PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

Sword32: A Bytecode Emulating Microprocessor for Object-Oriented Languages

Norihisa Suzuki, Koichi Kubota, and Takashi Aoki

Department of Mathematical Engineering
The University of Tokyo

ABSTRACT

Sword32 is a large, 32-bit microprocessor with an internal, 4K-word microstore, a large stack, and an instruction dispatch mechanism to efficiently emulate bytecode instruction sets. The microarchitecture is optimized for Smalltalk-80 so that the emulator for the Smalltalk-80 bytecodes can execute 1.4 million bytecodes per second in our preliminary performance evaluation.

1. Introduction and Motivation

Since the spring of 1982 we have been studying whether a late-bound, object-oriented language like Smalltalk-80 should be used as a principal programming language for our research in computer system software, VLSI design, and computer-assisted instruction. We concluded that Smalltalk-80 system has a number of attractive features including the fact that a large collection of useful software to support programming environments can be obtained for free, but the current implementation is inefficient and furthermore runs on very limited kinds of computers. Alternatives are UNIX and LISP, but the versions of software that are available to universities lack the sophisticated display-oriented programming systems.

Therefore, we decided to invest our research efforts in improving the performance of Smalltalk system, and devised new techniques for compiling and creating run-time systems for such a late-bound, object-oriented language (Suzuki and Terada 1984). We came up with a number of new techniques that make Smalltalk-80 much faster, and created two Smalltalk systems to try out our ideas on a standard microprocessor MC68000 (Suzuki et al. 1984). The attempt was quite satisfactory, and the performance of our systems were in line with other implementations (Deutsch and Schiffman 1984, Wirfs-Brock 1983, Unger 1984). However, we thought that we could create a system faster by an order of magnitude if we build a special purpose microprocessor with powerful features designed to execute Smalltalk. Therefore, we started to design a general-purpose, bytecode-emulating microprocessor tuned to execute Smalltalk bytecodes particularly efficiently in the summer of 1983.

Smalltalk-80 is the trademark of Xerox Corp.

While we could have written our Smalltalk system in microcode on Perq or Symbolics 3600, building our own VLSI microprocessor allows us to explore several new architectural ideas that should result in a system that is significantly faster than one implemented on Perq or Symbolics 3600. Furthermore, as one of our research goal is to design and build our own computer systems, particularly ones based on VLSI microprocessors, such an undertaking seems appropriate.

Nevertheless, because this is the first microprocessor we have designed, we have kept our aspirations modest. We have eliminated many features that might have improved the speed or flexibility, but at the cost of increased architectural and layout complexity. Yet, as we wrote the microcode for Smalltalk-80 (Suzuki 1983), we discovered that it may run significantly faster than many previous implementations, even those on sophisticated bytecode emulation computers built using ECL (Deutsch 1983). We also have kept our software aspirations modest. We made the instruction set of Sword32 to be completely compatible with the Smalltalk-80 Virtual Machine except that Sword32 uses 32-bit object pointers with slightly different encodings of small integers. This is because we did not want to spend our efforts in creating a special system in order for Smalltalk to run on Sword32. Furthermore, since we have no control over the future systems, we decided to conform to the standard format so that we need not keep creating versions of the Virtual Image.

We completed the first architectural design of the microprocessor in the autumn of 1983 and have designed an ALU in NMOS technology. This was a 16-bit microprocessor, which we called Sword (Suzuki et al. 1983). We first designed a 16-bit microprocessor, since we intended to run the Smalltalk-80 Virtual Image as is. We did not have enough experience to know the limitations of the 16-bit system and to know how to rewrite it for 32-bit computer. However, after we created the first virtual machine on SUN Workstation, we knew immediately that the size of the object memory is too small to implement any serious application programs. So we redesigned a virtual machine to have 32-bit object pointers, and have implemented on SUN workstation (Suzuki et al. 1984). From this experience and from the fact that we were able to have an access to fast CMOS technology, we decided to redesign the whole system to have 32-bit data paths. This is the Sword32 microprocessor, which we will describe in this paper.

