Vol.2021-SLDM-196 No.30
2021/12/2

Rocket-Chipへの三角関数演算命令の実装と評価

中尾 怜史[1,a)]　武内 良典[2,b)]

概要：本研究では，Rocket-Chip へのアクセラレータ部に三角関数演算命令を実装し，その性能の比較を行った．現在 IoT には低消費電力化などが求められている．そして特定用途に特化したプロセッサが必要とされているが，毎回設計し作成するには大きなコストを要してしまう．この問題に対し RISC-V を活用することが提案されている．RISC-V はライセンスフリー ISA で，シンプルな命令セットであることから現在注目されている．この ISA を実装するアーキテクチャとして Rocket-Chip がある．Rocket-Chip は設計者が独自の回路を追加でき，またアクセラレータを用いた複雑な処理の実行が可能なフリーのアーキテクチャである．
本報告では，Rocket-Chip のアクセラレータ部に対し CORDIC アルゴリズムで三角関数演算命令を実装することで，ハードウェアとソフトウェアでの命令の実行数や時間を比較した．

Implementation and Evaluation of Trigonometric Instructions on the Rocket-Chip

Abstract: In this study, we implemented trigonometric instructions in the accelerator part of the Rocket-Chip and compared the performance. Currently, low power consumption is required for the IoT. The IoT requires low power consumption, and processors specialized for specific applications are needed, but designing and creating them each time requires a large cost. RISC-V has been proposed as a solution to this problem, because it is a license-free ISA with a simple instruction set. The Rocket-Chip is a free architecture that allows designers to add their own circuits and to perform complex operations using accelerators.
In this report, we compared the number of instructions and the time to execute them in hardware and software by implementing trigonometric operations with the CORDIC algorithm in the accelerator part of the Rocket-Chip.

1. 序論

近年，あらゆる機器にプロセッサを取り付け，データのやりとりを行う IoT(Internet of Things) 化が進んでいる．家電といった生活圏から，常時電力供給が期待されない場所での運用も求められる．このため低消費電力化や高性能化などが必要とされている．しかしこれらの機能を実現するために，毎回特定用途に特化したプロセッサの開発を行うと，膨大なコストが必要となる．

一方 RISC-V と呼ばれる ISA(Instruction Set Architecture) を用いたプロセッサが提案されている [1], [2], [3]. RISC-V は UCB(カリフォルニア大学バークレー校) が開発したオープンな ISA である．RISC-V の特徴としては浮動小数点命令や圧縮命令など，実装する命令セットを選択できる拡張機能がある．また２つ目の特徴として，カスタム命令とよばれる設計者が任意に定義できる命令があり，特定用途に向けた専用の命令を実装することが可能である．

Rocket-Chip は RISC-V を実装とするアーキテクチャの一つである．Rocket-Chip は BSD ライセンスの SoC(System on Chip) で，UCB が開発し現在は SiFive 社がサポートをするフリーアーキテクチャである．Rocket-Chip は RISC-V のカスタム命令にも対応しており，RoCC(Rocket Custom Coprocessor) と呼ばれるカスタム可能な外部アクセラレータが設けられており，開発したいハードウェアに合わせた回路を追加することが可能である．本稿では Rocket-Chip に CORDIC の回路を実装し，ハードウェアとソフトウェアによる性能の違いを比較した．

ここで，本論文の構成を示す．第 2 節では RISC-V，Rocket-Chip と CORDIC について述べる．第 3 節では

[1] 近畿大学大学院総合理工学研究科
[2] 近畿大学理工学部電気電子工学科
[a)] 2033340434n@kindai.ac.jp
[b)] takeuchi@ele.kindai.ac.jp

CORDIC 手法を用いた回路の実装手法を述べる．第 4 節で適用を行い，第 5 節で本稿のまとめを述べる．

2. RISC-V, Rocket-Chip, CORDIC 概要

本節では，本論文で用いる RISC-V と，これを ISA として実装するアーキテクチャ Rocket-Chip の概要について述べる．また Rocket-Chip に追加で実装する回路で使用するアルゴリズム CORDIC の概要も述べる．

