Technical Report

UCAM-CL-TR-100
ISSN 1476-2986

Number 100

Proving a computer correct in higher order logic

Jeff Joyce, Graham Birtwistle, Mike Gordon

December 1986

ISSN 1476-2986

Proving a Computer Correct in Higher Order Logic

Jeff Joyce
Graham Birtwistle
Mike Gordon

Abstract

Technical Report No. 42, 'Proving a Computer Correct using the LCF_LSM Hardware Verification System', describes the specification and verification of a register-transfer level implementation of a simple general purpose computer. The computer has a microcoded control unit implementing eight user level instructions. We have subsequently redone this example in higher order logic using the HOL hardware verification system.

This report presents the specification and verification of Gordon's computer as an example of hardware specification and verification in higher order logic. The report describes how the structure and behaviour of digital circuits may be specified using the formalism of higher order logic. The proof of correctness also shows how digital behaviour at different granularities of time may be related by means of a temporal abstraction.

This report should be read with Technical Report No. 68, 'HOL, A Machine Oriented Formulation of Higher Order Logic', which describes the logic underlying the HOL hardware verification system.

Proving a Computer Correct in Higher Order Logic [1]

Jeff Joyce
Graham Birtwistle [2]
Mike Gordon

1 Introduction

Mike Gordon has described the specification and verification of a register-transfer level implementation of a simple general purpose computer using the LCF_LSM hardware verification system [1] [2]. We have subsequently redone this example in higher order logic using the HOL system [3]. In this paper we present the specification and verification of Gordon's computer as an example of hardware specification and verification in higher order logic.

2 Informal Description of Gordon's Computer

Gordon's computer is a simple general-purpose computer invented for this example of hardware specification and verification. Nevertheless the register-transfer implementation of this machine is sufficiently realistic to serve as an interesting and illustrative example.

Target Level

At the target level, the computer has a memory and two registers. The memory has a 13-bit address space of 16-bit words. The two registers are the 13-bit program counter PC and the 16-bit accumulator ACC.

The front panel of the computer is shown in Figure 1. 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 (ie. not executing a program). There are also sixteen two-position switches which are used for inputing data. [3]

[1]This report originally appeared as 'Research Report 85/208/21, Department of Computer Science, University of Calgary, Canada, August 1985'.

[2]Department of Computer Science, University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4

[3]This description of the computer is based on the description in 'Proving a Computer Correct using the LCF_LSM Hardware Verification System', Technical Report No. 42, University of Cambridge, 1983.

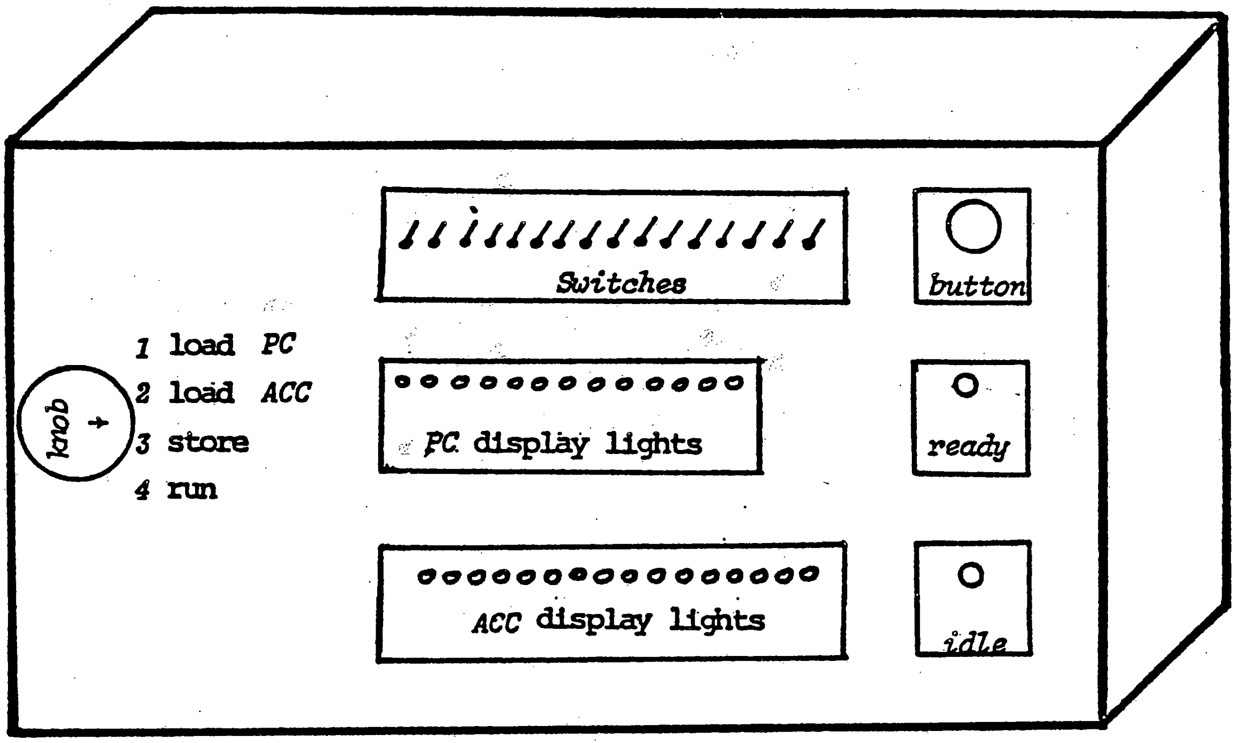

Figure 1: Front Panel of Gordon's Computer

Pushing the button on the front panel of the computer when the computer is running (ie. executing a program) interrupts the execution of the program and the computer idles. The effect of pushing the button on the front panel when the computer is idling is determined by the position of the knob. We shall refer to the four positions of the knob as 0, 1, 2 and 3. When the knob is in position 0, the effect of pushing the button is that the word determined by the state of the thirteen rightmost switches is loaded into the program counter. Pushing the button when the knob is in position 1 loads the word determined by the sixteen switches into the accumulator. When the knob is in position 2, the contents of the accumulator is stored in the memory at the location held in the program counter. Finally, knob position 3 is used to start the execution of the program stored in memory beginning at the location in the program counter.

knob = 0	load PC
knob = 1	load ACC
knob = 2	store ACC at PC
knob = 3	start execution at PC

When execution of a program begins, the idle light goes off and stays off until execution stops. Execution can only stop if a HALT instruction is encountered or an interrupt is generated by pushing the button.

The computer has eight machine instructions: HALT, JMP, JZR, ADD, SUB, LOAD, STORE, and SKIP. Each instruction consists of a 3-bit opcode and a thirteen bit address. The format is:

000—address—	HALT	Stops execution
001—address—	JMP L	Jump to address
010—address—	JZR L	Jump to address if ACC = 0
011—address—	ADD L	Add contents of address to ACC
100—address—	SUB L	Subtract contents of address from ACC
101—address—	LD L	Load contents of address into ACC
110—address—	ST L	Store contents of ACC in address
111—address—	SKIP	Skip to next instruction

This completes the target level description of the computer. A description such as this is typically what an assembler language programmer would need to write programs. Importantly, the description completely captures the relevant behaviour of computer and hides implementation details which are generally of little interest to the assembler language programmer. We assume that a description of the computer from the point of view of an assembler language programmer is the most appropriate for a target level description.

Implementation Level

The register-transfer level implementation of Gordon's computer is shown in Figure 2. The implementation has a number of registers in addition to the program counter and accumulator of the target machine. The instruction currently under execution is held in the instruction register IR; addresses of memory locations to be read or written to are held in the memory address register MAR; arguments to the arithmetic and logic unit ALU are held in the ARG register, and the results of the ALU are held in the buffer register BUF. The implementation also uses five gate devices G0, G1, G2, G3 and G4 to control the reading of data onto the 16-bit bus, BUS.

The fetch-decode-execute cycle is driven by a microcoded control unit. The microcode is stored in a read-only memory ROM which can hold 32 microcode instructions, each 30 bits wide. On every clock cycle, the microcode instruction addressed by the microcode program counter MPC is read from ROM and decoded by the decode unit DECODE. Output from the decode unit consists of signals which control the operation of the data part of the implementation. These signals correspond to control lines which are labelled in Figure 2 as rsw, wmar, memcntl, wpc, rpc, wacc, racc, wir, rir, warg, alucntl and rbuf. For instance, when the boolean signal rpc has the value T (or true), the low 13 bits of the bus are read into the program counter register PC. All of the control signals are boolean signals except for memcntl and alucntl which are both signals with values of type *:word2*. The decode unit also produces the address of the next microcode instruction which is loaded in the microcode program counter.

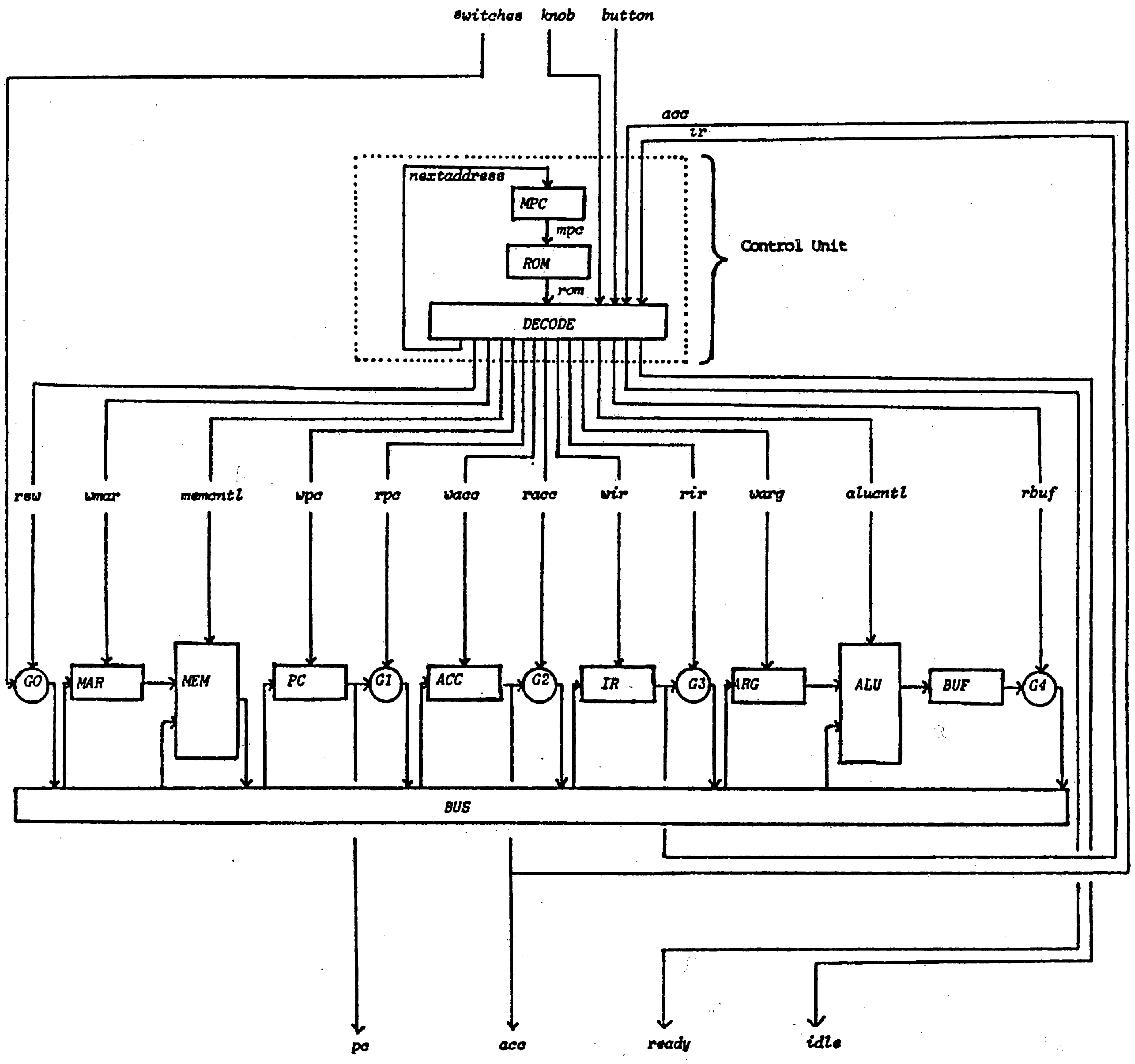

Figure 2: Register-Transfer Level Implementation of Gordon's Computer

Location	Control	Signals	Next Location	Explanation
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 instruction
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	SKIP, 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)
26			jump 0	unused
27			jump 0	unused
28			jump 0	unused
29			jump 0	unused
30			jump 0	unused
31			jump 0	unused

Figure 3: Microcode Program Pseudo-Code

The microcode is fully horizontal which means that each bit or group of bits has a unique function. For instance, bit 23 of each microcode instruction word uniquely controls the rpc signal. The microcode program we use in the specification is shown in Figure 3. The first column of each row indicates the address of the microcode instruction in the microcode store. The rest of the row specifies which bits are set in this microcode instruction. Columns 2 and 3 specify the settings of bits 28 to 13 in terms of which control signals should be generated. Bits 12 to 0 are used by the decode unit to determine the location of the next microcode instruction. These 13 bits consist of two 5-bit address fields and a 3-bit test field which indicates how the next location is to be obtained from the two address fields. The setting of bits 12 to 0 is indicated in column 4 of Figure 3.

