 rep inputs output flags)";
    "!inputs output flags.
      (module15 (rep:^alu_rep_ty) inputs output flags)==>
      (func15 rep inputs output flags)";
    ];;

%----------------------------------------------------------------
................ |- !rep rep switch in_A in_B cin output neg zero
                     ovfl carry.
                     ALU_IMP
                     rep
                     switch
                     (in_A,in_B,cin)
                     output
                     (neg,zero,ovfl,carry) ==>
                     ALU_SPEC
                     rep
                     switch
                     (in_A,in_B,cin)
                     output
                     (neg,zero,ovfl,carry)
Run time: 1081.2s
Intermediate theorems generated: 67847
----------------------------------------------------------------%

prove_thm
   (`ALU_CORRECT`,
    "!switch inputs output flags .
      ALU_IMP rep switch
            inputs output flags==>
      ALU_SPEC rep switch
            inputs output flags",
     REPEAT GEN_TAC
     THEN ONCE_REWRITE_TAC [alu_imp_def]
     THEN STRUCT_CASES_TAC
         (SPEC "switch:bool#bool#bool#bool" FOUR_TUPLE_VALUE_LEMMA)
     THEN ONCE_REWRITE_TAC [ALU_SPEC; MUX_16]
     THEN REPEAT STRIP_TAC
     THEN RES_TAC
     THEN ASM_REWRITE_TAC []
   );;

close_theory();;
```

3.4.3 The Arithmetic Logic Unit

The section presents the ML code that creates the theory `alu_def.th`.

```
%----------------------------------------------------------------

   File:          def_alu.ml

   Author:        (c) P. J. Windley 1989, 1990

   Date:          29 DEC 89

   Modified:      13 JAN 90

   Description:

   Defines a ALU used in subsequent specifications using
   generic operators from the auxilliary definitions theory
   and a generic ALU from the theory of generic alu's.


----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`]));;

system `/bin/rm alu_def.th`;;

new_theory `alu_def`;;

loadf `abstract`;;

map new_parent [`aux_def`;`gen_alu`];;

let rep_ty = abstract_type `aux_def` `opcode`;;

let add_without_carry_def = new_definition
   (`ADD_WITHOUT_CARRY`,
    "! (rep:^rep_ty) in_A in_B (cin:bool) out neg zero ovfl carry .
     ADD_WITHOUT_CARRY rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (add rep) (in_A,in_B) in
            let c = (addp rep)  (in_A,in_B,result) and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = (aovfl rep) (in_A,in_B,result) in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
```

```
                             (ovfl = v) /\ (carry = c))"
   );;

let add_with_carry_def = new_definition
   (`ADD_WITH_CARRY`,
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     ADD_WITH_CARRY rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (addc rep) (in_A,in_B,cin) in
            let c = (addcp rep) (in_A,in_B,result) and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = (aovfl rep) (in_A,in_B,result) in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
                             (ovfl = v) /\ (carry = c))"
   );;

let increment_def = new_definition
   (`INCREMENT`,
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     INCREMENT rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (inc rep) in_A in
            let c = (addp rep) (in_A,(wordn rep) 0,result) and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = F in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
                             (ovfl = v) /\ (carry = c))"
   );;

let sub_without_carry_def = new_definition
   (`SUB_WITHOUT_CARRY`,
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     SUB_WITHOUT_CARRY rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (sub rep) (in_A,in_B) in
            let c = (subp rep)  (in_A,in_B,result) and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = (sovfl rep) (in_A,in_B,result) in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
                             (ovfl = v) /\ (carry = c))"
   );;

let sub_with_carry_def = new_definition
   (`SUB_WITH_CARRY`,
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     SUB_WITH_CARRY rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (subc rep) (in_A,in_B,cin) in
            let c = (subp rep)  (in_A,in_B,result) and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = (sovfl rep) (in_A,in_B,result) in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
                            (ovfl = v) /\ (carry = c))"
   );;
```

```
let decrement_def = new_definition
   (`DECREMENT`,
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     DECREMENT rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (dec rep) in_A in
            let c = (subp rep) (in_A,(wordn rep) 0,result) and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = F in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
                              (ovfl = v) /\ (carry = c))"
   );;

let bitwise_and_def = new_definition
   (`BITWISE_AND`,
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     BITWISE_AND rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (band rep) (in_A,in_B) in
            let c = F and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = F in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
                               (ovfl = v) /\ (carry = c))"
   );;


let bitwise_xor_def = new_definition
   (`BITWISE_XOR`,
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     BITWISE_XOR rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (bxor rep) (in_A,in_B) in
            let c = F and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = F in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
                               (ovfl = v) /\ (carry = c))"
   );;

let bitwise_or_def = new_definition
   (`BITWISE_OR`,
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     BITWISE_OR rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (bor rep) (in_A,in_B) in
            let c = F and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = F in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
                               (ovfl = v) /\ (carry = c))"
   );;

let bitwise_not_def = new_definition
   (`BITWISE_NOT`,
```

```
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     BITWISE_NOT rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            let result = (bnot rep) in_A in
            let c = F and
                n = (negp rep) result and
                z = (zerop rep) result and
                v = F in
            ((out = result) /\ (neg = n) /\ (zero = z) /\
                                (ovfl = v) /\ (carry = c))"
   );;

let alu_noop_def = new_definition
   (`ALU_NOOP`,
    "! (rep:^rep_ty) in_A in_B cin out neg zero ovfl carry .
     ALU_NOOP rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
            ((out = in_A) /\ (neg = ((negp rep) in_A)) /\
                              (zero = ((zerop rep) in_A)) /\
                              (ovfl = F) /\ (carry = F))"
   );;

let dummy_module_def = new_definition
   (`DUMMY_MODULE_DEF`,
    "!(rep:^rep_ty)  (in_A in_B out:*wordn) (cin neg zero ovfl carry:bool) .
     DUMMY_MODULE_DEF rep (in_A,in_B,cin) out (neg,zero,ovfl,carry) = F"
   );;

let alu_spec_def = new_definition
   (`MAC2_ALU_SPEC_DEF`,
    "! (rep:^rep_ty) switch in_A in_B cin out (neg zero ovfl carry:bool) .
     MAC2_ALU_SPEC rep switch (in_A,in_B,cin) out (neg,zero,ovfl,carry) =
        ALU_SPEC (
           (ADD_WITHOUT_CARRY rep),
           (ADD_WITH_CARRY rep),
           (INCREMENT rep),
           (SUB_WITHOUT_CARRY rep),
           (SUB_WITH_CARRY rep),
           (DECREMENT rep),
           (BITWISE_AND rep),
           (BITWISE_XOR rep),
           (BITWISE_OR rep),
           (BITWISE_NOT rep),
           (ALU_NOOP rep),
           (ALU_NOOP rep),
           (ALU_NOOP rep),
           (ALU_NOOP rep),
           (ALU_NOOP rep),
           (ALU_NOOP rep),
           (DUMMY_MODULE_DEF rep),(DUMMY_MODULE_DEF rep),
           (DUMMY_MODULE_DEF rep),(DUMMY_MODULE_DEF rep),
           (DUMMY_MODULE_DEF rep),(DUMMY_MODULE_DEF rep),
           (DUMMY_MODULE_DEF rep),(DUMMY_MODULE_DEF rep),
           (DUMMY_MODULE_DEF rep),(DUMMY_MODULE_DEF rep),
           (DUMMY_MODULE_DEF rep),(DUMMY_MODULE_DEF rep),
           (DUMMY_MODULE_DEF rep),(DUMMY_MODULE_DEF rep),
           (DUMMY_MODULE_DEF rep),(DUMMY_MODULE_DEF rep),
```

```
           DUMMY_OP)
        switch (in_A,in_B,cin) out (neg,zero,ovfl,carry)"
   );;

let MAC2_ALU_SPEC = save_thm
   (`MAC2_ALU_SPEC`,
    instantiate_abstract_definition
          `gen_alu` `ALU_SPEC_DEF` alu_spec_def
   );;


%-----------------------------------------------------------------
yields...

MAC2_ALU_SPEC =
|- !rep switch in_A in_B cin out neg zero ovfl carry.
    MAC2_ALU_SPEC rep switch(in_A,in_B,cin)out(neg,zero,ovfl,carry) =
    ((switch = F,F,F,F) =>
     ADD_WITHOUT_CARRY rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,F,F,T) =>
     ADD_WITH_CARRY rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,F,T,F) =>
     INCREMENT rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,F,T,T) =>
     SUB_WITHOUT_CARRY rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,T,F,F) =>
     SUB_WITH_CARRY rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,T,F,T) =>
     DECREMENT rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,T,T,F) =>
     BITWISE_AND rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = F,T,T,T) =>
     BITWISE_XOR rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,F,F,F) =>
     BITWISE_OR rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,F,F,T) =>
     BITWISE_NOT rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,F,T,F) =>
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,F,T,T) =>
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,T,F,F) =>
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,T,F,T) =>
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
    ((switch = T,T,T,F) =>
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry) |
     ALU_NOOP rep(in_A,in_B,cin)out(neg,zero,ovfl,carry)))))))))))))))
Run time: 219.0s
Intermediate theorems generated: 4104
-----------------------------------------------------------------%

let COND_CONJ_LEMMA = TAC_PROOF
   (([],
    "! (a:bool) (b1 x1 y1:*) b2 b3 .
```

```
       (a => ((b1 = x1) /\ b2) | ((b1 = y1) /\ b3)) =
       ((b1 = (a => x1 | y1)) /\ (a => b2 | b3))"),
    REPEAT GEN_TAC
    THEN COND_CASES_TAC
    THEN REWRITE_TAC []
   );;

let COND_EQT_LEMMA = TAC_PROOF
   (([],
    "! (a:bool) (b1 x1 y1:*) b2 b3 .
       (a => (b1 = x1) | (b1 = y1)) =
       (b1 = (a => x1 | y1))"),
    REPEAT GEN_TAC
    THEN COND_CASES_TAC
    THEN REWRITE_TAC []
   );;

let COND_FUNC_LEMMA = TAC_PROOF
   (([],"! (a:*->**) b (c d:*) .
         (b => (a c) | (a d)) = (a (b => c | d))"),
    REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "b"
    THEN REWRITE_TAC []
   );;

let COND_NULL_LEMMA = TAC_PROOF
   (([],"! b (c: *) .
         (b => c | c) = c"),
    REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "b"
    THEN REWRITE_TAC []
   );;

let lemma_list =
    let MAC2_EXPANDED =
       EXPAND_LET_RULE (
       ONCE_REWRITE_RULE [add_without_carry_def;
                          add_with_carry_def;
                          increment_def;
                          sub_without_carry_def;
                          sub_with_carry_def;
                          decrement_def;
                          bitwise_and_def;
                          bitwise_xor_def;
                          bitwise_or_def;
                          bitwise_not_def;
                          alu_noop_def] MAC2_ALU_SPEC) in
    let rule1 = SPEC "carry:bool" (SYM_RULE EQ_CLAUSE4) and
        rule2 = SPEC "ovfl:bool" (SYM_RULE EQ_CLAUSE4) in
    let lemma1 = PURE_ONCE_REWRITE_RULE [rule1;rule2] MAC2_EXPANDED in
    let lemma2 = UNDISCH(fst(EQ_IMP_RULE (SPEC_ALL lemma1))) in
    let lemma3 = PURE_REWRITE_RULE
                  [COND_CONJ_LEMMA;COND_NULL_LEMMA] lemma2 in
        CONJUNCTS lemma3;;
```

```
let out_lemma = save_thm
   (`MAC2_OUT_LEMMA`,
    (GEN_ALL(DISCH_ALL (el 1 lemma_list)))
   );;

let neg_lemma = save_thm
   (`MAC2_NEG_LEMMA`,
    (GEN_ALL(DISCH_ALL
     (PURE_REWRITE_RULE [COND_FUNC_LEMMA] (el 2 lemma_list))))
   );;

let zero_lemma = save_thm
   (`MAC2_ZERO_LEMMA`,
    (GEN_ALL(DISCH_ALL
     (PURE_REWRITE_RULE [COND_FUNC_LEMMA] (el 3 lemma_list))))
   );;

let ovfl_lemma = save_thm
   (`MAC2_OVFL_LEMMA`,
    (GEN_ALL(DISCH_ALL (el 4 lemma_list)))
   );;

let carry_lemma = save_thm
   (`MAC2_CARRY_LEMMA`,
     (GEN_ALL(DISCH_ALL
      (PURE_REWRITE_RULE [COND_EQT_LEMMA] (el 5 lemma_list))))
   );;
```

3.4.4 The Shifter Unit

The section presents the ML code that creates the theory `shifter_def.th`.

```
%----------------------------------------------------------------

   File:         def_shift.ml

   Author:       (c) P. J. Windley 1990

   Date:         13 JAN 90

   Description:

   Defines a SHIFTER used in subsequent specifications using
   generic operators from the auxilliary definitions theory.

   Modification History:

   May 16 1990

   Added carry signal for shifter end bits.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

system `/bin/rm shift_def.th`;;

new_theory `shift_def`;;

loadf `abstract`;;

map new_parent [`aux_def`];;

let rep_ty = abstract_type `aux_def` `opcode`;;

let shifter_spec_def = new_definition
   (`SHIFTER_SPEC`,
    "! (rep:^rep_ty) switch in_A out .
     SHIFTER_SPEC rep switch in_A out c_flag =
       ((switch = (F,F)) => ((out = (shl rep) in_A) /\
                             (c_flag = (msb rep) in_A))    |
        (switch = (F,T)) => ((out = (shr rep) in_A) /\
                             (c_flag = (lsb rep) in_A))    |
        (switch = (T,F)) => ((out = (asr rep) in_A) /\
                             (c_flag = (lsb rep) in_A))    |
                            ((out = in_A)  /\
                             (c_flag = F))                )"
   );;
```

```
let COND_CONJ_LEMMA = TAC_PROOF
   (([],
    "! (a:bool) (b1 x1 y1:*) b2 b3 .
       (a => ((b1 = x1) /\ b2) | ((b1 = y1) /\ b3)) =
       ((b1 = (a => x1 | y1)) /\ (a => b2 | b3))"),
    REPEAT GEN_TAC
    THEN COND_CASES_TAC
    THEN REWRITE_TAC []
   );;

let COND_EQT_LEMMA = TAC_PROOF
   (([],
    "! (a:bool) (b1 x1 y1:*) b2 b3 .
       (a => (b1 = x1) | (b1 = y1)) =
       (b1 = (a => x1 | y1))"),
    REPEAT GEN_TAC
    THEN COND_CASES_TAC
    THEN REWRITE_TAC []
   );;

let COND_FUNC_LEMMA = TAC_PROOF
   (([],"! (a:*->**) b (c d:*) .
         (b => (a c) | (a d)) = (a (b => c | d))"),
    REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "b"
    THEN REWRITE_TAC []
   );;

let COND_NULL_LEMMA = TAC_PROOF
   (([],"! b (c: *) .
         (b => c | c) = c"),
    REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "b"
    THEN REWRITE_TAC []
   );;

let lemma_list =
    let rule1 = SPEC "c_flag:bool" (SYM_RULE EQ_CLAUSE4) in
    let lemma1 = PURE_ONCE_REWRITE_RULE [rule1] shifter_spec_def in
    let lemma2 = UNDISCH(fst(EQ_IMP_RULE (SPEC_ALL lemma1))) in
    let lemma3 = PURE_REWRITE_RULE
                  [COND_CONJ_LEMMA;COND_NULL_LEMMA] lemma2 in
    CONJUNCTS lemma3;;

let out_lemma = save_thm
   (`SHIFTER_OUT_LEMMA`,
    (GEN_ALL(DISCH_ALL (el 1 lemma_list)))
   );;

let carry_lemma = save_thm
   (`SHIFTER_CARRY_LEMMA`,
     (GEN_ALL(DISCH_ALL
      (PURE_REWRITE_RULE [COND_EQT_LEMMA] (el 2 lemma_list))))
   );;
```

```
close_theory();;
```

3.4.5 The Microprogram Counter Unit

The section presents the ML code that creates the theory `mpc_def.th`.

```
%----------------------------------------------------------------

   File:         def_mpc.ml

   Author:       (c) P. J. Windley 1990

   Date:         18 JAN 90

   Modified:

   Description:

   Defines a function specifying the behavior of the micorprogram
   counter unit.  The definition is used in the specification of
   the electronic block model and the phase level.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`;`assoc/`]));;


system `/bin/rm mpc_def.th`;;

new_theory `mpc_def`;;

map new_parent [`tuple`;`aux_thms`];;

let MPC_UNIT = new_definition
   (`MPC_UNIT`,
    "!(mpc:bt6) (opc:bt6) addr cond ireq_f ie sm.
      MPC_UNIT mpc opc addr cond ireq_f ie sm =
          let bt6_inc n     = (add_bt6 n 1) in
          ((cond = (F,F,F)) => (bt6_inc mpc) |
           (cond = (F,F,T)) => addr |
           (cond = (F,T,F)) => (add_bt6 (F,(SND opc)) 4) |
           (cond = (F,T,T)) => ((ireq_f /\ ie) => addr | (bt6_inc mpc)) |
           (cond = (T,F,F)) => (sm => addr | (bt6_inc mpc)) |
                               (bt6_inc mpc))"
   );;
```

```
close_theory();;
```

3.4.6 The State Selectors.

The section presents the ML code that creates the theory `select_def.th`.

```
%-----------------------------------------------------------------

   File:         def_select.ml

   Author:       (c) P. J. Windley 1990

   Date:         25 May 90

   Description:

   Defines selection functions for the electronic block model state
   and environment.

   Modification History:

-----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`;`assoc/`]));;


system `/bin/rm select_def.th`;;

new_theory `select_def`;;

map new_parent [`ucode_def`;`tuple`];;

new_type_abbrev (`time`,":num");;

%-----------------------------------------------------------------
 Define State and selector functions for s:time->^EBM_state
-----------------------------------------------------------------%
let EBM_state =
    ":((*wordn)list#*wordn#*wordn#*memory#
       *wordn#*wordn#*wordn#*wordn#bt6#
       *wordn#*wordn#bool#bool#ucode#(num->ucode)#bt2)";;

let EBM_env = ":bool";;

let Selector_TAC x =
    REPEAT GEN_TAC
    THEN CONV_TAC (TOP_DEPTH_CONV FUN_EQ_CONV)
```

```
    THEN PURE_ONCE_REWRITE_TAC [x]
    THEN BETA_TAC
    THEN REWRITE_TAC [];;


let RegS = new_definition
   (`RegS`,
    "!(t:time) (s:time->^EBM_state) .
     RegS s t = FST (s t)"
   );;

let RegS = prove_thm
   (`RegS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     RegS (\t.(reg t, psw t, pc t, mem t, ivec t,
                ir t, mar t, mbr t, mpc t,
                alatch t, blatch t, ireq_ff t,
                iack_ff t, mir t, urom, clk t)) = reg",
    Selector_TAC RegS
   );;

let PswS = new_definition
   (`PswS`,
    "!(t:time) (s:time->^EBM_state) .
     PswS s t = FST(SND(s t))"
   );;

let PswS = prove_thm
   (`PswS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     PswS (\t.(reg t, psw t, pc t, mem t, ivec t,
                ir t, mar t, mbr t, mpc t,
                alatch t, blatch t, ireq_ff t,
                iack_ff t, mir t, urom, clk t)) = psw",
    Selector_TAC PswS
   );;

let PcS = new_definition
   (`PcS`,
    "!(t:time) (s:time->^EBM_state) .
     PcS s t = FST(SND(SND(s t)))"
   );;

let PcS = prove_thm
   (`PcS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
```

```
     PcS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = pc",
    Selector_TAC PcS
   );;


let MemS = new_definition
   (`MemS`,
    "!(t:time) (s:time->^EBM_state) .
     MemS s t = FST(SND(SND(SND(s t))))"
   );;

let MemS = prove_thm
   (`MemS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     MemS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = mem",
    Selector_TAC MemS
   );;


let IvecS = new_definition
   (`IvecS`,
    "!(t:time) (s:time->^EBM_state) .
     IvecS s t = FST(SND(SND(SND(SND(s t)))))"
   );;

let IvecS = prove_thm
   (`IvecS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     IvecS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = ivec",
    Selector_TAC IvecS
   );;


let IrS = new_definition
   (`IrS`,
    "!(t:time) (s:time->^EBM_state) .
     IrS s t = FST(SND(SND(SND(SND(SND(s t))))))"
   );;

let IrS = prove_thm
```

```
   (`IrS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     IrS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = ir",
    Selector_TAC IrS
   );;

let MarS = new_definition
   (`MarS`,
    "!(t:time) (s:time->^EBM_state) .
     MarS s t = FST(SND(SND(SND(SND(SND(SND(s t)))))))"
   );;

let MarS = prove_thm
   (`MarS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     MarS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = mar",
    Selector_TAC MarS
   );;

let MbrS = new_definition
   (`MbrS`,
    "!(t:time) (s:time->^EBM_state) .
     MbrS s t = FST(SND(SND(SND(SND(SND(SND(SND(s t))))))))"
   );;

let MbrS = prove_thm
   (`MbrS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     MbrS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = mbr",
    Selector_TAC MbrS
   );;


let MpcS = new_definition
   (`MpcS`,
    "!(t:time) (s:time->^EBM_state) .
     MpcS s t = FST(SND(SND(SND(SND(SND(SND(SND(SND(s t)))))))))"
```

```
   );;

let MpcS = prove_thm
   (`MpcS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     MpcS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = mpc",
    Selector_TAC MpcS
   );;


let AlatchS = new_definition
   (`AlatchS`,
    "!(t:time) (s:time->^EBM_state) .
     AlatchS s t = FST(SND(SND(SND(SND(
                       SND(SND(SND(SND(SND(s t))))))))))"
   );;

let AlatchS = prove_thm
   (`AlatchS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     AlatchS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = alatch",
    Selector_TAC AlatchS
   );;

let BlatchS = new_definition
   (`BlatchS`,
    "!(t:time) (s:time->^EBM_state) .
     BlatchS s t = FST(SND(SND(SND(SND(
                       SND(SND(SND(SND(SND(SND(s t)))))))))))"
   );;

let BlatchS = prove_thm
   (`BlatchS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     BlatchS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = blatch",
    Selector_TAC BlatchS
   );;
```

```
let IreqS = new_definition
   (`IreqS`,
    "!(t:time) (s:time->^EBM_state) .
     IreqS s t = FST(SND(SND(SND(SND(SND
                    (SND(SND(SND(SND(SND(SND(s t)))))))))))"
   );;

let IreqS = prove_thm
   (`IreqS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     IreqS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = ireq_ff",
    Selector_TAC IreqS
   );;

let IackS = new_definition
   (`IackS`,
    "!(t:time) (s:time->^EBM_state) .
     IackS s t = FST(SND(SND(SND(SND(SND(SND(
                      SND(SND(SND(SND(SND(SND(s t))))))))))))"
   );;

let IackS = prove_thm
   (`IackS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     IackS (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)) = iack_ff",
    Selector_TAC IackS
   );;

let MirS = new_definition
   (`MirS`,
    "!(t:time) (s:time->^EBM_state) .
     MirS s t = FST(SND(SND(SND(SND(SND(SND(SND(
                      SND(SND(SND(SND(SND(SND(s t)))))))))))))"
   );;

let MirS = prove_thm
   (`MirS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     MirS (\t.(reg t, psw t, pc t, mem t, ivec t,
```

```
              ir t, mar t, mbr t, mpc t,
              alatch t, blatch t, ireq_ff t,
              iack_ff t, mir t, urom, clk t)) = mir",
    Selector_TAC MirS
   );;


let UromS = new_definition
   (`UromS`,
    "!(t:time) (s:time->^EBM_state) .
     UromS s t = FST(SND(SND(SND(SND(SND(SND(SND(SND(
                     SND(SND(SND(SND(SND(SND(s t))))))))))))))"
   );;

let UromS = prove_thm
   (`UromS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     UromS (\t.(reg t, psw t, pc t, mem t, ivec t,
              ir t, mar t, mbr t, mpc t,
              alatch t, blatch t, ireq_ff t,
              iack_ff t, mir t, urom, clk t)) = (\t:time.urom)",
    Selector_TAC UromS
   );;

let ClkS = new_definition
   (`ClkS`,
    "!(t:time) (s:time->^EBM_state) .
     ClkS s t = SND(SND(SND(SND(SND(SND(SND(SND(SND(
                     SND(SND(SND(SND(SND(SND(s t))))))))))))))"
   );;

let ClkS = prove_thm
   (`ClkS`,
    "!(reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode) (ireq_ff iack_ff:time->bool).
     ClkS (\t.(reg t, psw t, pc t, mem t, ivec t,
              ir t, mar t, mbr t, mpc t,
              alatch t, blatch t, ireq_ff t,
              iack_ff t, mir t, urom, clk t)) = clk",
    Selector_TAC ClkS
   );;

%----------------------------------------------------------------
 Selectors on the environment
----------------------------------------------------------------%
let IreqE = new_definition
   (`IreqE`,
    "! (t:time) (e:time->^EBM_env) .
     IreqE e t = (e t)"
   );;
```

```
let IreqE = prove_thm
   (`IreqE`,
    "! (t:time) (ireq_e:time->bool) .
     IreqE (\t. (ireq_e t)) = ireq_e",
    Selector_TAC IreqE
   );;


close_theory();;
```

3.4.7 The Electronic Block Model

The section presents the ML code that creates the theory `block_def.th`.

```
%----------------------------------------------------------------

   File:         def_block.ml

   Author:       (c) P. J. Windley 1990

   Date:         12 JAN 90

   Description:

   Defines the behavioral description of the electronic block
   model.

   Modification History:

   May 16 90

   Updated to reflect new design.
     -- Non-user registers have been moved out of register file.
     -- Jump condition calculator is added.
     -- PMUX added to multiplex input to MAR
     -- MAR loads from PMUX
     -- Shifter generates carry out
     -- CMUX multiplexes carry signals from ALU and Shifter

   May 25 90

   Corrected errors in the specification:
     -- Demuxes must not have floating lines
     -- Wired ORs cannot be used (lead to inconsistacies)

   Removed EBM state selection functions and placed in
   def_select.ml.

   Corrected IR_SPEC definition to reflect desired behavior.

   Connected C255 to B bus rather than the C bus.

   May 28 190

   Fixed IVEC unit so that ivec has state.

   Fixed PC unit selection so that it doesn't fall through.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;
```

```
let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`;`assoc/`]));;


loadf `abstract`;;

system `/bin/rm block_def.th`;;

new_theory `block_def`;;

map new_parent [`alu_def`;`shift_def`;`aux_thms`;`mpc_def`;
                `tuple`;`regs_def`;`ucode_def`;`jump_def`];;


load_parent `select_def`;;

let rep_ty = abstract_type `aux_def` `opcode`;;

%----------------------------------------------------------------
 Ground
----------------------------------------------------------------%

let GND = new_definition
   (`GND`,
    "! out . GND out = (out = F)"
   );;

%----------------------------------------------------------------
 n-bit Mux spec
----------------------------------------------------------------%

let MUX_SPEC = new_definition
   (`MUX_SPEC`,
    "! ctl (a:*wordn) b c .
     MUX_SPEC ctl a b c =
       c = (ctl => a | b)"
   );;

%----------------------------------------------------------------
 1-bit Mux spec
----------------------------------------------------------------%

let MUX_1_SPEC = new_definition
   (`MUX_1_SPEC`,
    "! ctl (a:bool) b c .
     MUX_1_SPEC ctl a b c =
       c = (ctl => a | b)"
   );;

%----------------------------------------------------------------
 Latch specification
```

```
-----------------------------------------------------------------%

let LATCH_SPEC = new_definition
   (`LATCH_SPEC`,
    " ! (i:time->*wordn) ld out .
      LATCH_SPEC i ld out =
       (! t:time . out(t+1) = ld t  => i t
                                      | out t)"
   );;


%-----------------------------------------------------------------
 Register specification
-----------------------------------------------------------------%

let REG_SPEC = new_definition
   (`REG_SPEC`,
    " ! (i:time->*wordn) ld out contents .
      REG_SPEC i ld prt out contents =
       ! t:time .
       (contents (t+1) = ld t  => i t | contents t) /\
       (prt t ==> (out = contents))"
   );;

%-----------------------------------------------------------------
 Flipflop
-----------------------------------------------------------------%
let FF_SPEC = new_definition
   (`FF_SPEC`,
    "! (in:time->bool) (ld:time->bool) (q:time->bool) .
     FF_SPEC in ld q =
       ! t:num . q(t+1) = ((ld t)  => in t | q t)"
   );;

%-----------------------------------------------------------------
 Register block
-----------------------------------------------------------------%

let REGISTER_BLOCK = new_definition
   (`REGISTER_BLOCK`,
    "! (rep:^rep_ty) c a b
       ld ld_ssp prt_A prt_D prt_B ssp (in:time->*wordn) outA outB  psw
       (reg_list:time->(*wordn)list).
     REGISTER_BLOCK rep c a b ld ld_ssp prt_A prt_D ssp prt_B
                      in outA outB psw reg_list =
        !t:time .
         (reg_list (t+1) =
           (ld t) => (UPDATE_REG rep (psw t) (reg_len rep (c t))
                                (reg_list t) (in t)) |
           (ld_ssp t) => (UPDATE_REG rep (psw t) ssp_reg
                                (reg_list t) (in t))
                   | (reg_list t)) /\
         (prt_A t ==> (outA t = (EL (reg_len rep (a t)) (reg_list t)))) /\
         (prt_D t ==> (outA t = (EL (reg_len rep (c t)) (reg_list t)))) /\
         (ssp t   ==> (outA t = (SSP_REG (reg_list t)))) /\
```

```
        (prt_B t ==> (outB t = (EL (reg_len rep (b t)) (reg_list t))))"
   );;

%----------------------------------------------------------------
 Instruction Register
----------------------------------------------------------------%

let IR_SPEC = new_definition
   (`IR_SPEC`,
    "! (rep:^rep_ty) set prt (in out contents:time->*wordn)
       opc_port dest_port srca_port srcb_port .
     IR_SPEC rep set prt in out contents
                opc_port dest_port srca_port srcb_port =
     (!t:time.
         (contents (t+1) = (set t) => in t | contents t) /\
         (opc_port t  = opcode rep (contents t)) /\
         (dest_port t = dest rep (contents t)) /\
         (srca_port t = srca rep (contents t)) /\
         (srcb_port t = srcb rep (contents t)) /\
         (prt t ==> (out t = (imm rep (contents t)))))"
   );;

%----------------------------------------------------------------
 PSW Register
----------------------------------------------------------------%

let PSW_SPEC = new_definition
   (`PSW_SPEC`,
    "! (rep:^rep_ty) set (in:time->*wordn) out contents
       s_sm c_sm s_ie c_ie ld_v ld_n ld_c ld_z
       vf nf cf zf ie sm.
     PSW_SPEC rep set clk prt in out ie sm contents
                 vf nf cf zf
                 s_sm c_sm s_ie c_ie ld_v ld_n ld_c ld_z =
     (!t:time.
         (contents (t+1) =
           ((set t) /\ (get_sm rep (contents t))) => (in t) |
           (clk t) =>
             (mk_psw rep (
              (s_sm t => T  | c_sm t => F | (get_sm rep (contents t))),
              (s_ie t => T  | c_ie t => F | (get_ie rep (contents t))),
              (ld_v t => vf | (get_vf rep (contents t))),
              (ld_n t => nf | (get_nf rep (contents t))),
              (ld_c t => cf | (get_cf rep (contents t))),
              (ld_z t => zf | (get_zf rep (contents t))))) |
             (contents t)) /\
         (sm t = get_sm rep (contents t)) /\
         (ie t = get_ie rep (contents t)) /\
         (prt t ==> (out = contents)))"
   );;


%----------------------------------------------------------------
 JUMP condition calculator
----------------------------------------------------------------%
```

```
let JUMP_SPEC = new_definition
   (`JUMP_SPEC`,
    "! (rep:^rep_ty) d psw out .
     JUMP_SPEC rep d psw out =
      !t:time .  (out t) = JUMP_COND rep (reg_len rep (d t)) (psw t)"
   );;