Table 1. Comparison of Smalltalk Systems

System	Dolphin	SUN	SOAR	Sword32
Implementation	Microcode emulator	Dynamic compiler	Static compiler	Microcode emulator
Object Pointer	16 bit	24 bit	28 bit	32 bit
Object Table	Yes	Yes	No	Yes
Garbage Collector	Reference Counting	Transaction	Generation Scavenger	Transaction
Instruction	bytecode	native code	native code	bytecode
Method Search	Method cache	Inline cache	Inline cache	Method cache
Context Allocation	heap	stack	stack	stack
Primitives	microcode + BCPL	assembler	C(?)	microcode + C
Multiprocessor	No	No	No	Yes

2. Comparison with Other Systems

We compare architectural features of several distinct implementations of Smalltalk-80 (Table 1).

The systems compared in this table are Dolphin sold by Xerox, SUN Workstation system (Deutsch and Schiffman 1984), SOAR microprocessor system implemented at UC Berkeley (Unger et al. 1984), and Sword32.

Dolphin system is a microcoded emulator of the Smalltalk-80 Virtual Machine implemented in a way very close to the book. SUN implementation is the fastest running system besides Dorado implementation; it dynamically compiles bytecodes into native codes. Throughout the rest of the paper, we mean Deutsch and Schiffman's implementation if we mention SUN implementation without any qualification. SOAR is a reduced instruction set microprocessor tailored for Smalltalk. All the object codes are native codes. Sword32 is a microcoded emulator of the Smalltalk-80 Virtual Machine just like Dolphin, but the implementation techniques are close to those of SUN.

Object pointers are unique identifiers of objects. In Dolphin an object pointer is an index of an object table, whose entry contains the real address of the object, the reference count, and several flags. The object pointer of the Sword32 is the absolute address of the object table entry. The object pointer of SOAR is the absolute address of the object, so it does not have an object table. We do not know the details of SUN implementation.

Dolphin employs reference counting garbage collection. Since this was a major performance bottleneck, the SUN implementation uses Deutsch-Bobrow transaction garbage collection (Deutsch and Bobrow 1976). We also use transaction garbage collection in Sword32. SOAR uses a generation scavenger, which is a variant of copying garbage collector.

Both Dolphin and Sword32 directly execute Smalltalk-80 bytecodes, which are very high-level instruction set for stack machines. Therefore, the decompilation can be done easily and the various system software does not have to be rewritten in order to run on Sword32. The programs of SUN runs 68000 native code but the compilation is done dynamically, so that it still retains the compatibility with the system software. SOAR uses its own instruction set so that system software has to be rewritten. Smalltalk-80 is a late-bound language. The link between a message and a method is determined at run-time every time a message sent. In order to speed up the linking, various implementations use some kind of cache. Dolphin uses a method cache, which is a global hash table where the keys are the class of the receiver and the message selector, and the values are the method and the primitive index. The table size is usually from 256 to 2048 entries, but the hit rate is quite high, 95% (Conroy 1983). SUN implementation uses an inline cache. After each message send instruction, the class and the address of the object code for the method are stored. If the receiver's class is the same as the class stored inline, the direct subroutine call is done. SOAR uses the same technique. Sword32 uses the method cache; in order to speed up the linking, we store the absolute address of the instruction, that is the address of the first bytecode of the method if the method is implemented by Smalltalk-80 code, and the microcode address if the method is implemented by a primitive, the absolute address of the method, and the number of temporaries other than the parameters.

Contexts are the objects that store information necessary for method activation; they are usually called stack frames in other programming languages. In Dolphin contexts are the first class objects; they are allocated and deallocated for all the method activations and returns, and even reference counted. This inefficiency has been eliminated in SUN by allocating contexts on the stack as long as the contexts are not retained. SOAR and Sword32 use the same technique.

We provide a hook to create a high performance multiprocessor system, in which each processor has its own cache and caches are connected to a common memory bus and share a main memory. A microinstruction is provided which can implement a test-and-set instruction for such a system.

3. Overview of the System Design

Most of our architectural features are driven by our knowledge of inefficiencies of Dolphin system. The bottlenecks are reference counting garbage collection, method search, and the context allocation in the heap.

