Swamp: A Fast Processor for Smalltalk-80

David M. Lewis, David R. Galloway, Robert J. Francis, and Brian W. Thomson

Computer Systems Research Institute
University of Toronto, Toronto, Canada

Abstract

A processor for the Smalltalk-80[†] programming language is described. This machine is implemented using a standard bit slice ALU and sequencer, TTL MSI, and NMOS LSI RAMS. It executes an instruction set similar to the Smalltalk-80 virtual machine instruction set. The data paths of the machine are optimized for rapid Smalltalk-80 execution by the inclusion of a context cache, tag checking, and a hardware method cache. Each context is only partly initialized when created, and has no memory allocated for it until a possibly non-LIFO reference to it is created. The machine is microprogrammed, and uses a simple next micro-address prediction strategy to obtain most of the performance of pipelining without the attendant complexity. The machine can execute simple instructions at over 7M bytecodes per second and has a predicted average throughput of 1.9M bytecodes per second.

1. Introduction

Smalltalk-80 [1] (henceforth, ST-80) is an object-oriented programming language which executes very slowly on conventional processors. The reasons for this slow execution are common to object oriented languages, and include:

- Message sending with no statically determinable type information means that the procedure to perform the actions requested by a message must be determined dynamically
- Non-LIFO control means that storage for activation records (called *contexts* in ST-80) must be allocated on the heap
- automatic storage management means that unused storage must be detected and reclaimed by the implementation

† Smalltalk-80 is a trademark of Xerox Corp.

This paper describes a processor that is designed for the fast execution of ST-80. This processor, called *Swamp* (Smalltalk Without All that Much Pipelining), applies an important and general principle that is often used in creating efficient systems. This principle is to identify those simple special cases which are dynamically frequent, and which do not require the expense of the general case, and optimize the handling of them. This principle is applied to the above problems in the following manner:

- messages sent to integers or other common classes of objects are handled efficiently
- contexts are allocated based on the assumption that they will be used in LIFO order, decreasing the memory management traffic
- storage is allocated on the assumption that recently allocated storage will not be used for long, while storage that has been used for a long time will continue to be.

The identification of these problems and the application of similar solutions is not new: they have been used in both software implementations [10] [11] and hardware implementations [3] [14] of ST-80. Where Swamp differs from previous implementations is the application of these solutions.

The ST-80 system is described in terms of an implementation called the ST-80 *virtual machine* (ST-80VM). Implementations on standard processors which adhere to this specification generally have poor performance, due to the time required to interpret the ST-80VM instruction set in software and due to the lack of any special hardware designed to speed ST-80 execution.

Much higher performance is available by translating the ST-80VM instructions into the native instruction set of the processor. Even when the native instruction set contains no support for ST-80, good performance can be achieved [11] [15]. The SOAR processor has demonstrated that when support for ST-80 is added to the processor, implementations which translate ST-80VM code into the native code can have good performance even with a relatively slow processor [3].

The use of the ST-80VM has several advantages. It is compact, which reduces both space requirements and instruction fetch bandwidth. The ST-80VM is also portable, and allows different implementations to use the same implementation of the compiler and debugger. Published implementations with special hardware for ST-80 have good performance, but we suspected better performance was possible for the same implementation technology. SOAR uses a RISC organization, which requires high instruction bandwidth. Sword-32 [14] uses the ST-80VM, but we believed better performance could be obtained with comparable technology. Although the cycle time of Sword-32 is 125ns, less than one fourth of the 550 ns cycle of SOAR, its performance is only 25% better. We believed that a processor carefully designed to suit the requirements of ST-80, and using an instruction set similar to the ST-80VM, could perform as well as a native code implementation, but still have a dense instruction set and low memory bandwidth.

The goal of Swamp project is to investigate high performance machines for the execution of ST-80. In the first phase, we intend to investigate machines using instruction sets with only very minor differences from the ST-80VM. This will also be used to gain experience with ST-80. In the longer term, we expect that changes to the ST-80VM will be necessary for maximum performance.

This paper describes a processor designed as a result of the first part of the project. The goals of this work were to design a processor capable of executing ST-80 at a rate comparable to a typical workstation executing programs in conventional languages such as C or Pascal. This meant a performance goal of one to two million byte codes per second. This implementation was constrained to using standard chips because we did not have the resources to devote to a VLSI implementation. Because this processor is intended as a "one-of" prototype to gain experience with ST-80, there is also little point in casting it in silicon. For this reason, bit slice technology was selected for the data paths and control sequencer.

2. Object Representation

Object representation is an important design decision in implementing a ST-80 system. The VM defines a representation suitable for small machines, and restricted to 32K objects. For a high performance machine such as ours, this is unsatisfactory. We use 32 bit object references (called OOPs).

Another disadvantage of the ST-80VM is its use of an object table and reference counting as the garbage collection method. These are very slow to implement, because every reference to a field of an object takes two memory references, and because of reference counting overhead. For this reason, we used an object representation in which object references point directly to the memory locations used by the object, as did BS [17] and SOAR [3] [16].

Garbage collection is done with a generation based scheme similar to [7] and [8]. Memory space is divided into a number of generations, 8 in this case. Generation 0 is the oldest generation, and objects are normally allocated in generation 7. All objects refer to other objects by their OOPs, but such references are only unconstrained for references from objects in generation i to objects in j when $j \leq i$. All other references require that the object containing the reference be noted in an entry table. There are 28 such entry tables, containing references to all of the objects in generation i which refer to objects in generation j, where $i < j$.

