A High Performance VME Processor Card When 32-Bit Super-Micros Can't Cut It

Phil Burnley
Xycom, Inc.
Northampton, England

and

Thomas Harkaway
Xycom, Inc.
Saline, Michigan

Abstract

Using High Level Languages (HLL) for demanding real time applications is usually not possible on conventional microprocessors. Typically the HLL is only used to program the user interface portion of the application while assembly language is used for the time critical portions. Because the advantages of HLL programming are so great, various approaches to this problem are taken including the use of multiple processors in the same system and using more expensive higher performance processors than the application really requires. This paper discusses an alternate approach; a VMEbus processor which was designed for programming real time applications completely in a HLL.

1. INTRODUCTION

The benefits of programming in High Level Languages (HLLs) are many and well known, yet despite the widespread availability of HLLs for conventional processors, in many applications, engineers are still forced to undertake substantial developments entirely in assembler language. This is especially true in demanding real-time applications where execution speed is of the utmost importance and even the most efficiently compiled HLL cannot compete with carefully hand-coded assembler. Furthermore, real-time applications often require input/output handling and interrupt servicing, features that are poorly supported in most HLLs.

It has been widely accepted as a fact-of-life that a program written in a HLL will incur a performance penalty, against the same program coded in assembler, but why should this be the case? Is it possible that the architecture of conventional processors is imposing a handicap upon HLLs which could be avoided with an alternative architecture?

This paper answers these questions by outlining the rationale behind a new two-stack architecture that has found commercial realization in the XYCOM XVME-616. This processor was designed for programming real-time application in HLLs. The three design objectives were:

1) Develop an architecture optimized for the execution of HLLs

2) Very high speed

3) Micro-programmable to allow the creation of Application Specific CPUs

2. HLLs ON CONVENTIONAL PROCESSORS

An analysis of code generated by compilers for procedural high level languages, such as Pascal and C, quickly reveals a reliance at runtime upon one or more stack (LIFO) data structures. Stacks are needed for:

- storage of local variables so that, during recursion, a new set of locals may be easily created
- storage of subroutine return addresses to facilitate procedure calls and returns
- temporary storage during execution of complex arithmetic expressions.

It is easy to show that procedural HLLs which support local variables and recursion cannot be executed at all without at least one stack data structure. As a result it is not surprising to discover that compiler writers often write their compilers for a 'virtual stack machine', as embodied for example in Pascal P-code.

Conventional, register based, processors, such as the 8086 and 68000 families, do not have integral hardware stacks and, as a consequence, must implement these essential data structures as reserved areas of the main memory. These areas are accessed in a simple LIFO manner, controlled by dedicated memory address registers within the CPU acting as stack pointers. It becomes clear in analysis of HLL program execution that this method of data organization is the principle cause of the poor performance of HLLs on these processors.

Most conventional processors implement special instructions such as PUSH

(decrement stack pointer and store data in location addressed by stack pointer) and POP (fetch data from location addressed by stack pointer and increment stack pointer) but these instructions do not overcome the overhead created by the large number of addresses and data that must be transferred between the processor and the main memory where the stack structure exists.

This can be easily demonstrated by the following examples.

2.1 Local Variables

The treatment of local variables on a conventional processor (Figure 1) is illustrated by considering the following fragment of C code:

```
int a,b,c;
a=b+c;
```

The three variables a, b, and c are locals therefore must be stored within a stack data structure in main memory. The arithmetic addition of b and c can only be carried out by the hardware Arithmetic Logic Unit (ALU), within the CPU, and so b and c must be moved to the ALU via one or more of the CPU registers. After the addition the result, a must be moved back from the CPU to the stack; a sequence of operations achieved by the following 68000 assembler code,

```
MOVE b'(SP),D0
ADD  c'(SP),D0
MOVE D0,a'(SP)
```

This sequence assumes that the locals a, b and c are offset from the top of stack by a', b' and c'. The time taken to execute this sequence is typically 36 clock cycles and 9 data bus transfers on a 68000 processor. At least one third of this effort is wasted transferring the data between the stack and the ALU.

2.2 Subroutine Call and Return

A similar performance penalty is incurred in the execution of subroutine calls and returns. Before loading the program counter (PC) in the CPU with the address of the start of the called subroutine the current value of the PC must first be pushed onto a return address stack in main memory. At the end of the subroutine the return address must be popped from the return stack in main memory into the PC in the CPU. The physical separation of the program counter and the return address stack imposes a penalty of 2 main memory transfers just as the separation of the ALU and the local variable stack imposes penalties in the previous example.