This completes our informal description of Gordon's computer and its register-transfer level implementation.

3 Hardware Specification in Higher Order Logic

The starting point in a discussion of hardware specification in higher order logic is a discussion of 'signals'.

A signal is a mapping from discrete points in time to primitive values. [4] In higher order logic, we treat a signal as a function of type *:num→<primitive type>*. In the computer example, the following are used as primitive types: *:bool*, *:word2*, *:word3*, *:word5*, *:word13*, *:word16*, *:tri_word16*, *:word30*, *:mem13_16* and *:mem5_30*. For example, the signal corresponding to the output of the the PC register is of type *:num→word13*. That is, this signal maps discrete points in time, which we regard as numbers, to 13-bit values. Note that a value of type *:word13*, for example, is not the same as a value of type *:num* although we have a means of obtaining the numerical value normally associated with a 13-bit word.

We describe a signal in higher order logic as a equation which gives the value of a signal at a point in time in terms of constants and values of this signal or other signals at that point or an earlier point in time. For example, we can describe the output signal of a 16-bit register with the equation shown below. [5]

```
!t:num. output (t+1) = ((load t) => (input t) | (output t))
```

Building upon this model of a signal, we view hardware devices as collections of input and output signals. We describe devices in higher order logic as simply a

[4] In this paper, we limit our discussion of signals to discrete time scales. Below the register-transfer level, one might choose to consider signals as mappings from a continuous time scale to primitive values.

[5] '!' and '?' are the symbols for universal and existential quantification, respectively, in the HOL system. A conditional expression has the form, `A => B | C`.

conjunction of equations for each output signal. The register only has one output signal and so the above equation serves as a specification of a register device.

It useful to associate the conjunction of equations describing a device with a constant. Moreover, since we often require more than one instance of a device in an implementation, we use formal parameters to name signals in the behaviour equations. The specification of a 16-bit register in higher order logic is shown below.

```
|- REG16 (input,load,output) =
    !t:num. output (t+1) = ((load t) => (input t) | (output t))
```

At this point we can begin the formal specification of Gordon's computer in higher order logic using the HOL system.

4 Formal Specification of Gordon's Computer

We begin by remarking that the HOL system has a number of built-in theories which provide such things as simple arithmetic. HOL also provides facilities which directly support the specification and verification of hardware in higher order logic. This includes types for words and memories of any dimension. At the present time, some of these facilities are not completely developed. For instance, a fully developed theory about values of type *:word<n>* would include the theorem that a value of type *:word2*, for example, has only four possible values, #00, #01, #10 and #11. [6] Currently, one must supply such facts as required in the form of axioms. However, we ignore such details in the rest of our discussion.

The specification of Gordon's computer begins with the definition of some constants which name operations used in both the target level specification and the specification of the implementation.

INC16	:word16→word16
ADD16	:word16→word16→word16
SUB16	:word16→word16→word16

It is unnecessary to specify details of the operation named by INC16, ADD16 and SUB16 because the verification of the computer implementation does not require any information about these operations except for their types. The operations named by INC16, ADD16 and SUB16 are taken as primitive operations in the specification of the arithmetic and logic unit ALU. INC16, ADD16 and SUB16 then reappear in the target level specification. We never have to synthesis the ADD16 operation from other operations nor do ever need to know any details

[6]A word<n> constant is a term starting with '#' followed by n 0's and 1's.

of the ADD16 operation to synthesis another operation. It is sufficient to only know that ADD16 names an operation which takes two 16-bit words and results in a 16-bit word. Similarly, only type specifications are required for INC16 and SUB16.

However, we do need to supply details of several other operations which are also common to the target and implementation levels of specification. These operations are named by the following constants.

PAD13_16	:word13→word16
CUT16_13	:word16→word13
INC13	:word13→word13

A fully developed theory about values of type *:word<n>* would include pad and cut operations. Currently, we must supply these operations. When a 13-bit address is transferred along the 16-bit bus in the implementation, it must be padded with 3 extra bits. When the 13-bit address arrives at its destination as a 16-bit word, it is normally truncated back to a 13-bit word. Padding and subsequently truncating a 13-bit word results in the original 13-bit word. This is reflected in the following axiom.

```
|- !w:word13. CUT16_13 (PAD13_16 w) = w
```

We also specify axiomatically that the INC13 operation is the result of padding a 13-bit word, applying the INC16 operation and then truncating the result back to a 13-bit word.

```
|- !w:word13. CUT16_13(INC16(PAD13_16 w)) = INC13 w
```

Finally, OPCODE names the operation which extracts the 3-bit opcode field of an target level instruction. Like INC16, ADD16 and SUB16, it is unnecessary to specify any details for this operation except for its type.

OPCODE	:word16→word3

Implementation Level

Following a hierarchical design methodology, we divide the register-transfer implementation of the computer into two parts, the data path and the control unit.

We first consider the specification of the data path of the implementation beginning with the registers. We have already seen the specification of a 16-bit register. ACC, IR and ARG are 16-bit registers which we specify as instances of REG16. We also require a 13-bit register which takes a 16-bit input but only uses the lower thirteen bits. The specification of REG13 uses the constant CUT16_13 to name the operation which truncates a 16-bit word to 13-bit word. MAR and PC are instances of REG13.

```
|- REG13 (input,load,output) =
    !t:num.
     output (t+1) = ((load t) => (CUT16_13 (input t)) | (output t))
```

Both REG16 and REG13 specify registers which are selectively loadable. A third type of register, BUF is a 16-bit register that is not selectively loadable. BUF loads from its input line every clock cycle.

```
|- BUF (input,output) = !t:num. output (t+1) = input t
```

So far we have only considered primitive values which are bi-stable, that is, values which, for each bit position, are either 0 or 1. We also need to consider tri-state values which, in addition to 0 and 1, have a high impedance or floating state. When we connect devices to the bus in the register-transfer level implementation, we need to consider the result of merging several signals. Merging bi-stable signals is not acceptable because merging 0 with 1 corresponds to connecting power to ground which is obviously undesirable from an electrical point of view. A third state of high impedance or floating represents a value on a wire which is electrically isolated from power or ground. Merging a floating value with another floating value results in the floating value, but merging a floating with 0 results in 0. Similarly merging a floating value with 1 results in 1.

The HOL system provides the type *:tri_word<n>* to model tri-state values. The floating value for the type *:tri_word<n>* is named by the constant FLOAT<n>. The constants MK_TRI<n> and DEST_TRI<n> are also provided to convert from bi-state to tri-state values and vice versa. Another constant, U<n>, names an infix function which represents the value which results when two tri-state values are merged. The functions satisfy the built-in axioms:

```
|- !w:word<n>. DEST_TRI<n> (MK_TRI<n> w) = w

|- !w:tri_state<n>. (FLOAT<n> U<n> w = w) /\ (w U<n> FLOAT<n> = w)
```

With this explanation of tri-state values, we can consider devices which gate values onto the bus. Five of the six devices which gate values onto the bus are called 'gates'. These devices receive bi-stable outputs from registers and the set of switches. If the control line of the gate is high, the input value is gated onto the bus otherwise the tri-state value FLOAT16 is produced. The definition of a GATE16 is shown below. G0, G2, G3 and G4 are instances of GATE16. G1 is an instance of GATE13 which is identical to GATE16 except that it receives a 13-bit value from the PC register and produces a 16-bit value by padding the 13-bit input with an unspecified value.

```
|- GATE16 (input,control,output) =
    !t:num.
     output t = (control t => MK_TRI16 (input t) | FLOAT16)

|- GATE13 (input,control,output) =
    !t:num.
     output t =
      (control t => MK_TRI16 (PAD13_16 (input t)) | FLOAT16)
```

The remaining components of the data path of the computer are the memory, the arithmetic and logic unit, and the bus.

Our specification of the memory treats the actual storage function as a signal, memory, and the memory device as simply a device which regulates access to this signal. The operation of the memory device is control by the 2-bit memory control signal, memcntl. When the memory control line signal is #00 or #11, the device is effectively inoperative. The tri-state floating value FLOAT16 is read onto the bus and the memory is left unchanged. However, when the value of the control signal is #01, the memory signal is left unchanged but the 16-bit word addressed by the value of mar is read onto the bus. Notice that the memory is only other device besides gates which reads values onto the bus. When the value of the control signal is #10, the memory device updates the memory signal using the built-in constant STORE13 to name the operation which stores values in the memory. We can see from the specification of the device that the value of the signal mar is the address at which the value of bus is stored in the memory.

```
|- MEM (memory,mar,bus,memcntl,memout) =
    !t:num.
     (memout t =
      ((VAL2 (memcntl t) = 1) =>
       MK_TRI16 (FETCH13 (memory t) (mar t)) | FLOAT16)) /\
     (memory(t+1) =
    ((VAL2 (memcntl t) = 2) =>
     STORE13 (mar t) (bus t) (memory t) | memory t))
```

The arithmetic and logic unit, ALU, is a combinational device with the functions: no operation, increment, addition and subtraction. The function is selected by the 2-bit alu control signal, alucntl.

```
|- ALU (arg,bus,alucntl,alu) =
    !t:num.
     (alu t =
      ((VAL2 (alucntl t) = 0) => bus t |
       (VAL2 (alucntl t) = 1) => INC16 (bus t) |
       (VAL2 (alucntl t) = 2) => ADD16 (arg t) (bus t) |
       SUB16 (arg t) (bus t)))
```

The final component in the data path of the register-transfer implementation of the computer is the 16-bit bus. This bus is used for the transfer of both 16-bit data words and 13-bit addresses. We model the bus as a signal of type *:num→word16*, that is, a signal which produces a bi-state 16-bit value from six tri-state input signals, memout, g0, g1, g2, g3 and g4. We have already mentioned that a non-floating value merged with a floating value produces the non-floating value. Our register-transfer level implementation of the computer will guarantee that at most one non-floating value will ever be read onto the bus.

```
|- BUS(memout,g0,g1,g2,g3,g4,bus) =
    !t:num.
     bus t = DEST_TRI16
      (memout t U16 g0 t U16 g1 t U16 g2 t U16 g3 t U16 g4 t)
```

Having provided specifications of all the components in the data path of the computer, we can now precede in a 'bottom-up' fashion by specifying the data path in terms of these primitive components. As we have said, a device is specified as the conjunction of equations describing the behaviour of the output signals of the device. In the specification of primitive component, we must provide these equations explicitedly. However, we specify a composite device as a conjunction of the specifications for each instance of a component used to implement the device. Furthermore, signals must be named to replace the formal parameters in the specifications of components in the device. For example, in the specification of the data path, the signals bus, wpc and pc replace the formal parameters input, load and output in the specification of PC. We can then 'expand' the specification of a composite device to produce a conjunction of equations for each output signal.