%----------------------------------------------------------------
 Mbr
----------------------------------------------------------------%
let MBR_SPEC = new_definition
   (`MBR_SPEC`,
    "! set clk rd_s wr_s (i:time->*wordn) value bus mem_port .
     MBR_SPEC set clk rd_s wr_s i value bus mem_port =
     (!t:time.
          (value (t+1) = (((clk t) /\ (rd_s t)) => mem_port t |
                         ((clk t) /\ (set t)) => i t | value t)) /\
          (wr_s t ==> (mem_port = value))) /\
     (bus = value)"
   );;

%----------------------------------------------------------------
 C255  (constant)
----------------------------------------------------------------%

let C255_SPEC = new_definition
   (`C255_SPEC`,
    "! (rep:^rep_ty) prt out .
     C255_SPEC rep prt out =
        prt ==> (out = (wordn rep 255))"
   );;

%----------------------------------------------------------------
 Interrupt vector register specification
----------------------------------------------------------------%

let IVEC_SPEC = new_definition
   (`IVEC_SPEC`,
    " ! (rep:^rep_ty) prt (out:time->*wordn) contents .
      IVEC_SPEC rep prt out contents =
       ! t:time .
        (contents (t+1) = (contents t)) /\
        (prt t ==> (out t = (int_fetch rep (contents t))))"
   );;

%----------------------------------------------------------------
 Decoder Specs
----------------------------------------------------------------%

let DEMUX_2_SPEC = new_definition
   (`DEMUX_2_SPEC`,
    "! s o0 o1 o2 o3 .
     DEMUX_2_SPEC s o0 o1 o2 o3 =
         (!t . o0 t = ((s t) = (F,F))) /\
```

```
         (!t . o1 t = ((s t) = (F,T))) /\
         (!t . o2 t = ((s t) = (T,F))) /\
         (!t . o3 t = ((s t) = (T,T)))"
   );;

let DEMUX_3_SPEC = new_definition
   (`DEMUX_3_SPEC`,
    "! s o0 o1 o2 o3 o4 o5 o6 o7 .
     DEMUX_3_SPEC s o0 o1 o2 o3 o4 o5 o6 o7 =
         (!t . o0 t = ((s t) = (F,F,F))) /\
         (!t . o1 t = ((s t) = (F,F,T))) /\
         (!t . o2 t = ((s t) = (F,T,F))) /\
         (!t . o3 t = ((s t) = (F,T,T))) /\
         (!t . o4 t = ((s t) = (T,F,F))) /\
         (!t . o5 t = ((s t) = (T,F,T))) /\
         (!t . o6 t = ((s t) = (T,T,F))) /\
         (!t . o7 t = ((s t) = (T,T,T)))"
   );;

%----------------------------------------------------------------
 Memory
----------------------------------------------------------------%

let MEM = new_definition
   (`MEM`,
    "! (rep:^rep_ty) wr_s rd_s addr data mem.
     MEM rep wr_s rd_s addr data mem =
     !t:time .
        (mem (t+1) =
           (wr_s t => store rep (mem t, address rep (addr t), (data t))
                   | mem t)) /\
        (rd_s t ==> (data t = (fetch rep (mem t, address rep (addr t)))))"
   );;

%----------------------------------------------------------------
 LOGIC gates
----------------------------------------------------------------%

let AND_SPEC = new_definition
   (`AND_SPEC`,
    "! a b out .
     AND_SPEC a b out =
       (!t:time . (out t) = (a t) /\ (b t))"
   );;

let OR_SPEC = new_definition
   (`OR_SPEC`,
    "! a b out .
     OR_SPEC a b out =
       (!t:time . (out t) = (a t) \/ (b t))"
   );;

let OR_3_SPEC = new_definition
   (`OR_3_SPEC`,
    "! a b c out .
```

```
     OR_3_SPEC a b c out =
      (!t:time . (out t) = (a t) \/ (b t) \/ (c t))"
   );;

let MAR_LOGIC_SPEC = new_definition
   (`MAR_LOGIC_SPEC`,
    "! pmux clk_3 clk_4 mar out .
     MAR_LOGIC_SPEC pmux clk_3 clk_4 mar out =
      !t:time. (out t) =
       ((((pmux t) /\ (clk_3 t)) \/ (~(pmux t) /\ (clk_4 t))) /\ (mar t))"
   );;

let PC_LOGIC_SPEC = new_definition
   (`PC_LOGIC_SPEC`,
    "! pc_enable pc_jmp_enable jump_flag out .
     PC_LOGIC_SPEC clk pc_enable pc_jmp_enable jump_flag out =
     !t:time. (out t) = (clk t) /\
                        ((pc_enable t) \/
                         ((pc_jmp_enable t) /\ (jump_flag t)))"
   );;

%-----------------------------------------------------------------
Data path
-----------------------------------------------------------------%

let DATAPATH = new_definition
   (`DATAPATH`,
    "! (rep:^rep_ty)
       mem reg mar mbr alatch blatch ir pc psw ivec
       iack_ff ireq_ff
       ireq_e
       amux_s alu_s shft_s mbr_s mar_s pmux_s cselect aselect bselect
       s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s
       iack_s rd_s wr_s
       opc ie sm
       clk_1 clk_2 clk_3 clk_4 .
     DATAPATH rep mem reg mar mbr alatch blatch ir pc psw ivec
              iack_ff ireq_ff
              ireq_e
              amux_s alu_s shft_s mbr_s mar_s pmux_s cselect aselect bselect
              s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s
              iack_s rd_s wr_s
              opc ie sm
              clk_1 clk_2 clk_3 clk_4 =
     !t:time.
      ? Abus Bbus Cbus MuxOut MuxIn MemData AluOut Gnd MarIn
        regd_enable ssp_enable psw_enable ir_enable pc_enable pc_jmp_enable
        reg_a_enable reg_sa_enable ssp_a_enable
        psw_a_enable C255_enable pc_a_enable
        reg_b_enable ivec_enable ir_b_enable
        ld_reg_block ld_ssp ld_ir ld_psw ld_mar ld_pc do_write
        dest_s srca_s srcb_s alu_c shift_c cf nf vf zf jump_flag
        pc_a_1 pc_a_2 pc_a_3 ir_b_1 ir_b_2
        float0 float1 .
         (GND (Gnd t)) /\
```

```
        (DEMUX_3_SPEC cselect regd_enable ssp_enable psw_enable
                              ir_enable pc_enable pc_jmp_enable
                              float0 float1) /\
        (DEMUX_3_SPEC aselect reg_a_enable reg_sa_enable ssp_a_enable
                              psw_a_enable C255_enable pc_a_1
                              pc_a_2 pc_a_3) /\
        (OR_3_SPEC pc_a_1 pc_a_2 pc_a_3 pc_a_enable) /\
        (DEMUX_2_SPEC bselect reg_b_enable ivec_enable
                              ir_b_1 ir_b_2) /\
        (OR_SPEC ir_b_1 ir_b_2 ir_b_enable) /\
        (AND_SPEC clk_4 regd_enable ld_reg_block) /\
        (AND_SPEC clk_4 ssp_enable ld_ssp) /\
        (REGISTER_BLOCK rep dest_s srca_s srcb_s
                            ld_reg_block ld_ssp reg_a_enable reg_sa_enable
                            ssp_a_enable reg_b_enable
                            Cbus Abus Bbus psw reg) /\
        (AND_SPEC clk_4 ir_enable ld_ir) /\
        (IR_SPEC rep ld_ir ir_b_enable Cbus Bbus ir
                       opc dest_s srca_s srcb_s) /\
        (LATCH_SPEC Abus clk_2 alatch) /\
        (LATCH_SPEC Bbus clk_2 blatch) /\
        (IVEC_SPEC rep ivec_enable Bbus ivec) /\
        (FF_SPEC iack_s clk_2 iack_ff) /\
        (FF_SPEC ireq_e clk_1 ireq_ff) /\
        (MUX_SPEC (amux_s t) (MuxIn t) (alatch t) (MuxOut t)) /\
        (MAC2_ALU_SPEC rep (alu_s t) (MuxOut t,blatch t,get_cf rep (psw t))
                     (AluOut t) (nf t, zf t,vf t,alu_c t)) /\
        (SHIFTER_SPEC rep (shft_s t) (AluOut t) (Cbus t) (shift_c t)) /\
        (MUX_1_SPEC (csrc_s t) (alu_c t) (shift_c t) (cf t)) /\
        (MBR_SPEC mbr_s clk_4 rd_s wr_s Cbus
                  mbr MuxIn MemData) /\
        (AND_SPEC clk_4 psw_enable ld_psw) /\
        (PSW_SPEC rep ld_psw clk_4 psw_a_enable Cbus Abus ie sm psw
                       (vf t) (nf t) (cf t) (zf t)
                       s_sm c_sm s_ie c_ie ld_v ld_n ld_c ld_z) /\
        (JUMP_SPEC rep dest_s psw jump_flag) /\
        (PC_LOGIC_SPEC clk_4 pc_enable pc_jmp_enable jump_flag ld_pc) /\
        (REG_SPEC Cbus ld_pc pc_a_enable Abus pc) /\
        (C255_SPEC rep (C255_enable t) (Abus t)) /\
        (MUX_SPEC (pmux_s t) (pc t) (Cbus t) (MarIn t)) /\
        (MAR_LOGIC_SPEC pmux_s clk_3 clk_4 mar_s ld_mar) /\
        (LATCH_SPEC MarIn ld_mar mar) /\
        (AND_SPEC clk_4 wr_s do_write) /\
        (MEM rep do_write rd_s mar MemData mem)"
   );;

%-----------------------------------------------------------------
 Control Unit
-----------------------------------------------------------------%

let MPC_SPEC = new_definition
   (`MPC_SPEC`,
    "! (rep:^rep_ty) clk opc mpc addr_s irq ie sm cond_s .
     MPC_SPEC rep mpc clk opc irq ie sm addr_s cond_s =
        ! t:time .
```

```
         mpc (t+1) =
          ((clk t) =>
             (MPC_UNIT (mpc t) (opc t)
                       (addr_s t) (cond_s t) (irq t) (ie t) (sm t)) |
             mpc t)"
   );;

let MIR_SPEC = new_definition
   (`MIR_SPEC`,
    "! (mir:time->ucode) clk in
       amux_s sh_s alu_s mbr_s mar_s pmux_s cselect aselect bselect
       s_sm_s c_sm_s s_ie_s c_ie_s
       ld_c_s ld_v_s ld_n_s ld_z_s csrc_s ftch_s
       iack_s rd_s wr_s addr_s cond_s .
     MIR_SPEC mir clk in
              amux_s sh_s alu_s mbr_s mar_s pmux_s cselect aselect bselect
              s_sm_s c_sm_s s_ie_s c_ie_s
              ld_c_s ld_v_s ld_n_s ld_z_s csrc_s ftch_s
              iack_s rd_s wr_s addr_s cond_s =
      !t:time .
          (mir (t+1) = (clk t => (in t) | (mir t))) /\
          (amux_s t =  (Amux (mir t))) /\
          (sh_s t =  (Shift (mir t))) /\
          (alu_s t = (Alu (mir t))) /\
          (mbr_s t = (Mbr (mir t))) /\
          (mar_s t = (Mar (mir t))) /\
          (pmux_s t = (Pmux (mir t))) /\
          (cselect t = (Trgt (mir t))) /\
          (aselect t = (SrcA (mir t))) /\
          (bselect t = (SrcB (mir t))) /\
          (s_sm_s t = (S_sm (mir t))) /\
          (c_sm_s t = (C_sm (mir t))) /\
          (s_ie_s t = (S_ie (mir t))) /\
          (c_ie_s t = (C_ie (mir t))) /\
          (ld_c_s t = (Ld_c (mir t))) /\
          (ld_v_s t = (Ld_v (mir t))) /\
          (ld_n_s t = (Ld_n (mir t))) /\
          (ld_z_s t = (Ld_z (mir t))) /\
          (csrc_s t = (Csrc (mir t))) /\
          (ftch_s t = (Ftch (mir t))) /\
          (iack_s t = (Iack (mir t))) /\
          (rd_s t = (Rd (mir t))) /\
          (wr_s t = (Wr (mir t))) /\
          (addr_s t = (Address (mir t))) /\
          (cond_s t = (Cond (mir t)))"
   );;


let CLOCK_SPEC = new_definition
   (`CLOCK_SPEC`,
    "! clk clk_1 clk_2 clk_3 clk_4 .
     CLOCK_SPEC clk clk_1 clk_2 clk_3 clk_4 =
         !t:time .
         (clk (t+1) = (((clk t) = (F,F)) => (F,T) |
                        ((clk t) = (F,T)) => (T,F) |
```

```
                     ((clk t) = (T,F)) => (T,T) | (F,F))) /\
        (clk_1 t = (clk t = (F,F))) /\
        (clk_2 t = (clk t = (F,T))) /\
        (clk_3 t = (clk t = (T,F))) /\
        (clk_4 t = (clk t = (T,T)))"
   );;


let CONTROL_UNIT = new_definition
   (`CONTROL_UNIT`,
    "! (rep:^rep_ty) (mpc:time->bt6) (mir:time->ucode) clk
       (urom:(time->num->ucode))
       clk_1 clk_2 clk_3 clk_4
       amux_s sh_s alu_s mbr_s mar_s pmux_s cselect aselect bselect
       s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s ftch_s
       iack_s rd_s wr_s
       opc sm ie ireq_f .
     CONTROL_UNIT rep
          mpc mir clk urom
          clk_1 clk_2 clk_3 clk_4
          amux_s sh_s alu_s mbr_s mar_s pmux_s cselect aselect bselect
          s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s ftch_s
          iack_s rd_s wr_s
          opc sm ie ireq_f =
       ? addr_s cond_s .
       (MPC_SPEC rep mpc clk_4 opc ireq_f ie sm addr_s cond_s) /\
       (MIR_SPEC mir clk_1 (\t.(urom t (bt6_val (mpc t))))
                 amux_s sh_s alu_s mbr_s mar_s pmux_s cselect aselect bselect
                 s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s ftch_s
                 iack_s rd_s wr_s addr_s cond_s) /\
       (CLOCK_SPEC clk clk_1 clk_2 clk_3 clk_4)"
   );;


%----------------------------------------------------------------
 Define State and selector functions for s:time->^EBM_state
----------------------------------------------------------------%
let EBM_state =
    ":((*wordn)list#*wordn#*wordn#*memory#
       *wordn#*wordn#*wordn#*wordn#bt6#
       *wordn#*wordn#bool#bool#ucode#(num->ucode)#bt2)";;

let EBM_env = ":bool";;

%----------------------------------------------------------------
 Define Electronic Block Model

 This definition uses the selection functions on the state and
 environment defined in def_select.ml.  This is done in order
 to have the definition be of the for "EBM rep s e = ..." so
 that it can be used with the generic interpreter theory.
----------------------------------------------------------------%

let EBM_def = new_definition
   (`EBM_def`,
```

```
   "! (rep:^rep_ty) (s:time->^EBM_state) (e:time->^EBM_env) .
    EBM rep s e =
      ? amux_s alu_s shft_s mbr_s mar_s pmux_s cselect aselect
        bselect
        s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s
        iack_s rd_s wr_s ftch_s
        opc ie sm
        clk_1 clk_2 clk_3 clk_4 .
      (DATAPATH rep
          (MemS s) (RegS s) (MarS s) (MbrS s)
          (AlatchS s) (BlatchS s) (IrS s) (PcS s) (PswS s)
          (IvecS s) (IackS s) (IreqS s)
          (IreqE e)
          amux_s alu_s shft_s mbr_s mar_s pmux_s cselect aselect
          bselect
          s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s
          iack_s rd_s wr_s
          opc ie sm
          clk_1 clk_2 clk_3 clk_4) /\
      (CONTROL_UNIT rep
          (MpcS s) (MirS s) (ClkS s) (UromS s)
          clk_1 clk_2 clk_3 clk_4
          amux_s shft_s alu_s mbr_s mar_s pmux_s cselect aselect
          bselect
          s_sm c_sm s_ie c_ie ld_c ld_v ld_n ld_z csrc_s ftch_s
          iack_s rd_s wr_s
          opc sm ie (IreqS s))"
  );;

let EBM = save_thm
  (`EBM`,
   GEN_ALL (
   PURE_ONCE_REWRITE_RULE
         [RegS;PswS;PcS;MemS;IvecS;IrS;MarS;
          MbrS;MpcS;AlatchS;BlatchS;IreqS;
          IackS;MirS;UromS;ClkS;IreqE] (
   SPECL ["rep:^rep_ty";
          "(\t.(reg t, psw t, pc t, mem t, ivec t,
                ir t, mar t, mbr t, mpc t,
                alatch t, blatch t, ireq_ff t,
                iack_ff t, mir t, urom, clk t)):time->^EBM_state";
          "(\t. (ireq_e t)):time->^EBM_env"] EBM_def))
  );;


let EBM_expanded = save_thm
  (`EBM_expanded`,
   CONV_RULE (TOP_DEPTH_CONV BETA_CONV) (
   PURE_ONCE_REWRITE_RULE [
      GND;MUX_SPEC;MUX_1_SPEC;LATCH_SPEC;REG_SPEC;
      FF_SPEC;REGISTER_BLOCK;IR_SPEC;PSW_SPEC;
      JUMP_SPEC;MBR_SPEC;C255_SPEC;DEMUX_2_SPEC;
      DEMUX_3_SPEC;MEM;AND_SPEC;OR_SPEC;OR_3_SPEC;
      MAR_LOGIC_SPEC;PC_LOGIC_SPEC;
      MPC_SPEC;MIR_SPEC;CLOCK_SPEC;IVEC_SPEC
```

```
      ] (
    PURE_ONCE_REWRITE_RULE [DATAPATH; CONTROL_UNIT] (
    SPEC_ALL EBM))
   );;


%----------------------------------------------------------------
 Define a function that maps EBM state to the EBM counter.
----------------------------------------------------------------%
let GetEBMClock = new_definition
   (`GetEBMClock`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr alatch blatch:*wordn)
      (mpc:bt6) (clk:bt2) (urom:num->ucode) (mir:ucode)
      (ireq_ff iack_ff int_e:bool).
     GetEBMClock rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                      alatch, blatch, ireq_ff, iack_ff, mir, urom, clk)
                     (int_e) = @x:bool.F"
   );;

%----------------------------------------------------------------
 Define the start state
----------------------------------------------------------------%

let EBM_Start = new_definition
   (`EBM_Start`,
    "EBM_Start = @x:bool.F"
   );;

close_theory();;
```

3.5 The Phase–Level

This section presents the theories that define the phase–level interpreter. Also presented is the theory that verifies the phase–level interpreter with respect to the electronic block model.

3.5.1 The Microcode Assembler

The section presents the ML code that defines the microcode assembler.

```
%----------------------------------------------------------------

   File:          ucode_aux.ml

   Author:        (c) P. J. Windley 1990

   Date:          JUN 23, 1990

   Modified:

   Description:

   Defines the ML functions and constants necessary to describe
   the microintrcutions.  This file is loaded by several files
   that draft theories.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root) [`decimal/`;`assoc/`;`tuple/`]));;

%----------------------------------------------------------------

A microinstruction has the following format:


 Bits  Mneumonic    Description

    1  AMUX         Toggle MUX on A-bus
    2  SHFT         Shifter function
    4  ALU          ALU function
    1  MAR          Load MAR from P-Mux
    1  MBR          Load MBR from C-bus
    1  PMUX         Toggle MUX loading MAR
    3  SRCA         A-bus source (includes SSP)
    2  SRCB         B-bus source
```

```
    3  TRGT         C-bus target (includes SSP)

    1  S_SM         Set supervisory mode bit in PSW
    1  C_SM         Clear supervisory mode bit in PSW
    1  S_IE         Set interrupt enable bit in PSW
    1  C_IE         Clear interrupt enable bit in PSW
    1  LD_C         Load carry bit in PSW
    1  LD_V         Load overflow bit in PSW
    1  LD_N         Load negative bit in PSW
    1  LD_Z         Load zero bit in PSW
    1  CSRC         Source of carry (shifter or alu)

    1  IACK         Interrupt acknowledge signal
    1  FTCH         Fetch signal
    1  RD           Read signal
    1  WR           Write signal

    3  COND         Microcode jump condition
    6  ADDR         Next address

-----------------------------------------------------------------%


%-----------------------------------------------------------------
 Shifter mnuemonics
-----------------------------------------------------------------%
let shl = "(F,F)";;

let shr = "(F,T)";;

let asr = "(T,F)";;

let nsh = "(T,T)";;

%-----------------------------------------------------------------
 ALU mnuemonics
-----------------------------------------------------------------%
let add = "(F,F,F,F)";;

let addc = "(F,F,F,T)";;

let inc = "(F,F,T,F)";;

let sub = "(F,F,T,T)";;

let subc = "(F,T,F,F)";;

let dec = "(F,T,F,T)";;

let band = "(F,T,T,F)";;

let bxor = "(F,T,T,T)";;

let bor = "(T,F,F,F)";;
```

```
let bnot = "(T,F,F,T)";;

let nop = "(T,F,T,F)";;


%-----------------------------------------------------------------
 Register mnuemonics
-----------------------------------------------------------------%

let reg_file = "(F,F,F,F)";;

let ssp = "(F,F,F,T)";;

let ir = "(F,F,T,F)";;

let psw = "(F,F,T,T)";;

let pc = "(F,T,F,F)";;

let pcj = "(F,T,F,T)";;

let mar = "(F,T,T,F)";;

let mbr = "(F,T,T,T)";;

let noreg = "(T,F,F,F)";;

let mar_gets_pc = "(T,F,F,T)";;

let reg_dest = "(T,F,T,F)";;

let C255 = "(T,F,T,T)";;

let ivec = "(T,T,F,F)";;

%-----------------------------------------------------------------

The effect of a microinstruction on the major components of the
datapath is described by an 5-tuple:

  Oper (target, shifterop, sourceA, aluop, sourceB)

     Target is the target register
     SourceA is the register fed to the A-latch
     SourceB is the register fed to the B-latch
     AluOp is the AluOp applied to SourceA and SourceB
     ShifterOp is the shifter operation applied to the result of
        AluOp

-----------------------------------------------------------------%


let Process_Trgt x =
     (x = reg_file) => "(F,F,F)" |
     (x = ssp)      => "(F,F,T)" |
```

```
     (x = psw)      => "(F,T,F)" |
     (x = ir)       => "(F,T,T)" |
     (x = pc)       => "(T,F,F)" |
     (x = pcj)      => "(T,F,T)" |
                       "(T,T,F)";;


let Process_Srca x =
     (x = reg_file) => "(F,F,F)" |
     (x = reg_dest) => "(F,F,T)" |
     (x = ssp)      => "(F,T,F)" |
     (x = psw)      => "(F,T,T)" |
     (x = C255)     => "(T,F,F)" |
                       "(T,F,T)";;

let Process_Srcb x =
     (x = reg_file) => "(F,F)" |
     (x = ivec)     => "(F,T)" |
                       "(T,F)";;

let Process_MBR x =
     (x = mbr) => "T" | "F";;

let Process_MAR x =
     ((x = mar) or (x = mar_gets_pc)) => "T" | "F";;

let Process_PMUX x =
     (x = mar_gets_pc) => "T" | "F";;

let Process_AMUX x =
     (x = mbr) => "T" | "F";;

let Oper (trgt, sop, srca, aop, srcb, special) =
     "(^(Process_AMUX srca),
       ^sop,
       ^aop,
       ^(Process_MBR special),
       ^(Process_MAR special),
       ^(Process_PMUX special),
       ^(Process_Trgt trgt),
       ^(Process_Srca srca),
       ^(Process_Srcb srcb))";;


%----------------------------------------------------------------

Oper(reg_file,nsh,reg_file,add,ir,noreg);;

Oper(reg_file,shl,mbr,band,reg_file,mar);;

----------------------------------------------------------------%

%----------------------------------------------------------------
The PSW loading is given by PSW_Control:
    1  S_SM         Set supervisory mode bit in PSW
```

```
    1  C_SM         Clear supervisory mode bit in PSW
    1  S_IE         Set interrupt enable bit in PSW
    1  C_IE         Clear interrupt enable bit in PSW
    1  LD_C         Load carry bit in PSW
    1  LD_V         Load overflow bit in PSW
    1  LD_N         Load negative bit in PSW
    1  LD_Z         Load zero bit in PSW
    1  CSRC         Source of carry (shifter or alu)
----------------------------------------------------------------%

let set_sm = 1;;

let clr_sm = 2;;

let set_ie = 1;;

let clr_ie = 2;;

let pass = 3;;

let ld_from_alu = 1;;

let ld_from_shifter = 2;;

let ld_vf = 4;;

let ld_nf = 4;;

let ld_zf = 4;;


let Set_PSW (sm,ie,vf,nf,cf,zf) =
   "(^((sm = set_sm) => "T" | "F"),
     ^((sm = clr_sm) => "T" | "F"),
     ^((ie = set_ie) => "T" | "F"),
     ^((ie = clr_ie) => "T" | "F"),
     ^((cf = ld_from_alu) or (cf = ld_from_shifter) => "T" | "F"),
     ^((vf = ld_vf) => "T" | "F"),
     ^((nf = ld_nf) => "T" | "F"),
     ^((zf = ld_zf) => "T" | "F"),
     ^((cf = ld_from_alu) => "T" | "F"))";;

%----------------------------------------------------------------

Set_PSW (set_sm, clr_ie, pass, pass, pass, pass);;

Set_PSW (pass, pass, ld_from_alu, ld_vf, ld_nf, ld_zf);;

Set_PSW (pass, pass, ld_from_shifter, ld_vf, ld_nf, ld_zf);;

----------------------------------------------------------------%

%----------------------------------------------------------------
The external signals are described by a function ExtSig
----------------------------------------------------------------%
```

```
let rd = 1;;

let wr = 2;;

let no_mem_op = 3;;

let i_ack = "T";;

let off = "F";;

let in_fetch = "T";;

let Process_Mem_Op memop =
   (memop = 1) => "(T,F)" |
   (memop = 2) => "(F,T)" |
                  "(F,F)";;

let ExtSig (iaction,fetch,memop) =
  "(^iaction,
    ^fetch,
    ^(Process_Mem_Op memop))";;

%----------------------------------------------------------------

ExtSig(off,off,rd);;

ExtSig(i_ack,in_fetch,no_mem_op);;

----------------------------------------------------------------%

%----------------------------------------------------------------
The next micro instruction is chosen by the result of

  Mpc (cond, address)

The cond field can take the following values:

  Value     Meaning

  step      Increment the program counter and go there
  jmp       Jump unconditionally
  jop       Jump relative to mpc based on current opcode
  jint      Jump on interrupt
  jsm       Jump in supervisory mode

Step is the default.

----------------------------------------------------------------%

let step = "(F,F,F)";;

let jmp = "(F,F,T)";;

let jop = "(F,T,F)";;
```

```
let jint = "(F,T,T)";;

let jsm = "(T,F,F)";;

let Mpc (cond, addr) = "(^cond, ^addr)";;

let TEST_ADDR = "(F,F,F,F,F,F)";;

%----------------------------------------------------------------

Mpc(step,TEST_ADDR);;

Mpc(jint,TEST_ADDR);;

----------------------------------------------------------------%
```

3.5.2 The Microcode Definition

The section presents the ML code that creates the theory `ucode_def.th`.

```
%----------------------------------------------------------------

   File:         def_ucode.ml

   Author:       (c) P. J. Windley 1990

   Date:         JUN 23, 1990

   Modified:

   Description:

   Defines the microcode for the machine in an abstract way.
   A mnuemonic microassmbly langauge is defined.  The theorems
   necessary for assembling the code are proven.  The assembler is
   actually defined in the file that defines the actual microcode
   for the machine.

   In addition a type for the assembled microcode, the structure
   of the assembled microcode, and selectors on the assembled
   microcode are defined.


----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root) [`decimal/`;`assoc/`;`tuple/`]));;

system `/bin/rm ucode_def.th`;;

new_theory `ucode_def`;;

map new_parent [`tuple`];;

%----------------------------------------------------------------

A microinstruction has the following format:


 Bits  Mneumonic    Description

    1  AMUX         Toggle MUX on A-bus
    2  SHFT         Shifter function
```

```
    4  ALU          ALU function
    1  MAR          Load MAR from P-Mux
    1  MBR          Load MBR from C-bus
    1  PMUX         Toggle MUX loading MAR
    3  SRCA         A-bus source (includes SSP)
    2  SRCB         B-bus source
    3  TRGT         C-bus target (includes SSP)

    1  S_SM         Set supervisory mode bit in PSW
    1  C_SM         Clear supervisory mode bit in PSW
    1  S_IE         Set interrupt enable bit in PSW
    1  C_IE         Clear interrupt enable bit in PSW
    1  LD_C         Load carry bit in PSW
    1  LD_V         Load overflow bit in PSW
    1  LD_N         Load negative bit in PSW
    1  LD_Z         Load zero bit in PSW
    1  CSRC         Source of carry (shifter or alu)

    1  IACK         Interrupt acknowledge signal
    1  FTCH         Fetch signal
    1  RD           Read signal
    1  WR           Write signal

    3  COND         Microcode jump condition
    6  ADDR         Next address

----------------------------------------------------------------%

%----------------------------------------------------------------
 Load the ucode auxilliary file.
----------------------------------------------------------------%

loadf `ucode_aux`;;

%----------------------------------------------------------------
 Now define a type for ucode.
----------------------------------------------------------------%
new_type_abbrev(`ucode`,
   type_of
      "(^(Oper(reg_file,nsh,reg_file,add,ir,noreg)),
        ^(Set_PSW (pass, pass, ld_from_alu, ld_vf, ld_nf, ld_zf)),
        ^(ExtSig(i_ack,in_fetch,no_mem_op)),
        ^(Mpc(jint,TEST_ADDR)))"
   );;

%----------------------------------------------------------------
Here are the selectors for the microcode
----------------------------------------------------------------%

let Amux = new_definition
   (`Amux`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Amux ((ax,sh,al,mb,ma,pc,tg,sa,sb),
```

```
          (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
          (ia,f,r,w),
          (jc,ad)) = ax"
   );;

let Shift = new_definition
   (`Shift`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Shift ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = sh"
   );;

let Alu = new_definition
   (`Alu`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Alu   ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = al"
   );;

let Mbr = new_definition
   (`Mbr`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Mbr   ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = mb"
   );;

let Mar = new_definition
   (`Mar`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Mar   ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = ma"
   );;

let Pmux = new_definition
   (`Pmux`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Pmux   ((ax,sh,al,mb,ma,pc,tg,sa,sb),
```

```
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = pc"
   );;

let Trgt = new_definition
   (`Trgt`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Trgt  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = tg"
   );;

let SrcA = new_definition
   (`SrcA`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      SrcA  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = sa"
   );;

let SrcB = new_definition
   (`SrcB`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      SrcB  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = sb"
   );;


let S_sm = new_definition
   (`S_sm`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      S_sm  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = ssm"
   );;

let C_sm = new_definition
   (`C_sm`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
```

```
     C_sm  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = csm"
  );;

let S_ie = new_definition
  (`S_ie`,
   "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
     (ssm csm sie cie lcf lvf lnf lzf lal:bool)
     (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
     S_ie  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = sie"
  );;

let C_ie = new_definition
  (`C_ie`,
   "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
     (ssm csm sie cie lcf lvf lnf lzf lal:bool)
     (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
     C_ie  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = cie"
  );;

let Ld_c = new_definition
  (`Ld_c`,
   "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
     (ssm csm sie cie lcf lvf lnf lzf lal:bool)
     (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
     Ld_c  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = lcf"
  );;

let Ld_v = new_definition
  (`Ld_v`,
   "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
     (ssm csm sie cie lcf lvf lnf lzf lal:bool)
     (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
     Ld_v  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
            (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
            (ia,f,r,w),
            (jc,ad)) = lvf"
  );;

let Ld_n = new_definition
  (`Ld_n`,
   "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
     (ssm csm sie cie lcf lvf lnf lzf lal:bool)
     (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
```

```
      Ld_n  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = lnf"
   );;

let Ld_z = new_definition
   (`Ld_z`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Ld_z  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = lzf"
   );;

let Csrc = new_definition
   (`Csrc`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Csrc  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = lal"
   );;



let Iack = new_definition
   (`Iack`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Iack  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = ia"
   );;

let Ftch = new_definition
   (`Ftch`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Ftch  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = f"
   );;

let Rd = new_definition
   (`Rd`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
```

```
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Rd    ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = r"
   );;

let Wr = new_definition
   (`Wr`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Wr    ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = w"
   );;

let Cond = new_definition
   (`Cond`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Cond  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = jc"
  );;

let Address = new_definition
   (`Address`,
    "!(ax:bool) (sh:bt2) (al:bt4) (ma mb pc r w ia f:bool)
      (ssm csm sie cie lcf lvf lnf lzf lal:bool)
      (tg:bt3) (sa:bt3) (sb:bt2) (jc:bt3) (ad:bt6) .
      Address  ((ax,sh,al,mb,ma,pc,tg,sa,sb),
             (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal),
             (ia,f,r,w),
             (jc,ad)) = ad"
   );;

close_theory();;
```

3.5.3 The Phase–Level Interpreter

The section presents the ML code that creates the theory `phase_def.th`.

