BL Architecture Documentation

Andrew Buck

Blacklight Software

5/2/2019

**1 APPENDIX**

**1.1 DESCRIPTION**

BL is a 32-bit Little Endian virtualized architecture (RISC) designed for protection of critical parts of security applications within Blacklight Software. Below describes the architecture and its structure.

**1.2 PURPOSE**

Some critical networking portions of the software require protection from tampering, and virtualization with a custom architecture seems to be a viable option.

**1.3 DESIGN**

BL has 12 general-purpose registers, (R\_A-R\_L), a PC (program counter) register (R\_PRG), a conditional register (R\_CND), a stack base register (R\_SB), and a stack frame register (R\_SF). Opcodes are anywhere from 1 byte up to 6 bytes long. Addressing has two modes, immediate (offset from R\_PRG) or register addressing. When each opcode is executed, R\_PRG is increased by the size of that opcode before any relative addressing is made. Up to 2^32 memory locations are accessible, with each memory location containing one 8-bit byte. By default, the base of the stack is by default at 0x1000 and grows to 0x0000, holding up to 0x400 (1024) integers. The x86 32-bit call table is immediately following the stack (the start of the table is represented by R\_SB) and extends to the entry point, and must be populated after mapping an image. The code is loaded into 0x2000 by default, unless otherwise specified. (see Appendix 1.7 for more information on executable structure) The rest of the memory can be used by the program as necessary, provided the space has been allocated by the memory controller.

**1.4 GENERAL**

    IMM - Immediate mode (operand is an offset from R\_PRG after the operand is executed)

    DST REG - Destination Register (R\_A - R\_PRG)

    SRC REG - Source Register

OFFSETXX - XX bit signed offset from R\_PC

DATAXX - XX bit signed data

**1.5 REGISTERS**

|  |  |
| --- | --- |
| R\_A(h/l) | 0 0 0 0 (0) |
| R\_B(h/l) | 0 0 0 1 (1) |
| R\_C(h/l) | 0 0 1 0 (2) |
| R\_D(h/l) | 0 0 1 1 (3) |
| R\_E(h/l) | 0 1 0 0 (4) |
| R\_F(h/l) | 0 1 0 1 (5) |
| R\_G(h/l) | 0 1 1 0 (6) |
| R\_H(h/l) | 0 1 1 1 (7) |
| R\_I(h/l) | 1 0 0 0 (8) |
| R\_J(h/l) | 1 0 0 1 (9) |
| R\_K(h/l) | 1 0 1 0 (10) |
| R\_L(h/l) | 1 0 1 1(11) |
| R\_SF | 1 1 0 0 (12) |
| R\_SB | 1 1 0 1 (13) |
| R\_PRG | 1 1 1 0 (14) |
| R\_CND | 1 1 1 1 (15) |

**1.6 FLAGS**

|  |  |  |
| --- | --- | --- |
| F\_N | Negative | 0 0 0 1 (1) |
| F\_E | Equal | 0 0 1 0 (2) |
| F\_P | Positive | 0 1 0 0 (4) |

**1.7 EXECUTABLE STRUCTURE**

    A valid executable begins with a 32-bit (4 byte) designation as to the stack size, which is 0x1000 by default. The stack is allocated in the first N bytes of memory. The succeeding value is another 32-bit (4 byte) designation as to where the program should be mapped in memory. By default, this is 0x2000, and it must be equal to or higher than the stack size. The gap between the stack and code is reserved for the x86 call table, and can be closed if x86 function calls are not necessary. A compact scenario with 50 bytes of bytecode that uses 4 DWORDs (16 bytes) of the stack could theoretically operate in a 66 byte space, with a stacksize of 0x10 and an entry point of 0x10, where the memory would appear as such:

|  |  |
| --- | --- |
| 0 - 16 | 16 - 66 |
| STACK (R\_SB = 0x16, R\_SF -> 0) | BYTECODE |

**2 INSTRUCTIONS**

**2.0 OPCODE FORMAT**

    The instructions in the BL architecture have a minimum of one byte and can extend up to 4 bytes. The first byte is always the instruction of an opcode, stored in the upper 4 bits of the byte. If the instruction uses immediate values, an addition bit is set in the top bit of the bottom 4 bits of the byte, indicating that the opcode is the immediate version. The bottom 3 bits of the instruction are reserved for the instruction’s use. If any registers are used, the REG byte follows, which designates the source and destination registers. An example is shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BYTE 0 |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 - 0 | 7 - 4 | 3 - 0 |
| INST (0 0 0 0) | IMM (0) | RES (0 0 0) | DST REG (0 0 0 0) | SRC REG (0 0 0 0) |