The 68000 requires 34 clock cycles and 5 data bus transfers for the BSR/RTS pair of instructions. The equivalent call and return pair in a stack architecture machine would only need 2 main memory transfers.

The execution speed of a HLL program is known to be particularly sensitive to the time required for subroutine call and return. A program of average complexity on a conventional processor will often spend 25% of runtime simply executing calls and returns due to the frequency of these instructions and the great length of time they take to execute. This can become worse for complex systems because of the increased number of subroutine nesting layers employed. An improvement in this area alone will dramatically improve the performance of HLL programs.

3. A TWO STACK ARCHITECTURE

The examples reviewed above, and others, suggest that the conventional register architecture is far from ideal for the execution of procedural HLLs and that a substantial performance advantage may be gained by implementing one or more stack data structures directly in hardware within the CPU.

Examination of the relative merits of one-stack versus two-stack architectures strongly favors a two-stack approach with one stack coupled to an ALU and reserved for local variable storage (the Arithmetic Stack), and the other coupled to the PC and reserved for return addresses (the Return Stack). The advantage of a two-stack approach is best illustrated by examining the requirements of parameter passing, via a stack, to a subroutine,as would be implemented in a HLL program. In a one-stack machine parameters placed on the Top-of-Stack (TOS) will no longer be on TOS after the subroutine call, but will be buried underneath the subroutines return address on the same stack. In a two-stack machine local variables and subroutine operands will retain the same relative position on the arithmetic stack despite any number of nested subroutine calls, and therefore the code to access locals can be simpler and more efficient.

A two-stack architecture with main memory is shown in Figure 2. The CPU now has two main units, the Arithmetic Stack Unit (ASU) and the Return Stack Unit (RSU). The ASU consists of a hardware stack with its own stack pointer (SP) and an ALU. The ALU is deliberately shown inside the arithmetic stack since it is blended within the stack in a manner that minimizes data movement between the stack and the ALU. For instance, TOS operands are always available at the inputs to the ALU. Thus the Arithmetic Stack Unit is neither just a stack nor just an ALU; hence it's terminology. Likewise the RSU consists of a hardware stack, with it's stack pointer (RP) and the program counter which points to the program code in the main memory, which for historical reasons, is known as the input pointer (IP).

Although the ASU and RSU are both part of the CPU, when coupled with main memory as shown in Figure 2, the complete machine has three independent memory structures, each fulfilling a specific functional requirement of HLLs.

- The ASU for local variables and temporary storage
- The RSU for subroutine and interrupt return addresses
- Main memory for program code and global (static) variables.

Not only does this architecture substantially reduce the number of data transfers from main memory during HLL program execution but the three memory structures may operate in parallel so that, for example, arithmetic on local variables may be carried out at the same time as a subroutine call.

4. PROGRAMMING A TWO STACK MACHINE

The choice of an assembly language for the two-stack machine described above should be optimized for the execution of HLLs, in this case optimized for the execution of C code, as the hardware is optimized for the execution of HLL programs. Why not base the assembly language on an HLL in the first place?

As described before, several procedural HLLs conceive of a virtual stack machine in their implementation. The closest match between a true HLL and the physical architecture described here is provided by FORTH which, in addition, provides a highly interactive programming and development environment that is ideal for the rapid development of real-time applications. There are some aspects of C language implementation that are not well served by FORTH and so, for these, there are additional instructions developed within the microprogram to provide efficient C language operation. The complete 'assembly language' is now made up of HLL primitives. There are seven classes of primitive operation as shown below;

Class	No. of primitives
Stack	18
Logical	13
Comparison	7
Control	17
Arithmetic	11
Memory	10
Literal	9

Typical primitives for the Comparison, Arithmetic, and Control operations are listed below.

COMPARISON

Name	t-states
U<	4
0<	4
0=	3
<	10
=	4
>	8
COMP	9+19n, where n is the number of comparisons made.

