A Stack-Frame Architecture Language Processor

R. D. Dixon
Computer Science Department
Wright State University

Abstract

An architecture for a fast 32-bit processor which is designed to support a variety of languages for use in a real-time setting is described. The processor is based on the augmentation of a simple CPU with a set of smart auxillary memories. These memories allow stack mode and direct access over a fast bus which is not shared with main memory. The possibility of hardware support for context switching in this design makes it a good candidate for demanding imbedded systems such as flight computers.

Introduction

Producing reliable real-time systems is one of the most difficult and important problems facing computer scientists. As techniques in non-real-time areas yield more impressive results, the expectations for imbedded systems increase. Much of the progress in programming complex tasks has come from better programming languages, and better software engineering techniques. The difficulty in transfering these techniques to real-time applications is that language and program structure can have profound effects on response times.

This paper is a report on a research effort to develop a flexible system approach to this problem. Many researchers in the Forth community have been successful in handling real-time systems where others have failed. Our research has in part been a matter of following the wide variety of approaches being taken and attempting to synthesize them. Starting from the top, we are looking for a language processor. By that we mean a system that can effectively process many languages, from natural language expressions to higher level programming languages to direct expression of machine operations. We need to be able to deal with different semantic models as well as different syntax and we need to produce systems that do not lose the name real-time, even though some operations, such as compilations, may be time consuming.

Starting at the bottom, we have concluded that a fast machine which contains the basic Forth operations as a subset of its instruction set will assure an assembly language (i.e. a superset of Forth) which will be easy to use. The experience with imbeddings of very high level languages in Forth assures us that this is a reasonable base on which to build.

The middle level languages such as C, Ada, Pascal, and Fortran have been less well supported from a pure stack machine. The standard approach is to provide register pointers to pieces of context, called frames. We have chosen to provide direct

hardware support for combined stack-frame structures by special purpose VLSI chips.

Any system that uses interrupts is really a multi-process system with the processor, its registers, and any other context mechanism used by more than one program, as shared resources. It is the cost of protecting the information in these shared resources that limits performance of real-time systems.

Several approaches can be used to confine the damage:

1. Limit shared resources.

2. Restrict the occurance of process switches.

3. Provide hardware support for process switches.

Our approach is to partition the processor into two parts. The first manages the computation but carries pratically no status from one instruction to another. Thus we limit the shared resources. The second part of the processor is all data: the stack-frames. They are implemented in VLSI, and communicate over a separate bus. This separate bus allows hardware support for context switches which do not follow a LIFO (a stack) protocol. This is done by controlling the select lines of the stack-frame chips.

In this paper we present the description of the machine, the SF1, and in a companion paper we present a description of the VLSI implementation.

The SF1 is a 32-bit architecture with an instruction set that combines some features of RISC machines, stack machines, and micro-coded machines to achieve high performance. Included is a description of the SF1 architecture, the core instruction set, how the SF1 generates frames, conditional branching, interrupts, compatibility with C and block structured languages, and some extensions. We consider the SF1 as a point of departure for exploring 32-bit data processor architectures for imbedded systems.

Many of the ideas in this architecture are derived for conversations and papers given at previous Rochester Forth Conferences. Alan Winfield, Ron Goodman, Phil Koopman Jr., Glen Haydon and the producers of the Novix chip have all freely described their ideas. Lew Odette has been very helpful in providing understanding about support for very high level languages.

