

Vicuna MDU Design Specification

|  |  |
| --- | --- |
| Document Number | DSP-00106 |
| Date Issued | 2016-11-21 |
| Status | In-progress |



**CONFIDENTIAL**

Revision History

| Rev. | Revision Date | Revised Chapter-Section | Revised Content | Author |
| --- | --- | --- | --- | --- |
| 1.0 | 2016-11-21 | All | Initial version | John Chen |
| 1.1 | 2016-12-01 | All | 1. Remove that MDU can start on DONE state 2. Add Function index to Chapter 2. 3. Move chapter4 into chapter 2. 4. Modify logic symbol 5. Modify Figure; MDU status | John Chen |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Review History** | | | | |
| **Rev.** | **Review Date** | **Reviewed Chapter-Section** | **Comments** | **Reviewer** |
| **0.1** | 2016-12-01 | All | 1. Remove that MDU can start on DONE state 2. Add Function index to Chapter 2. 3. Move chapter4 into chapter 2. 4. Modify logic symbol 5. Modify Figure; MDU status | Alex  Chris  Jack  Kylie |

|  |  |  |  |
| --- | --- | --- | --- |
| **Document Sign-off List** | | | |
| **Department** | **Name** | **Date** | **Comment** |
| VLSI |  | 2014-mm-dd |  |

Table of Contents

[Revision History ii](#_Toc468888210)

[List of Tables vi](#_Toc468888211)

[List of Figures vii](#_Toc468888212)

[1. Overview 1](#_Toc468888213)

[1.1. Introduction 1](#_Toc468888214)

[1.2. Features 1](#_Toc468888215)

[1.3. Block Diagram 2](#_Toc468888216)

[1.4. Function Description 2](#_Toc468888217)

[2. Signal Description 3](#_Toc468888218)

[2.1. MDU Signals 3](#_Toc468888219)

[2.2. MDU Operations 5](#_Toc468888220)

[2.2.1. Normal Operation 5](#_Toc468888221)

[2.2.2. Killed Operation 6](#_Toc468888222)

[3. Micro-Architecture 7](#_Toc468888223)

[3.1. MDU Status 7](#_Toc468888224)

[3.2. MDU Data Path block diagram 8](#_Toc468888225)

[3.3. MDU Arithmetic Algorithms 9](#_Toc468888226)

[3.3.1. Multiplication Operation 9](#_Toc468888227)

[3.3.2. Division Operation 12](#_Toc468888228)

[4. Design Configuration 16](#_Toc468888229)

[5. Programming Model 17](#_Toc468888230)

List of Tables

[Table 1. MDU Design Configurations 3](#_Toc468888190)

[Table 2. Signal Definition 4](#_Toc468888191)

[Table 3. MDU Function Encode 4](#_Toc468888192)

[Table 4 .RISCV Multiplication Operation Instructions 9](#_Toc468888193)

[Table 5. Hardware operation example of 8-bit Unsigned Multiplication 11](#_Toc468888194)

[Table 6. Hardware operation example of 8-bit signed Multiplication 11](#_Toc468888195)

[Table 7 .RISCV Division Operation Instructions 12](#_Toc468888196)

[Table 8 .The Summary of Division by Zero and Signed Division Overflow 13](#_Toc468888197)

[Table 9. Example of 8-bit Unsigned Non-Restoring Division 14](#_Toc468888198)

[Table 10. Example of 8-bit Signed Non-Restoring Division 15](#_Toc468888199)

List of Figures

[Figure 1. Block Diagram 2](#_Toc468888200)

[Figure 2. Logic Symbol 3](#_Toc468888201)

[Figure 3. MDU Operation Example 5](#_Toc468888202)

[Figure 4. MDU Killed Operation 6](#_Toc468888203)

[Figure 5 . MDU Status 7](#_Toc468888204)

[Figure 6. MDU Data Path Architecture 8](#_Toc468888205)

[Figure 7. Multiply-Accumulate Operation 10](#_Toc468888206)

