The ultimate hands-on eBook:

—

Getting started with RISC-V by IAR(日本語版)

From curiosity to mastery

目次

このeBook について 4
前書き 5
著者について 5
1. はじめに 7
1.1 なぜRISC-V？ 7
1.2 オープンソースのISAとRISC-V 8
1.3 RISCとは? 8
1.4 RISC-Vの命令セット 9
1.4.1 RISC-Vの基本命令 9
1.4.2 拡張命令とカスタム命令 11
1.4.3 汎用レジスタと浮動小数点レジスタ 11
1.4.4 CSR(Control and Status Register) 12
1.4.5 動作モード 12
1.4.6 簡単なアセンブラ命令 13
1.5 RISC-Vのプロファイル 13
1.5.1 RISC-V profiles 14
1.5.2 RISC-V platform specification 15
1.6 なぜRISC-Vを使うと良いのか？ 15
1.7 本書の構成 16
2. EWRISCV開発環境の基本操作 18
2.1 EWRISCVを使う時の注意点 18
2.2 プロジェクトの作成（sample 1） 18
2.2.1 新規プロジェクトを作成して実行する 18
2.2.2 プロジェクトの構成 24
2.2.3 マニュアルについて 24
2.3 オプション設定 26
2.3.1 General Options（一般的なオプション） 26
2.3.2 C/C++ Compiler 32
2.3.3 Output converter 37
2.3.4 Linker 38

2.3.5 Debugger 41
2.4 EWRISCVのプロジェクト全体を理解する 42
2.4.1 サンプル2を作る 42
2.4.2 サンプルプログラムを実行 43
2.4.3 GP相対について 45
2.5 C拡張命令 48
2.6 M拡張命令 50
2.6.1 サンプル3をつくる 50
2.6.2 M拡張命令を有効にする 51
2.6.3 RV32Mの拡張命令 53
2.7 A拡張命令 54
2.7.1 サンプル4：A拡張命令を使ったプログラム 55
2.8 N extension instructions 57
2.9 カスタム命令 58
2.9.1 インフォメーションセンターのサンプルを開く 58
2.9.2 RISC-Vの命令コードについて 59
2.9.3 カスタム命令を決定する 59
2.9.4 C言語コードの中のカスタム命令 60
2.9.5 シミュレータをカスタム命令に対応させる 61
2.10 関数呼び出しABIについて 62
2.10.1 C言語の関数呼び出し 62
2.10.2 関数呼び出しの時のルール 63
2.11 EWRISCVの出力ファイルについて 64
2.11.1 Executables/libraries 64
2.11.2 Object files 64
2.11.3 List files 64
2.11.4 Browse files 64
2.11.5 MAPファイル 65
3. 実ハードを動かしてRISC-Vを理解する 77
3.1 GigaDevice社のRISC-Vマイコンを動かす 77
3.1.1 デバッグプローブを接続する 77
3.1.2 I-jetでの接続確認 79
3.1.3 サンプル5：GPIOを用いたLED点滅 81
3.1.4 デバッガの設定を行い、実行開始 86
3.1.4 RISC-Vの割り込みを理解する 88
3.1.5 CSRを確認しよう 94
3.2 ルネサスのFBP-R9A02G021でRISC-Vを使う 105
3.2.1 サンプルプロジェクトを作成しデバッグする 106
4. RTOS、ワークフローの自動化、コード品質について 112
5. おわりに 115
References 115

このeBookについて

インフィニオン、クアルコム、ノルディック・セミコンダクター、ボッシュ、NXPは、RISC-Vアーキテクチャベースの製品を加速するために共同出資する会社を設立しました。

また、2024年3月にルネサスは汎用RISC-V MCUの提供開始を発表しました。まさに今RISC-Vが熱を帯びてきており、世界中の組み込み開発者は迅速にスキルアップする必要があります。

本書「Getting Started with RISC-V by IAR」は究極のハンズオンガイドです。

長年にわたり、開発者はRISC-Vを高く評価して来ましたが、それは特にオープン・アーキテクチャ、カスタマイズ性、スケーラビリティ性、コミュニティサポートという点からです。SiFive、Andes、GigaDeviceなどの企業は、ビジネスとしてRISC-Vコアとデバイスの製造や販売といった点ではすでに軌道に乗っていますが、しかし、今までのところ商業的なリフトオフ(離陸)としては何か足りておらず、今一つという状況でした。

RISC-Vを取り巻くエコシステムは急速に成長・成熟しています。これは、最近のRISC-Vでのパートナーシップなどの発表を見ると明確です。より多くのデバイスが市場に投入されるにつれて、競争が激化するにつれて性能と効率の向上が期待されています。さらに重要な点として、ルネサスのような大手ベンダーは、自信を持って需要を予測しない限り、RISC-Vの分野に参入しません。これまでRISC-Vが提案されてから長い時間が掛かりましたが、ようやくRISC-Vが商業的に正しい位置へリフトアップするための土台が出来上がりました。
IARでは、過去40年間にもわたり、たくさんのアーキテクチャやデバイス向けのソリューションとしてツールチェーンを構築し、世界中の組み込み開発者や企業などに提供しています。

このガイドは、IAR Embedded Workbench for RISC-Vでのコンパイルやデバッガの使い方を中心に構成しています。このツールには静的解析ツールIAR C-STAT for RISC-Vも統合されています。
ガイドを読み進める際には、ソフトウェアの無料評価版をダウンロードして実際に使ってみる事を強くお勧めします。

IARのRISC-V向けソリューションでは、車載アプリケーション向けのISO 26262や産業用オートメーション向けのIEC 61508などの機能安全規格に準拠した機能安全対応版も提供をしています。
これらのソリューションは、開発者がGUIから操作することも可能ですし、自動化されたワークフローのCIパイプラインから使用する事が可能です。もちろん、その両方の組み合わせで操作できます。
ローカルなPCで操作するのか、仮想環境で実行するのか、クラウドで実行するのかについてはユーザが自分で決めることができます。
私たちはこれを「オープンチョイス(Open Choice)」と呼んでいます。

我々は、皆さんがこの無料ですが網羅的なRISC-V情報を記載したガイドを見てスキルアップしていただきたいと考えています。気軽に仲間と共有し、他に質問がある場合は遠慮なく連絡してください。
RISC-Vのヒントやコツ、ベストプラクティスをもっと詳しく知りたい方は、このガイドに記載した情報をもとにRISC-Vの世界に飛び込んでください。こちらから

皆さんがRISC-Vを理解してRISC-Vを活用する準備が出来ることを心から願っており、それと同時に、楽しいコーディングとスキルアップを願っています。

ニコラス ケールマン(Niklas Källman),
シニア・プロダクト・マネージャ(Senior Product Manager),
RISC-Vソリューション(RISC-V solutions),
IARシステムズ(IAR Systems)

前書き

本書は、RISC-Vを利用して組込みソフトウェアを開発する開発者や専門家（プログラマ）を対象としています。
本書では、IAR Embedded Workbench for RISC-V(以下、EWRISCV)を使用する際に必要となるRISC-Vの特徴と機能について説明します。そのためにRISC-Vを理解するために、RISC-VというCPUがもつ命令セットとスタックの動作も見ていきます。
読者は、CPU に関する基本的な知識を持ち、アセンブラ命令の基本的な機能を理解していることが期待されます。
さらに、C言語とアセンブリ言語を使用してRISC-Vの説明をします。それらの記述は、本書で説明するRISC-Vプログラミングのためのに必要なC言語プログラミングの理解する基礎となります。

アセンブリ言語に関しては、RISC-Vアーキテクチャに必要な基本に焦点を当てています。アセンブリ言語のより深い知識については、参考文献にあるRISC-Vの命令セットなどの資料を参照することをお勧めします。
IAR Embedded Workbench for RISC-Vの評価版は、https://www.iar.com/products/architectures/risc-v/iar-embedded-workbench-for-risc-v/ で無償で入手可能です。

本書では、IARのRISC-Vソリューションを使用しながら、RISC-Vアーキテクチャについて学ぶことをお勧めします。

著者について

赤星博輝は、IARのシニア・フィールド・アプリケーション・エンジニアです。
九州大学で情報工学の学士号(1991年)、修士号(1993年)、博士号(1996年)など、数多くの学位を取得しています。
研究対象は主にCPUアーキテクチャとコンパイラですが、組み込みソフトウェア開発での経験があります。
これまで、20年以上にわたり自動車業界の安全性と機能性に焦点を当てた幅広いプロジェクトで活動をしてきました。

1. はじめに

1. はじめに

1.1　なぜRISC-V?

RISC-Vは、オープンソースとして提供されている命令セットアーキテクチャ(ISA:Instruction Set Architecture) の名称です。RISC-Vの命令セットを採用したマイクロプロセッサやマイクロコントローラ（マイコン）なども「RISC-V」と呼ばれることが多いので注意が必要です。RISC-Vは、2010年にKrste Asanović 氏が、米国University of California, Berkeleyにて開発を始めました。同大学で開発された5番目のRISC命令セットアーキテクチャという意味で、「RISC-V」と命名されたそうです。RISCとは Reduced Instruction Set Computer の頭文字をとったCPUの設計方式の一つで、シンプルな命令セットを備えたコンピュータ（プロセッサ）を意味します。

このRISC-Vは最近になって急に誕生したものではなく、数十年の間、研究やビジネスに適用されたCPUの命令セットを分析し、検討して作られています。例えば、David A. Patterson 氏と John L. Hennessy 氏が共同執筆した名著"Computer Architecture: A Quantitative Approach"（邦題『コンピュータ・アーキテクチャ、設計・実現・評価の定量的アプローチ』）参考文献[1]が1990年に出版されています。この書籍の中で、RISC-Vの祖先にあたるDLXアーキテクチャが紹介されています。
DLXは、教育の現場ではそこそこ使われたと思いますが、ビジネスの世界では使われませんでした。当時は、Hennessy 氏がMIPSアーキテクチャに基づくプロセッサを開発するMIPS Computer Systems社を立ち上げたため、DLXをビジネスに適用する動きは、あまりなかったと思います。その後、RISC方式のCPUアーキテクチャの技術は大いに発展し、高性能なプロセッサが続々と開発されました。このように、今のRISC-Vにつながる基本的なコンセプトは、30年前から存在しています。
そして、もう一人の著者であるPatterson 氏がRISC-Vの開発や普及推進を後押ししている、という点も、RISC-Vに期待が集まる大きなポイントとなっています。同氏は現在、RISC-Vについての情報発信などを行っています（参考文献 [7] ）。整理すると、CPU技術の有識者が集まり、既存の命令セットが抱えていた問題点などを踏まえて、新たに開発・リリースされたのがRISC-Vなのです。例えば、以下の点が改善されました

- アドレス空間の拡大に無理やり対応してきたこれまでのCPUとは異なり、32ビット、64ビット、128ビットの処理が最初から検討されている
- 命令セットをモジュラ構造にして、必要なものを実装し、不要なものを実装しない、という選択が行えるようになっており、限られたハードウェアリソースを効率よく使うことができる
- 遅延分岐や遅延ロードといった単一命令実行のパイプラインを前提とした仕組みを排除した

RISC-V普及のため、RISC-V Foundationという団体が2015年に設立され、同団体は2019年にスイスへ拠点を変えました。現在、RISC-V InternationalがRISC-VのISAの仕様を公開し、仕様策定はRISC-V Internationalの会員が行っています。
では、なぜ今、RISC-Vが大きな話題となっているのでしょう？ それはRISC-VのISAがオープンソースとして公開されているためです。仕様書は Creative Commons Attribution 4.0 International のライセンスに基づいて提供されており、配布や変更なども可能となっています。
ここで注意していただきたいのは、ISAの仕様がオープンというだけで、ハードウェアやその設計情報がタダ（無償配布）というわけではないことです。ISAがオープンになっているので、誰がその仕様に準拠したプロセッサを作っても差し支えありません。実際、多くの企業や研究機関、大学などがRISC-VのCPUコアを実装しています。

大学などを中心に、RISC-V仕様に準拠したCPUコアの設計情報（設計された回路の情報やそのEDAデータ）がオープンソースとして公開されているケースも、確かにあります。しかし、すべてのCPUコアの設計情報が公開されているわけではなく、またすべてのCPUコアの設計情報がタダというわけでもありません。実際、RISC-VのCPUコアを有償で販売している会社はいくつもあります。例えば、台湾Andes Technology社やドイツCodasip社、米国SiFive社などです。命令セットが決まっているのであれば、実装は似たようなものになるのでは？ と思われるかもしれませんが、そんなことはありません。例えば、パイプラインの段数や同時発行命令数、キャッシュメモリの構成や容量、メモリアクセスの方式、分岐予測などの仕様を各社が独自に策定し、それぞれ異なる特徴を持つCPUコアを設計しています。

オープンソースというと、GCC（GNU Compiler Collection）を思い浮かべる方も多いと思います。RISC-VもGCCに対応しており、Linuxなどを利用する時にはGCCが必須となります。RISC-Vのソフトウェアを作成する際の開発環境として、「GCCがあれば十分」と思われている方が多いのですが、組込みシステムの開発ではそれほど話が単純ではありません。状況に応じて必要とされるコンパイラが異なります。
例えば、自動車や産業用ロボットのように機能安全が要求されるシステムの開発ではツール認証を求められます。本来の業務の合間にコンパイラの検証を行うことは、なかなか大変な作業になります。
IARでは第三者機関による認定を受けたコンパイラを販売しています。本製品を使用すれば、ユーザによるツール認証作業の手間を大幅に軽減できます。
IAR Systems社のコンパイラは組込みソフトウェアの開発に最適化されている点にあり、生成されるコードのサイズや実行速度で優位な点もメリットとなります。

1.2 オープンソースのISAとRISC-V

上述したように、RISC-Vの特徴の一つは「オープンソースとして提供されているISAであること」ですが、それには具体的にどのようなメリットがあるのでしょう？世の中で、チップとして売られているプロセッサ製品の多くは、特定の企業がそのISAの権利を保有しています。

例えば、ルネサス エレクトロニクスの32ビットマイコンである「RXファミリ」は、同社が独自のISAを定義し、実装し、チップとして販売しています。英国Arm社のCortex-M/Cortex-R/Cortex-Aの場合は、Arm社が保有するIP（Intellectual Property：知的財産権）を半導体メーカにライセンス供与し、そのIPを使って半導体メーカがチップを作り、販売しています。このようなMCUやMPUはメーカ固有のISAをもっており、ユーザが命令セットを勝手に追加・改変することはできません。

契約の形態には、Arm社が設計・実装したCPUコア（の設計データ）をそのまま使う通常のライセンスと、Arm社が定義したISAの仕様に基づいて半導体メーカがCPUコアを設計・実装するArchitecture Licenseがあります。2020年には、ISAの仕様を部分的にカスタマイズできるようにしたCortex-X Custom Programも登場しています。
ベンダ固有のISAに対して、RISC-Vはオープンソース ISAとしてメリットがあります。例えば、アプリケーションコードを効率的に実行するため、ユーザが特殊な命令を追加する、といったことが可能です。ベンダ固有のISAでは、ユーザが命令を追加することは一般的には困難です。Arm社の場合、契約自体は可能ですが、ユーザがそれをすることは難しいと言う事です。

しかし、ISAの仕様がオープンなCPUのプロジェクトは少なからず存在します。中には、CPUコアの設計データが無償公開されているものもあります。なぜ、RISC-Vがここまで注目を集めているのでしょう？一つの理由は、RISC-Vでは、エコシステム（生態系）と呼ばれるコミュニティの構築に成功したからです。2023年までに、ハードウェアからソフトウェアに至るまで、多くの企業がRISC-Vのエコシステムに参加しています。例えば、IAR Systems社は2019年にRISC-V向けの独自開発のコンパイラをリリースし、2023年3月までに11回のバージョンアップを実施しています。さらに、RISC-V向けの機能安全認証取得済みコンパイラを2021年と2023年にリリースしています。

統合開発環境のEWRISCVについては、Azure RTOS ThreadXやFreeRTOS example projects、SAFERTOSなどのサンプルを導入できるようになっています。
コンパイラとリアルタイムOSのサポートがきちんとしているので、ユーザは安心して組込みシステムの開発を始められます。上述のIARの話は、エコシステムの一例に過ぎません。企業同士、関係者同士が相互に補完し合うエコシステムが存在することで、ユーザはより包括的なサービスを受けられるようになっています
RISC-Vのビジネスが長期的に成立しなければ、エコシステムに参加する企業は増えません。コミュニティの構築がうまくいっているということは、オープンなISAであるRISC-Vがビジネスの対象として認知されていることのあかしである、と言えます。

1.3RISCとは?

RISC-VのRISCとは、Reduced Instruction Set Computerの頭文字をとった言葉です。RISCとは何か？ という疑問に、簡単に答えておきます。
CPU（コンピュータ）の世界では、あらかじめ命令セットが定義されており、その命令セットを利用して（命令を並べて）プログラムを作っていきます。より高速に、より高機能な処理をCPUに実行させるため、多くのCPU開発者は命令の数をどんどん増やしていきました。
ソフトウェアが、高級言語とコンパイラを使って開発されるようになり、高級言語の構文をそのまま実行できる命令も追加されました。こうした方針に基づいて開発されたCPU（の設計方式）は、複雑な命令を追加したという意味で、Complex Instruction Set Computer（CISC）と呼ばれるようになりました。

しかしその後、コードの実行履歴を解析するなど、きちんと研究が進んでいくと、複雑な命令はほとんど使われておらず、コンパイラがそうした命令をうまく活用できていないことが分かってきました。

それなら、シンプルな命令のみを実装するほうがCPUの設計が簡素になりますし、動作速度も引き上げやすくなります。なくなった複雑な命令の部分はどう処理するのか？ という点については、シンプルな命令を組み合わせて複雑な処理を実現していくことになります。

このようにISAをシンプルな命令に限定することで、より高速な処理を実現できるCPUを開発する動きが、1980年ぐらいから盛んになり、RISC方式のCPUが広く普及しました。例えば、SPARCやMIPS R2000/R3000、Alpha、i860/i960、PA-RISC、PowerPCなどのアーキテクチャに基づくプロセッサ製品が市場に投入されました。
コンピュータ科学の教科書として、前述の“Computer Architecture : A Quantitative Approach” の初版が出版されたのもこの頃（1990年）で、1992年には日本語訳も出版されています。著者の一人であるPatterson教授が、RISC-Vの開発に関わっているのです。

CPUの高速化が進むと、CPUの動作速度とメモリのアクセス速度の差が広がっていきました。いくらCPUを高速にしても、メモリからデータを取得する時やメモリにデータを書き込む時に待ち状態が生じることが問題となってきたのです。これをフォンノイマンボトルネックと呼びます。
CISC方式のCPUでは、メモリ上のデータを操作する命令が用意されていることが普通ですが、メモリへのアクセスが遅いと、CPUもそのアクセスに影響され、動作速度が低下します。
RISC方式のCPUでは、汎用的に使えるレジスタを用意し、演算はレジスタに対して実行することが一般的で、これはロードストアアーキテクチャと呼ばれています。多くのRISC CPUが採用しているロードストアアーキテクチャですが、CPUの命令としてはメモリからデータを読み出すLoad命令と、メモリにデータを書き込むStore命令を持ちます。メモリにアクセスするのはLoad命令やStore命令で、加算やビット演算などの処理はレジスタに対して行います。

CPUがさらに高速化になると、CPUとメインメモリの間にキャッシュなどの高速なサブメモリ（ただし、サイズは小さい）を導入し、CPUの動作速度とメモリのアクセス速度の格差を埋めていく、という方法が採られています。
最近のCPUは、1次キャッシュ、2次キャッシュ、...というように、複数階層のキャッシュメモリを備えていることが珍しくありません。

1.4 RISC-Vの命令セット

1.4.1 RISC-Vの基本命令

RISC-Vの命令セットでは、基本命令と拡張命令が定義されており、さらにカスタム命令を追加することが出来ます。ここでは、それらを確認していきます。
アドレスについては、32ビット、64ビット、128ビットの3種類が用意されています。汎用レジスタは、32本のものに加えて、小さなマイコンを実現する際に利用される16本のものも用意されています。

- RV32I (32ビットアドレス, 整数演算, 32本の汎用レジスタ)
- RV64I (64ビットアドレス, 整数演算, 32本の汎用レジスタ)
- RV128I (128ビットアドレス, 整数演算, 32本の汎用レジスタ)
- RV32E (32ビットアドレス, 整数演算, 16本の汎用レジスタ)

現状、128ビットのRV128Iのニーズは少ないかもしれませんが、将来性を考えて128ビットまで基本命令に追加されたようです。既存のCPUの命令セットは、32ビットをベースとした構成に拡張を加えたものが多く、増改築を繰り返して、いびつな構造になってしまっているものも珍しくありません。これに対してRISC-Vは、32ビット、64ビット、128ビットのアドレスを最初からまとめて考慮することで、きれいな命令セットを作っています。これは、後発ならではの利点と言えます。さらに言うと、RISC-Vの命令については、今後の拡張の余地も残してあります。

まず、RISC-Vの命令の基本となる命令フォーマットですが、六つの命令フォーマットがあります。opcodeとfunctで命令を、rd、rs1、rs2でレジスタを指定します。immは即値データを表します。注目していただきたい点は、いずれのタイプでも、rd、rs1、rs2が32ビットの同じ位置に配置されているところです。このような特徴は、CPUハードウェアの設計を容易にします。

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

OPコード（opcode）の上位5ビット（6〜2ビットの部分）の割り当ては、以下のように決められています。下位2ビットは、inst[1:0]=11となっています。

inst[6:5] inst[4:2]	000	001	010	011	100	101	110	111
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/ rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/ rv128	>=80b

RV32Iが備える各種Load命令を見てみると、以下のビット割り当てになっています。inst[6:2]が00000になっていることを確認できると思います。
Load命令のバイトデータ（signedとunsigned）やハーフワードデータ（signedとunsigned）、ワードデータについては、inst[14:12]の部分で区別されていることが分かります。

	31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
lb (load byte)	offset12 [11:0]	rs1	000	rd	0000011
lbu (load byte unsigned)	offset12 [11:0]	rs1	100	rd	0000011
lh (load halfword)	offset12 [11:0]	rs1	001	rd	0000011
lhu (load halfword unsigned)	offset12 [11:0]	rs1	101	rd	0000011
lw (load word)	offset12 [11:0]	rs1	010	rd	0000011

1.4.2 拡張命令とカスタム命令

RISC-Vでは、拡張命令やカスタム命令を持つことが可能です。この点がRISC-Vのユニークな特徴となっており、この仕組みはモジュラ構造と呼ばれています。新たな拡張命令が、RISC-V Internationalに続々と提案されており、さまざまな命令について議論されたり、正式な規格となったりしています。
例えば、2019年のISA Specifications（Volume 1, Unprivileged Specification version 20191213[参考文献4]）を見ると、拡張命令について以下のように記載されています。

拡張命令	バージョン	ステータス
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
Counters	2.0	Ratified
L	0.0	Draft
B	0.0	Draft
J	0.0	Draft
T	0.0	Draft
P	0.2	Draft
V	0.7	Draft
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
Zam	0.1	Draft
Ztso	0.1	Frozen

表のStatusの意味は、以下のとおりです。
- Ratified：承認された。仕様変更は認められない
- Frozen：承認なく仕様を変更しない。クリティカルなケースがあれば、仕様を変更する可能性がある
- Draft：承認まで仕様を変更する可能性がある

当然、時間が経過するとステータスは変化します。常に最新の資料を確認するようにしてください。

1.4.3　汎用レジスタと浮動小数点レジスタ

RISC-Vは汎用レジスタ方式を採用しています。基本命令のRV32IやRV64I、RV128Iについては、整数レジスタを32本使用可能です。ただし、そのうちの一つは、常に値が 0 であるゼロレジスタになります。RV32Eについては、小さなマイコン向けということで、整数レジスタを16本のみ使用可能です。

RISC-Vでは、整数レジスタはレジスタ名とABI（Application Binary Interface）名の二つの名前を持っています。レジスタ名はx0〜x31で、ABI名はzero、ra、sp、gp、tp、t0〜t6、a0〜a7、s0〜s11となります。ABI名とレジスタの役割の対応は、以下のとおりです。

- zero：読み出しは常に 0 で、書き込みは無視するレジスタ。よって常に 0 を返す
- ra：関数呼び出しの時、戻り先アドレスをraに設定して呼び出すレジスタ
- sp：スタックポインタ
- gp：グローバルポインタで、メモリアクセスのために使用されるレジスタ。このgpを使ってメモリアクセスを行う際には、リンカ設定が必要となる
- tp：スレッドポインタ
- t0-t6：テンポラリレジスタ
- a0-a7：関数の引き数や戻り値に使用するレジスタ
- S0-S11 ：保存レジスタ

レジスタ名	ABI名	内容
x0	zero	ゼロレジスタ
x1	ra	戻りレジスタ
x2	sp	スタックポインタ
x3	gp	グローバルポインタ
x4	tp	スレッドポインタ
x5,x6,x7	t0,t1,t2	テンポラリレジスタ
x8	s0/fp	保存レジスタ または フレームポインタ
x9	s1	保存レジスタ
x10,x11	a0,a1	関数引数、戻り値
x12,x13,x14, x15,x16,x17	a2,a3,a4, a5,a6,a7	関数の引数
x18,x19,x20, x21,x22	s2,s3,s4, s5,s6	保存レジスタ
x23,x24,x25, x26,x27	s7,s8,s9, s10,s11	保存レジスタ
x28,x29, x30,x31	t3,t4,t5,t6	テンポラリレジスタ

レジスタ名のx0〜x31に対して、ABI名のグループがバラけていると感じるかもしれません。具体的には、保存レジスタの並びが複数に分割されています。これは、RV32Eのための工夫です。RV32Eはレジスタを16本しか使用しないので、レジスタの長さが半分になったときでも効率よくプログラムを動かせるように割り当てられています。

RISC-Vでは、拡張命令Fで単精度の浮動小数点演算命令を、拡張命令Dで倍精度の浮動小数点演算命令を扱えます。仕様上は、4倍精度の拡張命令Qや、整数レジスタを使うZfinx、Zdinxなどもありますが、ここでは基本のFとDについて述べます。浮動小数点レジスタは、32本のレジスタを持っています。単精度の場合は下位32ビットを、倍精度の場合は64ビットを使用します。
浮動小数点レジスタの名前は、単精度の場合がf0〜f31、倍精度の場合がd0〜d31となります。

1.4.4 CSR(Control and Status Register)

RISC-Vには、CSR（Control and Status Register）というレジスタ群が用意されており、これらのレジスタは12ビットの値で指定します。
参考文献 [5]には、さまざまな制御レジスタが定義されています。ただし、チップによって実装されているCSRの機能に差異がある点には注意が必要です。
詳しくは、RISC-Vのドキュメントや使用するマイコンのデータシートなどを確認してください。
その中の重要なものの一部を、以下に示します。3章でGigaDevice Semiconductor社のRISC-Vマイコンを動かしながら説明します。

0xF11	mvendorid	ベンダID
0xF12	marchid	アーキテクチャID
0xF13	mimpid	実装ID
0xF14	mhartid	ハードウェアスレッドID
0x300	mstatus	マシン状態レジスタ
0x301	misa	ISAと拡張
0x304	mie	マシン割り込み許可レジスタ
0x305	mtvec	マシントラップハンドラベースアドレス
0x341	mepc	マシン例外プログラムカウンタ
0x342	mcause	マシントラップ要因
0x343	mtval	マシン不良アドレス/命令
0x344	mip	マシン割り込み保留

1.4.5　動作モード

RISC-Vでは、CPUの特権と動作モードの仕様が定義されています（参考文献 [5] ）。現在のマイクロコントローラでは、マシンモードとユーザモードを見る事ができます。
定義された動作モードを、以下に示します。

レベル	2進エンコード	動作モード
0	00	ユーザ/アプリケーションモード (U)
1	01	スーパーバイザモード (S)
3	11	マシンモード (M)

マシンモードが最も特権レベルが高く、マシンモードで実行するものは、信頼されたソフトウェアやデバイスドライバなどが想定されます。
また、マシンモードはどのような実装でも実現する必要がありますが、その他のモードはオプションとなっています。実装される組み合わせには、下表の三つがあります。

サポートレベル数	サポートモード	想定する使い方
1	M	シンプルな組込みシステム
2	M,U	セキュアな組込みシステム
3	M,S,U	Unix, android, or Windows

基本的にユーザモードでは、割り込みを受けることが出来ません。ユーザモードで割り込みが入ると、マシンモードに遷移します（移譲する方法などもあるが、ここでは説明を省略する）。マシンモードの割り込み処理からユーザモードへ戻る際には、mretという命令を使います。

1.4.6　簡単なアセンブラ命令

最近では、アセンブリ言語でプログラムを作る人は多くないと思います。しかし、リアルタイムOSを実装したい場合やプログラムの性能を最適化したい場合は、今でもアセンブラが使われます。また、RISC-Vには拡張命令やカスタム命令があります。RISC-Vを使用する場合は、最低限のアセンブラの知識を身につけておくべきです。基本的には、演算とメモリアクセス、関数の呼び出し/戻りを憶えておけばよいでしょう。まず演算の例として、加算add命令と加算即値addi命令の二つを見てみましょう。

- add a0, a1, a2
- addi a0, a1, 1

命令addは三つのレジスタを指定し、a0が結果の格納レジスタ、a1が第1引き数、a2が第2引き数となります。数式で書くと、a0=a1+a2という感じとなり、アセンブリコードの記述順と一致します。この命令により、レジスタa1とa2の内容を加算してa0に書き込みます。

命令addiは、a0が結果の格納レジスタ、a1が第1引き数、1が即値となります。数式で書くと、a0=a1+1という感じで、レジスタa1に1を加算してa0に書き込みます。
次に、メモリアクセスのlw命令とsw命令について説明します。

- lw a0, -8(gp)
- sw a0, -8(gp)

命令lwはload wordの意味になり、メモリからwordデータを取得し、レジスタに格納します。メモリのアドレスは、レジスタにオフセットを加えたものになります。ここではgpの内容に-8したアドレスとなり、そのアドレスの示す内容をa0に格納します。

命令swはstore wordの意味になり、レジスタの値をメモリに書き込みます。メモリのアドレスは、命令lwの場合と同じで、レジスタにオフセットを加えたものになります。ここではgpの内容に-8したアドレスとなり、そのアドレスにレジスタa0の内容を書き込みます。
このようにメモリアクセス命令の場合、第1引き数がレジスタ、第2引き数がメモリアドレスになります。

最後にジャンプ命令です。

- jal ra, 0x18 (call 0x18 pseudo instruction)
- jalr zero, ra, 0x0 (ret pseudo instruction)

命令jalは、次の命令のアドレスをレジスタraに保存し、今のPC（Program Counter）の値に対して即値の0x18を加算したアドレスにPCを設定（ジャンプ）します。
call疑似命令の call 0x18 とまったく同じ意味になります。
命令jalrは、次の命令をzeroレジスタに保存し、レジスタraの値にオフセット0x0を加算したアドレスにPCを設定します。
RISC-Vでは、zeroレジスタへの書き込みは無意味なので、単なる関数から戻るときに使用します。そのため、ret疑似命令と同じ意味になります。

アセンブラは、最初は取っ付きにくいと思われるかもしれませんが、アセンブラを使ったり、アセンブリコードを見たりする機会が増えると、だんだん慣れてきます。アセンブラに早く慣れるためには、RISC-Vの規格書などを使って学習するとよいでしょう。

1.5　RISC-Vのプロファイル

RISC-Vの特徴の一つとして、命令セットがモジュラ構造になっており、拡張命令を選択できる、という話をしました。しかし、あまりにも自由に命令セットのモジュールを作っていくと、互換性のないものがたくさん出来てしまう可能性があります。実は、David A. Patterson氏はこうした懸念についても解説しています。

同氏が執筆したブログ記事 “Top Ten Fallacies About RISC-V”（RISC-Vに関する誤解のトップ10、参考文献 [3]）の6番目の「6. RISC-Vのモジュラ構造はソフトウェアエコシステムの細分化をもたらす」という項目で、「この誤解は、RISC-Vを提唱し始めた当初から提起されてきたものであり、無視されたわけではありません。一部の市場セグメントは、安定したISAとバイナリ互換性を要求しますが、RISC-Vはこの問題にプロファイルで対処します」と説明されています。

以下では、RISC-VのISAプロファイルについて紹介します。

1.5.1 RISC-V ISA profiles

現在、RISC-VのISAプロファイルとして、RVI20 Profiles、RVA20 Profiles、RVA22 Profilesが定義されています（2023年時点）。"Profiles"が複数形になっていますが、具体的には以下の6通りの仕様が定義されています。

- RVI20U32
- RVI20U64
- RVA20U64
- RVA20S64
- RVA22U64
- RVA22S64

最初の2文字の"RV"はRISC-Vを意味し、3文字目はプロファイルのファミリ名を示します。ファミリ名には、以下のものがあります。

- RVI: INTEGER(整数系)
- RVM: MICROCONTOLLER（MCU向け）
- RVA: Application（アプリケーション）

4、5文字目の2けたの数字は、このプロファイルが承認された年を示します。6文字目は動作モード（M：マシンモード、S：スーパーバイザモード、U：ユーザモード）を指定しています。7、8文字目の2けたの数字は、32ビット、64ビットを示します。それぞれのプロファイルは、Mandatory Base（必須の基本命令）、Mandatory Extensions（必須の拡張命令）、Optional Extensions（オプションの拡張命令）の三つの要素から構成されています。
例えば、RVI20U32プロファイルの仕様を見てみると、以下のようになっています。

- Mandatory Baseについては、RV32Iでリトルエンディアンが指定されている。さらに、fence.tso命令（Fence命令、Total Store Orderingモード）のサポートが必須
- Mandatory Extensionsはなし
- Optional Extensionsは、以下のとおり
  - > M Integer multiplication and division.
  - > A Atomic instructions.
  - > F Single-precision floating-point instructions.
  - > D Double-precision floating-point instructions.
  - > C Compressed Instructions.
  - > Zifencei Instruction-fetch fence instruction.
  - > Misaligned loads and stores may be supported.
  - > Zicntr Basic counters.
  - > Zihpm Hardware performance counters.

次に、RVA22U64プロファイルについても見てみましょう。RVA22U64の仕様は、以下のようになっています。

- Mandatory Baseについては、RV64Iでリトルエンディアンが指定されている。さらに、fence.tso命令のサポートが必須
- Mandatory Extensionsは、以下のとおり
  - > M Integer multiplication and division.
  - > A Atomic instructions.
  - > F Single-precision floating-point instructions.
  - > D Double-precision floating-point instructions.
  - > C Compressed Instructions.
  - > Zicsr CSR instructions. The presence of F implies these.
  - > Zicntr Base counters and timers.
  - > Zihpm Hardware performance counters.
  - > Ziccif Main memory regions with both the cacheability and coherence PMAs must support instruction fetch, and any instruction fetches of naturally aligned power-of-2 sizes up to min(ILEN,XLEN) (i.e., 32 bits for RVA22) are atomic.
  - > Ziccrse Main memory regions with cacheability and coherence PMAs must support RsrvEventual.
  - > Ziccamoa Main memory regions with cacheability and coherence PMAs must support AMOArithmetic.
  - > Zicclsm Misaligned loads and stores to main memory regions with both the cacheability and coherence PMAs must be supported.
  - > Za64rs Reservation sets are contiguous, naturally aligned, and have a maximum of 64 bytes.
  - > Zihintpause Pause instruction.
  - > Zba Address computation.
  - > Zbb Basic bit manipulation.
  - > Zbs Single-bit instructions.
  - > Zic64b Cache blocks must be 64 bytes in size, naturally aligned in the address space.
  - > Zicbom Cache-Block Management Operations.
  - > Zicbop Cache-Block Prefetch Operations.
  - > Zicboz Cache-Block Zero Operations.
  - > Zfhmin Half-Precision Floating-point transfer and convert.
  - > Zkt Data-independent execution time.
- Optional Extensionsは、以下のとおり
  - > Zfh Half-Precision Floating-Point.
  - > V Vector Extension.
  - > Zkn Scalar Crypto NIST Algorithms.
  - > Zks Scalar Crypto ShangMi Algorithms.

