

The BELLE-ISA Manual

v3

(Big Endian, Low Level Emulator Instruction Set Architecture)

A 16-bit RISC CPU architecture

The emulator, assembler, and
disassembler are available [on GitHub](#)

Overview

Memory

The BELLE-ISA can address 2^{16} addresses, numbered from $0x0000$ to $0xFFFF$. Addresses are all general purpose. The *first address available for a program* is address $0x0000$. By default, programs begin execution at address $0x64$ (**100**), with the stack and base pointer expanding to **lower addresses**, starting from $0x63$ (**99**).

Addresses $0xFF$ - $0x9C9$ are used for video memory. It can be written to and read from as normal, but do note that characters in video memory will be displayed. Aside from this, memory **can be used for any purpose**. All memory cells are 16-bits wide.

Call stack

The call stack is defined based on the addresses the **stack pointer** and **base pointer** contain. The call stack can expand “*upwards*” (to higher addresses) or “*downwards*” (to lower addresses), depending on the position of the stack and base pointers *relative to each other*. It is recommended that the call stack expands **downwards** to prevent overriding program memory.

A **common setup** of memory for a program is as follows

| $0x0 \dots 0x30$ | $0x30 \dots 0x40$ | $0x40 \dots 0x50$ | $0x50 \dots 0xFFFF$ |
|------------------|-------------------|-------------------|---------------------------------|
| Call stack | Constants | Dynamic data | Program (video memory included) |

All memory is **R/W**. Accessing uninitialized memory addresses will generate a **segmentation fault**. Memory is *all uninitialized by default*, until a **ROM** is loaded into memory.

Instructions

Instructions are 16-bits wide. Instructions can contain *certain bits* to identify the type of the next operand.

The BELLE-ISA specifies 20 different instructions that the processor can execute. Instructions take **zero, one, or two** operands, and the operands can *be of different types*.

Instructions can operate on *memory, registers, immediate values, flags*, and affect the *running state of the processor*.

Addressing modes

The BELLE-ISA can be addressed with *register immediate, register indirect, memory value, memory direct, memory indirect*, and *immediate value*. No offset addressing exists on the BELLE-ISA.

- Register immediate: The value at a register
- Register indirect: The value at a memory address at a register
- Memory value: The value *of* a memory address (e.g. `0xff` would refer to `0xff`, not the value at address `0xff`)
- Memory direct: The value *at* a memory address
- Memory indirect: The value *at* memory address *at* a memory address
- Immediate value: A value

Exceptions

A variety of exceptions, or errors, can occur, for a variety of reasons.

Errors include **segmentation faults, illegal instructions, divide by zero, invalid register, stack overflows, and stack underflows**. Execution will immediately halt upon an unrecoverable error, and the machine will enter a crash state.

Registers

8 general purpose registers, numbered 0-7 are present on the CPU. Register 8 is the program counter, and register 9 is the stack pointer. Register 8 and 9 can be accessed in *some instructions*. Registers 0-7 can be accessed as *any operand of most instructions*.

There is also a 16-bit wide **instruction register** that holds the *current instruction* being executed, however this register cannot be directly read or written to.

Program counter - **R8**

Holds the address of the *next instruction* in memory. *16 bits wide*.

Stack pointer - **R9**

Points to the *current top of the call stack*. *16 bits wide*.

| General purpose registers (GPRs) | | Program counter | Stack pointer |
|----------------------------------|---------------|-----------------|---------------|
| R0-R5 | R6-R7 | R8 | R9 |
| 16-bit | 32-bit, float | 16-bit | 16-bit |

Flags

4 total flags are present on the CPU. 3 control flags exist on the CPU to represent the result of a **compare** instruction. Sign, overflow, and zero. The 4th flag, the **remainder flag**, becomes set if a division instruction *creates a remainder*. Flags can be read with a *jump instruction* to conditionally branch to an address in memory. Flags can also be set with certain interrupt codes.