[Figure 8. Hardware Architecture of Multiple-Accumulate Operation 10](#_Toc468888207)

[Figure 9. Non-Restoring Division 13](#_Toc468888208)

[Figure 10. Hardware Architecture of Non-Restoring Division Operation 14](#_Toc468888209)

**Typographical Convention Index**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Document Element | Font | Font Style | Size | Color |
| Normal | Georgia | Normal | 12 | Black |
| Code | Lucida Console | Normal | 11 | Indigo |
| USER VARIABLE | Lucida Console | ALL-CAPS | 11 | INDIGO |
| Note/Warning | Georgia | Normal | 12 | Red |
| Hyperlink | Georgia | Bold + Underlined | 12 | Blue |

# Overview

## Introduction

Multiplication Division Unit (MDU) supplies integer multiplication and division based on RISCV M Standard Extension. The Operands can be signed or unsigned. The architecture of MDU includes a 64-bit adder (RV64, it would be 32-bit adder on RV32) which shared by Multiplication and Division.

## Features

List the features of the design:

* Parameterized Design (for RV32 and RV64).
* Dynamic latency which based operands to achieve much more performance.
* Calculated based on RISCV M Standard Extension.
* Resource sharing to minimize area.

## Block Diagram

Figure 1 shows the block diagram that the blocks communicate with MDU.

IIU applies the request, GPR index, operands and the function that required.

While the operation is done, MDU responses result with GPR index to update the proper GPR.

MDU

Request

Interface

MDU

Response

Interface

MDU

IIU

Figure 1. Block Diagram

## Function Description

MDU performs *Multiply-Accumulator* operation for multiplication and *non-restoring division* algorithm for division.

For further information, please refer to 3.3 MDU Arithmetic Algorithm at page 9.

# Signal Description

## MDU Signals

MDU calculates result depends on mdu\_req\_func[3:0], and response it to system with GPR index.

Table 1 performs the design configurations of MDU;

Table 1. MDU Design Configurations

|  |  |
| --- | --- |
| Parameter | Description |
| XLEN | **32** : Supports RV32M, operand size = 32 bits  **64** : Supports RV64M, operand size = 64 bits (default) |
| MUL\_RADIX | **2**: Radix-2 multiplication. Calculate 1 bit per cycle  **4**: Radix-4 multiplication. Calculate 2 bit per cycle  **16**: Radix-16 multiplication. Calculate 4 bit per cycle (default)  **256**: Radix-256 multiplication. Calculate 8 bit per cycle |

Table 2 performs the definition of MDU signals and Table 3 shows the encoding of MDU function.

MDU

core\_clk

core\_reset\_n

mdu\_kill

mdu\_req\_valid

mdu\_req\_func[3:0]

mdu\_req\_tag[4:0]

mdu\_req\_op0[XLEN-1:0]

mdu\_req\_op1[XLEN-1:0]

mdu\_req\_ready

Note: XLEN = 32 or 64

mdu\_resp\_valid

mdu\_resp\_tag[4:0]

mdu\_resp\_result[XLEN-1:0]

mdu\_resp\_ready

Figure 2. Logic Symbol

Table 2. Signal Definition

| **Signal Name** | **I/O Type** | | **Description** |
| --- | --- | --- | --- |
| **Processor Signals** | | | |
| core\_clk | I | | System bus clock |
| core\_reset\_n | I | | System bus reset |
| mdu\_kill | I | | Indicates that system discards all on-going or non-confirmed request |
| **MDU Request Signals** | | | |
| mdu\_req\_valid | I | | Indicates an operation start, if MDU not busy |
| mdu\_req\_func[3:0] | I | | Indicates which kinds of operation that MDU needs to perform |
| mdu\_req\_tag[4:0] | I | | GPR index that MDU need to update after the operation |
| mdu\_req\_op0[XLEN-1:0] | I | | Operand 1, Multiplicand/Dividend |
| mdu\_req\_op1[XLEN-1:0] | I | | Operand 2, Multiplier/Divisor |
| mdu\_req\_ready | O | | Indicates that MDU is ready for operation (Not busy) |
| **MDU Response Signals** | | | |
| mdu\_resp\_valid | | O | Indicates that the operation of MDU is finished with valid result |
| mdu\_resp\_tag[4:0] | | O | GPR index that system must updates with valid result |
| mdu\_resp\_result[XLEN-1:0] | | O | MDU calculated result |
| mdu\_resp\_ready | | I | System can handle the response of MDU |