```
%----------------------------------------------------------------

   File:         def_phase.ml

   Author:       (c) P. J. Windley 1990

   Date:         18 JAN 90

   Modified:     06 MAY 90

   Description:

   Defines the behavioral description of the phase level
   interpreter.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`;`assoc/`]));;

loadf `abstract`;;

system `/bin/rm phase_def.th`;;

new_theory `phase_def`;;

map new_parent [`mpc_def`;`aux_def`;`tuple`;
                `aux_thms`;`regs_def`;`jump_def`;
                `ucode_def`];;

let rep_ty = abstract_type `aux_def` `opcode`;;

%----------------------------------------------------------------
 Denotational descriptions of phase level instructions.
----------------------------------------------------------------%
let phase_one_def = new_definition
   (`phase_one_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr alatch blatch:*wordn)
      (mpc:bt6) (clk:bt2) (urom:num->ucode) (mir:ucode)
      (ireq_ff iack_ff int_e:bool).
     phase_one rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
```

```
                  alatch, blatch, ireq_ff, iack_ff, mir, urom, clk)
                 (int_e) =
        let new_mir = urom (bt6_val mpc) and
            new_ireq_ff = int_e and
            new_clk = (F,T) in
        (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
         alatch, blatch, new_ireq_ff, iack_ff, new_mir, urom, new_clk)"
   );;

let phase_two_def = new_definition
   (`phase_two_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr alatch blatch:*wordn)
      (mpc:bt6) (clk:bt2) (urom:num->ucode) (mir:ucode)
      (ireq_ff iack_ff int_e:bool).
     phase_two rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                  alatch, blatch, ireq_ff, iack_ff, mir, urom, clk)
                 (int_e) =
        let new_alatch = (
            ((SrcA mir) = (F,F,F)) => (EL (reg_len rep (srca rep ir)) reg) |
            ((SrcA mir) = (F,F,T)) => (EL (reg_len rep (dest rep ir)) reg) |
            ((SrcA mir) = (F,T,F)) => (SSP_REG reg) |
            ((SrcA mir) = (F,T,T)) => psw |
            ((SrcA mir) = (T,F,F)) => (wordn rep 255) |
                                      pc) in
        let new_blatch = (
            ((SrcB mir) = (F,F)) => (EL (reg_len rep (srcb rep ir)) reg) |
            ((SrcB mir) = (F,T)) => (int_fetch rep ivec) |
                                    (imm rep ir)) in
        let new_iack_ff = Iack mir and
            new_clk = (T,F) in
        (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
         new_alatch, new_blatch, ireq_ff, new_iack_ff,
         mir, urom, new_clk)"
   );;

let phase_three_def = new_definition
   (`phase_three_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr alatch blatch:*wordn)
      (mpc:bt6) (clk:bt2) (urom:num->ucode) (mir:ucode)
      (ireq_ff iack_ff int_e:bool).
     phase_three rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                    alatch, blatch, ireq_ff, iack_ff, mir, urom, clk)
                   (int_e) =
       let new_mar = (((Pmux mir) /\ (Mar mir)) => pc | mar) and
           new_clk = (T,T) in
       (reg, psw, pc, mem, ivec, ir, new_mar, mbr, mpc,
        alatch, blatch, ireq_ff, iack_ff, mir, urom, new_clk)"
   );;

%-----------------------------------------------------------------
 A few auxilliary definitions
-----------------------------------------------------------------%
```

```
let ALU_FUNC = new_definition
   (`ALU_FUNC`,
    "! (rep:^rep_ty) s a_input blatch carry_in .
     ALU_FUNC rep s a_input blatch carry_in  =
      ((s = (F,F,F,F)) => (add  rep (a_input,blatch)) |
       (s = (F,F,F,T)) => (addc rep (a_input,blatch,carry_in)) |
       (s = (F,F,T,F)) => (inc  rep a_input) |
       (s = (F,F,T,T)) => (sub  rep (a_input,blatch)) |
       (s = (F,T,F,F)) => (subc rep (a_input,blatch,carry_in)) |
       (s = (F,T,F,T)) => (dec  rep a_input) |
       (s = (F,T,T,F)) => (band rep (a_input,blatch)) |
       (s = (F,T,T,T)) => (bxor rep (a_input,blatch)) |
       (s = (T,F,F,F)) => (bor  rep (a_input,blatch)) |
       (s = (T,F,F,T)) => (bnot rep a_input) |
                          a_input)"
   );;

let ALU_CARRY_FUNC = new_definition
   (`ALU_CARRY_FUNC`,
    "!(rep:^rep_ty) switch in_A in_B cin .
     ALU_CARRY_FUNC rep switch in_A in_B cin =
       ((switch = F,F,F,F) =>
          addp rep(in_A,in_B,add rep(in_A,in_B)) |
        (switch = F,F,F,T) =>
          addcp rep(in_A,in_B,addc rep(in_A,in_B,cin)) |
        (switch = F,F,T,F) =>
          addp rep(in_A,wordn rep 0,inc rep in_A) |
        (switch = F,F,T,T) =>
          subp rep(in_A,in_B,sub rep(in_A,in_B)) |
        (switch = F,T,F,F) =>
          subp rep(in_A,in_B,subc rep(in_A,in_B,cin)) |
        (switch = F,T,F,T) =>
          subp rep(in_A,wordn rep 0,dec rep in_A) |
          F)"
   );;

let ALU_OVFL_FUNC = new_definition
   (`ALU_OVFL_FUNC`,
    "!(rep:^rep_ty) switch in_A in_B cin .
     ALU_OVFL_FUNC rep switch in_A in_B cin =
       ((switch = F,F,F,F) =>
          aovfl rep(in_A,in_B,add rep(in_A,in_B)) |
        (switch = F,F,F,T) =>
          aovfl rep(in_A,in_B,addc rep(in_A,in_B,cin)) |
        (switch = F,F,T,F) =>
          F |
        (switch = F,F,T,T) =>
          sovfl rep(in_A,in_B,sub rep(in_A,in_B)) |
        (switch = F,T,F,F) =>
          sovfl rep(in_A,in_B,subc rep(in_A,in_B,cin)) |
          F)"
   );;

let ALU_NEG_FUNC = new_definition
   (`ALU_NEG_FUNC`,
```

```
    "!(rep:^rep_ty) switch in_A in_B cin .
     ALU_NEG_FUNC rep switch in_A in_B cin =
      negp rep
       ((switch = F,F,F,F) =>
           add rep(in_A,in_B) |
        (switch = F,F,F,T) =>
           addc rep(in_A,in_B,cin) |
        (switch = F,F,T,F) =>
           inc rep in_A |
        (switch = F,F,T,T) =>
           sub rep(in_A,in_B) |
        (switch = F,T,F,F) =>
           subc rep(in_A,in_B,cin) |
        (switch = F,T,F,T) =>
           dec rep in_A |
        (switch = F,T,T,F) =>
           band rep(in_A,in_B) |
        (switch = F,T,T,T) =>
           bxor rep(in_A,in_B) |
        (switch = T,F,F,F) =>
           bor rep(in_A,in_B) |
        (switch = T,F,F,T) =>
           bnot rep in_A | in_A)"
   );;

let ALU_ZERO_FUNC = new_definition
   (`ALU_ZERO_FUNC`,
    "!(rep:^rep_ty) switch in_A in_B cin .
     ALU_ZERO_FUNC rep switch in_A in_B cin =
      zerop rep
       ((switch = F,F,F,F) =>
           add rep(in_A,in_B) |
        (switch = F,F,F,T) =>
           addc rep(in_A,in_B,cin) |
        (switch = F,F,T,F) =>
           inc rep in_A |
        (switch = F,F,T,T) =>
           sub rep(in_A,in_B) |
        (switch = F,T,F,F) =>
           subc rep(in_A,in_B,cin) |
        (switch = F,T,F,T) =>
           dec rep in_A |
        (switch = F,T,T,F) =>
           band rep(in_A,in_B) |
        (switch = F,T,T,T) =>
           bxor rep(in_A,in_B) |
        (switch = T,F,F,F) =>
           bor rep(in_A,in_B) |
        (switch = T,F,F,T) =>
           bnot rep in_A | in_A)"
   );;

let SHIFTER_FUNC = new_definition
   (`SHIFTER_FUNC`,
    "!(rep:^rep_ty) switch in_A .
```

```
     SHIFTER_FUNC rep switch in_A =
      ((switch = F,F) =>
         shl rep in_A |
       (switch = F,T) =>
         shr rep in_A |
       (switch = T,F) =>
         asr rep in_A | in_A)"
   );;

let SHIFTER_CARRY_FUNC = new_definition
   (`SHIFTER_CARRY_FUNC`,
    "!(rep:^rep_ty) switch in_A .
     SHIFTER_CARRY_FUNC rep switch in_A =
      ((switch = F,F) =>
         msb rep in_A |
       (switch = F,T) =>
         lsb rep in_A |
       (switch = T,F) =>
         lsb rep in_A | F)"
   );;

let phase_four_def = new_definition
   (`phase_four_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr alatch blatch:*wordn)
      (mpc:bt6) (clk:bt2) (urom:num->ucode) (mir:ucode)
      (ireq_ff iack_ff int_e:bool).
     phase_four rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                     alatch, blatch, ireq_ff, iack_ff, mir, urom, clk)
                    (int_e) =
     let a_input = ((Amux mir) => mbr | alatch) in
     let carry_in = (get_cf rep psw) in
     let alu_result =
         ALU_FUNC rep (Alu mir) a_input blatch carry_in in
     let cf =
         ALU_CARRY_FUNC rep (Alu mir) a_input blatch carry_in in
     let vf =
         ALU_OVFL_FUNC rep (Alu mir) a_input blatch carry_in in
     let nf =
         ALU_NEG_FUNC rep (Alu mir) a_input blatch carry_in in
     let zf =
         ALU_ZERO_FUNC rep (Alu mir) a_input blatch carry_in in
     let result = SHIFTER_FUNC rep (Shift mir) alu_result in
     let shft_c = SHIFTER_CARRY_FUNC rep (Shift mir) alu_result in
     let opc  = (opcode rep ir) in
     let ie   = (get_ie rep psw) and
         sm   = (get_sm rep psw) in
     let new_psw = (
         (((Trgt mir) = (F,T,F)) /\ sm) => result |
            (mk_psw rep (
              ((S_sm mir) => T | (C_sm mir) => F | sm),
              ((S_ie mir) => T | (C_ie mir) => F | ie),
              ((Ld_v mir) => vf | (get_vf rep psw)),
              ((Ld_n mir) => nf | (get_nf rep psw)),
              ((Ld_c mir) => ((Csrc mir) => cf | shft_c) | (get_cf rep psw)),
```

```
            ((Ld_z mir) => zf | (get_zf rep psw))))) in
     let new_reg = (
         ((Trgt mir) = (F,F,F)) =>
               (UPDATE_REG rep psw (reg_len rep (dest rep ir)) reg result) |
         ((Trgt mir) = (F,F,T)) =>
               (UPDATE_REG rep psw ssp_reg reg result) |
               reg) in
     let new_mpc = (
          MPC_UNIT mpc opc (Address mir) (Cond mir) ireq_ff ie sm) in
     let new_ir = (((Trgt mir) = (F,T,T)) => result | ir) in
     let jmp = (JUMP_COND rep (reg_len rep (dest rep ir)) psw) in
     let new_pc = (
         ((Trgt mir) = (T,F,F)) => result |
         (((Trgt mir) = (T,F,T)) /\ jmp) => result | pc) in
     let new_mbr = (
         (Rd mir)  => (fetch rep (mem, address rep mar)) |
         (Mbr mir) => result |
                      mbr) in
     let new_mar = ((~(Pmux mir) /\ (Mar mir)) => result | mar) in
     let new_mem = ((Wr mir) => store rep (mem,address rep mar,mbr)
                             | mem) in
     let new_clk = (F,F) in
       (new_reg, new_psw, new_pc, new_mem, ivec, new_ir, new_mar,
        new_mbr, new_mpc, alatch, blatch, ireq_ff, iack_ff, mir,
        urom, new_clk)"
   );;


%----------------------------------------------------------------
 Selector function on phase level state for the phase level
 counter.
----------------------------------------------------------------%

let GetPhaseClock = new_definition
   (`GetPhaseClock`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr alatch blatch:*wordn)
      (mpc:bt6) (clk:bt2) (urom:num->ucode) (mir:ucode)
      (ireq_ff iack_ff int_e:bool).
     GetPhaseClock rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                        alatch, blatch, ireq_ff, iack_ff, mir, urom, clk)
                       (int_e) = clk"
   );;


let PhaseClockBegin = new_definition
   (`PhaseClockBegin`,
    "PhaseClockBegin = F,F"
   );;

%----------------------------------------------------------------
 Substate the phasestate to the micro state.
----------------------------------------------------------------%
let Phase_Substate = new_definition
   (`Phase_Substate`,
```

```
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr alatch blatch:*wordn)
      (mpc:bt6) (clk:bt2) (urom:num->ucode) (mir:ucode)
      (ireq_ff iack_ff int_e:bool).
     Phase_Substate rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc,
                         alatch, blatch, ireq_ff, iack_ff, mir, urom,
                         clk) =
                        (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc) "
   );;

%-----------------------------------------------------------------
 I serves as the substate funtion since the state
 of the phase level is equivalent to the phase of the EBM.

 I also serves as the subenv function since the set of external
 lines in the phase level is the same as the set of external
 lines in the EBM.
-----------------------------------------------------------------%

close_theory ();;
```

3.5.4 The Phase–Level Proof

The section presents the ML code that creates the theory `phase.th`.

```
%----------------------------------------------------------------

   File:         mk_phase.ml

   Author:       (c) P. J. Windley 1990

   Date:         18 JAN 90

   Modified:

   Description:

   Defines the phase level interpreter in terms of the definitions
   in block_def.th, phase_def.th, and gen_I.th.

   Proves the lemmas meeting the theory obligations for the abstract
   theory gen_I.th and instantiates a proof of the phase level in
   terms of the EBM.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`;`assoc/`]));;


loadf `abstract`;;

system `/bin/rm phase.th`;;

new_theory `phase`;;

map new_parent [`gen_I`;`phase_def`;`block_def`];;

let GetPhaseClock = definition `phase_def` `GetPhaseClock`;;

let phase_one_def = EXPAND_LET_RULE (
       definition `phase_def` `phase_one_def`);;

let phase_two_def = EXPAND_LET_RULE (
       definition `phase_def` `phase_two_def`);;

let phase_three_def = EXPAND_LET_RULE (
```

```
      definition `phase_def` `phase_three_def`);;

let ALU_FUNC = definition `phase_def` `ALU_FUNC`;;

let ALU_CARRY_FUNC = definition `phase_def` `ALU_CARRY_FUNC`;;

let ALU_OVFL_FUNC = definition `phase_def` `ALU_OVFL_FUNC`;;

let ALU_NEG_FUNC = definition `phase_def` `ALU_NEG_FUNC`;;

let ALU_ZERO_FUNC = definition `phase_def` `ALU_ZERO_FUNC`;;

let SHIFTER_FUNC = definition `phase_def` `SHIFTER_FUNC`;;

let SHIFTER_CARRY_FUNC = definition `phase_def` `SHIFTER_CARRY_FUNC`;;

let phase_four_def = definition `phase_def` `phase_four_def`;;

let phase_four_expanded = EXPAND_LET_RULE phase_four_def;;

let EBM_expanded =
    REWRITE_RULE [definition `block_def` `IVEC_SPEC`]
                 (theorem `block_def` `EBM_expanded`);;

let GetEBMClock = definition `block_def` `GetEBMClock`;;

let EBM_Start = definition `block_def` `EBM_Start`;;

let Next = definition `time_abs` `Next`;;

let Temp_Abs_DEGENERATE = theorem `time_abs` `Temp_Abs_DEGENERATE`;;

loadf `tuple`;;

map autoload_theory [`mpc_def`;`alu_def`;`shift_def`];;

let rep_ty = abstract_type `aux_def` `opcode`;;

let I_rep_ty = abstract_type `gen_I` `Impl`;;

let Phase_state =
    ":((*wordn)list#**wordn#**wordn#**memory#
       *wordn#**wordn#**wordn#**wordn#bt6#
       *wordn#**wordn#bool#bool#ucode#(num->ucode)#bt2)";;

let Phase_env = ":bool";;

let EBM_state = Phase_state;;

let EBM_env = Phase_env;;

%----------------------------------------------------------------
 Define the phase level interpeter in terms of the generic
 interpreter definition.
----------------------------------------------------------------%
```

```
let Phase_Int_def = new_definition
   (`Phase_Int_def`,
    "! (rep:^rep_ty) (s:time->^Phase_state) (e:time->^Phase_env) .
     Phase_Int rep s e =
        INTERP
           ([(F,F),phase_one rep;
             (F,T),phase_two rep;
             (T,F),phase_three rep;
             (T,T),phase_four rep],
            bt2_val,
            (GetPhaseClock rep:^Phase_state->^Phase_env->bt2),
            (I:^EBM_state->^Phase_state),
            (I:^EBM_env->^Phase_env), EBM rep,
            (GetEBMClock rep:^EBM_state->^EBM_env->bool),
            EBM_Start, @x:one.F) s e"
   );;

let Phase_Int = save_thm
   (`Phase_Int`,
    BETA_RULE (
    EXPAND_LET_RULE
       (instantiate_abstract_definition
           `gen_I`
           `INTERP`
           Phase_Int_def))
   );;


%-----------------------------------------------------------------
PHASE_Int =
|- !rep s e.
    Phase_I rep s e =
    (!t.
      s(t + 1) =
      SND
      (EL
       (bt2_val(GetPhaseClock(s t)(e t)))
       [(F,F),phase_one rep;(F,T),phase_two rep;(T,F),phase_three rep;
        (T,T),phase_four rep])
      (s t)
      (e t))
Run time: 84.5s
Intermediate theorems generated: 1527
-----------------------------------------------------------------%

let Phase_Int_Inst_Correct_def = new_definition
   (`Phase_Int_Inst_Correct_def`,
    "! (rep:^rep_ty) s' e'.
     Phase_Int_Inst_Correct rep s' e'  =
       INST_CORRECT
           ([(F,F),phase_one rep;
             (F,T),phase_two rep;
             (T,F),phase_three rep;
             (T,T),phase_four rep],
```

```
          bt2_val,
          (GetPhaseClock rep:^Phase_state->^Phase_env->bt2),
          (I:^EBM_state->^Phase_state),
          (I:^EBM_env->^Phase_env), EBM rep,
          (GetEBMClock rep:^EBM_state->^EBM_env->bool),
          EBM_Start, @x:one.F) s' e'"
   );;


let Phase_Int_Inst_Correct =
    let Phase_Int_EXT =
       CONV_RULE (TOP_DEPTH_CONV FUN_EQ_CONV) Phase_Int_Inst_Correct_def in
    (REWRITE_RULE [I_THM] (
     BETA_RULE (
     EXPAND_LET_RULE (
     instantiate_abstract_definition
           `gen_I`
           `INST_CORRECT`
           Phase_Int_EXT))));;

%----------------------------------------------------------------
Phase_Int_Inst_Correct =
|- !rep s' e' p.
    Phase_Int_Inst_Correct rep s' e' p =
    EBM rep s' e' ==>
    (!t.
      (GetPhaseClock rep(s' t)(e' t) = FST p) /\
      (GetEBMClock rep(s' t)(e' t) = EBM_Start) ==>
      (?c.
        Next(\t'. GetEBMClock rep(s' t')(e' t') = EBM_Start)(t,t + c) /\
        (SND p(s' t)(e' t) = s'(t + c))))
Run time: 203.9s
Intermediate theorems generated: 2744
----------------------------------------------------------------%

let NEXT_LEMMA = TAC_PROOF
   (([],
    "!t. t < (t + 1) /\ (!t'. ~(t < t' /\ t' < (t + 1)))"),
    REPEAT GEN_TAC
    THEN CONJ_TAC
    THENL [ % 1 %
        REWRITE_TAC [SYM_RULE ADD1;LESS_THM]
    ; % 2 %
        REWRITE_TAC [LESS_LESS_SUC;SYM_RULE ADD1]
    ]
   );;

let NOT_IF_LEMMA = TAC_PROOF
   (([],
    "! x y (a b c:*wordn) .
     ((~x /\ y) => (x => a | b)
                | c) =
     ((~x /\ y) => b | c)"),
    REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "x"
```

```
    THEN REWRITE_TAC []
   );;

let IF_OR_LEMMA = TAC_PROOF
   (([],
    "! x y (a b:*wordn) .
     (x => a |
      y => a | b) =
     ((x \/ y) => a | b)"),
    REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "x"
    THEN REWRITE_TAC []
   );;


%----------------------------------------------------------------
 Cause these to be read in now so that we can delete the cache.
----------------------------------------------------------------%
TWO_TUPLE_VALUE_LEMMA;;

THREE_TUPLE_VALUE_LEMMA;;

%----------------------------------------------------------------
 Get rid of some bulk
----------------------------------------------------------------%
let ALU_FUNC_LEMMA =
    REWRITE_RULE [SYM_RULE ALU_FUNC] MAC2_OUT_LEMMA;;

let ALU_CARRY_FUNC_LEMMA =
    REWRITE_RULE [SYM_RULE ALU_CARRY_FUNC] MAC2_CARRY_LEMMA;;

let ALU_OVFL_FUNC_LEMMA =
    REWRITE_RULE [SYM_RULE ALU_OVFL_FUNC] MAC2_OVFL_LEMMA;;

let ALU_NEG_FUNC_LEMMA =
    REWRITE_RULE [SYM_RULE ALU_NEG_FUNC] MAC2_NEG_LEMMA;;

let ALU_ZERO_FUNC_LEMMA =
    REWRITE_RULE [SYM_RULE ALU_ZERO_FUNC] MAC2_ZERO_LEMMA;;

let SHIFTER_FUNC_LEMMA =
    REWRITE_RULE [SYM_RULE SHIFTER_FUNC] SHIFTER_OUT_LEMMA;;

let SHIFTER_CARRY_FUNC_LEMMA =
    REWRITE_RULE [SYM_RULE SHIFTER_CARRY_FUNC] SHIFTER_CARRY_LEMMA;;

map (delete_cache o fst) (cached_theories());;

let PHASE_ONE_EBM_LEMMA = TAC_PROOF
   (([],
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode)
      (ireq_ff iack_ff ireq_e:time->bool).
```

```
     Phase_Int_Inst_Correct rep
           (\t.(reg t, psw t, pc t, mem t, ivec t,
                ir t, mar t, mbr t, mpc t,
                alatch t, blatch t, ireq_ff t,
                iack_ff t, mir t, urom, clk t))
           (\t. (ireq_e t))
           ((F,F),phase_one rep)"),
    PURE_ONCE_REWRITE_TAC [Phase_Int_Inst_Correct]
    THEN REPEAT GEN_TAC
    THEN BETA_TAC
    THEN REWRITE_TAC [GetPhaseClock;Next;
                      GetEBMClock;EBM_Start;phase_one_def;]
    THEN SUBST_TAC [EBM_expanded]
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM_LIST (\asl. (MAP_EVERY (STRIP_ASSUME_TAC o SPEC_ALL) asl))
    THEN EXISTS_TAC "1"
    THEN ASM_REWRITE_TAC [PAIR_EQ;NEXT_LEMMA]
);;

let PHASE_TWO_EBM_LEMMA = TAC_PROOF
   (([],
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode)
      (ireq_ff iack_ff ireq_e:time->bool).
     Phase_Int_Inst_Correct rep
           (\t.(reg t, psw t, pc t, mem t, ivec t,
                ir t, mar t, mbr t, mpc t,
                alatch t, blatch t, ireq_ff t,
                iack_ff t, mir t, urom, clk t))
           (\t. (ireq_e t))
           ((F,T),phase_two rep)"),
    PURE_ONCE_REWRITE_TAC [Phase_Int_Inst_Correct]
    THEN REPEAT GEN_TAC
    THEN BETA_TAC
    THEN REWRITE_TAC [GetPhaseClock;Next;
                      GetEBMClock;EBM_Start;phase_two_def;]
    THEN SUBST_TAC [EBM_expanded]
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM_LIST (\asl.
        MAP_EVERY (STRIP_ASSUME_TAC o SPEC_ALL) asl)
    THEN EXISTS_TAC "1"
    THEN ASM_REWRITE_TAC [PAIR_EQ;NEXT_LEMMA]
    THEN CONJ_TAC
    THENL [
        ASSUM_LIST (\asl .
            let find_aselect_term tm = (
                let (x,y) = (dest_eq tm) in
                (x = "(aselect t):bt3")) ? false in
            UNDISCH_TAC (concl (hd (filter ((find_aselect_term) o concl)
                                           asl))))
        THEN STRUCT_CASES_TAC (SPEC "SrcA(mir t):bt3" THREE_TUPLE_VALUE_LEMMA)
        THEN STRIP_TAC
        THEN POP_ASSUM_LIST (\asl .
```

```
        let find_aselect_term tm = (
            let (w,(y,(x,z))) = (I # (I # dest_eq))
                                ((I # dest_eq) (dest_forall tm)) in
            (x = "(aselect t):bt3")) ? false in
        let SPEC_t x = (SPEC "t:time" x) ? x in
        let aselect_list =
             (filter ((find_aselect_term) o concl) (tl asl)) in
        let rest = subtract (tl asl) aselect_list in
        let aselect_thms =
             map (REWRITE_RULE [hd asl;PAIR_EQ] o SPEC_ALL) aselect_list in
        MAP_EVERY
             (CHECK_ASSUME_TAC o (REWRITE_RULE aselect_thms) o SPEC_t)
             (rev rest)
        THEN MAP_EVERY ASSUME_TAC aselect_thms)
    THEN RES_TAC
    THEN ASM_REWRITE_TAC [PAIR_EQ]
;
    ASSUM_LIST (\asl .
        let find_bselect_term tm = (
            let (x,y) = (dest_eq tm) in
            (x = "(bselect t):bt2")) ? false in
        UNDISCH_TAC (concl (hd (filter ((find_bselect_term) o concl)
                                       asl))))
    THEN STRUCT_CASES_TAC (SPEC "SrcB(mir t):bt2" TWO_TUPLE_VALUE_LEMMA)
    THEN STRIP_TAC
    THEN POP_ASSUM_LIST (\asl .
        let find_bselect_term tm = (
            let (w,(y,(x,z))) = (I # (I # dest_eq))
                                ((I # dest_eq) (dest_forall tm)) in
            (x = "(bselect t):bt2")) ? false in
        let SPEC_t x = (SPEC "t:time" x) ? x in
        let bselect_list =
             (filter ((find_bselect_term) o concl) (tl asl)) in
        let rest = subtract (tl asl) bselect_list in
        let bselect_thms =
             map (REWRITE_RULE [hd asl;PAIR_EQ] o SPEC_ALL)
                 bselect_list in
        MAP_EVERY
             (CHECK_ASSUME_TAC o (REWRITE_RULE bselect_thms) o SPEC_t)
             (rev rest))
    THEN RES_TAC
    THEN ASM_REWRITE_TAC [PAIR_EQ]
]
);;

let PHASE_THREE_EBM_LEMMA = TAC_PROOF
   (([],
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode)
      (ireq_ff iack_ff ireq_e:time->bool).
     Phase_Int_Inst_Correct rep
          (\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
```

```
             alatch t, blatch t, ireq_ff t,
             iack_ff t, mir t, urom, clk t))
        (\t. (ireq_e t))
        ((T,F),phase_three rep)"),
    PURE_ONCE_REWRITE_TAC [Phase_Int_Inst_Correct]
    THEN REPEAT GEN_TAC
    THEN BETA_TAC
    THEN REWRITE_TAC [GetPhaseClock;Next;
                      GetEBMClock;EBM_Start;phase_three_def;]
    THEN SUBST_TAC [EBM_expanded]
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM_LIST (\asl. (MAP_EVERY (STRIP_ASSUME_TAC o SPEC_ALL) asl))
    THEN EXISTS_TAC "1"
    THEN ASM_REWRITE_TAC [PAIR_EQ;NEXT_LEMMA]
    THEN BOOL_CASES_TAC "Pmux(mir t):bool"
    THEN REWRITE_TAC []
   );;

let PHASE_FOUR_EBM_LEMMA = TAC_PROOF
   (([],
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode)
      (ireq_ff iack_ff ireq_e:time->bool).
     Phase_Int_Inst_Correct rep
        (\t.(reg t, psw t, pc t, mem t, ivec t,
             ir t, mar t, mbr t, mpc t,
             alatch t, blatch t, ireq_ff t,
             iack_ff t, mir t, urom, clk t))
        (\t. (ireq_e t))
        ((T,T),phase_four rep)"),
    PURE_ONCE_REWRITE_TAC [Phase_Int_Inst_Correct]
    THEN REPEAT GEN_TAC
    THEN BETA_TAC
    THEN REWRITE_TAC [Next;
                      GetPhaseClock;
                      GetEBMClock;EBM_Start]
    THEN SUBST_TAC [EBM_expanded]
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM_LIST (\asl. (MAP_EVERY (STRIP_ASSUME_TAC o SPEC_ALL) asl))
    THEN EXISTS_TAC "1"
    THEN FIRST_ASSUM
        (\thm. (ASSUME_TAC (MATCH_MP ALU_FUNC_LEMMA thm)) ? NO_TAC)
    THEN FIRST_ASSUM
        (\thm. (ASSUME_TAC (MATCH_MP SHIFTER_FUNC_LEMMA thm)) ? NO_TAC)
    THEN FIRST_ASSUM
        (\thm. (ASSUME_TAC (MATCH_MP ALU_NEG_FUNC_LEMMA thm)) ? NO_TAC)
    THEN FIRST_ASSUM
        (\thm. (ASSUME_TAC (MATCH_MP ALU_ZERO_FUNC_LEMMA thm)) ? NO_TAC)
    THEN FIRST_ASSUM
        (\thm. (ASSUME_TAC (MATCH_MP ALU_CARRY_FUNC_LEMMA thm)) ? NO_TAC)
    THEN FIRST_ASSUM
        (\thm. (ASSUME_TAC (MATCH_MP ALU_OVFL_FUNC_LEMMA thm)) ? NO_TAC)
    THEN FIRST_ASSUM
```

```
        (\thm. (ASSUME_TAC
                (MATCH_MP SHIFTER_CARRY_FUNC_LEMMA thm)) ? NO_TAC)
    THEN ASM_REWRITE_TAC [PAIR_EQ;NEXT_LEMMA;
                          phase_four_expanded;
                          IF_OR_LEMMA;NOT_IF_LEMMA]
    THEN REPEAT CONJ_TAC
    THENL [ % 1 %
        ASM_CASES_TAC "Wr(mir t):bool"
        THENL [ % 1.1 %
            POP_ASSUM (\thm1 .
               FIRST_ASSUM (\thm2 . (
                  ASSUME_TAC (
                    EQ_MP (SYM_RULE thm2) thm1)) ? NO_TAC))
            THEN RES_TAC
        ; % 1.2 %
            ALL_TAC
        ]
    ; % 2 %
        ASM_CASES_TAC "Rd(mir t):bool"
        THENL [ % 1.1 %
            POP_ASSUM (\thm1 .
               FIRST_ASSUM (\thm2 . (
                  ASSUME_TAC (
                    EQ_MP (SYM_RULE thm2) thm1)) ? NO_TAC))
            THEN RES_TAC
        ; % 1.2 %
            ALL_TAC
        ]
    ]
    THEN ASM_REWRITE_TAC []
   );;


