Technical Report

UCAM-CL-TR-42
ISSN 1476-2986

Number 42

Proving a computer correct with the LCF_LSM hardware verification system

Mike Gordon

September 1983

ISSN 1476-2986

Proving a Computer Correct
with the LCF_LSM Hardware Verification System

Mike Gordon
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG

Abtract

A machine-generated correctness proof of a simple computer is described.

At the machine code level the computer has a memory and two registers: a 13-bit program counter and a 16-bit accumulator. There are 8 machine instructions: halt, unconditional jump, jump when the accumulator contains 0, add contents of a memory location to accumulator, subtract contents of a location from accumulator, load accumulator from memory, store contents of accumulator in memory, and skip. The machine can be interrupted by pushing a button on its front panel.

The implementation which we prove correct has 6 data registers, an ALU, a memory, and a microcode controller. This controller consists of a ROM holding 26 30-bit microinstructions, a microprogram counter, and some combinational microinstruction decode logic.

Formal specifications of the target and host machines are given, and we describe the main steps in proving that the host correctly fetches, decodes and executes machine instructions.

The utility of LCF_LSM for general manipulation is illustrated in two appendices. In Appendix 1 we show how to code a microassembler. In Appendix 2 we use the LCF_LSM inference rules to design a hard-wired controller equivalent to the original microcoded one.

N.B. This report should be read in conjunction with *LCF_LSM: A system for specifying and verifying hardware*. University of Cambridge computer laboratory technical report Number 41.

Proving a Computer Correct

with the LCF_LSM Hardware Verification System

Introduction

The example computer used in this report is taken from [Gordon1]; its specification and verification are done using the LCF_LSM system [Gordon2]. Familiarity with LCF_LSM is assumed.

As the proof of correctness is quite long, we will not give a complete transcript of it. We will, however, outline the main steps. The omitted details are mostly pure LCF manipulations. We hope eventually to design and implement derived inference rules which will perform these LCF inferences automatically.

Our aim in presenting the proof is to give an idea of what is involved in verifying a non-trivial system. It is not expected that the reader will follow all the details, but merely that he will get a feel for their rough shape. In particular we hope to demonstrates that:

1. Non-trivial hardware can be proved correct using existing techniques.

2. Machine assistance is essential for such proofs.

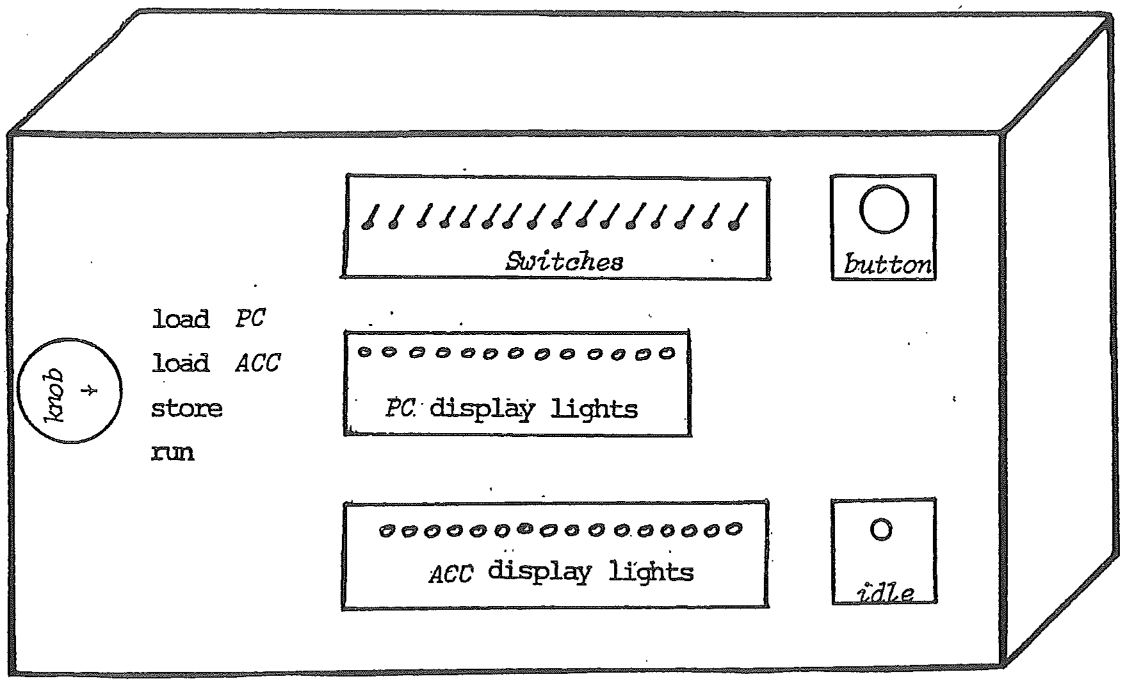
The entire specification and verification described here took several months, but this includes some extending and debugging of LCF_LSM (necessary, as this was our first big example). I estimate that it would take me two to four weeks to do another similar exercise now. The complete proof requires several hours CPU time on a 2 megabyte Vax750. I found it necessary to prove some of the bigger lemmas (e.g. *DERIVED_HOST_EQN* below) in batch mode overnight. These times seem to me to be reasonable in the context of the normal timescales associated with the design and implementation of harware systems.

If a small change were made to the computer (e.g. adding a separate incrementer for the program counter, or changing a few microinstructions), then the proof could be redone in a day or so. The main overhead of setting up the various theories, and creating ML programs to do the necessary inferences, need only be done once. The correctnes of small changes can be verified by editing these theories and programs, and then redoing the entire proof as a batch job.

To illustrate how the LCF_LSM system can be used for other things besides formal proof of correctness, we describe in appendices how a microassembler can be expressed in ML, and how a hard-wired controller for the computer can be systematically derived. We believe that LCF_LSM provides a good programming environment for developing many kinds of CAD/CAM tools. Futher evidence for this is the use of ML for mask-level graphical design in the Sticks & Stones system [Cardelli], this approach is entirely consistent with the framework used here; Cardelli's ideas could be nicely integrated into LCF_LSM.

Specification of The Target Machine

The front panel of the computer looks like:



This computer has two registers: the program counter *PC* which is 13 bits wide, and the accumulator *ACC* which is 16 bits wide. It has a random access memory with 13-bit addresses, each of which points to a 16-bit word (thus the memory holds 8192 16-bit words).

On the front panel there is a four position knob which determines what happens when the button on the right of the panel is pressed. There are three sets of lights: thirteen *PC* display lights which show the contents of the program counter; sixteen *ACC* display lights which show the contents of the accumulator, and the idle light which is on when the computer is idling (i.e. not executing a program). There are also sixteen two-position switches which are used for inputting data.

We shall refer to the four positions of the knob as *0*, *1*, *2* and *3*. The effect of pushing the button when the idle light is on is:

knob=0	The word determined by the state of the thirteen rightmost switches is loaded into the program counter.
knob=1	The word determined by the state of the sixteen switches is loaded into the accumulator.
knob=2	The contents of the accumulator is stored in the memory at the location held in the program counter.
knob=3	The program stored in memory is executed starting at the location in the program counter. The idle light will go off, and stay off until execution stops; this happens either when a halt instruction is reached, or when an interrupt is generated by pushing the button.

Each instruction has a three bit opcode part *op* and a thirteen bit address part *addr*. The format is:

```
|--op-|----------addr----------|
```

The eight instuctions are:

`|0|0|0|----------addr----------|`	*HALT*	Stops exececution
`|0|0|1|----------addr----------|`	*JMP L*	Jump to *addr*
`|0|1|0|----------addr----------|`	*JZR L*	Jump to *addr* if *ACC=0*
`|0|1|1|----------addr----------|`	*ADD L*	Add contents of *addr* to *ACC*
`|1|0|0|----------addr----------|`	*SUB L*	Subtract contents of *addr* from *add*
`|1|0|1|----------addr----------|`	*LD L*	Load contents of *addr* into *ACC*
`|1|1|0|----------addr----------|`	*ST L*	Store contents of *ACC* in *addr*
`|1|1|1|----------addr----------|`	*SKIP*	Skip to next instruction

In order to formally specify the computer in LSM it is convenient to define a few auxiliary constants:

PAD13_16	*:word13->word16*	Pads a 13-bit word with 3 0s at left
CUT16_13	*:word16->word13*	Extracts 13 rightmost bits
OPCODE	*:word16->word3*	Extracts 3 leftmost bits
INC13	*:word13->word13*	Adds 1 to a 13-bit word
INC16	*:word16->word16*	Adds 1 to a 16-bit word
ADD16	*:word16->word16->word16*	Addition on 16-bit words
SUB16	*:word16->word16->word16*	Subtraction on 16-bit words
ISZERO16	*:word16->bool*	Test for 0 on 16-bit words

The following axioms are needed:

```
CUT_PAD          !w:word13.  CUT16_13(PAD13_16 w) == w
CUT_INC_PAD      !w:word13.  CUT16_13(INC16(PAD13_16 w)) == INC13 w
```

We can now specify the computer using two constants:

```
NEXT          :mem13_16#word13#word16 -> mem13_16#word13#word16#bool
COMPUTER      :mem13_16#word13#word16#bool -> dev
```

The function *NEXT* defines a single step of the machine, and hence specifies the semantics of the machine instructions. *NEXT(m,w1,w2)* gives the state after executing the instruction stored in *m* at address *w1* when the accumulator holds *w2*, and is defined by the following axiom:

```
NEXT(m,w1,w2) ==
 let op   = VAL3(OPCODE(FETCH13 m w1)) in
 let addr = CUT16_13(FETCH13 m w1)      in
  (op=0 -> (m, w1, w2, T) |
   op=1 -> (m, addr, w2, F) |
   op=2 -> (VAL16 w2 = 0 -> (m, addr, w2, F) | (m, INC13 w1, w2, F)) |
   op=3 -> (m, INC13 w1, ADD16 w2 (FETCH13 m addr), F) |
   op=4 -> (m, INC13 w1, SUB16 w2 (FETCH13 m addr), F) |
   op=5 -> (m, INC13 w1, FETCH13 m addr, F) |
   op=6 -> (STORE13 addr w2 m, INC13 w1, w2, F) |
           (m, INC13 w1, w2, F))
```

COMPUTER(m,w1,w2,t) is the behaviour of the computer when the memory is *m*, the program counter is *w1*, the accumulator is *w2* and the run/idle status is *t* (*t=T* means the computer is idling; *t=F* means it is executing). The axiom defining this behaviour is:

```
COMPUTER(m,w1,w2,t) ==
 dev{knob,button,switches,pc,acc,idle}.
    {pc=w1, acc=w2, idle=t};
    COMPUTER(t ->
               (button ->
                ((VAL2 knob = 0) -> (m, CUT16_13 switches, w2, T) |
                  (VAL2 knob = 1) -> (m, w1, switches, T) |
                  (VAL2 knob = 2) -> (STORE13 w1 w2 m, w1, w2, T) |
                                     (m, w1, w2, F)) |
                (m, w1, w2, T)) |
               (button -> (m, w1, w2, T) | NEXT(m, w1, w2)))
```

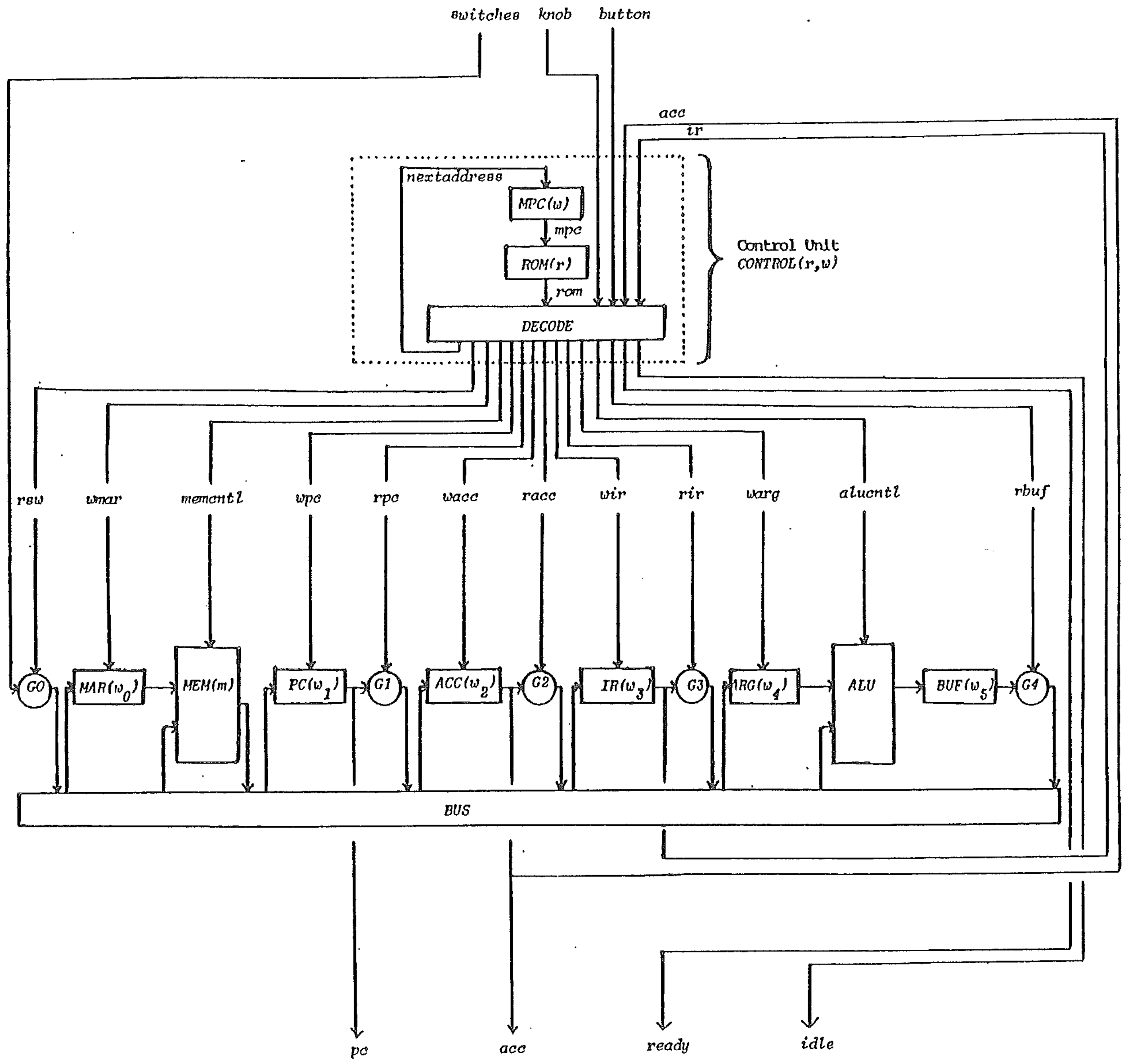
The two axioms above are a complete specification of the computer. Note how concise they are.