When there is no more space for objects in generation 7, garbage collection must take place. The entry tables provide a set of potential live references to objects in generation 7. All objects still accessible in generation 7 are moved into generation 6. Generation 7 is then reused as a pool of free space. The same approach is used when generation 6 or older generations fill up. Older generations are garbage collected less frequently, and generation 0 is only garbage collected off-line. The size of each generation can be optimized by observing the lifetime distribution of objects. A minor difficulty is to avoid having one generation fill up when garbage collecting another. This is most easily avoided by having a spare pool of storage of the same size as the largest non-0 generation.

The desire for maximum speed encourages the use of an object representation in which objects of class **SmallInteger** are represented by a bit pattern similar to the 2's complement representation of the value. Our processor carries this to the maximum extent possible. **SmallIntegers** are represented in 2's complement form, but restricted to a subset of the set of values representable in 32 bits. This means that the manipulation of **SmallInteger** data can take place without any masking or shifting.

During execution of a ST-80 program, most references are to data contained in contexts. Our architecture treats contexts specially in order to maximize the speed of operations on them. ST-80 allows program references to contexts to be made, however, requiring that the implementation appear to represent contexts in the prescribed form. Contexts are not often referred to by programs, so it is advantageous to represent them in a different, more efficient, manner, as long as the required information can be recovered. In order to do this, our architecture requires that contexts reside in a distinct address space, so that references to them can be easily detected and cause special handling to generate the correct information.

Table 1 gives the representation of objects in our system. This OOP representation has some similarities to that of SOAR [16]. OOPs with most significant bits (MSB's) of 11 or 00 are **SmallIntegers**, and the OOP contains the 2's complement representation of the integer. OOPs with MSB's of 01 are other object references, comprising a 3 bit generation number and a 27 bit memory address. Memory addresses are also 27 bits, so any field in an object can be read by adding the OOP of the object to the index of the desired field and reading that word in memory, without having to perform indirection in an object table. Contexts are distinguished with MSB's of 10.

Representation	Meaning
$00b_{29}b_{28}\cdots b_0$	2's complement **SmallInteger**
$11b_{29}b_{28}\cdots b_0$	2's complement **SmallInteger**
$01g_2g_1g_0a_{26}a_{25}\cdots a_0$	object at address A
$10g_2g_1g_0a_{26}a_{25}\cdots a_0$	context at address A

Figure 1. OOP Representation

Objects in memory consist of a 3 word header followed by the OOPs contained in the object. The first word of the header gives the class of the object, the second gives the size of the object, and the third gives a hash value, which is a **SmallInteger** that the OOP of the object can be mapped into.

Implementations of ST-80 which use OOPs that directly address memory do make one feature of ST-80 very expensive. This is the message **become:**, which causes all references to an object to subsequently refer to some other object. This is easy to implement with an object table, but requires sweeping memory to find all references to the first object in schemes in which OOPs directly reference memory. We dislike the presence of **become:** because of its non-local effects, and intend to rewrite ST-80 to eliminate all uses of it and eventually not support it. We initially considered a scheme similar to [9], using invisible pointers to "becomed" objects, supported by the hardware, but rejected it because the complexity and loss of speed didn't appear to be worth the minimal advantages of **become:**.

3. Architecture Overview

A block diagram of the Swamp processor is given in figure 2. The instruction fetch unit (IFU) is responsible for fetching instructions and decoding them. It maintains a 6 byte queue of instructions to execute, and provides the first byte of the next instruction in the IFU queue for use by the microsequencer, called *nextop*. It also provides two decoded bytes, *op1* and *op2*, for use as operands. Refill time from loading a PC to dispatch on a new instruction is 2 microcycles.

The primary block of importance is the context cache, labelled CCache, and the blocks for addressing it, CCptrs and CCA. The context cache is a dual ported set of 512 32 bit registers. It is divided into 128 directly addressable registers and 16 banks of 24 words. Each bank is large enough to hold a small ST-80 context. Two adjacent banks are used to hold a large context. The microcode is responsible for insuring that the contexts which can be directly referred to by a ST-80 instruction are in the context cache.

A four bit current context number register, called *CCN*, indicates the identity of the currently active context in the context cache. A five bit register, called *cflags*, describes the type of this context (method or block), and the size of this context and its caller (sender). A collection of 7 registers, called the CCPtrs, is used to refer to data in the context cache. Two registers, called *ccbase* and *hcbase* are used to refer to the currently active context and possibly another context (called the home context). Two other registers, called *ap* and *sp* are used to point to the arguments and temporaries, and the top of stack in the currently active context.

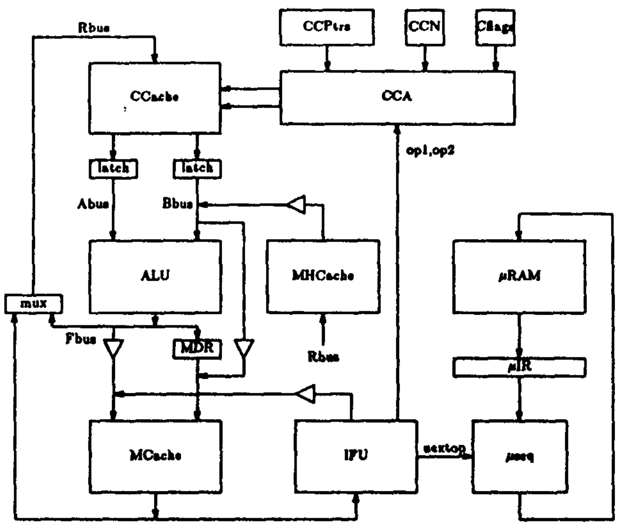

Figure 2. Architecture Overview

There are three other registers, called *ocbase*, *temp1*, and *temp2* used for scratch registers.

