Technical Report

UCAM-CL-TR-40
ISSN 1476-2986

Number 40

The SKIM microprogrammer's guide

W. Stoye

October 1983

ISSN 1476-2986

Contents

This document is divided into sections, as follows:

SKIM - Physical Organisation

```
=======================SKIM==rack============================
   16 bit ad, 40 bit data          10 bit ad, 24 bit data
     +---------------+  (backplane)  +---------------+-. . .
     |               |               |               |
|-----+------| |-----+------| |------------| |------------|
|            | |            | |            | |            |
| micro code | |            | |    RAM     | |    RAM     |
|    RAM     | |    CPU     | | controller | |   array    |
|    card    | |    card    | |    card    | |    card    |
|            | |            | |            | |            |
|------------| |------------| |------------| |------------|
                   |     |               |
                   |     +-------------+24 bit ad/data path
                   |                    (ribbon cable)
==================|=========================================
                   |
                   |8 bit bidirectional port
                   |debug and control lines
                   |(ribbon cable)
             |-----+------|
 screen <----|            |<----> local discs
             |   host     |
keyboard--->| processor  |<----> terminal line to Phoenix/MVT
             |            |
             |------------|
```

The prototype SKIM has been constructed on four wirewrapped extended double Eurocards. Up to three more RAM array cards could be added.

All I/O from the processor is done via a host computer system, currently a BBC microcomputer system. The host and SKIM together form a stand-alone working system. There is also a link to the local mainframe, where most of the system software is being developed.

TTL logic, mostly MSI and SSI, has been used throughout. The only non-TTL devices are memory chips. All four boards contain a total of 230 chips, of which 92 are memory chips.

Main Design Features

The CPU Card

24 bit ALU, based on 6 AMD2901 4-bit TTL slices

processor operation can be monitored, halted and
  single stepped under the control of the host

the host can usefully assist in the debugging
  and checking of hardware and microcode.

tagged architecture:
  24 bit word consists of 4 tag, 20 data
  carry from data does not overflow into tag in the ALU
  hardware assist for rapid testing of tag values

The Microcode Memory Card

40 bit wide microcode with one-word pipeline

250ns micro cycle time

microcode in RAM, under control of host

very large microcode store, up to 64K words

The Main Memory Controller Card

memory arranged in nodes
each node contains two 24 bit words and two flag bits

hardware assist for marking and garbage collection

parity checking on main memory

350 ns main memory cycle time - transparent refresh

read/modify/write memory cycles allow fast
  pointer operations

can control up to four memory array cards

The Main Memory Array Card

64K nodes of 54 64K ram chips on each card
can be upgraded to use 256K rams

total maximum memory 1M nodes (6Mbytes) on 4 cards

Operation

The CPU is based on AMD2901 4-bit ALU slices. It has 16 internal general-purpose registers, and provides simple arithmetic operations. Multiplication and division are specially assisted by conditional 'shift and add/subtract' instructions. It is contructed so that tag and data calculations are separate.

The CPU is controlled by 40-bit wide microcode which allows branch calculations, tag comparisons, ALU arithmetic, microcode instruction fetch and main memory access all to proceed concurrently. The microcode is controlled by a 16 bit microsequencer with its own subroutine stack. The CPU obeys one microcode cycle every 250ns.

The main memory is organised as a sequence of nodes, each of which contains two machine words of 24 bits and two flag bits, which can be written to independently. This allows a marking garbage collection to proceed at high speed, and means that the machine memory is viewed by the programmer as a heap of binary nodes. Read/modify/write memory cycles allow common operations such as pointer chain following and reversal to proceed at maximum memory bandwidth, while the tagged processor design means that the pointers can be checked concurrently as bona fide memory pointer values. This means that many tree and list processing operations can proceed at much higher speed than on a conventional architecture of similar bandwidth.

Software Development

SKIM is envisaged as a soft machine, running an interpreter written in microcode for a list/tree processing system.

Large Microcode (Writable Control) Store

Provision has been made for connecting a very large amount of microcode memory to SKIM, which would allow microcode to be used like the machine code of a conventional processor. Operating system primitives and common inner loops could be coded in microcode, or a compiler written that generates microcode. The changing relative cost of fast memory and CPU logic is rapidly making this the most cost-effective method of making the processor go quickly. Using 8K*8 static RAMs, 64K words of microcode memory can be placed on one card.

Tagged Architecture

SKIM's hardware provides support for calculations involving tagged machine words. Special microcode instructions allow the testing of tag values concurrently with other calculations in the CPU, so that run-time type checking will not have an associated performance penalty. This means that garbage collection of main memory can be performed by microcode at high speed. Also, the difference between integer data values represented by a single machine word and bigger, multiprecision values represented by a list of memory nodes can be transparent to the programmer, so that worries over 'bitswidth' disappear.

Debugging under Host Control

The prototype SKIM has been designed as a development system, and every effort has been made to ensure that bugs will show themselves. The host processor has complete control over the operation of the CPU, and can halt the system clock (the main memory continues to refresh itself) and single step the processor at low speed under operator control.

The host can write values into the microcode memory (eg for breakpoints and microcode debugging) and by inserting microcode values directly into the microcode pipeline can read the CPU registers without disturbing the progress of a calculation. Indeed the whole machine state (including main memory) can be non-destructively examined under control of the host. With these facilities development of microcode programs is no more difficult than that of machine code programs for a conventional machine.

Comparison With Other Machines

There are several hardware processors in existence that were designed specifically for list processing. Examples include the MIT Lisp Machine [Symbolcs 82] and FLATS [FLATS 79]. Such machines are mostly aimed at the efficient implementation of Lisp.

SKIM has been microcoded for combinator reduction [Turner 78], and it reduces about 80,000 combinators per second. Its flexibility should also make it ideal for supercombinators [Hughes 82] and for Prolog and Lisp. It is considerably smaller and simpler than most soft machines available today, and this should be an advantage in experimenting with new implementation techniques.

The SKIM prototype has not been designed with the most recent technology. The CPU is in fact very simple, a brief calculation of the number of circuits on it indicates that it is considerably less complex than a Motorola 68000 [68000 82]. It would fit very well on a single chip, and this would lead to a faster, cheaper machine but this is not sensible until more work has been done with software experimentation and development.