ARITHMETIC

Name	t-states
+	3
-	3
1+	3
1-	3
2+	4
2-	4
D+	11
DNEGATE	7
NEGATE	3
*STEP	7 or 11
/STEP	10,13, or 14

CONTROL

Name	t-states?
?BRANCH	6
BRANCH	6
NOOP	3
EXECUTE	3
LEXECUTE	9
(+LOOP)	11
(DO)	6
(LOOP)	8
(OF)	7
I	6
J	10
EXIT	4
LEXIT	7
THREAD	4
IM!	3
IM@	4
RESET	6

Examples of the implementation of a C program in the XVME-616 primitives are shown below:

C code:

```
static int a,b;
if(a<10)b=a*2;
```

Primitive code:

```
a @
10 < ?BRANCH
  a @ 2* b !
THEN
```

where:

```
 a - push address of a onto ASU
 @ - fetch value at address on TOS
10 - push 10 onto the stack
 < - compare TOS (10) with NOS (a) and
     leave true/false flag on the stack
?BRANCH -
     if flag on TOS is true then
     execute instructions up to THEN,
     otherwise continue from THEN
2* - multiply TOS by 2 (shift left 1
     bit) and leave result on TOS
 b - push address of b onto stack
 ! - store NOS at address indicated by
     TOS
```

Execution time:

```
     36 t-states if false = 1.8 uSecs
     76 t-states if true  = 3.8 uSecs
```

In this example the Primitive notation is little more than a rearrangement of the original C. The longest primitive in this sequence is 10 t-states.

In the second example we look at the same C code sequence reviewed earlier;

C code:

```
int a, b, c;
a=b+c;
```

Primitive code:

```
OVER OVER + 2UNPICK
```

Execution Time 19 t-states = 950 nSecs

Here a, b, and c are locals and rapidly accessible with the arithmetic stack manipulation primitives of which OVER and UNPICK are typical. OVER takes only 3 t-states to execute and copies the current NOS to the TOS. '+' adds the TOS and NOS leaving the result on NOS, 2UNPICK places the current TOS 2 places further down the stack leaving the stack as c, b, a from the top.

5. THE XVME-616 CPU

The practical implementation of this two-stack CPU architecture is shown in Figure 3. The XVME-616 CPU is implemented on a double high Eurocard with SSI and MSI Advanced Schottky TTL devices and a clock speed of 20MHz which results in a maximum execution speed of 6.67 million HLL instructions per second.

The two stacks are realized with 35 nSec SRAM, and both are 16 bits wide and 1024 words deep. The execution sequencer contains 512 56-bit words of microcode. The bus structure is based on a 24-bit address, 16-bit bi-directional data bus with synchronous operation at 6.67 million bus read or write cycles per second.

The instruction set is microprogrammable and extensible and at this time consists of 85 primitives associated with the execution of C language programs. In addition there are a number of primitives associated with the hardware and the VME bus interface. The architecture allows the CPU to be tailored not only to the execution of HLLs but also, by creating new microcoded primitives, to applications requiring specific data manipulation or processing procedures. In this fashion the processor may become an Applications Specific CPU.

6. THE XVME-616 PROCESSOR

The XVME-616 CPU is the heart of the XVME-616 VMEbus processor which is shown in total in Figure 4. The processor consists of two cards, the XVME-616 CPU card and the XVME-616 Memory, I/O, and VME (MIOV) card. These two cards are connected by a synchronous 6.67 Mhz bus. The MIOV card supports a number of functions, including:

- A24:D16 VMEbus master interface
- Peripheral bus interface which is used to access the on-board I/O devices
  - 2 RS232 serial Ports
  - real-time clock
  - 2 EPROM sites
- 512 byte Boot ROM
- 128K of 65nSec SRAM main memory
- Watchdog circuit for failsafe monitoring of critical applications
- Option One Arbiter for the VMEbus with a VMEbus timeout counter
- 8 level interrupt handler
  - Local I/O
  - Timer/Clock
  - Abort switch
  - VMEbus signals BCLR*, BERR*, and ACFAIL*
  - VMEbus interrupts, IRQ1*-IRQ7*.

The XVME-616 processor is provided with a complete multitasking operating system and debugger.

A novel aspect of the implementation is that the peripheral bus and the VMEbus are synthesized and arbitrated with microcode. The two buses appear to the XVME-616 CPU as simple memory mapped I/O ports. This approach arises from the large disparity in speed between the high speed bus and the VMEbus and it would be unacceptable to degrade the performance of the CPU while waiting for address arbitration logic and synchronization logic to permit accesses. A number of high level primitives are provided to access and control all the processor facilities including the VMEbus.

7. CONCLUSION

The power of the processor is illustrated by reviewing some typical processor operations.