The specification of the data path of the computer, DATA, in terms of the register-transfer devices, for which we have provided formal specifications, is shown below. The existential quantification of the signals g0, g1, g2, g3, g4, memout, alu and bus has the effect of 'hiding' these signals within the specification of the data path. We are able to hide these signals within the specification of the data path because the signals are isolated from the rest of the computer implementation. Hiding these signals is very desirable because it hides details of the implementation of the data path which are not required when we view the data path as a individual component in the implementation of the computer.

```
|- DATA
    (memory,mar,pc,acc,ir,arg,buf,switches,rsw,wmar,memcntl,
     wpc,rpc,wacc,racc,wir,rir,warg,alucntl,rbuf) =
    ?g0 g1 g2 g3 g4 memout alu bus.
     MEM(memory,mar,bus,memcntl,memout) /\
     MAR(bus,wmar,mar) /\
     PC(bus,wpc,pc) /\
     ACC(bus,wacc,acc) /\
     IR(bus,wir,ir) /\
     ARG(bus,warg,arg) /\
     BUF(alu,buf) /\
     G0(switches,rsw,g0) /\
     G1(pc,rpc,g1) /\
     G2(acc,racc,g2) /\
     G3(ir,rir,g3) /\
     G4(buf,rbuf,g4) /\
     ALU(arg,bus,alucntl,alu) /\
     BUS(memout,g0,g1,g2,g3,g4,bus)
```

We now turn to a specification of the control unit of the computer. The control unit consists of three components: the read-only microcode store ROM, the microcode program counter MPC and the decode unit DECODE. We must also supply the microcode program to complete the specification of the control unit.

ROM is a constant for the specification of a read-only device which, given a particular configuration of a 5-bit address space of 30-bit words of memory, outputs the word stored at the location addressed by the value of the input signal. FETCH5 is the built-in constant which names the operation of fetching a word from the memory.

```
|- ROM microcode (mpc,rom) =
    !t:num. rom t = FETCH5 microcode (mpc t)
```

The specification of the microcode program counter MPC is straightforward. This device is simply a 5-bit register that loads every clock cycle.

```
|- MPC (nextaddress,mpc) = !t:num. mpc (t+1) = nextaddress t
```

Before we can specify the decode unit, we need to define several operations for extracting fields of a microcode instruction word. CNTL_BIT is an operation which selects a single bit in the microcode instruction word. CNTL_FIELD selects a 2-bit field from the microcode instruction. The two 5-bit address fields, bits 3 to 7 and 8 to 12, are extracted by B_ADDR and A_ADDR. Finally, TEST extracts the test field, bits 0 to 2, which determines how the two address fields are used to construct the next microcode program address. Note that EL, V, SEG, BITS30 are built-in operations in the HOL system (see Appendix A).

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

The specification of the decode unit is considerably longer than the specification of any other primitive component in our implementation. Actually this specification is not as complex as it first appears. Moreover, it is reasonable to treat the decode unit as a primitive in our implementation because it is just a large combinational circuit that could be implemented almost automatically. The output equation for the signal nextaddress specifies how the next microcode program address is obtained from the two address fields and the test field in the microcode instruction word. The remaining equations identify individual bits or pairs of bits with signals that control the operation of the data path.

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

To complete the bottom level specification of the control unit, we need to specify the microcode program which we have already informally described. We specify the microcode program as a particular configuration of memory named by the constant MICROCODE. The specification consists of a set of 32 axioms, one axioms for each word in the microcode store.

```
|- FETCH5 MICROCODE #00000 = #000000000000000110000000001001
|- FETCH5 MICROCODE #00001 = #000000000000000000001000000011
|- FETCH5 MICROCODE #00010 = #010001000000000000000000000000
|- FETCH5 MICROCODE #00011 = #010000010000000000000000000000
|- FETCH5 MICROCODE #00100 = #001000100000000000001110000000
|- FETCH5 MICROCODE #00101 = #000000000000000100011000000001
|- FETCH5 MICROCODE #00110 = #001000100000000000010000000000
|- FETCH5 MICROCODE #00111 = #000100001000000000000000000000
|- FETCH5 MICROCODE #01000 = #000010000100000000100100000000
|- FETCH5 MICROCODE #01001 = #000000000000000000101000000100
|- FETCH5 MICROCODE #01010 = #000000000000000000000000000000
|- FETCH5 MICROCODE #01011 = #000001000010000000010100000000
|- FETCH5 MICROCODE #01100 = #000000000000000001000101011010
|- FETCH5 MICROCODE #01101 = #000000001001000001001100000000
|- FETCH5 MICROCODE #01110 = #000000001001000001011000000000
|- FETCH5 MICROCODE #01111 = #001000000010000001100000000000
|- FETCH5 MICROCODE #10000 = #001000000010000001100100000000
|- FETCH5 MICROCODE #10001 = #000000100000010001001000000000
|- FETCH5 MICROCODE #10010 = #000010000000001000010100000000
|- FETCH5 MICROCODE #10011 = #001000000010000001010000000000
|- FETCH5 MICROCODE #10100 = #000010000000100001010100000000
|- FETCH5 MICROCODE #10101 = #000000010000001001000100000000
|- FETCH5 MICROCODE #10110 = #001000000010000001011100000000
|- FETCH5 MICROCODE #10111 = #000010000000110001010100000000
|- FETCH5 MICROCODE #11000 = #000010010000000001000100000000
|- FETCH5 MICROCODE #11001 = #000100001000000001000100000000
|- FETCH5 MICROCODE #11010 = #000000000000000000000000000000
|- FETCH5 MICROCODE #11011 = #000000000000000000000000000000
|- FETCH5 MICROCODE #11100 = #000000000000000000000000000000
|- FETCH5 MICROCODE #11101 = #000000000000000000000000000000
|- FETCH5 MICROCODE #11110 = #000000000000000000000000000000
|- FETCH5 MICROCODE #11111 = #000000000000000000000000000000
```

We can now specify the control part as a top-level component of the computer implementation. Notice that the signals rom and nextaddress are hidden because they are completely internal to the control unit.

```
|- CONTROL
     microcode
     (mpc,knob,button,acc,ir,rsw,wmar,memcntl,
      wpc,rpc,wacc,racc,wir,rir,warg,alucntl,rbuf,ready,idle) =
    ?rom nextaddress.
     ROM microcode (mpc,rom) /\
     MPC (nextaddress,mpc) /\
     DECODE
      (rom,knob,button,acc,ir,nextaddress,rsw,wmar,memcntl,
       wpc,rpc,wacc,racc,wir,rir,warg,alucntl,rbuf,ready,idle)
```

The final step in the formal specification of the register-transfer implementation of Gordon's computer is the overall specification of the implementation as the composition of the data path and the control unit. Connecting the control unit with the data path allows us to hide the control signals generated by the control unit which govern the operation of the data path.

```
|- COMPUTER_IMP
    (mpc,mar,ir,arg,buf)
    (memory,knob,button,switches,pc,acc,idle,ready) =
    ?rsw wmar memcntl wpc rpc wacc racc wir rir warg alucntl rbuf.
     CONTROL
      MICROCODE
      (mpc,knob,button,acc,ir,rsw,wmar,memcntl,
       wpc,rpc,wacc,racc,wir,rir,warg,alucntl,rbuf,ready,idle) /\
     DATA
      (memory,mar,pc,acc,ir,arg,buf,switches,rsw,wmar,memcntl,
       wpc,rpc,wacc,racc,wir,rir,warg,alucntl,rbuf)
```

This completes the formal specification in higher order logic of the implementation of Gordon's computer. We now consider the target-level specification of the computer.

Target Level

For readability, we split the target level specification of the computer into two definitions. The first defines a constant EXECUTE which describes the execution of a single target level instruction. Hence, EXECUTE gives the semantics of the instruction set. The second definition defines a constant COMPUTER which is the overall specification of Gordon's computer including its behaviour when it is idling as well as when it is running. Notice that COMPUTER is defined in terms of EXECUTE.

The definition of EXECUTE evaluates to the next state of the computer which results from the execution of a single target level instruction. The next state is given as a 4-tuple consisting of the memory state, the values of the program counter and accumulator, and the run/idle status. The next state is obtained from the current memory state and the current values of the program counter and accumulator.

```
|- EXECUTE (memory_val,pc_val,acc_val) =
    let op   = VAL3(OPCODE(FETCH13 memory_val pc_val)) in
    let addr = CUT16_13(FETCH13 memory_val pc_val)     in
    ((op=0) => (memory_val, pc_val, acc_val, T) |
     (op=1) => (memory_val, addr, acc_val, F) |
     (op=2) =>
      ((VAL16 acc_val = 0) =>
       (memory_val, addr, acc_val, F) |
       (memory_val, INC13 pc_val, acc_val, F)) |
     (op=3) =>
    (memory_val, INC13 pc_val,
     ADD16 acc_val (FETCH13 memory_val addr), F) |
   (op=4) =>
    (memory_val, INC13 pc_val,
     SUB16 acc_val (FETCH13 memory_val addr), F) |
   (op=5) =>
    (memory_val, INC13 pc_val, FETCH13 memory_val addr, F) |
   (op=6) =>
    (STORE13 addr acc_val memory_val, INC13 pc_val, acc_val, F) |
    (memory_val, INC13 pc_val, acc_val, F))
```

The definition of COMPUTER is expressed in terms of an relationship between the value of the signals memory, knob, button, switches, pc, acc and idle from a time t1 to a time t2. We will see shortly that the choice of t1 and t2 is restricted in such a way that this relationship is at least a partial specification of the target level computer.

```
|- COMPUTER
    (t1,t2) (memory,knob,button,switches,pc,acc,idle) =
     (memory t2,pc t2,acc t2,idle t2) =
      (idle t1 =>
       (button t1 =>
        ((VAL2(knob t1) = 0) =>
          (memory t1, CUT16_13(switches t1), acc t1, T) |
         (VAL2(knob t1) = 1) =>
          (memory t1, pc t1, switches t1, T) |
         (VAL2(knob t1) = 2) =>
          (STORE13(pc t1)(acc t1)(memory t1), pc t1, acc t1, T) |
          (memory t1, pc t1, acc t1, F)) |
        (memory t1, pc t1, acc t1, T)) |
       (button t1 =>
        (memory t1, pc t1, acc t1, T) |
        EXECUTE(memory t1, pc t1, acc t1)))
```

Later on, we introduce another definition COMPUTER_abs which will ultimately serve as our target level specification of Gordon's computer. Nevertheless, the above definition of COMPUTER is the basis of an important step in the verification of the computer implementation. For the present, we accept the definition of COMPUTER as the target level specification with the understanding that certain assumptions such as the restriction on t1 and t2 are needed to make this definition useful as a target level specification.

5 Statement of Correctness

The implementation and target level specifications of Gordon's computer which we developed in the previous section only become interesting when they are used within a statement of correctness.

The first step in verifying Gordon's computer is to prove the correctness statement shown below. Correctness is expressed by the assertion that if the signals mpc, mar, ir, arg, buf, memory, knob, button, switches, pc, acc, idle and ready are in the relationship expressed by the predicate COMPUTER_IMP, then the values of the signals memory, knob, button, switches, pc, acc and idle at time t2 will be related to the values of the signals at time t1 in a relationship expressed by the predicate COMPUTER. Recall that COMPUTER_IMP and COMPUTER are the implementation and target specifications of the computer, respectively. The choice of the times t1 and t2 are restricted as follows. The boolean signal ready must be true at time t1 and furthermore, t2 must be the first time that ready is true after t1. This relationship between t1, t2 and ready is expressed by the predicate NEXT. There are also some extra assumptions about the stability of the signals knob and switches over the period of time from t1 to t2. The formal definitions of NEXT and STABLE are also shown below.

```
|- STABLE (t1,t2) sig =
    !t. ((t1 < t) /\ (t < t2)) ==> (((sig t):*) = ((sig t1):*))

|- NEXT (t1,t2) sig =
    (t1 < t2) /\ (sig t2)  /\ (!t. (t1 < t) /\ (t < t2) ==> ~sig t)

|- !t1 t2.
     COMPUTER_IMP
      (mpc,mar,ir,arg,buf)
      (memory,knob,button,switches,pc,acc,idle,ready) /\
     STABLE (t1,t2) switches /\
     STABLE (t1,t2) knob /\
     NEXT (t1,t2) ready /\
     ready t1 ==>
     COMPUTER (t1,t2) (memory,knob,button,switches,pc,acc,idle)
```

Examining the microcode program, we can see that the signal ready is only true when the microcode program counter is 0 or 5. Location 0 in the microcode program is the start of the idle cycle and location 5 is the start of the cycle which fetches, decodes and executes a target level instruction. This explains the restriction on t1 and t2 in terms of ready. We use ready to identify precisely the intervals described by the predicate COMPUTER.