Limitations

No Virtual Memory or Address Translation

The maximum addressing range is for 1M nodes of RAM. It should soon be economically feasable to connect this much physical RAM (ie 6M bytes), which should be enough for fairly large applications.

In order to make use of virtual memory a larger virtual address space would be necessary, which would involve windening all data paths and would generally lead to a larger machine. Also, address translation would slow main memory access down so the processor cycle time would have to be increased, or more advanced and complex logic used in constructing the processor.

No multiply, divide, shift or floating point hardware

It would be possible to add these if they were felt to be necessary, but the processor would then occupy more than one board and this was felt to be inappropriate in the prototype. Before any of these a fast hardware stack would probably be a more productive addition to the CPU. 20 by 20 bit unsigned integer multiplication and division both occur in 20 micrcocycles, ie 5 microseconds.

No memory cache

The main memory cycle time of 350ns is fairly quick and considering the overall size of the machine (four boards of circuitry), the amount of effort required to implement a memory cache was not considered worth while. Also, the main memory bandwidth is no more a bottleneck than the microcode memory bandwidth, and no great benefit would result from the increasing the bandwidth available on either one of these.

No parity checking on CPU data paths

This would be very difficult on an MSI implementation. If a chip were being made it would be very much easier. Great emphasis has been placed on the reduction of noise in the hardware design, in an effort to make a suitable, reliable tool for software research. At the time of writing the prototype has shown no signs of noise related problems.

All of the above features could be added to a future version of SKIM, and they would lead to a more powerful machine. But it would also be correspondingly more complex and expensive.

Introduction.

This document is intended as a guide for the writer of microcode for SKIM. It does not describe the hardware in any detail. SKIM has been microprogrammed to reduce combinators [Turner 78], but this should not be its only application and will not be covered here.

General knowledge of machine code programming concepts are assumed, and the reader is advised to read the SKIM hardware description first. SKIM's name is derived from its original application area ("S, K, I Machine" - S, K and I being the three most important combinators).

SKIM - Simple Block Diagram.

```
                                                to host  ^
    -----                                                |
 micro  |--------------------(control)----+              |
 instr  |----------(control)----+         |              V
pipeline|                       |         |          ---------
    |   |----+-(16)+            V         |          | PORT  |
    |   |    |     |       -----------    V          ---------
    -----    |   . V       | machine |  ----------         ^
      ^     .|  --------   |   regs  |->| ALU     |        |
     (40)    |  | ireg |   | 16*24   |->|         |        |
      |      |  --------   -----------  ----------      (8)
---------    |    ^             ^           ^            |
| micro |    |    |             |           |            |
| code  |    |    V             V           V            V
| ram.  |    |  =============24=bit=main=bus==============
|       |    |   ^          ^        |         ^         ^
| 2K*40 |    |   |          |        |         |         |
---------    V   V          V        V         |         |
    ^      --------    ---------  ---------  ---------  ---------
    +-(16)-| MPC  |    | MAR   |  | MIR   |  |HDOUT  |  |TLOUT  |
           --------    ---------  ---------  ---------  ---------
              ^           |         |          ^          ^
              |           |         |          |          |
              V           V addr    V din      | dout     | dout
           --------    ---------------------------------------
       16*4| subr |    |    Main   Memory.                   |
           |stack |    |        64K*50                       |
           --------    ---------------------------------------
```

The ALU.

SKIM has a CPU based around 6 AMD2901 ALU chips. Each is a 4-bit slice of an ALU, and they have been carefully wired to support SKIM's tagged architecture.

```
---------------------------------------------
| 4 bit |        20 bit value              |
|  tag  | ms                             ls|
---------------------------------------------
23     20 19                              0
```

Bits in SKIM are labelled as 0 for least significant, 23 for most significant. Bits 0 to 19 of the ALU are wired in a conventional manner, to allow add, subtract, shift and so on to proceed as usual. Bits 20 to 23 form a 'tag'. Carry from the data field does not overflow into the tag field. Tag values are usually just transferred rather that added or combined in some way. The tag specifies what kind of value this is. By convention, a tag value of 0..7 indicates that this is a memory address of some kind. A tag value of 8..15 means that it is some other value. A tag value of 15 means that the value is a microcode address.

ALU Registers.

SKIM has 16 machine registers (called 'regs', or r0 to r15) for holding intermediate values during computation. These all behave in a similar manner, ie there are no special operations that only work on r0. There is also a special register in the ALU (called Q) which is used during multiplys and divides only. Although these operations are not done in a single microcode cycle, special instructions have been provided to ensure that they can be executed fairly quickly.

The ALU also stores the carry bit from the previous cycle. This is very useful in some multi-stage computations.

Other Registers.

The CPU also has various registers that are external to the ALU, called 'xregs' - all have specialised and different names. they are as follows.

The <u>Memory Address Register</u> or <u>MAR</u> holds the address of the main memory cell currently being accessed. In order to access a cell of main memory the address of the cell must first be written to this register. This register can also be read, as it was found that this helps greatly in certain critical loops.

There is a convention when programming SKIM that values with a tag of 7 or less are memory addresses, and values with a tag of more than 7 are not. This makes garbage collection much simpler, because the garbage collection can decide which values are memory pointer.

This convention is enforced in the hardware of the MAR. If the value in the MAR is more than 7 then any memory access instructions are ignored. This saves a cycle in some programs, as checking for a value greater than 7 can happen concurrently with the initiation of a memory cycle.

The Memory Input Register or MIR. This holds the value to be written into memory. A value to be written into main memory must first be written into this register.

The Head Output Register, or HDOUT. Each cell of the main memory has two halves, 'head' and 'tail'. A read from memory will cause the head value in the cell being accessed to be latched into this register, which can then be read by the microcode program into an ALU register. A write to memory will result in the previous content of the head half of the cell written to to be latched into this register.

The Tail Output Register or TLOUT. Analogous to HDOUT, this latches the tail half of any cell accessed. Thus note that any access to memory reads both halves of the node.