Shown would be a complete opcode for LD R\_A, R\_A, which would load 4 bytes from the address specified at R\_A into R\_A. Note that if the IMM bit was set, 4 more bytes would follow with an immediate value, forming a 6-byte opcode.

**2.1 LD (LOAD)**

**Description:**

Load data into a register from an immediate address or an address specified by another register. The length of the data to be read is specified in the bottom 3 bits of the instruction. The behavior is undefined if multiple lengths are specified, or if no length is specified.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 0 0 0 0 | IMM | BYTE | WORD | DWORD | DST REG | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| DWORD 2 |
| OFFSET32 |

**Operation:**

|  |
| --- |
| if (imm)     DST = (BYTE/WORD/DWORD)mem[R\_PRG + OFFSET32]; else     DST = (BYTE/WORD/DWORD)mem[SRC]; |

**2.2 LDV (LOAD VALUE)**

**Description:**

Load data into a register that is immediate or is already in another register. The length of the data to be read is specified in the bottom 3 bits of the instruction. The behavior is undefined if multiple lengths are specified, or if no length is specified.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 0 0 0 1 | IMM | BYTE | WORD | DWORD | DST REG | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| BYTE/WORD/DWORD 2 |
| DATA8/16/32 |

**Operation:**

|  |
| --- |
| if (imm)     DST = DATA8/16/32; else     DST = (BYTE/WORD/DWORD)SRC; |

**2.3 ST (STORE)**

**Description:**

Store data from a register into an immediate destination or a destination held in a register. Other bytes are not cleared. The length of the data to be stored is specified in the bottom 3 bits of the instruction. The behavior is undefined if multiple lengths are specified, or if no length is specified.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 0 0 1 0 | IMM | BYTE | WORD | DWORD | DST REG | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| DWORD 2 |
| OFFSET32 |

**Operation:**

|  |
| --- |
| if (imm)     mem[R\_PRG + OFFSET32] = SRC; else     mem[DST] = SRC; |

**2.4 PUSH (PUSH TO STACK)**

**Description:**

Pushes either 4-byte data in a register or an immediate onto the stack and modifies the stack frame register.

**Memory Representation:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BYTE 0 |  |  | BYTE 1 (if IMM == 0) |  |
| 7 - 4 | 3 | 2 - 0 | 7 - 4 | 3 - 0 |
| 0 0 1 1 | IMM | 0 0 0 | 0 0 0 0 | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| DWORD 1 |
| DATA32 |

**Operation:**

|  |
| --- |
| R\_SF -= sizeof(int);  if (imm)     mem[R\_SF] = DATA32; else     mem[R\_SF] = SRC; |
|  |

**2.5 POP (POP STACK)**

**Description:**

Pops data off the stack into a register

**Memory Representation:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BYTE 0 |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 - 0 | 7 - 4 | 3 - 0 |
| 0 1 0 0 | 0 | 0 0 0 | DST REG | 0 0 0 0 |

**Operation:**

|  |
| --- |
| DST = mem[R\_SF]  R\_SF += sizeof(int); |
|  |

**2.6 ADD (UNSIGNED ADD)**

**Description:**

Adds data to a register specified by immediate data or another register. The length of the data to be added is specified in the bottom 3 bits of the instruction if the data is immediate. The behavior is undefined if multiple lengths are specified, or if no length is specified.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 0 1 0 1 | IMM | BYTE | WORD | DWORD | DST REG | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| BYTE/WORD/DWORD 2 |
| DATA8/16/32 |