Specification of the Host Machine

Next we describe an implementation. We will refer to this as the host machine and the machine it implements (namely the machine specified by *NEXT* and *COMPUTER* above) as the target machine. The host machine has a number of registers in addition to the program counter and accumulator of the target machine. The instruction currently being executed is held in the instruction register *IR*; addresses being looked-up in the memory are held in the memory address register *MAR*; arguments to the arithmetic and logic unit (the *ALU*) are held in the *ARG* register, and the results of the *ALU* are held in the buffer register *BUF*.

The fetch-decode-execute cycle is driven by a microcoded control unit. The microcode is stored in a read-only memory (the *ROM*) which can hold 32 microinstructions, each 30 bits wide. The actual microcode is represented by a constant *MICROCODE* of LSM type *mem5_30*.

The architecture of the host is described by the following diagram:



This diagram shows both the control and data parts of the host; it is represented in LSM by a constant:

```
HOST :word5#mem13_16#word13#word13#word16#word16#word16#word16->dev
```

Where, in *HOST(w,m,w0,w1,w2,w3,w4,w5)*: *w* is a 5-bit word representing the address of the next micorinstuction to be executed, *m* is the memory, *w0* is the contents of the *MAR* register, *w1* is the contents of the *PC* register, *w2* is the contents of the *ACC* register, *w3* is the contents of the *IR* register, *w4* is the contents of the *ARG* register and *w5* is the contents of the *BUF* register.

The correctness of the host is represented in LSM by the formula:

```
!m w0 w1 w2 w3 w4 w5 t.
 COMPUTER(m,w1,w2,t) ==
  until ready do HOST(WORD5(t->0|5),m,w0,w1,w2,w3,w4,w5)
```

The start address in *ROM* of the microcode for controlling the idling computer is *0*; the start address for the fetch-decode-execute cycle is *5*. The host has a done-line called ***ready*** which signals the end of a sequence of microcycles which implement a single target macrocycle.

We hierarchically specify *HOST* by introducing constants:

```
CONTROL : mem5_30#word5 -> dev
DATA    : mem13_16#word13#word13#word16#word16#word16#word16 -> dev
```

And then an axiom:

```
HOST(w,m,w0,w1,w2,w3,w4,w5) ==
 [| CONTROL(MICROCODE,w) | DATA(m,w0,w1,w2,w3,w4,w5) |]
 hide{rsw,wmar,memcntl,wpc,rpc,wacc,racc,wir,rir,warg,alucntl,rbuf,ir}
```

Thus the host is obtained by connecting together the control part (with the actual microcode) and the data part and then hiding the control lines. We specify these parts structurally by:

```
CONTROL(r,w) == [| ROM r | MPC w | DECODE |] hide{mpc,rom,nextaddress}
```

and

```
 DATA(m,w0,w1,w2,w3,w4,w5) ==
 [| MEM m | MAR w0 | PC w1 | ACC w2 | IR w3 | ARG w4 | BUF w5
  | G0 | G1 | G2 | G3 | G4 | ALU | BUS |]
 hide{mem,mar,arg,buf,g0,g1,g2,g3,g4,alu,bus}
```

Where for the component devices we have constants:

```
ROM    : mem5_30 -> dev
MPC    : word5 -> dev
DECODE : dev
MEM    : mem13_16 -> dev
MAR    : word13 -> dev
PC     : word13 -> dev
```

```
ACC     : word16 -> dev
IR      : word16 -> dev
ARG     : word16 -> dev
BUF     : word16 -> dev
G0      : dev
G1      : dev
G2      : dev
G3      : dev
G4      : dev
ALU     : dev
BUS     : dev
```

We then specify axioms defining the behaviour of the memories, registers, gates, ALU, bus, and control logic. We have arbitrarily chosen to take these devices as primitive, we could have chosen to further decompose them structurally, and then proven that these structures have the desired behaviour. The only complicated specification is the one for the purely combinational microinstruction decode logic - i.e. the device *DECODE*. We defer giving this for a while. The read-only memory (for holding the microcode) and microprogram counter are specified by:

```
ROM r == dev{mpc,rom}.{rom=FETCH5 r mpc};ROM r
MPC w == dev{nextaddress,mpc}.{mpc=w};MPC nextaddress
```

To model the bus we use tri-state values, thus the LSM type of values on the bus is *tri_word16* not *word16*. The LSM functions

```
MK_TRI16   : word16     -> tri_word16
DEST_TRI16 : tri_word16 -> word16
```

convert from 16-bit words to 16-bit tri-state words, and vice versa, respectively. These functions satisfy the (built in) axiom:

```
!w:word16. DEST_TRI16(MK_TRI16 w) == w
```

This axiom is invoked by *TRI_RULE* and *TRI_FCONV* as described in Appendix 3 of [Gordon2].

LSM has a special built in tri-state value *FLOAT16* to represent the floating (or high impedance) state. There is also a built in infix:

```
U16 : tri_word16#tri_word16 -> tri_word16
```

for representing the value which results when several values are simultaneously put on a bus. Thus if two tri-state words, *w1* and *w2* say, are put on the bus at the same time then the resulting value is *w1 U16 w2*.

We have the built in axiom:

```
!w:tri_word16. FLOAT16 U16 w == w /\ w U16 FLOAT16 == w
```

This axiom is invoked by *U_RULE* and *U_FCONV* as described in Appendix 3 of [Gordon2]. It says that if only one non-floating value, *w* say, is put on a bus, then the resulting value on the bus is *w*. We do not specify what happens if two or more different non-floating values are put on a bus (this should never happen; proving it doesn't is part of the work that has to be done in verifying correctness). In the definition of *BUS* below we choose to output not this tri-state value, but *DEST_TRI16* of it instead. This saves us having to invoke *DEST_TRI16* in all the devices that read from the bus.

We can now describe the behaviour of the main memory:

```
MEM m ==
  dev{mar, bus ,memcntl ,mem}.
     {mem=(VAL2 memcntl  =  1  -> MK_TRI16(FETCH13 m mar)  |  FLOAT16)};
     MEM(VAL2 memcntl  =  2  -> STORE13 mar bus m  | m)
```

Thus if the number denoted by the 2-bit word on line *memcntl* is *1* then the contents of the location input on line *mar* is put on the line *mem* (which is connected to the bus), otherwise the line is floated. The new state of *MEM* is identical to the old state unless the value on *memcntl* denotes *2* (i.e. is *#10*). If *#10* is input on line *memcntl* then the memory *m* becomes *STORE13 mar bus m* - i.e. a memory identical to *m* except that location *mar* contains value *bus*. (As mentioned above, the definition of *BUS* below ensures that the value on line *bus* is an ordinary 16-bit word, not a tri-state word).

The host uses three kinds of registers: 13 and 16-bit registers with load-enable control line:

```
REG13 : word13 -> dev
REG16 : word16 -> dev
```

With behaviour:

```
REG13 w == dev{i,ld,o13}.{o13=w};REG13(ld->CUT16_13 i|w)
REG16 w == dev{i,ld,o16}.{o16=w};REG16(ld->i|w)
```

Using these we can define:

```
ACC w == (REG16 w) rn[i=bus;ld=wacc;o16=acc]
IR  w == (REG16 w) rn[i=bus;ld=wir  ;o16=ir]
ARG w == (REG16 w) rn[i=bus;ld=warg;o16=arg]
MAR w == (REG13 w) rn[i=bus;ld=wmar;o13=mar]
PC  w == (REG13 w) rn[i=bus;ld=wpc  ;o13=pc]
```

The third kind of register has no load-enable line; the buffer register is the only example of this used, so we define it directly:

```
BUF w == dev{alu,buf}.{buf=w};BUF alu
```

The gate *G1* is the only gate which has a 13-bit input. We define it directly by:

```
G1 == dev{pc,rpc,g1}.{g1=(rpc->MK_TRI16(PAD13_16 pc)|FLOAT16)};G1
```

The remaining four gates all have 16-bit inputs. We will define them in terms of a generic gate device *GATE*:

```
GATE == dev{i,cntl,o}.{o=(cntl->MK_TRI16 i|FLOAT16)};GATE
```

Then:

```
G0 == GATE rn[i=switches;cntl=rsw; o=g0]
G2 == GATE rn[i=acc;     cntl=racc;o=g2]
G3 == GATE rn[i=ir;      cntl=rir; o=g3]
G4 == GATE rn[i=buf;     cntl=rbuf;o=g4]
```

The alu and bus are directly specified by:

```
ALU == dev{arg,bus,alucntl,alu}.
          {alu=(VAL2 alucntl = 0 -> bus |
                VAL2 alucntl = 1 -> INC16 bus |
                VAL2 alucntl = 2 -> ADD16 arg bus |
                                    SUB16 arg bus)};
          ALU

 BUS == dev{mem,g0,g1,g2,g3,g4}.
           {bus=DEST_TRI16(mem U16 g0 U16 g1 U16 g2 U16 g3 U16 g4)};
           BUS
```

Note that the value output on line *bus* has LSM type *word16* - i.e. it is not a tri-state value.

Before defining the microinstruction decoding device *DECODE*, we must desribe the 30-bit microinstructions of the host. Bits *0* to *2* are are 3-bit opcode called the test field. Bits *8* to *12* hold a 5-bit microinstruction address called the A-address, and bits *3* to *7* hold another 5-bit address called the B-address. The remaining bits are all control fields, each field determining the value on a control line during a microcycle. The microinstructions are thus fully "horizontal".

The address of the next microinstruction to be executed is normally the contents of the A-address field, unless:

1. The value denoted by the test field is *1* and *T* is input on line *button*, or the value denoted by the test field is *2* and the value denoted by the 16-bit word on the *acc* line is *0*. In either of these two cases the next address is the contents the B-address field.

2. The value denoted by the test field is *3*. In this case the next address is obtained by adding the value denoted by the 2-bit word on the *knob* line to the value in the A-address field.

3. The value denoted by the test field is *4*. In this case the next address is obtained by adding the value of bits *13* to *15* of the 16-bit word on line *ir* (i.e. the opcode) to the value in the A-address field.

Thus if the test field contains *#001* then pressing the button causes a branch to the B-address; if the test field contains *#010* then a branch to the B-address occurs if the accumulator contains *#0000000000000000*; if the test field contains *#011* then a branch to a microinstruction determined by the position of the knob occurs, and if the test field contains *#100* then a branch to a microinstruction determined by the opcode of the current machine instruction (i.e. the contents of the instruction register) occurs.

The actual microcode for the host is represented in LSM by a constant *MICROCODE : mem5_30*. This is defined by 26 axioms, one for each microinstruction:

```
FETCH5 MICROCODE(WORD5 0)  == #000000000000000110000000001001
FETCH5 MICROCODE(WORD5 1)  == #000000000000000000000100000011
FETCH5 MICROCODE(WORD5 2)  == #010001000000000000000000000000
FETCH5 MICROCODE(WORD5 3)  == #010000010000000000000000000000
FETCH5 MICROCODE(WORD5 4)  == #001000100000000000001110000000
FETCH5 MICROCODE(WORD5 5)  == #000000000000000100011000000001
FETCH5 MICROCODE(WORD5 6)  == #001000100000000000100000000000
FETCH5 MICROCODE(WORD5 7)  == #000100001000000000000000000000
FETCH5 MICROCODE(WORD5 8)  == #000010000100000000100100000000
FETCH5 MICROCODE(WORD5 9)  == #000000000000000000101000000100
FETCH5 MICROCODE(WORD5 10) == #000000000000000000000000000000
FETCH5 MICROCODE(WORD5 11) == #000001000010000000010100000000
FETCH5 MICROCODE(WORD5 12) == #000000000000000001000101011010
FETCH5 MICROCODE(WORD5 13) == #000000001001000001001100000000
FETCH5 MICROCODE(WORD5 14) == #000000001001000001011000000000
FETCH5 MICROCODE(WORD5 15) == #001000000001000001100000000000
FETCH5 MICROCODE(WORD5 16) == #001000000001000001100100000000
FETCH5 MICROCODE(WORD5 17) == #000000100000010001001000000000
FETCH5 MICROCODE(WORD5 18) == #000001000000001000010100000000
FETCH5 MICROCODE(WORD5 19) == #001000000001000001010000000000
FETCH5 MICROCODE(WORD5 20) == #000010000000100001010100000000
FETCH5 MICROCODE(WORD5 21) == #000000010000001001000100000000
FETCH5 MICROCODE(WORD5 22) == #001000000001000001011100000000
FETCH5 MICROCODE(WORD5 23) == #000010000000110001010100000000
FETCH5 MICROCODE(WORD5 24) == #000010010000000001000100000000
FETCH5 MICROCODE(WORD5 25) == #000100001000000001000100000000
```

These axioms were generated by a simple microassembler written in ML. The ML code for this microassembler, together with the symbolic input which generates the above 26 axioms is listed in Appendix 1 of [Gordon4]. We do not specify the microinstructions with addresses *26* to 31.

