Boneless-III

Architecture Reference Manual

Notice:

This document is a work in progress and subject to change without warning. However, the parts that are *especially* subject to change carry a notice similar to this one.

Contents

1 Introduction

TBD

2 Guide to Instruction Set

This guide first explains how to interpret the notation used in this document. After, it explains the available registers and their behavior.

2.1 Operation Syntax

This document uses the following syntax and operators to describe the operation of each instruction.

2.1.1 Undefined and Unpredictable Behavior

To describe the boundaries of legal program behavior, this document uses the words **UNDEFINED** and **UNPREDICTABLE**.

When execution encounters **UNPREDICTABLE** behavior, the implementation may perform any behavior, including but not limited to hanging and failing to continue execution. The resulting behavior may be different between executions even under the same circumstances.

Certain operations, including any operation with an **UNDEFINED** input, will produce an **UNDEFINED** result. Reading a register whose value is currently **UNDEFINED** may produce any bit pattern. Multiple consecutive reads of such a register may also produce different bit patterns on each read.

2.1.2 Reference Operators

The following operators reference parts of variables or the attached memory.

- `opB` ← `opA`: Store `opA` into `opB`. If necessary, `opA` is implicitly zero-extended or truncated to match the length of `opB`.
- `op[b:a]`: Reference bits `a` through `b`, inclusive, of `op`. If `a` is greater than `b`, the resulting length is zero.
- `mem[addr]`: Reference memory word at word address `addr`. The address is implicitly ANDed with `0xFFFF`.
- `ext[addr]`: Reference external bus word at word address `addr`. The address is implicitly ANDed with `0xFFFF`.
- `{opA, opB}`: Concatenate the bits of `opA` and `opB`. `opA` makes the high-order bits of the result and `opB` makes the low-order bits.
- `opB{opA}`: Construct the result by repeating `opA` `opB` times.

2.1.3 Arithmetic Operators

The arithmetic operators perform arithmetic or bitwise logic between the operands. All operands to these operators are unsigned. If one operand is shorter than the other, it is zero-extended to match the length of the other.

- `opA + opB`: Add `opA` and `opB`. The high bit of the result is a carry bit.
- `opA` **`and`** `opB`: Perform a bitwise AND between `opA` and `opB`.
- `opA` **`or`** `opB`: Perform a bitwise OR between `opA` and `opB`.
- `opA` **`xor`** `opB`: Perform a bitwise XOR between `opA` and `opB`.
- **`not`** `op`: Perform a bitwise negation of `op`.

2.1.4 Logical Operators

The logical operators yield 1 if the condition is satisfied and 0 if it is not. If one operand is shorter than the other, it is zero-extended to match the length of the other.

- `opA = opB`: Satisfied if `opA` equals `opB`.
- `opA <> opB`: Satisfied if `opA` does not equal `opB`.

2.1.5 Functions

- `sign_extend_16(op)`: Perform a two's complement sign extension of `op` by replicating the high bit until the total length is 16 bits.
- `decode_imm_al(op)`: Calculate the immediate value of an arithmetic or logical instruction according to the following table.

op	Result
0	`0x0000`
1	`0x0001`
2	`0x8000`
3	`TBD`
4	`0x00FF`
5	`0xFF00`
6	`0x7FFF`
7	`0xFFFF`

- `decode_imm_sr(op)`: Calculate the immediate value of a shift or rotate instruction according to the following table.

op	Result
0	8
1	1
2	2
3	3
4	4
5	5
6	6
7	7

2.2 Registers

The CPU contains a number of registers that are used to store data, computation results, and system state. The registers are operated on as described by the operation of each instruction. If a register is not modified by an instruction, its value is preserved, except where noted in this register definition.

2.2.1 Program Counter

The CPU features a 16-bit program counter named `PC`. All instructions are 16 bit, so all values of `PC` are valid.

Unless otherwise specified by an instruction's operation, `PC` ← `PC+1` after each instruction.

The behavior of `PC` ← `op` defines that `op` is truncated to 16 bits to fit `PC`. Thus, adjusting `PC` is defined to wrap.

At reset, `PC` ← `0x0000`, but this value can be changed by the implementation.

2.2.2 General Purpose Registers and the Window

The CPU has eight 16-bit general purpose (GP) integer registers, named `R0` through `R7`. Each register can hold any 16-bit value. Most instructions interpret the values as unsigned integers, but some treat values as two's complement signed integers (denoted by a "signed" operation).

The GP registers are fully interchangeable and any register can be used as source and/or destination for any instruction which uses a GP register. The register file is windowed: register values are stored in main memory, starting at the window address.

The current window address is stored in a 16-bit register named `W`. Its value can only be set and/or read by the four window instructions: `ADJW`, `LDW`, `STW`, and `XCHW`. Setting `W` logically changes the values of all GP registers simultaneously, enabling fast procedure calls and task switches.

`W` is added to the number of a GP register to calculate the address where that register is stored. For example, the operation `mem[W+3]` ← `mem[W+1]` sets `R3` equal to `R1`.

The behavior of `W` $\leftarrow$ `op` and `mem[op]` defines that `op` is truncated to 16 bits to fit `W`. Thus, adjusting `W` and calculating GP register addresses are defined to wrap.

At reset, `W` $\leftarrow$ `0xFFF8`, but this value can be changed by the implementation.

