XMC1000

32-bit Microcontroller Series for Industrial Applications

Math Coprocessor (MATH)

AP32307

Application Note

About this document

Scope and purpose

This document describes how to use the MATH Coprocessor for the XMC 32-bit Microcontroller. The document includes code snippets and examples for a variety of use cases.

Intended audience

This document is intended for engineers who are developing applications that require math-intensive computations with the XMC Microcontroller series.

Applicable Products

- XMC130x
- XMC140x
- DAVE ™

References

Infineon: Example code: http://www.infineon.com/XMC1000 Tab: Documents

Infineon: XMC Lib, http://www.infineon.com/DAVE

Infineon: DAVE™, http://www.infineon.com/DAVE

Infineon: XMC Reference Manual, http://www.infineon.com/XMC1000 Tab: Documents

Infineon: XMC Data Sheet, http://www.infineon.com/XMC1000 Tab: Documents

Table of Contents

1 MATH Coprocessor Overview

The Math Coprocessor (MATH) module provides assistance for math-intensive computations. The module comprises of two independent sub-blocks which are executed in parallel to the CPU core. The two sub-blocks are:

- A 32-bit Divider Unit (DIV) for signed and unsigned division functions
- A 24-bit CORDIC (COrdinate Rotation DIgital Computer) for trigonometric, linear and hyperbolic functions

1.1 Features

The MATH module includes the following features:

- Divide function with operand pre-processing and result post-processing
- CORDIC Coprocessor for computation of trigonometric, hyperbolic and linear functions
- Support kernel clock to interface clock ratio 2:1 for faster execution
- Support result chaining between the Divider Unit and CORDIC Coprocessor

1.2 MATH Library

The MATH library is a collection of Application Programming Interfaces (APIs) to compute common mathematical operations such as division, modulus and trigonometric functions. The APIs configure the respective registers of the MATH sub-blocks to perform the requested calculations. The library is provided as part of the XMC Library (XMC Lib) from Infineon.

The APIs provided in the MATH library can be categorized as Blocking and Non-blocking. The blocking APIs poll for the result by reading the result register. This adds wait states to the bus until the result is ready. While waiting for the result, all other operations are blocked, hence the name.

The non-blocking APIs start the desired calculations and then control is returned to the calling thread. This allows other operations to continue. The user can check if a calculation has ended by polling the busy flag of the MATH Coprocessor. When the busy flag is cleared, the user can read the calculation result by calling the GetResult APIs.

Note: The occurence of interrupts during the execution of non-blocking APIs may lead to erroneous results. For example, the execution of a divide instruction ('/') in an interrupt service routine during the execution of a non-blocking API may give erroneous results.

2 Division Operation

DIVS supports the truncated division operation, which is the ISO C99 standard and the most popular choice among modern processors:

- q = D divide by d
- r = D modulus by d

“D” is the dividend (DVD register)

“d” is the divisor (DVS register)

“q” is the quotient (QUOT register)

“r” is the remainder (RMD register)

2.1 Configuration for Signed or Unsigned division operation

Configuration is via the DIVCON.USIGN bit:

- DIVCON.USIGN = 0 Signed division
- DIVCON.USIGN = 1 Unsigned division

2.2 Configuration for starting the division operation

There are two methods for starting the division operation. Either:

1. Set the ST bit
2. Or have the division operation start automatically by loading a value into the DVS register

Starting the division operation by setting the ST bit

```
MATH->DIVCON = (1<<MATH_CON_ST_MODE_Pos);     // DIVCON.STMODE = 1
                                  // Calculation start when ST bit is set
MATH->DVD = 0x12345678;                 // Load the dividend value
MATH->DVS = 0x11223344;                 // Load the divisor value
MATH->DIVCON |= (1<<MATH_CON_ST_Pos);  // DIVCON.ST = 1
                                       // ST bit is set. The division begins
```

Starting the division operation automatically

```
MATH->DIVCON = (0<<MATH_CON_ST_MODE_Pos);     // DIVCON.STMODE = 0
                            // Calculation start when write to DVS register
MATH->DVD = 0x12345678;    // Load the dividend value
MATH->DVS = 0x11223344;    // Load the divisor value, the division begin
```

2.3 Poll for the result to be ready

The division operation takes 35 kernel clock cycles. The BSY flag is set to 1 when the division operation starts.

On completion, the quotient and remainder values are available in the QUOT and RMD registers, and the BSY flag is cleared.