3.1. Reference Counting

In order to create a personal computer system that provides real-time response for interaction, we have to use a real-time garbage collector. Therefore, the Dolphin implementation employs a reference-counting garbage collector. However, this consumes a substantial amount of computation time. According to our calculation 80% of the time spent for pushes and pops is spent for reference counting.

There are two algorithms for real-time garbage collection that lead to successful implementations of Smalltalk-80 (Deutsch and Schiffman 1984, Unger 1984). We implemented Deutsch-Bobrow transaction-based garbage collection with a satisfactory result in our MC68000 implementation (Suzuki et al. 1984). Therefore, we use this algorithm for Sword32.

There is no special hardware support for transaction garbage collector except that most of the program will be written in microcode.

3.2. Object Representations

References to objects are made through object pointers just like in Dolphin implementation. Object pointers are absolute addresses of object table entries, in which the absolute addresses of the objects and reference counts are stored (Figure 1).

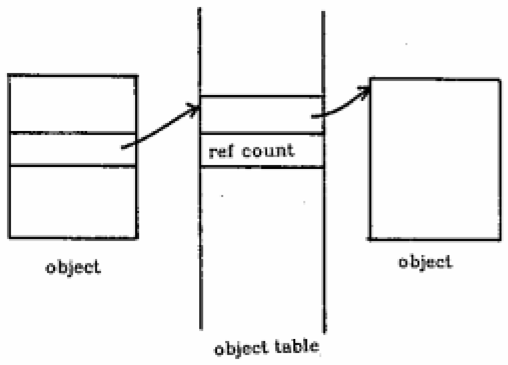

Fig.1. Object pointers are absolute addresses of entries in the object table.

Both Dolphin and SUN implementations use object pointers in order to simplify compaction. Furthermore, there is a method called become: which swaps objects that are pointed by two object pointers. This is difficult to implement without the object table. SmallIntegers are encoded in object pointers; if the least significant bit is zero, the rest of 31 bits denote SmallIntegers. We chose this form, because ordinary arithmetic on SmallIntegers can be performed by an ALU designed to work for unsigned integers. Therefore, the least significant three bits of object pointers are zero, zero, and one, if they are indexes to an object table. (Figure 2)

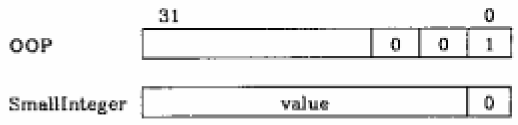

Fig.2. The formats of object pointers.

In order to accommodate special encodings of object pointers and perform type checks, we have unique memory operations. Memory operations of Sword32 are fetch, store, fetch byte, and store byte. Fetch and store read and write aligned four bytes, and fetch byte and store byte read and write any byte in the memory. For fetch and store the memory operation is started if the least significant bit in MAR is one, and the memory address put out from the chip has zero as the least significant bit. If the bit is zero, no operation is performed but the least significant bit of the address of the next microinstruction is or'ed with one. So fetch and store operations always cause conditional branches.

3.3. Method Lookup

In order to speed up the message linking most of Smalltalk-80 implementations use cache. Dolphin implementation use a method cache, each entry of which consists of selector, class, method, and primitive index. By a proper choice of a hash function, the hit rate can be quite high, as much as 95%, but still the time to compare two entries, selector and class, then decode the locations of bytecodes is substantial. Deutsch and Schiffman's implementation used an inline cache; after each send instruction the class and the address of the instruction of the method last called from the particular message send is stored. The link is fast but it rewrites and expands codes. This method is also used for SOAR. We also use a global method cache, but we store information that enable faster method activation; they are the absolute address of the method, the address of the instruction, and the number of temporaries other than the arguments. The format of an entry of the method cache is shown in Figure 3.