7.1 The high level language DO loop

The XVME-616 implements a run-time loop primitive, " (LOOP)", which maintains index and loop end values on the return stack, and performs a signed increment of the loop index, a comparison with the end value, and a conditional branch in 8 clock cycles or 400 nSecs, resulting in 2.5 million DO loops per second. The much less powerful 68000 decrement and branch if not zero instruction requires 10 clock cycles or 1 uSec in a 10Mhz, 0-wait state design.

7.2 Context Switch

A multi-tasking context switch on the XVME-616 simply requires the two stack pointers, SP and RP, be saved and the reloaded to point to the stack partition for the newly activated task. There are no other registers to be saved or restored and so a full context switch can be executed in 5 uSecs. The equivalent operation in PDOS running on a 68000 requires 140 microseconds and on a 20Mhz 68020 requires 30 microseconds.

7.3 High Level String Compare

The COMP primitive compares two byte strings, of length 'n', held in main memory for equality. The strings are pointed to by parameters on the stack and the length is defined by a parameter also on the stack. The primitive returns a the value (n-x), on the stack, where x equals the number of true comparisons. This primitive is a complete implementation of the C langauge, string compare DO loop and takes only 9 + 19n t-states and processes character comparisons at a rate in excess of 1 million bytes per second.

8. SUMMARY

The XVME-616 is a registerless, two-stack machine, specifically designed for the execution of High Level Languages and C language in particular. It represents a fundamental and significant change in the design of processor architectures. Even more fundamental than RISC (less of the same) or Parallel (more of the same) approaches. This processor can be regarded as the first in a series of machines designed to meet the needs of demanding, real-time applications without the heartache of machine level coding.

