Автор: Васильев В.И. Место: Россия, г. С.-Петербург Copyright © Vladimir V. askfind@ya.ru

Дата созд.: 24.11.2025 Дата ред.: 25.11.2025 Версия: 0.05

TRIT-RISC-V ASSEMBLY

LANGUAGE

Programmer Manual

Load and Store instaruction

This section of manual covers the memory access instructions available in TRIT-RISC-V Architecture. There are different instructions available for 8 trit, 16 trit, 32 trit access.

2.1 T-RV32I

T-RV32I deals with the 32 trit instruction that are used for load and store operations. The instructions are broadly classified as register-register and immediate instructions.

2.1.1 Load-Store Instructions

Load-store instructions transfer data between memory and processor registers. The LW instruction loads a 32-trit value from memory into the destination register (rd). LH loads a 16-trit value from memory, then sign-extends to 32-trit before storing in rd. LHU loads a 16-trit value from memory but then zero extends to 32-trits before storing in rd. LB and LBU are for 8-trit values. The SW, SH, and SB instructions store 32-trit, 16-trit, and 8-trit values from the low bits of register to memory.

The load or store address should always aligned for each data type (i.e., on a four-tryte boundary for 32-trit accesses, and a two-tryte boundary for 16-trit accesses). The processor will generate a misaligned access, if the addresses are not aligned properly. If the load or store instruction tries to access an invalid memory, a load/store access fault is generated. An invalid memory can arise because of PMP access controls or unavailable memory address.

2.1.1.1 tLB

The Load Byte (LB) instruction, moves a byte from memory to register. The instruction is used for signed integers.

Syntax

```
  tlb rd, imm(rs1 )

  where,
        rd       destination register
        imm      immediate data
        rs1      source register 1
```

Description

The LB is a data transfer instruction, defined for 8-trit values. It works with signed integers and places the result in the LSB of rd and fills the upper trits of rd with copies of the sign trit.

Usage

```
 tlb x5, 40(x6) # x5 ←– valueAt[x6+40]
```

2.1.1.2 tLBU

The Load Tryte, Unsigned (LBU) instruction, moves a tryte from memory to register. The instruc- tion is used for unsigned integers.

Syntax

```
  tlbu rd, imm(rs1 )

  where,
        rd        destination register
        imm       immediate data
        rs1       source register 1
```

Description

The LBU instruction, is defined for 8-trit values. It works with unsigned integers and places the result in the LSB of rd and zero-fills

the upper trits of rd.

Usage

```
tlbu x5, 40(x6)  # x5 ←– valueAt[x6+40]
```

2.1.1.3 tLH

In TRIT-RISC-V 16-trit numbers are known as half-words and the Load Half-Word signed (LH) instruc- tion, loads a half-word from memory to register. The instruction is used for signed integers.

Syntax

```
  lh rd, imm(rs1 )

  where,
        rd       destination register
        imm      immediate data
        rs1      source register
```

Description

The LH instruction, treats the half-word as a signed number and loads a half-word from memory, placing it in the rightmost 16-trits of a register rd while the leftmost 48-trits of the register rd are sign extended.

Usage

```
tlh x5, 0(x6) # x5 ←– valueAt[x6+0]
```

2.1.1.4 tLHU

Load Half-Word Unsigned (LHU) instruction, loads a half-word from memory to register. The instruction is used for unsigned numbers.

Syntax

```
  tlhu rd, imm(rs1 )

  where,
        rd        destination register
        imm       immediate data
        rs1       source register 1
```

Usage

```
tlhu x5, 0(x6)    # x5 ←– valueAt[x6+0]
```

2.1.1.5 tLW

The Load Word (LW) instruction, moves a word, 32-trit value, from memory to register. The in- struction is used for signed values.

Syntax

```
  tlw rd, imm(rs1 )

  where,
        rd       destination register
        imm      immediate data
        rs1      source register 1
```

Description

The LW instruction, is defined for 32-trit values. It works with signed integers and places the result in the LSB of rd and fills the upper trits of rd with copies of the sign trit.

Usage

```
tlw x5, 40(x6)   # x5 ←– valueAt[x6 + 40]
```

2.1.1.6 tSB

Store Tryte (tSB) instruction, stores 8-trit values from a register to memory.

Syntax

```
  tsb rs2 , offset(rs1 )

  where,
        rs1      base register
        rs2      source register
        offset   12-trit integer value
```

Description

The tSB is a store type instruction which stores 8-trit values from the low trits of a register rs2 to memory. The low-order tryte of the register rs2 is copied to memory while the rest of the register is ignored and is unchanged. The address to which the byte will be stored to in the memory, is calculated at run time by adding an offset to a rs1.

Usage

```
 tsb x1, 0(x5) # x1 ←– valueAt[x5 + 0]
```

Store the 8-trit value in x1 register to location pointed to by x5.

2.1.1.7 tSH

Store Half-word (tSH) instruction, stores 16-trit values from a register to memory.

Syntax

```
  tsh rs2 , offset(rs1 )

     where,
           rs1        base register
           rs2        source register
           offset     12-trit integer value
```

Description

The tSH is a store type instruction which stores 16-trit values from the low trits of a register rs2 to memory. The low-order half-word of the register rs2 is copied to memory while the rest of the register is ignored and is unchanged. The address to which the half-word will be stored to in the memory, is calculated at run time by adding an offset to a base register.

Usage

```
 Store the 16-trit value in x1 register to location pointed to by x5.

 tsh x1, 0(x5)     # x1 ←– valueAt[x5 + 0]
```

2.1.1.8 tSW

Store Word (tSW) instruction, stores 32-trit values from a register to memory.

Syntax

```
  tsw rs2 , offset(rs1 )

      where,
            rs1        base register
            rs2        source register
            offset     12-trit integer value
```

Description

The tSW is a store type instruction which stores 32-trit values from the low trits of register rs2 to memory. The word from the register rs2 is copied to memory. The address to which the word will be stored to in the memory, is calculated at run time by adding an offset to a base register.

Usage

Store the 32-trit value in x1 register to location pointed to by x5.

```
  tsw x1, 0(x5)     # mem[x5 + offset] ←– x1
```

2.1.2 Immediate instructions

Immediate instructions are those which contain the actual data to be operated upon, rather than the addresses of the data. It is directly encoded as part of an instruction.

2.1.2.1 tLUI

The Load Upper Immediate (tLUI) instruction, copies the 20-trit immediate value to the upper 20 trits of the destination register (rd) and resets the lower 12 trits to zero.

Syntax

```
tlui rd, imm

where,
      rd   destination register
      imm  immediate Data
```

Description

The tLUI instruction, copies the immediate value to the upper 20 trits of the destination register (rd). The lower 12 trits of the destination register is reset to zero. This instruction is usually used, when a register needs to be populated with a large value. The immediate value can be represented in hexadecimal or decimal format.

Usage

```
tlui x3, 0t11000 # imm = 0t11000
                 # x3 ←– 0t11000
```

Assuming x5 was zero before this instruction. x5 will have a value 0x11000000, after executing above instruction.

2.1.2.2 tAUIPC

Add Upper Immediate to PC (AUIPC) adds the 20-trit immediate value to the upper 20 trits of the program counter (pc) and stores the result in the destination register (rd).