**Operation:**

|  |
| --- |
| if (imm)      DST += DATA8/16/32;  else      DST += SRC; |
|  |

**2.7 SUB (UNSIGNED SUBTRACT)**

**Description:**

Subtracts data from a register specified by immediate data or another register. The length of the data to be subtracted is specified in the bottom 3 bits of the instruction if the data is immediate. The behavior is undefined if multiple lengths are specified, or if no length is specified.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 0 1 1 0 | IMM | BYTE | WORD | DWORD | DST REG | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| BYTE/WORD/DWORD 2 |
| DATA8/16/32 |

**Operation:**

|  |
| --- |
| if (imm)      DST -= DATA8/16/32;  else      DST -= SRC; |
|  |

**2.8 AND (BITWISE AND)**

**Description:**

Performs a bitwise AND operation on a register, with a zero-extended immediate value or data from a register. The length of the data to be operated on is specified in the bottom 3 bits of the instruction. The behavior is undefined if multiple lengths are specified, or if no length is specified.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 0 1 1 1 | IMM | BYTE | WORD | DWORD | DST REG | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| BYTE/WORD/DWORD 2 |
| DATA8/16/32 |

**Operation:**

|  |
| --- |
| if (imm)      DST &= DATA8/16/32;  else      DST &= SRC; |
|  |

**2.9 NOT (BITWISE NOT)**

**Description:**

Performs a bitwise NOT operation on a register.

**Memory Representation:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BYTE 0 |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 - 0 | 7 - 4 | 3 - 0 |
| 1 0 0 0 | 0 | 0 0 0 | DST REG | 0 0 0 0 |

**Operation:**

|  |
| --- |
| DST = ~DST; |
|  |

**2.10 CMP (UNSIGNED COMPARE)**

**Description:**

Compare an unsigned register value or unsigned zero-extended immediate with a 4-bit unsigned register, and set flags accordingly. F\_P is set if the difference between the source and destination is positive, F\_E is set if they are equal, and F\_N is set if the difference is negative. The length of the data to be compared is specified in the bottom 3 bits of the instruction. The behavior is undefined if multiple lengths are specified, or if no length is specified.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 1 0 0 1 | IMM | BYTE | WORD | DWORD | DST REG | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| BYTE/WORD/DWORD 2 |
| DATA8/16/32 |

**Operation:**

|  |
| --- |
| R\_CND = 0;  uint32\_t src;  if (imm)      src = (uint32\_t)DATA8/16/32;  else      src = SRC; |
| if (src > DST)      R\_CND |= F\_P;  else if (src == DST)      R\_CND |= F\_E;  else      R\_CND |= F\_N; |

**2.11 BR (BRANCH)**

**Description:**

Jumps to a destination specified by an immediate offset or register if the specified condition is met. The behavior is not well-defined if no condition is specified.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 (if IMM == 0) |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 1 0 1 0 | IMM | P | E | N | 0 0 0 0 | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| BYTE 1 |
| OFFSET32 |

**Operation:**

|  |
| --- |
| if ((P && (R\_CND & F\_P)) ||      (E && (R\_CND & F\_E)) ||      (N && (R\_CND & F\_N))  {      if (imm)          R\_PRG += OFFSET32;      else          R\_PRG = SRC;  } |
|  |

**2.12 JMP (UNCONDITIONAL JUMP)**

**Description:**

Jumps to a destination specified by a sign-extended immediate offset or register.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 (if IMM == 0) |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 1 0 1 1 | IMM | BYTE | WORD | DWORD | 0 0 0 0 | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| BYTE/WORD/DWORD 1 |
| OFFSET8/16/32 |

**Operation:**

|  |
| --- |
| if (imm)      R\_PRG += SignExtend(OFFSET8/16/32);  else      R\_PRG = SRC; |
|  |

**2.13 CALL (CALL SUBROUTINE)**

**Description:**

Pushes the current program counter to the stack and jumps to a destination specified by a sign-extended immediate offset or register.