The architecture is a two address one, in which two values may be read and placed onto the Abus and Bbus, entering the ALU, and the result written back into one of the locations read. A variety of special values can also be placed onto the Bbus. The addresses used to access the context cache can be generated in a variety of ways by the context cache addressing (CCA) box. The most complicated way allows a sum of one of the CCPtrs, plus or minus either of *op1* or *op2*, plus a small constant in the range -1..8 to be computed and used as an address in the context cache. This is useful when an instruction contains an operand which refers to some location in a context or a value buried in the stack. A related form of addressing allows one of the CCPtrs and the small constant to be added and used as an address in the context cache, and the CCPtr to be incremented or decremented after the microinstruction. This is useful for pushing or popping the stack. Only a subset of the CCptrs can auto-increment or decrement. The first 128 registers can be addressed via an absolute register number. The *CCN* and CCPtrs are also capable of being modified in a variety of special ways during message sending or returning.

In order to speed message sending where contexts are used in a strictly LIFO manner, contexts are not fully initialized when created. Contexts exist in a variety of states, called soft, mushy, and hard. These varieties are similar to the organizations of contexts in [11]. A soft context is the minimal context. It does not have a memory allocation reserved for it, and there is no OOP that can refer to it. The sender, receiver, and argument fields of a soft context are not initialized. The sender is implicitly the context at the next lower address in the context cache, and the arguments are on the sender's stack, and pointed to by *ap*. When a context is first created, it is created as a soft context. A soft context can only be used for LIFO control, and must be converted into another form when the possibility of non-LIFO control exists.

The next form of context, mushy, also has no memory associated with it, but has the sender field valid and the arguments have been copied into it. A mushy context is only used for LIFO control, but its sender might not be in the context cache.

A fully initialized context is called hard, and in addition to the sender field, receiver, and arguments, has a memory allocation associated with it. A hard context will be either in the context cache or in main memory at any time. An additional field in the context points to the memory locations reserved for it. The main memory copy of a context has a field which indicates whether the context is currently in the context cache or main memory. Contexts are allocated in a separate memory space so that program references to fields in contexts can be detected and the actual location of the context determined. Contexts are initially created soft, and will be converted into hard ones when an OOP must be generated for the context. Statistics in [11] indicate that less than 10% of contexts are used in a manner that requires that an OOP be generated for it. When a context is converted into a hard one, all of its callers must also be converted. Soft and mushy contexts are freed upon return, but hard contexts must be reclaimed by the garbage collector.

The data read onto the Abus and Bbus is latched and passed to an ALU which produces a result on the Fbus. If no memory reference is being made, this result is passed to the Rbus. If a memory operation is being performed, the Fbus is used as the address and the result read from the memory cache (Mcache) is placed on the Rbus. Memory writes use the Fbus as the address, and either the Bbus or MDR as the data.

Tag checking circuitry can detect if the Abus, Bbus, and Fbus each contain integer OOPs, whether the Fbus contains a context OOP, and whether a memory write is of a newer generation OOP into an older generation.

The block labelled MHcache is a method lookup cache which, given a class and message symbol, will hash the two values and perform tag checking, and read the method header into a special register. This takes place concurrently with other actions during message sending, and greatly increases the speed of message sending.

In order to simplify microprogramming, the microprogrammer's view of the architecture is a non-pipelined machine. For increased speed, writes are performed in the cycle after the cycle which performs the operation. Bypassing logic detects read-write conflicts, and forwards results without a time penalty. The CCA also computes the addresses in the cycle before they are used, but has access to the next-cycle values of the CCPtrs, so there is no hazard.

4. Microcode Sequencer

Our processor uses a standard micro sequencer [12] for micro program sequencing, but augments it with a powerful wide branching capability in order to be able to rapidly test a number of conditions simultaneously. The sequencer maintains an internal microprogram counter, and has a bus, the seqDbus, attached to it. This bus can be used to specify the destination address of conditional branches, or iteration counts. Our machine can drive this bus from several different sources under microcode control. In the most common case, the seqDbus is driven from a microword field called *branchbase*. It is also possible to cause *branchbase* to be modified by bitwise ORing of a small number taken from a variety of sources. These are called the wide branch sources. The seqDbus can also be driven from a two bit instruction set register and the nextop field from the IFU, allowing byte code dispatching. A special value outside of the range of byte codes is inserted if the IFU does not have a complete byte code in its buffer.

The wide branch sources are used to test a number of conditions simultaneously. These sources are arranged so that 0 is the most common result, where it is possible to predict the result statically. As an example, the *trapBranchNotIntABF* wide branch is used when performing operations on OOPs which are expected to be **SmallIntegers**. It produces the value 0 if the A, B, and F buses all contain integers, 1 if A is not an integer, 2 if A is an integer, but B is not, and 3 otherwise. This feature is used to implement a variety of tag checking traps, as does the Symbolics-3600 [18] and SOAR [16].