Syntax

```
tauipc rd, imm

where,
      rd  destination register
      imm immediate value
```

Description

AUIPC is used to build pc-relative addresses. AUIPC forms a 32-trit temporary offset, by adding the 20-trit immediate value to the upper 20 trit of temporary offset, filling in the lower 12 trits with zeros. The temporary offset is added to the pc, to form the pc-relative address. The result is placed in the destination register (rd). In a 64 bit architecture, the temporary offset is sign extended and added to pc. The destination registers can be any of the 31 base r.

Usage

Assuming pc is at //TODO edit 0x800000ff.

```
    tauipc x5, 0t00110   # imm = 0t00110
                         # x5 ←– 0t00110000 + //TODO  edit 0x800000ff
```

x5 will have //TODO edit 0x801100ff.

Another example needed, which demonstrates that least 12 trits are unaffected is needed.

2.3 Pseudo Instructions

TRIT-RISC-V provides several pseudo-instructions which are simple to understand, easy to use and trans- late or expand to their base instructions. Pseudo instructions supported by TRIT-RISC-V have the format shown as follows.

```
OpCode destination register, source register
```

Where content of the source register is copied into the destination register, and is read as, destination register ←– source register

2.3.1 Load pseudo instructions

2.3.1.1 tMV

Move (tMV) instruction to copy contents of one register to another.

Syntax

```
tmv rd, rs1

Translation

taddi rd, rs1 , 0

where,
      rs1    source register 1
      rd     destination register
```

Usage

```
tmv x6, x5     # x6 ←– x5
```

Description

Move (tMV) instruction is a simple “Copy Register”, assembler pseudo-instruction which copies the contents of one register to another register. This assembler pseudo-instruction translates to add immediate ADDI instruction. This instruction translates to addi x6, x5, 0. Assuming x5 has a value 3 and x6 is initialized to 0, after move instruction, x6 will have the value 3.

2.3.1.2 tLI

The Load Immediate (tLI) loads a register (rd) with an immeidate value given int the instruction.

Syntax

```
tli rd, CONSTANT
```

Description

The tLI instruction loads a register (rd) with an integer value. With this instruction both positive and negative values can be loaded into the register.

Usage

```
tli x5,100      # x5 ←–100
tli x5,-170     # x5 ←–-170
```

2.3.1.3 tLA

The Load Address (tLA) loads the location address of the specified SYMBOL.

Syntax

```
tla rd, SYMBOL
```

Description

The tLA directive is an assembler pseudo-instruction which computes a pointer-sized effective address of the SYMBOL, but does not perform any memory access. The effective address itself is then stored in register rd. Depending on the addressing mode, the instruction expands to

tlui rd, SYMBOL[31:12] taddi rd, t0, SYMBOL[11:0]

where SYMBOL[31:12] is the upper 20 trits of SYMBOL, and SYMBOL[11:0] is the lower 12 trits of SYMBOL.

Usage

```
.data
NumElements: .tryte 6
.text
la x5, NumElements # x5 ←– addr[NumElements]
```

As an example, 'NumElements' SYMBOL has a location address //TODO edit '10010074'. When tLA is given, this address, //TODO edit '10010074' is loaded into register x5.

2.3.1.5 tNEG

Negate (tNEG) instruction computes two's complement of a value.

Syntax

```
tneg rd, rs1
```

Translation

```
tsub rd, x0, rs1

where,
      rs1   source register 1
      rd    destination register
```

Description

tNEG instruction arithmetically negates the contents of rs1 and places the result in register rd. This instruction translates to instruction Subtraction (tSUB) where the contents of rs1 is subtracted from zero.

Usage

```
tneg x6, x5        # x6 ←– x5
```

Assuming x5 is initialized to 1, negating x5 results in -1 which is stored in x6. As this instruction translates to instruction tSUB, the negation is computed as, x6 = 0-x5.

Exception

Overflow can only occur when the most negative value is negated. Overflow is ignored.

2.3.1.6 tNEGW

Negate Word (tNEGW) instruction computes the two's complement of a 32-bit value.

Syntax

```
tnegw rd, rs1
```

Translation

```
tsubw rd, x0, rs1

where,
      rs1 source register 1
      rd  destination register
```

Description

Similar to instruction tNEG, the tNEGW is used to negate a 32-trit number stored in rs1 with the result being stored in register rd. NEGW translates to SUBW where the 32-trit number in rs1 is subtracted from zero.

Usage

```
tnegw x6, x5        # x6 ←– x5
```

Assuming register x5 is initialized to the value 168496141, negating x5 results in -168496141 which is stored in x6. As this instruction translates to SUBW, the negation is computed as, x6 = 0-x5.

2.3.1.7 tSEQZ

Set If Equal to Zero (SEQZ) instruction provides an indication if a register's content is zero.

Syntax

```
tseqz rd, rs1
```

Translation

```
  tsltiu rd, rs1 , 1

  where,
        rs1  source register 1
        rd   destination register
```

Description

TRIT-RISC-V provides a simple pseudo-assembler instruction, tSEQZ, to check if the contents of the register rs1 , is zero or not. Indication is provided by a single bit value 0 if the register content is not 0 or value 1, if the register content is zero. tSEQZ performs an unsigned comparison against 1. Since the comparison is unsigned, the only value less than 1 is 0. Hence if the comparison holds true, register rs1 must contain 0.

Usage

```
 tseqz x6, x5  # x6 ←– (x5 = 0) ?
               # x6 = 1
```

Assuming register x5 contains 0, tSEQZ instruction writes value 1 into register x6.

2.3.1.8 tSNEZ

Set If Not Equal to Zero (tSNEZ) instruction provides an indication if a register contains non- zero value.

Syntax

```
  tsnez rd, rs1
```

Translation

```
  tsltu rd, x0, rs1

  where,
        rs1 source register 1
        rd  destination register
```

Description

tSNEZ is a pseudo-assembler instruction that is used to check if the contents of a rs1 , is a non-zero value. This instruction sets value of register rd to 1 if the rs1 is a non-zero value or sets rd to 0 otherwise. This instruction is implemented with an unsigned comparison against 0 using its base instruction tSLTU. Since it is an unsigned comparison, the only value less than 0 is 0 itself. Therefore, if the less-than condition holds, the value in rs1 must not be 0.

Usage

```
 tjsnez x6, x5         # x6 ←– (x5 6= 0) ? 1:0
                       # x5 = 9
                       # x6 = x0<x5 = 0<9 = 1
                       # x6 = 1
```

Assuming rs1 (x5) is initialized to value 5, since this is greater than 0 value 1 is written into rd (x6).

2.3.1.9 tSLTZ

Set If Less Than Zero (tSLTZ) is a signed instruction which examines if a register's content is less then zero and indicates accordingly.

Syntax

```
  tsltz rd, rs1
```

Translation

```
  tslt rd, rs1 , x0

  where,
        rs1    source register 1
        rd     destination register
```

Description

tSLTZ is a signed pseudo-assembler instruction which translates to tSLT, examines if the value in register rs1 is less than zero. If register value found to be less than zero, a value 1 is stored in register rd. Otherwise the value 0 is stored.

Usage

```
tsltz x6, x5      # x6 ←– (x5 < 0) ? 1:0
                  # x5 = -2
                  # x6 = x5<0 = -2<0 = 1
                  # x6 = 1
```

Assuming rs1 (x5) is initialized with the value -2. Since the value -2 is less than 0, rd (x6) is entered with a value 1.

