1984

The Bluff Machine (Single Module Version)

John T. Korb
Purdue University, jtk@cs.purdue.edu

Report Number:
84-484

The Bluff Machine
(Single Module Version)

John T. Korb

May 1984

CSD-TR-484

Computer Science Department
Purdue University
West Lafayette, IN 47907

ABSTRACT

This report documents the target machine used by the Compiler Construction class (CS502) at Purdue University. The Bluff machine is a simple, stack-oriented machine designed to permit straightforward translation from C programs.

A Bluff assembler and machine interpreter have been implemented for VAX UNIX systems.

The version of Bluff defined in this report is designed for programs consisting of a single module (or for multiple modules combined into a single module at compile or link time). It provides no support for dynamic program loading. A multi-module version is forthcoming.

The Bluff Machine
(Single Module Version)

1. General Features

The Bluff machine is a general-purpose, stack-oriented processor. It uses one-byte opcodes and relative addresses to efficiently encode instructions. The word size is 32 bits.

Bluff provides a number of facilities for the implementation of high-level languages. It includes instructions for recursive procedure calls, automatic argument passing, and efficient access to local and global variables.

The Bluff machine uses two stacks, a procedure stack and a register stack. The procedure stack contains frames of partially executed procedures, with the currently executing procedure frame on top. The register stack is a block of accumulators that store the results of partially computed expressions.

The procedure stack is kept in main memory, although the top few elements may be kept in processor memory. The entire register stack is contained in processor memory.

2. Bluff Registers

All registers are thirty-two bits wide. There are two kinds of addresses in Bluff, word addresses and byte addresses. Integers and pointers are addressed using *word* addresses, instructions are addressed using *byte* addresses.

- `PC` The `PC` contains the *byte* address of the next instruction.
- `SP` Register `SP` points to the *next available location* on the procedure stack.
- `F` Register `F` (frame pointer) points into the procedure stack to the top of the currently executing procedure frame.
- `G` Register `G` (global pointer) points to the global variable table, a block of memory locations for storing values accessible to all procedures. Bluff provides instructions for accessing global variables relative to this pointer.
- `P` Register `P` (procedure pointer) points to the procedure entry point table. The procedure call instruction gives an offset into this list. The format of the procedure entry point table is described in the data structure section.

3. Initialization and Power-On Sequence

When the power is turned on to the Bluff machine, main memory is assumed to be initialized. Bluff initializes its registers by loading them from the following main memory locations.

register	*memory location*
`SP`	0
`G`	1
`P`	2

After initializing the three registers, Bluff calls the first procedure in the procedure table (that is, it executes a `CALLB 0` instruction). The procedure call initializes the remaining two registers (`PC` and `F`).

The Bluff machine is automatically turned off by executing a return instruction in the main procedure.

4. Data Structures

Bluff uses three data structures in main memory: a procedure stack, a global variable table, and a procedure entry point table.

4.1 Procedure Stack

The **procedure stack** is used to store procedure frames during execution. The stack grows downward in memory, toward higher addresses. It is located at the first word following the last instruction and occupies all of remaining memory.

4.2 Global Variable Table

The **global variable table** is a block of memory that contains variables that can be directly accessed by any procedure. This table is pointed to by the G register.

4.3 Procedure Entry Table

The **procedure entry table** contains a two-word entry for each procedure in the program. The first word is the *byte address* of the first instruction of the procedure. The second word is the initial size of the frame (including parameters) to be allocated when the procedure is called.

The first two-word entry of the procedure entry table contains the byte address of the main procedure and the initial length of its frame.

5. Typical Memory Layout

The diagram on the next page shows a typical memory layout for the Bluff machine. The first three memory locations contain the values required for initialization. The positions of the three data structures are determined from these values.

