An Overview of the Mesa Processor Architecture

Richard K. Johnsson
John D. Wick
Xerox Office Products Division
3333 Coyote Hill Road
Palo Alto, California 94304

Introduction

This paper provides an overview of the architecture of the Mesa processor, an architecture which was designed to support the Mesa programming system [4]. Mesa is a high level systems programming language and associated tools designed to support the development of large information processing applications (on the order of one million source lines). Since the start of development in 1971, the processor architecture, the programming language, and the operating system have been designed as a unit, so that proper tradeoffs among these components could be made. The three main goals of the architecture were:

- To enable the efficient implementation of a modular, high level programming language such as Mesa. The emphasis here is not on simplicity of the compiler, but on efficiency of the generated object code and on a good match between the semantics of the language and the capabilities of the processor.
- To provide a very compact representation of programs and data so that large, complex systems can run efficiently in machines with relatively small amounts of primary memory.
- To separate the architecture from any particular implementation of the processor, and thus accommodate new implementations whenever it is technically or economically advantageous, without materially affecting either system or application software.

We will present a general introduction to the processor and its memory and control structure; we then consider an example of how the Mesa instruction set enables significant reductions in code size over more traditional architectures. We will also discuss in considerable detail the control transfer mechanism used to implement procedure calls and context switches among concurrent processes. A brief description of the process facilities is also included.

General Overview

All Mesa processors have the following characteristics which distinguish them from other computers:

High Level Language

The Mesa architecture is designed to efficiently execute high level languages in the style of Algol, Mesa, and Pascal. Constructs in the programming languages such as modules, procedures and processes all have concrete representations in the processor and main memory, and the instruction set includes opcodes that efficiently implement those language constructs (e.g. procedure call and return) using these structures. The processor does not "directly execute" any particular high level programming language.

Compact Program Representation

The Mesa instruction set is designed primarily for a compact, dense representation of programs. Instructions are variable length with the most frequently used operations and operands encoded in a single byte opcode; less frequently used combinations are encoded in two bytes, and so on. The instructions themselves are chosen based on their frequency of use. This design leads to an asymmetrical instruction set. For example, there are twenty-four different instructions that can be used to load local variables from memory, but only twenty-one that store into such variables; this occurs because typical programs perform many more loads than stores. The average instruction length (static) is 1.45 bytes.

Compact Data Representation

The instruction set includes a wide variety of instructions for accessing partial and multiword fields of the memory's basic unit, the sixteen bit word. Except for system data structures defined by the architecture, there are no alignment restrictions on the allocation of variables, and data structures are generally assumed to be tightly packed in memory.

Evaluation Stack

The Mesa processor is a stack machine; it has no general purpose registers. The evaluation stack is used as the destination for load instructions, the source for store instructions, and as both the source and destination for arithmetic instructions; it is also used for passing parameters to procedures. The primary motivation for the stack architecture is not to simplify code generation, but to achieve compact program representation. Since the stack is assumed as the source and/or destination of one or more operands, specifying operand location requires no bits in the instruction. Another motivation for the stack is to minimize the register saving and restoring required in the procedure calling mechanism.

Control Transfers

The architecture is designed to support modular programming, and therefore suitably optimizes transfers of control between modules. The Mesa processor implements all control transfers with a single primitive called **XFER**, which is a generalization of the notion of a procedure or subroutine call. All of the standard procedure calling conventions (call by value, call by reference (result), etc.) and all transfers of control between contexts (procedure call and return, nested procedure calls, coroutine transfers, traps, and process switches) are implemented using the **XFER** primitive. To support arbitrary control transfer disciplines, activation records (called *frames*) are allocated by **XFER** from a heap rather than a stack; this allows the heap to be shared by multiple processes.

Process Mechanism

The architecture is designed for applications that expect a large amount of concurrent activity. The Mesa processor provides for the simultaneous execution of up to one thousand asynchronous preemptable processes on a single processor. The process mechanism implements monitors and condition variables to control the synchronization and mutual exclusion of processes and the sharing of resources among them. Scheduling is event driven, rather than time sliced. Interrupts, timeouts, and communication with I/O devices also utilize the process mechanism.

Virtual Memory

The Mesa processor provides a single large, uniformly addressed virtual memory, shared by all processes. The memory is addressed linearly as an array of 2^{32} sixteen-bit words, and, for mapping purposes, is further organized as an array of 2^{24} pages of 256 words each; it has no other programmer visible substructure. Each page can be individually write-protected, and the processor records the fact that a page has been written into or referenced.

