STM32G4 - CORDICコプロセッサ

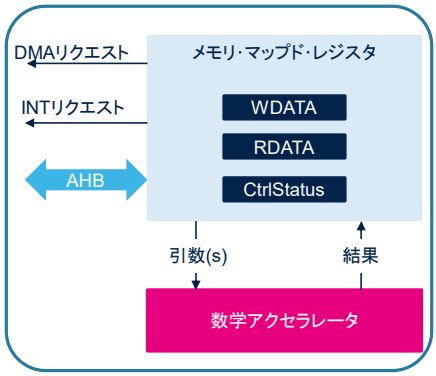
CORDIC

1.0版

life.augmented

こんにちは、STM32G4 Cordicコプロセッサブロックのプレゼンテーションへようこそ。
これは、三角関数の計算を加速するために使用されるブロックの主な機能をカバーします。

概要



- Cordic（COordinate Rotation DIgital Computer）数学関数のハードウェア·アクセラレーションを提供
  - 三角関数、対数、平方根
- ソフトウェアとの透過的な同期
  - 待機時は、結果が使用可能になるまでAHBスレーブ·インタフェースに挿入
  - 割込みチャネルとDMAチャネルも使用可能

アプリケーション側の利点

- 固定小数点
- パイプライン·オペレーション
- CPUをオフロードし、消費電流を削減し、パフォーマンスを向上

life.augmented

CORDICコプロセッサは、特定の数学関数（特に三角関数）のハードウェアアクセラレーションを提供し、モーター制御、計測、信号処理、および他の多くのアプリケーションで一般的に使用されます。
ソフトウェア実装と比較してこれらの関数の計算を高速化し、より低い動作周波数を可能にするか、他のタスクを実行するためにプロセッササイクルを解放します。
CordicブロックはAHBスレーブであり、Cortex-M4が結果を要求すると、操作が完了するまで待機状態を挿入します。したがって、入出力ドライバは必要ありません。
もう1つのアプローチは、Cordic-M4がCordic計算の進行中に他の処理を処理できるようにすることです。この場合、割り込み要求は結果が利用可能であることを示します。
DMAチャネルを実装して、メモリから引数を提供し、結果をメモリに書き込むことができます。
Cordicブロックはパイプライン処理をサポートしています。現在の引数を使用した計算の進行中に次の引数を提供できます。
Cordicブロックは固定小数点演算アクセラレータであることに注意してください。

Cordicの基本

- Cordic（COordinate Rotation DIgital Computer）は三角関数と双曲線関数を評価するための低コストの連続近似アルゴリズム
  - 三角関数（サーキュラ）モードでは、正弦と余弦は、回転角度の累積合計が入力角度に等しくなるまで、単位ベクトル[1, 0]を減少角度で回転させることによって決定
  - 逆に、ベクトル[x,y]の角度は、アークタンジェント（y/x）に対応し、[x,y]を連続的に減少する角度で回転させて単位ベクトル[1,0]を求める
- CORDICアルゴリズムは、双曲線に沿ったステップで連続する円形回転を置き換えることによって双曲線関数（sinh、cosh、atanh）を計算するためにも使用可能

life.augmented

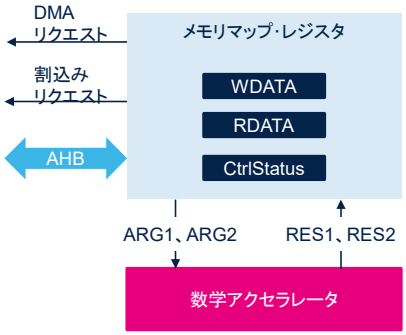
“coordinate rotation digital computer”を意味するCORDICは、ハードウェア効率が高い、繰り返し演算を使い基本関数を計算が可能です。
三角関数（サーキュラ）モードでは、角度θの正弦と余弦は、回転角度の累積合計が入力角度と等しくなるまで、減少する角度で単位ベクトル[1、0]を回転させることによって決定されます。回転したベクトルのxおよびyデカルト成分は、それぞれθのコサインとサインに対応します。
逆に、逆正接（y/x）に対応するベクトル[x、y]の角度は、[x、y]を次第に減少する角度で回転させて、単位ベクトル[1、0]を取得することによって決定されます。
回転角度の累積合計は、元のベクトルの角度を示します。
CORDICアルゴリズムは、双曲線に沿ったステップで連続する円回転を置き換えることにより、双曲線関数（sinh、cosh、atanh）の計算にも使用できます。

Cordic関数

- STM32G4 Cordicユニットは次の関数をサポート:
  - コサイン（cos θ）
  - サイン（sin θ）
  - 位相（atan2 y,x）
  - 絶対値（モジュラス）（$\sqrt{x^2+y^2}$）
  - アークタンジェント（tan-1 x）
  - 双曲線サイン（sinh x）
  - 双曲線コサイン（cosh x）
  - 双曲線アークタンジェント（tanh-1 x）
  - 自然対数（ln x）
  - 平方根（$\sqrt{x}$）
- 選択は、CORDIC_CRレジスタのFUNCフィールドで行われる

