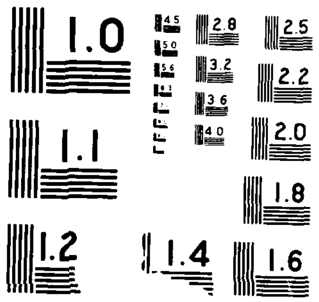
AD-A193 889 THE SMITE COMPUTER ARCHITECTURE(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) S WISEMAN ET AL. JAN 88 RSRE-MEMO-4126 DRIC-BR-105545 1/1

UNCLASSIFIED F/G 12/6 NL

END
DATE
FILMED
7 88



1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

UK105545

AD-A193 889

[illegible] No. [illegible]

ROYAL SIGNALS & RADAR ESTABLISHMENT

THE SMITE COMPUTER ARCHITECTURE

[illegible] Wiseman and H Field-Richards

88 5 10 009

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4126

Title : The SMITE Computer Architecture
Author : Simon R. Wiseman & Hugh S. Field-Richards
Date : January 1988

Summary

The SMITE computer architecture is being produced as a base for computer security applications. This paper describes the instruction set level of SMITE, which is based upon the Flex capability computer architecture.

1. Introduction

The SMITE project aims to provide Secure Multiprocessing of Information by Type Enforcement [Wiseman86]. To achieve this the software engineering techniques of type abstraction and object oriented programming will be used [Wiseman88]. However to get the best out of these relatively new techniques for building systems a sympathetic computer architecture is required.

Such an architecture has been developed at RSRE. It is called Flex [Foster.et.al.82] and has been available for a number of years, the latest implementation being for the ICL PERQ II workstation, though with new microcode and software [Currie.et.al.85]. Flex is a high level language oriented capability computer but is not specific to one particular language.

For the SMITE project a new hardware base is being produced to provide adequate performance and functionality for applications such as secure databases. The implementation is based on Flex to allow the Ten15 software engineering environment [Core&Foster86] to be used, but has additional functionality for security and performance reasons. This paper describes the instruction set level architecture of SMITE and the hardware structure that supports it.

2. Overview

The main store of SMITE is organised as a heap and capabilities are used for addressing. The address space of the heap is common to all software in the machine, which facilitates sharing of data. Basic protection is afforded by the capabilities, but higher levels of protection can be created using the type abstraction mechanisms [Wiseman88].

The SMITE instruction set is organised around a stack and a single register. Data can be treated as booleans, characters or words and may be organised into records or arrays. Procedures are "first class" objects because they are treated like any other data, in that once they are created they can be called in any context and still remain valid [Currie82]. Instructions support constructs for program control within a procedure, including FOR loops and CASE statements as well as various conditionals.

To provide a family of computers, offering a wide variety of processor, I/O and memory configurations, the SMITE computer has a multiprocessor architecture. This is centred around the IEEE Futurebus standard [IEEE87], a high performance asynchronous backplane bus. To give sufficient performance, each processor is equiped with its own private memory in addition to the shared global memory. However the microcode ensures that the software only observes a single uniform address space which spans all these memories. This allows software to be constructed independently of the multiprocessor configuration.

Each SMITE CPU is a flexible micro engine which is microprogrammed to emulate the high level SMITE instruction set. The hardware is designed to be comparatively simple to microprogram, by handling all timing constraints in the hardware, and yet should be relatively fast. For increased performance accesses to the local memory are cached and an instruction fetch unit prepares instructions for dispatch, in parallel with program execution.

The I/O subsystem of SMITE utilises standard backplane buses so that proprietory peripherals may be used. However to give the necessary protection all accesses to the bus are governed by capabilities. DMA and interrupt requests are mapped into the capability environment of the local memory on a per peripheral basis. These mappings are handled by an I/O CPU which is closely coupled to the main CPU.

2. Memory Organisation

2.1 Capabilities and Blocks

The primary memory of a SMITE computer is organised as a heap store. It is divided up into many, relatively small, blocks of various sizes. Unlike segmented memories the blocks are addressed by capabilities and no order is imposed upon them. A capability is the protected address of a block. Possession of a capability is all that is required to access the block it refers to. Conversely, no access to a block is possible without a capability that refers to it.

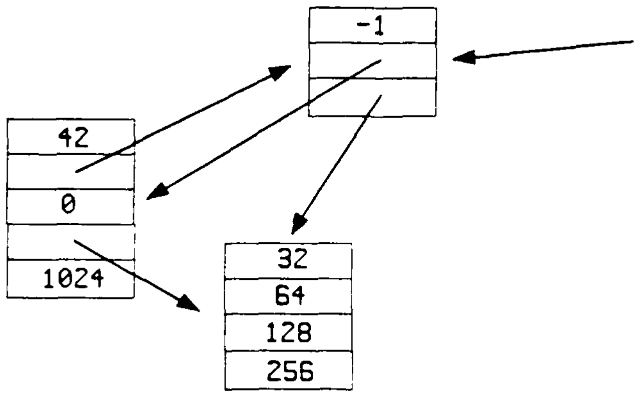


Fig2.1: The memory is divided into blocks which can contain both scalar data and capabilities, which refer to other blocks.

Blocks may contain a mixture of capabilities and scalar data. The distinction between them is maintained by the microcode, using a hidden bit on every word in the memory as a flag. The microcode ensures that it is not possible to treat scalar data as a capability, thus it is not possible to gain access to a block by guessing its address. This use of tagged memory gives software much greater flexibility in creating data structures than the method of partitioning used in other capability computers [Wiseman82].

When a block is accessed, the microcode ensures that all locations read or written are wholly within the bounds of the block. It is not possible to read or write beyond the end of a block, thereby gaining illegitimate access to another, by using, for example, an out of range index.

