The BLIZZARD Computer Architecture

Ronald L. Rivest
MIT Laboratory for Computer Science
February 6, 1978(Rev. 8/8/79)

We present a simple and powerful computer architecture with several novel features, the most interesting of which are probably the ways in which immediate and stack operands are treated in this register-oriented machine. The recent debate in this journal (see [1]-[5]) over the relative efficiencies of stacks and registers will take on a new form for the BLIZZARD.

The machine is called the BLIZZARD, since the design was initiated while the author was snow-bound at home during the Boston "Blizzard of '78".

A summary of the BLIZZARD's features are:

- Memory consists of up to 2**32 16-bit words. Addressing is by word.
- 16 double-word registers are available. Seven have special functions (like PC, SP); three of the seven are "window" registers for fetching immediate immediate operands and pushing/popping stack operands.
- Data lengths 1, 2, 4, 8, 16 or 32 bits (called bits, dabs, nibs, bytes, words, and double-words, or generically, "flakes") handled easily. Small flakes are stored as packed arrays; each flake is easily accessible.
- Each instruction is one word long (and is possibly followed by word or double-word immediate operands) and has a simple four-nib format.
- There are looping, subroutine enter/exit, and block move instructions, etc.

I. NOTATION AND DATA FORMATS

A double-word occupies consecutive words; its address is the same as the word with lower address (which contains the least-significant bits of the double-word). We let [X] denote the contents of the memory word at location X, and (X) denote the double-word stored at location X (consisting of [X] and [X+1]). We also use () for denoting register contents: (PC), etc.

Integers are in two's-complement; floating-point numbers are double-words with a sign, eight exponent bits (excess 128 code) and a 23-bit mantissa.

We use hexadecimal notation; a word is four hex digits (nibs). A hex constant appears as # followed by a nib sequence. Lower case letters denote arbitrary nibs: #8ABd is a word with nibs 8, A, B, and an arbitrary nib d.

The bits of a double-word or word are numbered from 31 or 15 for the sign bit to 0 for the least-significant bit. Similarly, flakes in a word are numbered starting with zero for the flake in the least-significant position.

II. REGISTERS

BLIZZARD's 16 double-word registers (R0 to RF) are carefully allocated between seven dedicated uses and general-purpose use. This frees the opcodes from having to specify operand sources (e.g. immediate, stack, or register). The registers are identified with double-words 0 to F of memory (Ri consists of memory words 2*i and 2*i+1).

R0 = Program Status Register PSR
carry bit (bit 31), overflow bit(bit 30), interrupt mask bit (bit 29), and flake size code: 1,2,4,8,16, or 32 (bits 0-5)

When used as a base or index register, PSR has a value of 0.

R1 = Link Register LR: Keeps the return address for subroutines.
LR gets the old PC, whenever PC is assigned a new value.

R2 - RA = General Purpose Registers

RB = Program Counter PC: Incremented by 1 after each instruction fetch.

RC = Stack Pointer SP
Push decrements SP by 2 then stores a double-word.
Pop fetches the double-word at (SP) then increments SP by 2.

RD = Word Immediate "Window" Register WIR
Fetch of WIR (even as base or index register) returns [(PC)] then adds 1 to PC. Store into WIR is a no-op.

RE = Double-word Immediate "Window" Register DWIR
Fetch of DWIR (even as base or index register) returns ((PC)) then adds 2 to PC. Store into DWIR is a no-op.

RF = Double-word Top-of-Stack "Window" Register TOS
Fetch of TOS (even as base or index register) returns [(SP)] then adds 2 to SP. (This is a "pop".) Store into TOS subtracts 2 from SP, then stores into location (SP). (This is a "push".)

III. THE INSTRUCTION SET

An instruction is one word long, but its execution may fetch the next word or double-word and bump PC by an additional 1 or 2 (due to WIR or DWIR). The instructions have similar four-nib formats. Using "0" for opcode, "r" for register, and "-" for immediate denotations, they look like:

```
Orr-    load/store a double-word/flake (base reg + immediate offset)
Orrr    load/store a double-word/flake (base reg + index)
OOr-    compare against immediate, not, enter, exit
OOrr    compare/add/subtract/exchange/etc. registers
Or--    load/add/subtract immediate, load PC (base reg + displacement)
O---    jump (relative to PC)
```

The instructions are listed below. In all cases the registers named Rx, Rb, and Ra are evaluated in that order. (This only matters if two of them refer to the stack via TOS or to immediate operands via WIR or DWIR.)