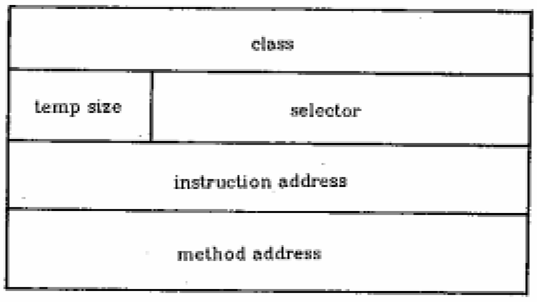

Fig.3. Each entry in the method cache occupies four words.

The address of the instruction is encoded in such a way that if the most significant bit is zero it is the absolute address of the bytecode, and if the bit is one, the least significant 12 bits are microcode address of the primitive. (Figure 4)

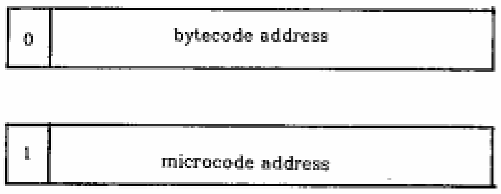

Fig.4. Format of an instruction address.

The number of temporaries other than the arguments are the number of NIL's that have to be pushed on the stack.

3.4. Stack Organization

Since procedures are invoked very frequently in Smalltalk-80, hardware support for procedure calls is important. Sword32 has a large (128 words) on-chip stack and 32 general purpose registers. We would use these registers to cache the top part of the contexts.

There are four stack pointers FP, CP, TP, SP; the data are accessed from the parts pointed by these pointers. In the Smalltalk-80 emulator, FP is a context base pointer, TP is the work pointer to retrieve local variables in the active context, and SP is the top of the stack pointer. FP itself is organized as an eight-level stack. The current frame pointer is pointed by CP. The first 16 words of the general purpose registers can also be used as two eight-level stacks, whose pointer is also CP. These two stacks can be accessed through CP, if we access the lowest address register in each stack. Thus, if the value of CP is 5 and if we fetch register 0, we actually obtain the value of register 5. If we fetch register 8, we actually fetch register 13. We can still access other registers at random. If we fetch register 2, we really fetch register 2.

The stack organization for emulating Smalltalk-80 is shown in Figure 5.

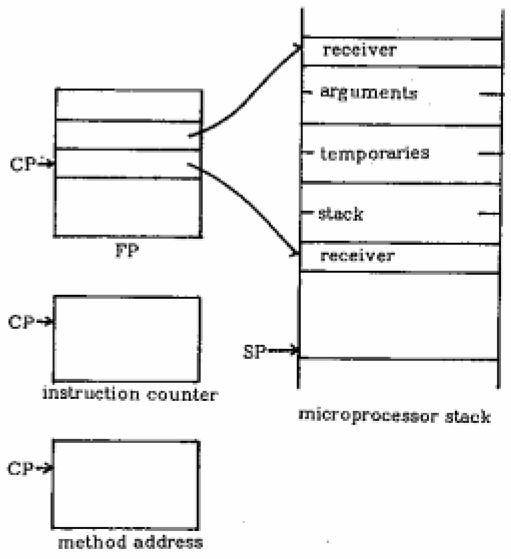

Fig.5. Stack organization for Smalltalk-80 emulator.

At most 8 contexts can be stored in the stack. FP points to the beginning of each context. Two 8-word stacks store the bytecode instruction counter and the method address. Each context occupies a stack region bounded by two consecutive FP's.

The stack is organized this way so that there is no need for copying arguments from the sender to the method as is done in Dolphin. The receiver and the arguments are pushed in the evaluation stack of the active context, and the message is invoked. Then the top most several locations defined by each bytecode become the bottom part of the newly activated context.

4. Microarchitecture

Our guiding principle throughout architectural design was to keep the microprocessor simple. We would supply just enough hardware, so that most of the complex jobs could be done efficiently with microcode. We encountered a number of occasions where if we added some hardware feature, a particular microprogram or a particular bytecode would run faster. However, in most cases we found that either a different microcoding for the same job would give us the same speedup, or that the part that was speeded up was executed relatively infrequently so that the hardware addition would not result in a significant overall performance improvement. Furthermore, adding hardware often has a negative effect on the overall speed of the machine because the basic machine cycle has to be slowed down to accommodate the addition. Therefore we were very careful whenever we were tempted to add new hardware features.