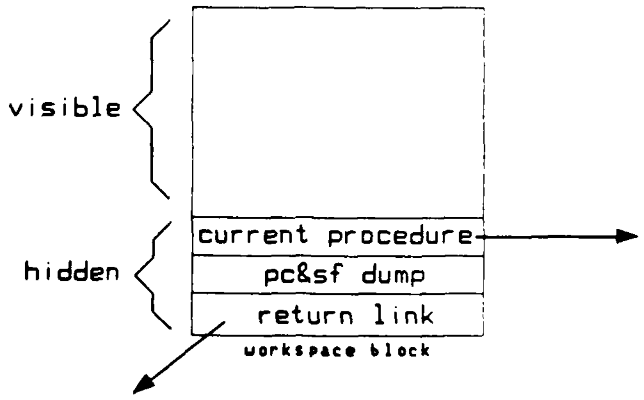


Fig2.2: Some types of block have words which are used by the microprogram in special ways and are hidden from the software.

There are eighteen different types of block in SMITE, which used to hold instructions and data and to represent objects such as processes and semaphores. The operations that may be performed on a block depend on its type. For example it is not possible to raise a data block or read from a semaphore block. A summary of the various types is given in an appendix.

Some block types have words which are used only by the microprogram and are completely hidden from the software. For example the procedure call return link in a workspace block is only manipulated by the procedure call and return instructions and cannot be accessed directly by software. In some types all the words are hidden, for example closures and peripheral blocks. They may only be accessed by specific instructions which manipulate the hidden words in a particular way.

2.2 Heap Memory Implementation

Capabilities are implemented using a level of indirection. That is, the capability contains an index into a table which contains the physical address of the block in memory. The table entries also contain the block's size, its type and flags for garbage collection purposes. The indirection table is managed by the microprogram and is completely hidden from the software.

A SMITE computer is a multi-processor with several separate memories (see figure 4.1), therefore another field in the capability indicates which memory the block resides in. If this is the local memory of some other processor, it must first be moved to the global memory, as described in section 4, before it can be accessed.

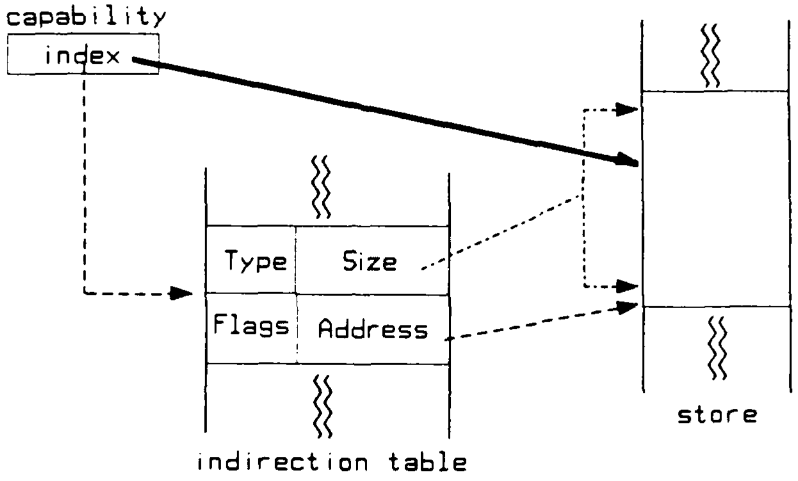


Fig2.3: Capabilities address a block via an indirection table, which also holds their type and some garbage collection flags.

To access a word within the block referred to by a capability, the microprogram first extracts the index field from the capability. This is used to read the indirection table entry that describes the block. The type of the block is checked to ensure that the access is allowed and the offset is checked to ensure it lies within the bounds of the block accessible to software. The address and offset are added together and the physical memory is accessed.

Translating capabilities into addresses is a comparatively lengthy process, however it is not done for every memory access. The capabilities which make up a process' context, for example the current instructions block and current expression stack, are translated once on procedure call and exit and process switch. The base and limit addresses are then kept in hardware registers where they can be accessed very quickly. Also, because the instruction set is high level language oriented, an instruction tends to move data in large quantities, such as whole arrays and records. Only one address translation would be required for each move.

The SMITE instruction set does not provide a means of deallocating a block. Instead a garbage collector [Wiseman85] is used which finds all blocks that are inaccessible and recovers them. This relieves the programmers from the burden of keeping track of references to variables in order to deallocate them only when they are no longer required. This is difficult to do well and causes errors which are hard to find. The use of a garbage collector therefore makes programs easier to write and maintain.

The garbage collection is performed by the microcode and is relatively fast. It is invoked by software using the garbage collect instruction, but operates incrementally. That is, the instruction is interruptable by processes of a higher priority. Usually a process running at the lowest priority will execute the instruction in a infinite loop, effectively giving a background garbage collector. If the free store starts to run low other processes may also execute the garbage collection instruction to ensure garbage is recovered more quickly.

Perhaps rather surprisingly SMITE does not provide a paged memory system. All blocks in the address space are found in physical memory, none are pushed out onto disc. Paging is usually necessary because processes running in the same computer are self contained and contain their own copies of code and data. They may also preallocate large amounts of space for stacks and private heaps. A much better utilization of memory is achieved in SMITE because code and other unalterable data is easily shared by all software in the system. This is made possible because all software operates in one capability based address space. The provision of an efficient system wide heap also removes the need to preallocate data space, instead space is allocated in small pieces as it is required.

The SMITE system also reduces memory requirements by providing dynamic linking and loading of software. Code is only loaded from disc as it is needed and once it is no longer required it is recovered by the garbage collector. So, for example, the initialisation code of a program is discarded once it is used and does not use up any memory space.

3. The Instruction Set

When a process is running its state is given by the context of the procedure it is executing. It has access to the procedure's code, its non-local variables and the local variables and stack associated with this particular call. A program counter gives the offset of the next instruction to be executed and a stack pointer gives the top of the expression stack. The process also has one general purpose register available to it.