1.5.2 RISC-V platform specification

RISC-VのISAプロファイルは、上述のように拡張命令などの指定になっており、ある意味、単純なリストで示されています。それに加えて、RISC-V Platform Specificationと呼ぶ仕様が定義されています。筆者としては、このプラットフォームの概念があって初めて、意味のあるプロファイルになると考えています。現状、プラットフォームのプロファイルには、LinuxやWindowsといった比較的リッチなOSの利用を想定したOS-A Platformと、組込み向けのマイコンで動作するリアルタイムOSの利用やベアメタルのシステムを想定したM Platformの2種類があります。

このうち、OS-A Platformについては、OS-A Embedded Platformの仕様とOS-A Server Platformの仕様が定義されています。一方、M Platformについては、Baseの仕様とPhysical Memory Protection（PMP）Extensionの仕様が定義されています。プラットフォームとISAプロファイルの関係ですが、OS-A Platformについては、RVA22U、RVA22S のプロファイルへの対応が要求されており、M Platformについては、RVM22Mのプロファイルへの対応が要求されています。ただし、現在はRVAの仕様検討を優先しているようで、RVM22Mについては、まだ資料がない状態です。このあたりは、LinuxやAndroidへの対応を急いでいるためのようです。RISC-Vマイコンを使ったり、自分たちでRISC-Vコアを作ったりする場合、規格標準化の動向を理解しておくことも重要になります。

1.6 なぜRISC-Vを使うと良いのか？

ここまで、RISC-VとそのISAの概要を、かいつまんで説明しました。例えばRISC-VのISAは、基本命令だけでなく、拡張命令やカスタム命令を利用できるので、さまざまなシステムの要求に対応しやすい、という特徴がありました。また、後発の新しいISAなので、構造がきれい（きれいな命令は実装が容易）という特徴もありました。RISC-Vの利点は「性能改善の選択肢が増える」事があります。プロセッサのパフォーマンスは、プログラムの実行にかかる時間（ランタイム）で表現できます。これを以下のように表します。

- パフォーマンス ＝ (時間 / プログラム)

ここで言うパフォーマンスは、一つのプログラムをどのくらいの時間で実行できるか、という指標です（具体的なCPUの処理速度や消費電力などではないことに注意）。ここで、さきほどの式の右辺の要素を以下のように3つの要素に分解します。

- パフォーマンス ＝(命令数 / プログラム) × (サイクル数 / 命令) × (時間 / サイクル)

ここで右辺の左側の (命令数 / プログラム) は、一つのプログラムを何個の命令で実行できるかです。右辺の中央の (サイクル数 / 命令) は、一つの命令を何サイクルで実行できるかです。右辺の右側の (時間 / サイクル数) は、1クロックサイクルを何秒で実行できるか（サイクルタイム）に対応します。これらについて考えてみましょう。

1. (命令数 / プログラム)は、命令セットアーキテクチャ（ISA）で決まります.
2. (サイクル数 / 命令)はCPUのマイクロアーキテクチャ（パイプランの段数やスーパースカラ構造、キャッシュ構成など）で決まります.
3. (時間 / サイクル数)は半導体の製造技術の影響を受けます。ここの性能を上げるには、高価な微細プロセス（シリコン上の回路の線幅が小さいプロセス）を採用する必要があります。

RISC-Vの場合、ユーザにより1と2（ISAとマイクロアーキテクチャ）の変更が可能です。1と2を変える事で、場合によっては、3の安価な半導体プロセスで実装できるケースも出てくるかもしれません。そのため、性能を決定する3つの要素をユーザによりすべて変更・選択ができます。これは、設計・製造で高い自由度があることになります。

では、Arm社のCPUの場合はどうでしょう？ この場合、Arm社から購入した（ライセンス調達した）CPUコアをそのまま使うケースがほとんどです（カスタマイズには別の契約が必要です）。性能を引き上げようとすると、多くの場合、半導体の製造プロセスを変更する方法を採ることになります。

しかし、RISC-Vでは新たな命令を追加したり、パイプライン構成を変更したりすることが可能です。例えば、アプリケーションの中に特殊な演算処理がある場合、その演算を直接実行するカスタム命令を追加することで、システムの性能を引き上げることが出来ます。既存のCPUコアでは命令追加することができないので、製品を差異化しやすいという利点があります。仮に競合他社が似たような製品を作ろうとしても、同じクロック周波数でシステムを実現することは難しいと考えられます。

性能改善のため、既存のCPUコアとは異なる手法を選択できることは、RISC-Vの大きな強みになっています。半導体を自社開発する場合、もしくはFPGA（Field Programmable Gate Array）を使ってカスタムハードウェアを実現する場合、CPUコアそのものに手を加えることで、システムの処理性能や消費電力を大幅に改善できるケースがあります。
近年、競争力のある製品を作るため、競争力のある半導体を自社で作ろう、という動きが目立っています。代表的な例は米国のApple社とGoogle社です。Apple社は自社専用のiOS、Google社はオープンなAndroidというように、それぞれ異なる戦略でビジネスを展開していますが、独自の半導体を開発して自社製品に組み込んでいる点は共通しています。競争力を高めるための半導体の自社開発について、今後、業界全体がどのような方向へ進んでいくのか、筆者は注目しています。

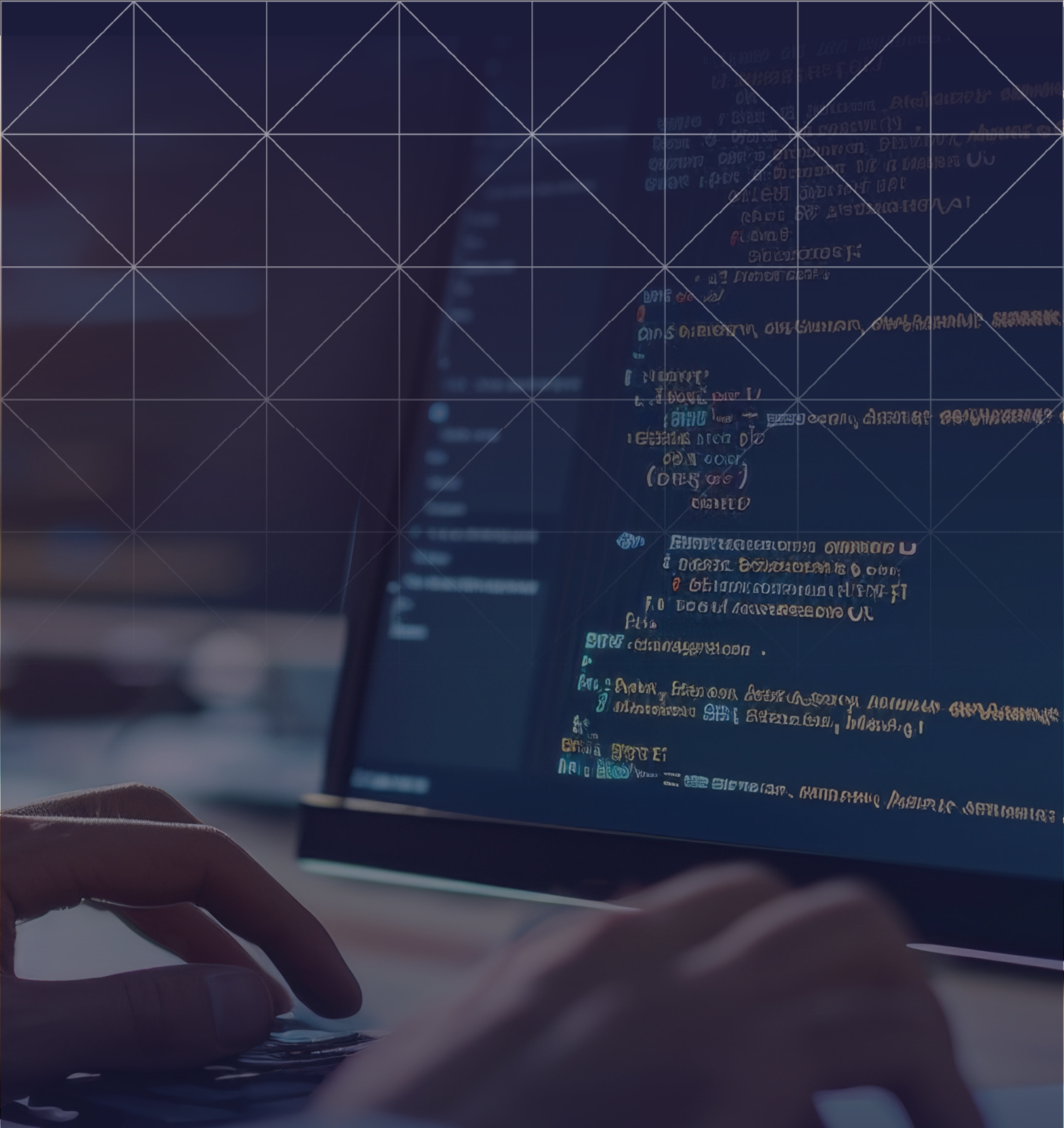
最後に、RISC-V採用の三つ目の利点である「課題の先送りや将来への備えが可能になる」という点について説明します。RISC-Vの利点は理解できるが、今すぐカスタム命令が必要なわけではない、あるいは必要性を感じることはあっても、今すぐ自分たちでCPUコアを作ったりすることは出来ない、という方がほとんどだと思います。しかし、将来、カスタム命令を使うかどうか分からない段階であっても、RISC-Vを採用する意味はあると考えます。それは、将来の性能改善やプラットフォームの更新に備えるためです。
いま自分たちにカスタム命令を使用する計画が無くても、RISC-Vを使っておけば、将来に拡張命令を追加することが出来るのです。新しい命令を追加することで、システムの性能が数倍になるということは珍しくありません。来たるべき時に備えて、公開されているRISC-Vプロセッサの実装を調査するなどして、RISC-Vのノウハウを蓄積することは重要だと思います。
RISC-VのCPUコアを自分たちで作れない場合は、市販されているCPUコアを購入する、という手もあります。

ここで視点を変えてみましょう。ビジネスから国際問題に話を広げてみます。かつて、東西冷戦の時代に、COCOM（Coordinating Committee for Export to Communist Area；対共産圏輸出統制委員会）という、軍事技術や戦略物資の輸出規制を行う国際機関がありました。これは1949年に創設され、1950年に活動を開始し、冷戦の終結などを受けて1994年に解散しました。「ココム違反」で検索すると、さまざまな事件が起こっていたことを確認できます。

COCOMは昔の話ですが、現在は米中貿易摩擦の問題が起こっています。実際、CPU技術を中国に輸出できないケースが発生しています。こうした国家間の対立が、皆さんの開発する製品の流通や部品調達に影響を及ぼす可能性は、決してゼロではありません。このような観点から、オープンソースのISAであるRISC-Vの採用を検討する価値はあると思います。特定のプロセッサを搭載した製品が輸出できなくなった場合でも、RISC-Vであれば、現地で互換性のある部品を入手できる可能性が高いからです。

1.7 本書の構成

これまでの説明により、RISC-Vについて、半導体メーカがこれまで提供してきたプロセッサのISAと異なる点を理解していただけたのではないかと思います。
2章では、IAR Systems社の統合開発環境であるEmbedded Workbench for RISC-Vを使用して、実際に生成されるRISC-Vの命令セット、およびRISC-Vマイコンを動かす時のポイントを確認していきます。



2. EWRISCV開発環境の基本操作

iar

2. EWRISCV開発環境の基本操作

RISC-Vを学習するためには、プログラムを作って実行することが一番の早道です。ここでは、IARの開発環境を使用します。IAR Embedded Workbench for RISC-V (以降、EWRISCVと省略)は、RISC-V向けの組み込みシステム開発ツールとなります。本章では、EWRISCVの使い方を説明しながら、RISC-Vの学習を進めます。EWRISCVは、グラフィカルなGUI (Graphical User Interface) 操作が基本の統合開発環境です。

コマンドラインで実行するためのiarbuild.exeというツールもあるのですが、開発環境上でオプション設定を行いながら実行するのが基本的な使い方となります。コンパイラ本体はiccriscv.exe、リンカ本体はilinkriscv.exeなど、ツールが個別に用意されているので、一般的な make コマンドを使って操作することも可能です。本書では、GUI操作による使い方を説明していきます。

2.1　EWRISCVを使う時の注意点

最初に、EWRISCVを使う場合の注意点を記載します。EWRISCVを動かすパソコンには、以下のものが求められます。

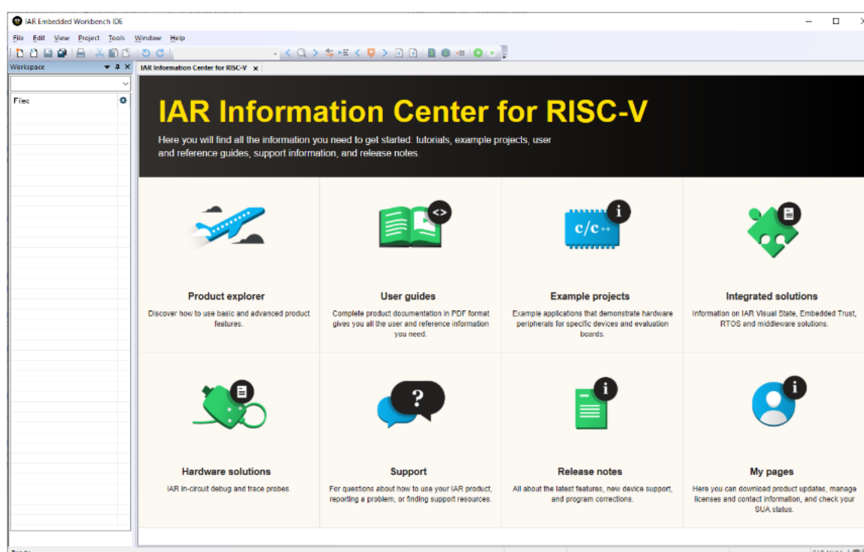
- Windows 10、またはWindows 11の64ビット版を搭載したPentium互換パソコン
- 最小4GバイトのRAMと10Gバイトの空きディスク容量
- 製品ドキュメントを表示するためのAdobe Acrobat Reader

パソコンのスペックについて個人的な意見を述べておくと、EWRISCVを快適に使うためには、RAMが8Gバイト以上、HDD (ハードディスク装置) の空き容量も20Gバイト程度は欲しいところです。
また、ソフトウェア開発には、複数のディスプレイを用いたマルチモニタ環境や、4Kディスプレイなど、解像度の高いモニタ環境の使用を推奨します。
併せて、作業フォルダ名やファイル名について、日本語を使用するのは避けるようお願いします。

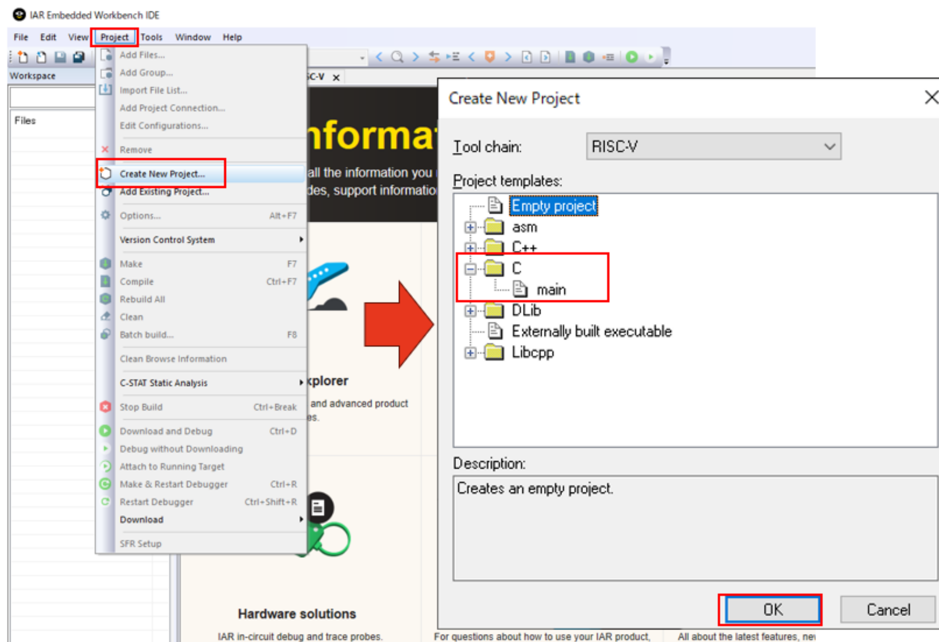
2.2　プロジェクトの作成 (sample 1)

2.2.1　新規プロジェクトを作成して実行する

EWRISCVを起動すると、以下の画面が現れます。

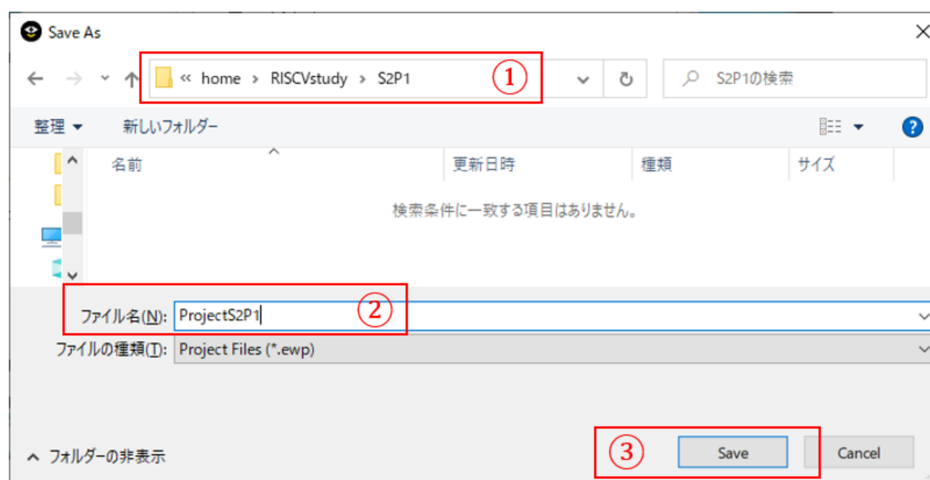


新規プロジェクトを作成するには、メニューバーの[Project]-[Create New Project] を選択します。Create New Project 画面の main を選び、[OK] ボタンをクリックしてください。

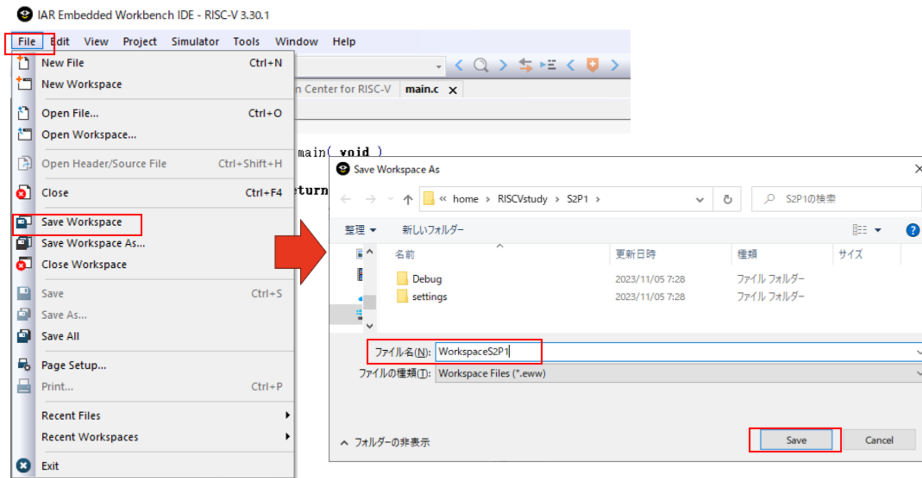


以下の画面で、①のアドレスバーでプロジェクトを作成するフォルダを指定し、②にプロジェクトのファイル名を入力し、最後に③の [保存] ボタンをクリックします。

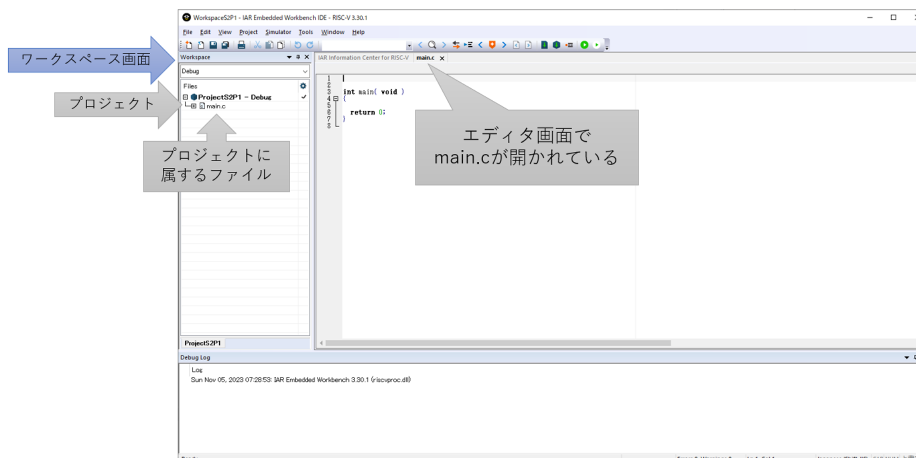
先ほど確認していただいたように、日本語のフォルダ名やファイル名を付けるのは避けてください。
今回はフォルダを /home/RISCVstudy/S2P1 とし、プロジェクトのファイル名を ProjectS2P1 としました。



これで main.c を含むプロジェクトが作成されたのですが、ここでワークスペースも作成します。メニューバーの [File]-[Save Workspace] を選ぶとセーブ画面が開きます。ワークスペースのファイル名を指定し、[保存] ボタンをクリックします。



すると、以下の状態でプロジェクトが作成されます。左側にワークスペース画面があり、その右側にエディタ画面があり、main.c が開かれた状態となっています。EWRISCVはMDI（Multiple Document Interface）型のWindowsアプリケーションとなっており、メイン画面の中に複数の子ウインドウを持っています。子ウインドウの位置は、変更したり、非表示にしたり出来ます。

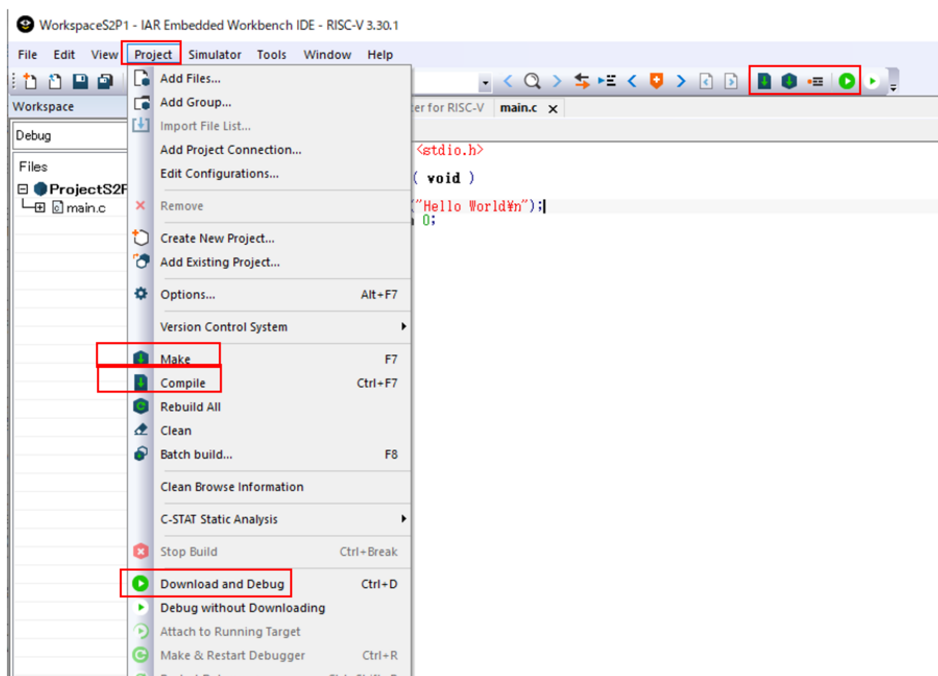


ここまでの操作により、C言語のプログラムを作成するプロジェクトのベースが出来上がりました。新規のプロジェクトを生成した時、EWRISCVでは、オプション設定などはデフォルトの状態になっています。今回は、このデフォルトの状態でソフトウェアを作成していきます。

せっかくなので、hello world プログラムを作成しましょう。hello world プログラムとは、C言語の教科書である『プログラミング言語 C』の最初に出てくるプログラムです。先ほど生成された main.c を以下のコードに変更してください。

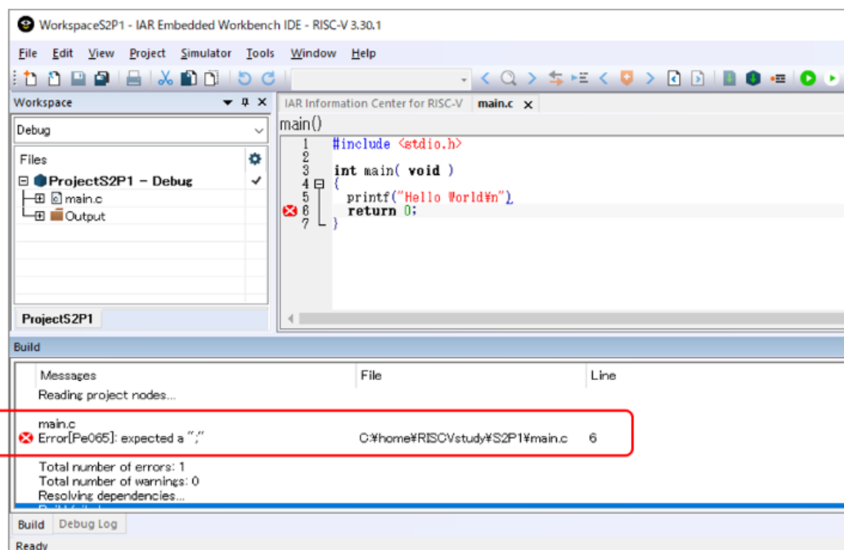
```
#include <stdio.h>
int main( void )
{
  printf("Hello World\n");
  return 0;
}
```

これで、ソースコードの準備が出来ました。続いてビルドしていきます。メイクするには、下図のように、メニューバーの [Project]-[Make] を選択するか、もしくはツールバーのメイクボタン（アイコン）をクリックします。単にコンパイルするだけであれば、メニューバーの [Project]-[Compile] を選択するか、もしくはツールバーのコンパイルボタンをクリックします。
ここで、コンパイルとメイクの違いについて説明します。拡張子 .c / .cpp のファイルを処理して、拡張子 .o のファイルを作るのがコンパイルです。.o のファイルはオブジェクトファイルと呼ばれますが、変数や関数などの配置場所をまだ決定していないので、そのままでは実行できません。メイクは、拡張子 .c / .cpp のファイルをコンパイルし、.o のファイルを集めて、実行形式のファイルを生成します。
最近のマイコンはELFという実行形式のフォーマットを利用します。その他のフォーマットのファイルが必要な場合は、ELFから変換することになります。



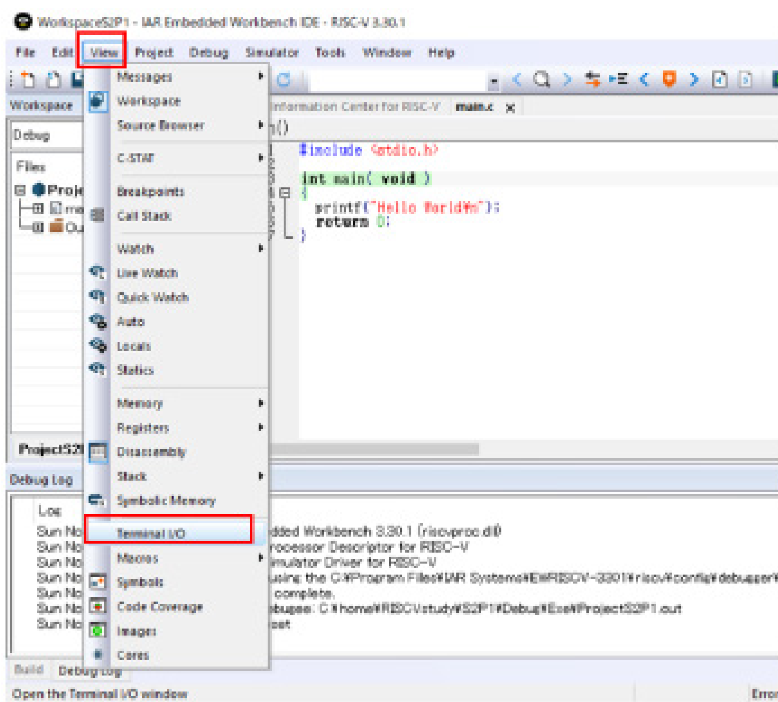
コンパイル時にエラーやワーニングが発生することもあります。EWRISCVでは、ビルド画面にその情報が出力されます。下図下部にビルド画面があり、Errorメッセージが出ています。

このメッセージには、エラーが出たファイル名と行数が示されます。また、この行をクリックすると、エディタ上のエラーが発生した行にジャンプ（遷移）します。

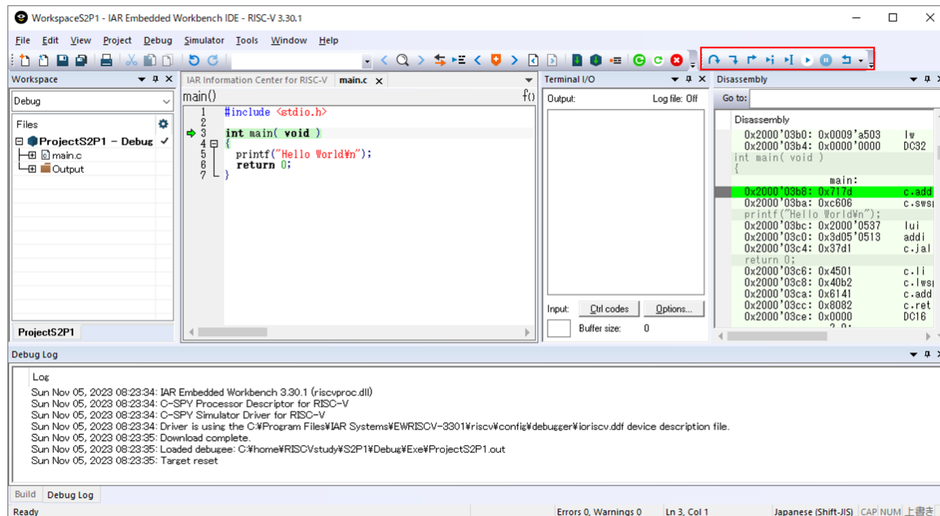


エラーが発生したときは、文字の打ち間違いなどがないか確認してください。よくあるケースとして、文字列やコメント以外のところに全角文字が入るとエラーになります。例えば、全角スペースやセミコロン、ダブルクオーテーション、カッコなどの入力ミスは、見た目だけでは気付きにくいです。エラーがなくなり、メイクが完了したら、デバッグを開始します。

デフォルトの設定では、シミュレータを使ってデバッグするようになっています。メニューバーの [Project]-[Download and Debug] を選択すると、EWRISCVはデバッグモードに入ります。
今回は、printf で文字列を出力します。その表示のため、メニューバーの [View]-[Terminal I/O] を選択します。

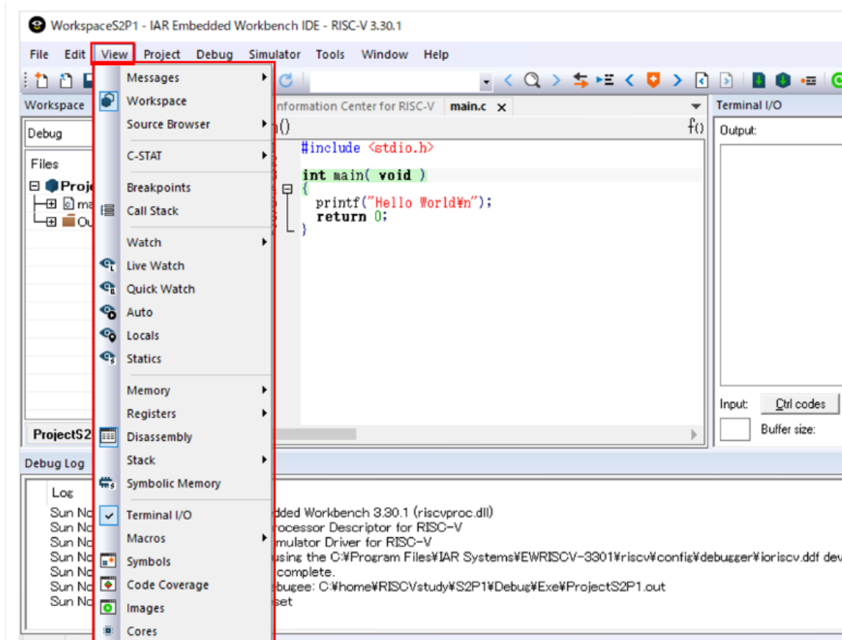


ツールバーの実行ボタンをクリックすると、Terminal I/O画面に Hello World と表示されます。プログラムの実行を細かく操作したい場合は、ステップオーバやステップイン、ステップアウトなどを組み合わせて、実行を進めます。リセットボタンをクリックすると、再度、実行することが可能です。



デバッグ時に使用できる画面は、Terminal I/Oだけではありません。下図のプルダウンメニューに示されているように、さまざまな情報を表示できます。よく使用されるのは、MemoryやWatch、Registersなどの画面です。詳細については、EWRISCVのマニュアルを確認してください。

なお、デバッグ中にファイルは編集できますが、オプション設定などの変更は行えません。デバッグが終わったら、ツールバーのデバッグの終了ボタンをクリックしてください。



2.2.2　プロジェクトの構成

それでは、EWRISCVのプロジェクトの構成を見ていきましょう。この段階では、プロジェクトを作成したフォルダに、以下のファイルやフォルダがあります。

- ワークスペース (.eww)
- プロジェクトファイル(.ewp)
- main.c
- [フォルダ]Debug
  - > [フォルダ] BrowseInfo
  - > [フォルダ] EXE
  - > [フォルダ] List
  - > [フォルダ] Obj
  - > [フォルダ] setting
- [フォルダ]setting

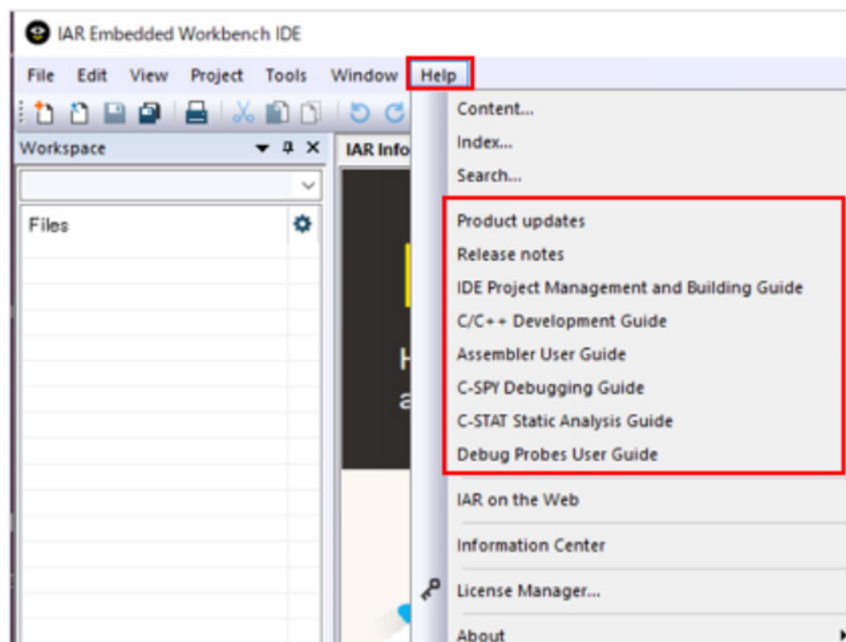
ワークスペースはプロジェクトを管理するための入れ物です。複数のプロジェクトを一つのワークスペースで管理できます。プロジェクトファイルは、DebugとReleaseの二つの構成を持ちます。デフォルトの状態ではDebugの構成になっており、この段階ではDebugフォルダのみ存在します。
そのフォルダの下には、BrowseInfo、Exe、List、Objという四つのフォルダがあり、コンパイルした結果（成果物）が保存されます。DebugからReleaseの構成に切り替わると、Releaseフォルダが作成され、同じようにコンパイルした結果が保存されます。
BrowseInfoフォルダには、ソースコードを解析した情報などが格納されます。このフォルダの中身をユーザが直接見る必要はありません。
Exeフォルダは、メイクされた実行形式のファイル（ELFフォーマット）の配置場所です。HEXファイルなどを生成すると、デフォルトではこのExeフォルダに格納されます。
Listフォルダには、MAPファイルなどが配置されます。
Objフォルダには、コンパイル後の .o ファイルが格納されます。

大きなプロジェクトになると、プロジェクト構成の確認が大変になります。今回のような小さいプロジェクトで、どこに何が作成されるのかを確認しておきましょう。

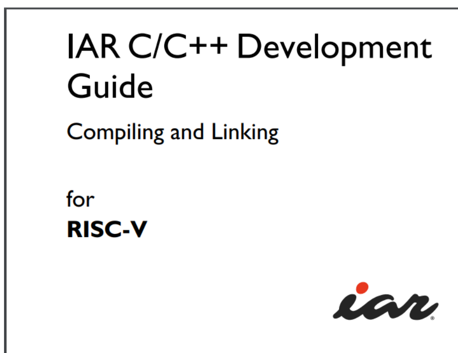
2.2.3 マニュアルについて

ソフトウェアを開発する際には、コンパイラやアセンブラなどの使い方を調べる必要があります。EWRISCVでは、PDFによるマニュアルとオンラインヘルプを提供しています（現状、英文のみの提供となっています）。
EWRISCVの開発画面においてメニューバーの [help] を選ぶと、下図のように表示され、マニュアルなどを選択できます。

- 統合開発環境のマニュアル：IDE Project Management and Building Guide
- C/C++コンパイラやリンカのマニュアル：C/C++ Development Guide
- アセンブラのマニュアル：Assembly User Guide
- デバッグのマニュアル：C-SPY Debugging Guide
- デバッグプローブのマニュアル：Debug Probe User Guide



例えば、C/C++ Development Guideを開くと、下図のようなPDFファイルが開きます。
このC/C++ Development Guideには、以下のようなC/C++やリンカについての説明が記載されています。



Part 1では基本的な考え方などを解説しており、Part 2ではオプションなどの詳細を説明しています。

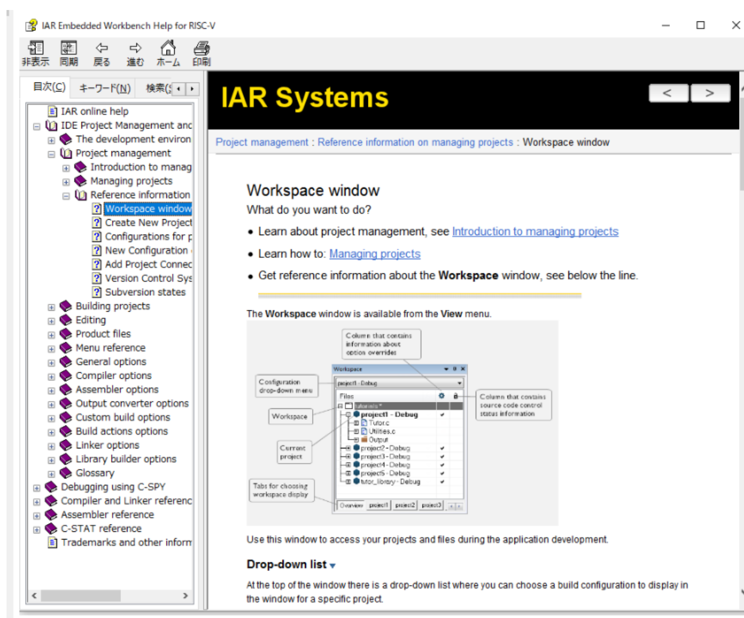
Part 1. 開発ツールの使いかた

- 開発環境
- 組込みアプリケーションの開発
- データの格納について
- 関数について
- ILINKを用いたリンクについて
- ランタイム環境について
- アセンブラとのインターフェースなど

Part 2. リファレンス情報

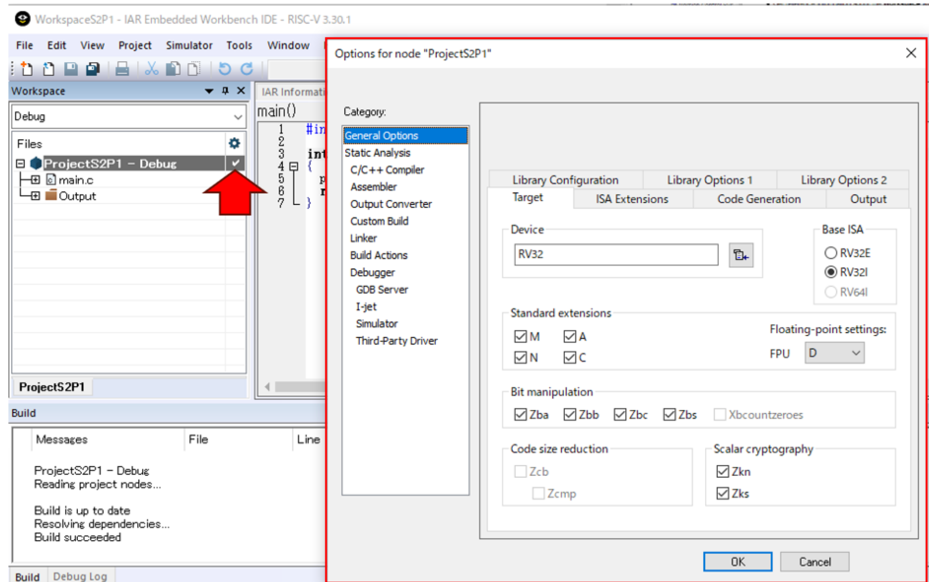
- コンパイラオプション
- リンカオプション
- データ型について
- 拡張キーワード
- プラグマについて
- 組込み関数
- リンカ設定について
- コンパイラで使用するセクションについて
- スタック解析設定ファイルについて
- C++/C言語の実装依存について

また、EWRISCVの開発画面においてF1キーを押すと、下図のようなオンラインヘルプが表示されます。



2.3 オプション設定

ここでは、プロジェクトのオプションを設定する方法と、その設定内容の詳細について確認していきます。EWRISCVにおいてプロジェクトのオプション設定の画面を開くには、下図のワークスペース画面のプロジェクト名の右にある✔（チェックマーク）をダブルクリックするか、もしくはワークスペース画面上のプロジェクト名を選択し、メニューバーの [Project]-[Options] を選択します。



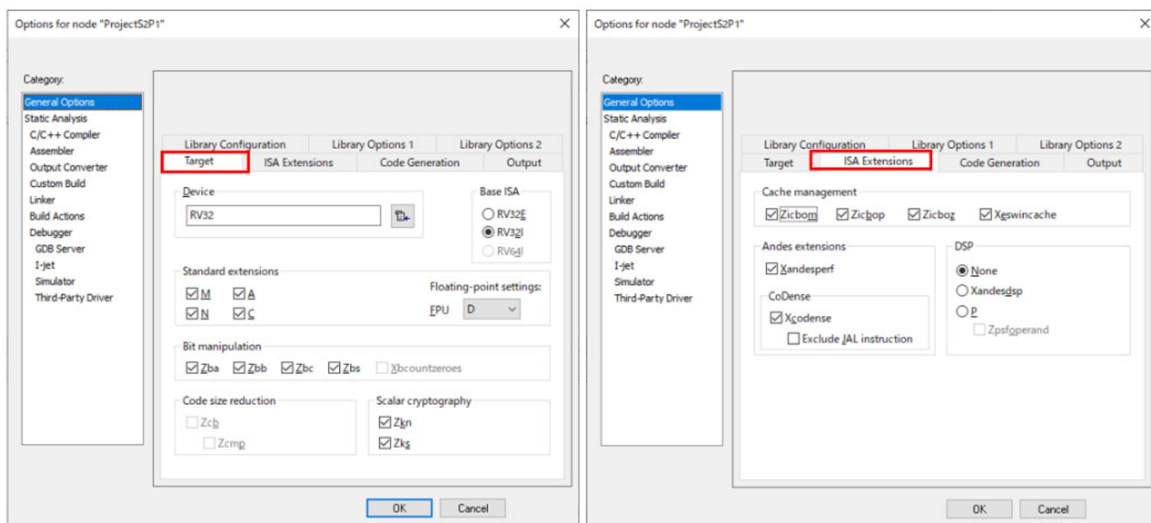
上図の右側の赤枠で囲った部分が、プロジェクトのオプション画面になります。オプション画面は、左側にカテゴリ（Category）を選択するメニューが、右側に実際にオプションを指定する場所があります。