Probably the most successful practice in the design of this microprocessor was that we started to write microcode within a week after the project was started. The first version of the microcode was written within two weeks. This helped immensely in tuning the machine; many features were found to be unnecessary, and many new features were added to improve the performance. Another successful practice was that we wrote a microassembler very early with only a couple of days efforts. This was probably made possible because we used LISP. The microassembler discovered many programming errors that tried to assign several different values to microinstruction fields. This resulted in microprograms that were more packed than logically possible. Without the error detection the machine would have been designed under the false assumption that efficient codings can be done under the architecture.

Some of the big surprises were that we thought, at the beginning, we need a special hardware for handling small objects such as integers that are encoded in the pointer fields. However, after writing the microcode we realized that we did not need any special hardware for treating encoded integers. All we are using are masking, shift, and standard arithmetic hardware. Another hardware which we dropped is a barrel shifter. A barrel shifter may be useful in Smalltalk-80 execution in three ways: decoding the bytecode, decoding the data, and executing shift bytecode. However, in all cases microprograms without a barrel shifter perform just as good or better than microprograms with a barrel shifter.

On the other hand we decided to include special hardware for bytecode dispatch after we wrote microcode. We first tried to use the general dispatch mechanism for the bytecode dispatch, but that uses one field (F2 field) of an instruction, and needs one instruction before the next bytecode is executed since the general dispatching requires one instruction delay. This additional hardware reduced the length of most microprograms by 1 or 2.

By continuously tuning the architecture we came up with a quite efficient bytecode emulator for Smalltalk. We describe the design strategies of the components in the following.

4.1. Bytecode Fetching

Efficient bytecode fetch machinery is important, but we did not support any sophisticated instruction fetch unit such as the one found in Dorado (Lampson et al. 1981). What we are providing are hardware for fast bytecode dispatching, automatic fetching of bytecode, process switch, and dynamic switching of bytecode dispatch tables.

As explained previously, there is no delay between the microinstruction to signal the end of a bytecode execution and the first instruction of the next bytecode; if the microinstruction is a branch to zero, the next instruction executed is the first instruction of the next bytecode. Furthermore, this does not use a microinstruction field. There is a 32-bit instruction buffer; if the bytecode read is the least significant byte of the buffer, the buffer is filled automatically by issuing a fetch. It is the responsibility of the microprogrammer to make sure that the bytecode fetch does not occur while a memory operation is in progress.

The process switch requirement of internal or external reasons are reported by setting a ProcessSwitch flag by a microinstruction. Then at the bytecode dispatch time, if this flag is set the control transfers to a fixed microprogram address to perform the process switch. The bytecode instruction counter does not increment, since the program may decide not to perform the process switch and continue to execute from the original process.

Another valuable mechanism is the two sets of bytecode dispatch tables, and the fast, dynamic switching of microcode dispatch tables. We cache top most contexts in the internal stack. It is, however, costly to maintain the top most context, which is called an active context, to be always in the stack at each bytecode dispatch. So at the beginning of each bytecode execution the active context may be on the stack or in the heap. Since most of the bytecodes and primitives behave very differently according to whether the active context is on the stack or in the heap, the interpreter has to know which state the machine is in at the beginning of each bytecode. However, we would not like to check by microcode the state of the contexts at the beginning of each bytecode; instead, we have two sets of microcode according to the states of the active context and switch among them, thus speeding up most of the bytecodes by two cycles.

4.2. I/O

Unlike Alto (Thacker et al. 1979) or Dorado (Lampson et al. 1981), we assume that there will be another microprocessor (MC68000) to perform most of the I/O work, such as disk I/O, raster operations, and keyboard I/O.

The input and output mechanisms of Sword32 resemble those of the Smalltalk-80 Virtual Machine. There is one input queue implemented as a cyclic buffer in the main memory. When some input events such as a keyboard depress and a mouse movement occur, it is detected by MC68000. The input is decoded and the input data is added to the queue, and MC68000 notifies Sword32 by asserting PWR. Sword32 increments the input semaphore, asserts PWRAck, and resumes the bytecode emulation.