Associated with HDOUT and TLOUT are two more one-bit registers, that latch the output flags from the memory word whenever HDOUT and TLOUT are updated. These cannot be accessed directly by the programmer, but are accessed by various conditional branches.

The PORT. This register is the only means that SKIM has of talking to the outside world. Peripherals, discs etc are controlled for SKIM by a host processor. The host can send input to SKIM by placing a character in this register. SKIM can output a character by writing to this register.

A read from the PORT register will only yield a significant result in the bottom eight bits of the databus. The top sixteen bits of the result will be garbage and should be masked out.

The Microcode Program Counter or MPC. This register is used to access the microcode memory. It contains the address of the next microcode instruction to be executed. The microcode programmer can read from it, in order to get a return address for a subroutine. In order to write to it a 'force jump' (q.v.) is necessary.

The Immediate Data Field or I. This register provides immediate data values. It is derived from a 16 bit field in the microcode word, which is used for three different purposes:

16 bit target of a JMP or JSR
12 bit target and 4 bit condition code of a branch
12 bit data and 4 bit tag of an immediate data value

Thus immeditate data loads cannot occur in the same microinstruction as a jump, subroutine jump or branch. Also, the immediate data value is only derived from 16 bits, so bits 12 to 19 of the I field will always be the same as bit 11 of the I field (ie Only a 12 bit value can be provided, which is sign extended to form a 20 bit value). Any tag value can be specified.

Timing.

One microcode cycle on SKIM takes 250ns. A main memory cycle takes 350ns, so a microcode cycle in which a main memory cycle occurs is stretched by 100ns. Small extra delays may also occur for main memory refresh cycles, but these are not visible to the programmer.

Main Memory.

SKIM has a 20 bit address bus, and thus can address 1M memory words. the terms 'cell' or 'node' are sometimes used instead of word. Each cell consists of 50 bits:

```
------------------------------  ------------------------------
|f|4bit|  20 bit data        |  |f|4bit|      20 bit data     |
|l|tag |                     |  |l|tag |                      |.
-a----------------------------  -a----------------------------
 g          HEAD                 g           TAIL
```

24 bits of 'head' or 'left' field - 4 tag, 20 data
24 bits of 'tail' or 'right'field - 4 tag, 20 data
1 head flag
1 tail flag.

Any microcode instruction can be used to initiate a memory access. The following types are available.

RDHD	read head field
XHD	write to head field, reading previous content.
XHDF0	same, but in the process, also write a 0 to the head flag.
XHDF1	same, but writing a 1 to the head flag.
RDHDF0	read from the head field, in the process write a 0 to the head flag.
RDHDF1	same, writing a 1.

The same operations are all available on the tail half, substitute TL for HD in all the above.

In a cycle in which any of these is specified the values in the MAR and possibly the MIR are significant. These values can be set up in the same cycle as memory cycle initialisation. On the next microcode cycle new values will be available in the HDOUT and TLOUT registers that reflect the values of the memory node accessed. So the memory cycle happens (conceptually) <u>between</u> two microcode cycles, although in practice the process is pipelined and no time is lost.

Note that on all memory cycles, the whole cell is read and values appear in both HDOUT and TLOUT registers. Thus all writing to the memory is done by read-modify-write memory cycles. This is fairly unusual, but it has been found in practice that this considerably speeds up many processes involving following chains of pointers.

The flags are usually used by the memory allocation and garbage disposal system, to mark which cells are in use and which are free. Note that it is possible to write to the flags in a cycle which does not write to the rest of the data.

Microcode Instruction Words.

Each microcode instruction is 1 word long. A word in the microcode memory is 40 bits wide, and is split into fields:

bits	field	use
0 - 11	IB	constant (low 12 bits of I field) or jump/branch target address
12 - 15	IA	Tag part of I field or branch condition selector or top 4 bits of jump target address
16 - 19	M	Memory cycle description
20 - 25	V	branch/jump description
26 - 30	F	Instruction code
31	X	Update MAR this cycle with current bus value
32 - 35	B	destination register number
36 - 39	A	source register number

Thus several things can occur in one instruction. To assist the writing of microcode an assembler and a linker exist, running under Phoenix MVT.

The instructions given in the next section are shown in the syntax of this assembler. 'ra' and 'rb' indicate any ALU register (ie 0 to 15), 'xa' and 'xb' indicate any external register (ie MIR, HDOUT, TLOUT, MPC, PORT) except MAR. A value can be written to MAR in any cycle, regardless of what else is happening. 'value' indicates an immediate value from the microcode I fields (IA and IB). A complete syntax of the assembler and details of labels etc. can be found elsewhere.

Quite a lot of things can happen in one microcode instruction, but not all reasonable looking combinations are in fact legal. This is because some fields of the microcode word are used for more than one purpose. The assembler will carefully reject illegal instructions and point out which microcode fields are being used twice. Microcode memory access to fetch the next instruction happens while the previous instruction is being executed, and this pipelining is 'intelligent' in that it notices unconditional jumps and fetches the correct microcode word. But conditional jumps, where the correct word is not known until some computations have occured, may cause a cycle to be lost.

Arithmetic Instructions.

Simple Assignment.

```
rb  :=  ra;
xb  :=  ra;
rb  :=  xa;
rb  :=  value;
xb  :=  xa;
```

The 24 bit value is moved from the source on the right to the destination on the left. If 'value' is specified, it is placed in the I field.

The assembler syntax for a value is:
tag-value, hash-sign, data-value
eg int#5 or char#'A'

The assembler documentation holds more precise details of the assembler syntax. Arithmetic operations are specified using infix assignment operators, in a style stolen from Algol68 and then absurdly exaggerated.

Tag assignment.

```
rb  :T=  ra;
rb  :T=  xa;
rb  :T=  value;
```

Bits 0 to 19 of the destination are unaffected, bits 20 to 23 are replaced by those from the source.

Q assignment.

```
QREG   :=  ra;
rb     :=  QREG;
```

Bits 0 to 19 are transferred to or from the Q register. Thus, Q is only 20 bits long.