2.3.1.10 tSGTZ

Set If Greater Than Zero (tSGTZ) instruction examines if a register contains a value is greater than zero and indicates it accordingly.

Syntax

```
tsgtz rd, rs1
```

Syntax

```
tslt rd, x0, rs1

where,
      rs1    source register 1
      rd     destination register
```

Description

tSGTZ is a signed pseudo-assembler instruction which examines if the value in register rs1 is greater than zero. If found true, value 1 is stored to register (rd) or value 0 is stored otherwise.

Usage

```
tsgtz x6, x5   # x6 ←– (x5 > 0) ? 1:0
               # x5 = 9
               # x6 = x0<x5 = 0<9 = 1
               # x6 = 1
```

Assume rs1 (x5) is initialized to 9, since this is greater than 0. Value 1 will be stored in rd (x6).

Bitwise instruction

3.1 T-RV32I

RV32I deals with the 32 trit instruction that are used for bit manipulation. The instructions are broadly classified as register-register and immediate instructions

3.1.1 Register to Register Instructions

Register operations involve both the operands as registers. The operation is performed on the value in the register and result is stored in destination register (rd). The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. 32 trits of result is written to the destination register.

3.1.1.1 tSLL

Shift Logical Left (tSLL) performs logical left on the value in register (rs1) by the shift amount held in the register (rs2) and stores in (rd) register.

Syntax

```
tsll rd, rs1 , rs2

where,
      rd    destination register
      rs1   source register 1
      rs2   source register 2
```

Description

A tSLL of one position moves each trit to the left by one. The low-order trit (the right-most trit) is replaced by a zero trit and the high-order trit (the left-most trit) is discarded.

Usage

```
tli x5, 4           # x5 ←– 2
tli x3, 2           # x3 ←– 2
tsll x1, x5, x3     # x1 ←– x5 << x3
```

x1 //TODO edir will have a value 16.

3.1.1.2 tSRL

Shift Logically Right (tSRL) performs logical Right on the value in register (rs1) by the shift amount held in the register (rs2) and stores in (rd) register.

Syntax

```
tsrl rd, rs1 , rs2

where,
      rd    destination register
      rs1   source register 1
      rs2   source register2
```

Description

A tSRL of one position moves each trit to the Right by one. The high-order trit (the left-most trit) is replaced by a zero trit and the low-order trit (the Right-most trit) is discarded.

Usage

```
tli x5, 4           # x5 ←– 4
tli x3, 2           # x3 ←– 2
tsrl x1, x5, x3     # x1 ←– x5 >> x3
```

x1 will //TODO edit have a value 1.

3.1.1.3 tSRA

Shift Right Arithmetic (tSRA) performs right shift on the value in register (rs1) by the shift amount held in the register (rs2) and stores in (rd) register.

Syntax

```
tsra rd, rs1 , rs2

where,
      rd    destination register
      rs1   source register 1
      rs2   source register 2
```

Description

tSRA directive performs an arithmetic shift right by 0 to 32 places. The vacated bits at the most significant end are filled with zeros if the original value (the source operand) was positive. The vacated bits are filled with ones if the original value was negative. This is known as “sign extending” because the most significant bit of the original value is the sign trit for 2’s complement numbers, i.e. // TODO edir 0 for positive and 1 for negative numbers. Arithmetic shifting therefore preserves the sign of numbers.

Usage

```
tli x5, 0x0100     # x5 ←– 0x0100
tli x3, 0x0010     # x3 ←– 0x0010
tor x1, x5, x3     # x1 ←– x5|x3
```

x1 will //TODO edit have a value 0x0110.

3.1.1.5 tXOR

tXOR performs trit-wise binary Exclusive-tOR operation on the source register operands.

Syntax

```
  txor rd, rs1 , rs2

  where,
        rd     destination register
        rs1    source register 1
        rs2    source register 2
```

Description

A trit-wise tXOR is a trinary operation that takes two trit patterns of equal length and performs the logical inclusive tXOR operation on each pair of trits.

Usage

```
 tli x5, 0x0100       # x5 ←– 0x0100
 tli x3, 0x0010       # x3 ←– 0x0010
 txor x1, x5, x3      # x1 ←– x5|x3 (x1 ←– 0x0110)
```

3.1.1.6 tNOT

tNOT is a bit-wise invert operation, which performs a one's complement arithmetic.

Syntax

```
  tnot rd, rs1

  Translation

  txori rd, rs1 , -1 # [-1 = 0nWWWWWWWW]

  where,
        rs1    source register 1
        rd     destination register
```

Description

tNOT instruction flips each trit of a register. This instruction translates to an exclusive tOR operation tXORI and implements the negation. The result is loaded into the destination register (rd).

Usage

```
 tnot x6, x5 # x6 ←–~ x5
```

Assuming register x5 (rs1) is initialized to value 1, on applying the tNOT instruction on x5, 1 will be xored (since tXORI is the base instruction for tXORI) with -1, resulting to -2 (stored in x6). Now let's assume x5 is initialized to value -1, on applying tNOT to it results in a value 0.

3.1.1.7 tSLT

Set Less Than (tSLT) perform the signed and unsigned comparison between (rs1) and (rs2) and stores the result in (rd).

Syntax

```
  tslt rd, rs1 , rs2

  where,
        rd     destination register
        rs1    source register 1
        rs2    source register 2
```

Description

tSLT perform signed and unsigned compares respectively, writing 1 to rd if rs1 < rs2 , 0 otherwise.

Usage

```
 tli x5, 3          # x5 ←– 3
```

```
tli x3, 5          # x3 ←– 5
tslt x1, x5, x3    # x1 ←– x5 < x3
```

x1 will have a value 1.

3.1.1.8 tSLTU

Set Less Than Unsigned (tSLTU) perform the signed and unsigned comparison between (rs1) and (rs2) and stores the result in (rd).

Syntax

```
sltu rd, rs1 , rs2

where,
      rd     destination register
      rs1    source register 1
      rs2    source register 2
```

Description

tSLTU sets rd to 1 if rs2 is not equal to zero, otherwise sets rd to zero .tSLTU perform signed and unsigned compares respectively, writing 1 to rd if rs1 ¡rs2 , 0 otherwise.

Usage

x1 will have a value 1.

```
tli x5, 3            # x5 ←– 3
tli x3, 5            # x3 ←– 5
tslt x1, x5, x3      # x1 ←– x5 < x3
```

3.1.2 Immediate instructions

Any instruction which contains an operand that is directly encoded as part of an instruction is called an immediate instruction and the operand as immediate operand. This section covers shift and logical operations with immediate operands as part of the instruction.

3.1.2.1 tSLLI

Shift Logically Left Immediate (SLLI) performs logical left on the value in register (rs1) by the shift amount held in the register (imm) and stores in (rd) register.

Syntax

```
tslli rd, rs1 , imm

where,
      rd      destination register
      rs1     source register 1
      imm     immediate data
```

Description

A tSLLI of one position moves each trit to the left by one. The low-order trit (the right-most trit) is replaced by a zero bit and the high-order bit (the left-most bit) is discarded.

Usage

```
tslli x1, x1, 1   # x1 ←– x1<<1
```

3.1.2.2 tSRLI

Shift Logically Right Immediate (tSRLI) performs logical Right on the value in register (rs1) by the shift amount held in the register (imm) and stores in (rd) register.