Protection

The architecture is designed for the execution of cooperating, not competing, processes. There is no protection mechanism (other than the write-protected page) to limit the sharing of resources among processes. There is no "supervisor mode," nor are there any "privileged" instructions.

Virtual Memory Organization

Virtual addresses are mapped into real addresses by the processor. The mapping mechanism can be modeled as an array of real page numbers indexed by virtual page numbers. The array can have holes so that an associative or hashed implementation of the map is allowed; the actual implementation is not specified by the architecture and differs among the various implementations of the Mesa processor.

Instructions are provided to enable a program (usually the operating system) to examine and modify the virtual-to-real mapping. The processor maintains "write-protected," "dirty," and "referenced" flags for each mapped virtual page which can also be examined and modified by the program.

The address translation process is identical for *all* memory accesses, whether they originate from the processor or from I/O devices. There is no way to bypass the mapping and directly reference a main memory location using a real address. Any reference to a virtual page which has no associated real page (*page fault*), or an attempt to store into a write-protected page (*writeprotect fault*) will cause the processor to initiate a process switch (as described below). The abstraction of faults is that they occur between instructions so that the processor state at the time of the fault is well defined. In order to honor this abstraction, each instruction must avoid all changes to processor state registers (including the evaluation stack) and main memory until the possibility of faults has passed, or such changes must be undone in the event of a fault.

Virtual memory is addressed by either long (two word) pointers containing a full virtual address or by short (one

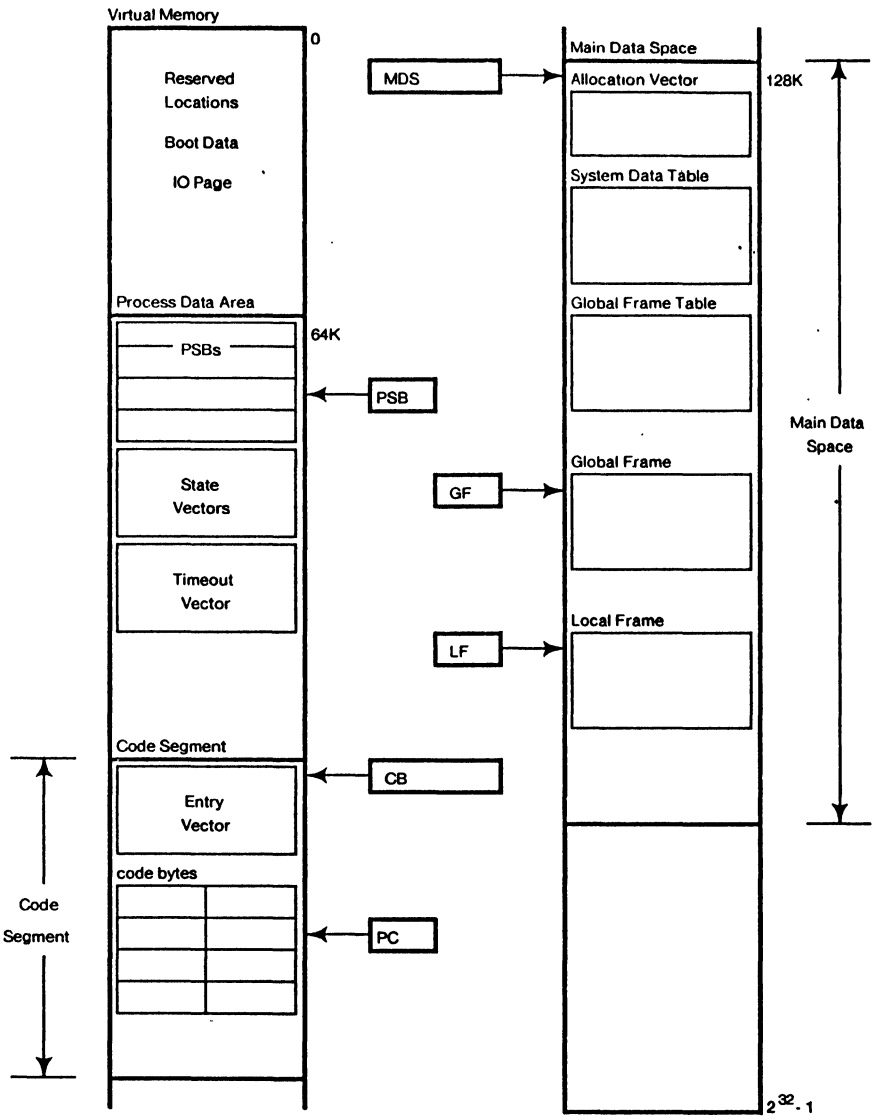

Figure 1. Virtual Memory Structure

word) pointers containing an offset from an implicit 64K word aligned base address. There are several uses of short pointers defined by the architecture:

- The first 64K words of virtual memory are reserved for booting data and communication with I/O devices. Virtual addresses known to be in this range are passed to I/O devices as short pointers with an implicit base of zero.
- The second 64K of virtual memory contains data structures relating to processes. Pointers to data structures in this area are stored as short pointers with an implicit base of 64K.
- Any other 64K region of virtual memory can be a main data space (**MDS**). Each process executes within some **MDS** in which its module and procedure variables are stored; these variables can be referenced by short pointers using as an implicit base the value stored in the processor's **MDS** register.

Code may be placed anywhere in virtual memory, although in general it is not located within any of the three regions mentioned above. A code segment contains read only instructions and constants for the procedures that comprise a Mesa module; it is never modified during normal execution and is usually write-protected. A code

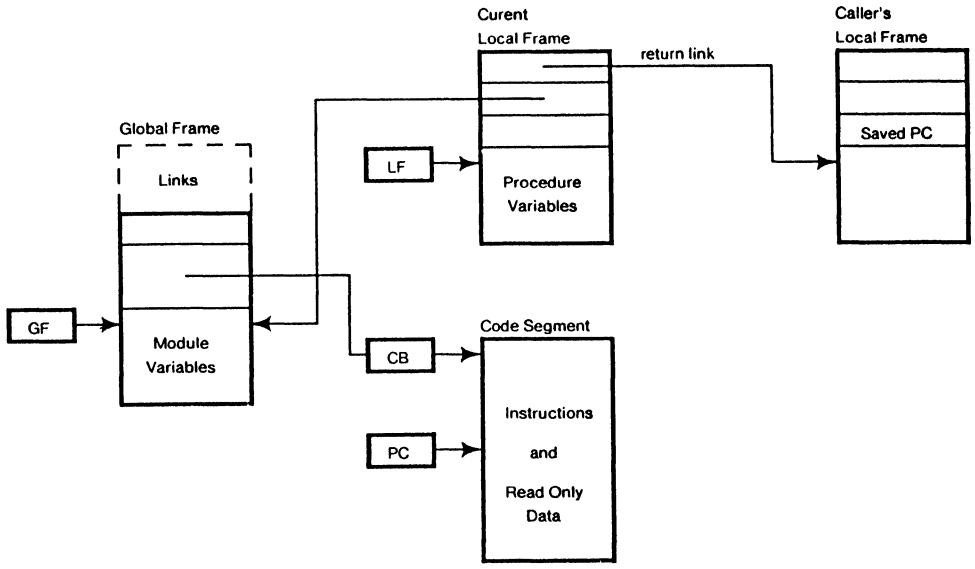

Figure 2. Local and Global Frames and Code Segments

segment is relocatable without modification; no information in a code segment depends on its location in virtual memory.

The data associated with a Mesa program is allocated in a main data space in the form of local and global frames. A global frame contains the data common to all procedures in the module, i.e. declared outside the scope of any procedure. The global frame is allocated when a module is loaded, and freed when the module is destroyed. A local frame contains data declared within a procedure; it is allocated when the procedure is called and freed when it returns.

Any region of the virtual memory, including any main data space, can contain additional dynamically allocated user data; it is managed by the programmer and referenced indirectly using long or short pointers. An MDS also contains a few system data structures used in the implementation of control transfers (discussed below). The overall structure of virtual memory is shown in Figure 1.

Besides enabling standard high level language features such as recursive procedures, multiple module instances, coroutines, and multiple processes, the representation of a program as local data, global data, and code segment tends to increase locality of reference; this is important in a paged virtual memory environment.

Contexts

In addition to a program's variables, there is a small amount of linkage and control information in each frame. A local frame contains a short pointer to the associated global frame and a short pointer to the local frame of its caller (the *return link*). A local frame also holds the code segment relative program counter for a procedure whose execution has been suspended (by preemption or by a call to another procedure). Each global frame contains a long pointer to the code segment of the module. A global frame optionally is preceded by an area called the *link space*, where links to procedures and variables in other modules are stored. This structure is shown in Figure 2.