2.3.1 General Options（一般的なオプション）

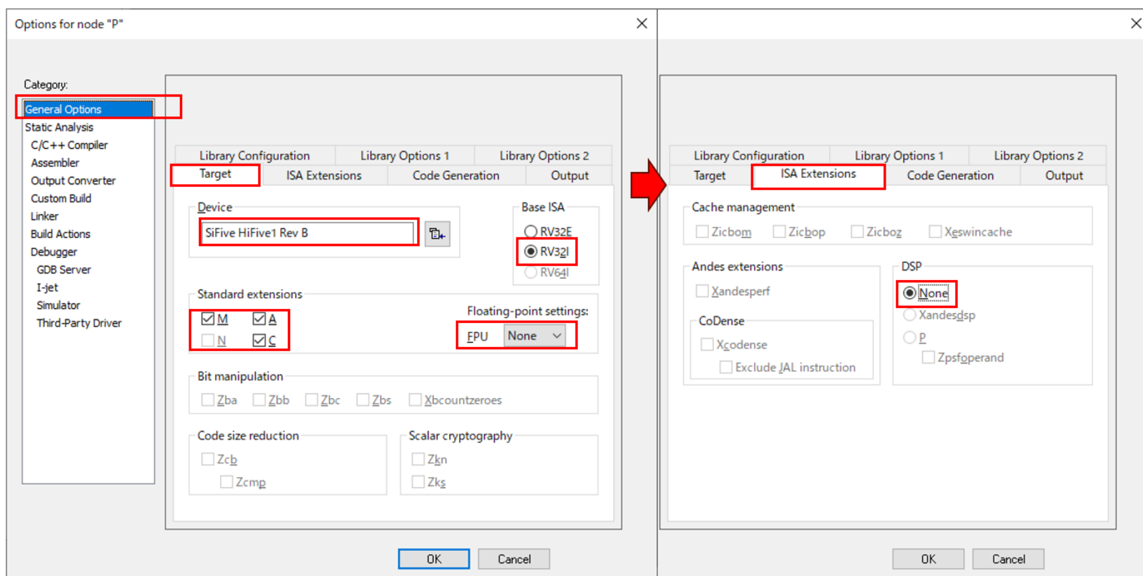
General Optionsでは、1) RISC-Vの命令セットに関するオプション設定、2) コードモデルとスタック/ヒープサイズの設定、3) ライブラリの設定、4) 出力形式と出力フォルダの設定、が行えます。それぞれの項目について、順番に説明していきます。

プロジェクトオプション画面

プロジェクトオプション画面で、カテゴリからGeneral Optionsを選択し、TargetとISA Extensionsでコンパイラが使用する命令セットを指定できます。

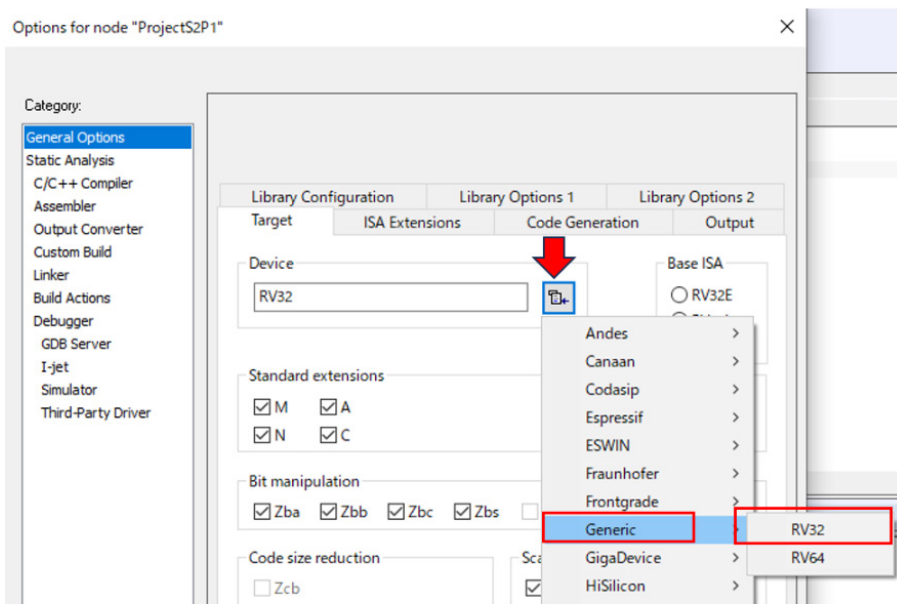


もっとも重要な部分は、TargetタブのDeviceの指定です。使用するマイコンや評価ボードなどが決まっている場合は、その指定することにより、拡張命令などが自動的に設定されます。
例えばSiFive社のHiFive1 Rev Bを選択すると、下図のように命令セットに関するオプションが自動設定されます。HiFive1 Rev Bの実装命令はRV32IMACなので、Targetタブにおいて、RV32I、M、A、Cのオプションが選択されています。



基本命令や拡張命令を個別に指定することも出来ます。その場合、下図のようにDeviceの右側にある選択ボタンをクリックし、プルダウンメニューのGenericを選択してください。RV32もしくはRV64を指定できるようになります。

他の拡張命令については、一つずつチェックを入れる・入れないの選択を行います。
これにより任意の拡張命令を指定できます（ただし依存関係のある命令が存在するので、完全にフリーに選択できるわけではない）。



基本命令セット(Base ISA)については、以下の三つの中から選択できます。

- RV32E:32ビット、汎用レジスタは16本
- RV32I:32ビット、汎用レジスタは32本
- RV64I:64ビット、汎用レジスタは32本

標準拡張命令(Standard extentions)については、以下のものを選択できます。

- M:整数の乗算・除算
- A:アトミック命令n
- C:圧縮命令(16ビット)
- N:ユーザレベル割り込み
- B:ビット操作命令(Bit manipulation)
    - Zba:アドレス計算に向いた命令
    - Zbb:基本となるビット命令
    - Zbc:キャリーレスの乗算
    - Zbs:シングルビット操作命令

浮動小数点演算に関する拡張命令(Floating-point settings、FPU)については、以下のものを選択できます。

- None:浮動小数点演算を使用しない
- F:浮動小数点レジスタの単精度浮動小数点演算
- Zfinx:整数レジスタの単精度浮動小数点演算
- D:浮動小数点レジスタの倍精度浮動小数点演算
- Zdinx:整数レジスタの倍精度浮動小数点演算

コードサイズ縮小(Code size reduction)については、以下のものを選択できます。

- Zcb:C拡張で追加する16ビットの拡張命令
- Zcmp:スタック操作命令でコードを小さくする16ビットの拡張命令

スカラ暗号拡張(Scalar cryptography)については、以下のものを選択できます。:

- Zkn:NISTアルゴリズム向けの拡張命令
- Zks:ShangMiアルゴリズム向けの拡張命令

キャッシュ管理に関する拡張(Cashe management)については、以下のものを選択できます。

- Zicbom:キャッシュブロック管理処理<
- Zicbop:キャッシュブロックプリフェッチ処理
- Zicboz:キャッシュブロックゼロ処理<
- Xeswincache:非標準のキャッシュ管理拡張命令

Andes Technology社が独自拡張した機能(Andes extensions)については、以下のものを選択できます。

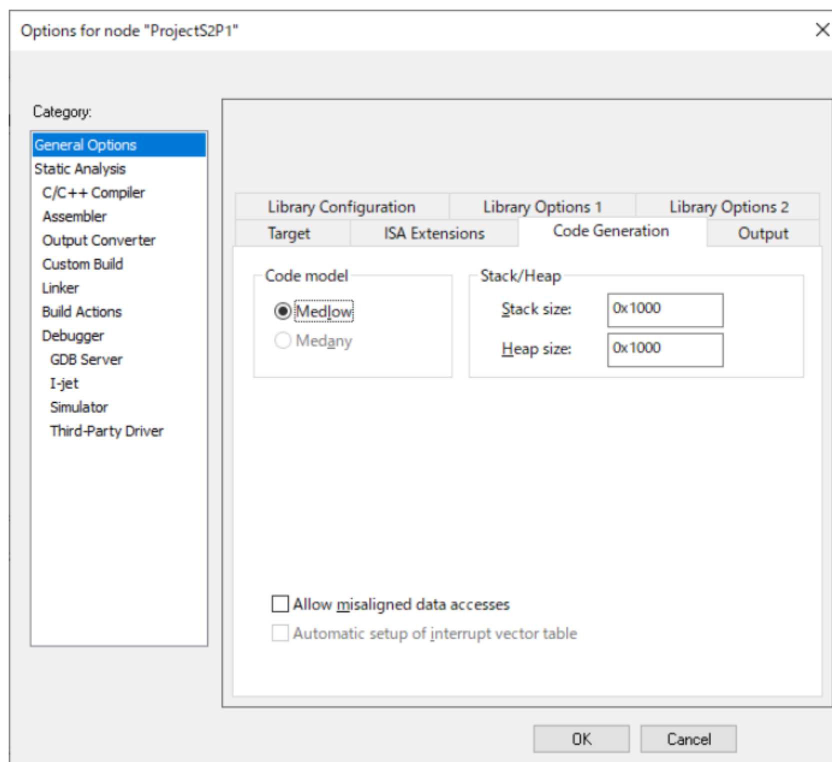
- Xandesperf:Andes Technology社の性能向上拡張(AndeStar V5 Performance)
- Xcodense:Andes Technology社のコードサイズ圧縮拡張(AndeStar V5 CoDense)

DSP拡張については、以下のものを選択できます。

- None:DSP拡張を使用しない
- Xandesdsp:Andes Technology社のDSP拡張を使用
- P:P拡張のサブセット。ZpnとZbpboを使用
- Zpsfoperand:すべてのP拡張を使用

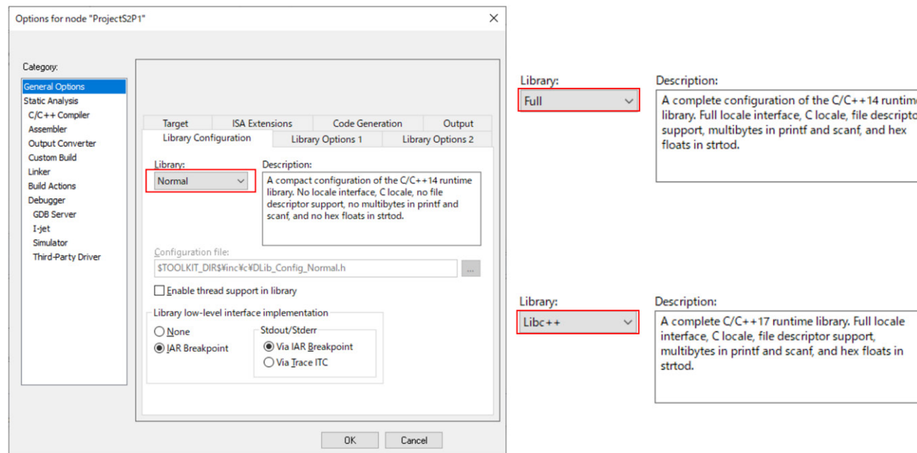
コードモデルとスタック/ヒープサイズの設定

CodeGenrationでは、入力するスタックサイズ（Stack size）とヒープサイズ（Heap size）の値を設定可能で、この値をリンカ設定ファイルで利用できます。



コードモデル（Code model）は、RV64I（64ビット）の時に選択するオプションです。Medlowを選択すると、32ビットの絶対値でアドレス指定を行います。符号付きで処理するため、アクセスできるメモリ領域は0x0000000000000000〜0x000000007fffffff、または0xffffffff80000000〜0xffffffffffffffff となります。
これに対してMedanyを選択すると、PC（program counter）相対でアドレス指定を行います。この場合、相対アドレスでー2Gバイト〜＋2Gバイトを指定することが可能です。Medlowのアクセス領域は0x0を中心とする±2Gバイトの空間でしたが、Medanyの場合はPC相対のアドレス指定となるので、Medlowより参照できる空間が広がります。
ただし、PCからー2Gバイト〜＋2Gバイトのアドレスしか参照できないので、RV64Iを使う際には注意しましょう。

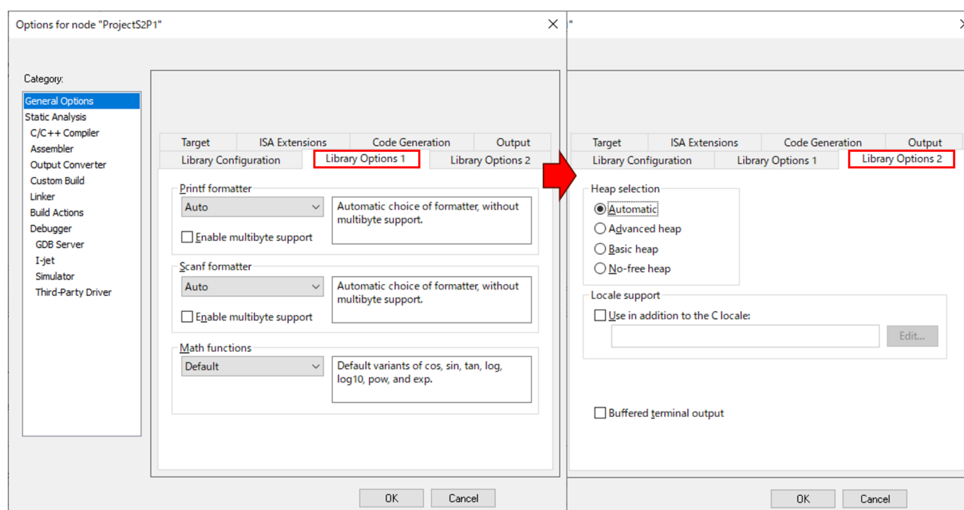
ライブラリの設定
続いて、ライブラリの設定について説明します。ライブラリの設定では、Library ConfigurationやLibrary Options 1、Library Options 2の三つのタブを利用します。Library Configurationタブでは、使用するライブラリについて、以下のものを選択できます。



- None：ライブラリを使用しない
- Normal：C/C++14のライブラリ（C localeのみサポート、マルチバイトサポートなし、Fileサポートなし）を使用する
- Full：C/C++14のライブラリ（フルセット）を使用する
- Libc++：C/C++17のライブラリ（フルセット）を使用する
- Custom：独自のC/C++ライブラリを使用する

また、Low Levelインターフェースのライブラリ（Library low-level interface implementation）の設定もこの画面で行います。これは、デバッガ接続時にパソコンにあるファイルを使用したり、printfなどの標準入出力をデバッガ画面で使用したりする機能に関する設定です。この機能を使用しない場合はNoneを、使用する場合はIAR Breakpointを選択します。Stdout/Stderrでは、標準出力をデバッガに出力する時のやり方を指定します。
通常は、ブレーク時にデータを出力するVia IAR Breakpointを選びます。ITC（Instrumentation Trace Component）を備えるデバイスの場合は、Via Trace ITCを選択可能です。

Library Options 1やLibrary Options 2タブでは、printf や scanf、数学関数について設定できます。



ライブラリの機能の多い・少ないは、実装コードのサイズに大きな影響を与えます。EWRISCVでは、ユーザが必要とする機能に応じて、ライブラリを変更できる構成になっています。

必要最低限のオプションを正しく選択してください。以下はprintf のフォーマット指定です。名前の後ろに「NoMB」とあるのは、「マルチバイトを使用しない」の意味です。

printf のフォーマット指定	**Tiny**	**Small/ SmallNoMB**	**Large/ LargeNoMB**	**Full/ FUllNoMB**
基本指定子 c, d, i, o, p, s, u, X, x, %	YES	YES	YES	YES
マルチバイト文字サポート	NO	YES/NO	YES/NO	YES/NO
浮動小数点数指定子 a and A	NO	NO	NO	YES
浮動小数点数指定子 e, E, f, F, g, and G	NO	NO	YES	YES
変換指定子 n	NO	NO	YES	YES
フォーマットフラグ +, -, #, 0, and space	NO	YES	YES	YES
サイズ修飾子 h, l, L, s, t, and Z	NO	YES	YES	YES
フィールド幅、精度 (* を含む)	NO	YES	YES	YES
long long サポート	NO	NO	YES	YES
wchar_t サポート	NO	NO	NO	YES

下表は、scanf のフォーマット指定です。

scanf のフォーマット指定	**Small/ SmallNoMB**	**Large/ LargeNoMB**	**Full/ FUllNoMB**
基本指定子 c, d, i, o, p, s, u, X, x, %	YES	YES	YES
マルチバイト文字サポート	YES/NO	YES/NO	YES/NO
浮動小数点数指定子 a, and A	NO	NO	YES
浮動小数点数指定子 e, E, f, F, g, and G	NO	NO	YES
変換指定子 n	NO	NO	YES
スキャン集合 [,]	NO	YES	YES
代入抑止 *	NO	YES	YES
long long のサポート	NO	NO	YES
wchar_t のサポート	NO	NO	YES

Library Options 2タブでは、ヒープアルゴリズムについて、以下のものを選択できます。なお、ヒープとはC言語の malloc / free、およびC++言語の new / delete で使用するメモリを指します。

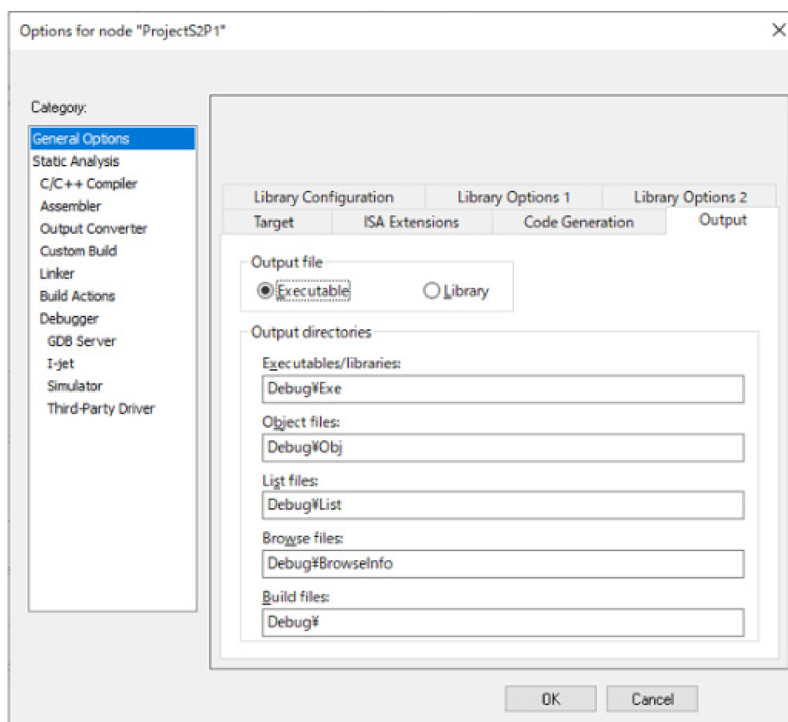
.

- Automatic：ツールが状況を判断して、以下の三つのオプションのどれかを自動的に選択する
- Advanced Heap：ヒープを多用する場合に推奨するオプション
- Basic Heap：ヒープをあまり多用しない場合に推奨するオプション
- No-free Heap：メモリを開放しない場合に推奨するオプション

出力形式と出力フォルダの設定

最後に、出力形式と出力フォルダの設定について説明します。
下図に示すOutputタブでは、出力ファイルを実行形式にするか、ライブラリにするかの設定が行えます。
また、生成物を出力するフォルダを指定できます。

特に問題がなければ、デフォルトのまま運用すればよいと思います。
最終的な生成物（実行形式やライブラリ）は Executables/libraries のフォルダに、マップファイルは List files のフォルダに格納されます。



2.3.2 C/C++ Compiler

ここでは、C/C++コンパイラの設定について説明します。
C/C++ Compilerでは、1) 言語の設定、2) 最適化の設定、3) 出力に関する設定、4) プリプロセッサの設定、5) Diagnostics（診断）の設定、が行えます。

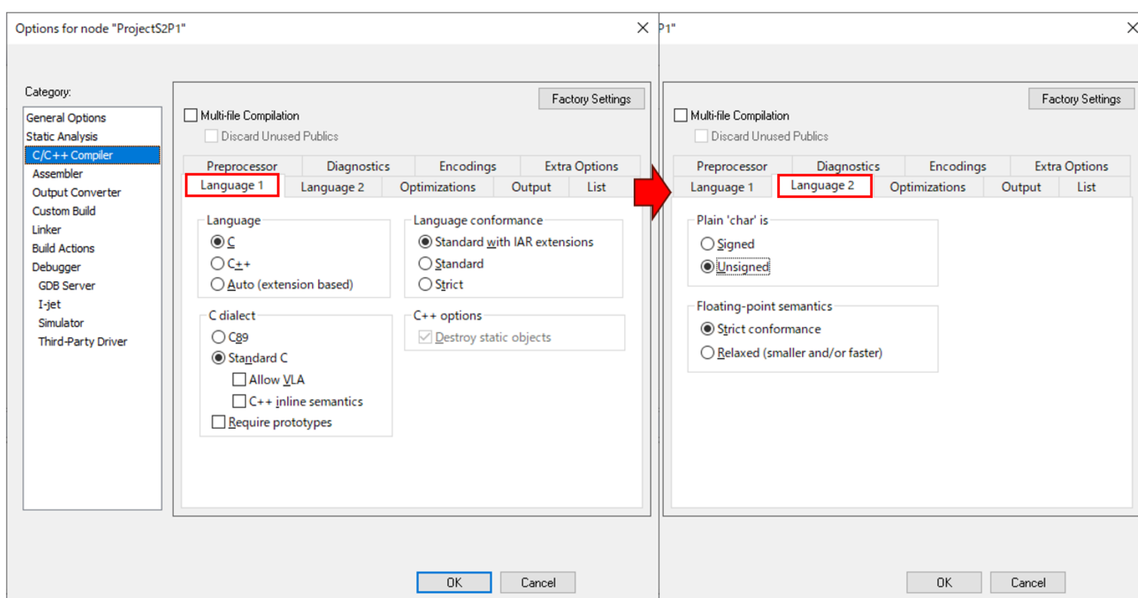
言語の設定

EWRISC-VはC言語とC++言語が使用できます。どのため、C/C++の選択が必要です。オプションLanguage1の画面で選択が出来ます。拡張子が .c の場合はC言語、.cpp、.cc、.cp、.cxx、.c++ の場合はC++言語、と確定している場合は、このAuto(extension based)を選択してもかまいません。EWRISCVが、ファイルの拡張子から自動的に言語を判別してくれます。C言語については、C dialect（C言語の方言）のところで、C89か現在のC18を選べます。
また、Language conformanceでは、規格の言語仕様に準拠するレベルを指定します。以下の三つのレベルから選択できます。

- Standard with IAR extensions：IAR Systems社が独自拡張した機能を有効にする。
- Standard：仕様を緩和する
- Strict：厳密に言語仕様を守る

組込みソフトウェアの開発では、CPU命令で直接記述したいというケースが存在しますが、アセンブラでのプログラムは面倒です。IARでは独自拡張した機能を用意しています。例えば、割り込みハンドラをC言語で宣言するため、__interrupt というキーワードがあります。
このような拡張機能を利用したい場合は、Standard with IAR extentionsを選択します。

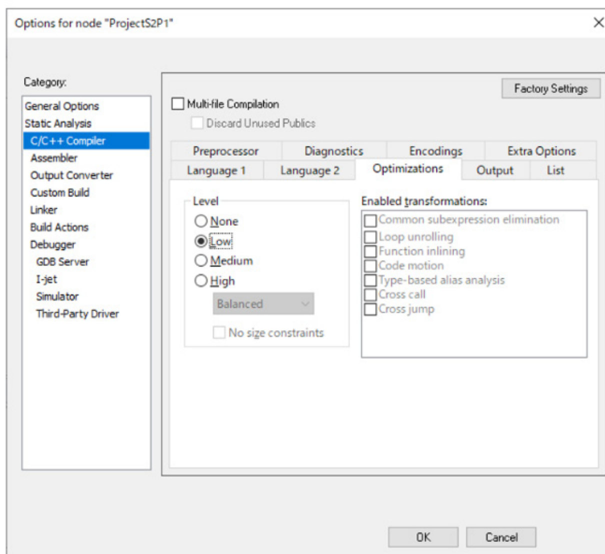
Language 2タブの設定については、最初に確認していただきたい点があります。日本のC言語の教科書を見ると、「char は signed で、正の整数だけを扱う場合は unsigned char で定義します」と書かれているものを見かけます。しかし、これは正しくありません。C言語の規格では、char の符号は実装まかせとなっています。char を unsigned char として使うケースも多くあります。EWRISCVでは、char について、signed か unsigned かをユーザが指定できるようになっています。Language 2タブのPlain 'char' isのところで、自分が使いたい型を選択してください。



最適化の設定

次に、コンパイラの最適化の設定について説明します。下図に示すOptimizationsタブでは、コンパイラの最適化のレベルを設定します。最適化のレベルは、None、Low、Medium、Highから選択でき、この設定に従ってコンパイラがそれぞれ個別の最適化手法を適用します。

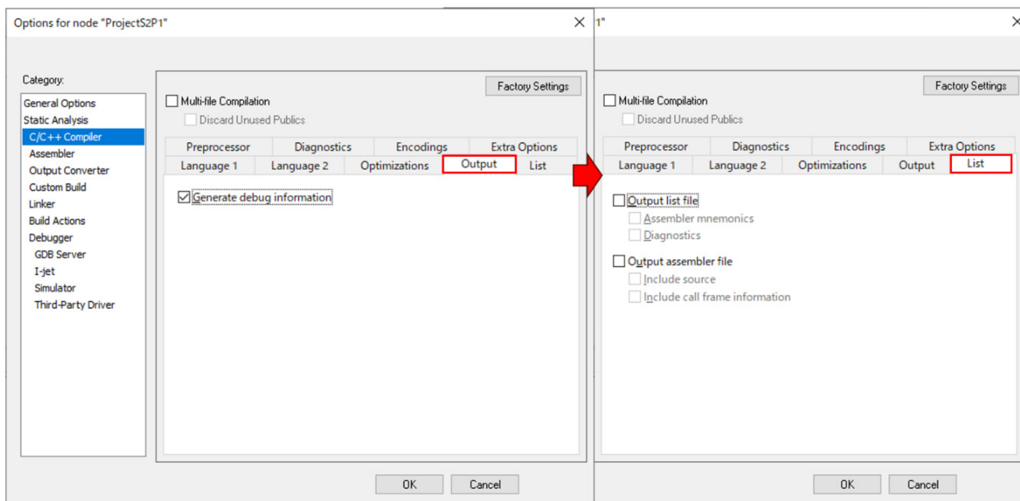
ただし、最適化しすぎると、デバッグ時にソースコードレベルのデバッグが難しくなることを、ぜひ憶えておいてください。最適化により、コンパイラがコード実行の順番を変更したり、ループを展開したり、特定の計算を事前に行ったりするため、記述順にステップ実行を行うことが困難になります。ソースコードデバッグを行いたい場合は、最適化のレベルについて、Noneもしくは Lowを選択します。



出力に関する設定

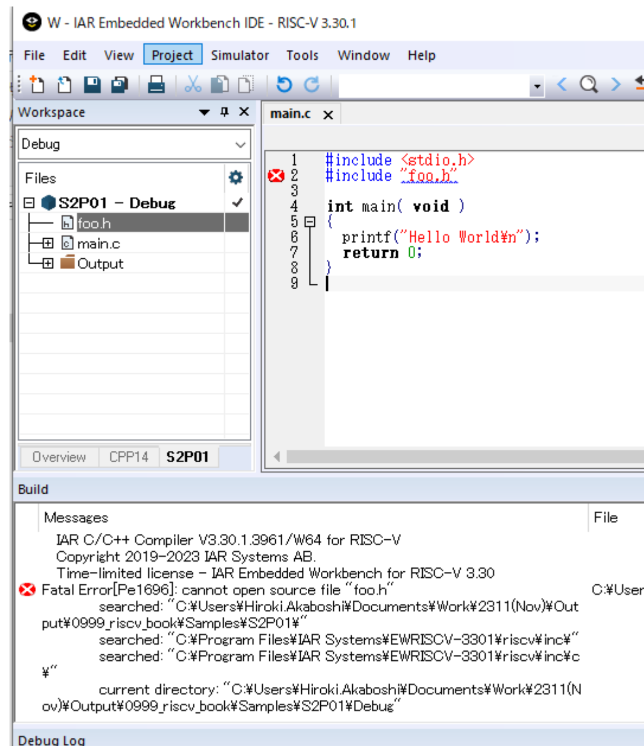
出力に関する設定について説明します。下図の左側に示すOutputタブでは、デバッグ情報の出力設定を行えます。ソースコードレベルのデバッグを行いたい場合は、必ずGenerate Debug Informationのチェックボックスにチェックを入れてください。

Listタブでは、コンパイルした結果のリストファイル、およびアセンブラファイルの出力を指定できます。C言語のプログラムがどのようにコンパイルされたかを確認したい場合、このリストファイルを利用するとよいと思います。なお、EWRISCVの評価版ではアセンブラファイルを出力できないので、注意してください。

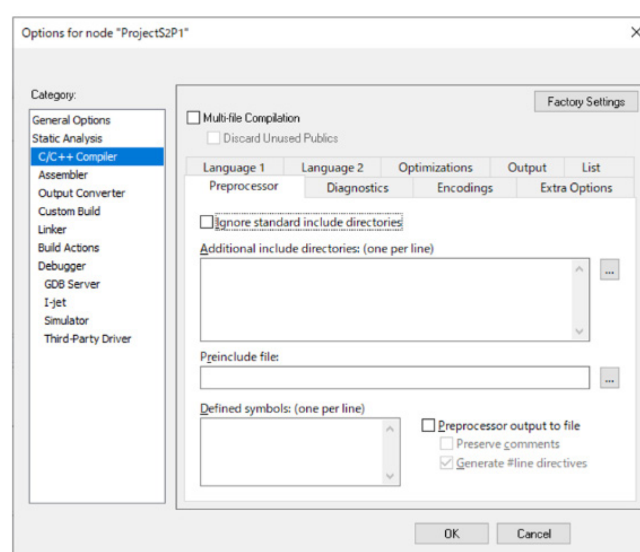


プリプロセッサの設定

EWRISCVでは、ヘッダファイルの指定が必要です。ワークスペース画面にはヘッダファイルを登録できますが、ここに登録しただけだと「ヘッダがない」ということで、下図のようにエラーが出てしまい、コンパイルを実行できません。



EWRISCVでは、ヘッダファイルが置かれているフォルダを指定する必要があります。なにも指定しないと、プロジェクトファイルのあるフォルダ、およびEWRISCVのincフォルダと inc/cフォルダにあるヘッダファイルを検索します。それ以外のフォルダを検索させたい場合は、下図に示すPreprocessorタブのAdditional include directories:のところにそのフォルダを指定していきます。



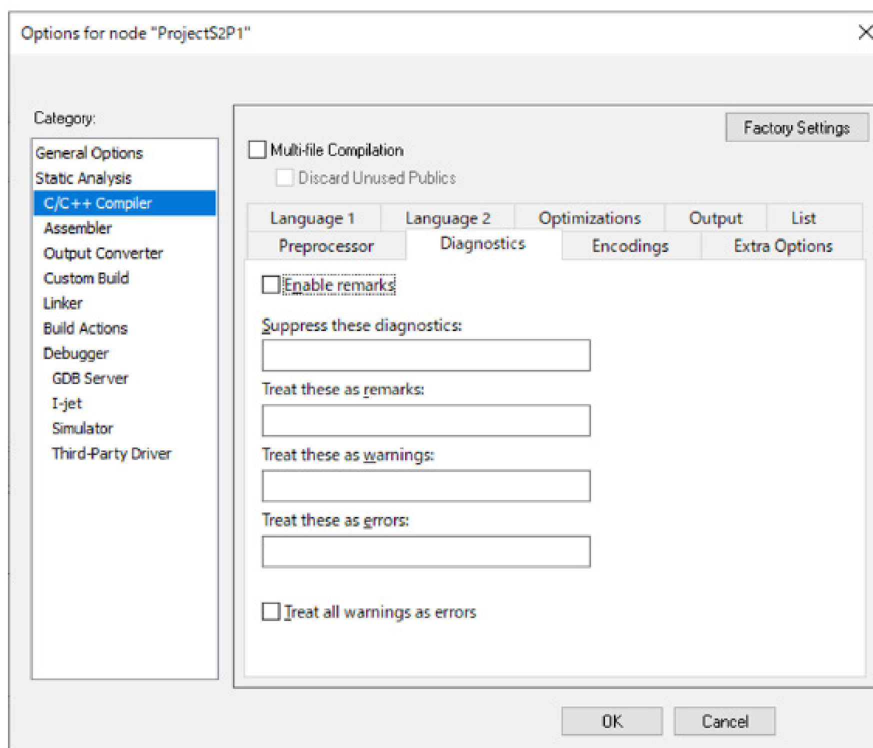
この時、プロジェクトを別の人と共有する時にトラブルが起こりやすいので、フォルダを絶対パスで指定するのは避けましょう。絶対パスを使わずにフォルダパスを設定するため、EWRISCVではプロジェクトフォルダの位置（.ewpファイルが置かれているフォルダ）を$PROJ_DIR$ で参照できます。例えば、$PROJ_DIR$\inc を指定すると、プロジェクトフォルダの直下にあるincフォルダを検索してくれます。

```
$PROJ_DIR$\inc
```

検索させたいフォルダが複数ある場合は、行を変えてフォルダパスを追加してください。
また、コンパイル時に #define SYM (1) と同等の設定を行いたい場合は、PreprocessorタブのDefined symbols:の欄を使って指定します。単純にSYMとだけ記述すると、値は 1 に設定されます。1 以外の値を指定したい場合は、SYM=100 などと記述してください。

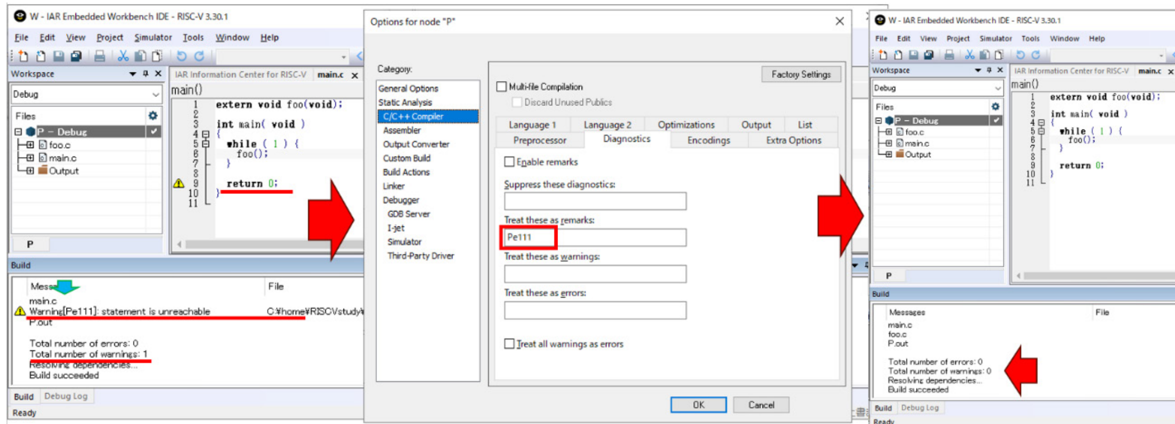
Diagnostics（診断）の設定

EWRISCVには、エラーやワーニングに関する設定を行う機能があります。下図に示すDiagnosticsタブでは、リマークを有効にすることが可能です。リマークとは、正式なエラーやワーニングではないのですが、「注意してください」という趣旨のメッセージです。より信頼性の高いプログラムの作成を目指す場合、このメッセージを確認しながら、必要に応じてコードを修正します。また、エラーやワーニングのレベルを変更したり、出力を停止したりすることも可能です。



組込みソフトウェアの開発では、main 関数において while(1) ループで処理を継続するため、それ以降の処理が実行されない、というケースがあります。下図のエディタ画面に示したプログラムの例では、while ループの後に return 0; という記述があります。しかし、実際にこの行が実行されることはありません。EWRISCVは、Pe111のワーニングを出します。

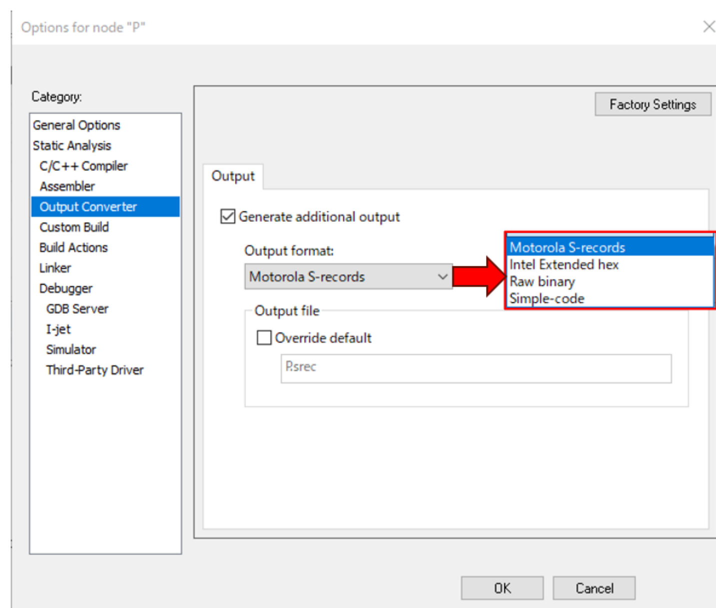
プログラムとしては問題ないので、このワーニングは無視してもよいのですが、Diagnosticsタブの設定により、このワーニングをリマークなどに変更する、という方法もあります。DiagnosticsタブのTreat these as remarks:のところにPe111を指定すると、ワーニングがリマークに変わります。設定後のビルド画面を見ると、ワーニングの数が1から0になったことが分かります。



2.3.3 Output converter

EWRISCVでメイクすると、実行形式のELFファイルが作成されます。しかし、ROMライタなどの別の機器で使用する際に、HEX形式やSREC形式のファイルが必要となる場合があります。Output Converterでこうしたファイルが生成可能です。追加の出力のファイル形式は、以下の四つから選択できます。この中でよく使われるのは、モトローラS形式とインテル形式です。

- モトローラS形式（Motorola S-records）
- インテル形式（Intel Extended hex）
- ローバイナリ（Raw binary）
- Simple-code



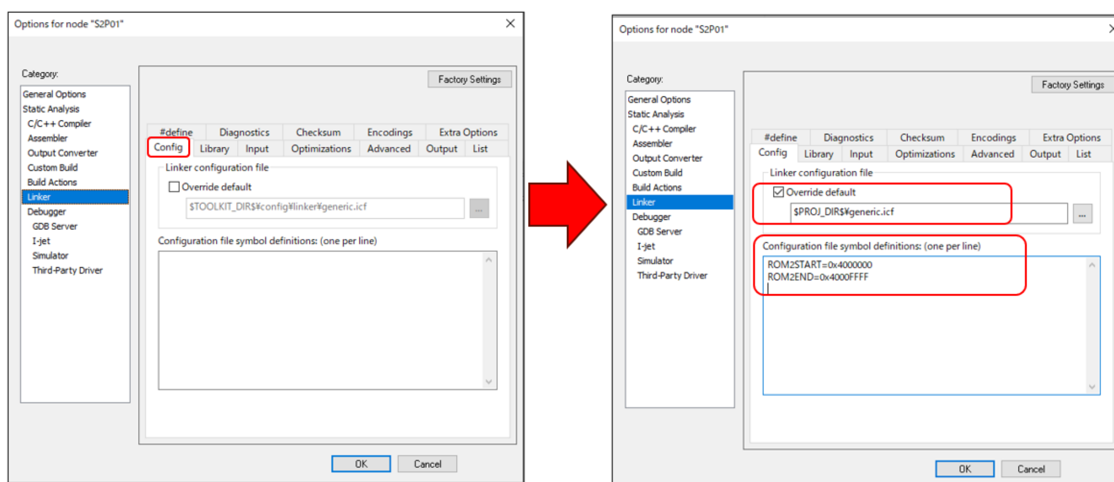
2.3.4 Linker

リンカは、C/C++コンパイラが作成した .o ファイルとその他の必要なデータを統合し、実行形式のファイルを出力します。リンカの設定を記述したファイルをリンカ設定ファイルと呼び、IAR Systems社の統合開発環境の場合、リンカ設定ファイルの拡張子は .icf となっています。
EWRISCVでは、マイコンに応じたリンカ設定ファイルが用意されており、これはConfigタブで設定できます。下図左側の画面を見ると、Linker Configuration fileはデフォルトの設定のままになっています。左側の画面を見ると、Linker Configuration fileはデフォルトの設定のままになっています。
このデフォルト状態のリンカ設定ファイルは、EWRISCVがインストールされたフォルダの下:

```
\riscv\config\linker
```

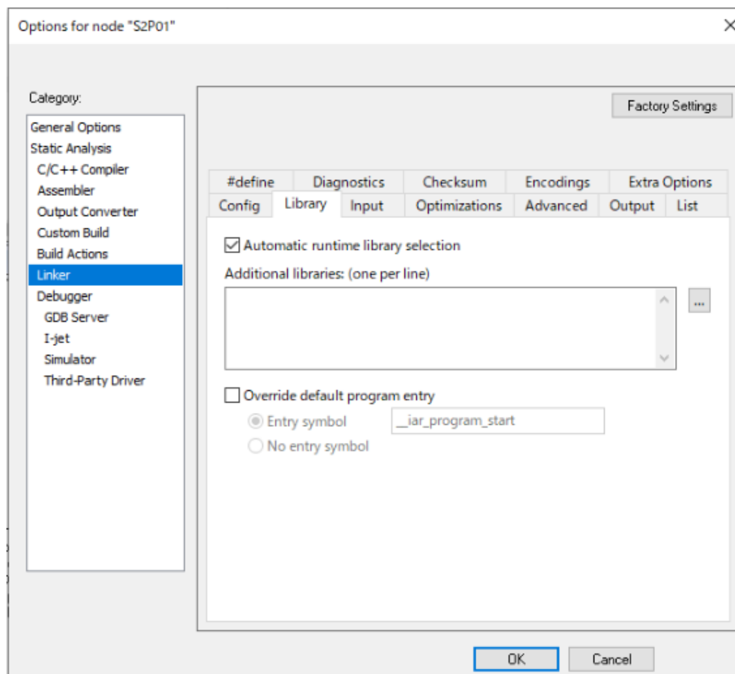
に格納されています。

しかし実際には、プロジェクトごとに個別に、リンカの設定を指定することは珍しくありません。デフォルトの状態のリンカ設定ファイルをプロジェクトフォルダにコピーして使うことが多いのです。プロジェクトファイルが置かれているフォルダがプロジェクトフォルダになり、そのフォルダは $PROJ_DIR$ で示せます。下図の右側の画面では、$PROJ_DIR$\generic.icf をリンカ設定ファイルとして指定しています。また、リンカ設定ファイルで使用するシンボルを、その下のConfiguration file symbol definitions:で定義しています。