| Flag | Zero (Z) | Sign (S) | Overflow (O) | Remainder (R) |
|-----------|----------------------|-------------------|-------------------------|------------------------|
| Set by | CMP L, R instruction | | Arithmetic instructions | DIV instruction |
| Condition | L = R | L < R | Register overflow | Has remainder |

ROM structure

All bytes in a ROM are stored in *big-endian*.

| Bytes | 0x0 .. 0x2 | 0x2 .. 0x4 | 0x2 |
|-------|----------------|-------------------------------|-------------------------------------|
| Use | Binary version | <code>.start</code> directive | Rest of program, including metadata |

In addition to instructions that are executed at runtime, the CPU has “**pseudo-instructions**” that set values in the CPU. They are executed *one time* when the **ROM** is loaded into memory, and affect the state of the CPU *at runtime*.

However, at *runtime*, these values can still be loaded into memory addresses.

Pseudo-instructions

| Name | Encoding | Function |
|---------------------|----------|--|
| <code>.start</code> | 0x02XX | Set the <i>starting address</i> of the program to the value of XX. The 3rd and 4th byte of the binary. |
| ---- | 0x010X | Version of the binary, denoted by the X |
| <code>.data</code> | 0x01XX | Metadata, an ASCII character denoted by the XX |

Instruction set

The **BELLE-ISA** has a reduced, but complete instruction set, containing **20** different instructions that are all **16** bits wide. The first **4** bits of an instruction denote the *opcode*, with the remaining **12** denoting the *operands* and operand *types*.

Notation

This table describes the **notation** that will be used to refer to *different operand types*. **x** is used in the table to refer to the amount of bits for the operand.

| Notation | Description |
|-----------|--|
| reg[x] | Register Direct - Value <i>in</i> register, R0-7 or R0-9 |
| regi[x] | Register Indirect - Value <i>at</i> address <i>in</i> register, R0-7 or R0-9 |
| mem[x] | Memory address direct - Value <i>at</i> a memory address |
| memi[x] | Memory address indirect - Value <i>at</i> the address <i>at</i> a memory address |
| memv[x] | Memory address value - Value <i>of</i> a memory address |
| imm[x] | Immediate signed value - Value <i>of</i> immediate value |
| ignore[x] | Ignored - Bits can be in any state and instruction still executes |
| var[x] | Varied - Bits will be defined later on in the document |
| opcode[x] | An opcode for an instruction |

Later in this document, *each instruction* will be documented, including the *possible instruction operands*, and the encoding for the instruction with *each type of operands*.

Refer back to the addressing modes section of this document to see a full description of all available addressing modes.

Pseudocode conventions

The instruction documentation uses *pseudocode* to represent the instruction in a concise, human-readable format. However, there are some conventions and consistencies within the pseudocode.

Below is a list of **symbols** in *pseudocode* and their function.

| Symbol | Description |
|--------------|--|
| INS(. . .) | Perform <i>an instruction</i> with the given operand(s) |
| Xflag | The value of one of the <u>CPU's flags</u> |
| operand | An operand's value, detailed in the <u>addressing mode section</u> |
| pc, sp, bp | The program counter or stack/base pointer, respectively |
| memory[i] | The value <i>at</i> memory address i |
| oob(X) | Returns true if X is <i>less than</i> 0x0 or <i>greater than</i> 0xFFFF |
| setpc(X) | Set the program counter to X and throw a segfault if OOB |
| throw X | Generate an <i>error</i> X |
| ARITH-OP | Any math operation. Set the overflow flag if an overflow occurred |