6 Formal Verification - Part 1

The formal verification of Gordon's computer is divided into two major parts. The first part involves proving the correctness statement presented in the previous section. Having proved this statement, we introduce a better statement of correctness which is based on a temporal abstraction between the *microcode time scale* and the *instruction set time scale*. We elaborate on the difference between these two time scales later in this paper. The proof of this improved statement of correctness is based upon the proof of the statement of correctness presently under consideration.

We begin by stating several preliminary lemmas about the relationship expressed by the predicate NEXT. These lemmas may be proved without difficulty from the definition of NEXT.

```
NEXT_INC_LEMMA =
 |- !sig t1. sig (t1 + 1) ==> NEXT (t1,t1 + 1) sig

NEXT_IDENTITY_LEMMA =
 |- !sig t1 t2 t2'.
     NEXT (t1,t2) sig /\ NEXT (t1,t2') sig ==> (t2 = t2')

NEXT_INC_INTERVAL_LEMMA =
 |- !sig t1 t2.
     ~sig (t1 + 1) /\ NEXT (t1 + 1,t2) sig ==> NEXT(t1,t2) sig
```

A number of simple arithmetic lemmas are also required such as `4 + 10 = 14`. These lemmas are easily proved in the HOL system.

The first step in the verification of the computer implementation is to fully 'expand' the definition of COMPUTER_IMP. Recall that COMPUTER_IMP is defined in terms of CONTROL and DATA which defined in terms of primitive components. We begin by expanding CONTROL and DATA. We then use the result of expanding CONTROL and DATA to expand COMPUTER_IMP.

Expanding the specification of a composite device involves three steps. First, we replace constants such as PC with the specifications of the components they name. For example, the term `PC (bus,wpc,pc)` in the specification of DATA is replaced with the following:

```
!t:num. pc (t+1) = ((wpc t) => (CUT16_13 (bus t)) | (pc t))
```

The second step is to 'unwind' the equations for all of the output signals. The result of unwinding these equations is that left-hand sides of equations are eliminated as much as possible in favor of right-hand sides in the right-hand sides of all the equations. The final step in expanding a composite specification is to 'prune away' equations for hidden signals. Recall that signals are 'hidden' by existential quantification. If the name of a hidden signal still appears in the right-hand side of an equation for a unhidden signal, this final step will fail. This is because the only signals which should appear on the right-hand sides of equations after unwinding are signals which serve as input to the composite device, that is, signals which are not internal.

We note that the expansion of a composite specification is a purely logical procedure. Fortunately, the HOL system provides a ML routine which automatically expands a composite specification in the manner we have described. This ML routine, EXPANDF, manipulates the composite specification in a strictly logical manner using the usual inference rules.

The result of fully expanding COMPUTER_IMP is a theorem with a conclusion several pages in length (see Appendix B). The conclusion is a term consisting of equations for each of the output signals of the computer implementation, namely, the signals mpc, mar, ir, arg, buf, memory, pc, acc, idle and ready. For example, the equation for the mpc signal is shown below. The theorem captures the complete behaviour of the register-transfer level implementation of the computer and is used as the unique source of information about the implementation for the rest of the proof of correctness.

```
(!t.
  mpc(t + 1) =
  (((V(SEG(0,2)(BITS30(FETCH5 MICROCODE(mpc t)))) = 1) /\
    button t) =>
   WORD5(V(SEG(3,7)(BITS30(FETCH5 MICROCODE(mpc t))))) |
   (((V(SEG(0,2)(BITS30(FETCH5 MICROCODE(mpc t)))) = 2) /\
     (VAL16(acc t) = 0)) =>
    WORD5(V(SEG(3,7)(BITS30(FETCH5 MICROCODE(mpc t))))) |
    ((V(SEG(0,2)(BITS30(FETCH5 MICROCODE(mpc t)))) = 3) =>
     WORD5
     ((VAL2(knob t)) +
      (VAL5
       (WORD5
        (V(SEG(8,12)(BITS30(FETCH5 MICROCODE(mpc t)))))))) |
     ((V(SEG(0,2)(BITS30(FETCH5 MICROCODE(mpc t)))) = 4) =>
      WORD5
      ((VAL3(OPCODE(ir t))) +
       (VAL5
        (WORD5
         (V(SEG(8,12)(BITS30(FETCH5 MICROCODE(mpc t)))))))) |
      WORD5(V(SEG(8,12)(BITS30(FETCH5 MICROCODE(mpc t))))))))))
```

It is important to understand how the the theorem in Appendix B can be used to determine the behaviour of the computer implementation under a certain set of assumptions. Suppose, for example that we assume the microcode program counter has the value #00000 at time t1. Furthermore, suppose that the button is not pressed at time t1. Using higher order logic, we can investigate the behaviour of the computer implementation under these assumptions. For instance, we can determine the value of the microcode program counter at the time (t1 + 1) by evaluating the equation for the output signal mpc under our assumptions about the value of mpc and button at t1.

First we instantiate the universally quantified variable 't' in the equation for mpc to t1. We then rewrite the equation with our assumptions which are formally expressed as `mpc t1 = #00000` and `button t1 = F` to obtain the following.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00000,
button t1 = F
|- mpc(t1 + 1) =
    (((V(SEG(0,2)(BITS30(FETCH5 MICROCODE #00000))) = 1) /\ F) =>
     WORD5(V(SEG(3,7)(BITS30(FETCH5 MICROCODE #00000)))) |
     (((V(SEG(0,2)(BITS30(FETCH5 MICROCODE #00000))) = 2) /\
       (VAL16(acc t1) = 0)) =>
      WORD5(V(SEG(3,7)(BITS30(FETCH5 MICROCODE #00000)))) |
      ((V(SEG(0,2)(BITS30(FETCH5 MICROCODE #00000))) = 3) =>
       WORD5
       ((VAL2(knob t1)) +
        (VAL5
         (WORD5
          (V(SEG(8,12)(BITS30(FETCH5 MICROCODE #00000)))))) |
       ((V(SEG(0,2)(BITS30(FETCH5 MICROCODE #00000))) = 4) =>
        WORD5
        ((VAL3(OPCODE(ir t1))) +
         (VAL5
          (WORD5
           (V(SEG(8,12)(BITS30(FETCH5 MICROCODE #00000)))))) |
        WORD5(V(SEG(8,12)(BITS30(FETCH5 MICROCODE #00000)))))))
```

Note that there are three hypotheses before the theoremhood symbol, '|-'. These are the two assumptions we have just introduced along with the overall hypothesis that the signals mpc, mar, ir, arg, buf, memory, knob, button, switches, pc, acc, idle and ready are in the relationship expressed by COMPUTER_IMP.

The next step in the evaluation is to use our specification of the microcode program to replace the term FETCH5 MICROCODE #00000 with the appropriate 30-bit word. In particular, we use the axiom for location 0 to rewrite the above partially evaluated theorem.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00000,
button t1 = F
|- mpc(t1 + 1) =
    (((V(SEG(0,2)
        (BITS30 #000000000000000110000000001001)) = 1) /\ F) =>
     WORD5(V(SEG(3,7)(BITS30 #000000000000000110000000001001))) |
    (((V(SEG(0,2)
        (BITS30 #000000000000000110000000001001)) = 2) /\
      (VAL16(acc t1) = 0)) =>
     WORD5(V(SEG(3,7)(BITS30 #000000000000000110000000001001))) |
    ((V(SEG(0,2)(BITS30 #000000000000000110000000001001)) = 3) =>
     WORD5((VAL2(knob t1)) +
      (VAL5
       (WORD5
        (V(SEG(8,12)(BITS30 #000000000000000110000000001001)))))) |
    ((V(SEG(0,2)(BITS30 #000000000000000110000000001001)) = 4) =>
     WORD5((VAL3(OPCODE(ir t1))) +
      (VAL5
       (WORD5
        (V(SEG(8,12)(BITS30 #000000000000000110000000001001)))))) |
     WORD5
      (V(SEG(8,12)(BITS30 #000000000000000110000000001001))))))
```

We now use some special purpose rules in HOL which evaluate terms containing the built-in constants V, SEG, BITS30 and WORD5 (see Appendix A). The result of BITS_RULE followed by successive applications of SEG_RULE, V_RULE, WORD_RULE and VAL_RULE is shown below.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
(mpc t1 = #00000),
(button t1 = F)
|- mpc(t1 + 1) =
    (((1 = 1) /\ F) => #00001 |
     (((1 = 2) /\ (VAL16(acc t1) = 0)) => #00001 |
      ((1 = 3) => WORD5((VAL2(knob t1)) + 0) |
       ((1 = 4) => WORD5((VAL3(OPCODE(ir t1))) + 0) | #00000))))
```

We complete the evaluation by using EQ_RULE which replaces the numerical inequalities with F, bool_RULE which simplifies boolean expressions, and finally COND_RULE which simplifies conditional expressions. The final result is the following theorem.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00000,
button t1 = F
|- mpc(t1 + 1) = #00000
```

The above theorem shows that, under our assumptions that the value of the microcode program counter is #00000 at time t1 and the button is not pressed, the value of the microcode program counter will be #00000 at time (t1 + 1) which is exactly what we would expect.

This simple example demonstrates the evaluation of output equations under a set of assumptions about initial values. We have shown how the value of the signal mpc at (t1 + 1) can be obtained on the basis of its value at t1 and the value of the signal button at t1.

Suppose that the button had been pressed at t1 and that the knob is in position 3. By a similar evaluation process, we would obtain the following.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00000,
button t1 = T,
VAL2 (knob(t1 + 1)) = 3
|- mpc(t1 + 1) = #00001
```

So far we have just seen how to determine the value of mpc at time (t1 + 1) from assumptions about the values of signals at time t1. We now show that this method can be extended to determine the value of mpc at time (t1 + n) for any n.

Suppose that we wish to determine the value of mpc at time ((t1 + 1) + 1) using the above result, `mpc (t1 + 1) = #00001`. We would begin by instantiating the universally quantified variable 't' to (t1 + 1) and then rewrite the equation with the above result that `mpc (t1 + 1) = #00001`.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00000,
button t1 = T,
VAL2 (knob(t1 + 1)) = 3
|- mpc((t1 + 1) + 1) =
    (((V(SEG(0,2)(BITS30(FETCH5 MICROCODE #00001))) = 1) /\
      button(t1 + 1)) =>
     WORD5(V(SEG(3,7)(BITS30(FETCH5 MICROCODE #00001)))) |
     (((V(SEG(0,2)(BITS30(FETCH5 MICROCODE #00001))) = 2) /\
       (VAL16(acc(t1 + 1)) = 0)) =>
      WORD5(V(SEG(3,7)(BITS30(FETCH5 MICROCODE #00001)))) |
      ((V(SEG(0,2)(BITS30(FETCH5 MICROCODE #00001))) = 3) =>
       WORD5
       (3 +
        (VAL5
         (WORD5
          (V(SEG(8,12)(BITS30(FETCH5 MICROCODE #00001)))))) |
       ((V(SEG(0,2)(BITS30(FETCH5 MICROCODE #00001))) = 4) =>
        WORD5
        ((VAL3(OPCODE(ir(t1 + 1)))) +
         (VAL5
          (WORD5
           (V(SEG(8,12)(BITS30(FETCH5 MICROCODE #00001)))))) |
        WORD5(V(SEG(8,12)(BITS30(FETCH5 MICROCODE #00001)))))))
```

Then by the usual evaluation procedure we would obtain the following.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00000,
button t1 = T,
VAL2 (knob(t1 + 1)) = 3
|- mpc((t1 + 1) + 1) = #00101
```

Not only can we investigate the behaviour of the signal mpc in this manner, but the value of any other signals can be determine by a similar evaluation procedure. For example, we can derive the following theorem about the value of the accumulator at time (((t1 + 1) + 1) + 1) under the initial assumptions `mpc t1 = #00000`, `button t1 = T` and `VAL2 (knob (t1 + 1)) = 1`. As we would expect, the value of the switches is loaded into the accumulator.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00000,
button t1 = T,
VAL2 (knob(t1 + 1)) = 1
|- acc(((t1 + 1) + 1) + 1) = switches((t1 + 1) + 1)
```

In general, it is possible to derive a theorem about a future value of any one of the output signals in the computer implementation. With the necessary assumptions about previous values of signals, for example, the initial value of the microcode program counter, the future value of a particular signal will be simplified to a primitive value such as #00000. We have offered several examples of the evaluation procedure which produces these theorems. Note that this procedure is a strictly logical derivation in higher order logic. Even the special purpose rules, BITS_RULE, SEG_RULE, V_RULE, WORD_RULE and VAL_RULE are really built-in inference patterns which use derived inference rules originating from theories about bits, words and lists.