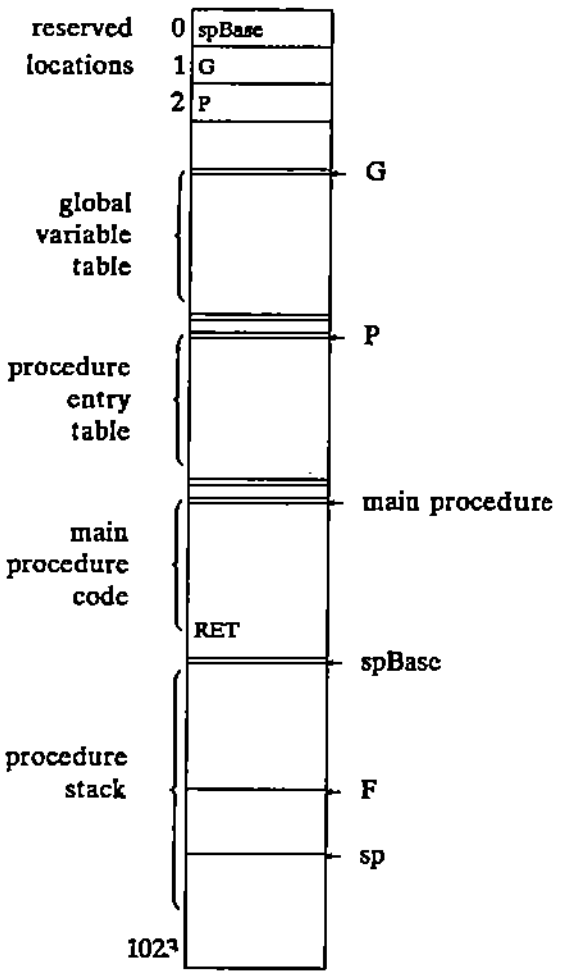
6. Instruction Set

Listed below are the basic Bluff instructions. All opcodes and operands are 8 bits wide. Instructions are either 8 or 16 bits wide and may cross word boundaries. Instructions that take an 8 bit argument end with the letter B, meaning byte (for example, see LLB, "Load Local Byte", below).

For some of the operations that expect values on the register stack, the order of the values on the stack is indicated by showing the order in which they are computed. For example,

```
⟨dividend⟩
⟨divisor⟩
DIV
```

means that the ⟨divisor⟩ is on the top of the stack (since it is computed second).



Typical Memory Layout

6.1 Load Instructions

Instructions in this group push values onto the register stack.

`LLB` i (Load Local Byte) Push the i^{th} variable in the current local procedure frame onto the register stack. The i^{th} local variable is at address (contents of `F`) + i.

`LGB` i (Load Global Byte) Push the i^{th} global variable onto the register stack. The i^{th} global variable is at address (contents of `G`) + i.

`LIB` $\pm i$ (Load Immediate Byte) Push the *sign-extended* value of i onto the register stack. Sign-extended means that if i is negative the left-most bits of the word pushed onto the stack are all ones.

`LLAB` i (Load Local Address Byte) Push the *address* of the i^{th} local variable onto the register stack. See `LLB`.

`LGAB` i (Load Global Address Byte) Push the *address* of the i^{th} global variable onto the register stack. See `LGB`.

6.2 Store Instructions

Instructions in this group pop values off the register stack into either a local variable or a global variable.

`SLB` *i* (Store Local Byte) Pop the top value off the register stack and store it into the i^{th} local variable in the current local procedure frame. The i^{th} local variable is located at address (contents of `F`) + *i*.

`SGB` *i* (Store Global Byte) Pop the top value off the register stack and store it into the i^{th} global variable. The i^{th} global variable is located at address (contents of `G`) + *i*.

6.3 Read and Write Memory Instructions

Instructions in this group can be used to read from and write to arbitrary memory locations.