The code block contains the constants required by the procedure, one of which is a capability for a block containing the instructions and another is the size of stack required by the procedure. The non-locals block is a data block which contains the non-local environment of the procedure, that is references to all variables declared outside the procedure's body. The workspace block contains locally declared variables and the expression evaluation stack.

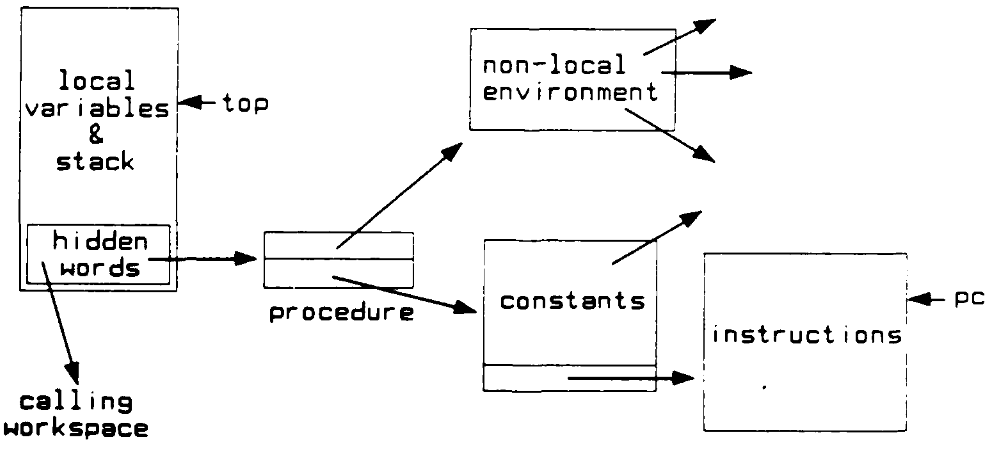


Fig3.1: A procedure has access to local variables, non-local variables and constants.

The non-local environment of a procedure takes the place of the display or static link in language run-time systems on conventional computers. When using displays, the overhead on accessing non-local variables occurs when the display is constructed as the procedure is called. If a static link is used, the overhead occurs each time the variable is accessed as the links are followed. With closures the overhead occurs when the procedure is declared, since references to all the non-local variables that may be accessed must be brought together and made into the non-locals block. The advantage of using closures is that procedures are treated like data objects, for example they can be delivered as the results of procedures and stored in data structures. Regardless of when they are called, closures remain valid. This is because they have a capability for their non-local environment bound into them, and this keeps the environment from being recovered by the garbage collector.

The universal register, called U, is able to contain any number of words, bytes or booleans, or nothing or special illegal values which are used for exception handling. Thus U carries with it typing information about the data it contains, which is used to give extra context to instructions. U cannot, however, hold mixtures of words, bytes and bools. For example, a structure of a word and a byte would be stored in U as two words.

The U register acts as the top of the stack, in that whenever a value is loaded into it, the current value is first pushed onto the top of the stack. For monadic instructions the operand is in U and the result remains in U. For dyadic instructions, the left operand is popped from the stack, the right operand is in U and the result is placed in U.

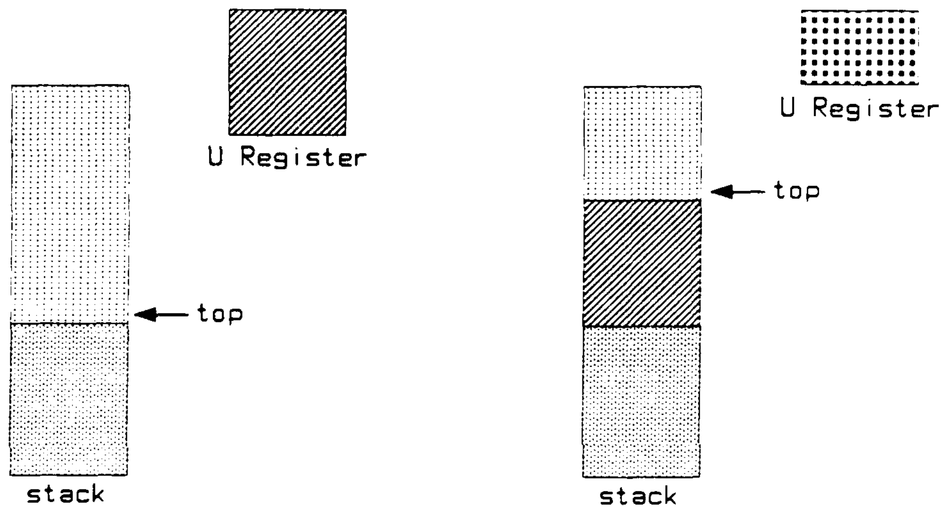


Fig 3.2: When the U register is loaded with new data, the contents are first pushed onto the stack.

For example the integer negate instruction expects a single word in U which is a scalar. It negates this value and places the result in U as a single scalar word. Integer addition pops one word from the stack, which must be a scalar, and adds this to the single scalar word in U, the result being placed in U as a single word. However, if U contains two scalar words, two scalar words are popped from the stack and long format addition is performed, with the resulting two word answer being placed in U.

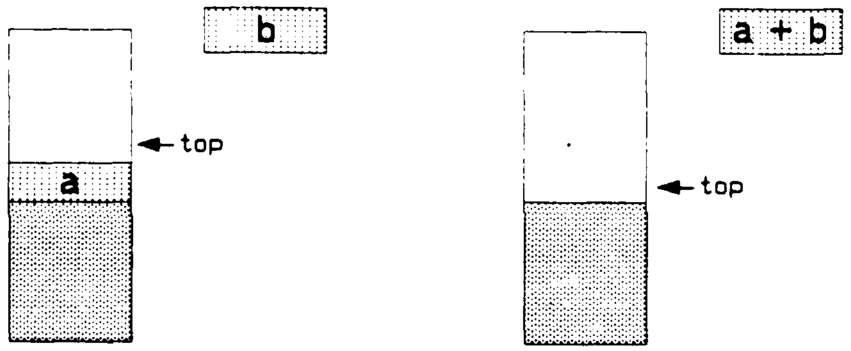


Fig 3.3: Dyadic operations like add pop the left operand from the stack and the right operand from U, and place the result in U.