%----------------------------------------------------------------
 The first obligation of the abstract interpreter theory
----------------------------------------------------------------%
let Phase_Int_Correct_LEMMA_AUX = TAC_PROOF
   (([],
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode)
      (mir:time->ucode)
      (ireq_ff iack_ff ireq_e:time->bool).
      EVERY (Phase_Int_Inst_Correct rep
              (\t.(reg t, psw t, pc t, mem t, ivec t,
                   ir t, mar t, mbr t, mpc t,
                   alatch t, blatch t, ireq_ff t,
                   iack_ff t, mir t, urom, clk t))
              (\t. (ireq_e t)))
               [(F,F),phase_one rep;
                (F,T),phase_two rep;
                (T,F),phase_three rep;
                (T,T),phase_four rep]"),
    REWRITE_TAC [EVERY_DEF]
    THEN REPEAT STRIP_TAC
```

```
    THEN FIRST [
        MATCH_ACCEPT_TAC PHASE_ONE_EBM_LEMMA;
        MATCH_ACCEPT_TAC PHASE_TWO_EBM_LEMMA;
        MATCH_ACCEPT_TAC PHASE_THREE_EBM_LEMMA;
        MATCH_ACCEPT_TAC PHASE_FOUR_EBM_LEMMA
    ]
   );;

let Phase_Int_Correct_LEMMA = (
    SPEC_ALL (
    PURE_ONCE_REWRITE_RULE [Phase_Int_Inst_Correct_def]  (
        Phase_Int_Correct_LEMMA_AUX)));;

%----------------------------------------------------------------
 The second obligation of the abstract interpreter theory
----------------------------------------------------------------%
let Phase_Int_LENGTH_LEMMA = TAC_PROOF
   (([],
    "! clk:bt2. bt2_val clk < (LENGTH [(F,F),phase_one (rep:^rep_ty);
                                       (F,T),phase_two rep;
                                       (T,F),phase_three rep;
                                       (T,T),phase_four rep])"),
    MATCH_ACCEPT_TAC bt2_LENGTH_LEMMA
   );;

%----------------------------------------------------------------
 The third obligation of the abstract interpreter theory
----------------------------------------------------------------%
let Phase_Int_ORDER_LEMMA = TAC_PROOF
   (([],
    "!clk:bt2 . clk = (FST (EL (bt2_val clk) [(F,F),phase_one (rep:^rep_ty);
                                              (F,T),phase_two rep;
                                              (T,F),phase_three rep;
                                              (T,T),phase_four rep]))"),
    REPEAT GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "clk:bt2" TWO_TUPLE_VALUE_LEMMA)
    THEN PURE_ONCE_REWRITE_TAC [bt2_val]
    THEN CONV_TAC (TOP_DEPTH_CONV num_CONV)
    THEN REWRITE_TAC [EL;FST;HD;TL]
   );;

%----------------------------------------------------------------
 Get the instantiation
----------------------------------------------------------------%
let theorem_list =
    instantiate_abstract_theorems
        `gen_I`
        [Phase_Int_Correct_LEMMA;
         Phase_Int_LENGTH_LEMMA;
         Phase_Int_ORDER_LEMMA]
        [
         ("rep:^I_rep_ty",
          "([(F,F),phase_one (rep:^rep_ty);
             (F,T),phase_two rep;
             (T,F),phase_three rep;
```

```
            (T,T),phase_four rep],
           bt2_val,
           GetPhaseClock rep:^Phase_state->^Phase_env->bt2,
           I:^EBM_state->^Phase_state,
           I:^EBM_env->^Phase_env,
           EBM rep,
           GetEBMClock rep:^EBM_state->^EBM_env->bool, EBM_Start)");
        ("e':time'->*env'",
         "(\t. (ireq_e t)):time->^EBM_env");
        ("s':time->*state'",
         "(\t.(reg t, psw t, pc t, mem t, ivec t,
               ir t, mar t, mbr t, mpc t,
               alatch t, blatch t, ireq_ff t,
               iack_ff t, mir t, urom, clk t)):time->^EBM_state");
      ]
     `PHASE`;;


%----------------------------------------------------------------
yields...

theorem_list =
[(`PHASE_IMPL_I_CORRECT`,
  |- let s t =
          I
          ((\t'.
             (reg t',psw t',pc t',mem t',ivec t',ir t',mar t',mbr t',
              mpc t',alatch t',blatch t',ireq_ff t',iack_ff t',mir t',
              urom,clk t'))
           t)
     and e t = I((\t'. (ireq_e t' t'))t)
     and f t =
          (GetEBMClock
           rep
           ((\t'.
              (reg t',psw t',pc t',mem t',ivec t',ir t',mar t',mbr t',
               mpc t',alatch t',blatch t',ireq_ff t',iack_ff t',mir t',
               urom,clk t'))
            t)
           ((\t'. (ireq_e t' t'))t) =
           EBM_Start)
     in
      let abs = Temp_Abs f
      in
       (EBM
        rep
        (\t.
          (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t,
           alatch t,blatch t,ireq_ff t,iack_ff t,mir t,urom,clk t))
        (\t. (ireq_e t t)) /\
        (?t. f t) ==>
        INTERP
        ([(F,F),phase_one rep;(F,T),phase_two rep;(T,F),phase_three rep;
          (T,T),phase_four rep],bt2_val,GetPhaseClock rep,I,I,EBM rep,
         GetEBMClock rep,EBM_Start,(@x. F))
```

```
        (s o abs)
        (e o abs)))]
: (string # thm) list
Run time: 526.4s
Intermediate theorems generated: 3903
-----------------------------------------------------------------%

let correct_lemma = snd(hd theorem_list);;

let TRUTH_EXISTS = TAC_PROOF
   (([],
    "?t:time.T"),
    EXISTS_TAC "0"
    THEN REWRITE_TAC []
   );;


%-----------------------------------------------------------------
 Rewrite the correctness lemma into a prettier form.
-----------------------------------------------------------------%
let PHASE_LEVEL_CORRECT_LEMMA = save_thm
   (`PHASE_LEVEL_CORRECT_LEMMA`,
    REWRITE_RULE [I_o_ID; Temp_Abs_DEGENERATE;TRUTH_EXISTS] (
    EXPAND_LET_RULE (
    ONCE_REWRITE_RULE [GetEBMClock;EBM_Start;I_THM] (
    BETA_RULE (
    ONCE_REWRITE_RULE [SYM_RULE Phase_Int_def] correct_lemma))))
   );;



%-----------------------------------------------------------------
yields...

PHASE_LEVEL_CORRECT_LEMMA =
|- EBM
   rep
   (\t.
     (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t,alatch t,
      blatch t,ireq_ff t,iack_ff t,mir t,urom,clk t))
   (\t. (ireq_e t t)) ==>
   Phase_Int
   rep
   (\t.
     (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t,alatch t,
      blatch t,ireq_ff t,iack_ff t,mir t,urom,clk t))
   (\t. (ireq_e t t))
Run time: 346.7s
Intermediate theorems generated: 4769
-----------------------------------------------------------------%
```

3.6 The Micro–Level

This section presents the theories that define the micro–level interpreter. Also presented is the theory that verifies the micro–level interpreter with respect to the phase–level interpreter.

3.6.1 The Micro–Level Interpreter

The section presents the ML code that creates the theory `micro_def.th`.

```
%----------------------------------------------------------------

   File:         def_micro.ml

   Author:       (c) P. J. Windley 1989

   Date:         05 APR 90

   Description:

   Defines the behavioral description of the micro interpreter
   level

   Modified:

   12 APR 90 -- Changed DECODE to use only 5 lsb in opcode.

----------------------------------------------------------------%
set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;


let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`numbers/`; `decimal/`; `assoc/`; `tuple/`]));;

loadf `abstract`;;

system `/bin/rm micro_def.th`;;

new_theory `micro_def`;;

map new_parent [`tuple`];;

map new_parent [`aux_def`; `aux_thms`; `regs_def`; `jump_def`];;

let rep_ty = abstract_type `aux_def` `opcode`;;

%----------------------------------------------------------------
```

```
 If you change these addresses, change the list in def_uinst.ml
 as well.
----------------------------------------------------------------%
let FETCH_ADDR   = "(F,F,F,F,F,F)";;

let CALL_u2_ADDR = "(T,F,F,T,F,F)";;

let CALL_u3_ADDR = "(T,F,F,T,F,T)";;

let CALL_u4_ADDR = "(T,F,F,T,T,F)";;

let INT_u2_ADDR  = "(T,F,F,T,T,T)";;

let INT_u3_ADDR  = "(T,F,T,F,F,F)";;

let INT_u4_ADDR  = "(T,F,T,F,F,T)";;

let RTI_u2_ADDR  = "(T,F,T,F,T,F)";;

let RTI_u3_ADDR  = "(T,F,T,F,T,T)";;

let RTN_u2_ADDR  = "(T,F,T,T,F,F)";;

let LD_u2_ADDR   = "(T,F,T,T,F,T)";;

let ST_u2_ADDR   = "(T,F,T,T,T,F)";;

let ST_u3_ADDR   = "(T,F,T,T,T,T)";;

let STI_u2_ADDR  = "(T,T,F,F,F,F)";;

let EINT_u1_ADDR = "(T,T,F,F,F,T)";;

let EINT_u2_ADDR = "(T,T,F,F,T,F)";;

let EINT_u3_ADDR = "(T,T,F,F,T,T)";;

let EINT_u4_ADDR = "(T,T,F,T,F,F)";;

let LD_u3_ADDR = "(T,T,F,T,F,T)";;


%----------------------------------------------------------------
 Micro instruction 0: fetch
----------------------------------------------------------------%
let FETCH = new_definition
   (`FETCH_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
     FETCH rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
               (int_e) =
           (reg, psw, pc, mem, ivec, ir,
            pc,
```

```
           fetch rep (mem, address rep pc),
           ((int_e /\ (get_ie rep psw)) => ^EINT_u1_ADDR | add_bt6 mpc 1))"
   );;

save_thm(`FETCH`,EXPAND_LET_RULE FETCH);;

%----------------------------------------------------------------
 Micro instruction 1: issue
----------------------------------------------------------------%
let ISSUE = new_definition
   (`ISSUE_def`,
    "!(rep:^rep_ty) reg mem
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
     ISSUE rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
               (int_e) =
           (reg, psw, pc, mem, ivec, mbr,
            mar, mbr, add_bt6 mpc 1)"
   );;

save_thm(`ISSUE`,EXPAND_LET_RULE ISSUE);;

%----------------------------------------------------------------
 Micro instruction 2: decode
----------------------------------------------------------------%
let DECODE = new_definition
   (`DECODE_def`,
    "!(rep:^rep_ty) reg mem
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
     DECODE rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                (int_e) =
           (reg, psw, inc rep pc, mem, ivec, ir,
            mar, mbr, add_bt6 (F,(SND(opcode rep ir))) 4)"
   );;

save_thm(`DECODE`,EXPAND_LET_RULE DECODE);;

%----------------------------------------------------------------
 table entry  0: first uinst for JMP
----------------------------------------------------------------%
let JMP_u1 = new_definition
   (`JMP_u1_def`,
    "!(rep:^rep_ty) reg mem
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      JMP_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir and
               d = reg_len rep (dest rep ir) in
           let result = add rep (a, i) in
           let jump_cond = JUMP_COND rep d psw in
           (reg, psw,
            (jump_cond => result | pc),
```

```
           mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`JMP_u1`,EXPAND_LET_RULE JMP_u1);;

%----------------------------------------------------------------
 table entry  1: first uinst for CALL
----------------------------------------------------------------%
let CALL_u1 = new_definition
   (`CALL_u1_def`,
    "!(rep:^rep_ty) reg mem
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      CALL_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           (reg, psw, pc,
            mem, ivec, ir, mar, pc, ^CALL_u2_ADDR)"
   );;

save_thm(`CALL_u1`,EXPAND_LET_RULE CALL_u1);;

let CALL_u2 = new_definition
   (`CALL_u2_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) mem
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      CALL_u2 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let d = reg_len rep (dest rep ir) in
           (reg, psw, pc,
            mem, ivec, ir, EL d reg, mbr, ^CALL_u3_ADDR)"
   );;

save_thm(`CALL_u2`,EXPAND_LET_RULE CALL_u2);;

let CALL_u3 = new_definition
   (`CALL_u3_def`,
    "!(rep:^rep_ty) reg mem
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      CALL_u3 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir in
           let result = add rep (a, i) in
           (reg, psw, result,
            store rep (mem, address rep mar, mbr),
            ivec, ir, mar, mbr, ^CALL_u4_ADDR)"
   );;

save_thm(`CALL_u3`,EXPAND_LET_RULE CALL_u3);;

let CALL_u4 = new_definition
   (`CALL_u4_def`,
    "!(rep:^rep_ty) reg mem
```

```
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      CALL_u4 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let d = reg_len rep (dest rep ir) in
           let result = inc rep (EL d reg) in
           (UPDATE_REG rep psw d reg result, psw, pc, mem,
            ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`CALL_u4`,EXPAND_LET_RULE CALL_u4);;

%----------------------------------------------------------------
 table entry  2: first uinst for INT
----------------------------------------------------------------%
let INT_u1 = new_definition
   (`INT_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      INT_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let cflag  = get_cf rep psw and
               vflag  = get_vf rep psw and
               nflag  = get_nf rep psw and
               zflag  = get_zf rep psw and
               sm     = T and
               ie     = F in
           (reg,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, pc, ^INT_u2_ADDR)"
   );;

save_thm(`INT_u1`,EXPAND_LET_RULE INT_u1);;

let INT_u2 = new_definition
   (`INT_u2_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      INT_u2 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           (reg, psw, pc,
            mem, ivec, ir, SSP_REG reg, mbr, ^INT_u3_ADDR)"
   );;

save_thm(`INT_u2`,EXPAND_LET_RULE INT_u2);;

let INT_u3 = new_definition
   (`INT_u3_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      INT_u3 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
```

```
          let result = inc rep (SSP_REG reg) in
          (UPDATE_REG rep psw ssp_reg reg result,
           psw, pc, mem, ivec, ir, mar, mbr, ^INT_u4_ADDR)"
   );;

save_thm(`INT_u3`,EXPAND_LET_RULE INT_u3);;

let INT_u4 = new_definition
   (`INT_u4_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      INT_u4 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
          let i = imm rep ir in
          let result = band rep (wordn rep 255, i) in
          (reg, psw, result,
           store rep (mem, address rep mar, mbr),
           ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`INT_u4`,EXPAND_LET_RULE INT_u4);;

%----------------------------------------------------------------
 table entry  3: first uinst for RTI
----------------------------------------------------------------%
let RTI_u1 = new_definition
   (`RTI_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      RTI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
          let result = dec rep (SSP_REG reg) in
          (UPDATE_REG rep psw ssp_reg reg result,
           psw, pc, mem, ivec, ir, result, mbr, ^RTI_u2_ADDR)"
   );;

save_thm(`RTI_u1`,EXPAND_LET_RULE RTI_u1);;

let RTI_u2 = new_definition
   (`RTI_u2_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      RTI_u2 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
          let cflag  = get_cf rep psw and
              vflag  = get_vf rep psw and
              nflag  = get_nf rep psw and
              zflag  = get_zf rep psw and
              sm     = F and
              ie     = T in
          (reg,
           mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
```

```
           pc, mem, ivec, ir, mar,
           fetch rep (mem, address rep mar), ^RTI_u3_ADDR)"
   );;

save_thm(`RTI_u2`,EXPAND_LET_RULE RTI_u2);;

let RTI_u3 = new_definition
   (`RTI_u3_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      RTI_u3 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
          (reg, psw, mbr, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`RTI_u3`,EXPAND_LET_RULE RTI_u3);;


%----------------------------------------------------------------
 table entry  4: first uinst for GPSW
----------------------------------------------------------------%
let GPSW_u1 = new_definition
   (`GPSW_u1_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      GPSW_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                (int_e) =
          let d = reg_len rep (dest rep ir) in
          (UPDATE_REG rep psw d reg psw,
           psw, pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`GPSW_u1`,EXPAND_LET_RULE GPSW_u1);;

%----------------------------------------------------------------
 table entry  5: first uinst for PPSW
----------------------------------------------------------------%
let PPSW_u1 = new_definition
   (`PPSW_u1_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      PPSW_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
          let d = reg_len rep (dest rep ir) and
              sm = get_sm rep psw in
          (reg,
           (sm => (EL d reg) | psw),
           pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`PPSW_u1`,EXPAND_LET_RULE PPSW_u1);;

%----------------------------------------------------------------
 table entry  6: first uinst for LD
```

```
 ---------------------------------------------------------------%
let LD_u1 = new_definition
   (`LD_u1_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      LD_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               b = EL (reg_len rep (srcb rep ir)) reg in
           let result = add rep (a, b) in
           (reg, psw, pc, mem, ivec, ir, result, mbr, ^LD_u2_ADDR)"
   );;

save_thm(`LD_u1`,EXPAND_LET_RULE LD_u1);;

%----------------------------------------------------------------
 LD_u2_ADDR: second uinst for LD.
 ---------------------------------------------------------------%
let LD_u2 = new_definition
   (`LD_u2_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      LD_u2 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                (int_e) =
           (reg, psw, pc, mem, ivec, ir, mar,
            fetch rep (mem, address rep mar), ^LD_u3_ADDR)"
   );;

save_thm(`LD_u2`,EXPAND_LET_RULE LD_u2);;

%----------------------------------------------------------------
 LD_u3_ADDR: third uinst for LD.
 ---------------------------------------------------------------%
let LD_u3 = new_definition
   (`LD_u3_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      LD_u3 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                (int_e) =
           let d = reg_len rep (dest rep ir) in
           (UPDATE_REG rep psw d reg mbr,
            psw, pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`LD_u3`,EXPAND_LET_RULE LD_u3);;

%----------------------------------------------------------------
 table entry  7: first uinst for ST
 ---------------------------------------------------------------%
let ST_u1 = new_definition
   (`ST_u1_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      ST_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                (int_e) =
```

```
          let a = EL (reg_len rep (srca rep ir)) reg and
              b = EL (reg_len rep (srcb rep ir)) reg in
          let result = add rep (a, b) in
          (reg, psw, pc, mem, ivec, ir, result, mbr, ^ST_u2_ADDR)"
   );;

save_thm(`ST_u1`,EXPAND_LET_RULE ST_u1);;

%----------------------------------------------------------------
 ST_u2_ADDR: second uinst for ST.
----------------------------------------------------------------%
let ST_u2 = new_definition
   (`ST_u2_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      ST_u2 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                (int_e) =
          let d = reg_len rep (dest rep ir) in
          (reg, psw, pc, mem, ivec, ir, mar, EL d reg, ^ST_u3_ADDR)"
   );;

save_thm(`ST_u2`,EXPAND_LET_RULE ST_u2);;

%----------------------------------------------------------------
 ST_u3_ADDR: third uinst for ST.
----------------------------------------------------------------%
let ST_u3 = new_definition
   (`ST_u3_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      ST_u3 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                (int_e) =
          (reg, psw, pc, store rep (mem, address rep mar, mbr),
           ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`ST_u3`,EXPAND_LET_RULE ST_u3);;


%----------------------------------------------------------------
 table entry  8: first uinst for LSL
----------------------------------------------------------------%
let LSL_u1 = new_definition
   (`LSL_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list)
      (mem:*memory) (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      LSL_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
          let a = EL (reg_len rep (srca rep ir)) reg and
              d = reg_len rep (dest rep ir) in
          let result = shl rep a in
          let cflag  = msb rep a and
              vflag  = get_vf rep psw and
```

```
              nflag  = get_nf rep psw and
              zflag  = get_zf rep psw and
              sm     = get_sm rep psw and
              ie     = get_ie rep psw in
          (UPDATE_REG rep psw d reg result,
           mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
           pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`LSL_u1`,EXPAND_LET_RULE LSL_u1);;

%----------------------------------------------------------------
 table entry  9: first uinst for LSR
----------------------------------------------------------------%
let LSR_u1 = new_definition
   (`LSR_u1_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      LSR_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
          let a = EL (reg_len rep (srca rep ir)) reg and
              d = reg_len rep (dest rep ir) in
          let result = shr rep a in
          let cflag  = lsb rep a and
              vflag  = get_vf rep psw and
              nflag  = get_nf rep psw and
              zflag  = get_zf rep psw and
              sm     = get_sm rep psw and
              ie     = get_ie rep psw in
          (UPDATE_REG rep psw d reg result,
           mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
           pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`LSR_u1`,EXPAND_LET_RULE LSR_u1);;

%----------------------------------------------------------------
 table entry 10: first uinst for ASR
----------------------------------------------------------------%
let ASR_u1 = new_definition
   (`ASR_u1_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      ASR_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
          let a = EL (reg_len rep (srca rep ir)) reg and
              d = reg_len rep (dest rep ir) in
          let result = asr rep a in
          let cflag  = lsb rep a and
              vflag  = get_vf rep psw and
              nflag  = get_nf rep psw and
              zflag  = get_zf rep psw and
              sm     = get_sm rep psw and
              ie     = get_ie rep psw in
          (UPDATE_REG rep psw d reg result,
```

```
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`ASR_u1`,EXPAND_LET_RULE ASR_u1);;

%----------------------------------------------------------------
 table entry 11: first uinst for RTN
----------------------------------------------------------------%
let RTN_u1 = new_definition
   (`RTN_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      RTN_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let d = reg_len rep (dest rep ir) in
           let result = dec rep (EL d reg) in
           (UPDATE_REG rep psw d reg result,
            psw, pc, mem, ivec, ir, result, mbr, ^RTN_u2_ADDR)"
   );;

save_thm(`RTN_u1`,EXPAND_LET_RULE RTN_u1);;

%----------------------------------------------------------------
 Return through RTI_u3 to transfer mbr to pc.
----------------------------------------------------------------%
let RTN_u2 = new_definition
   (`RTN_u2_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      RTN_u2 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           (reg, psw,
            pc, mem, ivec, ir, mar,
            fetch rep (mem, address rep mar), ^RTI_u3_ADDR)"
   );;

save_thm(`RTN_u2`,EXPAND_LET_RULE RTN_u2);;

%----------------------------------------------------------------
 table entry 12: first uinst for NOOP (used to fill urom)
----------------------------------------------------------------%
let NOOP_u1 = new_definition
   (`NOOP_u1_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      NOOP_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           (reg, psw, pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`NOOP_u1`,EXPAND_LET_RULE NOOP_u1);;
```

```
%---------------------------------------------------------------
 table entry 13: first uinst for NOOP (already defined)
---------------------------------------------------------------%

%---------------------------------------------------------------
 table entry 14: first uinst for LDI
 Jump to LD_u2_ADDR for second instruction of LDI
---------------------------------------------------------------%
let LDI_u1 = new_definition
   (`LDI_u1_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      LDI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir in
           let result = add rep (a, i) in
           (reg, psw, pc, mem, ivec, ir, result, mbr, ^LD_u2_ADDR)"
   );;

save_thm(`LDI_u1`,EXPAND_LET_RULE LDI_u1);;

%---------------------------------------------------------------
 table entry 15: first uinst for STI
---------------------------------------------------------------%
let STI_u1 = new_definition
   (`STI_u1_def`,
    "!(rep:^rep_ty) reg mem (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      STI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir in
           let result = add rep (a, i) in
           (reg, psw, pc, mem, ivec, ir, result, mbr, ^STI_u2_ADDR)"
   );;

save_thm(`STI_u1`,EXPAND_LET_RULE STI_u1);;

%---------------------------------------------------------------
 STI_u2_ADDR: second uinst for STI.
---------------------------------------------------------------%
let STI_u2 = new_definition
   (`STI_u2_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      STI_u2 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let d = reg_len rep (dest rep ir) in
           (reg, psw, pc, mem, ivec, ir, mar, EL d reg, ^ST_u3_ADDR)"
   );;

save_thm(`STI_u2`,EXPAND_LET_RULE STI_u2);;
```

```
%---------------------------------------------------------------
 table entry 16: first uinst for ADD
---------------------------------------------------------------%
let ADD_u1 = new_definition
   (`ADD_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      ADD_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               b = EL (reg_len rep (srcb rep ir)) reg and
               d = reg_len rep (dest rep ir) in
           let result = (add rep (a, b)) in
           let cflag  = addp rep (a, b, result) and
               vflag  = aovfl rep (a, b, result) and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`ADD_u1`,EXPAND_LET_RULE ADD_u1);;

%---------------------------------------------------------------
 table entry 17: first uinst for ADDC
---------------------------------------------------------------%
let ADDC_u1 = new_definition
   (`ADDC_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      ADDC_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               b = EL (reg_len rep (srcb rep ir)) reg and
               d = reg_len rep (dest rep ir) in
           let result = (addc rep (a, b, get_cf rep psw)) in
           let cflag  = addcp rep (a, b, result) and
               vflag  = aovfl rep (a, b, result) and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`ADDC_u1`,EXPAND_LET_RULE ADDC_u1);;

%---------------------------------------------------------------
```

```
 table entry 18: first uinst for SUB
----------------------------------------------------------------%
let SUB_u1 = new_definition
   (`SUB_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      SUB_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               b = EL (reg_len rep (srcb rep ir)) reg and
               d = reg_len rep (dest rep ir) in
           let result = (sub rep (a, b)) in
           let cflag  = subp rep (a, b, result) and
               vflag  = sovfl rep (a, b, result) and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`SUB_u1`,EXPAND_LET_RULE SUB_u1);;

%----------------------------------------------------------------
 table entry 19: first uinst for SUBC
----------------------------------------------------------------%
let SUBC_u1 = new_definition
   (`SUBC_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      SUBC_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               b = EL (reg_len rep (srcb rep ir)) reg and
               d = reg_len rep (dest rep ir) in
           let result = (subc rep (a, b, get_cf rep psw)) in
           let cflag  = subp rep (a, b, result) and
               vflag  = sovfl rep (a, b, result) and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`SUBC_u1`,EXPAND_LET_RULE SUBC_u1);;

%----------------------------------------------------------------
 table entry 20: first uinst for BAND
```

```
---------------------------------------------------------------%
let BAND_u1 = new_definition
   (`BAND_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      BAND_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               b = EL (reg_len rep (srcb rep ir)) reg and
               d = reg_len rep (dest rep ir) in
           let result = (band rep (a, b)) in
           let cflag  = get_cf rep psw and
               vflag  = get_vf rep psw and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`BAND_u1`,EXPAND_LET_RULE BAND_u1);;


%---------------------------------------------------------------
 table entry 21: first uinst for BOR
---------------------------------------------------------------%
let BOR_u1 = new_definition
   (`BOR_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      BOR_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               b = EL (reg_len rep (srcb rep ir)) reg and
               d = reg_len rep (dest rep ir) in
           let result = (bor rep (a, b)) in
           let cflag  = get_cf rep psw and
               vflag  = get_vf rep psw and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`BOR_u1`,EXPAND_LET_RULE BOR_u1);;

%---------------------------------------------------------------
 table entry 22: first uinst for BXOR
```

```
----------------------------------------------------------------%
let BXOR_u1 = new_definition
   (`BXOR_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      BXOR_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               b = EL (reg_len rep (srcb rep ir)) reg and
               d = reg_len rep (dest rep ir) in
           let result = (bxor rep (a, b)) in
           let cflag  = get_cf rep psw and
               vflag  = get_vf rep psw and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`BXOR_u1`,EXPAND_LET_RULE BXOR_u1);;

%----------------------------------------------------------------
 table entry 23: first uinst for BNOT
----------------------------------------------------------------%
let BNOT_u1 = new_definition
   (`BNOT_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      BNOT_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               d = reg_len rep (dest rep ir) in
           let result = (bnot rep a) in
           let cflag  = get_cf rep psw and
               vflag  = get_vf rep psw and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`BNOT_u1`,EXPAND_LET_RULE BNOT_u1);;

%----------------------------------------------------------------
 table entry 24: first uinst for ADDI
----------------------------------------------------------------%
let ADDI_u1 = new_definition
```

```
   (`ADDI_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      ADDI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir and
               d = reg_len rep (dest rep ir) in
           let result = (add rep (a, i)) in
           let cflag  = addp rep (a, i, result) and
               vflag  = aovfl rep (a, i, result) and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`ADDI_u1`,EXPAND_LET_RULE ADDI_u1);;

%-----------------------------------------------------------------
 table entry 25: first uinst for ADDCI
-----------------------------------------------------------------%
let ADDCI_u1 = new_definition
   (`ADDCI_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      ADDCI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                   (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir and
               d = reg_len rep (dest rep ir) in
           let result = (addc rep (a, i, get_cf rep psw)) in
           let cflag  = addcp rep (a, i, result) and
               vflag  = aovfl rep (a, i, result) and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`ADDCI_u1`,EXPAND_LET_RULE ADDCI_u1);;

%-----------------------------------------------------------------
 table entry 26: first uinst for SUBI
-----------------------------------------------------------------%
let SUBI_u1 = new_definition
   (`SUBI_u1_def`,
```

```
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      SUBI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir and
               d = reg_len rep (dest rep ir) in
           let result = (sub rep (a, i)) in
           let cflag  = subp rep (a, i, result) and
               vflag  = sovfl rep (a, i, result) and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`SUBI_u1`,EXPAND_LET_RULE SUBI_u1);;

%----------------------------------------------------------------
 table entry 27: first uinst for SUBCI
----------------------------------------------------------------%
let SUBCI_u1 = new_definition
   (`SUBCI_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      SUBCI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                   (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir and
               d = reg_len rep (dest rep ir) in
           let result = (subc rep (a, i, get_cf rep psw)) in
           let cflag  = subp rep (a, i, result) and
               vflag  = sovfl rep (a, i, result) and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`SUBCI_u1`,EXPAND_LET_RULE SUBCI_u1);;

%----------------------------------------------------------------
 table entry 28: first uinst for BANDI
----------------------------------------------------------------%
let BANDI_u1 = new_definition
   (`BANDI_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
```

```
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      BANDI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                   (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir and
               d = reg_len rep (dest rep ir) in
           let result = (band rep (a, i)) in
           let cflag  = get_cf rep psw and
               vflag  = get_vf rep psw and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`BANDI_u1`,EXPAND_LET_RULE BANDI_u1);;

%----------------------------------------------------------------
 table entry 29: first uinst for BORI
----------------------------------------------------------------%
let BORI_u1 = new_definition
   (`BORI_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      BORI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir and
               d = reg_len rep (dest rep ir) in
           let result = (bor rep (a, i)) in
           let cflag  = get_cf rep psw and
               vflag  = get_vf rep psw and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`BORI_u1`,EXPAND_LET_RULE BORI_u1);;

%----------------------------------------------------------------
 table entry 30: first uinst for BXORI
----------------------------------------------------------------%
let BXORI_u1 = new_definition
   (`BXORI_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
```

```
      (int_e:bool).
      BXORI_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                   (int_e) =
           let a = EL (reg_len rep (srca rep ir)) reg and
               i = imm rep ir and
               d = reg_len rep (dest rep ir) in
           let result = (bxor rep (a, i)) in
           let cflag  = get_cf rep psw and
               vflag  = get_vf rep psw and
               nflag  = negp rep result and
               zflag  = zerop rep result and
               sm     = get_sm rep psw and
               ie     = get_ie rep psw in
           (UPDATE_REG rep psw d reg result,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`BXORI_u1`,EXPAND_LET_RULE BXORI_u1);;

%----------------------------------------------------------------
 table entry 31: first uinst for NOOP (already defined)
----------------------------------------------------------------%


%----------------------------------------------------------------
 code for external interrupt
----------------------------------------------------------------%
let EINT_u1 = new_definition
   (`EINT_u1_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      EINT_u1 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                 (int_e) =
           let cflag  = get_cf rep psw and
               vflag  = get_vf rep psw and
               nflag  = get_nf rep psw and
               zflag  = get_zf rep psw and
               sm     = T and
               ie     = F in
           (reg,
            mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
            pc, mem, ivec, ir, mar, pc, ^EINT_u2_ADDR)"
   );;

save_thm(`EINT_u1`,EXPAND_LET_RULE EINT_u1);;

let EINT_u2 = new_definition
   (`EINT_u2_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      EINT_u2 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
```