- DIVST.BSY = 0 DIV is not running any division operation
- DIVST.BSY = 1 DIV is still running a division operation

```
// Insert code for division operation to start
while(MATH->DIVST);       // Wait until DIV is ready (Not busy)
// Insert code to read out result
```

2.4 Generate an Interrupt when result is ready

When the division is finished, the Divider event flag EVFR.DIVEOC is set. This can trigger an interrupt request to the NVIC by enabling the EVIER.DIVEOCIEN bit. This event flag can only be cleared by a software write to the EVFCR.DIVEOCC bit:

```
MATH->EVIER = (1<<MATH_EVIER_DIVEOCIEN_Pos); // EVIER.DIVEOCIEN = 1
             // End of divider calculation interrupt generation is enabled
NVIC_EnableIRQ(7);                      // Enabled MATH interrupt node
// Insert code for division operation to start
.....
// Inside the MATH ISR
MATH->EVFCR = (1<<MATH_EVFCR_DIVEOCC_Pos);          // EVFCR.DIVEOCC = 1
                          // Clear Divider end of calculation flag in EVFR
```

2.5 Divide by Zero Error

If a division operation is started with the divisor value equal to 0, the EVFR.DIVERR flag is set. The interrupt request to the NVIC can be generated by enabling it with EVIER.DIVERRIEN. This event flag can only be cleared by a software write to the EVFCR.DIVERRC bit.

```
MATH->EVIER = (1<<MATH_EVIER_DIVERRIEN_Pos); // EVIER.DIVERRIEN = 1
                          // Divider error interrupt generation is enabled
NVIC_EnableIRQ(7);        // Enabled MATH interrupt node
// Insert code for division operation to start
.....

// Inside the DIV_ERROR ISR
MATH->EVFCR = (1<<MATH_EVFCR_DIVERRC_Pos);          // EVFCR.DIVERRC = 1
                          // Clear the Divider error event flag in EVFR
```

2.6 Using XMC Lib for DIV operations

The MATH Library provides alternate implementations of the ARM Embedded Application Binary Interface (AEABI) functions for division and modulus operations. These alternate implementations use the DIV sub-block to perform the operations. The following examples demonstrate their usage:

Blocking division operation

```
uint32_t a = 5000;
uint32_t b = 250;
uint32_t c = a / b;
```

Blocking modulus operation

```
uint32_t a = 5000;
uint32_t b = 240;
uint32_t c = a % b;
```

In both examples above, the '/' and '%' operators are automatically recognized and the respective AEABI functions are called to perform the operations using the DIV sub-block.

Note: Only signed and unsigned integer division and modulus operations are supported by the MATH Library.

Non-blocking division operation

```
/* variable initialization */
uint32_t calculation_dividend = 5000;
uint32_t calculation_divisor = 250;
uint32_t calculation_result;
/* unsigned division calculation */
XMC_MATH_DIV_UnsignedDivNB(calculation_dividend,calculation_divisor);
while(XMC_MATH_DIV_IsBusy()); // wait for calculation to end
calculation_result = XMC_MATH_DIV_GetUnsignedDivResult();
```

In the example above, the *XMC_MATH_DIV_IsBusy()* API is used to check for the end of calculation. Alternatively, the user can also perform other operations and read the CORDIC calculation result only after a determined number of clock cycles or use an interrupt. The following example demonstrates the usage of an interrupt.

Non-blocking modulus operation

```
/* variable initialization */
uint32_t calculation_dividend = 5000;
uint32_t calculation_divisor = 240;
uint32_t calculation_result;
/* configure DIV end-of-calculation interrupt */
XMC_MATH_EnableEvent(XMC_MATH_EVENT_DIV_END_OF_CALC);
NVIC_SetPriority(MATH0_0_IRQn,3);
NVIC_EnableIRQ(MATH0_0_IRQn);
/* unsigned modulus calculation */
XMC_MATH_DIV_UnsignedModNB(calculation_dividend,calculation_divisor);

/* MATH Interrupt Handler */
void MATH0_0_IRQHandler(void)
{
  uint32_t calculation_result;

  XMC_MATH_ClearEvent(XMC_MATH_EVENT_DIV_END_OF_CALC);
  calculation_result = XMC_MATH_DIV_GetUnsignedModResult();
}
```

2.7 DIVS Benchmarking Results

The performance of the Divider is evaluated by benchmarking the execution time of a division operation running on the MATH Library against that of a similar operation running on a standard C library. The execution time is measured in terms of the number of MCLK cycles.