In keeping with the goal of providing a non-pipelined architecture to the microprogrammer, a wide branch can use results generated in the cycle that specifies the branch. In the interests of speed, however, the implementation attempts to pipeline instructions. When a wide branch is specified, the machine will fetch the next microinstruction based upon the assumption that the wide branch value will be 0. Near the end of the cycle, this assumption is checked, and if it is wrong, the cycle is stretched while the correct microinstruction is fetched. The dynamically frequent wide branches are organized so the the value 0 is the most common one. The machine will normally run with short cycles. This gives most of the performance of pipelining without the extra programming complexity. It is also doubtful that a pipelined machine would be able to accomplish any useful work in the cases where our machine cannot predict the branch.

Wide branches actually come in two flavours: traps and branches. A wide branch of the trap flavour will prevent any modification of the Ccache or CCPtrs if the branch value is non-0. This allows the microprogrammer to perform destructive operations on data at the same time as checking that the assumptions are true, without trashing the data if the assumptions are false.

The wide branching also interacts with the next byte code decoding. When next byte code decoding is specified, a wide branch may also be specified. If the wide branch modifier is 0, then the byte code decode will be done. Otherwise, the branch will be to *branchbase* ORed with the wide branch modifier. This allows the microcode to immediately execute the next byte code if the assumptions made were correct.

Two examples will illustrate the combined power of the data paths and sequencer. The first, in figure 3, is the microcode for the push-temp byte code. The syntax "A@reg#n+m" indicates that the Abus should be driven with the register pointed to by the sum of *reg*, *n* and *m*. *Reg* is a CCPtr, and *n* can be an operand or its complement, or an increment/decrement command, or 0. *M* is the small constant in -1..8. The push-temp byte code is implemented by a single microinstruction. The microcode places a temporary in the current context on the Abus. The temporary that is to be pushed is indicated by the op1 field of the current byte code. The ALU is set to pass it unmodified (zero extend a 32 bit quantity to a 32 bit quantity), and write the result back into the context cache, while incrementing the stack pointer. The next byte code can then be executed, by performing an unconditional branch with the "macro" field set. The push-temp byte code always takes one microcycle.

The second example, in figure 4, is the microcode for the send-+ byte code, which sends the message + and one argument to the receiver buried one deep on the stack. The receiver and argument will most often be **SmallIntegers**, so the microcode assumes that they are, and dispatches the next byte code if they are. If the receiver and argument are not both **SmallIntegers**, or overflow occurs, then a branch to sendPlus plus 1, 2, or 3 will occur. The send-+ byte code will complete in one microcycle in the vast majority of cases (over 95%, according to information contained in [5]). In the other 5% of the cases the cycle will be stretched while the target of the microbranch is fetched, and the general code to send the message + will be invoked.

```
A@hcbase#op1+ArgOffset B@sp#postinc+1
          aluZeroExtA writeB seqOpGoToD macro
```

Figure 3. Push-temp Microcode

```
A@sp#postdec B@sp#0+minus1 aluAdd writeB
          trapTrapNotIntABF seqOpGoToD
          sendPlus macro
```

Figure 4. Send-+ Microcode

The byte code to send a message requires 9 microcycles if two or fewer temporaries are created. The most common execution path is illustrated in figure 5. The first microinstruction fetches the class of the receiver, checking if it was a **SmallInteger**. The second fetches the message selector to be sent. The method header cache then begins its operations concurrently with the execution of the microcode. Microinstruction 3 saves the PC in the currently active context. Microinstruction 4 saves the flags (*cflags*, the *ap* and *sp*) in the current context, increments *CCN* and *ccbase* and *hcbase* to point to a new context via the special operation *sendNewContext*. It also dispatches on a wide branch that checks that the method header cache hit, and the kind of method found. Microinstruction 5 checks the flags of the new context to see if it was free, and initializes the ap and sp via *sendInitCCPtrs*. Microinstruction 6 initializes the method field of the new context by reading it from word 2 of the method header cache entry. Microinstruction 7 computes the new PC and sends it to the IFU, and dispatches on the number of temporaries required by the new method. Assuming one temporary, microinstruction 8 will initialize it to Nil, and microinstruction 9 will dispatch on the next byte code. Instruction 8 could dispatch, but the IFU would not be ready, and this would cause microinstruction 8 to be stretched, and the microcode for the dummy macro instruction to be executed for microinstruction 9. It is therefore faster to not dispatch until the IFU is ready. A message send that creates *n* temps takes *max (9, n + 7)* microcycles. This depends on several conditions which are usually true: The receiver is not a **SmallInteger**, the method is in the method header cache, and the newly allocated context is free. Sends to **SmallIntegers** take an extra cycle.

```
1  A@sp#op1bar+1 Br#R0 aluZeroExtA memRead
          writeMhcacheClass trapBranchNotIntABF
          seqOpGoToD sendCode
2  A@hcbase#0+MethodOffset B#op2+OO+OMethLiterals
          aluAdd memRead writeMhcacheMsg
          seqOpGoToD sendSavePC
3  A@ccbase#0+PCOffset pcnext aluZeroExtB writeA
4  A@ccbase#0+FlagOffset flags aluZeroExtB
          writeSendNewContext trapBranchMHCacheFlags
          seqOpGoToD sendCheckType
5  A@hcbase#0+FlagOffset aluZeroExtA
          writeSendInitCCPtrs trapBranchFlagBits
          seqOpGoToD sendCheckNew
6  A@hcbase#0+MethodOffset MHCacheWord#2
          aluZeroExtB writeA
          seqOpGoToD sendNilTemps
7  A@hcbase#0+MethodOffset MHCacheLitCount4
          aluAdd writePCWord seqOpGoToD
          trapBranchMHCacheNTemps nilNTemps
8  A@sp#postinc Br#Rnil writeA seqOpGoToD nil0Temps
9  seqOpGoToD macro
```