2.2.3 Result Flags

The CPU has four result flags to describe the result of ALU computations. The flags are updated by most ALU instructions. The CPU can act on the result of the flags by executing a conditional branch which transfers control if the flags are in the desired state. The exact contents of each flag after an instruction executes are explained in the instruction's operation, but the general purpose and behavior of each flag are explained below.

The `Z` (Zero) flag is set to 1 if the low 16 bits of the result of the operation were zero, and 0 otherwise.

The `S` (Sign) flag is set to the 15th bit of the result of the operation.

The `C` (Carry) flag is set to the 16th bit of the result of an arithmetic operation, or **UNDEFINED** if the operation was logical.

The `V` (oVerflow) flag is set if the arithmetic operation encountered two's complement overflow, or **UNDEFINED** if the operation was logical.

2.2.4 Extended Immediate

The CPU has two registers that help build a 16-bit immediate value.

The `EXTI` instruction sets the `ext13` register to its 13-bit immediate and the `has_ext13` register to 1.

Generally, if an instruction can use an immediate and `has_ext13` is 1, the instruction will use `ext13` as the high 13 bits of the the immediate value and take the low 3 bits from the instruction itself. The exact behavior of an instruction with regards to `has_ext13` and `ext13` is specified in the instruction's operation.

Except for after `EXTI`, `ext13` $\leftarrow$ **`UNDEFINED`** and `has_ext13` $\leftarrow$ `0` after every instruction, even those which do not use either register.

3 Quick Reference

This chapter summarizes the Boneless instruction set. Instructions are grouped according to their function.

3.1 ALU Instructions

3.1.1 Arithmetic

Mnemonic	Function

`ADD Rd, Ra, Rb`
`ADDI Rd, Ra, imm`
	Add register to register/immediate.

`ADC Rd, Ra, Rb`
`ADCI Rd, Ra, imm`
	Add register to register/immediate, including carry input. For multi-word addition.

`SUB Rd, Ra, Rb`
`SUBI Rd, Ra, imm`
	Subtract register/immediate from register.

`SBC Rd, Ra, Rb`
`SBCI Rd, Ra, imm`
	Subtract register/immediate from register, including carry input. For multi-word subtraction.

`CMP Ra, Rb`
`CMPI Ra, imm`
	Compare register with register/immediate, then set flags according to result.

3.1.2 Logic

Mnemonic	Function

`AND Rd, Ra, Rb`
`ANDI Rd, Ra, imm`
	Bitwise AND between register and register/immediate.

`OR Rd, Ra, Rb`
`ORI Rd, Ra, imm`
	Bitwise OR between register and register/immediate.

`XOR Rd, Ra, Rb`
`XORI Rd, Ra, imm`
	Bitwise XOR between register and register/immediate.

`NOT Rd, Ra`	Bitwise NOT between register and register.

3.1.3 Shift & Rotate

Mnemonic	Function

`SLL Rd, Ra, Rb`
`SLLI Rd, Ra, imm`
	Shift register left by register/immediate amount.

`SRL Rd, Ra, Rb`
`SRLI Rd, Ra, imm`
	Shift register right by register/immediate amount, with zero extension.

`SRA Rd, Ra, Rb`
`SRAI Rd, Ra, imm`
	Shift register right by register/immediate amount, with sign extension.

`ROL Rd, Ra, Rb`
`ROLI Rd, Ra, imm`
	Rotate register left by register/immediate amount.

`RORI Rd, Ra, imm`	Rotate register right by immediate amount.

3.2 Data Transfer Instructions

3.2.1 General Purpose Registers

Mnemonic	Function

`MOV Rd, Rs`
`MOVI Rd, imm`
	Move register/immediate into register.

`MOVR Rd, off`	Move PC-relative offset into register.
`XCHG Ra, Rb`	Exchange values of two registers.
`NOP`	Do nothing.
`EXTI imm`	Extend immediate of following instruction. Automatically placed by assembler; should not be manually written.

3.2.2 Window Register

Mnemonic	Function
`ADJW imm`	Add signed immediate to `W`.
`LDW Rd, imm`	Add signed immediate to `W`, then store previous `W` to register.
`STW Rb`	Move register into `W`.
`XCHW Rd, Rb`	Move `Rb` into `W`, then store previous `W` to `Rd`.

3.2.3 Memory

Mnemonic	Function
`LD Rd, Ra, off`	Load `Rd` from memory at `Ra` plus offset.
`LDR Rd, Ra, off`	Load `Rd` from memory at `Ra` plus PC-relative offset.
`ST Rs, Ra, off`	Store `Rs` to memory at `Ra` plus offset.
`STR Rs, Ra, off`	Store `Rs` to memory at `Ra` plus PC-relative offset.

3.2.4 External Bus

Mnemonic	Function
`LDX Rd, Ra, off`	Load `Rd` from external bus at `Ra` plus offset.
`LDXA Rd, off`	Load `Rd` from external bus at absolute offset.
`STX Rs, Ra, off`	Store `Rs` to external bus at `Ra` plus offset.
`STXA Rs, off`	Store `Rs` to external bus at absolute offset.

3.3 Control Transfer Instructions

3.3.1 Unconditional

Mnemonic	Function
`J label`	Jump to `label`.
`JAL Rd, label`	Store address of next instruction into register, then jump to `label`. For calling subroutines.
`JR Rs, off`	Jump to `Rs` plus PC-relative offset.
`JRAL Rd, Rb`	Store address of next instruction into `Rd`, then jump to `Rb`.
`JST Rs, off`	Jump through entry `Rs` of switch table at PC-relative offset.
`JVT Rd, off`	Jump through entry `off` of virtual table at `Rd`.