```
           (reg, psw, pc,
            mem, ivec, ir, SSP_REG reg, mbr, ^EINT_u3_ADDR)"
   );;

save_thm(`EINT_u2`,EXPAND_LET_RULE EINT_u2);;

let EINT_u3 = new_definition
   (`EINT_u3_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      EINT_u3 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let result = inc rep (SSP_REG reg) in
           (UPDATE_REG rep psw ssp_reg reg result,
            psw, pc,
            store rep (mem, address rep mar, mbr),
            ivec, ir, mar, mbr, ^EINT_u4_ADDR)"
   );;

save_thm(`EINT_u3`,EXPAND_LET_RULE EINT_u3);;

let EINT_u4 = new_definition
   (`EINT_u4_def`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      EINT_u4 rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
                  (int_e) =
           let result = band rep (wordn rep 255, int_fetch rep ivec) in
           (reg, psw, result, mem,
            ivec, ir, mar, mbr, ^FETCH_ADDR)"
   );;

save_thm(`EINT_u4`,EXPAND_LET_RULE EINT_u4);;

let micro_state = ":((*wordn)list#*wordn#*wordn#*memory#
                     *wordn#*wordn#*wordn#*wordn#bt6)";;

let micro_env = ":bool";;

%----------------------------------------------------------------
 The miro_inst_list will be used to instantiate inst_list in
 mk_micro.ml.
----------------------------------------------------------------%
let micro_inst_list = new_definition
   (`micro_inst_list`,
    "! rep:^rep_ty .
     micro_inst_list rep =
      [((F,F,F,F,F,F),(FETCH rep));
       ((F,F,F,F,F,T),(ISSUE rep));
       ((F,F,F,F,T,F),(DECODE rep));
       ((F,F,F,F,T,T),(NOOP_u1 rep));
       ((F,F,F,T,F,F),(JMP_u1 rep));
       ((F,F,F,T,F,T),(CALL_u1 rep));
```

```
((F,F,F,T,T,F),(INT_u1 rep));
((F,F,F,T,T,T),(RTI_u1 rep));
((F,F,T,F,F,F),(GPSW_u1 rep));
((F,F,T,F,F,T),(PPSW_u1 rep));
((F,F,T,F,T,F),(LD_u1 rep));
((F,F,T,F,T,T),(ST_u1 rep));
((F,F,T,T,F,F),(LSL_u1 rep));
((F,F,T,T,F,T),(LSR_u1 rep));
((F,F,T,T,T,F),(ASR_u1 rep));
((F,F,T,T,T,T),(RTN_u1 rep));
((F,T,F,F,F,F),(NOOP_u1 rep));
((F,T,F,F,F,T),(NOOP_u1 rep));
((F,T,F,F,T,F),(LDI_u1 rep));
((F,T,F,F,T,T),(STI_u1 rep));
((F,T,F,T,F,F),(ADD_u1 rep));
((F,T,F,T,F,T),(ADDC_u1 rep));
((F,T,F,T,T,F),(SUB_u1 rep));
((F,T,F,T,T,T),(SUBC_u1 rep));
((F,T,T,F,F,F),(BAND_u1 rep));
((F,T,T,F,F,T),(BOR_u1 rep));
((F,T,T,F,T,F),(BXOR_u1 rep));
((F,T,T,F,T,T),(BNOT_u1 rep));
((F,T,T,T,F,F),(ADDI_u1 rep));
((F,T,T,T,F,T),(ADDCI_u1 rep));
((F,T,T,T,T,F),(SUBI_u1 rep));
((F,T,T,T,T,T),(SUBCI_u1 rep));
((T,F,F,F,F,F),(BANDI_u1 rep));
((T,F,F,F,F,T),(BORI_u1 rep));
((T,F,F,F,T,F),(BXORI_u1 rep));
((T,F,F,F,T,T),(NOOP_u1 rep));
((T,F,F,T,F,F),(CALL_u2 rep));
((T,F,F,T,F,T),(CALL_u3 rep));
((T,F,F,T,T,F),(CALL_u4 rep));
((T,F,F,T,T,T),(INT_u2 rep));
((T,F,T,F,F,F),(INT_u3 rep));
((T,F,T,F,F,T),(INT_u4 rep));
((T,F,T,F,T,F),(RTI_u2 rep));
((T,F,T,F,T,T),(RTI_u3 rep));
((T,F,T,T,F,F),(RTN_u2 rep));
((T,F,T,T,F,T),(LD_u2 rep));
((T,F,T,T,T,F),(ST_u2 rep));
((T,F,T,T,T,T),(ST_u3 rep));
((T,T,F,F,F,F),(STI_u2 rep));
((T,T,F,F,F,T),(EINT_u1 rep));
((T,T,F,F,T,F),(EINT_u2 rep));
((T,T,F,F,T,T),(EINT_u3 rep));
((T,T,F,T,F,F),(EINT_u4 rep));
((T,T,F,T,F,T),(LD_u3 rep));
((T,T,F,T,T,F),(NOOP_u1 rep));
((T,T,F,T,T,T),(NOOP_u1 rep));
((T,T,T,F,F,F),(NOOP_u1 rep));
((T,T,T,F,F,T),(NOOP_u1 rep));
((T,T,T,F,T,F),(NOOP_u1 rep));
((T,T,T,F,T,T),(NOOP_u1 rep));
((T,T,T,T,F,F),(NOOP_u1 rep));
```

```
      ((T,T,T,T,F,T),(NOOP_u1 rep));
      ((T,T,T,T,T,F),(NOOP_u1 rep));
      ((T,T,T,T,T,T),(NOOP_u1 rep))]"
   );;

%----------------------------------------------------------------
 Select MPC from state.  This is used to instantiate gen_I.th.
----------------------------------------------------------------%
let GetMPC = new_definition
   (`GetMPC`,
    "!(reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6)
      (int_e:bool).
      GetMPC (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc)
             (int_e) = mpc"
   );;


close_theory();;
```

3.6.2 The Micro–Level Instructions

The section presents the ML code that creates the theory `uinst_def.th`.

```
%----------------------------------------------------------------

   File:         def_uinst.ml

   Author:       (c) P. J. Windley 1990

   Date:         JUN 23, 1990

   Modified:

   Description:

   Defines the microinstructions and microrom for the
   micro--level.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root) [`decimal/`;`assoc/`]));;

system `/bin/rm uinst.th`;;

new_theory `uinst`;;

loadf `ucode_aux`;;

new_parent `ucode_def`;;

%----------------------------------------------------------------
 If you change these addresses, change the list in def_uinst.ml
 as well.
----------------------------------------------------------------%
let FETCH_ADDR   = "(F,F,F,F,F,F)";;

let CALL_u2_ADDR = "(T,F,F,T,F,F)";;

let CALL_u3_ADDR = "(T,F,F,T,F,T)";;

let CALL_u4_ADDR = "(T,F,F,T,T,F)";;

let INT_u2_ADDR  = "(T,F,F,T,T,T)";;

let INT_u3_ADDR  = "(T,F,T,F,F,F)";;
```

```
let INT_u4_ADDR  = "(T,F,T,F,F,T)";;

let RTI_u2_ADDR  = "(T,F,T,F,T,F)";;

let RTI_u3_ADDR  = "(T,F,T,F,T,T)";;

let RTN_u2_ADDR  = "(T,F,T,T,F,F)";;

let LD_u2_ADDR   = "(T,F,T,T,F,T)";;

let ST_u2_ADDR   = "(T,F,T,T,T,F)";;

let ST_u3_ADDR   = "(T,F,T,T,T,T)";;

let STI_u2_ADDR  = "(T,T,F,F,F,F)";;

let EINT_u1_ADDR = "(T,T,F,F,F,T)";;

let EINT_u2_ADDR = "(T,T,F,F,T,F)";;

let EINT_u3_ADDR = "(T,T,F,F,T,T)";;

let EINT_u4_ADDR = "(T,T,F,T,F,F)";;

let LD_u3_ADDR = "(T,T,F,T,F,T)";;


let OFFSET = "(F,F,F,T,F,F)";;

let DUMMY = "(T,T,T,T,T,T)";;

let FETCH_mc = new_definition
   (`FETCH_mc`,
    "FETCH_mc =
     (^(Oper(noreg,nsh,noreg,nop,noreg,mar_gets_pc)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,rd)),
      ^(Mpc(jint,EINT_u1_ADDR)))"
   );;

%-----------------------------------------------------------------
FETCH_mc =
|- FETCH_mc =
   (F,(T,T),(T,F,F,T),F,T,T,(T,T,F),(T,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,T,F),(F,T,T),T,T,F,F,F,T
-----------------------------------------------------------------%

let ISSUE_mc = new_definition
   (`ISSUE_mc`,
    "ISSUE_mc =
     (^(Oper(ir,nsh,mbr,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(step,DUMMY)))"
```

```
   );;

%---------------------------------------------------------------
ISSUE_mc =
|- ISSUE_mc =
   (T,(T,T),(T,F,F,T),F,F,F,(F,T,T),(T,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,F),T,T,T,T,T,T
---------------------------------------------------------------%

let DECODE_mc = new_definition
   (`DECODE_mc`,
    "DECODE_mc =
     (^(Oper(pc,nsh,pc,inc,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jop,DUMMY)))"
   );;

%---------------------------------------------------------------
DECODE_mc =
|- DECODE_mc =
   (F,(T,T),(F,F,T,F),F,F,F,(T,F,F),(T,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,T,F),T,T,T,T,T,T
---------------------------------------------------------------%

let NOOP_u1_mc = new_definition
   (`NOOP_u1_mc`,
    "NOOP_u1_mc =
     (^(Oper(noreg,nsh,noreg,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%---------------------------------------------------------------
NOOP_u1_mc =
|- NOOP_u1_mc =
   (F,(T,T),(T,F,F,T),F,F,F,(T,T,F),(T,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,F),T,T,T,T,T,T
---------------------------------------------------------------%

let JMP_u1_mc = new_definition
   (`JMP_u1_mc`,
    "JMP_u1_mc =
     (^(Oper(pcj,nsh,reg_file,add,ir,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%---------------------------------------------------------------
JMP_u1_mc =
|- JMP_u1_mc =
   (F,(T,T),(F,F,F,F),F,F,F,(T,F,T),(F,F,F),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
```

```
----------------------------------------------------------------%

let CALL_u1_mc = new_definition
   (`CALL_u1_mc`,
    "CALL_u1_mc =
     (^(Oper(noreg,nsh,pc,nop,noreg,mbr)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,CALL_u2_ADDR)))"
   );;

%----------------------------------------------------------------
CALL_u1_mc =
|- CALL_u1_mc =
   (F,(T,T),(T,F,F,T),T,F,F,(T,T,F),(T,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,F,T,F,F
----------------------------------------------------------------%

let CALL_u2_mc = new_definition
   (`CALL_u2_mc`,
    "CALL_u2_mc =
     (^(Oper(noreg,nsh,reg_dest,nop,noreg,mar)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,CALL_u3_ADDR)))"
   );;

%----------------------------------------------------------------
CALL_u2_mc =
|- CALL_u2_mc =
   (F,(T,T),(T,F,F,T),F,T,F,(T,T,F),(F,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,F,T,F,T
----------------------------------------------------------------%

let CALL_u3_mc = new_definition
   (`CALL_u3_mc`,
    "CALL_u3_mc =
     (^(Oper(pc,nsh,reg_file,add,ir,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,wr)),
      ^(Mpc(jmp,CALL_u4_ADDR)))"
   );;

%----------------------------------------------------------------
CALL_u3_mc =
|- CALL_u3_mc =
   (F,(T,T),(F,F,F,F),F,F,F,(T,F,F),(F,F,F),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,T),(F,F,T),T,F,F,T,T,F
----------------------------------------------------------------%

let CALL_u4_mc = new_definition
   (`CALL_u4_mc`,
    "CALL_u4_mc =
     (^(Oper(reg_file,nsh,reg_dest,inc,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
```

```
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR))"
   );;

%----------------------------------------------------------------
CALL_u4_mc =
|- CALL_u4_mc =
   (F,(T,T),(F,F,T,F),F,F,F,(F,F,F),(F,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
----------------------------------------------------------------%

let INT_u1_mc = new_definition
   (`INT_u1_mc`,
    "INT_u1_mc =
     (^(Oper(noreg,nsh,pc,nop,noreg,mbr)),
      ^(Set_PSW (set_sm, clr_ie, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,INT_u2_ADDR))"
   );;

%----------------------------------------------------------------
INT_u1_mc =
|- INT_u1_mc =
   (F,(T,T),(T,F,F,T),T,F,F,(T,T,F),(T,F,T),T,F),(T,F,F,T,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,F,T,T,T
----------------------------------------------------------------%

let INT_u2_mc = new_definition
   (`INT_u2_mc`,
    "INT_u2_mc =
     (^(Oper(noreg,nsh,ssp,nop,noreg,mar)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,INT_u3_ADDR))"
   );;

%----------------------------------------------------------------
INT_u2_mc =
|- INT_u2_mc =
   (F,(T,T),(T,F,F,T),F,T,F,(T,T,F),(F,T,F),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,T,F,F,F
----------------------------------------------------------------%

let INT_u3_mc = new_definition
   (`INT_u3_mc`,
    "INT_u3_mc =
     (^(Oper(ssp,nsh,ssp,inc,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,INT_u4_ADDR))"
   );;

%----------------------------------------------------------------
INT_u3_mc =
|- INT_u3_mc =
```

```
   (F,(T,T),(F,F,T,F),F,F,F,(F,F,T),(F,T,F),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,T,F,F,T
-----------------------------------------------------------------%

let INT_u4_mc = new_definition
   (`INT_u4_mc`,
    "INT_u4_mc =
     (^(Oper(pc,nsh,C255,band,ir,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,wr)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
INT_u4_mc =
|- INT_u4_mc =
   (F,(T,T),(F,T,T,F),F,F,F,(T,F,F),(T,F,F),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,T),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let RTI_u1_mc = new_definition
   (`RTI_u1_mc`,
    "RTI_u1_mc =
     (^(Oper(ssp,nsh,ssp,dec,noreg,mar)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,RTI_u2_ADDR)))"
   );;

%-----------------------------------------------------------------
RTI_u1_mc =
|- RTI_u1_mc =
   (F,(T,T),(F,T,F,T),F,T,F,(F,F,T),(F,T,F),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,T,F,T,F
-----------------------------------------------------------------%

let RTI_u2_mc = new_definition
   (`RTI_u2_mc`,
    "RTI_u2_mc =
     (^(Oper(noreg,nsh,noreg,nop,noreg,noreg)),
      ^(Set_PSW (clr_sm, set_ie, pass, pass, pass, pass)),
      ^(ExtSig(off,off,rd)),
      ^(Mpc(jmp,RTI_u3_ADDR)))"
   );;

%-----------------------------------------------------------------
RTI_u2_mc =
|- RTI_u2_mc =
   (F,(T,T),(T,F,F,T),F,F,F,(T,T,F),(T,F,T),T,F),(F,T,T,F,F,F,F,F,F),
   (F,F,T,F),(F,F,T),T,F,T,F,T,T
-----------------------------------------------------------------%

let RTI_u3_mc = new_definition
   (`RTI_u3_mc`,
    "RTI_u3_mc =
```

```
     (^(Oper(pc,nsh,mbr,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
RTI_u3_mc =
|- RTI_u3_mc =
   (T,(T,T),(T,F,F,T),F,F,F,(T,F,F),(T,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,T,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let GPSW_u1_mc = new_definition
   (`GPSW_u1_mc`,
    "GPSW_u1_mc =
     (^(Oper(reg_file,nsh,psw,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
GPSW_u1_mc =
|- GPSW_u1_mc =
   (F,(T,T),(T,F,F,T),F,F,F,(F,F,F),(F,T,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let PPSW_u1_mc = new_definition
   (`PPSW_u1_mc`,
    "PPSW_u1_mc =
     (^(Oper(psw,nsh,reg_dest,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
PPSW_u1_mc =
|- PPSW_u1_mc =
   (F,(T,T),(T,F,F,T),F,F,F,(F,T,F),(F,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let LD_u1_mc = new_definition
   (`LD_u1_mc`,
    "LD_u1_mc =
     (^(Oper(noreg,nsh,reg_file,add,reg_file,mar)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,LD_u2_ADDR)))"
   );;
```

```
let LD_u2_mc = new_definition
   (`LD_u2_mc`,
    "LD_u2_mc =
     (^(Oper(noreg,nsh,noreg,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,rd)),
      ^(Mpc(jmp,LD_u3_ADDR)))"
   );;

let LD_u3_mc = new_definition
   (`LD_u3_mc`,
    "LD_u3_mc =
     (^(Oper(reg_file,nsh,mbr,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;


let ST_u1_mc = new_definition
   (`ST_u1_mc`,
    "ST_u1_mc =
     (^(Oper(noreg,nsh,reg_file,add,reg_file,mar)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,ST_u2_ADDR)))"
   );;

%-----------------------------------------------------------------
ST_u1_mc =
|- ST_u1_mc =
   (F,(T,T),(F,F,F,F),F,T,F,(T,T,F),(F,F,F),F,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,T,T,T,F
-----------------------------------------------------------------%

let ST_u2_mc = new_definition
   (`ST_u2_mc`,
    "ST_u2_mc =
     (^(Oper(noreg,nsh,reg_dest,nop,reg_file,mbr)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,ST_u3_ADDR)))"
   );;

%-----------------------------------------------------------------
ST_u2_mc =
|- ST_u2_mc =
   (F,(T,T),(T,F,F,T),T,F,F,(T,T,F),(F,F,T),F,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,T,T,T,T
-----------------------------------------------------------------%

let ST_u3_mc = new_definition
   (`ST_u3_mc`,
    "ST_u3_mc =
     (^(Oper(noreg,nsh,noreg,nop,noreg,noreg)),
```

```
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,wr)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%----------------------------------------------------------------
ST_u3_mc =
|- ST_u3_mc =
   (F,(T,T),(T,F,F,T),F,F,F,(T,T,F),(T,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,T),(F,F,T),F,F,F,F,F,F
----------------------------------------------------------------%

let LSL_u1_mc = new_definition
   (`LSL_u1_mc`,
    "LSL_u1_mc =
     (^(Oper(reg_file,shl,reg_file,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, ld_from_shifter, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%----------------------------------------------------------------
LSL_u1_mc =
|- LSL_u1_mc =
   (F,(F,F),(T,F,F,T),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,T,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
----------------------------------------------------------------%

let LSR_u1_mc = new_definition
   (`LSR_u1_mc`,
    "LSR_u1_mc =
     (^(Oper(reg_file,shr,reg_file,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, ld_from_shifter, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%----------------------------------------------------------------
LSR_u1_mc =
|- LSR_u1_mc =
   (F,(F,T),(T,F,F,T),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,T,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
----------------------------------------------------------------%

let ASR_u1_mc = new_definition
   (`ASR_u1_mc`,
    "ASR_u1_mc =
     (^(Oper(reg_file,asr,reg_file,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, ld_from_shifter, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%----------------------------------------------------------------
ASR_u1_mc =
```

```
|- ASR_u1_mc =
   (F,(T,F),(T,F,F,T),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,T,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let RTN_u1_mc = new_definition
   (`RTN_u1_mc`,
    "RTN_u1_mc =
     (^(Oper(reg_file,nsh,reg_dest,dec,noreg,mar)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,RTN_u2_ADDR)))"
   );;

%----------------------------------------------------------------
RTN_u1_mc =
|- RTN_u1_mc =
   (F,(T,T),(F,T,F,T),F,T,F,(F,F,F),(F,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,T,T,F,F
-----------------------------------------------------------------%

let RTN_u2_mc = new_definition
   (`RTN_u2_mc`,
    "RTN_u2_mc =
     (^(Oper(noreg,nsh,noreg,nop,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,rd)),
      ^(Mpc(jmp,RTI_u3_ADDR)))"
   );;

%----------------------------------------------------------------
RTN_u2_mc =
|- RTN_u2_mc =
   (F,(T,T),(T,F,F,T),F,F,F,(T,T,F),(T,F,T),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,T,F),(F,F,T),T,F,T,F,T,T
-----------------------------------------------------------------%

let LDI_u1_mc = new_definition
   (`LDI_u1_mc`,
    "LDI_u1_mc =
     (^(Oper(noreg,nsh,reg_file,add,ir,mar)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,LD_u2_ADDR)))"
   );;


%----------------------------------------------------------------
LDI_u1_mc =
|- LDI_u1_mc =
   (F,(T,T),(F,F,F,F),F,T,F,(T,T,F),(F,F,F),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,T,F),(F,F,T),T,F,T,T,F,T
-----------------------------------------------------------------%

let STI_u1_mc = new_definition
```

```
   (`STI_u1_mc`,
    "STI_u1_mc =
     (^(Oper(noreg,nsh,reg_file,add,ir,mar)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,STI_u2_ADDR)))"
   );;

%-----------------------------------------------------------------
STI_u1_mc =
|- STI_u1_mc =
   (F,(T,T),(F,F,F,F),F,T,F,(T,T,F),(F,F,F),F,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,T,T,T,F
-----------------------------------------------------------------%

let STI_u2_mc = new_definition
   (`STI_u2_mc`,
    "STI_u2_mc =
     (^(Oper(noreg,nsh,reg_dest,nop,reg_file,mbr)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,ST_u3_ADDR)))"
   );;

%-----------------------------------------------------------------
STI_u2_mc =
|- STI_u2_mc =
   (F,(T,T),(T,F,F,T),T,F,F,(T,T,F),(F,F,T),F,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,F,T,T,T,T
-----------------------------------------------------------------%

let ADD_u1_mc = new_definition
   (`ADD_u1_mc`,
    "ADD_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,add,reg_file,noreg)),
      ^(Set_PSW (pass, pass, ld_vf, ld_nf, ld_from_alu, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
ADD_u1_mc =
|- ADD_u1_mc =
   (F,(T,T),(F,F,F,F),F,F,F,(F,F,F),(F,F,F),F,F),(F,F,F,F,T,T,T,T,T),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let ADDC_u1_mc = new_definition
   (`ADDC_u1_mc`,
    "ADDC_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,addc,reg_file,noreg)),
      ^(Set_PSW (pass, pass, ld_vf, ld_nf, ld_from_alu, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;
```

```
%----------------------------------------------------------------
ADDC_u1_mc =
|- ADDC_u1_mc =
   (F,(T,T),(F,F,F,T),F,F,F,(F,F,F),(F,F,F),F,F),(F,F,F,F,T,T,T,T,T),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
----------------------------------------------------------------%

let SUB_u1_mc = new_definition
   (`SUB_u1_mc`,
    "SUB_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,sub,reg_file,noreg)),
      ^(Set_PSW (pass, pass, ld_vf, ld_nf, ld_from_alu, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%----------------------------------------------------------------
SUB_u1_mc =
|- SUB_u1_mc =
   (F,(T,T),(F,F,T,T),F,F,F,(F,F,F),(F,F,F),F,F),(F,F,F,F,T,T,T,T,T),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
----------------------------------------------------------------%

let SUBC_u1_mc = new_definition
   (`SUBC_u1_mc`,
    "SUBC_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,subc,reg_file,noreg)),
      ^(Set_PSW (pass, pass, ld_vf, ld_nf, ld_from_alu, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%----------------------------------------------------------------
SUBC_u1_mc =
|- SUBC_u1_mc =
   (F,(T,T),(F,T,F,F),F,F,F,(F,F,F),(F,F,F),F,F),(F,F,F,F,T,T,T,T,T),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
----------------------------------------------------------------%

let BAND_u1_mc = new_definition
   (`BAND_u1_mc`,
    "BAND_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,band,reg_file,noreg)),
      ^(Set_PSW (pass, pass, pass, ld_nf, pass, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%----------------------------------------------------------------
BAND_u1_mc =
|- BAND_u1_mc =
   (F,(T,T),(F,T,T,F),F,F,F,(F,F,F),(F,F,F),F,F),(F,F,F,F,F,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
----------------------------------------------------------------%
```

```
let BOR_u1_mc = new_definition
   (`BOR_u1_mc`,
    "BOR_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,bor,reg_file,noreg)),
      ^(Set_PSW (pass, pass, pass, ld_nf, pass, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
BOR_u1_mc =
|- BOR_u1_mc =
   (F,(T,T),(T,F,F,F),F,F,F,(F,F,F),(F,F,F),F,F),(F,F,F,F,F,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let BXOR_u1_mc = new_definition
   (`BXOR_u1_mc`,
    "BXOR_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,bxor,reg_file,noreg)),
      ^(Set_PSW (pass, pass, pass, ld_nf, pass, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
BXOR_u1_mc =
|- BXOR_u1_mc =
   (F,(T,T),(F,T,T,T),F,F,F,(F,F,F),(F,F,F),F,F),(F,F,F,F,F,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let BNOT_u1_mc = new_definition
   (`BNOT_u1_mc`,
    "BNOT_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,bnot,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, ld_nf, pass, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
BNOT_u1_mc =
|- BNOT_u1_mc =
   (F,(T,T),(T,F,F,T),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,F,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let ADDI_u1_mc = new_definition
   (`ADDI_u1_mc`,
    "ADDI_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,add,ir,noreg)),
      ^(Set_PSW (pass, pass, ld_vf, ld_nf, ld_from_alu, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
```

```
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
ADDI_u1_mc =
|- ADDI_u1_mc =
   (F,(T,T),(F,F,F,F),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,T,T,T,T,T),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let ADDCI_u1_mc = new_definition
   (`ADDCI_u1_mc`,
    "ADDCI_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,addc,ir,noreg)),
      ^(Set_PSW (pass, pass, ld_vf, ld_nf, ld_from_alu, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
ADDCI_u1_mc =
|- ADDCI_u1_mc =
   (F,(T,T),(F,F,F,T),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,T,T,T,T,T),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let SUBI_u1_mc = new_definition
   (`SUBI_u1_mc`,
    "SUBI_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,sub,ir,noreg)),
      ^(Set_PSW (pass, pass, ld_vf, ld_nf, ld_from_alu, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
SUBI_u1_mc =
|- SUBI_u1_mc =
   (F,(T,T),(F,F,T,T),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,T,T,T,T,T),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let SUBCI_u1_mc = new_definition
   (`SUBCI_u1_mc`,
    "SUBCI_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,subc,ir,noreg)),
      ^(Set_PSW (pass, pass, ld_vf, ld_nf, ld_from_alu, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
SUBCI_u1_mc =
|- SUBCI_u1_mc =
   (F,(T,T),(F,T,F,F),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,T,T,T,T,T),
```

```
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let BANDI_u1_mc = new_definition
   (`BANDI_u1_mc`,
    "BANDI_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,band,ir,noreg)),
      ^(Set_PSW (pass, pass, pass, ld_nf, pass, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
BANDI_u1_mc =
|- BANDI_u1_mc =
   (F,(T,T),(F,T,T,F),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,F,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let BORI_u1_mc = new_definition
   (`BORI_u1_mc`,
    "BORI_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,bor,ir,noreg)),
      ^(Set_PSW (pass, pass, pass, ld_nf, pass, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
BORI_u1_mc =
|- BORI_u1_mc =
   (F,(T,T),(T,F,F,F),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,F,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let BXORI_u1_mc = new_definition
   (`BXORI_u1_mc`,
    "BXORI_u1_mc =
     (^(Oper(reg_file,nsh,reg_file,bxor,ir,noreg)),
      ^(Set_PSW (pass, pass, pass, ld_nf, pass, ld_zf)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
BXORI_u1_mc =
|- BXORI_u1_mc =
   (F,(T,T),(F,T,T,T),F,F,F,(F,F,F),(F,F,F),T,F),(F,F,F,F,F,T,F,T,F),
   (F,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%

let EINT_u1_mc = new_definition
   (`EINT_u1_mc`,
    "EINT_u1_mc =
     (^(Oper(noreg,nsh,pc,nop,noreg,mbr)),
```

```
      ^(Set_PSW (set_sm, clr_ie, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,EINT_u2_ADDR)))"
   );;

%-----------------------------------------------------------------
EINT_u1_mc =
|- EINT_u1_mc =
   (F,(T,T),(T,F,F,T),T,F,F,(T,T,F),(T,F,T),T,F),(T,F,F,T,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,T,F,F,T,F
-----------------------------------------------------------------%

let EINT_u2_mc = new_definition
   (`EINT_u2_mc`,
    "EINT_u2_mc =
     (^(Oper(noreg,nsh,ssp,nop,noreg,mar)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,no_mem_op)),
      ^(Mpc(jmp,EINT_u3_ADDR)))"
   );;

%-----------------------------------------------------------------
EINT_u2_mc =
|- EINT_u2_mc =
   (F,(T,T),(T,F,F,T),F,T,F,(T,T,F),(F,T,F),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,F),(F,F,T),T,T,F,F,T,T
-----------------------------------------------------------------%

let EINT_u3_mc = new_definition
   (`EINT_u3_mc`,
    "EINT_u3_mc =
     (^(Oper(ssp,nsh,ssp,inc,noreg,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(off,off,wr)),
      ^(Mpc(jmp,EINT_u4_ADDR)))"
   );;

%-----------------------------------------------------------------
EINT_u3_mc =
|- EINT_u3_mc =
   (F,(T,T),(F,F,T,F),F,F,F,(F,F,T),(F,T,F),T,F),(F,F,F,F,F,F,F,F,F),
   (F,F,F,T),(F,F,T),T,T,F,T,F,F
-----------------------------------------------------------------%

let EINT_u4_mc = new_definition
   (`EINT_u4_mc`,
    "EINT_u4_mc =
     (^(Oper(pc,nsh,C255,band,ivec,noreg)),
      ^(Set_PSW (pass, pass, pass, pass, pass, pass)),
      ^(ExtSig(i_ack,off,no_mem_op)),
      ^(Mpc(jmp,FETCH_ADDR)))"
   );;

%-----------------------------------------------------------------
EINT_u4_mc =
```

```
|- EINT_u4_mc =
   (F,(T,T),(F,T,T,F),F,F,F,(T,F,F),(T,F,F),F,T),(F,F,F,F,F,F,F,F,F),
   (T,F,F,F),(F,F,T),F,F,F,F,F,F
-----------------------------------------------------------------%


%-----------------------------------------------------------------
This list must contain the microinstructions that implement the
behavior in the definition micro_inst_list defined in def_micro.ml.
-----------------------------------------------------------------%

let micro_rom = new_definition
   (`micro_rom`,
    "!n . micro_rom n =
      EL n
      [FETCH_mc; ISSUE_mc; DECODE_mc; NOOP_u1_mc; JMP_u1_mc; CALL_u1_mc;
       INT_u1_mc; RTI_u1_mc; GPSW_u1_mc; PPSW_u1_mc; LD_u1_mc; ST_u1_mc;
       LSL_u1_mc; LSR_u1_mc; ASR_u1_mc; RTN_u1_mc; NOOP_u1_mc; NOOP_u1_mc;
       LDI_u1_mc; STI_u1_mc; ADD_u1_mc; ADDC_u1_mc; SUB_u1_mc; SUBC_u1_mc;
       BAND_u1_mc; BOR_u1_mc; BXOR_u1_mc; BNOT_u1_mc; ADDI_u1_mc;
       ADDCI_u1_mc; SUBI_u1_mc; SUBCI_u1_mc; BANDI_u1_mc; BORI_u1_mc;
       BXORI_u1_mc; NOOP_u1_mc; CALL_u2_mc; CALL_u3_mc; CALL_u4_mc;
       INT_u2_mc; INT_u3_mc; INT_u4_mc; RTI_u2_mc; RTI_u3_mc; RTN_u2_mc;
       LD_u2_mc; ST_u2_mc; ST_u3_mc; STI_u2_mc; EINT_u1_mc; EINT_u2_mc;
       EINT_u3_mc; EINT_u4_mc; LD_u3_mc; NOOP_u1_mc; NOOP_u1_mc;
       NOOP_u1_mc; NOOP_u1_mc; NOOP_u1_mc; NOOP_u1_mc; NOOP_u1_mc;
       NOOP_u1_mc; NOOP_u1_mc; NOOP_u1_mc]"
   );;


save_thm(`micro_rom_expanded`,
          SUBS [FETCH_mc;ISSUE_mc;DECODE_mc;NOOP_u1_mc;JMP_u1_mc;
                CALL_u1_mc;INT_u1_mc;RTI_u1_mc;GPSW_u1_mc;
                PPSW_u1_mc;LD_u1_mc;ST_u1_mc;LSL_u1_mc;LSR_u1_mc;
                ASR_u1_mc;RTN_u1_mc;NOOP_u1_mc;NOOP_u1_mc;
                LDI_u1_mc;STI_u1_mc;ADD_u1_mc;ADDC_u1_mc;SUB_u1_mc;
                SUBC_u1_mc;BAND_u1_mc;BOR_u1_mc;BXOR_u1_mc;
                BNOT_u1_mc;ADDI_u1_mc;ADDCI_u1_mc;SUBI_u1_mc;
                SUBCI_u1_mc;BANDI_u1_mc;BORI_u1_mc;BXORI_u1_mc;
                NOOP_u1_mc;CALL_u2_mc;CALL_u3_mc;CALL_u4_mc;
                INT_u2_mc;INT_u3_mc;INT_u4_mc;RTI_u2_mc;
                RTI_u3_mc;RTN_u2_mc;LD_u2_mc;ST_u2_mc;ST_u3_mc;
                STI_u2_mc;EINT_u1_mc;EINT_u2_mc;EINT_u3_mc;
                EINT_u4_mc;LD_u3_mc] micro_rom
    );;


close_theory();;
```

3.6.3 The Micro–Level Proof

The section presents the ML code that creates the theory `micro.th`.