Figure 5. Send Microcode

Another example of the use of the wide branching capability is to speed up the subscripting messages **at:** and **at:put:**. These messages request an indexable object to read or write an object reference at a specified location in the indexable object. Each class contains a word that specified whether that class is indexable, whether the indexable units are bytes or words, and whether the data in the object are OOPs or bits. It is still necessary to perform a message lookup when doing a subscripting message because a class can redefine **at:** and **at:put:**. Because of the importance of this, we added a fourth bit to the class description. This bit is set if this class or a superclass of it redefines **at:** or **at:put**. As a result, the byte codes which send the subscripting messages can check all 4 of these bits via a single 16-way branch, saving the 7 microcycles needed to do a message lookup and enter the builtin subscripting microcode through the normal control path. This method is also possible for other common messages, but only subscripting messages were found to be frequent enough to justify it, as was determined by using the statistics from [5] in the spreadsheet. Some messages to contexts can bypass method lookup by doing a check of the address space of the receiver.

5. Performance Modelling

Performance modelling of our architecture was an essential part of optimizing performance, taking place concurrently with architectural design. The dynamic frequencies of each byte code were calculated from [5] and entered in a spreadsheet calculator. [5] contains a large number of statistics derived from a large test of the system's performance. To quote from it:

> During this test, the browser, compiler, decompiler, and window system are exercised in every conceivable way. With full performance monitoring, the test covers millions of bytecodes and takes over six hours.

The result is shown in the appendix. The frequencies include only those byte codes that complete in the fastest possible manner. For example, the send-+ byte code frequency is only for send-+ byte codes that operate on **SmallIntegers**. Send-+ byte codes that perform a full send are included in the full send category. Another point to note is that both micro and macro branch delays are fully accounted for in this estimate.

As the architecture evolved, the microcode for each byte code type was sketched and the overall system performance was calculated by the spreadsheet calculator. The time was estimated in minor cycles, where a microcycle is 2 minor cycles if it does not reference memory, and 3 if it does. As we began with a conventional implementation of message sending requiring about 25 microcycles, the spreadsheet immediately pointed out that message sending was the bottleneck. Adding the method header cache and partial context initialization reduced this to an estimated 7 microcycles, with 9 microcycles as the eventual result. With our microcycle time of 136 ns., the current performance estimate is 1.9M bytecodes per second.

The spreadsheet indicates the relatively successful parts of the architecture, and those which still need work. About 45% of the byte codes execute in one cycle, and 15% execute in two cycles, which together use only 20% of the processor's cycles. Sends and returns, which comprise about 18% of the byte codes executed, use 40% of the processor's cycles. The byte codes causing context cache manipulation, although only 1% in frequency, are estimated as using 8% of the processor's cycles. This is based on the assumption that they cause a context to be transferred to or from main storage.

A few caveats are in order whenever performance estimates are made. Our spreadsheet does not include any allowance for memory cache misses or method header cache misses. It only partly accounts for context cache overhead due to copying hard contexts out of the context cache. Some allowance for copying contexts into the cache is made based on the assumption that byte codes invoking non-LIFO control cause one context to be copied in. The time estimate for "all other primitives" is also a guess based on average time predicted for the large number of other primitive operations.

A set of standard benchmarks [2] provide a method of comparing the speeds of various ST-80 systems. Figure 6 contains a comparison of several benchmarks. K means 1000 (not 1024). Part (a) gives some raw performance data for Swamp. The first column is the name of the benchmark. **3Plus4** tests the speed of integer arithmetic. **LoadTempNRef** tests the speed of pushing **SmallInteger** local variables onto the stack. **LoadTempRef** tests the same, using an object which is subject to garbage collection. **LoadTempRef** will be slower than **LoadTempNRef** in reference counting systems. **ActivationReturn** tests the speed of message sending. **ArrayAt** tests the speed of array indexing.

The second column gives the number of byte codes executed in the test. The third column gives the time for the Dorado, the standard against which ST-80 systems are usually compared, to execute the benchmark. The fourth column gives the time for Swamp to execute the benchmark. This is not simply a multiple of the microcycle time because of the inclusion of memory access time and cycle stretching. The fifth column gives the execution rate of the benchmark in bytecodes per second.

