NASA Contractor Report 187547

Toward a Formal Verification of a Floating-Point Coprocessor and its Composition with a Central Processing Unit

Jing Pan
K. Levitt
University of California
Davis, California

G. C. Cohen
Boeing Advanced Systems
Seattle, Washington

NASA Contract NAS1-18586
August 1991

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665

PREFACE

Current hardware verification efforts have only begun to address the problem of composing asynchronously communicating units. This report presents work underway to formally specify and verify a floating-point coprocessor based on the MC68881. Our work uses the HOL verification system developed at Cambridge University. The coprocessor consists of two independent units: the bus interface unit to communicate with the CPU and the arithmetic processing unit to perform the actual calculation. We illustrate how the specification and verification process can be organized and simplified by a generalized hierarchical decomposition methodology that supports reasoning about horizontal interaction between processes. Techniques of composing processes having independent time scales are formalized. Reasoning about the interaction and synchronization among processes using higher-order logic is demonstrated.

The CPU instructions and the floating-point instructions are allowed to execute concurrently to improve performance. However, the coprocessor interface is designed to maintain a strictly sequential execution model to reflect the assembly language programmer's view of the system. We discuss the combination of the CPU and the coprocessor to form a computer system, and explore techniques to map from the underlying concurrent implementation to the programmer's view of sequential execution of instructions.

The NASA technical monitor for this work is Sally C. Johnson of NASA Langley Research Center, Hampton, Virginia.

The work was done at Boeing Military Airplanes, Seattle, Washington and the University of California, Davis, California. Personnel responsible for this work include:

Boeing Military Airplanes:
D. Gangsaas, responsible manager
T. M. Richardson, program manager
G. C. Cohen, principal investigator

University of California:
Dr. K. N. Levitt, chief researcher
Jing Pan, PhD candidate

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

1.0 INTRODUCTION

Formal hardware verification involves using mechanized theorem-proving techniques to verify that the design of a system satisfies its specification. Because exhaustive simulation is often too time consuming, and because simulation that is non-exhaustive might miss cases that are incorrect, there is increasing interest in using a formal approach to reason about hardware designs. This report describes work in progress on verifying a floating-point coprocessor based on the MC68881 ((ref. 1), (ref. 2)) [1]. The coprocessor consists of a bus interface unit (BIU) which communicates with the CPU, and an arithmetic processing unit (APU).

There has been significant interest in formal verification in recent years ((ref. 3), (ref. 4), (ref. 5), (ref. 6), (ref. 7), (ref. 8), (ref. 9), (ref. 10)). Formal proofs of complex systems are not trivial, requiring significant machine time and human effort. Perhaps the best known verification effort is that of the VIPER microprocessor ((ref. 3), (ref. 4)). VIPER is the first microprocessor intended for commercial distribution where a formal verification has been attempted.

Recent work ((ref. 11), (ref. 12)) has shown that the verification of microprocessors can be simplified through insertion of intermediate levels of abstraction between the instruction set and the electronic block model (EBM); the EBM is, generally, the lowest level in the hierarchy and, logically, represents the object being verified. Through these appropriate intermediate levels, long and complex proofs are replaced by many more simple proofs. Furthermore, each level is a self-contained abstraction that has meaning in the explanation of the system. That is, the overall approach of abstraction reflects the way complex microprocessors are designed. In (ref. 10), microprocessors with four levels of abstraction are considered. The macro level reflects the programmer's view of instruction execution. At the micro level, an instruction is interpreted by executing a sequence of microinstructions. The phase level description decomposes the interpretation of a single microinstruction into the parallel execution of a set of elementary operations. The lowest level is the electronic block model, where a number of blocks such as the registers, the ALU and the microROM are connected together. It is generally accepted that the correctness of the EBM can be established by simulation.

This report concentrates on two tasks: the first involves the formal composition of three interpreters (the CPU service, the BIU top level, and the APU top-level interpreter), the composition producing the floating-point coprocessor (FPC). The CPU service interpreter is that part of the CPU

[1] The coprocessor was designed based on the information contained in the MC68881 user's manual

specification concerned with communication with the BIU. The second task is concerned with the composition of the CPU and the FPC to form a more abstract view of a computer system, consisting of both the CPU and FPC. This composition would be verified with respect to a specification of a single abstract interpreter that provides both integer and floating-point instructions.

There are five sections to this report. Section two briefly describes the basic architecture of the floating-point coprocessor being verified, emphasizing the communication and synchronization functions; section three presents a methodology for verifying the composition of three interacting interpreters, the goal being the verification of the coprocessor; section four discusses our approach to the verification of the CPU-FPC combination.

2.0 THE FLOATING-POINT COPROCESSOR ARCHITECTURE

We now present the design of a floating-point coprocessor (FPC) that is a simplification of the MC68881. Our coprocessor is designed to interface with a CPU to provide a logical extension to the CPU's integer data processing capabilities. The assembly language programmer can view the FPC registers as though they are resident in the CPU. Thus, the CPU and FPC pair appears to the programmer to be one processor that supports both floating-point and integer operations. That the floating-point and integer operations are processed by different processors is an implementation detail to the programmer. Moreover, the coprocessor interface allows the FP instructions to execute concurrently with the CPU instructions to improve efficiency, but this, too, is an implementation detail.

2.1 THE HARDWARE OVERVIEW

The coprocessor is internally divided into two processing elements: the bus interface unit (BIU) and the arithmetic processing unit (APU) (see fig. 2.1-1). Though the BIU monitors the state of the APU, it operates independently of the APU. The APU operates on the opcode and operands that the BIU passes to it. In return, the APU reports its internal state to the BIU. The BIU contains the coprocessor interface registers (CIRs) and status flags. The CIR register select and acknowledgement control logic are also contained in the BIU. The coprocessor interface implements a protocol that controls the access of these registers by the CPU and the APU. Since the bus is asynchronous, the FPC need not run at the same clock speed as the CPU.

The APU executes all the floating-point instructions. The floating-point data, control and status registers are located inside the APU. In addition to these registers, the APU contains an arithmetic unit used for both mantissa and exponent calculations, and a barrel shifter.

2.2 THE COMMUNICATION PROTOCOL

As mentioned in the last section, the FPC contains a number of coprocessor interface registers (CIRs) which are addressed by the CPU in the same manner as memory is addressed. When the CPU fetches a coprocessor instruction, the CPU writes the instruction to the command CIR and reads the response CIR. In this response, the BIU encodes requests for any additional service required by the FPC. Data values read by the CPU from the FPC response CIR are referred to as

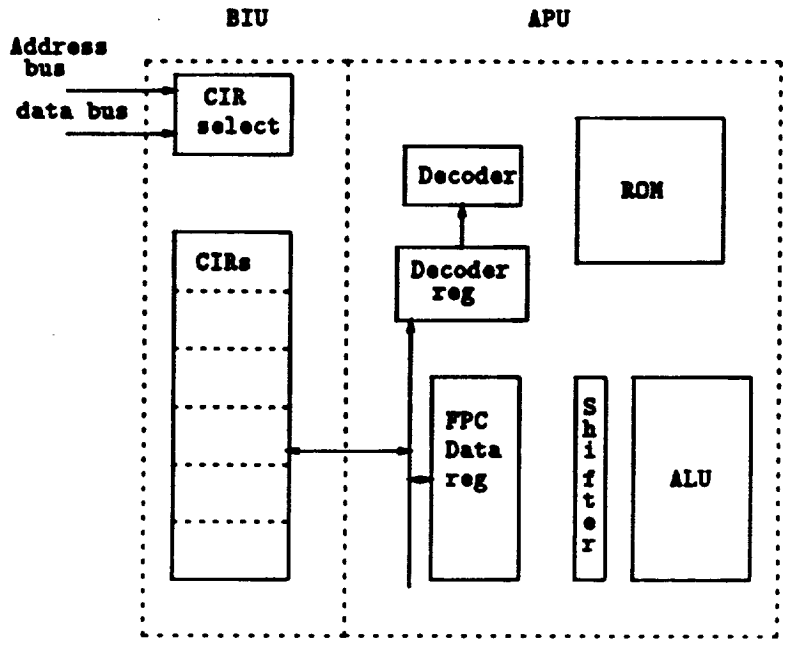


Figure 2.1-1: Simplified Block Diagram of the Floating-Point Coprocessor

"primitives". The response has one of the following meanings:

a. The FPC is busy. The CPU then checks for interrupts, processes them, and queries the FPC again. In this case, the CPU will not execute the next instruction in the program.

b. There is a FPC service request. For example, this request might be to evaluate the effective address and deliver data from the CPU data registers/memory to the FPC.

c. The CPU is not needed. Communication is terminated, and the CPU is free to execute the next instruction.

There are four CIRs that reside inside the BIU. The read-only/write-only designations apply to the CPU's access to these registers.

a. Response CIR. This read-only register is used to communicate service requests from the FPC to the CPU.

b. Command CIR. This write-only register is used by the CPU to initiate a dialog for a coprocessor instruction. When the FPC detects a write to this CIR, the data value is latched from

the data bus.

c. Condition CIR. This write-only register is used by the CPU to initiate the dialog for a conditional coprocessor instruction.

d. Operand CIR. This read/write register is used by the CPU to transfer data to and from the FPC.

Currently we do not have exception-handling capabilities built into the coprocessor. We plan to consider this feature in the future. Several additional CIRs will be needed to accommodate exception-handling capabilities.

2.3 THE INSTRUCTION SET

The FPC instructions can be separated into three groups:

a. Data movement instructions. The FLD and FSTR instructions are used to transfer data between the floating-point data registers and the main memory or the CPU data registers.

b. Arithmetic operation instructions. One or two operands are specified. The results is always stored into one of the internal FPC data registers. At most one operand may come from outside (the CPU registers or the memory).

c. System control instructions.

2.4 CONCURRENCY AND SYNCHRONIZATION

Since one FPC instruction usually takes much more time to complete than a CPU instruction, it is desirable for both the CPU and the FPC to be executing instructions simultaneously. When the CPU encounters an FPC instruction, it (1) busy-waits until the FPC becomes idle (although it can service interrupts), (2) causes the FPC to begin the instruction, and then (3) immediately services the next instruction.

The concurrency between the CPU and the FPC must be implemented to maintain a programming model based on sequential instruction execution. There are two possible ways the CPU and the FPC may interfere with one another; they must both be disallowed:

a. The CPU cannot start another FPC instruction if the FPC is still busy executing the previous instruction.

b. The CPU cannot reference a memory location that is being referenced by the previous (but still executing) FPC instruction.

The first coordination problem is solved by the CPU's busy wait feature, while the second one is solved by the assumption underlying the FPC instruction set. There is no arithmetic instruction that can store the result to the memory or the CPU registers. The only way to move data from the FPC to the outside world is through the FSTR instruction, which finishes execution as soon as the communication between the CPU and the FPC is finished. Therefore the CPU guarantees that all programs will produce the same result as if there were only one processor instead of two.

3.0 VERIFYING THE FPC TOP LEVEL FROM THE THREE COMMUNICATING UNITS

As we recall from the previous section, one floating-point instruction is accomplished by the cooperative effort of three units: the CPU, the BIU and the APU. The verification of the communication among the three units is the main focus of this section.

The FPC top-level interpreter is implemented by three interpreters: the BIU top-level interpreter, the APU top-level interpreter and the CPU service interpreter (fig. 3.1-1). Each interpreter has a separate specification, making it possible to reason independently about each interpreter. The specifications are then composed in order to reason about their interaction. Thus, a proof that each interpreter is implemented correctly can be carried out independently of the others.

Since each interpreter is specified independently, we do not assume there is a master clock shared by all the units. Each unit has its own clock. Moreover, for each unit, interpreters on different abstraction levels also have different time scales. All communication between the units is assumed to be performed asynchronously. The busy waiting mechanism used by Joyce (ref. 7) was used to specify and verify the asynchronous interaction among the units.

3.1 HIERARCHICAL DECOMPOSITION

Windley (ref. 10) describes a hierarchical decomposition of a computer system as a sequence of interpreters:

$$S \Rightarrow B_1 \Rightarrow ... \Rightarrow B_n$$

where S is the structural description, and B_1 through B_n represent increasingly abstract specification of the system. Generalizing this principle leads to a tree structure or possibly a directed, acyclic graph, if the computer system structure is a collection of independently operating units. In the case of a tree, at the root of the tree we have the top-level specification for the whole system. The leaves of the tree correspond to the structural specifications of the physical units. The nodes in the middle represent various intermediate abstract specifications for the units.

By applying the above generalized decomposition principle to the floating-point coprocessor, we obtain the decomposition illustrated in fig. 3.1-1. This model extends the one Windley adapted for his microprocessor by allowing an interpreter to be implemented by more than one lower-level interpreter. Therefore, besides the vertical interaction between the lower-level and the higher-level in-

terpreters, there is *horizontal* interaction between the lower-level interpreters. The verification that the composition of several lower-level interpreters implements a higher-level interpreter presents a problem not previously formalized for hardware verification, and is addressed here.

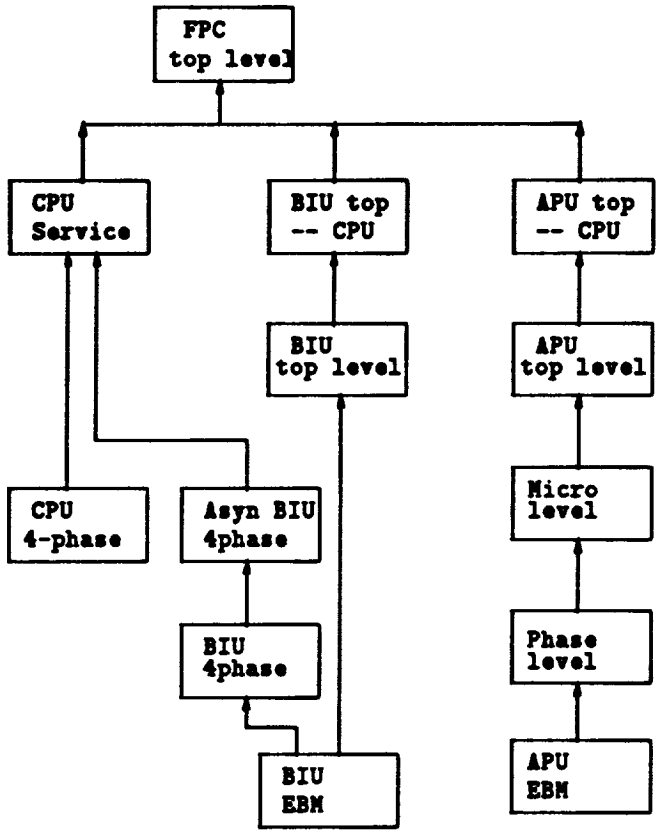


Figure 3.1-1: An Extended Hierarchy of Interpreters for the FPC

We adapted a model for describing the interpreters at various abstraction levels. An interpreter is a process organized as a state transition system defining the state s, the environment e, and a set of transition functions J that relate the state at time t+1 to the state and the environment at time t. In our model, the environment is used only for input; output to the environment is modeled as part of the state. Each level in the interpreter hierarchy has its own state, environment and time scale. A mapping function is used to relate the states, environments and the time scale of vertically adjacent interpreters.

3.2 THE FOUR-PHASE HANDSHAKING PROTOCOL

The FPC executes two different bus cycles, according to the direction of the transfer. They are the asynchronous read and asynchronous write bus cycles. The four-phase handshaking protocol is used to implement these two types of bus cycles. For the read cycle, the FPC detects the start of an asynchronous read cycle when the read request line is asserted by the CPU. The FPC puts its data value on the bus and asserts the acknowledgement line (dtack). The acknowledgement remains asserted until the read request line is lowered by the CPU to signify the end of the read cycle. The BIU then lowers the acknowledgement line (fig. 3.2-1).

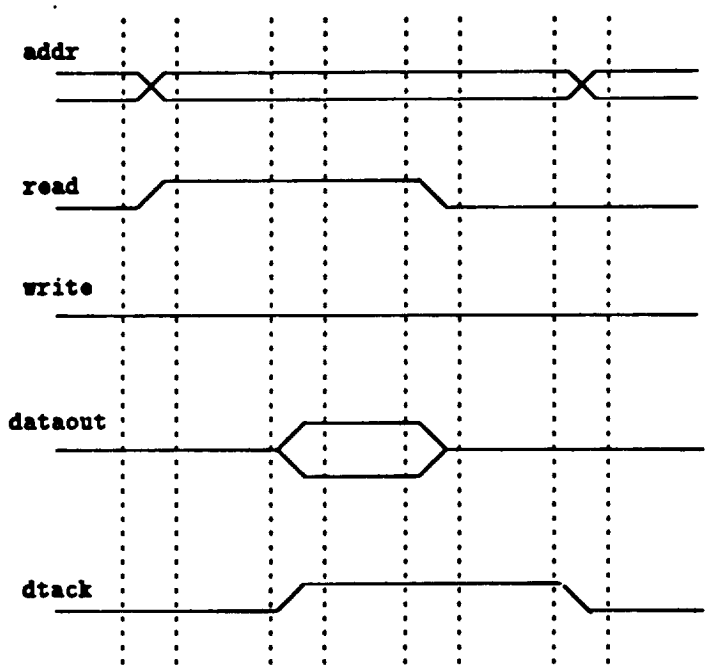


Figure 3.2-1: The 4-Phase Protocol Time Diagram

Fig. 3.2-1 represents the actual signal exchange in the implementation of the four-phase handshaking protocol. The implementation of this protocol is represented in the lower levels of the interpreter hierarchy (e.g., CPU 4-phase). We would like, however, to define a more abstract view of message passing between the CPU and the BIU where the handshaking detail is hidden. This more abstract view of communication can then be used in the next higher level, that is, in reasoning about the interaction between the CPU and the BIU during the execution of an FPC instruction.

The definition `cir_read_write` states that the read or write cycle will be successfully accomplished in one time unit on the more abstract (higher-level) time scale. In addition, proper signals such as

new_instr and operand_ready are raised so that the BIU will know which CIR the data is latched into. Since the data transmitted on the data bus are of type *wordn (abstract type of a word of length n), type conversion is needed in both reading and writing cycles. When an instruction is written to the command CIR, signal new_instr is raised. Similarly, operand_ready will be raised if the data is written into the output operand CIR.

```
cir_read_write rep address read write datain dataout
                  command response operand_in operand_out condition
                  control new_instr operand_ready =
⊢ ∀ t.
   (read t) ⇒  %read cycle%
     ((address t =1) ⇒ ((datain(t+1) = (numtow rep (response t))) ∧
                        ~(new_instr t) ∧
                        ~(operand_ready t)) |
                        ((datain(t+1) = (fptow rep (operand_in t))) ∧
                        ~(new_instr t) ∧
                        ~(operand_ready t))) |
   (write t) ⇒  %write cycle%
     ((address t =0) ⇒ (((command(t+1))= (wtonum rep (dataout t))) ∧
                        (new_instr t = T)) |
                        ((operand_out(t+1) = (wtofp rep (dataout t))) ∧
                        (operand_ready t = F) ∧
                        (operand_ready (t+1) = T))) |
    %idle cycle %
                        ((new_instr t = F) ∧
                        (operand_ready t = F))
```

To prove that the implementation of the four-phase handshaking protocol actually implements the more abstract description cir_read_write, a temporal abstraction from the lower-level time scale to the upper-level time scale has to be used (ref. 13). Temporal abstraction deals with the sequential or time-dependent behavior of a device viewed at different "grains" of discrete time.

3.3 SPECIFICATION OF THE THREE INTERPRETERS

This section describes the specification of the top-level interpreters of CPU service, BIU and APU. The state tuple and the environment tuple for each of the interpreters are listed in the following

two tables:

Interpreters	*state*
CPU_Service	(c_ac,c_reg,mem,dataout,address,read,write, cpu_state)
BIU	(response,operand_out,decode_reg,resp_ready,f_ac, biu_state,start)
APU	(f_ac,f_reg,cw,sw,done)

Interpreters	*environment*
CPU_Service	(resp_ready,datain,ir)
BIU	(command,condition,control,operand_in,done,new_instr, operand_ready)
APU	(start,decode_reg)

Although the data bus physically is one bi-directional bus, we chose to represent it as two uni-directional buses: datain and dataout. Fig. 3.3-1 depicts the connection between the CPU and BIU. Instructions and data fetched from the memory are written by the CPU to the command and output operand CIR through dataout. Similarly, response and data fetched from the FPC accumulator are sent by the BIU through the datain bus. The CPU and FPC accumulator and registers are denoted as c_ac, c_reg, f_ac and f_reg, respectively.

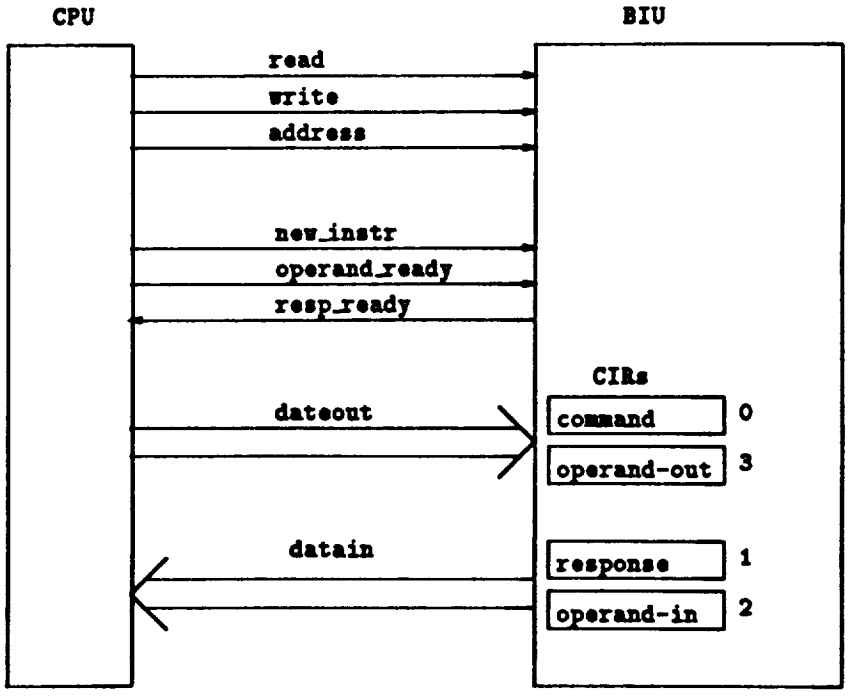


Figure 3.3-1: The States and Environment

3.3.1 SPECIFYING THE CPU SERVICE INTERPRETER

The CPU service interpreter is the part of the CPU specification that is responsible for communicating with the BIU in order to execute a floating-point instruction. Since the CIRs reside in the BIU, reading from or writing to any CIR by the CPU is accomplished by the four-phase handshaking protocol.

The CPU service interpreter specification is organized as a state machine that has six states. In state 0, the CPU starts the communication by sending out the current FPC instruction to the BIU. The CPU then "busy waits" in state 1 for the BIU to put the proper service request in the response CIR. In state 3, the CPU reads the response CIR, and enters the appropriate state to provide the requested service. If the response is a null primitive, then the CPU is no longer needed and, therefore, is free to execute other instructions; if the response is a read primitive (for FLD instruction), then the CPU fetches the data from the memory, raises the write request line, puts the data on the data bus, and goes back to state 0; if the response is a write primitive (for the FSTR instruction), then the CPU waits until the data is ready in the operand CIR, retrieves it from the data bus, and stores it to the memory.

The function cpu_service_state determines the new state for the CPU service interpreter from the current state and the environment by the current value of the state counter cpu_state.

```
cpu_service_state n rep cpu_state =
⊢ ((cpu_state = 0) ⇒ (cpu_begin n rep) |
    (cpu_state = 1) ⇒ (cpu_wait_for_response n rep) |
    (cpu_state = 2) ⇒ (cpu_wait_4phase n rep) |
    (cpu_state = 3) ⇒ (cpu_read_response n rep) |
    (cpu_state = 4) ⇒ (cpu_wait_read n rep) |
                      (cpu_put_data n rep))
```

There are four parameters for each state transition function. The second parameter rep is the representation parameter for abstract data types such as *wordn and *memory. A number of abstract operations are defined on the abstract data types. For example, the function wtonum converts a *wordn typed object to type number. The third parameter is the state tuple, and the last one is the environment tuple. The following is the definition of cpu_begin where the CPU sends out a new instruction to the BIU. The CPU raises the write request line and puts the instruction on the data bus.

```
cpu_begin n rep (c_ac, c_reg, mem, ir, dataout, address, read, write,
           cpu_state) (resp_ready, datain) =
⊢ (c_ac, c_reg, mem, ir, (numtow rep ir), 0, F, T, 1)
```

Note a few details regarding state 1, where the CPU waits for the BIU to finish putting the appropriate response primitive into the response CIR. Since the CPU and the FPC do not share the same master clock, the FPC will have to raise a response ready line (resp_ready) to inform the CPU when the response is ready. The MC68881 does not use the technique of a resp_ready line. Instead, a so-called "synchronous" read cycle is used to read the response CIR. This "synchronous" read cycle relates the bus cycle timing directly to the FPC clock to allow the proper response primitive to be prepared.