Addition and Subtraction.

```
rb    +:=  ra;
rb   +C:=  ra;
rb   +1:=  ra;
rb    -:=  ra;
rb   -C:=  ra;
rb   -1:=  ra;
rb   -R:=  ra;
rb  -RC:=  ra;
```

Register rb is replaced by its own contents combined with those of ra, in one of six ways: add, subtract, add with carry, subtract with carry, add one, subtract one, reverse subtract, reverse subtract with carry. Bits 20 to 23 of rb are unaffected.

Logical operations.

```
rb    &:=   ra;
rb    |:=   ra;
rb    ~:=   ra;
rb  :XOR=   ra;
rb    XOR   ra;
rb    XOR   xa;
rb    XOR   value;
```

Bits 0 to 19 of ra are replaced by a bit-by-bit logical operation. And, or, negate and exclusive or are available. Bits 20 to 23 are unaffected. The operator XOR does not update the contents of rb, but merely places the XOR of the values on the bus. This is usually accompanied by a branch that tests for equality. Bits 20 to 23 are included in this comparison, so the equality of the tags can be tested at the same time.

Multiply and divide.

```
ra  :MUL=  rb;
ra  :DIV=  rb;
```

The specification of these instructions is very complex, and they are unlikely to be of any use outside their intended field of application, ie optimal coding of multiply and divide routines. Using these instructions, a multiply or divide of two unsigned twenty bit operands will occur in twenty cycles.

Shift Instructions.

```
rb   >1=  ra;
rb   >0=  ra;
rb   >C=  ra;
```

Instructions are provided that shift the operand from ra one place to the right, placing the result in rb. The vacated bit (bit 19) is filled with 1, 0 or the carry bit, depending on the instruction. Shift left instructions are not provided, the effect can be achieved by adding a register to itself.

Jumps.

Usually microcode instructions are fetched and executed sequentially. The MPC automatically increments itself and fetches the next instruction while the current one is being executed.

There are two ways of transferring control: jumps and branches. A jump is an unconditional 16-bit transfer of control. There are four types of jump:

```
            JMP  label;
            JSR  label;
            RTS;
            FJMP tagvalue;
```

The last of these results in one cycle being wasted. The other three involve no delay at all.

Simple Jump.

The keyword JMP with a destination means that the next instruction to be obeyed will be that one.

```
eg:
0          r0 := r4      JMP l1;
1      l2: r1 := r5      JMP l3;
2      l1: r2 := r6      JMP l2;
3      l3:  ...
```

Instructions 0, 2, 1 and 3 are obeyed in that order, with no wasted cycles. Note that an instruction with a jump in it cannot assign a constant value from the I field, because the I field of the microcode word contains the jump address.

Subroutine Jump.

The MPC has associated with it a hardware push-down stack that is used for high-speed subroutine jumps. The keyword JSR with a destination specifies that the next instruction to be executed will be the destination one, and that the address of the instruction after the current one should be pushed onto the stack.

```
    eg:
0         r0  := int#1;
1         r0 +1:= r0     JSR l;
2         r0 +1:= r0     JSR l
3         r0 +1:= r0     JMP l2;
4      l: r0 +1:= r0     RTS;
5     l2: ...
```

Instruction 0 assigns a value from the I field into register r0, with tag value 'int' (previously declared as some value) and data value 1. After obeying this program fragment, r0 will contain 5. It obeys instructions 0, 1, 4, 2, 4, 3, 5 in that order, with no wasted cycles.

Subroutine Return.

the keyword RTS specifies that the next instruction to be executed is found at the address on the top of the stack, which is popped. Note that the stack is only four addresses deep, and that it cannot be accessed in any other way.

Arbitrarily Nested Subroutines.

If deeper nesting is required, return addresses must be held in machine registers, eg:

```
 10:     r15 := MPC;
 11:     r15 +1:= r15   JMP subr_entry;
 12:
```

After executing 10, r15 will hold t15#11. After executing 11, r15 will hold t15#12 and the MPC will hold 'subr_entry'. The first instruction of the subroutine will then be obeyed, with no wasted cycles. This instruction will effectively cause a subroutine jump, keeping the return address in register r15 with a tag value of 15.

Note that the following will NOT work

```
        r15 := MPC      JMP subr_entry;     /WRONG
```

The value loaded into r15 will be the value 'tag15#subr_entry'. This is due to the pipelineing of unconditional jumps. reading MPC yields the address of the next instruction to be executed, unless an FJMP or branch is taken in which case it yields one more than the address of the current instruction.

Return from this subroutine is achieved by a forced jump:

```
        r15 := r15      FJMP tag15;
```

Forced Jumps

Forced jumps are only useful in specialised circumstances. The value on the databus in this cycle (as a result of a move or arithmetic instruction) is examined, and if its tag value matches the tag specified in the force jump then the value is forced into the MPC.

eg:

```
        r0 := comb#1;
        r0 := r0 FJMP comb;
```

This would cause a transfer to label 1, which is declared somewhere else. One cycle is wasted in the transfer. This jump does <u>not</u> use up the I field of the microcode word.

The specified tag value can only be one of 12, 13, 14 or 15.

Branches.

A branch can occur in the same instruction as a jump, but there is only room for one destination address in a microcode instruction word so in practice this is only useful if the jump is RTS or an FJMP.

A branch is a conditional transfer of control. It can only change the bottom 12 bits of the MPC so it is only for local transfers of control, but in practice the linker should resolve any problems that this might cause.

If a branch and an RTS are both specified in the same instruction and the branch condition is true, then the RTS is ignored (no value is popped from the stack) and the branch taken.

If a branch and a FJMP are both specified in the same instruction and the FJMP condition is true then the branch is ignored and the FJMP taken. If the FJMP condition is false then the branch is examined in the normal way.