Table 3. MDU Function Encode

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Encode | Instruction | Encode |
| MUL | 4’h0 | DIV | 4’h4 |
| MULH | 4’h1 | DIVU | 4’h5 |
| MULHU | 4’h2 | REM | 4’h6 |
| MULHSU | 4’h3 | REMU | 4’h7 |
| MULW | 4’h8 | DIVW | 4’hC |
|  | | DIVUW | 4’hD |
| REMW | 4’hE |
| REMUW | 4’hF |

## MDU Operations

### Normal Operation

Figure 3 shows several examples of normal operation. While system asserts

mdu\_req\_valid, MDU would start an operation and MDU would not accept when any new request before operation is done.

After operation is done, MDU assert mdu\_resp\_valid to indicate that the operation is done and result is valid.

The system may not able to receive the result; therefore MDU would assert mdu\_resp\_valid and keep the result till system asserts mdu\_resp\_ready.

{signal: [
  {name: 'core_clk', wave: 'p...|....|.......|...'},
  {name: 'MDU status', wave:'2.2.|22.2|.2.2.2.|22.', data:["IDLE","BUSY","Done","IDLE","BUSY","Done","IDLE","BUSY","Done","IDLE"]},
  ['Request',
  {name: 'mdu_req_valid',    wave: '01x.|0.1x|.0..1x.|0..' },
  {name: 'mdu_req_ready',    wave: '1.0.|.1.0|...1.0.|.1.'},
   {name: 'mdu_req_tag[4:0]',   wave: 'x3x.|..4x|....5x.|...', data:["TAG","TAG","TAG"]},
  {name: 'mdu_req_func[3:0]',   wave: 'x3x.|..4x|....5x.|...', data:["Func","Func","Func"]},
   ['operand',
  {name: 'mdu_req_op0[63:0]',   wave: 'x3x.|..4x|....5x.|...', data:["Op0","Op0","Op0"]},
  {name: 'mdu_req_op1[63:0]',   wave: 'x3x.|..4x|....5x.|...', data:["Op1","Op1","Op1"]}]],
  ['Response',
  {name: 'mdu_resp_valid',    wave: '0...|10..|.1.0...|10.'},
   {name: 'mdu_resp_result[63:0]',  wave: 'x...|3x..|.4.x...|5x.', data:["result","result","result","result"]},
  {name: 'mdu_resp_tag[4:0]',   wave: 'x...|3x..|.4.x...|5x.', data:["TAG","TAG","TAG","TAG"]},
   {name: 'mdu_resp_ready',   wave: 'x...|1x..|.01x...|1x.'}]
],
 head:{
   text:'MDU Operation Example',
   tick:1,
 }
 }

Figure 3. MDU Operation Example

### Killed Operation

Figure 4 passes a simple concept; mdu\_kill can be asserted at any time and when it asserts, MDU goes to IDLE state immediately.

That means MDU drops unfinished operation or un-confirmed result and would not response to mdu\_req\_valid while mdu\_kill asserts.