Part (b) compares the relative performance of Swamp and some other systems. The second column gives the relative speed of Swamp versus the Dorado, with numbers greater than one being faster than the Dorado. The third and fourth give the relative speeds of SOAR and Sword-32 compared to a Dorado. The last column gives the relative speed of PS/68020 compared to a Dorado [15]. PS/68020 is a Smalltalk implementation based on the ideas in [11] running on a 68020 microprocessor.

test	#inst	Dorado time	Swamp time	Swamp inst/sec
3Plus4	400K	.16	.095	4.2M
LoadTempNRef	400K	.28	.082	3.25M
LoadTempRef	400K	.40	.082	4.65M
ActivationReturn	344K	1.01	.126	2.6M
ArrayAt	80K	.19	.147	.54M

(a) Raw Swamp Performance

test	Swamp perf	SOAR perf	Sword-32 perf	PS/68020 perf
3Plus4	1.68	.58†	.89†	.66
LoadTempNRef	3.25	-	2.24†	.86
LoadTempRef	4.65	-	-	1.22
ActivationReturn	8.01	4.1	5.0	1.84
ArrayAt	1.29	.51	-	.72

(b) Performance Relative to Dorado

test	SOAR perf	Sword-32 perf	PS/68020 perf	Dorado perf
3Plus4	1.39†	.49†	.36	.26
LoadTempNRef	-	.63†	.23	.14
LoadTempRef	-	-	.22	.09
ActivationReturn	2.07	.57	.20	.05
ArrayAt	1.60	-	.49	.34

(c) Performance Relative to Swamp With Normalized Cycle Times

†: computed by us from published data

Figure 6. Benchmark Results

We believe that our spreadsheet is more accurate than estimates based on a few benchmarks. Using only a few micro benchmarks appears to inflate predicted performance. This is also supported by [5], which states: "Early investigative work on our system used the **testStandardTests** benchmarks until we noticed that the results bore little relation to statistics gathered from normal usage." For example, using the 5 standard benchmarks, our machine has a net performance of 3.2M bytecodes per second (MBCPS), and the Dorado has a predicted performance of .8MBCPS. Since the Dorado is observed to have a typical performance of .4MBCPS, use of a few benchmarks appears to inflate performance, which is consistent with our spreadsheet. The most accurate benchmark is believed to be **testActivationReturn**, in which the Dorado executes at .36MBCPS, close to its observed average throughput. Although Swamp executes this benchmark at 2.6MBCPS, this throughput would not be achieved in a real instruction mix because sends and returns have been sped up relative to other parts of the implementation.

Architectural analyses should include the effects of different implementation technologies to compare the relative merits of architecture in a technology independent manner. Because most processor's cycle times are close to the time required to perform a read and write into a register file, we attempt to compare the performance of the various architectures and instruction sets by comparing the number of cycles required to execute a given program. Part (c) compares the performance of all implementations assuming that the implementation used the same 136ns cycle time that Swamp does. Each column gives the performance of that implementation relative to that of Swamp. In terms of normalized cycle times, SOAR has better performance, ranging from 39% to 107% faster for the same cycle time. Sword-32 is slower than Swamp, as is PS/68020. Although both PS and SOAR use native code, PS is much slower than SOAR on a normalized cycle basis, presumably due to the lack of ST-80 specific hardware in the 68020 architecture. The Dorado is by far the slowest architecture.

Another use of the spreadsheet is to estimate memory bandwidth. The average ST-80VM byte code is less than 1.1 bytes long, requiring about 2.1MB/s or .5MW/s of memory bandwidth. Another study using the spreadsheet shows that the average byte code makes .58 memory references, for a bandwidth of 1.1MW/s. The total bandwidth of 1.6MW/s is small for a machine of 1.9M byte codes per second performance. SOAR [3], for example, has an instruction bandwidth of 1.8MW/s even though it executes the three benchmarks compared from .35 to .51 times as fast as Swamp. A SOAR with a 136 ns cycle would have a bandwidth of 7.4MW/S, much larger than its relative performance would justify. The caveats for this number are stronger than before, however, as much of the total memory bandwidth appears to be used by features that we have only poor estimates of.

Adherence to the ST-80VM seems to incur a performance penalty. It is worthwhile seeing how small the performance penalty due to the use of a byte coded instruction set can be made. We would like to execute as much of the source level program as possible in each register read plus write cycle, provided that this does not lengthen the cycle. This suggests the use of a multiple address architecture. A three address version of Swamp would run at the same speed, and hopefully accomplish more work per cycle. We investigated the performance implications of such an architecture. It could speed up some instructions that currently take 2 microcycles to execute, and increase the fraction of instructions executed in one cycle to 56%, but this would increase performance by less than 3%. With the current ST-80VM, a three address architecture is not useful enough to justify the cost.

We believe that a significant component of the penalty is due to the single address nature of the ST-80VM. We believe that this significantly reduces performance for Swamp. For example, the expression **a < b ifTrue: [ ] ifFalse: [ ]** compiles into two push instructions, a send-<, and a branch instruction, requiring a total of 7 microcycles. Implementing a branch-on-< instruction would decrease the instruction count by one, and the number of microcycles by 2-3. As another example, the code **a + b** produces two pushes and a send-+ instruction requiring 3 microcycles. If both operands are temps, a three address version of Swamp could perform all of the actions in one microcycle.

We believe that an alternate instruction byte coded instruction set might offer a significant performance advantage. One reason for this is the ability to use a more efficient multiple address instruction set. Future work will look at some approaches for reducing this penalty. First, instruction set changes can be made to the ST-80VM to introduce multiple address or multiple operation instructions. An example is a push-temp-send-+ instruction, which pushes a temp and then sends the message +. This replaces two ST-80VM instructions by one. The current version of Swamp can execute this in one microcycle.