Currently available branch conditions:

```
    BTDEQ   label    br if tag and data both zero
    BTDNE   label    br if tag and data not both zero
    BDEQ    label    br if data zero
    BDNE    label    br if data not zero
    BPOS    label    br if bit 19 of data is clear
    BNEG    label    br if bit 19 of data is set
    BVS/BVC label    br on overflow set/clear
    BCS/BCC label    br on carry set/clear

    BDIN    label    br if data waiting in PORT for input
    BNDIN   label    br if no data waiting for input
    BDOUT   label    br if data output to PORT
                      has not yet been collected by host
    BNDOUT  label    br if safe to output to PORT

    BTAGEQ tag_value label    br if tag = tag_value
    BTAGNE tag_value label    br if tag /= tag_value
    BTAGGE tag_value label    br if tag >= tag_value
    BTAGLT tag_value label    br if tag <  tag_value

    BHDF1   label    br if head flag from memory output is 1
    BHDF0   label    br if head flag from memory output is 0
    BTLF1   label    br if tail flag from memory output is 1
    BTLF0   label    br if tail flag from memory output is 0
```

Note that this may not in fact be a complete list, extra branch conditions are easy to add and the hardware is still under developement, so further branch conditions can be added if they are found to be useful.

In the above references to 'tag' and 'data' refer to the value on the main bus during the instruction, i.e. the result of any calculation or ALU operation or the value being moved in the case of a simple transfer instruction. Carry is the carry out of bit 19 of the current instruction, which must be addition or subtraction if this is to be meaningful.

Overflow is (carry out of bit 18) XOR (carry out of bit 19), ie 2's complement overflow, from this instruction.

```
  eg:
0   10:   r1 := r1     BNDIN 10;
1         r1 := PORT;
2         r0 := char#255;
3         r1 :T= r0;
4         r1 &:= r0;
5         r0 := char#'A';
6         r1 XOR r0        BTDEQ Its_an_A;
7         r1 := r1         BTAGNE char Something_Wrong;
```

Instruction 0 waits for a character to arrive at the port. It is then fetched, given the tag value 'char' and masked so that all but the bottom 8 data bits are cleared. It is then tested to see if it's an upper case A. The last branch would clearly not be taken in this case, but demonstrates the testing of a tag value.

Memory Control Registers.

Various xregs are important when accessing main memory.

write to:	MAR address
	MIR data
output at:	HDOUT head output
	TLOUT tail output

Assignments to and from these registers happens like conventional ALU registers, but note that arithmetic on them is not possible. MAR is assigned to in a special way, however.

```
        MIR := ra;         / data input from register
        MIR := int#57;  / constant data input
        MAR, rb +1:= rb;  / MAR and rb both := rb + 1
        rb := HDOUT;
        rb := TLOUT;
```

Memory Accesses.

Microcode memory cannot be written to, and can only be read as instructions to be obeyed.

In order to write to main memory, the input registers must be set up, and a main memory cycle initiated. This can happen in any instruction and occurs by specifying one of the following in the instruction:

```
    RDHD   RDHDF0   RDHDF1   XHD   XHDF0   XHDF1
    RDTL   RDTLF0   RDTLF1   XTL   XTLF0   XTLF1
```

The operation of these different memory cycles has already been explained, in the section 'Main Memory'. The memory cycle word should appear in the instruction after the arithmetic part, and before any jumps or branches.

Here are some typical memory accesses. More can be found in the example programs at the end of this document.

```
        reg  := tag0#3;
        MAR, reg := reg RDHD;
        rx   := HDOUT;          / rx has HD of location 3

        MAR, reg := tag0#12;
        MIR := tag12#47 XHD;    / HD loc 3 := 12#47
```

Memory pointers should use tag values of 7 or less.

Small Example Section.

This example is taken from the combinator reducer. A complete section is shown, which might form a part of a larger system.

```
        SECTION  Reducer; / Declare section name

/ Tell linker which names in this section
/ are to be exported.
        GLOBAL Reducer_r_loop, Reducer_Patchup,
               Reducer_p_loop, Reducer_Reduced,
               Reducer_Entry;

/ Set up mnemonics for referring to registers.
        REG    forw = 11, back = 12, ret = 15, x = 8;

/ Set up mnemonics for referring to tag values.
        TAG    appl = 7, comb = 12, nil = 8, retad = 15;

Reducer_Entry:           / Label specified by a colon
                           / Entrypoint of section.
                           / This section was called
                           / as a subroutine,
                           / with return address in 'ret'.

        back := nil#0;   / Assign constant value into 'back'
                           / values written tag#constant

        forw := forw  FJMP comb BTAGNE appl Reducer_Patchup;
                           / Look at the value in 'forw'.
                           / If its tag = comb, jump to it.
                           / Otherwise, if its tag /= appl,
                           /  jump to 'Reducer_Patchup'.

 r:     MIR  := back;      / Set up next value to be written
                           / to memory.
/ The next three instructions are quite subtle, and deserve
/ close attention.
/ The intention is to follow the chain of head pointers
/ pointed at by 'forw', reversing the pointers as you go
/ with 'back' pointing to the chain of back pointers.
/ as soon as something that is not an application pointer
/ is reached, the process is stopped.
/ If it is a combinator, (tag = comb) then the data
/ field is a microcode address where the routine for
/ that combinator will be found. This address must be
/ jumped to.
/ The loop is written to proceed as fast as possible!
Reducer_r_loop:
        MAR, back := forw  XHD;
        forw := HDOUT   FJMP comb BTAGNE appl Reducer_Patchup;
        MIR  := back            JMP Reducer_r_loop;
/ Total time taken - 850ns per cell.

/ Execution has finished and the expression cannot be
/ further reduced.
/ Follow the chain of back pointers, re-reversing them,
/ and exit.
Reducer_Patchup:
        back := back            BTAGNE appl Reducer_Reduced;
        MIR  := forw;
Reducer_p_loop:
        MAR, forw := back  XHD;
        back := HDOUT           BTAGNE appl Reducer_Reduced;
        MIR  := forw            JMP Reducer_p_loop;

Reducer_Reduced:
        ret  := ret            FJMP retad; / return to caller
```

Extended example.