The U register is also used for passing parameters to procedures and receiving results. A procedure is called by first evaluating the actual parameters one by one. These naturally get pushed onto the stack, leftmost parameter first. The final parameter is evaluated and is held in U. Then, in one instruction, this is pushed onto the stack and all the parameters are popped off into U. The procedure is then called. If the procedure returns a result (a function in Pascal) this will be in U. If it is to be the right operand of some operation, this can immediately be executed. If it is to be the left operand it will be pushed onto the stack by evaluating the right operand.

The instruction set offers the usual arithmetic operations, though if overflow or underflow occurs an exception is always generated. Arithmetic comparisons, such as less than, are made with specific instructions which return a boolean value in U, thereby avoiding problems with arithmetic overflow. Instructions are also provided that test for equality and inequality. These compare arbitrary data and return a boolean in U. For example, if U contains n-words another n-words are popped from the stack and compared.

Jump instructions allow loops and conditional statements to be implemented. Conditional jumps are made according to a boolean value in U, not by testing condition flags typical of modern microprocessors. This makes it particularly easy to implement complex conditional expressions involving ANDs and ORs.

Instructions for manipulating arrays and vectors of any element type are provided. Arrays are general n-dimension objects with arbitrary upper and lower bounds, while vectors are simple one dimensional objects with a lower bound fixed at one. Indexing, trimming and slicing instructions allow array and vector templates to be manipulated, while movement and comparison instructions operate on the actual data.

Instructions also provide functions usually associated with an operating system kernel, for example, operations on semaphores. These instructions are designed so that policy decisions, such as high level scheduling and quota management, can be made by software yet are actioned by microcode. This enables the microcode to present a virtual view of the hardware by effectively hiding the presence of multiple physical memories and processors.

A formal specification of the SMITE instruction set has been produced [Cooper&Diss87], using the language Z [Hayes87]. This is to avoid the problems usually encountered with ambigous English specifications, which is particularly important in view of the security requirements of intended applications.

4. Multiprocessor Architecture

The SMITE hardware base is a multiprocessor configuration centred around the IEEE Futurebus standard. This gives a range of processing power, memory size and I/O throughput from one processor architecture. The microcode ensures that multiprocessing is made transparent to the software, by performing memory management and primitive scheduling. Typical configurations will range from two processo - for a high performance workstation, up to ten processors for a database en .

The Futurebus has a number of important characteristics which make it uniquely suitable for the main system bus. It is the only standard bus which can transfer tagged data, because it is effectively a 33 bit bus. This allows processors to transfer scalar and capability data amongst themselves in the same format that it is stored in memory. Several buses may be used in parallel to achieve higher throughput and to give some redundancy for higher reliability. However, unlike other buses, sufficient locking primitives are provided to properly control access to shared resources. Futurebus defines protocols which allows various coherent caching schemes to be implemented for shared memory [Sweazey&Smith86], though it is not currently envisaged that SMITE will use these. Also Futurebus allows processor to processor communication, is asynchronous and fast.

A SMITE system consists of a number of processors and some tagged global memory attached to a dual Futurebus. Each processor comprises a CPU and some private tagged local memory. A processor only uses the Futurebus to access data in global memory and to communicate with other processors. Peripherals are attached to I/O buses, such as VME. These are controlled by an I/O CPU which is an optional part of a processor. The main and I/O CPUs share access to the processor's local memory. The main CPU emulates the SMITE instruction set, while the I/O CPU maps DMA requests, interrupts and peripheral bus accesses.

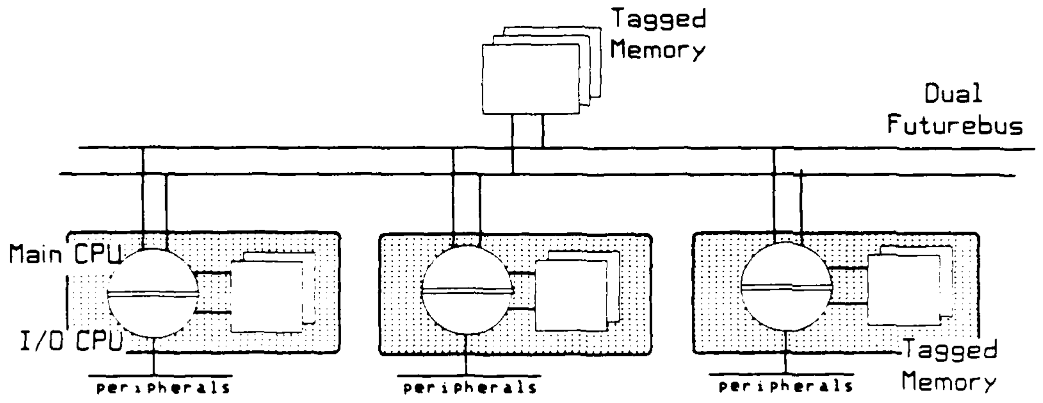


Fig4.1: A SMITE Computer comprises several processors and some memory attached to a Dual Futurebus. The processors have their own local memory, two CPUs and a peripheral bus.

To allow software to be written independently of the hardware configuration, the microcode implements a capability based virtual memory, or heap, which hides the separate physical memories. The hardware architecture prevents a processor from accessing the local memory of any other processor. However if a processor must access a block in another local memory, the block is first moved into global memory where it can be accessed. This is performed by microcode without the software's knowledge.

Capabilities contain the address of the memory in which the block they refer to resides. A capability may reside in any physical memory and can be freely copied without requiring any translation or modification. If the software requests an access to a block in another local memory, the processor's microcode communicates with the microcode of the other processor. This processor moves the block into global memory and reports back its new position. The access may now continue because all processors have access to global memory.