The SF1 Architecture

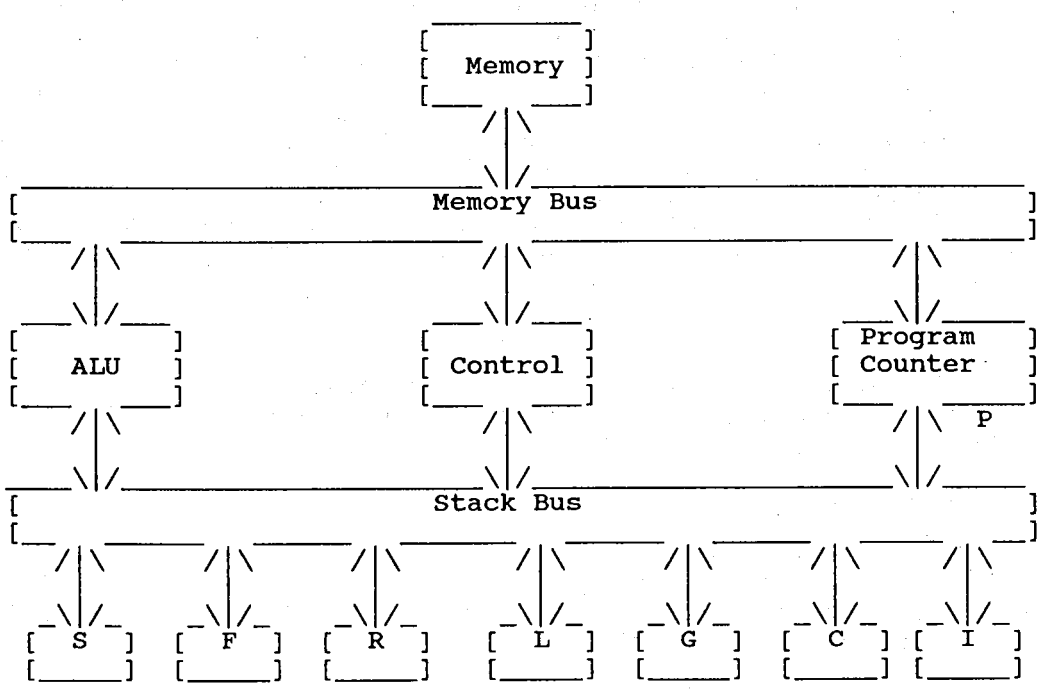
The current implementation of the SF1 has a memory bus and a main memory addressing space which contains both data and instructions. The addresses are 32-bits long but most addresses are word aligned and are thus multiples of four. Instructions are 32-bits long and all but main memory references execute in a single cycle whose length is determined by the speed of memory. An instruction is fetched every memory bus cycle and executed the

next. With a pipeline of depth two, a throughput of one instruction per cycle is obtained.

This implementation of the SF1 has eight accessible structures called stack-frames. These structures are connected to the ALU via a second bus called the stack bus. This bus operates at twice the speed of the memory bus, having a read cycle and a write cycle for every read cycle of the memory bus. Every instuction is read from main memory and causes a read from one stack-frame and a write to another. The ALU operation is overlapped with the stack bus write.

Memory data fetches and deposits are in effect DMAed with the other operations and so occupy the whole machine for exactly one cycle. Branches, which normally empty the pipeline, are restricted and optimized so that in each case the effective throughput is one instruction per memory bus cycle.

The following is a block level diagram of the current implementation of the SF1. The chip breakdown is roughly along the lines of this diagram.



The SF1 core instruction set is given in a later section but we describe the "feel" of it here. The instructions primarily move data from one stack-frame to the arithmetic and logic unit (ALU), and move the result of the previous operation to another stack frame while performing the current operation. Stack-frames are structures which have both direct access (memory or register like) and stack access (push, pop). The ALU has a single register

called TOS. A typical instruction,

```
ADD  F(10) S(<) ;
```

pushes a copy of the TOS onto the stack-frame S, and adds the contents of F(10) to TOS and places the result in TOS.

Access to main memory requires getting the address into TOS, so that complicated addressing requires several instructions.

Although jumps are optimized, conditionals are not and so they also must be constructed from other instructions.

A word address in the instruction stream is interpreted as a jump-to-subroutine (because the two low order bits are zero) just as in Forth. Thus, it is easy to extend the apparent instruction set of the machine by subroutine-instruction-augmentations (SIA).

The SF1 Core Instruction Set

Registers

The abstract SF1 machine has three registers that are carried over from one instruction to another and each is refreshed on each instruction. The first is called TOS (which stands for "top of stack") The result of every ALU operation is written to TOS. It is also one of two operands to every ALU operation. The PC register is the program counter. IR is the instruction register. Every instruction is one word (32-bits) long. There are three temporary registers, ALUI, which is the second operand to every ALU operation, VALUE which is used to hold the old value of TOS for certain operations, and PCJ which is used to save the value of PC during the execution of the call subroutine instruction.

Main Memory

Main memory (MM(address)) is byte addressed with 4 byte words aligned so they are addressed on multiples of 4. The core instruction set has only word (32-bit) operations.

Stack-Frames

The SF1 can address structures called stack-frames. We use x as a name for a generic stack-frame. We need a minimum of four stack-frames to define the core instruction set, nominally we name eight, but more can be used.

1. Frame Addressing

   x may be addressed as a frame by giving a displacement in the instruction. It is then a random access structure with values x(0), x(1), . . . ,x(i), . . .

   The maximum value for i is implementaion defined. Nominally we use 16K.