The initialisation section from the combinator reducer is presented in its entirity, to demonstrate for the interested reader some of the techniques that are used.

```
/*****************************************************
/*                                                   *
/*    Initialisation.                                *
/*                                                   *
/*****************************************************

/ Author: WRS

        SECTION Init;
        GLOBAL Supervisor_Entry;
        GLOBAL Reducer_Entry,
               Entrypoint_to_R_Combinator;
        REG r0 = 0, r1 = 1, r2 = 2, r3 = 3, r4 = 4,
            r5 = 5, r6 = 6, r7 = 7,
            x = 8, y = 9, z = 10, forw = 11, back = 12,
            stack = 13, free = 14, ret = 15;
        TAG pair= 1, appl = 7, int = 9, char = 11,
            comb = 12, nil = 8;

        / Important addresses
        MANIFEST heap_bot            = 35,
                 comb_count          = 1,
                 numeric_comb_count = 2,
                 mem_full_flag       = 3,
                 input_stream_ptr    = 4,
                 spare_free_ptr      = 5,
                 GC_number           = 6,
                 top_of_heap         = 8;

/ This will be physical location zero in the micro-memory.
Init_Entry:
        r0 := r3 := r4 := r5 := r6 := r7 := int#0;
        forw := back := stack := free := ret := int#0;
        r2 := PORT; / clear it just in case there's junk.
        r1 := nil#0;

Create_Memory_Top:
/ 64K nodes of main memory
/ it would be nice to find the top of memory
/ but that could cause a parity error
        x := appl#0;      / bottom of memory
        y := appl#2047;   / Largest constant I can input !
        r0 := y;
        r0 +:= r0;        / 4097 -'1
        r0 +:= r0;        / 8195 - 3
        r0 +:= r0;        / 16...
        r0 +:= r0;        / 32...
        r0 +:= r0;        / 64...
        r0 |:= y;         / 65535 at last !
        r0 -1:= r0;       / location FFFF is used by BBCSKIM
                          / to allow memory inspection
/ clearing memory ensures that all statistic counters
/ are cleared to 0, and that there are no bogus pointers
/ on the heap.
clear_memory:
        MIR := int#0;
        MAR, x := x     XHDF0;
c_loop: x := x; / dud because MAR-write problem
        x XOR r0        XTLF0     BTDEQ c_exit;
        MAR, x +1:= x   XHDF0     JMP   c_loop;
c_exit:

/ Now we must chain together all cells on the heap.
/ The last free cell is not in the heap,
/ becuase sweep in the GC is faster
/ if the last cell is always free.
        r0 -1:= r0;         / Last cell not chained on.
        x := appl#heap_bot;
        MIR := r1;         / 0 signifies end of chain.

/ 500 cells are reserved as the spare free chain.
/ They are only used if we run out of memory.
spare_500:
        z := int#500;     / counter
        MAR, x := x      XHD;
s_loop: MIR := x;
        z -1:= z                 BDEQ s_exit;
        MAR, x +1:= x    XHD     JMP  s_loop;
s_exit:

/ write various useful things to page 0 locations
        y -1:= x;
/ HD is an assembler macro, and produces two microcode words
        HD appl#spare_free_ptr := y; / 500 spare cells
        HD appl#top_of_heap := r0;

/ Now all the rest of memory is chained together
/ to form a freechain.
make_freechain:
        MIR := int#0;     / start of main chain
        MAR, x := x     XHD;
m_loop: MIR := x;
        x XOR r0                BTDEQ m_exit;
        MAR, x +1:= x   XHD     JMP   m_loop;
m_exit:
        free := x;              / CONS will now work
        x := y := r0 := int#0;  / GC'able

main_program:
        / set up input stream
        x    := comb#Entrypoint_to_R_Combinator;
        / CONS is an assembler macro
        / it generates three microcode words
        y    := CONS(x, x); / input stream buffer
        HD pair#input_stream_ptr := y;

        ret := MPC;
        ret +1:= ret JMP Supervisor_Entry; / main CLI
        / done - output debug signal and wait
        r2 := char#255;
done:   PORT := r2                JMP done;
```

Comparison with other microcoded machines.

The conventional idea of microcode is that it is a method of implementing a standard von Neumann computer. A small microcode program reads in an instruction stream from main memory and dispatches to the various handlers for the separate instructions. SKIM leans more towards the idea that this is a machine with separate program and data memory. The 16 bit microcode address bus is much bigger than is considered necessary for most machines, and it is envisaged that compilers can be made to produce microcode. This may seem wasteful of space, as traditionally microcode resides in expensive, fast RAM but the cost of memory is falling all the time and this policy now seems the best way to improve the price/performance ratio.

One difference that this leads to is that most machines, such as the PERQ[ICL 82] or Lilith[Wirth 81], include hardware for the rapid reading of an instruction stream. SKIM does not have this.

Another specialisation envisaged in SKIM is that the memory is designed with heap organisation in mind. The tagged architecture and flags for marking make garbage collection easy to automate. The only other processors known to the author that have this are considerably more expensive [Symbolics 82].

Using the Assembler on Phoenix/MVT

On Phoenix/MVT, the following instructions can be used to run the SKIM Microcode Assembler and Linker.

c ws12.skim.asm:r program <filename>

This runs the assembler on the source program in file <filename>. Error messages will appear at the terminal. If there are no errors, the following output files will be created:

&list	assembley listing of the program
&asm	numeric/symbolic form of microcode words generated

The &list file is arranged in pages for the printer in Reception. 'Printout &list' will cause a printed listing, showing the exact microcode generated for each instruction.

c ws12.skim.lked:r

This takes its input from &asm and produces the error messages at the terminal. If there are no errors, the following output files will be produced:

&mu	numeric form of microcode, suitable for loading
&index	not useful at present

If the program is composed of more than one section, each should be assembled separately, and their &asm output files concatenated and placed in &asm. The link editor will then resolve references between sections.

The structure of a SKIM Microcode Assembler Program

A program to run as microcode for SKIM is split into sections. Each section must have a different name. The sections must be compiled one at a time. The assembler is one-pass and generates two output files, a listing suitable for the line printer and the 'asm' output file, which is in a format suitable for the link editor. The listing will include error messages and details of the code produced, split into fields. References to labels and so on are not checked at this stage, labels are passed to the linker as text strings.

In order to link the resulting code, the 'asm' output of all the sections should be concatenated, with the initialisation segment at the front, and fed into the link editor. The link editor produces two output streams, one consisting of error messages (usually unresolved label references) and the other is a 'mu' load module, in a format that will be accepted by SKIM's host processor for loading into SKIM. SKIM will obey the microcode program, starting at word zero.