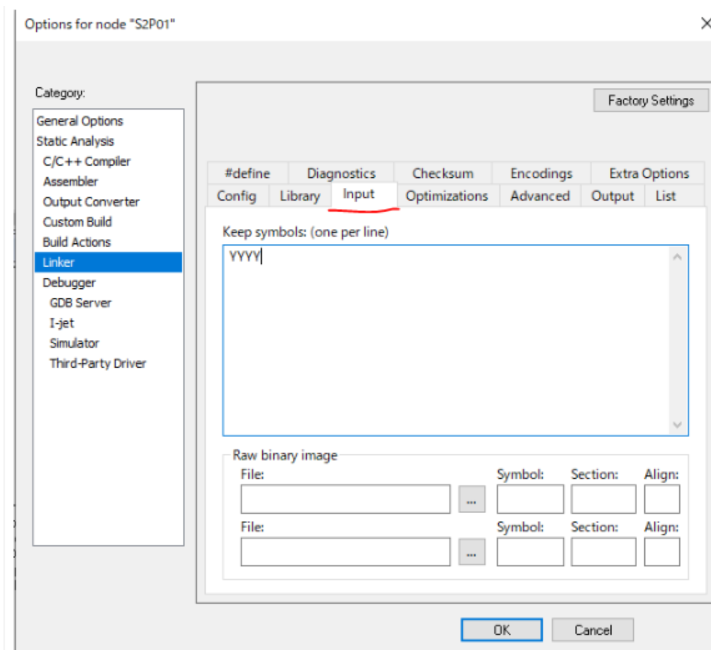
ライブラリが自動的に組み込まれることを嫌がる方もいますが、コンパイラ側で工夫してコードを生成しているので、基本的にここは有効にして使ってください。もちろん、コンパイラが生成するコードを完全に理解し、自分でマッピングの対応付けができる方であれば、手動で設定できると思いますが、そうでない方の場合はリンクできないケースが頻発することになると思います。
Libraryオプションではさらにプログラムエントリを指定できます。下図のようにデフォルトでは __iar_program_start となっており、これはユーザによる変更が可能です。プログラムエントリの情報は、リンカが実行形式のファイルを作るときに必要なものです。

リセット直後に実行を開始する関数（ラベル）を指定することで、ここからプログラムの実行に必要な関数や変数を配置していきます。その時に、明示的に呼び出しがない関数や明示的にアクセスがない変数は、リンカが不要と判断してリンクされません。そのためリセット直後に実行を開始する関数（ラベル）を指定することで、ここからプログラムの実行に必要な関数や変数を配置していきます。
明示的に呼び出しがない関数や明示的にアクセスがない変数は、リンカが不要と判断してリンクされません。そのようなケースでは-keepを用いて明示的に指定してください（つぎに出てきます）。



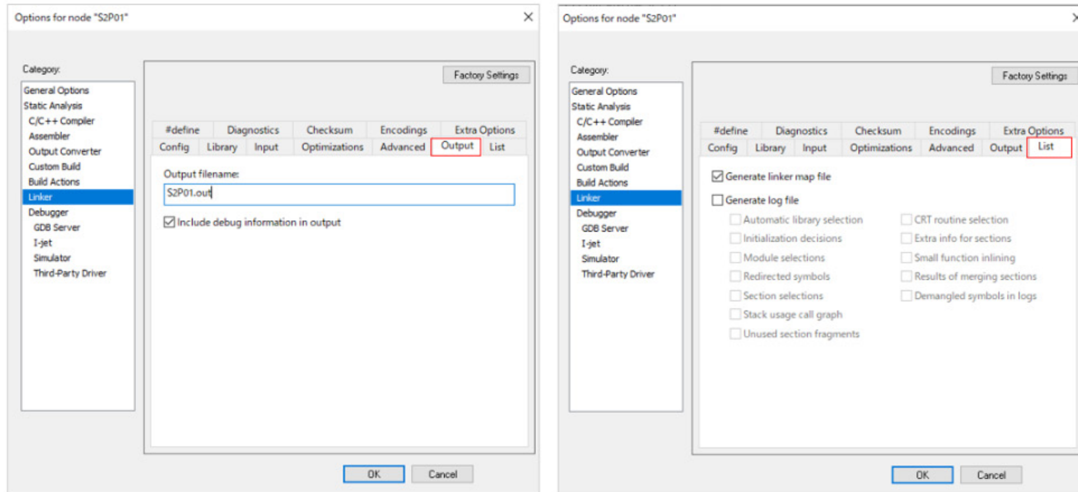
Inputタブでは、先ほど述べた明示的に呼び出しがない関数や明示的にアクセスがない変数が削除されない（実行形式に残る）ように、シンボルを残す（Keepする）指定が行えます。下図の画面を見ると、Keep symbols:のところで YYYY を残すように指定しています。

EWRISCVでは、画像ファイルや音声ファイルのようなバイナリデータを実行形式に取り込む場合、Inputタブの下のRaw binary Imageのところに設定します。そのデータはシンボルとして扱いますので、シンボル名やセクション、アラインを指定します。

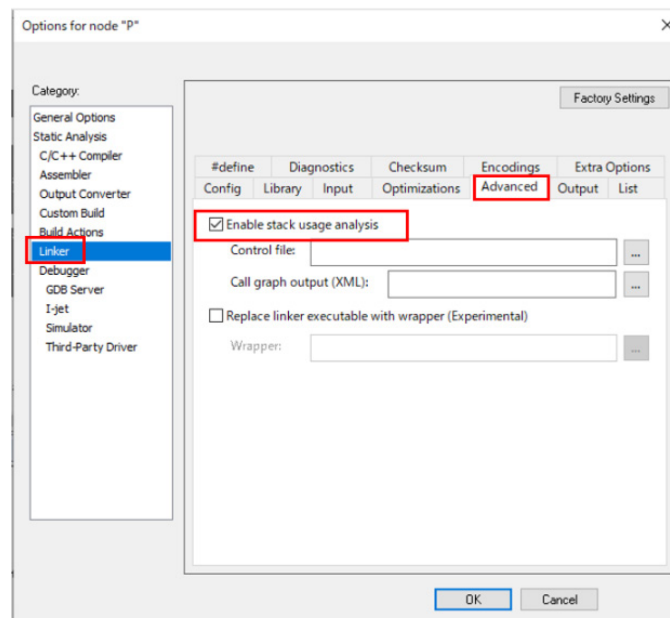


リンカの出力については、OutputタブとListタブで設定します。下図の左側Outputタブで、出力するファイル名を指定できます。デフォルトでは「プロジェクト名.out」になります。その下に、デバッグ情報を含めるかどうかを指定するチェックボックス（Include debug information in output）がありますが、基本的にチェックを入れるようにしてください。デバッグ情報を含めないと、ソースコードデバッグが行えません。
デバッグ情報を含めるとファイルサイズが大きくなるので、最終的なROMサイズが大きくなると勘違いされている方がいますが、デバッグ情報を含めても含めなくても、ROMサイズは同じです。また、デバッグ情報は後から取り除くことも可能です（EWRISCVが提供するコマンド ielftool を使用する）。

Generate linker map fileのチェックボックスを有効にしておくことで、MAPファイルを出力します。もし、メイクが完了したのにMAPファイルが見あたらない場合は、まず、ここのチェックがはずれていないかを確認してください。

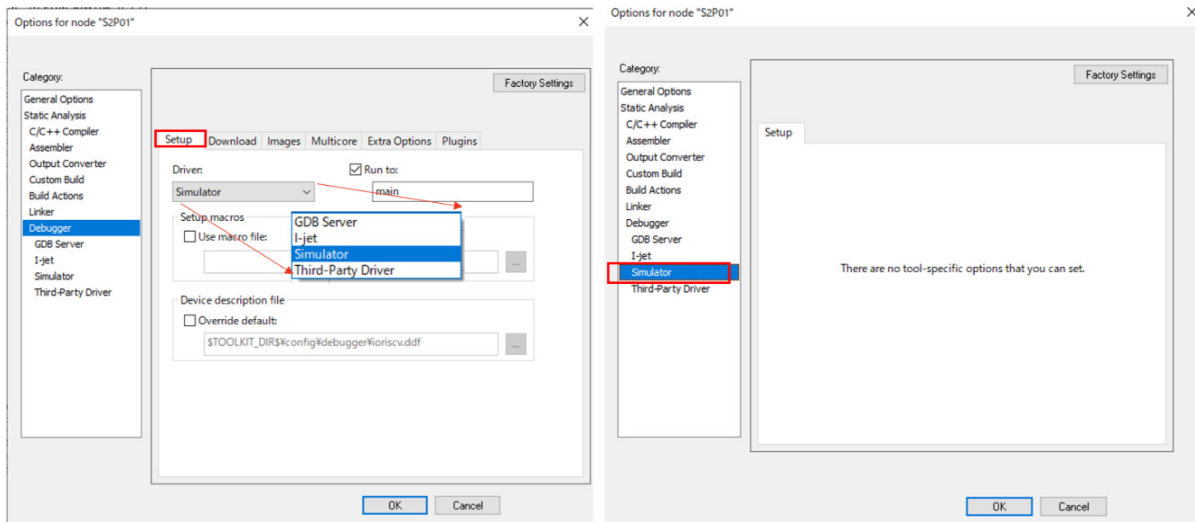


EWRISCVでは、静的なスタック解析も行えます。AdvancedタブのEnable stack usage analysisのチェックボックスを有効にすると、スタック解析が実行されます。解析結果はMAPファイルに出力することが基本ですが、XMLで出力することも出来ます。



2.3.5 Debugger

EWRISCVは、いくつかのデバッグインターフェースを持っており、プロジェクトのオプション画面で切り替えて使用します。
オプション画面左側のカテゴリのDebuggerを選択します。
Setupタブを見ると、DrivesでGDB ServerやI-jet、Simulatorなどを選べるようになっています。
本章ではシミュレータで動作確認を行っていくので、Simulatorを指定します。
なお、オプション画面の左側のカテゴリにもSimulatorという項目がありますが、下図右側に示すように、指定可能な設定オプションは出てきません。



2.4 EWRISCVのプロジェクト全体を理解する

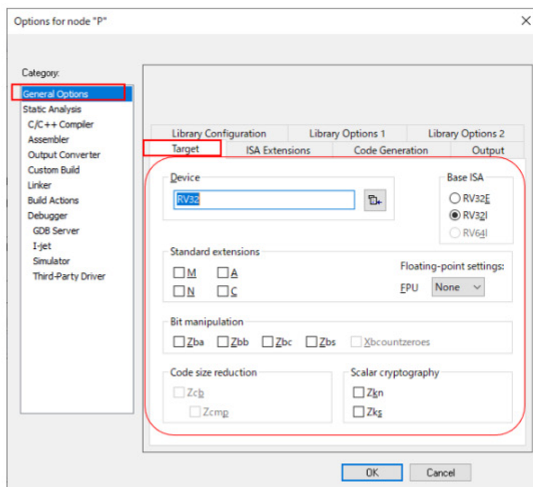
組込みソフトウェアの開発では、C言語のソースプログラムを作るだけでは作業が完了しません。スタートアップコード、CPUコアや周辺ハードウェアの設定、および変数の初期化が必要です。そのため、リンカ設定ファイルを作らなければなりません。
開発する環境により、オプション設定や設定ファイルの詳細は異なります。ここでは、ソフトウェア開発の手順に沿って、EWRISCVの基本的なオプション設定などを説明していきます。

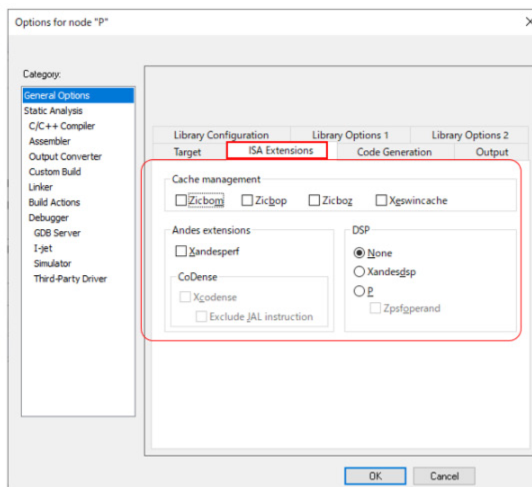
2.4.1 サンプル2を作る

以下の簡単なC言語プログラム（サンプル2）を見てください。
二つの配列 da と db の値を加算して配列 ha に格納し、配列 da と db の値を減算して配列 hb に格納し、その結果で配列 da と db を更新する、という簡単なプログラムです。

```
#define N (10)
int da[N]={11,12,13,14,15,16,17,18,19,20};
int db[N]={ 1, 2, 3, 4, 5, 6, 7, 8, 9,10};
int ha[N];
int hb[N];
int main( void )
{
  int i;
  for ( i=0 ; i < N ; i++ )  {
    ha[i]=da[i]+db[i];
    hb[i]=da[i]-db[i];
  }
  for ( i=0 ; i < N ; i++ )  {
     da[i]= ha[i];
     db[i]= hb[i];
  }
  return 0;
}
```

ここで、CPUの命令セットをRV32Iとして、このプログラムをメイクします。その時のGeneral OptionsのTargetタブとISA Extentionsタブの設定を下に示します。



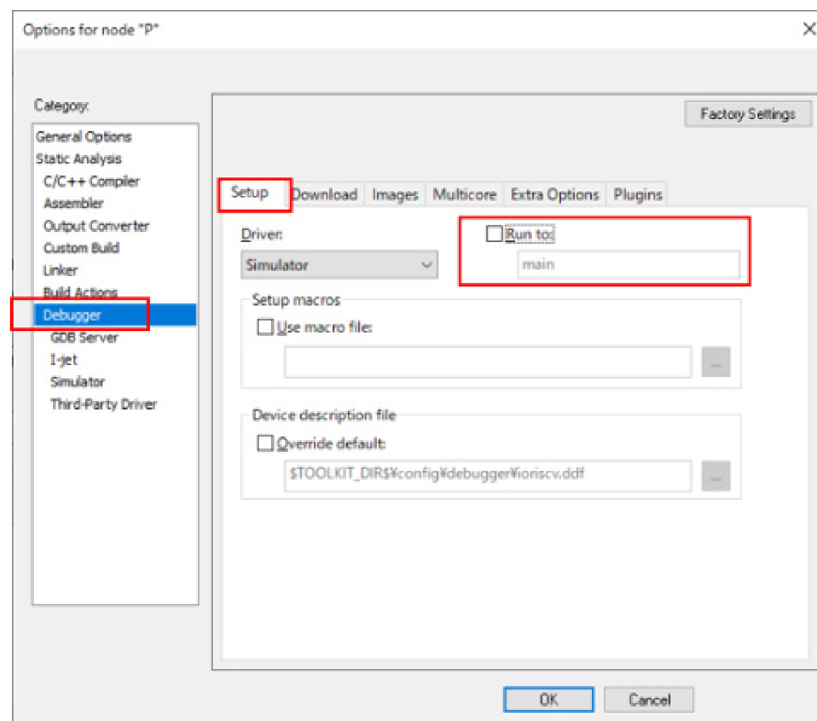


2.4.2　サンプルプログラムを実行

メイクが終わったら、メニューバーの [Project]-[Download and Debug] をクリックします。デバッグモードに入ったら、エディタ上で①の変数 da を選択し、右クリックして、Add to Watch: 'da' を選択します。これを変数 db、ha、hb についても実施します。すると、main 関数に到着した時点で、Watch画面上で変数がすべて初期化されていることが分かります。
このほかにも、プログラムを実行するために必須だが、プログラムでやっていないことがあります。以下の内容が、スタートアップコードによって実施されます。

- 配列 da、db の初期値による初期化
- 配列 ha、hb の 0 による初期化
- スタックポインタの設定
- グローバルポインタの設定
- 割り込みベクタの設定（今回は割り込みを使わないので、設定しなくてOK）

デフォルトの設定では、デバッグを開始した時点で main 関数のところまで実行が進んでしまいます。スタートアップコードの実行を確認する場合は、オプション設定を変更する必要があります。下図に示すようにDebuggerのSetupタブを表示し、Run to:のチェックボックスのチェックを外してください。



再度、メニューバーの [Project]-[Download and Debug] をクリックしてデバッグを開始すると、リセット直後からのプログラムの動作を確認できます。次の図に動作の説明を付けていますが、重要な点を補足します。
RISC-Vでは、グローバルポインタ（GP）のレジスタを用意しています。メモリにアクセスする際に、グローバルポインタの値からのオフセットで計算させることにより、こうした仕組みがない場合より高速に実行させることが可能となります。図では、lui 命令と addi 命令の二つで設定していることを確認できます。lui は20ビットの即値をレジスタの上位側に設定する命令です。また、addi は12ビットの値を即値で指定します。この二つの命令を使うことで、32ビットのアドレスをグローバルポインタに設定しています。

EWRISCVでは、変数の初期化をコンパイラに任せる方法とユーザ自身が作成したコードで初期化する方法のいずれかを選択できます。今回は、リンカ設定ファイルでコンパイラに初期化を任せた設定になっており、関数 __iar_data_init2 によって変数を初期化しています。詳細については、参考文献 [2] を確認してください。

```
        __iar_program_start:
0x2000'0004: 0x0000'0013    nop
        __iar_cstart_init_gp:
0x2000'0008: 0x8000'01b7    lui       gp, 0x80000
0x2000'000c: 0x0501'8193    addi      gp, gp, 0x50
0x2000'0010: 0xf140'2573    csrr      a0, mhartid
0x2000'0014: 0x8000'1537    lui       a0, 0x80001
0x2000'0018: 0x0a05'0513    addi      a0, a0, 0xA0
0x2000'001c: 0xff05'7113    andi      sp, a0, -0x10
0x2000'0020: 0x2000'0537    lui       a0, 0x20000
0x2000'0024: 0x3005'0513    addi      a0, a0, 0x300
0x2000'0028: 0x3055'1073    csrw      mtvec, a0
0x2000'002c: 0x2100'00ef    call20    __low_level_init
0x2000'0030: 0x0005'0463    beqz      a0, 0x2000'0038
0x2000'0034: 0x2100'00ef    call20    __iar_data_init2
0x2000'0038: 0x0000'0013    nop
0x2000'003c: 0x0000'0513    mv        a0, zero
0x2000'0040: 0x00c0'00ef    call20    main
0x2000'0044: 0x23c0'00ef    call20    exit
0x2000'0048: 0x0000'006f    j         0x2000'0048
```


スタートアップコードでは、グローバルポインタの値とスタックポインタの値、ベクタテーブルの値が設定されました。この時にスタートアップコードが使っているアドレスなどを確認する方法について説明します。

リンカが配置した変数や関数のアドレスは、MAPファイルに記載されています。下図に示すように、メイクが終わると、MAPファイルはワークスペース画面のOutputの下に表示されます。ここではP.mapというファイルです。これをダブルクリックするとエディタ画面が開きます。

MAPファイルの中のENTRY LISTで、変数や関数などの配置アドレスを確認できます。$$Base はその領域の先頭、$$Limit はその領域の次のアドレスを示します。例えばCSTACKで言うと、0x800000a0と0x800010a0のアドレスがスタックとして定義されています。実際に使用されるのは、0x800000a0〜0x8000109fの領域です。


```
214
215  Entry                          Address      Size  Type
216  -----                          --------     ----  ----
217  .iar.init_table$$Base          0x2000'0320
218  .iar.init_table$$Limit         0x2000'033c          --
219  CSTACK$$Base                   0x8000'00a0          --
220  CSTACK$$Limit                  0x8000'10a0
221  MINTERRUPTS$$Base              0x2000'0300          --
222  MINTERRUPTS$$Limit             0x2000'031c          --
223  RW_DATA$$Base                  0x8000'0000          --   Gb  - Linker created -
224  RW_DATA$$Limit                 0x8000'00a0          --   Gb  - Linker created -
225  Region$$Table$$Base            0x2000'0320          --   Gb  - Linker created -
226  Region$$Table$$Limit           0x2000'033c          --   Gb  - Linker created -
227  __DebugBreak                   0x2000'01c4   0x4  Code   Gb  __dbg_break.o [2]
228  __exit                         0x2000'01c8  0x28  Code   Gb  __dbg_xxexit.o [2]
229  __iar_copy_init2               0x2000'02a4  0x30  Code   Gb  copy_init.o [4]
230  __iar_cstart_init_gp           0x2000'0008        Code   Gb
231  __iar_data_init2               0x2000'0244  0x3c  Code   Gb
232  __iar_default_minterrupt_handler
233                                 0x2000'0300  0x24  Code   Gb
234  __iar_program_start            0x2000'0004        Code   Wk
235  __iar_static_base$$GPREL
236                                 0x8000'0050          --   Gb
237  __iar_zero_init2               0x2000'033c  0x20  Code   Gb
238  __low_level_init               0x2000'023c   0x8  Code   Gb
239  _exit                          0x2000'0284        Code   Gb  cexit.o [4]
240  abort                          0x2000'0124  0xa8  Code   Gb  __dbg_abort.o [2]
241  da                             0x8000'0000  0x28  Data   Gb  main.o [1]
242  db                             0x8000'0028  0x28  Data   Gb  main.o [1]
243  exit                           0x2000'0280   0x8  Code   Gb  exit.o [4]
244  ha                             0x8000'0050  0x28  Data   Gb  main.o [1]
245  hb                             0x8000'0078  0x28  Data   Gb  main.o [1]
246  main                           0x2000'004c  0xe8  Code   Gb  main.o [1]
```


__iar_static_base$$GPREL について説明します。RISC-Vでは、メモリにアクセスするとき、レジスタ gp が示す値からの相対値（GPREL；GP相対）でアクセスすることが可能です。グローバルポインタを使ってメモリにアクセスする場合、EWRISCVではリンカ設定ファイルによる指定が必要です。具体的な指定方法を次の図に示します。
このリンカ設定ファイルの内容について、簡単に説明します。define region で使用するメモリ領域を定義しています。ここでは、RAMとROMの領域を定義しました。重要なのが38行目のブロック RW_DATA の定義です。このブロックには readwrite 属性のdataが配置されますが、ここで with static base GPREL という記述により、GP相対でアクセスすることが指定されています。この RW_DATA をメモリ領域 RAM_region32 に place して配置しています。ILINKの詳細については、参考文献 [6] を参照してください。

IAR Information Center for RISC-V | main.c | P.map | P.out | generic.icf | __dbg_break.c [RO]

```
1   ////////////////////////////////////////////////////////////////////////////
2   // RISC-V ilink configuration file.
3   //
4
5   build for rom;
6
7   define exported symbol _link_file_version_2 = 1;
8   keep symbol __iar_cstart_init_gp; // defined in cstartup.s
9
10  define memory mem with size = 4G;
11
12  define region RAM_region32 = mem:[from 0x80000000 to 0x8003FFFF];
13  define region ROM_region32 = mem:[from 0x20000000 to 0x3FFFFFFF];
14
```

RAM領域を0x80000000～0x8003FFFFで定義

readwrite属性のdataをブロックRW_DATAに配置。
その時、GPREL（GP相対）で配置すると宣言。

```
36  }
37
38  define block RW_DATA with static base GPREL { rw data };
39
40  "CSTARTUP32" : place at start of ROM_region32 { ro section .cstartup };
41
42  "ROM32":place in ROM_region32        { ro,
43                                         block MINTERRUPTS };
44
45  "RAM32":place in RAM_region32        { block RW_DATA,
46                                         block HEAP,
47                                         block CSTACK };
48
```

RAM領域にブロックRW_DATA、ヒープとスタックを配置。
ヒープは使用しないので最終的には削除された。

2.4.3　GP相対について

近年のソフトウェア開発では、アセンブラ命令を意識しないでソフトウェアを作ることがほとんどだと思います。そのため、「GP相対」と言われてもピンとこないかもしれません。ここでは、C言語の変数がGP相対でどうアクセスされるのかを確認します。また、GP相対を使わない場合についても考えてみます。

2.4.1項で示したプログラム（サンプル2）において、GP相対で実装されている部分を抽出したものが下図です。GPに設定される値は、__iar_static_base$$GPREL で示される 0x80000058 です。それに対して変数 ha は 0x80000054、変数 hb は x8000007C です。変数 ha や変数 hb にアクセスするためには、このアドレスをレジスタに設定する必要がありますが、32ビットのアドレスを汎用レジスタに設定する処理は、lui 命令と addi 命令を組み合わせて実行することで可能です。
しかし、変数アクセスが発生するたびに毎回設定していると、コードサイズが大きくなり、実行時間も長くなります。

GP相対は、グローバルポインタ（レジスタ gp）にあるアドレスを起点にメモリアクセスのアドレスを計算する方法です。下図の右側のコードを見てください。ha[i] に演算結果を書き込むために、まずレジスタ gp の値をレジスタ a2 にコピーし、インデックス ix4（int型のため）の値を a2 に加えて、sw 命令で値を書き込んでいます。インデックスを4倍する処理は、左シフトで実装しています。

主要な変数の配置とサイズ

__iar_static_base$$GPREL	0x8000'0058
da	0x8000'0000
db	0x8000'0028
ha	0x8000'0054
hb	0x8000'007c

// ha[i]=da[i]+db[i];のha[i]を代入する部分

```
addi    a2, gp, -4
slli    a3, a0, 2
add     a2, a2, a3
sw      a1, 0(a2)
```

GPから-４すると、haの先頭アドレスとなる。

a0で配列のインデックスを持っているのでx4して
int型のサイズを計算

a2の先頭アドレスにインデックスx4 した値を加算し、
ha[i]のアドレスを算出し、a2に格納

a2のアドレスに演算結果を書き込む。

次に、GP相対を使わない場合にどうなるかを見てみましょう。GP相対を使わないようにするためには、リンカ設定ファイルを変更する必要があります。EWRISCV のリンカ設定は、デフォルトの状態では、下図の上側に示すようにGP相対を使う設定になっています。
これまでの処理は、GP相対でアクセスするデータ（ここではreadwrite data属性データが対象）をブロック RW_DATA に格納し、そのブロックを RAM_region32 に配置する、というものでした。これに対して、GP相対を使わない場合は、ブロック RW_DATA の定義をやめて、rw data をそのまま RAM_region32 に配置します。

```
define block RW_DATA with static base GPREL { rw data };

"RAM32":place in RAM_region32      { block RW_DATA,
                         block HEAP,
                         block CSTACK };
```

```
// define block RW_DATA with static base GPREL { rw data }; //コメントアウト

"RAM32":place in RAM_region32      { rw  data,
                         block HEAP,
                         block CSTACK };
```

ただし、これだけではスタートアップコードでエラーが発生します。リンカにGP相対の使用を指示しないので、レジスタ gp に設定する値がないことが、エラー発生の理由です。スタートアップコード（cstartup.s）において、レジスタ gp に値を設定している部分をコメントアウトしました。

```
136          PUBLIC __iar_cstart_init_gp
137  __iar_cstart_init_gp:
138          cfi ?RET Undefined
139          EXTERN  __iar_static_base$$GPREL
140          .option push
141          .option norelax
142          ;; lui  gp, %hi(__iar_static_base$$GPREL)
143          ;; addi gp, gp, %lo(__iar_static_base$$GPREL)
144  //         la      gp,  __iar_static_base$$GPREL
145          .option pop
146          REQUIRE ?cstart_init_sp
147
148          CfiEnd  1
149
```

その結果、生成される下図のように変更されました。先に述べたように、lui 命令と addi 命令を使ってアドレスを設定するようになっています

多くの場合、GP相対を使った方が短いコードが生成されます。生成されるコードが短ければ、実行速度も速くなります。

da	0x8000'0000
db	0x8000'0028
ha	0x8000'0050
hb	0x8000'0078

```
//ha[i]=da[i]+db[i];のhaへの書込み（a1に加算した結果を保持）
  lui      a3, 0x80000
  addi     a3, a3, 0x50
  slli     a2, a0, 2
  add      a2, a3, a2
  sw       a1, 0(a2)
```

配列haは0x8000 0050なので、lui命令とaddi命令でa2を設定

配列はint型なので、indexに4倍必要し、a２=a2+s3としてアクセスアドレスを計算

演算結果a1をa2が保持するアドレスに書き込む

出力された命令列を比較したのが次の図です。GP相対を使った場合、変数のアドレスを設定する部分を1命令で実行しています。一方、GP相対を使わない場合、lui と addi の2命令で実行しています。配列や変数アクセスが発生するたびに、この差が現れます。そのため、コードサイズや実行速度の観点では、GP相対を使えるのであれば使ったほうがよい、ということになります。

With GP relative reference

```
addi    a2, gp, -4
slli    a3, a0, 2
add     a2, a2, a3
sw      a1, 0(a2)
```

Without GP relative reference

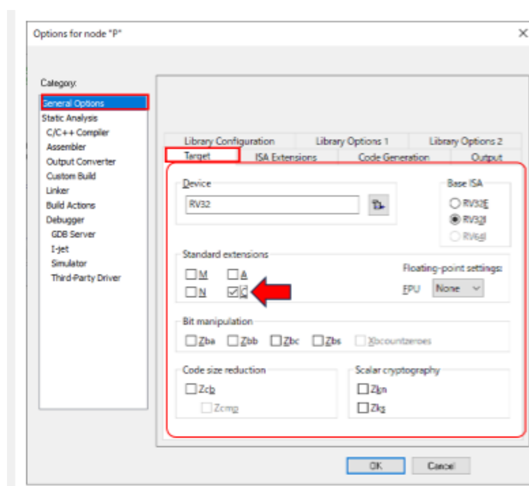
```
lui     a3, 0x80000
addi    a3, a3, 0x50
slli    a2, a0, 2
add     a2, a3, a2
sw      a1, 0(a2)
```

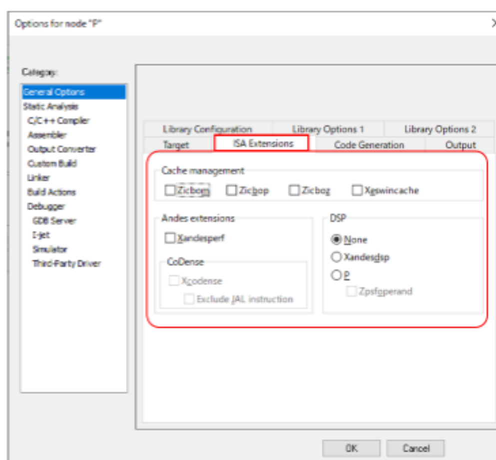
まとめると、GP相対はRISC-Vを使う上で重要な機能であり、使用したほうがコードサイズが小さく、また実行速度が速くなる可能性があります。EWRISCVでGP相対に対応したコードを生成する際には、リンカ設定ファイルによる指定が必要です。GP相対でアクセスするブロックを、with static base GPREL を付けて定義し、そのブロックを配置しなければなりません。

2.5　C拡張命令

RISC-Vの基本命令は32ビットなのですが、C拡張命令（Compress命令、圧縮命令とも呼ぶ）では16ビットの命令を用意しています。これにより、プログラムのコードサイズを小さくすることが可能となり、限られたROM領域（マイコン内蔵のフラッシュROMなど）を有効に使えます。実際にどのくらいの差が出るのかを見てみましょう。

2.4節で作ったプロジェクトでは、命令セットをRV32Iでメイクしました。今回は、C拡張命令も使えるように設定してメイクしてみます。プロジェクトのオプション画面を開き、左側のカテゴリのGeneral Optionsを選択します。下図に示すように、TargetタブのStandard extentionsのところにあるCのチェックボックスにチェックを入れます。





メイクした結果を比較してみます。C拡張命令によるコードサイズの違いを、下表に示します。コードメモリ（readonly code memory）については、708バイトから476バイトへ大きく減少しています。C拡張命令は基本的にコードメモリに対して影響を与えるのですが、配置上のアライメントの関係で、データメモリにも少し影響が出ています。組込みシステムはパソコンなどと比べると使用できるメモリ量が少ないので、C拡張命令は大変有効に働きます。

	RV32I	RV32I+C
readonly code memory	708	476
readonly data memory	108	112
readwrite data memory	4,300	4,344

違う例を用いて確認してみます。以下のコードは、八つの引き数を加算して、その結果を返す関数です。

```
int hoo(int a, int b, int c, int d, int e, int f, int g, int h ) {
  return a+b+c+d+e+f+g+h;
}
```

違う例を用いて確認してみます。以下のコードは、八つの引き数を加算して、その結果を返す関数です。16ビットのC拡張命令には、命令の接頭語として c. が付きます。例えば、加算命令は c.add になります。RV32Iでは add 命令が三つのオペランドで動作を指定しているのですが、c.add 命令は16ビットであるため、二つのオペランドしか取れません。第1オペランドが結果を格納するレジスタであるとともに、入力にもなっています。

c.add a0 a1 という命令を計算式的な記述で書くと、a0 = a0 + a1 になります。機械語（Machine Code）の部分を確認すると、RV32Iでは32ビット（8桁の16進数）になっていますが、RV32ICでは16ビット（4桁の16進数）になっていることが分かります。つまり、同じ処理が半分のコードサイズで実現できています。

RV32I

```
hoo:
0x00b50533  add   a0, a0, a1
0x00c50533  add   a0, a0, a2
0x00d50533  add   a0, a0, a3
0x00e50533  add   a0, a0, a4
0x00f50533  add   a0, a0, a5
0x01050533  add   a0, a0, a6
0x01150533  add   a0, a0, a7
0x00008067  ret
```

Machine code

RV32I + C

```
hoo:
0x952e  c.add  a0, a1
0x9532  c.add  a0, a2
0x9536  c.add  a0, a3
0x953a  c.add  a0, a4
0x953e  c.add  a0, a5
0x9542  c.add  a0, a6
0x9546  c.add  a0, a7
0x8082  c.ret
```

Machine code

すべての命令に対して16ビットの拡張命令が用意されているわけではありませんが、C拡張命令を活用することで、コードサイズを小さくできる可能性があることを理解していただけたと思います。
この拡張命令をマニュアルコーディングする場合、使用する・使用しないの選択が重要なポイントになるのですが、C/C++言語を使う人にとっては、コンパイラのオプション設定を切り替えるだけです。
組込みシステムの開発では、限られたROMを有効活用するため、C拡張命令を使用することを前提にコーディングの方針を検討していただければと思います。

2.6 M拡張命令

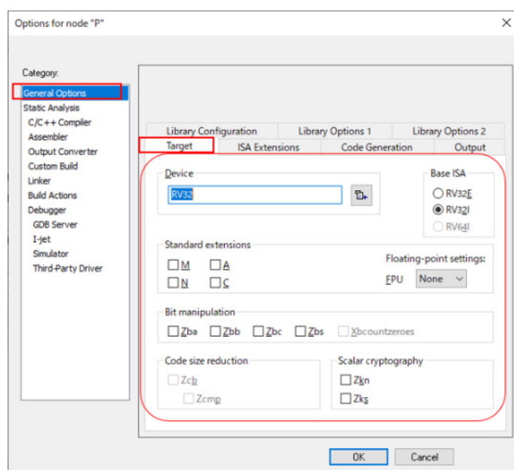
M拡張命令では、整数の乗算・除算を実行できます。もしM拡張命令がなければ、ソフトウェアによって乗算や除算をプログラムして実行することになります。C/C++レベルのコードだけを見ていると、実行できるのなら実行したほうがよいのでは？ と思われるかもしれません。ここでは実際のコードを見ながら、M拡張命令のあり・なしによって生成されるコードがどのように変わるのかを見ていきます。

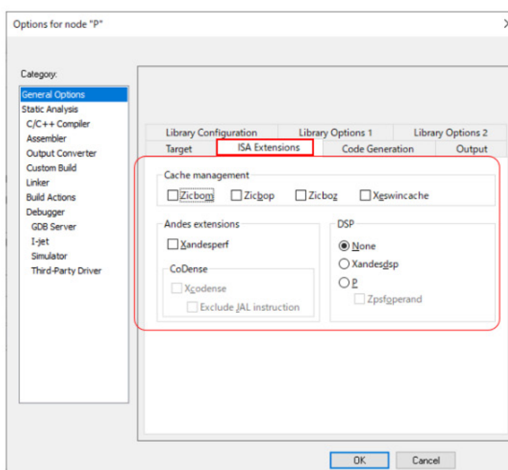
2.6.1 サンプル3をつくる

以下のプログラム（サンプル3）から生成されるコードを見ていきます。このプログラムは、乗算（*）と除算（/）を使っています。今回のプロジェクトでは、まず、RV32Iでメイクを実施します。

```
int a=100;
int b=4;
volatile int c;
volatile int d;
int main( ) {
  c = a*b;
  d = a/b;
  return 0;
}
```

このコードでは、乗算*と除算/を使用していますが、プロジェクトはRV32Iで作成しています。RV32Iの命令セットに、ハードウェアで実装される乗算命令と除算命令はありません。





下図に結果を示します。MAPファイルを見ると乗算を計算する __iar_imul や、除算を計算する __iar_idivmod といったソフトウェアライブラリが使用されています。

生成されたコートでは、乗算と除算でソフトウェアライブラリにジャンプしているのが確認できます。

主要な変数と関数の配置とサイズ

```
__iar_imul        0x2000'01fc
__iar_idivmod  0x2000'0048
__iar_static_base$$GPREL
                           0x8000'0008
a                          0x8000'0000   0x4
b                          0x8000'0004   0x4
c                          0x8000'0008   0x4
d                          0x8000'000c   0x4
```

生成されたコード

```
// c = a*b;
  lw      a0, -8(gp)
  lw      a1, -4(gp)
  jal     t0, __iar_imul
  sw      a0, 0(gp)

// d = a/b;
  lw      a0, -8(gp)
  lw      a1, -4(gp)
  jal     t0, __iar_idivmod
  sw      a0, 4(gp)
```

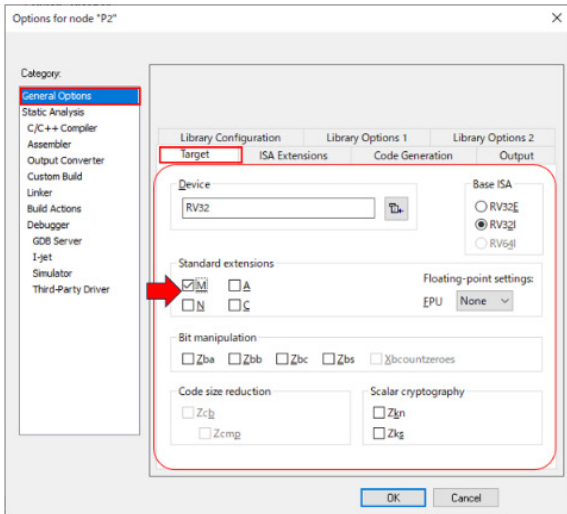
- GP-8は変数aのアドレスとなりそこからロード
- GP-4は変数bのアドレスとなりそこからロード
- __iar_imulと言う関数で乗算を実施
- 演算結果はa0にあるので, GP＋0は変数cとなり、そこに値を書き込む
- GP-8は変数aのアドレスとなりそこからロード
- GP-4は変数bのアドレスとなりそこからロード
- __iar_idivmodと言う関数で除算を実施
- 演算結果はa0にあるので, GP＋4は変数dとなり、そこに値を書き込む

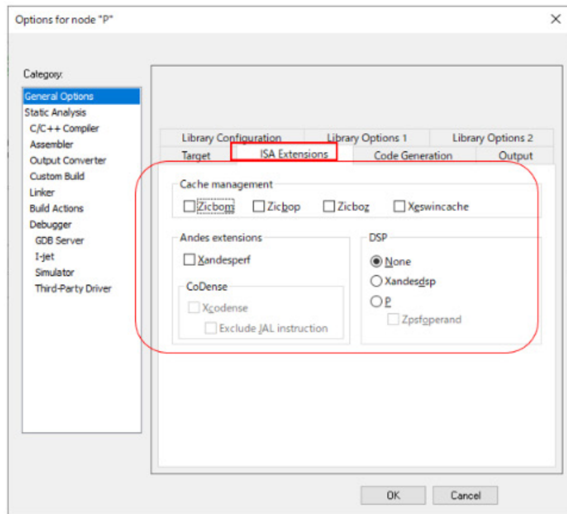
RV32Iでサンプル3をビルドしたの時のコードサイズを
示します。

```
  820 bytes of readonly  code memory
   32 bytes of readonly  data memory
4'236 bytes of readwrite data memory
```