2. Stack Addressing

x may be accessed as a stack to be pushed by the destination, x(<), which has the effect:

For all i x(i)->x(i+1), TOS->x(0).

x may also be addessed as a stack to be popped by the source, x(<), which has the effect:

x(0)->ALUI, For all i, x(i+1)->x(i).

Three of the stack frames are special purpose.

C The constant stack, C, is a special notation to allow the passage of immediate data from the instruction to the ALU.

For all i, C(i)=i.

I The I/O frame is used to address I/O devices.

P P is always addressed as a stack, but only P(0) is defined and it is the PC.

Instructions

The syntax for instructions in bnf is:

<instructions>:=<sf-instructions>|<mm-instructions>|<jumps>

<sf-instructions>

The first category, which includes most instructions, is:

<sf-instrutions>:=<sf-inst>";"|<sf-inst>"D;"

<sf-inst>:=<operation><source><destination>

<operation>:=<operation-result>|<operation-status>

<operation-status>:=<operation-result>"_ST"

<operation-result>:=LOAD|NOOP|ADD|SUB|SUBR|
SHIFTL|SHIFTRA|SHIFTRL|
AND|OR|EXOR

<source>:=x(displacement)|x(>)|x()

<destination>:=x(displacement)|x(<)|x()

x := S|F|R|L|G|C|I|P

Comment: Only one displacement may be used in an instruction. When x() is used, then the displacement from the other argument is used. If there is no displacement in the other argument then a displacement of 0 is used. Using C as a destination has the effect that no write to a stack-frame is done.

The semantics of these instructions can be given symbolically since the instructions are all the same length and the decoding is very regular. Here we use ":" to separate operations that are done concurrently. In cases where the results of some concurrent operations are inputs to others then we require that the updates be made after inputs are latched. We use ";" to separate sequential operations. Decoding of instructions is done from the value in IR retained from the previous instruction execution and is symbolized as decode(IR).

If IR contains <operation><source><destination>; then the execution is

```
decode(IR);

<source>->ALUI : MM(PC)->IR ;

PC+4->PC : TOS-><destination> : <operation>(TOS,ALUI)->TOS ;
```

As an example let TOS=4, S(0)=5, and execute

```
ADD  S(>) F(7) ;
```

Then F(7) becomes 4. TOS is 9, and S(0) gets the old value of S(1).

In the case where the <destination> is P, the transfer of TOS into the PC overrides the increment of the old PC.

The "D;" indicates a direct instruction which has a single change in the above semantics.

TOS -> <destination> is replaced by ALUI -> <destination>

Operations

The values computed by the ALU for the operations allowed in the syntax is given by the equations below:

```
LOAD(TOS,ALUI)=ALUI

NOOP(TOS,ALUI)=TOS

ADD(TOS,ALUI)=TOS+ALUI

     (Number representation is 32-bit, two's complement)

SUB(TOS,ALUI)=ALUI-TOS

SUBR(TOS,ALUI)=TOS-ALUI
```

```
SHIFTL(TOS,ALUI)=TOS*2

SHIFTRA(TOS,ALUI)=TOS/2

SHIFTRL(TOS,ALUI)=TOS/2 and 011...1   (bitwise)

AND(TOS,ALUI)=TOS and ALUI            (bitwise)

OR(TOS,ALUI)=TOS inclusive or ALUI    (bitwise)

XOR(TOS,ALUI)=TOS exclusive or ALUI   (bitwise)
```

For status operations, for example, ADD_ST, we return a status word in TOS instead of the result. This is always the result of the current operation. The status word always has the 32-bit form

00000000000000000000000vcnz00.

v,c,n,c stand for overflow, carry, negative, zero respectively. They are always set in the standard way.

Main Memory Instructions

There are only two ways to access main memory, fetch and store, but they can be done with any other instruction.

<MM-instructions>:=<sf-inst>("@"|"D@") | <sf-inst>("->"|"D->")

The operation of a memory fetch instruction <sf-inst>@ is:

```
decode(IR);

<source>->ALUI : MM(PC)->IR ;

PC+4->PC : TOS-><destination> : <operation>(TOS,ALUI)->TOS ;

MM(TOS)->TOS ;
```

Thus, as an example, LOAD C(1000) S(<) @ pushes the old TOS onto S and fetches the contents of MM(1000) into TOS.