{signal: [
  {name: 'core_clk', wave: 'p...|...|......|.....'},
  {name: 'MDU status', wave:'2...|.2.|.2..2.|.2.2.', data:["IDLE","BUSY","IDLE","BUSY","Done","IDLE"]},
  {name: 'mdu_kill',   wave:'010.|...|10....|..10.'},
  ['Request',
  {name: 'mdu_req_valid',    wave: '010.|.1x|x0.1x.|.0x0.' },
  {name: 'mdu_req_ready',    wave: '1...|..0|.1..0.|...1.'},
   {name: 'mdu_req_tag[4:0]',   wave: 'x3x.|.4x|...5x.|.....', data:["TAG","TAG","TAG"]},
  {name: 'mdu_req_func[3:0]',   wave: 'x3x.|.4x|...5x.|.....', data:["Func","Func","Func"]},
   ['operand',
  {name: 'mdu_req_op0[63:0]',   wave: 'x3x.|.4x|...5x.|.....', data:["Op0","Op0","Op0"]},
  {name: 'mdu_req_op1[63:0]',   wave: 'x3x.|.4x|...5x.|.....', data:["Op1","Op1","Op1"]}]],
  ['Response',
  {name: 'mdu_resp_valid',    wave: '0...|...|......|.1.0.'},
   {name: 'mdu_resp_result[63:0]',  wave: 'x...|...|......|.5.x.', data:["result","result"]},
  {name: 'mdu_resp_tag[4:0]',   wave: 'x...|...|......|.5.x.', data:["TAG","TAG"]},
   {name: 'mdu_resp_ready',   wave: 'x...|...|......|.0x..'}]
],
 head:{
   text:'MDU Kill Operation Example',
   tick:1,
 }
 }

Figure 4. MDU Killed Operation

# Micro-Architecture

## MDU Status

As shown in Figure 5, MDU is at IDLE state at the beginning. Then if MDU receive a valid request, it enters BUSY state and starts to calculate. While the calculation of MDU is done, it goes to DONE state and response valid result. When MDU is at DONE state, system can response a ready to MDU to indicate that system has received the result and that would make MDU goes to IDLE state. System can also send a new request while response a ready and MDU would start a new calculation without goes to IDLE state.

A Kill signal is equivalent to reset, they both result MDU goes to IDLE state which cancel the undone operation on MDU.

Calculate complete

mdu\_req\_valid

& ~kill

mdu\_resp\_ready

Kill

Figure 5 . MDU Status

## MDU Data Path block diagram

As shown in Figure 6, MDU has a (XLEN+RADIX) bit Adder for both multiplication and division.

At the time the multiplication begins, Reg0 contains accumulator and multiplier, and Reg1 contains multiplicand. After multiplication is done, the product is located in Reg0.

When MDU performs division, the dividend and remainder are placed in Reg0, and divisor is placed in Reg1. While division is complete, the high part of Reg0 would be remainder and low part would be quotient.

Div (done)

Div (init)

Mul (done)

Mul (init)

Reg1

Reg0

mdu\_resp\_tag[4:0]

mdu\_resp\_result[XLEN-1:0]

mdu\_req\_tag[4:0]

mdu\_req\_op1[XLEN-1:0]

mdu\_req\_op0[XLEN-1:0]

(XLEN+RADIX) bits

Adder

OP0\_Data

Pre-processing

Reg0

(2\*XLEN+RADIX) bits

Reg1

(Multiplicand/Divisor)

(XLEN+1) bits

OP1\_Data

Pre-processing

 Tag\_Reg

Accumulator (set to zero)

Multiplier (op1)

Product

Remainder (set to zero)

Dividend (op0)

Remainder

Quotient

Multiplicand (op0)

Divisor (op1)

MUL

Partial Product

Figure 6. MDU Data Path Architecture

## MDU Arithmetic Algorithms

### Multiplication Operation

MDU uses *Multiply-Accumulate operation* to perform multiplication. MDU can also speed up by examining multiplier, multiplication is completed while multiplier reaches zero.