The conditions for the benchmarking are as follows:

- Execution time refers to complete function execution, inclusive of co-processor configuration, writing of operands and state checking.
- The ratio of PCLK to MCLK is 2:1.
- Compilers from Infineon, Keil™ and IAR were used.

Division Operation

Table 1 Benchmarking results for division operation

Compiler	Number of MCLK cycles	
	With MATH Library	With Standard C Library
IAR EWARM v7.10	99	712
Keil™ µVision v5.10	95	230
DAVE™ v3.1.10	114	415

From the benchmarking results, a division operation with the MATH library can be up to 7 times faster than a similar operation with the standard C library.

3 CORDIC

The CORDIC algorithm is a useful convergence method for 24-bit computation of trigonometric (circular), linear (multiply-add), hyperbolic and related functions. It allows performance of vector rotation not only in the Euclidian plane, but also in the Linear and Hyperbolic planes.

CORDX, CORDY, and CORDZ are Data registers which are used to initialize the X, Y and Z parameters.

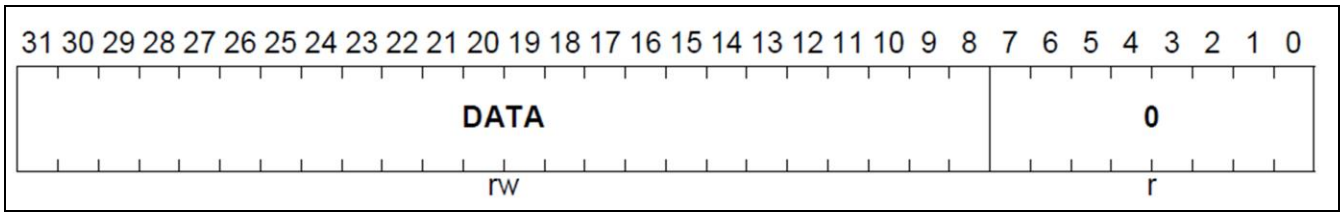

Figure 1 CORDIC data register structure

CORRX, CORRY, and CORRZ are Result registers from the CORDIC calculation.

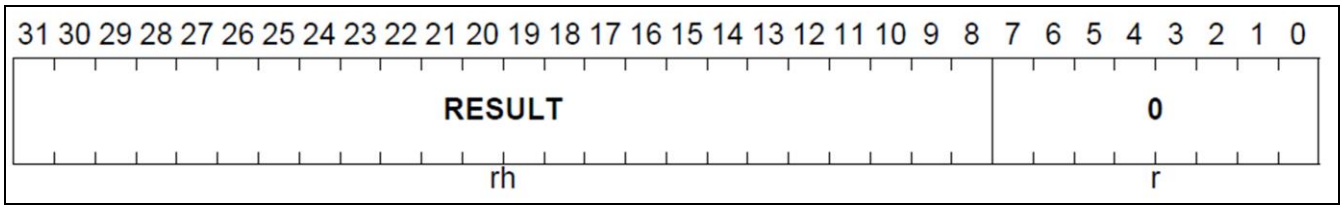

Figure 2 CORDIC result register structure

3.1 Configuration for CORDIC operation

Table 2 gives an overview of the different CORDIC operating modes.

X, Y and Z represent the initial data and X_{final}, Y_{final} & Z_{final} represent the final result data when the CORDIC computation is completed.

Table 2 Operating Modes of CORDIC

Function	Rotation Mode	Vectoring Mode
Circular	$X_{final} = K[X \cos(Z) - Y \sin(Z)] / MPS$ $Y_{final} = K[Y \cos(Z) + X \sin(Z)] / MPS$ $Z_{final} = 0$ $K \approx 1.646760258121$	$X_{final} = K \sqrt{(X^2+Y^2)} / MPS$ $Y_{final} = 0$ $Z_{final} = Z + \text{atan}(Y / X)$ $K \approx 1.646760258121$
Linear	$X_{final} = X / MPS$ $Y_{final} = [Y + X Z] / MPS$ $Z_{final} = 0$	$X_{final} = X / MPS$ $Y_{final} = 0$ $Z_{final} = Z + Y / X$

Function	Rotation Mode	Vectoring Mode
Hyperbolic	X_{final} = k[X cosh(Z) + Y sinh(Z)] / MPS Y_{final} = k[Y cosh(Z) + X sinh(Z)] / MPS Z_{final} = 0 k ≈ 0.828159360960	X_{final} = k sqrt(X^2-Y^2) / MPS Y_{final} = 0 Z_{final} = Z + atanh(Y / X) k ≈ 0.828159360960

The different modes are configured via the ROTVEC and MODE fields in the CON control register.