2.1 RISC-V

RISC-V はレジスタ長を 32, 64, 128 ビット，またアドレッシングモードも同じ 3 種類から選択することができる ISA である [4], [5], [6]．RISC-V は従来の ISA と比べ単純性が高められており，実行命令数は多くなるが 1 命令当たりのクロック数が少なくなる命令セットになるよう設計されている．

2.1.1 命令フォーマット

RISC-V の命令ビットフィールドはレジスタ長に関わらず 32 ビット長で統一されている. フォーマットは R, I, S, U，B, J の 6 種類が用意されており，レジスタやオペコードの位置が共通となるよう設計がされている．

RISC-V のオペコードは 28 種類用意されており，命令タイプによって共通のオペコードが利用されている．また命令フィールドに用意された特定の領域に個別の値を設定することで，1 つのオペコードに対し複数の命令を設定させることが可能である．また，命令ビットフィールドの [1,0] ビット目は，C 拡張の命令を除く全ての命令が 0x3 の値で固定されている．

2.1.2 命令セット

RISC-V には基本命令と拡張命令の 2 種類の命令があり，拡張命令は任意で着脱が可能な命令セットである．基本命令は算術命令, 論理演算命令の整数値を扱う命令の他，ロード, ストア命令やジャンプ命令といった制御系命令も定義され，アプリケーションを動作させるために必要最低限な命令セットとなっている．また，演算で使用する 0 レジスタと，整数値を保持する 31 個の汎用レジスタが用意されている．

拡張命令は M 拡張の乗除算命令，A 拡張のメモリに対して不可分の操作を行うアトミック命令，F, D, Q 拡張の単精度, 倍精度, 4 倍精度浮動小数点命令などがある．また，E 拡張は組込みシステムなどの小さなコア設計に対応する拡張として，汎用レジスタを半数に抑えた拡張となっている．

他，ベクトル命令を含む V 拡張といった現在各種実装法が提案されている拡張命令も存在し，RISC-V は現在も改良が続いている ISA である．

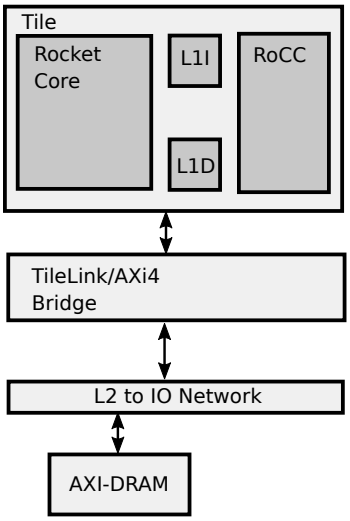

図 1 Rocket-Chip outline
Fig. 1 Outline of Rocket-Chip

2.2 Rocket-Chip

図 1 に Rocket-Chip の概略図を示す．Rocket-Chip は，RISC-V を ISA とした CPU を含む Tile と呼ばれるコア部分と，AXI バスで接続されたメモリを持つ SoC アーキテクチャである [2]．L2 キャッシュや IO へ接続部分には，複数のマスターデバイスとスレイブデバイスをコヒーレントで接続するためのプロトコルとして TileLink が採用されている．Rocket-Chip は Chisel と呼ばれるオブジェクト指向を含むハードウェア構築言語を用いて記述されている．そのため CPU コアやキャッシュの数などを容易に変更でき，独自のアーキテクチャを構築することが可能である．対応する RISC-V の命令は，基本命令, 拡張命令の M, A, F, D, C である．アドレッシングモードは 32, 64, 128 ビットに対応している．

Rocket-Chip に用いられるハードウェア構築言語 Chisel[7] は，プログラミング言語 Scala の埋め込み言語である．Chisel では予め用意されたクラスを継承することで，クロックやリセット信号を考慮する必要がないという特徴を持つ．

2.3 CORDIC

CORDIC(COrdinate Rotation DIgital Computer) は，1957 年に Jack E. Volder によって提案された三角関数などの初等関数の演算を可能とするアルゴリズムである [8]. 最大の特徴として乗除算を使用せず初等関数を求めることができることである. これにより小規模な回路で実装できるため，関数電卓や組み込み IC に組み込まれることが多い．また 1971 年には John Stephen Walther が新たに提案し，対数関数や双曲線関数などが演算可能となるといった拡張がなされた．