The operation of a memory store instruction <sf-inst>-> is:

```
decode(IR);

<source>->ALUI : MM(PC)->IR ;

PC+4->PC : TOS-><destination> : <operation>(TOS,ALUI)->TOS :
                                          TOS->VALUE ;
```

```
VALUE->MM(TOS) ;
```

As an example here suppose TOS=3. Then execute

```
LOAD C(1000), S(<) -> .
```

MM(1000) becomes 3, 3 is pushed on S, and 1000 remains in TOS.

As before, in the direct instructions, "D@" and "D->", the value for the stack bus write comes from ALUI rather than TOS.

Jump Instructions

The syntax of jump instructions is

```
<jumps>:= JSR <word-address>| JMP <word-address> .
```

The execution of the instruction JSR <word-address> is

```
decode(IR) ;
PC->PCJ : <word-address>->PC;
PCJ->ALUI : MM(PC)->IR ;
PC+4->PC : TOS->S(<) : ALUI->TOS ;
```

The execution of the instruction JMP <word-address> is

```
decode(IR);
<word-address>->PC ;
MM(PC)->IR ;
PC+4->PC : TOS->TOS ;
```

The JSR can be omitted so that if we write the address XXX in the instruction stream, it means JSR XXX.

The SF1 has no return instruction. The instruction

```
LOAD S(>) P(<) ;
```

serves that purpose. Because of pipelining, the next instruction following that will be executed before the branch.

Instruction Coding

The instruction semantics for the SF1 were given without reference to the mapping between the instructions and the binary coding of those instructions. This allows flexibility in the mapping. A sample coding is is given in the accompaning paper.

How the SF1 Generates Frames

A useful organization that we adopt is that writing to I(2) exchanges the S and F stack-frames. This can be done in a single cycle by changing control states. Suppose we make the convention that the arguments for a procedure are placed on the S-stack before calling the routine. The called routine then allots local variables on the stack and makes a frame so that local variables and parameters may be addressed relative to that frame. Before exit the subroutine deletes the local variables and parameters, places a single return value on the stack, restores the previous frame and returns. The code in a subroutine AAA which does this is

```
AAA: NOOP S() I(2);        Switch the role of S and F
     NOOP C(0) F(<) D;     Create a temporary variable
     NOOP C(0) F(<) D;     Create another one
                           Begin the subroutine body
          ...
                           What follows are sample
                             instructions
     LOAD  G(77) S(<);     Using the stack leaves the return
                            address under and does not affect
                            the frame
     LOAD  F(1) S(<);      F(1) is the first variable declared
     LOAD  S(>) F(0);      F(0) is the last
     LOAD  F(2) S(<);      F(2) is the last parameter
          ...
                           Suppose TOS contains a return value
                           End of the subroutine body
     NOOP  F(>) C();       Begin deleting variables
     NOOP  F(>) C();
     NOOP  F(>) C();       Begin deleting parameters
     NOOP  S(>) P(<) D;    Send return address to P
                           The return value is still in TOS
     NOOP  S() I(2);       Restore S and F
```

The overhead for a subroutine call using this convention is 2 cycles each for the entry and exit and 2 cycles for each parameter and variable.

Conditional Branching

The SF1 has no conditional branches so they must be composed of other instructions. Since we are not tied to any structure we may optimize for the particular type of language being used. We give here an example of some skip on condition instructions.

First assume FALSE=0, TRUE=4. We define an SIA (subroutine instruction augmentation) skip_on_true, skip_on_false as routines which expect a boolean (TRUE or FALSE) in TOS. These routines delete the boolean and skip the next instruction on the appropriate condition.

```
skip_on_true :
```

```
                                    The return address is in TOS
                                    The boolean is in S(0)
        ADD  S(>) C()   ;           Add the boolean (4 or 0 ) to
                                    the return
        LOAD S(>) P(<)  ;           return from subroutine
        NOOP S()  C()   ;           noop to execute before the
                                    return
```

This routine is used as follows:

```
        TEST which generates a boolean
        skip_on_true
        INSTRUCTION which might be skipped
        CONTINUATION
```

The skip_on_true routine, including the call takes 4 cycles.
Similarly we define the routine

```
skip_on_false :
        ADD  C(TRUE) C();       Add 4 to the return address
                                to make that the default
        SUBR S(>) C()   ;       Use the boolean to decide
        LOAD S(>) P(<)  ;       return
        NOOP S()  C()   ;       trailing noop
```

This routine takes 5 cycles.

An IF-THEN-ELSE structure using these constructs has an overhead of 5 cycles.