The multiplication operation instructions of RISCV ISA are shown in Table 4

Table .RISCV Multiplication Operation Instructions

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Type** | **Function Description** |
| MUL  (encode:4’h0) | RV32/64M, Signed | MUL performs an XLEN-bit\*XLEN-bit **signed multiplication** and places the **lower** XLEN bits in the destination register.  Result [XLEN-1:0] = (op0\*op1) [XLEN-1:0] |
| MULH  (encode:4’h1) | RV32/64M, Signed | MULH performs an XLEN-bit\*XLEN-bit **signed multiplication** and places the **higher** XLEN bits in the destination register.  Result[XLEN-1:0] = (op0\*op1)[2XLEN-1:XLEN] |
| MULHU  (encode:4’h2) | RV32/64M, Unsigned | MULH performs an XLEN-bit\*XLEN-bit **unsigned multiplication** and places the **higher** XLEN bits in the destination register.  Result[XLEN-1:0] = (op0\*op1)[2XLEN-1:XLEN] |
| MULHSU  (encode:4’h3) | RV32/64M, Signed | MULHSU performs (**signed** XLEN-bit) \* (**unsigned** XLEN-bit) **multiplication** and places the **higher** XLEN bits in the destination register.  Signed(op0) multiply unsigned(op1) :  Result[XLEN-1:0] = (op0\*op1)[2XLEN-1:XLEN] |
| MULW  (encode:4’h8) | RV64M, Unsigned | MULW is only valid for **RV64**, and multiplies the **lower 32 bits** of the source registers, placing the **sign-extension** of the lower 32 bits of the result into the destination register.  Result[63:0] = (op0\*op1)[31:0] ,sign extend to 64 bits |

#### Multiply-Accumulate Operation

The basic concept of *multiply-accumulate operation* is shown below:

Figure 9 shows the flow chart of *multiply-accumulate operation* and Figure 10 performs the hardware architecture of *multiple-accumulate operation*.

Product-register initial value:

High part: accumulator (set to zero)

Low part: multiplier

NO

YES

0

1

Product [0]

Product >> 1

Product high part = Product high part + Multiplicand

Repeat XLEN times?

DONE

Figure 7. Multiply-Accumulate Operation

Multiplicand

Product

XLEN bit

Adder

Write

RShift

Figure 8. Hardware Architecture of Multiple-Accumulate Operation

Table 5 and Table 6 perform hardware operation example of 8-bit unsigned and signed multiplication using *multiply-accumulation operation* respectively.

Example: Op0 = 0100\_1011, Op1 = 1001\_0110

Table 5. Hardware operation example of 8-bit Unsigned Multiplication

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MDU Status** | **reg0: Multiplier (Product/Accumulator/Multiplier)**  **1001\_0110 = 150** | | **reg1:Multiplicand**  **0100\_1011 = 75** | **Comment** |
| IDLE | **Accumulator** | **Multiplier** | **Multiplicand** |  |
| BUSY | 0000\_0000 | 1001\_0110 | 0100\_1011 | Initial value |
| 0000\_0000 | 0100\_1011 | Accu = Accu + Multiplicand & {XLEN{Multiplier[0]}},  reg0 = reg0 >> 1 |
| 0010\_0101 | 1010\_0101 |
| 0011\_1000 | 0101\_0010 |
| 0001\_1100 | 0010\_1001 |
| 0011\_0011 | 1001\_0100 |
| 0001\_1001 | 1100\_1010 |
| 0000\_1100 | 1110\_0101 |
| DONE | 0010\_1011 | 1111\_0010 | Product = reg0 = 0010\_1011\_1111\_0010  75 x 150 = 11250 |

Table 6. Hardware operation example of 8-bit signed Multiplication

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MDU Status** | **reg0: Multiplier (Product/Accumulator/Multiplier)**  **1001\_0110 = -106**  **Transfer to unsigned = 0110\_1010** | | **reg1:Multiplicand**  **0100\_1011 = 75** | **Comment** |
| IDLE | **Accumulator** | **Multiplier** | **Multiplicand** |  |
| BUSY | 0000\_0000 | **0110\_1010** | 0100\_1011 | Initial value |
| 0000\_0000 | **0011\_0101** | Accu = Accu + Multiplicand & {XLEN{Multiplier[0]}},  reg0 = reg0 >> 1 |
| 0010\_0101 | **1001\_1010** |
| 0001\_0010 | **1100\_1101** |
| 0010\_1110 | **11100110** |
| 0001\_0111 | **0111\_0011** |
| 0011\_0001 | **0011\_1001** |
| 0011\_1110 | **0001\_1100** |
| DONE | 0001\_1111 | 0000\_1110 | Product = reg0\*(-1) = -7950  75 x (-106) = -7950 |

### Division Operation

MDU performs division by the *non-restoring division* algorithm. MDU speeds up division by checking the leading zero of operands to determine the bits that remainder/quotient register should left shift.