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|------|-------|----|----|----|----------------|----|---|--------|--------|---|---|---|---|--------|---|---|--|
| HLT | 0000 | | | | ignore[12] | | | | | | | | | | | | |
| ADD | 0001 | | | | reg[3] | | | var[9] | | | | | | | | | |
| BO | 00100 | | | | var[11] | | | | | | | | | | | | |
| BNO | 00101 | | | | var[11] | | | | | | | | | | | | |
| POP | 0011 | | | | var[12] | | | | | | | | | | | | |
| DIV | 0100 | | | | reg[3] | | | var[9] | | | | | | | | | |
| RET | 0101 | | | | 0000 0000 0000 | | | | | | | | | | | | |
| BL | 01010 | | | | var[11] | | | | | | | | | | | | |
| BG | 01011 | | | | var[11] | | | | | | | | | | | | |
| LD | 0110 | | | | reg[3] | | | var[9] | | | | | | | | | |
| ST | 0111 | | | | var[9] | | | | | | | | | reg[3] | | | |
| JMP | 10000 | | | | var[11] | | | | | | | | | | | | |
| BZ | 10010 | | | | var[11] | | | | | | | | | | | | |
| BNZ | 10011 | | | | var[11] | | | | | | | | | | | | |
| CMP | 1010 | | | | reg[3] | | | var[9] | | | | | | | | | |
| NAND | 1011 | | | | reg[3] | | | var[9] | | | | | | | | | |
| PUSH | 1100 | | | | 000 | | | var[9] | | | | | | | | | |
| INT | 1101 | | | | 000 | | | 1 | imm[8] | | | | | | | | |
| MOV | 1110 | | | | reg[3] | | | var[9] | | | | | | | | | |
| LEA | 1111 | | | | reg[3] | | | var[9] | | | | | | | | | |

HLT - Halt

Description

Halt the CPU. As long as the HLT **opcode** is read, the instruction will be *treated as* a HLT instruction. This means that attempting to execute data *such as ASCII characters* will simply halt the processor.

Operands and format

No operands

HLT

Encoding

| | |
|------|------------|
| 0000 | ignore[12] |
|------|------------|

Examples

```
HLT ; Stop execution
```

ADD – Add two operands

Description

Add a value from a *source* to a *destination*, storing the **result in the destination**.
ADD can take a negative number, allowing it to perform subtraction.

If an ADD destination is a *floating point register* but the source is not, the source is *implicitly cast to a float*.

Operands and format

LHS - Register

RHS - Register, Register Indirect, Memory Indirect, Immediate

ADD LHS, RHS

Encodings

| | | | | | |
|------|--------|---|--------|--|--|
| 0001 | reg[3] | 1 | imm[8] | | |
|------|--------|---|--------|--|--|

| | | | | |
|------|--------|--------|--|--------|
| 0001 | reg[3] | 0000 0 | | reg[4] |
|------|--------|--------|--|--------|

| | | | | | |
|------|--------|----|---|----|---------|
| 0001 | reg[3] | 00 | 1 | 00 | regi[4] |
|------|--------|----|---|----|---------|

| | | | | | |
|------|--------|---|---|---------|--|
| 0001 | reg[3] | 0 | 1 | memi[7] | |
|------|--------|---|---|---------|--|

Example

```
ADD R5, 3 ; Add 3 to register 5
```

Pseudocode

```
lhs += rhs
```

POP – Pop from the stack

Description

First, read the value in memory at the *stack pointer*. If the stack pointer points to an uninitialized address, a stack underflow will be generated. Then **copy the value** into the destination and uninitialized the address the stack pointer is at. Uninitialize the address in the stack pointer. Finally, *change* the stack pointer based on the *base pointer's* value.

Operands and format

DST - Memory address or register

POP DST

Encodings

| | | |
|------|-----------|--------|
| 0011 | 0000 0000 | reg[4] |
|------|-----------|--------|

| | | |
|------|---|---------|
| 0011 | 1 | mem[11] |
|------|---|---------|

Examples

```
POP R4 ; Pop into register 4
```

Pseudocode

```
if memory[sp] == NULL {  
    throw underflow  
}  
dst = memory[sp]  
memory[sp] = NULL  
if sp > bp {  
    sp -= 1  
} else if sp < bp {  
    sp += 1  
} // if sp == bp, sp is unchanged
```

DIV - Divide two operands

Description

Divide a value from the *destination* by a *source*, storing the **result in the destination**. DIV can take a negative number.