```
%-----------------------------------------------------------------

   File:         mk_micro.ml

   Author:       (c) P. J. Windley 1990

   Date:         JUN 23, 1990

   Modified:

   Description:

   Proves the micro--level correct with respect to the phase--level
   using the generic interpreter proof, phase.th and micro_def.th.

-----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`]));;

loadf `abstract`;;

system `/bin/rm micro.th`;;

new_theory `micro`;;

loadf `tuple`;;

map new_parent [`gen_I`;`micro_def`;`phase`;`uinst`];;

new_autoload_theory `ucode_def`;;

%-----------------------------------------------------------------
 From micro_def
-----------------------------------------------------------------%
let load_micro_inst = (\x. theorem `micro_def` x);;

%-----------------------------------------------------------------
: thm list
Run time: 2824.7s
-----------------------------------------------------------------%
let instructions = map load_micro_inst
```

```
   [`FETCH`;`ISSUE`;
    `DECODE`;`NOOP_u1`;
    `JMP_u1`;`CALL_u1`;
    `INT_u1`;`RTI_u1`;
    `GPSW_u1`;`PPSW_u1`;
    `LD_u1`;`ST_u1`;
    `LSL_u1`;`LSR_u1`;
    `ASR_u1`;`RTN_u1`;
    `NOOP_u1`;`NOOP_u1`;
    `LDI_u1`;`STI_u1`;
    `ADD_u1`;`ADDC_u1`;
    `SUB_u1`;`SUBC_u1`;
    `BAND_u1`;`BOR_u1`;
    `BXOR_u1`;`BNOT_u1`;
    `ADDI_u1`;`ADDCI_u1`;
    `SUBI_u1`;`SUBCI_u1`;
    `BANDI_u1`;`BORI_u1`;
    `BXORI_u1`;`NOOP_u1`;
    `CALL_u2`;`CALL_u3`;
    `CALL_u4`;`INT_u2`;
    `INT_u3`;`INT_u4`;
    `RTI_u2`;`RTI_u3`;
    `RTN_u2`;`LD_u2`;
    `ST_u2`;`ST_u3`;
    `STI_u2`;`EINT_u1`;
    `EINT_u2`;`EINT_u3`;
    `EINT_u4`;`LD_u3`;
    `NOOP_u1`;`NOOP_u1`;
    `NOOP_u1`;`NOOP_u1`;
    `NOOP_u1`;`NOOP_u1`;
    `NOOP_u1`;`NOOP_u1`;
    `NOOP_u1`;`NOOP_u1`];;

let micro_inst_list = definition `micro_def` `micro_inst_list`;;

let GetMPC = definition `micro_def` `GetMPC`;;

%-----------------------------------------------------------------
 From phase_def
-----------------------------------------------------------------%
let load_phase_inst = (\x. definition `phase_def` x);;

let phases = map load_phase_inst
   [`phase_one_def`;`phase_two_def`;`phase_three_def`;`phase_four_def`];;

let Phase_Substate = definition `phase_def` `Phase_Substate`;;

let GetPhaseClock = definition `phase_def` `GetPhaseClock`;;

let PhaseClockBegin = definition `phase_def` `PhaseClockBegin`;;

let ALU_FUNC = definition `phase_def` `ALU_FUNC`;;

let ALU_CARRY_FUNC = definition `phase_def` `ALU_CARRY_FUNC`;;
```

```
let ALU_NEG_FUNC = definition `phase_def` `ALU_NEG_FUNC`;;

let ALU_ZERO_FUNC = definition `phase_def` `ALU_ZERO_FUNC`;;

let ALU_OVFL_FUNC = definition `phase_def` `ALU_OVFL_FUNC`;;

let SHIFTER_FUNC = definition `phase_def` `SHIFTER_FUNC`;;

let SHIFTER_CARRY_FUNC = definition `phase_def` `SHIFTER_CARRY_FUNC`;;


let Phase_Int = theorem `phase` `Phase_Int`;;

%----------------------------------------------------------------
 Misc. stuff
----------------------------------------------------------------%
let Next = definition `time_abs` `Next`;;

let micro_rom_expanded = theorem `uinst` `micro_rom_expanded`;;

let MPC_UNIT =
    BETA_RULE (
    EXPAND_LET_RULE (
    definition `mpc_def` `MPC_UNIT`));;

%----------------------------------------------------------------
 The representation types
----------------------------------------------------------------%
let rep_ty = abstract_type `aux_def` `opcode`;;

let I_rep_ty = abstract_type `gen_I` `Impl`;;

let micro_state = ":((*wordn)list#**wordn#**wordn#**memory#
                     *wordn#**wordn#**wordn#**wordn#bt6)";;

let micro_env = ":bool";;


let Phase_state =
    ":((*wordn)list#**wordn#**wordn#**memory#
       *wordn#**wordn#**wordn#**wordn#bt6#
       *wordn#**wordn#bool#bool#ucode#(num->ucode)#bt2)";;

let Phase_env = ":bool";;

%----------------------------------------------------------------
 Define the micro level interpeter in terms of the generic
 interpreter definition.
----------------------------------------------------------------%



let Micro_Int_def = new_definition
   (`Micro_Int_def`,
    "! (rep:^rep_ty) (s:time->^micro_state) (e:time->^micro_env) .
```

```
     Micro_Int rep s e =
        INTERP
           (micro_inst_list rep,
            bt6_val, GetMPC,
            Phase_Substate rep, I, Phase_Int rep,
            GetPhaseClock rep, PhaseClockBegin, @x:one.F) s e"
   );;


let Micro_Int = save_thm
   (`Micro_Int`,
    ONCE_REWRITE_RULE [GetMPC] (
    BETA_RULE (
    EXPAND_LET_RULE
       (instantiate_abstract_definition `gen_I` `INTERP` Micro_Int_def)))
   );;

%----------------------------------------------------------------
Micro_Int =
|- !rep s e.
    Micro_Int rep s e =
    (!t.
      s(t + 1) =
      SND
      (EL(bt6_val(GetMPC(s t)(e t)))(micro_inst_list rep))
      (s t)
      (e t))
Run time: 15.4s
Intermediate theorems generated: 921
----------------------------------------------------------------%


let Micro_Int_Inst_Correct_def = new_definition
   (`Micro_Int_Inst_Correct_def`,
    "! (rep:^rep_ty) (s:time->^Phase_state) (e:time->^Phase_env) .
     Micro_Int_Inst_Correct rep s e =
        INST_CORRECT
           (micro_inst_list rep,
            bt6_val, GetMPC,
            Phase_Substate rep, I, Phase_Int rep,
            GetPhaseClock rep, PhaseClockBegin, @x:one.F) s e"
   );;

let Micro_Int_Inst_Correct =
    let Micro_Int_EXT =
       CONV_RULE (TOP_DEPTH_CONV FUN_EQ_CONV) Micro_Int_Inst_Correct_def in
    (REWRITE_RULE [I_THM] (
     BETA_RULE (
     EXPAND_LET_RULE (
     instantiate_abstract_definition
           `gen_I`
           `INST_CORRECT`
           Micro_Int_EXT))));;

%----------------------------------------------------------------
```

```
Micro_Int_Inst_Correct =
|- !rep s e p.
    Micro_Int_Inst_Correct rep s e p =
    Phase_Int rep s e ==>
    (!t.
      (GetMPC(Phase_Substate rep(s t))(e t) = FST p) /\
      (GetPhaseClock rep(s t)(e t) = PhaseClockBegin) ==>
      (?c.
        Next
        (\t'. GetPhaseClock rep(s t')(e t') = PhaseClockBegin)
        (t,t + c) /\
        (SND p(Phase_Substate rep(s t))(e t) =
         Phase_Substate rep(s(t + c)))))
----------------------------------------------------------------%

map (delete_cache o fst) (cached_theories());;

%----------------------------------------------------------------
 Some ML function for the inference rules that follow.
----------------------------------------------------------------%
let last l = (el (length l) l);;

letrec term_list_el n l = (
   let tm_hd x = rand(fst(dest_comb x)) and
       tm_tl x = snd(dest_comb x) in
   if (n = 0) then  tm_hd l else
   term_list_el (n-1) (tm_tl l)) ?
   failwith `term_list_el`;;


%----------------------------------------------------------------
 This is insecure for right now.  If anyone is seriously concerned
 that this isn't right, I'll do it over.
----------------------------------------------------------------%
let EL_CONV tm = (
   let ((c,n),l) = ((dest_comb#I)o dest_comb) tm in
   let n_int = term_to_int n in
   mk_thm([],"^tm = ^(term_list_el n_int l)")) ?
   failwith `EL_CONV`;;

%----------------------------------------------------------------
 Some other nice conversions
----------------------------------------------------------------%
let is_SND_term t =
  if is_comb t then
    fst(dest_const(fst(strip_comb t))) = `SND`
  else
    false;;

%----------------------------------------------------------------
   SND_CONV  "SND (x,y)" --> |- SND (x,y) = y
----------------------------------------------------------------%
let SND_CONV t =
   if is_SND_term t then
      let op,pr = dest_comb t in
```

```
      let op,[t1;t2] = strip_comb pr in
      SPECL [t1;t2] (
         INST_TYPE [((type_of t1),":*");
                    ((type_of t2),":**")] SND)
   else
      failwith `SND_CONV`;;

%-----------------------------------------------------------------
   ADD_ASSOC_CONV "a+(b+c)"  --> |- a +(b+c) = (a+b)+c
-----------------------------------------------------------------%
let ADD_ASSOC_CONV t =
 let op1,[t1;t2] = strip_comb t
 in
 let op2,[t3;t4] = strip_comb t2
 in
 if op1 = "$+"  & op2 = "$+"
  then SPECL[t1;t3;t4]ADD_ASSOC
  else fail;;

%-----------------------------------------------------------------
   INV_ADD_ASSOC_CONV "(a+b)+c"  --> |- (a+b)+c = a+(b+c)
-----------------------------------------------------------------%
let INV_ADD_ASSOC = (GEN_ALL o SYM o SPEC_ALL) ADD_ASSOC;;

let INV_ADD_ASSOC_CONV t =
 let op1,[t1;t2] = strip_comb t
 in
 let op2,[t3;t4] = strip_comb t1
 in
 if op1 = "$+"  & op2 = "$+"
  then SPECL[t3;t4;t2] INV_ADD_ASSOC
  else fail;;

%-----------------------------------------------------------------
 inv_num_CONV inv_num_CONV "(SUC 2)" --> |- SUC 2 = 3
-----------------------------------------------------------------%
let inv_num_CONV n = (
   let x,y = dest_comb n in
   let y_inc = int_to_term ((term_to_int y) + 1) in
   if not(x = "SUC") then fail else
   SYM_RULE (num_CONV y_inc))
   ? failwith `inv_num_CONV`;;



%-----------------------------------------------------------------
Using MK_Phase_Int_Inst_LEMMA, we can prove a lemma of the form

|- Phase_Int
    rep
    (\t.
      (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t,alatch t,
       blatch t,ireq_ff t,iack_ff t,mir t,urom,clk t))
    (\t. (int_e t)) ==>
    (!t.
```

```
      (clk t = F,F) ==>
      (reg(t + 1),psw(t + 1),pc(t + 1),mem(t + 1),ivec(t + 1),ir(t + 1),
       mar(t + 1),mbr(t + 1),mpc(t + 1),alatch(t + 1),blatch(t + 1),
       ireq_ff(t + 1),iack_ff(t + 1),mir(t + 1),urom,clk(t + 1) =
       (let new_mir = urom(bt6_val(mpc t))
        and new_clk = F,T
        in
         reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t,alatch t,
         blatch t,ireq_ff t,iack_ff t,new_mir,urom,new_clk)))

----------------------------------------------------------------%

let Phase_Int_SPEC =
    PURE_ONCE_REWRITE_RULE [GetPhaseClock] (
    BETA_RULE (
    SPECL ["rep:^rep_ty";
           "(\t. (reg t,psw t,pc t,mem t,
                  ivec t,ir t,mar t,mbr t,mpc t,
                  alatch t, blatch t, ireq_ff t, iack_ff t,
                  mir t, urom, clk t)):time->^Phase_state";
           "(\t. (int_e t)):time->^Phase_env"] Phase_Int));;


let MK_Phase_Int_Inst_LEMMA inst =
    let tp = mk_n_tuple_from_int 2 inst in
    let clk_term = "clk t = ^tp" in
    DISCH_ALL (
    GEN "t" (
    DISCH clk_term (
    SUBS [SPECL ["rep:^rep_ty";
                 "reg t:(*wordn)list";
                 "mem t:*memory";
                 "psw t:*wordn";
                 "pc t:*wordn";
                 "ivec t:*wordn";
                 "ir t:*wordn";
                 "mar t:*wordn";
                 "mbr t:*wordn";
                 "alatch t:*wordn";
                 "blatch t:*wordn";
                 "mpc t:bt6";
                 tp;
                 "urom:num->ucode";
                 "mir t:ucode";
                 "ireq_ff t:bool";
                 "iack_ff t:bool";
                 "int_e t:bool"] (el (inst+1) phases)] (
    CONV_RULE (DEPTH_CONV SND_CONV) (
    CONV_RULE (ONCE_DEPTH_CONV EL_CONV) (
    SUBS [bt2_val_CONV "bt2_val ^tp"] (
    SUBS [ASSUME clk_term] (
    SPEC_ALL (
    SUBS [Phase_Int_SPEC] (
    ASSUME
      "Phase_Int (rep:^rep_ty)
```

```
          (\t. (reg t,psw t,pc t,mem t,
                ivec t,ir t,mar t,mbr t,mpc t,
                alatch t, blatch t, ireq_ff t, iack_ff t,
                mir t, urom, clk t))
          (\t. (int_e t))"))))))))));;

let mk_num_list n =
   letrec mk_num_list_aux n m =
      if n = m then [m] else
      (n . (mk_num_list_aux (n+1) m)) in
   mk_num_list_aux 0 n;;

let Phase_Int_Inst_list = map MK_Phase_Int_Inst_LEMMA (mk_num_list 3);;

let Micro_Inst_Correct_LEMMA =
    REWRITE_RULE [GetPhaseClock; Phase_Substate;Next;
                  PhaseClockBegin;GetMPC;] (
    BETA_RULE (
    SPECL ["rep:^rep_ty";
           "(\t. (reg t,psw t,pc t,mem t,
                  ivec t,ir t,mar t,mbr t,mpc t,
                  alatch t, blatch t, ireq_ff t, iack_ff t,
                  mir t, micro_rom, clk t)):time->^Phase_state";
           "(\t. (int_e t)):time->^Phase_env"]
          Micro_Int_Inst_Correct));;

let BEGIN_ADDR = "F,F";;

%----------------------------------------------------------------
 Create a goal for instruction n
----------------------------------------------------------------%
let MK_INST_CORRECT_GOAL n =
    let inst = term_list_el n
                (snd(dest_eq(
                 snd(dest_forall(concl micro_inst_list))))) in
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode) (mir:time->ucode)
      (ireq_ff iack_ff int_e:time->bool).
     (!p. mk_psw rep
            (get_sm rep p,get_ie rep p,get_vf rep p,
             get_nf rep p,get_cf rep p,get_zf rep p) = p) ==>
     Micro_Int_Inst_Correct rep
          (\t. (reg t,psw t,pc t,mem t,
                ivec t,ir t,mar t,mbr t,mpc t,
                alatch t, blatch t, ireq_ff t, iack_ff t,
                mir t, micro_rom, clk t))
          (\t. (int_e t)) ^inst";;

let phase_one_expanded =
     EXPAND_LET_RULE (el 1 Phase_Int_Inst_list);;

let phase_two_expanded =
     EXPAND_LET_RULE (el 2 Phase_Int_Inst_list);;
```

```
let phase_three_expanded =
     EXPAND_LET_RULE (el 3 Phase_Int_Inst_list);;

let RANGE_LEMMA = TAC_PROOF
   (([],
    "!t1 t2 (clk:time->bt2) x .
     (!t'. t1 < t' /\  t' < t2 ==> ~(clk t' = x)) /\
      ~(clk t2 = x) ==>
     (!t'. t1 < t' /\ t' < (t2 + 1) ==> ~(clk t' = x))"),
    REPEAT STRIP_TAC
    THEN ASSUM_LIST (\asl. ASSUME_TAC (
       SPEC "t':time" (el 5 asl)))
    THEN ASSUM_LIST (\asl. STRIP_ASSUME_TAC (
       REWRITE_RULE [SYM_RULE ADD1;LESS_THM] (el 3 asl)))
    THENL [
        ASSUM_LIST (\asl. ASSUME_TAC (
           REWRITE_RULE [el 1 asl] (el 3 asl)))
    ;
        ALL_TAC
    ]
    THEN RES_TAC
   );;


let LESS_SQUEEZE_LEMMA =
    let LESS_EQ_SUC =
          SYM_RULE (
          PURE_ONCE_REWRITE_RULE [DISJ_SYM] LESS_THM) in
    PURE_ONCE_REWRITE_RULE [ADD1] (
    PURE_ONCE_REWRITE_RULE [LESS_EQ_SUC] (
    PURE_ONCE_REWRITE_RULE [LESS_OR_EQ] LESS_EQ_ANTISYM));;

%----------------------------------------------------------------
 Specialize the selectors on the ucode for a particular uinst.
----------------------------------------------------------------%
let SPEC_SELECTOR x thm =
    let inst = snd(dest_eq x) in
    let (oper,(psw,(sig,mpc))) = (I # (I # dest_pair)) (
                                 (I # dest_pair) (
                                 (dest_pair inst))) in
    let (ax,sh,al,mb,ma,pc,tg,sa,sb) =
                (I # (I # (I # (I # (I # (I # (I # dest_pair))))))) (
                (I # (I # (I # (I # (I # (I # dest_pair)))))) (
                (I # (I # (I # (I # (I # dest_pair))))) (
                (I # (I # (I # (I # dest_pair)))) (
                (I # (I # (I # dest_pair))) (
                (I # (I # dest_pair)) (
                (I # dest_pair) (
                (dest_pair oper)))))))) in
    let (ssm,csm,sie,cie,lcf,lvf,lnf,lzf,lal) =
                (I # (I # (I # (I # (I # (I # (I # dest_pair))))))) (
                (I # (I # (I # (I # (I # (I # dest_pair)))))) (
                (I # (I # (I # (I # (I # dest_pair))))) (
                (I # (I # (I # (I # dest_pair)))) (
                (I # (I # (I # dest_pair))) (
```

```
                (I # (I # dest_pair)) (
                (I # dest_pair) (
                (dest_pair psw))))))) in
    let (ia,f,r,w) =
                (I # (I # dest_pair)) (
                (I # dest_pair) (
                (dest_pair sig))) in
    let (jc,ad) = dest_pair mpc in
    SPECL [ax;sh;al;ma;mb;pc;r;w;ia;f;
          ssm;csm;sie;cie;lcf;lvf;lnf;lzf;lal;
          tg;sa;sb;jc;ad] thm;;

let SPEC_ALL_SELECTORS x =
   map (SPEC_SELECTOR x)
       [Amux;Shift;Alu;Mbr;Mar;Pmux;Trgt;SrcA;SrcB;
        S_sm;C_sm;S_ie;C_ie;Ld_c;Ld_v;Ld_n;Ld_z;
        Csrc;Iack;Ftch;Rd;Wr;Cond;Address];;

map (delete_cache o fst) (cached_theories());;

%----------------------------------------------------------------
 Prove the instruction correctness lemma for instruction n
----------------------------------------------------------------%
let INST_CORRECT_TAC n =
    let inst = term_list_el n
                (snd(dest_eq(
                 snd(dest_forall(concl micro_inst_list))))) in
    let thm = el (n+1) instructions in
    let find_Phase_Int_term tm = (
        let ((x,y),z) = ((dest_comb # I)
                          (dest_comb tm)) in
        (x = "Phase_Int (rep:^rep_ty)")) ? false in (
    REPEAT STRIP_TAC
    THEN SUBST_TAC [SPEC inst Micro_Inst_Correct_LEMMA]
    THEN ASM_REWRITE_TAC [thm]
    THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC (
         CONJUNCTS (
         REWRITE_RULE [PAIR_EQ] (
         SUBS [CONV_RULE (ONCE_DEPTH_CONV EL_CONV) (
               SPEC (int_to_term n) micro_rom_expanded)] (
         CONV_RULE (ONCE_DEPTH_CONV bt6_val_CONV) (
         SUBS [el 2 x] (
         (\y. MP y (el 1 x)) (
         SPEC "t:time" (
         MATCH_MP phase_one_expanded
                 (hd (filter (find_Phase_Int_term o concl) x)) )))))))))
    THEN ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC (
         CONJUNCTS (
         REWRITE_RULE [PAIR_EQ] (
         SUBS (SPEC_ALL_SELECTORS (concl (el 2 x))) (
         SUBS [el 2 x] (
         (\y. MP y (el 1 x)) (
         SPEC "t+1" (
         MATCH_MP phase_two_expanded
```

```
                  (hd (filter (find_Phase_Int_term o concl) x)) ))))))))
     THEN ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC (
          CONJUNCTS (
          REWRITE_RULE [PAIR_EQ] (
          SUBS (SPEC_ALL_SELECTORS (concl (el 2 x))) (
          SUBS [el 2 x] (
          (\y. MP y (el 1 x)) (
          SPEC "(t+1)+1" (
          MATCH_MP phase_three_expanded
                  (hd (filter (find_Phase_Int_term o concl) x)) ))))))))
     THEN ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC (
          CONJUNCTS (
          REWRITE_RULE [PAIR_EQ] (
          EXPAND_LET_RULE (
          REWRITE_RULE [PAIR_EQ;
                        ALU_FUNC;ALU_CARRY_FUNC;ALU_OVFL_FUNC;
                        ALU_NEG_FUNC;ALU_ZERO_FUNC;SHIFTER_FUNC;
                        SHIFTER_CARRY_FUNC] (
          SUBS (SPEC_ALL_SELECTORS (concl (el 2 x))) (
          SUBS [el 2 x] (
          (\y. MP y (el 1 x)) (
          SPEC "((t+1)+1)+1" (
          MATCH_MP (el 4 Phase_Int_Inst_list)
                  (hd (filter (find_Phase_Int_term o concl) x)))))))))))
     THEN EXISTS_TAC "((1 + 1) + 1) + 1"
     THEN CONV_TAC (TOP_DEPTH_CONV ADD_ASSOC_CONV)
     THEN BETA_TAC
     THEN ASM_REWRITE_TAC [PAIR_EQ;MPC_UNIT]
     THEN REPEAT CONJ_TAC
     THEN FIRST [ % 1 %
         GEN_TAC
         THEN SPEC_TAC ("t':time","t':time")
         THEN PURE_ONCE_REWRITE_TAC [ADD1]
         THEN CONV_TAC (TOP_DEPTH_CONV ADD_ASSOC_CONV)
         THEN REPEAT (
             ((MATCH_MP_TAC RANGE_LEMMA) ORELSE ALL_TAC)
             THEN CONJ_TAC
             THEN ONCE_REWRITE_TAC [LESS_SQUEEZE_LEMMA])
         THEN ASM_REWRITE_TAC [PAIR_EQ]
     ;
         PURE_ONCE_REWRITE_TAC [SYM_RULE ADD1]
         THEN CONV_TAC (TOP_DEPTH_CONV INV_ADD_ASSOC_CONV)
         THEN REWRITE_TAC [
             REWRITE_RULE [ADD_CLAUSES;NOT_SUC] (
             GEN_ALL (
             SPECL ["m:num";"SUC n"] LESS_ADD_NONZERO))]
     ;
         ALL_TAC
     ]);;

let PROVE_INST_CORRECT_LEMMA n = (
   TAC_PROOF (([],
       MK_INST_CORRECT_GOAL n),
       INST_CORRECT_TAC n))
   ? BOOL_CASES_AX;;
```

```
%----------------------------------------------------------------
 Save lemmas for recovery in the event of a crash.
----------------------------------------------------------------%
let SAVE_INST_LEMMA n =
   let name = (concat `INST_` (string_of_int n)) in
   save_thm(name,PROVE_INST_CORRECT_LEMMA n);;


map (delete_cache o fst) (cached_theories());;

letrec mk_num_list n m =
      if n = m then [m] else
      (n . (mk_num_list (n+1) m));;

let inst_lemma_list =
     (map SAVE_INST_LEMMA (mk_num_list 0 15));;

map (delete_cache o fst) (cached_theories());;

let inst_lemma_list =
     inst_lemma_list @
     (map SAVE_INST_LEMMA (mk_num_list 16 31));;

map (delete_cache o fst) (cached_theories());;

let inst_lemma_list =
     inst_lemma_list @
     (map SAVE_INST_LEMMA (mk_num_list 32 47));;

map (delete_cache o fst) (cached_theories());;

let inst_lemma_list =
     inst_lemma_list @
     (map SAVE_INST_LEMMA (mk_num_list 48 63));;

map (delete_cache o fst) (cached_theories());;

%----------------------------------------------------------------
 The first obligation of the abstract interpreter theory
----------------------------------------------------------------%

let Micro_Int_CORRECT_LEMMA_AUX = TAC_PROOF
   (([],
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2) (urom:num->ucode) (mir:time->ucode)
      (ireq_ff iack_ff int_e:time->bool).
     (!p. mk_psw rep
            (get_sm rep p,get_ie rep p,get_vf rep p,
             get_nf rep p,get_cf rep p,get_zf rep p) = p) ==>
      EVERY (Micro_Int_Inst_Correct rep
             (\t. (reg t,psw t,pc t,mem t,
```

```
                 ivec t,ir t,mar t,mbr t,mpc t,
                 alatch t, blatch t, ireq_ff t, iack_ff t,
                 mir t, micro_rom, clk t))
             (\t. (int_e t))) (micro_inst_list rep)"),
    REWRITE_TAC [EVERY_DEF;micro_inst_list]
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM (\asl. MP_TAC asl)
    THENL (map MATCH_ACCEPT_TAC inst_lemma_list)
   );;

let Micro_Int_CORRECT_LEMMA = (
    UNDISCH_ALL (
    SPEC_ALL (
    PURE_ONCE_REWRITE_RULE [Micro_Int_Inst_Correct_def]
        Micro_Int_CORRECT_LEMMA_AUX)));;

save_thm(`Micro_Int_CORRECT_LEMMA`,Micro_Int_CORRECT_LEMMA);;

%----------------------------------------------------------------
 The second obligation of the abstract interpreter theory
----------------------------------------------------------------%
let Micro_Int_LENGTH_LEMMA = TAC_PROOF
   (([],
    "! mpc. bt6_val mpc < (LENGTH (micro_inst_list (rep:^rep_ty)))"),
    REPEAT GEN_TAC
    THEN REWRITE_TAC [micro_inst_list;LENGTH]
    THEN STRUCT_CASES_TAC (SPEC "mpc:bt6" SIX_TUPLE_VALUE_LEMMA)
    THEN CONV_TAC (DEPTH_CONV bt6_val_CONV)
    THEN CONV_TAC (TOP_DEPTH_CONV num_CONV)
    THEN REWRITE_TAC [LESS_0;LESS_MONO_EQ]
   );;

save_thm(`Micro_Int_LENGTH_LEMMA`,Micro_Int_LENGTH_LEMMA);;

map (delete_cache o fst) (cached_theories());;

%----------------------------------------------------------------
 The third obligation of the abstract interpreter theory
----------------------------------------------------------------%
let Micro_Int_ORDER_LEMMA = TAC_PROOF
   (([],
    "!mpc:bt6 . mpc = (FST (EL (bt6_val mpc)
                                (micro_inst_list (rep:^rep_ty))))"),
    REPEAT GEN_TAC
    THEN SUBST_TAC [SPEC "rep:^rep_ty" micro_inst_list]
    THEN STRUCT_CASES_TAC (SPEC "mpc:bt6" SIX_TUPLE_VALUE_LEMMA)
    THEN CONV_TAC (ONCE_DEPTH_CONV bt6_val_CONV)
    THEN CONV_TAC (ONCE_DEPTH_CONV EL_CONV)
    THEN REWRITE_TAC []
   );;

save_thm(`Micro_Int_ORDER_LEMMA`,Micro_Int_ORDER_LEMMA);;

map (delete_cache o fst) (cached_theories());;
```

```
let theorem_list =
    instantiate_abstract_theorems
        `gen_I`
        [Micro_Int_CORRECT_LEMMA;
         Micro_Int_LENGTH_LEMMA;
         Micro_Int_ORDER_LEMMA]
        [
         ("rep:^I_rep_ty",
          "(micro_inst_list (rep:^rep_ty),
            bt6_val,
            GetMPC:^micro_state->^micro_env->bt6,
            (Phase_Substate rep):^phase_state->^micro_state,
            (I:^phase_env->^micro_env),
            Phase_Int rep,
            (GetPhaseClock rep):^phase_state->^phase_env->bt2,
            PhaseClockBegin:bt2,@x:one.F)");
         ("e':time'->*env'",
          "(\t:time. (int_e t):bool)");
         ("s':time->*state'",
          "(\t. (reg t,psw t,pc t,mem t,
                 ivec t,ir t,mar t,mbr t,mpc t,
                 alatch t, blatch t, ireq_ff t, iack_ff t,
                 mir t, micro_rom, clk t)):time->^phase_state")
       ]
      `MICRO`;;

let correct_lemma = snd(hd theorem_list);;


let MICRO_LEVEL_CORRECT_LEMMA = save_thm
   (`MICRO_LEVEL_CORRECT_LEMMA`,
    BETA_RULE (
    EXPAND_LET_RULE (
    ONCE_REWRITE_RULE [Phase_Substate;I_THM;GetPhaseClock;PhaseClockBegin] (
    BETA_RULE (
    ONCE_REWRITE_RULE [SYM_RULE Micro_Int_def] correct_lemma))))
   );;
```

3.7 The Macro–Level

This section presents the theories that define the macro–level interpreter. Also presented is the theory that verifies the macro–level interpreter with respect to the micro–level interpreter.

3.7.1 The Macro–Level Interpreter

The section presents the ML code that creates the theory `macro_def.th`.

```
%----------------------------------------------------------------

   File:         def_macro.ml

   Author:       (c) P. J. Windley 1989

   Date:         24 OCT 89

   Modified:     03 APR 90

   Description:

   Defines the behavioral description of the macro interpreter
   level

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;


let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`numbers/`; `decimal/`; `assoc/`; `tuple/`]));;

loadf `abstract`;;

system `/bin/rm macro_def.th`;;

new_theory `macro_def`;;

map new_parent [`aux_def`; `tuple`; `aux_thms`;
                `regs_def`; `jump_def`];;

let rep_ty = abstract_type `aux_def` `opcode`;;

%----------------------------------------------------------------'
The instruction formats are given below:
```

```
    Format 1:

     31       25     20     15     10            0
    +--------+------+------+------+--------------+
    | opcode | dest |  A   |  B   | unused       |
    +--------+------+------+------+--------------+

    Format 2:

     31       25     20     15                   0
    +--------+------+------+---------------------+
    | opcode | dest |  A   | imm                 |
    +--------+------+------+---------------------+

The following instructions select fields from the instructions.