The top-level specification of CPU service relates the state at time t+1 to the state and environment at t by the next state function:

```
cpu_service n rep (c_ac, c_reg, mem, ir, dataout, address, read, write,
           cpu_state) (resp_ready, datain) =
⊢ ∀ t. (c_ac(t+1), c_reg(t+1), mem(t+1), ir(t+1), dataout(t+1),
        address(t+1), read(t+1), write(t+1), cpu_state(t+1)) =
    cpu_service_state n rep (cpu_state t)
        (c_ac t, c_reg t, mem t, ir t, dataout t, address t, read t,
         write t, cpu_state t)
        (resp_ready t, datain t)
```

3.3.2 SPECIFYING THE BIU TOP-LEVEL

In order to allow a FPC instruction to start executing, the BIU communicates with the CPU to obtain the necessary information, and to inform the APU to start processing in the case of an arithmetic instruction. There are four states for the BIU top-level state machine. In state 0, the BIU is idle and waits for the next instruction to be sent over by the CPU. In state 1, the BIU decodes the current instruction in the command CIR and determines whether the CPU is needed to transfer data. If a transfer is required, a read or write primitive is placed into the response CIR. Otherwise, the null primitive is issued to inform the CPU of the termination of the communication for arithmetic instructions. Further, the APU is started when the instruction is an arithmetic instruction. If the instruction is FLD, the BIU waits for the CPU to place data in the operand CIR

and then the BIU transfers the data to the FPC accumulator. If the instruction is FSTR, the BIU fetches data from the FPC accumulator and stores it in the operand CIR, making it available for the CPU. For arithmetic instructions, the BIU waits till the APU finishes executing.

```
biu_top_state n rep biu_state =
⊢   (biu_state = 0) ⇒ (biu_idle n rep) |
    (biu_state = 1) ⇒ (biu_decode n rep) |
    (biu_state = 2) ⇒ (biu_wait_op n rep) |
    (biu_state = 3) ⇒ (biu_fld n rep) |
                      (biu_wait_apu n rep)
```

In order to compose the CPU, the BIU and the APU interpreter, the three interpreters have to be expressed with respect to a *common* time scale. We chose the cpu_service clock rate as the common clock rate. Therefore, the behavior of biu_top and apu_top must be described in terms of the clock rate of cpu_service. We chose to use existential quantification to describe the BIU top-level behavior. More specifically, the specification states that there exists some future time t' such that the BIU finishes executing. If the clock of apu_top is faster or of the same speed as the clock of cpu_service, t' will be equal to t+1. However, if the apu_top clock is slower, t' will be bigger than t+1.

The following definition is the APU top-level behavior from the CPU's point of view. The interpreter for the BIU top-level relates the state tuple at time t+1 or t' to the state and environment tuple at time t, depending on whether the BIU is waiting for the CPU or is doing some work required for the current FPC instruction. For example, at state 1, if the FPC instruction arrives at CPU time t, then at CPU time t' the instruction will be decoded and the proper response is sent. Further, t' is the first time that the response is ready (First (t, t') resp_ready), and the start signal to the APU will remain stably low from t to t' (Stable (start, F, t, t')).

```
biu_top_cpu n rep (response, operand_out, decode_reg, resp_ready, f_ac,
                   biu_state, start)
            (command, condition, control, operand_in, done, new_instr,
             operand_ready) =
  ∀ (t:num). ∃ (t':num). ((t+1)<=t') ∧
  (let state_tuple = (state_tuple_biu t) and
       state_tuple1 = (state_tuple_biu (t+1)) and
       env_tuple = (env_tuple_biu t) and
       state_tuple' = (state_tuple_biu t') in

  (((biu_state t = 0) ∨ (biu_state t = 2)) ⇒
       (state_tuple1 = biu_top_state n rep (biu_state t)
                                           state_tuple env_tuple) |
   (biu_state t = 1) ⇒
       ((First (t, t') (resp_ready)) ∧
       (Stable (start, F, t, t')) ∧
       (state_tuple' = (biu_decode n rep state_tuple env_tuple))) |
   (biu_state t = 3) ⇒
       ((~(operand_ready t)) ⇒
            (state_tuple1 = (response t, operand_out t, decode_reg t,
                   resp_ready t, f_ac t, 3, start t)) |
            ((Stable (start, F, t, t')) ∧
            (state_tuple' = (response t, operand_out t, decode_reg t,
                   resp_ready t, operand_out t, 0, start t)))) |
       (state_tuple' = (biu_top_state n rep (biu_state t)
            state_tuple env_tuple))))
```

3.3.3 SPECIFYING THE APU TOP-LEVEL

The APU top-level interpreter describes the APU behavior with respect to each floating-point arithmetic instruction. The state on the APU top-level does not contain the instruction register, the CPU accumulator, the CPU registers and the memory. Instead the APU gets the current instruction from the decoder register that resides inside the APU. For example, the specification APU_FADD asserts that the content of the FPC accumulator is added to the content of a designated FPC register, and the result is stored in the accumulator.

```
APU_FADD n rep (f_ac, f_reg, cw, sw, done)
               (start, decode_reg) =
⊢ let (addr:num) = (Addr n decode_reg) in
          ((FST (FP_ADD n1 n2 f_ac (f_reg addr))), f_reg,
             cw, sw, T)
```

NextState_apu determines the new state for the APU top-level interpreter from the current state and environment according to the opcode of the current instruction stored in the decoder register. The execution of load and store instructions do not involve the APU. For this discussion, we are only concerned with four arithmetic instructions, that is addition, subtraction, multiplication and division.

```
NextState_apu n rep opcode =
⊢  ((opcode = 2) ⇒ (APU_FADD n rep) |
    (opcode = 3) ⇒ (APU_FSUB n rep) |
    (opcode = 4) ⇒ (APU_FMUL n rep) |
                   (APU_FDIV n rep))
```

Similar to biu_top_cpu, apu_top_cpu specifies the APU top-level behavior from the CPU's point of view. It states that if the start signal has not arrived, the APU will remain idle. If the APU starts to execute at CPU time t, however, then there exists some future CPU time t' that the APU will finish the execution of the instruction.

```
apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
          (start, decode_reg) =
⊢ ∀ t . (¬start t) ⇒
    ((f_ac(t+1), f_reg(t+1), cw(t+1), sw(t+1), done(t+1)) =
        apu_idle n rep (f_ac t, f_reg t, cw t, sw t, done t)
                 (start t, decode_reg t)) |
     (∃ t' . (t≤t') ∧
            (First (t, t') (done)) ∧
       ((f_ac t', f_reg t', cw t', sw t', done t') =
       NextState_apu n rep (Opc n (decode_reg t))
                 (f_ac t, f_reg t, cw t, sw t, done t)
                 (start t, decode_reg t)))
```

3.4 VERIFYING THE FPC TOP-LEVEL

This section describes the methodology used to verify that the cooperative effort of the three interpreters implements the FPC top-level specification. The key to this proof is the asynchronous communication among the three interpreters.

3.4.1 SPECIFYING THE FPC TOP-LEVEL

The function performed by each FPC instruction is not accomplished by the coprocessor alone, but requires interaction with the CPU. Hence, the state visible at this level includes the FPC registers plus the CPU registers and the memory contained in the CPU service interpreter state. The CPU registers and the instruction register do not reside inside the FPC, nor does the coprocessor directly reference the memory, but the FPC top-level interpreter reflects the assembly language programmer's point of view of floating-point instruction execution and treats the FPC as a single unit. For example, from the programmer's point of view, the current FPC instruction being executed is in the instruction register, and the FLD and FSTR instructions directly load from and store to the memory.

The specification for the FPC instruction FLD at the FPC top-level is as follows, where c_ac, c_reg, mem and ir are the CPU accumulator, the CPU data registers, the main memory and the instruction register, respectively. The FLD instruction fetches data from the main memory and stores it to the FPC accumulator. Function (Addr n ir) returns the memory address of the operand that is fetched.

```
FLD n rep (f_ac, f_reg, c_ac, c_reg, mem) (ir) =
⊢ let data = fetch rep mem (Addr n ir) in
      (((wtofp rep) data), f_reg, c_ac, c_reg, mem)
```

The interpreter for the FPC top-level is expressed as a predicate relating the value of the state tuple at time t' to the value of the state tuple at time t according to the function NextState_fpc. The reason that each instruction is specified to finish at some time t' instead of t+1 is that the behavior of the FPC is specified from the CPU's point of view. This specification will be used later to compose the CPU top-level specification and the FPC top-level specification to form a complete computer system.

```
fpc_top n rep (f_ac, f_reg, c_ac, c_reg, mem) (ir) =
⊢ ∀ t . ∃ t'.
    (f_ac t', f_reg t', c_ac t', c_reg t', mem t') =
      NextState_fpc n rep (ir t) (f_ac t, f_reg t, c_ac t, c_reg t,
                    mem t) (ir t)
```

3.4.2 WAITING STATES

Sometimes the APU top-level interpreter and the CPU service interpreter must wait until some signal becomes high. During these waiting periods, the state tuple remains unchanged. For example, for the CPU service interpreter, its state machine waits in state 1 until the signal `resp_ready` becomes high. Using induction on the length of the wait period, we can prove theorem `wait_cpu` stating that the state counter `cpu_state` (which is an element of the `cpu_service` state tuple) remains stable while the CPU is waiting in state 1 for the signal `resp_ready`.

```
wait_cpu =
⊢ cpu_service n rep (c_ac, c_reg, mem, ir, dataout,
                address, read, write, cpu_state) (resp_ready, datain)
        ⟹
        ∀ (t:num). (cpu_state t = 1) ⟹
            StableUntil (cpu_state, 1, t, resp_ready)
```

There are a number of theorems in this form. For example, theorem `wait_cpu2` states that the memory and the instruction register remain stable until the response has arrived (because the CPU has not been freed from its floating-point execution duty at this point). Similarly, `wait_apu` guarantees that the FPC accumulator and registers will remain stable until the signal `start` from the BIU arrives. Theorems such as this will be used in the next section to reason about the interaction of the three implementing interpreters to prove the FPC top-level interpreter.

3.4.3 VERIFYING THE FLD INSTRUCTION

To verify the FPC top-level interpreter, the interaction of the three implementing interpreters must be formalized. Although the FPC top-level interpreter is defined in terms of state transitions that represent the execution of single instructions, each FPC transition represents some unspecified

number of transitions by the lower-level interpreters. An example of this is shown in the following theorem, FLD_Correct.

```
FLD_Correct =
⊢ ∀rep response operand_out decode_reg biu_state start command
      condition control operand_in done new_instr operand_ready
      resp_ready f_ac f_reg cw sw c_ac c_reg mem ir cpu_state
      address read write datain dataout n.
      (biu_top_cpu n rep state_tuple_biu env_tuple_biu) ∧
      (apu_top_cpu n rep state_tuple_apu env_tuple_apu) ∧
      (cpu_service n rep state_tuple_cpu_service env_tuple_cpu_service) ∧
      (cir_read_write rep
        address read write datain dataout command
        response operand_in operand_out condition control
        new_instr operand_ready) ⟹
      (∀t.
        (Opc n(ir t) = 0) ∧ %FLD%
        ~start t ∧
        (biu_state t = 0) ∧
        (cpu_state t = 0) ⟹
        (∃t'.
          (f_ac t',f_reg t') =
          FPCstate(FLD n rep(f_ac t,f_reg t,c_ac t,c_reg t,mem t,ir t))))
```

Theorem FLD_Correct states that the FLD instruction is correctly implemented by the three interacting units: the CPU service, the BIU and the APU. More formally, the definitions of the CPU service, BIU, APU and the correctness of the four-phase protocol imply that for any time t, where the current instruction is FLD and each of the three units are in their initial states, the effect of running the three lower-level interpreters together is the same as that of running the top-level FPC interpreter. Function FPCstate is used to filter out some elements from the state tuple that might be changed by the CPU after the CPU is freed from its floating-point activity. These elements are the CPU accumulator and registers, the memory, and the instruction register. The same is true for the arithmetic instructions. However, for the FSTR instruction, the entire state will be used since the CPU will not be free until the FSTR is finished execution.

To illustrate the verification technique, we examine the interaction between the three units for the FLD instruction. At time t, the BIU and APU are idle and cpu_service initiates the communication

by sending out the current FPC instruction to the BIU. More specifically, the CPU raises the write request line and places the instruction on the data bus (state transition function cpu_begin). The correctness property of the four-phase handshaking protocol (cir_read_write) ensures that the instruction is delivered to the command CIR and the signal new_instr is raised.

At time t+1 the state machine of the CPU service interpreter is in state 1 and waiting for the response ready signal resp_ready from the BIU. At time t+2 the BIU discovers that a new instruction is placed in its command CIR; it then decodes the instruction and puts a proper response primitive into the response CIR. In this case, the response is a read primitive. Since the BIU operates on a different clock, the decoding will be done at time t', with t' bigger than t (state transition function biu_decode).

During the interval t+2 to t', the CPU and the APU are idle. The waiting theorems described in the previous section ensure that the state tuples of the CPU service and the APU remain unchanged. The CPU detects that the proper response is ready at t', and enters state 2 to wait for the response to be put on the databus. The CPU reads the response from the databus at t'+2 and realizes that data needs to be fetched from the memory. Thus, the CPU fetches the data, raises the write request line and puts the data on the databus (cpu_read_response). From now on, the CPU is no longer needed, and is free to execute other CPU instructions. The four-phase protocol ensures that the data will be put into the operand CIR. At time t'+4 the BIU stores the data to the accumulator and the whole operation is finished (biu_fld). During the entire process, the APU remains idle. A more detailed view of the communication among the three interpreters is illustrated in fig. 3.4-1.

3.4.4 VERIFYING THE FPC TOP-LEVEL INTERPRETER

Theorems like those above are proved for every FPC instruction (currently six of them). These theorems are used to establish a theorem stating that the three top-level interpreters of CPU service, BIU and APU imply the FPC top-level. The initial condition is more complicated in this proof. Usually when proving the correctness of a microprocessor, for instance from micro level to macro level, the initial condition will be that the microprogram counter is zero. Here a function Initial_State is defined to include the initial conditions for all three components.

	t	t+1	t+2	t'	t'+1
CPU	begin by sending instr	idle	idle (induc)	idle	
BIU	idle	idle	decode send response	idle	
APU	idle	idle	idle (induc)	idle	
4-ph		write command cir			

	t'+1	t'+2	t'+3	t'+4	t''
CPU	idle	read response send data	free now		
BIU	idle	idle	idle	put data to ac	
APU	idle	idle	idle	idle	
4-ph	read response	idle	write data		

Figure 3.4-1: The Interaction of the Three Units for FLD Instruction

```
fpc_top_w_initial n rep (f_ac,f_reg,c_ac,c_reg,mem) (ir)
        (start,new_instr,resp_ready,read,biu_state,cpu_state) =
⊢ (∀t.
      Initial_State
      (start,new_instr,resp_ready,read,biu_state,cpu_state,t) ∧
      ValidOpcode n rep(ir t) ⟹
      (∃t'.
        (f_ac t',f_reg t') =
        FPCstate
        (NextState_fpc n rep (ir t)
           (f_ac t,f_reg t,c_ac t,c_reg t,mem t) (ir t))))
```

Using the theorems for each valid FPC instruction and the definitions that were given above, we can prove the final theorem for this level:

```
FPC_Correct =
⊢ ∀rep response operand_out decode_reg biu_state start command
      condition control operand_in done new_instr operand_ready
      resp_ready f_ac f_reg cw sw c_ac c_reg mem ir cpu_state address
      read write datain dataout n.
      (biu_top_cpu n rep state_biu env_biu) ∧
      (apu_top_cpu n rep state_apu env_apu) ∧
      (cpu_service n rep state_cpu_service env_cpu_service) ∧
      (cir_read_write rep
        address read write datain dataout command
        response operand_in operand_out condition control
        new_instr operand_ready) ⟹
     fpc_top_w_initial n rep (f_ac,f_reg,c_ac,c_reg,mem,ir)
            (start,new_instr,resp_ready,read,biu_state,cpu_state)
```

4.0 COMBINING THE CPU AND THE FPC

With the inclusion of the FPC, the system provides an instruction set which contains both the CPU instruction set and the FPC instruction set. At the abstraction level of the assembly language programmer, the system appears as a single unit. Furthermore, at this level of abstraction, the instructions appear to be executed in a sequential manner. However, the underlying implementation of instructions is concurrent. This sequential execution of instructions is "equivalent" to the underlying concurrent execution only in the sense that the final result of the program will be identical. This section discusses our methodology for specifying and verifying the computer system consisting of the CPU and the FPC as shown in fig. 4.0-1. More specifically, we are going to discuss the approach to abstract the FPC top-level to the concurrent top-level and subsequently to the sequential top-level.

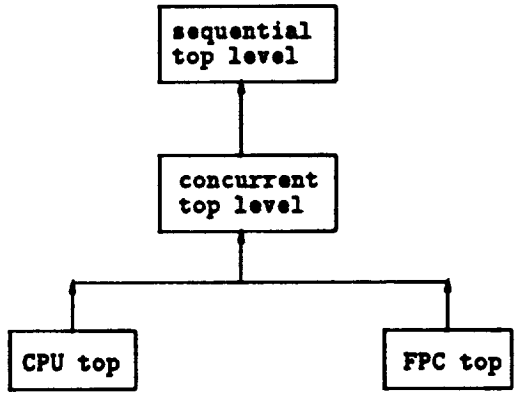


Figure 4.0-1: Hierarchical Decomposition of the CPU-FPC System

4.1 SPECIFYING THE SEQUENTIAL TOP-LEVEL INTERPRETER

The sequential top-level interpreter (shown in outline form in `Sequential_Top_Level`) reflects the assembly language programmer's view of sequential execution. The functions `FPC_NextState` and `CPU_NextState` execute the current FPC or CPU instruction pointed to by the program counter.

```
Sequential_Top_Level state env =
⊢ ∀ time t.
   if current instr is a FPC instr ⇒
        state(t+1) = FPC_NextState (state t) (env t) |
        state(t+1) = CPU_NextState (state t) (env t)
```

The total state at this level state can be partitioned into three disjoint sets: (1) cpu_s, which includes the CPU data registers and the memory; (2) fpc_s, which includes the FPC data registers; and (3) joint_s, which has the program counter. If the current function is FSTR, the function FPC_NextState modifies the total state. However, if the current instruction is a FPC instruction other than FSTR, then FPC_NextState only modifies fpc_s and joint_s. Similarly, the function CPU_NextState only modifies cpu_s and joint_s, but not fpc_s. Note that cpu_s is not the state at the CPU top-level. The same applies for fpc_s. This will be discussed in the next section.

At the concurrent top-level, each clock tick represents the start of a new instruction. A CPU instruction only needs one clock cycle to complete, while a FPC instruction may need more than one cycle. In addition, a FPC instruction cannot be started unless the FPC has finished the previous instruction. The state and the environment at this level is the same as those at the sequential top-level.

```
Concurrent_Top_Level state env =
⊢ ∀ time t.
  if the current instr is of FPC type ⇒
        (∃ c. at t+c, the FPC instr will be completed ∧
        ∀ t'. (0<t'<c ⇒
            the instr at t+t' is not a FPC instr|
        at t+1, the CPU instr will be completed)
```

We need to prove the following sequential top-level correctness statement (with the correct abstraction function).

```
Concurrent_Top_Level ∧ Non_Interference ⟹ Sequential_Top_Level
```

The non-interference property has the following four aspects:

a. The state of the CPU cpu_s and the FPC fpc_s are disjoint.

b. The CPU never modifies the FPC state.

c. The FPC never modifies the CPU state except using FSTR instruction.

d. A FPC instruction is never started unless the FPC has finished executing the previous instruction.

Conditions (a) to (c) are implicitly provided by the nature of the instruction set, and are modeled in Non-Interference in the above correctness statement. The fourth condition is guaranteed by the implementation and modeled in the specification of the concurrent top-level interpreter.

4.2 THE ABSTRACTION FROM THE CONCURRENT LEVEL TO THE SEQUENTIAL LEVEL

Suppose the sequential execution flow of a instruction stream is "$F_1C_1C_2C_3F_2F_3C_4F_4C_5C_6$", where F's and C's represent FPC and CPU instructions, respectively[2]. The sequential and concurrent execution sequence of the above instruction stream is depicted in fig. 4.2-1. The dotted line indicates the mapping points of the abstraction, which will be discussed later. The instruction labels marked above the line are the starting time of the corresponding instructions, while the labels below the line represent the finishing time. For example, on the concurrent level, instruction F_1 starts at time 0, finishes at time 3.

Fig. 4.2-2 shows the sequential top-level state and the concurrent top-level state for the above instruction stream at each clock tick. The CPU state cpu_s on both levels are always the same at each clock tick. However, at some point the FPC state fpc_s on the concurrent level is different from fpc_s on the sequential level. For instance, at time 2 the total state on the sequential level equals the CPU state modified by C1, the FPC state modified by instruction F1, and the incremented programmer counter (the first table). At the corresponding time the total state on the concurrent level equals the CPU state modified by C1, the FPC state at time 1 and the incremented programmer counter (the second table). In this particular example, the total state on the sequential level and the total state on the concurrent level are equivalent at time 3, 4, 5, 7 and 10; while at all other points the total states on the two levels are different.

[2]because of branching instructions, the program and its sequential execution sequence might be different

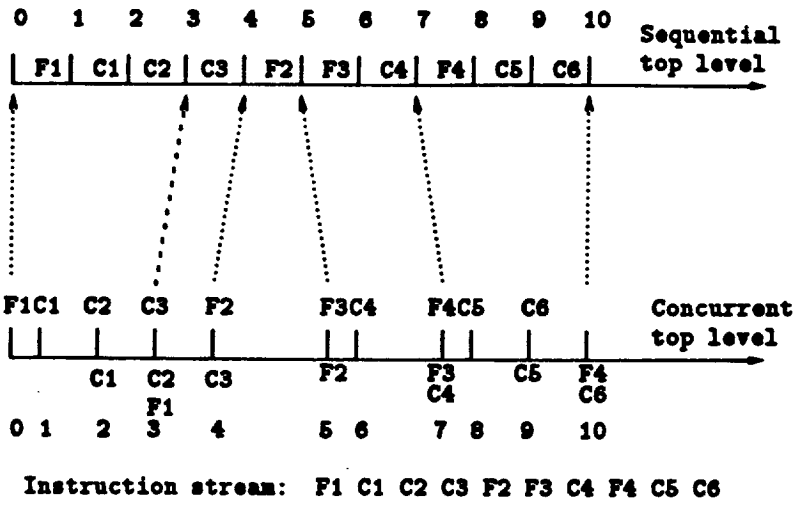


Figure 4.2-1: The mapping between the sequential top-level and the concurrent top-level

More formally, assume the instruction stream is:

$$F, C_1, C_2, ..., C_n, F'$$

Since CPU instructions C_1 to C_n do not interfere with F, any one of the following execution sequences will have the same result as the above. The only requirement is that F must be executed before F'. Depending on the instruction type of F and how fast each instruction is executed, the concurrent top-level interpreter will specify one of the following sequences:

$$F, C_1, C_2, C_3, ..., C_n, F'$$

$$C_1, F, C_2, C_3, ..., C_n, F'$$

$$C_1, C_2, F, C_3, ..., C_n, F'$$

.....

$$C_1, C_2, C_3, ..., C_n, F, F'$$

Suppose F does not finish until C_i starts. This is equivalent to:

$$C_1, C_2, ..., C_{i-1}, F, C_i, ..., C_n, F'$$

Further assume the instruction stream starts at time 0. Then the FPC state on the concurrent level from time 0 to time i-1 is always equivalent to the FPC state at time 0 on the sequential level, since F does not finish executing until time i. However, from time i to n, the total state

sequential:

time / state	1	2	3	4	5	6	7	8	9	10
cpu_s		C1	C2	C3	C3	C3	C4	C4	C5	C6
fpc_s	F1	F1	F1	F1	F2	F3	F3	F4	F4	F4
joint_s	1	2	3	4	5	6	7	8	9	10

concurrent:

time / state	1	2	3	4	5	6	7	8	9	10
cpu_s		C1	C2	C3	C3	C3	C4	C4	C5	C6
fpc_s			F1	F1	F2	F2	F3	F3	F3	F4
joint_s	1	2	3	4	5	6	7	8	9	10

Figure 4.2-2: The state on the sequential top-level

on the two levels are identical. Although the exact time that F finishes is unknown, F will finish *before* F' starts. Therefore at time n+1 (the time F' starts), the total state on the sequential level is guaranteed to be identical to the total state on the concurrent level.

Instead of looking at the state after the execution of each instruction, a "chunk" of instructions is examined. Each "chunk" is defined by a sequence of the form FC^*, denoting one floating-point instruction followed by zero or many CPU instructions. The start of a "chunk" is a floating-point instruction, and the termination is the last CPU instruction that precedes a floating-point instruction. In the example presented in fig. 4.2-1, the "chunks" are "$F_1C_1C_2C_3$", "F_2", "F_3C_4", "$F_4C_5C_6$". At the end of each "chunk", the states on the two levels are always identical. Those points, therefore, become the mapping points between the concurrent top-level and the sequential top-level.

4.3 VERIFYING THE CONCURRENT TOP-LEVEL

As depicted in fig. 4.0-1, the concurrent top-level interpreter is implied by two lower-level interpreters: the CPU top-level and the FPC top-level interpreters. The synchronization between the CPU and the FPC is specified by both the CPU and the FPC top-level interpreters.