サポートされている10個の数学関数のリストを示しています。
コプロセッサを使用するときの最初のステップは、CORDIC_CSR
レジスターのFUNCフィールドを適宜プログラミングすることにより、
必要な機能を選択することです。
したがって、一度にアクティブになる機能は1つだけです。

関数パラメータ



- 引数
  - Cordic関数は1つまたは2つの引数を取ることができる（ARG1、ARG2）
  - ドメイン（入力範囲）は関数によって制限されるか、それ以外の場合はコード・アルゴリズムの収束の範囲によって制限
- 答え
  - Cordic関数は1つまたは2つの結果を出力（RES1、RES2）

life.augmented

いくつかの関数は、ARG1とARG2の2つの入力引数を取り、RES1とRES2の2つの結果を同時に生成します。
これはCordicアルゴリズムの副作用であり、2つの値を取得するために必要な操作は1つだけであることを意味します。
これは、例えば極座標を直交座標に変換する場合に当てはまります。sin θ はcos θ も生成し、cos θ はsin θ も生成します。
同様に、直交座標から極座標への変換（phase（x、y）、modulus（x、y））および双曲線関数（cosh θ 、sinh θ ）の場合もです。

数値の表現

- 固定小数点
  - CORDICは固定小数点符号付き整数フォーマットで動作
  - 入力値と出力値は、q1.31またはq1.15のいずれか

q1.31 フォーマット

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	c_{30}	c_{29}	c_{28}	c_{27}	c_{26}	c_{25}	c_{24}	c_{23}	c_{22}	c_{21}	c_{20}	c_{19}	c_{18}	c_{17}	c_{16}	c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

小数部 $= (-1)^s \sum_{k=0}^{30} \frac{1}{2^{31-k}} c_k$

q1.15 フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

小数部 $= (-1)^s \sum_{k=0}^{14} \frac{1}{2^{15-k}} c_k$

- パッキング、アンパッキングはq1.15

q1.31形式では、数字は1つの符号ビットと31個の小数ビット（バイナリの場所）で表されます。
したがって、数値の範囲は−1（0x80000000）から、$1-2^{-31}$（0x7FFFFFFF）になります。
精度は2^{-31}（約5×10^{-10}）です。
q1.15形式では、数値の範囲は1(0x8000)から、$1-2^{-15}$（0x7FFF）です。
この形式には、2つの入力引数を1つの32ビット書き込みにパックでき、32ビット読み取りで2つの結果を取得できるという利点があります。
ただし、精度は2^{-15}（約3×10^{-5}）に減少します。

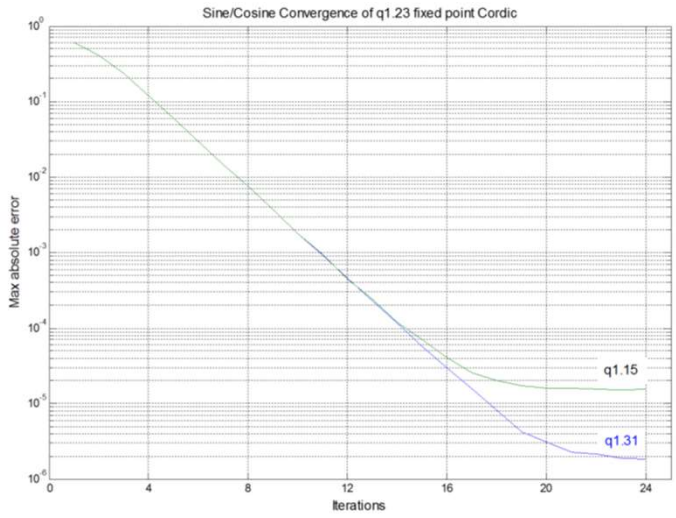
数値の表現

- 角度表現
  - 角度は固定小数点式で効率的に表現するため、πで除算
  - したがって-1は角度-πに対応し、+1は+πに対応
  - +πを超えて角度を大きくすると、自動的に-πに折り返す
- スケーリング係数
  - 一部の関数には、固定小数点の数値範囲を超えるドメインがある
  - この場合、2のべき乗の倍率（右シフト）を適用
  - スケーリング係数はCordic入力パラメータ（SCALE）で考慮され、出力で元に戻す必要がある

角度は、π で割ったラジアンで表されます。
したがって、間隔[−1,+1]のみが使用されます。
いくつかの関数は、スケールファクターSCALEを指定します。
これにより、入力レジスタ、出力レジスタ、または内部レジスタを飽和させることなく、Cordicでサポートされる値の範囲全体をカバーするように関数入力範囲を拡張できます。
スケーリング係数が必要な場合は、ソフトウェアで計算し、CORDIC_CSRレジスタのSCALEフィールドにプログラムする必要があります。
CORDIC_WDATAレジスタでスケール値をプログラミングする前に、入力引数をそれに応じてスケーリングする必要があります。
スケーリングは、CORDIC_RDATAレジスタから読み取られた結果でも元に戻す必要があります。
スケール係数には、スケール値の切り捨てによる精度の損失が伴います。