To speed access to code and data, the processor contains registers which hold the local and global frame addresses (LF and GF), and the code base and program counter (CB and PC) for the currently executing procedure; these are collectively called a context. When a procedure is suspended, the single sixteen bit value which is the MDS relative pointer to its local frame is sufficient to reestablish this complete context by fetching GF and PC from the local frame and CB from the global frame. The management of these registers during context switches is discussed in the section on control transfers below.

The Mesa Instruction Set

As mentioned above, a primary goal of the Mesa architecture is compact representation of programs. The general idea is to introduce special mechanisms into the instruction set so that the most frequent operations can be represented in a minimum number of bytes. See [5] for a description of how the instruction set is tuned to accomplish this goal. Below we enumerate a representative sample of the instruction set.

Many functions are implemented with a family of instructions with the most common forms being a single

byte. In the descriptions of instructions below, operand bytes in the code stream are represented by α and β; $\alpha\beta$ represents two bytes that are taken together as a sixteen bit quantity. The suffix **n** on an opcode mnemonic represents a group of instructions with **n** standing for small integers, e.g. **LIn** represents **LI0**, **LI1**, **LI2**, etc. A trailing **B** in an opcode indicates a following operand byte (α); **W** indicates a word ($\alpha\beta$); **P** indicates that the operand byte is a pair of four bit quantities, α.left and α.right.

Operations' on the stack. These instructions obtain arguments from and return results to the evaluation stack. Although elements in the stack are sixteen bits, some instructions treat two elements as single thirty-two bit quantities. Numbers are represented in two's complement.

DIS Discard the top element of the stack (decrement the stack pointer).

REC Recover the previous top of stack (increment the stack pointer).

EXCH Exchange the top two elements of the stack.

DEXCH Exchange the top two doubleword elements of the stack.

DUP Duplicate the top element of the stack.

DDUP Duplicate the top doubleword element of the stack.

DBL Double the top of stack (multiply by 2).

unary operations: **NEG, INC, DEC,** etc.

logical operations: **IOR, AND, XOR.**

arithmetic: **ADD, SUB, MUL.**

doubleword arithmetic: **DADD, DSUB.**

Divide and other infrequent operations are relegated to a multibyte escape opcode that extends the instruction set beyond 256 instructions.

Simple Load and Store instructions. These instructions move data between the evaluation stack and local or global variables.

LIn Load Immediate n.

LIB α Load Immediate Byte.

LIW $\alpha\beta$ Load Immediate Word.

LLn Load Local n; load the word at offset n from **LF.**

LLB α Load Local Byte; load the word at offset α from **LF.**

SLn Store Local n.

SLB α Store Local Byte.

PLn Put Local n; equivalent to **SLn REC,** i.e. store and leave the value on the stack.

LGn Load Global n; load the word at offset n from **GF.**

LGB α Load Global Byte; load the word at offset α from **GF.**

SGB α Store Global Byte.

LLKB α Load Link; load a word at offset α in the link space.

There are also versions of these instructions that load doubleword quantities. Note that there are no three-byte versions of these loads and stores and no one-byte Store Global instructions. These do not occur frequently enough to warrant inclusion in the instruction set.

Jumps. All jump distances are measured in bytes relative to the beginning of the jump instruction; they are specified as signed eight or sixteen bit numbers.

Jn short positive jumps.

JB α jump -128 to $+127$ bytes.

JW $\alpha\beta$ long positive or negative jumps.

JLB α compare (unsigned) top two elements of stack and jump if less; also **JLEB, JEB, JGB, JGEB** and unsigned versions.

JEBB $\alpha\ \beta$ if top of stack is equal to to α, jump distance in β; also **JNBB.**

JZB α jump if top of stack is zero; also **JNZB.**

JEP α if top of stack is equal to α.left, jump distance in α.right; also **JNEP.**

JIB $\alpha\beta$ at offset $\alpha\beta$ in the code segment find a table of eight bit distances to be indexed by the top of stack; also **JIW** with a table of sixteen bit distances.

Read and Write through pointers. These instructions read and write data through pointers on the stack or stored in local variables.

Rn Read through pointer on stack plus small offset.

RB α Read through pointer on stack plus offset α.

WB α Write through pointer on stack plus offset α.

RLIP α Read Local Indirect; use pointer in local variable α.left; add offset α.right.

WLIP α Write Local Indirect.

RnF α — Read Field using pointer on the stack plus n; α contains starting bit and bit count as four bit quantities.