The CPU top-level interpreter determines if the current instruction is a FPC instruction. If it is, then the CPU will synchronize with the FPC by waiting until the FPC is idle. Otherwise, the interpreter relates the state at t+1 to the state and environment at t through the CPU instruction selected according to the current program counter: At this level, the state has part of the concurrent top-level state which does not concern the FPC, for example the CPU registers, the memory, the program counter, plus the busy line emitted by the FPC.

```
CPU_Top_Level state_cpu env_cpu =
⊢ ∀ time t.
    if the current instr is FPC type ⇒
        (if FPC is busy ⇒
            state_cpu(t+1) = wait until FPC is idle  |
            state_cpu(t+1) = finish the communication) |
        state_cpu(t+1) =
            (Selected CPU instr) (state_cpu t) (env t)
```

The concurrent top-level correctness statement is as follows:

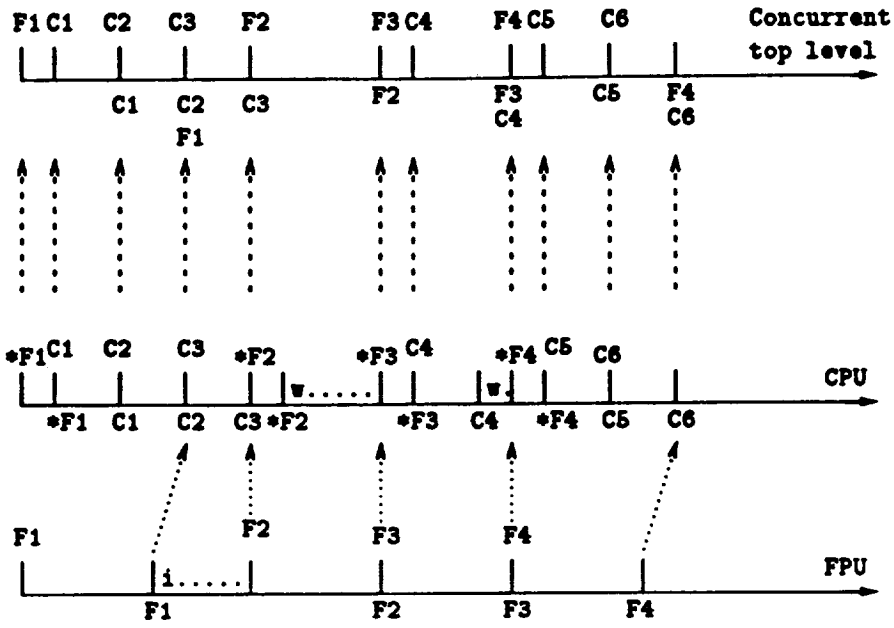


Figure 4.3-1: The mapping between the concurrent top-level and the CPU, FPC top-level

$$(\text{CPU\_Top\_Level} \wedge \text{FPC\_Top\_Level}) \Longrightarrow \text{Concurrent\_Top\_Level}$$

Usually, when proving an implementation (lower-level interpreter) implies a specification (an upper-level interpreter), a temporal abstraction is defined from the lower-level time scale to the upper-level time scale (fig. 4.3-1). However, in this case there are two units on the lower level, each with its own independent time scale. One way to solve this problem is to map the FPC time to the CPU time scale.

For example, in fig. 4.3-1, F_1 is actually completed by the FPC before the CPU completes the execution of C_2. However, since the clock ticks on the concurrent top-level only represent the starting time of each instruction, F_1 is not considered to be finished until C_3 starts executing, which is the same time C_2 finishes. The notation *F1 denotes the communication between the CPU and the FPC for the instruction F_1. The waiting of the CPU for the FPC to finish execution is signified by "w.....", and the FPC's idle period is signified by "i.....". As we can see in fig. 4.3-1, the waiting period of the CPU is collapsed into the execution time of the previous instruction, for example, C_4.

5.0 CONCLUSION

This report describes work on the verification of a floating-point coprocessor from three communicating implementing processes. Traditionally, hierarchical decomposition only involves vertical abstraction: an abstraction level is implemented by a single implementing level under it. In this project, three interacting processes are used to prove the correctness of the implemented process, that is, the FPC top-level interpreter.

Furthermore, we described the methodology for the composition of the CPU and coprocessor top-level specifications to form a more abstract view of the whole system consisting of the CPU and the FPC. Two levels of abstraction are used, the concurrent top level and the sequential top level. We believe this kind of abstraction approach presents a very interesting problem. Extending this approach, a complete system composed of many devices can be shown to correctly implement an abstract system abstraction.

REFERENCES

1. *MC68881/882 Floating-Point Coprocessor User's Manual.* Englewood Cliffs, N.J. 07632, 1989.

2. A. D. Wilcox, *68000 Microcomputer Systems.* Englewood Cliffs, N.J. 07632: Prentice Hall, 1987.

3. A. Cohn, "A Proof of Correctness of the VIPER Microprocessor: the First Level," *VLSI Specification, Verification, and synthesis, G. Birtwhistle and P. S ubrahmanyam, eds.*, 1988.

4. A. Cohn, "A Proof of Correctness of the VIPER Microprocessor: the Second Level," *University of Cambridge computer Laboratory Technical Report*, 1989.

5. J. Joyce, "Formal Specification and Verification of Microprocessor Systems," *Microprocessing and Microprogramming, North-Holland*, (24)1988.

6. J. Joyce, "Using Higher-Order Logic to Specify Computer Hardware and Architecture," in *Proceedings of the IFIP TC10 Working Conference on Design Methodology in VLSI and Computer Architecture*, 1988.

7. J. Joyce, "Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic," technical report, University of Cambridge, Computer Laboratory, 1988.

8. W. A. Hunt, "A Verified Microprocessor," *Technical Report 47, The University of Texas at Austin*, Dec., 1985.

9. G. Milne and P. Subrahmanyam, *Formal Aspect of VLSI Design.* Publishers B.V., 1986.

10. P. J. Windley, "A Hierarchical Methodology for Verifying Microprogrammed Microprocessors," *Proceedings on IEEE Computer Society Symposium on Research in Security and Privacy*, 1990.

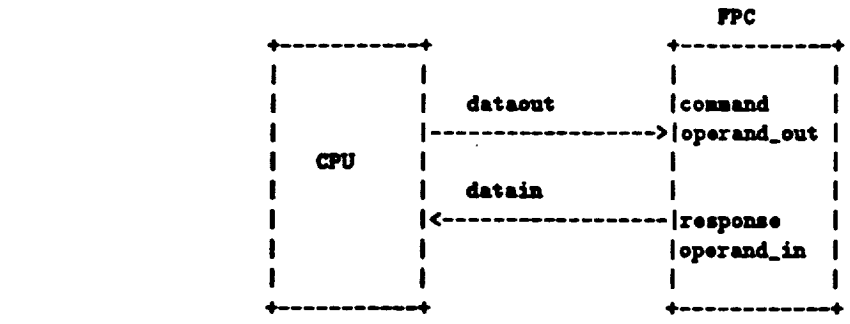
11. J. Joyce, *Multi-Level Verification of Microprocessor-Based Systems.* PhD thesis, Cambridge University, December 1989.

12. P. J. Windley, "The Formal Verification of Generic Interpreters," *Ph.D Thesis*, 1990.

13. T. Melham, "Abstraction Mechanisms for Hardware Verification," in *VLSI specification, Verification and Synthesis, Proceeding of the Workshop on Hardware Verification, Calgary, Canada, 1987*, (G. Birtwistle and P. Subrahmanyam, eds.), (Boston), Kluwer Academic Publisher, 1988.

APPENDIX A: SPECIFICATION OF THE INTERPRETERS

```
%-------------------------------------------------------------
File: cpu_service.ml
Author: Jing Pan
Date: March 1991
Purpose:        The CPU part concerning communication between
-               the CPU and the FPU.
-------------------------------------------------------------%


loadf '/csgrad/panj/holdir/init.ml';;

loadf 'abstract.ml';;

loadf 'tactics.ml';;
system '/bin/rm -f cpu_service.th';;

set_flag ('sticky', true);;

new_theory 'cpu_service';;

loadf 'aux_defs.ml';;

map new_parent ['aux'; 'interface'];;

autoload_defs_and_thms 'aux';;

let rep_ty = abstract_type 'interface' 'fetch';;




%-----------------------------------------------
  state = (c_ac, c_reg, mem, ir, dataout, address, read, write,
           cpu_state)
  env = (resp_ready, datain)
-----------------------------------------------%



%-----------------------------------------------
                cir_read_write
    correctness property of the 4-phase handshanking
  protocol.

  the address of the cirs:
      command          0
      response         1
      operand_in       2
      operand_out      3
      condition        4
      control          5

  datain :  cpu <-- biu
  dataout : cpu --> biu
```




```
Note setting new_instr and operand_ready is instant!!
-----------------------------------------------------%
let cir_read_write = new_definition
 ('cir_read_write',
 "! (rep:^rep_ty) (address:num->num) (read write :num-> bool)
    (datain dataout:num->*wordn) (command:num->num) (response:num->num)
    (operand_in:num->fp) (operand_out:num->fp)
    (condition control:num->num) (operand_ready:num->bool)
    (new_instr:num->bool) .
    cir_read_write rep address read write datain dataout
                   command response operand_in operand_out condition
                   control new_instr operand_ready =
  ! t.
  (read t) =>
     (address t =1) => ((datain(t+1) = (numtow rep (response t))) /\
       ~(new_instr t) /\
                       ~(operand_ready t)) |
                       ((datain(t+1) = (fptow rep (operand_in t))) /\
                       ~(new_instr t) /\
                       ~(operand_ready t)) |
  (write t) =>
     ((address t =0) => (((command(t+1))= (wtonum rep (dataout t))) /\
                         (new_instr t = T)) |
                         ((operand_out(t+1) = (wtofp rep (dataout t))) /\
                         (operand_ready t = F) /\
                         (operand_ready (t+1) = T))) |
   % No read or write %
                         ((new_instr t = F) /\
                         (operand_ready t = F))"
);;




%-------------------------------------------------
       Define the state transition

 CPU_service stays (wait) at every state until
something happens to prompt it to go to another
state.
-------------------------------------------------%


%0%
```

```
%-----------------------------------------------
                cpu_begin

  put the instr into the command cir, goto 1

  Since all the CIR's are resided inside the biu,
any read or write by the cpu from or to the CIRs are
implemented by a four phase handshaking protocol

  CPU writes to the BIU cirs, therefore "write = T"
and "read = F". We have to specify read being false
since last time read might be true. Then read (t+1) = read t
will make read true all the time.
-----------------------------------------------%
let cpu_begin = new_definition
  ('cpu_begin',
  "! (rep: ^rep_ty) (c_ac:num) (c_reg:num->num) (mem:*memory) (ir:num)
     (dataout : *wordn) (address : num) (read write : bool)
     (cpu_state: num)
     (resp_ready:bool) (datain : *wordn) (n:num) .
  cpu_begin n rep (c_ac, c_reg, mem, ir, dataout, address, read, write,
           cpu_state) (resp_ready, datain) =
       (c_ac, c_reg, mem, ir, (numtow rep ir), 0, F, T, 1)"
);;



%1%
%-----------------------------------------------
            cpu_wait_for_response

 wait for the response from biu to come

 note since everything is completely asynchronous,
the cpu has wait for possibly several cycles
before the biu response is ready (resp_ready = T)
-----------------------------------------------%
let cpu_wait_for_response = new_definition
  ('cpu_wait_for_response',
 "! (rep:^rep_ty) c_ac c_reg mem ir dataout address read write
    cpu_state resp_ready datain n .
  cpu_wait_for_response n rep (c_ac, c_reg, mem, ir, dataout, address,
                       read, write, cpu_state) (resp_ready, datain) =
  (^resp_ready) =>
     (c_ac, c_reg, mem, ir, dataout, address, F, F, 1) |
     (c_ac, c_reg, mem, ir, dataout, 1, T, F, 2)"
);;


%2%
%-----------------------------------------------
            cpu_wait_4phase

   the CPU waits once cycle for the 4phase protocol
 finish reading the response (put the response on
 the datain bus)
-----------------------------------------------%
let cpu_wait_4phase = new_definition
```

```
  ('cpu_wait_4phase',
 "! (rep:^rep_ty) c_ac c_reg mem ir dataout address read write
    cpu_state resp_ready datain n .
  cpu_wait_4phase n rep (c_ac, c_reg, mem, ir, dataout, address,
                         read, write, cpu_state) (resp_ready, datain) =
     (c_ac, c_reg, mem, ir, dataout, address, F, F, 3)"
);;


%3%
%-------------------------------------------------
              cpu_read_response

   if the response is to transfer data into fpc
      then goto 2
   if the response is to transfer data out of fpc
      then got 4
      else (null primitive) goto 0
-------------------------------------------------%
let cpu_read_response = new_definition
  ('cpu_read_response',
 "! (rep:^rep_ty) c_ac c_reg mem ir dataout address read write
    cpu_state resp_ready datain n .
  cpu_read_response n rep (c_ac, c_reg, mem, ir, dataout, address,
                   read, write, cpu_state) (resp_ready, datain) =
  let data = (fetch rep) mem (Addr n ir) in
   (((wtonum rep datain) = read_p) =>  %fld%
       (c_ac, c_reg, mem, ir, data, 2, F, T, 0) |
    ((wtonum rep datain) = write_p) =>  %fstore%
       (c_ac, c_reg, mem, ir, dataout, 2, T, F, 4) |
    ((wtonum rep datain) = wait_p) =>
       (c_ac, c_reg, mem, ir, dataout, address, F, F, 1) |
       (c_ac, c_reg, mem, ir, dataout, address, F, F, 0))"
);;



%4%
%-------------------------------------------------
                 cpu_wait_read

  CPU wait one cycle for the 4-phase to finish reading
-------------------------------------------------%
let cpu_wait_read = new_definition
 ('cpu_wait_read',
 "! (rep:^rep_ty) c_ac c_reg mem ir dataout address read write
    cpu_state resp_ready datain n .
  cpu_wait_read n rep (c_ac, c_reg, mem, ir, dataout, address,
                   read, write, cpu_state) (resp_ready, datain) =
      (c_ac, c_reg, mem, ir, dataout, address, F, F, 5)"
);;



%5%
%-------------------------------------------------
```

```
                cpu_put_data

   write the data in the operand_out cir to the memory,
and goto 0.
-------------------------------------------------%
let cpu_put_data = new_definition
 ('cpu_put_data',
 "! (rep:^rep_ty) c_ac c_reg mem ir dataout address read write
    cpu_state resp_ready datain n .
  cpu_put_data n rep (c_ac, c_reg, mem, ir, dataout, address,
                      read, write, cpu_state) (resp_ready, datain) =
  let new_mem = (store rep) mem (Addr n ir) datain in
       (c_ac, c_reg, new_mem, ir, dataout, address, F, F, 0)"
);;



%-----------------------------------------------
              cpu_service_state

    Next state function for CPU service.
-------------------------------------------------%
let cpu_service_state = new_definition
  ('cpu_service_state',
  "! (rep:^rep_ty) (n : num) (cpu_state:num) .
  cpu_service_state n rep cpu_state =
     ((cpu_state = 0) => (cpu_begin n rep) |
      (cpu_state = 1) => (cpu_wait_for_response n rep) |
      (cpu_state = 2) => (cpu_wait_4phase n rep) |
      (cpu_state = 3) => (cpu_read_response n rep) |
      (cpu_state = 4) => (cpu_wait_read n rep) |
                         (cpu_put_data n rep))"
);;



%-----------------------------------------------------
                  cpu_service
   The top level description for CPU service.
-----------------------------------------------------%
let cpu_service = new_definition
  ('cpu_service',
  "! (rep: ^rep_ty) (c_ac:num->num) (c_reg:num->num->num)
     (mem:num->*memory)
     (ir:num->num) (dataout :num->*wordn) (address :num->num)
     (read write :num->bool) (cpu_state:num->num)
     (resp_ready:num->bool) (datain :num->*wordn) (n:num) .
  cpu_service n rep (c_ac, c_reg, mem, ir, dataout, address, read, write,
            cpu_state) (resp_ready, datain) =
  ! t. (c_ac(t+1), c_reg(t+1), mem(t+1), ir(t+1), dataout(t+1),
        address(t+1), read(t+1), write(t+1), cpu_state(t+1)) =
    cpu_service_state n rep (cpu_state t)
        (c_ac t, c_reg t, mem t, ir t, dataout t, address t, read t,
         write t, cpu_state t)
        (resp_ready t, datain t)"
);;
```

```
close_theory();;
quit();;
```

```
%-------------------------------------------------------------
File: biu_top.ml
Author: Jing Pan
Date: March 1991
Purpose:        The top level spec of the BIU, view it as a
                state transition machine.
-------------------------------------------------------------%


loadf '/csgrad/panj/holdir/init.ml';;
loadf 'abstract.ml';;

system '/bin/rm -f biu_top.th';;

set_flag ('sticky', true);;

new_theory 'biu_top';;

loadf 'aux_defs.ml';;

map new_parent ['aux'; 'interface'; 'fptype'];;

autoload_defs_and_thms 'fptype';;

let rep_ty = abstract_type 'interface' 'fetch';;


%-----------------------------------------------
state = (response, operand_out, decode_reg, resp_ready, f_ac,
         biu_state, start)
env = (command, condition, control, operand_in, done, new_instr,
       operand_ready)
-----------------------------------------------%



%0%
%-------------------------------------------------------
                    biu_idle

  if ~new_instr then busy wait
         else set fpc status to busy, and goto 1
-------------------------------------------------------%
let biu_idle = new_definition
  ('biu_idle',
  "! (rep: ^rep_ty) (response: num) (operand_out: fp)
     (decode_reg: num) (resp_ready:bool) (biu_state: num) (start:bool)
     (command: num) (condition: num) (control: num)
     (operand_in: fp) (done: bool) (new_instr:bool) (f_ac:fp)
```

```
     (operand_ready:bool) (n:num).
    biu_idle n rep (response, operand_out, decode_reg, resp_ready, f_ac,
             biu_state, start) (command, condition, control, operand_in,
             done, new_instr, operand_ready) =
    (~new_instr) =>
         (response, operand_out, decode_reg, resp_ready, f_ac, 0, start) |
         (response, operand_out, decode_reg, resp_ready, f_ac, 1, start)"
);;


%1%
%-------------------------------------------------------
                    biu_decode

  BIU decodes the instruction and send back the response.
In the case of FSTR, it also send back the data.
-------------------------------------------------------%
let biu_decode = new_definition
  ('biu_decode',
"! (rep:^rep_ty) response operand_out  decode_reg resp_ready biu_state
    command condition control operand_in done new_instr f_ac operand_ready
    n .
  biu_decode n rep (response, operand_out, decode_reg, resp_ready, f_ac,
             biu_state, start) (command, condition, control, operand_in,
             done, new_instr, operand_ready) =
      (need_read n command) =>  %fld%
             (read_p, operand_out, decode_reg, T, f_ac, 2, start) |
      (need_write n command) =>  %fstore%
             (write_p, f_ac, decode_reg, T, f_ac, 0, start) |
             (null_p, operand_out, command, T, f_ac, 4, T)"  %fadd%
);;


%2%
%-------------------------------------------------------
                  biu_wait_op
  The BIU waits one cycle
-------------------------------------------------------%
let biu_wait_op = new_definition
 ('biu_wait_op',
"! (rep:^rep_ty) response operand_out  decode_reg resp_ready biu_state
    command condition control operand_in done new_instr f_ac operand_ready
    n .
  biu_wait_op n rep (response, operand_out, decode_reg, resp_ready, f_ac,
       biu_state, start) (command, condition, control, operand_in,
       done, new_instr, operand_ready) =
     (response, operand_out, decode_reg, resp_ready, f_ac, 3, start)"
);;


%3%
%-------------------------------------------------------
                  biu_fld
-------------------------------------------------------%
let biu_fld = new_definition
  ('biu_fld',
```

```
"! (rep:^rep_ty) response operand_out  decode_reg resp_ready biu_state
    command condition control operand_in done new_instr f_ac operand_ready
    n .
  biu_fld n rep (response, operand_out, decode_reg, resp_ready, f_ac,
       biu_state, start) (command, condition, control, operand_in,
       done, new_instr, operand_ready) =
  (^operand_ready) =>
     (response, operand_out, decode_reg, resp_ready, f_ac, 3, start) |
     (response, operand_out, decode_reg, resp_ready, operand_out, 0, start)"
);;



%4%
%-------------------------------------------------------
                 biu_wait_apu

  if apu is not done with the current instr.
              then wait
              else set status to idle and goto 0
-------------------------------------------------------%
let biu_wait_apu = new_definition
 ('biu_wait_apu',
"! (rep:^rep_ty) response operand_out  decode_reg biu_state
    command condition control operand_in done new_instr f_ac operand_ready
    n .
  biu_wait_apu n rep (response, operand_out, decode_reg, resp_ready, f_ac,
             biu_state, start) (command, condition, control, operand_in,
             done, new_instr, operand_ready) =
  (done) =>
     (response, operand_out, decode_reg, resp_ready, f_ac, 0, start) |
     (new_instr =>
        (wait_p, operand_out, decode_reg, resp_ready, f_ac, 4, start) |
        (response, operand_out, decode_reg, resp_ready, f_ac, 4, start))"

);;



%-------------------------------------------------------
                 biu_top_state
-------------------------------------------------------%
let biu_top_state = new_definition
  ('biu_top_state',
  "! (rep:^rep_ty) (n : num) (biu_state:num) .
  biu_top_state n rep biu_state =
     (biu_state = 0) => (biu_idle n rep) |
     (biu_state = 1) => (biu_decode n rep) |
     (biu_state = 2) => (biu_wait_op n rep) |
     (biu_state = 3) => (biu_fld n rep) |
                        (biu_wait_apu n rep)"
);;



%-----------------------------------------------
                 biu_top
```

```
  BIU top level behavior
-----------------------------------------------%

let biu_top = new_definition
 ('biu_top',
 "! (rep : ^rep_ty) (respnse:num->num) (operand_out:num->fp)
    (decode_reg:num->num) (resp_ready:num->bool) (biu_state:num->num)
     (start : num->bool)
     (command:num->num) (condition:num->num) (control:num->num)
     (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (f_ac:num->fp) (operand_ready:num->bool)
     (n:num) .
  biu_top n rep  (response, operand_out, decode_reg, resp_ready, f_ac,
          biu_state, start)
         (command, condition, control, operand_in, done, new_instr,
          operand_ready) =
  ! t. (response(t+1), operand_out(t+1),
       decode_reg(t+1), resp_ready(t+1), f_ac(t+1), biu_state(t+1),
       start(t+1)) =
    biu_top_state n rep (biu_state t) (response t, operand_out t,
          decode_reg t, resp_ready t, f_ac t, biu_state t, start t)
       (command t, condition t, control t, operand_in t, done t,
        new_instr t, operand_ready t)"
);;



%-----------------------------------------------
                 biu_top_cpu

  BIU top level behavior, CPU's point of view
-----------------------------------------------%
let biu_top_cpu = new_definition
  ('biu_top_cpu',
 "! (rep : ^rep_ty) (response:num->num) (operand_out:num->fp)
    (decode_reg:num->num) (resp_ready:num->bool) (biu_state:num->num)
     (start : num->bool)
     (command:num->num) (condition:num->num) (control:num->num)
     (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (f_ac:num->fp) (operand_ready:num->bool)
     (n:num) .
  biu_top_cpu n rep (response, operand_out, decode_reg, resp_ready, f_ac,
          biu_state, start)
         (command, condition, control, operand_in, done, new_instr,
          operand_ready) =
  ! (t:num). ? (t':num). ((t+1)<=t') /\
  (let state_tuple = (response t, operand_out t, decode_reg t, resp_ready t,
                      f_ac t, biu_state t, start t) and
      state_tuplet1 = (response(t+1), operand_out(t+1), decode_reg(t+1),
                resp_ready(t+1), f_ac(t+1), biu_state(t+1), start(t+1)) and
      env_tuple = (command t, condition t, control t, operand_in t, done t,
                   new_instr t, operand_ready t) and
      state_tuplet' = (response t', operand_out t', decode_reg t',
                       resp_ready t', f_ac t', biu_state t', start t') in
      (((biu_state t = 0) \/ (biu_state t = 2)) =>
           (state_tuplet1 = biu_top_state n rep (biu_state t)
                                          state_tuple env_tuple) |
       (biu_state t = 1) =>
```

```
          ((First (t, t') (resp_ready)) /\
           (Stable (start, F, t, t')) /\
           (state_tuplet' = (biu_decode n rep state_tuple env_tuple))) |
        (biu_state t = 3) =>
          ((~(operand_ready t)) =>
                (state_tuplet1 = (response t, operand_out t, decode_reg t,
                 resp_ready t, f_ac t, 3, start t)) |
                ((Stable (start, F, t, t')) /\
                 (state_tuplet' = (response t, operand_out t, decode_reg t,
                   resp_ready t, operand_out t, 0, start t)))) |
          (state_tuplet' = (biu_top_state n rep (biu_state t)
                                          state_tuple env_tuple))))"
);;


close_theory();;

quit();;
```

```
%-------------------------------------------------------------
File: apu_top.ml
Author: Jing Pan
Date: March 1991
Purpose:        The top level description of the APU
-------------------------------------------------------------%


loadf '/csgrad/panj/holdir/init.ml';;

loadf 'abstract.ml';;

system '/bin/rm -f apu_top.th';;

set_flag ('sticky', true);;

new_theory 'apu_top';;

loadf 'aux_defs.ml';;

map new_parent ['aux'; 'interface'; 'fptype'];;

autoload_defs_and_thms 'aux';;
autoload_defs_and_thms 'fptype';;


let rep_ty = abstract_type 'interface' 'fetch';;
```

```
%-------------------------------------------------
  state on the top level for APU:

     (f_ac, f_reg, cw, sw, done)

  env on the micro level for MEU:
     (start, decode_reg, operand_in)

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



%-------------------------------------------------
Arithmetic Instruction
  perform arthmetic operation on single or double
precision floating point (real) numbers:

  FADD, FSUB, FMUL, FDIV, FREM

These operations are entirely internal to the coprocessor.
-------------------------------------------------%



let n1 = new_definition
  ('n1', "n1 = 8");;

let n2 = new_definition
  ('n2', "n2 = 23");;



%-------------------------------------------------
C_FADD

  The floating point add instruction, where n is the
instruction length
         (f_ac) + (f_reg i) -> f_ac
-------------------------------------------------%
let C_FADD = new_definition
  ('C_FADD',
  "! (rep : ^rep_ty) (f_ac : fp) (f_reg : num->fp)
     (cw sw :bool#bool#bool#bool) (done : bool)
     (start :bool) (decode_reg : num) (n:num) .
    C_FADD n rep (f_ac, f_reg, cw, sw, done)
               (start, decode_reg) =
    let (addr:num) = (Addr n decode_reg) in
          ((FST (FP_ADD n1 n2 f_ac (f_reg addr))), f_reg,
             cw, sw, T)"
);;



%-------------------------------------------------
C_FSUB
```

```
  The floating point sub instruction, where n is the
instruction length
         (f_ac) - (f_reg i) -> f_ac
-------------------------------------------------------%
let C_FSUB = new_definition
  ('C_FSUB',
  "! (rep : ^rep_ty) f_ac f_reg cw sw done start
     decode_reg n .
    C_FSUB n rep (f_ac, f_reg, cw, sw, done)
                 (start, decode_reg) =
    let (addr:num) = (Addr n decode_reg) in
((PST (FP_SUB n1 n2 f_ac (f_reg addr))), f_reg,
          cw, sw, T)"
);;



%-------------------------------------------------
C_FMUL

  The floating point mul instruction, where n is the
instruction length
         (f_ac) * (f_reg i) -> f_ac
-------------------------------------------------------%
let C_FMUL = new_definition
  ('C_FMUL',
  "! (rep : ^rep_ty) f_ac f_reg cw sw done start
     decode_reg n .
    C_FMUL n rep (f_ac, f_reg, cw, sw, done)
                 (start, decode_reg) =
    let (addr:num) = (Addr n decode_reg) in
   ((PST (FP_MUL n1 n2 f_ac (f_reg addr))), f_reg,
             cw, sw, T)"
);;



%-------------------------------------------------
C_FDIV

  The floating point div instruction, where n is the
instruction length
         (f_ac) / (f_reg i) -> f_ac
-------------------------------------------------------%
let C_FDIV = new_definition
  ('C_FDIV',
  "! (rep : ^rep_ty) f_ac f_reg cw sw done start
     decode_reg n .
    C_FDIV n rep (f_ac, f_reg, cw, sw, done)
                 (start, decode_reg) =
    let (addr:num) = (Addr n decode_reg) in
          ((PST (FP_DIV n1 n2 f_ac (f_reg addr))), f_reg,
             cw, sw, T)"
);;
```

```
%-----------------------------------------------
                  apu_idle

    APU wait for the BIU to pass an instr
-----------------------------------------------%
let apu_idle = new_definition
  ('apu_idle',
 "! (rep : ^rep_ty) f_ac f_reg cw sw done start
     decode_reg n .
    apu_idle n rep (f_ac, f_reg, cw, sw, done)
               (start, decode_reg) =
        (f_ac, f_reg, cw, sw, done)"
);;



%-----------------------------------------------
NextState_apu

  Define the next state of APU top level
-----------------------------------------------%
let NextState_apu = new_definition
  ('NextState_apu',
 "! (rep : ^rep_ty) (opcode:num) n .
    NextState_apu n rep opcode =
 %     ((opcode = 0) => (C_FLD n rep) |
        (opcode = 1) => (C_FSTR n rep) | %
       ((opcode = 2) => (C_FADD n rep) |
        (opcode = 3) => (C_FSUB n rep) |
        (opcode = 4) => (C_FMUL n rep) |
                        (C_FDIV n rep))"
);;



%-----------------------------------------------
apu_top

  The top level description of APU.
-----------------------------------------------%
let apu_top = new_definition
  ('apu_top',
 "! (rep : ^rep_ty) (f_ac : num->fp) (f_reg : num->num->fp)
    (cw sw : num->bool#bool#bool#bool)
    (done : num->bool) (start : num->bool) (decode_reg : num->num)
    (n:num) .
  apu_top n rep (f_ac, f_reg, cw, sw, done)
          (start, decode_reg) =
    ! t . (f_ac(t+1), f_reg(t+1), cw(t+1), sw(t+1),
           done(t+1)) =
    (^start t) =>
       apu_idle  n rep (f_ac t, f_reg t, cw t, sw t,
              done t) (start t, decode_reg t)  |
       NextState_apu n rep (Opc n (decode_reg t))
              (f_ac t, f_reg t, cw t, sw t, done t)
              (start t, decode_reg t)"
);;
```

```
%-------------------------------------------------
apu_top_cpu

  The top level description of APU, from the CPU
point of view.
-------------------------------------------------%
let apu_top_cpu = new_definition
  ('apu_top_cpu',
 "! (rep : ^rep_ty) (f_ac : num->fp) (f_reg : num->num->fp)
    (cw sw : num->bool#bool#bool#bool)
    (done : num->bool) (start : num->bool) (decode_reg : num->num)
    (n:num) .
  apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
          (start, decode_reg) =
    ! t . (^start t) =>
    ((f_ac(t+1), f_reg(t+1), cw(t+1), sw(t+1), done(t+1)) =
         apu_idle n rep (f_ac t, f_reg t, cw t, sw t, done t)
                  (start t, decode_reg t)) |
     (? t' . (t<=t') /\
              (First (t, t') (done)) /\
       ((f_ac t', f_reg t', cw t', sw t', done t') =
       NextState_apu n rep (Opc n (decode_reg t))
                  (f_ac t, f_reg t, cw t, sw t, done t)
                  (start t, decode_reg t)))"
);;


close_theory();;
quit();;
```