To avoid having to update all capabilities for a block which is moved, the indirection tables have special entries which indicate that a block has moved. These are put in place of the block's original entry and redirect all references to the new entry in the global memory. Eventually the garbage collector will ensure that all capabilities are updated to refer directly to the new position, at which point the entry can be recovered.

The microcode of the processors also cooperate in scheduling processes, so that software may be written independently of the number of processors. While low level scheduling is performed by microcode, software is responsible for high level scheduling. This can perform load balancing to gain optimum performance from the multiprocessor.

Each process has a base processor, on which it will usually run, though a special form of procedure call can be used to make a process execute on another processor. When the procedure exits, the process reverts to being run by the previous processor.

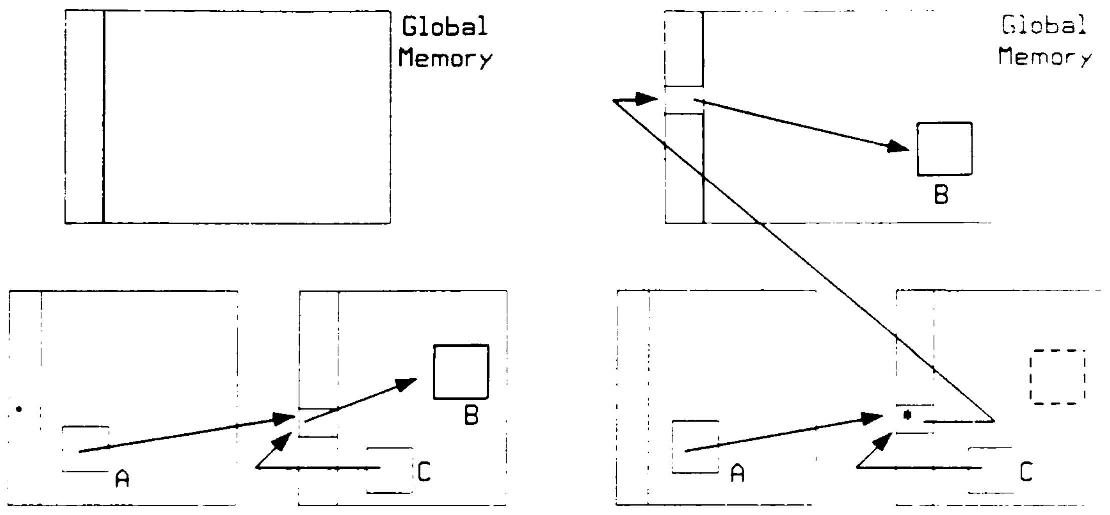


Fig4.2: Blocks A and C contain capabilities for B. If B has to be moved into global memory the indirection table entry shows it has moved.

For example, this mechanism is used by peripheral driver software to ensure that a process is being executed by the appropriate processor when the peripheral is accessed. Note that only that part of the process which is concerned with driving the peripheral is actually moved onto the new processor. If drivers are carefully written this will be no more than the data to be sent or received.

5. Main CPU Architecture

The main CPU of a SMITE processor is a microprogrammable engine which is microcoded to emulate the SMITE instruction set. The engine is programmed using a high level micro assembler [Poulter86] which allows the parallelism to be expressed in a natural way.

5.1 Data Paths

The basic CPU comprises of a sequencer, ALU, register file (AMD 29300 series components), scratchpad memory and memory interface. It has a basic cycle time of 90ns with transparent clock stretching where necessary and is able to perform several activities in parallel during one cycle. This requires a comparatively wide micro instruction of 128 bits.

The micro engine's data paths are generally 33 bits wide, to allow for transporting tagged data. The register file, which is four-ported, and scratchpad memory are able to store 33' words. The ALU is only 32', but a simple one bit ALU provides tag bit operations.

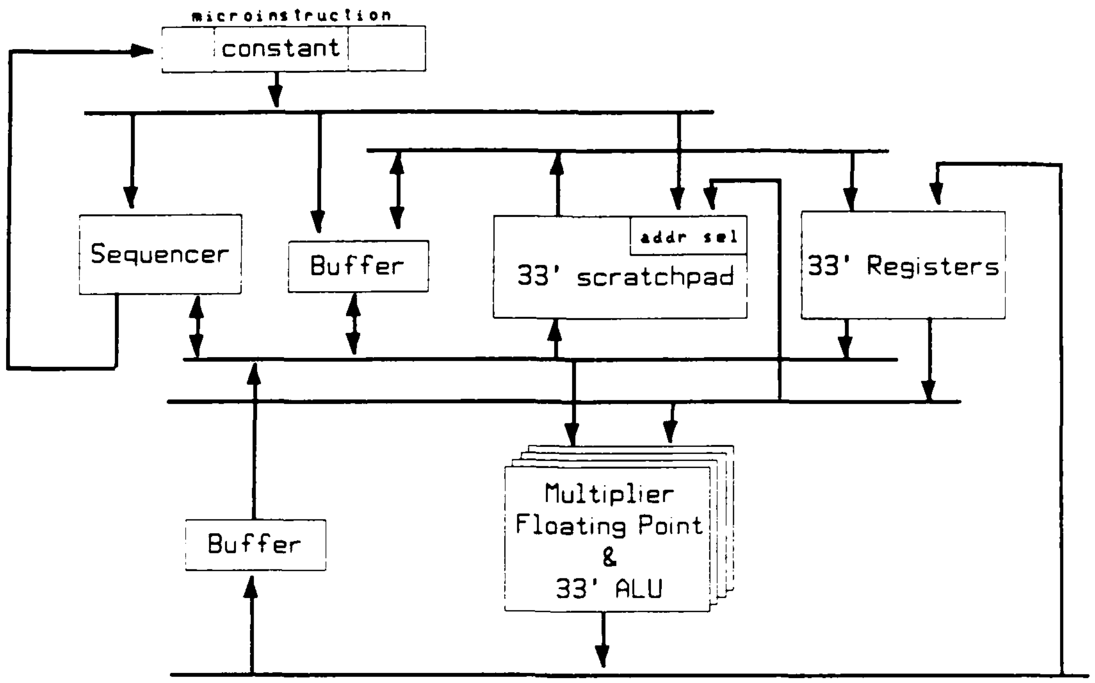


Fig5.1: The SMITE Microengine has 33' data paths and storage to facilitate processing tagged data.