`RD` (Read) Pop the top value off the register stack and push the value located at that address onto the register stack. `RD` expects one value on top of the register stack and it overwrites that value. There is no net change in the height of the register stack.

```
⟨pw⟩
RD
```

`WR` (Write) Pop two values off the register stack. Store the second value popped at the address given by the first value popped. `WR` expects two values on the register stack and it pops both values off.

```
⟨w⟩
⟨pw⟩
RD
```

6.4 Arithmetic and Logical Instructions

Instructions in this group take their operands from the register stack and push their results back onto the register stack.

`ADD` (Add) Pop the top two elements off the register stack, add them, and push the sum back onto the stack.

`SUB` (Subtract) Pop the top two elements off the register stack, subtract them (the second element minus the top element), and push the difference back onto the stack.

`MUL` (Multiply) Pop the top two elements off the register stack, multiply them, and push the product back onto the stack.

`DIV` (Divide) Pop the top two elements off the register stack, divide them (the second element divided by the top element), and push the integer quotient back onto the stack.

`NOT` (Logical Not) Replace the top element of the register stack with its (bitwise) complement.

AND (Logical And) Pop the top two elements off the register stack, form the logical (bitwise) and, and push the result back onto the stack.

OR (Logical Or) Pop the top two elements off the register stack, form the logical (bitwise) or, and push the result back onto the stack.

6.5 Character Instructions

Character pointers are constructed from word pointers by multiplying the latter by four and adding the offset of the desired character. Characters are stored in memory such that the low-order eight bits in a word correspond to the character whose address is four times the word address.

RDCH The RDCH instruction replaces the character pointer on the top of the register stack by the character to which it points. The character is right-justified in the word and zero-filled (suitable for output using the OUTCH instruction).

```
⟨pch⟩
RDCH
```

WRCH The WRCH instruction writes the character ⟨ch⟩ to the location pointed to by ⟨pch⟩. Other characters in the word are unaffected.

```
⟨ch⟩
⟨pch⟩
WRCH
```

PWXPCH The PWXPCH instruction converts a pointer to a word to a pointer to a character. The value on the top of the stack is shifted left two bits. The height of the stack is unchanged.

```
⟨pw⟩
PWXPCH
```

PCHXPW The PCHXPW instruction converts a pointer to a character to a pointer to a word. The value on the top of the stack is shifted right two bits. The height of the stack is unchanged.

```
⟨pch⟩
PCHXPW
```

6.6 Comparison Instructions

Each comparison instruction pops two words off the stack, compares them according to the opcode ((second element) **op** ⟨top element⟩), and pushes the boolean result back on the stack (-1 for true, 0 for false).

CMPLT (Compare Less Than) Pop and compare the top two elements on the stack. If the second element is less than the top element, push -1 else push 0.

CMPLE (Compare Less or Equal) Pop and compare the top two elements on the stack. If the second element is less than or equal to the top element, push -1 else push 0.

CMPGT (Compare Greater Than) Pop and compare the top two elements on the stack. If the second element is greater than the top element, push -1 else push 0.

CMPGE (Compare Greater or Equal) Pop and compare the top two elements on the stack. If the second element is greater than or equal to the top element, push -1 else push 0.

CMPEQ (Compare Equal) Pop and compare the top two elements on the stack. If the second element is equal to the top element, push -1 else push 0.

CMPNE (Compare Not Equal) Pop and compare the top two elements on the stack. If the second element is not equal to the top element, push -1 else push 0.

6.7 Unconditional and Conditional Jump Instructions

Instructions in this group jump forward or backward either unconditionally or as a result of testing (and discarding) the top element on the register stack.

JMPB $\pm i$ (Jump Byte) Add the *sign-extended byte offset*, i, to the PC. This instruction can jump forward 127 locations or backward 128 locations. Since the instruction is two bytes long, JMPB -2 is an infinite loop.

JEQB $\pm i$ (Jump Equal Byte) Pop the top element off the register stack. If it is zero, jump to the destination determined by $\pm i$ (as in the JMPB instruction).