精度

- 三角関数(サイン、コサイン、フェーズ、絶対値（モジュラス）)
  - アルゴリズムは、反復ごとに1つの2進数の一定の速度で収束します

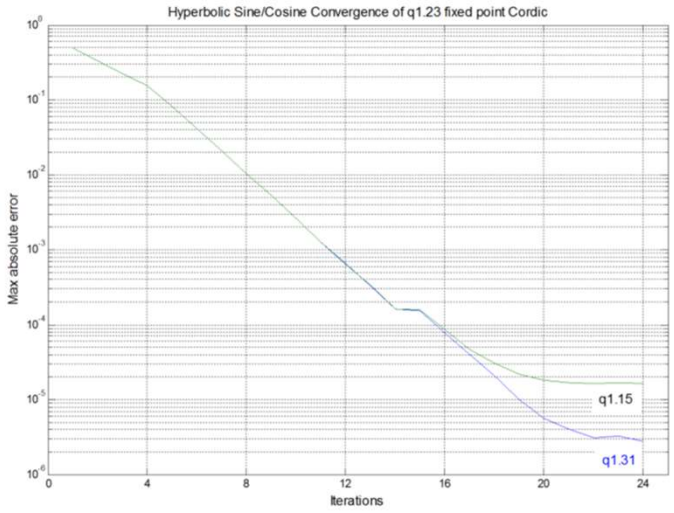


life.augmented

精度は、CORDIC反復の数によって異なります。
このアルゴリズムは、三角関数の反復ごとに1つの2進数字の一定の速度で収束します

精度

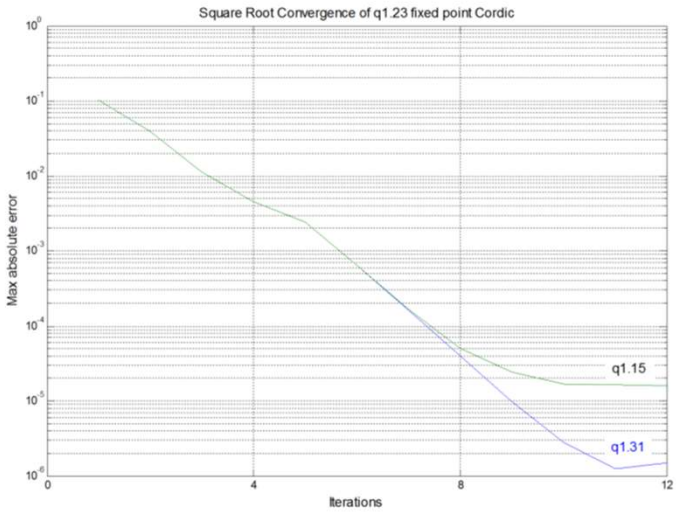
- 双曲線関数（双曲線サイン、双曲線コサイン、自然対数）
  - このアルゴリズムは、反復ごとに<1バイナリで収束



双曲線関数（双曲線サイン、双曲線コサイン、自然対数）の場合、収束率はCORDICアルゴリズムの特殊性のために小さく一定です。

精度

- 平方根関数
  - アルゴリズムは双曲線関数の約2倍の速度で収束



life.augmented

平方根関数は双曲線関数の約2倍の速度で収束します。

精度

- 内部のワード長
  - 数値はq1.23形式で内部的に表現
  - これは、丸め誤差が2^{-19}の精度になることを意味する
- 入出力のワード長
  - 最大精度を得る場合は、Q1.31形式を入力および出力に使用する必要がある
    - しかし、出力はせいぜい20ビットの精度に制限
  - 入力にq1.15形式を使用する場合、精度は出力フォーマットに関わらず、精度はq1.15に制限

life.augmented

引数と結果の形式は、CORDIC_CSRレジスタのARGSIZEおよびRESSIZEフィールドq1.15またはq1.31のいずれかで個別にプログラムされます。
内部では、コーディックアクセラレータはq1.23形式を実装します。これは、丸め誤差が2^{-19}の精度になることを意味します。
最大精度に達した後にCordicで反復を続けると、精度が徐々に低下します。
最大精度を得る場合は、q1.31形式を入力および出力に使用する必要があります。
ただし、内部的に実装された形式を考えると、出力は最高で20ビットの精度に制限されます。
q1.15形式を入力に使用する場合、出力形式に関する精度はq1.15に制限されます。

精度

- 繰り返し処理
  - Cordicのプログラミング時に、反復の数を4の倍数で指定可能
  - 各クロックサイクルで4つの反復を実行可能
  - 最高速度を得るには、必要な精度の最小反復回数をプログラムする必要がある

必要な精度は、CORDIC_CSRレジスタのフィールドPRECISIONにプログラムする必要がある反復の数によって異なります。
反復回数は、このフィールドに4を掛けた値と同じです。
最高速度を実現するには、必要な精度の最小反復回数をプログラムする必要があります。
ほとんどの関数では、このフィールドの推奨範囲は3~6です。

コサイン関数

パラメータ	説明	範囲
ARG1	角度θラジアン、πで除算	[-1 1]
ARG2	絶対値（モジュラス）m	[0 1]
RES1	m.cos θ	[-1 1]
RES2	m.sin θ	[-1 1]
SCALE	該当なし	0