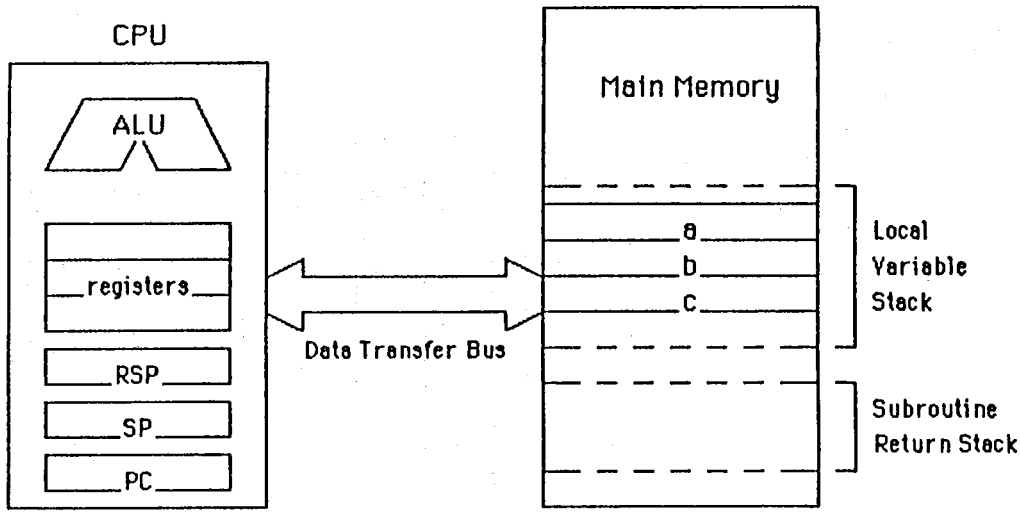

Figure 1
Stack Structures in a Conventional Architecture

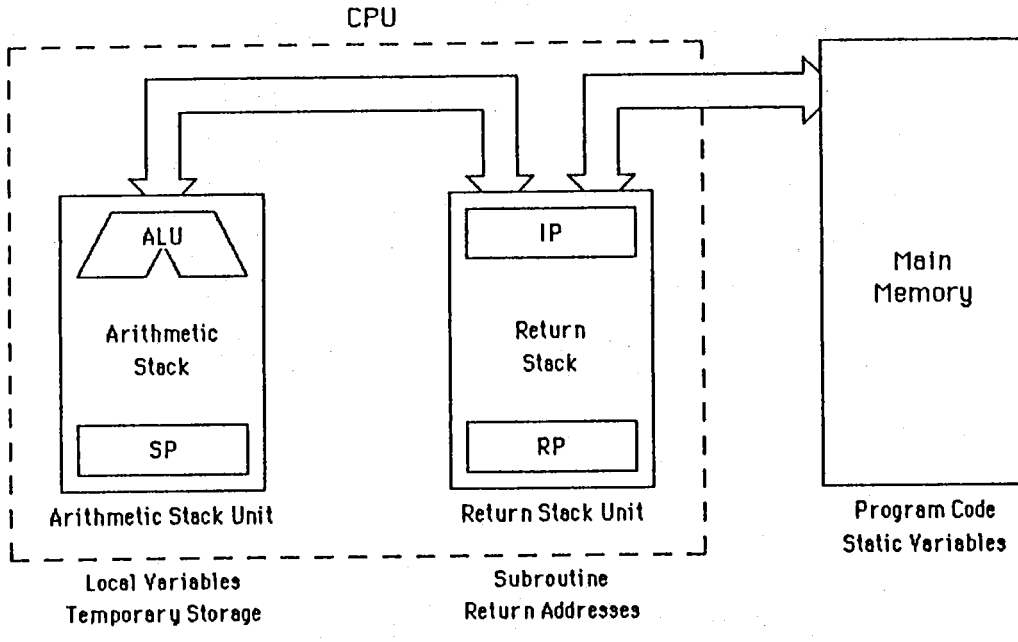

Figure 2
A Two-stack Architecture