To make the definition of *DECODE* more readable we introduce some auxiliary functions.

```
CNTL_BIT   : num -> word30 -> bool
CNTL_FIELD : num#num -> word30 -> word2
A_ADDR     : word30 -> word5
B_ADDR     : word30 -> word5
TEST       : word30 -> num
```

These are defined by axioms:

```
CNTL_BIT n w        == EL n (BITS30 w)
```

```
CNTL_FIELD (m,n) w == WORD2(V(SEG(m,n)(BITS30 w)))
A_ADDR w           == WORD5(V(SEG(8,12)(BITS30 w)))
B_ADDR w           == WORD5(V(SEG(3,7)(BITS30 w)))
TEST w             == V(SEG(0,2)(BITS30 w))
```

Thus if *w* is a microinstruction (i.e. a 30-bit word) then: *CNTL_BIT n w* is the *n*'th bit of *w*; *CNTL_FIELD(m,n)w* is the 2-bit word consisting of bits *m* and *n* of *w*; *A_ADDR w* is the A-address of *w*; *B_ADDR w* is the B-address of *w*, and *TEST w* is the test field of *w*. Notice how we have defined these functions by first converting *w* to a list of booleans and then using the list-processing functions *EL* and *SEG*.

The interpretation of microinstructions described above is embodied in the specification of *DECODE*:

```
DECODE ==
 dev{rom,knob,button,acc,ir,nextaddress,rsw,wmar,memcntl,wpc,rpc,
     wacc,racc,wir,rir,warg,alucntl,rbuf,ready,idle}.
    {nextaddress = ((TEST rom = 1) AND button
                      -> B_ADDR rom |
                    (TEST rom = 2) AND (VAL16 acc = 0)
                      -> B_ADDR rom |
                    (TEST rom = 3)
                      -> WORD5((VAL2 knob + 1) + VAL5(A_ADDR rom)) |
                    (TEST rom = 4)
                      -> WORD5(VAL3(OPCODE ir) + VAL5(A_ADDR rom)) |
                         A_ADDR rom),
     rsw          = CNTL_BIT 28 rom,
     wmar         = CNTL_BIT 27 rom,
     memcntl      = CNTL_FIELD (25,26) rom,
     wpc          = CNTL_BIT 24 rom,
     rpc          = CNTL_BIT 23 rom,
     wacc         = CNTL_BIT 22 rom,
     racc         = CNTL_BIT 21 rom,
     wir          = CNTL_BIT 20 rom,
     rir          = CNTL_BIT 19 rom,
     warg         = CNTL_BIT 18 rom,
     alucntl      = CNTL_FIELD (16,17) rom,
     rbuf         = CNTL_BIT 15 rom,
     ready        = CNTL_BIT 14 rom,
     idle         = CNTL_BIT 13 rom};
    DECODE
```

This completes the specification of the host.

Proof of Correctness

To verify that the host machine correctly implements the target machine we must show:

```
!m w0 w1 w2 w3 w4 w5 t.
 COMPUTER(m,w1,w2,t) ==
  until ready do HOST(WORD5(t->0|5),m,w0,w1,w2,w3,w4,w5)
```

It is convenient to introduce a constant:

```
COMPUTER_IMP : bool#mem13_16#word13#word13#word16#word16#word16#word16-
```

Defined by :

```
COMPUTER_IMP(t,m,w0,w1,w2,w3,w4,w5) ==
 until ready do HOST(WORD5(t->0|5),m,w0,w1,w2,w3,w4,w5)
```

Then correctness is simply:

```
!m w0 w1 w2 w3 w4 w5 t.
 COMPUTER(m,w1,w2,t) ==
  COMPUTER_IMP(t,m,w0,w1,w2,w3,w4,w5)
```

We start the proof by deriving a behaviour equation for *HOST*. This is done in two stages. First we use *EXPAND_IMP* to derive an equation for *CONTROL* which we call *CONTROL_EQN*.

```
!r w.
 CONTROL(r,w) ==
  dev{knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,
      wacc,racc,wir,rir,warg,alucntl,rbuf,ready,idle}.
     {rsw=(CNTL_BIT 28(FETCH5 r w)),
      umar=(CNTL_BIT 27(FETCH5 r w)),
      memcntl=(CNTL_FIELD(25,26)(FETCH5 r w)),
      wpc=(CNTL_BIT 24(FETCH5 r w)),
      rpc=(CNTL_BIT 23(FETCH5 r w)),
      wacc=(CNTL_BIT 22(FETCH5 r w)),
      racc=(CNTL_BIT 21(FETCH5 r w)),
      wir=(CNTL_BIT 20(FETCH5 r w)),
      rir=(CNTL_BIT 19(FETCH5 r w)),
      warg=(CNTL_BIT 18(FETCH5 r w)),
      alucntl=(CNTL_FIELD(16,17)(FETCH5 r w)),
      rbuf=(CNTL_BIT 15(FETCH5 r w)),
      ready=(CNTL_BIT 14(FETCH5 r w)),
      idle=(CNTL_BIT 13(FETCH5 r w))};
     CONTROL
     (r,
      (((TEST(FETCH5 r w)) = 1) AND button ->
       B_ADDR(FETCH5 r w) |
       (((TEST(FETCH5 r w)) = 2) AND ((VAL16 acc) = 0) ->
        B_ADDR(FETCH5 r w) |
        ((TEST(FETCH5 r w)) = 3 ->
         WORD5(((VAL2 knob) + 1) + (VAL5(A_ADDR(FETCH5 r w)))) |
         ((TEST(FETCH5 r w)) = 4 ->
          WORD5((VAL3(OPCODE ir)) + (VAL5(A_ADDR(FETCH5 r w)))) |
          A_ADDR(FETCH5 r w))))))
```

Next we derive a behaviour equation for *DATA* which we call *DATA_EQN*. In order to derive this we must use *EXPAND_DEF* to deduce the following equations for the primitives which are defined in terms of generic devices.

```
!w. ACC w == dev{bus,wacc,acc}.{acc=w};ACC(wacc -> bus | w)

!w. IR w == dev{bus,wir,ir}.{ir=w};IR(wir -> bus | w)

!w. ARG w == dev{bus,warg,arg}.{arg=w};ARG(warg -> bus | w)

!w. MAR w == dev{bus,umar,mar}.{mar=w};MAR(umar -> CUT16_13 bus | w)

!w. PC w == dev{bus,wpc,pc}.{pc=w};PC(wpc -> CUT16_13 bus | w)

G0 ==  dev{switches,rsw,g0}.{g0=(rsw -> MK_TRI16 switches | FLOAT16)};G0

G2 == dev{acc,racc,g2}.{g2=(racc -> MK_TRI16 acc | FLOAT16)};G2

G3 == dev{ir,rir,g3}.{g3=(rir -> MK_TRI16 ir | FLOAT16)};G3
```

```
G4 == dev{buf,rbuf,g4}.{g4=(rbuf -> MK_TRI16 buf | FLOAT16)};G4
```

Using these theorems, together with the behaviour equations for the rest of the primitives, we can derive the following behaviour equation for the data part of the host.

```
!m w0 w1 w2 w3 w4 w5.
 DATA(m,w0,w1,w2,w3,w4,w5) ==
  dev{memcntl,wmar,wpc,pc,wacc,acc,wir,ir,
      warg,switches,rsw,rpc,racc,rir,rbuf,alucntl}.
     {pc=w1,
      acc=w2,
      ir=w3,
      bus=(DEST_TRI16
            (((VAL2 memcntl) = 1 -> MK_TRI16(FETCH13 m w0) | FLOAT16) U16
             ((rsw -> MK_TRI16 switches | FLOAT16) U16
              ((rpc -> MK_TRI16(PAD13_16 w1) | FLOAT16) U16
               ((racc -> MK_TRI16 w2 | FLOAT16) U16
                ((rir -> MK_TRI16 w3 | FLOAT16) U16
                 (rbuf -> MK_TRI16 w5 | FLOAT16)))))))};
     DATA
     (((VAL2 memcntl) = 2 -> STORE13 w0 bus m | m),
      (wmar -> CUT16_13 bus | w0),
      (wpc -> CUT16_13 bus | w1),
      (wacc -> bus | w2),
      (wir -> bus | w3),
      (warg -> bus | w4),
      ((VAL2 alucntl) = 0 ->
        bus |
        ((VAL2 alucntl) = 1 ->
          INC16 bus |
          ((VAL2 alucntl) = 2 -> ADD16 w4 bus | SUB16 w4 bus))))
```

Now we can put together the behaviour equations for the control and data parts to get, via *EXPAND_IMP*, the following equation for *HOST*:

```
!w m w0 w1 w2 w3 w4 w5.
 HOST(w,m,w0,w1,w2,w3,w4,w5) ==
  dev{knob,button,acc,ready,idle,pc,switches}.
     {ready=(CNTL_BIT 14(FETCH5 MICROCODE w)),
      idle=(CNTL_BIT 13(FETCH5 MICROCODE w)),
      pc=w1,
      acc=w2,
      bus=(DEST_TRI16
            (((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE w))) = 1 ->
              MK_TRI16(FETCH13 m w0) |
              FLOAT16) U16
             ((CNTL_BIT 28(FETCH5 MICROCODE w) ->
               MK_TRI16 switches |
               FLOAT16) U16
              ((CNTL_BIT 23(FETCH5 MICROCODE w) ->
                MK_TRI16(PAD13_16 w1) |
                FLOAT16) U16
               ((CNTL_BIT 21(FETCH5 MICROCODE w) ->
                 MK_TRI16 w2 |
                 FLOAT16) U16
                ((CNTL_BIT 19(FETCH5 MICROCODE w) ->
                  MK_TRI16 w3 |
                  FLOAT16) U16
                 (CNTL_BIT 15(FETCH5 MICROCODE w) ->
                  MK_TRI16 w5 |
                  FLOAT16)))))))};
     HOST
     ((((TEST(FETCH5 MICROCODE w)) = 1) AND button ->
       B_ADDR(FETCH5 MICROCODE w) |
       (((TEST(FETCH5 MICROCODE w)) = 2) AND ((VAL16 w2) = 0) ->
        B_ADDR(FETCH5 MICROCODE w) |
        ((TEST(FETCH5 MICROCODE w)) = 3 ->
      WORD5
      (((VAL2 knob) + 1) + (VAL5(A_ADDR(FETCH5 MICROCODE w)))) |
      ((TEST(FETCH5 MICROCODE w)) = 4 ->
       WORD5
       ((VAL3(OPCODE w3)) + (VAL5(A_ADDR(FETCH5 MICROCODE w)))) |
       A_ADDR(FETCH5 MICROCODE w))))),
 ((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE w))) = 2 ->
  STORE13 w0 bus m |
  m),
 (CNTL_BIT 27(FETCH5 MICROCODE w) -> CUT16_13 bus | w0),
 (CNTL_BIT 24(FETCH5 MICROCODE w) -> CUT16_13 bus | w1),
 (CNTL_BIT 22(FETCH5 MICROCODE w) -> bus | w2),
 (CNTL_BIT 20(FETCH5 MICROCODE w) -> bus | w3),
 (CNTL_BIT 18(FETCH5 MICROCODE w) -> bus | w4),
 ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 0 ->
  bus |
  ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 1 ->
   INC16 bus |
   ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 2 ->
    ADD16 w4 bus |
    SUB16 w4 bus))))
```

We call this *HOST_EQN*. Next we use the rule *UNTIL* to derive an equation for *until ready do HOST(w,m,w0,w1,w2,w3,w4,w5)*. The resulting theorem, which we call *HOST_UNTIL_EQN*, is rather big. It is:

```
!host.
 (!button knob switches w m w0 w1 w2 w3 w4 w5.
   <equation>
   ==>
   (!w m w0 w1 w2 w3 w4 w5.
     until ready do HOST(w,m,w0,w1,w2,w3,w4,w5) ==
     dev{knob,button,acc,idle,pc,switches}.
        {idle=(CNTL_BIT 13(FETCH5 MICROCODE w)),
         pc=w1,
         acc=w2,
         bus=(DEST_TRI16
               (((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE w))) = 1 ->
                 MK_TRI16(FETCH13 m w0) |
                 FLOAT16) U16
                ((CNTL_BIT 28(FETCH5 MICROCODE w) ->
                  MK_TRI16 switches |
                  FLOAT16) U16
                 ((CNTL_BIT 23(FETCH5 MICROCODE w) ->
                   MK_TRI16(PAD13_16 w1) |
                   FLOAT16) U16
                  ((CNTL_BIT 21(FETCH5 MICROCODE w) ->
                    MK_TRI16 w2 |
                    FLOAT16) U16
                   ((CNTL_BIT 19(FETCH5 MICROCODE w) ->
                     MK_TRI16 w3 |
                     FLOAT16) U16
                    (CNTL_BIT 15(FETCH5 MICROCODE w) ->
                     MK_TRI16 w5 |
                     FLOAT16)))))))};
        until
         ready
         do
         HOST
         (host
          (button,
           knob,
           switches,
           (((TEST(FETCH5 MICROCODE w)) = 1) AND button ->
            B_ADDR(FETCH5 MICROCODE w) |
            (((TEST(FETCH5 MICROCODE w)) = 2) AND ((VAL16 w2) = 0) ->
             B_ADDR(FETCH5 MICROCODE w) |
             ((TEST(FETCH5 MICROCODE w)) = 3 ->
              WORD5
              (((VAL2 knob) + 1) +
                (VAL5(A_ADDR(FETCH5 MICROCODE w)))) |
               ((TEST(FETCH5 MICROCODE w)) = 4 ->
                WORD5
                ((VAL3(OPCODE w3)) +
                 (VAL5(A_ADDR(FETCH5 MICROCODE w)))) |
                A_ADDR(FETCH5 MICROCODE w))))),
           ((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE w))) = 2 ->
            STORE13 w0 bus m |
            m),
           (CNTL_BIT 27(FETCH5 MICROCODE w) -> CUT16_13 bus | w0),
           (CNTL_BIT 24(FETCH5 MICROCODE w) -> CUT16_13 bus | w1),
           (CNTL_BIT 22(FETCH5 MICROCODE w) -> bus | w2),
           (CNTL_BIT 20(FETCH5 MICROCODE w) -> bus | w3),
           (CNTL_BIT 18(FETCH5 MICROCODE w) -> bus | w4),
           ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 0 ->
            bus |
            ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 1 ->
             INC16 bus |
             ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 2 ->
              ADD16 w4 bus |
              SUB16 w4 bus))))))
```