APPENDIX B: USEFUL LEMMAS

```
%-------------------------------------------------------------
File: service_lemma.ml
Author: Jing Pan
Date: March 1991
Purpose: Lemmas for "cpu_service".
-------------------------------------------------------------%


loadf '/csgrad/panj/holdir/init.ml';;

loadf 'abstract.ml';;

system '/bin/rm -f service_lemma.th';;

set_flag ('sticky', true);;

new_theory 'service_lemma';;

loadf 'aux_defs.ml';;

map new_parent ['cpu_service'];;

autoload_defs_and_thms 'cpu_service';;

let rep_ty = abstract_type 'interface' 'fetch';;

map loadf ['digit'; 'decimal'];;



%-------------------------------------------------------------
             cpu_service_lemma
-------------------------------------------------------------%
let cpu_service_lemma = prove_thm
 ('cpu_service_lemma',
 "! rep:^rep_ty (n : num) .
  ((cpu_service_state n rep 0 = (cpu_begin n rep)) /\
   (cpu_service_state n rep 1 = (cpu_wait_for_response n rep)) /\
   (cpu_service_state n rep 2 = (cpu_wait_4phase n rep)) /\
   (cpu_service_state n rep 3 = (cpu_read_response n rep)) /\
   (cpu_service_state n rep 4 = (cpu_wait_read n rep)) /\
   (cpu_service_state n rep 5 = (cpu_put_data n rep)))",
  REPEAT GEN_TAC
  THEN ONCE_REWRITE_TAC [cpu_service_state]
  THEN DEC_EQ_TAC
);;



let MICRO_RULE address instr =
  (GEN_ALL (DISCH_ALL
   (GEN "t"
    (DISCH "(cpu_state t = ^address)"
     (REWRITE_RULE [cpu_service_lemma; instr]
      (SUBS [ASSUME "(cpu_state t = ^address)"]
```

```
        (SPEC_ALL
         (REWRITE_RULE [cpu_service]
          (ASSUME "cpu_service n (rep:^rep_ty) (c_ac, c_reg, mem, ir,
                                  dataout, address, read, write, cpu_state)
                                  (resp_ready, datain)"
))))))))));;



%----------------------------------------
  One lemma for each individual state
----------------------------------------%

let cpu_begin_lemma = save_thm
  ('cpu_begin_lemma',
    MICRO_RULE "0" cpu_begin
);;


let cpu_wait_for_response_lemma = save_thm
  ('cpu_wait_for_response_lemma',
    MICRO_RULE "1" cpu_wait_for_response
);;


let cpu_wait_4phase_lemma = save_thm
  ('cpu_wait_4phase_lemma',
    MICRO_RULE "2" cpu_wait_4phase
);;


let cpu_read_response_lemma = save_thm
  ('cpu_read_response_lemma',
    MICRO_RULE "3" cpu_read_response
);;


let cpu_wait_read_lemma = save_thm
  ('cpu_wait_read_lemma',
   MICRO_RULE "4" cpu_wait_read
);;


let cpu_put_data_lemma = save_thm
  ('cpu_put_data_lemma',
   MICRO_RULE "5" cpu_put_data
);;



close_theory();;
quit();;
```

```
%-------------------------------------------------------------
File: biu_lemma.ml
Author: Jing Pan
Date: March 1991
Purpose:        A group of lemmas aabout the top level spec
                of the BIU.
-------------------------------------------------------------%


loadf '/csgrad/panj/holdir/init.ml';;
loadf 'abstract.ml';;
loadf 'tactics.ml';;

system '/bin/rm -f biu_lemma.th';;

set_flag ('sticky', true);;

new_theory 'biu_lemma';;

loadf 'aux_defs.ml';;

map new_parent ['aux'; 'interface'; 'fptype'; 'biu_top'];;

autoload_defs_and_thms 'fptype';;
autoload_defs_and_thms 'biu_top';;

let rep_ty = abstract_type 'interface' 'fetch';;


map loadf ['digit'; 'decimal'];;



%-------------------------------------------
              biu_top_lemma
-------------------------------------------%
let biu_top_lemma = prove_thm
 ('biu_top_lemma',
 "! rep:*rep_ty (n : num) .
  ((biu_top_state n rep 0 = (biu_idle n rep)) /\
   (biu_top_state n rep 1 = (biu_decode n rep)) /\
   (biu_top_state n rep 2 = (biu_wait_op n rep)) /\
   (biu_top_state n rep 3 = (biu_fld n rep)) /\
   (biu_top_state n rep 4 = (biu_wait_apu n rep)))",
  REPEAT GEN_TAC
  THEN ONCE_REWRITE_TAC [biu_top_state]
  THEN DEC_EQ_TAC
);;



let TAC1 =
  REPEAT STRIP_TAC
  THEN REWRITE_ASM_THM_TAC biu_top_cpu 2
  THEN ASSUM_LIST (\ths. ASSUME_TAC
       (SPEC "t" (el 1 ths)))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC LET_DEF 1
  THEN RM2LAST_TAC
```

```
  THEN POP_ASSUM (\th1. ASSUME_TAC
        (BETA_RULE th1))
% RES_TAC -- resolution only handles implication, not conditions %
  THEN REWRITE_ASM 2 1
  THEN POP_ASSUM (\th1. ASSUME_TAC
        ((CONV_RULE DEC_EQ_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
        (CONV_RULE (ONCE_DEPTH_CONV INV_dec_CONV) th1))
  THEN REWRITE_ASM_THM_TAC biu_top_lemma 1
  THEN ASM_REWRITE_TAC[];;


let state1_TAC =
  REPEAT STRIP_TAC
  THEN REWRITE_ASM_THM_TAC biu_top_cpu 2
  THEN ASSUM_LIST (\ths. ASSUME_TAC
       (SPEC "t" (el 1 ths)))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC LET_DEF 1
  THEN RM2LAST_TAC
  THEN POP_ASSUM (\th1. ASSUME_TAC
        (BETA_RULE th1))
% RES_TAC -- resolution only handles implication, not conditions %
  THEN REWRITE_ASM 2 1
  THEN POP_ASSUM (\th1. ASSUME_TAC
        ((CONV_RULE DEC_EQ_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
        (CONV_RULE (ONCE_DEPTH_CONV INV_dec_CONV) th1))
  THEN ASM_REWRITE_TAC[];;



%-----------------------------------------
  One lemma for each individual state
-----------------------------------------%


%-----------------------------------------
      biu_decode_lemma
-----------------------------------------%
let biu_decode_lemma = prove_thm
 ('biu_decode_lemma',
  "! (rep: ^rep_ty) (response: num->num) (operand_out: num->fp)
     (decode_reg:num->num) (resp_ready:num->bool) (biu_state:num->num)
     (start:num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr:num->bool) (f_ac:num->fp) (operand_ready:num->bool) (n:num).
    biu_top_cpu n rep (response, operand_out, decode_reg, resp_ready, f_ac,
             biu_state, start) (command, condition, control, operand_in,
             done, new_instr, operand_ready) ==>
    (!t.
       (biu_state t = 1) ==>
         ? t'. ((t+1)<=t') /\
               (First (t, t') (resp_ready)) /\
               (Stable (start, F, t, t')) /\
         ((response t', operand_out t', decode_reg t', resp_ready t', f_ac t',
          biu_state t', start t') =
          biu_decode n rep (response t, operand_out t, decode_reg t,
                    resp_ready t, f_ac t, biu_state t, start t)
```

```
                    (command t, condition t, control t, operand_in t,
                    done t, new_instr t, operand_ready t)))",
   state1_TAC);;


%-----------------------------------------
      biu_fld_lemma
-----------------------------------------%
let biu_fld_lemma = prove_thm
 ('biu_fld_lemma',
  "! (rep: ^rep_ty) (response: num->num) (operand_out: num->fp)
     (decode_reg:num->num) (resp_ready:num->bool) (biu_state:num->num)
     (start:num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr:num->bool) (f_ac:num->fp) (operand_ready:num->bool) (n:num).
    biu_top_cpu n rep (response, operand_out, decode_reg, resp_ready, f_ac,
             biu_state, start) (command, condition, control, operand_in,
             done, new_instr, operand_ready) ==>
    (!t.
      (biu_state t = 3) ==>
        ? t'. ((t+1)<=t') /\
        ((~(operand_ready t)) =>
           ((response(t+1), operand_out(t+1), decode_reg(t+1),
             resp_ready(t+1), f_ac(t+1), biu_state(t+1), start(t+1)) =
           (response t, operand_out t, decode_reg t, resp_ready t, f_ac t,
            3, start t)) |
            ((Stable (start, F, t, t')) /\
            (((response t', operand_out t', decode_reg t', resp_ready t',
                 f_ac t', biu_state t', start t') =
                (response t, operand_out t, decode_reg t, resp_ready t,
                 operand_out t, 0, start t))))))",
  REPEAT STRIP_TAC
  THEN REWRITE_ASM_THM_TAC biu_top_cpu 2
  THEN ASSUM_LIST (\ths. ASSUME_TAC
       (SPEC "t" (el 1 ths)))
  THEN POP_ASSUM (\th1.
         STRIP_ASSUME_TAC th1)
  THEN RM3LAST_TAC
  THEN REWRITE_ASM_THM_TAC LET_DEF 1
  THEN RM2LAST_TAC
  THEN POP_ASSUM (\th1. ASSUME_TAC
         (BETA_RULE th1))
% RES_TAC -- resolution only handles implication, not conditions %
  THEN REWRITE_ASM 3 1
  THEN POP_ASSUM (\th1. ASSUME_TAC
         ((CONV_RULE DEC_EQ_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
         (CONV_RULE (ONCE_DEPTH_CONV INV_dec_CONV) th1))
  THEN EXISTS_TAC "t':num"
  THEN ASM_REWRITE_TAC□
);;



let TAC2 =
  REPEAT STRIP_TAC
```

```
  THEN REWRITE_ASM_THM_TAC biu_top_cpu 2
  THEN ASSUM_LIST (\ths. ASSUME_TAC
       (SPEC "t" (el 1 ths)))
  THEN POP_ASSUM (\th1.
         STRIP_ASSUME_TAC th1)
  THEN RM3LAST_TAC
  THEN REWRITE_ASM_THM_TAC LET_DEF 1
  THEN RM2LAST_TAC
  THEN POP_ASSUM (\th1. ASSUME_TAC
         (BETA_RULE th1))
% RES_TAC -- resolution only handles implication, not conditions %
  THEN REWRITE_ASM 3 1
  THEN POP_ASSUM (\th1. ASSUME_TAC
         ((CONV_RULE DEC_EQ_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
         (CONV_RULE (ONCE_DEPTH_CONV INV_dec_CONV) th1))
  THEN REWRITE_ASM_THM_TAC biu_top_lemma 1
  THEN ASM_REWRITE_TAC[];;


%-----------------------------------------
      biu_wait_apu_lemma
-----------------------------------------%
let biu_wait_apu_lemma = prove_thm
 ('biu_wait_apu_lemma',
  "! (rep: *rep_ty) (response: num->num) (operand_out: num->fp)
     (decode_reg:num->num) (resp_ready:num->bool) (biu_state:num->num)
     (start:num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr:num->bool) (f_ac:num->fp) (operand_ready:num->bool) (n:num).
    biu_top_cpu n rep (response, operand_out, decode_reg, resp_ready, f_ac,
             biu_state, start) (command, condition, control, operand_in,
             done, new_instr, operand_ready) ==>
      (!t.
       (biu_state t = 4) ==>
         ? t'. ((t+1)<=t') /\
         ((response t', operand_out t', decode_reg t', resp_ready t', f_ac t',
          biu_state t', start t') =
          biu_wait_apu n rep (response t, operand_out t, decode_reg t,
                     resp_ready t, f_ac t, biu_state t, start t)
                     (command t, condition t, control t, operand_in t,
                     done t, new_instr t, operand_ready t)))",
  TAC1
);;


%-----------------------------------------
      biu_wait_op_lemma
-----------------------------------------%
let biu_wait_op_lemma = prove_thm
 ('biu_wait_op_lemma',
  "! (rep: *rep_ty) (response: num->num) (operand_out: num->fp)
     (decode_reg:num->num) (resp_ready:num->bool) (biu_state:num->num)
     (start:num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr:num->bool) (f_ac:num->fp) (operand_ready:num->bool) (n:num).
    biu_top_cpu n rep (response, operand_out, decode_reg, resp_ready, f_ac,
```

```
            biu_state, start) (command, condition, control, operand_in,
            done, new_instr, operand_ready) ==>
        (!t.
       (biu_state t = 2) ==>
         ((response(t+1), operand_out(t+1), decode_reg(t+1),
           resp_ready(t+1), f_ac(t+1), biu_state(t+1), start(t+1)) =
          biu_wait_op n rep (response t, operand_out t, decode_reg t,
                     resp_ready t, f_ac t, biu_state t, start t)
                     (command t, condition t, control t, operand_in t,
                     done t, new_instr t, operand_ready t)))",
    TAC2
);;


%----------------------------------------
      biu_idle_lemma
----------------------------------------%
let biu_idle_lemma = prove_thm
 ('biu_idle_lemma',
  "! (rep: ^rep_ty) (response: num->num) (operand_out: num->fp)
     (decode_reg:num->num) (resp_ready:num->bool) (biu_state:num->num)
     (start:num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr:num->bool) (f_ac:num->fp) (operand_ready:num->bool) (n:num).
    biu_top_cpu n rep (response, operand_out, decode_reg, resp_ready, f_ac,
            biu_state, start) (command, condition, control, operand_in,
            done, new_instr, operand_ready) ==>
    (!t.
       (biu_state t = 0) ==>
         ((response(t+1), operand_out(t+1), decode_reg(t+1),
           resp_ready(t+1), f_ac(t+1), biu_state(t+1), start(t+1)) =
          biu_idle n rep (response t, operand_out t, decode_reg t,
                     resp_ready t, f_ac t, biu_state t, start t)
                     (command t, condition t, control t, operand_in t,
                     done t, new_instr t, operand_ready t)))",
    TAC2
);;


close_theory();;

quit();;
```

```
%-----------------------------------------------------------
File: apu_lemma.ml
Author: Jing Pan
Date: March 1991
Purpose:        Lemmas for "apu_top.ml".
-----------------------------------------------------------%
```

```
loadf '/csgrad/panj/holdir/init.ml';;
loadf 'abstract.ml';;
loadf 'tactics';;
system '/bin/rm -f apu_lemma.th';;

set_flag ('sticky', true);;

new_theory 'apu_lemma';;

loadf 'aux_defs.ml';;

map new_parent ['aux'; 'interface'; 'fptype'; 'apu_top'];;

autoload_defs_and_thms 'fptype';;
autoload_defs_and_thms 'apu_top';;



let rep_ty = abstract_type 'interface' 'fetch';;


map loadf ['digit'; 'decimal'];;



%----------------------------------------------
                apu_top_lemma
----------------------------------------------%
let apu_top_lemma = prove_thm
 ('apu_top_lemma',
 "! rep:^rep_ty (n : num) .
  ((NextState_apu n rep 2 = (C_FADD n rep)) /\
   (NextState_apu n rep 3 = (C_FSUB n rep)) /\
   (NextState_apu n rep 4 = (C_FMUL n rep)) /\
   (NextState_apu n rep 5 = (C_FDIV n rep)))",
  REPEAT GEN_TAC
  THEN ONCE_REWRITE_TAC [NextState_apu]
  THEN DEC_EQ_TAC
);;



let TAC1 =
  REPEAT STRIP_TAC
  THEN REWRITE_ASM_THM_TAC apu_top_cpu 3
  THEN ASSUM_LIST (\ths. ASSUME_TAC
       (SPEC "t" (el 1 ths)))
  THEN RM2LAST_TAC
  % RES_TAC -- resolution only handles implication, not conditions %
  THEN REWRITE_ASM 2 1
  THEN REWRITE_ASM 3 1
  THEN REWRITE_ASM_THM_TAC apu_top_lemma 1
  THEN ASM_REWRITE_TAC[];;



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

```
  One lemma for each individual state
-----------------------------------------%

let apu_FADD_lemma = prove_thm
 ('apu_FADD_lemma',
  "! (rep : ^rep_ty) (f_ac : num->fp) (f_reg : num->num->fp)
    (cw sw : num->bool#bool#bool#bool)
    (done : num->bool) (start : num->bool) (decode_reg : num->num)
    (n:num) .
  apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
          (start, decode_reg) ==>
    ! t .
      (start t)  /\
      (Opc n (decode_reg t) = 2) ==>
      ? t'. (t<=t') /\
        (First (t, t') (done)) /\
        ((f_ac t', f_reg t', cw t', sw t', done t') =
         C_FADD n rep (f_ac t, f_reg t, cw t, sw t, done t)
                  (start t, decode_reg t))",
   TAC1
);;



let apu_FSUB_lemma = prove_thm
 ('apu_FSUB_lemma',
  "! (rep : ^rep_ty) (f_ac : num->fp) (f_reg : num->num->fp)
    (cw sw : num->bool#bool#bool#bool)
    (done : num->bool) (start : num->bool) (decode_reg : num->num)
    (n:num) .
  apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
          (start, decode_reg) ==>
    ! t .
      (start t)  /\
      (Opc n (decode_reg t) = 3) ==>
      ? t'. (t<=t') /\
        (First (t, t') (done)) /\
        ((f_ac t', f_reg t', cw t', sw t', done t') =
         C_FSUB n rep (f_ac t, f_reg t, cw t, sw t, done t)
                  (start t, decode_reg t))",
   TAC1
);;



let apu_FMUL_lemma = prove_thm
 ('apu_FMUL_lemma',
  "! (rep : ^rep_ty) (f_ac : num->fp) (f_reg : num->num->fp)
    (cw sw : num->bool#bool#bool#bool)
    (done : num->bool) (start : num->bool) (decode_reg : num->num)
    (n:num) .
  apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
          (start, decode_reg) ==>
    ! t .
      (start t)  /\
      (Opc n (decode_reg t) = 4) ==>
      ? t'. (t<=t') /\
        (First (t, t') (done)) /\
        ((f_ac t', f_reg t', cw t', sw t', done t') =
```