The register file is four ported, allowing two operands to be extracted and two values to be loaded in each cycle. The larger scratchpad memory can be indirectly addressed, and is therefore used to hold various internal tables. Some registers are shared with the I/O CPU, allowing commands and parameters to be passed between the two CPUs.

5.2 Instruction Fetch

The processor contains an Instruction Fetch Unit which assembles and dispatches instructions. Unlike some IFUs [Lampsonet.al.84], it is a rather simple affair because the format of SMITE instructions is so simple. Each SMITE instruction consists of a number of bytes. There are just over 200 different instructions, so one byte gives the instruction opcode. Some instructions are qualified by one or two constants which are generally used to specify an offset or a size. These qualifiers are usually one byte, but an extended format using two bytes can be specified by preceeding the opcode with a byte of FE_{16}.

For example the integer add instruction is opcode 64_{16} and has no qualifiers, so only one byte specifies it. The instruction that loads an integer constant into U is opcode $1E_{16}$. This has one qualifier which is the value to be loaded. Most constants are quite small and one byte suffices. However for constants up to 2^{16}-1 two bytes can be used by preceeding the opcode with an extra byte of FE_{16}. Larger constants, including capabilities, can be loaded from the code block.

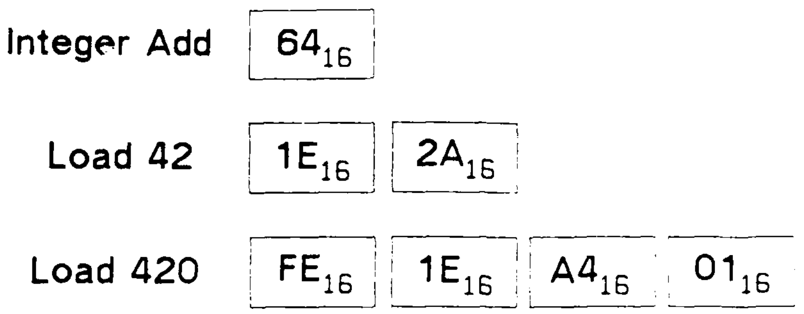


Fig5.2: Instructions have a one byte opcode. Some have constant qualifiers of one byte, though this can be extended to two.

Unlike most register based architectures, no complex addressing modes are used. The instruction stream only specifies constant data, like offsets and sizes, and not variable data like the value in a register. The task of the IFU is to ascertain the start address of the microprogram sequence that emulates the instruction and to assemble the qualifiers for the instruction, taking one or two bytes as appropriate. This saves several cpu cycles for each instruction dispatched, which is particularly important for short instructions such as arithmetic.

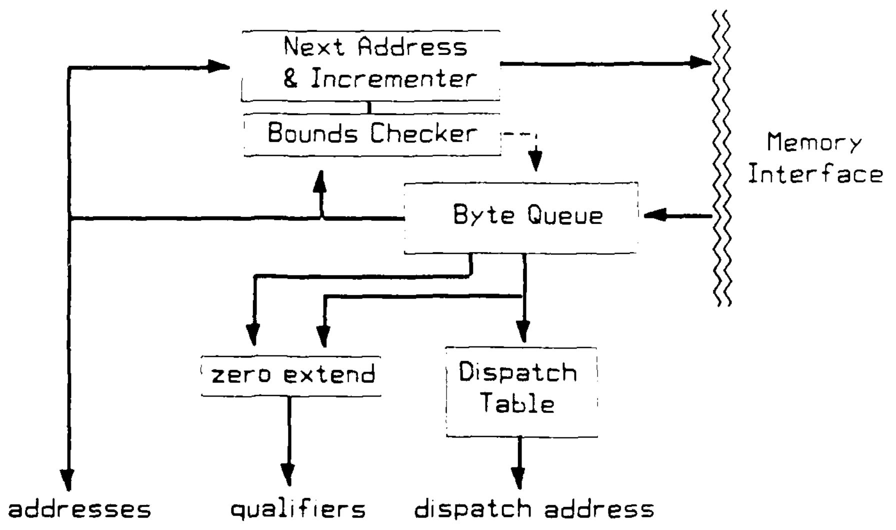


Fig5.3: The IFU generates dispatch addresses and assembles qualifiers.

Words are read from the instructions block autonomously, though at a lower priority than the main microprogram accesses, and assembled into a byte stream. The IFU also checks for attempts to run code off the end of the instructions block. Loops and branches only occur in programs where they are specified by the high level language statements, no loops are created to copy blocks of data because this type of operation is handled by single instructions. This obviates the need for complex hardware to reduce the time wasted by breaks in the instruction pipeline.

Many SMITE instructions begin with pushing the current value contained in U onto the stack. Depending on the type of value held in U, different microprogram sequences must be executed to do this. To save on a few cpu cycles, the IFU first dispatches these instructions to the appropriate push code, according to the value in the hardware U-type register.

5.3 Data Caching

The SMITE instruction set is stack oriented and it is well known that stack based expression evaluation can use more memory bandwidth than register based evaluation [Myers77]. To improve performance, therefore, accesses to local memory are cached. That is, the values for those addresses which are likely to be accessed in the near future are held in a small, but very fast, memory.

A simple direct mapped cache is used. In this, part of the address is used to index the cache array, which contains the cached data and the address of that data. A comparison is made between the incoming address and the address of the cached data. If they are equal a cache hit has occured and the cached data is used rather than the main memory. In a fully associative cache the incoming address is compared with all cached addresses in parallel, which is very expensive to implement.