- この関数は、-πからπの範囲の角度のコサインを計算
  - 極座標を直交座標に変換するためにも使用可能

life.augmented

コサイン関数の機能について説明します。
主引数は、ラジアン単位の角度θです。
ARG1をプログラミングする前に、πで除算する必要があります。
2番目の引数mは絶対値（モジュラス）です。
mが1より大きい場合、ソフトウェアでスケーリングを適用して、ARG2のq1.31範囲に適合させる必要があります。
第1の結果RES1は角度の余弦で、絶対値（モジュラス）を掛けた値です。
2次結果RES2は角度の副次的な値であり、絶対値（モジュラス）を掛けた値です。

サイン関数

パラメータ	説明	範囲
ARG1	角度θラジアン、πで除算	[-1 1]
ARG2	絶対値（モジュラス）m	[0 1]
RES1	m.sin θ	[-1 1]
RES2	m.cos θ	[-1 1]
SCALE	該当なし	0

- この関数は、-πからπの範囲の角度のサインを計算
  - 極座標を直交座標に変換するためにも使用可能

life.augmented

サイン関数の機能について説明します。
主引数は、ラジアン単位の角度θです。
ARG1をプログラミングする前に、πで除算する必要があります。
2番目の引数mは絶対値（モジュラス）です。
mが1より大きい場合、ソフトウェアでスケーリングを適用して、ARG2のq1.31範囲に適合させる必要があります。
第1の結果RES1は角度の中のサインに絶対値（モジュラス）を掛けた値です。
2次結果RES2は角度の余弦であり、絶対値（モジュラス）を掛けた値です。

フェーズ関数

パラメータ	説明	範囲
ARG1	x座標	[-1 1]
ARG2	y座標	[-1 1]
RES1	フェーズ角度θラジアン、πで除算	[-1 1]
RES2	絶対値（モジュラス）m	[0 1]
SCALE	該当なし	0

- この関数は、ベクトルv=[x y]の範囲-π~πの位相角度を計算
  - 直交から極への変換を実行するために使用することが可能

life.augmented

フェーズ関数の機能について説明します。
主引数はx座標、つまりx軸の方向のベクトルの大きさです。
|x|>1、ARG1のq1.31範囲に適応させるには、スケーリングをソフトウェアに適用する必要があります。
第2引数はy座標、つまりy軸の方向のベクトルの大きさです。
|y|>1、ARG2のq1.31範囲に適応させるには、スケーリングをソフトウェアに適用する必要があります。
第1の結果、RES1は、ベクトルvの位相角度θになります。
角度をラジアンで取得するには、RES1にπを掛ける必要があります。
πに近い値は、位相角度の円形の性質により-πに折り返されることがあります。
2次結果RES2は、次の値で示される絶対値（モジュラス）です。
|v|=√(x^2+y^2)。
|v|>1の場合、RES2の結果は1に飽和されます。

絶対値（モジュラス）関数

パラメータ	説明	範囲
ARG1	x座標	[-1 1]
ARG2	y座標	[-1 1]
RES1	絶対値（モジュラス）m	[0 1]
RES2	フェーズ角度θラジアン、πで除算	[-1 1]
SCALE	該当なし	0

- この関数は、ベクトルv=[x y]の大きさ（絶対値（モジュラス））を計算
  - 直交から極への変換を実行するために使用することができる

life.augmented

絶対値（モジュラス）関数の機能について説明します。
主引数はx座標、つまりx軸の方向のベクトルの大きさです。
|x|>1、ARG1のq1.31範囲に適応させるには、スケーリングをソフトウェアに適用する必要があります。
第2引数はy座標、つまりy軸の方向のベクトルの大きさです。
|y|>1、ARG2のq1.31範囲に適応させるには、スケーリングをソフトウェアに適用する必要があります。
主な結果であるRES1は、次の値で示される係数です。
|v|=√(x^2+y^2)
|v|>1の場合、RES1の結果は1に飽和されます。
2次結果、RES2は、ベクトルvの位相角度θである。
角度をラジアンで取得するには、RES2にπを掛ける必要があります。
πに近い値は、位相角度の円形の性質により−πに折り返されることがあります。

アークタンジェント関数

パラメータ	説明	範囲
ARG1	$x.2^{-n}$	[-1, 1]
ARG2	N/A	-
RES1	2-n.tan-1 x、ラジアン、πで割る	[-1, 1]
RES2	N/A	-
SCALE	n	[0, 7]

- この関数は、入力引数xでアークタンジェント（逆タンジェント）を計算

life.augmented

アークタンジェント関数の機能について説明します。
ARG1の主引数は、入力値$x=\tan\theta$です。
$|x|>1$、$-1<x*2^{-n}<1$のようなソフトウェアで、2^{-n}のスケーリング係数を適用する必要があります。
スケール値$x*2^{-n}$はARG1でプログラムされ、スケール係数nはSCALEパラメーターでプログラムする必要があります。
許容される最大入力値は、$\tan\theta=128$で、角度$\theta=89.55$度に相当します。
$|x|>128$、$\tan^{-1}x$を見つけるにはソフトウェアメソッドを使用する必要があります。
2次引数ARG2は使用されていません。
1次結果のRES1は、角度$\theta=\tan^{-1}x$.RES1に$2^n.\pi$を掛けてラジアンで角度を求める必要があります。
2次結果のRES2は使用されません。