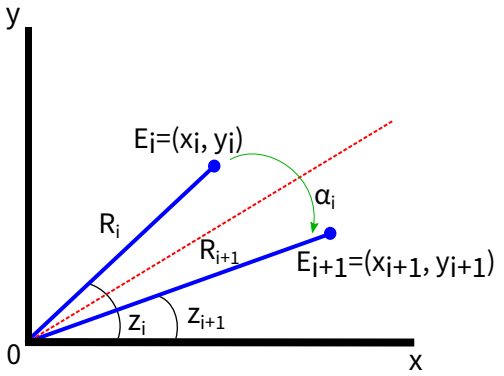

図 2 CORDIC のベクトル回転
Fig. 2 Vector rotation of CORDIC

図 2 は CORDIC アルゴリズムにおけるベクトル回転を表したものである [9]．例えば図 2 のように点 E_i から点 E_{i+1} へ角度 α_i だけベクトル回転した時，このときの角度 z_{i+1} は，

$$z_{i+1} = z_i - \alpha_i \tag{1}$$

となる．ただし実際の回転移動は疑似回転と呼ばれるものであり，移動した時のベクトル長 R_{i+1} は移動前のベクトル長 R_i を用いて，

$$R_{i+1} = R_i(1 + \tan^2(\alpha_i))^{\frac{1}{2}} \tag{2}$$

だけ延長されることになる．つまり移動後の各座標の値は，

$$x_{i+1} = x_i - y_i \tan(\alpha_i) \tag{3}$$
$$y_{i+1} = y_i + x_i \tan(\alpha_i) \tag{4}$$

となる．CORDIC では回転移動を複数回繰り返す．したがって，m 回疑似回転を行った場合各値の一般式は，

$$\begin{aligned} x_m &= (x\cos(\sum \alpha_i) - y\sin(\sum \alpha_i)) \prod (1 + \tan^2 \alpha_i)^{\frac{1}{2}} \\ &= \kappa(x\cos(\sum \alpha_i) - y\sin(\sum \alpha_i)) \end{aligned} \tag{5}$$
$$\begin{aligned} y_m &= (y\cos(\sum \alpha_i) + x\sin(\sum \alpha_i)) \prod (1 + \tan^2 \alpha_i)^{\frac{1}{2}} \\ &= \kappa(y\cos(\sum \alpha_i) + x\sin(\sum \alpha_i)) \end{aligned} \tag{6}$$
$$z_m = z - (\sum \alpha_i) \tag{7}$$

になる．x, y, z は初期値で，拡大係数 κ は α_i に依存する．ここでいつも同じ角度で回転した場合，定数として事前に求めることができるようにる．したがって，実際の回転よりも簡単な疑似回転となり，既知の定数を用いて各値を求めることが可能となる．

2.3.1 CORDIC の回転操作

CORDIC は 3 つの回転方法があり，各回転方法の違いは角度 α_i のとり方にある．

2.3.1.1 円回転

円回転では角度 α_i が以下の式

$$\tan(\alpha_i) = d_i 2^{-i} \tag{8}$$

を満たす時である．ただし $d_i \in \{-1, 1\}$ である．円回転における疑似回転後の各値は改めて書き直すと，

$$x_{i+1} = x_i - d_i y_i 2^{-i} \tag{9}$$
$$y_{i+1} = y_i + d_i x_i 2^{-i} \tag{10}$$
$$z_{i+1} = z_i - d_i \arctan(2^{-1}) \tag{11}$$

となる．式 (9)〜式 (11) より各要素を求めるには，加減算，右シフト演算と定数テーブル参照を組み合わせることで可能であることが分かる．また拡大係数 κ は 1.646760258... となる．

2.3.1.2 線形回転

線形回転は角度 α_i が 2^{-i} を満たすときである．ただしこの時の回転では，x_m は x_0 と等しく，ベクトル長は変化しない．

2.3.1.3 双曲線回転

双極線回転は角度 α_i が $\tanh^{-1}(2^{-i})$ を満たす時であり，ベクトル長は縮小していく．

2.3.2 モード

CORDIC には 1 つの回転方法に対し 2 つのモードがある．以下では各モードについて述べる．なお本稿では円回転を用いるため，本説で説明したモードの動作となる．