III.1 LOAD and STORE DOUBLE-WORD INSTRUCTIONS

0abd LOAD DOUBLE-WORD: loads Ra with the double-word at address (Rb)+2*d.
Assembler: L a,d(b) (PUSH d(b) for L TOS,d(b))

1abx LOAD DOUBLE-WORD INDEXED: loads Ra with the double-word at (Rb)+2*(Rx).
Assembler: LX a,(b,x) (PUSHX (b,x) for LX TOS,(b,x))

2abd STORE DOUBLE-WORD: stores double-word (Ra) into location (Rb)+2*d.
Assembler: S a,d(b) (POP d(b) for S TOS,d(b))

3abx STORE DOUBLE-WORD INDEXED: puts (Ra) into location (Rb)+2*(Rx).
Assembler: SX a,(b,x) (or POPX (b,x) for SX TOS,(b,x))

These instructions load registers, jump (Ra=PC), change the current flake size (Ra=PSR), push/pop data from the stack (Ra=TOS), etc. If the base or index register (or both) is TOS, the value used is popped from the stack. Small displacements (0 to 15) are coded directly; larger displacements require the LOAD/STORE INDEXED format with WIR or DWIR as the index register (the displacement appears as an immediate operand). Note that for all of these instructions the displacement d or (Rx) is in double-words from location (Rb).

III.2 LOAD and STORE FLAKE INSTRUCTIONS

4abd	LOAD FLAKE	(Assembler: LF a,d(b) or PUSHF d(b))
5abx	LOAD FLAKE INDEXED	(Assembler: LFX a,(b,x) or PUSHFX (b,x))
6abd	STORE FLAKE	(Assembler: SF a,d(b) or POPF d(b))
7abx	STORE FLAKE INDEXED	(Assembler: SFX a,(b,x) or POPFX (b,x))

A flake (of size given in PSR) is loaded (right-justified with leading zeros) into or stored from Ra. The flake is the d-th or (Rx)-th flake from the least significant flake in (Rb) (0-origin indexing). Thus if PSR=1, the instruction LF 2,5(3) loads R2 with the 5-th bit of word [R3].

III.3 SIMPLE IMMEDIATE INSTRUCTIONS

8abc	LOAD IMMEDIATE	(Assembler: LI a,#bc or PUSHI #bc) Ra gets #bc.
9abc	ADD IMMEDIATE	(Assembler: ADDI a,#bc) Add #bc to Ra.
Aabc	SUBTRACT IMMEDIATE	(Assembler: SUBI a,#bc) Subtract #bc from Ra.

These mildly redundant (viz. WIR/DWIR) instructions provide for compact code when the immediate operand is one byte or less.

III.4 JUMP AND SUBROUTINE CALL INSTRUCTIONS

Babc JUMP: Two's complement value #abc is added to PC. (-2048<=#abc<=2047)
Assembler: J label

Cabc LOAD PC: The PC is loaded with the double-word at (Ra)+2*bc.
Assembler: LPC bc(a) or just LPC subrname

These are also mildly redundant, but help to compact code. JUMP provides enough displacement to handle most local jumps. LPC provides a concise way of referring to a relatively large (256) set of subroutines. For example, by dedicating one register as base register for a dispatch table, each of 256 common procedures can be called with a one-word instruction.

Since LR gets the old PC whenever PC is changed (except by the post-instruction fetch incrementation), any of the instructions LOAD, STORE, JUMP, LPC, etc., can call a subroutine. The subroutine need only save the return address in LR (say on the stack) immediately upon entry. The ENTER instruction (see later) provides a neat way of doing this.

III.5 COMPARE AND LOOP-CONTROL INSTRUCTIONS

Dfab Decrement/Increment/leave alone Ra, then
compare (Ra) against either b or (Rb) and skip. f-codes:
0=DSLI, 1=SLI, 2=SEI, 3=SLEI, 4=SGI, 5=SNEI, 6=SGEI, 7=ISGI,
8=DSL, 9=SL, A=SE, B=SLE, C=SG, D=SNE, E=SGE, F=ISG.
DSLI and DSL first decrement Ra. ISGI and ISG first increment Ra.