Suppose we wish to generate a boolean value based on the truth of the statement S(0)<TOS. For every such condition we keep a table in low memory such as:

```
zero_gt_table :
        FALSE ; FALSE ; TRUE ; FALSE
        FALSE ; FALSE ; TRUE ; FALSE
        FALSE ; FALSE ; TRUE ; FALSE
        FALSE ; FALSE ; TRUE ; FALSE
```

The meaning of this table is related to the fact that any status instruction generates a word

```
        00000000000000000000000000vcnz00
```

If the status word is used as an index into the table above it generates a boolean value for each of the 16 status values. The table given here gives a TRUE precisely when n is 1 an z is 0. A table which accounts for overflow would have different values in the last two rows.

The code to accomplish the stated purpose is

```
        SUB_ST    S(>) C()  ;           produce the
                                            status word
        ADD  C(zero_gt_table) C() @     Fetch the boolean
```

from the table

This code takes 3 cycles. A separate table is necessary for each type of condition.

Interrupts on the SF1

The exact nature of interrupt control is frequently determined by the demands of the devices and should not be too tightly coupled to processor design. Thus, we assume that an interrupt controller chip will mediate between the devices and the SF1. Control of these devices and communication between the interrupt controller chip and the SF1 will take place through the I frame. When an interrupt takes place the following happens. During an instruction fetch cycle the memory bus is taken over by the controller and it inserts an address onto the line. In the normal course of things the SF1 interprets this as a jump to subroutine. This causes a vectored interrupt to the location placed on the bus. The return address stored is that of the instruction following the address which was placed on the memory bus by the SF1 during the interrupt cycle. The return from interrupt is

```
ADD  C(-4) C()   ;
LOAD S(>) P(<)   ;
NOOP S()  I(?)   ;      This resets the stacks and the
                        interrupt hardware
```

For this basic interrupt scheme, the interrupt service routine will be using the interrupted process's stack-frames. For many systems this will be satisfactory. In other systems, it may be useful for the interrupt to trigger a complete change of context. This can be done by the interrupt controller or by the SF1 program through its I frame. The mapping between stack-frame hardware and instruction fields is redefined by such operations.

It should also be mentioned that interrupts should be disabled for one instruction following any instruction that writes to P. This is usually the return instruction and is easy to detect by the P hardware. The return instruction is really takes place over two cyles and no machine can suffer interrupts in the middle of an instruction.

C and Other Block Structured Languages

In the C benchmark program below, the arguments and local variables are nested in stack-frames. This system requires one stack-frame for each visible block. In C this requires three; the local stack, the arguments and local variables, and the global variables. Arrays and variables to which pointers are passed must reside in main memory.

In a language where arbitrary nesting of blocks is possible, then the entire block structure must be maintained in main memory. Stack-frames can be used effectively to maintain the "display lists" of pointers to the bases of storage of each nested

block. In general, an access of this type takes three cycles. An example of accessing the third word in the the seventh frame is

```
LOAD F(6) S(<) ;          F(6) is the seventh element
ADD  C(8) C()  @     .
```

An array with its index on the stack can also be accessed in two cycles

```
ADD  F(6) C()  ;
ADD  C(8) C()  @     .
```

In summary, the stack-frames, at their best, give very fast access to local data from a program. If the context is too deeply nested or too large for the implemented stack-frames then the stack frames can act as fast and very flexible register sets.

Processes

Process context can be switched very fast as far as the ALU is concerned because it has no registers. The stack-frames belong to a process so they must be switched in hardware, much as register sets are switched in a standard minicomputer, if the switches are to be very fast. If such hardware is not provided. Then they must be drained and restored as is usually done with microcomputers. The stack bus and the ability to provide as many stack-frames as needed makes the hardware switching, which takes only one cycle, attractive for demanding real-time applications.

An Example of a Compiled C Program

The following example is a simple bubble sort program written in naive C.

```
bsort(list,nn)
int list[],nn;
{
     int xx,yy,qq,zz;
     xx=o;
     qq=0;
     while (qq==0)
     {
          qq=1;
          xx=0;
          while (xx<nn)
          {
               yy=xx+1;
               if (list[xx]<list[yy])
               {
                    zz=list[xx];
                    list[xx=list[yy];
                    list[yy]=zz;
                    qq=0;
               }
```

```
                    xx=yy;
                }
            }
}
```