JNEB $\pm i$ (Jump Not Equal Byte) Pop the top element off the register stack. If it is not zero, jump to the destination determined by $\pm i$ (as in the JMPB instruction).

6.8 Stack Manipulation Instructions

Instructions are provided to modify the heights of the procedure stack and the register stack, as well as to discard the contents of the register stack.

MSPB $\pm i$ (Modify Stack Pointer Byte) Add the *sign-extended* byte to the procedure stack pointer, SP.

SRS (Save Register Stack) Store the contents of the register stack at the end of the procedure stack. Decrement the register stack pointer and increment the procedure stack pointer by the appropriate amounts. (Used with RRSB instruction below.)

RRSB i (Restore Register Stack Byte) Restore the i values at the end of the procedure stack to the register stack. Increment the register stack pointer and decrement the procedure stack pointer by i. If any values were on the register stack before the RRSB instruction, they are placed *on top of* the register stack after the instruction. Thus, the SRS/RRSB instructions can be used to save the register stack before a procedure call, restore it after the call, and leave the return value from the procedure on top of the stack.

IRSP (Increment Register Stack Pointer) Add one to the register stack pointer. IRSP is useful for recovering a value that was just popped off the stack.

DRSP (Decrement Register Stack Pointer) Subtract one from the register stack pointer. DRSP is useful for discarding a value that is no longer needed.

DUP (Duplicate) Push the top value of the register stack onto the register stack. That is, make a duplicate of the top of the stack.

EXCH (Exchange) Exchange the top two elements of the register stack.

SST (String to Stack) Copy a string of characters in the code segment to the procedure stack. The Bluff assembler format for this instruction is

```
SST "This string uses a tab \t and newline \n character."
```

The assembler stores the SST opcode immediately followed by the string. The string is stored four characters to a word with the end of the string marked by a null (0) byte. The SST instruction causes the string to be copied to the procedure stack with the last byte (the null char) at the top of the stack. A pointer to the start of the string is pushed onto the register stack.

6.9 Procedure Call and Return Instructions

These instructions are used to call and return from procedures. Argument transmission is call by value (although other techniques may be simulated).

CALLB i (Call Procedure Byte) Call the i^{th} procedure in the procedure table. The entry point to the procedure is the first word of the i^{th} pair in the table pointed to by P, the procedure entry table register.

The procedure call involves the following steps.

(1) Push the return program counter onto the procedure stack.
(2) Push the current frame pointer (F) onto the procedure stack.
(3) Set F equal to SP. This step sets up a new local environment for the called procedure.
(4) Increment SP by the number of local variables needed in the called procedure. This number is contained in the second word of the entry in the procedure entry point table.
(5) Move the contents of the register stack onto the procedure stack starting at F. All values on the register stack are popped off. This step passes parameters to the called procedure. Note that the value at the base of the register stack is copied to the location pointed to by F.
(6) Set the PC to the entry point of the called procedure.

CALLS (Call Procedure on Stack) Like CALLB, except the index of the procedure in the procedure entry point table is popped from the top of the register stack.

RET (Return) Return from procedure. The contents of the register stack are taken to be the return value(s) from the procedure. That is, the

called procedure may return one or more values to the caller by simply leaving them on the register stack. Note that the calling procedure must know how many values are being returned so it can correctly use the returned stack.

6.10 Input/Output Instructions

`INN` (Input Number) Read the next integer from the input stream, convert it to binary, and push the value on the stack.

`OUTN` (Output Number) Pop the integer value off the top of the stack, convert it to a string, and write it to the output stream.

`INCH` (Input Character) Read the next character from the input stream and push it on the stack (as a small integer).

`OUTCH` (Output Character) Pop the top value off the register stack and write it onto the output stream (as a character).

`OUTS` (Output String) Pop the top of the stack and use it as a pointer to the string to be printed.

`DUMP` (Dump) Perform a machine-dependent register and partial memory dump. The exact details of this instruction are determined by the hardware used to implement the Bluff machine.