If a DIV destination is a *floating point register* but the source is not, the source is *implicitly cast to a float*.

Operands and format

LHS - Register

RHS - Register, Register Indirect, Memory Indirect, Immediate

DIV LHS, RHS

Encodings

| | | | | | |
|------|--------|---|--------|--|--|
| 0100 | reg[3] | 1 | imm[8] | | |
|------|--------|---|--------|--|--|

| | | | | |
|------|--------|--------|--|--------|
| 0100 | reg[3] | 0000 0 | | reg[4] |
|------|--------|--------|--|--------|

| | | | | | |
|------|--------|----|---|----|---------|
| 0100 | reg[3] | 00 | 1 | 00 | regi[4] |
|------|--------|----|---|----|---------|

| | | | | | |
|------|--------|---|---|---------|--|
| 0100 | reg[3] | 0 | 1 | memi[7] | |
|------|--------|---|---|---------|--|

Examples

```
DIV R0, 3 ; Divide r0 by 3
```

Pseudocode

```
lhs /= rhs  
rflag = lhs % rhs == 0
```

RET - Return

Description

Pop the last value from the call stack into the *program counter* and continue execution. This instruction **is equivalent to** POP R8.

Note: a BL instruction to address 0x0 *will* be interpreted as a RET instruction, as the encodings are *identical* and they *share the same opcode*.

Operands and format

No operands
RET

Encoding

| | |
|------|----------------|
| 0101 | 0000 0000 0000 |
|------|----------------|

Example

RET ; Return

Pseudocode

POP(pc)

LD - Load direct

Description

Load a value from a memory address into a register. If the address is uninitialized, throw a segmentation fault.

Operands and format

LHS - Register

RHS - Memory address

LD LHS, RHS

Encoding

| | | |
|------|--------|--------|
| 0110 | reg[3] | mem[9] |
|------|--------|--------|

Example

```
LD R4, [NEAT] ; Load the value at label NEAT into R4
```

Pseudocode

```
if memory[rhs] == NULL || oob(rhs) {  
    throw segfault  
}  
lhs = memory[rhs]
```

ST – Store direct and indirect

Description

Store a value from a **register** to a *memory address* or register indirect. If the destination is out of bounds, throw a segmentation fault.

Operands and format

LHS - Memory address or register indirect

RHS - Register

ST LHS, RHS

Encodings

| | | | |
|------|---|----------|----------|
| 0111 | 0 | mem[8] | reg[3] |
|------|---|----------|----------|

| | | | | |
|------|---|----------|------|----------|
| 0111 | 1 | reg[4] | 0000 | reg[3] |
|------|---|----------|------|----------|

Example

```
ST [0x45], R2 ; Store R2's value at address 0x45
```

Pseudocode

```
if oob(lhs) {  
    throw segfault  
}  
memory[lhs] = rhs
```

JMP - Unconditional jump

Description

Unconditionally set the *program counter* to the value of a given *destination*. Destinations can be register indirects or memory addresses. The return address is ***not*** pushed onto the stack.

Operands and format

DST - Memory address or register indirect

JMP DST

Encodings

| | | | |
|-------|---|-----------|-----------|
| 10000 | 0 | mem[10] | |
| 10000 | 1 | 0000 00 | regi[4] |

Example

JMP APPLE ; Jump to the address of the APPLE label

Pseudocode

setpc(dst)

Bcc - Branch if condition

Description

If the **condition is met**, push the current value of the program counter onto the call stack (return address). Then, set the *program counter* to the specified memory address or the address in a register if a register indirect is the destination. If the condition is **not** met, do not branch, and move onto the next instruction.