In the hand compiled version of this program, let the following abbreviations be used:
qq is F(0), yy is F(1), xx is F(2), nn is F(3), list is F(4). frame is NOOP S(), I(2);

```
bsort:
        frame                   ;    exchange F and S
        NOOP C(0) F(<)        D;     First local variable
        NOOP C(0) F(<)        D;     Second loacal variable
        NOOP C(TRUE) F(<)     D;     true is 4
        LOAD qq S(<)            ;
        skip_on_false           ;    While test
        bsort1                  ;    Body of the while
        NOOP F(>) C()           ;    Delete variable
        NOOP F(>) C()           ;    Delete variable
        NOOP F(>) C()           ;    Delete variable
        NOOP F(>) C()           ;    Delete parameter
        NOOP F(>) C()           ;    Delete parameter
        LOAD F(>) P(<)          ;    return
        frame                   ;


bsort1:                             Body of the outside loop
        LOAD C(FALSE) S(<)      ;
        LOAD S(>) qq            ;    qq gets the FALSE
        LOAD C(0) S(<)          ;
        NOOP S() xx             ;    Now xx gets the 0
        SUBR_ST nn C()          ;
        ADD C(zero_gt_table) S() @   returns a true if negative
        skip_on_false           ;    while test
        bsort2                  ;    Body of loop
        SUB  qq C()             ;    return test is fast
        LOAD S(>) P(<)          ;    return
        NOOP S() C()            ;


bsort2:                             Body of the inside loop
        LOAD xx S(<)            ;
        ADD  C(4) C()           ;    Increment is 4 because of size
                                     of words
        LOAD S(>) yy            ;
        LOAD list S(<)          ;    Array addressing takes 2
        ADD  xx  C()            @    instructions and 3 cycles
        LOAD list S(<)          ;
        ADD  yy C()             @
        SUBR_ST S(>) C()        ;    Convert status to true or false
        ADD  C(zero_gt_table) C() ;
        skip_on_true            ;    if test
        JMP bsort3              ;    body of the if
```

```
          LOAD list S(<)        ;
          ADD  xx C()           ;    Address of list[xx]
          LOAD list S(<)        ;
          ADD  yy C()           ;    Address of list[yy]
          LOAD S(0) S(<)        @    list[xx]
          LOAD S(0) S(<)        @    list[yy]
          LOAD S(2) C()         ->   Write in list[xx]
          LOAD S(>) C()         ;    Drop address
          LOAD S(>) C()         ->   Write in list[yy]
          LOAD S(>) C()         ;    Clear stack
          LOAD S(>) C()         ;
          LOAD C(true) S(<)     ;
          LOAD S(>) qq          ;


bsort3 :
          LOAD yy S(<)          ;
          NOOP S() xx           ;
          SUB_ST nn C()         ;    Condition is at the end of loop
          ADD C(zero_lt_table) C() @
          SUB S(>) C()          ;
          LOAD S(>) P(<)        ;    return
          NOOP S() C()          ;


     skip_on_false:
          as above
```

Assuming a 10 Mhz clock on the SF1, this program runs about 40 times faster than C86 on the Zenith 158 and about 2 times as fast as C on the SUN-3.

Inference Machines on the SF1

L.L.Odette and W. H. Wilkinson have reported at the FORML 86 Conference in the paper "Prolog at 20,000 LIPS on the Novix?". They suggest such performance on the (to be released) 10Mhz version of the 16-bit Novix chip which is a pure stack machine. The SF1 has most of the capabilities of the Novix and many that the Novix does not. Following Odette and Wilkinson's model we expect the performance to be comparable to Novix system. That would match the SF1 performance on C, twice the performance of the SUN-3 or the VAX-780 on this type of problem.

Extensions of the SF1

ALU Operations

The addition of operations to the ALU is consistent with the SF1 design provided the operation has at most two inputs and produces a single 32-bit result. An example of an instruction that could be there but is not is an arbitrary shift instruction. The main memory operations should be extended to include byte

operations. Internalizing some of the tables like the zero_gt_table might be reasonable. In general, a more powerful conditional branch facility would be helpful.

Operations like multiply and divide might be handled carefully within the ALU. A multiply which yields a 32-bit result is possible. One operation for the high order 32-bits, another for the low order 32-bits is more inclusive. A 32-bit divide and a 32-bit remainder operation suffice for longer divides.

I/O Space operations

In order to keep the ALU chip simple, more complicated operations can be done on chips connected to the I/O bus. Thus a floating point chip or an array processer could be placed there. In this case, these chips will retain status information over several cycles. It will then be necessary to shut off interrupts during the several cycles they take to operate, or have them stack inputs and outputs or to have them be virtual for each new process.