Now that we have seen how to obtain results about the values of output signals we can continue with the verification of the register-transfer implementation of the computer.

We now prove several simple theorems which are intuitively obvious from examination of the microcode program. We can be see from examining the microcode program that the only microcode instruction which enables both the ready and idle signals is in location 0. Hence, if both of these signals are true at time t1, then the microcode program counter must hold the value #00000. Similarly, when ready is true but idle is false, then the value of the microcode program counter must be #00101. These observations can be formally proved by a simple evaluation of the equations for the ready and idle signals for every possible value of the signal mpc. Note that since the microcode program counter is a 5-bit register, there are exactly 32 cases to consider.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
ready t1,
idle t1
|- mpc t1 = #00000

COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
ready t1,
~idle t1
|- mpc t1 = #00101
```

Next we observe that the ready will be true infinitely often. Starting at any location we can see that the microcode program counter must always return to either location 0 or 5. When the computer is operating normally, we know that it will always return to start of either the idle loop or the instruction execution cycle which begin at locations 0 and 5, respectively. However, we must also consider what could happen when the computer powers up. [7] Upon power up, the microcode program counter could take any 5-bit value including those locations, 26 - 31, which are in the 'unused' section of the microcode store. [8] We must exhaustively demonstrate that from any possible location in the microcode store, the control will return to either location 0 or 5. By repeated evaluation of the equation for mpc, we can show that mpc must eventually have the value 0 or 5 for every possible initial value of mpc. By a simple evaluation of ready for mpc equal to #00000 or #00101, we can then prove that ready will be true infinitely often. This result is expressed by the theorem below.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready)
```

[7] In fact, verification failed to expose a design error concerning power up which was later discovered when the design was implemented. See the discussion in the 'Epilogue' section.

[8] Gordon's original specification of the microcode program ignored the contents of the 'unused' section of the microcode store. Without a complete specification of the microcode program, it would have been impossible to prove the above three theorems. Moreover, one could easily construct a case for which the implementation fails to power up 'correctly' if the unused section of the microcode store is left unspecified.

```
|- (!t1. ?t2. t1 < t2 /\ ready t2)
```

The next step in the verification of Gordon's computer is the cornerstone of the proof. Recall the restriction placed on t1 and t2 in the correctness statement where the signal ready must be true at time t1 and next true at time t2. We have already said that this interval corresponds to a single iteration of the idle loop beginning with the microcode instruction in location 0 or the execution of a single target level instruction beginning at location 5. Examination of the microcode program shows that there are exactly fifteen different possible execution paths where an execution path is a sequence of microcode instruction from t1 to t2. In particular, there are five different possible execution paths which begin at location 0 and ten different possible execution paths beginning at location 5. The next step in the proof is to examine the behaviour of the computer implementation for each one these fifteen different possible execution paths.

Using the evaluation procedure described above, we can determine the stepwise behaviour of the entire implementation in what resembles a simulation of the machine. The most important difference between the simulation we proposed to do here and what is usually meant by a 'simulation' is that instead of actual 'values', the state of the machine in the simulation will be represented by a collection of theorems. These theorems will describe the current value of each output signal in terms of the prior values of input and output signals.

As an example of how we propose to simulate the behaviour of the computer, we consider the simulation of one of the fifteen possible execution paths, in particular, the path which carries out the execution of the target level instruction ADD.

We can obtain the first state in the simulation with the following assumptions which determine the particular execution path under consideration.

```
mpc t1 = #00101
button t1 = F
VAL3 (OPCODE (FETCH13 (memory t1) (pc t1))) = 3
```

By evaluating the equations for each of the output signals with 't' instantiated to t1, we obtain the first state in simulation at time (t1 + 1). Notice that the state at time (t1 + 1) is described in terms of the state at time t1. The '....' stand for the four hypothesis in each theorem, namely the three assumptions above along with the hypothesis that the signals mpc, mar, ir, arg, buf, memory, knob, button, switches, pc, acc, idle and ready are in the relationship expressed by the predicate COMPUTER_IMP.

```
.... |- buf(t1+1) = DEST_TRI16 FLOAT16
.... |- memory(t1+1) = memory t1
```

```
.... |- mar(t1+1) = mar t1
.... |- pc(t1+1) = pc t1
.... |- acc(t1+1) = acc t1
.... |- ir(t1+1) = ir t1
.... |- arg(t1+1) = arg t1
.... |- mpc(t1+1) = #00110
```

From this state we can obtain the second state and subsequently the third state at times ((t1+1)+1) and (((t1+1)+1)+1), respectively. We can see that by the end of the third cycle the target level instruction word addressed by the program counter has been fetched and loaded into the instruction register.

```
.... |- buf((t1+1)+1) = PAD13_16(pc t1)
.... |- memory((t1+1)+1) = memory t1
.... |- mar((t1+1)+1) = pc t1
.... |- pc((t1+1)+1) = pc t1
.... |- acc((t1+1)+1) = acc t1
.... |- ir((t1+1)+1) = ir t1
.... |- arg((t1+1)+1) = arg t1
.... |- mpc((t1+1)+1) = #01000

.... |- buf(((t1+1)+1)+1) = FETCH13(memory t1)(pc t1)
.... |- memory(((t1+1)+1)+1) = memory t1
.... |- mar(((t1+1)+1)+1) = pc t1
.... |- pc(((t1+1)+1)+1) = pc t1
.... |- acc(((t1+1)+1)+1) = acc t1
.... |- ir(((t1+1)+1)+1) = FETCH13(memory t1)(pc t1)
.... |- arg(((t1+1)+1)+1) = arg t1
.... |- mpc(((t1+1)+1)+1) = #01001
```

Seven more iterations in this manner would complete the simulation of the microcode instruction sequence implementing the ADD instruction. The final state is shown below. [9]

```
.... |- buf((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1) =
         INC16(PAD13_16(pc t1))
.... |- memory((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1) =
         memory t1
.... |- mar ((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1) =
```

[9]The states for each of the ten steps in this example are shown in Appendix C.

```
          CUT16_13(FETCH13(memory t1)(pc t1))
 .... |- pc(((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          INC13(pc t1)
 .... |- acc(((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          ADD16
           (acc t1)
           (FETCH13
            (memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))))
 .... |- ir(((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          FETCH13(memory t1)(pc t1)
 .... |- arg(((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          acc t1
 .... |- mpc(((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          #00101
```

At the end of this sequence, the microcode program counter would hold the value #00101, the state of the memory would be unchanged, and the program counter would be incremented. Furthermore, the accumulator would hold the sum of the previous value in the accumulator and the memory value addressed by the address field of the target level instruction word. This value is denoted by:

```
ADD16
 (acc t1)
 (FETCH13(memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))))
```

We record the results of the simulation for this particular execution path in a single theorem. The theorem only records the final values of the signals which are part of the target level specification, namely, memory, pc, acc and idle. The theorem also records that the ready signal is true at both time t1 and time (((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) but false at all times in between. The value of the ready signal at these times is easily derived from the successive values of the microcode program counter.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00101,
button t1 = F,
VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 3
|- (memory ((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1)
    = memory t1) /\
(pc(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)
 = INC13(pc t1)) /\
(acc (((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) =
 ADD16
  (acc t1)
  (FETCH13(memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))))) /\
~idle (((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) /\
ready (((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) /\
~ready ((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) /\
~ready (((((((t1+1)+1)+1)+1)+1)+1)+1) /\
~ready ((((((t1+1)+1)+1)+1)+1)+1) /\
~ready (((((t1+1)+1)+1)+1)+1) /\
~ready ((((t1+1)+1)+1)+1) /\
~ready (((t1+1)+1)+1) /\
~ready ((t1+1)+1) /\
~ready ((t1+1)+1) /\
~ready (t1+1) /\
ready t1
```

Similar theorems can be derived for each of the other fourteen possible execution paths using this simulation procedure. Collectively, these theorems capture the behaviour of the implementation. The rest of the verification is really a matter of refining these results and stitching them together into the desired correctness statement.

The next major step in the verification of Gordon's computer removes all terms of the form `(...t1 ...+1)` from the fifteen theorems obtained in the previous step. The result will only contain t1 and t2 as terms denoting points in time.

At this point, we introduce one of the hypotheses in the correctness statement, in particular, the hypothesis that t2 is the first time that the signal ready is true after t1 which is expressed as `NEXT (t1,t2) ready`.

The theorem obtained in the previous step records that the signal ready is true at time t1 and then false until $(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)$. Using the NEXT_INC_LEMMA and NEXT_INC_INTERVAL_LEMMA we can show:

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00101,
button t1 = F,
VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 3
|- NEXT(t1,(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)) ready
```

Then by NEXT_IDENTITY_LEMMA and the hypothesis NEXT (t1,t2) ready, we can establish that ((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1) is identical with t2. This allows us to replace occurrences of this term with t2 in the theorem obtained in the previous step. Similarly for the other fourteen theorems, the term denoting the time when the signal ready first becomes true after t1 can be identified with and replaced by t2.

For several of the execution paths, in particular those execution paths where the value of the knob or switches signals is used, the theorems obtained by the simulation of the execution path will contain time terms in between t1 and t2. By introducing hypotheses about the stability of the knob and switches signals between t1 and t2, we can replace these intermediate time terms with t1.

The result of this step in the verification of Gordon's computer is a simplification of the fifteen theorems obtained in the previous step. The theorems are simplified by eliminating all time terms except t1 and t2. The theorems are further simplified by eliminating information about all signals except for memory, pc, acc and idle. For example, the result of this simplification for the ADD execution path is the following.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
NEXT(t1,t2)ready,
mpc t1 = #00101,
button t1 = F,
VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 3
|- (memory t2 = memory t1) /\
   (pc t2 = INC13(pc t1)) /\
   (acc t2 =
    ADD16
     (acc t1)
     (FETCH13(memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))))) /\
   ~idle t2
```

The next step in the verification procedure is another refinement of the theorems for each of the fifteen possible execution paths.

Each of the fifteen theorems from the previous step have as an assumption either mpc t1 = #00000 or mpc t1 = #00101. This step in the verification procedure 'trades in' this assumption for assumptions about the signals idle and ready at time t1. Using previously mentioned theorems, we can replace the assumption mpc t1 = #00000 with the assumptions ready t1 and idle t1. Similarly, we can replace the assumption mpc t1 = #00101 with the assumptions ready t1 and ~idle t1.

The second refinement in this stage of the proof is to restate each of the fifteen theorems as an equation for the four-tuple, (memory t2, pc t2, acc t2, idle t2). The result of these refinements for the theorem about the ADD execution path is the following.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
NEXT(t1,t2)ready,
~idle t1,
ready t1,
button t1 = F,
VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 3
|- memory t2, pc t2, acc t2, idle t2 =
    memory t1, INC13(pc t1),
     ADD16
      (acc t1)
      (FETCH13(memory t1)(CUT16_13(FETCH13(memory t1)(pc t1)))), F
```

With this refinement, we obtain fifteen theorems which state, for each of the fifteen possible execution paths, the behaviour expressed by the correctness statement on page 18. The fifteen theorems are listed in Appendix D.

At this point, this correctness statement may be easily proved by a case analysis on the possible values of the signals idle, ready and knob at time t1 and on the seven possible values of the 3-bit opcode field in the instruction word addressed by the program counter at t1. This result concludes the first part of the verification of Gordon's computer.

7 An Improved Statement of Correctness

In many respects, the proof of the correctness statement on page 18 is a acceptable verification of the register-transfer implementation of Gordon's computer. The correctness statement states that the implementation has a behaviour which matches our specification of the computer as a register-transfer device. Nevertheless, the correctness statement on page 18 is not entirely satisfactory because the target level specification of the computer is stated as if we intended to use the computer as a register-transfer device. However, our informal description of the computer is clearly aimed at a level above the execution of microcode instructions at the register-transfer level. The target level description is stated in terms of assembler language instructions such as ADD. Moreover, the execution of a target level instruction is presented an event which occurs in a single time interval instead of some number of microcycles. This distinction leads to the recognition of two different but related time scales.