2.3.2.1 ローテーションモード

ローテーションモードは角度を 0 に近づけるよう疑似回転を行うモードである．m 回疑似回転した時，各値は初期値 x, y, z を用いると以下の式で表される．

$$x_m = \kappa(x\cos(z) - y\sin(z)) \tag{12}$$
$$y_m = \kappa(y\cos(z) + x\sin(z)) \tag{13}$$
$$z_m = 0 \tag{14}$$

ここで $x = 1/\kappa, y = 0$ とすることで，$x_m = \cos(z), y_m = \sin(z)$ と任意の角度 z に対し $\sin, \cos$ 関数を求めることができる．またこれらを除算することで $\tan$ 関数も求めることができる．

2.3.2.2 ベクタリングモード

ベクタリングモードは y 座標を 0 に近づけるよう疑似回転を行うモードである．m 回疑似回転した時，各値は初期値 x, y, z を用いると以下の式で表される．

$$x_m = \kappa(x^2 + y^2)^{\frac{1}{2}} \tag{15}$$
$$x_m = 0 \tag{16}$$
$$z_m = z + \arctan\frac{y}{x} \tag{17}$$

ここで $x = 1, z = 0$ とすると，z_m の値から任意の y の値に対する $\arctan$ 関数を求めることができる．

これ以外にも関数を組み合わたり，入力値を工夫することで実現できる関数も存在する．

3. 実装手法

本稿では拡張性を持つ RISC-V と Rocket-Chip に対し CORDIC の回路を実装することで，ハードウェアとソフトウェアで実行した時の性能の違いを比較した．まず RISC-V のソフトウェア環境を拡張し，CORDIC で実装する初等関数をカスタム命令として登録した．また Rocket-Chip の拡張はアクセラレータ RoCC を用いて CORDIC を実装した回路の記述を追加し実現した．

3.1 ソフトウェアの拡張

RISC-V へのソフトウェア環境の追加は，RISC-V の riscv-gnu-toolchain に対しカスタム命令の定義文を作成し追加した．また RISC-V の命令動作の確認を行うシミュレータ spike を含む riscv-tools に対してもカスタム命令の追加記述を行った．Rocket-Chip のアクセラレータを実行させるカスタム命令は，**図 3** の R タイプの命令フォーマットを使用した．命令フォーマットの 14 - 12 ビット目は，アクセラレータで使用できるカスタム命令のオペランドを格納するレジスタを表している．fucnt7 はアクセラレータ内部の挙動を変化させる領域で，本実験では命令ごとに固有の値を持たせることで各関数の演算をするための挙動を指定した．

3.2 ハードウェア部への拡張

Rocket-Chip への追加について述べる．Rocket-Chip への処理構造の拡張については複数の方法が存在する [10]．本実験では，適応する CORDIC アルゴリズムは複数のサイクル処理を要するため，これを実現する方法として RoCC を用いた拡張方法を採用した．

図 4 に RoCC の概要を示す．RoCC(Rocket Custom Coprocessor) は Rocket-Chip のアクセラレータであり，複雑な処理を実行することで全体を高速処理することを目的として設計されている．RoCC Interface を介して，CPU コアからオペランドの値などを含むデータ cmd を受取り，演算結果と出力レジスタのアドレスを含むデータ resp を返す．データのやり取りは，専用の制御信号を用いて行われる．また，RoCC は CPU コア以外にもキャッシュメモリなどと直接通信させ演算処理を行わせることも可能である．

RoCC は，カスタム命令用のオペコード 1 つにつき 1 つの RoCC を実行することができ，funct7 の値によって動作を制御させることができる．また RoCC は，status レジスタの mstatus_xs に 0 以外の値を設定することで動作させることができる．

3.2.1 CORDIC の実装

ここでは実装した CORDIC の仕様と，処理手順について述べる．

3.2.1.1 数値フォーマット

CORDIC を実装するに当たって，入力する値は全て整数値とし，アルゴリズム中では固定小数点を用いて処理を行った．固定小数点とは整数型を用いた小数値を表す方法である．ビット長に対し任意の小数点位置を設定し，小数点位置よりも左側を整数部, 右側を小数部に分け，整数型で演算することで変換することができる．また演算結果を確認する場合は, 右シフト (2 の冪乗で除算) することで小数部を含む値を得られる．固定小数点は浮動小数点数と比べ，小数部のビット長が短くなるため値の精度は落ちる．