Syntax

```
tsrli rd, rs1 , imm

where,
      rd     destination register
      rs1    source register 1
```

```
      imm    immediate data
```

Description

A Shift Right Logical Immediate (tSRLI) of one position moves each trit to the Right by one. The most significant trit is replaced by a zero trit and the least significant trit is discarded.

Usage

```
tsrli x1, x1, 1  # x1 ←– x1>>1
```

3.1.2.3 tSRAI

Shift Right Arithmetic Immediate (tSRAI) performs right shift on the value in register (rs1) by the shift amount held in the (imm) and stores in (rd) register.

Syntax

```
tsrai rd, rs1 , imm

where,
      rd      destination register
      rs1     source register 1
      imm     Immediate data
```

Description

tSRAI is arithmetic shift right of a number by 'N' places. The vacated trits at the most significant end are filled with value of sign trit (0 for +ve sign and 1 for -ve sign). This is known as "sign extending".The most significant trit of the original value is the sign trit for 2's complement numbers.

Usage

```
tsrai x1, x1, 1           # x1 ←– x1>>1
```

3.1.2.4 tANDI

AND Immediate (tANDI) performs trinary operation between contents of register (rs1) and immediate data (imm) and stores in (rd) register.

Syntax

```
tandi rd, rs1 , imm

where,
      rd      destination register
      rs1     source register 1
      imm     immediate data
```

Description

A Bitwise tANDI is a binary operation that takes two trit patterns of equal length and performs the logical inclusive tAND Immediate operation over each trits. The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. 32 trits of result is written to the destination register.

Usage

```
tandi x5, x5, 4   # x5 ←– x5 & 4
```

3.1.2.5 tORI

tOR Immediate (tORI) performs binary operation between register (rs1) and Immediate data (imm) and stores in (rd) register.

Syntax

```
ori rd, rs1 , imm

where,
      rd    destination register
      rs1   source register 1
      imm   Immediate data
```

Description

A bitwise tORI is a binary operation that takes two trit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding trits.

Usage

```
tli x5, 0x0100            # x5 ←– 0x0100
tori x1, x5, 0x0010       # x1 ←– x5|2
```

x1 will //TODO edit have a value 0x0110.

3.1.2.6 tXORI

Exclusive-OR Immediate (tXORI) performs bit-wise binary operation between register contents (rs1) and Immediate data (imm) and stores in (rd) register.

Syntax

```
txori rd, rs1 , imm

where,
      rd     destination register
      rs1    source register 1
      imm    Immediate data
```

Description

A bitwise tXORI is a binary operation that takes two trit patterns of equal length and performs logical inclusive tXOR operation on each pair of corresponding trits.

Usage

tslti x5, x1, 2 # x5 ←– x1 < 2

3.1.2.8 tSLTIU

Set Less Than Immediate Unsigned (tSLTIU) does comparison between register contents (rs1) and Immediate data (imm) and sets value in (rd) register.

Syntax

```
tsltiu rd, rs1 , imm

where,
      rd       destination register
      rs1      source register 1
      imm      Immediate data
```

Description

A tSLTIU is a comparison to the contents of register using unsigned comparison. If the value in register is less than the immediate value, the value 1 is stored in destination Register, otherwise, the value 0 is stored in destination register.

Usage

```
tslti x5, x1, 2  # x5 ←– x1 < 2
```

Arithmetic instruction

4.1 T-RV32I

T-RV32I deals with the 32 trit instruction that are used for arithmetic operations. The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. The instructions are broadly classified as register- register and immediate instructions

4.1.1 Register to Register instructions

Register to register instruction involves, both the operands as a register. The contents of the register holds the content of the

operands.

4.1.1.1 tADD

Addition (tADD) adds the contents of two registers and stores the result in another register.

Syntax

```
  tadd rd, rs1 , rs2

  where,
        rd     destination register
        rs1    source register 1
        rs2    source register 2
```

Description

The tADD instruction adds content of the two registers rs1 and rs2 and stores the resulting value in rd register. The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. Overflows are ignored and the lower 32 trits of result is written to the destination register.

Usage

```
 tli x2, 3        # x2 ←– 3
 tli x3, 4        # x3 ←– 4
 tadd x1, x2, x3  # x1 ←– x2 + x3
```

Assuming rs1 (x2) and rs2 (x3) contain values 3 and 4 respectively, an addition operation on them will result in value 7 which will be stored in rd (x1). x1 will have a value 7.

4.1.1.2 tSUB

Subtraction (tSUB) subtracts contents of one register from another and stores the result in another register.

Syntax

```
  tsub rd, rs1 , rs2

  where,
        rd     destination register
        rs1    source register 1
        rs2    source register 2
```

Description

The tSUB instruction subtracts content of the source register rs2 from rs1 and stores the value in the register rd. Overflows are ignored and the lower XLEN trits of the result is written to rd. The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. The overflows as well as borrow are ignored and the lower 32 trits of result is written to the destination register.

Usage

```
 tli x2, 4         # x2 ←– 4
 tli x3, 3         # x3 ←– 3
 tsub x1, x2, x3   # x1 ←– x2 - x3
```

x1 will have a value 1

4.1.1.3 tMUL

Multiplication (tMUL) calculates the product of the multiplier in source register 1 (rs1) and mul- tiplicand in source register 2 (rs2), with the resulting product being stored in destination register (rd).

Syntax

```
  tmul rd, rs1 , rs2

  where,
        rd     destination register
        rs1    source register 1
        rs2    source register 2
```

Description

tMUL calculates the product of two XLEN-bit operands in the source registers 1 and 2 (rs1 , rs2). This instruction stores the less significant part of the result in the destination register and any overflow is ignored.

Usage

```
tmul x4, x9, x13  # x4 ←– Low //TODO edit  Bits [x9 * x13]
```

4.1.1.4 tMULH

Multiply signed and return upper trits (tMULH)) calculates the product of signed values in source registers (rs1) and (rs2) and stores result in the specified destination register (rd).

Syntax

```
tmulh rd, rs1 , rs2

where,
      rd      destination register
      rs1     source register 1
      rs2     source register 2
```

Description

tMULH calculates the product of signed multiplier and signed multiplicand (present in the two source registers specified respectively), and places the upper XLEN trits of the full 2*XLEN product, into the destination register. tMULH has to be used with tMUL to get the complete 2*XLEN trits result.

Usage

```
tli x1,-80           # x1 ←– -80
tli x5,20            # x5 ←– 20
tmulh x5, x5, x1     # x5 ←– High Bits[x5*x1]
```

4.1.1.5 tMULHU

Multiply Unsigned and return upper trits (tMULHU)) calculates the product of two unsigned values in source registers rs1 and rs2 . The resulting value is placed in the specified destination register (rd).

Syntax

```
tmulhu rd, rs1 , rs2

where,
      rd       destination register
      rs1      source register 1
      rs2      source register 2
```

Description

tMULHU multiplies two unsigned operands in the source registers and the most significant part of result is stored in the destination register.

Usage

```
tli x1,-80           # x1 ←– -80
tli x5,20            # x5 ←– 20
tmulhu x5, x5, x1    # x5 ←– High Bits [x5*x1]
```

4.1.1.6 tMULHSU

Multiply Signed-Unsigned and return upper bits (tMULHSU)) calculates the product of a signed value in source register rs1 with an unsigned value in source register rs2 and the result- ing product is stored in destination register, rd.