3.3.2 Conditional on Comparison

These instructions branch to `label` after a `CMP Ra, Rb` instruction if the given condition is met when the operands are treated as numbers with the given signedness.

Condition	Signed Comparison	Unsigned Comparison
`Ra = Rb`	`BEQ label`	`BEQ label`
`Ra > Rb`	`BGTS label`	`BGTU label`
`Ra >= Rb`	`BGES label`	`BGEU label`
`Ra <= Rb`	`BLES label`	`BLEU label`
`Ra < Rb`	`BLTS label`	`BLTU label`
`Ra <> Rb`	`BNE label`	`BNE label`

3.3.3 Conditional on Result

These instructions branch to `label` if the last arithmetic or logical operation met the given condition.

Condition	is True	is False
Result is equal to zero	`BZ label`	`BNZ label`
Result is negative	`BS label`	`BNS label`
Operation encountered unsigned overflow	`BC label`	`BNC label`
Operation encountered signed overflow	`BV label`	`BNV label`

3.3.4 Conditional on Flags

These instructions branch to `label` if the given flag is in the given state.

Flag	is Set	is Clear
Zero	`BZ1 label`	`BZ0 label`
Sign	`BS1 label`	`BS0 label`
Carry	`BC1 label`	`BC0 label`
oVerflow	`BV1 label`	`BV0 label`

4 List of Instructions

The following pages provide a detailed description of instructions, arranged in alphabetical order.

Executing any instruction with an encoding not present on the following pages has **UNPREDICTABLE** behavior.

4.1 ADC Add Register with Carry

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`ADC`	`00010`					Rd			Ra			`01`		Rb		

Assembly:

```
ADC Rd, Ra, Rb
```

Purpose:

To add 16-bit integers in registers, with carry input.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← opA + opB + C
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = opB[15]) and (opA[15] <> res[15])
```

Remarks:

A 32-bit addition with both operands in registers can be performed as follows:

```
; Perform {R1, R0} ← {R3, R2} + {R5, R4}
    ADD  R0, R2, R4
    ADC  R1, R3, R5
```

4.2 ADCI Add Immediate with Carry

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`ADCI`	`00011`					Rd			Ra			`01`		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`ADCI`	`00011`					Rd			Ra			`01`		imm3		

Assembly:

```
ADCI Rd, Ra, imm
```

Purpose:

To add a constant to a 16-bit integer in a register, with carry input.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_al(imm3)
res ← opA + opB + C
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = opB[15]) and (opA[15] <> res[15])
```

Remarks:

A 32-bit addition with a register and an immediate operand can be performed as follows:

```
; Perform {R1, R0} ← {R3, R2} + 0x40001
    ADDI R0, R2, 1
    ADCI R1, R3, 4
```

4.3 ADD — Add Register

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`ADD`	`00010`					Rd			Ra			`00`		Rb		

Assembly:

```
ADD Rd, Ra, Rb
```

Purpose:

To add 16-bit integers in registers.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← opA + opB
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = opB[15]) and (opA[15] <> res[15])
```

4.4 ADDI Add Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`ADDI`	`00011`					Rd			Ra			`00`		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`ADDI`	`00011`					Rd			Ra			`00`		imm3		

Assembly:

```
ADDI Rd, Ra, imm
```

Purpose:

To add a constant to a 16-bit integer in a register.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_al(imm3)
res ← opA + opB
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = opB[15]) and (opA[15] <> res[15])
```

4.5 ADJW — Adjust Window Address

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`ADJW`	`10100`					000			`010`			imm5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`ADJW`	`10100`					000			`010`			imm5				

Assembly:

`ADJW imm`

Purpose:

To increase or decrease the address of the register window.

Restrictions:

If `imm` contains a value that is not a multiple of 8, the behavior is **UNPREDICTABLE**. If the long form is used, and `imm5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then imm ← {ext13, imm5[2:0]}
else imm ← sign_extend_16(imm5)
W ← W + imm
```

Remarks:

This instruction may be used in a function prologue or epilogue.

Notice:

The interpretation of the immediate field of this instruction is not final.

4.6 AND — Bitwise AND with Register

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`AND`	`00000`					Rd			Ra			`00`		Rb		

Assembly:

```
AND Rd, Ra, Rb
```

Purpose:

To perform bitwise AND between 16-bit integers in registers.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← opA and opB
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.7 ANDI Bitwise AND with Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`ANDI`	`00001`					Rd			Ra			`00`		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`ANDI`	`00001`					Rd			Ra			`00`		imm3		

Assembly:

```
ANDI Rd, Ra, imm
```

Purpose:

To perform bitwise AND between a 16-bit integer in a register and a constant.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_al(imm3)
res ← opA and opB
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.8 BC — Branch if Carry

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BC`	`1011`				`1010`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BC`	`1011`				`1010`				off8							

Assembly:

```
BC label
```

Purpose:

To transfer control if the last arithmetic operation resulted in unsigned overflow.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (C)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BC1`.

4.9 BC0 Branch if Carry is 0

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BC0`	`1011`				`0010`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BC0`	`1011`				`0010`				off8							

Assembly:

```
BC0 label
```

Purpose:

To transfer control if the carry (C) flag is 0.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not C)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.10 BC1 Branch if Carry is 1

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BC1`	`1011`				`1010`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BC1`	`1011`				`1010`				off8							

Assembly:

```
BC1 label
```

Purpose:

To transfer control if the carry (C) flag is 1.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (C)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.11 BEQ — Branch if Equal

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BEQ`	`1011`				`1000`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BEQ`	`1011`				`1000`				off8							

Assembly:

```
BEQ label
```

Purpose:

To transfer control after a `CMP Ra, Rb` instruction if `Ra` is equal to `Rb`.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (Z)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BZ1`.

4.12 BGES Branch if Greater or Equal, Signed

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BGES`	`1011`				`0101`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BGES`	`1011`				`0101`				off8							

Assembly:

```
BGES label
```

Purpose:

To transfer control after a `CMP Ra, Rb` instruction if `Ra` is greater than or equal to `Rb` when interpreted as signed integers.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not (S xor V))
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.13 BGEU Branch if Greater or Equal, Unsigned

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BGEU`	`1011`				`1010`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BGEU`	`1011`				`1010`				off8							

Assembly:

```
BGEU label
```

Purpose:

To transfer control after a `CMP Ra, Rb` instruction if `Ra` is greater than or equal to `Rb` when interpreted as unsigned integers.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (C)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BC1`.

4.14 BGTS Branch if Greater Than, Signed

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BGTS`	`1011`				`0110`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BGTS`	`1011`				`0110`				off8							

Assembly:

```
BGTS label
```

Purpose:

To transfer control after a `CMP Ra, Rb` instruction if `Ra` is greater than `Rb` when interpreted as signed integers.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not ((S xor V) or Z))
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.15 BGTU Branch if Greater Than, Unsigned

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BGTU`	`1011`				`0100`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BGTU`	`1011`				`0100`				off8							

Assembly:

```
BGTU label
```

Purpose:

To transfer control after a `CMP Ra, Rb` instruction if `Ra` is greater than `Rb` when interpreted as unsigned integers.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not ((not C) or Z))
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.16 BLES Branch if Less or Equal, Signed

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BLES`	`1011`				`1110`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BLES`	`1011`				`1110`				off8							

Assembly:

```
BLES label
```

Purpose:

To transfer control after a CMP `Ra, Rb` instruction if `Ra` is less than or equal to `Rb` when interpreted as signed integers.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if ((S xor V) or Z)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.17 BLEU Branch if Less or Equal, Unsigned

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BLEU`	`1011`				`1100`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BLEU`	`1011`				`1100`				off8							

Assembly:

```
BLEU label
```

Purpose:

To transfer control after a `CMP Ra, Rb` instruction if `Ra` is less than or equal to `Rb` when interpreted as unsigned integers.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if ((not C) or Z)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.18 BLTS — Branch if Less Than, Signed

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BLTS`	`1011`				`1101`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BLTS`	`1011`				`1101`				off8							

Assembly:

```
BLTS label
```

Purpose:

To transfer control after a `CMP Ra, Rb` instruction if `Ra` is less than `Rb` when interpreted as signed integers.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (S xor V)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.19 BLTU — Branch if Less Than, Unsigned

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BLTU`	`1011`				`0010`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BLTU`	`1011`				`0010`				off8							

Assembly:

```
BLTU label
```

Purpose:

To transfer control after a `CMP Ra, Rb` instruction if `Ra` is less than `Rb` when interpreted as unsigned integers.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not C)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BC0`.

4.20 BNC — Branch if Not Carry

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BNC`	`1011`				`0010`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BNC`	`1011`				`0010`				off8							

Assembly:

```
BNC label
```

Purpose:

To transfer control if the last arithmetic operation did not result in unsigned overflow.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not C)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BC0`.

4.21 BNE — Branch if Not Equal

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BNE`	`1011`				`0000`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BNE`	`1011`				`0000`				off8							

Assembly:

```
BNE label
```

Purpose:

To transfer control after a `CMP Ra, Rb` instruction if `Ra` is not equal to `Rb`.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not Z)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BZ0`.

4.22 BNS — Branch if Not Sign

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
BNS	1011				0001				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
EXTI	110			ext13												
BNS	1011				0001				off8							

Assembly:

```
BNS label
```

Purpose:

To transfer control if the last arithmetic operation did not produce a negative result.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not S)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BS0`.

4.23 BNV — Branch if Not Overflow

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BNV`	`1011`				`0011`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BNV`	`1011`				`0011`				off8							

Assembly:

```
BNV label
```

Purpose:

To transfer control if the last arithmetic operation did not result in signed overflow.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not V)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BVO`.

4.24 BNZ — Branch if Not Zero

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
BNZ	1011				0000				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
EXTI	110			ext13												
BNZ	1011				0000				off8							

Assembly:

```
BNZ label
```

Purpose:

To transfer control if the last operation produced a result not equal to zero.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not Z)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BZ0`.

4.25 BS — Branch if Sign

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BS`	`1011`				`1001`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BS`	`1011`				`1001`				off8							

Assembly:

```
BS label
```

Purpose:

To transfer control if the last arithmetic operation produced a negative result.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (S)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BS1`.

4.26 BS0 Branch if Sign is 0

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BS0`	`1011`				`0001`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BS0`	`1011`				`0001`				off8							

Assembly:

```
BS0 label
```

Purpose:

To transfer control if the sign (S) flag is 0.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not S)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.27 BS1 — Branch if Sign is 1

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BS1`	`1011`				`1001`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BS1`	`1011`				`1001`				off8							

Assembly:

```
BS1 label
```

Purpose:

To transfer control if the sign (S) flag is 1.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (S)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.28 BV — Branch if Overflow

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BV`	`1011`				`1011`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BV`	`1011`				`1011`				off8							

Assembly:

```
BV label
```

Purpose:

To transfer control if the last arithmetic operation resulted in signed overflow.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (V)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BV1`.

4.29 BV0 Branch if Overflow is 0

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BV0`	`1011`				`0011`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BV0`	`1011`				`0011`				off8							

Assembly:

```
BV0 label
```

Purpose:

To transfer control if the overflow (V) flag is 0.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not V)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.30 BV1 — Branch if Overflow is 1

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BV1`	`1011`				`1011`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BV1`	`1011`				`1011`				off8							

Assembly:

```
BV1 label
```

Purpose:

To transfer control if the overflow (V) flag is 1.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (V)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.31 BZ — Branch if Zero

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BZ`	`1011`				`1000`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BZ`	`1011`				`1000`				off8							

Assembly:

```
BZ label
```

Purpose:

To transfer control if the last operation produced a result equal to zero.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (Z)
then PC ← PC + 1 + off
else PC ← PC + 1
```

Remarks:

This instruction has the same encoding as `BZ1`.

4.32 BZ0 — Branch if Zero is 0

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BZ0`	`1011`				`0000`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BZ0`	`1011`				`0000`				off8							

Assembly:

```
BZ0 label
```

Purpose:

To transfer control if the zero (Z) flag is 0.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (not Z)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.33 BZ1 Branch if Zero is 1

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`BZ1`	`1011`				`1000`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`BZ1`	`1011`				`1000`				off8							

Assembly:

```
BZ1 label
```

Purpose:

To transfer control if the zero (Z) flag is 1.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
if (Z)
then PC ← PC + 1 + off
else PC ← PC + 1
```

4.34 CMP — Compare to Register

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
CMP	00000					000			Ra			11		Rb		

Assembly:

```
CMP Ra, Rb
```

Purpose:

To compare 16-bit two's complement integers in registers.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← opA + not opB + 1
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = not opB[15]) and (opA[15] <> res[15])
```

Remarks:

This instruction behaves identically to `SUB`, with the exception that it discards the computed value.

4.35 CMPI — Compare to Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`CMPI`	`00001`					000		Ra			`11`		imm3			

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`CMPI`	`00001`					000		Ra			`11`		imm3			

Assembly:

```
CMPI Ra, imm
```

Purpose:

To compare a two's complement constant to a 16-bit two's complement integer in a register.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_al(imm3)
res ← opA + not opB + 1
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = not opB[15]) and (opA[15] <> res[15])
```

Remarks:

This instruction behaves identically to `SUBI`, with the exception that it discards the computed value.

4.36 EXTI — Extend Immediate

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			imm13												

Assembly:

```
EXTI imm
```

Purpose:

To extend the range of immediate in the following instruction.

Restrictions:

None.

Operation:

```
ext13 ← imm13
has_ext13 ← 1
```

Remarks:

This instruction is automatically emitted by the assembler while translating other instructions. As it changes both the meaning of and the constraints placed on the immediate field in the following instruction, placing it manually may lead to unexpected results.

4.37 J **Jump**

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`J`	`1011`				`1111`				off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`J`	`1011`				`1111`				off8							

Assembly:

```
J label
```

Purpose:

To unconditionally transfer control.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
PC ← PC + 1 + off
```

4.38 JAL — Jump and Link

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`JAL`	`10101`					Rd			off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`JAL`	`10101`					Rd			off8							

Assembly:

```
JAL Rd, label
```

Purpose:

To transfer control to a subroutine.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
mem[W+Rd] ← PC + 1
PC ← PC + 1 + off
```

4.39 JR Jump to Register

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`JR`	`10100`					Rs			`100`			off5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`JR`	`10100`					Rs			`100`			off5				

Assembly:

```
JR Rs, off
```

Purpose:

To transfer control to a variable absolute address contained in a register, with a constant offset.

Restrictions:

If the long form is used, and `off5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off5[2:0]}
else off ← sign_extend_16(off5)
PC ← mem[W+Ra] + off
```

4.40 JRAL — Jump to Register and Link

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`JRAL`	`10100`					Rd			`101`			00		Rb		

Assembly:

```
JRAL Rd, Rb
```

Purpose:

To transfer control to a subroutine whose variable absolute address is contained in a register.

Restrictions:

None.

Operation:

```
addr ← mem[W+Rb]
mem[W+Rd] ← PC + 1
PC ← addr
```

4.41 JST Jump through Switch Table

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`JST`	`10100`					Rs			`111`			off5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`JST`	`10100`					Rs			`111`			off5				

Assembly:

```
JST Rs, off
```

Purpose:

To transfer control to an address contained in a jump table at a variable offset, where the address is relative to the location of the table.

Restrictions:

If the long form is used, and `off5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off5[2:0]}
else off ← sign_extend_16(off5)
table ← PC + 1 + off
entry ← mem[W+Rs]
addr ← mem[table + entry]
PC ← table + addr
```

4.42 JVT — Jump through Virtual Table

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`JVT`	`10100`					Rs			`110`			off5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`JVT`	`10100`					Rs			`110`			off5				

Assembly:

```
JVT Rs, off
```

Purpose:

To transfer control to an address contained in a jump table at a constant offset, where the address is relative to the location of the table.

Restrictions:

If the long form is used, and `off5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off5[2:0]}
else off ← sign_extend_16(off5)
table ← mem[W+Rs]
addr ← mem[table + off]
PC ← table + addr
```

4.43 LD — Load

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`LD`	`01000`					Rd			Ra			off5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`LD`	`01000`					Rd			Ra			off5				

Assembly:

```
LD Rd, Ra, off
```

Purpose:

To load a word from memory at a variable address, with a constant offset.