2.6.2 M拡張命令を有効にする

次に、M拡張命令を有効にしてみます。オプション設定を、下図に示します。れにより、RV32IMの命令セットでメイクされます。





マップファイルから、今回、関係する変数と関数を抽出したものが下図の左側です。ソフトウェアライブラリの乗算・除算の関数は存在しません。
右側に示した生成コードを見ると、M拡張の mul 命令と div 命令が使われていることが分かります。

マップファイルから、今回、関係する変数と関数を抽出したものが下図の左側です。ソフトウェアライブラリの乗算・除算の関数は存在しません。右側に示した生成コードを見ると、M拡張の mul 命令と div 命令が使われていることが分かります。

主要な変数と関数の配置とサイズ

```
__iar_static_base$$GPREL
                 0x8000'0008
a                0x8000'0000   0x4
b                0x8000'0004   0x4
c                0x8000'0008   0x4
d                0x8000'000c   0x4
```

生成されたコード

```
// c = a*b;
   lw     a0, -8(gp)
   lw     a1, -4(gp)
   mul    a0, a0, a1
   sw     a0, 0(gp)

// d = a/b;
   lw     a0, -8(gp)
   lw     a1, -4(gp)
   div    a0, a0, a1
   sw     a0, 4(gp)
```

GP-8は変数aのアドレスとなりそこからロード
GP-4は変数bのアドレスとなりそこからロード
乗算命令mulを実行
演算結果はa0にあるので,
GP＋0は変数cとなり、そこに値を書き込む
GP-8は変数aのアドレスとなりそこからロード
GP-4は変数bのアドレスとなりそこからロード
除算命令divを実行
演算結果はa0にあるので,
GP＋4は変数dとなり、そこに値を書き込む

RV32IMでサンプル3をビルドしたの時のコードサイズを、以下に示します。

```
500 bytes of readonly  code memory
     32 bytes of readonly  data memory
  4'168 bytes of readwrite data memory
```

M拡張命令のあり・なしで、コードメモリ（readonly code memory）のサイズが変わっています。M拡張命令を利用すると、コードメモリのサイズが820バイトから500バイトへ、つまり約60％に縮小しています。
ただし、常にコードメモリが60％になるわけではありません。ソフトウェアライブラリの利用により、300バイト程度増加したということなので、開発対象となるソフトウェアの規模が大きくなると、その影響は小さくなると考えられます。
この時の実行サイクル数を調べてみると、下表のようになりました。M拡張命令ありの場合、乗算・除算を少ないサイクル数で（高速に）実行できていることが分かります。なお、実行サイクル数は与えるデータによって変動するので、数字はあくまでも参考程度のもの、と考えてください。

	RV32I	**RV32I+M**
c = a*b;	24	4
d = a/b;	59	4

2.6.3 RV32Mの拡張命令

RV32Mの拡張命令を、以下に示します。
- MUL rd,rs1,rs2
- MULH rd,rs1,rs2
- MULHSU rd,rs1,rs2
- MULHU rd,rs1,rs2
- DIV rd,rs1,rs2
- DIVU rd,rs1,rs2
- REM rd,rs1,rs2
- REMU rd,rs1,rs2

命令の命名規則を説明します。MUL、DIV、REMが、基本的な32ビットの乗算命令、除算命令、剰余算（除算の余り）命令です。MULでは、32ビットと32ビットの乗算の結果が64ビットの値をとります。MULHの場合、MULの演算結果の上位32ビットを格納します。この時、ソースとなるレジスタの値が signed か unsigned かによって、MULH、MULHSU、MULHUを使い分ける必要があります。MULHの場合は二つのソースが signed 、MULHUの場合は二つのソースが unsigned 、MULHSUの場合は rs1 が signed 、rs2 が unsigned になります。

日頃、C言語でソフトウェアを作成していると、乗算・除算のためにこれほど多くの命令が用意されていることを意識する機会は少ないと思いまし、RISC-Vを習得するためには、こうした点も理解しておくほうがよいと思います。というのは、カスタム命令などを作る時に、どのようなデータ型に対して処理を行うかを考える必要があるからです。また、これらの命令がどのような時に使われるのか、命令数を減らしたい場合にどの命令を残せばよいのか、などを考える訓練にもなると思います。

2.7　A拡張命令

続いて、A拡張命令について説明します。A拡張命令はアトミック（Atomic）命令とも呼ばれます。こうした命令は、複数の処理を同時に実行する場合に必要となります。例えば、OS（リアルタイムOSを含む）を使ったマルチタスク環境において、複数のタスクが共有データに正しくアクセスするために行う排他制御に使用されます。また、OSを使わない場合でも、割り込み処理で共有データを操作する時などに使用されます。
RV32Aのアトミック命令を、以下に示します。先頭にAMOが付く9個の命令がAMO命令、残りの2個がLR/SC命令です。

- AMOSWAP.W rd, rs2,(rs1)
- AMOADD.W rd, rs2,(rs1)
- AMOXOR.W rd, rs2,(rs1)
- AMOAND.W rd, rs2,(rs1)
- AMOOR.W rd, rs2,(rs1)
- AMOMIN.W rd, rs2,(rs1)
- AMOMAX.W rd, rs2,(rs1)
- AMOMINU.W rd, rs2,(rs1)
- AMOMAXU.W rd, rs2,(rs1)
- LR.W rd, (rs1)
- SC.W rd,rs1,(rs2)

まず、AMO命令について説明します。この命令は、Read-Modify-Write（データの読み出し、更新、書き込み）を実行します。例えば、AMOOR.W rd, rs2,(rs1) の場合、以下の動作を不可分で実行します。

1. rs1 で示すメモリアドレスから値を読み出す
2. メモリから読み出した値と rs2 のレジスタ値でORを実行する
3. ORの結果をメモリに書き込むと同時に、先ほどの rs2 の値をレジスタ rd に保存する

LR.W はLoad-Reserved命令です。LR.w rd, (rs1) では、以下の動作を不可分で実行します。

1. rs1 で示すメモリアドレスから値を読み出す
2. メモリから読み出した値をレジスタ rd に格納する
3. そのメモリ上に対する予約を記録する

SC.W はStore-Conditional命令です。SC.w rd, rs1, (rs2) では、以下の動作を不可分で実行します。

1. rs1 で示すメモリアドレスに対して予約がある場合、rs2 の内容を書き込み、ストアに成功したら rd の値を 0 とする
2. そうでない場合は、0 以外のエラーコードを書き込む

Load-Reserved命令とStore-Conditional命令の使用例については、以下のアセンブリコードが、RISC-V命令セットマニュア仕様書に紹介されています。

```
        # a0 holds address of memory location
        # a1 holds expected value
        # a2 holds desired value
        # a0 holds return value, 0 if successful, !0 otherwise
cas:
        lr.w t0, (a0) # Load original value.
        bne t0, a1, fail # Doesn’t match, so fail.
        sc.w t0, a2, (a0) # Try to update.
        bnez t0, cas # Retry if store-conditional failed.
        li a0, 0 # Set return to success.
        jr ra # Return.
fail:
        li a0, 1 # Set return to failure.
        jr ra # Return.
```

2.7.1 サンプル4：A拡張命令を使ったプログラム

A拡張命令については、C/C++言語で記述したプログラムから直接生成されることはありません。アセンブラ命令、もしくはインラインアセンブラを使用して記述する必要があります。実際に、C言語からアセンブラを呼び出し、そのアセンブラでAMOADD命令を動かすプログラムを見てみます。
プロジェクトのオプション画面を開き、左側のカテゴリのGeneral Optionsを選択してください。A拡張命令を使用するための設定を、下図に示します。

C言語コードとアセンブリコードのプログラム（サンプル4）を、次の図に示します。変数 d2 の宣言のところに @ 0x80000000 が付いているのは、IAR Systems社が独自拡張した機能です。この機能は、宣言した変数を @ 以降の数字のアドレスに配置します。これにより、変数 d2 は 0x80000000 に配置されます。main 関数の中で関数 f001 を呼び出していますが、関数 f001 はアセンブリコードで定義されています。
次に、右側のアセンブリコードを見ていきしょう。アセンブリコードをあまり見たことのない方が多いと思うので、ここでは細かく説明します。アセンブラでは、1列目（左端）から記述したものがラベルとなり、ラベル以外のものは2列目以降に記述します。図2-58の右側のアセンブリコードにおいて、1列目から記述しているのは、5行目の f001 のみです。アセンブラでは、変数も関数もラベルで名前を付けることを憶えておきましょう。

2行目の public f001 は、f001 というラベルを外部のファイルから参照できるようにするための記述です。この public 定義がなければ、そのファイル内でのみのアクセスとなります。4行目の SECTION は、これ以降に記述するものを配置するセクションを設定しています。セクションの名称が '.text' で、メモリタイプがCODEとなります。メモリタイプには、CODE、CONST、DATAの3種類があり、それぞれコード、定数、変数の意味になります。
その後ろのNOROOTは、この部分が不要な場合にリンクで削除してもよい、という指示です。使用しない関数などを残したい場合は、ROOTというキーワードを使います。最後の (2) は、2バイトでアラインすることを意味します。

main.c / main()

```
#include <stdio.h>

extern void foo1();

int d2 @  0x80000000;

int main( void ) {
  int t;
  t=f001();
  printf("d2=%d, ret=%d¥n", d2,t);
  t=f001();
  printf("d2=%d, ret=%d¥n", d2,t);
  return 0;
}
```

test.S

```

  public f001

  SECTION `.text`:CODE:NOROOT(2)
f001:
 lui t0, 0x80000;
 addi t0,t0, 0x000;
 addi t1, zero, 0x001;

 amoadd.w a0,t1,(t0)
 ret


 END
```

6行目以降は、アセンブラ命令を使ってプログラムしている部分です。
この部分の説明を、下図に示します。アセンブリコードのファイルは、最後にENDがないとエラーになる点も注意してください。

```
f001:
 lui t0, 0x80000;
 addi t0,t0, 0x000;
 addi t1, zero, 0x001;

 amoadd.w a0,t1,(t0)
 ret


 END
```

- lui t0, 0x80000; / addi t0,t0, 0x000; → レジスタt0に0x8000 0000をセットする。
- addi t1, zero, 0x001; → レジスタt1に0x001を設定する（zero+0x001）
- amoadd.w a0,t1,(t0) → t0のアドレスの値をa0に格納し、その値＋t1の結果をt0のアドレスに書き戻す。
- ret → 関数f001から戻る、返り値は a0に入っている

実行結果となるターミナルI/Oの画面を、下図に示します。amoaddは変更の前の値を返していることが分かると思います。

main.c / main()

```
#include <stdio.h>

extern void foo1();

int d2 @  0x80000000;

int main( void ) {
  int t;
  t=f001();
  printf("d2=%d, ret=%d¥n", d2,t);
  t=f001();
  printf("d2=%d, ret=%d¥n", d2,t);
  return 0;
}
```

Terminal I/O

Output: Log file: Off

```
d2=1, ret=0
d2=2, ret=1
```

Input: Ctrl codes Options...

Buffer size: 0

2.8 N extension instructions

1.4.5項の動作モードの説明の中で、「基本的にユーザモードでは割り込みを受けることが出来ません」と述べました。N拡張命令は、ユーザレベルで割り込みや例外を実現するための命令です。ユーザレベルで割り込みを受けるために、CSR（Control and Status Register）と命令の追加が行われています。N拡張命令のために追加された主要なCSRを、下表に示します。

CSR address	名前	説明
0x000	ustatus	User Status Register（ユーザ状態レジスタ）
0x004	uie	User level interrupt enable registe（ユーザ割り込み許可レジスタ）
0x005	utvec	User interrupt handler base address（ユーザトラップハンドラベースアドレス）
0x040	uscratch	Scratch register for user interrupts（ユーザトラップハンドラのためのスクラッチレジスタ）
0x041	uepc	user interrupt PC（ユーザ例外プログラムカウンタ）
0x042	ucause	User exception cause（ユーザトラップ要因）
0x043	utval	User bad address or instruction（ユーザ不良アドレス/命令）
0x044	uip	User interrupt pending register（ユーザ割り込み保留）

また、N拡張命令のために追加された命令を、以下に示します。

- uret

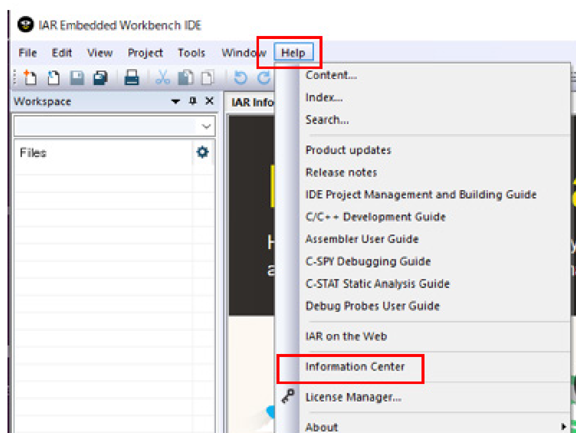
2.9 カスタム命令

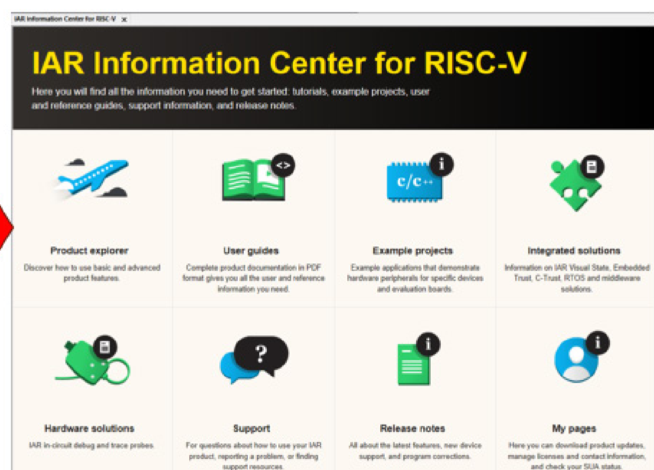
RISC-Vには、命令セットがモジュラ構造になっており、拡張命令やカスタム命令を追加できる、という特徴がありました。EWRISCVにはカスタム命令を扱う仕組みがあるので、ここで説明します。

2.9.1 インフォメーションセンターのサンプルを開く

まず、EWRISCVが提供しているカスタム命令のサンプルプロジェクトを開きます。メニューバーの [Help]-[Information Center] を選択します。

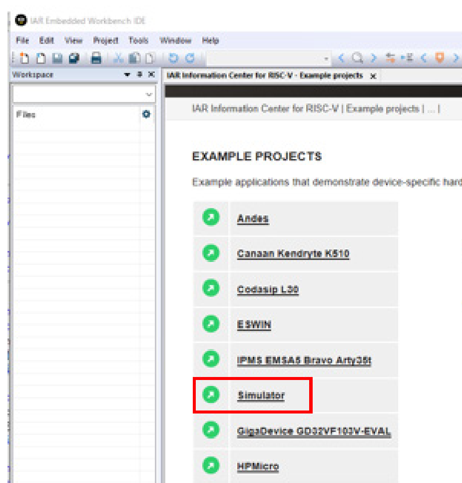
このInformation Center（インフォメーションセンター）では、さまざまな情報を提供しており、その一つにサンプルプロジェクトがあります。それでは、Example Projectsのところをクリックします。

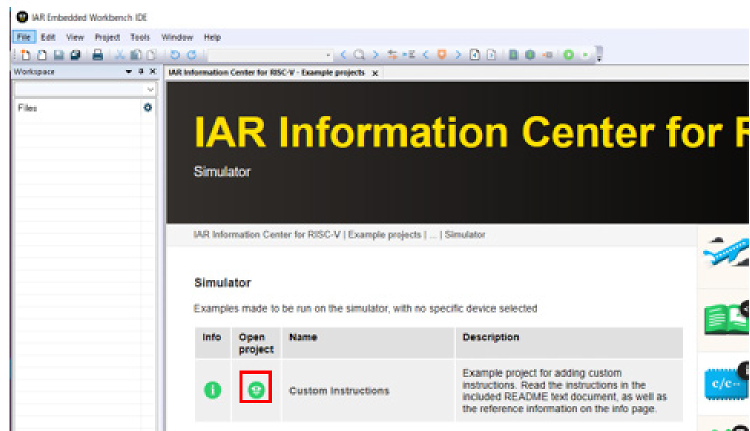




すると、画面が変わります。図に示すようにSimulatorを選択し、続いてCustom Instructions（カスタム命令）の左のOpen projectのところをクリックします。

格納するフォルダを聞いてくるので、適当なフォルダを指定してください。これでカスタム命令のサンプルプロジェクトが開きます。





2.9.2 RISC-Vの命令コードについて

RISC-Vではカスタム命令を作れますが、命令コードは下表のように決められています。今回のサンプルプロジェクトでは、reservedの領域を使用しています。

これは、参考文献 [6] を参考にして今回のサンプルが作られたためです。皆さんが実際にカスタム命令を作る場合は、custom-0、custom-1、custom-2、custom-3などと記載された領域を使ったほうがよいかもしれません。

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

カスタム命令を作る前に、RISC-Vの六つの命令タイプを理解する必要があります。
命令によって即値を指定する場合やレジスタを指定する場合があり、それぞれレジスタの数も異なります。

- Rタイプ：R＝R＋R のように、二つのレジスタ入力と一つのレジスタ出力を指定する命令タイプ
- Iタイプ：R＝R＋Iのように、一つのレジスタ入力と即値と一つのレジスタ出力を指定する命令タイプ
- Sタイプ：メモリへの書き込み（ストア）などを指定する命令タイプ.
- Bタイプ：条件分岐のように、二つのレジスタを指定する命令タイプ
- Uタイプ：20ビットの即値を扱う命令タイプ
- Jタイプ：無条件ジャンプなどで使用する命令タイプ

31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

2.9.3 カスタム命令を決定する

参考文献 [6] をもとにした今回のサンプルは、整数のmod演算（剰余算）を実装しています。その時のカスタム命令を決定します。mod演算命令は R=R%R という形になるので、上述の命令タイプの中のRタイプに該当します。

R-type（一番上の行）のビット構成を見ると、inst（opcodeと記載されている部分）とfunct3、funct7を決定する必要があります。それぞれの値を、以下のとおりとします。
• inst[6:0] の値は、2進数で 1101011（0x6B）とする
• funct3 の値は、2進数で 000（0x0）とする
• funct7 の値は、2進数で 0010000（0x10）とする

2.9.4 C言語コードの中のカスタム命令

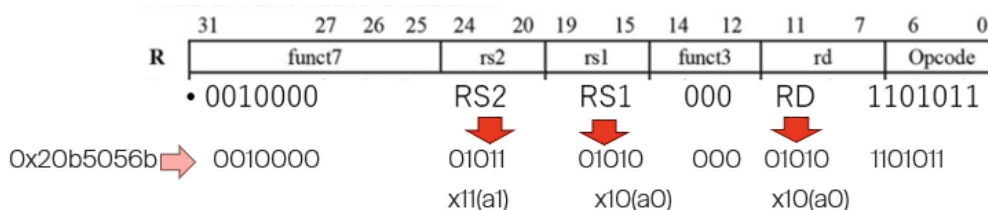
EWRISCVでは、カスタム命令に対応したアセンブリコードを記述できます。詳細はマニュアルを見ていただくとして、ここでは先ほど決定したmod演算命令の呼び出し方法を確認しましょう。以下のように、インラインアセンブラを使ってC言語コードの中でカスタム命令を使っています。.insn r は、Rタイプの命令であることを示すディレクティブです。inst、funct3、funct7の値を0x6B、0x0、0x10と指定し、その後ろでレジスタを指定しています。%0 がデスティネーション、%1 と %2 がソースになります。さらにその後ろで、%0、%1、%2と変数の対応付けを行っています。 %0、%1、%2については、コンパイラが適当なレジスタに割り当てます。C言語上でインラインアセンブラを使う場合に便利な記法です。

```
int modulo(int a, int b) {
  int r;
  __asm(".insn r 0x6B, 0x0,0x10,   %0,%1,%2":"=r"(r) : "r"(a), "r"(b) );
  return r;
};
```

実際にメイクした結果を、以下に示します。今回のカスタム命令は32ビット命令として実装されており、0x20b5056b となっています。EWRISCVの逆アセンブラ表示では、カスタム命令は、残念ながらUnknown32となります。

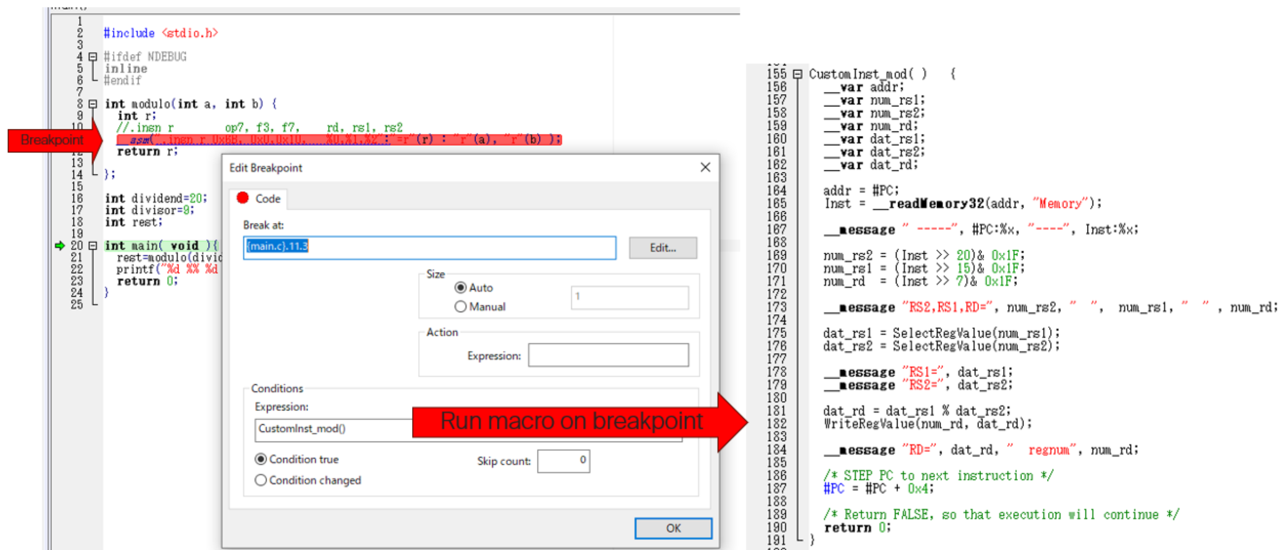
```
  __asm(".insn r 0x6B, 0x0,0x10,   %0,%1,%2":"=r"(r) : "r"(a), "r"(b) );
                modulo:
  0x2000'0320: 0x20b5'056b     Unknown32
  return r;
  0x2000'0324: 0x8082          c.ret
```

0x20b5056b となった機械語を分析してみると、下図のようになります。
機械語ではレジスタ名で指定されるので、rs1はx10、rs2はx11です。ABI名で表記すると、rs1はa0、rs2はa1です。このa0とa1は、関数の第1引き数と第2引き数になります



2.9.5　シミュレータをカスタム命令に対応させる

EWRISCVのシミュレータは、そのままではカスタム命令を実行できませんが、少し工夫することでカスタム命令を実行できるようになります。今回のプロジェクトでは、カスタム命令を置いたアドレスでブレークポイントを設定し、そのブレークポイントに来た時に実行するマクロ関数を用意し、そのマクロ関数でカスタム命令に相当する処理を実行しています。



なお、詳細を知りたい場合は、EWRISCVで実際にサンプルを動かしながら、動作を確認してみてください。今回のサンプルでは、ブレークポイントをセットし、ブレークポイントに達した時にマクロ関数 CustomInst_mod が実行されます。このマクロ関数の中で、レジスタ rd、rs1、rs2を分析し、MOD演算の処理を模擬しています。

このマクロ関数が実行されることで、PC＝PC＋4 となります。これにより、シミュレータがカスタム命令を知らなくても、カスタム命令を実行できました。

2.10　関数呼び出しABIについて
2.10.1　C言語の関数呼び出し

ここでは、C言語の関数を呼び出したり、関数から戻ったりする時、RISC-Vではどのような命令が使われているのかを見てみましょう。

```
int cnt=0;
void add1( )
{
    cnt++;
}
void foo() {
    add1();
}
```

以下のプログラムが、出力されたアセンブラ命令となります。ここでは、最適化レベルを低にしてコンパイルしています関数 foo から関数 add1 を呼び出すところで、call add1 を呼び出しています。

実は、RISC-Vの正式な命令に call はなく、ここで示されている call は JAL ra imm の疑似命令です。
また、ret は jalr x0, 0(x1) の疑似命令で、関数から戻る命令です。x0 はゼロレジスタ、x1 は ra（戻りレジスタ）となります。

```
add1:
     lw        a0, 0x18(gp)
     addi      a0, a0, 1
     sw        a0, 0x18(gp)
     ret
foo:
     addi      sp, sp, -0x10
     sw        ra, 0xC(sp)
     call20    add1
     lw        ra, 0xC(sp)
     addi      sp, sp, 0x10
     ret
```

2.10.2　関数呼び出しの時のルール

ここではRV32Iを例に、関数呼び出しの時のルールについて説明します。まずレジスタを、以下の三つに分類します。

1. スクラッチレジスタ：t0～t6、ft0～ft11、a0～a7、fa0～fa7
2. 保存レジスタ：s0～s11、fs0～fs11
3. 特定用途レジスタ：sp/x2、gp/x3、ra/x1

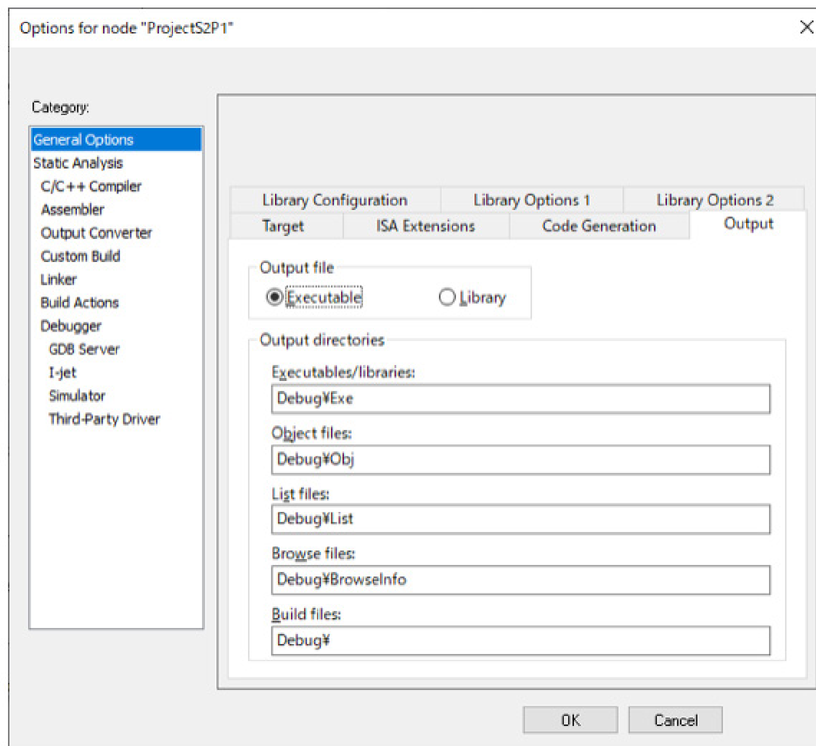
スクラッチレジスタは、関数呼び出しにおいて、関数内で値を自由に書き換えてよいレジスタです。
これに対して保存レジスタは、関数の中で値を変える場合に、プログラムで保存し、関数が終わるときに値を復元する必要があります。
特定用途レジスタには、スタックポインタ sp 、変数アクセスに利用するグローバルポインタ gp、および関数の戻り値を保持する戻りレジスタ ra があります。

関数呼び出しの時にレジスタがどうなっているのかを見ていきます。整数値やポインタ値はレジスタ a0 ～ a7 を用いて渡されます。領域が不足する場合は、スタックを利用します。RV32Iの戻り値は、a0 で返します。64ビットのデータの場合は、a0 と a1 を使って返します。
実際に、関数呼び出しはどのように実現されるのでしょう。以下の関数には、int 型の12個の引き数があります。引き数で渡せるのは8個までなので、四つはスタック渡しになります。以下が生成されたアセンブリコードです。
関数の初めに lw 命令を使い、スタック経由で四つの変数の値をテンポラリレジスタに積んでいます。その他の8変数については、a0 ～ a7 のレジスタで渡しています。

```
f:
    lw      t0, 0(sp)
    lw      t1, 4(sp)
    lw      t2, 8(sp)
    lw      t3, 0xC(sp)
    add     a0, a0, a1
    add     a0, a0, a2
    add     a0, a0, a3
    add     a0, a0, a4
    add     a0, a0, a5
    add     a0, a0, a6
    add     a0, a0, a7
    add     a0, a0, t0
    add     a0, a0, t1
    add     a0, a0, t2
    add     a0, a0, t3
    ret
```

2.11 EWRISCVの出力ファイルについて

本章の最後に、EWRISCVが出力するファイルについて整理しておきます。下図に示すように、プロジェクトのオプションの[General Options]-[Output]タブに、各ファイルの出力先のフォルダが指定されています。



2.11.1 Executables/libraries

Executables/Librariesで指定したフォルダには、ELFの実行形式、もしくはLIBRARYが出力されます。Output ConverterでHEX形式やSREC形式のファイルを生成した場合も、このフォルダに出力されます。

2.11.2 Object files

Object Filesで指定したフォルダには、C/C++ファイルをコンパイルした結果となる .o ファイルが出力されます。

2.11.3 List files

List Filesで指定したフォルダには、MAPファイルやコンパイル時に生成するLISTファイルが出力されます。C/C++コンパイル時に生成するアセンブラファイルも、このフォルダに出力されます。

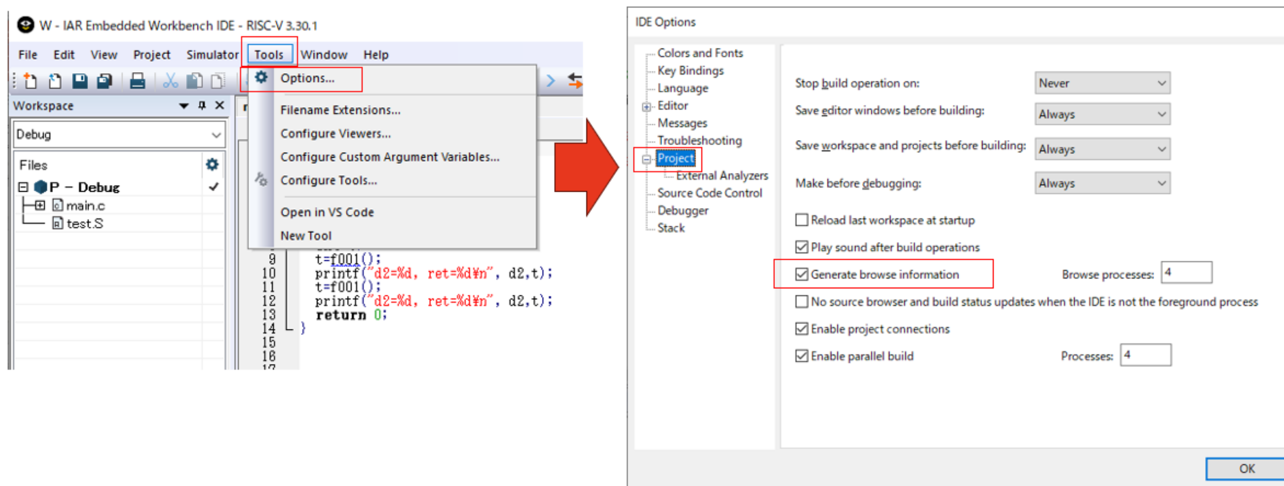
2.11.4 Browse files

EWRISCVには、ソースコードの入力支援機能があります。これは、構造体のメンバを提案したり、ソースコードの入力中に関数や変数などを補完したりする機能です。例えば、入力中に次図に示すような選択肢が表示されますが、この時の情報などが、Browse Filesで指定したフォルダに格納されています。

```
int main( void )
{
  printf("Hello World¥n");
  printf
}
```

```
printf(const char *restrict, ...)                                   int
snprintf(char *restrict, size_t, const char *restrict, ...)         int
sprintf(char *restrict, const char *restrict, ...)                  int
vprintf(const char *restrict, __Va_list)                            int
vsnprintf(char *restrict, size_t, const char *restrict, __Va_list)  int
vsprintf(char *restrict, const char *restrict, __Va_list)           int
int
_Wintt
main()                                                              int
```

ソースコードの入力支援を行うため、EWRISCVは随時、ソースコードを解析しています。この解析機能は、設定によってON / OFFすることが可能です。ソースコードの解析に多大な時間がかかっている時、EWRSICVがフリーズしているような状態になることがあります。その場合は、ソースコードの解析をいったん停止して、原因を突き止めます。
メニューバーの [Tools]-[Options] を選択し、IDE Options画面を開きます。Projectをクリックし、Generate Browse Informationのチェックボックスを有効または無効にすることで、ソースコードの解析機能を制御できます。



2.11.5　MAPファイル

コンパイル後のリンクの際に生成されるMAPファイルについて、簡単に説明しておきます。
オプションの設定やコードの記述などにより出力される内容が変わる場合もありますが、MAPファイルに含まれる基本的な項目は、以下の八つになります。

- リンカ設定オプション
- RUNTIME MODEL ATTRIBUTES
- HEAP SELECTION
- PLACEMENT SUMMARY
- INIT TABLE
- STACK USAGE
- MODULE SUMMARY
- ENTRY LIST

プログラム（サンプル2）から生成されたMAPファイルを参照しながら、各項目について、順番に説明します。

MAP:リンカ設定オプション

MAPファイルの先頭には、以下のようなリンカ(ILINK)のオプションの情報が記載されています。
EWRISCVを利用している時に、リンカのオプションを確認したくなった場合、通常はオプション画面を表示し、さまざまなタブを開いて求める情報を探すことが多いと思います。
しかし、リンカのオプションの情報だけが必要なのであれば、MAPファイルでも確認できます。

```
# IAR ELF Linker V3.20.1.3352/W64 for RISC-
V              30/Aug/2023  09:10:27
# Copyright 2019-2023 IAR Systems AB.
#
#    Output file  =
#        C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Exe\P.o
ut
#    Map file     =
#        C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\List\P.
map
#    Command line =
#        -f
#        "C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Exe\P.
out.rsp"
#        ("C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Obj\m
ain.o"
#        --config_def CSTACK_SIZE=0x1000 --config_def HEAP_SIZE=0x1000
#        --no_out_extension -o
#        "C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Exe\P.
out"
#        --define_symbol __iar_static_base$$GPREL=0 --map
#        "C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\List\P
.map"
#        --config
#        "C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\generic.icf"
#        --entry __iar_program_start --vfe --enable_stack_usage --
text_out
#        locale --debug_lib --core=RV32I)
#
############################################################################
######
```

MAP:RUNTIME MODEL ATTRIBUTES

EWRISCVではランタイムのライブラリをリンクしていきますが、その時のオプションやバージョンなどをMAPファイルに記録しています。サンプル2から生成されたMAPファイルには、以下のように記載されています。

この例では、__SystemLibrary に DLib が使用されています。ここではEWRISCVの標準ライブラリが指定されていますが、ユーザが独自のライブラリを組み込むことも可能です。

```
******
*** RUNTIME MODEL ATTRIBUTES
***
__SystemLibrary              = DLib
__dlib_file_descriptor       = 0
__dlib_version               = 6
__iar_optimize_for_size      = 1
__iar_riscv_atomic           = *
__iar_riscv_base_isa         = rv32i
__iar_riscv_compact          = *
__iar_riscv_div              = *
__iar_riscv_enum_size        = *
__iar_riscv_extension_atomic = *
__iar_riscv_extension_div    = 0
__iar_riscv_extension_mul    = 0
__iar_riscv_fpu              = none
__iar_riscv_i_version        = 2.0
__iar_riscv_mul              = *
__iar_riscv_xbcountzeroes    = *
__iar_riscv_xlen             = 32
__iar_riscv_zba              = *
__iar_riscv_zbb              = *
__iar_riscv_zbc              = *
__iar_riscv_zbe              = *
__iar_riscv_zbf              = *
__iar_riscv_zbm              = *
__iar_riscv_zbp              = *
__iar_riscv_zbr              = *
__iar_riscv_zbs              = *
__iar_riscv_zbt              = *
__rt_version                 = 1
```

MAP:HEAP SELECTION

EWRISCVでは、ヒープで使用するアルゴリズムを選択できます。ヒープ選択についてのMAPファイルの記述を、以下に示します。

```
******
*** HEAP SELECTION
***
The basic heap was selected because --advanced_heap
was not specified, and the application did not appear to
be primarily optimized for speed.
```

MAP:PLACEMENT SUMMARY

PLACEMENT SUMMARYには、セクションやブロックなどの配置の情報が記載されています。前半はリンカ設定ファイルについての情報、後半はセクションの配置情報です。
まず、前半のリンカ設定ファイルについての情報を見ていきます。サンプル2から生成されたMAPファイルには、以下のように記載されています。
この部分には、ブロック情報や配置情報、初期化情報などが記されています。ブロック情報は、サイズを指定したり、セクションをまとめたりするグループとして定義されます。例えば、MVECTOR というブロックは、セクション .mintvec の readonly を含むブロック（アライメントは128）を定義しています。HEAPやCSTACKといったブロックは、サイズを4Kバイト、アライメントを16として定義しています。
配置は place を含む文で示されており、readwrite のデータ、およびブロックのHEAPとCSTACKを、メモリ領域 0x8000’0000〜0x8003’ffff に配置する指定になっています。

これらの情報は、リンカ設定ファイルの内容が反映されたものです。
重要なのは “No sections matched the following patterns:”という記述です。これは、リンカ設定ファイル上には配置のための記述があったのに、プログラム上にはなかった場合に出力されます
この例では、ブロック MVECTOR に配置されるセクション .mintvec の readonly のデータがない、と記されています。これが正しいか否かは、プログラムを作った人にしか分かりません。今回は、プログラムがセクション .mintvec を使用しなかったため、問題ありませんでした。ただしEWRISCVは、直接、参照しない変数や関数を削除してしまうので、間接的に参照される変数や関数が削除されていないかどうかを確認する必要があります。

```
******
*** PLACEMENT SUMMARY
***
build for rom;
"CSTARTUP32":
       place at start of [from 0x2000'0000 to 0x3fff'ffff] {
          ro section .cstartup };
define block MVECTOR with alignment = 128 { ro section .mintvec };
define block MINTERRUPTS
   with maximum size = 64K { ro section .mtext, block MVECTOR };
"ROM32":
       place in [from 0x2000'0000 to 0x3fff'ffff] { ro, block MINTERRUPTS
};
define block HEAP with size = 4K, alignment = 16 { };
define block CSTACK with size = 4K, alignment = 16 { };
"RAM32":
       place in [from 0x8000'0000 to 0x8003'ffff] {
          rw data, block HEAP, block CSTACK };
initialize by copy { rw };
keep symbol __iar_cstart_init_gp;
No sections matched the following patterns:
ro section .mintvec  in block MVECTOR
```

次に、PLACEMENT SUMMARYの後半の部分を見ていきます。サンプル2から生成されたMAPファイルには、以下のように記載されています。この部分には、実際にどのセクションやブロックをどこに配置したのかの情報が記されています。ここには五つの列があり、それぞれSection、Kind、Address、Size、Objectとなっています。Section列にはセクションやブロック名が、Kind列にはその領域の情報が記載されています。そして、Address列にその領域のアドレスが、Size列にサイズが、Object列にそのファイルの定義場所（.o）が記載されています。

```
Section             Kind         Address       Size  Object
  -------             ----         -------       ----  ------
"CSTARTUP32":                                    0x48
  .cstartup           ro code  0x2000'0000       0x48  cstartup.o [5]
                             - 0x2000'0048       0x48
"ROM32":                                        0x334
  .text               ro code  0x2000'0048       0xe8  main.o [1]
  .text               ro code  0x2000'0130       0xa0  __dbg_abort.o [2]
  .text               ro code  0x2000'01d0        0x4  __dbg_break.o [2]
  .text               ro code  0x2000'01d4       0x24  __dbg_xxexit.o [2]
  Initializer bytes   const    0x2000'01f8       0x50          <for RAM32-1>
  .text               ro code  0x2000'0248        0x8  low_level_init.o [4]
  .text               ro code  0x2000'0250       0x3c  data_init.o [4]
  .text               ro code  0x2000'028c        0x4  exit.o [4]
  .text               ro code  0x2000'0290       0x20  cexit.o [4]
  .text               ro code  0x2000'02b0       0x30  copy_init.o [4]
  MINTERRUPTS                  0x2000'0300       0x20          <Block>
    .mtext            ro
code  0x2000'0300     0x20  default_interrupt_handler.o [3]
  .iar.init_table     const    0x2000'0320       0x1c  - Linker created -
  .text               ro code  0x2000'033c       0x20  main.o [1]
  .text               ro code  0x2000'035c       0x20  zero_init.o [4]
                             - 0x2000'037c      0x334
"RAM32", part 1 of 3:                            0x50
  RAM32-1                      0x8000'0000       0x50          <Init block>
    .data             inited   0x8000'0000       0x28  main.o [1]
    .data             inited   0x8000'0028       0x28  main.o [1]
                             - 0x8000'0050       0x50
"RAM32", part 2 of 3:                            0x50
  .bss                zero     0x8000'0050       0x28  main.o [1]
  .bss                zero     0x8000'0078       0x28  main.o [1]
                             - 0x8000'00a0       0x50
"RAM32", part 3 of 3:                          0x1000
  CSTACK                       0x8000'00a0     0x1000          <Block>
    CSTACK            uninit   0x8000'00a0     0x1000          <Block tail>
                             - 0x8000'10a0     0x1000

Unused ranges:
From             To          Size
         ----            --          ----
  0x2000'037c  0x3fff'ffff  0x1fff'fc84
  0x8000'10a0  0x8003'ffff     0x3'ef60
```