双曲線コサイン関数

パラメータ	説明	範囲
ARG1	$x.2^{-n}$	[-0.559 0.559]
ARG2	N/A	-
RES1	$2^{-n}.\cosh x$	[0.5 0.846]
RES2	$2^{-n}.\sinh x$	[-0.683 0.683]
SCALE	n	1

- この関数は、双曲線角度xの双曲線コサインを計算
  - また、次の計算にも使用可能
    - 指数関数$e^x = \cosh x + \sinh x$
    - $e^{-x} = \cosh x - \sinh x$

双曲線コサイン関数の機能について説明します。
主引数は双曲線角度xです。
サポートされているのは、−1.118~+1.118の範囲のxの値だけです。
cosh xの最小値は1で、q1.31形式の範囲を超えているため、ソフトウェアでは2^{-n}のスケーリング係数を適用する必要があります。
演算項目n=1は、SCALEパラメーターでプログラムする必要があります。
2次引数ARG2は使用されていません。
主な結果であるRES1は、双曲線コサイン、cosh xです。
正しい結果を得るには、RES1に2を掛ける必要があります。
2次結果RES2は双曲線正弦、sinh xです。
正しい結果を得るには、RES2に2を掛ける必要があります。

双曲線サイン関数

パラメータ	説明	範囲
ARG1	$x.2^{-n}$	[-0.559 0.559]
ARG2	N/A	-
RES1	$2^{-n}.\sinh x$	[-0.683 0.683]
RES2	$2^{-n}.\cosh x$	[0.5 0.846]
SCALE	n	1

- この関数は、双曲線角度xの双曲線サインを計算
  - また、次の計算にも使用可能
    - 指数関数$e^x = \cosh x + \sinh x$
    - $e^{-x} = \cosh x - \sinh x$

life.augmented

双曲線サイン関数の機能について説明します。
主引数は双曲線角度xです。
サポートされているのは、−1.118~+1.118の範囲のxの値だけです。
すべての入力値について、ソフトウェアで2^{-n}のスケーリング係数
を適用する必要があります(n=1)。
スケール値x·0.5はARG1でプログラムされ、演算項目n=1は
SCALEパラメーターにプログラムする必要があります。
2次引数ARG2は使用されていません。
主な結果であるRES1は、双曲線サイン、sinh xです。
正しい結果を得るには、RES1に2を掛ける必要があります。
2次結果RES2は双曲線コサイン、cosh xです。
正しい結果を得るには、RES2に2を掛ける必要があります。

双曲線アークタンジェント関数

パラメータ	説明	範囲
ARG1	$x.2^{-n}$	[-0.403 0.403]
ARG2	N/A	-
RES1	2^{-n}.atanh x	[-0.559 0.559]
RES2	N/A	-
SCALE	n	1

- この関数は、入力引数xの双曲線アークタンジェントを計算

双曲線アークタンジェント関数の機能について説明します。
主引数は、入力値xです。
サポートされているのは、−0.806~+0.806の範囲のxの値だけです。
値xは、因子2^{-n}N=1でスケールする必要があります。
スケール値x·0.5はARG1でプログラムされ、演算項目n=1は
　SCALEパラメーターにプログラムする必要があります。
2次引数ARG2は使用されていません。
1次結果RES1は双曲線アークタンジェント、atanh xです。
正しい値を取得するには、RES1に2を掛ける必要があります。
2次結果は使用されません。

自然対数関数

パラメータ	説明	範囲
ARG1	$x.2^{-n}$	[0.054 0.875]
ARG2	N/A	-
RES1	$2^{-(n+1)}.\ln x$	[-0.279 0.137]
RES2	N/A	-
SCALE	n	[1 4]

- この関数は、入力引数xの自然対数を計算

life.augmented

自然対数関数の特徴について説明します。
主引数は、入力値xです。
0.107~+9.35の範囲のxの値のみがサポートされます。
値xは、$2^{-n}<1-2^{-n}$のように、因子2^{-n}でスケールする必要があります。
スケール値$x\cdot2^{-n}$はARG1でプログラムされ、演算項目n=1はSCALEパラメーターにプログラムする必要があります。
2番目の引数は使用されていません。
主な結果であるRES1は、自然対数です。
正しい値を取得するには、RES1に$2^{(n+1)}$を掛ける必要があります。
2次結果は使用されません。

平方根関数

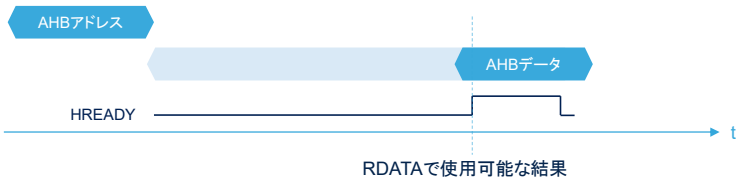
パラメータ	説明	範囲
ARG1	$x.2^{-n}$	[0.027, 0.703]
ARG2	N/A	-
RES1	2^{-n}.	[0.04, 1]
RES2	N/A	-
SCALE	n	[0, 2]