Syntax.

The following is a fairly loose syntax description of the language accepted by the assembler. A BNF-like form is used, with square brackets to denote an optional part of the input and

```
          [    ]*
```

to denote something that can be repeated 0 or more times.

Input is free format, spaces and newlines can be added to improve legibility. A slash (/) denotes a comment - all of the rest of the line is ignored.

A complete section has the form:

```
     SECTION ident; [[label:]* (command | statement)]*
```

The rest is as follows.

```
command        ::=  ;                     / superfluous ";" ignored.
                    SECTION name
                    GLOBAL namelist
                    MANIFEST name = value[, name = value]*
                    REG      name = value[, name = value]*
                    TAG      name = value[, name = value]*
                    LIST
                    NOLIST
                    COND value
                    ENDCOND

statement      ::=  statement body [memcode] [jump] [branch];

statement body::=  PUSH  reg [, reg]*
                    POP   reg [, reg]*
                    [MAR,] assgn
```

```
assgn        ::=  reg      := assgn2
                  reg    op:= assgn2
                  reg      := CONS(assgn2, assgn2)
                  reg      := CONS tag (assgn2, assgn2)
                  reg      := memsrc

                  reg      := xsource
                  reg      :T= xsource
                  reg      XOR xrource

                  xreg     := xsource
                  xreg     := assgn2
                  mem      := assgn2
                  QREG     := assgn2
                  reg      := QREG

assgn2       ::=  reg
                  assgn              //provided that it
                                     //starts with a reg

memsrc       ::=  mem
                  mem :=:= assgn2

mem          ::=  HD memaddr
                  TL memaddr

memaddr      ::=  reg
                  xsource
                  (reg := assgn2)
                  (reg := xsource)

memcode      ::=  one of:
                   RDHD RDTL  // identical in effect
                   XHD  XTL   // exchange
                   // as above, but also write to
                   // corresponding flag bit:
                   RDHDF0 RDTLF0 RDHDF1 RDTLF1
                   XHDF0  XTLF0  XHDF1  XTLF1

jump         ::=  FJMP tag
                  RTS
                  JSR label
                  JMP label

branch       ::=  ccode label
                  tagcode tag label

xsource      ::=  xreg
                  literal

literal      ::=  tag#value

value        ::=  number
                  label
                  charconst
                  name     //previously declared as MANIFEST

number       ::=  digit[digit]*  // no blanks in middle
```

```
charconst      ::=  'anychar'

label          ::=  name   // that appears somewhere
                           // with a colon after it.
tag            ::=  name   // that has been declared as TAG
reg            ::=  name   // that has been declared as REG

name           ::=  letter[alphanum]*

alphanum       ::=  letter
                    digit
                    underscore    // _

op:=           ::=  :=
                    :T=     // move tag field
                    +:=     // add
                    -:=     // subtract
                    +1:=    // increment
                    -1:=    // decrement
                    +C:=    // add with carry
                    -C:=    // subtract with carry
                    -R:=    // reverse subtract
                    -RC:=   // reverse subtract with carry
                    &:=     // bit-by-bit logical and
                    |:=     // bit-by-bit logical or
                    :XOR=   // bit-by-bit exclusive or
                    :MUL=   // multiply step
                    :DIV=   // divide step
                    XOR     // bit-by-bit compare

xreg           ::=  MPC     // micro program counter, read only
                    PORT    // communication with host, r/w
                    MIR     // memory input register, write only
                    HDOUT   // memory output register, read only
                    TLOUT   // memory output register, read only
                    MAR     //memory address register, read only
                    DUD     // non-existant - write only,
                            // value thrown away.
(note that writing to MAR is done in a special way,
 see 'statement body' syntax)

ccode          ::=  BDEQ    // branch if data equal
                    BDNE    // branch if data not equal
                    BTDEQ   // br. if tag and data both equal 0
                    BTDNE   // br. if tag and data not both = 0
                    BCC     // br. if carry 0
                    BCS     // br. if carry 1
                    BVC     // br. if overflow 0
                    BVS     // br. if overflow 1
                    BDIN    // br. if data input waiting in PORT
                    BNDIN   // br. if no data input waiting
                    BDOUT   // br. if data output waiting
                    BNDOUT  // br. if no data output waiting
                    BHDF0   // br. if head flag 0
                    BHDF1   // br. if head flag 1
                    BTLF0   // br. if tail flag 0
                    BTLF1   // br. if tail flag 1
          // the last four refer to the latched output
          // of the last memory cycle.
```

```
tagcode         ::=  BTAGEQ     // b if tag = given tag value
                     BTAGNE     // b if tag /= given tag value
                     BTAGGE     // b if tag >= given tag value
                     BTAGLT     // b if tag < given tag value
```

Compound Instructions

PUSH, POP and CONS are pseudo-instructions, that generate several instructions each. The default tag value of a CONS is whatever the freechain uses. They use the following mechanism.

```
    REG stack = 13, freechain = 14;
    Both of these are head-linked lists.

    PUSH r1, r2, ... generates
      stack := CONS(stack, r);    for each register
```

The following code is generated by popping a register.

```
      MIR    :=  free pointer;
      free pointer := stack pointer    XHD;
      stack pointer := HDOUT;
      reg           := TLOUT;
```

Note that the node is reclaimed and placed on the front of the free chain. The stack is NOT checked for underflow.

In the case of a multiple pop, the following is generated.

```
    MIR   := free;
    MAR, free := stack  RDHD;
    reg1 := TLOUT;
  / Then this, for all registers except the first and last.
    regn := TL(stack := HDOUT);
  / Then this for the last one.
    MAR, stack := HDOUT XHD;
    stack := HDOUT;
    lastreg := TLOUT;
```

Thus the cost of a pop is 600ns for each register, +500ns.