Where <equation> is the formula:

```
host(button,knob,switches,w,m,w0,w1,w2,w3,w4,w5) ==
 let bus =
  DEST_TRI16
  (((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE w))) = 1 ->
    MK_TRI16(FETCH13 m w0) |
    FLOAT16) U16
   ((CNTL_BIT 28(FETCH5 MICROCODE w) ->
     MK_TRI16 switches |
     FLOAT16) U16
    ((CNTL_BIT 23(FETCH5 MICROCODE w) ->
      MK_TRI16(PAD13_16 w1) |
      FLOAT16) U16
     ((CNTL_BIT 21(FETCH5 MICROCODE w) ->
       MK_TRI16 w2 |
       FLOAT16) U16
      ((CNTL_BIT 19(FETCH5 MICROCODE w) ->
        MK_TRI16 w3 |
        FLOAT16) U16
       (CNTL_BIT 15(FETCH5 MICROCODE w) ->
        MK_TRI16 w5 |
        FLOAT16)))))
 in (CNTL_BIT 14(FETCH5 MICROCODE w) ->
     (w,m,w0,w1,w2,w3,w4,w5) |
     host
     (button,
      knob,
      switches,
      (((TEST(FETCH5 MICROCODE w)) = 1) AND button ->
       B_ADDR(FETCH5 MICROCODE w) |
       (((TEST(FETCH5 MICROCODE w)) = 2) AND ((VAL16 w2) = 0) ->
        B_ADDR(FETCH5 MICROCODE w) |
        ((TEST(FETCH5 MICROCODE w)) = 3 ->
         WORD5
         (((VAL2 knob) + 1) +
          (VAL5(A_ADDR(FETCH5 MICROCODE w)))) |
         ((TEST(FETCH5 MICROCODE w)) = 4 ->
          WORD5
          ((VAL3(OPCODE w3)) +
           (VAL5(A_ADDR(FETCH5 MICROCODE w)))) |
          A_ADDR(FETCH5 MICROCODE w))))),
      ((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE w))) = 2 ->
       STORE13 w0 bus m |
       m),
      (CNTL_BIT 27(FETCH5 MICROCODE w) -> CUT16_13 bus | w0),
      (CNTL_BIT 24(FETCH5 MICROCODE w) -> CUT16_13 bus | w1),
      (CNTL_BIT 22(FETCH5 MICROCODE w) -> bus | w2),
      (CNTL_BIT 20(FETCH5 MICROCODE w) -> bus | w3),
(CNTL_BIT 18(FETCH5 MICROCODE w) -> bus | w4),
((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 0 ->
 bus |
 ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 1 ->
  INC16 bus |
  ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 2 ->
   ADD16 w4 bus |
   SUB16 w4 bus)))))
```

We now introduce a constant called *HOST_fn* to name the function defined by <equation>. This constant has OL type:

```
bool#
 word2#
  word16#
   word5#mem13_16#word13#word13#word16#word16#word16#word16
-> word5#mem13_16#word13#word13#word16#word16#word16#word16
```

HOST_fn is defined to satisfy the recursive definition generated by the *UNTIL* rule (i.e. the equation <equation> for ***host*** which is the antecedent of the implication in *HOST_UNTIL_EQN*). The axiom defining *HOST_fn* is thus the formula <equation> with the variable ***host*** replaced by the constant *HOST_fn*, i.e.

```
HOST_fn(button,knob,switches,w,m,w0,w1,w2,w3,w4,w5) ==
 let bus =
  DEST_TRI16
  (((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE w))) = 1 ->
    MK_TRI16(FETCH13 m w0) |
    FLOAT16) U16
   ((CNTL_BIT 28(FETCH5 MICROCODE w) ->
     MK_TRI16 switches |
     FLOAT16) U16
    ((CNTL_BIT 23(FETCH5 MICROCODE w) ->
      MK_TRI16(PAD13_16 w1) |
      FLOAT16) U16
     ((CNTL_BIT 21(FETCH5 MICROCODE w) -> MK_TRI16 w2 | FLOAT16) U16
      ((CNTL_BIT 19(FETCH5 MICROCODE w) -> MK_TRI16 w3 | FLOAT16) U16
       (CNTL_BIT 15(FETCH5 MICROCODE w) -> MK_TRI16 w5 | FLOAT16))))))
 in (CNTL_BIT 14(FETCH5 MICROCODE w) ->
     (w,m,w0,w1,w2,w3,w4,w5) |
     HOST_fn
     (button,
      knob,
      switches,
      (((TEST(FETCH5 MICROCODE w)) = 1) AND button ->
        B_ADDR(FETCH5 MICROCODE w) |
        (((TEST(FETCH5 MICROCODE w)) = 2) AND ((VAL16 w2) = 0) ->
          B_ADDR(FETCH5 MICROCODE w) |
          ((TEST(FETCH5 MICROCODE w)) = 3 ->
           WORD5
           (((VAL2 knob) + 1) + (VAL5(A_ADDR(FETCH5 MICROCODE w)))) |
           ((TEST(FETCH5 MICROCODE w)) = 4 ->
            WORD5
            ((VAL3(OPCODE w3)) +
             (VAL5(A_ADDR(FETCH5 MICROCODE w)))) |
            A_ADDR(FETCH5 MICROCODE w))))),
      ((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE w))) = 2 ->
        STORE13 w0 bus m |
        m),
      (CNTL_BIT 27(FETCH5 MICROCODE w) -> CUT16_13 bus | w0),
      (CNTL_BIT 24(FETCH5 MICROCODE w) -> CUT16_13 bus | w1),
      (CNTL_BIT 22(FETCH5 MICROCODE w) -> bus | w2),
      (CNTL_BIT 20(FETCH5 MICROCODE w) -> bus | w3),
      (CNTL_BIT 18(FETCH5 MICROCODE w) -> bus | w4),
      ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 0 ->
        bus |
((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 1 ->
 INC16 bus |
 ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE w))) = 2 ->
  ADD16 w4 bus |
  SUB16 w4 bus)))))
```

We can now simplify *HOST_UNTIL_EQN* by specializing the variable *host* to the constant *HOST_fn*, and then doing modus ponens with the above axiom. This yields the following theorem:

```
until ready do HOST(w,m,w0,w1,w2,w3,w4,w5) ==
 dev
  {knob,button,acc,idle,pc,switches}.
  {idle=(EL 13(BITS30(FETCH5 MICROCODE w))),
   pc=w1,
   acc=w2,
   bus=(DEST_TRI16
         (((VAL2(WORD2(V(SEG(25,26)(BITS30(FETCH5 MICROCODE w)))))) =
           1 ->
           MK_TRI16(FETCH13 m w0) |
           FLOAT16) U16
          ((EL 28(BITS30(FETCH5 MICROCODE w)) ->
            MK_TRI16 switches |
            FLOAT16) U16
           ((EL 23(BITS30(FETCH5 MICROCODE w)) ->
             MK_TRI16(PAD13_16 w1) |
             FLOAT16) U16
            ((EL 21(BITS30(FETCH5 MICROCODE w)) ->
              MK_TRI16 w2 |
              FLOAT16) U16
             ((EL 19(BITS30(FETCH5 MICROCODE w)) ->
               MK_TRI16 w3 |
               FLOAT16) U16
              (EL 15(BITS30(FETCH5 MICROCODE w)) ->
               MK_TRI16 w5 |
               FLOAT16)))))))};
   until
    ready
    do
    HOST
    (HOST_fn
     (button,
      knob,
      switches,
      (((V(SEG(0,2)(BITS30(FETCH5 MICROCODE w)))) = 1) AND button ->
       WORD5(V(SEG(3,7)(BITS30(FETCH5 MICROCODE w)))) |
       (((V(SEG(0,2)(BITS30(FETCH5 MICROCODE w)))) = 2) AND
        ((VAL16 w2) = 0) ->
        WORD5(V(SEG(3,7)(BITS30(FETCH5 MICROCODE w)))) |
        ((V(SEG(0,2)(BITS30(FETCH5 MICROCODE w)))) = 3 ->
         WORD5
         (((VAL2 knob) + 1) +
          (VAL5(WORD5(V(SEG(8,12)(BITS30(FETCH5 MICROCODE w))))))) |
         ((V(SEG(0,2)(BITS30(FETCH5 MICROCODE w)))) = 4 ->
          WORD5
          ((VAL3(OPCODE w3)) +
           (VAL5(WORD5(V(SEG(8,12)(BITS30(FETCH5 MICROCODE w))))))) |
          WORD5(V(SEG(8,12)(BITS30(FETCH5 MICROCODE w)))))))),
      ((VAL2(WORD2(V(SEG(25,26)(BITS30(FETCH5 MICROCODE w)))))) = 2 ->
       STORE13 w0 bus m |
       m),
      (EL 27(BITS30(FETCH5 MICROCODE w)) -> CUT16_13 bus | w0),
      (EL 24(BITS30(FETCH5 MICROCODE w)) -> CUT16_13 bus | w1),
      (EL 22(BITS30(FETCH5 MICROCODE w)) -> bus | w2),
      (EL 20(BITS30(FETCH5 MICROCODE w)) -> bus | w3),
      (EL 18(BITS30(FETCH5 MICROCODE w)) -> bus | w4),
      ((VAL2(WORD2(V(SEG(16,17)(BITS30(FETCH5 MICROCODE w)))))) = 0 ->
       bus |
       ((VAL2(WORD2(V(SEG(16,17)(BITS30(FETCH5 MICROCODE w)))))) = 1 ->
        INC16 bus |
((VAL2(WORD2(V(SEG(16,17)(BITS30(FETCH5 MICROCODE w)))))) = 2 ->
 ADD16 w4 bus |
 SUB16 w4 bus)))))
```

The variable *w* here is the address of the current microinstruction. From the definition of *COMPUTER_IMP* we see that two important cases are when this address is (a 5-bit word denoting) *0*, and when it is *5*. These cases correspond to the start of the microcode for idling and executing respectively. Using the various evaluation rules and the definition of *MICROCODE* we can derive the following two theorems (we precede them by their names).

HOST_UNTIL_EQN_0

```
until ready do HOST(#00000,m,w0,w1,w2,w3,w4,w5) ==
 dev{knob,button,acc,idle,pc,switches}.
    {idle=T,pc=w1,acc=w2};
    until
     ready
     do
     HOST
     (HOST_fn
      (button,
       knob,
       switches,
       (button -> #00001 | #00000),
       m,
       w0,
       w1,
       w2,
       w3,
       w4,
       DEST_TRI16 FLOAT16))
```

HOST_UNTIL_EQN_5

```
until ready do HOST(#00101,m,w0,w1,w2,w3,w4,w5) ==
 dev{knob,button,acc,idle,pc,switches}.
    {idle=F,pc=w1,acc=w2};
    until
     ready
     do
     HOST
     (HOST_fn
      (button,
       knob,
       switches,
       (button -> #00000 | #00110),
       m,
       w0,
       w1,
       w2,
       w3,
       w4,
       DEST_TRI16 FLOAT16))
```