- この関数は、入力引数xの自然対数を計算

平方根関数の機能について説明します。
主引数は、入力値xです。
0.027~2.34の範囲のxの値のみがサポートされます。
値xは、$2^{-n}<1-2^{(-n-2)}$のように、因子2^{-n}でスケールする必要があります。
スケール値$x\cdot2^{-n}$はARG1でプログラムされ、演算項目n=1はSCALEパラメーターにプログラムする必要があります。
2番目の引数は使用されていません。
第1の結果RES1は、xの平方根です。
正しい値を取得するには、RES1に2^nを掛ける必要があります。
2次結果は使用されません。

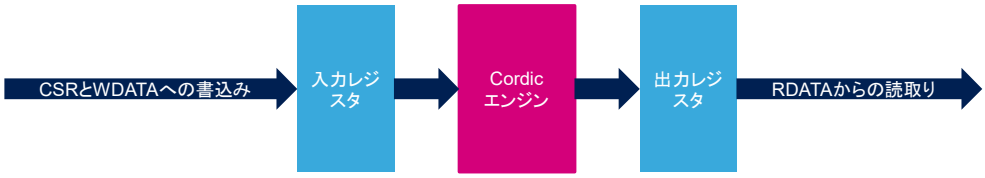
結果(リザルト)レジスタ

- 結果(リザルト)レジスタ(CORDIC_RDATA)
  - 演算が進行中CORDIC_RDATAへの読取りアクセスは、結果が使用可能になるまで待機し完了となる。
  - これは、RRDYフラグをポーリングする必要がないことを意味する
  - 結果は、使用可能になるとすぐにCPUに返される



Cordicブロックに計算を任せたソフトウェアでは、この計算がいつ完了するかを判断するためのフラグをポーリングする必要はありません。
単にAHBバスを介してRDATAレジスタの読み取り要求を開始します。
AHBトランザクションと同様に、スレーブはHREADYシグナルを低く維持することによって待機状態を挿入することが許可されます。
結果の1つが使用可能な場合、CordicブロックはHREADYをアサートし、トランザクションを完了します。
その間、Cortex-M4プロセッサはフリーズします。
このアプローチは、ゼロオーバーヘッドモードと呼ばれます。

パイプライン・オペレーション



- WDATAが書き込まれると、引数と設定が入力レジスタに格納（NARGSの値に応じて1回または2回）
  - プログラム操作が保留中になる
  - 実行中の操作が終了すると、結果は出力レジスタに格納され、RRDYフラグが設定される

結果がRDATAから読み取られると（NRESの値に応じて1回または2回読み取られる）、ペンディング操作が開始されます。
ペンディング中の操作がない限り、新しい引数（および設定）のセットを書き込むことができます。
つまり、Cordic操作の完了を待機するのに費やす時間を使用して次の操作を準備することができ、Cordicがアイドルになることはありません。
CORDIC_CSRレジスタは、計算の進行中の計算の結果に影響を与えることなく、計算の進行中に再プログラムすることができます。

Cordicの使い方

ゼロオーバーヘッド･シングルショット･モード

- ゼロオーバーヘッド･シングルショット･モードは、単一の計算を実行する最速の方法

1. 必要に応じて、CORDIC_CSRレジスタを適切な設定でプログラム
2. CORDIC_WDATAレジスタで計算の引数をプログラム
   - ➢ これにより、計算がスタート
3. CORDIC_RDATAレジスタから結果を読み取る

life.augmented

このシーケンスでは、シングルショットオペレーションを想定したゼロオーバーヘッドモードでのCORDIC_IPの使用について説明します。
プロセッサは現在のオペレーションの完了で待機するので、それ以上の計算はスケジュールされません。

ゼロオーバーヘッドパイプラインモード

- ゼロオーバーヘッドパイプラインモードは、いくつかの連続した計算を実行する最速の方法

1. 適切な設定でCORDIC_CSRレジスタにプログラム
2. CORDIC_WDATAレジスタの最初の計算の引数をプログラム
   - これにより、最初の計算がスタート
3. 必要に応じて、次の計算のためにCORDIC_CSRレジスタ設定を更新（ステップ3）
4. CORDIC_WDATAレジスタの次の計算の引数をプログラム
5. CORDIC_RDATAレジスタから結果を読み取る
   - これは次の計算をトリガ
6. ステップ3へ
7. シーケンスの最後に、最終の計算結果を取得するために追加の読取りが必要

life.augmented

このシーケンスでは、パイプライン処理を想定したゼロオーバーヘッドモードでのCORDIC_IPの使用について説明します。
手順3~6を繰り返すことで、ソフトウェアは引数の配列に対して同じ操作を再実行できます。
最後の操作の結果を取得するには、7番目の手順が必要です。

Cordicの使い方

ポーリング・モード

- RRDYフラグをポーリングし結果を読み取る間の遅延があり、時間が必要
  - ただし、結果を待っている間にプロセッサを中断できる

1. 必要に応じて、適切な設定でCORDIC_CSRレジスタをプログラム
2. CORDIC_WDATAレジスタに引数をプログラム
   - ➢ これにより、計算がスタート
3. CORDIC_CSRレジスタのRRDYフラグが立つまでポーリング
4. CORDIC_RDATAレジスタから結果を読み取る