Syntax

```
tmulhsu rd, rs1 , rs2

where,
      rd    destination register
```

```
        rs1   source register 1
        rs2   source register 2
```

Description

tMULHSU computes the product of the signed, most significant word of the multiplier and the unsigned, least significant word of the multiplicand. The most significant part of the resulting product is stored in the specified destination register. The resulting value is a signed value.

Usage

```
 tli x1,-80            # x1 ←– -80
 tli x5,20             # x5 ←– 20
 tmulhsu x5, x5, x1    # x5 ←– //TODO edit  High Bits[x5*x1]
```

4.1.1.7 tDIV

Division (tDIV) performs division on the value in source register (rs1) with the value in the source register (rs2) and stores quotient in (rd) register.

Syntax

```
  tdiv rd, rs1 , rs2

  where,
        rd     destination register
        rs1    source register 1
        rs2    source register 2
```

Description

tDIV does the division of operands in source registers and stores quotient in the destination register. Both operands and the result are signed values.

Usage

```
 tli x9, -400            # x9 ←– -400
 tli x13, 200            # x13 ←– 200
 tdiv x4, x9, x13        # x4 ←– x9/x13
```

4.1.1.8 tDIVU

Division Unsigned (DIVU) performs unsigned Division on the value in source register (rs1) by the value in the source register (rs2) and stores quotient in the destination register (rd).

Syntax

```
  tdivu rd, rs1 , rs2

  where,
        rd      destination register
        rs1     source register 1
        rs2     source register 2
```

Description

tDIVU does the division of unsigned operands in source registers and stores quotient in the destination register. Both operands and the result are unsigned values.

Usage

```
 tli x9, 400            # x9 ←– 400
 tli x13,200            # x13 ←– 200
 tdivu x4, x9, x13      # x4 ←– x9/x13
```

4.1.1.9 tREM

Reminder (tREM) performs division on the value in source register (rs1) with the value in the source register (rs2) and stores remainder in (rd) register.

Syntax

```
trem rd, rs1 , rs2

where,
      rd     destination register
      rs1    source register 1
      rs2    source register 2
```

Description

tREM does the signed division of operands in source registers and stores the remainder in the desti- nation register. Both operands and the result are signed values.

Usage

```
tli x9, 400              # x9 ←– 400
tli x13,200              # x13 ←– 200
trem x4, x9, x13         # x4 ←– x9%x13
```

NOTE:

Sometime's a programmer needs both quotient and remainder. In such cases it is recommended to perform tDIV first and tREM later

4.1.2 Immediate Instructions

Instructions involving a constant operand are immediate instructions. Here we are going to load and store immediate instructions.

4.1.2.1 tLI

Load Immediate (tLI) load register rd with a value that is immediately available

Syntax

```
tli rd, imm

where,
      rd       destination register
      imm      Immediate data
```

Description

The tLI instruction loads a positive or negative value that is immediately available, without going into memory. The value maybe a 16-trit or a 32-trit integer.

Usage

```
tli x5, 24 # x5 ←– 24
```

4.1.2.2 tADDI

Add Immediate (tADDI) adds content of the source registers rs1 , immediate data (imm) and store the result in the destination register (rd).

Syntax

```
taddi rd, rs1 , imm

where,
      rd      destination register
      rs1     source register 1
      imm     Immediate data
```

Description

The tADDI instruction adds content of a source register with an absolute value and stores the result in the destination register. Overflows are ignored and the lower 32 trits of result is written to the destination register.

Usage

```
tli x2,24              # x2 ←– 24
taddi x1, x2,64        # x1 ←– x2 + 64
```

x1 will have a value 88.

Control Transfer instruction

5.1 Branch Instructions

A branch instruction in a program causes the system to execute a different instruction sequence, making the system deviate from its normal course of action of executing instructions in sequence. Branches are useful for implementing logical constructs since the architecture allows compares and dependent branches to be scheduled in the same cycle.

5.1.0.1 tBEQ

Branch If Equal (tBEQ) the contents of source register rs1 is compared with source register rs2 , if found equal, the control is transferred to the specified label.

Syntax

```
  tbeq rs1 , rs2 , label

  where,
        rs1
        rs2
        label
```

Description

The tBEQ instruction compares contents of (rs1) is compared to the contents of (rs2). If equal, control jumps. The target address is given as a PC-relative offset. More precisely, the offset is sign-extended, multiplied by 2, and added to the value of the PC. The value of the PC used is the address of the instruction following the branch, not the branch itself. The offset is multiplied by 2, since all instructions must be half word aligned.

Usage

```
 loop: addi x5, x1, 1      # x5 ←– x1 + 1
       beq x0, x0, loop    # x0 = x0 jump to loop
```

5.1.0.2 tBNE

Branch If Not Equal (tBNE) the contents of source register rs1 , is compared with source register rs2 if they are not equal control is transferred to the label as mentioned.

Syntax

```
  tbne rs1 , rs2 , label

  where,
        rs1
        rs2
        label
```

The tBNE instruction compares contents of (rs1) is compared to the contents of (rs2). If not equal, control jumps. The target address is given as a PC-relative offset.

Usage

```
 label: taddi x4, x9,123      # x4 ←– x9 + 123
        tbne x4, x9, label    # x4 6= x9 jump to label
```

5.1.0.3 tBLT

Branch If Less Than (BLT) the contents of source register rs1 , is compared with contents of source register rs2 . If (rs1) is less than (rs2) control is transferred to the label as mentioned.

Syntax

```
  tblt rs1 , rs2 , label

  where,
        rs1      source register 1
```

```
        rs2      source register 2
        label
```

Description

The tBLT instruction compares contents of (rs1) is compared to the contents of (rs2). If (rs1) contents is less than (rs2)(signed comparison), control jumps. The target address is given as a PC-relative offset.

Usage

```
 label: taddi x4, x9, 123           # x4 ←− x9 + 123
        tblt x4, x9, label          # x4 < x9 jump to label
```

5.1.0.4 tBLTU

Branch If Less Than Unsigned (BLTU) the contents of source register rs1 , is compared with con- tents of source register rs2 if (rs1) is less than (rs2) control is transferred to the label as mentioned.

Syntax

```
  tbltu rs1 , rs2 , label

  where,
        rs1       source register 1
        rs2       source register 2
        label
```

Description

The tBLTU instruction compares contents of (rs1) is compared with the contents of (rs2). If (rs1) contents is less than (rhan (rs2), (unsigned comparison) control jumps. The target address is given as a PC-relative offset.

Usage

```
 loop: taddi x1, x0, 1       # x1 ←− x0 + 1
       taddi x5, x0, 3       # x5 ←− x0 + 3
       tbltu x1, x5, loop    # x1 < x5 jump to loop
```

5.1.0.5 tBGE

Branch If Greater Than or Equal, signed (BGE) the contents of source register rs1 , is com- pared with contents of source register rs2 if (rs1) is greater than (rs2) control is transferred to the label as mentioned.

Syntax

```
  tbge rs1 , rs2 , label

  where,
        rs1      source register 1
        rs2      source register 2
        label    reference to a valid memory location
```

Description

The tBGE instruction compares contents of (rs1) with the contents of (rs2). If (rs1) contents is greater than or equal to contents of (rs2), (signed comparison) control jumps to the specified location. The target address is given as a PC-relative offset.