We must now consider each or the 26 microinstructions separately. Using the axioms defining *MICROCODE* followed by simplification using the various evaluation rules, we can derive:

```
HOST_fn(button,knob,switches,#00000,m,w0,w1,w2,w3,w4,w5) ==
 #00000,m,w0,w1,w2,w3,w4,w5

HOST_fn(button,knob,switches,#00001,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  WORD5(((VAL2 knob) + 1) + 1),
  m,
  w0,
  w1,
  w2,
  w3,
  w4,
  DEST_TRI16 FLOAT16)

HOST_fn(button,knob,switches,#00010,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  #00000,
  m,
  w0,
  CUT16_13 switches,
  w2,
  w3,
  w4,
  switches)

HOST_fn(button,knob,switches,#00011,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,knob,switches,#00000,m,w0,w1,switches,w3,w4,switches)

HOST_fn(button,knob,switches,#00100,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,knob,switches,#00111,m,w1,w1,w2,w3,w4,PAD13_16 w1)

HOST_fn(button,knob,switches,#00101,m,w0,w1,w2,w3,w4,w5) ==
 #00101,m,w0,w1,w2,w3,w4,w5

HOST_fn(button,knob,switches,#00110,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,knob,switches,#01000,m,w1,w1,w2,w3,w4,PAD13_16 w1)

HOST_fn(button,knob,switches,#00111,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,knob,switches,#00000,STORE13 w0 w2 m,w0,w1,w2,w3,w4,w2)

HOST_fn(button,knob,switches,#01000,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  #01001,
  m,
  w0,
  w1,
  w2,
  FETCH13 m w0,
  w4,
  FETCH13 m w0)

HOST_fn(button,knob,switches,#01001,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  WORD5((VAL3(OPCODE w3)) + 10),
  m,
  w0,
  w1,
  w2,
  w3,
  w4,
  DEST_TRI16 FLOAT16)

HOST_fn(button,knob,switches,#01010,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  #00000,
  m,
  w0,
  w1,
  w2,
  w3,
  w4,
  DEST_TRI16 FLOAT16)

HOST_fn(button,knob,switches,#01011,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,knob,switches,#00101,m,w0,CUT16_13 w3,w2,w3,w4,w3)

HOST_fn(button,knob,switches,#01100,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  ((VAL16 w2) = 0 -> #01011 | #10001),
  m,
  w0,
  w1,
  w2,
  w3,
  w4,
  DEST_TRI16 FLOAT16)

HOST_fn(button,knob,switches,#01101,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn(button,knob,switches,#10011,m,w0,w1,w2,w3,w2,w2)

HOST_fn(button,knob,switches,#01110,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn(button,knob,switches,#10110,m,w0,w1,w2,w3,w2,w2)

HOST_fn(button,knob,switches,#01111,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,knob,switches,#11000,m,CUT16_13 w3,w1,w2,w3,w4,w3)

HOST_fn(button,knob,switches,#10000,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,knob,switches,#11001,m,CUT16_13 w3,w1,w2,w3,w4,w3)

HOST_fn(button,knob,switches,#10001,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  #10010,
  m,
  w0,
  w1,
  w2,
  w3,
  w4,
  INC16(PAD13_16 w1))

HOST_fn(button,knob,switches,#10010,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,knob,switches,#00101,m,w0,CUT16_13 w5,w2,w3,w4,w5)

HOST_fn(button,knob,switches,#10011,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,knob,switches,#10100,m,CUT16_13 w3,w1,w2,w3,w4,w3)

HOST_fn(button,knob,switches,#10100,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
```

```
     (button,
      knob,
      switches,
      #10101,
      m,
      w0,
      w1,
      w2,
      w3,
      w4,
      ADD16 w4(FETCH13 m w0))

   HOST_fn(button,knob,switches,#10101,m,w0,w1,w2,w3,w4,w5) ==
    HOST_fn(button,knob,switches,#10001,m,w0,w1,w5,w3,w4,w5)

   HOST_fn(button,knob,switches,#10110,m,w0,w1,w2,w3,w4,w5) ==
    HOST_fn
    (button,knob,switches,#10111,m,CUT16_13 w3,w1,w2,w3,w4,w3)

   HOST_fn(button,knob,switches,#10111,m,w0,w1,w2,w3,w4,w5) ==
    HOST_fn
    (button,
     knob,
     switches,
     #10101,
     m,
     w0,
     w1,
     w2,
     w3,
     w4,
     SUB16 w4(FETCH13 m w0))

   HOST_fn(button,knob,switches,#11000,m,w0,w1,w2,w3,w4,w5) ==
    HOST_fn
    (button,
     knob,
     switches,
     #10001,
     m,
     w0,
     w1,
     FETCH13 m w0,
     w3,
     w4,
     FETCH13 m w0)

   HOST_fn(button,knob,switches,#11001,m,w0,w1,w2,w3,w4,w5) ==
    HOST_fn
    (button,knob,switches,#10001,STORE13 w0 w2 m,w0,w1,w2,w3,w4,w2)
```

By unwinding these equations we can derive a further 26 theorems.

```
   HOST_fn(button,knob,switches,#00000,m,w0,w1,w2,w3,w4,w5) ==
    #00000,m,w0,w1,w2,w3,w4,w5

   HOST_fn(button,knob,switches,#00001,m,w0,w1,w2,w3,w4,w5) ==
    HOST_fn
    (button,
     knob,
     switches,
     WORD5(((VAL2 knob) + 1) + 1),
     m,
     w0,
     w1,
     w2,
     w3,
     w4,
     DEST_TRI16 FLOAT16)

   HOST_fn(button,knob,switches,#00010,m,w0,w1,w2,w3,w4,w5) ==
 #00000,m,w0,CUT16_13 switches,w2,w3,w4,switches

HOST_fn(button,knob,switches,#00011,m,w0,w1,w2,w3,w4,w5) ==
 #00000,m,w0,w1,switches,w3,w4,switches

HOST_fn(button,knob,switches,#00100,m,w0,w1,w2,w3,w4,w5) ==
 #00000,STORE13 w1 w2 m,w1,w1,w2,w3,w4,w2

HOST_fn(button,knob,switches,#00101,m,w0,w1,w2,w3,w4,w5) ==
 #00101,m,w0,w1,w2,w3,w4,w5

HOST_fn(button,knob,switches,#00110,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  WORD5((VAL3(OPCODE(FETCH13 m w1))) + 10),
  m,
  w1,
  w1,
  w2,
  FETCH13 m w1,
  w4,
  DEST_TRI16 FLOAT16)

HOST_fn(button,knob,switches,#00111,m,w0,w1,w2,w3,w4,w5) ==
 #00000,STORE13 w0 w2 m,w0,w1,w2,w3,w4,w2

HOST_fn(button,knob,switches,#01000,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  WORD5((VAL3(OPCODE(FETCH13 m w0))) + 10),
  m,
  w0,
  w1,
  w2,
  FETCH13 m w0,
  w4,
  DEST_TRI16 FLOAT16)

HOST_fn(button,knob,switches,#01001,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  WORD5((VAL3(OPCODE w3)) + 10),
  m,
  w0,
  w1,
  w2,
  w3,
  w4,
  DEST_TRI16 FLOAT16)

HOST_fn(button,knob,switches,#01010,m,w0,w1,w2,w3,w4,w5) ==
 #00000,m,w0,w1,w2,w3,w4,DEST_TRI16 FLOAT16

HOST_fn(button,knob,switches,#01011,m,w0,w1,w2,w3,w4,w5) ==
 #00101,m,w0,CUT16_13 w3,w2,w3,w4,w3

HOST_fn(button,knob,switches,#01100,m,w0,w1,w2,w3,w4,w5) ==
 HOST_fn
 (button,
  knob,
  switches,
  ((VAL16 w2) = 0 -> #01011 | #10001),
  m,
  w0,
  w1,
  w2,
   w3,
   w4,
   DEST_TRI16 FLOAT16)

HOST_fn(button,knob,switches,#01101,m,w0,w1,w2,w3,w4,w5) ==
  #00101,
  m,
  CUT16_13 w3,
  INC13 w1,
  ADD16 w2(FETCH13 m(CUT16_13 w3)),
  w3,
  w2,
  INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#01110,m,w0,w1,w2,w3,w4,w5) ==
  #00101,
  m,
  CUT16_13 w3,
  INC13 w1,
  SUB16 w2(FETCH13 m(CUT16_13 w3)),
  w3,
  w2,
  INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#01111,m,w0,w1,w2,w3,w4,w5) ==
  #00101,
  m,
  CUT16_13 w3,
  INC13 w1,
  FETCH13 m(CUT16_13 w3),
  w3,
  w4,
  INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#10000,m,w0,w1,w2,w3,w4,w5) ==
  #00101,
  STORE13(CUT16_13 w3)w2 m,
  CUT16_13 w3,
  INC13 w1,
  w2,
  w3,
  w4,
  INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#10001,m,w0,w1,w2,w3,w4,w5) ==
  #00101,m,w0, INC13 w1,w2,w3,w4, INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#10010,m,w0,w1,w2,w3,w4,w5) ==
  #00101,m,w0, CUT16_13 w5,w2,w3,w4,w5

HOST_fn(button,knob,switches,#10011,m,w0,w1,w2,w3,w4,w5) ==
  #00101,
  m,
  CUT16_13 w3,
  INC13 w1,
  ADD16 w4(FETCH13 m(CUT16_13 w3)),
  w3,
  w4,
  INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#10100,m,w0,w1,w2,w3,w4,w5) ==
  #00101,
  m,
  w0,
  INC13 w1,
  ADD16 w4(FETCH13 m w0),
  w3,
  w4,
  INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#10101,m,w0,w1,w2,w3,w4,w5) ==
  #00101,m,w0, INC13 w1,w5,w3,w4, INC16(PAD13_16 w1)
```

```
HOST_fn(button,knob,switches,#10110,m,w0,w1,w2,w3,w4,w5) ==
 #00101,
 m,
 CUT16_13 w3,
 INC13 w1,
 SUB16 w4(FETCH13 m(CUT16_13 w3)),
 w3,
 w4,
 INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#10111,m,w0,w1,w2,w3,w4,w5) ==
 #00101,
 m,
 w0,
 INC13 w1,
 SUB16 w4(FETCH13 m w0),
 w3,
 w4,
 INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#11000,m,w0,w1,w2,w3,w4,w5) ==
 #00101,m,w0,INC13 w1,FETCH13 m w0,w3,w4,INC16(PAD13_16 w1)

HOST_fn(button,knob,switches,#11001,m,w0,w1,w2,w3,w4,w5) ==
 #00101,STORE13 w0 w2 m,w0,INC13 w1,w2,w3,w4,INC16(PAD13_16 w1)
```

From these theorems and *HOST_UNTIL_EQN_0* and *HOST_UNTIL_EQN_5* one can derive theorems *HOST_IDLE* and *HOST_RUN* where:

```
HOST_IDLE

!m w0 w1 w2 w3 w4 w5.
 until ready do HOST(#00000,m,w0,w1,w2,w3,w4,w5) ==
 dev
   {knob,button,acc,idle,pc,switches}.
   {idle=T,pc=w1,acc=w2};
  until
   ready
   do
   HOST
   (button ->
    ((VAL2 knob) = 0 ->
     (#00000,m,w0,CUT16_13 switches,w2,w3,w4,switches) |
     ((VAL2 knob) = 1 ->
      (#00000,m,w0,w1,switches,w3,w4,switches) |
      ((VAL2 knob) = 2 ->
       (#00000,STORE13 w1 w2 m,w1,w1,w2,w3,w4,w2) |
       (#00101,m,w0,w1,w2,w3,w4,DEST_TRI16 FLOAT16)))) |
    (#00000,m,w0,w1,w2,w3,w4,DEST_TRI16 FLOAT16))

HOST_RUN

!m w0 w1 w2 w3 w4 w5.
 until ready do HOST(#00101,m,w0,w1,w2,w3,w4,w5) ==
 dev
   {knob,button,acc,idle,pc,switches}.
   {idle=F,pc=w1,acc=w2};
  until
   ready
   do
   HOST
   (button ->
    (#00000,m,w0,w1,w2,w3,w4,DEST_TRI16 FLOAT16) |
    ((VAL3(OPCODE(FETCH13 m w1))) = 0 ->
     (#00000,m,w1,w1,w2,FETCH13 m w1,w4,DEST_TRI16 FLOAT16) |
     ((VAL3(OPCODE(FETCH13 m w1))) = 1 ->
      (#00101,
       m,
 w1,
 CUT16_13(FETCH13 m w1),
 w2,
 FETCH13 m w1,
 w4,
 FETCH13 m w1) |
((VAL3(OPCODE(FETCH13 m w1))) = 2 ->
  ((VAL16 w2) = 0 ->
    (#00101,
     m,
     w1,
     CUT16_13(FETCH13 m w1),
     w2,
     FETCH13 m w1,
     w4,
     FETCH13 m w1) |
    (#00101,m,w1,INC13 w1,w2,FETCH13 m w1,w4,INC16(PAD13_16 w1))) |
  ((VAL3(OPCODE(FETCH13 m w1))) = 3 ->
    (#00101,
     m,
     CUT16_13(FETCH13 m w1),
     INC13 w1,
     ADD16 w2(FETCH13 m(CUT16_13(FETCH13 m w1))),
     FETCH13 m w1,
     w2,
     INC16(PAD13_16 w1)) |
    ((VAL3(OPCODE(FETCH13 m w1))) = 4 ->
      (#00101,
       m,
       CUT16_13(FETCH13 m w1),
       INC13 w1,
       SUB16 w2(FETCH13 m(CUT16_13(FETCH13 m w1))),
       FETCH13 m w1,
       w2,
       INC16(PAD13_16 w1)) |
      ((VAL3(OPCODE(FETCH13 m w1))) = 5 ->
        (#00101,
         m,
         CUT16_13(FETCH13 m w1),
         INC13 w1,
         FETCH13 m(CUT16_13(FETCH13 m w1)),
         FETCH13 m w1,
         w4,
         INC16(PAD13_16 w1)) |
        ((VAL3(OPCODE(FETCH13 m w1))) = 6 ->
          (#00101,
           STORE13(CUT16_13(FETCH13 m w1))w2 m,
           CUT16_13(FETCH13 m w1),
           INC13 w1,
           w2,
           FETCH13 m w1,
           w4,
           INC16(PAD13_16 w1)) |
          (#00101,
           m,
           w1,
           INC13 w1,
           w2,
           FETCH13 m w1,
           w4,
           INC16(PAD13_16 w1))))))))))
```

Recall that *COMPUTER_IMP* is defined by:

```
COMPUTER_IMP(t,m,w0,w1,w2,w3,w4,w5) ==
 until ready do HOST(WORD5(t->0|5),m,w0,w1,w2,w3,w4,w5)
```