For 0<=f<=7 comparison is (Ra) against b, otherwise it is (Ra) vs. (Rb)
E.g.: SG skips if (Ra)>(Rb), ISG a,b is equivalent to ADDI a,1; SG a,b.
Assembler: SE a,b or SEI a,b etc.

The immediate form can compare a register to zero. The instruction skipped must be a one-word instruction (no immediate operands). (If Ra or Rb is WIR/DWIR, the instruction skipped appears after an immediate operand.) The floating-point format causes comparisons to be consistent with integer comparisons. DSLI, DSL, ISGI, ISG are useful in loop control; typically they are followed by a jump back to the beginning of the loop.

III.6. BINARY OPERATIONS

Efab Ra gets binary operation "f" of (Ra) and (Rb). f:
0=ADD, 1=SUB, 2=MUL, 3=DIV, 4=FADD, 5=FSUB, 6=FMUL,
7=FDIV, 8=REM, 9=AND, A=OR, B=XOR, C=LSH, D=RSH,
E,F unused
Assembler: ADD a,b etc.

MUL and DIV have double-word product and dividend.

III.7. UNARY and UTILITY OPERATIONS

F0ad	NOT a,d	Set Ra to complement of (Ra), plus d (d=0 is complement, d=1 for 2's complement)
F1ad	ENTER a,d	Saves R0..Ra on stack, while taking d arguments from the stack and loading them into R2..R[d+1] (Top goes into R2, etc.)
F2ad	EXIT a,d	Restores R0..Ra from save area on stack, underneath d result double-words. Eliminates save area by moving d results down a+1 positions and adjusting SP. (I.e. stack goes from [d results, (a+1) regs, other] to [d results, other]) Then loads PC from LR.
F3ab	EXCH a,b	Exchange registers a and b (Note that a=b=TOS exchanges top two elements on stack.)
F4ab	BLOCK a,b	Moves a block of [TOS] words from (Rb),... to (Ra),... . Stack is popped. "Smart" version.
F5.. to FF..		Unused.

This completes our listing of the BLIZZARD instruction set.

IV. THE WINDOW REGISTERS WIR, DWIR AND TOS

These give much versatility to the simple instruction set. They are not registers in the usual sense, since they are just shorthand:

WIR is the word pointed to by PC,
DWIR is the double-word pointed to by PC, and
TOS is the double-word pointed to by SP.

That is, (WIR)=[(PC)], so that "register" WIR has a value equal to the immediate word operand for the instruction. Analogously, (DWIR) equals the double-word just after the instruction word. Similarly, (TOS) equals the the double-word on the top of the stack. Thus the programmer can talk about immediate operands or top-of-stack operands as simply and naturally as he can talk about register operands. This includes any use of these registers; even

for base or index-registers purposes. Freeing the opcodes of the necessity of making this distinction helps make a flexible instruction set.

To make this work smoothly, these "registers" have an "auto-increment" property relative to their defining pointers (PC or SP) when they are fetched: PC is incremented by 1 (resp. 2) after any fetch of WIR (resp. DWIR), and SP is incremented by 2 after any fetch of TOS. The increment of PC after fetching WIR or DWIR keeps BLIZZARD from subsequently trying to execute that operand.

When a value is to be stored into TOS, SP is first decremented by 2, and then that double-word is placed in the memory location then pointed to by SP. Thus a load into TOS is a "push". An attempt to store into WIR or DWIR is a no-op.

The concept of a "window register" is different than the "autoincrement indirect" addressing modes of the PDP-11 and VAX. There they obtain the effect of a window register by specifying the defining pointer register (say PC or SP) and then specifying via additional control bits the autoincrement indirect addressing mode. Here immediate and top-of-stack operands are fully equivalent to register operands and may be used wherever register operands are permitted (even as base or index registers); no additional control bits are required.

The stack is automatically checked for overflow or underflow whenever it is modified. The location of the bottom of the stack is given in memory double-word 16 and the location of the top of the stack area is given in memory double-word 17.