Usage

```
 label: addi x4, x9, 123       # x4 ←− x9 + 123
        bge x4, x9, label      # if x4 ≥ x9 jump to label
```

5.1.0.6 tBGEU

Branch If Greater Than or Equal, Unsigned (tBGEU) the contents of source register rs1 , is com- pared with contents of source register rs2 . If rs1 is greater than or equal to rs2 , control is transferred to the label as mentioned.

Syntax

```
  tbgeu rs1 , rs2 , label
```

```
where,
      rs1
      rs2
      label
```

Description

The tBGEU instruction compares contents of (rs1) is compared with the contents of (rs2). If (rs1) contents is greater than (rs2), (unsigned comparison) control jumps. The target address is given as a PC-relative offset.

5.1.1 Pseudo Instructions

Branching instructions in this section are pseudo or convenient instructions to be used in place of the base instructions.

5.1.1.1 tBEQZ

Branch if Equal to Zero (BEQZ) instruction jumps to a specified location in the program if the condition, equal to zero is met.

Syntax

```
tbeqz rs1 , label

Translation

tbeq rs1 , x0, label

where,
      rs1     source register
      label   Address to JUMP to
```

Description

The tBEQZ translates to beq rs1 , x0, label, as the expansion reveals, the (rs1) contents is com- pared with the zero register (x0) and the program counter branches to the specified label if the condition equal to zero is met.

Usage

```
      tli x6, 0           # x6 = 0
loop: tli x5, x5, 100     # Example operation
      tbeqz x6, loop      # x6 = 0 branch to loop
```

Assume rs1 (x6) is initialized to 0 and there is an example operation within the specified label (loop). tBEQZ on register rs1 (x6) will shift the program counter to the specified label since the contents of rs1 (x6) is indeed 0.

5.1.1.2 tBNEZ

Branch if Not Equal to Zero (tBNEZ) jumps to a specified location in the program if the condi- tion, not equal to zero is met.

Syntax

```
tbnez rs1 , label

Translation

tbne rs1 , x0, label

where,
      rs1          source register 1
      label        Address to JUMP to
```

Description

The tBNEZ instruction translates to tBNE. As the translation reveals, the contents of rs1 is compared with the zero register (x0) and branches to the specified label, if the condition that the contents of rs1 register is not equal to zero, is met.

Usage

```
       tli x6, 50           # x6 = 50
loop:  taddi x5, x6, 100    # Example operation
       tbnez x6, loop       # x6 6= 0 jump to loop
```

Assume rs1 (x6) is initialized to 50 and there is an example operation within the specified label (loop). tBNEZ on register rs1 (x6) will shift the program counter to the specified label since the contents of rs1 (x6) is indeed not equal to 0.

5.1.1.3 tBLEZ

Branch if Less Than or Equal to Zero (tBLEZ) the program counter branches to the specified location if the condition, less than or equal to zero.

Syntax

```
tblez rs1 , label

Translation

tbge x0, rs1 , label

where,
      rs1     source register 1
      label   Address to JUMP to
```

Description

The tBLEZ expands to BGE. This instruction is a signed comparison instruction which shifts the program counter to the specified location if value in rs1 is less than or equal to 0.

Usage

```
      tli x6, -50           # x6 = −50
loop: taddi x5, x6, 100     # Example operation
      tblez x6, loop        # x6 ≤ 0 jump to loop
```

Assuming rs1 (x6) is initialized to -50, tBLEZ, shifts the program counter to label (loop) since the condition that rs1 (x6) should to either less than or equal to 0, is met.

5.1.1.4 tBGEZ

Branch if greater than or equal to Zero (BGEZ) checks if register rs1 is greater than or equal to zero, if the condition is met, the program counter branches to the specified label.

Syntax

```
tbgez rs1 , label

Translation

tbge rs1 , x0, label

where,
      rs1      source register 1
      label    Address to JUMP to
```

Description

The tBGEZ expands to tBGE. This instruction compares if contents of rs1 is greater than or equal to zero (x0). If the conditions are met, the program counter branches to the specified label.

Usage

```
      tli x6, 50           # x6 = 50
loop: taddi x5, x6, 100    # Example operation
      tbgez x6, loop       # x6 ≥ 0 jump to loop
```

Assuming that rs1 (x6) is initialized to a value 50, tBGEZ instruction shifts the program counter to label (loop) since the condition, rs1 (x6) must be greater than or equal to 0, is satisfied.

5.1.1.5 tBLTZ

Branch if Less Than Zero (tBLTZ) shifts the program counter to a specified location if the value in a register is less than zero.

Syntax

```
tbltz rs1 , label
```

```
Translation

tblt rs1 , x0, label

where,
      rs1      source register 1
      label    Address to JUMP to
```

Description

tBLTZ is a signed comparison instruction with its base instruction being BLT. The value in rs1 is compared with x0 and shifts the program counter to the specified location in case its contents are less than 0.

Usage

```
      tli x6, -20           # x6 = −20
loop: taddi x5, x6, 100     # Example instruction
      tbltz x6, loop        # x6 < 0 jump to loop
```

Assuming rs1 (x6) is initialized to -20, BLTZ shifts the program counter to label (loop) since the contents of rs1 (x6) is indeed less than 0. The program then executes the instructions within the label (loop).

5.1.1.6 tBGTZ

Branch if Greater Than Zero (BGTZ) shifts the program counter to a specified location, if the contents of a register is found to be greater than zero.

Syntax

```
tbgtz rs1 , label
```

Syntax

```
tblt x0, rs1 , label

where,
      rs1      source register 1
      label    Address to JUMP to
```

Description

The tBGTZ is a signed comparison instruction which translates to its base instruction tBLT. If the contents of rs1 is greater than x0, the program counter shifts and continues its execution with the instructions in the location specified.

Usage

```
        tli x6, 5               # x6 = 5
 loop:  taddi x5, x6, 100       # Example instruction
        tbgtz x6, loop          # x6 > 0 jump to label
```

Assuming that rs1 (x6) is initialized to value 5, the tBGTZ instruction shifts the program counter to label (loop), since rs1 (x6) is greater than 0. Program execution continues with what label (loop) contains.

5.1.1.7 tBGT

Branch if Greater Than (BGT) instruction shifts the program counter to the specified location if the value in a register is greater than that of another.

Syntax

```
tbgt rs1 , rs2 , label
```

Translation

```
  tblt rs2 , rs1 , label

  where,
       rs1      source register 1
       rs2      source register 2
```

```
label    Address to JUMP to
```

Description

The tBGT is a signed comparison instruction which translates to tBLT. In this instruction, it is examined if the contents of rs2 is less than the contents of register rs1 . If the condition is satisfied, program counter branches to the location specified.

Usage

```
       tli x5, 30          # x5 = 30
       tli x6, -25         # x6 = −25
 loop: taddi x7, x6, 100   # Example instruction
       tbgt x5, x6, loop   # x6 < x5 jump to loop
```

Assuming rs1 (x5) is initialized to 30 and rs2 (x6) is initialized to -25. Since the condition rs2 (x6) should be less than rs1 (x5) to branch, is true (tBGT translates to tBGT), the program branches to label (loop) and continues execution.

5.1.1.8 tBLE

Branch if Less Than or Equal (tBLE) instruction shifts the program counter to the specified lo- cation if the value in a register is less than or equal to that of another.

Syntax

```
tble rs1 , rs2 , label
```

Translation

```
tbge rs2 , rs1 , label

where,
      rs1      source register 1
      rs2      source register 2
      label    Address to JUMP to
```

Description

The tBLE is a signed comparison instruction which examines if the contents of rs1 is less than or equal to the contents of register rs2 . If the condition is satisfied, program counter branches to the location specified.