- CON.ROTVEC = 0 Vectoring Mode
- CON.ROTVEC = 1 Rotation Mode
- CON.MODE = 00b Linear Mode
- CON.MODE = 01b Circular Mode
- CON.MODE = 11b Hyperbolic Mode

The X and Y Magnitude Prescaler (MPS) prevents the result data from overflowing. At the end of calculation, the computed values of X and Y are each divided by the MPS factor to yield the final result.

Note: Refer to the appendix for other mathematical calculations supported by CORDIC.

3.2 Configuration for starting CORDIC operation

There are two methods to start the CORDIC operation:

- Set the ST bit
- Or have the operation automatically start by loading a value into the CORDX register

Starting the CORDIC operation by setting the ST bit

```
MATH->CON = (1<<MATH_CON_ST_MODE_Pos); // CON.STMODE = 1;
                          // Calculation start when ST bit is set to 1
MATH->CORDZ = 0x12345600;
MATH->CORDY = 0x11223300;
MATH->CORDX = 0x33221100;
MATH->CON |= (1<<MATH_CON_ST_Pos);           // CON.ST = 1;
                          // Start the CORDIC operation
```

Starting the CORDIC operation automatically

```
MATH->CON = (0<<MATH_CON_ST_MODE_Pos); // CON.STMODE = 0;
                          // Calculation start with a write to CORDX
MATH->CORDZ = 0x12345600;
MATH->CORDY = 0x11223300;
MATH->CORDX = 0x33221100; // Load CORDX value and start CORDIC operation
```

3.3 Poll for the result to be ready

The CORDIC operation takes 62 kernel clock cycles. The BSY flag is set when operation starts.

On completion, the BSY flag is cleared.

- STATC.BSY = 0 CORDIC is not computing any calculation
- STATC.BSY = 1 CORDIC is still computing a calculation

```
// Insert code for CORDIC to start
while((MATH->STATC)&(1<<MATH_STATC_BSY_Pos));
                                    // wait until CORDIC is ready (Not busy)
// Insert code to read out result
```

3.4 Generate an Interrupt on completion

At the end of the CORDIC computation, the event flag EVFR.CDEOC is set. An interrupt request to the NVIC can be triggered by enabling the EVIER.CDEOCIEN bit. This event flag can only be cleared by a software write to the EVFCR.CDEOCC bit.

```
MATH->EVIER = (1<<MATH_EVIER_CDEOCIEN_Pos);  // EVIER.CDEOCIEN = 1
             // End of CORDIC calculation interrupt generation is enabled
NVIC_EnableIRQ(7);         // Enable MATH interrupt node
// Insert code for CORDIC to start
……..

// Inside the MATH ISR
MATH->EVFCR = (1<<MATH_EVFCR_CDEOCC_Pos);    // EVFCR.CDEOCC = 1
                    // Clear CORDIC end of calculation event flag in EVFR
```

3.5 Using XMC Lib for CORDIC calculation

The MATH Library supports the following CORDIC calculations:

- Trigonometric: Sin, Cos, Tan, Atan
- Hyperbolic: Sinh, Cosh, Tanh

When using the XMC Lib APIs for calculations where the input data is an angle, it is essential that the input angle is converted using the following equation:

$$Input\_angle = (angle\_in\_rad) * 8388608 / \pi$$

For example, to calculate cos(π/6), the input angle is:

$$Input\_angle = (\pi/6) * 8388608 / \pi = 1398101 \text{ or } 0x155555$$

Example using Blocking APIs

```
/* variable initialization */
XMC_MATH_Q0_23_t angle = 0x2AAAAA; // (pi/3)*8388608/pi
XMC_MATH_Q0_23_t calculation_result;
/* cosine of angle calculation */
calculation_result = XMC_MATH_CORDIC_Cos(angle);
```

Example using Non-Blocking APIs

```
/* variable initialization */
XMC_MATH_Q0_23_t angle = 0x155555; // (pi/6)*8388608/pi
XMC_MATH_Q0_23_t calculation_result;
/* cosine of angle calculation */
XMC_MATH_CORDIC_CosNB(angle);
while(XMC_MATH_CORDIC_IsBusy()); // wait for calculation to end
calculation_result = XMC_MATH_CORDIC_GetCosResult();
```

In the non-blocking API example above, the *XMC_MATH_CORDIC_IsBusy()* API is used to check for the end of the calculation. The user can instead perform other operations and read the CORDIC calculation result only after a determined number of clock cycles or use an interrupt. The following example demonstrates the use of an interrupt.