Some sample code sequences are:

```
LOC     VALUE           ASSEMBLER       COMMENT

200     020D            L 2,WIR         % loads the constant 1000 into R2
201     03E8            #03E8           % (Note that 1000 = #03E8.)

202     12E3            LX 2,(DWIR,3)   % loads the (R3)th element of the array
203     00123000        #123000         % beginning at #123000 into R2
```

In our future examples, we shall replace WIR or DWIR by the constant so obtained, preceded by @. (E.g. for above: L 2,@1000 and LX 2,(@#123000,3))

```
205     0F0B            PUSH PC         % (same as L TOS,PC) push the program
                                        % counter (206) on the stack.
206     0F0D            PUSH @259       % push constant 259 on the stack.
207     0103
208     0FC0            PUSH 0(SP)      % (same as L TOS,0(SP)) duplicates the top
                                        % double-word of the stack
209     2F0B            POP PC          % (same as S TOS,PC) pop the top of stack
                                        % into the PC
20A     12CD            LX 2,(SP,@999)  % load R2 with 999th stack element
20B     03E7
20C     8008            LI PSR,8        % Set byte size to 8-bit bytes.
20D     E1FF            SUB TOS,TOS     % Subtract the double-word which is on top of
                                        % the stack from the double-word which is
                                        % second on stack, save result on stack
20E     E1FD            SUB TOS,@259    % Decrement double-word on top of stack by 259
20F     0103
210     2DD0            S @1000,0(@259) % Store 1000 into location 259
```

```
211     0103
212     03E8
213     2FF0            POP 0(TOS)      % (Same as S TOS,0(TOS)) Puts the double-
                                        % word which is second the stack in the
                                        % location given on top of the stack.
```

V. LOADING AND STORING FLAKES

BLIZZARD has a flexible way of manipulating flakes. The flake length used in a load/store flake instruction is obtained from PSR when the instruction is executed.

The motivation for the BLIZZARDS's flake addressing method is that flakes are usually kept in packed arrays. The way to address a flake is thus to give the beginning of the array as a word address, and the offset in flakes. Since the flake offset can be given in a register, it is easy to randomly access flakes in an array of packed flakes.

For example, suppose we'd like a "flag bit" for each double-word of memory. Then if LX 2,(0,3) loads R2 with the (R3)-rd double-word of memory, LFX 4,(5,3) loads R4 with the (R3)-rd bit of the table of flag bits beginning at (R5), when PSR = 1. That is, R4 now contains the flag bit for the double-word in R2.

A flake is always loaded into a register right-justified with leading zeros. When a flake is stored, only the indicated flake is modified; other flakes in the same word are not changed. Some examples are:

```
LOC     VALUE           ASSEMBLER       COMMENT
200     8008            LI    PSR,8     % Set flake-size to 8-bit bytes
201     4840            LF    8,0(4)    % Get rightmost byte located at (R4)
202     6851            SF    8,1(5)    % Store in leftmost byte at (R5)
203     6859            SF    8,9(5)    % and in leftmost byte of (R5)+4.
204     8004            LI    PSR,4     % Switch to 4-bit flakes
205     5904            LFX   9,(@1000,4)  %Load (R4)-th flake from flake array
206     03E8                                % which begins at location 1000
207     7956            SFX   9,(5,6)   % Store in (R6)-th flake of array which
                                        % begins at location (R5)
208     595F            LFX   9,(5,TOS)   % load flake from array whose origin is
                                        % given in R5, index is on top of stack
209     4A0F            LF    A,#F      % Get #F-th flake of array beginning at
                                        % 0, which is thus rightmost nib of R1.
```

The load/store double-word and the load/store flake instructions share a common philosophy about offsets: the offset (d or (Rx)) is always measured from (Rb) in units equal to the size of the item being loaded. Thus, L a,d(b) and LF a,d(b) have the same effect when the current flake size is 32. The fact that the current flake size is given in natural form (rather than in a more compact logarithmic notation) in PSR both provides more perspicuous code and permits later expansion of the design to handle flakes of other sizes.

VI. THE JUMP INSTRUCTION AND SUBROUTINE CALLS

The BLIZZARD's one-word jump instructions (J and LPC) are meant to encourage modular code. They have the side effect of setting LR, the link register, to the location one beyond the current instruction. Thus these instructions can be used both for ordinary jumps and for subroutine calls, since the link register will contain the return address for a subroutine call.