```
        C_FMUL n rop (f_ac t, f_reg t, cw t, sw t, done t)
                (start t, decode_reg t))",
   TAC1
);;


let apu_FDIV_lemma = prove_thm
 ('apu_FDIV_lemma',
  "! (rop : ^rop_ty) (f_ac : num->fp) (f_reg : num->num->fp)
    (cw sw : num->bool#bool#bool#bool)
    (done : num->bool) (start : num->bool) (decode_reg : num->num)
    (n:num) .
  apu_top_cpu n rop (f_ac, f_reg, cw, sw, done)
         (start, decode_reg) ==>
    ! t .
      (start t)  /\
      (Opc n (decode_reg t) = 5) ==>
      ? t'. (t<=t') /\
      (First (t, t') (done)) /\
      ((f_ac t', f_reg t', cw t', sw t', done t') =
         C_FDIV n rop (f_ac t, f_reg t, cw t, sw t, done t)
                (start t, decode_reg t))",
   TAC1
);;


let TAC2 =
  REPEAT STRIP_TAC
  THEN REWRITE_ASM_THM_TAC apu_top_cpu 2
  THEN ASSUM_LIST (\ths. ASSUME_TAC
       (SPEC "t" (el 1 ths)))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 2 1
  THEN REWRITE_ASM_THM_TAC apu_top_lemma 1
  THEN ASM_REWRITE_TAC[];;


let apu_idle_lemma = prove_thm
 ('apu_idle_lemma',
  "! (rop : ^rop_ty) (f_ac : num->fp) (f_reg : num->num->fp)
    (cw sw : num->bool#bool#bool#bool)
    (done : num->bool) (start : num->bool) (decode_reg : num->num)
    (n:num) .
  apu_top_cpu n rop (f_ac, f_reg, cw, sw, done)
         (start, decode_reg) ==>
    ! t . (^start t) ==>
        ((f_ac(t+1), f_reg(t+1), cw(t+1), sw(t+1),
         done(t+1)) =
         apu_idle n rop (f_ac t, f_reg t, cw t, sw t, done t)
                (start t, decode_reg t))",
   TAC2
);;
```

```
close_theory();;
quit();;
```

```
%-------------------------------------------------------------
File: induc.ml
Author: Jing Pan
Date: March 1991
Purpose:        Induction on wait time, for used for
        existential quatified state transition.
-------------------------------------------------------------%


loadf '/csgrad/panj/holdir/init.ml';;

loadf 'abstract.ml';;

loadf 'tactics.ml';;
system '/bin/rm -f induc.th';;

set_flag ('sticky', true);;

new_theory 'induc';;

loadf 'aux_defs.ml';;

map new_parent ['aux'; 'interface'; 'cpu_service'; 'service_lemma';
                'apu_top'; 'apu_lemma'];;

autoload_defs_and_thms 'aux';;
autoload_defs_and_thms 'cpu_service';;
autoload_defs_and_thms 'service_lemma';;
autoload_defs_and_thms 'apu_lemma';;
autoload_defs_and_thms 'apu_top';;

let rep_ty = abstract_type 'interface' 'fetch';;



let lemma_suc = prove_thm
 ('lemma_suc',
  "(!t'. t <= t' /\ t' < (SUC (t + n')) ==> ~resp_ready t')
                    ==>
             (!t'. t <= t' /\ t' < (t + n') ==> ~resp_ready t')",
  STRIP_TAC
  THEN GEN_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
      (SPEC "t':num"(el 1 ths)))
  THEN STRIP_TAC
  THEN REWRITE_ASM1 2 3
       %LESS_SUC: m<n ==> m<SUC n%
  THEN ASSUM_LIST (\ths. ASSUME_TAC
     (SPECL ["t':num"; "t+n'"] LESS_SUC))
```

```
      %1 = "t' < (t + n') ==> t' < (SUC(t + n'))"
       3 = "t' < (t + n')" %
 THEN REWRITE_ASM1 3 1
      % 1 = "t' < (SUC(t + n'))"
        3 = "t' < (SUC(t + n')) ==> ~resp_ready t'" %
 THEN REWRITE_ASM 1 3
 THEN ASM_REWRITE_TAC[]);;



let lemma_suc2 = prove_thm
 ('lemma_suc2',
  "(!t'. t <= t' /\ t' < (SUC (t + n')) ==> ~start t')
                    ==>
             (!t'. t <= t' /\ t' < (t + n') ==> ~start t')",
  STRIP_TAC
  THEN GEN_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
      (SPEC "t':num"(el 1 ths)))
  THEN STRIP_TAC
  THEN REWRITE_ASM1 2 3
       %LESS_SUC: m<n ==> m<SUC n%
  THEN ASSUM_LIST (\ths. ASSUME_TAC
     (SPECL ["t':num"; "t+n'"] LESS_SUC))
       %1 = "t' < (t + n') ==> t' < (SUC(t + n'))"
        3 = "t' < (t + n')" %
  THEN REWRITE_ASM1 3 1
       % 1 = "t' < (SUC(t + n'))"
         3 = "t' < (SUC(t + n')) ==> ~start t'" %
  THEN REWRITE_ASM 1 3
  THEN ASM_REWRITE_TAC[]);;



let lemma_suc3 = prove_thm
 ('lemma_suc3',
 "(!t'. t <= t' /\ t' < (SUC (t + n')) ==> (~(read t') /\ ~(write t')))
                    ==>
          (!t'. t <= t' /\ t' < (t + n') ==> (~(read t') /\ ~(write t')))",
  STRIP_TAC
  THEN GEN_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
      (SPEC "t':num"(el 1 ths)))
  THEN STRIP_TAC
  THEN REWRITE_ASM1 2 3
       %LESS_SUC: m<n ==> m<SUC n%
  THEN ASSUM_LIST (\ths. ASSUME_TAC
     (SPECL ["t':num"; "t+n'"] LESS_SUC))
       %1 = "t' < (t + n') ==> t' < (SUC(t + n'))"
        3 = "t' < (t + n')" %
  THEN REWRITE_ASM1 3 1
       % 1 = "t' < (SUC(t + n'))"
         3 = "t' < (SUC(t + n')) ==> ~start t'" %
  THEN REWRITE_ASM 1 3
  THEN ASM_REWRITE_TAC[]);;
```

```
let TAC_induc =
  REPEAT STRIP_TAC
  THEN REWRITE_TAC[StableUntil; Stable]
  THEN IMP_RES_TAC cpu_wait_for_response_lemma
  THEN RM2LAST_TAC
  %induction on the length of wait period -- "n"%
  THEN INDUCT_TAC
  THENL [
  %base case %
  REWRITE_TAC[ADD_CLAUSES] %t+0=t%
  %NOT_LESS: ! m n. ~m<n = n <= m;
   EQ_SYM_EQ: ! x y. (x=y) = (y=x) %
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (ONCE_REWRITE_RULE [EQ_SYM_EQ] (NOT_LESS)))
  THEN ASM_REWRITE_TAC[];

  % induction step %
  REWRITE_TAC[ADD_CLAUSES] %m+(SUC n)=SUC (m+n)%
  THEN STRIP_TAC
  THEN IMP_RES_TAC lemma_suc
  THEN REWRITE_ASM 1 3 % now we have cpu_state(t+n')=1 %
  THEN IMP_RES_TAC cpu_wait_for_response_lemma
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (SPEC "t+n'" (el 3 ths)))
  %LESS_EQ_ADD: m<=(m+n)%
  THEN REWRITE_ASM_THM_TAC  LESS_EQ_ADD 1
  %LESS_EQ_SUC_REFL: n < (SUC n)%
  THEN REWRITE_ASM_THM_TAC LESS_SUC_REFL 1
   %now we have "~resp_ready(t + n')"%
  THEN REWRITE_ASM1 1 4
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths))) %now we got "cpu_state((t + n') + 1) = 1"%
  THEN REWRITE_TAC[ADD1] %ADD1: m+1 = SUC m%
  THEN ASM_REWRITE_TAC[]
];;



%-------------------------------------------------------
   wait_cpu
-------------------------------------------------------%
let wait_cpu = prove_thm
 ('wait_cpu',
  " ! (rep: ~rep_ty) (c_ac:num->num) (c_reg:num->num->num)
     (mem:num->*memory) (ir:num->num) (dataout :num->*wordn)
     (address :num->num) (read write :num->bool) (cpu_state:num->num)
     (resp_ready:num->bool) (datain :num->*wordn) (n:num) .
     cpu_service n rep (c_ac, c_reg, mem, ir, dataout,
                address, read, write, cpu_state) (resp_ready, datain)
          ==>
          ! (t:num). (cpu_state t = 1) ==>
               StableUntil (cpu_state, 1, t, resp_ready)",
     TAC_induc
);;
```

```
let TAC_induc2 stableuntilx =
  REPEAT STRIP_TAC
  THEN REWRITE_TAC[stableuntilx; Stable]
  THEN IMP_RES_TAC cpu_wait_for_response_lemma
  THEN RM2LAST_TAC
  %induction on the length of wait period -- "n"%
  THENL DUP_TAC (INDUCT_TAC
  THENL [
  %base case %
  REWRITE_TAC[ADD_CLAUSES] %t+0=t%
  %NOT_LESS: ! m n. ~m<n = n <= m;
   EQ_SYM_EQ: ! x y. (x=y) = (y=x) %
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (ONCE_REWRITE_RULE [EQ_SYM_EQ] (NOT_LESS)))
  THEN ASM_REWRITE_TAC[];

  % induction step %
  REWRITE_TAC[ADD_CLAUSES] %m+(SUC n)=SUC (m+n)%
  THEN STRIP_TAC
  THEN IMP_RES_TAC lemma_suc
  THEN REWRITE_ASM1 1 3 % now we have "mem(t+n') = mem t" %
  THEN IMP_RES_TAC wait_cpu
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (REWRITE_RULE [StableUntil; Stable] (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (SPEC "n'" (el 1 ths)))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 3 1  % now we get cpu_state(t+n') = 1 %

%to get (t + n') < (SUC(t + n')) ==> ~resp_ready(t + n')%
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (SPEC "t+n'" (el 4 ths)))
  %LESS_EQ_ADD: m<=(m+n)%
  THEN REWRITE_ASM_THM_TAC  LESS_EQ_ADD 1
  %LESS_EQ_SUC_REFL: m < (SUC m)%
  THEN REWRITE_ASM_THM_TAC LESS_SUC_REFL 1
 %now we have "~resp_ready(t + n')"%

  THEN IMP_RES_TAC cpu_wait_for_response_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC cpu_wait_for_response 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM  2 1
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

 %now we got "f_ac((t + n') + 1) = 1"%
  THEN REWRITE_TAC[ADD1] %ADD1: m+1 = SUC m%
  THEN ASM_REWRITE_TAC[]
]);;


%-------------------------------------------------------
   wait_cpu2
-------------------------------------------------------%
```

```
let wait_cpu2 = prove_thm
 ('wait_cpu2',
  " ! (rep: ^rep_ty) (c_ac:num->num) (c_reg:num->num->num)
     (mem:num->*memory) (ir:num->num) (dataout :num->*wordn)
     (address :num->num) (read write :num->bool) (cpu_state:num->num)
     (resp_ready:num->bool) (datain :num->*wordn) (n:num) .
     cpu_service n rep (c_ac, c_reg, mem, ir, dataout,
                address, read, write, cpu_state) (resp_ready, datain)
          ==>
          ! (t:num). (cpu_state t = 1) ==>
               StableUntil2 (mem, t, resp_ready) /\
               StableUntil2 (ir, t, resp_ready)",
    TAC_induc2 StableUntil2
);;



%-----------------------------------------------------------
   wait_cpu3
-----------------------------------------------------------%
let wait_cpu3 = prove_thm
 ('wait_cpu3',
 " ! (rep: ^rep_ty) (c_ac:num->num) (c_reg:num->num->num)
     (mem:num->*memory) (ir:num->num) (dataout :num->*wordn)
     (address :num->num) (read write :num->bool) (cpu_state:num->num)
     (resp_ready:num->bool) (datain :num->*wordn) (n:num) .
     cpu_service n rep (c_ac, c_reg, mem, ir, dataout,
                address, read, write, cpu_state) (resp_ready, datain)
          ==>
          ! (t:num). (cpu_state t = 1) ==>
               StableUntil3 (read, F, t, resp_ready) /\
               StableUntil3 (write, F, t, resp_ready)",
  REPEAT STRIP_TAC
  THEN REWRITE_TAC[StableUntil3; Stable]
  THEN IMP_RES_TAC cpu_wait_for_response_lemma
  THEN RM2LAST_TAC
  %induction on the length of wait period -- "n"%
  THENL DUP_TAC (INDUCT_TAC
  THENL [
  %base case %
  REWRITE_TAC[ADD_CLAUSES] %t+0=t%
  %NOT_LESS: ! m n. ^m<n = n <= m;
   EQ_SYM_EQ: ! x y. (x=y) = (y=x) %
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (ONCE_REWRITE_RULE [EQ_SYM_EQ] (NOT_LESS)))
  THEN ASM_REWRITE_TAC[]
  THEN STRIP_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (SPEC "t:num" (el 1 ths)))
  THEN ASSUM_LIST (\ths. ASSUME_TAC   %LESS_REFL:^n<n%
         (REWRITE_RULE[LESS_REFL] (el 1 ths)))
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (REWRITE_RULE[SYM_RULE ADD1] (el 1 ths)))
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (REWRITE_RULE[LESS_SUC_REFL] (el 1 ths)))
                %LESS_SUC_REFL:n<SUC n%
  %now we get ^resp_ready t%
  THEN REWRITE_ASM 1 7
```

```
  THEN ASSUM_LIST (\ths.
       TAC1 (el 1 ths))  %now we get ~read(t+1)%
  THEN ASM_REWRITE_TAC[];

  % induction step %
  REWRITE_TAC[ADD_CLAUSES] %m+(SUC n)=SUC (m+n)%
  THEN STRIP_TAC
  THEN IMP_RES_TAC lemma_suc
  THEN REWRITE_ASM1 1 3 % now we have "~read(t+(n'+1))" %

  THEN IMP_RES_TAC wait_cpu
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (REWRITE_RULE [StableUntil; Stable] (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (SPEC "(n'+1)" (el 1 ths)))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 3 1  % now we get cpu_state(t+(n'+1)) = 1 %

%to get (t + n') < (SUC(t + n')) ==> ~resp_ready(t + n')%
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (SPEC "t+(n'+1)" (el 4 ths)))
  %LESS_EQ_ADD: m<=(m+n)%
  THEN REWRITE_ASM_THM_TAC  LESS_EQ_ADD 1
  %LESS_EQ_SUC_REFL: m < (SUC m)%
  THEN REWRITE_ASM_THM_TAC LESS_SUC_REFL 1
 %now we have "~resp_ready(t + (n'+1))"%

  THEN IMP_RES_TAC cpu_wait_for_response_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC cpu_wait_for_response 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM  2 1
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

 %now we got "~read((t + (n'+1)) + 1)"%
  THEN REWRITE_TAC[ADD1] %ADD1: m+1 = SUC m%
  THEN ASM_REWRITE_TAC[]
]));;



let TAC2_induc =
  REPEAT STRIP_TAC
  THEN REWRITE_TAC[StableUntil2; Stable]
  THEN IMP_RES_TAC cpu_idle_lemma
  THEN RM2LAST_TAC

  %induction on the length of wait period -- "n"%
  THENL DUP_TAC (INDUCT_TAC

  THENL [
  %base case %
  REWRITE_TAC[ADD_CLAUSES] %t+0=t%
  THEN ASM_REWRITE_TAC[];

  % induction step %
```

```
  REWRITE_TAC[ADD_CLAUSES] %m+(SUC n)=SUC (m+n)%

  THEN STRIP_TAC % manipulate the induction assumption %

  THEN IMP_RES_TAC lemma_suc2
  THEN REWRITE_ASM 1 3 % now we have f_ac(t+n')=f_ac t %

  %to get (t + n') < (SUC(t + n')) ==> ~start(t + n')%
  THEN ASSUM_LIST (\ths.
         (ASSUME_TAC (SPEC "t+n'" (el 2 ths))))
  THEN REWRITE_ASM_THM_TAC  LESS_EQ_ADD 1   %m<=(m+n)%
  THEN REWRITE_ASM_THM_TAC  LESS_SUC_REFL 1   %m < (SUC m)%

  %now we have "~start(t + n')"%
  THEN IMP_RES_TAC apu_idle_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))

 %now we got "f_ac((t + n') + 1) = 1"%
  THEN REWRITE_TAC[ADD1] %ADD1: m+1 = SUC m%
  THEN ASM_REWRITE_TAC□
]);;



%-------------------------------------------------------
   wait_cpu
-------------------------------------------------------%
let  wait_apu = prove_thm
  ('wait_apu',
  "! (rep : ~rep_ty) (f_ac : num->fp) (f_reg : num->num->fp)
    (cw sw : num->bool#bool#bool#bool)
    (done : num->bool) (start : num->bool) (decode_reg : num->num)
    (n:num) .
  apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
          (start, decode_reg)
          ==>
          ! (t:num). ~(start t) ==>
               StableUntil2 (f_ac, t, start) /\
               StableUntil2 (f_reg, t, start)",
  TAC2_induc
);;



%-------------------------------------------------------
   wait_cpu2
-------------------------------------------------------%
let wait_apu2 = prove_thm
  ('wait_apu2',
  "! (rep : ~rep_ty) (f_ac : num->fp) (f_reg : num->num->fp)
    (cw sw : num->bool#bool#bool#bool)
    (done : num->bool) (start : num->bool) (decode_reg : num->num)
    (n:num) .
  apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
```

```
          (start, decode_reg)
          ==>
          ! (t:num). ~(start t) ==>
               StableUntil2 (cw, t, start) /\
               StableUntil2 (sw, t, start)",
  TAC2_induc
);;


close_theory();;
quit();;
```

```
%-------------------------------------------------
arith.ml

  Theorems concerning number manipulation, used in
conjunction with the theroems in the file "induc.ml".
-------------------------------------------------%


loadf '/csgrad/panj/holdir/init.ml';;
loadf 'tactics.ml';;

system '/bin/rm -f a.th';;

set_flag ('sticky', true);;

new_theory 'a';;

loadf 'aux_defs.ml';;

map new_parent ['aux'; 'induc'; 'num_thms'];;
autoload_defs_and_thms 'induc';;

loadf 'normalize';;
loadf 'num_thms';;


%-------------------------------------------
                num_lemma1
-------------------------------------------%
let num_lemma1 = prove_thm
 ('num_lemma1',
  "! t t':num . ((((t+1)+1)<=t') ==> ((t+1)+ ((t'- t) - 1)=t'))",
  REPEAT STRIP_TAC
  THEN NORMALIZE_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
    (SPECL ["t'-t"; "1"; "1"] SUB_ADD_SUB))

%now we need to get 1<=(t'-t) from ((t+1)+1) <= t' %
  % first to get SUC SUC t < t'%
  THEN REWRITE_ASM_THM_TAC (SYM_RULE ADD1) 2
```

```
  % to get SUC SUC t<t' \/ SUC SUC t=t' %
  THEN REWRITE_ASM_THM_TAC LESS_OR_EQ 1
  THEN RM2LAST_TAC
  % to get SUC t < t'%
  THEN REWRITE_ASM_THM_TAC (SYM_RULE LESS_EQ_SUC_LESS) 1
  THEN RM2LAST_TAC
  %to get t<t'%
  THEN IMP_RES_TAC SUC_LESS
  % to get 0 < (t'-t)%
  THEN IMP_RES_TAC (SYM_RULE SUB_LESS_0)
  THEN POP_TOP_ASSUMP_TAC
  % to get SUC 0 <= (t'-t)%
  THEN REWRITE_ASM_THM_TAC (LESS_EQ) 1
  % to get SUC 0%
  THEN REWRITE_ASM_THM_TAC ADD1 1
  THEN REWRITE_ASM_THM_TAC ADD 1

  % clean up, to get rid of unnecessary assumptions %
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC

  % to switch -1 and +1 %
  THEN REWRITE_ASM1 1 4
  THEN ASM_REWRITE_TAC[]
  THEN REWRITE_TAC [ADD_SUB]

 % now we have got: (t'-t)+t = t' as the goal %
  THEN ASSUM_LIST (\ths. ASSUME_TAC
    (SPECL ["t'"; "t"; "t"] SUB_ADD_SUB))

 % to get t<=t' ......%
  THEN REWRITE_ASM_THM_TAC LESS_OR_EQ 1
  THEN REWRITE_ASM 5 1   % 5= t<t' %
  THEN ASM_REWRITE_TAC[]
  THEN REWRITE_TAC [ADD_SUB]);;



%-----------------------------------------------
                num_lemma2b
-----------------------------------------------%
let num_lemma2b = prove_thm
 ('num_lemma2b',
  "! t t':num . ((((t+1)+1)<=t') ==>
                   (((t+1)+1) + (((t'- t) - 1)-1) = t'))",
  REPEAT STRIP_TAC
  THEN NORMALIZE_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
    (SPECL ["(t'-t)-1"; "1"; "1"] SUB_ADD_SUB))

%now we need to get 1<=(t'-t)-1 from ((t+1)+1) <= t' %
  % first to get SUC SUC t < t'%
  THEN REWRITE_ASM_THM_TAC (SYM_RULE ADD1) 2
  % to get SUC SUC t<t' \/ SUC SUC t=t' %
  THEN REWRITE_ASM_THM_TAC LESS_OR_EQ 1
  THEN RM2LAST_TAC
  % to get SUC t < t'%
  THEN REWRITE_ASM_THM_TAC (SYM_RULE LESS_EQ_SUC_LESS) 1
```

```
 THEN RM2LAST_TAC
 % to get t+1 < t' %
 THEN REWRITE_ASM_THM_TAC ADD1 1
 % to get 0 < (t'-(t+1))%
 THEN IMP_RES_TAC (SYM_RULE SUB_LESS_0)
 THEN POP_TOP_ASSUMP_TAC
 % to get 0 < (t'-t)-1 %
 THEN REWRITE_ASM_THM_TAC (SYM_RULE SUB_ADD_ASSOC) 1
 THEN RM2LAST_TAC
 % to get SUC 0 <= (t'-t)-1%
 THEN REWRITE_ASM_THM_TAC (LESS_EQ) 1
 % to get SUC 0 = 0+1%
 THEN REWRITE_ASM_THM_TAC ADD1 1
 % to get 0+1 = 1%
 THEN REWRITE_ASM_THM_TAC ADD 1

 % Clean up, to get rid of unnecessary assumptions %

 THEN RM2LAST_TAC
 THEN RM2LAST_TAC

 % to switch -1 and +1 %
 THEN REWRITE_ASM1 1 5
 THEN ASM_REWRITE_TAC[]
 THEN REWRITE_TAC [ADD_SUB]

 % now we have got: (((t'-t)-1)+1)+t = t' as the goal %
 THEN POP_TOP_ASSUMP_TAC
 THEN POP_TOP_ASSUMP_TAC
 THEN POP_TOP_ASSUMP_TAC
 THEN POP_TOP_ASSUMP_TAC
 %to get t<t'%
 THEN IMP_RES_TAC SUC_LESS

 % to get 0 < (t'-t)%
 THEN IMP_RES_TAC (SYM_RULE SUB_LESS_0)
 THEN POP_TOP_ASSUMP_TAC
 % to get SUC 0 <= (t'-t)%
 THEN REWRITE_ASM_THM_TAC (LESS_EQ) 1
 % to get SUC 0 = 0+1%
 THEN REWRITE_ASM_THM_TAC ADD1 1
 % to get 0+1 = 1%
 THEN REWRITE_ASM_THM_TAC ADD 1
 % to get "1 <= (t' - t) ==> (((t' - t) - 1) + 1 = ((t' - t) + 1) - 1)" %
 THEN ASSUM_LIST (\ths. ASSUME_TAC
   (SPECL ["t'-t"; "1"; "1"] SUB_ADD_SUB))
 % to get "((t' - t) - 1) + 1 = ((t' - t) + 1) - 1"%
 THEN REWRITE_ASM 2 1
 THEN ASM_REWRITE_TAC[]
 THEN REWRITE_TAC [ADD_SUB]

% to get t<=t' .....%
 THEN POP_TOP_ASSUMP_TAC
 THEN POP_TOP_ASSUMP_TAC
 THEN POP_TOP_ASSUMP_TAC
 THEN POP_TOP_ASSUMP_TAC
 THEN POP_TOP_ASSUMP_TAC

 %(t' - t) + t = t'%
```

```
  THEN ASSUM_LIST (\ths. ASSUME_TAC
    (SPECL ["t'"; "t"; "t"] SUB_ADD_SUB))
  THEN REWRITE_ASM_THM_TAC LESS_OR_EQ 1
  THEN REWRITE_ASM 3 1   % 3 = t<t' %
  THEN ASM_REWRITE_TAC□
  THEN REWRITE_TAC [ADD_SUB]);;



%-------------------------------------------
                num_lemma2

  Used in conjunction with (1) "wait_cpu" to get
"cpu_state t' = 1"; (2) "wait_cpu2" to get
"mem t' = mem (t+1+1)" and "ir t' = ir (t+1+1)";
(3) "wait_apu" to get "f_ac t'.." and "f_reg t'.."
-------------------------------------------%
let num_lemma2 = prove_thm
 ('num_lemma2',
 "! t t':num . (((((t+1)+1)+1)<=t') ==>
                  (((t+1)+1) + (((t'- t) - 1)-1) = t'))",
  REPEAT STRIP_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (SPECL ["((t+1)+1)"; "t'"] (SYM_RULE LESS_EQ)))
         %LESS_EQ = m<n = (SUC m <= n)%
  THEN REWRITE_ASM_THM_TAC ADD1 1
  THEN REWRITE_ASM1 3 1
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (SPECL ["t"; "t'"] num_lemma2b))
  THEN REWRITE_ASM_THM_TAC LESS_OR_EQ 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 2 1
  THEN ASM_REWRITE_TAC□
);;