The instruction set design must be done carefully to avoid instructions that may cause a full message send in the middle. For example, the branch-on-< instruction is not feasible, because sending < might require a full message send. Upon return, the branch should be performed, however, since this corresponds to part of the instruction, there is no program counter that corresponds to this point in the execution of the branch-on-< instruction. One approach to avoid this in which the IFU recognizes instruction pairs dynamically and generates pseudo instructions corresponding to the pair will be investigated. If a full message send is required, then the IFU can backup and process the instructions one at a time. Another approach is to encode the branch-on-< instruction so the last part of the instruction contains a correct branch-on-true instruction.

SOAR [3] [16] contains aspects of both of these improvements, which we believe is a primary reason for its good perfomance. We believe that a multiple address byte coded instruction set can be designed to obtain comparable performance with lower instruction bandwidth.

The other significant performance limit is due to the cost of message sending. Further simplification of the actions performed during message sending and added hardware may reduce the 40% of the processor cycles used simply for message sending and returns. One minor improvement to allow the method header cache to calculate a new PC and send it directly to the IFU would speed up message sending by 33%, and increase the speed of the **testActivationReturn** benchmark by 11.5%. Its omission was an oversight, as it would increase the complexity of the machine by less than 2%.

In spite of this minor problem, message sending is one of the relatively successful parts of the processor. The benchmarks clearly demonstrate the relative performance increase gained by the resources devoted to message sending and returns. Message sending and returns is the one area where Swamp performance is most dramatically improved over the Dorado implementation.

6. Implementation

A processor based upon the architecture described is being constructed. It comprises about 550 chips, with 340 in the CPU and the remainder in the memory and display. A workstation is used to load the microcode and perform disk IO. The processor uses 45 ns 2K × 8 static RAMs for all memories in the CPU. A master clock with a 34ns period controls a small machine which generates the processor clock. Note that the minor cycles in the spread sheet are two master clock cycles[1]. A microcycle is a minimum of 4 master clock cycles, for a time of 136ns. The shortest possible clock cycle capable of multiplexing two addresses and performing a read and write into the RAMs used is 112ns, so there is not much room for improvement before hitting this limit. The first 3 master clock cycles of each microcycle are used to allow the IFU to use the memory cache. A memory reference that results in a cache hit causes the microcycle to be stretched to 6 master clock cycles. Cache misses take 5 more master clock cycles. A wrongly predicted branch requires stretching the microcycle by 4 master clock cycles. Memory references complete in the microcycle in which they are initiated.

A *C* compiler which compiles to microcode is used for dynamically infrequent code and first implementations of complex routines such as BitBlt and the garbage collector. Our implementation of BitBlt is based on the ideas in [13], and generates optimal code for the operation using a small number of instructions defined in a second instruction set.

A 1Khz timer interrupt causes the execution of a small routine which checks the amount of storage free, and invokes the garbage collector if this falls below the amount that can be allocated by the processor in 1ms. This means that all code that allocates a small bounded size of storage does not need to check if there is free storage.

1 This clock is double the minor cycle frequency to allow cache accesses to take 3 × 34ns clock cycles for 102ns, rather than 2 × 68ns for 136ns This only speeds up execution in the case of instruction fetch, and so improves performance by .8%, which is probably not worth the difficulties encountered squeezing the logic functions of the processor clock generator into a 34ns cycle.

There are two strategies for reducing the CPU's cost and complexity from its present 350 chips through the use of VLSI. Gate arrays could be used to integrate all logic other than RAM's and speed up the CPU somewhat. Although the CPU's complexity of about 15000 gates suggests taht a single chip could be used to implement it, the roughly 300 pins that would be required is excessive. Four arrays ranging from 3000 to 6000 gate complexity and reasonable pin counts could be used. The total CPU complexity would be reduced to 40-60 chips depending on RAM density. An implementation based on a full custom chip containing the microcode ROM and context cache RAM would increase its speed further, and reduce the CPU to 1 custom chip, 10 RAMs, and 8 bus interface chips. An advanced process, 1.5μ or better, would be required to integrate the 300Kbits of ROM and 12Kbits of RAM as well as the data paths.

7. Conclusions

We have discussed a high speed processor for ST-80 capable of 1.9M bytecode per second performance. This processor executes a byte coded instruction set. This processor handles the dynamically frequent cases of ST-80 with circuitry that allows it to execute rapidly. Wide branching in microcode allows the execution of the special case to take place concurrently with the checking of the assumptions that it is a special case.

Swamp performs faster than any other ST-80 processor known to us. It ranges from 44% to 89% faster than its closest competitor, even though Swamp's microcycle time is 9% longer. It clearly demonstrates that byte coded implementations of the Smalltalk-80 programming language can obtain good performance it sufficient attention is paid to optimizing the handling of the important cases. Performance modelling concurrent with architectural design is a useful tool for improving performance. A simple spreadsheet can be used to allow rapid estimation of the performance implications of architectural features.

Although Swamp demonstrates that machines designed to execute byte coded instruction sets can obtain good performance, some limits of the ST-80VM have been exposed. The Swamp processor contains sufficient resources to perform more operations than are specified in most ST-80VM instructions. Because Swamp can already execute most instructions in a single cycle, we do not believe that significantly better performance can be obtained with the current ST-80VM. These limits have not previously been apparent because the performance of previous implementations has been limited by other factors. The problems do not suggest that the ST-80VM be scrapped, but modified. We believe that further speed improvements will require a redesign of the ST80-VM to include multiple address instructions. Future work will concentrate on increasing the speed of Swamp while keeping the same low instruction bandwidth provided by byte coded instruction sets.