The normal way to save the return address upon subroutine entry is to execute an ENTER a,d instruction as the first instruction. This

(1) removes the d parameters from the top of the stack and saves them in a "safe place",
(2) pushes Ra, ... , R0 onto the stack (in that order),
(3) arranges the d parameters in R2, ... , R[d+1]. (The parameter which was originally on the top ends up in R2.)

If a>=1, then this saves the return address. If a>1, the registers 2,...,a are freed for use as temporaries. If d>0, the first d of these temporaries are loaded with the d parameters from the top of the stack. (These parameters are removed from the stack.)

When the subroutine is finished, the instruction EXIT a,d restores the registers and returns the d results on top of the stack as the subroutine results. (This loads the registers from a save area underneath the d results on the stack, and then moves the d results down a+1 positions on the stack.)

VII. INPUT/OUTPUT AND INTERRUPTS

Input/ouput is handled by memory-mapped i/o. A write to (or read from) an IO device looks like a write to (or read from) a particular memory location.

When an interrupt occurs: the PC is stored in double-word 18 of memory, and then the PC is loaded from a double-word reserved to store the address of an interrupt-handling routine (i.e. vectored interrupts). (The link register LR is not modified.) Simultaneously, the interrupt mask bit in the PSR is turned on, which masks out other interrupts.

VIII. A SAMPLE PROGRAM: THE EIGHT QUEEN'S PROBLEM

To illustrate what typical BLIZZARD programs look like, we present here a solution to the Eight Queen's problem. We first present a PASCAL program for its solution due to Niklaus Wirth [Algorithms + Data Structures = Programs, Prentice-Hall, 1976, p. 347-348], and then what BLIZZARD code for the compiled program might look like.

```
program eightqueens(output);
var i: integer,
    a: array [1..8] of boolean,
    b: array [2..16] of boolean,
    c: array [-7..7] of boolean,
    x: array [1..8] of integer;

procedure print;
    var k: integer;
    begin for k:=1 to 8 do write(x[k]:4);
          writeln
    end {print};

procedure try(i:integer);
    var j: integer;
    begin
        for j := 1 to 8 do
            if a[j] and b[i+j] and c[i-j] then
            begin x[i]:=j;
                a[j]:=false; b[i+j]:=false;c[i-j]:=false;
                if i < 8 then try(i+1) else print;
                a[j]:=true; b[i+j]:=true; c[i-j]:= true
            end
```

```
    end {try}

begin
    for i := 1 to 8 do a[i] :=true;
    for i := 2 to 16 do b[i] := true;
    for i := -7 to 7 do c[i] := true;
    try(1)
end.
```