- パイプライン処理は、ポーリング・モードでも使用可能

life.augmented

このシーケンスは、ポーリングモードでのCORDICの使用方法です。
CORDIC_RDATAレジスタで新しい結果が得られた場合、RRDYフラグはCORDIC_CSRレジスタに設定されます。
このレジスタを読み取ることで、フラグをポーリングできます。
CORDIC_RDATAレジスタを読み取ることによってリセットされます。
（CORDIC_CSRレジスタのNRESフィールドに応じて1回または2回）
RRDYフラグのポーリングは、CORDIC_RDATAレジスタを直接読み取るよりも少し時間がかかります。
ただし、プロセッサとバスインタフェイスはCORDIC_CSRレジスタの読み取り中に停止しませんので、プロセッサの停止が許容されない場合(たとえば、低遅延割り込みを処理する必要がある場合、このモードは使用可能な場合があります。

Cordicの使い方

割込みモード

- RRDYフラグをポーリングする代わりに、結果が得られるとCPUに割り込みが発生
  - 割込みの処理には余分なサイクルが必要だが、このモードでは、他のタスクに関してCordicの優先順位が設定可能

1. 適切な設定でCORDIC_CSRレジスタをプログラムし、IENビットをセット
2. CORDIC_WDATAレジスタの引数をプログラム
   - ➢ これにより、最初の計算がスタート
3. 結果の準備が整うと、Cordic割込みが発生
4. 割り込みハンドラのCORDIC_RDATAレジスタから結果を読み取る
   - ➢ これにより、RRDYフラグがリセットされ、割り込み要求がクリア
5. パイプライン処理は割り込みモードで使用可能
   - ➢ ただし、書き込みと読み取りの順序を維持するように注意が必要

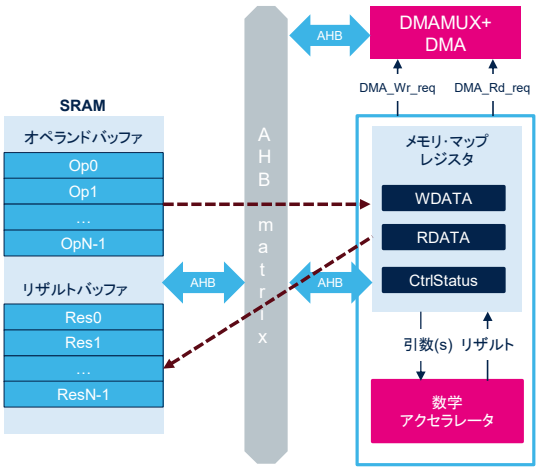
life.augmented

このシーケンスは、割り込みモードでのCORDIC_IPの使用方法です。
CORDIC_CSRレジスタに割り込み可能(IEN)ビットを設定することにより、RRDYフラグが設定されるたびに割り込みが生成されます。
フラグがリセットされると、割り込みはクリアされます。
このモードでは、計算の結果を割り込みサービスルーチンで読み取ることができ、したがって他のタスクに対する優先順位が与えられます。
ただし、割り込み処理の遅延により、結果を直接読み取ったり、フラグをポーリングしたりするよりも遅くなります。

Cordicの使い方

DMAモード

- DMAモードを使用して、同じ設定を使用して複数の計算を実行可能
  - ソフトウェア実装と比較して計算を高速化するだけでなく、他のタスクのためにCPUを解放する
- DMAを使用して、Cordicに引数を書き込むことが出来る
  - 新しい計算が開始されるたびに、DMA書込みチャネル要求が生成され、次の引数がメモリからフェッチされる
- 同様に、DMAを使用して結果を読取ることが可能
  - 計算が完了するたびに(RRDYフラグがアクティブになる)、DMA読み取りチャネル要求が生成され、結果がメモリにコピーされる



DMAモードは、同じ設定を使用して複数の計算を実行する場合に非常に効率的です。
DMAでCORDIC_CSRレジスタを変更することはできません。
したがって、設定を変更する必要がある場合は、まずDMAを停止し、新しい設定をプログラムした後に再起動する必要があります。
DMA書込みは、DMA、ポーリング、または読込みメソッドの割り込みと組み合わせることができます。
パイプライン処理は、常にDMAモードで使用されます。
DMA書込み要求は、CORDIC_CSRレジスタでDMAWENビットを設定することによって有効になります。
DMA読取り要求は、CORDIC_CSRレジスタでDMARENビットを設定することによって有効になります。

Cordic vs ARM fast math関数

固定小数点のサイン

- メモリに格納された3024アングルのバッファを、48kspsでサンプリングされた1kHzの正波を表すサンプルに変換し、メモリに保存
  - アングルはCPUによって事前に計算される

q1.15固定小数点、精度=4		
ARM fast math arm_sin_q15()	Cordic:ゼロオーバーヘッドモード	Cordic: DMAイン/アウトモード
36サイクル/サンプル	7サイクル/サンプル	11サイクル/サンプル
-	x5 高速	x3 高速
100%CPU	100% CPU	0% CPU
最大エラー: 0.00012	最大エラー: 0.00004	最大エラー: 0.00004
-	x3 精度	x3 精度

q1.31固定小数点、精度=6		
ARM fast math arm_sin_q31()	Cordic:ゼロオーバーヘッド･モード	Cordic: DMAイン/アウト･モード
41サイクル/サンプル	8サイクル/サンプル	12サイクル/サンプル
-	x5 高速	x3 高速
100%CPU	100% CPU	0% CPU
最大エラー: 0.00002	最大エラー: 0.000002	最大エラー: 0.000002
-	x10 精度	x10 精度