%-------------------------------------------
                num_lemma3b
-------------------------------------------%
let num_lemma3b = prove_thm
 ('num_lemma3b',
 "! t t':num . ((((((t+1)+1)+1)+1)+1)<=t') ==>
                  (((((t+1)+1)+1)+1) + (((((t'-t)-1)-1)-1)-1) = t'))",
  REPEAT STRIP_TAC
  THEN ASSUME_TAC num_lemma2b
  THEN ASSUM_LIST (\ths. ASSUME_TAC
    (SPECL ["(t+1)+1"; "t':num"] (el 1 ths)))
  THEN ASSUM_LIST (\ths. ASSUME_TAC
      (SPECL ["t'"; "(t+1)"; "1"] (SYM_RULE SUB_ADD_ASSOC)))
  THEN REWRITE_ASM1 1 2
    % now get "((((t + 1) + 1) + 1) + 1) <= t' ==>
       (((((t + 1) + 1) + 1) + 1) + ((((t' - (t + 1)) - 1) - 1) - 1) = t')%
  THEN ASSUM_LIST (\ths. ASSUME_TAC
      (SPECL ["t'"; "t"; "1"] (SYM_RULE SUB_ADD_ASSOC)))
  THEN REWRITE_ASM1 1 2
    % now get "((((t + 1) + 1) + 1) + 1) <= t' ==>
       (((((t + 1) + 1) + 1) + 1) + (((((t' - t) - 1) - 1) - 1) - 1) = t')"%
```

```
  THEN REWRITE_ASM1 7 1
  THEN ASM_REWRITE_TAC[]
);;



%-------------------------------------------
                num_lemma3
  Used in conjunction with (1) "wait_cpu" to get
"f_reg t''''' = f_reg (t'+1+1+1+1)".
-------------------------------------------%
let num_lemma3 = prove_thm
 ('num_lemma3',
 "! t t':num . (((((((t+1)+1)+1)+1)+1)+1)<=t') ==>
                    (((((t+1)+1)+1)+1) + (((((t'-t)-1)-1)-1)-1)-1) = t'))",
  REPEAT STRIP_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (SPECL ["((((t+1)+1)+1)+1)"; "t'"] (SYM_RULE LESS_EQ)))
         %LESS_EQ = m<n = (SUC m <= n)%
  THEN REWRITE_ASM_THM_TAC ADD1 1
  THEN REWRITE_ASM1 3 1
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (SPECL ["t"; "t'"] num_lemma3b))
  THEN REWRITE_ASM_THM_TAC LESS_OR_EQ 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 2 1
  THEN ASM_REWRITE_TAC[]
);;



%-------------------------------------------
                num_lemma4
-------------------------------------------%
let num_lemma4 = prove_thm
 ('num_lemma4',
 "! t t':num . (((((t+1)+1)+1)+1)<=t') ==>
                    (((t+1)+1) + (((((t'- t) - 1)-1)-1)+1) = t'))",
  REPEAT STRIP_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
    (SPECL ["((t'-t)-1)-1"; "1"; "1"] SUB_ADD_SUB))

%now we need to get 1<=((t'-t)-1)-1 from (((t+1)+1)+1) <= t' %
  % first to get SUC SUC SUC t < t'%
  THEN REWRITE_ASM_THM_TAC (SYM_RULE ADD1) 2
  % to get SUC SUC SUC t<t' \/ SUC SUC SUC t=t' %
  THEN REWRITE_ASM_THM_TAC LESS_OR_EQ 1
  THEN RM2LAST_TAC
  % to get SUC SUC t < t'%
  THEN REWRITE_ASM_THM_TAC (SYM_RULE LESS_EQ_SUC_LESS) 1
  THEN RM2LAST_TAC
  % to get 0 < (t'-SUC SUC t)%
  THEN IMP_RES_TAC (SYM_RULE SUB_LESS_0)
  % to get SUC 0 <= (t'-SUC SUC t)%
  THEN REWRITE_ASM_THM_TAC (LESS_EQ) 1
  % to get SUC 0 = 1%
```

```
  THEN REWRITE_ASM_THM_TAC ADD1 1
  THEN REWRITE_ASM_THM_TAC ADD 1
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC (SYM_RULE SUB_ADD_ASSOC) 1
         %SUB_ADD_ASSOC = |- !a b c. (a - b) - c = a - (b + c)%
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 1 3
  THEN REWRITE_ASM_THM_TAC ADD_SUB 1  %ADD_SUB = (n+m)-m=n%

% now we get "((((t' - t) - 1) - 1) - 1) + 1 = ((t' - t) - 1) - 1"%
  THEN ASM_REWRITE_TAC□
  THEN IMP_RES_TAC num_lemma2
);;



close_theory();;
quit();;
```

APPENDIX C: VERIFICATION OF FPC TOP-LEVEL INTERPRETER

```
%-------------------------------------------------------------
File: fpc_top.ml
Author: Jing Pan
Date: Aug 1990

  The top level description of the APU
-------------------------------------------------------------%


loadf '/csgrad/panj/holdir/init.ml';;

loadf 'abstract.ml';;

system '/bin/rm -f fpc_top.th';;

set_flag ('sticky', true);;

new_theory 'fpc_top';;

loadf 'aux_defs.ml';;

map new_parent ['aux'; 'interface'; 'fptype'; 'apu_top'];;


autoload_defs_and_thms 'aux';;
autoload_defs_and_thms 'fptype';;
autoload_defs_and_thms 'apu_top';;


let rep_ty = abstract_type 'interface' 'fetch';;




%-------------------------------------------------
  state on the top level for APU:
     (f_ac, f_reg, c_ac, c_reg, mem)
  env on APU top level:
     (ir)

note: (c_ac, c_reg, mem, ir) are actually belong to
       CPU state. But since FSTR instruction changes
       CPU state, c_ac, c_reg and mem has to be
       included here.
-------------------------------------------------%




%-------------------------------------------------
Arithmetic Instruction
  perform arthmetic operation on single or double
precision floating point (real) numbers:
```

```
  FADD, FSUB, FMUL, FDIV, FREM

These operations are entirely internal to the coprocessor.


        (f_ac) + (f_reg i) -> f_ac
-------------------------------------------------%

let FADD = new_definition
  ('FADD',
  "! (rep : `rep_ty) (f_ac : fp) (f_reg : num->fp) (c_ac : num)
     (c_reg :num->num) (mem:*memory) (ir :num) (n:num) .
    FADD n rep (f_ac, f_reg, c_ac, c_reg, mem) (ir) =
    let (addr:num) = (Addr n ir) in
    ((FST (FP_ADD n1 n2 f_ac (f_reg addr))), f_reg, c_ac, c_reg,
            mem)"
);;


let FSUB = new_definition
  ('FSUB',
  "! (rep : `rep_ty) f_ac f_reg c_ac c_reg mem ir n .
    FSUB n rep (f_ac, f_reg, c_ac, c_reg, mem) (ir) =
   let (addr:num) = (Addr n ir) in
    ((FST (FP_SUB n1 n2 f_ac (f_reg addr))),  f_reg, c_ac, c_reg,
            mem)"
);;


let FMUL = new_definition
  ('FMUL',
  "! (rep : `rep_ty) f_ac f_reg c_ac c_reg mem ir n .
    FMUL n rep (f_ac, f_reg, c_ac, c_reg, mem) (ir) =
  let (addr:num) = (Addr n ir) in
    ((FST (FP_MUL n1 n2 f_ac (f_reg addr))), f_reg, c_ac, c_reg,
            mem)"
);;


let FDIV = new_definition
  ('FDIV',
 "! (rep : `rep_ty) f_ac f_reg c_ac c_reg mem ir n .
    FDIV n rep (f_ac, f_reg, c_ac, c_reg, mem) (ir) =
     let (addr:num) = (Addr n ir) in
    ((FST (FP_DIV n1 n2 f_ac (f_reg addr))), f_reg, c_ac, c_reg,
            mem)"
);;




%-------------------------------------------------
Load & Store

  Load from operand cir to F_AC and store F_AC to
the operand cir.
-------------------------------------------------%

let FLD = new_definition
```

```
  ('FLD',
 "! (rep : *rep_ty) f_ac f_reg c_ac c_reg mem ir n .
    FLD n rep (f_ac, f_reg, c_ac, c_reg, mem) (ir) =
    let data = fetch rep mem (Addr n ir) in
       (((wtofp rep) data), f_reg, c_ac, c_reg, mem)"
);;


let FSTR = new_definition
  ('FSTR',
 "! (rep : *rep_ty) f_ac f_reg c_ac c_reg mem ir n .
    FSTR n rep (f_ac, f_reg, c_ac, c_reg, mem) (ir) =
    let new_mem = (store rep) mem (Addr n ir) (fptow rep f_ac) in
              (f_ac, f_reg, c_ac, c_reg, new_mem)"
);;



%-------------------------------------------------
NextState_fpc
  Define the next state of instr level
-------------------------------------------------%
let NextState_fpc = new_definition
  ('NextState_fpc',
 "! (rep : *rep_ty) ir n .
    NextState_fpc n rep ir =
      (((Opc n ir) = 0) => (FLD n rep) |
       ((Opc n ir) = 1) => (FSTR n rep) |
       ((Opc n ir) = 2) => (FADD n rep) |
       ((Opc n ir) = 3) => (FSUB n rep) |
       ((Opc n ir) = 4) => (FMUL n rep) |
                           (FDIV n rep))"
);;



%-------------------------------------------------
fpc_top

  The top level specification of the floating-point
coprocessor
-------------------------------------------------%
let fpc_top = new_definition
  ('fpc_top',
 "! (rep : *rep_ty) (f_ac :num->fp) (f_reg :num->num->fp) (c_ac :num->num)
     (c_reg :num->num->num) (mem:num->*memory) (ir :num->num)
     (n:num) .
  fpc_top n rep (f_ac, f_reg, c_ac, c_reg, mem) (ir) =
    ! t . ? t'.
     (f_ac t', f_reg t', c_ac t', c_reg t', mem t') =
       NextState_fpc n rep (ir t) (f_ac t, f_reg t, c_ac t, c_reg t,
                    mem t) (ir t)"
);;



%-------------------------------------------------
          FPCstate
```

```
 the cpu accumulator and register, memory, and instruction
register will be different when the current FP instruction
finishes, since some CPU instruction might have been executed
-------------------------------------------------%

let FPCstate = new_definition
 ('FPCstate',
 "! (f_ac:fp) (f_reg:num->fp) (c_ac:num) (c_reg:num->num)
    (mem:*memory) (ir:num) .
  FPCstate (f_ac, f_reg, c_ac, c_reg, mem) = (f_ac, f_reg)"
);;


close_theory();;

quit();;









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

File: f_arith_correct.ml
Author: Jing Pan
Date: Jan. 1991
Purpose:        Verification of FPC top level against top level
                spec of the BIU, APU, and CPU_service, in the
                case of an arithmetic instruction (FADD, FSUB,
                FMUL, FDIV).

Theories Used:  aux, biu_top, biu_lemma, apu_top, apu_lemma,
                cpu_service, service_lemma, fpc_top, induc, a.

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



loadf '/csgrad/panj/holdir/init.ml';;

loadf 'abstract.ml';;
loadf 'tactics.ml';;

system '/bin/rm -f f_arith_correct.th';;

set_flag ('sticky', true);;

new_theory 'f_arith_correct';;


loadf 'aux_defs.ml';;

map new_parent ['aux'; 'biu_top'; 'biu_lemma'; 'apu_top';
                'apu_lemma'; 'cpu_service'; 'service_lemma';
```

```
                'fpc_top'; 'induc'; 'a'];;

autoload_defs_and_thms 'biu_top';;
autoload_defs_and_thms 'biu_lemma';;
autoload_defs_and_thms 'apu_top';;
autoload_defs_and_thms 'apu_lemma';;

autoload_defs_and_thms 'cpu_service';;
autoload_defs_and_thms 'service_lemma';;
autoload_defs_and_thms 'fpc_top';;
autoload_defs_and_thms 'aux';;
autoload_defs_and_thms 'induc';;
autoload_defs_and_thms 'a';;


let rep_ty = abstract_type 'interface' 'fetch';;

new_theory_obligations [
  "! (x:num) (rep:^rep_ty). ((wtonum rep (numtow rep x)) = x)";
  "! (x:fp) (rep:^rep_ty). ((wtofp rep (fptow rep x)) = x)";
];;


let double_num = prove_thm
  ('double_num',
   "! (x:num) (rep:^rep_ty). ((wtonum rep (numtow rep x)) = x)",
    REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;


let double_fp = prove_thm
  ('double_fp',
   "! (x:fp) (rep:^rep_ty). ((wtofp rep (fptow rep x)) = x)",
    REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

map loadf ['digit'; 'decimal'];;



let TAC_storm apu_instr_lemma C_instr instr =
%t->t+1%
  REPEAT STRIP_TAC
%(1) APU: apu_idle_lemma -- idle, waiting for start signal%
  THEN IMP_RES_TAC apu_idle_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
%(2) BIU -- idle, waiting for the new_instr signal%
  THEN IMP_RES_TAC biu_idle_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_idle 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 11 1  %11= "~new_instr t"%
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
%(3) CPU -- start up the communication by sending the instr%
```

```
   THEN IMP_RES_TAC cpu_begin_lemma
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC cpu_begin 1
   THEN RM2LAST_TAC
   THEN ASSUM_LIST (\ths.
          (TAC1 (el 1 ths)))

 %t+1 --> t+1+1%
 %(1) correctness of the 4-phase handshaking: cir_read_write --
      get "command (t+1+1) = ir t" %
   THEN REWRITE_ASM_THM_TAC cir_read_write 29 % 29 = cir_read_write%
   THEN ASSUM_LIST (\ths.
           (ASSUME_TAC (SPEC "t+1" (el 1 ths))))
   THEN RM2LAST_TAC
   THEN REWRITE_ASM1 25 4 %25="read t, 4="read(t+1)=read t"%
   THEN REWRITE_ASM1 1 2
   THEN RM3LAST_TAC
   THEN REWRITE_ASM 4 1 % 4 = write (t+1) %
   THEN REWRITE_ASM 6 1 %6="address(t+1)=0"%
   THEN ASSUM_LIST (\ths.
          (TAC1 (el 1 ths)))
   THEN REWRITE_ASM1 8 2   %8="dataout=.." and 2="command=dataout..."%
   THEN REWRITE_ASM_THM_TAC double_num 1
   THEN RM2LAST_TAC
 % (2) biu idles until  new_instr is high %
   THEN IMP_RES_TAC biu_idle_lemma
   THEN POP_TOP_ASSUMP_TAC
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC biu_idle 1
   THEN RM2LAST_TAC
   THEN REWRITE_ASM 3 1 %2 = new_instr(t+1)%
   THEN ASSUM_LIST (\ths.
          (TAC1 (el 1 ths)))

 % (3) CPU waits until the response come %
   THEN IMP_RES_TAC cpu_wait_for_response_lemma
   THEN RM2LAST_TAC
   THEN REWRITE_ASM1 37 25 % get "resp_ready (t+1) %
   THEN REWRITE_ASM 1 2
   THEN RM2LAST_TAC
   THEN ASSUM_LIST (\ths.
          (TAC1 (el 1 ths)))

 % (4) APU still idles %
   THEN ASSUM_LIST (\ths. ASSUME_TAC
          (REWRITE_RULE [SYM_RULE (el 30 ths)] (el 47 ths)))
                  % to get ""start (t+1)" first.
                    47 = ""start t", 30="start(t+1) = start t" %
   THEN IMP_RES_TAC apu_idle_lemma
   THEN POP_TOP_ASSUMP_TAC
   THEN RM2LAST_TAC
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC apu_idle 1
   THEN RM2LAST_TAC
   THEN ASSUM_LIST (\ths.
          (TAC1 (el 1 ths)))

 %t+1+1 -> t'%
 % (1) biu decodes, sends back response, and starts up the apu.
```

```
  first time nondeterministic.......?t'%
  THEN IMP_RES_TAC biu_decode_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_decode 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (REWRITE_RULE [need_read; need_write] (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
         (REWRITE_OTHER_TAC 1))
  THEN RM2LAST_TAC
  THEN POP_ASSUM (\th1. ASSUME_TAC
         ((CONV_RULE DEC_EQ_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
         (CONV_RULE (ONCE_DEPTH_CONV
             INV_dec_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
         (REWRITE_RULE [First] th1))
  THEN POP_ASSUM (\th1.
         STRIP_ASSUME_TAC th1)  % get rid of ?t' and make (A/\B) two
                                      separate assumptions A, B %
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

% (2) CPU -- waiting for the response from BIU, induction theroem
      wait_cpu is used here%
  THEN IMP_RES_TAC wait_cpu  %resolve against (cpu_state t+1 = 1) and
                              (cpu_state t+1+1 = 1)%
  THEN RM3LAST_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil 1
% specialize n%
  THEN POP_ASSUM (\th1.
        ASSUME_TAC (SPEC "((t'-t)-1)-1" th1))
  THEN IMP_RES_TAC num_lemma2 %a numerical lemma %
  THEN REWRITE_ASM 1 2  % to rewrite use the theorem %
  THEN REWRITE_ASM 13 1  % 13 = Stable(resp_ready,F,(t + 1) + 1,t') %

% (3) APU -- waiting to be started, induction thereomm
      wait_apu and wait_apu2 are used here%

  % (3.1) to get f_ac t' = f_ac t+1+1, and f_reg t' = f_reg t+1+1 %
  % to get ~start(t+1+1) first
    29 = start t+1+1, 51 = start t+1, 69 = ~start t %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
         (REWRITE_OTHER_TAC 29))
  THEN IMP_RES_TAC wait_apu  % resolve against ~start t and ~start(t+1+1) %
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil2 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil2 2
  THEN RM3LAST_TAC
  THEN POP_ASSUM (\th1.
        ASSUME_TAC (SPEC "((t'-t)-1)-1" th1))
  THEN POP_ASSUM (\th1.  POP_ASSUM (\th2.
          ASSUME_TAC (SPEC "((t'-t)-1)-1" th2)
          THEN ASSUME_TAC th1))
```

```
   THEN IMP_RES_TAC num_lemma2 %get ((t + 1) + 1) + (((t' - t) - 1) - 1) = t'%
   THEN REWRITE_ASM1 1 2  % to rewrite use the theorem %
   THEN RM3LAST_TAC
   THEN REWRITE_ASM1 2 3
   THEN REWRITE_ASM 16 1   % 16 = Stable(start,F,(t + 1) + 1,t') %
   THEN REWRITE_ASM1 16 2
   THEN RM3LAST_TAC

 % (3.2) to get cw t' = cw t+1+1 and sw t' = sw t+1+1 %
   THEN IMP_RES_TAC wait_apu2  % resolve against ~start t and ~start(t+1+1) %
   THEN POP_TOP_ASSUMP_TAC
   THEN POP_TOP_ASSUMP_TAC
   THEN RM3LAST_TAC
   THEN REWRITE_ASM_THM_TAC StableUntil2 1
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC StableUntil2 2
   THEN RM3LAST_TAC
   THEN POP_ASSUM (\th1.
         ASSUME_TAC (SPEC "((t'-t)-1)-1" th1))
   THEN POP_ASSUM (\th1.  POP_ASSUM (\th2.
           ASSUME_TAC (SPEC "((t'-t)-1)-1" th2)
           THEN ASSUME_TAC th1))
   THEN REWRITE_ASM 5 1 %rewrite use the numerical thereom %
   THEN REWRITE_ASM1 5 2
   THEN RM3LAST_TAC
   THEN REWRITE_ASM 18 1   % 19 = Stable(start,F,(t + 1) + 1,t') %
   THEN REWRITE_ASM1 18 2
   THEN RM3LAST_TAC

 %t'-->t'+1%
 %now we get cpu_state t' = 1, f_ac t' = f_ac t+1+1, f_reg t' = f_reg t+1+1 %
 % (1) CPU goes to state 2 to analyze the response, and find out
   it doesn't have to do anything else %
   THEN IMP_RES_TAC cpu_wait_for_response_lemma
   THEN RM2LAST_TAC
   THEN POP_TOP_ASSUMP_TAC
   THEN RM2LAST_TAC
   THEN REWRITE_ASM1 15 1   % 15 = resp_ready t' %
   THEN RM2LAST_TAC
   THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

 % t'-->t''%
 %  (1) APU execute C_instr.
    we need to get (start t') /\ (Opc n (decode_reg t') = 2) first%
 %24 = "decode_reg t' = ir t" and
  83 = "Opc n(ir t) = 2" %
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (REWRITE_RULE [SYM_RULE (el 24 ths)] (el 83 ths)))
  THEN IMP_RES_TAC apu_instr_lemma
 % to match against "! t. (start t) /\ (Opc n (ir t)) ==> "%
  THEN  ASSUM_LIST (\ths. ASSUME_TAC
         (HOL_MATCH_MP (el 1 ths)
          (CONJ (el 22 ths ) (el 2 ths)))) % 22 = start t'%
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC C_instr 1
  THEN RM2LAST_TAC
  THEN POP_ASSUM (\th1.
         STRIP_ASSUME_TAC th1)  % get rid of ?t' and make (A/\B) two
```

```
                                        separate assumptions A, B %
 THEN ASSUM_LIST (\ths. ASSUME_TAC
        (REWRITE_RULE[LET_DEF] (el 1 ths)))
 THEN RM2LAST_TAC
 THEN ASSUM_LIST (\ths. ASSUME_TAC
        (BETA_RULE (el 1 ths)))
 THEN RM2LAST_TAC
 THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
        (REWRITE_OTHER_TAC 1))
 THEN RM2LAST_TAC
 THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))

% t'+1-->t''%
% (1) BIU idles, waiting for the next FP instruction %
% (2) CPU can proceed with the next CPU instruction %

% To rewrite the goal %
 THEN ONCE_REWRITE_TAC[instr]
 THEN ONCE_REWRITE_TAC [LET_DEF]
 THEN BETA_TAC
 THEN REWRITE_TAC[FPCstate]
 THEN EXISTS_TAC "t'':num"
 THEN ASM_REWRITE_TAC[];;




%-------------------------------------------------
            FADD_Correct

  Tp prove the correctness of FADD, upon the interaction
among the BIU, APU and CPU.
-------------------------------------------------%

let FADD_Correct = prove_thm
 ('FADD_Correct',
 "! (rep : ^rep_ty) (response:num->num) (operand_out:num->fp)
     (decode_reg:num->num) (biu_state:num->num)
     (start : num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (operand_ready:num->bool) (resp_ready:num->bool)
     (f_ac : num->fp) (f_reg : num->num->fp)
     (cw sw : num->bool$bool$bool$bool)
     (c_ac:num->num) (c_reg:num->num->num) (mem:num->*memory)
     (ir:num->num) (cpu_state:num->num) (address:num->num)
     (read write :num->bool) (datain dataout:num->*wordn) (n:num).

  (biu_top_cpu n rep  (response, operand_out, decode_reg, resp_ready,
              f_ac, biu_state, start)
             (command, condition, control, operand_in, done, new_instr,
              operand_ready)  /\

   apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
         (start, decode_reg)       /\

   cpu_service n rep (c_ac, c_reg, mem, ir, dataout, address, read, write,
           cpu_state) (resp_ready, datain)       /\
```

```
   (cir_read_write rep address read write datain dataout
                   command response operand_in operand_out condition
                   control new_instr operand_ready))
   ==>
   (!t.  ((Opc n (ir t) = 2) /\
         (~(start t)) /\
         (~(new_instr t)) /\
         (~(resp_ready t)) /\
         (~(read t)) /\
         (biu_state t = 0) /\
         (cpu_state t = 0)) ==>
         (? (t':num).
             ((f_ac t', f_reg t') =
               FPCstate (FADD n rep (f_ac t, f_reg t, c_ac t,
                                     c_reg t, mem t, ir t)))))",
  TAC_storm apu_FADD_lemma C_FADD FADD
);;




%-------------------------------------------------
            FSUB_Correct

  Tp prove the correctness of FSUB, upon the interaction
among the BIU, APU and CPU.
-------------------------------------------------%

let FSUB_Correct = prove_thm
 ('FSUB_Correct',
  "! (rep : ~rep_ty) (response:num->num) (operand_out:num->fp)
     (decode_reg:num->num) (biu_state:num->num)
     (start : num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (operand_ready:num->bool) (resp_ready:num->bool)
     (f_ac : num->fp) (f_reg : num->num->fp)
     (cw sw : num->bool#bool#bool#bool)
     (c_ac:num->num) (c_reg:num->num->num) (mem:num->*memory)
     (ir:num->num) (cpu_state:num->num) (address:num->num)
     (read write :num->bool) (datain dataout:num->*wordn) (n:num).