Many systems have several direct mapped caches working in parallel and perform an associative search to determine which cache, if any, contains the contents of a location. Two-way set associative is most common. In SMITE the memory is divided into blocks of various types. By caching blocks containing instructions (readonly scalar), blocks containing local variables and the stack (workspace) and blocks containing other data in different caches the effect of set associative caches is achieved with much simpler hardware. Another cache is used for the indirection tables which contain the addresses of all the blocks.

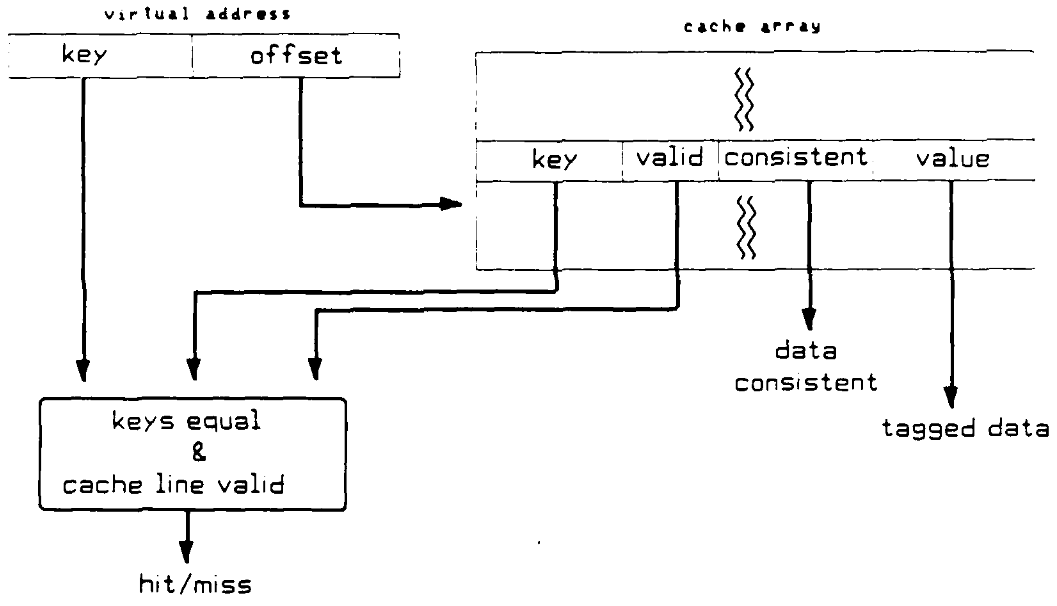


Fig5.4: Each SMITE cache is direct mapped and uses a write back scheme.

In general a write back caching scheme is used. This is where data written to a location that is cached, is stored in the cache but not in the main memory. If the cache line is needed to cache the value of another location, the data in it must first be written back to main memory. This has the advantage of reducing memory traffic, but at the expense of more complex hardware. It does however increase the memory bandwidth that is available to the I/O CPU for DMA transfers. It is also particularly advantageous for stack based architectures because accesses to the top of stack do not use main memory bandwidth.

The microprogram is emulating an instruction set which operates in a well structured environment. This enables it to make certain optimisations which are just not possible in a conventional architecture. For example, the hidden words of a workspace block are only accessed during process switches and procedure calls and exits. These locations are therefore rarely accessed, so if they are cached it will be at the expense of some other more useful data. When these locations are accessed, the microprogram can direct the memory system to not cache them. Another example is that data popped from the stack will never be accessed again, so it can be removed from the cache to allow more useful data in.

6. I/O Architecture

SMITE is intended for applications which demand highly assured security. Capabilities and abstract types are used internally to provide this, however equal protection must be given to the computer's interface to the outside world. For this reason all interactions with the peripherals are monitored by the I/O CPU. This acts as the controller of the bus and handles requests for DMA transfers.

The I/O CPU receives commands from the main CPU. These are generated in response to the use of instructions which transfer data to and from the Control and Status Registers of a peripheral. The main CPU is responsible for generating correct addresses. This is controlled by a peripheral block, there is one for each peripheral on the bus, which contains the range of valid addresses for the peripheral. For example, the instruction which reads one of a peripheral's status registers takes a capability for the peripheral block on the stack and an offset in U. The offset is added to the address of the first register and, after a check is made to to ensure the offset is ok, it is passed to the I/O CPU. This accesses the register and passes the result back to the main CPU which places it in U.

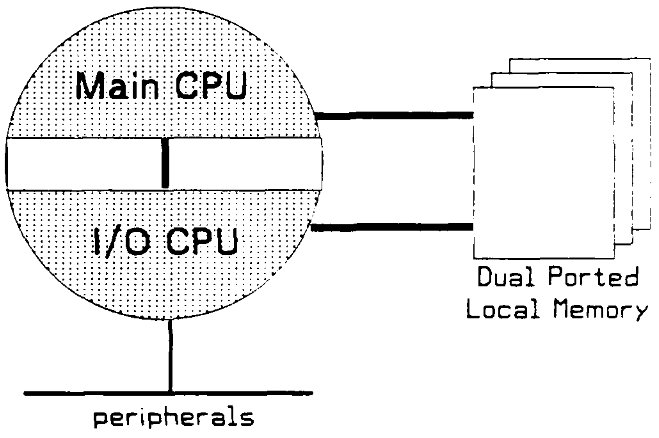


Fig6.1: The I/O CPU maps bus accesses, DMA requests and interrupts.

Each peripheral has a number of interrupt vectors associated with it. A semaphore can be associated with each of these so that if the peripheral interrupts on that vector, the semaphore is released. Thus the peripheral is able to signal to a device driver process waiting on this semaphore.