Now we present the BLIZZARD code for the above program.

```
LOC      VALUE    LABEL   ASSEMBLER          COMMENT
1000              A:      WORD               % bits array for A
1001              B:      DOUBLE-WORD                % bits array for B
1003                      WORD               % bits array for C
1004              C:      WORD
1005              X:      9 DOUBLE-WORD              % integer array X
1100     F130     PRINT:  ENTER   3,0        % enter, save to R3
1101     8201             LI      2,1       % use R2 for k. do k:=1
1102     1FD21005 PR1:    PUSHX   (@X,2)     % push x[k]
1104     8F04             PUSHI   4          % push 4
1105     C055             LPC     WRITE      % call system write routine
1106     D728             ISGI    2,8        % k:=k+1; to PR1 if k<=8
1107     BFFA             J       PR1
1108     C056             LPC     WRITELN    % call system writeln rouine
1109     F230             EXIT    3,0        % return

1120     F171     TRY:    ENTER   7,1        % save to R7, get i into R2
1121     8001             LI      0,1        % set flake-size to bits
1122     8301             LI      3,1        % initialize j (R3) to 1
1123     0602     TR1:    L       6,2        % set R6 to i+j
1124     E063             ADD     6,3
1125     0702             L       7,2        % set R7 to i-j
1126     E173             SUB     7,3
1127     55031000         LFX     5,(@A,3)   % load bit A[j]
1129     D250             SEI     5,0        % skip if A[j]=0 (false)
112A     B008             J       TR2        % else enter body of if stmt
112B     55061001         LFX     5,(@B,6)   % load bit B[i+j]
112D     D250             SEI     5,0        % skip if B[i+j]=0 (false)
112E     B004             J       TR2        % else enter body of if stmt
112F     55071004         LFX     5,(@C,7)   % load bit C[i-j]
1131     D550             SNEI    5,0        % skip if C[i-j]=1
1132     B018             J       TR5        % skip over if stmt body
1133     43021005 TR2:    SX      3,(@X,2)   % X[i]:=j
1135     8500             LI      5,0        % set R5 to false
1136     75031000         SFX     5,(@A,3)   % A[j]:=false
1138     75061001         SFX     5,(@B,6)   % B[i+j]:=false
113A     75071004         SFX     5,(@C,7)   % C[i-j]:=false
113C     D128             SLI     2,8        % test if i<8
113D     B004             J       TR3
113E     0F02             PUSH    2          % push i+1
113F     9F01             ADDI    F,1
1140     BFDF             J       TRY        % do try(i+1)
1141     B001             J       TR4
1142     BFCA     TR3:    J       PRINT      % if i>=8 call print
1143     8501     TR4:    LI      5,1        % set R5 to true
1144     75031000         SFX     5,(@A,3)   % A[j]:=true
1146     75061001         SFX     5,(@B,6)   % B[i+j]:=true
1148     75071004         SFX     5,(@C,7)   % C[i-j]:=true
114A     D738     TR5:    ISGI    3,8        % loop
```

```
114B    BFE8            J       TR1
114C    F270            EXIT    7,0             % exit from try

1150    8201    EIGHTQ: LI      2,1             % set R2 to true
1151    8001            LI      0,1             % set flake-size to bits
1152    8301            LI      3,1             % set i to 1
1153    72D31000 E1:    SFX     2,(@A,3)        % A[i]:=true
1155    D738            ISGI    3,8             % loop
1156    BFFC            J       E1
1157    8302            LI      3,2             % i:=2
1158    72D31001 E2:    SFX     2,(@B,3)        % B[i]:=true
115A    F43D0010        ISG     3,@16           % loop
115C    BFFC            J       E2
115D    8307            LI      3,7             % R3 := -7
115E    F031            NOT     3,1
115F    72D31004 E3:    SFX     2,(@C,3)        % C[i]:=true
1161    D727            ISGI    2,7             % loop
1162    BFFC            J       E3
1163    8F01            PUSHI   1               % do try(1)
1164    BFB8            J       TRY
1165    C0FF            LPC     SYSEXIT         % call system exit routine
```

IX. FINAL COMMENTS AND SUMMARY

The design presented here emphasizes those aspects of the architecture that I was most interested in studying: the generalization of the register-structure of the typical von-Neumann machine to handle more general operands and the development of a clean way of handling flakes. By comparison with similar solutions to these problems in other machines (e.g. the addressing modes of the PDP-11 or the byte pointers of the PDP-10), I feel that BLIZZARD is significantly more powerful and flexible.

Other aspects of the design need to be clarified and expanded before BLIZZARD becomes well-defined. The interrupt structure and the possible addition of supervisor/user mode bits in the PSR or test-and-set instructions, for example, need study.

The design of BLIZZARD is also an exercise in aesthetics. An attempt was made to achieve the maximum of capability with a minimum of mechanism. The reader can judge to what extent BLIZZARD achieves this goal.

X. REFERENCES

[1] Myers, G. J., "The Case Against Stack-Oriented Instruction Sets", Computer Architecture News Vol. 6, No. 3 (Aug. 1977), 7-10.

[2] Keedy, J. L., "On the Use of Stacks in the Evaluation of Expressions", Computer Architecture News, Vol. 6, No. 6 (Feb. 1978), 22-28.

[3] Myers, G. J., "The Evaluation of Expressions in a Storage-to-Storage Architecture, Computer Architecture News Vol. 6., No 9(June 1978), 20-23.

[4] Keedy, J. L., "On the Evaluation of Expressions using Accumulators, Stacks, and Store-to-Store Instructions", Computer Architecture News, Vol. 7, No. 4 (Dec. 1978), 24-27.

[5] Keedy, J. L., "More on the Use of Stacks in the Evaluation of Expressions", Computer Architecture News, Vol. 7, No. 8 (June 1979), 18-21.