---------------------------------------------------------------%

let GetSrcA = new_definition
   (`GetSrcA`,
    "! (rep:^rep_ty) mem reg .
     GetSrcA rep reg mem =
      reg_len rep (srca rep (fetch rep (mem, address rep reg)))"
   );;

let GetSrcB = new_definition
   (`GetSrcB`,
    "! (rep:^rep_ty) mem reg .
     GetSrcB rep reg mem =
      reg_len rep (srcb rep (fetch rep (mem, address rep reg)))"
   );;

let GetImm = new_definition
   (`GetImm`,
    "! (rep:^rep_ty) mem reg .
     GetImm rep reg mem =
      (imm rep (fetch rep (mem, address rep reg)))"
   );;

let GetDest = new_definition
   (`GetDest`,
    "! (rep:^rep_ty) mem reg .
     GetDest rep reg mem =
      reg_len rep (dest rep (fetch rep (mem, address rep reg)))"
   );;

%---------------------------------------------------------------
Arithmetic functions:
---------------------------------------------------------------%

let ADD = new_definition
   (`ADD`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     ADD rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
```

```
            b = EL (GetSrcB rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = add rep (a, b) in
        let cflag  = addp rep (a, b, result) and
            vflag  = aovfl rep (a, b, result) and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

let ADDC = new_definition
   (`ADDC`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     ADDC rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            b = EL (GetSrcB rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = addc rep (a, b, get_cf rep psw) in
        let cflag  = addcp rep (a, b, result) and
            vflag  = aovfl rep (a, b, result) and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

let SUB = new_definition
   (`SUB`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     SUB rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            b = EL (GetSrcB rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = sub rep (a, b) in
        let cflag  = subp rep (a, b, result) and
            vflag  = sovfl rep (a, b, result) and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
```

```
         ivec)"
   );;

let SUBC = new_definition
   (`SUBC`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     SUBC rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            b = EL (GetSrcB rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = subc rep (a, b, get_cf rep psw) in
        let cflag  = subp rep (a, b, result) and
            vflag  = sovfl rep (a, b, result) and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

%----------------------------------------------------------------
Immediate arithmetic functions:
----------------------------------------------------------------%
let ADDI = new_definition
   (`ADDI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     ADDI rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem in
        let result = add rep (a, i) in
        let cflag  = addp rep (a, i, result) and
            vflag  = aovfl rep (a, i, result) and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

let ADDCI = new_definition
   (`ADDCI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     ADDCI rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem in
```

```
        let result = addc rep (a, i, get_cf rep psw) in
        let cflag  = addcp rep (a, i, result) and
            vflag  = aovfl rep (a, i, result) and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
       (UPDATE_REG rep psw d reg result,
        mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
        inc rep pc,
        mem,
        ivec)"
   );;

let SUBI = new_definition
   (`SUBI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     SUBI rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem in
        let result = sub rep (a, i) in
        let cflag  = subp rep (a, i, result) and
            vflag  = sovfl rep (a, i, result) and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
       (UPDATE_REG rep psw d reg result,
        mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
        inc rep pc,
        mem,
        ivec)"
   );;

let SUBCI = new_definition
   (`SUBCI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     SUBCI rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem in
        let result = subc rep (a, i, get_cf rep psw) in
        let cflag  = subp rep (a, i, result) and
            vflag  = sovfl rep (a, i, result) and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
       (UPDATE_REG rep psw d reg result,
        mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
        inc rep pc,
        mem,
        ivec)"
   );;
```

```
%----------------------------------------------------------------
Shifting functions:
----------------------------------------------------------------%
let LSL = new_definition
   (`LSL`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     LSL rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = shl rep a in
        let cflag  = msb rep a and
            vflag  = get_vf rep psw and
            nflag  = get_nf rep psw and
            zflag  = get_zf rep psw and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

let LSR = new_definition
   (`LSR`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     LSR rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = shr rep a in
        let cflag  = lsb rep a and
            vflag  = get_vf rep psw and
            nflag  = get_nf rep psw and
            zflag  = get_zf rep psw and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

let ASR = new_definition
   (`ASR`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     ASR rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = asr rep a in
        let cflag  = lsb rep a and
            vflag  = get_vf rep psw and
            nflag  = get_nf rep psw and
            zflag  = get_zf rep psw and
```

```
           sm     = get_sm rep psw and
           ie     = get_ie rep psw in
       (UPDATE_REG rep psw d reg result,
        mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
        inc rep pc,
        mem,
        ivec)"
   );;

%-----------------------------------------------------------------
Logical functions:
-----------------------------------------------------------------%
let BAND = new_definition
   (`BAND`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     BAND rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            b = EL (GetSrcB rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = band rep (a, b) in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

let BOR = new_definition
   (`BOR`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     BOR rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            b = EL (GetSrcB rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = bor rep (a, b) in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

let BXOR = new_definition
```

```
   (`BXOR`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     BXOR rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            b = EL (GetSrcB rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = bxor rep (a, b) in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

let BNOT = new_definition
   (`BNOT`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     BNOT rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            b = EL (GetSrcB rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = bnot rep a in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

%-----------------------------------------------------------------
Immediate Logical functions:
-----------------------------------------------------------------%
let BANDI = new_definition
   (`BANDI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     BANDI rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem in
        let result = band rep (a, i) in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = negp rep result and
```

```
          zflag  = zerop rep result and
          sm     = get_sm rep psw and
          ie     = get_ie rep psw in
      (UPDATE_REG rep psw d reg result,
       mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
       inc rep pc,
       mem,
       ivec)"
   );;

let BORI = new_definition
   (`BORI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     BORI rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem in
        let result = bor rep (a, i) in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

let BXORI = new_definition
   (`BXORI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     BXORI rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem in
        let result = bxor rep (a, i) in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = negp rep result and
            zflag  = zerop rep result and
            sm     = get_sm rep psw and
            ie     = get_ie rep psw in
        (UPDATE_REG rep psw d reg result,
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         inc rep pc,
         mem,
         ivec)"
   );;

%-----------------------------------------------------------------
Load and Store
-----------------------------------------------------------------%
```

```
let LD = new_definition
   (`LD`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     LD rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            b = EL (GetSrcB rep pc mem) reg and
            d = GetDest rep pc mem in
        let result = fetch rep (mem, address rep (add rep (a, b))) in
        (UPDATE_REG rep psw d reg result,
         psw,
         inc rep pc,
         mem,
         ivec)"
   );;

let ST = new_definition
   (`ST`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     ST rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            b = EL (GetSrcB rep pc mem) reg and
            d = EL (GetDest rep pc mem) reg in
        let new_address = address rep (add rep (a, b)) in
        (reg,
         psw,
         inc rep pc,
         store rep (mem, new_address, d),
         ivec)"
   );;

%-----------------------------------------------------------------
Immediate Load and Store:
-----------------------------------------------------------------%
let LDI = new_definition
   (`LDI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     LDI rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem in
        let result = fetch rep (mem, address rep (add rep (a, i))) in
        (UPDATE_REG rep psw d reg result,
         psw,
         inc rep pc,
         mem,
         ivec)"
   );;

let STI = new_definition
   (`STI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     STI rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = EL (GetDest rep pc mem) reg in
```

```
        let new_address = address rep (add rep (a, i)) in
        (reg,
         psw,
         inc rep pc,
         store rep (mem, new_address, d),
         ivec)"
   );;

%----------------------------------------------------------------
Jump
----------------------------------------------------------------%
let JMP = new_definition
   (`JMP`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     JMP rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem in
        let jump_cond = JUMP_COND rep d psw in
        (reg,
         psw,
         (jump_cond => (add rep (a, i)) | inc rep pc),
         mem,
         ivec)"
   );;

%----------------------------------------------------------------
CALL a subroutine
----------------------------------------------------------------%
let CALL = new_definition
   (`CALL`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     CALL rep (reg, psw, pc, mem, ivec) =
        let a = EL (GetSrcA rep pc mem) reg and
            i = GetImm rep pc mem and
            d = GetDest rep pc mem and
            cd = (EL (GetDest rep pc mem) reg) in
        (UPDATE_REG rep psw d reg (inc rep cd),
         psw,
         add rep (a, i),
         store rep (mem, address rep cd, inc rep pc),
         ivec)"
   );;

let RTN = new_definition
   (`RTN`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     RTN rep (reg, psw, pc, mem, ivec) =
        let cd = EL (GetDest rep pc mem) reg and
            d = GetDest rep pc mem in
        (UPDATE_REG rep psw d reg (dec rep cd),
         psw,
         fetch rep (mem, address rep (dec rep cd)),
         mem,
         ivec)"
```

```
   );;

%----------------------------------------------------------------
Interrupt instruction
----------------------------------------------------------------%
let INT = new_definition
   (`INT`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     INT rep (reg, psw, pc, mem, ivec) =
        let i = GetImm rep pc mem in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = get_nf rep psw and
            zflag  = get_zf rep psw and
            sm     = T and
            ie     = F in
        let new_psw = mk_psw rep (sm, ie, vflag, nflag, cflag, zflag) in
        (UPDATE_REG rep new_psw ssp_reg reg (inc rep (SSP_REG reg)),
         new_psw,
         band rep (wordn rep 255, i),
         store rep (mem, address rep (SSP_REG reg), inc rep pc),
         ivec)"
   );;

let RTI = new_definition
   (`RTI`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     RTI rep (reg, psw, pc, mem, ivec) =
        let cd = SSP_REG reg in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = get_nf rep psw and
            zflag  = get_zf rep psw and
            sm     = F and
            ie     = T in
        (UPDATE_REG rep psw ssp_reg reg (dec rep cd),
         mk_psw rep (sm, ie, vflag, nflag, cflag, zflag),
         fetch rep (mem, address rep (dec rep cd)),
         mem,
         ivec)"
   );;


%----------------------------------------------------------------
Get and put program status word

For future reference, it would be nice to store the psw banded
with imm.
----------------------------------------------------------------%
let GPSW = new_definition
   (`GPSW`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     GPSW rep (reg, psw, pc, mem, ivec) =
        let d = GetDest rep pc mem in
        (UPDATE_REG rep psw d reg psw,
```

```
         psw,
         inc rep pc,
         mem,
         ivec)"
   );;


let PPSW = new_definition
   (`PPSW`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     PPSW rep (reg, psw, pc, mem, ivec) =
        let d = EL (GetDest rep pc mem) reg in
        let sm  = get_sm rep psw in
        (reg,
         (sm => d | psw),
         inc rep pc,
         mem,
         ivec)"
   );;

%----------------------------------------------------------------
No operation
----------------------------------------------------------------%
let NOOP = new_definition
   (`NOOP`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     NOOP rep (reg, psw, pc, mem, ivec) =
        (reg,
         psw,
         inc rep pc,
         mem,
         ivec)"
   );;


%----------------------------------------------------------------
 Pseudoinstruction for external interrupt
----------------------------------------------------------------%
let EINT = new_definition
   (`EINT`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn) .
     EINT rep (reg, psw, pc, mem, ivec) =
        let cd = SSP_REG reg and
            d = ssp_reg in
        let cflag  = get_cf rep psw and
            vflag  = get_vf rep psw and
            nflag  = get_nf rep psw and
            zflag  = get_zf rep psw and
            sm     = T and
            ie     = F in
        let new_psw =  mk_psw rep (sm, ie, vflag, nflag, cflag, zflag) in
        (UPDATE_REG rep new_psw d reg (inc rep cd),
         new_psw,
         band rep (wordn rep 255, int_fetch rep ivec),
         store rep (mem, address rep cd, pc),
```

```
        ivec)"
   );;


let macro_state = ":((*wordn)list#*wordn#*wordn#*memory)";;

let macro_env = ":bool";;

%-----------------------------------------------------------------
 ABS_ENV takes a function of type (macro_state -> macro_state)
 and creates a function of type (macro_state -> macro_env -> macro_state).
 The purpose of this function is to make the functions defining the
 instructions have the right type for use in the instruction list.
-----------------------------------------------------------------%
let ABS_ENV = new_definition
   (`ABS_ENV`,
    "! (f:^macro_state->^macro_state) (x:^macro_state) (y:^macro_env) .
      ABS_ENV f x y = f x"
   );;

%-----------------------------------------------------------------
 The macro_inst_list will be used to instantiate inst_list in
 mk_macro.ml.
-----------------------------------------------------------------%
let macro_inst_list = new_definition
   (`macro_inst_list`,
    "! rep:^rep_ty .
     macro_inst_list rep =
     [(INL(F,F,F,F,F),ABS_ENV (JMP rep));
      (INL(F,F,F,F,T),ABS_ENV (CALL rep));
      (INL(F,F,F,T,F),ABS_ENV (INT rep));
      (INL(F,F,F,T,T),ABS_ENV (RTI rep));
      (INL(F,F,T,F,F),ABS_ENV (GPSW rep));
      (INL(F,F,T,F,T),ABS_ENV (PPSW rep));
      (INL(F,F,T,T,F),ABS_ENV (LD rep));
      (INL(F,F,T,T,T),ABS_ENV (ST rep));
      (INL(F,T,F,F,F),ABS_ENV (LSL rep));
      (INL(F,T,F,F,T),ABS_ENV (LSR rep));
      (INL(F,T,F,T,F),ABS_ENV (ASR rep));
      (INL(F,T,F,T,T),ABS_ENV (RTN rep));
      (INL(F,T,T,F,F),ABS_ENV (NOOP rep));
      (INL(F,T,T,F,T),ABS_ENV (NOOP rep));
      (INL(F,T,T,T,F),ABS_ENV (LDI rep));
      (INL(F,T,T,T,T),ABS_ENV (STI rep));
      (INL(T,F,F,F,F),ABS_ENV (ADD rep));
      (INL(T,F,F,F,T),ABS_ENV (ADDC rep));
      (INL(T,F,F,T,F),ABS_ENV (SUB rep));
      (INL(T,F,F,T,T),ABS_ENV (SUBC rep));
      (INL(T,F,T,F,F),ABS_ENV (BAND rep));
      (INL(T,F,T,F,T),ABS_ENV (BOR rep));
      (INL(T,F,T,T,F),ABS_ENV (BXOR rep));
      (INL(T,F,T,T,T),ABS_ENV (BNOT rep));
      (INL(T,T,F,F,F),ABS_ENV (ADDI rep));
      (INL(T,T,F,F,T),ABS_ENV (ADDCI rep));
      (INL(T,T,F,T,F),ABS_ENV (SUBI rep));
```

```
      (INL(T,T,F,T,T),ABS_ENV (SUBCI rep));
      (INL(T,T,T,F,F),ABS_ENV (BANDI rep));
      (INL(T,T,T,F,T),ABS_ENV (BORI rep));
      (INL(T,T,T,T,F),ABS_ENV (BXORI rep));
      (INL(T,T,T,T,T),ABS_ENV (NOOP rep));
      (INR(one),      ABS_ENV (EINT rep));
      ]"
   );;

%----------------------------------------------------------------
 Opcode will be used to instantiate select in mk_macro.ml.
----------------------------------------------------------------%
let Opcode = new_definition
   (`Opcode`,
    "!(rep:^rep_ty) reg mem (psw pc ivec:*wordn)
      (int_e:bool).
     Opcode rep (reg, psw, pc, mem, ivec) (int_e) =
        (int_e /\ (get_ie rep psw)) =>
            INR(one) |
            INL(SND (opcode rep (fetch rep (mem, address rep pc))))"
   );;

%----------------------------------------------------------------
 Opc_Val will be used to instantiate key in mk_macro.ml
----------------------------------------------------------------%
let Opc_Val = new_definition
   (`Opc_Val`,
    "! x .
     Opc_Val (x:((bool#bool#bool#bool#bool) + one)) =
        (ISL x) => (bt5_val (OUTL x))
                 | 32"            % there's only one pseudo instruction %
   );;

let Micro_Substate = new_definition
   (`Micro_Substate`,
    "!(rep:^rep_ty) (reg:(*wordn)list) (mem:*memory)
      (psw pc ivec ir mar mbr :*wordn) (mpc:bt6) .
     Micro_Substate rep (reg, psw, pc, mem, ivec, ir, mar, mbr, mpc) =
                        (reg, psw, pc, trans rep mem, int_trans rep ivec)"
   );;


close_theory();;
```

3.7.2 The Macro–Level Proof

The section presents the ML code that creates the theory `macro.th`.

```
%----------------------------------------------------------------

   File:         mk_macro.ml

   Author:       (c) P. J. Windley 1990

   Date:         JUN 23, 1990

   Modified:

   Description:

   Proves the macro--level correct with respect to the micro--level
   using the generic interpreter theory, micro.th, and macro_def.th.

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`]));;

loadf `abstract`;;

system `/bin/rm macro.th`;;

new_theory `macro`;;

map new_parent [`macro_def`;`gen_I`];;

map loadf [`tuple`;`digit`;`decimal`];;


%----------------------------------------------------------------
 Load stuff from macro_def
----------------------------------------------------------------%

let load_macro_inst = (\x. definition `macro_def` x);;

let macro_defn_list = map load_macro_inst
    [`JMP`;`CALL`;`INT`;`RTI`;`GPSW`;`PPSW`;`LD`;`ST`;
     `LSL`;`LSR`;`ASR`;`RTN`;`NOOP`;`NOOP`;`LDI`;`STI`;
     `ADD`;`ADDC`;`SUB`;`SUBC`;`BAND`;`BOR`;`BXOR`;`BNOT`;
     `ADDI`;`ADDCI`;`SUBI`;`SUBCI`;`BANDI`;`BORI`;`BXORI`;`NOOP`];;
```

```
let GetSrcA = definition `macro_def` `GetSrcA`;;
let GetSrcB = definition `macro_def` `GetSrcB`;;
let GetImm = definition `macro_def` `GetImm`;;
let GetDest = definition `macro_def` `GetDest`;;

let ABS_ENV = definition `macro_def` `ABS_ENV`;;
let Opcode = definition `macro_def` `Opcode`;;
let Opc_Val = definition `macro_def` `Opc_Val`;;
let Micro_Substate = definition `macro_def` `Micro_Substate`;;

let macro_inst_list = definition `macro_def` `macro_inst_list`;;

%-----------------------------------------------------------------
 Load stuff from micro_def.
-----------------------------------------------------------------%

new_parent `micro`;;

let load_micro_inst = (\x. theorem `micro_def` x);;

%-----------------------------------------------------------------
: thm list
Run time: 2824.7s
-----------------------------------------------------------------%
let instructions = map load_micro_inst
   [`FETCH`;`ISSUE`;
    `DECODE`;`NOOP_u1`;
    `JMP_u1`;`CALL_u1`;
    `INT_u1`;`RTI_u1`;
    `GPSW_u1`;`PPSW_u1`;
    `LD_u1`;`ST_u1`;
    `LSL_u1`;`LSR_u1`;
    `ASR_u1`;`RTN_u1`;
    `NOOP_u1`;`NOOP_u1`;
    `LDI_u1`;`STI_u1`;
    `ADD_u1`;`ADDC_u1`;
    `SUB_u1`;`SUBC_u1`;
    `BAND_u1`;`BOR_u1`;
    `BXOR_u1`;`BNOT_u1`;
    `ADDI_u1`;`ADDCI_u1`;
    `SUBI_u1`;`SUBCI_u1`;
    `BANDI_u1`;`BORI_u1`;
    `BXORI_u1`;`NOOP_u1`;
    `CALL_u2`;`CALL_u3`;
    `CALL_u4`;`INT_u2`;
    `INT_u3`;`INT_u4`;
    `RTI_u2`;`RTI_u3`;
    `RTN_u2`;`LD_u2`;
    `ST_u2`;`ST_u3`;
    `STI_u2`;`EINT_u1`;
    `EINT_u2`;`EINT_u3`;
    `EINT_u4`;`LD_u3`;
    `NOOP_u1`;`NOOP_u1`;
    `NOOP_u1`;`NOOP_u1`;
```

```
    `NOOP_u1`;`NOOP_u1`;
    `NOOP_u1`;`NOOP_u1`;
    `NOOP_u1`;`NOOP_u1`];;

let micro_inst_list = definition `micro_def` `micro_inst_list`;;

let GetMPC = definition `micro_def` `GetMPC`;;

%----------------------------------------------------------------
 Other misc. loads.
----------------------------------------------------------------%

let Micro_Int = theorem `micro` `Micro_Int`;;

let Next = definition `time_abs` `Next`;;

let add_bt6 = definition `aux_thms` `add_bt6`;;

let OFFSET_NOT_BEGINNING = theorem `aux_thms` `OFFSET_NOT_BEGINNING`;;


%----------------------------------------------------------------
 Load abstract type definitions.
----------------------------------------------------------------%
let rep_ty = abstract_type `aux_def` `opcode`;;

let I_rep_ty = abstract_type `gen_I` `Impl`;;

%----------------------------------------------------------------
 Define type terms for the state and env.
----------------------------------------------------------------%
let macro_state = ":((*wordn)list#*wordn#*wordn#*memory#*wordn)";;

let macro_env = ":bool";;

let micro_state = ":((*wordn)list#*wordn#*wordn#*memory#
                     *wordn#*wordn#*wordn#*wordn#bt6)";;

let micro_env = ":bool";;

%----------------------------------------------------------------
 Beginning of MPC
----------------------------------------------------------------%
let FETCH_ADDR   = "(F,F,F,F,F,F)";;

%----------------------------------------------------------------
 Offset into microrom lookup table
----------------------------------------------------------------%
let OFFSET = "4";;


%----------------------------------------------------------------
 Define the macro level interpeter in terms of the generic
 interpreter definition.
----------------------------------------------------------------%
```

```
let Macro_Int_def = new_definition
   (`Macro_Int_def`,
    "! (rep:^rep_ty) (s:time->^macro_state) (e:time->^macro_env) .
     Macro_Int rep s e =
        INTERP
           (macro_inst_list rep,
            Opc_Val, Opcode rep,
            (Micro_Substate rep):^micro_state->^macro_state,
            (I:^micro_env->^macro_env),
            (Micro_Int rep):(time->^micro_state)->(time->^micro_env)->bool,
            GetMPC:^micro_state->^micro_env->bt6,
            ^FETCH_ADDR:bt6, @x:one.F) s e"
   );;


let Macro_Int = save_thm
   (`Macro_Int`,
    ONCE_REWRITE_RULE [Opcode] (
    BETA_RULE (
    EXPAND_LET_RULE (
    instantiate_abstract_definition `gen_I` `INTERP` Macro_Int_def)))
   );;

%----------------------------------------------------------------
Macro_Int =
|- !rep s e.
    Macro_Int rep s e =
    (!t.
      s(t + 1) =
      SND
      (EL(Opc_Val(Opcode rep(s t)(e t)))(macro_inst_list rep))
      (s t)
      (e t))
Run time: 21.6s
Intermediate theorems generated: 929
----------------------------------------------------------------%

let Macro_Inst_Correct_def = new_definition
   (`Macro_Inst_Correct_def`,
    "! (rep:^rep_ty) s' e' .
     Macro_Inst_Correct rep s' e' =
        INST_CORRECT
           (macro_inst_list rep,
            Opc_Val, Opcode rep,
            Micro_Substate rep, I, Micro_Int rep,
            GetMPC, ^FETCH_ADDR, @x:one.F) s' e'"
   );;

let Macro_Inst_Correct = save_thm
   (`Macro_Inst_Correct`,
    let Macro_Inst_EXT =
       CONV_RULE (TOP_DEPTH_CONV FUN_EQ_CONV) Macro_Inst_Correct_def in
    REWRITE_RULE [I_THM] (
    BETA_RULE (
    EXPAND_LET_RULE (
```

```
    instantiate_abstract_definition
         `gen_I` `INST_CORRECT` Macro_Inst_EXT)))
   );;

%-----------------------------------------------------------------
Macro_Inst_Correct =
|- !rep s' e' p.
    Macro_Inst_Correct rep s' e' p =
    Micro_Int rep s' e' ==>
    (!t.
      (Opcode rep(Micro_Substate rep(s' t))(e' t) = FST p) /\
      (GetMPC(s' t)(e' t) = F,F,F,F,F,F) ==>
      (?c.
        Next(\t'. GetMPC(s' t')(e' t') = F,F,F,F,F,F)(t,t + c) /\
        (SND p(Micro_Substate rep(s' t))(e' t) =
         Micro_Substate rep(s'(t + c)))))
Run time: 74.3s
Intermediate theorems generated: 4267
-----------------------------------------------------------------%


%-----------------------------------------------------------------
 I need some theorems about SUM not provided in the theory
-----------------------------------------------------------------%
let sum_axiom =
    BETA_RULE (
    REWRITE_RULE [o_DEF] (
    CONV_RULE (TOP_DEPTH_CONV FUN_EQ_CONV) sum_Axiom));;

let INJECTION_ONE_ONE = prove_constructors_one_one sum_axiom;;

let INJECTION_DISTINCT = prove_constructors_distinct sum_axiom;;

let INJ_LEMMA_ONE = TAC_PROOF
  (([],
    "! (b:bool) (x:**) (y z:*) .
       ((b => INR x | INL y) = (INL z)) ==>
        (b = F) /\ (y = z)"),
    REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "b:bool"
    THEN REWRITE_TAC []
    THEN STRIP_TAC
    THEN IMP_RES_TAC (SYM_RULE INJECTION_DISTINCT)
    THEN MATCH_MP_TAC (fst (EQ_IMP_RULE
                           (SPEC_ALL
                           (CONJUNCT1 INJECTION_ONE_ONE))))
    THEN POP_ASSUM (\thm . MATCH_ACCEPT_TAC thm)
   );;


let INJ_LEMMA_TWO = TAC_PROOF
  (([],
    "! (b:bool) (x z:**) (y:*) .
       ((b => INR x | INL y) = (INR z)) ==>
        (b = T) /\ (x = z)"),
```

```
    REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "b:bool"
    THEN REWRITE_TAC []
    THEN STRIP_TAC
    THEN IMP_RES_TAC INJECTION_DISTINCT
    THEN MATCH_MP_TAC (fst (EQ_IMP_RULE
                           (SPEC_ALL
                           (CONJUNCT2 INJECTION_ONE_ONE))))
    THEN POP_ASSUM (\thm . MATCH_ACCEPT_TAC thm)
   );;


%----------------------------------------------------------------
 Some ML function for the inference rules that follow.
----------------------------------------------------------------%
let last l = (el (length l) l);;

letrec term_list_el n l = (
   let tm_hd x = rand(fst(dest_comb x)) and
       tm_tl x = snd(dest_comb x) in
   if (n = 0) then  tm_hd l else
   term_list_el (n-1) (tm_tl l)) ?
   failwith `term_list_el`;;


%----------------------------------------------------------------
 This is insecure for right now.  If anyone is seriously concerned
 that this isn't right, I'll do it over.
----------------------------------------------------------------%
let EL_CONV tm = (
   let ((c,n),l) = ((dest_comb#I)o dest_comb) tm in
   let n_int = term_to_int n in
   mk_thm([],"^tm = ^(term_list_el n_int l)")) ?
   failwith `EL_CONV`;;


%----------------------------------------------------------------
 Some other nice conversions
----------------------------------------------------------------%
let is_SND_term t =
  if is_comb t then
    fst(dest_const(fst(strip_comb t))) = `SND`
  else
    false;;

%----------------------------------------------------------------
   SND_CONV  "SND (x,y)" --> |- SND (x,y) = y
----------------------------------------------------------------%
let SND_CONV t =
   if is_SND_term t then
      let op,pr = dest_comb t in
      let op,[t1;t2] = strip_comb pr in
      SPECL [t1;t2] (
         INST_TYPE [((type_of t1),":*");
                    ((type_of t2),":**")] SND)
```

```
   else
      failwith `SND_CONV`;;

%-----------------------------------------------------------------
   ADD_ASSOC_CONV "a+(b+c)"  --> |- a +(b+c) = (a+b)+c
-----------------------------------------------------------------%
let ADD_ASSOC_CONV t =
 let op1,[t1;t2] = strip_comb t
 in
 let op2,[t3;t4] = strip_comb t2
 in
 if op1 = "$+"  & op2 = "$+"
  then SPECL[t1;t3;t4]ADD_ASSOC
  else fail;;


%-----------------------------------------------------------------
   INV_ADD_ASSOC_CONV "(a+b)+c"  --> |- (a+b)+c = a+(b+c)
-----------------------------------------------------------------%
let INV_ADD_ASSOC = (GEN_ALL o SYM o SPEC_ALL) ADD_ASSOC;;

let INV_ADD_ASSOC_CONV t =
 let op1,[t1;t2] = strip_comb t
 in
 let op2,[t3;t4] = strip_comb t1
 in
 if op1 = "$+"  & op2 = "$+"
  then SPECL[t3;t4;t2] INV_ADD_ASSOC
  else fail;;

%-----------------------------------------------------------------
 inv_num_CONV inv_num_CONV "(SUC 2)" --> |- SUC 2 = 3
-----------------------------------------------------------------%
let inv_num_CONV n = (
   let x,y = dest_comb n in
   let y_inc = int_to_term ((term_to_int y) + 1) in
   if not(x = "SUC") then fail else
   SYM_RULE (num_CONV y_inc))
   ? failwith `inv_num_CONV`;;

%-----------------------------------------------------------------
Using MK_Micro_Int_Inst_LEMMA, we can prove a lemma of the form

 |- Micro_Int
    rep
    (\t. (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t))
    (\t. (int_e t)) ==>
    (!t.
      (mpc t = F,F,T,F,T,T) ==>
      (reg(t + 1),psw(t + 1),pc(t + 1),mem(t + 1),ivec(t + 1),ir(t + 1),
       mar(t + 1),mbr(t + 1),mpc(t + 1) =
       ST_u1
       rep
       (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,F,F,T,F,T,T)
       (int_e t)))
```

```
for every microinstruction, by simply giving its position in the
list.  Mapping the inference rule onto a list of integers from 0
to 63 yields a list of lemmas for each micro instruction.  The
entire process (exclusive of autoloading time) takes < 700 sec.
---------------------------------------------------------------%

let Micro_Int_SPEC =
    PURE_ONCE_REWRITE_RULE [micro_inst_list;GetMPC] (
    BETA_RULE (
    SPECL ["rep:^rep_ty";
           "(\t. (reg t,psw t,pc t,mem t,
                  ivec t,ir t,mar t,mbr t,mpc t)):time->^micro_state";
           "(\t. (int_e t)):time->^micro_env"] Micro_Int));;


let MK_Micro_Int_Inst_LEMMA inst =
    let tp = mk_n_tuple_from_int 6 inst in
    let mpc_term = "mpc t = ^tp" in
    DISCH_ALL (
    GEN "t" (
    DISCH mpc_term (
    SUBS [SPECL ["rep:^rep_ty";
                 "reg t:(*wordn)list";
                 "mem t:*memory";
                 "psw t:*wordn";
                 "pc t:*wordn";
                 "ivec t:*wordn";
                 "ir t:*wordn";
                 "mar t:*wordn";
                 "mbr t:*wordn";
                 tp;
                 "int_e t:bool"] (el (inst+1) instructions)] (
    CONV_RULE (DEPTH_CONV SND_CONV) (
    CONV_RULE (ONCE_DEPTH_CONV EL_CONV) (
    SUBS [bt6_val_CONV "bt6_val ^tp"] (
    SUBS [ASSUME mpc_term] (
    SPEC_ALL (
    SUBS [Micro_Int_SPEC] (
    ASSUME
      "Micro_Int (rep:^rep_ty)
         (\t. (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t))
         (\t. (int_e t))"))))))))));;

let mk_num_list n =
   letrec mk_num_list_aux n m =
      if n = m then [m] else
      (n . (mk_num_list_aux (n+1) m)) in
   mk_num_list_aux 0 n;;

let Micro_Int_Inst_list = map MK_Micro_Int_Inst_LEMMA (mk_num_list 63);;

%---------------------------------------------------------------
 Normalize top assumption (get rid of add_bt6)
---------------------------------------------------------------%
```

```
let NORMAL_POP_ASSUM_TAC =
    POP_ASSUM (\thm. ASSUME_TAC (
       CONV_RULE (ONCE_DEPTH_CONV bt6_ival_CONV) (
       CONV_RULE DEC_ADD_CONV (
       % DEC_ADD_CONV broken for "0 + 1" %
       PURE_ONCE_REWRITE_RULE [ADD_CLAUSES] (
       CONV_RULE (ONCE_DEPTH_CONV bt6_val_CONV) (
       REWRITE_RULE [add_bt6] thm))))));;