RF α β — Read Field using pointer on the stack plus α; β contains starting bit and bit count as four bit quantities.

WF α β — Write Field.

RKIB α — Read Link Indirect; use the word at offset α in the link space as a pointer.

There are also versions of these instructions that take long pointers and versions that read or write doubleword quantities.

Control Transfers. These instructions handle procedure call and return. Local calls (in the same module) specify the *entry point number* of the destination procedure; external calls (to another module) specify an index of a *control link* in the module's link space (see the section on Control Transfers).

LFCn — Local Function Call using entry point n.

LFCB α — Local Function Call using entry point α.

EFCn — External Function Call using control link n.

EFCB α — External Function Call Byte using control link α.

SFC — Stack Function Call; use control link from the stack.

RET — Return. XFER using the return link in the local frame as the destination; free the frame.

BRK — Breakpoint; a distinguished one-byte instruction that causes a trap.

Miscellaneous. These instructions are used to generate and manipulate pointer values.

LAn — Local Address n; put the address of local variable n on the stack.

LAB α — Local Address Byte; put the address of local variable α on the stack.

LAW $\alpha\beta$ — Local Address Word; put the address of local variable $\alpha\beta$ on the stack.

GAn — Global Address n; put the address of global variable n on the stack.

GAB α — Global Address Byte; put the address of global variable α on the stack.

GAW $\alpha\beta$ — Global Address Word; put the address of global variable $\alpha\beta$ on the stack.

LP — Lengthen Pointer; convert the short pointer on the stack to a long pointer by adding MDS; includes a check for invalid pointers.

An example. Consider the program fragment below. The statement c ← Q[p.f + i] means "call procedure Q, passing the sum of i and field f of the record pointed to by local variable p; store the result in global variable c." The statement RETURN [a[i].c] means "return as the value of the procedure Q field c of the ith record of global array a."

```
Prog: PROGRAM =
   BEGIN

   c: CHARACTER;
   a: ARRAY INTEGER OF RECORD [
      b: BOOLEAN,
      c: CHARACTER,
      s: INTEGER[0..128),
      w: CARDINAL];

   P: PROCEDURE =
      BEGIN
      i: INTEGER = 2;
      p: POINTER TO RECORD [. . . , f: INTEGER];
      . . . ;
      c ← Q[p.f + i];
      . . . ;
      END;

   Q: PROCEDURE [i: INTEGER]
         RETURNS [CHARACTER] =
      BEGIN
      RETURN [a[i].c];
      END;

   END.
```

Below we have shown the code generated for this program fragment in a generalized Mesa instruction set, and then in the current optimized version of the instruction set.

Source	Mesa/Gen	Mesa/Opt
p.f	LL p	RLIP (p, f)
	R f	
i	LI i	LIi
p.f + i	ADD	ADD
Q[..]	LFC q	LFCq
n ←	SG n	SG n
	11 Code Bytes	7 Code Bytes
	6 Instructions	5 Instructions
Q:	SL i	PLi
i	REC	
	DBL	DBL
	GAB a	GAa
a[i]	ADD	ADD
a[i].c	RF 0,(1,8)	ROF (1,8)
RETURN	RET	RET
	11 Code Bytes	7 Code Bytes
	7 Instructions	6 Instructions

Although this is admittedly a contrived example, it cannot be called pathological, and it does illustrate quite well several of the ways the Mesa instruction set achieves code size reduction. In particular:

- *Use of the evaluation stack.* The stack is the implicit destination or source for load and store operations; instructions can be smaller because they need not specify all operand locations. Since the stack is also used to pass parameters, no extra instructions are needed to set up for the procedure call. Most statements and expressions are quite simple so that the added generality of a general register architecture is a liability rather than an asset.
- *Control transfer primitive.* By using a single, standard calling convention with built-in storage allocation, almost all of the overhead associated with a call is eliminated. There is minimal register saving and restoring.
- *Common operations are single instructions.* Operations that occur frequently are encoded in single instructions. Reading a word from a record given a pointer to the record in a local variable is a good example (**RLIP**). There are similar instructions for storing values through pointers. There are instructions that deal with partial word quantities or that include runtime as well as compile time offsets. Procedure calls are also given single instructions.
- *Frequently referenced variables are stored together.* Most operands are addressed with small offsets from local or global frame pointers or from variable pointers stored in the local or global frame. Using small offsets means that instructions can be smaller because fewer bits are needed to record the offset. The compiler assists by assigning variable locations based on static frequency so that the smallest offsets occur most often.