For call a := CONS tag (b, c).
a = b = c is not allowed.
If a /= c, the following code is generated.

```
    MIR  := b;
    MAR, a := free XHD;
    MIR  := c               JSR GC_check;
    a :T= tag#0;
```

Otherwise, the following code is generated.

```
    MIR  := c;
    MAR, a := free XTL;
    MIR  := b               JSR GC_check_2;
    a :T= tag#0;            / omitted if no tag specified
```

```
  The time taken is 1 microsecond
  + 250ns for the tag assignment.
  At the (global) check labels is something of the form:
    gc_check:    free := HDOUT  XTL  RTS  BTAGNE appl oops;
    gc_check_2:  free := HDOUT  XHD  RTS  BTAGNE appl oops;
  These two labels are internally declared to be global,
  so it is not necessary for the programmer to do so.
```

Use of HD or TL on their own generate two instructions, i.e.

```
        a := HD b;
```

is equivalent to

```
        MAR, b := b;
        a := HDOUT;
```

Labels that are defined as global will have their definitions exported, and can be referred to by other sections.

Register, tag, section name, manifest and label names must all be distinct.

16 July 1983 - Known Bugs

SKIM's hardware has now been finished and testing of the assembler really begins. It has been discovered that many of the multiple instructions described above do not in fact assemble correctly. People intending to use SKIM microcode should avoid such things as

```
      a := HD b :=:= c;
```

but should write out such combinations individually. PUSH, POP and CONS all work, and so should any statement that only generates a single instruction. Simple ones like

```
        HD a := b;
        b := TL c;
```

are also OK. If in doubt, use only commands that generate one single microcode instruction, as per a conventional assembler.

February 1984

The assembler has now been in use for over a year, and about 4K words of microcode have been written in it for the reduction of combinators. Extensive use is now made of a macro prepass over the input source, using the MLI general-purpose macro processor.

INTERACTIVE DEBUGGING OF SKIM MICROCODE

The SKIM processor is connected to a host processor, through which all peripheral transfers occur. But the host is also useful for debugging microcode programs. In addition to an 8-bit bidirectional port joining them, the host has control of SKIM's clock logic and microcode pipeline.

The following operations are possible from the host:
    stop/start/single step SKIM's CPU clock
    load microcode words into SKIM's microcode memory
    examine SKIM's ALU registers
    examine memory locations in SKIM's main memory
    execute single microcode instructions
    examine and change SKIM's microcode sequencer (MPC)

This means that debugging microcode programs is not a difficult operation. The main cycle for debugging is:

    create/edit the microcode program source
      on a mainframe computer
    assemble and link the microcode program
    transfer the microcode load module to SKIM's host
    load up SKIM's front panel program
    load the microcode load module
        into SKIM's microcode memory
    execute or single step the microcode program.

The front panel program keeps a copy of the microcode symbol table, and gives a continuous full-screen display of SKIM CPU registers and memory.

At the time of writing the host is a BBC Microcomputer System, with facilities for transferring files to and from the local IBM mainframe.

Screen Display of SKIM Microcode Front Panel

```
1       BBCSKIM  V2.4 (July 83)    ('H' for help)
2
3       Pc=Supv#Supv_wait_for_input_char
4         =0000=0                           r0=000000
5       IO=FF   no output char              r1=000000
6                   input char              r2=000000
7               no parity error             r3=000000
8       Q =0FFFFF, C=0                      r4=000000
9       MAR  =000000                        r5=000000
10      HDOUT=000000 T                      r6=000000
11      TLOUT=000000 T                      r7=000000
12      MIR  =000000                        r8=000000
13      000000: 000000 000000 TT            r9=000000
14      000000: 000000 000000 TT            ra=000000
15      000000: 000000 000000 TT            rb=000000
16      000000: 000000 000000 TT            rc=000000
17      000000: 000000 000000 TT            rd=000000
18      000000: 000000 000000 TT            re=000000
19      000000: 000000 000000 TT            rf=000000
20          Command:_
```

In the diagram above 000000 stands for a 6-digit hex number. The front panel takes single key commands, and updates the screen after every key. Thus stepping results in a continuous display of what is going on.

Lines 13 to 19 display lines of memory, as address, head, tail and both flags. Either a consecutive block can be displayed, or a connected tree structure.

Line 3 shows the microcode label fixed to the instruction that will be executed next, in the form

section_name#label_name + offset

This allows the programmer to see instantly where he is in the microcode program.

Debugging Hardware

These facilities aid the debugging of not only microcode programs, but also of the hardware. If the hardware for communication with the Host is working then individual instructions can all be independently checked, without relying on the microcode memory functioning properly. This allows wiring errors and chip failiures to be quickly identified. If a parity error occurs in the main memory when a program is running then the processor stops immediately and the front panel is displayed. Thus the occurrence can be logged, with an exact record of what instruction was being executed, etc.

REFERENCES

[Clarke 80]: T. J. W. Clarke, P. J. S. Gladstone,
C. D. MacLean and A. C. Norman,
SKIM - the S, K, I Reduction Machine,
Proceedings of the 1980 ACM Lisp Conference, pp 128 - 135,
August 1980.

[FLATS 78]:
Report of The FLATS Project, Volume 1,
Information Science Laboratory,
Institute of Physical and Chemical Research,
Wako-Shi, Saitama, 351 Japan,
October 1978

[Symbolics 82]:
Symbolics 3600 reference manual,
Symbolics Inc, 21150 Califa Street, Woodland Hills,
California 91367

[PERQ 82]:
Introduction to PERQ,
UK Software and Litererary Supply Centre,
International Computers Limited,
ICL House, Putney, London SW15

[Wirth 81]:
The Personal Computer Lilith,
Institut fur Informatik,
Eidgenossische Technische Hochshule,
Zurich,
1981

[68000 82]:
MC68000 16-bit Microprocessor User's Manual,
Prentice-Hall,
1982.

[BBC 82]: J. Coll, edited by D. Allen,
BBC Microcomputer System User Guide,
British Broadcasting Corporation,
35 Marylebone High Street, London W1M 4AA,
1982.

[ACM 82]:
Conference record of the
1982 ACM Symposium on Lisp and Functional Programming,
August 15-18, 1982,
ACM order no 552820

[Hughes 82]: R. J. M. Hughes, Oxford University,
Super Combinators: A New Implementation Method for
Applicative Languages,
[ACM 82].

[Turner 79]: D. A. Turner,
A New Implementation Technique for Applicative Languages,
Software Practice and Experience, Volumn 19, pp31-34, 1979