ここで arctan 関数について考える．入力 y 対して $y = 0$ の時 $\arctan(y)$ は 0, $y = 1$ の時は 45 となり，45° 未満の値が出力されない．よって y に小数の値を持たせるため 10^n 倍し，小数部を整数部に上げることとした．今回は小数第 4 桁までを確保することとし，10^4 倍をかけた値を入力とした．その結果整数部のビット長を確保しなければならず，小数部が短くなり精度が下がってしまった．そこで CORDIC 中は扱うビット長を 128 ビットに拡張した．

3.2.1.2 処理工程

CORDIC を実装するに当たり，各処理を 8 ステージに分けて実装した．第 1 ステージでは，制御信号を用いて CPU コアから cmd を受け取り，レジスタに格納する．また funct7 を読み込み命令の判定を行う．第 2 ステージは値のチェックを行う．今回は $\sin, \cos$ 関数をカバーする範囲として $-360° \leq \theta \leq 360°$ とし，範囲を超える値は全て 0° とした．また arctan は出力できる範囲を $-89° \leq \theta \leq 89°$ に限定した．またどちらも入力が負の値の場合，絶対値変換して演算を行う．第 3 ステージでは角度の象限判定を行う．判定を行った後は減算を行うことで全ての値を 0° から 90° の範囲で演算する．第 4 ステージは参照テーブルの決定と初期値 x, y, z の設定を行う．入力した値はこのステージで左シフトされ固定小数点となる．第 5, 6 ステージは CORDIC アルゴリズムを実行するステージで，今回は 14 回疑似回転を行う，第 5 ステージではモード別に x, y, z の加減算を行う．また第 6 ステージでは，カウンタを用いることで右シフトする回数を制御し 2^{-i} した値を演算する．第 7 ステージは第 2, 3 の判定によって出力する値を正負や値の交換を行う．最後の第 8 ステージでは制御信号を操作し，CPU コアへ演算結果を送信する．

4. 実行結果と性能比較

本節では，第 3 節で提示した手法で実装し，ソフトウェアとハードウェアで実行した時の性能を比較し検討する．**表 1** に実装したカスタム命令を示す．実装した命令は全て整数命令として実装した．

4.1 出力値の誤差について

最初に実装した回路の精度について調査を行った．なお

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
funct7							rs2					rs1					xd	xs1	xs2	rd					opcode						
roccinst[6:0]							rs2					rs1					1	1	1	rd					custom0/1/2/3						

図 3 RoCC で動作する命令フォーマット
Fig. 3 RoCC's Instruction Format

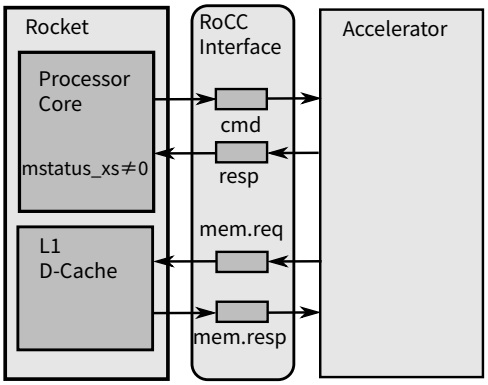

図 4 RoCC の概要
Fig. 4 Outline of RoCC

表 1 実装命令
Table 1 Implement Instructions

name	behavior
mysin	$rd = \sin(rs1)$
mycos	$rd = \cos(rs1)$
myatan	$rd = \arctan(rs1), rs1 = (decimal fraction) * 10^4$

sin, cos 関数は $-360°$ ～ $360°$ で，arctan は桁上げ前の数値で -58 ～ 58 の範囲で入力した．その結果を**図 5**, **図 6** に示す．RoCC へ実装した CORDIC 回路で出力した値は小数第 15 位まで演算することができた．