Operands and format

DST - Memory address or register indirect

Bcc DST

Encodings

If the inverted instruction (BNO/BNZ/BG) is to be executed, the *iv* bit will be **on**.

| Instruction | Opcode | Condition |
|-------------|--------|-----------|
| BO | 0010 | OFlag |
| BZ | 1001 | ZFlag |
| BL | 0101 | SFlag |

| | | | | |
|-----------|----|---|---------|--------|
| opcode[4] | iv | 1 | 0000 00 | reg[4] |
|-----------|----|---|---------|--------|

| | | | |
|-----------|----|---|---------|
| opcode[4] | iv | 0 | mem[10] |
|-----------|----|---|---------|

Examples

```
BL [0x3] ; Branch if less to address 0x3
```

Pseudocode

```
if condition {  
    PUSH(pc)  
    setpc(dst)  
}
```

CMP – Compare two operands

Description

CMP will *read the values of two operands*, then set **flags** on the processor depending on the *values* of the *operands*. If the left side of the instruction is less than the right side, the **sign flag** will be set. If both values are equal, the **zero flag** will be set. The flags will be **unset** for the inverse conditions.

Operands and format

LHS - Register

RHS - Register, Register Indirect, Memory Indirect, Immediate

CMP LHS, RHS

Encodings

| | | | | | |
|------|--------|---|--------|--|--|
| 1010 | reg[3] | 1 | imm[8] | | |
|------|--------|---|--------|--|--|

| | | | | |
|------|--------|--------|--|--------|
| 1010 | reg[3] | 0000 0 | | reg[4] |
|------|--------|--------|--|--------|

| | | | | | |
|------|--------|----|---|----|---------|
| 1010 | reg[3] | 00 | 1 | 00 | regi[4] |
|------|--------|----|---|----|---------|

| | | | | | |
|------|--------|---|---|---------|--|
| 1010 | reg[3] | 0 | 1 | memi[7] | |
|------|--------|---|---|---------|--|

Example

```
CMP R1, 33 ; Compare the value in R1 with 33
```

Pseudocode

```
sflag = lhs < rhs  
zflag = lhs == rhs
```

NAND – NAND two operands

Description

NAND all the bits of a destination and a source, storing the **result in the destination**.

Operands and format

LHS - Register

RHS - Register, Register Indirect, Memory Indirect, Immediate

NAND LHS, RHS

Encodings

| | | | | | |
|------|--------|---|--------|--|--|
| 1011 | reg[3] | 1 | imm[8] | | |
|------|--------|---|--------|--|--|

| | | | | |
|------|--------|--------|--|--------|
| 1011 | reg[3] | 0000 0 | | reg[4] |
|------|--------|--------|--|--------|

| | | | | | |
|------|--------|----|---|----|---------|
| 1011 | reg[3] | 00 | 1 | 00 | regi[4] |
|------|--------|----|---|----|---------|

| | | | | | |
|------|--------|---|---|---------|--|
| 1010 | reg[3] | 0 | 1 | memi[7] | |
|------|--------|---|---|---------|--|

Example

```
NAND R3, R8 ; NAND R3 with the program counter
```

Pseudocode

```
lhs = !(lhs & rhs)
```

PUSH – Push onto the stack

Description

Read the **value** in the *stack pointer*. Then, change the **value** of the stack pointer based on the base pointer's value. Lastly, copy the value from the *source* to the address of the stack pointer.

Operands and format

SRC - Register, Immediate

PUSH SRC

Encodings

| | | | |
|------|-----|---|--------|
| 1100 | 000 | 1 | imm[8] |
|------|-----|---|--------|

| | | | |
|------|-----|--------|--------|
| 1100 | 000 | 0000 0 | reg[4] |
|------|-----|--------|--------|

Example

`PUSH 4 ; Push the value of 4 onto the stack`

Pseudocode

```
if sp > bp {  
    sp += 1  
} else if sp < bp {  
    sp -= 1  
} // if sp == bp, sp is unchanged  
  
memory[sp] = src
```

INT - Interrupt/"pseudo instruction"