Usage

```
      tli x5, -25          # x5 = −25
      tli x6, 30           # x6 = 30
loop: tble x5, x6, loop    # Example instruction
```

Assume rs1 (x5) is initialized to -25 and rs2 (x6) is initialized to 30, the program branches to the specified label (loop) since rs1 (x5) is less than rs2 (x6).

5.1.1.9 tBGTU

Branch if Greater Than, Unsigned (BGTU) an unsigned comparison instruction to examine if contents of one register is greater than the other, according to which the program counter branches to the specified label.

Syntax

```
tbgtu rs1 , rs2 , label
```

Translation

```
tbltu rs2 , rs1 , label

where,
      rs1       source register 1
      rs2       source register 2
      label     Address to JUMP to
```

Description

The tBGTU is an unsigned comparison instruction which examines if the contents of rs1 is greater than rs2 . If the condition is

satisfied, the program counter shifts to the specified location and continues executing instructions from there on.

Usage

```
      tli x6, 50         # x6 = 50
      tli x7, 10         # x7 = 10
loop: tbgtu x6, x7, loop # x6 > x7 Jump to loop
```

Assume rs1 (x6) is initialized to 50 and rs2 (x7) is initialized to 10. The program shifts to the specified label (loop) as rs1 is greater than rs2 .

5.1.1.10 tBLEU

Branch if Less Than or Equal, Unsigned (BLEU) instruction examines whether the of one reg- ister is less than or equal to the other and the program counter shifts accordingly.

Syntax

```
  tbleu rs1 , rs2 , label
```

Translation

```
  tbgeu rs2 , rs1 , label

  where,
        rs1      source register 1
        rs2      source register 2
        label    Address to JUMP to
```

Description

tBLEU is an unsigned comparison instruction which examines if contents of rs1 is less than or equal to that of rs2 . If the condition is satisfied, the program counter branches to the specified label.

Usage

```
       tli x6, 20             # x6 = 20
       tli x7, 25             # x7 = 25
 loop: taddi x5, x7, 100      # Example instruction
       tbleu x6, x7, loop     # x6 ≤ x7 Jump to loop
```

Assuming rs1 (x6) is initialized to 20 and rs2 (x7) is initialized to 25. Since rs1 (x6) is less than rs2 (x7), the tBLEU instruction branches the program counter to the specified label (loop).

5.1.1.11 tRET

Return from Subroutine (tRET) pseudo-instruction used at the end of a subroutine to return to its caller.

Syntax

```
  label: tret

  where,
        label      sub-routine
```

Description

The tRET translates to jalr x0, 0(ra). This instruction jumps to the address in the ra, but does not save a return address. The instruction will ensure that execution continues from where the call was made.

Usage

```
 tli x6, 50
 tli x7, 20
 taddi x5, x7, 100
 tret             # Return back to caller
```

5.2 Unconditional Jump Instructions

Unconditional Jump Instructions transfers the program sequence to the specified memory address without a condition.

5.2.0.1 Jump and Link

Jump and Link (tJAL) is used to call a subroutine (i.e., function).

Syntax

```
  tjal rd, offset

  where,
        rd          destination register
        offset      offset value
```

Description

The tJAL instruction is used to call a subroutine (i.e., function). The return address (i.e., the PC, which is the address of the instruction following the tJAL) is saved in the destination register. The target address is given as a PC-relative offset, more precisely, the offset is sign-extended, multiplied by 2, and added to the value of the PC. The value of the PC used is the address of the instruction following the JAL, not the JAL itself. The offset is multiplied by 2, since all instructions must be half word aligned.

Usage

```
 loop: taddi x5, x4, 1     # x5 ←– x4 + 1
       tjal x1, loop       # Goto loop x1 ←– address[loop]
```

5.2.0.2 tJALR

Jump and Link Register (JALR) is used to invoke a subroutine call (i.e., function/method/pro- cedure).

Syntax

```
  tjalr rd, offset

  where,
        rd        destination register
        offset    offset value
```

Description

The tJALR instruction is used to call a subroutine (i.e., function). The return address (i.e., the PC, which is the address of the instruction following the tJALR) is saved in the destination register. The target address is given as a PC-relative offset, more precisely, the offset is sign-extended and added to the value of the destination register. The offset is not multiplied by 2.

Usage

```
       taddi x1, x0, 3      # x1 ←– x0 + 3
 loop: taddi x5, x0, 1      # x5 ←– x0 + 1
       tjalr x0, 0(x1)      # x0 ←– mem[x1 + 0]
```

5.2.0.3 tJ

Jump (tJ) is a pseudo-instruction which uses Jump and Link (JAL) instead and sets the destination register to zero to discard return address.

Syntax tj label

```
  where,
        tj      Jump
        label   A string that points to an instruction
```

Description

tJ is a plain unconditional jump (tUJ-type) instruction used to jump to anywhere in the code memory. This instruction translates to tjal x0, label, which sets the return address to zero thus discarding the return address.

Usage

```
 loop:  tli x6, 100             # x6 ←– 100
        tli x7, 100             # x7 ←– 100
        tli x1, 1000            # x1 ←– 1000
        tadd x5, x6, x7         # x5 ←– x6 + x7
        tbge x5, x1, load1      # x5 ≥ x1
```

```
load1: tli x5, x0              # x5 ←– 0
       tj loop                 # Jump to loop
```

5.2.0.4 tJR

Jump Register (tJR) is a pseudo-instruction which translates to Jump and Link Register (tJALR) which jumps to the address and places the return address in a general purpose register (GPR).

Syntax

```
tjr rs1

where,
      jr     Jump Register
      rs1    Return Address
```

Description

tJR is translated to tjalr rd, rs1 , imm where, rd is zero register, rs1 contains the target address and imm is given the value 0. In this instruction, the rd field is set to zero thereby performing the jump to the address in ra register but does not save a return address.

Usage

```
label: tli x28, 100            # x1 ←– 100
       tli x5, 200             # x5 ←– 200
       tli x6, 50              # x6 ←– 50
       tjal ra, loop           # ra ←– loop
       tli x2, 10              # x2 ←– 10
loop:  tadd x4, x28, x5        # x4 ←– x28 + x5
       tsub x7, x6, x4         # x7 ←– x6 + x4
       tjr ra                  # JumpRegister
```

5.3 System Instructions

SYSTEM instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions. CSR instructions are described in this

5.3.1 ECALL

Environment Call (ECALL) instruction is used to implement system calls. Also, ECALL is used to transfer control from lower privilege level to higher privilege level.

Syntax

```
ecall
```

Description

The ECALL instruction is used to implement system calls. System calls are subroutine calls made from a lower privilege code to a higher privilege code. The execution happens in the higher privilege level and result is given back to the lower privilege code. Once the desired operation is over, the control returns back to the lower privilege level. Generally, if an operation needs to be done at a higher privilege level, ECALL is used. For example, the implementations of libraries for FILE operations in a Unix operating system, uses ECALL. On execution of ECALL, one of the following exception arise:

• Environment Call from User Mode • Environment Call from Supervisor Mode • Environment Call from Machine Mode

As described in the section “mcause”, the above exceptions have a dedicated exception code. The trap handler in higher privilege level handles the exception and redirects the call to the corresponding subroutine. The arguments are passed through argument registers (ai) and result is saved in Saved register (si).