The division operation instructions of RISCV ISA are shown in Table 7

Table .RISCV Division Operation Instructions

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Type** | **Function Description** |
| DIV  (encode:4’h4) | RV32/64M, Signed | DIV performs **signed** integer division of XLEN bits by XLEN bits and provides XLEN bits **quotient**.  Result [XLEN-1:0] = (op0/op1) [XLEN-1:0] |
| DIVU  (encode:4’h5) | RV32/64M, Unsigned | DIVU performs **unsigned** integer division of XLEN bits by XLEN bits and provides XLEN bits **quotient**.  Result [XLEN-1:0] = (op0/op1) [XLEN-1:0] |
| REM  (encode:4’h6) | RV32/64M, Signed | REM provides the **remainder** of the **signed** integer division operation.  Result [XLEN-1:0] = (op0 % op1) [XLEN-1:0] |
| REMU  (encode:4’h7) | RV32/64M, Unsigned | REMU provides the **remainder** of the **unsigned** integer division operation.  Result [XLEN-1:0] = (op0 % op1) [XLEN-1:0] |
| DIVW  (encode:4’hC) | RV64M, Signed | DIVW performs **signed division** by divides lower 32 bits of op0 by lower 32 bits of op1 and provides a **32 bits quotient signed-extended to 64 bits.**  Result[63:0] = (op0[31:0]/op1[31:0]), sign extend to 64 bits |
| DIVUW  (encode:4’hD) | RV64M, Unsigned | DIVUW performs **unsigned division** by divides lower 32 bits of op0 by lower 32 bits of op1 and provides a **32 bits quotient signed-extended to 64 bits.**  Result[63:0] = (op0[31:0]/op1[31:0]), sign extend to 64 bits |
| REMW  (encode:4’hE) | RV64M, Signed | REMW performs **signed division** by divides lower 32 bits of op0 by lower 32 bits of op1 and provides a **32 bits remainder signed-extended to 64 bits.**  Result[63:0] = (op0[31:0] % op1[31:0]), sign extend to 64 bits |
| REMUW  (encode:4’hF) | RV64M, Unsigned | REMUW performs **unsigned division** by divides lower 32 bits of op0 by lower 32 bits of op1 and provides a **32 bits remainder signed-extended to 64 bits.**  Result[63:0] = (op0[31:0] % op1[31:0]), sign extend to 64 bits |

#### Division by Zero and Division Overflow

RISCV has specific definition of both division by zero and division overflow.

The quotient of division by zero has **all bits set**, i.e. 2XLEN- 1 for unsigned division or -1 for signed division. And the remainder of division by zero equals to dividend (OP0).

Signed division overflow occurs only when the most negative integer -2XLEN-1 dividesby -1. The quotient of signed division overflow is equal to dividend, and the remainder is equal to zero.

The summary of division by zero and signed division overflow are shown in Table 8

Table .The Summary of Division by Zero and Signed Division Overflow

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Condition** | **Dividend** | **Divisor** | **DIVU** | **REMU** | **DIV** | **REM** |
| Divide by zero | x | 0 | 2XLEN- 1 | x | -1 | x |
| Divide overflow | -2XLEN-1 | -1 | - | - | -2XLEN-1 | 0 |

#### Non-Restoring Division

RQ\_Reg << 1 bit

Note:

RQ\_Reg = Remainder/Quotient Register

RQ\_LHP = RQ\_Reg Left Half Part = Remainder

RQ\_RHP = RQ\_Reg Right Half Part = Quotient

Initial: RQ\_RHP = Dividend

RQ\_LHP = RQ\_LHP - Divisor

RQ\_LHP = RQ\_LHP + Divisor

RQ\_Reg << 1bit,

and set RQ\_Reg[LSB] = 0

RQ\_Reg << 1bit, and set RQ\_Reg[LSB] = 1

RQ\_LHP Negative?