These last two points are the guiding principles of the Mesa instruction set. If an operation, even a complex one involving indirection and indexing, occurs frequently in "real" programs, then it should be a single instruction or family of instructions. For instruction families with compile time constant operands such as offsets, assigning operand values by frequency increases the payoff of merging small operand values into the opcode or packing multiple values into a single operand byte. There are a small number of cases in which an infrequently used function is provided as an instruction because it is required for technical reasons or for efficiency (e.g. disable interrupts or block transfer).

Control Transfers

The Mesa architecture supports several types of transfers of control, including procedure call and return, nested procedure calls, coroutine transfers, traps and process switches, using a single primitive called **XFER** [1]. In its simplest form, **XFER** is supplied with a destination *control link* in the form of a pointer to a local frame; **XFER** then establishes the context associated with that frame by loading the processor state registers: the **PC** and global frame pointer **GF** are obtained from the local frame, and the code base **CB** is obtained from the global frame. Most control transfer instructions perform some initial setup before invoking the **XFER** primitive; some specify action to be taken after the **XFER**. If after the **XFER** we add code to free the source frame, we have the mechanism for performing a procedure return. On the other hand, if we add code before the **XFER** to save the current context (only the **PC**), we have the basic mechanism to implement a coroutine transfer between any two existing contexts.

A process switch is little more than a coroutine transfer, except that it may be preemptive, in which case the evaluation stack must be saved and restored on each side of the **XFER**. In the Mesa architecture, we have also added the ability to change the main data space on a process switch (see the next section).

The procedure call is the most interesting form of control transfer in any architecture; it is complicated by the fact that the destination context does not yet exist, and must be created out of whole cloth. We represent the context of a not-yet-executing procedure by a control link called a *procedure descriptor.* It must contain enough information to derive all of the following:

The global frame pointer of the module containing the procedure,

The address of the code segment of the module,

The starting **PC** of the procedure within the code segment, and

The size of the frame to allocate for the procedure's local variables.

Note that in the case of a local call within the current module, only the last two items are needed; the first two remain unchanged.

It is desirable to pack all of this information into a single word, and at the same time make room for a tag bit to distinguish between local frames and procedure descriptors, so the two can be used interchangeably. Then, at the Mesa source level, a program need not concern itself with whether it is calling a procedure or a coroutine.