Kind列でよく使用される項目の分類を、以下に示します

- ro code： readonly 属性のコード
- const：定数などのデータ
- inited：ゼロ以外の初期化変数や領域
- zero：ゼロの初期化変数や領域
- uninit：初期化しない変数や領域

MAP:INIT TABLE

INIT TABLEには、変数の初期化についての情報が記載されています。サンプル2から生成されたMAPファイルには、以下のように記されています。
INIT TABLEは、コンパイラに変数の初期化を実行させた場合に出力されます。初期化する変数がまったくない場合は出力されません。C言語の変数の初期化には、ゼロ初期化と非ゼロ初期化の2通りがあります。一般的には、ゼロ初期化は .bss 領域の、非ゼロ初期化は .data の変数初期化となります。

今回の例では、Zero の部分に、使用する初期化関数 __iar_zero_init2 と、ゼロ初期化する領域の先頭アドレス、およびサイズが記載されています。
また、Copyの部分には、使用する初期化関数 __iar_copy_init2 と、非ゼロ初期化する領域の先頭アドレス、およびサイズが記載されています。
スタートアップコードを変更したり、自分で作成したりする場合は、この初期化関数を正しく呼ぶ必要があります。

```
**********
*** INIT TABLE
***
Address      Size
          -------      ----
Zero (__iar_zero_init2)
    1 destination range, total size 0x50:
          0x8000'0050   0x50
Copy (__iar_copy_init2)
    1 source range, total size 0x50:
          0x2000'01f8   0x50
    1 destination range, total size 0x50:
          0x8000'0000   0x50
```

MAP:STACK USAGE

STACK USAGEは、リンカのスタック解析機能を有効にする（具体的には、LinkerカテゴリのAdvancedタブにあるEnable stack usage analysisのチェックボックスを有効にする）ことで出力されます。細かい指定を行っていない場合、EWRISCVはプログラムエントリからスタック量の解析を行います。サンプル2から生成されたMAPファイルには、以下のように記載されています。今回の例では、最初に Program entry と interrupt と Uncalled function の3項目の情報が表示されています。Program entryについては、リセット直後に実行を開始したプログラムエントリから、スタックの使用量の解析を行います。
割り込みについては、__interrupt などで指定したハンドラを interrupt として解析します。どの関数からも呼ばれない関数（Uncalled function）が存在する場合は、その情報も表示されます。

```
******
*** STACK USAGE
***
Call Graph Root Category  Max Use  Total Use
  -----------------------  -------  ---------
  interrupt                    112        112
  Program entry                 48         48
  Uncalled function              0          0
Program entry
  "__iar_program_start": 0x2000'0000
Maximum call chain                               48  bytes
"__iar_program_start"                           0
    "exit"                                          0
    "_exit"                                        16
    "__exit"                                       32
    "__DebugBreak"                                  0
interrupt
  "__iar_default_minterrupt_handler": 0x2000'0300
Maximum call chain                              112  bytes
"__iar_default_minterrupt_handler"             16
    "abort"                                        64
    "__exit"                                       32
    "__DebugBreak"                                  0
Uncalled function
  "hoo": 0x2000'033c
Maximum call chain                                0  bytes
"hoo"                                           0
```

それではスタック解析の値はどれだけ信用できるのでしょうか?残念ながらいくつか課題があります。スタック解析が正しく実施されない場合は、以下の状況が考えられます。

- プログラムを作成する際に、関数ポインタを使ったり、ジャンプ先アドレスを固定アドレスで記述したりした場合
- アセンブリ言語で記述した関数が含まれる場合r
- 再帰関数などで複数回、同じ関数が実行される場合

また、リアルタイムOSなどを使用した場合は、タスクごとのスタック使用量の情報が必要になると思います。正しく解析したい、あるいはさらに細かい情報が欲しいという時、EWRISCVには二つの対処法があります。#pragma を使ってソースコード上で指示する方法と、別ファイルで指示する方法です。

詳細はマニュアルを確認していただきたいのですが、まず、ソースコード上で指示する方法の簡単な例を、以下に示します。
task1とtask2は、リアルタイムOSのタスクを想定しています。タスクごとに個別のスタック解析が必要となる場合、関数の前に #pragma call_graph_root と記述します。
関数ポインタを使うなどした時に、EWRISCVが関数の呼び出しを理解できない場合は、#pragma calls で呼び出す関数を指定します。

```
#pragma call_graph_root
void task1(void ) {
  . . .
}
#pragma call_graph_root
void task2(void ) {
  . . .
}
extern int (*fp)(void);
int main( void )
{
#pragma calls=foo,goo
  fp();
```

次に、別ファイルで指示する方法の例を、以下に示します。

call graph root: task1, task2;
possible calls fp: foo, goo;
max recursion depth recursive: 32;

別ファイルの方が指定できる内容が多くなります。以下の例では、1行目と2行目は #pragma を使ってソースコード上で指定した内容と同じです。
3行目は再帰呼び出しのための記述です。再帰呼び出しがあると、それが何回実行されるのかコンパイラには分かりません。ユーザが上限を指定することで、スタック解析を正しく実行することが可能となります。

MAP: MODULE SUMMARY

MODULE SUMMARYには、ファイルやライブラリごとのコードのサイズ、readonly データ(定数など、値の変更がないデータ) のサイズ、および readwrite データ(変数など) のサイズなどの情報が出力されます。モジュールのサイズなので、リンカの最適化などにより、最終的な実行ファイルのサイズと若干の違いが出る可能性があります。最初にユーザコードのサイズが表示され、その後にライブラリのサイズが記されています。

ライブラリについては、名称に意味があります。例えば dbg-rv32i.a はデバッグ用のI/Oライブラリで、命令セットがRV32iであることを示しています。また、di-rv32iはデフォルトの割り込みハンドラを提供するライブラリ、dl-rv32i.a は printf / scanf などのライブラリ、dlmath-rv32i.a はC言語の数学関数ライブラリです。なお、拡張命令によって使用するライブラリが変わる場合があります。

```
******
*** MODULE SUMMARY
***
Module                        ro code  ro data  rw data
    ------                    -------  -------  -------
command line/config:
    ---------------------------------------------------
    Total:
C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Obj: [1]
    main.o                        264       80      160
    ---------------------------------------------------
    Total:                        264       80      160
dbg-rv32i.a: [2]
    __dbg_abort.o                 160
    __dbg_break.o                   4
    __dbg_xxexit.o                 36
    ---------------------------------------------------
    Total:                        200
di-rv32i.a: [3]
    default_interrupt_handler.o    32
    ---------------------------------------------------
    Total:                         32
dl-rv32i.a: [4]
    cexit.o                        32
    copy_init.o                    48
    data_init.o                    60
    exit.o                          4
    low_level_init.o                8
    zero_init.o                    32
    ---------------------------------------------------
    Total:                        184
dlmath-rv32i.a: [5]
    cstartup.o                     72
    ---------------------------------------------------
    Total:                         72
Linker created                              28    4'096
-------------------------------------------------------
    Grand Total:                  752      108    4'256
```

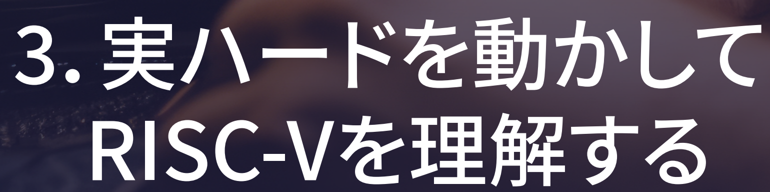
MAP:ENTRY LIST

組込み技術者がもっともよく利用する部分が、このENTRY LISTになると思います。ENTRY LISTでは、関数や変数の配置状況を確認できます。Entryは、関数や変数、ラベル名です。その右側にアドレス、サイズ、タイプ、Gb / Wk / Lc、オブジェクトが記載されています。タイプについては、Codeが関数、Dataが定数や変数となります。その右隣のGbは、グローバルに定義された変数・関数であることを示します。Wkは weak 属性がついた関数・変数であることを、Lcはローカルに定義された変数・関数であることを示します。

オブジェクトについては、リンカによって作成されたものと、ファイル名が指定されたものがあります。ファイル名の後ろの角かっこの数字（[1]、[2]、[3] など）は、リストの下の凡例（[1] = ...、[2] = ...、[3] = ... の部分）により、どのファイルからリンクされたのかを示しています。ライブラリについても、もともとはC言語やアセンブリ言語のファイルなので、*.o という名称があり、凡例のところにその実体が示されています。
最後に、プログラムのサイズに関する情報、およびビルド時のワーニングやエラーの数なども記載されています。

```
*** ENTRY LIST
***
Entry                       Address  Size  Type      Object
-----                       -------  ----  ----      ------
.iar.init_table$$Base    0x2000'0320          --   Gb  - Linker created -
.iar.init_table$$Limit   0x2000'033c          --   Gb  - Linker created -
CSTACK$$Base             0x8000'00a0          --   Gb  - Linker created -
CSTACK$$Limit            0x8000'10a0          --   Gb  - Linker created -
MINTERRUPTS$$Base        0x2000'0300          --   Gb  - Linker created -
MINTERRUPTS$$Limit       0x2000'031c          --   Gb  - Linker created -
Region$$Table$$Base      0x2000'0320          --   Gb  - Linker created -
Region$$Table$$Limit     0x2000'033c          --   Gb  - Linker created -
__DebugBreak             0x2000'01d0   0x4  Code   Gb  __dbg_break.o [2]
__exit                   0x2000'01d4  0x28  Code   Gb  __dbg_xxexit.o [2]
__iar_copy_init2         0x2000'02b0  0x30  Code   Gb  copy_init.o [4]
__iar_cstart_init_gp     0x2000'0004        Code   Gb  cstartup.o [5]
__iar_data_init2         0x2000'0250  0x3c  Code   Gb  data_init.o [4]
__iar_default_minterrupt_handler
                         0x2000'0300  0x24  Code   Gb  default_interrupt_ha
```

```
ndler.o [3]
__iar_program_start      0x2000'0000         Code  Wk  cstartup.o [5]
__iar_static_base$$GPREL {Abs}
                                 0x0         Data  Gb  <internal module>
__iar_zero_init2         0x2000'035c  0x20   Code  Gb  zero_init.o [4]
__low_level_init         0x2000'0248   0x8   Code  Gb  low_level_init.o [4]
_exit                    0x2000'0290         Code  Gb  cexit.o [4]
abort                    0x2000'0130  0xa8   Code  Gb  __dbg_abort.o [2]
da                       0x8000'0000  0x28   Data  Gb  main.o [1]
db                       0x8000'0028  0x28   Data  Gb  main.o [1]
exit                     0x2000'028c   0x8   Code  Gb  exit.o [4]
ha                       0x8000'0050  0x28   Data  Gb  main.o [1]
hb                       0x8000'0078  0x28   Data  Gb  main.o [1]
hoo                      0x2000'033c  0x20   Code  Gb  main.o [1]
main                     0x2000'0048  0xe8   Code  Gb  main.o [1]
[1] = C:\Users\Documents\Work\Output\riscv_book\Samples\S2P02\Debug\Obj
[2] = dbg-rv32i.a
[3] = di-rv32i.a
[4] = dl-rv32i.a
[5] = dlmath-rv32i.a
[6] = dln-rv32i.a
752 bytes of readonly  code memory
    108 bytes of readonly  data memory
  4'256 bytes of readwrite data memory
Errors: none
Warnings: 1
```



3. 実ハードを動かして RISC-Vを理解する

3. 実ハードを動かしてRISC-Vを理解する

3.1　GigaDevice社のRISC-Vマイコンを動かす

ここでは評価ボードとして、GD32VF103（型名はGD32VF103CBT6）を搭載する中国Seeed Technology社製の「Wio Lite RISC-V」を使用します。このボードは、2024年4月現在、秋月電子通商の通販サイトにて500円で販売されています（ただし、在庫限りの数量限定品。在庫僅少）。デバッガの接続方法を含めて、開発手順の概要を説明するのにちょうど良いボードなので、最初に取り上げます。なお、本ボードには中国Espressif Systems社のWi-Fiモジュールである「ESP8266」も実装されていますが、ここではGD32VF103のみを使用しています。GD32VF103の特徴として紹介されている記述の中から、RISC-VのISAと関係しそうなものを抜粋して、以下に示します。

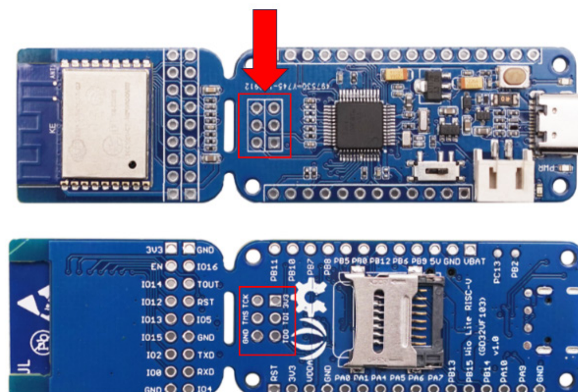
- RISC-V compliant little-endian RV32IMAC (32GPRs)〔RISC-V規格に準拠するリトルエンディアンのRV32IMAC（32個の汎用レジスタを搭載）〕
- Machine (M) and User (U) Privilege levels support〔マシンモード（M）とユーザモード（U）の特権レベルをサポート〕
- Single-cycle hardware multiplier and Multi-cycles hardware divider support（シングルサイクルのハードウェア乗算器と複数サイクルのハードウェア除算器をサポート
- Misaligned load/store hardware support（整列されていないロード/ストア命令のハードウェアをサポート）
- Atomic instructions hardware support（アトミック命令のハードウェアをサポート）
- Non-maskable interrupt (NMI) support〔マスク不可割り込み（NMI）をサポート〕
- WFI (Wait for Interrupt) support〔WFI（割り込み待ち）をサポート〕
- WFE (Wait for Event) support〔WFE（イベント待ち）をサポート〕

3.1.1　デバッグプローブを接続する

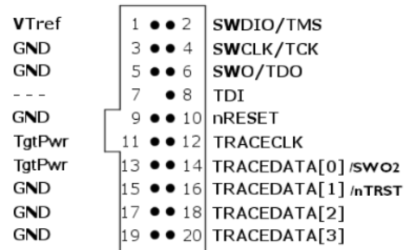
ボードに搭載されたマイコンにソフトウェアを書き込んだり、ソフトウェアをデバッグしたりするためには、ハードウェアデバッガと呼ばれる機器が必要です。
ここでは、IARのデバッグプローブ（ハードウェアプローブ）「I-jet」を使用します。デバッグプローブの接続イメージを示します。

I-jetは、マイコンのJTAGポートに接続して使います。JTAGポートには、以下の六つのピン（端子）があります。Wio Lite RISC-Vボードの赤い線で囲ったところが、JTAGポートの6ピンです

- TCK：クロック供給
- TMS：TAPコントローラの遷移に使用
- TDO：データ出力
- TDI：データ入力
- 3V3：3V電源出力
- GND：グラウンド（接地）

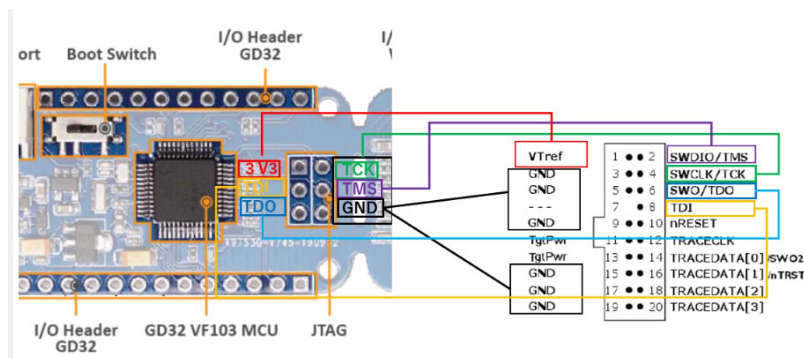


I-jet側のポートも見てみます。こちらのポートは、下図のような20ピンのMIPI-20コネクタに引き出されています。



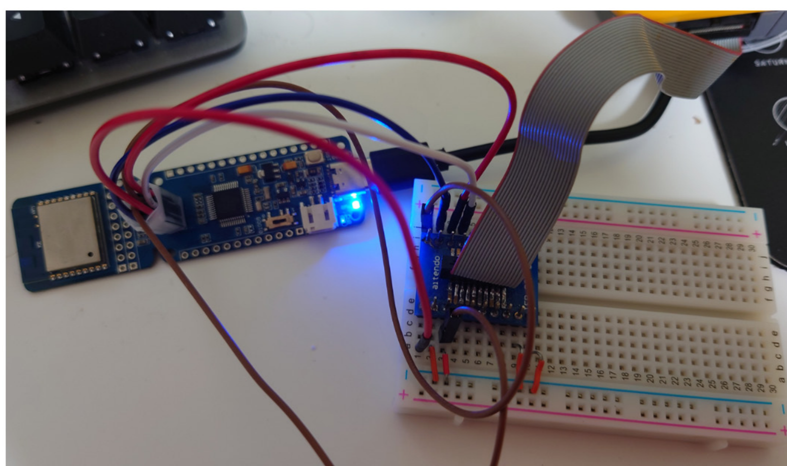
Wio Lite RISC-V上のTCK、TMS、TDO、TDI、GNDの各ピンは、それぞれMIPI-20コネクタの同じ名前のピンに接続します。

Wio Lite RISC-V上の3V3ピン（3V電源）は、MIPI-20コネクタのVTrefピンに接続します。このVTrefとの接続は、信号レベルを設定するために必須です。



MIPI-20コネクタの信号だと扱いにくいので、GD32のJTAGとの接続するためブレッドボードとピッチ変換基板を入れて接続しています。

しかし、高速な信号を扱う場合にはこのタイプの接続は避けた方が良いでしょう。

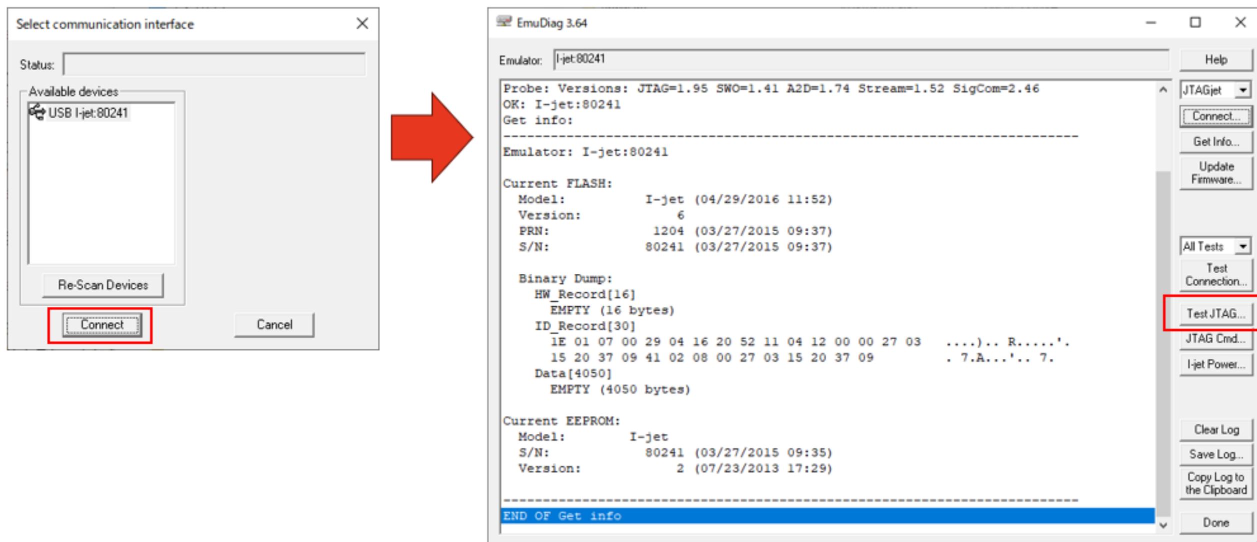


3.1.2　I-jetでの接続確認

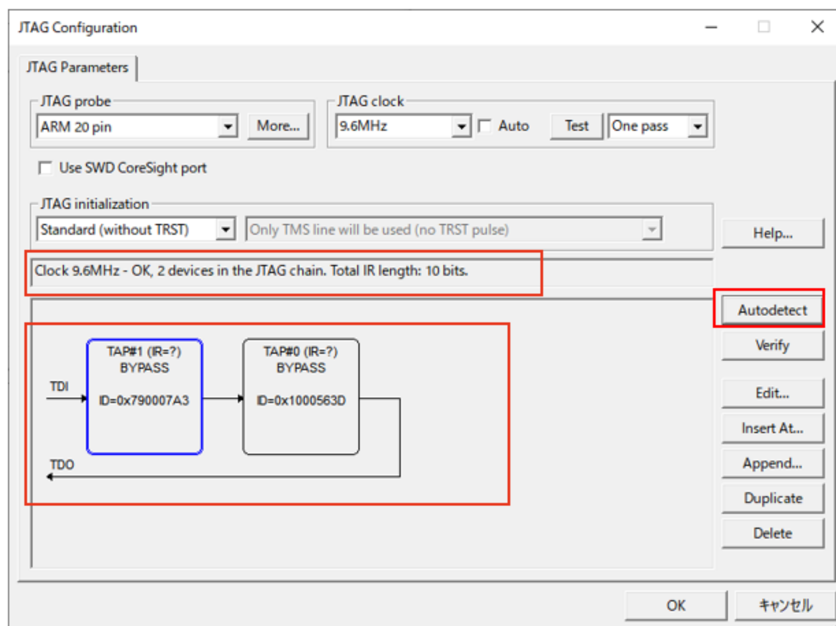
Wio Lite RISC-VとI-jetと開発用パソコンを接続したら、正しく接続できているかどうかをチェックします。一般にマイコンの評価・開発ボードは、デバッグプローブをきちんと接続すれば確実に動くようになっていますが、今回は、ブレッドボードなどを利用して1ピンずつ接続しているので、不具合が発生する可能性があります。

開発用パソコンにEWRISCVをインストールすると、ツールがインストールされたフォルダ以下の \riscv\bin\jet にEmudiag.exeというプログラムが配置されます。

Emudiag.exeを起動し、下図のように [Connect] ボタンをクリックし、さらに [Test JTAG...] ボタンをクリックします。

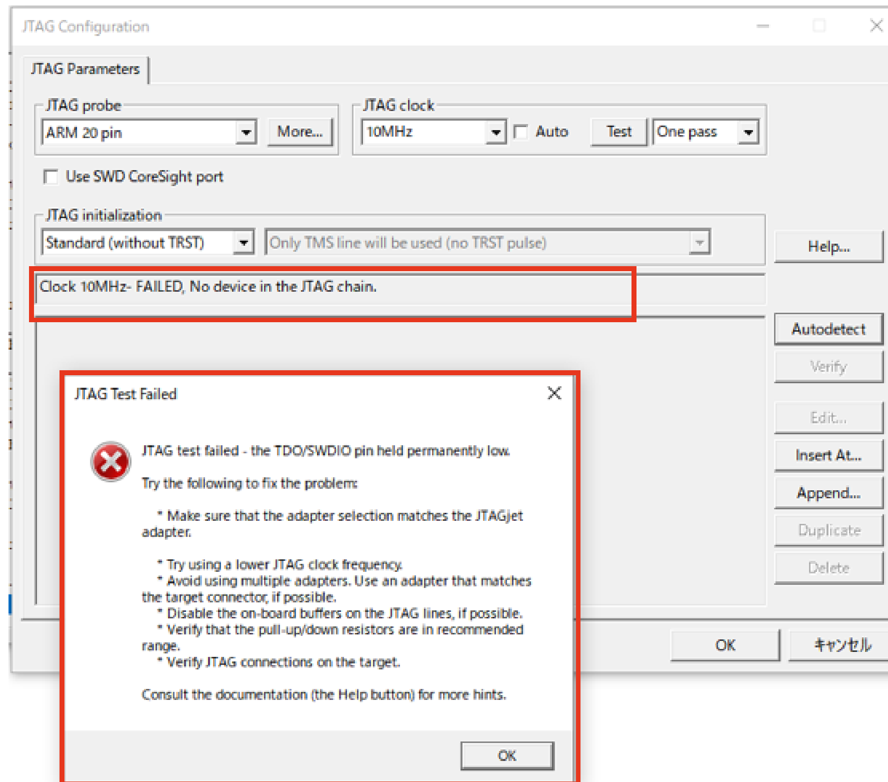


JTAG Configuration画面が出てきます。ここで、[Autodetect] ボタンをクリックします。正しく接続できていれば、赤の線で囲んだ箇所に情報が表示されます。



うまく接続できなかった時の例を示します。これは、正しく接続された状態からTMSの信号線を外した場合です。

JTAGポートが正しく接続されていないと、このようなエラーメッセージが出てきます。このような時は、落ち着いて一つ一つ接続を確かめてください。



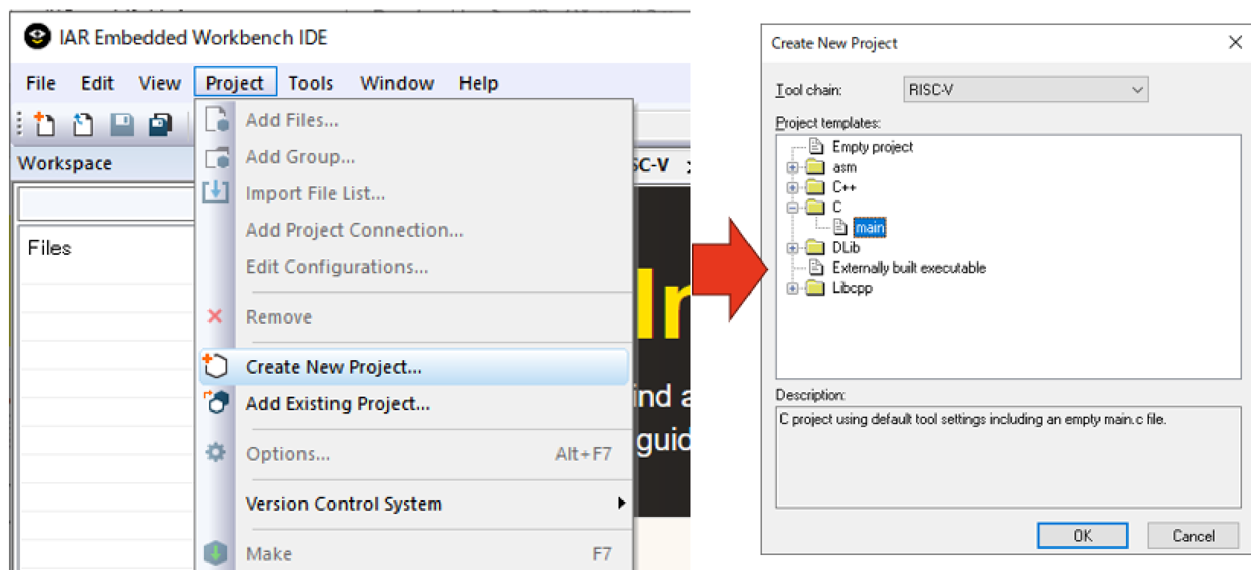
接続確認が終わったら、[OK]ボタンと[Done]ボタンをクリックして、Emudiag.exeを終了させます。Emudiag.exeが起動したままだと、EWRISCVからデバッグの制御を行えなくなる可能性があります。

3.1.3 サンプル5：GPIOを用いたLED点滅

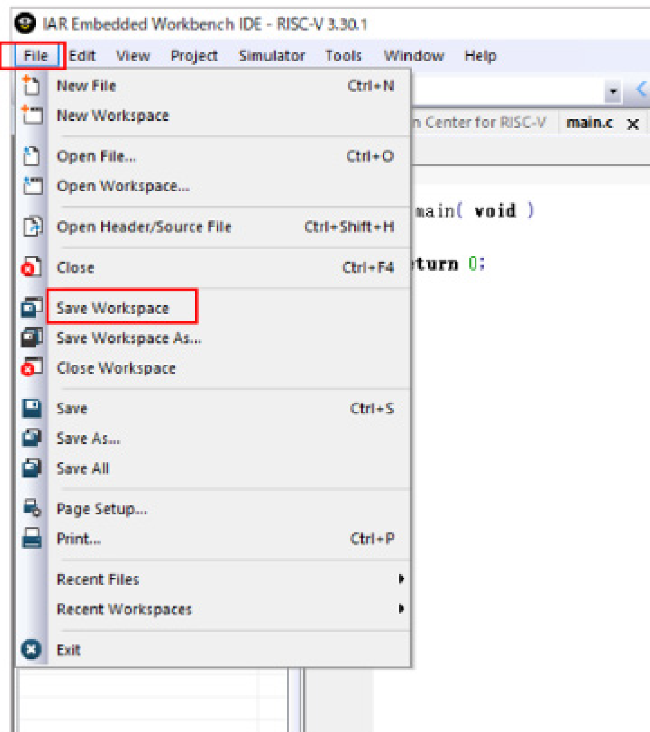
Wio Lite RISC-Vボードの回路図を見て、LEDの接続を確認します。PA8にLEDが接続されていることが分かります。

それでは、プロジェクトを作成していきます。下図に示すように、EWRISCVのメニューバーの [Project]-[Create New Project...] を選択し、C のフォルダの下の main をクリックします。

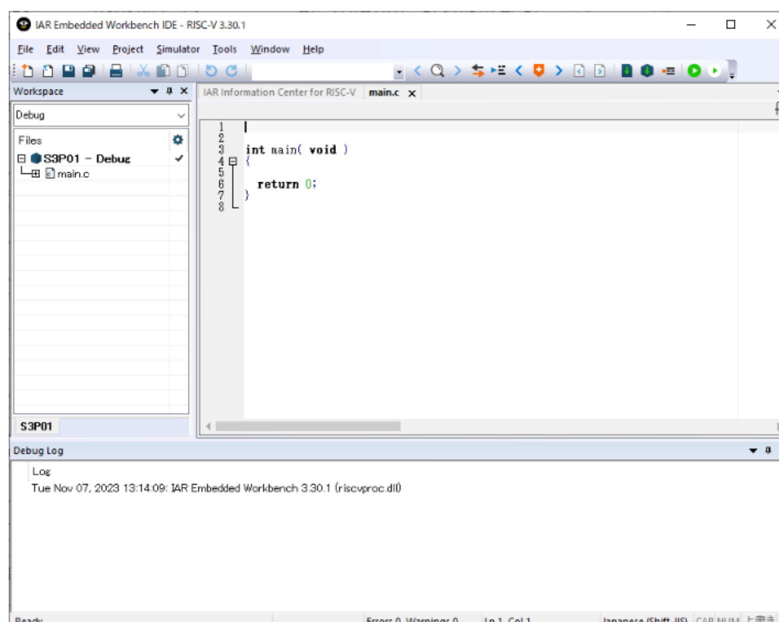
次に、プロジェクトを作成するフォルダを選び、プロジェクト名を付けます。ここで、適宜、プロジェクトファイル名を指定します。今回は、S3P01 という名前にしました。



続いて、メニューバーの [File]-[Save Workspace] を選択し、適宜、ワークスペースファイル名を指定します。

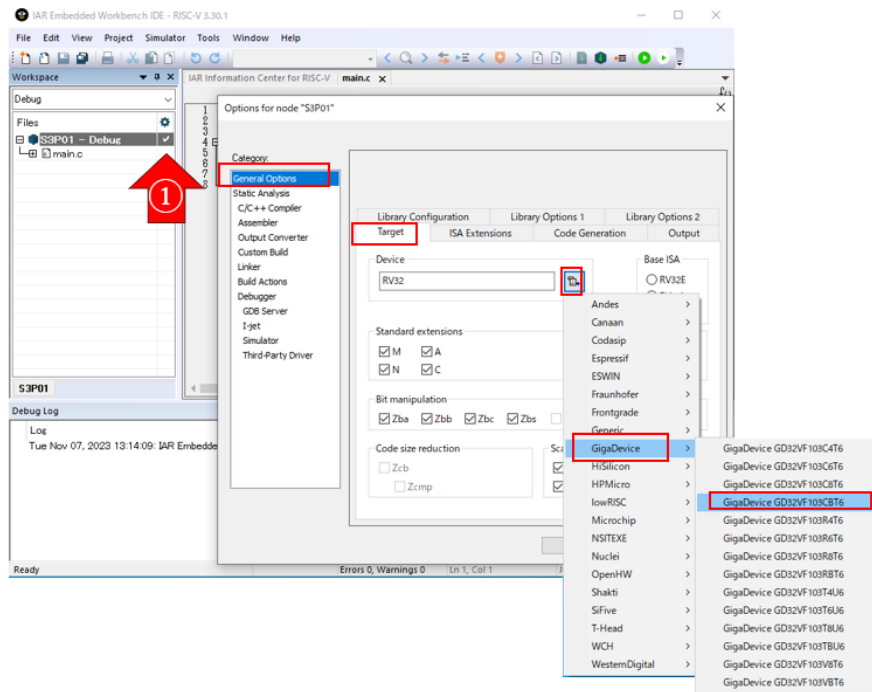


これで main.c が一つあるプロジェクトとワークスペースが作成されました。下図のようになっていると思います。

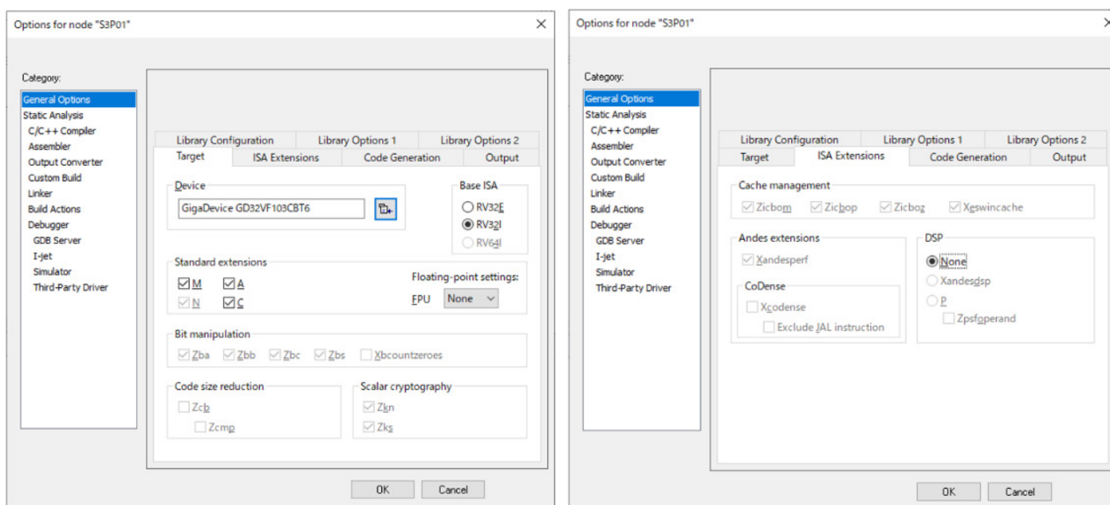


EWRISCVで、使用するマイコンを指定しましょう。下図に示すワークスペース画面で、プロジェクト名（S3P01）の横にある✔（①のチェックマーク）をダブルクリックすると、オプション設定の画面が開きます。

左側のカテゴリのGeneral Optionsを選択し、TargetタブのDeviceのところで、GigaDevice社のGD32VF103CBT6を指定します。



2章では拡張命令の細かい設定方法を説明しましたが、EWRISCVがサポートしているデバイスやボードの場合は、デバイス名やボード名を指定するだけで、拡張命令などの設定が自動的に行われます。
通常、命令についての設定は、このままで良いと思います。ただし、複数のマイコンを使用する時など、特定のオプション設定にそろえたい場合もあります。その場合は、ユーザ自身で設定を変更してください。
下図に、GD32VF103CBT6を指定した時の命令に関するオプション設定の画面を示します。

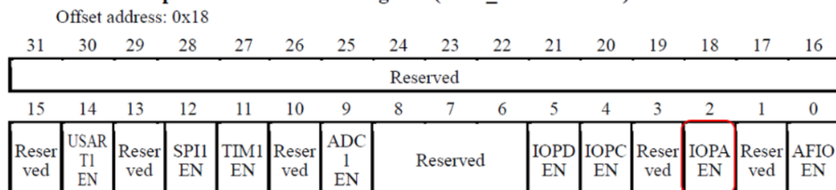


EWRISCVがサポートしているデバイス（またはボード）を指定した場合、周辺アクセス用のヘッダファイルも提供されます。#include <gigadevice/ioGD32VF103.h> と記述することで、周辺機能（ペリフェラル）へのアクセスが可能となります。LEDを点滅させるためには、GPIOAのクロックを有効にし、PA8を出力、かつPushPullに設定する必要があります。以下にその記述（サンプル5）を示します。

```
#include <gigadevice/ioGD32VF103.h>
void delay(void ) {
  volatile int i;
  for (i=0; i< 500000;i++);
}
int main( void ){
RCU_APB2EN_bit.PAEN=1; /* GPIO port A clock enable */
  GPIOA_CTL1_bit.MD8  = 0x1;  /*PA8: Output 10MHz */
  GPIOA_CTL1_bit.CTL8 = 0x0;  /*PA8: push-pull    */

  while( 1 ) {
     GPIOA_OCTL_bit.OCTL8 = 0; /* Drive Low(0) */
     delay();
     GPIOA_OCTL_bit.OCTL8 = 1; /* Drive High(1) */
     delay();
  }
return 0;
}
```

ヘッダファイルでどのように定義されているかを、APB2 Peripheral Clock Enable Register（APB2EN）で確認してみます。下図の左側がヘッダの中のAPB2ENを定義した部分、右側がレジスタビットです。定義を見ると、union になっており、32ビットでアクセスする場合は RCU_APB2EN を使います。ビットフィールドでアクセスする場合は RCU_APB2EN_bit.XXXX（XXXXは個別のフィールド名）を使います
先ほどの記述（サンプル5）ではPAENのビットだけを設定するので、RCU_APB2EN_bit.PAEN=1; となっています。GPIOAのCTL1やOCTLについても同様です。

```
__no_init volatile union
  {
    unsigned int RCU_APB2EN;
    struct
    {
      unsigned int AFEN     : 1;
      unsigned int          : 1;
      unsigned int PAEN     : 1;
      unsigned int PBEN     : 1;
      unsigned int PCEN     : 1;
      unsigned int PDEN     : 1;
      unsigned int PEEN     : 1;
      unsigned int          : 2;
      unsigned int ADC0EN   : 1;
      unsigned int ADC1EN   : 1;
      unsigned int TIMER0EN : 1;
      unsigned int SPI0EN   : 1;
      unsigned int          : 1;
      unsigned int USART0EN : 1;
      unsigned int          : 17;
    } RCU_APB2EN_bit;
  } @ 0x40021018;
```


3.4.7 APB2 Peripheral Clock Enable Register (RCC_APB2PCENR)

Offset address: 0x18

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

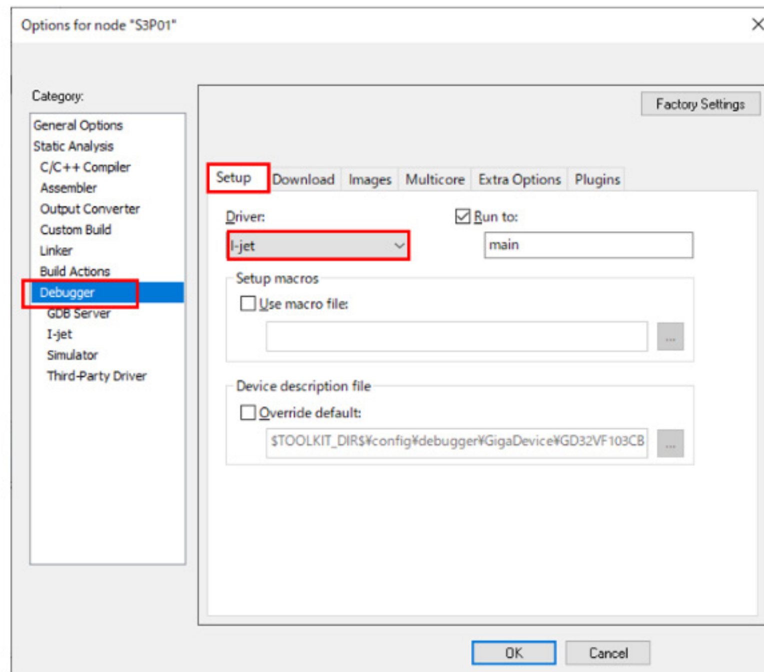
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	USART1 EN	Reserved	SPI1 EN	TIM1 EN	Reserved	ADC1 EN	Reserved			IOPD EN	IOPC EN	Reserved	IOPA EN	Reserved	AFIO EN