As we have seen, Gordon's computer is implemented as a device consisting of register-transfer level devices such as registers, combinational circuits, a knob, a button, switches and memories. At the register-transfer level, the microcoded computer has a behaviour which consists of fetching and executing microcode instructions stored in the microcode store. The time scale at this level is defined by the system-wide clocking of these devices. Each time interval corresponds to the execution of one microcode instruction. We refer to this time scale as the *microcode time scale.*

A second time scale is the *instruction set time scale.* A target level view of the behaviour of Gordon's computer consists of the execution of one of the eight instructions, HALT, JMP, JZR, ADD, SUB, LOAD, STORE, and SKIP in a single interval of time. Because the execution of one of these instructions involves the execution of several microcode instructions, the *instruction set time scale* is a temporal abstraction of the *microcode time scale.* An illustrative view of these two time scales during the execution of a short sequence of instructions is shown below.

Instruction	Microcode Program Counter	Microcode Time Scale	Instruction Set Time Scale
JMP	5	0	0
	6	1	
	8	2	
	9	3	
	11	4	
ADD	5	5	1
	6	6	
	8	7	
	9	8	
	13	9	
	19	10	
	20	11	
	21	12	
	17	13	
	18	14	
HALT	5	15	2
	6	16	
	8	17	
	9	18	
	10	19	
	0	20	3

Our goal is to prove a statement of correctness for a target level specification which states the behaviour of the computer at the assembler language level, that is, in terms of the *instruction set time scale.*

A target level specification of the computer stated in terms of the *instruction set time scale* consists of assertions about signals defined at this time scale. Up until this point in our discussion, all signals have been defined at the same time scale, namely, the *microcode time scale.* However, there is no difficultly in defining a second set of signals for a different time scale, in particular, the *instruction set time scale.* Just as we have treated signals defined for the *microcode time scale* as mappings from numbers to primitive values in higher order logic, we shall likewise treat signals defined for the *instruction set time scale* as mappings from numbers to primitive values. Nevertheless, we shall be careful not to confuse these two time scales, and for this reason we adopt the convention of using the suffix '_abs' to denote a signal defined at the *instruction set time scale.*

Our final version of the target level specification of Gordon's computer is shown below. The important difference between the definition of COMPUTER_abs and the earlier definition of COMPUTER is that t1 and t2 are no longer formal parameters which must be specified when the specification is used in a correctness statement. Rather, COMPUTER_abs describes a relationship between the signals memory_abs, knob_abs, button_abs, switches_abs, pc_abs, acc_abs and idle_abs over adjacent points of time on the *instruction set time scale.*

```
|- COMPUTER_abs
    (memory_abs,
     knob_abs,button_abs,switches_abs,pc_abs,acc_abs,idle_abs) =
    !t:num.
     (memory_abs (t+1),pc_abs (t+1),acc_abs (t+1),idle_abs (t+1)) =
     (idle_abs t =>
      (button_abs t =>
       ((VAL2(knob_abs t) = 0) =>
        (memory_abs t, CUT16_13(switches_abs t), acc_abs t, T) |
        (VAL2(knob_abs t) = 1) =>
         (memory_abs t, pc_abs t, switches_abs t, T) |
        (VAL2(knob_abs t) = 2) =>
         (STORE13(pc_abs t)(acc_abs t)(memory_abs t),
          pc_abs t, acc_abs t, T) |
         (memory_abs t, pc_abs t, acc_abs t, F)) |
       (memory_abs t, pc_abs t, acc_abs t, T)) |
      (button_abs t =>
       (memory_abs t, pc_abs t, acc_abs t, T) |
       EXECUTE(memory_abs t, pc_abs t, acc_abs t)))
```

We now consider a statement of correctness for the computer implementation which uses this new target level specification. The specification of the implementation describes a relationship between signals defined in terms of the *microcode time scale* whereas the target level specification describes a relationship between signals defined in terms of the *instruction set time scale*. As we warned earlier, these two sets of signals cannot be confused. Instead we must formally define a relationship between these two sets of signals based on our suggestion that the *instruction set time scale* is an abstraction of the *microcode time scale*. This relationship between the two sets of signals will form the necessary link between the implementation level specification and the target level specification.

(abs ready) [10] [11] is a function which maps the nth point on the *instruction set time scale* to a point on the *microcode time scale*. The primitive recursive definition of **abs** is shown below. [12] Note that '@' is the symbol used in the HOL system for Hilbert's epsilon operator which may be read as 'the t such that ...'.

```
|- abs signal 0 = @t. signal t /\ (!t'. t' < t ==> ~signal t')
|- abs signal n+1 =
    @t.
     signal t /\
     (abs n signal) < t /\
     (!t'. (abs n signal) < t' < t ==> ~signal t')
```

The function (abs ready) simply identifies points on the *microcode time scale* when the signal ready is true. That is, (abs ready n) is the nth point on the *microcode time scale* when the signal ready is true. For example, (abs ready 2) = 15 for the execution sequence on page 33.

Once we have defined the *instruction set time scale* as a function of the *microcode time scale*, the next step is to introduce a set of signals defined at the *instruction set time scale* which are simply the result of looking at memory, knob, button, switches, pc, acc and idle only when the signal ready is true. memory_abs, knob_abs, button_abs, switches_abs, pc_abs, acc_abs and idle_abs are easily defined in terms of the function abs as shown below. For example, memory_abs is defined as the *instruction set time scale* signal which maps n to the value of the *microcode time scale* signal memory at the time (abs ready n), that is, the nth time ready is true. The reader is reminded of our convention of denoting *instruction set time scale* signals with the suffix '_abs'.

[10] The function **abs** and its definition were motivated by the research work of T.Melham [4].

[11] **abs** is a function of type *:(num→bool)→num→num* and ready is a function of type *:num→bool* making (**abs ready**) a function of type *:num→num*.

[12] A typographical error in the definition of **abs** as it appeared in the original publication of this report, 'Research Report 85/208/21, Department of Computer Science, University of Calgary, August 1985', has been corrected here.

```
memory_abs n = memory (abs ready n)
knob_abs n = knob (abs ready n)
button_abs n = button (abs ready n)
switches_abs n = switches (abs ready n)
pc_abs n = pc (abs ready n)
acc_abs n = acc (abs ready n)
idle_abs n = idle (abs ready n)
```

Having defined the above *instruction set time scale* signals in terms of *microcode time scale* signals, we can present a correctness statement which relates the register-transfer level implementation to our target level specification. The following is our final version of the correctness statement for the register-transfer implementation of Gordon's computer.

```
|- COMPUTER_IMP
    (mpc,mar,ir,arg,buf)
    (memory,knob,button,switches,pc,acc,idle,ready) /\
   (!n. STABLE(abs ready n,abs ready (n+1))switches) /\
   (!n. STABLE(abs ready n,abs ready (n+1))knob) /\
   (!n.
     (m_abs n = m(abs ready n)) /\
     (knob_abs n = knob(abs ready n)) /\
     (button_abs n = button(abs ready n)) /\
     (switches_abs n = switches(abs ready n)) /\
     (pc_abs n = pc(abs ready n)) /\
     (acc_abs n = acc(abs ready n)) /\
     (idle_abs n = idle(abs ready n))) ==>
   COMPUTER_abs
    (memory_abs,
     knob_abs,button_abs,switches_abs,pc_abs,acc_abs,idle_abs)
```

In this correctness statement we have assumptions about the stability of the *microcode time scale* signals switches and knob. While it is necessary to include these assumptions, given the implementation described by Gordon, the need for these assumptions points to what may be considered a design flaw in the implementation. We reserve further comment until later in this report.

1 Formal Verification - Part 2

We complete the verification of Gordon's computer by proving the statement of correctness given at the end of the previous section.

A preliminary step is to establish that the function (abs ready) is well-defined. A theorem stating that this function is well-defined can be proven from an earlier result that the signal ready will be true infinitely often. This theorem is shown below.

```
|- ?t.
      ready t /\
      (abs ready n) < t /\
      (!t'. (abs ready n) < t' /\ t' < t ==> ~ready t')
```

At this point we outline our strategy for proving this correctness statement. We start with the *microcode time scale* correctness statement shown on page 18 with t1 and t2 instantiated as (abs ready n) and (abs ready (n+1)). We then prove the following two lemmas.

```
|- ready (abs ready n)

|- NEXT ((abs ready n),(abs ready (n+1))) ready
```

Both of these lemmas follow immediately from the theorem that function (abs ready) is well-defined. [13] These two lemmas, along with assumptions about the stability of switches and knob, allows us to use the implementation level statement of correctness, proved in the first part of the verification procedure, to obtain:

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
(!n. STABLE(abs ready n,abs ready (n+1))switches),
(!n. STABLE(abs ready n,abs ready (n+1))knob)
|- COMPUTER
    ((abs ready n),(abs ready (n+1)))
    (memory,knob,button,switches,pc,acc,idle)
```

Rewriting with the definition of COMPUTER, we then obtain a theorem with a conclusion containing terms such as memory (abs ready n) and memory (abs ready (n+1)).

[13] An axiom that 'whenever there exists a number with some property, then there exists a least number with that property' was introduced at this point. This axiom will be proved as a theorem in the future.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
(!n. STABLE(abs ready n,abs ready (n+1))switches),
(!n. STABLE(abs ready n,abs ready (n+1))knob)
|- memory(abs ready (n+1)),
    pc(abs ready (n+1)),
     acc(abs ready (n+1)),
      idle(abs ready (n+1)) =
    (idle(abs ready n) =>
     (button(abs ready n) =>
      ((VAL2(knob(abs ready n)) = 0) =>
       (memory(abs ready n),
        CUT16_13(switches(abs ready n)),acc(abs ready n),T) |
      ((VAL2(knob(abs ready n)) = 1) =>
       (memory(abs ready n),
        pc(abs ready n),switches(abs ready n),T) |
      ((VAL2(knob(abs ready n)) = 2) =>
       (STORE13
         (pc(abs ready n))(acc(abs ready n))(memory(abs ready n)),
        pc(abs ready n),acc(abs ready n),T) |
       (memory(abs ready n),
        pc(abs ready n),acc(abs ready n),F)))) |
      (memory(abs ready n),pc(abs ready n),acc(abs ready n),T)) |
     (button(abs ready n) =>
      (memory(abs ready n),pc(abs ready n),acc(abs ready n),T) |
      EXECUTE
       (memory(abs ready n),pc(abs ready n),acc(abs ready n))))
```

We now rewrite with the definitions of the signals memory_abs, knob_abs, button_abs, switches_abs, pc_abs, acc_abs and idle_abs to obtain:

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
(!n. STABLE(abs ready n,abs ready (n+1))switches),
(!n. STABLE(abs ready n,abs ready (n+1))knob),
(!n.
  (m_abs n = m(abs ready n)) /\
  (knob_abs n = knob(abs ready n)) /\
  (button_abs n = button(abs ready n)) /\
  (switches_abs n = switches(abs ready n)) /\
  (pc_abs n = pc(abs ready n)) /\
  (acc_abs n = acc(abs ready n)) /\
  (idle_abs n = idle(abs ready n)))
|- memory_abs(n+1),pc_abs(n+1),acc_abs(n+1),idle_abs(n+1) =
     (idle_abs n  =>
      (button_abs n  =>
       ((VAL2(knob_abs n ) = 0) =>
        (memory_abs n ,CUT16_13(switches_abs n ),acc_abs n ,T) |
       ((VAL2(knob_abs n ) = 1) =>
        (memory_abs n ,pc_abs n ,switches_abs n ,T) |
       ((VAL2(knob_abs n ) = 2) =>
        (STORE13(pc_abs n )(acc_abs n )(memory_abs n ),
         pc_abs n ,acc_abs n ,T) |
        (memory_abs n ,pc_abs n ,acc_abs n ,F))) |
       (memory_abs n ,pc_abs n ,acc_abs n ,T)) |
      (button_abs n  =>
       (memory_abs n ,pc_abs n ,acc_abs n ,T) |
       EXECUTE(memory_abs n ,pc_abs n ,acc_abs n )))
```

Generalizing for the free variable 'n' followed by alpha-converting 'n' to 't' results in the definition of COMPUTER_abs, the target level specification of the computer. Finally, we obtain the correctness statement on page 36 simply by rewriting with the definition of COMPUTER_abs and discharging the premise set. This concludes our verification of Gordon's computer.

9 Discussion

Abstraction is a fundamental technique for managing complexity. In computer science, examples of abstraction abound especially in the realm of software. As software solutions are constantly adapted to hardware problems, abstraction will be an increasingly important technique for managing the complexity of computer hardware.

Perfection is reached not when there is no longer anything to add, but when there is no longer anything to take away. [A. Saint-Exupery]

The concept of information hiding underlies abstraction. For instance, an abstract data type presents a view of a specialized data object which allows the data object to be fully manipulated by a programmer but hides details concerning the implementation of the data object. Similarly, the target level description of Gordon's computer in terms of the eight instructions, HALT, JMP, JZR, ADD, SUB, LOAD, STORE, and SKIP and the use of the button, knob, and switches on the front panel of the computer constitutes an abstract view of the register-transfer level behaviour of the device. This abstraction hides, for example, the implementation detail that the ADD instruction results in the execution of ten microcode instructions. Aside from real-time applications, in which case this abstraction is not appropriate, it should matter little how many microcode instructions are executed for any target level instruction. Nevertheless, the success of the abstraction does not depend merely on the volume of detail hidden in the abstraction. A successful abstraction presents a unified view without 'gaps'. For example, if the SKIP instruction was left out of the target level description of Gordon's computer, then what happens when opcode 7 is encountered would be a gap in the abstraction. If successful, the abstraction, as explanation of the behaviour of the machine, should stand alone from register-transfer level details.

The first step in verifying Gordon's computer results in the correctness statement shown on page 18. This step is based on a structural abstraction where the behaviour of a device is derived from the behaviour of its components. This is the most common type of abstraction in hardware verification. A second type of abstraction, temporal abstraction, attempts to recast the behaviour of a device in terms of a coarser grain of time. The second part of the verification procedure we have described for Gordon's computer is based on a temporal abstraction. A third type of abstraction, data abstraction, is not used in the computer example. An example of data abstraction would be to prove the correctness of a register-transfer implementation of a Lisp machine where the correctness statement makes reference to (potentially infinite) data objects such as lists.

An obvious motivation for abstraction is to present a simplified, but nevertheless useful view of something complex. An instructor would never (hopefully) introduce assembler language programming to first year students with an explanation of the microcode behaviour of the machine. For similar reasons of clarity, a statement of correctness for hardware designers should be as simple as possible while still capturing the essential behaviour of the device.

A less obvious motivation for hardware abstractions is that abstraction forms the basis of hierarchical design. An hierarchical design is simply a series of nested abstractions. For example, Gordon's computer might be incorporated as a microprocessor in a larger system. In verifying the correctness of the larger system, the correctness statement on page 36 would serve as a specification of the microprocessor as a primitive component in the larger system.

Yet another motivation for abstraction is that the failure to establish a desired abstraction may point out flaws in the design. In fact, the verification of Gordon's computer is a example of this. Earlier we observed that the correctness statement on page 36 requires assumptions about the stability of the knob and switches signals. Ideally, we desire a correctness statement which does not refer to events between points on the *instruction set time scale*. However, these assumptions about the stability of knob and switches do make reference to events between points on the *instruction set time scale*. Therefore, the temporal abstraction we have presented in this paper is in fact a failure. The abstraction fails because we need to take into account events at the *microcode time scale* to make sense of the correctness statement.

Nevertheless, the origin of the failure lies not in the abstraction but in the design of the computer itself. Suppose, as we have already suggested, that Gordon's computer is used as a component in a larger system. Suppose further, that this larger system is clocked by the ready signal of the computer, that is, the system operates at the *instruction set time scale.* In particular, suppose that the signals knob_abs and switches_abs are clocked by ready. This means that these signals are not necessarily stable with respect to the *microcode time scale* between points on the *instruction set time scale.* Hence, the computer could fail to behave as expected. Fortunately, verification brings to light the possibility of this failure.

10 Summary

The entire specification and verification of Gordon's computer consists of 3000 lines of documented source text. The verification procedure requires approximately six hours of running time on a lightly loaded, 14 megabyte VAX 11/780. The proof required about two man-months of effort which included learning how to specify and verify hardware in higher order logic and how to use the HOL proof assistant system.

The main overhead of producing the initial specification and writing the ML programs to do the necessary inferences need only be done once. Thereafter, the finished proof could be easily modified to accommodate small changes in the hardware. For example, two registers could be added to latch the values of the knob and switches signals whenever the ready signal is true. This would eliminate the need for assumptions about the stability of these signals. The correctness of incremental changes to a design can be verified by editing the specification and verification procedure, usually in a very minor way, and re-running the proof as a batch job.

11 Epilogue

Since the original publication of this report an 8-bit version of Gordon's computer has been implemented as a 5,000 transistor CMOS microchip. The purpose of this exercise was to study the role of formal specification in the design process; this experience is reported in [5].

One of the most interesting discoveries of this exercise was that the formal verification of the design missed a design error: there is no reset button to initialize the microcode program counter (this is different from the function of the interrupt button).

The HOL version of the computer example improves upon the original LCF_LSM version by ensuring that for any initial state of the microcode program counter, even an address into the 'unused' part of the microcode, the computer will eventually reach the start of an execution cycle, ie. mpc = #00000 or mpc = #00101. However, we incorrectly assumed that the mpc would power up to a proper state and not some undefined value.

After the design was submitted for fabrication and long after its formal verification, we decided to simulate the design using a switch-level simulator involving a more accurate model of a signal value [6]. The design failed to simulate properly because the initial state of the mpc signal was undefined and there was no way to make it become defined. Fortunately, when the actual chips were returned and tested we found that the mpc signal tended to #00000 due to electrical factors not modelled at the switch-level and the chip worked correctly.

Why did formal verification fail to discover this design error ? Clearly, the actual verification process, ie. formal deduction, was correct; instead, the source of the problem was the incorrect assumption that the mpc signal was bi-stable. This assumption was introduced by our behavioural specification of the MPC as a primitive component.

Bi-stability is an abstract view of more complex forms of data in digital circuits, eg. tri-state data, voltage values, etc. This abstraction is valid only under certain conditions. For instance, we could prove that the output of a register such as the MPC is bi-stable provided that the input to the register is bi-stable. In the formal specification of the computer we simply assumed that the output of the MPC was bi-stable. If instead we had been required to establish that the output of the MPC was bi-stable by showing that the input to the MPC was bi-stable (following a reset), then the need for a reset button would have inevitability been discovered.

More generally, verification failed here because our register-transfer level primitives were not adequate models for reliable verification. Behaviours at the register-transfer level should be derived from some lower level model such as a switch-level model. Behavioural models of register-transfer level components derived from a lower level model would clearly state the conditions under which the behaviour is valid. A library of register-transfer component behaviours and their respective validity conditions would serve as a basis for a hardware verification methodology for VLSI design.

References

[1] Gordon, M., "LCF_LSM, A System for Specifying and Verifying Hardware", Technical Report No. 41, Computer Laboratory, The University of Cambridge, September 1983.

[2] Gordon, M., "Proving a Computer Correct using the LCF_LSM Hardware Verification System", Technical Report No. 42, Computer Laboratory, The University of Cambridge, September 1983.

[3] Gordon, M., "HOL: A Machine Oriented Formulation of Higher Order Logic", Technical Report No. 68, Computer Laboratory, The University of Cambridge, July 1985.

[4] Melham, T., "Abstraction in Hardware Verification", Progress Report and Thesis Proposal, Computer Laboratory, The University of Cambridge, October 1985.

[5] Joyce, J., "Formal Verification and Implementation of a Microprocessor", Hardware Verification Workshop Proceedings, The University of Calgary, January 1987.

[6] Bryant, R., "An Algorithm for MOS Logic Simulation", Lambda Magazine, Fourth Quarter, 1980.

12 Appendix A

HOL Constants for 'Built-in' Operations

The constants EL, SEG, V, VAL<n>, WORD<n> and BITS<n> are built into the HOL system. Each of these constants names an operation whose type is shown below along with a description of the operation.

EL	:num→* list→*	nth element of a list
SEG	:num#num→* list→* list	sublist of a list
V	:bool list→num	number denoted by a bit list
VAL	:word<n>→num	number denoted by a word
WORD<n>	:num→word<n>	word representing a number
BITS<n>	:word<n>→bool list	list of bits in a WORD

For each of these constants, there is a built-in evaluation rule which evaluates applications of the constant to primitive data object. These rules are implemented as ML routines of type *:thm→thm*. For example, applying VAL_RULE to a theorem containing an occurrence of `VAL2 #11` would result in another theorem with `VAL2 #11` replaced by 3. In addition to VAL_RULE, these rules are EL_RULE, SEG_RULE, WORD_RULE and BITS_RULE.

13 Appendix B

Expanded Version of COMPUTER_IMP

The following is the result of expanding the definitions of COMPUTER_IMP with the definitions of CONTROL and DATA and the definitions of primitive components. To retain a measure of readability, the signal bus has not been hidden and the expansion of CNTL_BIT, CNTL_FIELD, B_ADDR, A_ADDR, and TEST has been suppressed.

This theorem captures the entire behaviour of the register-transfer level implementation.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf,bus)
 (memory,knob,button,switches,pc,acc,idle,ready) =
|- (!t.
     buf(t + 1) =
     ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE(mpc t))) = 0) =>
      bus t |
      ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE(mpc t))) = 1) =>
       INC16(bus t) |
       ((VAL2(CNTL_FIELD(16,17)(FETCH5 MICROCODE(mpc t))) = 2) =>
        ADD16(arg t)(bus t) |
        SUB16(arg t)(bus t))))) /\
   (!t.
     memory(t + 1) =
     ((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE(mpc t))) = 2) =>
      STORE13(mar t)(bus t)(memory t) |
      memory t)) /\
   (!t.
     mar(t + 1) =
     (CNTL_BIT 27(FETCH5 MICROCODE(mpc t)) =>
      CUT16_13(bus t) | mar t)) /\
   (!t.
     pc(t + 1) =
     (CNTL_BIT 24(FETCH5 MICROCODE(mpc t)) =>
      CUT16_13(bus t) | pc t)) /\
   (!t.
     acc(t + 1) =
     (CNTL_BIT 22(FETCH5 MICROCODE(mpc t)) => bus t | acc t)) /\
   (!t.
     ir(t + 1) = (CNTL_BIT 20(FETCH5 MICROCODE(mpc t)) =>
      bus t | ir t)) /\
(!t.
  arg(t + 1) =
  (CNTL_BIT 18(FETCH5 MICROCODE(mpc t)) => bus t | arg t)) /\
(!t.
  bus t =
  DEST_TRI16
  (((VAL2(CNTL_FIELD(25,26)(FETCH5 MICROCODE(mpc t))) = 1) =>
    MK_TRI16(FETCH13(memory t)(mar t)) |
    FLOAT16) U16
   ((CNTL_BIT 28(FETCH5 MICROCODE(mpc t)) =>
     MK_TRI16(switches t) |
     FLOAT16) U16
    ((CNTL_BIT 23(FETCH5 MICROCODE(mpc t)) =>
      MK_TRI16(PAD13_16(pc t)) |
      FLOAT16) U16
     ((CNTL_BIT 21(FETCH5 MICROCODE(mpc t)) =>
       MK_TRI16(acc t) |
       FLOAT16) U16
      ((CNTL_BIT 19(FETCH5 MICROCODE(mpc t)) =>
        MK_TRI16(ir t) |
        FLOAT16) U16
       (CNTL_BIT 15(FETCH5 MICROCODE(mpc t)) =>
        MK_TRI16(buf t) |
        FLOAT16)))))) /\
(!t.
  mpc(t + 1) =
  (((TEST(FETCH5 MICROCODE(mpc t)) = 1) /\ button t) =>
   B_ADDR(FETCH5 MICROCODE(mpc t)) |
   (((TEST(FETCH5 MICROCODE(mpc t)) = 2) /\
      (VAL16(acc t) = 0)) =>
    B_ADDR(FETCH5 MICROCODE(mpc t)) |
    ((TEST(FETCH5 MICROCODE(mpc t)) = 3) =>
     WORD5((VAL2(knob t)) +
      (VAL5(A_ADDR(FETCH5 MICROCODE(mpc t))))) |
     ((TEST(FETCH5 MICROCODE(mpc t)) = 4) =>
      WORD5
      ((VAL3(OPCODE(ir t))) +
       (VAL5(A_ADDR(FETCH5 MICROCODE(mpc t))))) |
      A_ADDR(FETCH5 MICROCODE(mpc t))))))) /\
(!t. ready t = CNTL_BIT 14(FETCH5 MICROCODE(mpc t))) /\
(!t. idle t = CNTL_BIT 13(FETCH5 MICROCODE(mpc t)))
```

14 Appendix C

Simulation of the Execution Path for ADD

The following is the sequence of machine states in the simulation of the execution path for the target level instruction ADD. Each state is represented by eight theorems. Each theorem describes the current value of one of the eight output signals of the implementation. Theorems for the signals idle and ready are excluded because the value of these signals can be obtained directly from the current value of the microcode program counter. Note that the "...." before theoremhood symbol are the hypotheses shown below.