Example using Non-Blocking API and Interrupt

```
/* variable initialization */
XMC_MATH_Q0_23_t angle = 0x860A91; // (pi/3)*8388608/pi
/* configure CORDIC end-of-calculation interrupt */
XMC_MATH_EnableEvent(XMC_MATH_EVENT_CORDIC_END_OF_CALC);
NVIC_SetPriority(MATH0_0_IRQn,3);
NVIC_EnableIRQ(MATH0_0_IRQn);
/* cosine of angle calculation */
XMC_MATH_CORDIC_CosNB(angle);
/* MATH Interrupt Handler */
void MATH0_0_IRQHandler(void)
{
```

```
  XMC_MATH_Q0_23_t calculation_result;

  XMC_MATH_ClearEvent(XMC_MATH_EVENT_CORDIC_END_OF_CALC);
  calculation_result = XMC_MATH_CORDIC_GetCosResult();
}
```

3.5.1 Calculating exp(z)

It is known that:

$$exp(z) = sinh(z) + cosh(z)$$

The following example demonstrates how exp(0.5) can be calculated using the XMC Lib.

```
 /* variable initialization */
XMC_MATH_Q0_23_t angle = 0x145F30; // 0.5*8388608/pi
XMC_MATH_Q0_23_t calculation_result;
/* hyperbolic sine of angle calculation */
calculation_result = XMC_MATH_CORDIC_Sinh(angle);
/* hyperbolic cosine of angle calculation (also final result) */
calculation_result += XMC_MATH_CORDIC_Cosh(angle);
```

Although the MATH Library supports only the abovementioned calculations, the CORDIC sub-block is capable of computing many other calculations. These calculations can be performed by manually configuring the registers of CORDIC. Refer to Figure 6 for a more complete view on the different computations that can be performed with CORDIC.

3.6 CORDIC Examples

This section provides some CORDIC use-cases.

3.6.1 Calculating Vector Magnitude and Angle

The following example illustrates the use of CORDIC in the Circular Vectoring Mode for the calculation of the magnitude and angle of two vectors.

Table 3 CORDIC Circular Vectoring Mode

Function	Vectoring Mode
Circular	X_{final} = K sqrt(X^2+Y^2) / MPS Y_{final} = 0 Z_{final} = Z + atan(Y / X) K ≈ 1.646760258121

```
MATH->CON = 0x0002;        // MODE = 01b, Circular Mode
                           // ROTVEC = 0, Vectoring Mode
                           // ST_MODE = 0, Auto start when CORDX is written

MATH->CORDZ = 0;                  // Load the initial angle value
MATH->CORDY = (vector2<<8);       // Load the magnitude of vector 2
MATH->CORDX = (vector1<<8);       // Load the magnitude of vector 1
                                  // CORDIC will automatically start
while((MATH->STATC)&(1<<MATH_STATC_BSY_Pos));
                                  // wait until CORDIC is ready (Not busy)
Result_Mag = MATH->CORRX;         // Read out the result
Result_Ang = MATH->CORRZ;
```

3.6.2 Calculating ln(x)

It is known that:

$$ln(x) = 2*atanh[(x-1)/(x+1)]$$

CORDIC can be used in the Hyperbolic Vectoring mode for the calculation above by setting the initial input data as follows:

$X = x + 1$

$Y = x - 1$

The following example illustrates the calculation of *ln(variable_x)* using CORDIC.

```
MATH->CON = 0x0006;        // MODE = 11b, Hyperbolic Mode
                           // ROTVEC = 0, Vectoring Mode
                           // ST_MODE = 0, Auto start when CORDX is written

MATH->CORDZ = 0;                  // Load the initial angle value
MATH->CORDY = ((variable_x-1)<<8);      // Load (x-1)
MATH->CORDX = ((variable_x+1)<<8);      // Load (x+1)
                                  // CORDIC will automatically start
while((MATH->STATC)&(1<<MATH_STATC_BSY_Pos));
                                  // wait until CORDIC is ready (Not busy)
Result = (MATH->CORRZ>>8);        // Read out the result of atanh
```

```
Result = 2*Result;                  // final result is scaled by 8388608/pi
```

Note: The result of this calculation has a scaling of 8388608/π.