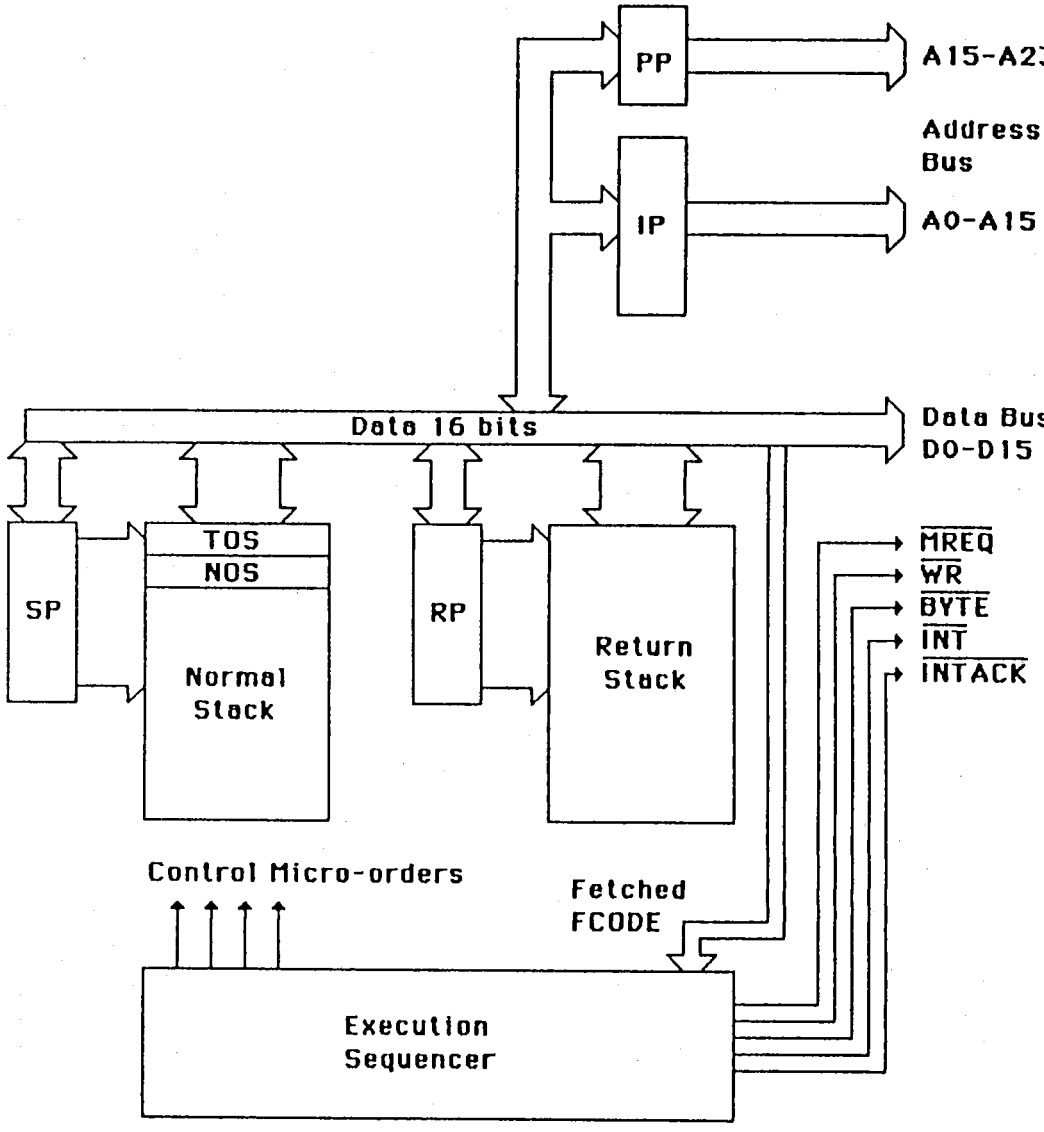

Figure 3
XVME-616 CPU Block Diagram

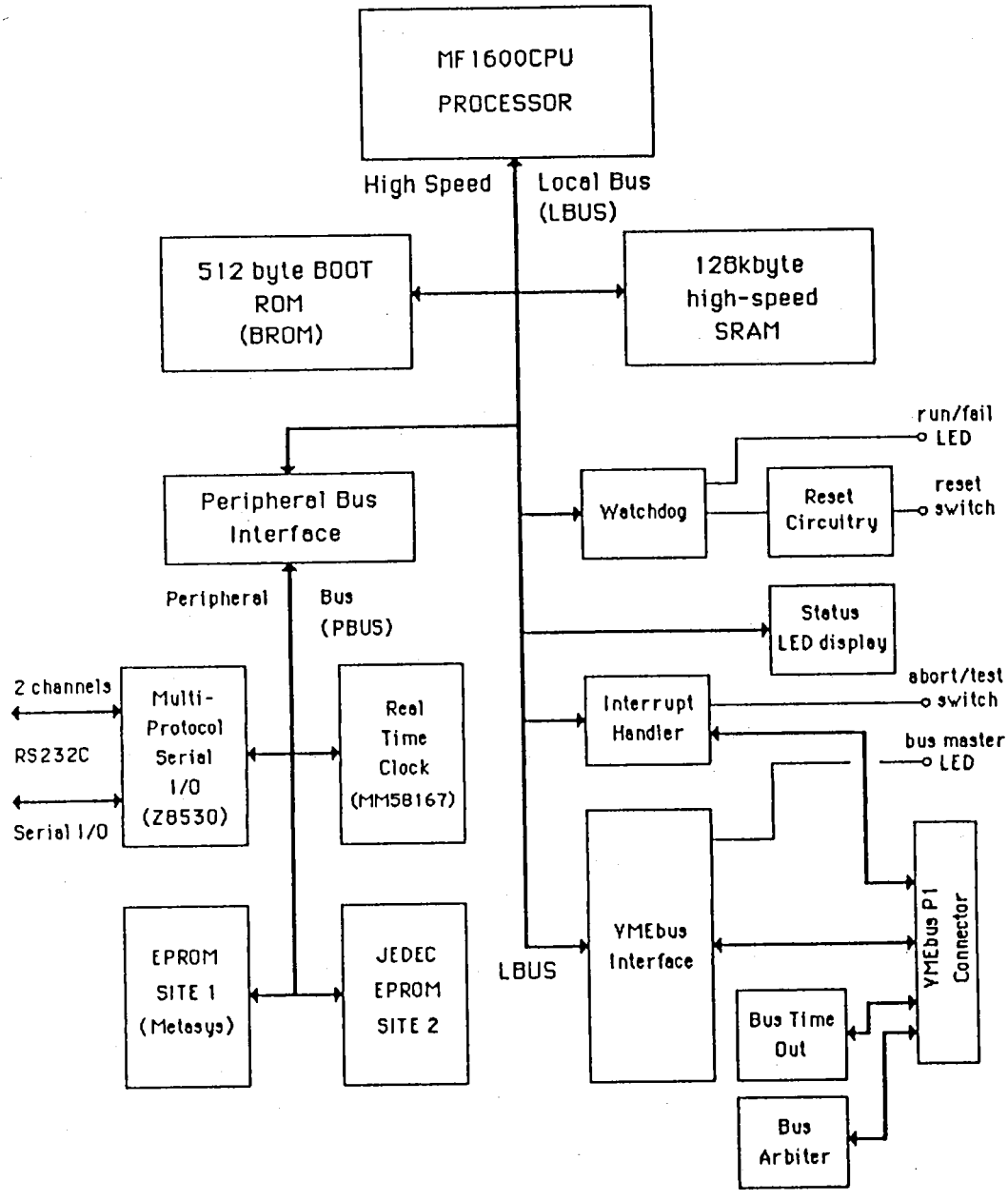

Figure 4
XVME-616 Processor Block Diagram