精度は C プログラムの倍精度による計算結果を真の値とし相対誤差を求めた．結果 sin, cos 関数の演算誤差は約 0.4% 以下となった．また sin, cos 関数の値の絶対値が 1 に近づくと誤差が小さく，0 に近づくと大きくなることが分かった．次の arctan 関数での相対誤差は約 0.1% 以下となった．こちらも関数の値が 0 に近づくと誤差が大きくなることが分かった．

4.2 CSR を用いた測定方法

それぞれの性能を調べるため，実行するのにかかった命令数とサイクル数を測定し比較した．RISC-V には CSR(Control and Status Registers) が用意されており，その中に mcycle, minstret と呼ばれるレジスタが存在する．mcycle はサイクル数, minstret は実行命令数が記録されており，これを読み込むことで性能を確認することができる．CSR にアクセスするには実行モード:Machine Mode の特権命令として実行する必要がある．

rsicv-tests には Machine Mode で実行するよう予め設定

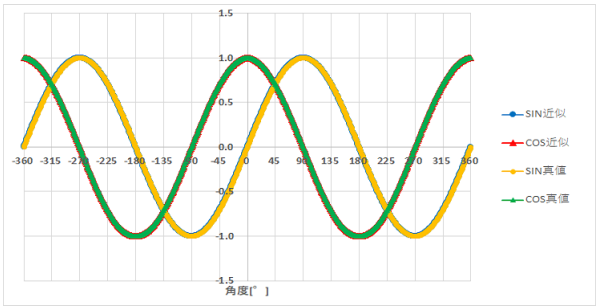

図 5 RoCC に実装した mysin, mycos 命令と c プログラムで求めた値との比較
Fig. 5 Comparison between the mysin and mycos instructions implemented in RoCC and the values obtained by the c program

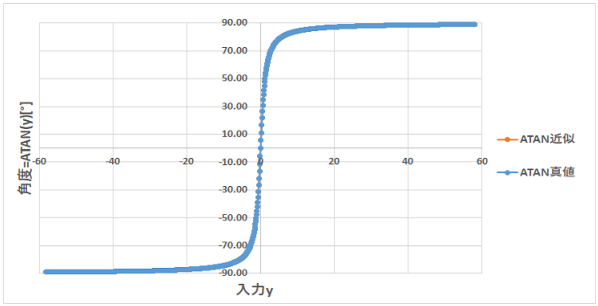

図 6 RoCC に実装した myatan 命令と c プログラムで求めた値との比較
Fig. 6 Comparison between the myatan instruction implemented in RoCC and the value obtained by the c program

がされているので，調査したい処理の前後での差をもとめること出力した．

4.3 性能比較

表 2, **表 3**, **表 4**, **表 5**, **表 6** に，それぞれの初等関数を RoCC に実装した CORDIC 回路と，テイラー展開, 固定小数点と単精度浮動小数の CORDIC を用いたソフトウェア実現の 4 種類の方法で作成したプログラムで実行した時の実行命令数と実行サイクル数を示す．入力は sin, cos は $60°$ と $45°$, atan は 1 と 2 の交互にし，それぞれ 10 回演算を行った．

最初に実行命令数について見ると, RoCC とテイラー展開とでは約 $\frac{1}{70}$ に抑えられていることが分かった．CORDIC を固定小数, 単精度浮動小数でソフトウェア実現した結果とでは，約 $\frac{1}{130}$ に抑えることができている．またコンパイ

表 2　RoCC とテイラー展開の実行命令数
Table 2　Number of instructions for RoCC and Taylor Expansion

Compile Optimaization	RoCC[count]			Tayler Expansion[count]		
	sin	cos	arctan	sin	cos	arctan
O0	68	68	77	5008	5013	5438
O1	42	42	43	3832	3832	4125
O2	42	42	43	3398	3418	3376

表 3　ソフトウェア実装した CODIC 固定小数と単精度浮動小数の実行命令数
Table 3　Number of instructions in fixed point number and single precision floating point number of CODIC in software

Compile Optimaization	Fixed-point CORDIC[count]			Float-point CORDIC[count]		
	sin	cos	arctan	sin	cos	arctan
O0	8863	8863	7890	35763	35763	34793
O1	6992	6992	6517	24286	24311	23681
O2	6922	6922	6374	21137	21132	20463