6.11 Miscellaneous Operations

`RDB` i The `RDB` instruction is used for structure references. The pointer on the top of the stack is assumed to point to a structure. The constant i selects a field (word) within the structure. The RDB instruction replaces the top of the stack with the value stored at $(pw + i)$.

```
⟨pw⟩
RDB    i
```

`WRB` i The `WRB` instruction is the complement of the `RDB` instruction. It writes the value ⟨w⟩ to location ⟨pw⟩+i.

```
⟨w⟩
⟨pw⟩
WRB    i
```

`BFORW` i, L The `BFORW/EFORB` instructions implement a Pascal-style **for** loop in Bluff. The upper and lower limits of the loop are on the register stack. The argument i is an index in the local frame of a three-word block that is used as temporary storage by the two instructions (the first word is the current value of i, the second is the upper limit, and the third is the address of the first instruction in the loop). The label L is after the loop and is used to allow the loop to be executed zero times if the upper limit is less than the lower limit.

```
⟨iLower⟩
⟨iUpper⟩
```

```
        BFORW    i, L
          .
          .
          .
        EFORB    i
L:
```

`SWITCH` k The `SWITCH` instruction is used to implement the C **switch** statement. The value on the register stack is the value being compared. The switch table has k pairs (W_i, L_i). Control transfers to label L_i if w is equal to W_i. If none of the values are equal, the instruction following the last `CASE` statement is executed next (where the **default** should be located).

```
(w)
SWITCH    k
CASE      W1, L1
CASE      W2, L2
  .
  .
  .
CASE      Wk, Lk
```

`NOP` (No Operation) Perform no operation (except to increment the `PC`).

7. Assembler Syntax

The Bluff assembler accepts Bluff statements in the format given in this document:

⟨label⟩: ⟨opcode⟩ ⟨operands⟩,... ; ⟨comments⟩

Any instruction can be given a label. A label can also appear on a line by itself, in which case it is associated with the byte-address of the next instruction. Comments (and the initial semi-colon) are optional.

The assembler automatically converts labels used in jump instructions (`JMPB`, `JNEB`, and `JEQB`) to the appropriate signed offset required by the Bluff machine.

The assembler provides two pseudo-instructions for allocating and initializing memory locations. These instructions can be used to setup the global data structures needed by the Bluff runtime system.

`DW` $n,\ldots$ (Define Word) The `DW` instruction takes one or more labels or numeric operands. The value of each operand is put in the next available memory word. If a label is present, it becomes the byte-address of the first generated word.

`DS` i (Define Space) The `DS` instruction allocates i words of memory starting at the current location. If a label is present, it becomes the byte-address of the first generated word.

8. Sample Bluff Program

Below are a simple C program and a corresponding compiler-generated Bluff program.

```
int x;

main ()
{
    int a, b, y;

    a = 10;
    b = 2;
    y = 3;

    x = a + b*f(y);
    bluff { LGB 0; OUTN; }
}

int f (w)
int w;
{
    return (w+1);
}
```

C Program

```
    DW      SP,G,P    ; define first three memory locations
G:
    DS      1         ; storage for x
P:
    DW      main,3    ; main has 3 local variables
    DW      f,1       ; f has 1 local variable
main:
    LIB     10
    SLB     0         ; a = 10
    LIB     2
    SLB     1         ; b = 2
    LIB     3
    SLB     2         ; y = 3
    LLB     0
    LLB     1
    SRS               ; save register stack
    LLB     2         ; load y
    CALLB   1         ; call f
```

```
    RRSB    2       ; restore register stack
    MUL
    ADD
    SGB     0       ; x = a + b*f(y)
    LGB     0
    OUTN
    RET
f:
    LLB     0       ; load w
    LIB     1
    ADD             ; w+1
    RET
SP:                 ; put stack at end of code
```

Bluff Program