4.3. Microsequencing

Microprogrammable microprocessors often use microsequencer to eliminate address fields in microcodes. In Sword32 we decided to use a next field, because a large portion of the microprograms are shared and there are a number of placement constraints so that without a next field explicit branch instructions would be very frequent.

4.4. Pipeline Organization

The pipeline organization of instruction decoding and execution depends on how we organize the data paths structures. We considered two approaches for the pipeline organization. One approach is to organize like Dorado (Lampson et al. 1981) where the basic machine operation is to read an accumulator and a data word from the register file, perform an operation, and store the result back into the register file. This organization requires three-stage pipeline and a by-pass circuit in order to attain maximum speed: a stage to read a microcode, a stage to read two registers and put data into ALU, and a stage to get result from ALU and write it into a register. Since the write register cycle of an instruction is overlapped with the read register cycle of the next instruction, it must be possible to read and write different locations in the register file in one cycle. Another organization is used in the Alto (Thacker et al. 1981) in which the pipeline has two stages: one stage that reads an instruction, and another stage that either reads one data word from the register file, performs an operation and stores the result in special latches, or else gets a data word from the latches and stores it into the register file.

The Dorado approach is suited if the time that takes to read and write the register file is short compared with the ALU operation time. In particular, this approach requires to read from a register file in the first half of the cycle and write to the register file at the second half of the cycle; the time to read and then write the register file should be approximately equal to the time to perform the ALU operation. On the other hand the Alto approach is suited if the register file read and write times are longer than the ALU operation time. Since in MOS VLSI register files are implemented using (in our case, static) memory arrays, and read and write times are relatively large compared with the ALU operation time, we adopted the Alto approach for the pipeline organization.

4.5. Data Paths

Figure 6 shows the logical organization of data paths, latches, and registers in the Sword32 microprocessor.

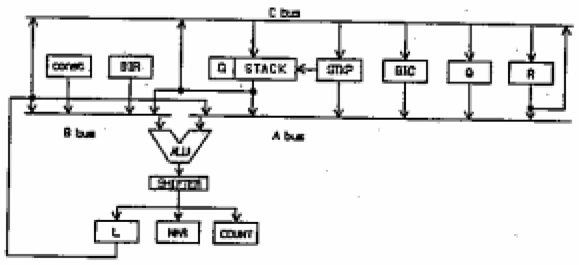

Fig.6. The logical organization of data paths.

The data paths in Sword32 are 32 bits wide. Even though the standard Smalltalk-80 Virtual Image is built on 16-bit data path computers, we adopted to have 32-bit data paths, because the large object space is inevitable if Smalltalk is going to be used for serious applications. The actual layout of the data paths are more orderly. (Figure 7)

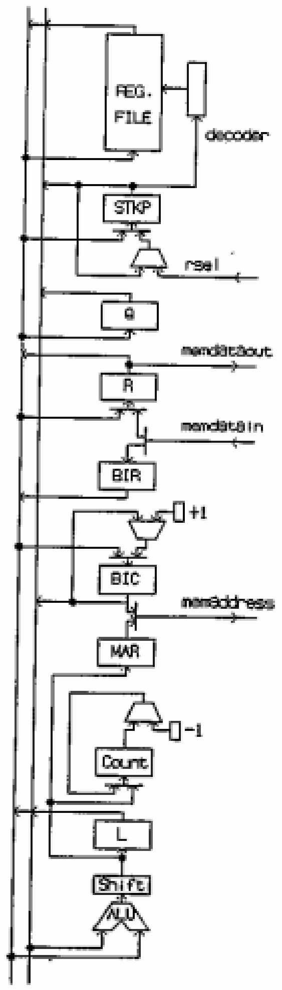

Fig.7. The diagram of the data paths that closely correlates the physical layout.

4.5.1. Register File

The largest overall performance gain comes from having many internal registers. They are used for various purposes--stack, context cache, or cache of the real addresses of frequently accessed objects. If we organized these registers into separate groups, the number of busses and the number of ports to the busses would be very large. Therefore, we put as many registers as possible into one large register file so that the number of separate ports and wires in the entire chip is small.