%----------------------------------------------------------------
 A few interesting lemmas
----------------------------------------------------------------%
let T_PLUS_3_LEMMA = TAC_PROOF
   (([],"! t . t + 3 = ((t + 1) + 1) + 1"),
    GEN_TAC
    THEN REPEAT (
        PURE_ONCE_REWRITE_TAC [SYM_RULE ADD_ASSOC]
        THEN DEC_ADD_TAC)
    THEN REFL_TAC
   );;

let RANGE_LEMMA = TAC_PROOF
   (([],
    "!t1 t2 (mpc:time->bt6) x .
     (!t'. t1 < t' /\  t' < t2 ==> ~(mpc t' = x)) /\
      ~(mpc t2 = x) ==>
     (!t'. t1 < t' /\ t' < (t2 + 1) ==> ~(mpc t' = x))"),
    REPEAT STRIP_TAC
    THEN ASSUM_LIST (\asl. ASSUME_TAC (
       SPEC "t':time" (el 5 asl)))
    THEN ASSUM_LIST (\asl. STRIP_ASSUME_TAC (
       REWRITE_RULE [SYM_RULE ADD1;LESS_THM] (el 3 asl)))
    THENL [
        ASSUM_LIST (\asl. ASSUME_TAC (
           REWRITE_RULE [el 1 asl] (el 3 asl)))
    ;
        ALL_TAC
    ]
    THEN RES_TAC
   );;

let LESS_SQUEEZE_LEMMA =
    let LESS_EQ_SUC =
          SYM_RULE (
          PURE_ONCE_REWRITE_RULE [DISJ_SYM] LESS_THM) in
    PURE_ONCE_REWRITE_RULE [ADD1] (
    PURE_ONCE_REWRITE_RULE [LESS_EQ_SUC] (
    PURE_ONCE_REWRITE_RULE [LESS_OR_EQ] LESS_EQ_ANTISYM));;

%----------------------------------------------------------------
 Lemma about FETCH-ISSUE-DECODE sequence.
----------------------------------------------------------------%
let FID_LEMMA = TAC_PROOF
   (([],
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
```

```
   (psw pc ivec ir mar mbr :time->*wordn) (mpc:time->bt6)
   (int_e:time->bool).
  Micro_Int rep (\t. (reg t,psw t,pc t,mem t,ivec t,
                      ir t,mar t,mbr t,mpc t))
                (\t. (int_e t)) ==>
  !t. (int_e t /\ get_ie rep (psw t) = F) /\
      (mpc t = (F,F,F,F,F,F)) ==>
      ((reg(t + 3),psw(t + 3),pc(t + 3),mem(t + 3),ivec(t + 3),
        ir(t + 3),mar(t + 3),mbr(t + 3),mpc(t + 3)) =
      (reg t,psw t,inc rep(pc t),mem t,ivec t,
       fetch rep(mem t,address rep(pc t)),pc t,
       fetch rep(mem t,address rep(pc t)),
       add_bt6 (F,SND(opcode rep
                      (fetch rep
                       (mem t,address rep(pc t))))) ^OFFSET)) /\
  ~(mpc(t + 1) = F,F,F,F,F,F) /\
  ~(mpc((t + 1) + 1) = F,F,F,F,F,F) /\
  ~(mpc(((t + 1) + 1) + 1) = F,F,F,F,F,F)"),
 REPEAT GEN_TAC
 THEN STRIP_TAC
 THEN GEN_TAC
 THEN STRIP_TAC
 THEN IMP_RES_TAC (el 1 Micro_Int_Inst_list)
 THEN ASSUM_LIST (\asl. MAP_EVERY ASSUME_TAC (
    CONJUNCTS(REWRITE_RULE [(el 4 asl); PAIR_EQ] (el 1 asl))))
 THEN NORMAL_POP_ASSUM_TAC
 THEN ASSUM_LIST (\asl. MAP_EVERY ASSUME_TAC (
    CONJUNCTS (
    REWRITE_RULE [PAIR_EQ] (
    (\y. MATCH_MP y (el 1 asl)) (
    SPEC "t+1:time" (
    MATCH_MP (el 2 Micro_Int_Inst_list) (last asl)
    ))))))
 THEN NORMAL_POP_ASSUM_TAC
 THEN ASSUM_LIST (\asl. MAP_EVERY ASSUME_TAC (
    CONJUNCTS (
    REWRITE_RULE [PAIR_EQ] (
    (\y. MATCH_MP y (el 1 asl)) (
    SPEC "(t+1)+1:time" (
    MATCH_MP (el 3 Micro_Int_Inst_list) (last asl)
    ))))))
 THEN ASM_REWRITE_TAC [T_PLUS_3_LEMMA;PAIR_EQ;
                       OFFSET_NOT_BEGINNING]
);;

let Macro_Inst_Correct_LEMMA =
    BETA_RULE (
    REWRITE_RULE [Opcode;Opc_Val; GetMPC; Micro_Substate;Next] (
    BETA_RULE (
    SPECL ["rep:^rep_ty";
           "(\t. reg t, psw t, pc t, mem t, ivec t,
                 ir t, mar t, mbr t, mpc t):time->^micro_state";
           "(\t. int_e t):time->^micro_env"]
          Macro_Inst_Correct)));;
```

```
let EXPAND_MACRO_INST_RULE x =
    PURE_REWRITE_RULE [GetDest; GetImm; GetSrcA; GetSrcB] (
    EXPAND_LET_RULE x);;

%----------------------------------------------------------------
 Performs repeated symbolic execution on the suumption list
 until the MPC has returned the FETCH_ADDR.  Keeps track of
 the number of iterations and supplies the number as a witness
 for the existential quantification.
----------------------------------------------------------------%
let (INST_LOOP_TAC tm_init):tactic =
    let is_begin thm =
       snd(dest_eq thm) = FETCH_ADDR in
    let tuple_val thm =
       term_to_int(bt_val_func(snd(dest_eq  thm))) in
    letrec INST_LOOP_TAC_AUX tm ((asl,w):goal) =
        let INST_TAC n =
               IMP_RES_TAC (el n Micro_Int_Inst_list) THEN
               ASSUM_LIST (\x. MAP_EVERY ASSUME_TAC (
                   CONJUNCTS (
                   REWRITE_RULE [PAIR_EQ] (el 1 x)))) in
        let n = (tuple_val (el 1 asl)) + 1 in
        let gl,p = INST_TAC n (asl,w) in
        let (asl',w') = (hd gl) in
        let gll,pl = split (
        if (is_begin (el 1 asl')) then
           map (EXISTS_TAC tm) gl else
           map (INST_LOOP_TAC_AUX "(^tm)+1") gl) in
        (flat gll,(p o mapshape(map length gll)pl)) in
    INST_LOOP_TAC_AUX "(^tm_init + 1)";;

%----------------------------------------------------------------
 Create a goal for instruction n
----------------------------------------------------------------%
let MK_INST_CORRECT_GOAL n =
    let inst = term_list_el n
                 (snd(dest_eq(
                  snd(dest_forall(concl macro_inst_list))))) in
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr :time->*wordn) (mpc:time->bt6)
      (int_e:time->bool).
      (! m . int_fetch rep (int_trans rep m) = (int_fetch rep m)) /\
      (! m a . fetch rep (trans rep m,a) = fetch rep (m,a)) /\
      (! m a x . store rep (trans rep m,a,x) =
                 trans rep (store rep (m,a,x))) ==>
      Macro_Inst_Correct rep
         (\t. reg t, psw t, pc t, mem t, ivec t,
              ir t, mar t, mbr t, mpc t)
         (\t. int_e t) ^inst";;

%----------------------------------------------------------------
 Prove the instruction correctness lemma for instruction n
----------------------------------------------------------------%
let INST_CORRECT_TAC (n,thm) =
    let inst_lemma = EXPAND_MACRO_INST_RULE thm in
```

```
    let inst = term_list_el n
                 (snd(dest_eq(
                  snd(dest_forall(concl macro_inst_list))))) in
    REPEAT STRIP_TAC
    THEN SUBST_TAC [SPEC inst Macro_Inst_Correct_LEMMA]
    THEN ASM_REWRITE_TAC [inst_lemma;ABS_ENV]
    THEN REPEAT STRIP_TAC
    THEN IMP_RES_TAC INJ_LEMMA_ONE
    THEN IMP_RES_TAC FID_LEMMA
    THEN RES_TAC
    THEN ASSUM_LIST (\asl. MAP_EVERY ASSUME_TAC (
        CONJUNCTS (
        REWRITE_RULE [el 10 asl;PAIR_EQ] (el 7 asl)
        )))
    THEN NORMAL_POP_ASSUM_TAC
    THEN INST_LOOP_TAC "3"
    THEN CONV_TAC (TOP_DEPTH_CONV ADD_ASSOC_CONV)
    THEN BETA_TAC
    THEN ASM_REWRITE_TAC [PAIR_EQ]
    THEN REPEAT CONJ_TAC
    THENL [ % 1 %
        PURE_ONCE_REWRITE_TAC [SYM_RULE ADD1]
        THEN CONV_TAC (TOP_DEPTH_CONV INV_ADD_ASSOC_CONV)
        THEN REWRITE_TAC [
            REWRITE_RULE [ADD_CLAUSES;NOT_SUC] (
            GEN_ALL (SPECL ["m:num";"SUC n"] LESS_ADD_NONZERO))]
    ; % 2 %
        PURE_ONCE_REWRITE_TAC [T_PLUS_3_LEMMA]
        THEN REPEAT (
            ((MATCH_MP_TAC RANGE_LEMMA) ORELSE ALL_TAC)
            THEN CONJ_TAC
            THEN ONCE_REWRITE_TAC [LESS_SQUEEZE_LEMMA])
        THEN (SUBST_TAC [SYM_RULE (SPEC_ALL T_PLUS_3_LEMMA)]
              ORELSE ALL_TAC)
        THEN ASM_REWRITE_TAC [PAIR_EQ]
    ];;

map (delete_cache o fst) (cached_theories());;

%----------------------------------------------------------------
 Prove EINT instruction correctness lemma (special case)
----------------------------------------------------------------%
let EINT_inst = definition `macro_def` `EINT`;;

let EINT_CORRECT_LEMMA = (TAC_PROOF
   (([], MK_INST_CORRECT_GOAL 32),
    REPEAT GEN_TAC
    THEN SUBST_TAC [
          SPEC "(INR one:bt5+one,ABS_ENV(EINT (rep:^rep_ty)))"
               Macro_Inst_Correct_LEMMA]
    THEN ASM_REWRITE_TAC [ABS_ENV;
                 EXPAND_MACRO_INST_RULE EINT_inst]
    THEN REPEAT STRIP_TAC
    THEN IMP_RES_TAC INJ_LEMMA_TWO
    THEN IMP_RES_TAC (el 1 Micro_Int_Inst_list)
```

```
    THEN ASSUM_LIST (\asl. MAP_EVERY ASSUME_TAC (
       CONJUNCTS(REWRITE_RULE [(el 6 asl); PAIR_EQ] (el 1 asl))))
    THEN NORMAL_POP_ASSUM_TAC
    THEN INST_LOOP_TAC "1"
    THEN CONV_TAC (TOP_DEPTH_CONV ADD_ASSOC_CONV)
    THEN BETA_TAC
    THEN ASM_REWRITE_TAC [PAIR_EQ]
    THEN REPEAT CONJ_TAC
    THENL [ % 1 %
        PURE_ONCE_REWRITE_TAC [SYM_RULE ADD1]
        THEN CONV_TAC (TOP_DEPTH_CONV INV_ADD_ASSOC_CONV)
        THEN REWRITE_TAC [
            REWRITE_RULE [ADD_CLAUSES;NOT_SUC] (
            GEN_ALL (SPECL ["m:num";"SUC n"] LESS_ADD_NONZERO))]
    ; % 2 %
        REPEAT (
            ((MATCH_MP_TAC RANGE_LEMMA) ORELSE ALL_TAC)
            THEN CONJ_TAC
            THEN ONCE_REWRITE_TAC [LESS_SQUEEZE_LEMMA])
        THEN ASM_REWRITE_TAC [PAIR_EQ]
    ]) ? BOOL_CASES_AX
   );;

save_thm(`EINT_CORRECT_LEMMA`,EINT_CORRECT_LEMMA);;


%----------------------------------------------------------------
 If PROVE_INST_CORRECT_LEMMA fails, I don't want it to stop the
 make, so we'll return a dummy theorem.
----------------------------------------------------------------%

let PROVE_INST_CORRECT_LEMMA n = (
   TAC_PROOF (([], MK_INST_CORRECT_GOAL n),
              INST_CORRECT_TAC (n,el (n+1) macro_defn_list)))
   ? BOOL_CASES_AX;;


%----------------------------------------------------------------
 Save lemmas for recovery in the event of a crash.
----------------------------------------------------------------%
let SAVE_INST_LEMMA n =
   let name = (concat `MAC_INST_` (string_of_int n)) in
   save_thm(name,PROVE_INST_CORRECT_LEMMA n);;


map (delete_cache o fst) (cached_theories());;

letrec mk_num_list n m =
      if n = m then [m] else
      (n . (mk_num_list (n+1) m));;

let inst_lemma_list = map SAVE_INST_LEMMA (mk_num_list 0 7);;

map (delete_cache o fst) (cached_theories());;
```

```
let inst_lemma_list =
     inst_lemma_list @
     (map SAVE_INST_LEMMA (mk_num_list 8 15));;

map (delete_cache o fst) (cached_theories());;

let inst_lemma_list =
     inst_lemma_list @
     (map SAVE_INST_LEMMA (mk_num_list 16 23));;

map (delete_cache o fst) (cached_theories());;

let inst_lemma_list =
     inst_lemma_list @
     (map SAVE_INST_LEMMA (mk_num_list 24 31));;


let inst_lemma_list =
     inst_lemma_list @
     [EINT_CORRECT_LEMMA];;

%----------------------------------------------------------------
 The first obligation of the abstract interpreter theory
----------------------------------------------------------------%
let Macro_Int_CORRECT_LEMMA_AUX = TAC_PROOF
   (([],
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr :time->*wordn) (mpc:time->bt6)
      (int_e:time->bool).
      (! m . int_fetch rep (int_trans rep m) = (int_fetch rep m)) /\
      (! m a . fetch rep (trans rep m,a) = fetch rep (m,a)) /\
      (! m a x . store rep (trans rep m,a,x) =
                 trans rep (store rep (m,a,x))) ==>
      EVERY (Macro_Inst_Correct rep
              (\t. reg t, psw t, pc t, mem t, ivec t,
                   ir t, mar t, mbr t, mpc t)
              (\t. int_e t)) (macro_inst_list rep)"),
    REWRITE_TAC [EVERY_DEF;macro_inst_list]
    THEN REPEAT STRIP_TAC
    THEN POP_ASSUM_LIST (\asl. MP_TAC (LIST_CONJ (rev asl)))
    THENL
          (map MATCH_ACCEPT_TAC inst_lemma_list)
   );;

let Macro_Int_CORRECT_LEMMA = (
    UNDISCH_ALL (
    SPEC_ALL (
    PURE_ONCE_REWRITE_RULE [Macro_Inst_Correct_def]
        Macro_Int_CORRECT_LEMMA_AUX)));;

%----------------------------------------------------------------
 The second obligation of the abstract interpreter theory
----------------------------------------------------------------%
let Macro_Int_LENGTH_LEMMA = TAC_PROOF
   (([],
```

```
    "! opc. Opc_Val opc < (LENGTH (macro_inst_list (rep:^rep_ty)))"),
    REPEAT GEN_TAC
    THEN REWRITE_TAC [macro_inst_list;LENGTH;Opc_Val]
    THEN COND_CASES_TAC
    THENL [
        STRUCT_CASES_TAC (SPEC "(OUTL (opc:bt5+one))" FIVE_TUPLE_VALUE_LEMMA)
        THEN REWRITE_TAC [bt5_val;SYM_RULE ADD1;OUTL]
    ;
        ALL_TAC
    ]
    THEN CONV_TAC (TOP_DEPTH_CONV num_CONV)
    THEN REWRITE_TAC [LESS_0;LESS_MONO_EQ]
   );;


letrec DEPTH_FIRST_CONV conv tm =
    FIRST_CONV
      [conv;                               % try it here               %
       RATOR_CONV (DEPTH_FIRST_CONV conv);  % or else try left subtree  %
       RAND_CONV  (DEPTH_FIRST_CONV conv);  % or else try right subtree %
       ABS_CONV   (DEPTH_FIRST_CONV conv)]  % or go through a lambda    %
    tm;;

let ONCE_LEFT_REWRITE_TAC =
    GEN_REWRITE_TAC DEPTH_FIRST_CONV basic_rewrites;;

let NOT_ISL_LEMMA = TAC_PROOF
    (([],
     "!opc:bt5+one . ~(ISL opc) ==> (ISR opc)"),
     REPEAT STRIP_TAC
     THEN STRUCT_CASES_TAC (SPEC "opc:bt5+one"
            (INST_TYPE [(":bt5",":*");(":one",":**")] ISL_OR_ISR))
     THENL [
         RES_TAC
     ;
         POP_ASSUM (\thm. ACCEPT_TAC thm)
     ]
    );;


let NOT_ISR_LEMMA = TAC_PROOF
    (([],
     "!opc:bt5+one . ~(ISR opc) ==> (ISL opc)"),
     REPEAT STRIP_TAC
     THEN STRUCT_CASES_TAC (SPEC "opc:bt5+one"
            (INST_TYPE [(":bt5",":*");(":one",":**")] ISL_OR_ISR))
     THENL [
         POP_ASSUM (\thm. ACCEPT_TAC thm)
     ;
         RES_TAC
     ]
    );;
```

```
%---------------------------------------------------------------
 The third obligation of the abstract interpreter theory
---------------------------------------------------------------%
let Macro_Int_ORDER_LEMMA = TAC_PROOF
   (([],
    "!opc:bt5+one . opc = (FST (EL (Opc_Val opc)
                                   (macro_inst_list (rep:^rep_ty))))"),
    REPEAT GEN_TAC
    THEN REWRITE_TAC [Opc_Val;macro_inst_list]
    THEN COND_CASES_TAC
    THENL [
        POP_ASSUM (\thm. ONCE_LEFT_REWRITE_TAC [
           (SYM_RULE
           (MP (SPEC "opc:bt5+one"
                (INST_TYPE [(":bt5",":*");(":one",":**")] INL))
               (REWRITE_RULE [] thm)))])
        THEN STRUCT_CASES_TAC (SPEC "(OUTL (opc:bt5+one))"
                                   FIVE_TUPLE_VALUE_LEMMA)
        THEN REWRITE_TAC [bt5_val;OUTL]
    ;
        POP_ASSUM (\thm. ONCE_LEFT_REWRITE_TAC [
           (SYM_RULE
           (MP (SPEC "opc:bt5+one"
                (INST_TYPE [(":bt5",":*");(":one",":**")] INR))
               (REWRITE_RULE [thm] (SPEC_ALL NOT_ISL_LEMMA))
               ))])
        THEN SUBST_TAC [SPEC "(OUTR (opc:bt5+one))" one]
        THEN REWRITE_TAC [OUTR]
    ]
    THEN CONV_TAC (ONCE_DEPTH_CONV EL_CONV)
    THEN REWRITE_TAC []
   );;

let theorem_list =
    instantiate_abstract_theorems
        `gen_I`
        [Macro_Int_CORRECT_LEMMA;
         Macro_Int_LENGTH_LEMMA;
         Macro_Int_ORDER_LEMMA]
        [
         ("rep:^I_rep_ty",
          "(macro_inst_list (rep:^rep_ty),
            Opc_Val,
            Opcode rep,
            Micro_Substate rep,
            (I:^micro_env->^macro_env),
            Micro_Int rep,
            GetMPC:^micro_state->^micro_env->bt6, ^FETCH_ADDR,@x:one.F)");
         ("e':time'->*env'",
          "(\t:time. (int_e t):bool)");
         ("s':time->*state'",
          "(\t:time. (reg t):(*wordn)list, (psw t):*wordn,
                     (pc t):*wordn, (mem t):*memory, (ivec t):*wordn,
                     (ir t):*wordn, (mar t):*wordn, (mbr t):*wordn,
                     (mpc t):bt6)")
```

```
       ]
      `MACRO`;;

let correct_lemma = snd(hd theorem_list);;

%-----------------------------------------------------------------
MACRO_LEVEL_CORRECT_LEMMA =
|- (!m. int_fetch rep(int_trans rep m) = int_fetch rep m) /\
   (!m a. fetch rep(trans rep m,a) = fetch rep(m,a)) /\
   (!m a x. store rep(trans rep m,a,x) = trans rep(store rep(m,a,x))) ==>
   Micro_Int
   rep
   (\t. (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t))
   (\t. (int_e t)) /\
   (?t. mpc t = F,F,F,F,F,F) ==>
   Macro_Int
   rep
   ((\t. (reg t,psw t,pc t,trans rep(mem t),int_trans rep(ivec t))) o
    (Temp_Abs(\t. mpc t = F,F,F,F,F,F)))
   ((\t. (int_e t)) o (Temp_Abs(\t. mpc t = F,F,F,F,F,F)))
Run time: 254.3s
Intermediate theorems generated: 4257
-----------------------------------------------------------------%

let MACRO_LEVEL_CORRECT_LEMMA = save_thm
   (`MACRO_LEVEL_CORRECT_LEMMA`,
    BETA_RULE (
    EXPAND_LET_RULE (
    ONCE_REWRITE_RULE [Micro_Substate;I_THM;GetMPC] (
    BETA_RULE (
    ONCE_REWRITE_RULE [SYM_RULE Macro_Int_def] correct_lemma))))
   );;
```

3.8 The Final Result

The section presents the ML code that creates the theory `avm.th`.

```
%----------------------------------------------------------------

   File:         mk_avm.ml

   Author:       (c) P. J. Windley 1990

   Date:         JUN 23, 1990

   Modified:

   Description:

   Uses the correctness proofs from each level to prove an overall
   correctness result for AVM-1

----------------------------------------------------------------%

set_search_path (search_path() @ [`/muztag/home/windley/hol/tactics/`;
                                  `/muztag/home/windley/hol/ml/`;
                                 ]);;

let Library_Root = `/muztag/home/windley/hol/Library/`;;

set_search_path
    (search_path() @
        (map (concat Library_Root)
             [`tuple/`;`decimal/`]));;

loadf `abstract`;;

system `/bin/rm avm.th`;;

new_theory `avm`;;

new_parent `macro`;;

let MACRO_LEVEL_CORRECT_LEMMA =
    theorem `macro` `MACRO_LEVEL_CORRECT_LEMMA`;;

let MICRO_LEVEL_CORRECT_LEMMA =
    theorem `micro` `MICRO_LEVEL_CORRECT_LEMMA`;;

let PHASE_LEVEL_CORRECT_LEMMA =
    REWRITE_RULE [I_o_ID] (
    theorem `phase` `PHASE_LEVEL_CORRECT_LEMMA`);;

let Micro_Int = theorem `micro` `Micro_Int`;;

%----------------------------------------------------------------
 Load abstract type definitions.
```

```
----------------------------------------------------------------%
let rep_ty = abstract_type `aux_def` `opcode`;;

let I_rep_ty = abstract_type `gen_I` `Impl`;;

%----------------------------------------------------------------
 Define type terms for the state and env.
----------------------------------------------------------------%
let macro_state = ":((*wordn)list#*wordn#*wordn#*memory#*wordn)";;

let macro_env = ":bool";;

let micro_state = ":((*wordn)list#*wordn#*wordn#*memory#
                     *wordn#*wordn#*wordn#*wordn#bt6)";;

let micro_env = ":bool";;

let Phase_state =
    ":((*wordn)list#*wordn#*wordn#*memory#
       *wordn#*wordn#*wordn#*wordn#bt6#
       *wordn#*wordn#bool#bool#ucode#(num->ucode)#bt2)";;

let Phase_env = ":bool";;

let EBM_state = Phase_state;;

let EBM_env = Phase_env;;

%----------------------------------------------------------------
Note that micro_rom is substituted for urom.  The general version
doesn't imply the higher levels, only the EBM coupled with the
microcode does.
----------------------------------------------------------------%

let EBM_MICRO_CORRECT_LEMMA = prove_thm
   (`EBM_MICRO_CORRECT_LEMMA`,
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2)
      (mir:time->ucode)
      (ireq_ff iack_ff ireq_e:time->bool).
     let f = (Temp_Abs(\t. clk t = F,F)) in (
     (!p. mk_psw rep (get_sm rep p,get_ie rep p,
                      get_vf rep p,get_nf rep p,
                      get_cf rep p,get_zf rep p) = p) ==>
     EBM rep
        (\t. (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,
              mbr t,mpc t,alatch t,blatch t,ireq_ff t,
              iack_ff t,mir t,micro_rom,clk t))
        (\t. (ireq_e t t)) /\
     (?t. clk t = F,F) ==>
     Micro_Int rep
       ((\t. (reg t,psw t,pc t,mem t,
              ivec t,ir t,mar t,mbr t,mpc t)) o f)
       ((\t. (ireq_e t t)) o f))",
```

```
    EXPAND_LET_TAC
    THEN REPEAT (
       STRIP_GOAL_THEN (\thm. (MAP_EVERY CHECK_ASSUME_TAC (CONJUNCTS thm))))
    THEN IMP_RES_TAC PHASE_LEVEL_CORRECT_LEMMA
    THEN IMP_RES_TAC MICRO_LEVEL_CORRECT_LEMMA
   );;

let new_o_THM =
    GEN_ALL (
    SPECL ["f:time->***";
           "Temp_Abs(\t. clk t = F,F):time->time";
           "x:time"] (
    INST_TYPE [(":time",":*");
               (":time",":**")] o_THM));;

let new_o_DEF =
    GEN_ALL (
    SPECL ["f:time->***";
           "Temp_Abs(\t. clk t = F,F):time->time"] (
    INST_TYPE [(":time",":*");
               (":time",":**")] o_DEF));;


let EBM_MICRO_CORRECT_LEMMA_EXPANDED =
    ONCE_REWRITE_RULE [SYM_RULE new_o_THM] (
    BETA_RULE (
    REWRITE_RULE [o_DEF] (
    EXPAND_LET_RULE EBM_MICRO_CORRECT_LEMMA)));;

%-----------------------------------------------------------------
EBM_MICRO_CORRECT_LEMMA_EXPANDED =
|- !rep reg mem psw pc ivec ir mar mbr alatch blatch mpc clk mir
    ireq_ff iack_ff ireq_e.
    (!p.
      mk_psw
      rep
      (get_sm rep p,get_ie rep p,get_vf rep p,get_nf rep p,get_cf rep p,
       get_zf rep p) =
      p) ==>
    EBM
    rep
    (\t.
      (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t,alatch t,
       blatch t,ireq_ff t,iack_ff t,mir t,micro_rom,clk t))
    (\t. (ireq_e t t)) /\
    (?t. clk t = F,F) ==>
    Micro_Int
    rep
    (\x.
      ((reg o (Temp_Abs(\t. clk t = F,F)))x,
       (psw o (Temp_Abs(\t. clk t = F,F)))x,
       (pc o (Temp_Abs(\t. clk t = F,F)))x,
       (mem o (Temp_Abs(\t. clk t = F,F)))x,
       (ivec o (Temp_Abs(\t. clk t = F,F)))x,
       (ir o (Temp_Abs(\t. clk t = F,F)))x,
```

```
       (mar o (Temp_Abs(\t. clk t = F,F)))x,
       (mbr o (Temp_Abs(\t. clk t = F,F)))x,
       (mpc o (Temp_Abs(\t. clk t = F,F)))x))
    (\x.
      ((ireq_e o (Temp_Abs(\t. clk t = F,F)))x))
Run time: 142.2s
Intermediate theorems generated: 4272
-----------------------------------------------------------------%

let AVM_CORRECT = prove_thm
   (`AVM_CORRECT`,
    "!(rep:^rep_ty) (reg:time->(*wordn)list) (mem:time->*memory)
      (psw pc ivec ir mar mbr alatch blatch:time->*wordn)
      (mpc:time->bt6) (clk:time->bt2)
      (mir:time->ucode)
      (ireq_ff iack_ff ireq_e:time->bool).
     let micro_abs = (Temp_Abs(\t. clk t = F,F)) in
     let abs = micro_abs o
               (Temp_Abs(\t. (mpc o micro_abs) t = F,F,F,F,F,F)) in (
     (!m. int_fetch rep(int_trans rep m) = int_fetch rep m) /\
     (!m a. fetch rep(trans rep m,a) = fetch rep(m,a)) /\
     (!m a x. store rep(trans rep m,a,x) = trans rep(store rep(m,a,x))) ==>
     (!p. mk_psw rep (get_sm rep p,get_ie rep p,
                      get_vf rep p,get_nf rep p,
                      get_cf rep p,get_zf rep p) = p) ==>
     EBM rep
        (\t. (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,
              mbr t,mpc t,alatch t,blatch t,ireq_ff t,
              iack_ff t,mir t,micro_rom,clk t))
        (\t. (ireq_e t t)) /\
     (?t. (clk t = F,F)) /\
     (?t. ((mpc o micro_abs) t = F,F,F,F,F,F)) ==>
     Macro_Int rep
        ((\t. (reg t,psw t,pc t,
               trans rep(mem t),int_trans rep(ivec t))) o abs)
        ((\t. (ireq_e t t)) o abs))",
    EXPAND_LET_TAC
    THEN REPEAT (
       STRIP_GOAL_THEN (\thm. (MAP_EVERY CHECK_ASSUME_TAC (CONJUNCTS thm))))
    THEN IMP_RES_TAC EBM_MICRO_CORRECT_LEMMA_EXPANDED
    THEN IMP_RES_TAC MACRO_LEVEL_CORRECT_LEMMA
    THEN ONCE_REWRITE_TAC [o_ASSOC]
    THEN ONCE_REWRITE_TAC [new_o_DEF]
    THEN BETA_TAC
    THEN ONCE_REWRITE_TAC [SYM_RULE new_o_THM]
    THEN POP_ASSUM (\thm . MATCH_ACCEPT_TAC thm)
);;

%-----------------------------------------------------------------
AVM_CORRECT =
|- !rep reg mem psw pc ivec ir mar mbr alatch blatch mpc clk mir
    ireq_ff iack_ff ireq_e.
    let micro_abs = Temp_Abs(\t. clk t = F,F)
    in
     let abs =
```

```
        micro_abs o (Temp_Abs(\t. (mpc o micro_abs)t = F,F,F,F,F,F))
  in
   ((!m. int_fetch rep(int_trans rep m) = int_fetch rep m) /\
    (!m a. fetch rep(trans rep m,a) = fetch rep(m,a)) /\
    (!m a x.
      store rep(trans rep m,a,x) = trans rep(store rep(m,a,x))) ==>
    (!p.
      mk_psw
      rep
      (get_sm rep p,get_ie rep p,get_vf rep p,get_nf rep p,
       get_cf rep p,get_zf rep p) =
      p) ==>
    EBM
    rep
    (\t.
      (reg t,psw t,pc t,mem t,ivec t,ir t,mar t,mbr t,mpc t,alatch t,
       blatch t,ireq_ff t,iack_ff t,mir t,micro_rom,clk t))
    (\t. (ireq_e t t)) /\
    (?t. clk t = F,F) /\
    (?t. (mpc o micro_abs)t = F,F,F,F,F,F) ==>
    Macro_Int
    rep
    ((\t. (reg t,psw t,pc t,trans rep(mem t),int_trans rep(ivec t))) o
     abs)
    ((\t. (ireq_e t t)) o abs))
Run time: 238.1s
Intermediate theorems generated: 3280
----------------------------------------------------------------%
```

References

[Adv83] Advanced Micro Devices. *Bipolar Microprocessor Logic and Interface Data Book*. AMD Inc., 1983.

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

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

[Kat85] Manolis G. H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI*. MIT Press, 1985.

[Win90a] Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, Division of Computer Science, June 1990.

[Win90b] Phillip J. Windley. A hierarchical methodology for the verification of microprogrammed microprocessors. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1990.