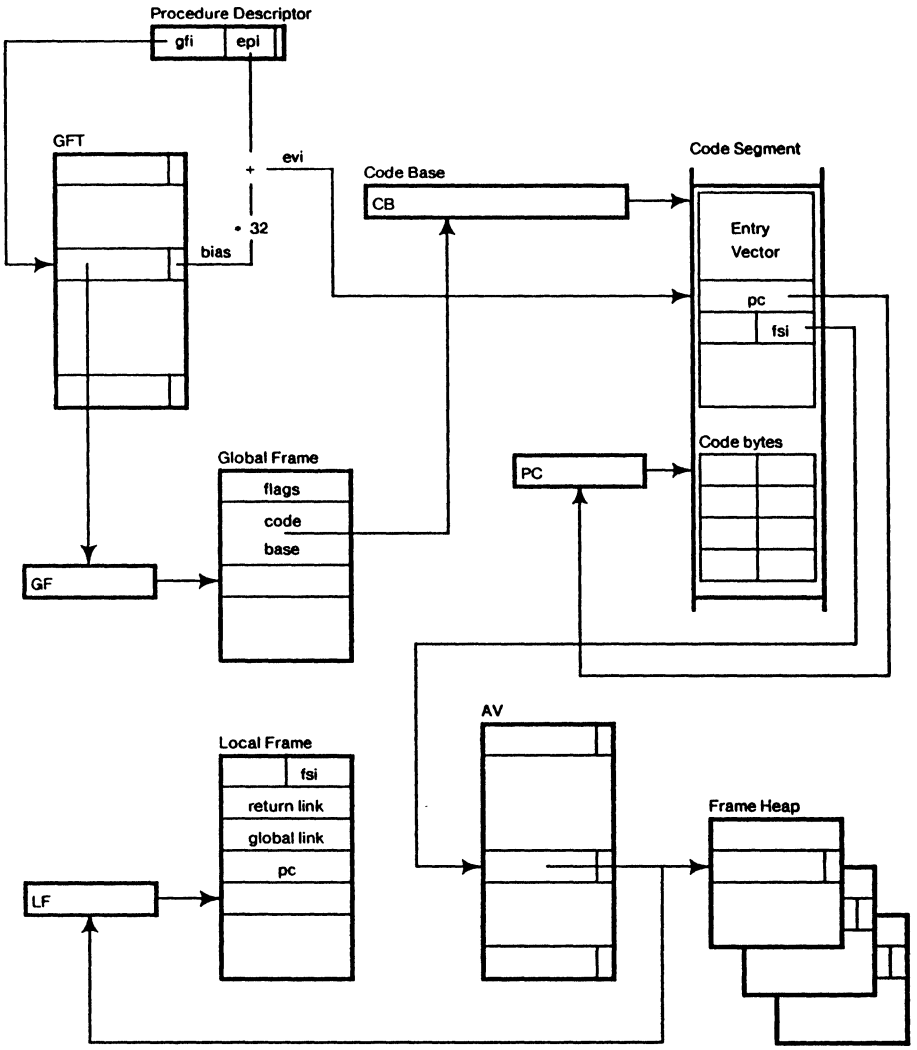

Figure 3. Procedure Calls

The obvious representation of a procedure descriptor would include the global frame address (sixteen bits), the code segment address (thirty-two bits), the starting **PC** (sixteen bits), and the local frame size (sixteen bits), for a total of eighty bits. We use a combination of indirection, auxiliary tables, and imposed restrictions to reduce this to the required fifteen bits, leaving one bit for the frame/procedure tag (refer to Figure 3).

We eliminate the code segment address by noticing that it is available in the global frame of the destination module, at the cost of a double word fetch.

We replace the **PC** and frame size by a small (five bit) *entry point index* into a table at the beginning of each code segment containing these values for each procedure. This costs another double word fetch, and limits the number of procedures per module to a maximum of thirty-two. (By an encoding trick, we will increase this to 128 later.)

We replace the global frame pointer by a ten bit index into an **MDS**-unique structure called the global frame table (**GFT**); it contains a global frame pointer for each module in the main data space. This costs one additional memory reference per **XFER** and limits the number of modules in an **MDS** to 1024 and the number of procedures in an **MDS** to 32,768.

We obtain our tag bit by aligning local frames to at least even addresses; the low order bit of all procedure descriptors is one.

To increase the maximum number of procedures per module, we first free up two bits in each entry of the global frame table by aligning all global frames on quad

word boundaries. We use these two bits to indicate that the entry point index should be increased by 0, 32, 64, or 96 before it is used to index the code segment entry vector. Of course, this requires multiple entries in the global frame table for modules with more than thirty-two procedures.

So, XFER's job in the case of a procedure call is conceptually the same as a simple frame transfer, except that it must pick apart the procedure descriptor and reference all the auxiliary data structures created above. It also needs a mechanism for allocating a new local frame, given its size.

As mentioned above, local frames are allocated from a heap rather than a stack, so that a pool of available frames can be shared among several processes executing in the same MDS. We organize this pool as an array of lists of frames of the most frequently used sizes; each list contains frames of only one size. Rather than actual frame sizes, the code segment entry vector contains frame size indexes into this array, called the allocation vector, or AV (see Figure 3).

Assuming that a frame is present on the appropriate list, it costs three memory references to remove the frame from the list and update the list head. This scheme requires that the frame's frame size index be kept in its overhead words, so that it can be returned to the proper list; it therefore requires four memory references to free a frame. Again we take advantage of the fact that frames are aligned to make use of the low order bits of the list pointers as a tag to indicate an empty list. There is also a facility for chaining a list to a larger frame size list.

In the (rare) event that no frame of the required size (or larger) is available, a trap to software is generated; it may resume the operation after supplying more frame storage. Of course, the frequency of traps depends on the initial allocation of frames of each size, as well as the calling patterns of the application; this is determined by the obvious static and dynamic analysis of frame usage.

Calling a nested procedure involves additional complexity because the new context must be able to access the local variables of the lexically enclosing procedure. The semantics of procedure variables in the Mesa language dictate that the caller of a nested procedure cannot be aware of its context or depth of nesting; all of the complexity must be handled by the called procedure. The implementation of this is beyond the scope of this paper.

Concurrent Processes

The Mesa architecture implements concurrent processes as defined by the Mesa programming language for controlling the execution of multiple processes and guaranteeing mutual exclusion [2].