  (biu_top_cpu n rep  (response, operand_out, decode_reg, resp_ready,
              f_ac, biu_state, start)
             (command, condition, control, operand_in, done, new_instr,
              operand_ready)  /\

   apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
         (start, decode_reg)       /\

   cpu_service n rep (c_ac, c_reg, mem, ir, dataout, address, read, write,
           cpu_state) (resp_ready, datain)       /\

   (cir_read_write rep address read write datain dataout
                   command response operand_in operand_out condition
                   control new_instr operand_ready))
```

```
  ==>
  (!t.  ((Opc n (ir t) = 3) /\
        (~(start t)) /\
        (~(new_instr t)) /\
        (~(resp_ready t)) /\
        (~(read t)) /\
        (biu_state t = 0) /\
        (cpu_state t = 0)) ==>
        (? (t':num).
            ((f_ac t', f_reg t') =
              FPCstate (FSUB n rop (f_ac t, f_reg t, c_ac t,
                                    c_reg t, mem t, ir t)))))",
 TAC_storm apu_FSUB_lemma C_FSUB FSUB
);;




%-------------------------------------------------
            FMUL_Correct

  Tp prove the correctness of FMUL, upon the interaction
among the BIU, APU and CPU.
-------------------------------------------------%

let FMUL_Correct = prove_thm
 ('FMUL_Correct',
  "! (rop : ~rop_ty) (response:num->num) (operand_out:num->fp)
     (decode_reg:num->num) (biu_state:num->num)
     (start : num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (operand_ready:num->bool) (resp_ready:num->bool)
     (f_ac : num->fp) (f_reg : num->num->fp)
     (cw sw : num->bool#bool#bool#bool)
     (c_ac:num->num) (c_reg:num->num->num) (mem:num->*memory)
     (ir:num->num) (cpu_state:num->num) (address:num->num)
     (read write :num->bool) (datain dataout:num->*wordn) (n:num).

  (biu_top_cpu n rop  (response, operand_out, decode_reg, resp_ready,
              f_ac, biu_state, start)
             (command, condition, control, operand_in, done, new_instr,
              operand_ready)  /\

   apu_top_cpu n rop (f_ac, f_reg, cw, sw, done)
         (start, decode_reg)       /\

   cpu_service n rop (c_ac, c_reg, mem, ir, dataout, address, read, write,
           cpu_state) (resp_ready, datain)       /\

   (cir_read_write rop address read write datain dataout
                  command response operand_in operand_out condition
                  control new_instr operand_ready))
   ==>
   (!t.  ((Opc n (ir t) = 4) /\
         (~(start t)) /\
         (~(new_instr t)) /\
         (~(resp_ready t)) /\
```

```
        (~(read t)) /\
        (biu_state t = 0) /\
        (cpu_state t = 0)) ==>
        (? (t':num).
            ((f_ac t', f_reg t') =
              FPCstate (FMUL n rop (f_ac t, f_reg t, c_ac t,
                                    c_reg t, mem t, ir t)))))",
  TAC_storm apu_FMUL_lemma C_FMUL FMUL
);;




%-------------------------------------------------
            FDIV_Correct

  Tp prove the correctness of FDIV, upon the interaction
among the BIU, APU and CPU.
-------------------------------------------------%

let FDIV_Correct = prove_thm
 ('FDIV_Correct',
  "! (rop : ~rop_ty) (response:num->num) (operand_out:num->fp)
     (decode_reg:num->num) (biu_state:num->num)
     (start : num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (operand_ready:num->bool) (resp_ready:num->bool)
     (f_ac : num->fp) (f_reg : num->num->fp)
     (cw sw : num->bool#bool#bool#bool)
     (c_ac:num->num) (c_reg:num->num->num) (mem:num->*memory)
     (ir:num->num) (cpu_state:num->num) (address:num->num)
     (read write :num->bool) (datain dataout:num->*wordn) (n:num).

  (biu_top_cpu n rop  (response, operand_out, decode_reg, resp_ready,
              f_ac, biu_state, start)
             (command, condition, control, operand_in, done, new_instr,
              operand_ready)  /\

   apu_top_cpu n rop (f_ac, f_reg, cw, sw, done)
         (start, decode_reg)       /\

   cpu_service n rop (c_ac, c_reg, mem, ir, dataout, address, read, write,
           cpu_state) (resp_ready, datain)       /\

   (cir_read_write rop address read write datain dataout
                   command response operand_in operand_out condition
                   control new_instr operand_ready))
   ==>
   (!t.  ((Opc n (ir t) = 5) /\
         (~(start t)) /\
         (~(new_instr t)) /\
         (~(resp_ready t)) /\
         (~(read t)) /\
         (biu_state t = 0) /\
         (cpu_state t = 0)) ==>
         (? (t':num).
             ((f_ac t', f_reg t') =
```

```
              FPCstate (FDIV n rop (f_ac t, f_reg t, c_ac t,
                                    c_reg t, mem t, ir t)))))",
  TAC_storm apu_FDIV_lemma C_FDIV FDIV
);;


close_theory();;
quit();;
```

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

File: fld_correct.ml
Author: Jing Pan
Date: Jan. 1991
Purpose:        Verification of FPC top level against top level
                spec of the BIU, APU, and CPU_service, in the
                case of an FLD instruction.

Theories Used:  aux, biu_top, biu_lemma, apu_top, apu_lemma,
                cpu_service, service_lemma, fpc_top, induc, a.

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


loadf '/csgrad/panj/holdir/init.ml';;

loadf 'abstract.ml';;
loadf 'tactics.ml';;

system '/bin/rm -f fld_correct.th';;

set_flag ('sticky', true);;

new_theory 'fld_correct';;


loadf 'aux_defs.ml';;

map new_parent ['aux'; 'biu_top'; 'biu_lemma'; 'apu_top';
                'apu_lemma'; 'cpu_service'; 'service_lemma';
                'fpc_top'; 'induc'; 'a'];;

autoload_defs_and_thms 'biu_top';;
autoload_defs_and_thms 'biu_lemma';;
autoload_defs_and_thms 'apu_top';;
autoload_defs_and_thms 'apu_lemma';;

autoload_defs_and_thms 'cpu_service';;
autoload_defs_and_thms 'service_lemma';;
autoload_defs_and_thms 'fpc_top';;
autoload_defs_and_thms 'aux';;
autoload_defs_and_thms 'induc';;
```

```
autoload_defs_and_thms 'a';;


let rep_ty = abstract_type 'interface' 'fetch';;



new_theory_obligations [
  "! (x:num) (rep:^rep_ty). ((wtonum rep (numtow rep x)) = x)";
  "! (x:fp) (rep:^rep_ty). ((wtofp rep (fptow rep x)) = x)";
];;


let double_num = prove_thm
  ('double_num',
   "! (x:num) (rep:^rep_ty). ((wtonum rep (numtow rep x)) = x)",
    REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;


let double_fp = prove_thm
  ('double_fp',
   "! (x:fp) (rep:^rep_ty). ((wtofp rep (fptow rep x)) = x)",
    REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

map loadf ['digit'; 'decimal'];;



%-------------------------------------------------
            FLD_Correct

  Tp prove the correctness of FLD, upon the interaction
among the BIU, APU and CPU.
-------------------------------------------------%

let FLD_Correct = prove_thm
 ('FLD_Correct',
 "! (rep : ^rep_ty) (response:num->num) (operand_out:num->fp)
     (decode_reg:num->num) (biu_state:num->num)
     (start : num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (operand_ready:num->bool) (resp_ready:num->bool)
     (f_ac : num->fp) (f_reg : num->num->fp)
     (cw sw : num->bool#bool#bool#bool)
     (c_ac:num->num) (c_reg:num->num->num) (mem:num->*memory)
     (ir:num->num) (cpu_state:num->num) (address:num->num)
     (read write :num->bool) (datain dataout:num->*wordn) (n:num).

  (biu_top_cpu n rep  (response, operand_out, decode_reg, resp_ready,
              f_ac, biu_state, start)
             (command, condition, control, operand_in, done, new_instr,
              operand_ready)  /\

   apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
         (start, decode_reg)        /\

   cpu_service n rep (c_ac, c_reg, mem, ir, dataout, address, read, write,
```

```
          cpu_state) (resp_ready, datain)       /\

   (cir_read_write rep address read write datain dataout
                   command response operand_in operand_out condition
                   control new_instr operand_ready))
   ==>
   (!t.  ((Opc n (ir t) = 0) /\  % FLD %
         (~(start t)) /\
         (~(new_instr t)) /\
         (~(resp_ready t)) /\
         (~(read t)) /\
         (biu_state t = 0) /\
         (cpu_state t = 0)) ==>
         (? (t':num).
             ((f_ac t', f_reg t') =
               FPCstate (FLD n rep (f_ac t, f_reg t, c_ac t,
                                    c_reg t, mem t, ir t)))))",
%t->t+1%
  REPEAT STRIP_TAC
%(1) APU: idle %
  THEN IMP_RES_TAC apu_idle_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
% (2) BIU -- idle, waiting for the new_instr signal%
  THEN IMP_RES_TAC biu_idle_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_idle 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 11 1  % 11= "~new_instr t"%
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
%(3) CPU -- start up the communication by sending the instr%
  THEN IMP_RES_TAC cpu_begin_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC cpu_begin 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

%t+1 --> t+1+1%
%(1) correctness of the 4-phase handshaking,, "write": cir_read_write --
     get "command (t+1+1) = ir t" %
  THEN REWRITE_ASM_THM_TAC cir_read_write 29 % 29 = cir_read_write%
  THEN ASSUM_LIST (\ths.
          (ASSUME_TAC (SPEC "t+1" (el 1 ths))))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM1 25 4 %25=~read t, 4="read(t+1)=read t"%
  THEN REWRITE_ASM1 1 2
  THEN RM3LAST_TAC
  THEN REWRITE_ASM 4 1  % 4=write(t+1) %
  THEN REWRITE_ASM 6 1    % 6="address(t+1)=0"%
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
  THEN RM3LAST_TAC
  THEN REWRITE_ASM1 7 2   %7="dataout=.." and 2="command=dataout..."%
  THEN REWRITE_ASM_THM_TAC double_num 1
```

```
  THEN RM2LAST_TAC

% (2) biu idles until  new_instr is high %
  THEN IMP_RES_TAC biu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_idle 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 3 1 %2 = new_instr(t+1)%
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))
% (3) CPU waits until the response come %
  THEN IMP_RES_TAC cpu_wait_for_response_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM1 36 24        % 36 = ~resp_ready t,
                                   24 = resp_ready(t+1) = resp_ready t %
  THEN REWRITE_ASM 1 2
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))
% (4) APU still idles %
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (REWRITE_RULE [SYM_RULE (el 29 ths)] (el 46 ths)))
                % to get "~start (t+1)" first.
                  46 = "~start t", 29="start(t+1) = start t" %
  THEN IMP_RES_TAC apu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))

%t+1+1 -> t'%
% (1) BIU decodes, sends back response, and starts up the apu.
  first time nondeterministic.......?t'%
  THEN IMP_RES_TAC biu_decode_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_decode 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (REWRITE_RULE [need_read; need_write] (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
        (REWRITE_OTHER_TAC 1))
  THEN RM2LAST_TAC
  THEN POP_ASSUM (\th1. ASSUME_TAC
        ((CONV_RULE DEC_EQ_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
        (CONV_RULE (ONCE_DEPTH_CONV
            INV_dec_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
        (REWRITE_RULE [First] th1))
  THEN POP_ASSUM (\th1.
        STRIP_ASSUME_TAC th1)  % get rid of ?t' and make (A/\B) two
                                 separate assumptions A, B %
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))
```

```
% (2) CPU -- waiting for the response from BIU, induction theroom
      wait_cpu is used here.
      At the end we get cpu_state t' = 1, mem t' = mem (t+1+1)
      and ir t' = ir(t+1+1) %
  THEN IMP_RES_TAC wait_cpu %resolve against (cpu_state t+1 = 1) and
                             (cpu_state t+1+1 = 1)%
  THEN RM3LAST_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil 1
% specialize n%
  THEN POP_ASSUM (\th1.
        ASSUME_TAC (SPEC "((t'-t)-1)-1" th1))
  THEN IMP_RES_TAC num_lemma2 %a numerical lemma %
  THEN REWRITE_ASM 1 2  % to rewrite use the theorem %
  THEN REWRITE_ASM 13 1  % 13 = Stable(resp_ready,F,(t + 1) + 1,t') %
% to get mem and ir right %
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN IMP_RES_TAC wait_cpu2 %resolve against (cpu_state t+1 = 1) and
                             (cpu_state t+1+1 = 1)%
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil2 1
  THEN REWRITE_ASM_THM_TAC StableUntil2 3
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
% specialize n%
  THEN POP_ASSUM (\th1.
        ASSUME_TAC (SPEC "((t'-t)-1)-1" th1))
  THEN POP_ASSUM (\th1. POP_ASSUM (\th2.
        ASSUME_TAC th1
        THEN ASSUME_TAC (SPEC "((t'-t)-1)-1" th2)))
  THEN IMP_RES_TAC num_lemma2 %a numerical lemma %
  THEN REWRITE_ASM1 1 2  % to rewrite use the theorem %
  THEN REWRITE_ASM1 2 4
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM 13 1  % 13 = Stable(resp_ready,F,(t + 1) + 1,t') %
  THEN REWRITE_ASM1 13 2
  THEN RM3LAST_TAC

% (3) APU -- waiting to be started, induction theroomm
      wait_apu used here to get f_ac t' = f_ac t+1+1,
      and f_reg t' = f_reg t+1+1 %
  % to get ~start(t+1+1) first,  29 = start t+1+1 %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
        (REWRITE_OTHER_TAC 29))
  THEN IMP_RES_TAC wait_apu  % resolve against ~start t and ~start(t+1+1) %
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil2 1
```

```
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC StableUntil2 2
   THEN RM3LAST_TAC
   THEN POP_ASSUM (\th1.
         ASSUME_TAC (SPEC "((t'-t)-1)-1" th1))
   THEN POP_ASSUM (\th1.  POP_ASSUM (\th2.
           ASSUME_TAC (SPEC "((t'-t)-1)-1" th2)
           THEN ASSUME_TAC th1))
   THEN IMP_RES_TAC num_lemma2 %get ((t + 1) + 1) + (((t' - t) - 1) - 1) = t'%
   THEN REWRITE_ASM1 1 2  % to rewrite use the theorem %
   THEN RM3LAST_TAC
   THEN REWRITE_ASM1 2 3
   THEN RM3LAST_TAC
   THEN RM3LAST_TAC
   THEN REWRITE_ASM 14 1   % 14 = Stable(start,F,(t + 1) + 1,t') %
   THEN REWRITE_ASM1 14 2
   THEN RM3LAST_TAC

 %t'-->t'+1%
 % (1) CPU goes to state 2 to analyze the response %
   THEN IMP_RES_TAC cpu_wait_for_response_lemma
   THEN RM2LAST_TAC
   THEN POP_TOP_ASSUMP_TAC
   THEN RM2LAST_TAC
   THEN REWRITE_ASM 11 1   % 11 = resp_ready t' %
   THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

 % (2) BIU wait for the operand in state 2 (since ~operand_ready t') %
   THEN IMP_RES_TAC biu_wait_op_lemma
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC biu_wait_op 1
   THEN RM2LAST_TAC
   THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

 % (3) APU idles %
   THEN IMP_RES_TAC apu_idle_lemma
   THEN POP_TOP_ASSUMP_TAC
   THEN RM2LAST_TAC
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC apu_idle 1
   THEN RM2LAST_TAC
   THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))


 % t'+1 --> t'+1+1 %
 % (1) "read" -- correctness of the 4-phase insures that the response
       is put into the datain bus so that the CPU can read it %
   THEN REWRITE_ASM_THM_TAC cir_read_write 91 % 91 = cir_read_write%
   THEN ASSUM_LIST (\ths.
          (ASSUME_TAC (SPEC "t'+1:num" (el 1 ths))))
   THEN RM2LAST_TAC
   THEN REWRITE_ASM 16 1  % 16 = read(t'+1) %
   THEN REWRITE_ASM 17 1  % 17 ="address(t'+1)=1"%
   THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
```

```
  % now we get "datain((t' + 1) + 1) = numtow rep(response(t' + 1))" %
% (2) CPU waits for one cycle for the 4phase %
  THEN IMP_RES_TAC cpu_wait_4phase_lemma
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
% (3) BIU idles %
  THEN IMP_RES_TAC biu_fld_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 11 1
  THEN POP_ASSUM (\th1.
         STRIP_ASSUME_TAC th1)  % get rid of ?t' %
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
% (4) APU idles %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
         (REWRITE_OTHER_TAC 26)) % 26="start(t' + 1) = start t' " %
  THEN IMP_RES_TAC apu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))


% t'+1+1 --> t'+1+1+1 %
% (1) 4-phase "idle" %
  THEN REWRITE_ASM_THM_TAC cir_read_write 117 % 117 = cir_read_write%
  THEN ASSUM_LIST (\ths.
          (ASSUME_TAC (SPEC "(t'+1)+1:num" (el 1 ths))))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 18 1   % 18 ="~read(t'+1+1)"%
  THEN REWRITE_ASM 17 1   % 17 = ~write(t'+1+1) %
  THEN POP_ASSUM (\th1.
         STRIP_ASSUME_TAC th1)  % make (A/\B) 2 assumptions %

% (2) CPU reads the response and puts the data (operand fetched
     from the mem) to the databus %
  THEN IMP_RES_TAC cpu_read_response_lemma
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (REWRITE_RULE [LET_DEF] (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (BETA_RULE (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. (ASSUME_TAC
         (REWRITE_RULE [el 29 ths; double_num] (el 1 ths))))
  THEN RM2LAST_TAC    % 29 = datain(t'+1+1)=... %
  THEN REWRITE_ASM1 63 41     % 41 = response(t'+1)=response t',
                                63 = response t' = read_p %
  THEN REWRITE_ASM1 1 2
  THEN RM3LAST_TAC
  THEN POP_ASSUM (\th1. ASSUME_TAC
         ((CONV_RULE DEC_EQ_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
```

```
        (CONV_RULE (ONCE_DEPTH_CONV
           INV_dec_CONV) th1))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))
% (3) BIU waits in state 3 %
  THEN IMP_RES_TAC biu_fld_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 11 1   % 11 = ~operand_ready (t'+1+1) %
  THEN POP_ASSUM (\th1.
       STRIP_ASSUME_TAC th1)  % get rid of ?t' %
  THEN ASSUM_LIST (\ths.
       (TAC1 (el 1 ths)))

% (4) APU idles %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
       (REWRITE_OTHER_TAC 26)) % 26="start(t'+1+1) = start(t'+1) " %
  THEN IMP_RES_TAC apu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
       (TAC1 (el 1 ths)))


% t'+1+1+1 --> t'+1+1+1+1 %
% (1) "write" -- the coorectness property of the 4-phase protocol
  insures that the data in put into the operand_in cir %
  THEN REWRITE_ASM_THM_TAC cir_read_write 142   %142 = cir_read_write%
  THEN ASSUM_LIST (\ths.
        (ASSUME_TAC (SPEC "((t'+1)+1)+1" (el 1 ths))))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 18 1   % 18 = "~read(t'+1+1+1)"%
  THEN REWRITE_ASM 17 1   % 17 = write(t'+1+1+1) %
  THEN REWRITE_ASM 19 1    % 19="address(t'+1+1+1)=2  (operand_in)"%
  THEN POP_ASSUM (\th1. ASSUME_TAC
       ((CONV_RULE DEC_EQ_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
       (CONV_RULE (ONCE_DEPTH_CONV
          INV_dec_CONV) th1))
  THEN ASSUM_LIST (\ths.
       (TAC1 (el 1 ths)))
  THEN REWRITE_ASM1 22 3
        %22="dataout(t'+3)=fetch rep mem (Addr n ir(t'+3))" and
         3="operand_out(t'+4) = wtofp rep dataout(t'+3)"%

% (2) BIU waits in state 2 %
  THEN IMP_RES_TAC biu_fld_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 4 1   % 4 = ~operand_ready (t'+1+1+1) %
  THEN POP_ASSUM (\th1.
       STRIP_ASSUME_TAC th1)  % get rid of ?t' %
```

```
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))


% (3) APU idles %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
         (REWRITE_OTHER_TAC 19))   %19="start(t'+1+1+1) = start(t'+1+1)"%
  THEN IMP_RES_TAC apu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
% (4) CPU is free now %


% t'+1+1+1+1 --> t''''' %
% (1) BIU put the content of operand_in to f_ac %
% the rest idles or do other things %
  THEN IMP_RES_TAC biu_fld_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 17 1   % 17 = operand_ready (t'+1+1+1+1) %
  THEN POP_ASSUM (\th1.
         STRIP_ASSUME_TAC th1)  % get rid of ?t' %
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

% (2) APU idles -- induction. To get f_reg t''''' = f_reg(t'+1+1+1+1) %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
         (REWRITE_OTHER_TAC 16)) % 16="start(t'+1+1+1+1) = start(t'+1+1+1) " %
  THEN IMP_RES_TAC wait_apu  % resolve against ~start t and ~start(t+1+1) %
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil2 1
  THEN RM2LAST_TAC
  THEN POP_ASSUM (\th1.
         ASSUME_TAC (SPEC "((((t'''''-t')-1)-1)-1)-1" th1))
  THEN IMP_RES_TAC num_lemma3
         %get ((((t+1)+1)+1)+1) + (((((t'''''-t)-1)-1)-1)-1) = t'''''%
```

```
   THEN REWRITE_ASM1 1 2  % to rewrite use the theorem %
   THEN RM3LAST_TAC
   THEN REWRITE_ASM 11 1   % 11 = Stable(start,F,(t'+1+1+1+1),t''''') %

 % To rewrite the goal %
  THEN REWRITE_ASM1 27 6  % to get f_sc t''''' = wtofp rop (...) %
  THEN ONCE_REWRITE_TAC[FLD]
  THEN ONCE_REWRITE_TAC [LET_DEF]
  THEN BETA_TAC
  THEN REWRITE_TAC[FPCstate]
  THEN EXISTS_TAC "t''''':num"
  THEN ASM_REWRITE_TAC[]
 );;



close_theory();;
quit();;




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

File: fstr_correct.ml
Author: Jing Pan
Date: Feb. 1991
Purpose:        Verification of FPC top level against top level
                spec of the BIU, APU, and CPU_service, in the
                case of an FSTR instruction.

Theories Used:  aux, biu_top, biu_lemma, apu_top, apu_lemma,
                cpu_service, service_lemma, fpc_top, induc, a.

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


loadf '/csgrad/panj/holdir/init.ml';;

loadf 'abstract.ml';;
loadf 'tactics.ml';;

system '/bin/rm -f fstr_correct.th';;

set_flag ('sticky', true);;

new_theory 'fstr_correct';;


loadf 'aux_defs.ml';;

map new_parent ['aux'; 'biu_top'; 'biu_lemma'; 'apu_top';
                'apu_lemma'; 'cpu_service'; 'service_lemma';
```

```
              'fpc_top'; 'induc'; 'a'];;

autoload_defs_and_thms 'biu_top';;
autoload_defs_and_thms 'biu_lemma';;
autoload_defs_and_thms 'apu_top';;
autoload_defs_and_thms 'apu_lemma';;

autoload_defs_and_thms 'cpu_service';;
autoload_defs_and_thms 'service_lemma';;
autoload_defs_and_thms 'fpc_top';;
autoload_defs_and_thms 'aux';;
autoload_defs_and_thms 'induc';;
autoload_defs_and_thms 'a';;


let rep_ty = abstract_type 'interface' 'fetch';;



new_theory_obligations [
  "! (x:num) (rep:^rep_ty). ((wtonum rep (numtow rep x)) = x)";
  "! (x:fp) (rep:^rep_ty). ((wtofp rep (fptow rep x)) = x)";
];;


let double_num = prove_thm
  ('double_num',
   "! (x:num) (rep:^rep_ty). ((wtonum rep (numtow rep x)) = x)",
    REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;


let double_fp = prove_thm
  ('double_fp',
   "! (x:fp) (rep:^rep_ty). ((wtofp rep (fptow rep x)) = x)",
    REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

map loadf ['digit'; 'decimal'];;



%-------------------------------------------------
             FSTR_Correct

  Tp prove the correctness of FSTORE, upon the interaction
among the BIU, APU and CPU.
-------------------------------------------------%

let FSTR_Correct = prove_thm
 ('FSTR_Correct',
 "! (rep : ^rep_ty) (response:num->num) (operand_out:num->fp)
     (decode_reg:num->num) (biu_state:num->num)
     (start : num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (operand_ready:num->bool) (resp_ready:num->bool)
     (f_ac : num->fp) (f_reg : num->num->fp)
     (cw sw : num->bool#bool#bool#bool)
     (c_ac:num->num) (c_reg:num->num->num) (mem:num->*memory)
```

```
      (ir:num->num) (cpu_state:num->num) (address:num->num)
      (read write :num->bool) (datain dataout:num->*wordn) (n:num).

   (biu_top_cpu n rep  (response, operand_out, decode_reg, resp_ready,
               f_ac, biu_state, start)
              (command, condition, control, operand_in, done, new_instr,
               operand_ready)  /\

    apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
          (start, decode_reg)       /\

    cpu_service n rep (c_ac, c_reg, mem, ir, dataout, address, read, write,
            cpu_state) (resp_ready, datain)       /\

    (cir_read_write rep address read write datain dataout
                    command response operand_in operand_out condition
                    control new_instr operand_ready))
    ==>
    (!t.  ((Opc n (ir t) = 1) /\  % FSTR %
          (~(start t)) /\
          (~(new_instr t)) /\
          (~(resp_ready t)) /\
          (~(read t)) /\
          (biu_state t = 0) /\
          (cpu_state t = 0)) ==>
          (? (t':num).
              ((f_ac t', f_reg t') =
                FPCstate (FSTR n rep (f_ac t, f_reg t, c_ac t,
                                c_reg t, mem t, ir t)))))",

%t->t+1%
*(REPEAT STRIP_TAC
%(1) APU: idle %
  THEN IMP_RES_TAC apu_idle_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
% (2) BIU -- idle, waiting for the new_instr signal%
  THEN IMP_RES_TAC biu_idle_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_idle 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 11 1  % 11= "~new_instr t"%
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))
%(3) CPU -- start up the communication by sending the instr%
  THEN IMP_RES_TAC cpu_begin_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC cpu_begin 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

%t+1 --> t+1+1%
%(1) correctness of the 4-phase handshaking,, "write": cir_read_write --
     get "command (t+1+1) = ir t" %
  THEN REWRITE_ASM_THM_TAC cir_read_write 29 % 29 = cir_read_write%
```

```
  THEN ASSUM_LIST (\ths.
         (ASSUME_TAC (SPEC "t+1" (el 1 ths))))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM1 25 4 %25="read t, 4="read(t+1)=read t"%
  THEN REWRITE_ASM1 1 2
  THEN RM3LAST_TAC
  THEN REWRITE_ASM 4 1  % 4=write(t+1) %
  THEN REWRITE_ASM 6 1    % 6="address(t+1)=0"%
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))
  THEN RM3LAST_TAC
  THEN REWRITE_ASM1 7 2   %7="dataout=.." and 2="command=dataout..."%
  THEN REWRITE_ASM_THM_TAC double_num 1
  THEN RM2LAST_TAC

% (2) biu idles until  new_instr is high %
  THEN IMP_RES_TAC biu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_idle 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 3 1 %2 = new_instr(t+1)%
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))
% (3) CPU waits until the response come %
  THEN IMP_RES_TAC cpu_wait_for_response_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM1 36 24        % 36 = "resp_ready t,
                                   24 = resp_ready(t+1) = resp_ready t %
  THEN REWRITE_ASM 1 2
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))
% (4) APU still idles %
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (REWRITE_RULE [SYM_RULE (el 29 ths)] (el 46 ths)))
                % to get ""start (t+1)" first.
                  46 = ""start t", 29="start(t+1) = start t" %
  THEN IMP_RES_TAC apu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))

%t+1+1 -> t'%
% (1) BIU decodes, sends back response, and starts up the apu.
  first time nondeterministic.......?t'%
  THEN IMP_RES_TAC biu_decode_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_decode 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (REWRITE_RULE [need_read; need_write] (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
        (REWRITE_OTHER_TAC 1))
```

```
  THEN RM2LAST_TAC
  THEN POP_ASSUM (\th1. ASSUME_TAC
       ((CONV_RULE DEC_EQ_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
       (CONV_RULE (ONCE_DEPTH_CONV
           INV_dec_CONV) th1))
  THEN POP_ASSUM (\th1. ASSUME_TAC
       (REWRITE_RULE [First] th1))
  THEN POP_ASSUM (\th1.
       STRIP_ASSUME_TAC th1)  % get rid of ?t' and make (A/\B) two
                                 separate assumptions A, B %
  THEN ASSUM_LIST (\ths.
       (TAC1 (el 1 ths)))

% (2) CPU -- waiting for the response from BIU, induction theroem
     wait_cpu is used here.
     At the end we get cpu_state t' = 1, mem t' = mem (t+1+1)
     and ir t' = ir(t+1+1) %
  THEN IMP_RES_TAC wait_cpu %resolve against (cpu_state t+1 = 1) and
                            (cpu_state t+1+1 = 1)%
  THEN RM3LAST_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil 1
% specialize n%
  THEN POP_ASSUM (\th1.
       ASSUME_TAC (SPEC "((t'-t)-1)-1" th1))
  THEN IMP_RES_TAC num_lemma2 %a numerical lemma %
  THEN REWRITE_ASM 1 2  % to rewrite use the theorem %
  THEN REWRITE_ASM 13 1  % 13 = Stable(resp_ready,F,(t + 1) + 1,t') %
% to get mem and ir right %
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN IMP_RES_TAC wait_cpu2 %resolve against (cpu_state t+1 = 1) and
                            (cpu_state t+1+1 = 1)%
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil2 1
  THEN REWRITE_ASM_THM_TAC StableUntil2 3
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
% specialize n%
  THEN POP_ASSUM (\th1.
       ASSUME_TAC (SPEC "((t'-t)-1)-1" th1))
  THEN POP_ASSUM (\th1. POP_ASSUM (\th2.
       ASSUME_TAC th1
       THEN ASSUME_TAC (SPEC "((t'-t)-1)-1" th2)))
  THEN IMP_RES_TAC num_lemma2 %a numerical lemma %
  THEN REWRITE_ASM1 1 2  % to rewrite use the theorem %
  THEN REWRITE_ASM1 2 4
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM 13 1  % 13 = Stable(resp_ready,F,(t + 1) + 1,t') %
  THEN REWRITE_ASM1 13 2
  THEN RM3LAST_TAC
```

```
% (3) APU -- waiting to be started, induction thereomm
      wait_apu used here to get f_ac t' = f_ac t+1+1,
      and f_reg t' = f_reg t+1+1 %
  % to get ~start(t+1+1) first,  29 = start t+1+1 %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
         (REWRITE_OTHER_TAC 29))
  THEN IMP_RES_TAC wait_apu  % resolve against ~start t and ~start(t+1+1) %
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil2 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil2 2
  THEN RM3LAST_TAC
  THEN POP_ASSUM (\th1.
        ASSUME_TAC (SPEC "((t'-t)-1)-1" th1))
  THEN POP_ASSUM (\th1.  POP_ASSUM (\th2.
          ASSUME_TAC (SPEC "((t'-t)-1)-1" th2)
          THEN ASSUME_TAC th1))
  THEN IMP_RES_TAC num_lemma2 %get ((t + 1) + 1) + (((t' - t) - 1) - 1) = t'%
  THEN REWRITE_ASM1 1 2  % to rewrite use the theorem %
  THEN RM3LAST_TAC
  THEN REWRITE_ASM1 2 3
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM 14 1   % 14 = Stable(start,F,(t + 1) + 1,t') %
  THEN REWRITE_ASM1 14 2
  THEN RM3LAST_TAC
%(4) CPU induction -- to get ~read t' and ~write t'%
  THEN IMP_RES_TAC wait_cpu3 %resolve against (cpu_state t+1 = 1) and
                              (cpu_state t+1+1 = 1)%
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM_THM_TAC StableUntil3 1
  THEN REWRITE_ASM_THM_TAC StableUntil3 3
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
% specialize n%
  THEN POP_ASSUM (\th1.
        ASSUME_TAC (SPEC "(((t'-t)-1)-1)-1" th1))
  THEN POP_ASSUM (\th1. POP_ASSUM (\th2.
        ASSUME_TAC th1
        THEN ASSUME_TAC (SPEC "(((t'-t)-1)-1)-1" th2)))
  THEN IMP_RES_TAC num_lemma4 %a numerical lemma %
  THEN REWRITE_ASM1 1 2  % to rewrite use the theorem %
  THEN REWRITE_ASM1 2 4
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN RM3LAST_TAC
  THEN REWRITE_ASM 18 1  % 18 = Stable(resp_ready,F,(t + 1) + 1,t') %
  THEN REWRITE_ASM1 18 2
  THEN RM3LAST_TAC
```

```
%t'-->t'+1%
% (1) 4-phase idles %
  THEN REWRITE_ASM_THM_TAC cir_read_write 72 % 72 = cir_read_write%
  THEN ASSUM_LIST (\ths.
         (ASSUME_TAC (SPEC "t':num" (el 1 ths))))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 2 1   % 2 ="~read t'"%
  THEN REWRITE_ASM 3 1   % 3 = ~write t' %
  THEN POP_ASSUM (\th1.
        STRIP_ASSUME_TAC th1)  % make (A/\B) 2 assumptions %

% (2) CPU goes to state 2 to analyze the result %
  THEN IMP_RES_TAC cpu_wait_for_response_lemma
  THEN RM2LAST_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 15 1   % 15 = resp_ready t' %
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))

% (3) BIU wait in state 0 -- 1st time %
  THEN IMP_RES_TAC biu_idle_lemma
  THEN RM2LAST_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_idle 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 12 1  %12=~new_instr t'%
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))

% (4) APU idles %
  THEN IMP_RES_TAC apu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))


% t'+1 --> t'+1+1 %
% (1) "read" -- correctness of the -phase insures that the response
     is put into the datain bus so that the CPU can read it %
  THEN REWRITE_ASM_THM_TAC cir_read_write 95 % 95 = cir_read_write%
  THEN ASSUM_LIST (\ths.
         (ASSUME_TAC (SPEC "t'+1:num" (el 1 ths))))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 16 1  % 16 = read(t'+1) %
  THEN REWRITE_ASM 17 1  % 17 ="address(t'+1)=1"%
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))

  % now we get "datain((t' + 1) + 1) = numtow rep(response(t' + 1))" %
% (2) CPU waits for one cycle for the 4phase %
  THEN IMP_RES_TAC cpu_wait_4phase_lemma
  THEN RM2LAST_TAC
```

```
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))
% (3) BIU idles in state 0 -- 2nd time %
  THEN IMP_RES_TAC biu_idle_lemma
  THEN RM2LAST_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_idle 1
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 12 1  %12=~new_instr(t'+1)%
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))


% (4) APU idles %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
        (REWRITE_OTHER_TAC 25)) % 25="start(t' + 1) = start t' " %
  THEN IMP_RES_TAC apu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))


% t'+1+1 --> t'+1+1+1 %
% (1) 4-phase "idle" %
  THEN REWRITE_ASM_THM_TAC cir_read_write 120 % 120 = cir_read_write%
  THEN ASSUM_LIST (\ths.
         (ASSUME_TAC (SPEC "(t'+1)+1:num" (el 1 ths))))
  THEN RM2LAST_TAC
  THEN REWRITE_ASM 17 1   % 17 ="~read(t'+1+1)"%
  THEN REWRITE_ASM 16 1   % 16 = ~write(t'+1+1) %
  THEN POP_ASSUM (\th1.
        STRIP_ASSUME_TAC th1)  % make (A/\B) 2 assumptions %

% (2) CPU reads the response and raise the read line to read the
operand now is on the datain bus %
  THEN IMP_RES_TAC cpu_read_response_lemma
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (REWRITE_RULE [LET_DEF] (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
        (BETA_RULE (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. (ASSUME_TAC
        (REWRITE_RULE [el 28 ths; double_num] (el 1 ths))))
  THEN RM2LAST_TAC    % 28 = datain(t'+1+1)=... %
  THEN REWRITE_ASM1 66 40     % 40 = response(t'+1)=response t',
                                66 = response t' = write_p %
  THEN REWRITE_ASM1 1 2
  THEN RM3LAST_TAC
  THEN ASSUM_LIST (\ths. (ASSUME_TAC
        (REWRITE_RULE [read_p; write_p] (el 1 ths))))
```

```
   THEN POP_ASSUM (\th1. ASSUME_TAC
          ((CONV_RULE DEC_EQ_CONV) th1))
   THEN POP_ASSUM (\th1. ASSUME_TAC
          (CONV_RULE (ONCE_DEPTH_CONV
              INV_dec_CONV) th1))
   THEN RM2LAST_TAC
   THEN ASSUM_LIST (\ths.
          (TAC1 (el 1 ths)))

 % (3) BIU waits in state 0 -- 3rd time %
   THEN IMP_RES_TAC biu_idle_lemma
   THEN POP_TOP_ASSUMP_TAC
   THEN POP_TOP_ASSUMP_TAC
   THEN POP_TOP_ASSUMP_TAC
   THEN POP_TOP_ASSUMP_TAC
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC biu_idle 1  % the 5th one %
   THEN RM2LAST_TAC
   THEN REWRITE_ASM 13 1  %13="new_instr(t'+1+1)%
   THEN ASSUM_LIST (\ths.
          (TAC1 (el 1 ths)))

 % (4) APU idles %
   THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
          (REWRITE_OTHER_TAC 26))   % 26="start(t'+1+1) = start(t'+1)" %
   THEN IMP_RES_TAC apu_idle_lemma
   THEN POP_TOP_ASSUMP_TAC
   THEN POP_TOP_ASSUMP_TAC
   THEN POP_TOP_ASSUMP_TAC
   THEN POP_TOP_ASSUMP_TAC
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC apu_idle 1
   THEN RM2LAST_TAC
   THEN ASSUM_LIST (\ths.
          (TAC1 (el 1 ths)))


 % t'+1+1+1 --> t'+1+1+1+1 %
 % (1) "read" -- the correctness property of the 4-phase protocol
   insures that the data in operand_in cir is put on datain bus%
   THEN REWRITE_ASM_THM_TAC cir_read_write 145   %145 = cir_read_write%
   THEN ASSUM_LIST (\ths.
          (ASSUME_TAC (SPEC "((t'+1)+1)+1" (el 1 ths))))
   THEN RM2LAST_TAC
   THEN REWRITE_ASM 17 1   % 17 = "read(t'+1+1+1)"%
   THEN REWRITE_ASM 16 1   % 16 = "write(t'+1+1+1) %
   THEN REWRITE_ASM 18 1    % 18="address(t'+1+1+1)=2  (operand_in)"%
   THEN POP_ASSUM (\th1. ASSUME_TAC
          ((CONV_RULE DEC_EQ_CONV) th1))
   THEN POP_ASSUM (\th1. ASSUME_TAC
          (CONV_RULE (ONCE_DEPTH_CONV
              INV_dec_CONV) th1))
   THEN ASSUM_LIST (\ths.
          (TAC1 (el 1 ths)))

 % (2) CPU wait one cycle%
   THEN IMP_RES_TAC cpu_wait_read_lemma
   THEN RM2LAST_TAC
   THEN REWRITE_ASM_THM_TAC cpu_wait_read 1
```

```
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

% (3) BIU waits in state 0 -- 4th time %
  THEN IMP_RES_TAC biu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC biu_idle 1  % the 6th one %
  THEN RM2LAST_TAC
  THEN e(REWRITE_ASM 12 1  %12="new_instr(t'+1+1+1)%
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

% (3) APU idles %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
         (REWRITE_OTHER_TAC 26))   %26="start(t'+1+1+1) = start(t'+1+1)"%
  THEN IMP_RES_TAC apu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))


% t'+1+1+1+1 --> t'+1+1+1+1+1%
% (1) CPU stores the value on data bus to the memory %
  THEN IMP_RES_TAC cpu_put_data_lemma
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC cpu_put_data 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (REWRITE_RULE [LET_DEF] (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths. ASSUME_TAC
         (BETA_RULE (el 1 ths)))
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
         (TAC1 (el 1 ths)))

% (2) BIU idles in state 0 %
% (3) APU idles %
  THEN ASSUM_LIST (\ths.  %rewrite use all other assumptions%
         (REWRITE_OTHER_TAC 16))   %16="start(t'+1+1+1+1) = start(t'+1+1+1)"%
  THEN IMP_RES_TAC apu_idle_lemma
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
  THEN POP_TOP_ASSUMP_TAC
```

```
  THEN POP_TOP_ASSUMP_TAC
  THEN RM2LAST_TAC
  THEN REWRITE_ASM_THM_TAC apu_idle 1
  THEN RM2LAST_TAC
  THEN ASSUM_LIST (\ths.
        (TAC1 (el 1 ths)))

% To rewrite the goal %
 THEN REWRITE_ASM1 27 6  % to get f_sc t''''' = wtofp rop (...) %
 THEN ONCE_REWRITE_TAC[FSTR]
 THEN ONCE_REWRITE_TAC [LET_DEF]
 THEN BETA_TAC
 THEN REWRITE_TAC[FPCstate]
 THEN EXISTS_TAC "((((t'+1)+1)+1)+1)+1:num"
 THEN ASM_REWRITE_TAC[]
);;



close_theory();;
quit();;




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

File:           fpc_correct.ml
Author:         Jing Pan
Date:           March. 1991
Purpose:        Verification of FPC top level against top level
                spec of the BIU, APU, and CPU_service.

Theories Used:  aux, fpc_top, f_arith_correct.ml fld_correct.ml
                fstr_correct.ml

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


loadf '/csgrad/panj/holdir/init.ml';;

loadf 'abstract.ml';;
loadf 'tactics.ml';;

system '/bin/rm -f fpc_correct.th';;

set_flag ('sticky', true);;

new_theory 'fpc_correct';;

loadf 'aux_defs.ml';;

map new_parent ['aux'; 'time_abs'; 'fpc_top'; 'f_arith_correct';
                'fld_correct'; 'fstr_correct'];;
```

```
autoload_defs_and_thms `fpc_top`;;
autoload_defs_and_thms `aux`;;
autoload_defs_and_thms `f_arith_correct`;;
autoload_defs_and_thms `fld_correct`;;
autoload_defs_and_thms `fstr_correct`;;
autoload_defs_and_thms `time_abs`;;


let rep_ty = abstract_type `interface` `fetch`;;

new_theory_obligations [
  "! (x:num) (rep:^rep_ty). ((wtonum rep (numtow rep x)) = x)";
  "! (x:fp) (rep:^rep_ty). ((wtofp rep (fptow rep x)) = x)";
];;

map loadf [`digit`; `decimal`];;



%-------------------------------------------------------------
                    Initial_State
  Specify the initial states of the CPU, BIU, APU and
the correctness of the 4 phase protocol.
-------------------------------------------------------------%
let Initial_State = new_definition
  (`Initial_State`,
  "! (start new_instr resp_ready read:num->bool)
     (biu_state cpu_state:num->num) t.
   Initial_State (start, new_instr, resp_ready, read, biu_state,
                  cpu_state, t) =
        ((^(start t)) /\
        (^(new_instr t)) /\
        (^(resp_ready t)) /\
        (^(read t)) /\
        (biu_state t = 0) /\
        (cpu_state t = 0))"
);;



%-------------------------------------------------------------
                    ValidOpcode
  Specify the valid opcodes
-------------------------------------------------------------%
let ValidOpcode = new_definition
  (`ValidOpcode`,
  "! (rep:^rep_ty) n ir . ValidOpcode n rep ir =
     ((Opc n ir = 0) \/
      (Opc n ir = 2) \/
      (Opc n ir = 3) \/
      (Opc n ir = 4) \/
      (Opc n ir = 5))"
);;



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