EWRISCVがサポートしているデバイス（またはボード）については、ヘッダファイルに加えて、通常、リンカ設定ファイルも提供されます。
今回は、デバイスとしてGD32VF103CBT6を指定した時に、リンカ設定ファイルもいっしょに選択されています。リンカ設定ファイルの中身を理解していなくても、プログラムの作成を始められます。

```
///////////////////////////////////////////////////////////////////////
// RISC-V ilink configuration file
// for the GD32VF103CBT6
//
define exported symbol _link_file_version_2 = 1;
define exported symbol _auto_vector_setup = 1;
define exported symbol _max_vector = 96;
if (isdefinedsymbol(_disable_clic))
{
  define exported symbol _CLINT = 1;
}
else
{
  define exported symbol _uses_clic=1;
  define exported symbol _CLIC_GIGADEVICE = 1;
}
keep symbol __iar_cstart_init_gp; // defined in cstartup.s
keep { ro section .alias.hwreset };
define memory mem with size = 4G;
define region ROM_region32 = mem:[from 0x08000000 to 0x0801FFFF];
define region RAM_region32 = mem:[from 0x20000000 to 0x20007FFF];
initialize by copy { rw };
do not initialize  { section *.noinit };
define block CSTACK with alignment = 16, size = CSTACK_SIZE { };
define block HEAP   with alignment = 16, size = HEAP_SIZE   { };
define block MVECTOR with alignment = 128, size = _max_vector*4 { ro
section .mintvec };
if (isdefinedsymbol(_uses_clic))
{
  define block MINTERRUPT with alignment = 128 { ro section .mtext };
  define block MINTERRUPTS { block MVECTOR,
                             block MINTERRUPT };
}
else
{
  define block MINTERRUPTS with maximum size =  64k { ro section .mtext,
                                                     midway block
MVECTOR };
}
define block RW_DATA with static base GPREL { rw data };
"CSTARTUP32" : place at start of ROM_region32 { ro section
.alias.hwreset,
                                                ro section .cstartup };
"ROM32":place in ROM_region32        { ro,
                                       block MINTERRUPTS };
"RAM32":place in RAM_region32        { block RW_DATA,
                                       block HEAP,
                                       block CSTACK };
```

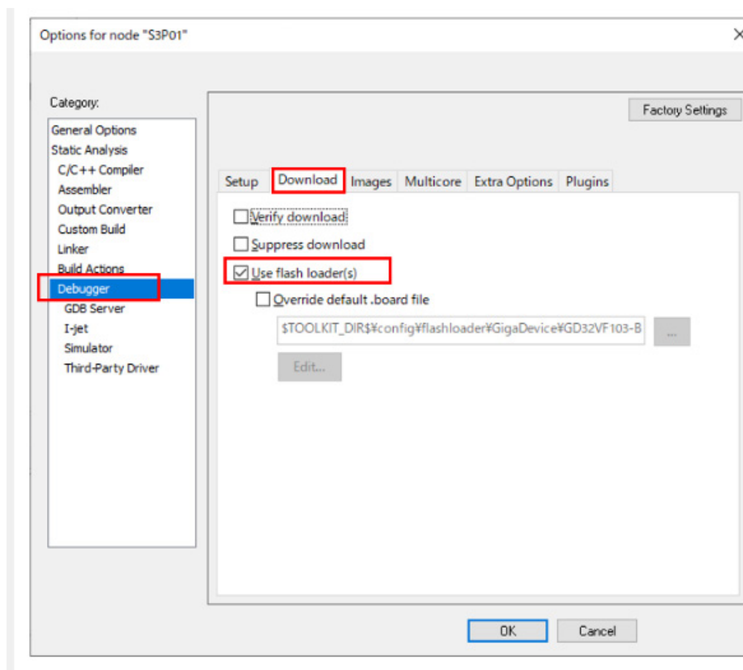
3.1.4　デバッガの設定を行い、実行開始

プログラムの準備が出来たので、プログラムをボード上のRISC-Vマイコンにダウンロードして実行したいと思います。今回はデバッグプローブとしてI-jetを使用するので、そのための設定を行います。プロジェクトのオプション画面を開き、下図のように左側のカテゴリのDebuggerを選択し、SetupタブのDriverのところでI-jetを指定します。



EWRISCVがサポートしているデバイス（またはボード）を指定した場合、フラッシュローダ（マイコンのRAMに転送され、内蔵フラッシュROM、または外付けのフラッシュROMのデータの書き換えを可能とする小規模なプログラム）を使用できます。

GD32VF103CBT6では、オンチップ・フラッシュのサポートされています。下図に示すように、Downloadタブの Use flashu loader(s) のチェックボックスを有効にします。これを有効にしていないと、フラッシュROMへの書き込みが行えないので注意してください。



なお、EWRISCVがサポートしているデバイス（または ボード）であっても、デバイス内蔵（オンチップ）のフラッシュROMへの書き込みしか対応していない場合や、特定のボードの外部フラッシュROMへの書き込みしか対応していない場合があります。サポート外のフラッシュROMに書き込みたい場合は、ユーザ自身で対応する必要があります。その場合は、EWRISCVがインストールされたフォルダ以下にフラッシュローダのマニュアル（FlashLoaderGuide.ENU.pdf）があります。

```
\riscv\doc
```

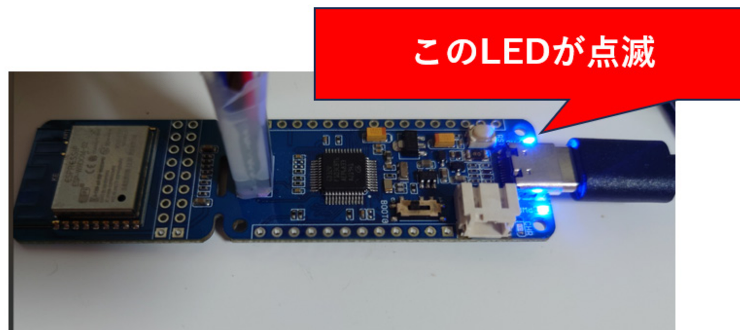
フラッシュローダのサンプルのソースコードは以下にあります。

```
\riscv\src\flashloader
```

たとえば、GD32VF103CBT6のフラッシュローダは以下にあります。

```
\riscv\src\flashloader\GigaDevice\GD32VF103
```

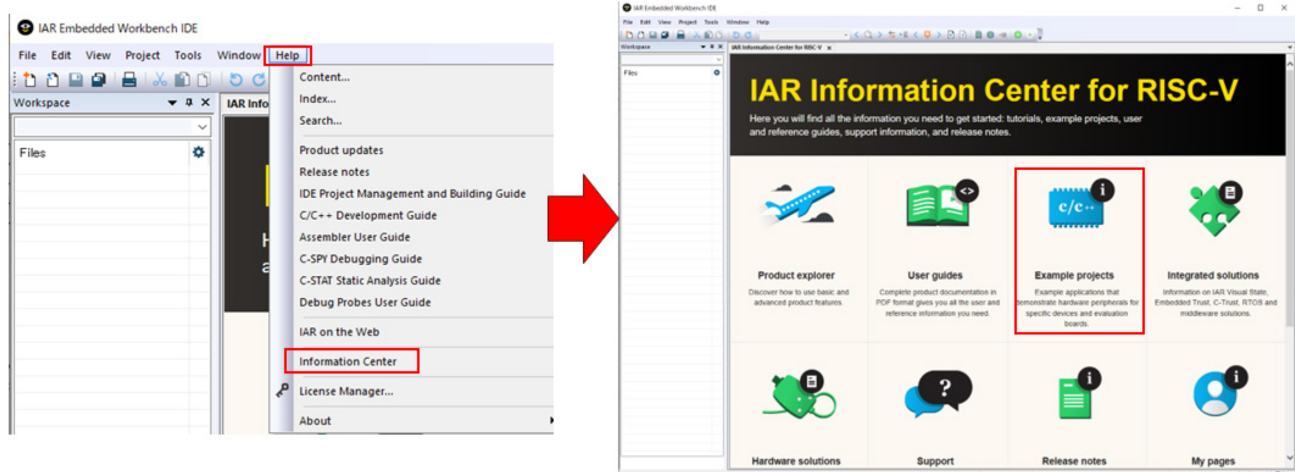
EWRISCVでダウンロードが終了すると、EWRISCVはデバッグモードになります。Goボタンをクリックすると、プログラムの実行を開始します。



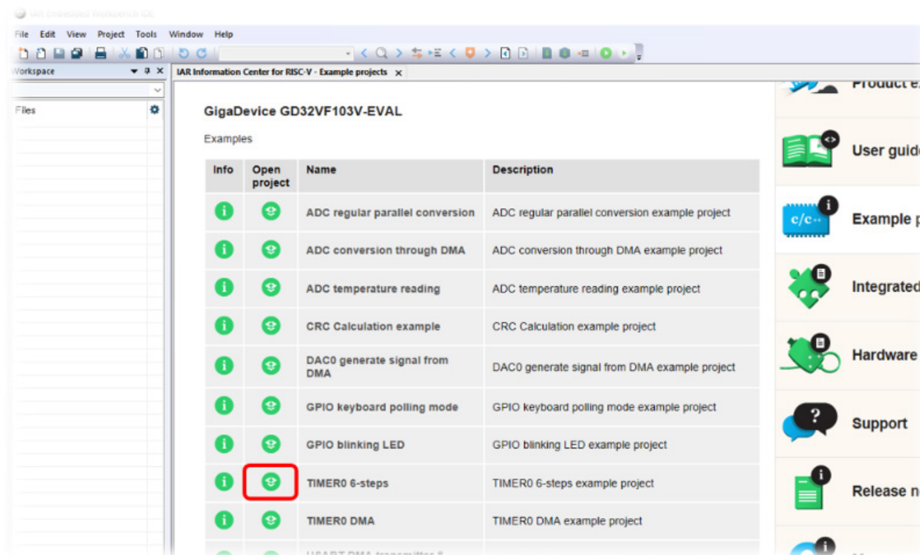
3.1.4　RISC-Vの割り込みを理解する

初めて採用したマイコンの割り込みプログラムを作成するのは、なかなか難しいものです。EWRISCVには、GigaDevice社の評価ボードである「GD32VF103V-EVAL」のサンプルプロジェクトが用意されています。ボードは異なるのですが、このプロジェクトに含まれるサンプルコードを参考にして、Wio Lite RISC-Vボードを動かしてみます。初めて取り扱うマイコンの割り込みプログラムを作る場合、サンプルコードの内容を確認しながら理解するのが早道となります。GD32VF103V-EVALのサンプルプロジェクトは、EWRISCVのインフォメーションセンターに置いてあります。

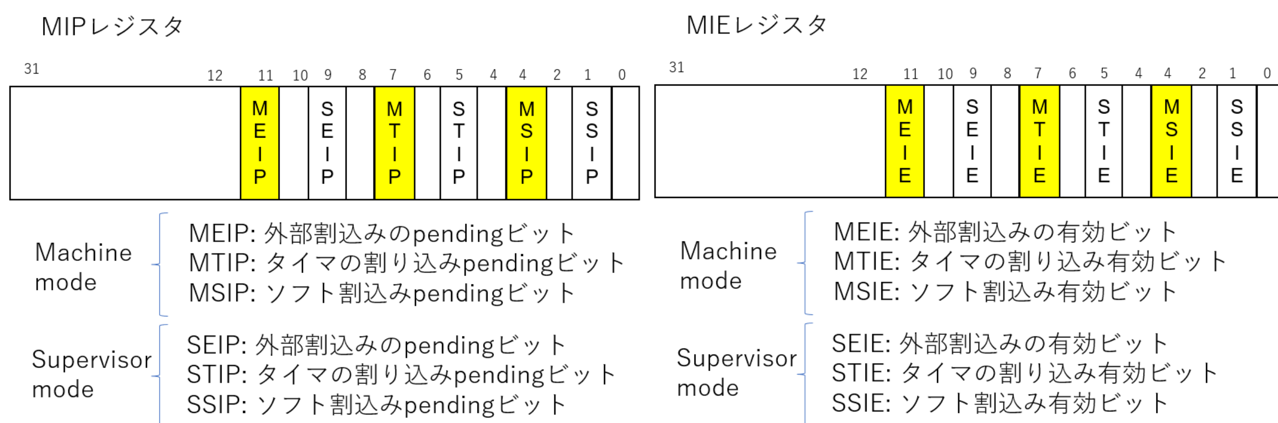
メニューバーの [Help]-[Information Center] を選択してインフォメーションセンターを開き、Example projectsのところをクリックします。



GigaDevice GD32VF103V-EVALを選択し、下図に示すように、TIMER0 6-stepsの左のOpen projectのところをクリックします。



ここで保存フォルダを指定すると、サンプルプロジェクトを開くことが出来ます。
GD32VF103の割り込みは一つではなく、設定によって切り替えて使用できます。以下では、RISC-Vの割り込みについて説明します。RISC-Vでは、CPUコアに近いローカルな内部割り込みのコントローラとして、CLINT（Core Local Interrupt）、CLIC（Core-Local Interrupt Controller）、CLINTの上位互換となるACLINT（Advanced Core Local Interruptor）という仕様も公開されています。外部割り込みついては、PLIC（Platform-Level Interrupt Controller）というコントローラがあります。CLICについては割り込みコントローラについては、参考文献 [7] などを読むと理解が進むと思います。CLINTの仕様では、CLINTは内部割り込みを取り扱い、外部割り込みについてはPLICが取り扱いし、
CSR（Control and Status Register）として用意されるMEIP（Machine External Interrupt Pending）やMEIE（Machine External Interrupt Enable）といったレジスタでまとめて取り扱う、というシンプルな構成になっています。
下図に、割り込みを管理するCSRのMIP（Machine Interrupt Pending）レジスタとMIE（Machine Interrupt Enable）レジスタを示します。



これに対してCLICでは、MIPレジスタやMIEレジスタを使用せず、CLICの仕様で定義された管理用のレジスタを使って割り込みを管理します。割り込みの優先度やトリガタイプ、割り込みごとのベクタの対応などを設定できます。GD32VF103は、ECLIC（Enhanced Core-Local Interrupt Controller）と呼ぶ、CLICをベースとした割り込みコントローラを実装しています。
ここですべての仕様を説明することは出来ませんが、重要な部分をかいつまんで紹介します。詳しく調べたい場合は、サンプルコードを見ながら動作を確認してください。
まず、サンプルコードで使われている関数について説明します。以下の関数は、EWRISCVが用意しているintrinsics関数（組込み関数）と呼ばれているもので、C言語では記述できない機能を実現しています。

- __disable_interrupt()：CPUの割り込みを禁止する
- __set_bits_csr(reg, val)： regで指定したCSRに対して、ビットを val の値でセットする
- __clear_bits_csr(reg,val)：regで指定したCSRに対して、ビットを val の値でクリアする
- __write_csr(reg,val)：regで指定したCSRに対して、値 val を設定する
- __read_csr(reg)：regで指定したCSRを読み出し、内容を return する

次ページのサンプルコードでは、関数 __low_level_init で割り込みに関する基本的な設定を行っています。関数 __low_level_init のコードとその説明を示します。まず、MTVT、MTVT2、MTVECを設定しています。CLICモードで動作させ、割り込み発生時には irq_entry を呼び出し、TRAPやNMI（Non-maskable Interrupt）の発生時には trap_entry を呼び出しています。

```c
int __low_level_init()
{
  __disable_interrupt();
  /* Set the the NMI base to share with mtvec by setting CSR_MMISC_CTL */
  /* li t0, 0x200           */
  /* csrs CSR_MMISC_CTL, t0 */
  __set_bits_csr(/*CSR_MMISC_CTL*/ 0x7D0, 0x200);

  /* Initialize the mtvt */
  /* la t0, vector_base      */
  /* csrw CSR_MTVT, t0       */
  __write_csr(_CSR_MTVT, ((unsigned int)&gd_vector_base));
  /* Initialize the mtvt2 and enable it */
  /* la t0, irq_entry
     csrw CSR_MTVT2, t0
     csrs CSR_MTVT2, 0x1
  */
  __write_csr(/*_CSR_MTVT2*/ 0x7EC, 0x1 | ((unsigned int)&irq_entry));

  /* Initialize the CSR MTVEC for the Trap ane NMI base addr*/
  /* la t0, trap_entry
     csrw CSR_MTVEC, t0
  */
  __write_csr(_CSR_MTVEC, 0x03 | ((unsigned int)&trap_entry));

  /* Enable mycycle_minstret */
  __clear_bits_csr(/*CSR_MCOUNTINHIBIT*/ 0x320, 0x5);
  return 1;
}
```

EWRISC-Vが提供するintrinsics関数で、
CPUの割り込みを禁止する

NMI発生時に割込みを発生させる設定

ベクタモード時に使用するアドレスの設定（このサンプル
ではベクタモードは使用しない）

外部割込みの処理を実施するベクタをMTVT2に登録。
割込み発生時に、irq_entryを呼び出す。
0x1を設定することで割込み時mtvec2のアドレスを使用。
下位1ビットに0x0を設定すると、mtvecを使用する。

TRAP,NMI発生時に実施するベクタをMTVECに登録。
TRAP,NMI発生時に、trap_entryを呼び出す。
0x03でCLICモードに設定している。下位2ビットが0x3以
外の場合にはCLINTモードになる。

次に、割り込みベクタについて説明します。割り込みベクタは配列で作成されており、割り込みハンドラがない部分には 0 が設定されています。下図に、割り込みベクタのコードを示します。配列の中で関数が指定されており、割り込み要因により個別の関数が割り込みハンドラによって呼び出されます。

```c
typedef void(*__fp)();
const __fp gd_vector_base[96] =
{
    0,
    0,
    0,
    eclic_msip_handler,
    0,
    0,
    0,
    eclic_mtip_handler,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    eclic_bwei_handler,
    eclic_pmovi_handler,
    WWDGT_IRQHandler,
    LVD_IRQHandler,
    TAMPER_IRQHandler,
    RTC_IRQHandler,
    FMC_IRQHandler,
    RCU_IRQHandler,
    EXTI0_IRQHandler,
    EXTI1_IRQHandler,
    EXTI2_IRQHandler,
    EXTI3_IRQHandler,
    EXTI4_IRQHandler,
    DMA0_Channel0_IRQHandler,
    DMA0_Channel1_IRQHandler,
    DMA0_Channel2_IRQHandler,
    DMA0_Channel3_IRQHandler,
    DMA0_Channel4_IRQHandler,
    DMA0_Channel5_IRQHandler,
    DMA0_Channel6_IRQHandler,
    ADC0_1_IRQHandler,
    CAN0_TX_IRQHandler,
    CAN0_RX0_IRQHandler,
    CAN0_RX1_IRQHandler,
    CAN0_EWMC_IRQHandler,
```

続いて、割り込みハンドラに進みます。まず、EWRISCVでハンドラを定義する方法を説明します。EWRISCVは、拡張キーワードとして __interrupt を用意しています。以下のように、関数の定義の前に __interrupt を付けることでハンドラになります。ハンドラには引き数がなく、戻り値もありません。そのため、引き数および戻り値は、必ず void になります。

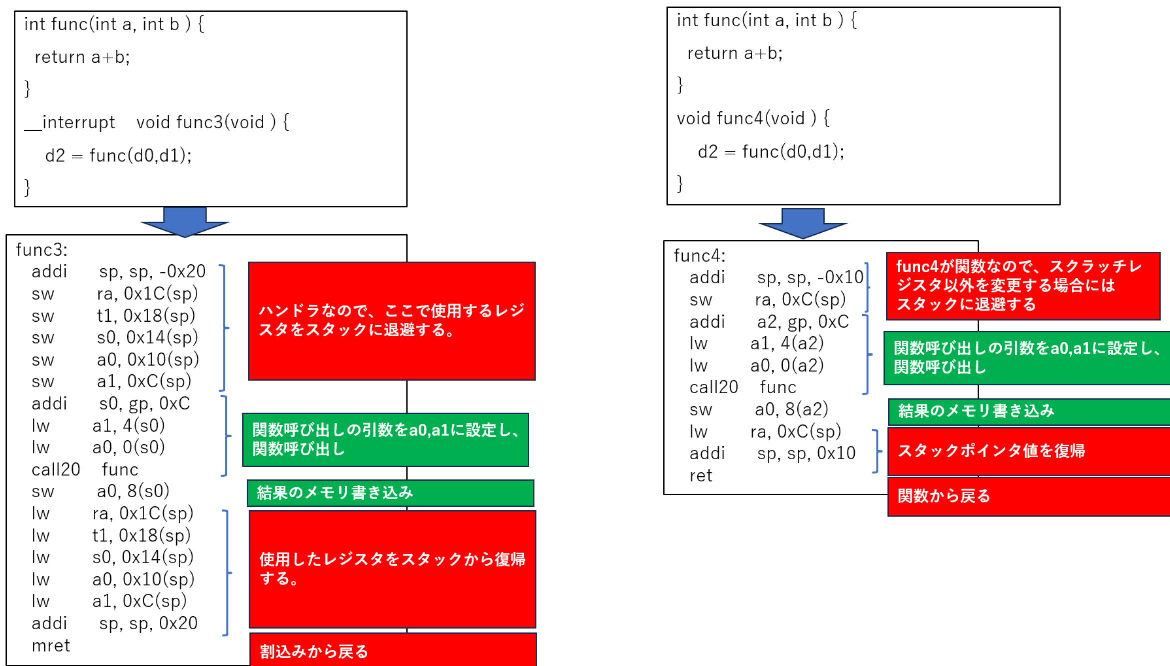
```
__interrupt void my_interrupt_handler(void);
```

それでは、なぜこの __interrupt を使用する必要があるのでしょう？この __interrupt がない場合は、通常の関数になります。2.10.2項（関数呼び出しの時のルール）で説明したように、関数呼び出しの時に使用されるレジスタは、以下のとおりです。
スクラッチレジスタは、関数内で値を自由に書き換えてよいレジスタです。関数として呼び出す場合、関数を呼び出す側がスクラッチレジスタの値をスタックに保存して、関数を呼び出します。これにより、呼び出された関数はスクラッチレジスタの値を自由に書き換えられます。

1. スクラッチレジスタ：t0〜t6、ft0〜ft11、a0〜a7、fa0〜fa7
2. 保存レジスタ：s0〜s11、fs0〜fs11
3. 特定用途レジスタ：sp/x2、gp/x3、ra/x1

ハンドラで使用するレジスタは、割り込み発生時にスタックに値を退避し、ハンドラを終了する時にレジスタに値を戻す必要があります。
そうしないと、ハンドラから戻ったときにレジスタの値が変わり、動作がおかしくなる可能性があるからです。特に問題となるのが、スクラッチレジスタの部分です。

下図左側が __interrupt を付けた割り込みハンドラ、右側が通常の関数です。
左のハンドラの場合、ハンドラ内で使用するレジスタの値をすべて退避しています。右の関数の場合、スタックポインタのみ退避しています。
関数 func そのものの呼び出しは、ハンドラから呼ぶ場合も関数から呼ぶ場合も同じになっていることが分かります（緑色の説明の部分）。関数 func の処理が終わり、結果をメモリに格納した後は、退避したレジスタの復帰処理となります。



このプログラムでは、外部割り込みは irq_entry が、TRAPやNMIは trap_entry がハンドラとなります。そのため、irq_entry と trap_entry については __interrupt が付けられています。irq_entry では、CSRのMCAUSEやMEPC、MSUBMなどを退避して、ハンドラを呼び出しています。その後、退避したレジスタの復帰処理を行い、割り込みから戻ります。

なお、このサンプルコードでは完全には実装されていませんが、記述を見ると、多重割り込みを行えるようにする準備をしていたものと思われます（残念ながら、多重割り込みが動くところまで実装されていない）。

```
uintptr_t handle_trap(uintptr_t mcause, uintptr_t sp)
{
  __fp fp;
  mcause &= 0xFFF;
  fp = gd_vector_base[mcause];
  if (fp)
    fp();
  return 0;
}

__interrupt void trap_entry()
{
  uintptr_t mcause = __read_csr(_CSR_MCAUSE);
  handle_trap(mcause, 0);
}

__interrupt void irq_entry()
{
  uintptr_t mcause = __read_csr(_CSR_MCAUSE);
  uintptr_t mepc = __read_csr(_CSR_MEPC);
  uintptr_t msubm = __read_csr(0x7C4);
  handle_trap(mcause, 0);
  //asm("csrrw ra,0x7ED, ra");
  __disable_interrupt();
  __write_csr(_CSR_MCAUSE, mcause);
  __write_csr(_CSR_MEPC, mepc);
  __write_csr(0x7C4, msubm);

}
```

mcauseに格納された番号で、処理するハンドラを選択し、呼び出す。

EWRISC-Vでは割込みハンドラは__interruptを付ける

CSRのMCAUSEに要因が入っているのでローカル変数に保存

CSRのMEPCに割込みの戻り先が保存、その値をローカル変数に保存

CSRのMSUBMに現在の割り込み状態と、直前の割り込み状態を保存。
これをローカル変数に保存。

割込み要因を引数に、handle_trapを呼び出す。

あとは、CLICの設定を行うだけです。GD32VF103では、ECLICの設定となります。ECLICについて詳しく知りたい場合は、GD32VF103のマニュアルを確認してください。ECLICでは下表に示すレジスタが用意されていますclicintattr（CLIC割り込み属性）では、一つ一つの割り込みに対して、割り込みベクタで受けるかどうかを設定できます。また、レベルエッジなどの割り込みタイプを指定できます。それぞれの割り込みは、clicintie（CLIC割り込み許可）、clicintip（CLIC割り込み保留）というフラグを個別に持っています。そのため、CSRのMIEレジスタやMIPレジスタなどは使用しません。

Offset	Permission	Register	Bit	説明
0x0000	RW	cliccfg	8	Level / priorityビット幅指定
0x0004	R	clicinfo	32	CLICの実装情報
0x000b	RW	mth	8	割り込みスレショルドの定義
0x1000+4*i	RW	clicintip[i]	8	i番目の割り込み保留フラグ
0x1001+4*i	RW	clicintie[i]	8	i番目の割り込み許可フラグ
0x1002+4*i	RW	clicintattr[i]	8	i番目の割り込みタイプ指定・ベクタ設定
0x1003+4*i	RW	clicintctl[i]	8	i番目のLevel / Priority設定

以下に実際のCLICの設定などを実施するmain関数を示します。参考となるサンプルコードを読み込むと、RISC-Vの仕様への理解が深まります。EWRISCVを使って、実際にハードウェアを動かしながらサンプルコードに目を通すことをお勧めします。

```
int main(void)
{
    gpio_config();
    eclic_global_interrupt_enable();
    eclic_set_nlbits(ECLIC_GROUP_LEVEL3_PRIO1);
    eclic_irq_enable(TIMER0_TRG_CMT_IRQn, 6,1);
    timer_config();

    while(1){
        delay_1ms(10);
        timer_event_software_generate(TIMER0,TIMER_EVENT_SRC_CMTG);
    }
}
[EOF]
```

CPUの割り込みを許可する

CLICの割り込みレベルとプライオリティを設定

CLICのTIMER0_TRG_CMT_IRQを有効にし、
その時level＝6，priority=1とする。

ソフトからTIMER0のTIMERイベントを発生させる。

3.1.5　CSRを確認しよう

RISC-Vでは、ハードウェアの動作状況を確認したり、割り込みを制御したりする際に、CSRの理解が重要になります。ここでは、CSRにアクセスする方法を説明します。また、いくつかのCSRのレジスタについては、GD32VF103CBT6で動作を確認してみます。

RISC-Vには演算フラグがない

CSRの説明に入る前に、演算フラグの話をしておきます。
組込みシステムでよく使用されるArm Cortex-Mマイコンでは、以下の五つの演算フラグが用意されています。しかし、RISC-Vにはこのような演算フラグがありません。「演算フラグはない」と理解しておいてください。

- Nフラグ：ネガティブフラグ。演算結果が負の場合に 1 となる
- Zフラグ：ゼロフラグ。演算結果が 0 の場合に 1 となる
- Cフラグ：キャリ/ボローフラグ。演算結果にキャリまたはボローが発生した場合に 1 となる
- Vフラグ：オーバーフローフラグ。オーバーフローが発生した場合に 1 となる
- Qフラグ：飽和フラグ。飽和演算で飽和が発生した場合に 1 となる

Machine-Level CSR

RISC-V命令セットマニュアル第2巻（参考文献 [5]）のCSRの章には、Machine-Level CSRのリストが掲載されています。重要な部分を抜粋して下に示します。

Number	**Name**	**Memo**
Machine Information Registers		
0xF11	mvendorid	Vendor ID
0xF12	marchid	Architecture ID
0xF13	mimpid	Implementation ID
0xF14	mhartid	Hardware thread ID
0xF15	mconfigptr	Pointer to configuration data structure
Machine Trap Setup		
0x300	mstatus	Machine status register
0x301	misa	ISA and extensions
0x302	medeleg	Machine exception delegation register
0x303	mideleg	Machine interrupt delegation register
0x304	mie	Machine interrupt-enable register
0x305	mtvec	Machine trap-handler base address
0x306	mcounteren	Machine counter enable
0x310	mstatush	Additional machine status register(RV32 only)
Machine Trap Handling		
0x340	mscratch	Scratch register for machine trap handlers
0x341	mepc	Machine exception program counter
0x342	mcause	Machine trap cause
0x343	mtval	Machine bad address or instruction
0x344	mip	Machine interrupt pending
0x34A	mtinst	Machine trap instruction (transformed)
0x34B	mtval2	Machine bad guest physical address

マニュアルではこれ以降もCSRの定義が説明されています。ただし、本書ではここまで細かく説明しません。

Number	**Name**	**Memo**
Machine Configuration		
0x30A	menvcfg	Machine environment configuration register
0x31A	menvcfgh	Additional machine env. conf. register(RV32 only)
0x747	mseccfg	Machine security configuration register
0x757	mseccfgh	Additional machine security conf. register(RV32 only)
Machine Memory Protection		
0x3A0	pmpcfg0	Physical memory protection configuration
0x3A1	pmpcfg1	Physical memory protection configuration(RV32 only)
0x3A2	pmpcfg2	Physical memory protection configuration
0x3A3	pmpcfg3	Physical memory protection configuration(RV32 only)
...	...	...
0x3AE	pmpcfg14	Physical memory protection configuration
0x3AF	pmpcfg15	Physical memory protection configuration(RV32 only)
0x3B0	pmpaddr0	Physical memory protection address register
0x3B1	pmpaddr1	Physical memory protection address register
...	...	...
0x3EF	pmpaddr63	Physical memory protection address register
Machine Counter/Timers		
0xB00	mcycle	Machine cycle counter
0xB02	minstret	Machine instructions-retired counter
0xB03	mhpmcounter3	Machine performance-monitoring counter
0xB04	mhpmcounter4	Machine performance-monitoring counter
...	...	...
0xB1F	mhpmcounter31	Machine performance-monitoring counter
0xB80	mcycleh	Upper 32 bits of mcycle(RV32 only)
0xB82	minstreth	Upper 32 bits of minstret(RV32 only)
0xB83	mhpmcounter3h	Upper 32 bits of mhpmcounter3(RV32 only)
0xB84	mhpmcounter4h	Upper 32 bits of mhpmcounter4(RV32 only)
...	...	...
0xB9F	mhpmcounter31h	Upper 32 bits of mhpmcounter31(RV32 only)
Machine Counter Setup		
0x320	mcountinhibit	Machine counter-inhibit register
0x323	mhpmevent3	Machine performance-monitoring event selector
0x324	mhpmevent4	Machine performance-monitoring event selector
...	...	...
0x33F	mhpmevent31	Machine performance-monitoring event selector

Debug/Trace Registers (shared with Debug Mode)		
0x7A0	tselect	Debug/Trace trigger register select
0x7A1	tdata1	First, Debug/Trace trigger data register
0x7A2	tdata2	Second Debug/Trace trigger data register
0x7A3	tdata3	Third, Debug/Trace trigger data register
0x7A8	mcontext	Machine-mode context register
Debug Mode Registers		
0x7B0	dcsr	Debug control and status register
0x7B1	dpc	Debug PC
0x7B2	dscratch0	Debug scratch register 0
0x7B3	dscratch1	Debug scratch register 1

CSRにアクセスするプログラミング

RISC-Vには、CSRにアクセスするための拡張命令が用意されており、RISC-V命令セットマニュアル第1巻（参考文献 [4]）に、Zicsrとして定義されています。
以下に、Zicsrの命令を示します。基本となる命令はCSRRWです。これらの命令は指定されたCSRをrdに出力し、命令により入力がrs1を利用しCSR値を更新します。

- CSRRW rd, csr, rs1;　：Read/Write
- CSRRS rd, csr, rs1;　：Read and Set bit
- CSRRC rd, csr, rs1;　：Read and Clear bit
- CSRRWI rd, csr, imm;　：Read/Write 即値
- CSRRSI rd, csr, imm;　：Read and Set bit 即値
- CSRRCI rd, csr, imm;　：Read and Clear bit 即値

ただし、アセンブリ言語でプログラムを組むことが苦手な方も多いと思います。EWRISCVでは、CSRにアクセスするためのintrinsics関数を用意しています。以下に、その関数を示します。
どの関数も csr を指定し、操作を行います。また、どの関数もCSRを操作する前の値を戻り値として返します。

- __clear_bits_csr(csr, value)　：指定のCSRを value でクリアする
- __set_bits_csr(csr, value)　：指定のCSRを value でセットする
- __read_csr(csr)　：指定のCSRを読み出す
- __write_csr(csr, value)　：指定のCSRを読み出し、かつ value を書き換える

それでは、実際にCSRにアクセスするプログラムを作成し、実行してみます。CSRにアクセスするプログラムを、以下に示します。intrinsics関数を使用してCSRを読み出し、printf を使って標準出力に書き出しています。CSRを指定する部分ですが、EWRISCVが用意している csr.h に定義されているものについては、そこで定義されている define 名を使用しました。csr.h に定義されていないものについては、CSR番号で指定していますが、この部分はコメントアウトしてあります。
これは、実行するとエラーになった部分です。仕様書が改版されたり、実装する・しないがベンダに委ねられていたりすることもあり、このような不具合が発生していると見ています。

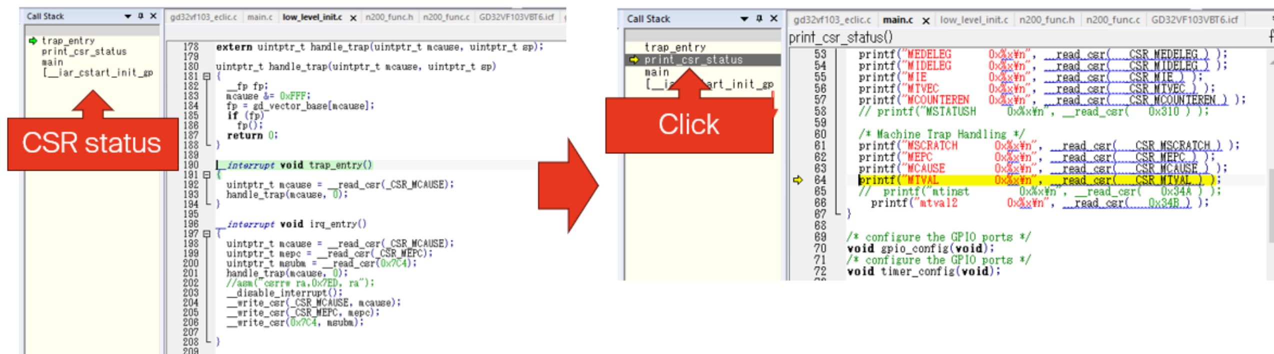
```
#include <intrinsics.h>
void print_csr_status(void ) {
    /* Machine Information Registers */
  printf("MVENDORID     0x%x\n", __read_csr(  _CSR_MVENDORID ) );
  printf("MARCHID       0x%x\n", __read_csr(  _CSR_MARCHID ) );
  printf("MIMPID        0x%x\n", __read_csr(  _CSR_MIMPID ) );
  printf("MHARTID       0x%x\n", __read_csr(  _CSR_MHARTID ) );
  //  printf("mconfigptr   0x%x\n", __read_csr(   0xF15 ) );
```

```
  /* Machine Trap Setup  */
  printf("MSTATUS      0x%x\n", __read_csr(  _CSR_MSTATUS ) );
  printf("MISA         0x%x\n", __read_csr(  _CSR_MISA ) );
  printf("MEDELEG      0x%x\n", __read_csr(  _CSR_MEDELEG ) );
  printf("MIDELEG      0x%x\n", __read_csr(  _CSR_MIDELEG ) );
  printf("MIE          0x%x\n", __read_csr(  _CSR_MIE ) );
  printf("MTVEC        0x%x\n", __read_csr(  _CSR_MTVEC ) );
  printf("MCOUNTEREN   0x%x\n", __read_csr(  _CSR_MCOUNTEREN ) );
  // printf("MSTATUSH     0x%x\n", __read_csr(   0x310 ) );

  /* Machine Trap Handling */
  printf("MSCRATCH      0x%x\n", __read_csr(  _CSR_MSCRATCH ) );
  printf("MEPC          0x%x\n", __read_csr(  _CSR_MEPC ) );
  printf("MCAUSE        0x%x\n", __read_csr(  _CSR_MCAUSE ) );
  printf("MTVAL         0x%x\n", __read_csr(  _CSR_MTVAL ) );
  //  printf("mtinst        0x%x\n", __read_csr(   0x34A ) );
  //  printf("mtval2        0x%x\n", __read_csr(   0x34B ) );
}
```

上記のコードを実行した結果を、以下に示します。

```
VENDORID     0x31e
ARCHID       0x80000022
IMPID        0x100
HARTID       0x0
MSTATUS      0x0
MISA         0x40901105
MEDELEG      0x0
MIDELEG      0x0
MIE          0x0
MTVEC        0x8000b83
MCOUNTEREN   0x0
MSCRATCH      0x0
MEPC          0x80011d4
MCAUSE        0x0
MTVAL         0x34b02573
```

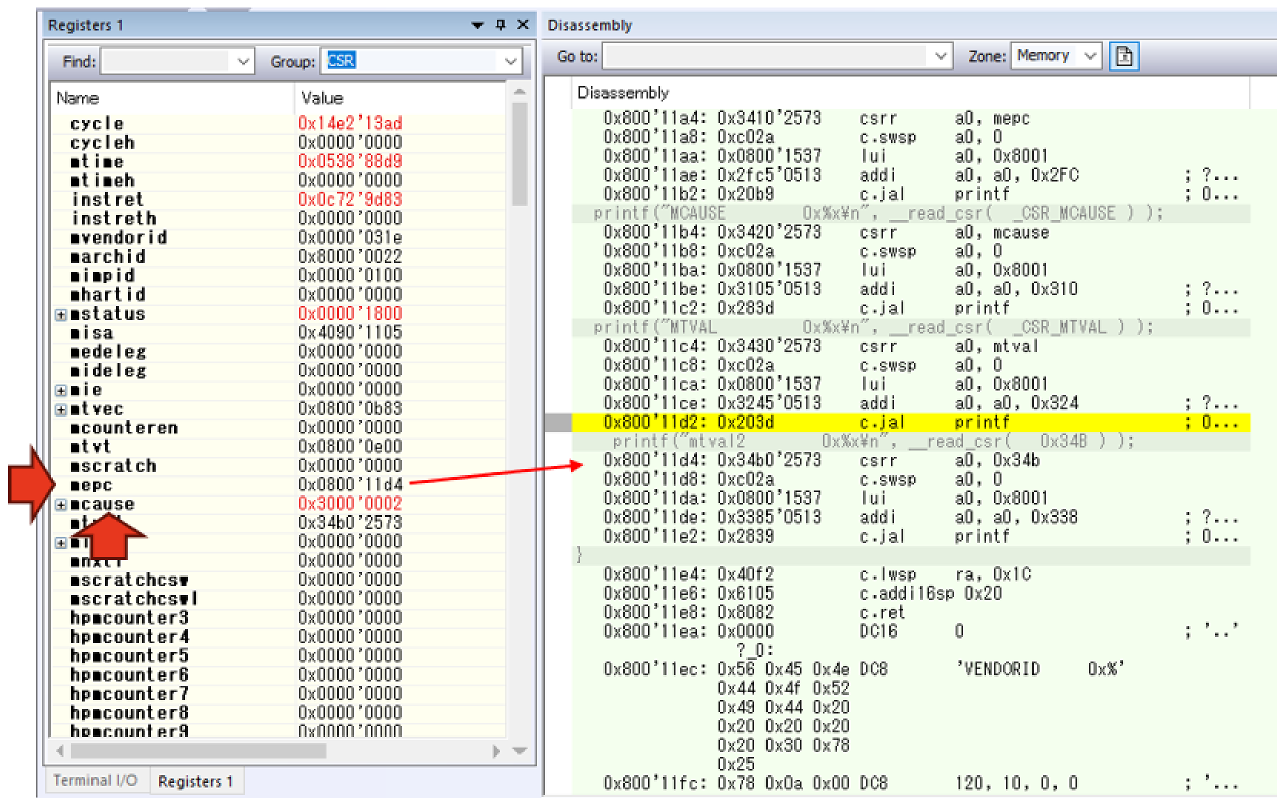
せっかくなので、エラーが出てきた時の状況を調べてみます。
ここでは、CSRの mtval2 にアクセスしてエラーが発生した時の状況を確認します。
エラーが発生し、例外が発生して、trap_entryに遷移しましたが、あいにくと正常系の割り込みベクタしか定義されていないので、trap_entryで無限にループしている状態でした。
EWRISCVでは、Call Stack（コールスタック）画面を利用すると、問題が発生したおおよその場所を確認できます。下図の左側は、ブレークをかけた時の状態です。Call Stack画面に、そこに至るまでの状況が表示されています。main 関数の実行中に、print_csr_status を呼び出し、そこで trap_entry が発生しています。
Call Stack画面に表示されている関数などをクリックすると、問題が発生したおおよその場所を示してくれます。今回のケースでは、print_csr_status をクリックすると、下図の右側の64行目を指示しました。