The process implementation is based on queues of small objects called Process State Blocks (PSBs), each representing a single process. When a process is not running, its PSB records the state associated with the process, including the process's MDS and the local frame it was last executing. If the process was preempted, its evaluation stack is also saved in an auxiliary data structure; the evaluation stack is known to be empty when a process stops running voluntarily (by waiting on a condition or blocking on a monitor). The PSB also records the process's priority and a few flag bits.

When a process is running, its state is contained in the evaluation stack and in the processor registers that hold pointers to the current local and global frames, code segment and MDS. An MDS may be shared by more than one process or may be restricted to a single process. All of these processor registers are modified when a process switch takes place.

Each PSB is a member of exactly one process queue. There is one queue for each monitor lock, condition variable, and fault handler in the system. A process that is not blocked on a monitor, waiting on a condition variable, or faulted (e.g. suspended by a page fault) is on the ready queue and is available for execution by the processor. The process at the head of the ready queue is the one currently being executed.

The primary effect of the process instructions is to move PSBs back and forth between the ready queue and a monitor or condition queue. A process moves from the ready to a monitor queue when it attempts to enter a locked monitor; it moves from the monitor queue to the ready queue when the monitor is unlocked (by some other process). Similarly, a process moves from the ready queue to a condition queue when it waits on a condition variable, and it moves back to the ready queue when the condition variable is notified, or when the process has timed out. The instruction set includes both notify and broadcast instructions, the latter having the effect of moving all processes waiting on a condition variable to the ready queue.

Each time a process is requeued, the scheduler is invoked; it saves the state of the current process in the process's PSB, loads the state of the highest priority ready process, and continues execution. To simplify the task of choosing the highest priority task from a queue, all queues are kept sorted by priority.

In addition to normal interaction with monitors and condition variables, certain other conditions result in process switches. Faults (e.g. page faults or write-protect faults) cause the current process to be moved to a fault queue (specific to the type of fault); a condition variable associated with the fault is then notified. An interrupt

(from an I/O device) causes one of a set of preassigned condition variables to be notified. Finally, a timeout causes a waiting process to be moved to the ready queue, even though the condition variable on which it was waiting has not been notified by another process.

Conclusions

The Mesa architecture accomplishes its goals of supporting the Mesa programming system and allowing significant code size reduction. Key to this success is that the architecture has evolved in conjunction with the language and the operating system, and that the hardware architecture has been driven by the software architecture, rather than the other way around.

The Mesa architecture has been implemented on several machines ranging from the Alto [6] to the Dorado [3], and is the basis of the Xerox 8000 series products and the Xerox 5700 electronic printing system. The ability to transport almost all Mesa software (i.e. all except unusual I/O device drivers) among these machines while retaining the advantages of the semantic match between the language and the architecture has been invaluable. The code size reduction over conventional architectures (which averages about a factor of two) has allowed considerable shoehorning of software function into relatively small machines.

Acknowledgments

The first version of the Mesa architecture was designed and implemented by the Computer Science Laboratory of the Xerox Palo Alto Research Center. Butler Lampson was responsible for much of the overall design and many of the encoding tricks. Subsequent development and maintenance have been done by the Systems Development Department of the Office Products Division. Chuck Geschke, Richard Johnsson, Butler Lampson, Roy Levin, Jim Mitchell, Dave Redell, Jim Sandman, Ed Satterthwaite, Dick Sweet, Chuck Thacker, and John Wick have all made major technical contributions.

References

[1] Lampson, B., Mitchell, J., and Satterthwaite, E. On the transfer of control between contexts. *Lecture Notes in Computer Science 19,* (1974).

[2] Lampson, B. W. and Redell, D. D. Experience with processes and monitors in Mesa. *Comm. ACM 23,* 2 (Feb. 1980), 105-117.

[3] Lampson, B. W. *et. al.* The Dorado: A high-performance personal computer—three papers. Tech. Rep. CSL 81-1, Xerox Palo Alto Res. Ctr., 1981.

[4] Mitchell, J. G., Maybury, W., and Sweet, R. Mesa Language Manual. Tech. Rep. CSL 79-3, Xerox Palo Alto Res. Ctr., 1979.

[5] Sweet, R. E. and Sandman, J. G. Empirical Analysis of the Mesa Instruction Set, ACM Symposium on Architectural Support for Programming Languages & Operating Systems, March 1982.

[6] Thacker, C.P. *et. al.* Alto: a personal computer, in *Computer Structures: Readings and Examples,* Second edition, Sieworek, Bell, and Newell, Eds., McGraw-Hill, 1981. Also available as Tech. Rep. CSL 81-1, Xerox Palo Alto Res. Ctr., 1981.