3.6.3 Calculating sqrt(x)

It is known that:

$$sqrt(x) = sqrt[(x+0.25)^2-(x-0.25)^2]$$

CORDIC can be used in the Hyperbolic Vectoring mode for the calculation above by setting the initial input data as follows:

$X = (x + 0.25) / k$

$Y = (x - 0.25) / k$ *where* $k = 0.82815936960$

The user should ensure that the input data lie within the domain of convergence, meaning $atanh|Y/X| \leq 1.11$ *radians*.

The following example demonstrates the square root calculation of a Q1.8 number. The calculated result is a Q5.12 number.

```
#define GAIN 0x0135       // (2^8)/0.82815936960

uint32_t number = 0x01C0;

MATH->CON = 0x0006;        // MODE = 11b, Hyperbolic Mode
                           // ROTVEC = 0, Vectoring Mode
                           // ST_MODE = 0, Auto start when CORDX is written

MATH->CORDZ = 0;                  // Load the initial angle value
MATH->CORDY = ((GAIN*(number-0x40))<<8);       // Load (x-0.25)/k
MATH->CORDX = ((GAIN*(number+0x40))<<8);       // Load (x+0.25)/k
                                  // CORDIC will automatically start
while((MATH->STATC)&(1<<MATH_STATC_BSY_Pos));
                                  // wait until CORDIC is ready (Not busy)
Result = (MATH->CORRX>>8);              // Read out the result
```

3.7 CORDIC Benchmarking Results

The performance of the CORDIC co-processor is evaluated by benchmarking the execution time of a cosine calculation running on the MATH Library against that of a similar operation running on a standard C library. The execution time is measured in terms of the number of MCLK cycles.

The conditions for the benchmarking are as follows:

- Execution time refers to complete function execution, inclusive of co-processor configuration, writing of operands and state checking.
- The ratio of PCLK to MCLK is 2:1.
- Compilers from Infineon, Keil™ and IAR were used.

Table 4 Benchmarking results for cosine calculation

Compiler	Number of MCLK cycles	
	With MATH Library	With Standard C Library
IAR EWARM v7.10	234	4574
Keil µVision v5.10	238	6514
DAVE™ v3.1.10	258	9832

From the benchmarking results, a cosine calculation with the MATH library can be up to 38 times faster than a similar operation with the standard C library.

4 Result Chaining

The MATH Coprocessor supports result chaining between the DIV and CORDIC modules. The DIV result can be passed to the input of the CORDIC data register. Similarly, the CORDIC result can also be passed to the input of the DIV DVD and DVS registers.

GLBCON.DVDRC and GLBCON.DVSRC

- 000b No result chaining
- 001b QUOT register is the input to DIV
- 010b RMD register is the input to DIV
- 011b CORRX register is the input to DIV
- 100b CORRY register is the input to DIV
- 101b CORRZ register is the input to DIV

GLBCON.CORDXRC, GLBCON.CORDYRC and GLBCON.CORDZRC

- 00b No result chaining
- 01b QUOT register is the input to DIV
- 10b RMD register is the input to DIV

4.1 Data Compatibility between 32-bit DIV and 24-bit CORDIC

The data and result register for the DIV are assigned to bits[0 to 31].

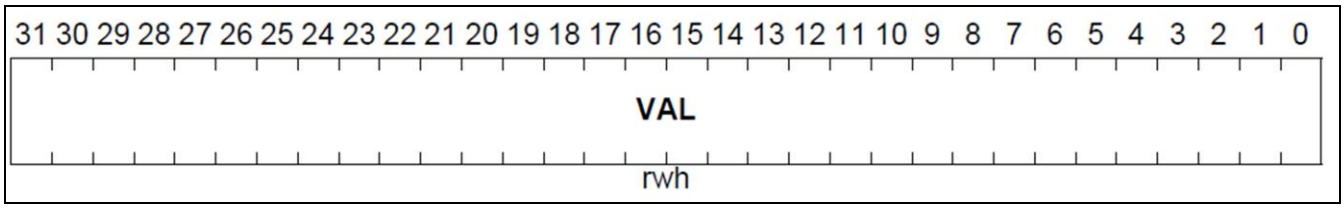

Figure 3 DIV data and result register structure

The data and result register for the CORDIC are assigned to bits[8 to 31].