Using some case analysis, the theorems *HOST_UNTI EQN_0*, *HOST_UNTIL_EQN_5*, *HOST_IDLE* and *HOST_RUN* we can derive:

```
DERIVED_HOST_EQN

 COMPUTER_IMP(t,m,w0,w1,w2,w3,w4,w5) ==
  dev{knob,button,acc,idle,pc,switches}.
     {idle=t,pc=w1,acc=w2};
     COMPUTER_IMP(term1,term2,term3,term4,term5,term6,term7,term8)
```

where:

```
term1 = (t  ->
          (button ->
           ((VAL2 knob) = 0 ->
            T |
            ((VAL2 knob) = 1 -> T | ((VAL2 knob) = 2 -> T | F))) |
           T) |
          (button ->
           T |
           ((VAL3(OPCODE(FETCH13 m w1))) = 0 ->
            T |
            ((VAL3(OPCODE(FETCH13 m w1))) = 1 ->
             F |
             ((VAL3(OPCODE(FETCH13 m w1))) = 2 ->
              ((VAL16 w2) = 0 -> F | F) |
              ((VAL3(OPCODE(FETCH13 m w1))) = 3 ->
               F |
               ((VAL3(OPCODE(FETCH13 m w1))) = 4 ->
                F |
                ((VAL3(OPCODE(FETCH13 m w1))) = 5 ->
                 F |
                 ((VAL3(OPCODE(FETCH13 m w1))) = 6 -> F | F)))))))))
```

and

```
term2 = (t  ->
          (button ->
           ((VAL2 knob) = 0 ->
            m |
            ((VAL2 knob) = 1 ->
             m |
             ((VAL2 knob) = 2 -> STORE13 w1 w2 m | m))) |
           m) |
          (button ->
           m |
           ((VAL3(OPCODE(FETCH13 m w1))) = 0 ->
            m |
            ((VAL3(OPCODE(FETCH13 m w1))) = 1 ->
             m |
             ((VAL3(OPCODE(FETCH13 m w1))) = 2 ->
              ((VAL16 w2) = 0 -> m | m) |
              ((VAL3(OPCODE(FETCH13 m w1))) = 3 ->
               m |
               ((VAL3(OPCODE(FETCH13 m w1))) = 4 ->
                m |
                ((VAL3(OPCODE(FETCH13 m w1))) = 5 ->
                 m |
                 ((VAL3(OPCODE(FETCH13 m w1))) = 6 ->
                  STORE13(CUT16_13(FETCH13 m w1))w2 m |
                  m)))))))))
```

and

```
term3 = (t  ->
          (button ->
           ((VAL2 knob) = 0 ->
            w0 |
            ((VAL2 knob) = 1 -> w0 | ((VAL2 knob) = 2 -> w1 | w0))) |
           w0) |
                (button ->
                 w0 |
                  ((VAL3(OPCODE(FETCH13 m w1))) = 0 ->
                   w1 |
                    ((VAL3(OPCODE(FETCH13 m w1))) = 1 ->
                     w1 |
                      ((VAL3(OPCODE(FETCH13 m w1))) = 2 ->
                       ((VAL16 w2) = 0 -> w1 | w1) |
                       ((VAL3(OPCODE(FETCH13 m w1))) = 3 ->
                        CUT16_13(FETCH13 m w1) |
                        ((VAL3(OPCODE(FETCH13 m w1))) = 4 ->
                         CUT16_13(FETCH13 m w1) |
                         ((VAL3(OPCODE(FETCH13 m w1))) = 5 ->
                          CUT16_13(FETCH13 m w1) |
                          ((VAL3(OPCODE(FETCH13 m w1))) = 6 ->
                           CUT16_13(FETCH13 m w1) |
                           w1)))))))))
```

and

```
term4 = (t ->
                (button ->
                 ((VAL2 knob) = 0 ->
                  CUT16_13 switches |
                  ((VAL2 knob) = 1 -> w1 | ((VAL2 knob) = 2 -> w1 | w1))) |
                 w1) |
                (button ->
                 w1 |
                  ((VAL3(OPCODE(FETCH13 m w1))) = 0 ->
                   w1 |
                    ((VAL3(OPCODE(FETCH13 m w1))) = 1 ->
                     CUT16_13(FETCH13 m w1) |
                      ((VAL3(OPCODE(FETCH13 m w1))) = 2 ->
                       ((VAL16 w2) = 0 -> CUT16_13(FETCH13 m w1) | INC13 w1) |
                       ((VAL3(OPCODE(FETCH13 m w1))) = 3 ->
                        INC13 w1 |
                        ((VAL3(OPCODE(FETCH13 m w1))) = 4 ->
                         INC13 w1 |
                         ((VAL3(OPCODE(FETCH13 m w1))) = 5 ->
                          INC13 w1 |
                          ((VAL3(OPCODE(FETCH13 m w1))) = 6 ->
                           INC13 w1 |
                           INC13 w1))))))))
```

and

```
term5 = (t ->
                (button ->
                 ((VAL2 knob) = 0 ->
                  w2 |
                  ((VAL2 knob) = 1 ->
                   switches |
                   ((VAL2 knob) = 2 -> w2 | w2))) |
                 w2) |
                (button ->
                 w2 |
                  ((VAL3(OPCODE(FETCH13 m w1))) = 0 ->
                   w2 |
                    ((VAL3(OPCODE(FETCH13 m w1))) = 1 ->
                     w2 |
                      ((VAL3(OPCODE(FETCH13 m w1))) = 2 ->
                       ((VAL16 w2) = 0 -> w2 | w2) |
                       ((VAL3(OPCODE(FETCH13 m w1))) = 3 ->
                        ADD16 w2(FETCH13 m(CUT16_13(FETCH13 m w1))) |
                        ((VAL3(OPCODE(FETCH13 m w1))) = 4 ->
                         SUB16 w2(FETCH13 m(CUT16_13(FETCH13 m w1))) |
                         ((VAL3(OPCODE(FETCH13 m w1))) = 5 ->
                          FETCH13 m(CUT16_13(FETCH13 m w1)) |
                          ((VAL3(OPCODE(FETCH13 m w1))) = 6 -> w2 | w2))))))))
```

and

```
term6 = (t ->
          (button ->
           ((VAL2 knob) = 0 ->
            w3 |
            ((VAL2 knob) = 1 -> w3 | ((VAL2 knob) = 2 -> w3 | w3))) |
           w3) |
          (button ->
           w3 |
           ((VAL3(OPCODE(FETCH13 m w1))) = 0 ->
            FETCH13 m w1 |
            ((VAL3(OPCODE(FETCH13 m w1))) = 1 ->
             FETCH13 m w1 |
             ((VAL3(OPCODE(FETCH13 m w1))) = 2 ->
              ((VAL16 w2) = 0 -> FETCH13 m w1 | FETCH13 m w1) |
              ((VAL3(OPCODE(FETCH13 m w1))) = 3 ->
               FETCH13 m w1 |
               ((VAL3(OPCODE(FETCH13 m w1))) = 4 ->
                FETCH13 m w1 |
                ((VAL3(OPCODE(FETCH13 m w1))) = 5 ->
                 FETCH13 m w1 |
                 ((VAL3(OPCODE(FETCH13 m w1))) = 6 ->
                  FETCH13 m w1 |
                  FETCH13 m w1))))))))
```

and

```
term7 = (t ->
          (button ->
           ((VAL2 knob) = 0 ->
            w4 |
            ((VAL2 knob) = 1 -> w4 | ((VAL2 knob) = 2 -> w4 | w4))) |
           w4) |
          (button ->
           w4 |
           ((VAL3(OPCODE(FETCH13 m w1))) = 0 ->
            w4 |
            ((VAL3(OPCODE(FETCH13 m w1))) = 1 ->
             w4 |
             ((VAL3(OPCODE(FETCH13 m w1))) = 2 ->
              ((VAL16 w2) = 0 -> w4 | w4) |
              ((VAL3(OPCODE(FETCH13 m w1))) = 3 ->
               w2 |
               ((VAL3(OPCODE(FETCH13 m w1))) = 4 ->
                w2 |
                ((VAL3(OPCODE(FETCH13 m w1))) = 5 ->
                 w4 |
                 ((VAL3(OPCODE(FETCH13 m w1))) = 6 -> w4 | w4))))))))
```

and

```
term8 = (t ->
          (button ->
           ((VAL2 knob) = 0 ->
            switches |
            ((VAL2 knob) = 1 ->
             switches |
             ((VAL2 knob) = 2 -> w2 | DEST_TRI16 FLOAT16))) |
           DEST_TRI16 FLOAT16) |
          (button ->
           DEST_TRI16 FLOAT16 |
           ((VAL3(OPCODE(FETCH13 m w1))) = 0 ->
            DEST_TRI16 FLOAT16 |
            ((VAL3(OPCODE(FETCH13 m w1))) = 1 ->
             FETCH13 m w1 |
             ((VAL3(OPCODE(FETCH13 m w1))) = 2 ->
((VAL16 w2) = 0 -> FETCH13 m w1 | INC16(PAD13_16 w1)) |
((VAL3(OPCODE(FETCH13 m w1))) = 3 ->
 INC16(PAD13_16 w1) |
 ((VAL3(OPCODE(FETCH13 m w1))) = 4 ->
  INC16(PAD13_16 w1) |
  ((VAL3(OPCODE(FETCH13 m w1))) = 5 ->
   INC16(PAD13_16 w1) |
   ((VAL3(OPCODE(FETCH13 m w1))) = 6 ->
    INC16(PAD13_16 w1) |
    INC16(PAD13_16 w1))))))))
```

This is in the right form for applying the *UNIQUENESS* rule. Before we can use this we must derive a similar equation for the target machine *COMPUTER*. The required theorem is:

```
COMPUTER_EQN

 COMPUTER(m,w1,w2,t) ==
  dev{knob,button,switches,pc,acc,idle}.
     {pc=w1,acc=w2,idle=t};
     COMPUTER(term2,term4,term5,term1)
```

Where *term1*, *term2*, *term4*, *term5* are the terms occuring in *DERIVED_HOST_EQN* defined above.

Finally we just execute:

```
UNIQUENESS COMPUTER_EQN DERIVED_HOST_EQN
```

which generates the theorem showing correctness, namely:

```
 COMPUTER(m,w1,w2,t) == COMPUTER_IMP(t,m,w0,w1,w2,w3,w4,w5)
```

Appendix 1: Microassembler and symbolic microcode for the host

In this appendix we give the ML code for the microassembler, followed by the input used to generate the axioms specifying *MICROCODE*. Comments are enclosed between percent signs (i.e. %s). Although this code will only be understandable if you are familiar with ML, I hope it illustrates how simple it is to implement the microassembler in ML.

```
% The code that follows generates a theory called MUCODE containing
  axioms defining the constant MICROCODE which was declared in the
  theory HOST %

new_theory'MUCODE';;

new_parent'HOST';;

% The identifiers below give symbolic names to subfields of a
  microinstruction %

let rsw    = [28]
and umar   = [27]
and write  = [26]
and read   = [25]
and upc    = [24]
and rpc    = [23]
and uacc   = [22]
and racc   = [21]
and wir    = [20]
and rir    = [19]
and warg   = [18]
and add    = [17]
and inc    = [16]
and sub    = [16;17]
and rbuf   = [15]
and ready  = [14]
and idle   = [13];;

% map2 f (l1,l2)

 maps f in parallel down l1 and l2 %

letrec map2 f (l1,l2) =
 if null l1 & null l2 then []
                         else f(hd l1,hd l2).map2 f (tl l1,tl l2);;

% mk_ADDR [b4;b3;b2;b1;b0] n

 gives a list of the positions of bits that are 1 if n is represented in the
 5-bit field defined by bit positions b4,b3,b2,b1,b0 %

let mk_ADDR l n =
 if n<0 or n>31 or (not(length l= 5))
 then failwith 'mk_A_ADDR'
 else
  mapfilter
    (\n. n=0=>fail|n)
    (map2 $* (map(. t='0'=>0|1)(mk_bin rep(5,n)), l));;

% mk_A_ADDR n

  gives a list of the positions of bits in the A-address field which are 1
  if number n is held in that field

  mk_B_ADDR n

  gives a list of the positions of bits in the B-address field which are 1
  if number n is held in that field %
```

```
let mk_A_ADDR = mk_ADDR [12;11;10;9;8]
and mk_B_ADDR = mk_ADDR [7 ;6 ;5 ;4;3];;

% test_button(n,m)

   gives a list of the bits that are 1 if the address of the next
   microinstruction is (button->n|m) %

let test_button (n,m) =
 [0]@(mk_A_ADDR m)@(mk_B_ADDR n);;

% test_acc(n,m)

   gives a list of the bits that are 1 if the address of the next
   microinstruction is (acc=0->n|m) %

let test_acc (n,m) =
 [1]@(mk_A_ADDR m)@(mk_B_ADDR n);;

% jump n

   gives a list of the bits that are 1 if the address of the next
   microinstruction is n %

let jump = mk_A_ADDR;;

% jump_knob n

   gives a list of the bits that are 1 if the address of the next
   microinstruction is knob+1 %

let jump_knob n =
 [1;0]@(mk_A_ADDR n);;

% jump_opcode n

   gives a list of the bits that are 1 if the address of the next
   microinstruction is opcode+10 %

let jump_opcode n =
 [2]@(mk_A_ADDR n);;

let word30_ty = ":word30";;

% mk_micro_ins l

   takes a list of numbers representing the positions of bits in a
   microinstuction that should be 1, and constructs a micro-word (i.e. a
   value of type :word30) with the appropriate bits on %

let mk_micro_ins l =
 mk_const
   (implode
     (`#`.
       map
        (\n. mem n l=>`1`|`0`)
        [29;28;27;26;25;24;23;22;21;20;19;18;17;16;15;14;13;12;11;10;
         9;8;7;6;5;4;3;2;1;0]),
    word30_ty);;
```

```
% define_microinstruction(addr,cont)

  generates an axiom of the form: |- FETCH5 MICROCODE (WORD5 n) == w
  where n is the ROM-address corresponding to the ML integer
  addr, and w is a microinstruction word generated from the list
  cont of bits that are on %

let define_microinstruction (addr,cont) =
 new_axiom
  (concat `MICROCODE` (tok_of_int addr),
   "FETCH5 MICROCODE (WORD5 ^(int_to_term addr)) ==
     ^(mk_micro_ins cont)");;

% The following generates axioms defining the host's microcode MICROCODE.
  The comments should be interpreted as follows: A --> B is a tranfer with source
  and target B (this is sometimes written B:=A); MEM(MAR) is the location in
  memory pointed to by the address in MAR. Each element of the list %

map
 define_microinstruction
  [0 , ready @ idle  @ (test_button(1,0)); % begin idling cycle %
   1 ,                 (jump_knob 1);      % decode knob position %
   2 , rsw   @ wpc   @ (jump 0);           % switches --> PC %
   3 , rsw   @ wacc  @ (jump 0);           % switches --> ACC %
   4 , rpc   @ wmar  @ (jump 7);           % PC --> MAR %
   5 , ready @         (test_button(0,6)); % begin instruction execution %
   6 , rpc   @ wmar  @ (jump 8);           % PC --> MAR %
   7 , racc  @ write @ (jump 0);           % ACC --> MEM(MAR) %
   8 , read  @ wir   @ (jump 9);           % MEM(MAR) --> IR %
   9 ,                 (jump_opcode 10);   % decode %
   10,                 (jump 0);           % HALT %
   11, rir   @ wpc   @ (jump 5);           % JMP    IR --> PC %
   12,                 (test_acc(11,17));  % JZR %
   13, racc  @ warg  @ (jump 19);          % ADD    ACC --> ARG %
   14, racc  @ warg  @ (jump 22);          % SUB    ACC --> ARG %
   15, rir   @ wmar  @ (jump 24);          % LD     IR --> MAR %
   16, rir   @ wmar  @ (jump 25);          % ST     IR --> MAR %
   17, rpc   @ inc   @ (jump 18);          % PC+1 --> BUF %
   18, rbuf  @ wpc   @ (jump 5);           % BUF --> PC %
   19, rir   @ wmar  @ (jump 20);          % IR --> MAR %
   20, read  @ add   @ (jump 21);          % ARG+MEM(MAR) --> BUF %
   21, rbuf  @ wacc  @ (jump 17);          % BUF --> ACC %
   22, rir   @ wmar  @ (jump 23);          % IR --> MAR %
   23, read  @ sub   @ (jump 21);          % ARG-MEM(MAR) --> BUF %
   24, read  @ wacc  @ (jump 17);          % MEM(MAR) --> ACC %
   25, racc  @ write @ (jump 17)];;        % ACC -> MEM(MAR) %

close theory();;
```