Restrictions:

If the long form is used, and `off5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off5[2:0]}
else off ← sign_extend_16(off5)
addr ← mem[W+Ra] + off
data ← mem[addr]
mem[W+Rd] ← data
```

4.44 LDR Load PC-relative

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`LDR`	`01001`					Rd			Ra			off5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`LDR`	`01001`					Rd			Ra			off5				

Assembly:

```
LDR Rd, Ra, off
```

Purpose:

To load a word from memory at a constant PC-relative address, with a variable offset.

Restrictions:

If the long form is used, and `off5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off5[2:0]}
else off ← sign_extend_16(off5)
addr ← PC + 1 + off + mem[W+Ra]
data ← mem[addr]
mem[W+Rd] ← data
```

4.45 LDW Adjust and Load Window Address

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`LDW`	`10100`					Rd			`011`			imm5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`LDW`	`10100`					Rd			`011`			imm5				

Assembly:

```
LDW Rd, imm
```

Purpose:

To increase or decrease the address of the register window, and retrieve the prior address of the register window.

Restrictions:

If `imm` contains a value that is not a multiple of 8, the behavior is **UNPREDICTABLE**. If the long form is used, and `imm5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then imm ← {ext13, imm5[2:0]}
else imm ← sign_extend_16(imm5)
old ← W
W ← W + imm
mem[W+Rd] ← old
```

Remarks:

See also `STW`. This instruction may be used in a function prologue, where `Rd` is any register chosen to act as a frame pointer.

Notice:

The interpretation of the immediate field of this instruction is not final.

4.46 LDX — Load External

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`LDX`	`01100`					Rd			Ra			off5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`LDX`	`01100`					Rd			Ra			off5				

Assembly:

```
LDX Rd, Ra, off
```

Purpose:

To complete a load cycle on external bus at a variable address, with a constant offset.

Restrictions:

If the long form is used, and `off5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off5[2:0]}
else off ← sign_extend_16(off5)
addr ← mem[W+Ra] + off
data ← ext[addr]
mem[W+Rd] ← data
```

4.47 LDXA Load External Absolute

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`LDXA`	`01101`					Rd			off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`LDXA`	`01101`					Rd			off8							

Assembly:

```
LDXA Rd, off
```

Purpose:

To complete a load cycle on external bus at a constant absolute address.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
data ← ext[off]
mem[W+Rd] ← data
```

4.48 MOV **Move**

Assembly:

```
MOV Rd, Rs
```

Purpose:

To move a value from register to register.

Restrictions:

None.

Remarks:

The assembler does not translate any instructions for `MOV` with identical `Rd` and `Rs`, and translates `MOV` with any other register combination to

```
    AND  Rd, Rs, Rs
```

4.49 MOVI Move Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`MOVI`	`10000`					Rd			imm8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`MOVI`	`10000`					Rd			imm8							

Assembly:

```
MOVI Rd, imm
```

Purpose:

To load a register with a constant.

Restrictions:

If the long form is used, and `imm8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then imm ← {ext13, imm8[2:0]}
else imm ← sign_extend_16(imm8)
mem[W+Rd] ← imm
```

4.50 MOVR — Move PC-relative Address

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`MOVR`	`10001`					Rd			off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`MOVR`	`10001`					Rd			off8							

Assembly:

```
MOVR Rd, off
```

Purpose:

To load a register with an address relative to PC with a constant offset.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
mem[W+Rd] ← PC + 1 + off
```

4.51 NOP — No Operation

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`NOP`	`1011`				`0111`				`00000000`							

Assembly:

```
NOP
```

Purpose:

To perform no operation while consuming one instruction word.

Operation:

`PC` ← `PC + 1`

Remarks:

The `NOP` instruction is encoded as a conditional branch, whose condition is always false, that targets the next instruction.

4.52 NOT Logical NOT

Assembly:

```
NOT Rd, Rs
```

Purpose:

To invert all the bits in a register.

Restrictions:

None.

Remarks:

The translates to a XORI with an 0xFFFF immediate

```
    XORI Rd, Rs, 0xFFFF
```

4.53 OR Bitwise OR with Register

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
OR	00000					Rd			Ra			01		Rb		

Assembly:

```
OR Rd, Ra, Rb
```

Purpose:

To perform bitwise OR between 16-bit integers in registers.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← opA or opB
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.54 ORI — Bitwise OR with Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`ORI`	`00001`					Rd			Ra			`01`		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`ORI`	`00001`					Rd			Ra			`01`		imm3		

Assembly:

```
ORI Rd, Ra, imm
```

Purpose:

To perform bitwise OR between a 16-bit integer in a register and a constant.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_al(imm3)
res ← opA or opB
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.55 ROL — Rotate Left

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`ROL`	`00100`					Rd			Ra			`01`		Rb		

Assembly:

```
ROL Rd, Ra, Rb
```

Purpose:

To perform a left rotate of a 16-bit integer in a register by a variable bit amount.

Restrictions:

If `Rb` contains a value greater than 15, the behavior is **UNPREDICTABLE**.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← {opA[15-opB:0], opA[15:16-opB]}
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.56 ROLI Rotate Left Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
ROLI	00101					Rd			Ra			01		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
EXTI	110			ext13												
ROLI	00101					Rd			Ra			01		imm3		

Assembly:

```
ROLI Rd, Ra, amount
```

Purpose:

To perform a left rotate of a 16-bit integer in a register by a constant bit amount.

Restrictions:

If `amount` is greater than 15, the behavior is **UNPREDICTABLE**.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_sr(imm3)
res ← {opA[15-opB:0], opA[15:16-opB]}
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.57 RORI — Rotate Right Immediate

Assembly:

```
RORI Rd, Ra, amount
```

Purpose:

To perform a right rotate of a 16-bit integer in a register by a constant bit amount.

Restrictions:

If `amount` is greater than 15, the behavior is **UNPREDICTABLE**.

Remarks:

The assembler translates `RORI` with `amount` of 0 to

```
ROLI Rd, Ra, 0
```

and `RORI` with any other `amount` to

```
ROLI Rd, Ra, (16 - amount)
```

4.58 SBC Subtract Register with Carry

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
SBC	00010					Rd			Ra			11		Rb		

Assembly:

```
SBC Rd, Ra, Rb
```

Purpose:

To subtract 16-bit two's complement integers in registers, with carry input. If the carry input is 1, the previous operation did not borrow.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← opA + not opB + C
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = not opB[15]) and (opA[15] <> res[15])
```

Remarks:

A 32-bit subtraction with both operands in registers can be performed as follows:

```
; Perform {R1, R0} ← {R3, R2} - {R5, R4}
    SUB  R0, R2, R4
    SBC  R1, R3, R5
```

4.59 SBCI Subtract Immediate with Carry

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`SBCI`	`00011`					Rd			Ra			`11`		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`SBCI`	`00011`					Rd			Ra			`11`		imm3		

Assembly:

```
SBCI Rd, Ra, imm
```

Purpose:

To subtract a two's complement constant from a 16-bit two's complement integer in a register, with carry input. If the carry input is 1, the previous operation did not borrow.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_al(imm3)
res ← opA + not opB + C
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = not opB[15]) and (opA[15] <> res[15])
```

Remarks:

A 32-bit subtraction with a register and an immediate operand can be performed as follows:

```
; Perform {R1, R0} ← {R3, R2} - 0x40001
    SUBI R0, R2, 1
    SBCI R1, R3, 4
```

4.60 SLL — Shift Left Logical

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
SLL	00100					Rd			Ra			00		Rb		

Assembly:

```
SLL Rd, Ra, Rb
```

Purpose:

To perform a left logical shift of a 16-bit integer in a register by a variable bit amount.

Restrictions:

If `Rb` contains a value greater than 15, the behavior is **UNPREDICTABLE**.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← {opA[15-opB:0], opB{0}}
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.61 SLLI Shift Left Logical Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
SLLI	00101					Rd			Ra			00		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
EXTI	110			ext13												
SLLI	00101					Rd			Ra			00		imm3		

Assembly:

```
SLLI Rd, Ra, amount
```

Purpose:

To perform a left logical shift of a 16-bit integer in a register by a constant bit amount.

Restrictions:

If `amount` is greater than 15, the behavior is **UNPREDICTABLE**.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_sr(imm3)
res ← {opA[15-opB:0], opB{0}}
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.62 SRA — Shift Right Arithmetical

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`SRA`	`00100`					Rd			Ra			`11`		Rb		

Assembly:

```
SRA Rd, Ra, Rb
```

Purpose:

To perform a right arithmetical shift of a 16-bit integer in a register by a variable bit amount.

Restrictions:

If `Rb` contains a value greater than 15, the behavior is **UNPREDICTABLE**.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← {opB{opA[15]}, opA[15:opB]}
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.63 SRAI Shift Right Arithmetical Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
SRAI	00101					Rd			Ra			11		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
EXTI	110			ext13												
SRAI	00101					Rd			Ra			11		imm3		

Assembly:

```
SRAI Rd, Ra, amount
```

Purpose:

To perform a right arithmetical shift of a 16-bit integer in a register by a constant bit amount.

Restrictions:

If `amount` is greater than 15, the behavior is **UNPREDICTABLE**.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_sr(imm3)
res ← {opB{opA[15]}, opA[15:opB]}
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.64 SRL — Shift Right Logical

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
SRL	00100					Rd			Ra			10		Rb		

Assembly:

```
SRL Rd, Ra, Rb
```

Purpose:

To perform a right logical shift of a 16-bit integer in a register by a variable bit amount.

Restrictions:

If `Rb` contains a value greater than 15, the behavior is **UNPREDICTABLE**.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← {opB{0}, opA[15:opB]}
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.65 SRLI Shift Right Logical Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
SRLI	00101					Rd			Ra			10		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
EXTI	110			ext13												
SRLI	00101					Rd			Ra			10		imm3		

Assembly:

```
SRLI Rd, Ra, amount
```

Purpose:

To perform a right logical shift of a 16-bit integer in a register by a constant bit amount.

Restrictions:

If `amount` is greater than 15, the behavior is **UNPREDICTABLE**.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_sr(imm3)
res ← {opB{opA[15]}, opA[15:opB]}
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.66 ST — Store

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`ST`	`01010`					Rs			Ra			off5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`ST`	`01010`					Rs			Ra			off5				

Assembly:

```
ST Rs, Ra, off
```

Purpose:

To store a word to memory at a variable address, with a constant offset.

Restrictions:

If the long form is used, and `off5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off5[2:0]}
else off ← sign_extend_16(off5)
addr ← mem[W+Ra] + off
data ← mem[W+Rs]
mem[addr] ← data
```

4.67 STR Store PC-relative

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`STR`	`01011`					Rs			Ra			off5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`STR`	`01011`					Rs			Ra			off5				

Assembly:

```
STR Rs, Ra, off
```

Purpose:

To store a word to memory at a constant PC-relative address, with a variable offset.