8. References

[1] Adele Goldberg and David Robson, *Smalltalk-80 The Language and its Implementation*, Addison Wesley, 1983

[2] Kim McCall, *The Smalltalk-80 Benchmarks*, in *Smalltalk-80 Bits of History, Words of Advice*, Glenn Krasner (ed.), Adison Wesley, 1983

[3] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson, *Architecture of SOAR: Smalltalk on a RISC*, Proceedings of the Eleventh Annual Symposium on Computer Architecture, 1984

[4] Kenneth A. Pier, *A Retrospective on the Dorado*, Proceedings of the Tenth Annual Symposium on Computer Architecture, 1983

[5] Joseph R Falcone, *The Analysis of the Smalltalk-80 System at Hewlett-Packard*, in *Smalltalk-80 Bits of History, Words of Advice*, Glenn Krasner (ed.), Adison Wesley, 1983

[6] D. A. Patterson and C.H. Sequin, *A VLSI RISC*, Computer, 15,9, September 1982

[7] Henry Lieberman and Carl Hewitt, *A Real-Time Garbage Collector Based on the Lifetime of Objects*, CACM 26,6, June 1983

[8] David Ungar, *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm*, Software Engineering Notes, 9,3, May 1984

[9] Glenn Krasner, David Ungar, and Michael Malcolm, *About Become*, in *Smalltalk-80 Newsletter 4*, Xerox Corp, Sept 1984.

[10] Norihisa Suzuki and Minoru Terada, *Creating Efficient Systems for Object Oriented Languages*, ACM Conference on Principles of Programming Languages, 1984

[11] L. Peter Deutsch and Allan M Schiffman, *Efficient Implementation of the Smalltalk-80 System*, ACM conference on Principles of Programming Languages, 1984

[12] Advanced Micro Devices, *29300 Family Handbook*, Advanced Micro Devices, April, 1985

[13] Rob Pike and Bart Locanthi, *Hardware/Software Tradeoffs for Bitmap Graphics in the Blit*, Software Practice and Experience, Vol 15(2), Feb. 1985

[14] Norihisa Suzuki, Koichi Kubota, and Takashi Aoki, *Sword32: A Bytecode Emulating Microprocessor for Object-Oriented Languages*, in Proceedings of the International Conference on Fifth Generation Computer Systems, 1984

[15] *Xerox PARC/SCL's PS/68020*, in Smalltalk-80 Newsletter 7

[16] David M. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, Report No. UCB/CSD 86/287, March 1986, University of California at Berkeley

[17] David M. Ungar and David A. Patterson, *Berkeley Smalltalk: Who knows Where the Time Goes?*, in *Smalltalk-80: Bits of History, Words of Advice*, Ed. Glenn Krasner,

[18] David A. Moon, *Architecture of the Symbolics 3600*, in Proceedings of the Twelfth Annual Symposium on Computer Architecture, 1985

9. Appendix

Performance Spreadsheet

Instruction	%	#minor cycles	contribution
full sends, prim. failure	6.99	24.00	1.68
Pseudo primitive send	3.36	18.00	0.60
send +,-,and,or	5.21	2.00	0.10
send <,>,<=,>=,=,~=	3.72	5.00	0.19
send @	0.56	25.00	0.14
send *,//,\\,/	0.35	40.00	0.14
send bitShift:	0.24	6.00	0.01
<Object> at:	2.12	14.00	0.30
<Object> ==	1.63	5.00	0.08
<Blockcontext> value	0.66	50.00	0.33
<Object> size	0.64	19.00	0.12
<String> at:	0.47	18.00	0.08
<Object> at:put:	0.47	14.00	0.07
<ContextPart> blockCopy:	0.36	75.00	0.27
<Behavior> new	0.35	50.00	0.18
<Object> class	0.23	5.00	0.01
<BitBlt> copyBits	0.20	0.00	0.00
all other primitive sends	1.06	30.00	0.32
push temporary	19.72	2.00	0.39
push special constant	12.70	2.00	0.25
push active context	0.42	50.00	0.21
push receiver variable	7.63	5.00	0.38
push extended	1.32	2.97	0.04
push literal variable	1.31	8.00	0.10
push literal constant	1.27	5.00	0.06
store temporary	3.94	2.00	0.08
pop	2.41	2.00	0.05
store receiver variable	1.76	5.00	0.09
pop and store extended	1.51	2.87	0.04
store extended	1.17	2.87	0.03
pop and branch on false	4.25	7.00	0.30
pop and jump on false	1.94	7.00	0.14
jump	1.84	6.00	0.11
branch	0.56	6.00	0.03
return tos from method	4.93	10.00	0.49
return self	1.43	10.00	0.14
return tos from block	0.62	30.00	0.19
return false	0.43	10.00	0.04
return true	0.20	10.00	0.02
return nil	0.00	10.00	0.00
TOTAL	99.98		7.68
minor cycles per bytecode	7.68		
minor cycle (ns)	68.00		
bytecodes per second	1870677.48		

Notes: A microcycle is 2 or 3 minor cycles. Times for instructions that have more than one possible time (eg, send-<, branch-on-false) are averages of all possibilities. Non-integral times are weighted averages of all possible times for the instruction. Bitblt is ignored.