Usage

```
taddi x5, x0, 4          # x5 ←– 0 + 4
ecall                    # Atomic jump to location 0x80000180
```

5.3.2 EBREAK

Environment Break (EBREAK) is an assembly instruction that is used to stop the execution sud- denly.

Syntax

```
ebreak
```

Description

The EBREAK instruction is used to invoke a debugger, by causing a “Breakpoint” exception. Typically the debugging software will insert this instruction at various places in the application code sequence, in order to gain control from an executing program.

Usage

```
tla x1, msg             # x1 ←– address[msg]
tli x2, 0x11100111      # x2 ←– 0x11100111
ebreak                  # Debugger Breakpoint to test code

sw x5, 0(x1)            # ValueAt[x1 + 0] ←– x5
.section .rodata
msg: .string "Hello World!"
```

5.3.3 WFI

Wait For Interrupt (WFI) instruction causes the processor to suspend instruction execution. The processor will wake up when an asynchronous interrupt occurs and resumes execution.

Syntax

```
WFI
```

Description

On execution of WFI trap handler will be invoked and upon return to the code sequence containing the WFI instruction, the next instruction following the WFI will be executed.

5.3.4 tNOP

The No Operation (tNOP) instruction executes silently. It does not change registers, memory or processor statues. Only the program counter is advanced.

Syntax

```
tnop
```

Description

tNOP is a pseudo instruction that expands to taddi x0, x0, 0. The x0 is a read-only register holding the value zero. Anything, written to x0 register is discarded. The tNOP instruction does not change any architecturally visible state, except for advancing the pc and increment any applicable performance counters. As RISC-V has no arithmetic flags (i.e., carry, overflow, zero, sign flags), any arithmetic operation whose destination register is x0 will endup as a no operation instruction regardless of the source registers.

Usage

Lets say pc is at 0n3000000000000000. After execution of below instruction.

```
nop # pc ←– pc + 2
```

c becomes 0n3000000000000002. The state of the machine is unchanged.

Trap’s in TRIT-RISC-V

Trap is a specific scenario caused by a exceptional condition or interrupt. In TRIT-RISC-V, the term trap refers to, transfer of control to a trap handler caused either by an exception or an interrupt. Exception is an unusual condition occurring at run time of an instruction in the current TRIT-RISC- V hart. An exception disrupts the normal flow of instruction execution. Exceptions are usually synchronous. Interrupts are another form of a trap, where the origin of interrupt is from Timer or peripherals. Interrupt is a scenario designed to service a specific external input. All the Traps can be handled or ignored. It is upto the software to decide. A “trap handler” is a subroutine that handles the trap in a software. The way of handling a trap is left to the software designer and varies from one type of trap to another.

6.1 Exceptions

Exceptions are usually synchronous and always tied to an assembly instruction. A exception can arise at any stage of execution of an

instruction. For example, during instruction decode stage, the hardware may detect a bad opcode field. This will trigger a “illegal instruction” exception. When an exception happens, the hardware sets the mcause register with the corresponding exception code. The pc is set to the trap handler base address. The exception code helps to identify the type of exception. The possible exceptions in TRIT-RISC-V are listed in Table

• Illegal instruction • Instruction/Load/Store address misaligned • Instruction/Load/Store access fault • Environment call • Break point

6.1.1 Illegal Instruction Exception

The exception occurs when the programs tries to execute any illegal instruction. For example trying to write on a read-only tCSR register will generate a illegal instruction exception.

Example:

tli t0, 8 # t0 ←– 8 tcsrrs x0, mhartid, t0 # Attempt to write to a read-only tCSR, generates exception

6.1.2 Instruction Address Misaligned Exception

The exception occurs when the programs tries to execute an unconditional jump or take a branch, wherein the target address is not 4 byte aligned. For example, executing a program with start address as 0x80000001. This will generate a instruction address misalignment exception on a unconditional jump.

Note:

Instruction address misaligned exceptions are not possible on machines that support extensions with 16-bit aligned instructions, such as the compressed instruction-set extension, C.

Example:

start address set to 0x80000001 (start not aligned to 4 byte boundary.

start: tla x15, loop # x15 ←– Address (loop) tjalr ra, x15 ,0 # Jumping to a label (loop) which is not 4 byte aligned loop: # This causes an Instruction address misalignment exception tj loop # x10 ←– x10+1 taddi x10, x10,1 # Jump to loop

6.1.3 Load Address Misaligned Exception

The exception occur when the programs tries to execute an load instruction to access data from misaligned address or an address that is not 4 byte aligned. For example, trying to access a data section without using a properly aligning it would cause this exception.

Example:

```
tla x15, data1           # x15 ←– Address ( data1)
tlw x10, 0 ( x15 )       # x10 ←– Content(x15)
# Trying to load from a misaligned address ( data1)
tli t0, 8
data1:                    # data1 section is not aligned to 4 byte boundary
.word 3                   # Load access at data1 causes a misaligned exception
.word 2
```

6.1.4 Store Address Misaligned Exception

The exception occurs when the programs tries to execute an store instruction at a misaligned address (Address that is not four byte aligned). For example trying to store data into a data section without using proper alignment, would cause this exception.

Example:

```
tla x15, data1        # x15 ←– ( data1) memory address
tsw x10, 0 ( x15 )    # mem[x15+0] ←– x10
                      # Trying to store at a misaligned address ( data1)
data1:                # data1 section is not aligned to 4 byte boundary
.word 3               # Store access at data1 causes a misaligned exception
.word 2
```

6.1.5 Instruction Access Fault

The exception occurs when the programs tries to access an instruction on a invalid memory location. For example executing

unconditional jump instruction to a memory location which is out of bounds of the physical memory.

Example:

```
tla x15, data1              # x15 ←– Address of label ( data1)
tjalr ra,-1(x15)            # Jumping to wrong addr, decoding contents at that addr

data1:
.word 100
.word 99
```

In the above case, data1 holds data values. The data values are aligned at word boundary. Now, we jump to a location, that is data1 - 1 byte memory location. Here, when we execute 'jalr', an instruction access fault happens. The jump should have happened at 4 byte aligned address.

6.1.6 Load Access Fault

The exception occurs when the programs attempt to do a load on a invalid memory location. For example trying to load from address which is more than the bound of memory or inaccessible by memory. Certain registers are 32 bits of size. A 64 bit load operation might thrown an error.

Example:

```
start:
tla x15, start           # x15 ←– Address ( start)
tld x16, -16 ( x15 )     # x16 ←– Content(x15-16) -Exception generated
```

6.1.7 Store Access Fault

The exception occurs when the programs attempts to do a store on an invalid memory location. For example, trying store to address which is more than the bound of memory or inaccessible by memory.

Example:

```
start:
tla x15, start               # x15 ←– Address ( start)
tsd x16, -16 ( x15 )         # x16 –→ Content(x15-16) -Exception generated
```

6.1.8 Break Point

The exception occurs when the programs executes a break-point set in the program to enter debug mode.

6.1.9 Environment Call

This exception occurs when the programs executes a system call. The system call is realized in RISC-V using ecall instruction. The ecall instructions can also used to switch from lower privilege modes to higher privilege modes. An example ecall instruction is demonstrated below.

Example:

```
taddi x10, x10, 2        # Environment call exception generated
ecall
```