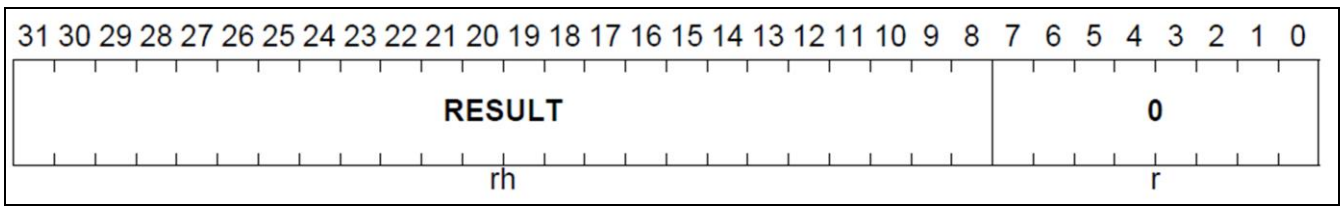

Figure 4 CORDIC data and result register structure

4.2 Transfer of data from DIV to CORDIC

The DIV's quotient final value can be left-shifted by 8 to fit into the CORDIC data register format.

- DIVCON.QSCNT Number of bits the quotient is shifted after the division
- DIV.QSDIR = 0 Left shift
- DIV.QSDIR = 1 Right shift

4.3 Transfer of data from CORDIC to DIV

The DIV's final divisor value can be right-shifted by 8 when it is updated by the CORDIC result register.

- DIVCON.DVSSRC Number of bits the divisor is shifted right before the division
- DIV.QSDIR = 0 Left-shift

4.4 Handling Busy Flags during Result Chaining

When the DIV result is chained to the CORDIC's CORDX, if CON.ST_MODE = 0 the start of the DIV calculation sets the DIV's busy flag and also sets the CORDIC's busy flag.

After completion of the DIV operation, the result is written into the DIV's register and CORDX. The DIV's busy flag is not immediately cleared. Instead, both the DIV and CORDIC busy flags are cleared after the CORDIC calculation is completed.

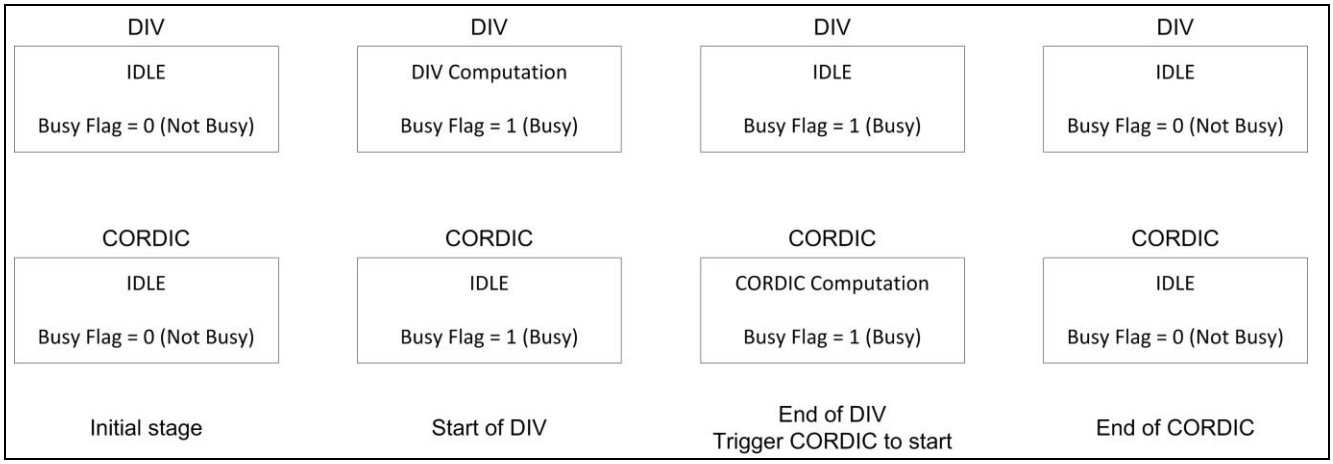

Figure 5 Busy flags during Result Chaining

The rule described above is applied in the other direction when the CORDIC result is chained to DIV's DVS register and DIVCON.STMODE = 0.

4.5 Result Chaining Example

The following example illustrates the use of result chaining by updating the input data of DIV's divisor using the CORDIC's CORRX output result.