表 4　RoCC とテイラー展開の実行サイクル数
Table 4　Number of cycles for RoCC and Taylor Expansion

Compile Optimaization	RoCC[cycle]			Tayler Expansion[cycle]		
	sin	cos	arctan	sin	cos	arctan
O0	1574	1594	1621	21760	21760	21694
O1	1536	1536	1567	19057	19053	21184
O2	1520	1520	1557	16448	16634	16775

表 5　ソフトウェア実装した CODIC 固定小数の実行サイクル数
Table 5　Number of cycles in fixed point number of CODIC in software

Compile Optimaization	Fixed-point number CORDIC(Software)[cycle]		
	sin	cos	arctan
O0	18057	18057	14633
O1	15005	14989	13637
O2	14864	15055	13179

ラの最適化レベル O1 から O2 では RoCC の命令数への効果は無かった．次に実行サイクル数では，RoCC と他 3 種類の方法とでは，同じく大きく削減できていることが分かった．CORDIC をソフトウェア実現した方法の内，固定小数で実装した方法が浮動小数で実装した結果よりも短いサイクル数で実現することができた．これはシフト演算を浮動小数では for 文で実装していたことや，乗除算を有無からなどから差が出たのだと考えられる．これらのことから，ソフトウェアと比べ RoCC の回路を用いた場合の方がより速い速度で実行することができた．またコンパイラの各最適化後の結果と比較しても，専用命令追加による効果は大きいと考えられる．

表 6　ソフトウェア実装した CODIC 単精度浮動小数の実行サイクル数
Table 6　Number of cycles in single precision floating point number of CODIC in software

Compile Optimaization	Float-point number CORDIC[cycle]		
	sin	cos	arctan
O0	136206	136206	132772
O1	64463	64653	62334
O2	58814	58632	56956

5. まとめ

本論文では RISC-V, Rocket-Chip と CORDIC を用いることで，sin, cos, arctan 関数を実装した回路を作成した．そして性能比較として，ソフトウェアで実装した場合の実行も行い，RISC-V の CSR の値を参照することで調査を行った．結果，回路で実装した場合の方が実行命令, サイクル数の削減を行えることが分かった．またコンパイラの最適化を行ったソフトウェア実装の結果と比較しても，ハードウェアとして実装した場合の方がより性能を出しやすいことが知られた．

謝辞　本研究は JSPS 科研費 JP20H00606 の助成を受けている．また，東京大学大規模集積システム設計教育研究センターを通し，メンターグラフィクス株式会社，シノプシス株式会社の協力で行われたものである．

参考文献

[1] 金森拓斗, 宮崎広夢, 吉瀬謙二,「組み込みシステムに適した RISC-V ソフトプロセッサの設計と実装」, 情処究報, pp. 1–8, Vol.2020-EMB-54, 2020.
[2] Krste Asanović, et al., "The Rocket Chip Generator", Technical Report No. UCB/EECS-2016-17 (2016).
[3] Christopher Celio, et al., "BOOM v2: an open-source out-of-order RISC-V core", Technical Report No. UCB/EECS-2017-157 (2017).
[4] A.Waterman, et al., "The RISC-V Instruction Set Manual Volume I:Unprivileged ISA Document", University of California, Berkeley(2021).
[5] David A.Patterson, A.Waterman,「RISC-V 原典 オープンアーキテクチャのススメ」, 日経 BP(2018).
[6] David A.Patterson, John L.Hennessy, "COMPUTER ORGANIZATION AND DESIGN THE HARDWARE/SOFTWARE INTERFACE RISC-V EDITION", Morgan Kaufmann(2017).
[7] Jonathan Bachrach, et al.,「Chisel: Constructing Hardware in a Scala Embedded Language」, DAC Design Automation Conference 2012, p1216–1225 (2012).
[8] Jack E. Volder, "The CORDIC Trigonometric Computing Technique", IRE Transactions on Electronic Computers Volume EC-8, Number 3 (1959).
[9] Behrooz Parhami, "Computer Arithmetic ALGORITHMS AND HARDWARE DESIGNS", Oxford Univ Pr (1999).
[10] 中尾怜史, 武内良典,「RISC-V を用いた命令拡張のためのプロセッサ開発環境の検討」, DA シンポジウム 2021 論文集, p52-59, 2021.