**Memory Representation:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 (if IMM == 0) |  |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 4 | 3 - 0 |
| 1 1 0 0 | IMM | BYTE | WORD | DWORD | 0 0 0 0 | SRC REG |

**If the IMM flag is set:**

|  |
| --- |
| DWORD 1 |
| OFFSET8/16/32 |

**Operation:**

|  |
| --- |
| R\_SF -= sizeof(int);  mem[R\_SF] = R\_PRG;  if (imm)      R\_PRG += SignExtend(OFFSET8/16/32);  else      R\_PRG= SRC; |
|  |

**2.14 RET (RETURN FROM SUBROUTINE)**

**Description:**

Jumps back to the caller

**Memory Representation:**

|  |  |  |
| --- | --- | --- |
| BYTE 0 |  |  |
| 7 - 4 | 3 | 2 - 0 |
| 1 1 0 1 | 0 | 0 0 0 |

**Operation:**

|  |
| --- |
| R\_PRG = mem[R\_SF];  R\_SF += sizeof(int); |
|  |

**2.15 CX (CALL X86 METHOD)**

**Description:**

Calls a 32-bit x86 function at the specified index in the x86 call table (whose size is stored in the top word of R\_CND), with the arguments passed in general registers, in order starting in R\_A, with the function index in R\_K, and the number of arguments stored in R\_L, so arguments must not be passed in R\_K or R\_L. Returns the result of the call in R\_A. Due to this, the maximum number of arguments that can be passed is 11. Functions must be specified to be FASTCALL, STDCALL, or CDECL, and behavior is not well-defined if multiple are specified, or if none are specified.

**Memory Representation:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| BYTE 0 |  |  |  |  | BYTE 1 |
| 7 - 4 | 3 | 2 | 1 | 0 | 7 - 0 |
| 1 1 1 0 | 0 | FASTCALL | STDCALL | CDECL | INDEX |

**Register Representation:**

|  |  |
| --- | --- |
| R\_A - R\_K | R\_L |
| ARGS[] | ARGC |

**Operation:**

|  |
| --- |
| if (FASTCALL)      R\_A = CallFastcall(x86CallTbl[INDEX], ARGS[], ARGC);  else if (STDCALL)      R\_A = CallStdcall(x86CallTbl[INDEX], ARGS[], ARGC);  else if (CDECL)      R\_A = CallCdecl(x86CallTbl[INDEX], ARGS[], ARGC); |
|  |

**2.16 TRAP (TRAP ROUTINE)**

**Description:**

Executes a routine specified in the table below. Arguments are passed in the specified registers, and return values are returned in R\_A;

**Memory Representation:**

|  |  |  |  |
| --- | --- | --- | --- |
| BYTE 0 |  |  | BYTE 1 |
| 7 - 4 | 3 | 2 - 0 | 7-0 |
| 1 1 1 1 | 0 | 0 0 0 | TRAPCODE |

**Register Representation:**

|  |
| --- |
| R\_A - R\_K |
| ARGS[] |

**Trap Table:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| TRAPNUM | ROUTINE | R\_A | R\_B | R\_C | R\_D | R\_E | Return |
| 0x0 | PUTC | char c |  |  |  |  | void |
| 0x1 | PUTS | char\* str |  |  |  |  | size\_t |
| 0x2 | GETCHAR |  |  |  |  |  | int |
| 0x3 | SOCKET | int domain | int type | int protocol |  |  | int |
| 0x4 | LISTEN | int fd | int backlog |  |  |  | int |
| 0x5 | BIND | int fd | const sockaddr\* addr | socklen\_t len |  |  | int |
| 0x6 | ACCEPT | int fd | const sockaddr\* addr | socklen\_t\* len |  |  | int |
| 0x7 | SELECT | int nfds | fd\_set\* readfds | fd\_set\* writefds | fd\_set\* exceptfds | timeval\* timeout | int |
| 0x8 | RECV | int fd | void\* buf | size\_t len | int flags |  | int |
| 0x9 | SEND | int fd | const void\* buf | size\_t len | int flags |  | int |
| 0xA | SHUTDOWN | int socket | int how |  |  |  | int |
| 0xB | HALT |  |  |  |  |  | void |

**Operation:**

|  |
| --- |
| trapTable[TRAPCODE](ARGS[]); |
|  |