Result Chaining

```
/* DVSRC = 011b, DVS result will be updated when CORRX has new result */
MATH->GLBCON = XMC_MATH_DIV_DVSRC_CORRX_IS_SOURCE;
/* STMODE = 0, Auto start when DVS is written */
/* DVSSRC = 8, DVS value right shifted by 8 */
MATH->DIVCON = (0<<MATH_DIVCON_STMODE_Pos)+(8<<MATH_DIVCON_DVSSRC_Pos);
/* Preload the dividend value first */
MATH->DVD = 0x12345678;
/* MODE = 01b, Circular Mode */
/* ROTVEC = 0, Vectoring Mode */
/* ST_MODE = 0, Auto start when CORDX is written */
MATH->CON = XMC_MATH_CORDIC_OPERATING_MODE_CIRCULAR +
XMC_MATH_CORDIC_ROTVEC_MODE_VECTORING + (0<<MATH_CON_ST_MODE_Pos);
/* Load the initial angle value */
MATH->CORDZ = 0;
/* Load the magnitude of Vector2 */
MATH->CORDY = (0x123456<<8);
/* Load the magnitude of Vector1 */
MATH->CORDX = (0x112233<<8);

// CORDIC will automatically start
// ……..
// CORDIC result to DVS will start DIV

while(XMC_MATH_DIV_IsBusy()); // wait until DIV is ready (Not busy)
Result = MATH->QUOT;      // Read out the result
```

5 Appendix

Function	Rotation Mode	Vectoring Mode
	d_i = sign (z_i), $z_i \rightarrow 0$	d_i = -sign (y_i), $y_i \rightarrow 0$
Circular m = 1 e_i = atan(2^{-i})	X_{final} = K[X cos(Z) - Y sin(Z)] / MPS Y_{final} = K[Y cos(Z) + X sin(Z)] / MPS Z_{final} = 0 where K ≈ 1.646760258121	X_{final} = K sqrt(X^2+Y^2) / MPS Y_{final} = 0 Z_{final} = Z + atan(Y / X) where K ≈ 1.646760258121
	For solving cos(Z) and sin(Z), set X = 1 / K, Y = 0. Useful domain: Full range of X, Y and Z supported due to pre-processing logic.	For solving magnitude of vector (sqrt(x^2+y^2)), set X = x / K, Y = y / K. Useful domain: Full range of X and Y supported due to pre- and post-processing logic. For solving atan(Y / X), set Z = 0. Useful domain: Full range of X and Y, except X = 0.
	Relationships: tan(v) = sin(v) / cos(v)	Relationships: acos(w) = atan[sqrt(1-w^2) / w] asin(w) = atan[w / sqrt(1-w^2)]
Linear m = 0 e_i = 2^{-i}	X_{final} = X / MPS Y_{final} = [Y + X Z] / MPS Z_{final} = 0	X_{final} = X / MPS Y_{final} = 0 Z_{final} = Z + Y / X
	For solving $X \cdot Z$, set Y = 0. Useful domain: |Z| ≤ 2.	For solving ratio Y / X, set Z = 0. Useful domain: |Y / X\] ≤ 2, X > 0.
Function	**Rotation Mode**	**Vectoring Mode**
Hyperbolic m = -1 e_i = atanh(2^{-i})	X_{final} = k[X cosh(Z) + Y sinh(Z)] / MPS Y_{final} = k[Y cosh(Z) + X sinh(Z)] / MPS Z_{final} = 0 where k ≈ 0.828159360960	X_{final} = k sqrt(X^2-Y^2) / MPS Y_{final} = 0 Z_{final} = Z + atanh(Y / X) where k ≈ 0.828159360960
	For solving cosh(Z) and sinh(Z) and e^Z, set X = 1 / k, Y = 0. Useful domain: |Z| ≤ 1.11rad, Y = 0.	For solving sqrt(x^2-y^2), set X = x / k, Y = y / k. Useful domain: |y| < |x|, X > 0. For solving atanh(Y / X), set Z = 0. Useful domain: |atanh(Y / X)| ≤ 1.11rad, X > 0.
	Relationships: tanh(v) = sinh(v) / cosh(v) e^v = sinh(v) + cosh(v) w^t = $e^{t\ ln(w)}$	Relationships: ln(w) = 2 atanh[(w-1) / (w+1)] sqrt(w) = sqrt((w+0.25)^2-(w-0.25)^2) acosh(w) = ln[w+sqrt(1-w^2)] asinh(w) = ln[w+sqrt(1+w^2)]

Figure 6 CORDIC Coprocessor Operating Modes and Corresponding Result Data

6 Revision History

Current Version is V1.0, 2015-07

Page or Reference	Description of change
V1.0, 2015-07	
	Initial Version

www.infineon.com

Edition 2015-07

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference

AP32307

Legal Disclaimer

THE INFORMATION GIVEN IN THIS APPLICATION NOTE (INCLUDING BUT NOT LIMITED TO CONTENTS OF REFERENCED WEBSITES) IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office. Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.