```
                FPC_NextState_Correct

  To prove the correctness of NextState_fpc, upon
interaction among the CPU, BIU and APU.
-------------------------------------------------------%
let NextState_Correct = prove_thm
  ('NextState_Correct',
  "! (rop : ^rop_ty) (response:num->num) (operand_out:num->fp)
     (decode_reg:num->num) (biu_state:num->num)
     (start : num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (operand_ready:num->bool) (resp_ready:num->bool)
     (f_ac : num->fp) (f_reg : num->num->fp)
     (cw sw : num->bool#bool#bool#bool)
     (c_ac:num->num) (c_reg:num->num->num) (mem:num->*memory)
     (ir:num->num) (cpu_state:num->num) (address:num->num)
     (read write :num->bool) (datain dataout:num->*wordn) (n:num).

  (biu_top_cpu n rop  (response, operand_out, decode_reg, resp_ready,
              f_ac, biu_state, start)
             (command, condition, control, operand_in, done, new_instr,
              operand_ready)  /\

   apu_top_cpu n rop (f_ac, f_reg, cw, sw, done)
         (start, decode_reg)       /\

   cpu_service n rop (c_ac, c_reg, mem, ir, dataout, address, read, write,
           cpu_state) (resp_ready, datain)       /\

   (cir_read_write rop address read write datain dataout
                  command response operand_in operand_out condition
                  control new_instr operand_ready))
  ==>
  (!t.  (Initial_State (start, new_instr, resp_ready, read, biu_state,
                 cpu_state, t) /\
         ValidOpcode n rop (ir t)) ==>
        (? (t':num).
            ((f_ac t', f_reg t') =
              FPCstate (NextState_fpc n rop (ir t) (f_ac t, f_reg t, c_ac t,
                              c_reg t, mem t, ir t)))))",
ONCE_REWRITE_TAC[Initial_State]
THEN REPEAT STRIP_TAC
THEN ONCE_REWRITE_TAC[NextState_fpc]
THEN POP_ASSUM (\th1. DISJ_CASES_THEN STRIP_ASSUME_TAC
                        (REWRITE_RULE [ValidOpcode] th1))
THEN ONCE_ASM_REWRITE_TAC[]
THEN DEC_EQ_TAC
THEN CONV_TAC (ONCE_DEPTH_CONV INV_dec_CONV)
THENL [ %0%
   IMP_RES_TAC FLD_Correct
; %2%
   IMP_RES_TAC FADD_Correct
; %3%
   IMP_RES_TAC FSUB_Correct
; %4%
   IMP_RES_TAC FMUL_Correct
; %5%
   IMP_RES_TAC FDIV_Correct
]
```

```
  THEN RES_TAC
  THEN ASM_REWRITE_TAC□
);;


let fpc_top3 = new_definition
  ('fpc_top3',
 "! (rep : ^rep_ty) (f_ac :num->fp) (f_reg :num->num->fp) (c_ac :num->num)
     (c_reg :num->num->num) (mem:num->*memory) (ir :num->num)
     (n:num) (start new_instr resp_ready read:num->bool)
     (biu_state cpu_state :num->num) .
  fpc_top3 n rep (f_ac, f_reg, c_ac, c_reg, mem, ir)
           (start, new_instr, resp_ready, read, biu_state, cpu_state) =
      ! t. (Initial_State (start, new_instr, resp_ready, read, biu_state,
                  cpu_state, t) /\
          ValidOpcode n rep (ir t)) ==>
            ? t'. (f_ac t', f_reg t') =
                  (FPCstate (NextState_fpc n rep (ir t)
                       (f_ac t, f_reg t, c_ac t, c_reg t, mem t, ir t)))"
);;



%-----------------------------------------------------------------
                      FPC_Correct
  To prove the correctness of fpc_top, upon
interaction among the CPU, BIU and APU.
-----------------------------------------------------------------%
let FPC_Correct = prove_thm
  ('FPC_Correct',
   "! (rep : ^rep_ty) (response:num->num) (operand_out:num->fp)
     (decode_reg:num->num) (biu_state:num->num)
     (start : num->bool) (command:num->num) (condition:num->num)
     (control:num->num) (operand_in:num->fp) (done:num->bool)
     (new_instr : num-> bool) (operand_ready:num->bool) (resp_ready:num->bool)
     (f_ac : num->fp) (f_reg : num->num->fp)
     (cw sw : num->bool#bool#bool#bool)
     (c_ac:num->num) (c_reg:num->num->num) (mem:num->*memory)
     (ir:num->num) (cpu_state:num->num) (address:num->num)
     (read write :num->bool) (datain dataout:num->*wordn) (n:num).

  (biu_top_cpu n rep  (response, operand_out, decode_reg, resp_ready,
               f_ac, biu_state, start)
              (command, condition, control, operand_in, done, new_instr,
               operand_ready)  /\

   apu_top_cpu n rep (f_ac, f_reg, cw, sw, done)
         (start, decode_reg)       /\

   cpu_service n rep (c_ac, c_reg, mem, ir, dataout, address, read, write,
            cpu_state) (resp_ready, datain)       /\

   (cir_read_write rep address read write datain dataout
                   command response operand_in operand_out condition
                   control new_instr operand_ready))
                  ==>
   (fpc_top3 n rep (f_ac, f_reg, c_ac, c_reg, mem, ir)
          (start, new_instr, resp_ready, read, biu_state, cpu_state))",
  ONCE_REWRITE_TAC[fpc_top3]
```

```
  THEN REPEAT STRIP_TAC
  THEN IMP_RES_TAC NextState_Correct
  THEN POP_ASSUM (\th1. ASSUME_TAC
       (SPEC "t:num" th1))
  THEN ASSUM_LIST (\ths. ASSUME_TAC
       (REWRITE_RULE [(el 5 ths); (el 6 ths)] (el 1 ths)))
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN RM2LAST_TAC
  THEN ASM_REWRITE_TAC[]
);;


close_theory();;
quit();;
```

NASA

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA-CR - 187547		

4. Title and Subtitle
Toward a Formal Verification of a Floating-Point Coprocessor and its composition with a Central Processing Unit

5. Report Date
August 5, 1991

6. Performing Organization Code

7. Author(s)
Jing Pan, Karl Levitt, and Gerald C. Cohen

8. Performing Organization Report No.

10. Work Unit No.
505-64-10-07

9. Performing Organization Name and Address
Boeing Military Airplanes
P.O. Box 3707, M/S 7J-24
Seattle, WA 98124-2207

11. Contract or Grant No.
NAS1-18586

13. Type of Report and Period Covered
Contractor Report

12. Sponsoring Agency Name and Address
NASA Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes
Langley Technical Monitor: Sally C. Johnson
Task 3, Subtask 8 Report

16. Abstract
This report presents work underway to formally specify and verify a floating-point coprocessor based on the MC68881. Our work uses the HOL verification system developed at Cambridge University. The coprocessor consists of two independent units: the bus interface unit to communicate with the cpu and the arithmetic processing unit to perform the actual calculation. Reasoning about the interaction and synchronization among processes using higher order logic is demonstrated.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Floating-Point Coprocessor MC68881 HOL Synchronization	Unclassified - Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	105	A06

NASA FORM 1626 OCT 86