Appendix 2: Derivation of a hard-wired controller

From the theorem *CONTROL_EQN* given above, and the axioms defining *MICROCODE* we can derive the following theorems, one for each microinstruction:

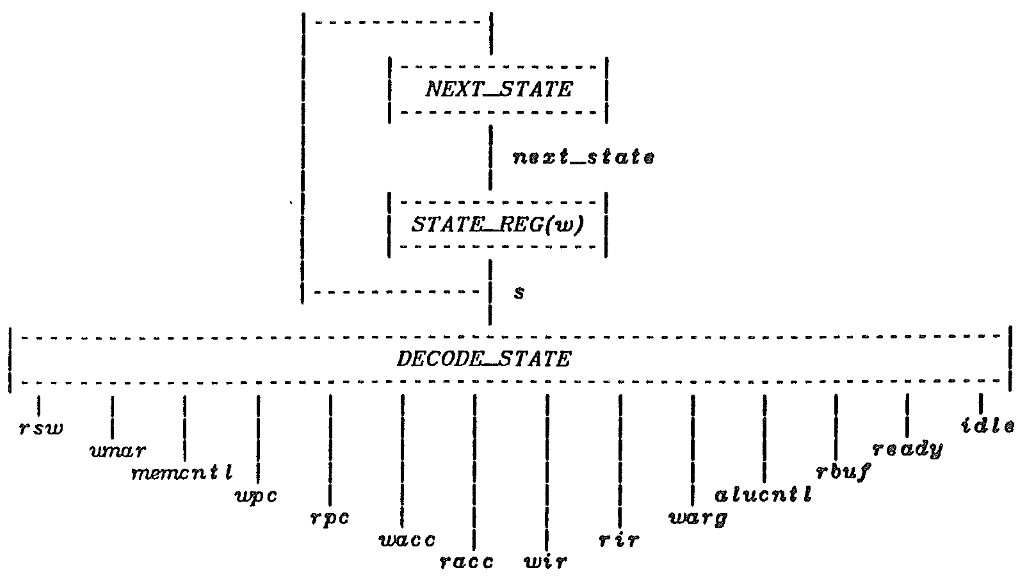
```
CONTROL(MICROCODE,#00000) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,upc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=F,
   umar=F,
   memcntl=#00,
   upc=F,
   rpc=F,
   wacc=F,
   racc=F,
   wir=F,
   rir=F,
   warg=F,
   alucntl=#00,
   rbuf=F,
   ready=T,
   idle=T};
  CONTROL(MICROCODE,(button -> #00001 | #00000))";

CONTROL(MICROCODE,#00001) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,upc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=F,
   umar=F,
   memcntl=#00,
   upc=F,
   rpc=F,
   wacc=F,
   racc=F,
   wir=F,
   rir=F,
   warg=F,
   alucntl=#00,
   rbuf=F,
   ready=F,
   idle=F};
  CONTROL(MICROCODE,WORD5(((VAL2 knob) + 1) + 1))";

CONTROL(MICROCODE,#00010) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,upc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=T,
   umar=F,
   memcntl=#00,
   upc=T,
   rpc=F,
   wacc=F,
   racc=F,
   wir=F,
   rir=F,
   warg=F,
   alucntl=#00,
   rbuf=F,
   ready=F,
   idle=F};
  CONTROL(MICROCODE,#00000)";

CONTROL(MICROCODE,#00011) ==
 dev
```

```
   {knob, button, acc, ir, rsw, umar, memcntl, upc, rpc, wacc, racc, wir, rir,
    warg, alucntl, rbuf, ready, idle}.
   {rsw=T,
    umar=F,
    memcntl=#00,
    upc=F,
    rpc=F,
    wacc=T,
    racc=F,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE, #00000)";

CONTROL(MICROCODE, #00100) ==
 dev
   {knob, button, acc, ir, rsw, umar, memcntl, upc, rpc, wacc, racc, wir, rir,
    warg, alucntl, rbuf, ready, idle}.
   {rsw=F,
    umar=T,
    memcntl=#00,
    upc=F,
    rpc=T,
    wacc=F,
    racc=F,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE, #00111)";

CONTROL(MICROCODE, #00101) ==
 dev
   {knob, button, acc, ir, rsw, umar, memcntl, upc, rpc, wacc, racc, wir, rir,
    warg, alucntl, rbuf, ready, idle}.
   {rsw=F,
    umar=F,
    memcntl=#00,
    upc=F,
    rpc=F,
    wacc=F,
    racc=F,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=T,
    idle=F};
   CONTROL(MICROCODE, (button -> #00000 | #00110))";

CONTROL(MICROCODE, #00110) ==
 dev
   {knob, button, acc, ir, rsw, umar, memcntl, upc, rpc, wacc, racc, wir, rir,
    warg, alucntl, rbuf, ready, idle}.
   {rsw=F,
    umar=T,
    memcntl=#00,
    upc=F,
    rpc=T,
    wacc=F,
    racc=F,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE,#01000)";

 CONTROL(MICROCODE,#00111) ==
  dev
   {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
    warg,alucntl,rbuf,ready,idle}.
   {rsw=F,
    umar=F,
    memcntl=#10,
    wpc=F,
    rpc=F,
    wacc=F,
    racc=T,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE,#00000)";

 CONTROL(MICROCODE,#01000) ==
  dev
   {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
    warg,alucntl,rbuf,ready,idle}.
   {rsw=F,
    umar=F,
    memcntl=#01,
    wpc=F,
    rpc=F,
    wacc=F,
    racc=F,
    wir=T,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE,#01001)";

 CONTROL(MICROCODE,#01001) ==
  dev
   {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
    warg,alucntl,rbuf,ready,idle}.
   {rsw=F,
    umar=F,
    memcntl=#00,
    wpc=F,
    rpc=F,
    wacc=F,
    racc=F,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE,WORD5((VAL3(OPCODE ir)) + 10))";

 CONTROL(MICROCODE,#01010) ==
  dev
   {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
    warg,alucntl,rbuf,ready,idle}.
   {rsw=F,
    umar=F,
    memcntl=#00,
    wpc=F,
    rpc=F,
    wacc=F,
    racc=F,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE,#00000)";

CONTROL(MICROCODE,#01011) ==
 dev
   {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
    warg,alucntl,rbuf,ready,idle}.
   {rsw=F,
    umar=F,
    memcntl=#00,
    wpc=T,
    rpc=F,
    wacc=F,
    racc=F,
    wir=F,
    rir=T,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE,#00101)";

CONTROL(MICROCODE,#01100) ==
 dev
   {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
    warg,alucntl,rbuf,ready,idle}.
   {rsw=F,
    umar=F,
    memcntl=#00,
    wpc=F,
    rpc=F,
    wacc=F,
    racc=F,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE,((VAL16 acc) = 0 -> #01011 | #10001))";

CONTROL(MICROCODE,#01101) ==
 dev
   {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
    warg,alucntl,rbuf,ready,idle}.
   {rsw=F,
    umar=F,
    memcntl=#00,
    wpc=F,
    rpc=F,
    wacc=F,
    racc=T,
    wir=F,
    rir=F,
    warg=T,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
```

```
  CONTROL(MICROCODE,#10011)";

CONTROL(MICROCODE,#01110) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=F,
   umar=F,
   memcntl=#00,
   wpc=F,
   rpc=F,
   wacc=F,
   racc=T,
   wir=F,
   rir=F,
   warg=T,
   alucntl=#00,
   rbuf=F,
   ready=F,
   idle=F};
  CONTROL(MICROCODE,#10110)";

CONTROL(MICROCODE,#01111) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=F,
   umar=T,
   memcntl=#00,
   wpc=F,
   rpc=F,
   wacc=F,
   racc=F,
   wir=F,
   rir=T,
   warg=F,
   alucntl=#00,
   rbuf=F,
   ready=F,
   idle=F};
  CONTROL(MICROCODE,#11000)";

CONTROL(MICROCODE,#10000) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=F,
   umar=T,
   memcntl=#00,
   wpc=F,
   rpc=F,
   wacc=F,
   racc=F,
   wir=F,
   rir=T,
   warg=F,
   alucntl=#00,
   rbuf=F,
   ready=F,
   idle=F};
  CONTROL(MICROCODE,#11001)";

CONTROL(MICROCODE,#10001) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=F,
   umar=F,
   memcntl=#00,
   wpc=F,
   rpc=T,
   wacc=F,
    racc=F,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#01,
    rbuf=F,
    ready=F,
    idle=F};
  CONTROL(MICROCODE,#10010)";

CONTROL(MICROCODE,#10010) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=F,
   umar=F,
   memcntl=#00,
   wpc=T,
   rpc=F,
   wacc=F,
   racc=F,
   wir=F,
   rir=F,
   warg=F,
   alucntl=#00,
   rbuf=T,
   ready=F,
   idle=F};
  CONTROL(MICROCODE,#00101)";

CONTROL(MICROCODE,#10011) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=F,
   umar=T,
   memcntl=#00,
   wpc=F,
   rpc=F,
   wacc=F,
   racc=F,
   wir=F,
   rir=T,
   warg=F,
   alucntl=#00,
   rbuf=F,
   ready=F,
   idle=F};
  CONTROL(MICROCODE,#10100)";

CONTROL(MICROCODE,#10100) ==
 dev
  {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
   warg,alucntl,rbuf,ready,idle}.
  {rsw=F,
   umar=F,
   memcntl=#01,
   wpc=F,
   rpc=F,
   wacc=F,
   racc=F,
   wir=F,
   rir=F,
   warg=F,
   alucntl=#10,
   rbuf=F,
   ready=F,
   idle=F};
  CONTROL(MICROCODE,#10101)";

CONTROL(MICROCODE,#10101) ==
 dev
```

```
    {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
    {rsw=F,
     umar=F,
     memcntl=#00,
     wpc=F,
     rpc=F,
     wacc=T,
     racc=F,
     wir=F,
     rir=F,
     warg=F,
     alucntl=#00,
     rbuf=T,
     ready=F,
     idle=F};
    CONTROL(MICROCODE,#10001)";

CONTROL(MICROCODE,#10110) ==
  dev
    {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
    {rsw=F,
     umar=T,
     memcntl=#00,
     wpc=F,
     rpc=F,
     wacc=F,
     racc=F,
     wir=F,
     rir=T,
     warg=F,
     alucntl=#00,
     rbuf=F,
     ready=F,
     idle=F};
    CONTROL(MICROCODE,#10111)";

CONTROL(MICROCODE,#10111) ==
  dev
    {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
    {rsw=F,
     umar=F,
     memcntl=#01,
     wpc=F,
     rpc=F,
     wacc=F,
     racc=F,
     wir=F,
     rir=F,
     warg=F,
     alucntl=#11,
     rbuf=F,
     ready=F,
     idle=F};
    CONTROL(MICROCODE,#10101)";

CONTROL(MICROCODE,#11000) ==
  dev
    {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
    {rsw=F,
     umar=F,
     memcntl=#01,
     wpc=F,
     rpc=F,
     wacc=T,
     racc=F,
     wir=F,
     rir=F,
     warg=F,
       alucntl=#00,
       rbuf=F,
       ready=F,
       idle=F};
     CONTROL(MICROCODE,#10001)";

 CONTROL(MICROCODE,#11001) ==
  dev
   {knob,button,acc,ir,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
    warg,alucntl,rbuf,ready,idle}.
   {rsw=F,
    umar=F,
    memcntl=#10,
    wpc=F,
    rpc=F,
    wacc=F,
    racc=T,
    wir=F,
    rir=F,
    warg=F,
    alucntl=#00,
    rbuf=F,
    ready=F,
    idle=F};
   CONTROL(MICROCODE,#10001)
```

We can use these theorems to design a controller which just has a 5-bit register (to hold a state counter) instead of a ROM with microcode. The structure of this hard-wired controller is:



Each state corresponds to a microinstruction address, and the decode logic *DECODE_STATE* and sequencing logic *NEXT_STATE* mimic the effect of the microcode as exhibited in the 26 theorems above. For abstractness let us take the state to be a number (rather than, say, a value of OL type *word5*). The specification of *STATE_REG* is:

```
STATE_REG n == dev{next state,s}.{s=n};STATE_REG next state
```

The purely combinational sequencing logic can be read off from the 26 theorems:

```
NEXT_STATE ==
 dev{s,button,knob,ir,acc,next state}.
    {next state = (s=0  -> (button->1|0)                   |
                   s=1  -> (VAL2 knob)+2                   |
                   s=2  -> 0                               |
                   s=3  -> 0                               |
                   s=4  -> 7                               |
                   s=5  -> (button->0|6)                   |
                   s=6  -> 8                               |
                   s=7  -> 0                               |
                   s=8  -> 9                               |
                   s=9  -> (VAL3(OPCODE ir))+10            |
                   s=10 -> 0                               |
                   s=11 -> 5                               |
                   s=12 -> ((VAL16 acc)=0->11|17)          |
                   s=13 -> 19                              |
                   s=14 -> 22                              |
                   s=15 -> 24                              |
                   s=16 -> 25                              |
                   s=17 -> 18                              |
                   s=18 -> 5                               |
                   s=19 -> 20                              |
                   s=20 -> 21                              |
                   s=22 -> 23                              |
                   s=23 -> 21                              |
                   17                                      )};
    NEXT_STATE
```

The purely combinational decode logic can also be constructed directly from the 26 theorems:

```
DECODE_STATE ==
 dev
  {s,rsw,wmar,memcntl,wpc,rpc,wacc,racc,wir,rir,warg,
   alucntl,rbuf,ready,idle}.
  {rsw     = (s=2)OR(s=3),
   wmar    = (s=4)OR(s=6)OR(s=15)OR(s=16)OR(s=19)OR(s=22),
   memcntl = ((s=17)OR(s=25)->#10|
               (s=8)OR(s=20)OR(s=23)OR(s=24)->#01|#00),
   wpc     = (s=2)OR(s=11)OR(s=18),
   rpc     = (s=14)OR(s=6)OR(s=17),
   wacc    = (s=2)OR(s=21)OR(s=24),
   racc    = (s=7)OR(s=13)OR(s=14)OR(s=25),
   wir     = (s=8),
   rir     = (s=11)OR(s=15)OR(s=16)OR(s=19)OR(s=22),
   warg    = (s=13)OR(s=14),
   alucntl = ((s=23)->#11|(s=20)->#10|(s=17)->#01|#00),
   rbuf    = (s=18)OR(s=21),
   ready   = (s=0)OR(s=5),
   idle    = (s=0)};
  DECODE_STATE
```

Using these devices we can then define a controller *CNTL* by:

```
CNTL n ==
 [| STATE_REG n | NEXT_STATE | DECODE_STATE |] hide{s,next state}
```

Although *CNTL* was derived directly from the microcode, pehaps we made a silly error when typing in the definitions. To check we didn't we can prove that *CNTL* satisfies the same equations as the microprogrammed controller *CONTROL*. First we use *EXPAND_IMP* to derive a behaviour equation for *CNTL n*, then we simplify the cases $n=0$, $n=1$, $n=2$, ... , $n=25$ separately to derive:

```
CNTL(VAL5 #00000) ==
 dev{button,knob,ir,acc,rsw,umar,memcntl,upc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
    {rsw=F,
     umar=F,
     memcntl=#00,
     upc=F,
     rpc=F,
     wacc=F,
     racc=F,
     wir=F,
     rir=F,
     warg=F,
     alucntl=#00,
     rbuf=F,
     ready=T,
     idle=T};
    CNTL(button -> 1 | 0)

CNTL(VAL5 #00001) ==
 dev{button,knob,ir,acc,rsw,umar,memcntl,upc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
    {rsw=F,
     umar=F,
     memcntl=#00,
     upc=F,
     rpc=F,
     wacc=F,
     racc=F,
     wir=F,
     rir=F,
     warg=F,
     alucntl=#00,
     rbuf=F,
     ready=F,
     idle=F};
    CNTL((VAL2 knob) + 2)

CNTL(VAL5 #00010) ==
 dev{button,knob,ir,acc,rsw,umar,memcntl,upc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
    {rsw=T,
     umar=F,
     memcntl=#00,
     upc=T,
     rpc=F,
     wacc=T,
     racc=F,
     wir=F,
     rir=F,
     warg=F,
     alucntl=#00,
     rbuf=F,
     ready=F,
     idle=F};
    CNTL 0

CNTL(VAL5 #00011) ==
 dev{button,knob,ir,acc,rsw,umar,memcntl,upc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
    {rsw=T,
     umar=F,
     memcntl=#00,
     upc=F,
     rpc=F,
     wacc=F,
     racc=F,
     wir=F,
     rir=F,
     warg=F,
     alucntl=#00,
       rbuf=F,
       ready=F,
       idle=F};
      CNTL 0

CNTL(VAL5 #00100) ==
 dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
      {rsw=F,
       umar=T,
       memcntl=#00,
       wpc=F,
       rpc=F,
       wacc=F,
       racc=F,
       wir=F,
       rir=F,
       warg=F,
       alucntl=#00,
       rbuf=F,
       ready=F,
       idle=F};
      CNTL 7

CNTL(VAL5 #00101) ==
 dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
      {rsw=F,
       umar=F,
       memcntl=#00,
       wpc=F,
       rpc=F,
       wacc=F,
       racc=F,
       wir=F,
       rir=F,
       warg=F,
       alucntl=#00,
       rbuf=F,
       ready=T,
       idle=F};
      CNTL(button -> 0 | 6)

CNTL(VAL5 #00110) ==
 dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
      {rsw=F,
       umar=T,
       memcntl=#00,
       wpc=F,
       rpc=T,
       wacc=F,
       racc=F,
       wir=F,
       rir=F,
       warg=F,
       alucntl=#00,
       rbuf=F,
       ready=F,
       idle=F};
      CNTL 8

CNTL(VAL5 #00111) ==
 dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
     warg,alucntl,rbuf,ready,idle}.
      {rsw=F,
       umar=F,
       memcntl=#00,
       wpc=F,
       rpc=F,
       wacc=F,
       racc=T,
        wir=F,
        rir=F,
        warg=F,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL 0

  CNTL(VAL5 #01000) ==
   dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=F,
        memcntl=#01,
        wpc=F,
        rpc=F,
        wacc=F,
        racc=F,
        wir=T,
        rir=F,
        warg=F,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL 9

  CNTL(VAL5 #01001) ==
   dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=F,
        memcntl=#00,
        wpc=F,
        rpc=F,
        wacc=F,
        racc=F,
        wir=F,
        rir=F,
        warg=F,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL((VAL3(OPCODE ir)) + 10)

  CNTL(VAL5 #01010) ==
   dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=F,
        memcntl=#00,
        wpc=F,
        rpc=F,
        wacc=F,
        racc=F,
        wir=F,
        rir=F,
        warg=F,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL 0

  CNTL(VAL5 #01011) ==
   dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=F,
        memcntl=#00,
        wpc=T,
        rpc=F,
        wacc=F,
        racc=F,
        wir=F,
        rir=T,
        warg=F,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL 5

  CNTL(VAL5 #01100) ==
   dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=F,
        memcntl=#00,
        wpc=F,
        rpc=F,
        wacc=F,
        racc=F,
        wir=F,
        rir=F,
        warg=F,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL((VAL16 acc) = 0 -> 11 | 17)

  CNTL(VAL5 #01101) ==
   dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=F,
        memcntl=#00,
        wpc=F,
        rpc=F,
        wacc=F,
        racc=T,
        wir=F,
        rir=F,
        warg=T,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL 19

  CNTL(VAL5 #01110) ==
   dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=F,
        memcntl=#00,
        wpc=F,
        rpc=T,
        wacc=F,
        racc=T,
        wir=F,
        rir=F,
        warg=T,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL 22

  CNTL(VAL5 #01111) ==
   dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=T,
        memcntl=#00,
        wpc=F,
        rpc=F,
        wacc=F,
        racc=F,
        wir=F,
        rir=T,
        warg=F,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL 24

 CNTL(VAL5 #10000) ==
  dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=T,
        memcntl=#00,
        wpc=F,
        rpc=F,
        wacc=F,
        racc=F,
        wir=F,
        rir=T,
        warg=F,
        alucntl=#00,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL 25

 CNTL(VAL5 #10001) ==
  dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=F,
        memcntl=#10,
        wpc=F,
        rpc=T,
        wacc=F,
        racc=F,
        wir=F,
        rir=F,
        warg=F,
        alucntl=#01,
        rbuf=F,
        ready=F,
        idle=F};
       CNTL 18

 CNTL(VAL5 #10010) ==
  dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
        warg,alucntl,rbuf,ready,idle}.
       {rsw=F,
        umar=F,
        memcntl=#00,
        wpc=T,
        rpc=F,
        wacc=F,
        racc=F,
        wir=F,
        rir=F,
        warg=F,
        alucntl=#00,
        rbuf=T,
        ready=F,
        idle=F};
```

```
      CNTL 5

 CNTL(VAL5 #10011) ==
  dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
      warg,alucntl,rbuf,ready,idle}.
     {rsw=F,
      umar=T,
      memcntl=#00,
      wpc=F,
      rpc=F,
      wacc=F,
      racc=F,
      wir=F,
      rir=T,
      warg=F,
      alucntl=#00,
      rbuf=F,
      ready=F,
      idle=F};
     CNTL 20

 CNTL(VAL5 #10100) ==
  dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
      warg,alucntl,rbuf,ready,idle}.
     {rsw=F,
      umar=F,
      memcntl=#01,
      wpc=F,
      rpc=F,
      wacc=F,
      racc=F,
      wir=F,
      rir=F,
      warg=F,
      alucntl=#10,
      rbuf=F,
      ready=F,
      idle=F};
     CNTL 21

 CNTL(VAL5 #10101) ==
  dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
      warg,alucntl,rbuf,ready,idle}.
     {rsw=F,
      umar=F,
      memcntl=#00,
      wpc=F,
      rpc=F,
      wacc=T,
      racc=F,
      wir=F,
      rir=F,
      warg=F,
      alucntl=#00,
      rbuf=T,
      ready=F,
      idle=F};
     CNTL 17

 CNTL(VAL5 #10110) ==
  dev{button,knob,ir,acc,rsw,umar,memcntl,wpc,rpc,wacc,racc,wir,rir,
      warg,alucntl,rbuf,ready,idle}.
     {rsw=F,
      umar=T,
      memcntl=#00,
      wpc=F,
      rpc=F,
      wacc=F,
      racc=F,
      wir=F,
      rir=T,
      warg=F,
          alucntl=#00,
          rbuf=F,
          ready=F,
          idle=F};
        CNTL 23

    CNTL(VAL5 #10111) ==
     dev{button,knob,ir,acc,rsw,wmar,memcntl,wpc,rpc,wacc,racc,wir,rir,
          warg,alucntl,rbuf,ready,idle}.
        {rsw=F,
          wmar=F,
          memcntl=#01,
          wpc=F,
          rpc=F,
          wacc=F,
          racc=F,
          wir=F,
          rir=F,
          warg=F,
          alucntl=#11,
          rbuf=F,
          ready=F,
          idle=F};
        CNTL 21

    CNTL(VAL5 #11000) ==
     dev{button,knob,ir,acc,rsw,wmar,memcntl,wpc,rpc,wacc,racc,wir,rir,
          warg,alucntl,rbuf,ready,idle}.
        {rsw=F,
          wmar=F,
          memcntl=#01,
          wpc=F,
          rpc=F,
          wacc=T,
          racc=F,
          wir=F,
          rir=F,
          warg=F,
          alucntl=#00,
          rbuf=F,
          ready=F,
          idle=F};
        CNTL 17

    CNTL(VAL5 #11001) ==
     dev{button,knob,ir,acc,rsw,wmar,memcntl,wpc,rpc,wacc,racc,wir,rir,
          warg,alucntl,rbuf,ready,idle}.
        {rsw=F,
          wmar=F,
          memcntl=#10,
          wpc=F,
          rpc=F,
          wacc=F,
          racc=T,
          wir=F,
          rir=F,
          warg=F,
          alucntl=#00,
          rbuf=F,
          ready=F,
          idle=F};
        CNTL 17
```

One can now compare these equations for *CNTL* with the previous set of 26 equations derived for *CONTROL*. It will be seen that they are equivalent.

If one wanted more security one could then go on to prove that:

```
COMPUTER(m,w1,w2,t) ==
 until ready
 do [| CNTL(t->0|5) | DATA(m,w0,w1,w2,w3,w4,w5) |]
    hide{rsw,wmar,memcntl,wpc,rpc,wacc,racc,
         wir,rir,warg,alucntl,rbuf,ir}
```

One could either do this directly, or by extending *MICROCODE* so that:

```
!w. CONTROL(MICROCODE,w) == CNTL(VAL5 w)
```

If this held, then the original proof of correctness would not need to be redone. Unfortunately, because we have not specified what happens when the microinstruction address lies in the range *26* to *31*, this equation does not follow from the axioms given above.

References

[Cardelli]
L. Cardelli. *Sticks & Stones: An Applicative VLSI Design Language*, Internal Report CSR-85-81, Department of Computer Science, University of Edinburgh, 1981.

[Gordon1]
M. Gordon. *A Model of register Transfer Systems with Applications to Microcode and VLSI Correctness*. Internal Report CSR-82-81, Dept. of Computer Science, University of Edinburgh, 1981

[Gordon2]
M. Gordon. *LCF_LSM: A system for specifying and verifying hardware*. University of Cambridge computer laboratory technical report Number 41, 1983.