The register file is 160 words of 32 bits each. The first 128 words are used as a stack that can be accessed through the stack pointers. The other 32 words are general-purpose registers. The first 16 words can be used as general-purpose registers or two stacks of 8 words each. They are accessed either through RSEL field of a microinstruction or through a stack pointer CP.

4.5.2. Stack Pointer Manipulation

The four stack pointers can be manipulated in two ways. First, each stack pointer can be a bus source. Therefore, in one cycle a stack pointer can be read onto the bus and subjected to an ALU operation, then, in the next cycle, it can be written back to the stack pointer register. However, most of the stack pointer operations increment or decrement a stack pointer by some small constants. So, there is a second mechanism for altering a stack pointer in which a special adder is used to add or subtract small amounts from a stack pointer in one cycle; one can read from or write to the stack using the value of a stack pointer at the beginning of the cycle. In the same cycle one can modify that stack pointer.

4.5.3. Temporary Registers

L, R, and Q are temporary registers to store the intermediate results. Q can be shifted left or right by one bit to be used for multiplication and division. In Smalltalk-80 emulator R contains the top of the stack and SP points to the second from the top of the stack.

4.5.4. External Memory Interface Registers

The micromachine interfaces with the memory system through MAR, and the temporary register R. When writing to memory, R must be loaded at least a cycle before the store operation starts. MAR is loaded by store instruction and the memory operation is started. When reading from memory, MAR is loaded by fetch instruction and the data comes into R two cycles later. If a reference is made to R a cycle after the fetch operation, the instruction is held until the data transfer is completed.

4.5.5. Shifters

As was explained earlier, we did not include a barrel shifter. Instead we have a byte swapper, and a one-bit shifter. Both shifters operate on the result of the ALU before the data is loaded to L, Count, or MAR. The byte swapper transposes the 8-bit halves of the least significant 16-bit result of the ALU.

4.6. Control System

Microprograms are executed continuously at a rate of one microinstruction per machine cycle. There are six different ways to choose the next microinstruction.

MPC is the program counter of the microinstruction. This value is set at the end of the microcode read stage from several sources. The sources of MPC are, NEXT field of the previous instruction, 1024 when a wake-up request signal is detected, 1025 if a process switch is waiting, RETURN register when it is returning from a micro-subroutine, INTRETURN register when it is returning from interrupt processing, the logical disjunction of Next field and Abus if dispatching, and BIR when it dispatches to a new bytecode.

The order for granting the sources is: 1024 is the highest priority, 1025, RETURN, INTRETURN, dispatch, and IR should be mutually exclusive and comes next, and finally NEXT field.

5. Performance

Performance of Sword32 is measured using the standard benchmark (McCall 1983). We compiled several benchmarks by the existing compiler to produce bytecodes. Then we wrote microprograms for these bytecodes and measured the time to compute benchmarks. Since we are calculating the performance estimates by hand simulation, it is only feasible to compute for some small sample programs. Smalltalk-80 benchmarks consist of microbenchmarks and macrobenchmarks. Microbenchmarks consist of small programs that test several specific bytecodes. The microbenchmark whose performance correlate best with the performance of the overall system performance as well as the performance of macrobenchmarks is the method activation and return benchmark.

The source program for the method activation and return is

```
recur: t1
  t1=0 ifTrue: [↑self].
  self recur: t1-1.
  ↑self recur: t1-1
```

This method is called with 14 as the argument; thus, recur: is called 2 to the 15 minus 1, or 32767 times. It is compiled into the following sequence of bytecodes. The number on the right is the number of microinstructions to implement the bytecodes.

pushTemp: 0	3
pushConstant: 0	2
send: =	3
jumpFalse: 10	3 or 6
returnSelf	4
pushSelf	2
pushTemp: 0	3
pushConstant: 1	2
send: -	4
send: recur:	21
pop	1
pushSelf	2
pushTemp: 0	3
pushConstant: 1	2
send: -	4
send: recur:	21
returnTop	4

When the argument is 0, it takes 15 cycles to complete the execution, and when the argument is not 0, it takes 83 cycles. Since the number of times recur: is sent with 0 is one more than the number of times recur: is sent with non-zero argument, it takes 49 cycles on the average. If we can accomplish the initial plan and can fabricate the microprocessor with 125ns cycle time, the average execution time is 6.125μsec. This translates to 17 bytecodes per 12.25μsecond, or 1.4 million bytecodes per second. This is 5.03 times faster than Dorado Smalltalk, or 23 per cent faster than SOAR for this test.