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
mpc t1 = #00101,
button t1 = F,
VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 3
```

Beginning with the state at time (t1 + 1), the following ten states occur. The final state occurs at ((((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1).

```
State 1

 .... |- buf(t1+1) = DEST_TRI16 FLOAT16
 .... |- memory(t1+1) = memory t1
 .... |- mar(t1+1) = mar t1
 .... |- pc(t1+1) = pc t1
 .... |- acc(t1+1) = acc t1
 .... |- ir(t1+1) = ir t1
 .... |- arg(t1+1) = arg t1
 .... |- mpc(t1+1) = #00110

State 2

 .... |- buf((t1+1)+1) = PAD13_16(pc t1)
 .... |- memory((t1+1)+1) = memory t1
 .... |- mar((t1+1)+1) = pc t1
 .... |- pc((t1+1)+1) = pc t1
 .... |- acc((t1+1)+1) = acc t1
 .... |- ir((t1+1)+1) = ir t1
 .... |- arg((t1+1)+1) = arg t1
```

```
.... |- mpc((t1+1)+1) = #01000
```

State 3

```
.... |- buf(((t1+1)+1)+1) = FETCH13(memory t1)(pc t1)
.... |- memory(((t1+1)+1)+1) = memory t1
.... |- mar(((t1+1)+1)+1) = pc t1
.... |- pc(((t1+1)+1)+1) = pc t1
.... |- acc(((t1+1)+1)+1) = acc t1
.... |- ir(((t1+1)+1)+1) = FETCH13(memory t1)(pc t1)
.... |- arg(((t1+1)+1)+1) = arg t1
.... |- mpc(((t1+1)+1)+1) = #01001
```

State 4

```
.... |- buf((((t1+1)+1)+1)+1) = DEST_TRI16 FLOAT16
.... |- memory((((t1+1)+1)+1)+1) = memory t1
.... |- mar((((t1+1)+1)+1)+1) = pc t1
.... |- pc((((t1+1)+1)+1)+1) = pc t1
.... |- acc((((t1+1)+1)+1)+1) = acc t1
.... |- ir((((t1+1)+1)+1)+1) = FETCH13(memory t1)(pc t1)
.... |- arg((((t1+1)+1)+1)+1) = arg t1
.... |- mpc((((t1+1)+1)+1)+1) = #01101
```

State 5

```
.... |- buf(((((t1+1)+1)+1)+1)+1) = acc t1
.... |- memory(((((t1+1)+1)+1)+1)+1) = memory t1
.... |- mar(((((t1+1)+1)+1)+1)+1) = pc t1
.... |- pc(((((t1+1)+1)+1)+1)+1) = pc t1
.... |- acc(((((t1+1)+1)+1)+1)+1) = acc t1
.... |- ir(((((t1+1)+1)+1)+1)+1) = FETCH13(memory t1)(pc t1)
.... |- arg(((((t1+1)+1)+1)+1)+1) = acc t1
.... |- mpc(((((t1+1)+1)+1)+1)+1) = #10011
```

State 6

```
.... |- buf((((((t1+1)+1)+1)+1)+1)+1) = FETCH13(memory t1)(pc t1)
.... |- memory((((((t1+1)+1)+1)+1)+1)+1) = memory t1
.... |- mar((((((t1+1)+1)+1)+1)+1)+1) =
        CUT16_13(FETCH13(memory t1)(pc t1))
.... |- pc((((((t1+1)+1)+1)+1)+1)+1) = pc t1
.... |- acc((((((t1+1)+1)+1)+1)+1)+1) = acc t1
```

```
 .... |- ir((((((t1+1)+1)+1)+1)+1)+1) = FETCH13(memory t1)(pc t1)
 .... |- arg((((((t1+1)+1)+1)+1)+1)+1) = acc t1
 .... |- mpc((((((t1+1)+1)+1)+1)+1)+1) = #10100

State 7

 .... |- buf(((((((t1+1)+1)+1)+1)+1)+1)+1) =
        ADD16
         (acc t1)
         (FETCH13
          (memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))))
 .... |- memory(((((((t1+1)+1)+1)+1)+1)+1)+1) = memory t1
 .... |- mar(((((((t1+1)+1)+1)+1)+1)+1)+1) =
        CUT16_13(FETCH13(memory t1)(pc t1))
 .... |- pc(((((((t1+1)+1)+1)+1)+1)+1)+1) = pc t1
 .... |- acc(((((((t1+1)+1)+1)+1)+1)+1)+1) = acc t1
 .... |- ir(((((((t1+1)+1)+1)+1)+1)+1)+1) =
        FETCH13(memory t1)(pc t1)
 .... |- arg(((((((t1+1)+1)+1)+1)+1)+1)+1) = acc t1
 .... |- mpc(((((((t1+1)+1)+1)+1)+1)+1)+1) = #10101

State 8

 .... |- buf((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) =
        ADD16
         (acc t1)
         (FETCH13
          (memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))))
 .... |- memory((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) = memory t1
 .... |- mar((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) =
        CUT16_13(FETCH13(memory t1)(pc t1))
 .... |- pc((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) = pc t1
 .... |- acc((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) =
        ADD16
         (acc t1)
         (FETCH13
          (memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))))
 .... |- ir((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) =
        FETCH13(memory t1)(pc t1)
 .... |- arg((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) = acc t1
 .... |- mpc((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) = #10001

State 9
```

```
 .... |- buf((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) =
          INC16(PAD13_16(pc t1))
 .... |- memory((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) = memory t1
 .... |- mar((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) =
          CUT16_13(FETCH13(memory t1)(pc t1))
 .... |- pc((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) = pc t1
 .... |- acc((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) =
          ADD16
           (acc t1)
           (FETCH13
            (memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))))
 .... |- ir((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) =
          FETCH13(memory t1)(pc t1)
 .... |- arg((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) = acc t1
 .... |- mpc((((((((t1+1)+1)+1)+1)+1)+1)+1)+1) = #10010

State 10

 .... |- buf(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          INC16(PAD13_16(pc t1))
 .... |- memory(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          memory t1
 .... |- mar(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          CUT16_13(FETCH13(memory t1)(pc t1))
 .... |- pc(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          INC13(pc t1)
 .... |- acc(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          ADD16
           (acc t1)
           (FETCH13
            (memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))))
 .... |- ir(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) =
          FETCH13(memory t1)(pc t1)
 .... |- arg(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) = acc t1
 .... |- mpc(((((((((t1+1)+1)+1)+1)+1)+1)+1)+1)+1) = #00101
```

15 Appendix D

Theorems for the Fifteen Possible Execution Paths

The following theorems state, for each of the fifteen possible execution paths, the behaviour of the implementation expressed by the correctness statement on page 18. The correctness statement is proved by a case analysis based on these fifteen theorems.

```
Case:  idling, button pressed, knob = 0

    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    idle t1,
    ready t1 ,
    button t1 = T,
    VAL2 (knob t1) = 0
    |- memory t2,pc t2,acc t2,idle t2
        = memory t1,CUT16_13(switches t1),acc t1,T


Case:  idling, button pressed, knob = 1

    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    idle t1,
    ready t1,
    button t1 = T,
    VAL2 (knob t1) = 1
    |- memory t2,pc t2,acc t2,idle t2
        = memory t1,pc t1,switches t1,T


Case:  idling, button pressed, knob = 2
```

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
STABLE (t1,t2) knob,
STABLE (t1,t2) switches,
NEXT (t1,t2) ready,
idle t1,
ready t1,
button t1 = T,
VAL2 (knob t1) = 2
|- memory t2,pc t2,acc t2,idle t2
    = STORE13(pc t1)(acc t1)(memory t1),pc t1,acc t1,T
```

Case: idling, button pressed, knob = 3

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
STABLE (t1,t2) knob,
STABLE (t1,t2) switches,
NEXT (t1,t2) ready,
idle t1,
ready t1,
button t1 = T,
VAL2 (knob t1) = 3
|- memory t2,pc t2,acc t2,idle t2
    = memory t1,pc t1,acc t1,F
```

Case: idling, button not pressed

```
COMPUTER_IMP
 (mpc,mar,ir,arg,buf)
 (memory,knob,button,switches,pc,acc,idle,ready),
STABLE (t1,t2) knob,
STABLE (t1,t2) switches,
NEXT (t1,t2) ready,
idle t1,
ready t1,
button t1 = F
|- memory t2,pc t2,acc t2,idle t2
    = memory t1,pc t1,acc t1,T
```

Case: running, button pressed

```
    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    ~idle t1,
    ready t1,
    button t1 = T,
    NEXT(t1,t2)ready, button t1 = T,
    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    ~idle t1, ready t1
    |- memory t2,pc t2,acc t2,idle t2
        = memory t1,pc t1,acc t1,T
```

Case: running, button not pressed, opcode = 0

```
    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    ~idle t1,
    ready t1,
    button t1 = F,
    VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 0
    |- memory t2,pc t2,acc t2,idle t2
        = memory t1,pc t1,acc t1,T
```

Case: running, button not pressed, opcode = 1

```
    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    ~idle t1,
    ready t1,
    button t1 = F,
    VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 1
    |- memory t2,pc t2,acc t2,idle t2
        = memory t1,CUT16_13(FETCH13(memory t1)(pc t1)),acc t1,F
```

```
Case:  running, button not pressed, opcode = 2, ACC = 0

    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    ~idle t1,
    ready t1,
    button t1 = F,
    VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 2,
    (VAL16(acc t1) = 0) = T
    |- memory t2,pc t2,acc t2,idle t2
        = memory t1,CUT16_13(FETCH13(memory t1)(pc t1)),acc t1,F
```

```
Case:  running, button not pressed, opcode = 2, not ACC = 0

    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    ~idle t1,
    ready t1,
    button t1 = F,
    VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 2
    (VAL16(acc t1) = 0) = F
    |- memory t2,pc t2,acc t2,idle t2
        = memory t1,INC13(pc t1),acc t1,F
```

Case: running, button not pressed, opcode = 3

```
    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    ~idle t1,
    ready t1,
    button t1 = F,
    VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 3
    |- memory t2,pc t2,acc t2,idle t2
        = memory t1,INC13(pc t1),
           ADD16
            (acc t1)
            (FETCH13
             (memory t1)
             (CUT16_13(FETCH13(memory t1)(pc t1)))),F
```

Case: running, button not pressed, opcode = 4

```
    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    ~idle t1,
    ready t1,
    button t1 = F,
    VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 4
    |- memory t2,pc t2,acc t2,idle t2 =
        memory t1,INC13(pc t1),
         SUB16
          (acc t1)
          (FETCH13
           (memory t1)
           (CUT16_13(FETCH13(memory t1)(pc t1)))),F
```

Case: running, button not pressed, opcode = 5

```
    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    ~idle t1,
    ready t1,
    button t1 = F,
    VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 5
    |- memory t2,pc t2,acc t2,idle t2 =
        memory t1,INC13(pc t1),
         FETCH13(memory t1)(CUT16_13(FETCH13(memory t1)(pc t1))),F
```

Case: running, button not pressed, opcode = 6

```
    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
    NEXT (t1,t2) ready,
    ~idle t1,
    ready t1,
    button t1 = F,
    VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 6
    |- memory t2,pc t2,acc t2,idle t2 =
        STORE13
         (CUT16_13(FETCH13(memory t1)(pc t1)))(acc t1)(memory t1),
         INC13(pc t1),acc t1,F
```

Case: running, button not pressed, opcode = 7

```
    COMPUTER_IMP
     (mpc,mar,ir,arg,buf)
     (memory,knob,button,switches,pc,acc,idle,ready),
    STABLE (t1,t2) knob,
    STABLE (t1,t2) switches,
NEXT (t1,t2) ready,
~idle t1,
ready t1,
button t1 = F,
VAL3(OPCODE(FETCH13(memory t1)(pc t1))) = 7
|- memory t2,pc t2,acc t2,idle t2
    = memory t1,INC13(pc t1),acc t1,F
```