MEPCとMCAUSEで詳細調べる

さらに詳細な状況を調べるには、CSRの値を確認する必要があります。

EWRISCVでは、Registers画面でCSRの値を確認できます。



mepcはMachine exception program counter（マシン例外プログラムカウンタ）で、割り込み発生時のPC（プログラムカウンタ）の値を記録しています。上図の右側のアセンブリコードを見ると、CSRR a0, 0x34B を発行したところで問題が発生したらしいことが分かります。

mcauseはMachine trap cause（マシントラップ要因）で、この値により問題の原因を特定できます。GD32VF103の場合、mcauseのビット設定は標準仕様から拡張されており、以下のようになっています。

Field	bit	説明
INTERRUPT	31	0: Exception or NMI, 1: Interrupt
MINHV	30	プロセッサが割り込みベクタテーブルを読み出している
MPP	29~28	割り込み前の特権モード
MPIE	27	割り込み前の割り込み許可
MPIL	23~16	前の割り込みレベル
EXCCODE	11~0	例外/割り込みエンコーディン

Registers画面を見ると、mcauseの値は0x30000002です。INTERRUPTビットは0で、発生原因はException（例外）またはTRAPと考えられます。また、MPP=3で、割り込み前はマシンモードです。さらに、EXCCODE（例外要因）=2となっています。
この例外要因の2が何なのかを調べます。GD32VF103マイコンの仕様書では発見できず、RISC-V命令セットマニュアル第2巻（参考文献 [5]）を調べると、下表の情報が見つかりました。
アクセスしてはいけないCSRへのアクセスが発生したため、Illegal instruction（不正な命令）の例外コードを出していると考えられます。

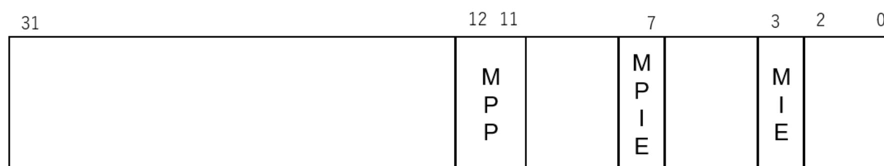
INTEERUPT	Code	説明
0	0	命令アドレスのミスアライン
0	1	命令アクセスの失敗
0	2	不正な命令
0	3	ブレークポイント
0	4	ロードアドレスがミスアライン
0	5	ロードアクセスの失敗
0	6	ストア/アトミックメモリ操作アドレスがミスアライン
0	7	ストア/アトミックメモリ操作アクセスの失敗
0	8	ユーザモードからのecall命令が呼び出し
0	9	スーパーバイザモードらのecall命令が呼び出し
0	10	Reserved
0	11	マシンモードからのecall命令が呼び出し
0	12	命令ページフォルト
0	13	ロードページフォルト
0	14	Reserved
0	15	ストア/アトミックメモリ操作ページフォルト
0	16~23	Reserved
0	24~31	カスタム利用のための指定
0	32~47	Reserved
0	48~63	カスタム利用のための指定
0	≥64	Reserved

- **mstatus**

次に、CSRのmstatusの値に注目します。mstatusはMachine status register（マシン状態レジスタ）で、マシンモードの実行中の状態を示します。GD32VF103マイコンのmstausのビット構成は、下図のようになっています。

MIEは、CPUそのものの割り込みの許可・禁止を設定するビットです（1：許可、0：禁止）。MPIEは、割り込みが入る前のMIEの状態を示すビットです。MPPは、割り込みが入る前の特権モードの状態を2ビットで示します（00：ユーザモード、11：マシンモード）。

mstatus: Machine Status Registers



MIE:　割込み許可ビット
MPIE: MIEビットの前回値を保存
MPP: 前の特権状態

割り込み発生前と割り込み発生時のmstatusの値を下図に示します。マシンモードで動作している場合、MIEが1の時に割り込みを受け付けることが可能です。ここで割り込みが入ると、MPPにマシンモードを示す値（11）が、MPIEに割り込み前のMIEの値（1）が入ります。RISC-Vでは、割り込みが入った段階で、次の割り込みを受けることが出来ないので、MIEの値は0に変わります。割り込みから戻った時は下図の左側の状態に戻ります。ここで注意していただきたいのですが、Arm Cortex-Mマイコンの場合は、割り込みが入った段階で、次の割り込みを受けることが可能です。これまでArm Cortex-Mマイコンを使っていた方は、この点に気をつけてください。

割込み前
mie　1
mip　0
mpp　00

割込み発生時
mie　0
mip　1
mpp　11

エラー発生時の分析作業とは関係ありませんが、実行状況に依存しないCSRの値も見ておきます。

- **mvendorid（ベンダID）を確認する**

CSRのmvendorid（ベンダID）の値は、JEDEC manufacturer IDを示します。読み出した値を見ると、0x0000031Eでした。いろいろ調べたところ、これはRISC-Vコアベンダである台湾Andes Technology社のIDであることが、参考文献 [8]（同社のAX45MP-1CというRISC-Vコア製品のリファレンスマニュアル）から判明しました。しかしこのことについて、筆者は最初、意味がよく分かりませんでした。

GD32Vシリーズのマイコンについて、さらに調べていくと、同じくRISC-Vコアベンダである中国Nuclei System Technology社のWebサイトで、顧客事例の紹介ページ（参考文献 [9] ）を見つけました。このページでは、GigaDevice社のGD32VF103が顧客の製品化の事例として紹介されており、さらにAndes Technolgy社のN22 RISC-Vコアも事例として紹介されていました。つまり、GD32VF103のCPUコアまわりの開発には、協業関係にあるNuclei System Technology社とAndes Technology社が関与している可能性がある、と筆者は推測しています。

- **marchid（アーキテクチャID）を確認する**

CSRのmarchid（アーキテクチャID）の値は、実行するCPUのマイクロアーキテクチャに付与されるIDを示します。下表に、marchidのビット設定を示します。marchidのMSB（最上位ビット）が0の場合はオープンソースプロジェクトです。オープンソースプロジェクトのArchitecture IDは、参考文献 [10] に示されています。一方、marchidのMSBが1の場合は商用プロジェクトで、ベンダがArchitecture IDを指定します。GD32VF103のmarchidの値を確認すると、0x80000022でした。MSBが1で、Architecture IDは0x22になります。先ほどのAndes Technology社のAX45MP-1C RISC-VコアのArchitecture IDが0x8a45となっていたので、0x22は、同社のN22 RISC-Vコアを意味しているのではないか、と筆者は想像しています。

Field	bit	memo
MSB	31	0 : OSS, 1 : Business Project
Architecture ID	30~0	Architecture ID

- **mimpidとmhartidを確認する**

CSRのmimpid（実装ID）は、実装時に付与されるIDを返すようです。GD32VF103のmimpidの値を確認すると、0x00000100でした。Andes Technology社の命名方法を参考にすると、MAJOR=1、MINOR=0、EXTENSION=0というIDが付与されているようです。CSRのmhartid（ハードウェアスレッドID）は、コードが実行されるハードウェアスレッドのIDを示します。最近のCPUの中には、同時に複数のコードを実行するマルチスレッド実行のためのハードウェア機構を備えているものがあります。このようなCPUの場合、mhartidを見ると、コードがどのハードウェアスレッド上で動いているのかが分かると思います。GD32VF103のmhartidの値を確認すると、0x0でした。mhartidの仕様では、「一つのハードウェアスレッドは必ず 0 を返す」とあります。GD32VF103のように一つしか実行スレッドがない場合、0x0を返すことになります。

- **misa（ISAと拡張）を確認する**

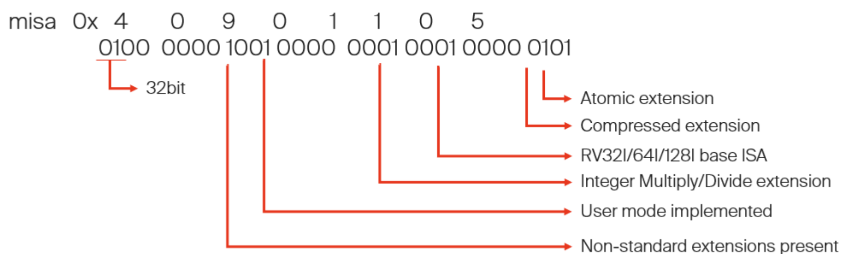
CSRのmisa（ISAと拡張）は、命令セットについての情報を示します。下図のようにビット長（MXL）と命令セット（Extensions）で指定しています。

MXL	XLEN
01	32bit
10	64bit
11	128bit

Extensionsは、下図のように定義されています。

Bit	Name	Description
0	A	Atomic extension（アトミック拡張）
1	B	Tentatively reserved for Bit-Manipulation extension
2	C	Compressed extension(圧縮拡張)
3	D	Double-precision floating-point extension（倍精度浮動小数点演算拡張）
4	E	RV32E base ISA（RV32EベースのISA）
5	F	Single-precision floating-point extension（単精度浮動小数点演算拡張）
6	G	Reserved
7	H	Hypervisor extension（ハイパーバイザ拡張）
8	I	RV32I/64I/128I base ISA（RV32I/64I/128IベースISA）
9	J	Tentatively reserved for Dynamically Translated Languages extension
10	K	Reserved
11	L	Reserved
12	M	Integer Multiply/Divide extension（整数乗算/除算拡張）
13	N	Tentatively reserved for User-Level Interrupts extension
14	O	Reserved
15	P	Tentatively reserved for Packed-SIMD extension
16	Q	Quad-precision floating-point extension（4倍精度浮動小数点演算拡張）
17	R	Reserved
18	S	Supervisor mode implemented（スーパーバイザモード実装）
19	T	Reserved
20	U	User mode implemented（ユーザモード実装）
21	V	Tentatively reserved for Vector extension
22	W	Reserved
23	X	Non-standard extensions present（非標準の拡張が存在）
24	Y	Reserved
25	Z	Reserved

GD32VF103のmisaの値を確認すると、0x40901105でした。この値の意味を、下図に示します。RISC-VのISAの書き方では、これはRV32IMACに対応します。マイコンに実装されている基本命令や拡張命令については、もちろんマイコンのデーターシートやマニュアルにも書いてあるのですが、このような内容がCSRに含まれていると知っていることが、何かの機会に役に立つかもしれません。



3.2 ルネサスのFBP-R9A02G021でRISC-Vを使う

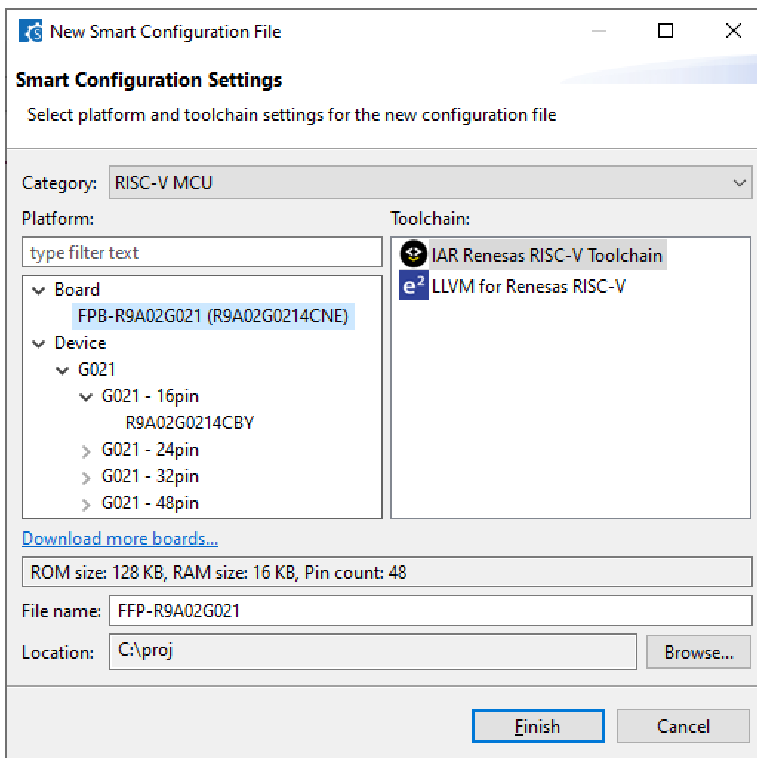
こちらが、ルネサスが汎用RISC-Vマイコンとして最初に出したR9A02G021を搭載した評価ボードFBP-R9A02G021です。

この評価ボード（FBP-R9A02G021）は、オンボード・デバッガも搭載していますが、外部のデバッガを接続するコネクタも搭載されています。
EWRISCVで使用する場合には、外部デバッガ接続コネクタを使いI-jetを接続します。EWRISCVではI-jetを使う方がより多くのデバッグ機能を使う事が可能ですので、まず外部デバッガでI-jetを使用します。本章のおわりに、オンボード・デバッガの使い方も説明します。

3.2.1 サンプルプロジェクトを作成しデバッグする

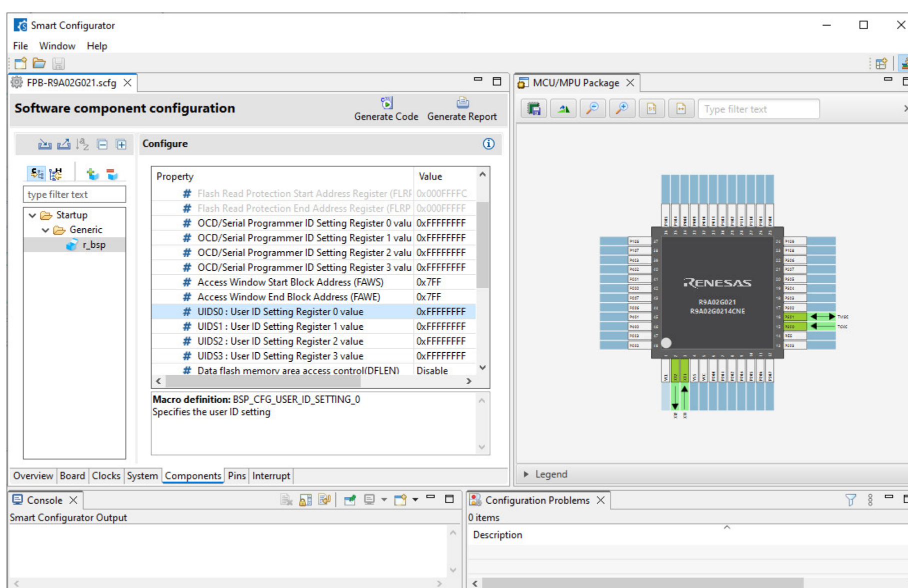
本評価ボードに対してサンプルやBSP（ボードサポートパッケージ）コードを生成するには、ルネサス社のスマート・コンフィグレータを使うことが推奨されます。

EWRISCV向けにFBP-R9A02G021ボードのプロジェクトを作成するためには、スマート・コンフィグレータを起動し、[File]-[New]を選択すると、以下の画面が表示されます。

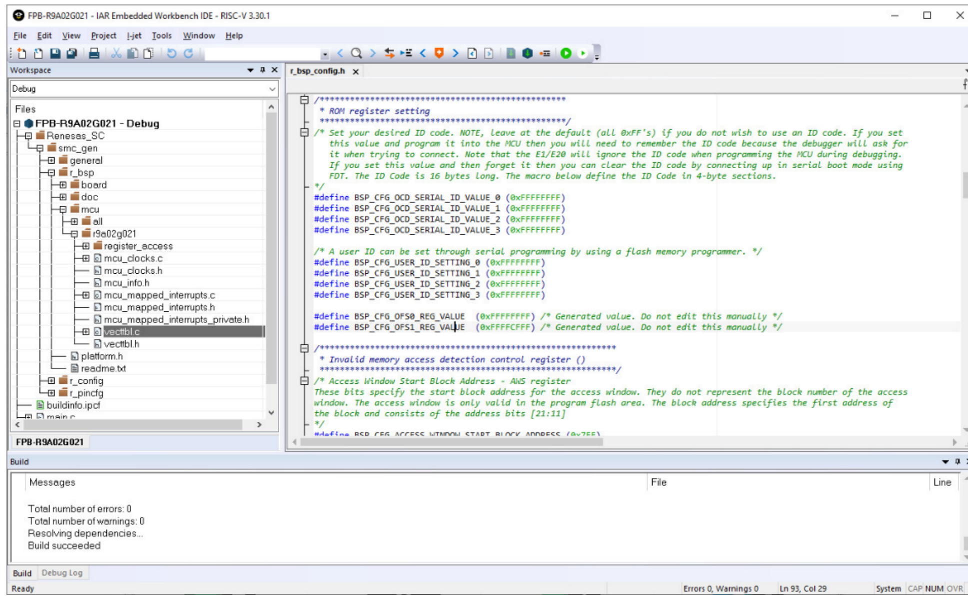


ボード選択でFPB-R9A02G021を選択し、ToolchainでIAR Renesas RISC-V Toolchain を選択します。

その後、クロックやコンポーネントやピンなどの設定をすることで、システム詳細を設定可能です。

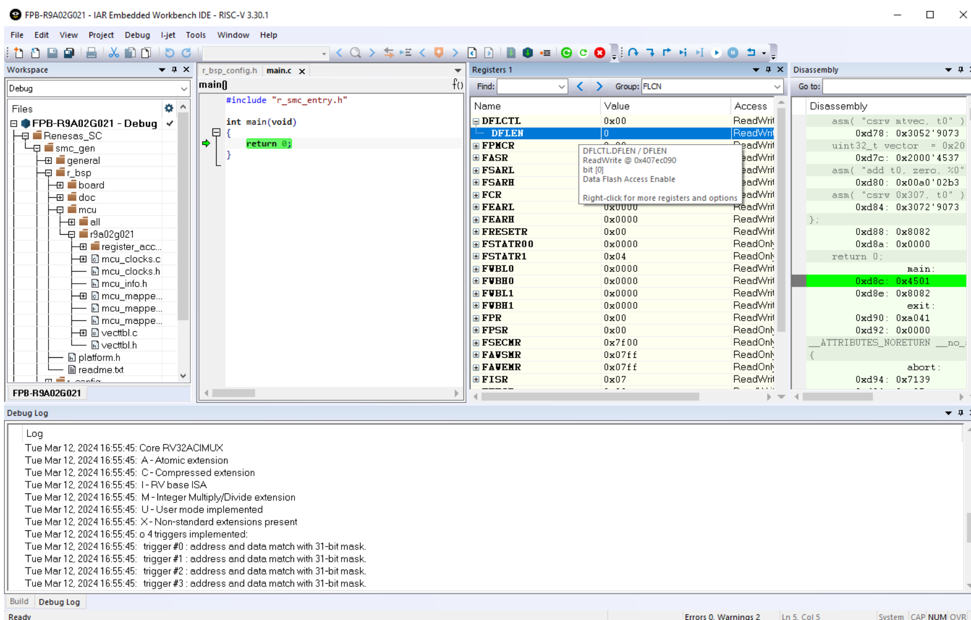


スマート・コンフィグレータで設定が終わったら、コード生成ボタンをクリックします。スマートコンフィギュレータではコードを生成すると、コンソール画面に生成したコードのリンクも出力します。
この例では、EWRISCVが使用するワークスペースファイル（FPB-R9A02G021.eww）が生成されており、EWRISCVでそのファイルを開く事で、EWRISCVが以下のようになります。:生成されたコードを見ると、ヘッダファイルr_bsp_config.hで、UIDS0のUser ID Setting Register値が0xFFFFFFFFがマクロBSP_CFG_USER_ID_SETTING_0として定義されている事が確認できます。



ここまで来たら、[Project]-[Make]を実施します。プロジェクトは通常I-jetをデバッグプローブとして使用するように設定されています。
もし、そうなっていない場合には、[Project]-[Options]-[Debugger]で、DriverをI-jetに変更してください。

設定が終わりましたら、評価ボードにプログラムをダウンロードします。そのためには、[Project]-[Download and debug]を実施すると、デバッグセッションが開始されます。



デバッグ中に、レジスタ画面でCSRの値を観測することが出来ます。たとえば、フラッシュメモリに関するFLCNレジスタグループを開くと、Data flash memory area access control register（DFLCTL）のDFLENを見ることが出来ます。もし、この値がゼロであればデータフラッシュメモリは無効な状態ですし（リセット状態）、有効にするためにはプログラムで設定が必要です。下図の左のコードではヘッダファイルplatform.hをインクルードすることで、R_FLCN->DFLCTL_b.DFLEN = 1 としての設定を実施しました。
実際にレジスタ定義は、生成されたBSPのiodefine.hに記載されており、フォルダは以下になります。

```
r_bsp\mcu\r9a02g021\register_access
```

また、Debug画面では、ログを確認することが出来ます。本ボードでは命令セットがRV32ACIMUであると表示されます。これは、Atomic（アトミック命令）, Compressed（圧縮命令）, Integer（整数演算）, Multiplication（整数乗除算）, and User Mode extensions（ユーザ割込み）の拡張に対応しています。

main.c

main()

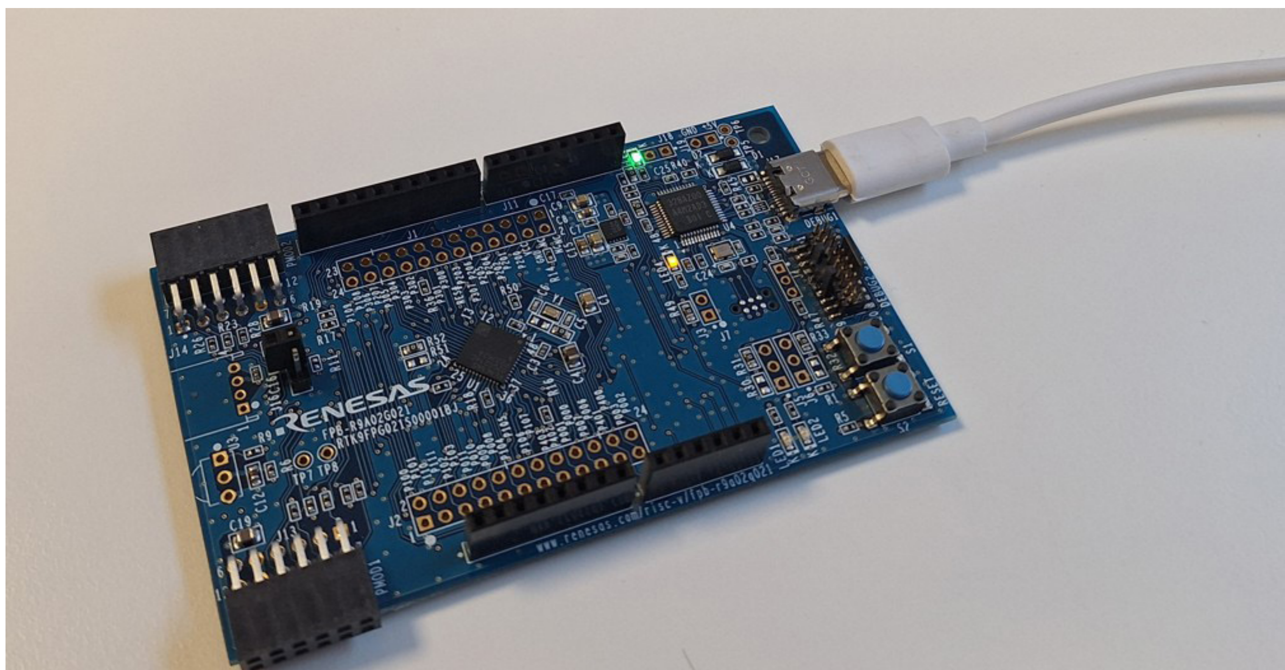
```c
#include "r_smc_entry.h"
#include "platform.h"

int main(void)
{
    R_FLCN->DFLCTL_b.DFLEN = 1;
    return 0;
}
```

Registers 1

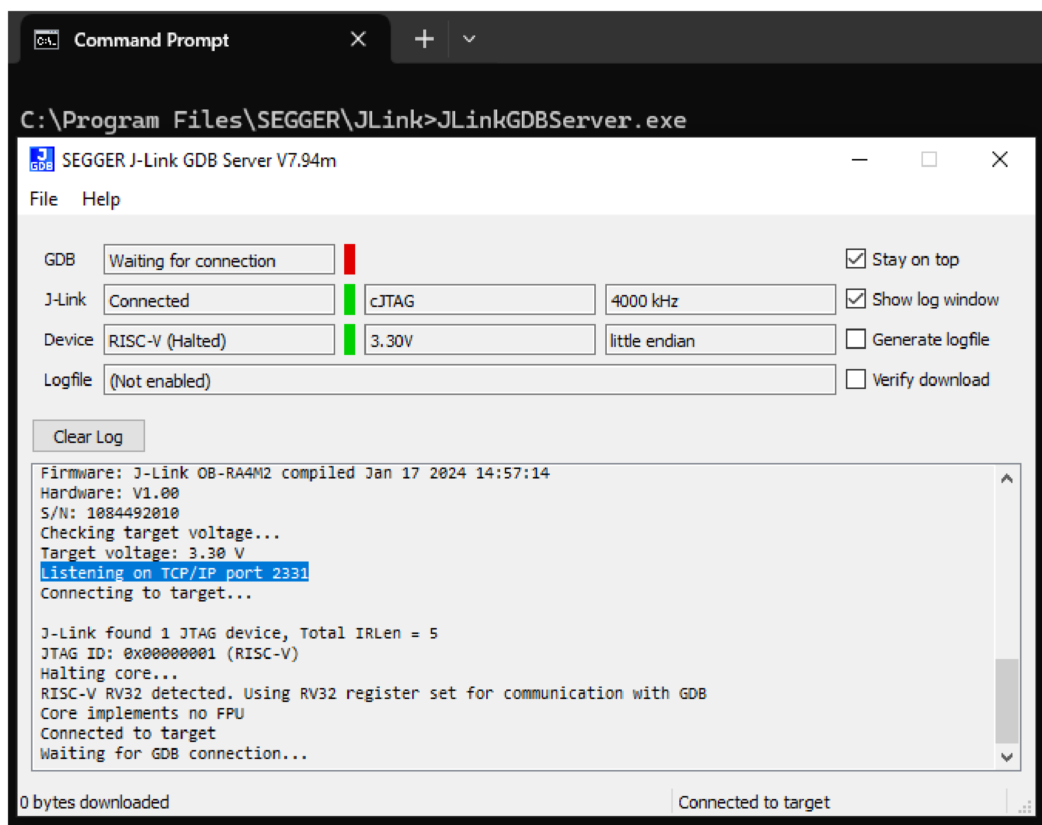
Find: Group: FLCN

Name	Value
DFLCTL	0x01
DFLEN	1
FPMCR	0x08
FASR	0x00
FSARL	0x0000
FSARH	0x0000

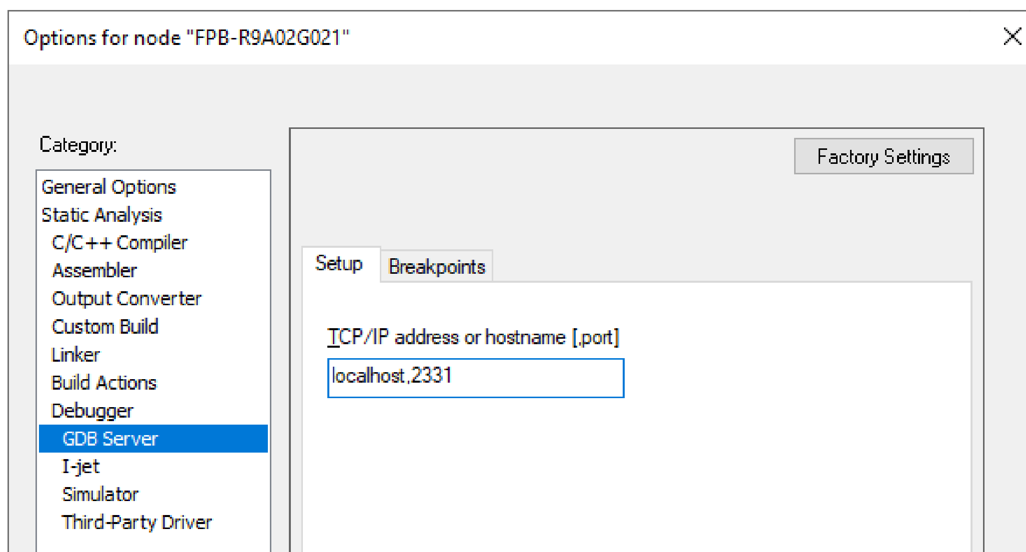
このFPB- R9A02G021 boardという評価ボードでは、オンボードデバッガとしてJ-Linkが搭載されています。これを使ってダウンロードやデバッグをするには、USB-Cポートにケーブルを接続するだけです（I-jet接続は不要となります）。



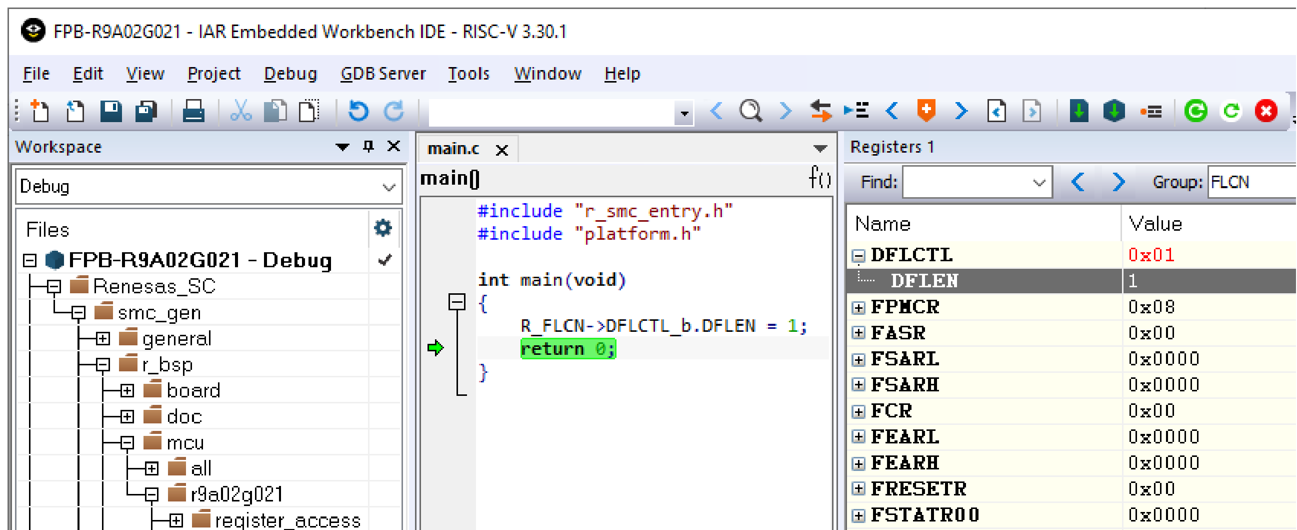
そして、SEGGER社のJLinkGDBServer.exeをスタートして、RISC-Vコアと接続をさせます。



この段階では JLinkGDBServer.exeはポート2331でTCP/IP通信によるデバッガ接続を待っています。EWRISCVではこのオンボードJ-LINKと接続するためには、[Project]-[Options]-[Debugger]を選択し、GDB Serverを選択します。Setupタブを選択し、TCP/IPアドレスをlocalhost,2331と以下のように設定します。



設定が終わったら、デバッグセッションを開始できます。I-jetで観測していたレジスタが確認できます。



注意点をあげておきます。EWRISCVでGDB Serverを使用した場合、性能面と機能面で制限があります。たとえば、GDB Serverの実装ではCSRを観測することが出来ません。今後のリリースで改善され鵜可能性がありますが、IARとしてはそうした制限の無いI-jetを使用する事を推奨します。

4. RTOS、ワークフローの自動化、コード品質について

4. RTOS、ワークフローの自動化、コード品質について

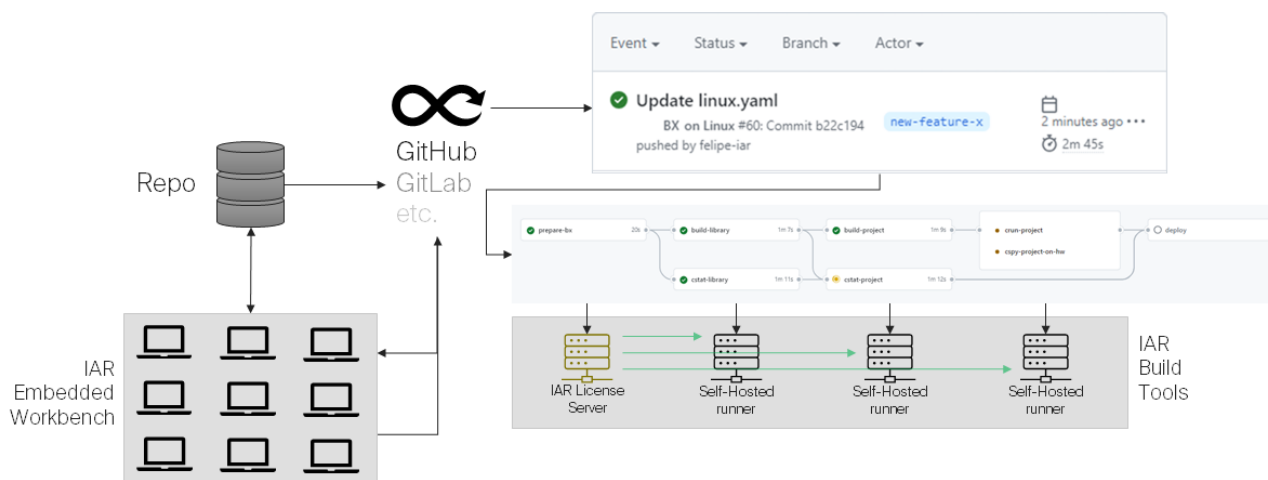
RISC-Vアーキテクチャとそのエコシステムの世界に飛び込む（実際に活用）時、効果的な学習と開発には適切なツールを理解しさらに活用することが重要です。その中でも、特にソフトウェア・プロジェクトが複雑になるにつれて、リアルタイム・オペレーティング・システム (RTOS) が重要な役割を果たします。
Azure RTOS (ThreadX) や FreeRTOS などの RTOS は、IAR Embedded Workbench for RISC-V (EWRISCV) のサンプルがあり即座に利用できます。RTOSっはリアルタイム・アプリケーションに不可欠な構造化されたリソース管理、タスク管理、タイミング管理に関する機能を提供します。さらに、SAFERTOS は、最大限の安全性と信頼性を必要とするアプリケーション向けに、認定された決RTOS ソリューションを提供します。

本書では、RTOSの詳細やステップバイステップの使い方などについては詳しく説明しません。
情報が必要な場合には提供されているすぐに使用できるサンプルを試し、RTOS ベンダーの Web サイトまたは GitHubを探すことをお勧めします。
さらに、最新のワークフロー、自動化、継続的インテグレーション/継続的デプロイメント (CI/CD) パイプラインに関連する一般的な開発課題に対処するために、IARはRISC-V向けの開発ツールとしての機能を向上させています。具体的なイメージを下図に示します。

効率的なソフトウェアの構築とテストのために、IAR C/C++ コンパイラー、アセンブラー、リンカー、IARBuild を含む包括的なツールをサポートしています。こうしたツールにより大規模な重要なソフトウェアの効率的な構築とテストを促進し、導入による信頼性とパフォーマンスを確保します。

さまざまな開発環境に対する適用性とパフォーマンスのために、組織のさまざまなニーズに適応するように設計されており、少数のライセンスで小規模なサーバーに導入することも、数百の並列ビルドをサポートするように拡張することもできます。
IARの開発環境は、規模を問わず高いパフォーマンスを保証します。

最新の開発ワークフローとの統合のため、最新のソフトウェア開発実践を念頭に置いて構築された IAR Build Tools は、CI/CD パイプラインにシームレスに統合することが可能です。
仮想マシン、Dockerコンテナ、およびGitHUBのセルフ・ホスティッド・ランナーをサポートします。
この互換性により、開発者は最新のソフトウェア開発にとって重要な、効率的なCI/CDプロセスを維持できるようになります。



最後に、RTOS と自動化されたワークフローに加えて、コードの品質を確保し、コードの再利用を促進することが、ビジネスの継続とソフトウェアプロジェクトの効率化にとって極めて重要です。
IAR は、IAR Embedded Workbench に完全に統合された強力な静的解析ツール C-STAT を使用して、ここコード品質・再利用性に対処します。
C-STAT はソース コード レベルで徹底的な分析を実行し、開発プロセスの初期段階で潜在的な問題を特定します。C-STATを利用した静的解析ツールを使用するプロアクティブなアプローチは、開発者がMISRA、CWE、CERT C/C++セキュア・コーディングなどの業界標準のコーディング標準に準拠するのに役立ちます。

5. おわりに

5. おわりに

本書は、RISC-V エコシステムを使い、複雑な組み込みソフトウェアを開発する開発者や専門家のための包括的なガイドとして役立つと考え出版をしました。
RISC-V の機能と可能性について説明をし、IAR Embedded Workbench for RISC-V (EWRISCV) が提供する様々な機能についても説明をしました。
これは、CPUの命令セット、スタックの動作についての基本理解が出来るように説明し、複雑なソフトウェアプロジェクトを管理するためのRTOSについては多くのサポートがある事を説明しました。

また、今後のソフトウェア開発では開発ワークフローを強化し、コードの品質を確保し、コードの再利用が重要となります。その時に、IARではWindowsで使う統合開発環境だけでなく、Linuxで使用できる用ビルドツール、さらには、C-STATといった静的解析ツールも用意しています。こうした点も今後重要性となります。
これらのソリューションは、これまで重要とされた効率的なソフトウェアの構築とテストだけでなはく、今後必要とされる業界標準のコーディング慣行への準拠まで、幅広い機能を提供します。
それによってセキュリティ・リスクやコーディング・エラーを軽減します。さらにIARでは第三者認証取得済みコンパイラの販売もしているので、機能安全に関するシステムの開発には有効となります。

このガイドでは、CI/CD パイプラインを含む最新のソフトウェア開発実践に焦点を当てましたが、RISC-Vのハードウェアをどのように使えばよいソフトウェアが出来るのか？についての実践的な知見についても触れました。このガイドをもとにぜひ実際に手を動かしてさらなるRISC-Vの学習・実験・調査をしていただけると思います。

本書で紹介した技術的内容とサンプル、および IAR Embedded Workbench の強力な機能を活用することで、RISC-Vの開発には十分な準備が整いました。これにより直近のプロジェクトの効率性と信頼性が確保されるだけでなく、将来の取り組みに対する保守性と安全性も確保されます。
これを機会にRISC-Vの世界に一歩踏み出して頂ければと思います。

References

1. Computer Architecture: A Quantitative Approach to Design, Implementation, and Evaluation, David A.Patterson, John L.Hennessy
2. IAR C/C++ Development Guide, Linking using ILINK https://wwwfiles.iar.com/riscv/EWRISCV_DevelopmentGuide.ENU.pdf
3. Top Ten Fallacies About RISC-V, David Patterson https://riscv.org/blog/2023/03/top-ten-fallacies-about-risc-v/
4. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA
5. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Document Version 20211203
6. https://hsandid.github.io/posts/risc-v-custom-instruction/
7. https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc
8. AndesCore AX45MP-1C Processor Reference Manual, https://www.andestech.com/wp-content/uploads/AX45MP-1C-Rev.-5.0.0-Datasheet.pdf
9. Nucleisys Customer Cases, https://www.nucleisys.com/product/rvipes/lcxp/
10. Open-Source RISC-V Architecture IDs, https://github.com/riscv/riscv-isa-manual/blob/latex/marchid.md