6. Conclusion

We have designed a general-purpose, bytecode-emulating microprocessor. We completed the logic design and the microprogramming; the layout design has been carried out by a manufacturer.

We did not even consider an architecture that would have required substantial rewriting of any part of the source program of the standard Smalltalk-80 Virtual Image. This is because we would like to run Smalltalk-80 as soon as the chip is fabricated. This requirement virtually ruled out a register-oriented architecture. Even though we have kept our architectural aspirations modest, the Smalltalk-80 emulation microcode we wrote for Sword32 would be 25% faster than the SOAR Smalltalk, and would probably be many times faster than Dorado Smalltalk.

There are many features that we might include in a future version of the microprocessor. These features include: multi-level pipelining, internal cache for instructions and data, and tag support hardware.

ACKNOWLEDGEMENTS

Discussions with Ichiro Ogata, Edward Frank, Minoru Terada, Shumpei Kawasaki, Hideo Inayoshi, Yuzo Kita, and Touru Imai are very helpful.

REFERENCES

Deutsch, L.P., "The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture", in *Smalltalk-80: Bits of History, Words of Advice*, Ed. Krasner, Addison-Wesley, Reading, MA, 1983.

Deutsch, L.P. and Bobrow, D.G., "An Efficient Incremental Automatic Garbage Collector", *CACM*, vol.19, no.9, pp.522-526, Sept. 1976.

Deutsch, L.P. and Schiffman, A.M., "Efficient Implementation of the Smalltalk-80 System", *Conf. Rec. of the 11th Ann. ACM Symp. on Princ. of Prog. Lang.*, Salt Lake City, Utah, January, 1984.

Goldberg, A.J. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

Lampson, B., et al., *The Dorado: A High-Performance Personal Computer*, Technical Report, CSL-81-1, Xerox Palo Alto Research Center, 1981.

Mitchell, J.G., et al., *Mesa Language Manual*, Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1979.

McCall, K., "The Smalltalk-80 Benchmarks", in *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, MA, 1983.

Suzuki, N., *Implementing Smalltalk-80 on Sword*, Technical Memo, The University of Tokyo, Tokyo, Japan, September, 1983.

Suzuki, N., Frank, E., Kubota, K., and Ogata, I., *Sword: A Bytecode Emulating Microprocessor*, Technical Memo, The University of Tokyo, Tokyo, Japan, September, 1983.

Suzuki, N., Ogata, I., and Terada, M., "Design of a 32-bit Virtual Machine for Smalltalk-80", *Kigoushori*, 28-3, Information Processing Society of Japan, June, 1984.

Suzuki, N. and Terada, M., "Creating Efficient Systems for Object-Oriented Languages", *Conf. Rec. of the 11th Ann. ACM Symp. on Princ. of Prog. Lang.*, Salt Lake City, Utah, January, 1984.

Thacker, C., et al., "Alto: A Personal Computer", in *Computer Structures: Readings and Examples*, 2nd Edition, Eds. Sieworek, Bell, and Newell, McGraw-Hill, New York, 1981.

Unger, D., "Generation Scavenger: A Nondisruptive High Performance Storage Reclamation Algorithm", *Proceedings of Symposium on Practical Software Development Environments*, Pittsburgh, PA, April, 1984.

Unger, D., et al., "Architecture of SOAR: Smalltalk on a RISC", *Proc. of 11th Ann Int'l. Symp. on Comp. Architecture*, Ann Arbor, MI, June, 1984.

Wirfs-Brock, A., "Design Decisions for Smalltalk-80 Implementors", in *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, MA, 1983.