Description

INT will pause the program, or interrupt it, and perform a task before returning to execution. INT takes a single argument, and depending on its value, it will perform a different task. INT allows I/O to take place, as well as unconditionally setting flags and generating errors. Interrupts are not blocks of code in memory.

| Code | Action |
|-----------|---|
| -6 . . -1 | Generate a <u>stack underflow</u> , <u>segmentation fault</u> , <u>illegal instruction</u> , <u>divide by zero</u> , <u>invalid register</u> , or <u>stack overflow exception</u> , respectively. |
| 0 . . 7 | Print the value in a register, then a newline |
| 8 | Print the values in memory from the address in R0 to the address in R1 to the console. If an address is uninitialized, simply <u>skip</u> it |
| 9 | Read a single character from standard input without echoing it back to the console, storing its ASCII value in R0 |
| 10 | Sleep for the number of 10ths of a second specified in R4 |
| 11 . . 13 | Set/unset/invert the zero flag |
| 21 . . 23 | Set/unset/invert the overflow flag |
| 31 . . 33 | Set/unset/invert the remainder flag |
| 40 | Read and echo a line and parse it to a 16 bit integer that is loaded into R0 |
| 41 . . 43 | Set/unset/invert the sign flag |
| 60 | Set the stack pointer to the value in R4 |
| 61 | Set the base pointer to the value in R4 |
| 70 | Bcc instructions will push return addresses (true by default) |
| 71 | Bcc instructions will not push return addresses |

Continued on next page

INT (cont..)

Operands and format

VAL - Immediate

INT VAL

Encoding

| | | | |
|------|-----|---|--------|
| 1101 | 000 | 1 | imm[8] |
|------|-----|---|--------|

Example

```
INT 7 ; Print the value of R7
```

Pseudocode

```
// None available here...
```

MOV – Move

Description

MOV will read the value of the **source operand**, and *copy* it to the **destination operand**. The destination cannot be the *program counter* or *stack pointer*. Use a JMP instruction to modify the *program counter*'s value.

Operands and format

LHS - Register

RHS - Register, Register Indirect, Memory Indirect, Immediate

MOV LHS, RHS

Encodings

| | | | | | |
|------|--------|---|--------|--|--|
| 1110 | reg[3] | 1 | imm[8] | | |
|------|--------|---|--------|--|--|

| | | | | |
|------|--------|--------|--|--------|
| 1110 | reg[3] | 0000 0 | | reg[4] |
|------|--------|--------|--|--------|

| | | | | | |
|------|--------|----|---|----|---------|
| 1110 | reg[3] | 00 | 1 | 00 | regi[4] |
|------|--------|----|---|----|---------|

| | | | | | |
|------|--------|---|---|---------|--|
| 1110 | reg[3] | 0 | 1 | memi[7] | |
|------|--------|---|---|---------|--|

Example

```
MOV R3, R4 ; Copy R4 into R3
```

Pseudocode

```
lhs = rhs
```

LEA - Load effective address

Description

Copy the value *of* a memory address into a register. Unlike LD, LEA does not copy the value *at* the address, but rather the *address itself* is copied. This means that a LEA instruction with [0xFF] as its source will load 0xFF and **not** the *value at* 0xFF

Operands and format

LHS - Register

RHS - Memory address

LEA LHS, RHS

Encoding

| | | |
|------|--------|---------|
| 1111 | reg[3] | memv[9] |
|------|--------|---------|

Example

```
LEA R4, [BEEP] ; Load the address of label BEEP into R4
```

Pseudocode

```
lhs = rhs
```


Video memory and screen

Video memory begins at `0xFF` and ends at `0x9C9`. Each **word** in video memory has its lower 8 bits read as an **ASCII** value, with the corresponding character *displayed on screen*. The screen can display 75 characters horizontally, and 30 characters vertically. The emulator's screen is approximately **680** by **480** pixels in width and height, respectively, with a **16** size font. The display runs at 60FPS in the emulator. Uninitialized addresses in video memory are treated as spaces.

Screen