Repeat XLEN times?

DONE

YES

No

YES

No

Figure 9. Non-Restoring Division

Figure 9 shows the flow chart of non-restoring division operation. And Figure 10 performs the hardware architecture of non-restoring division operation.

Divisor

Remainder/Quotient

XLEN bit

Adder

Write

LShift

Figure . Hardware Architecture of Non-Restoring Division Operation

Table 9 and Table 10 show the examples of 8-bit unsigned and signed non-restoring division operation respectively.

Example: Op0 = 1100\_0011, Op1 = 0010\_1011

Table . Example of 8-bit Unsigned Non-Restoring Division

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MDU**  **Status** | **Reg0: RQ\_Reg**  Op0 = 1100\_0011 = 193 | | **Reg1: Divisor**  Op1 = 0010\_1011= 43 | **Comments** |
|  | LHP(Rem) | RHP (Q) | Divisor |  |
| IDLE | - | - | - |  |
| BUSY | 0000\_0000 | 1100\_0011 | 0010\_1011 | Initial value |
| 0000\_0001 | 1000\_0110 | If (RQ\_LHP – Divisor ≧ 0)  RQ\_LHP = {(RQ\_LHP – divisor)<<1, RQ\_RHP[MSB]}  RQ\_RHP = {RQ\_RHP <<1 , 1}  Else  RQ\_LHP = {RQ\_LHP << 1, RQ\_RHP[MSB]}  RQ\_RHP = {RQ\_RHP << 1 , 0} |
| 0000\_0011 | 0000\_1100 |
| 0000\_0110 | 0001\_1000 |
| 0000\_1100 | 0011\_0000 |
| 0001\_1000 | 0110\_0000 |
| 0011\_0000 | 1100\_0000 |
| 0000\_1011 | 1000\_0001 |
| 0001\_0111 | 0000\_0010 |
| DONE | 0010\_1110 | 0000\_0100 | Quotient =RHP, Remainder = LHP >> 1 |

Table . Example of 8-bit Signed Non-Restoring Division

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MDU**  **Status** | **Reg0: RQ\_Reg**  Op0 = 1100\_0011 = -61  Transfer to unsigned = 0011\_1101 | | **Reg1: Divisor**  Op1 = 0010\_1011 = 43 | **Comments** |
|  | LHP(Rem) | RHP (Q) | Divisor |  |
| IDLE | - | - | - |  |
| BUSY | 0000\_0000 | 0011\_1101 | 0010\_1011 | Initial value |
| 0000\_0000 | 0111\_1010 | If (RQ\_LHP – Divisor ≧ 0)  RQ\_LHP = {(RQ\_LHP – divisor)<<1, RQ\_RHP[MSB]}  RQ\_RHP = {RQ\_RHP <<1 , 1}  Else  RQ\_LHP = {RQ\_LHP << 1, RQ\_RHP[MSB]}  RQ\_RHP = {RQ\_RHP << 1 , 0} |
| 0000\_0000 | 1111\_0100 |
| 0000\_0001 | 1110\_1000 |
| 0000\_0011 | 1101\_0000 |
| 0000\_0111 | 1010\_0000 |
| 0000\_1111 | 0100\_0000 |
| 0001\_1110 | 1000\_0000 |
| 0011\_1101 | 0000\_0000 |
| DONE | 0010\_0100 | 0000\_0001 | Quotient =RHP\*(-1) = -1 ,  Remainder = (LHP >> 1)\*(-1) = -17 |

# Design Configuration

Refer to Table 1 in Page 3.

# Programming Model

N/A