Restrictions:

If the long form is used, and `off5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off5[2:0]}
else off ← sign_extend_16(off5)
addr ← PC + 1 + off + mem[W+Ra]
data ← mem[W+Rs]
mem[addr] ← data
```

4.68 STW — Store to Window Address

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`STW`	`10100`					000			`000`			00		Rb		

Assembly:

`STW Rb`

Purpose:

To arbitrarily change the address of the register window.

Restrictions:

If `Rb` contains a value that is not a multiple of 8, the behavior is **UNPREDICTABLE**.

Operation:

`W` $\leftarrow$ `mem[W+Rb]`

Remarks:

See also `LDW`. This instruction may be used in a function epilogue, where `Rb` is any register chosen to act as a frame pointer.

4.69 STX — Store External

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`STX`	`01110`					Rs			Ra			off5				

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`STX`	`01110`					Rs			Ra			off5				

Assembly:

```
STX Rs, Ra, off
```

Purpose:

To complete a store cycle on external bus at a variable address, with a constant offset.

Restrictions:

If the long form is used, and `off5[4:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off5[2:0]}
else off ← sign_extend_16(off5)
addr ← mem[W+Ra] + off
data ← mem[W+Rs]
ext[addr] ← data
```

4.70 STXA Store External Absolute

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`STXA`	`01111`					Rs			off8							

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`STXA`	`01111`					Rs			off8							

Assembly:

```
STXA Rs, off
```

Purpose:

To complete a store cycle on external bus at a constant absolute address.

Restrictions:

If the long form is used, and `off8[7:3]` are non-zero, the behavior is **UNPREDICTABLE**.

Operation:

```
if (has_ext13)
then off ← {ext13, off8[2:0]}
else off ← sign_extend_16(off8)
data ← mem[W+Rs]
ext[off] ← data
```

4.71 SUB — Subtract Register

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
SUB	00010					Rd			Ra			10		Rb		

Assembly:

```
SUB Rd, Ra, Rb
```

Purpose:

To subtract 16-bit two's complement integers in registers.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← opA + not opB + 1
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = not opB[15]) and (opA[15] <> res[15])
```

4.72 SUBI — Subtract Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
SUBI	00011					Rd			Ra			10		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
EXTI	110			ext13												
SUBI	00011					Rd			Ra			10		imm3		

Assembly:

```
SUBI Rd, Ra, imm
```

Purpose:

To subtract a two's complement constant from a 16-bit two's complement integer in a register.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_al(imm3)
res ← opA + not opB + 1
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← res[16]
V ← (opA[15] = not opB[15]) and (opA[15] <> res[15])
```

4.73 XCHG Exchange Registers

Assembly:

`XCHG Ra, Rb`

Purpose:

To exchange the values of two registers.

Restrictions:

None.

Remarks:

The assembler does not translate any instructions for `XCHG` with identical `Ra` and `Rb`, and translates `XCHG` with any other register combination to

```
XOR  Ra, Ra, Rb
XOR  Rb, Rb, Ra
XOR  Ra, Ra, Rb
```

4.74 XCHW — Exchange Window Address

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`XCHW`	`10100`					Rd			`001`			00		Rb		

Assembly:

```
XCHW Rd, Rb
```

Purpose:

To exchange the address of the register window with a general purpose register.

Restrictions:

If `Rb` contains a value that is not a multiple of 8, the behavior is **UNPREDICTABLE**.

Operation:

```
old ← W
W ← mem[W+Rb]
mem[W+Rd] ← old
```

Remarks:

This instruction may be used in a context switch routine. For example, if multiple register windows are set up such that each contains the address of the next one in `R7`, the following code may be used to switch contexts:

```
yield:
    XCHW R7, R7
    JR   R0
; Elsewhere:
    JAL R0, yield
```

4.75 XOR — Bitwise XOR with Register

Encoding:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`XOR`	`00000`					Rd			Ra			`10`		Rb		

Assembly:

```
XOR Rd, Ra, Rb
```

Purpose:

To perform bitwise XOR between 16-bit integers in registers.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
opB ← mem[W+Rb]
res ← opA xor opB
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

4.76 XORI Bitwise XOR with Immediate

Encoding (short form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`XORI`	`00001`					Rd			Ra			`10`		imm3		

Encoding (long form):

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
`EXTI`	`110`			ext13												
`XORI`	`00001`					Rd			Ra			`10`		imm3		

Assembly:

```
XORI Rd, Ra, imm
```

Purpose:

To perform bitwise XOR between a 16-bit integer in a register and a constant.

Restrictions:

None.

Operation:

```
opA ← mem[W+Ra]
if (has_ext13)
then opB ← {ext13, imm3}
else opB ← decode_imm_al(imm3)
res ← opA xor opB
mem[W+Rd] ← res
Z ← res[15:0] = 0
S ← res[15]
C ← UNDEFINED
V ← UNDEFINED
```

5 List of Assembly Directives

TBD

6 Function Calling Sequence

TBD