Each peripheral also has a number of DMA channels associated with it. An uncached scalar data block may be associated with each of these for use as DMA buffers. DMA addresses generated by a peripheral contain the channel number and an offset. The I/O CPU maps this into an address within the appropriate buffer, but only after checking that the offset and channel numbers are valid. The I/O CPU has direct access to the local memory and carries out the DMA request without the intervention of the main CPU.

To gain the highest assurance in the security of a SMITE computer, it is necessary to be sure that peripherals are not able to access data that is not intended for them. Also a peripheral must not be able to communicate with the I/O CPU while pretending to be another peripheral. Unfortunately this can only be achieved by using special interface logic between standard peripheral cards and the bus, or by using specially designed cards. The costs involved in doing this will limit its usage to only a few very important systems.

7. Summary

The SMITE project proposes using a computer capable of providing user definable, protected abstract data types for use in computer security applications. Such facilities cannot be supported adequately by conventional processor architectures, but are provided by capability architectures. The architecture of the RSRE Flex computer has been taken as the starting point for a new multi-processor computer capable of providing the performance and protection features that will be required by future multi-level computer security applications.

An initial prototype of a primitive SMITE processor has been built and a full single processor SMITE computer workstation is under construction. The existing Flex software development facilities are being used on PERQ workstations to develop SMITE microprogram and operating system code.

Acknowlegements

Thanks go to the designers of the Flex architecture whose original research provides a foundation for the SMITE project.

References

D.G.Cooper & M.J.Diss
The Specification of the SMITE Instruction Set in Z
Report Number 72/87/R528/U
Plessey Research Roke Manor Limited
December 1987

P.W.Core & J.M.Foster
Ten15: An Overview
RSRE Memo 3977, September 1986

I.F.Currie
In Praise of Procedures
RSRE Memo 3499, July 1982

I.F.Currie, J.M.Foster & P.W.Edwards
PerqFlex Firmware
RSRE Report 85015, December 1985

J.M.Foster, I.F.Currie & P.W.Edwards
Flex: A Working Computer with an Architecture Based on Procedure Values
RSRE Memo 3500, July 1982

I Hayes (Ed.)
Specification Case Studies
Prentice-Hall International Series in Computer Science
1987

IEEE87
IEEE896.1:1987 Futurebus Backplane Specification

B.W.Lampson, G.McDaniel & S.M.Ornstein
An Instruction Fetch Unit for a High Performance Personal Computer
IEEE Trans on Computers
Vol C-33, Num 8, August 1984, pp712..730

G.J.Myers
The Case Against Stack-Oriented Instruction Sets
Computer Architecture News
Vol 6, Num 3, 1977, pp7..10

N.D.Poulter
A Compiler-Compiler for the SCP3 Micro-Program
RSRE Memo 3982, September 1986

P.Sweazey & A.J.Smith
A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus
Procs. 13th Ann. Symp on Computer Architecture, Tokyo, Japan
June 1986

S.R.Wiseman
Two Advanced Computer Architectures: A Study of their Support for Languages and Operating Systems
RSRE Report 82013, July 1982

S.R.Wiseman
A Garbage Collector for a Large Distributed Address Space
RSRE Report 85009, June 1985

S.R.Wiseman
A Secure Capability Computer System
IEEE Symposium on Security and Privacy
Oakland, California
April 1986

S.R.Wiseman
Protection & Security Mechanisms in the SMITE Capability Computer
RSRE Memo 4117, January 1988

Appendix

The following summary lists the different types of block and indicates their role in the SMITE architecture.

workspace	used to hold the local variables and an expression evaluation stack for a procedure call. Hidden words contain the link to the workspace of the calling procedure and space to dump the context during an inner procedure call.
data	holds a mixture of scalar data and capability data.
scalar	holds scalar data, but not capabilites.
uncached scalar	holds scalar data, but not capabilites, and is not cached. This type is used for DMA buffers and video display rasters.
readonly scalar	a data block which can only hold scalar data, but cannot be written to once it is created. This is used for instructions.
readonly data	a data block which may hold a mixture of scalar and capability data, but cannot be written to once it is created.
code	holds the constants of a procedure, which may be scalar or capabilities, along with a capability for the instructions.
closure	this is a representation of a procedure, and comprises a binding of the code with its non local environment.
immortal closure	represent procedures which cannot be failed by asynchronous exceptions, such as break in.
mode	used to represent modes (or types) as values in very high level languages, such as Ten15 [Core&Foster86].
readonly mode	as for mode, but can never be altered. In particular, used when mode blocks are brought in from backing store.
keyed	used to hide data from its users in a way which allows software to create protected abstract data types.
readonly keyed	as for keyed, but can never be altered. In particular, used when keyed blocks are brought in from backing store.
process	carries the context of a suspended process along with quota and scheduling information.
peripheral	represents a peripheral located on the I/O bus. Contains information necessary to map between the capability world and the linear address space of the I/O bus.
semaphore	gives simple binary semaphores for process synchronisation.
area	these control the usage of memory, allowing quotas to be imposed on the amount of resource consumed.
hash table	give a fast, microcoded search function. Used for translating external capabilities, such as those for objects in remote machines or on backing store.

DOCUMENT CONTROL SHEET

Overall security classification of sheet .... UNCLASSIFIED ....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference MEMO 4126	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known) 778400	6. Originator (Corporate Author) Name and Location RSRE St Andrews Road, Malvern, Worcs. ~~WR13 5LA~~		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title SMITE COMPUTER ARCHITECTURE
7a. Title in Foreign Language (in the case of translations)
7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials Wiseman, S.R.	9(a) Author 2 Field-Richards H.S.	9(b) Authors 3,4...	10. Date 1988.1	pp. ref. 14
11. Contract Number	12. Period	13. Project	14. Other Reference	

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The SMITE computer architecture is being produced as a base for computer security applications. This paper describes the instruction set level of SMITE, which is based upon the Flex capability computer architecture.

S80/48

first dispatches these instructions to the appropriate push code, according to
the value in the hardware U-type register.