固定小数点でサインを計算する場合のCordicとARMのfast math関数のパフォーマンスを比較します。
q1.15 およびq1.31形式を使用すると、パフォーマンスの比率は同じです。
ゼロオーバーヘッドモードではCordicが5倍、DMAのイン/アウトモードでは3倍高速です。

Cordic vs ARM fast math関数

浮動小数点のサイン

- 32ビット浮動小数点数は、ソフトウェアの乗算とキャストによって固定小数点との間で変換できる（Cortex-M4 FPUを使用）
  - value_q31 = (int31_t)(value_f32*0x80000000); /* f32 to q1.31 */
  - value_f32 = (float)value_q31/(float)0x8000000; /* q1.31 to f32 */

32ビット浮動小数点、精度=6	
ARM fast math arm_sin_f32()	**Cordic:ゼロオーバー・ヘッドモード**
66サイクル/サンプル	21サイクル/サンプル ➢ float32からint32への変換を含み、戻る
-	x3 高速
100%CPU	100% CPU
最大エラー: 0.00002	最大エラー: 0.000002
-	x10 精度

life.augmented

この浮動小数点の計算では、サインを計算するときのCordicとARMのfast math関数のパフォーマンスを比較することです。Cordicは、float32からint32および戻り値への変換時間を含み、ゼロオーバーヘッドモードで3倍高速です。

Cordic vs ARM fast math関数

平方根、固定小数点

- バッファード平方根
  - メモリに格納されている3024値のバッファの平方根を計算し、メモリに格納

q1.15固定小数点、精度=3	
ARM fast math arm_sqrt_q15()	Cordic:ゼロオーバーヘッド・モード
101サイクル/サンプル	7サイクル/サンプル
-	x14 高速
100%CPU	100% CPU
最大エラー: 0.0002	最大エラー: 0.000015
-	x13 精度

q1.31固定小数点、精度=3	
ARM fast math arm_sqrt_q31()	Cordic:ゼロオーバーヘッド・モード
98サイクル/サンプル	7サイクル/サンプル
-	x14 高速
100%CPU	100% CPU
最大エラー: 0.000000004	最大エラー: 0.0000015
-	x0.003 精度

life.augmented

この固定小数点平方根を計算する際のCordicとARMのfast math関数のパフォーマンスを比較です。
q1.15およびq1.31形式を使用すると、パフォーマンスの比率は同じです。
Cordicのゼロオーバーヘッドモードでは14倍高速です。

Cordic vs ARM fast math関数

バッファード平方根、浮動小数点

32ビット浮動小数点、精度=3	
ARM fast math arm_sqrt_f32()	**Cordic:ゼロオーバーヘッド・モード**
27サイクル/サンプル	21サイクル/サンプル ➢ float32からint32への変換を含み、戻る
-	x1.3 高速
100%CPU	100% CPU
最大エラー: 0.00000003	最大エラー: 0.0000015
-	x0.02 精度

- 浮動小数点での平方根演算では、ARM fast math関数に対してCordicを使用する利点はほとんどない
  - math.hのsqrtf()関数のパフォーマンスが似ている

life.augmented

この浮動小数点での平方根を計算する際のCordicとARMのfast math関数のパフォーマンスを比較します、
コーディックは、float32からint32および戻り値への変換時間を含め、ゼロオーバーヘッドモードでは1.3倍高速です。

Cordic vs ARM fast math関数

パーク変換

- パーク変換は、モータ制御アプリケーションで広く使用されている:
  - X = D.cos θ - Q.sin θ
  - Y = D.sin θ + Q.cos θ

q1.15ビット固定小数点,、精度=4	
ARM fast math arm_sqrt_q15()	Cordic:ゼロオーバーヘッド・モード
243サイクル	48サイクル
-	x5 高速

この固定小数点q1.15パーク変換でCordicとARMのfast math関数のパフォーマンスを比較します。
ゼロオーバーヘッドモードでは、コーディックが5倍高速です。

低電力モード

モード	説明
RUN	有効
SLEEP	有効 • ペリフェラルからの割込みにより、デバイスはSLEEPモードを終了
低電力 RUN	有効
低電力 SLEEP	有効 • ペリフェラルからの割込みにより、デバイスが低電力SLEEPモードを終了
STOP0/ STOP1	利用不可
STOP2	
STANDBY	
SHOUTDOWN	

life.augmented

コーディックユニットは、RUN、低電力RUN、SLEEP、低電力SLEEPモードでアクティブです。
他の低電力モードでは使用できません。

関連ペリフェラル

- 必要に応じて、このペリフェラルにリンクされている以下のトレーニングを参照してください。:
  - DMA - ダイレクト･メモリ･アクセス･コントローラ
  - 割込み - ネスト化されたベクタ割込みコントローラ

life.augmented

これらのペリフェラルは、Cordicブロックで正しく使用できるように特別に構成する必要があります。
詳細については、対応するペリフェラルトレーニングモジュールを参照してください。