

FPGAとRISC-Vプロセッサを活用して アクセラレータを持つ組み込みシステム を開発しよう

吉瀬 謙二 (Kise Kenji)

東京科学大学 情報理工学院 教授

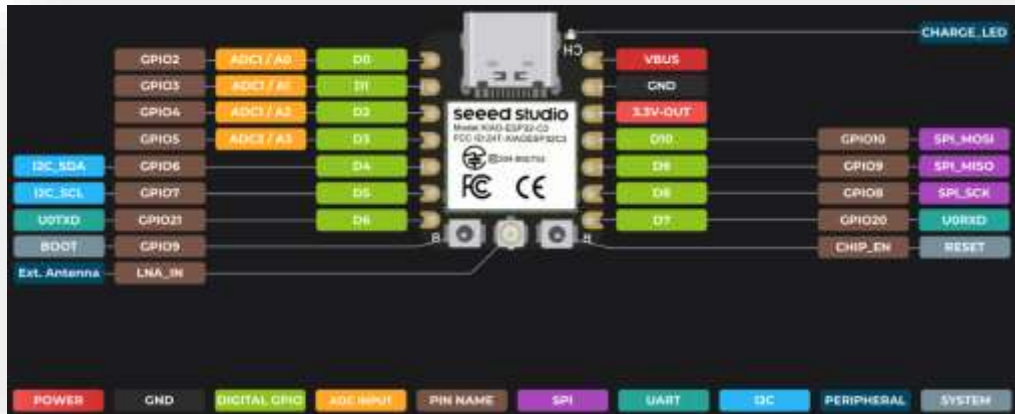
アダプティブコンピューティング研究推進体 代表



- 特定のアプリケーションに適した命令セットアーキテクチャのプロセッサを利用することで、**組み込みシステムの効率を改善**できます。このような、ASIP (Application-Specific Instruction-set Processor) を用いるシステム開発では、**独自の命令セットをサポートするプロセッサ設計**とそのプロセッサを活用する**ソフトウェア設計**という**複雑な協調設計**が必要になり、その開発時間の長期化が課題になっています。
- この課題の解決を目指して、**CFU Proving Ground**と呼ばれる、**FPGAをターゲットするハードウェア／ソフトウェアの協調設計の環境**を提供しています。
- この環境では、**CFU (Custom Function Unit)** と呼ばれる**アクセラレータを搭載するRISC-Vソフトプロセッサ**を用いて**組み込みシステムの効率化**を狙います。
- また、**軽量**なライブラリ、**明確**なファイル依存関係、**柔軟**なメモリマップを提供することでユーザービリティの向上を狙います。
- 本セッションでは、CFU Proving Groundの構成と使い方、ACRiルームでの動かし方などを説明します。

RISC-Vマイコンボードの例

- Seeed Studio **XIAO ESP32C3**
 - RISC-V 32bit CPU (RV32IMC)
 - 4段のパイプライン処理
160MHzの動作周波数
 - 400KB RAM
 - **1,080円**
 - Arduino IDE



秋月電子通商

キーワードから探す

お知らせ 振込先 新商品一覧 全ジャンル一覧 詳細検索 一括注文

カタログダウンロード ご質問・お問合せ 注文履歴 ログイン・会員登録 買い物かご

ホーム > 開発ツール・ボード > マイコンボード > マイコンボード本体 > Seeed Studio XIAO ESP32C3

[117454]Seeed Studio XIAO ESP32C3

在庫グレード: AA

RoHS2

販売コード: 117454
型番: 113991054
発売日: 2022/08/30
メーカー: Seeed Studio(シードスタジオ)

よくある質問(Q&A)
商品選定・製作の参考にしてください。

1個 [117454]

1個 ¥1,080 (税込) 税抜 ¥982

在庫あり

購入数量: 1 個

かごに入れる

お気に入り追加

通販購入可能数: 792個

店舗在庫情報

Seeed Studio XIAO ESP32C3は、親指サイズの小型設計ながらWi-FiとBluetooth 5.0 (BLE)に対応した、コストパフォーマンスに優れたマイコンボードです。

主な特徴

- RISC-V 32bit CPU; 省電力かつパワフルなESP32-C3チップを搭載。
- 超小型設計: 21 x 17.5mmのサイズで、ウェアラブルやIoT機器に最適。
- 豊富な機能: リチウム電池の充電管理機能を内蔵し、ディープスリープ時の消費電力は44μAと極めて低いです。
- 開発のしやすさ: ArduinoやMicroPythonに対応。表面実装可能な切欠き端子を採用しています。

Seeed Studio XIAO ESP32C3は低価格でワイヤレス機能を実現したい電子工作や産業用IoTに最適です。

FPGAボードの例

- Digilent **Cmod A7-35T** (AMD Artix-7 35T FPGA)
 - 17,900円
- Tang Nano 9K (Gowin GW1NR-9 FPGA)
 - 3,780円

秋月電子通商

キーワードから探す

お知らせ 商品一覧 新商品一覧 全ジャンル一覧 詳細検索 一括注文

ホーム > 開発ツール > ボード > FPGA・CPLD・SPLDボード > FPGA・CPLD・SPLDボード本体 > Tang Nano 9K

[117448]Tang Nano 9K

在庫グレード: **AAA**
FPGA・CPLD・SPLDボード本体: **新品未開封**

販売コード: 117448
型番: Tang Nano 9K
発売日: 2022/08/18
メーカー: [Sipeed](#)

[よくある質問\(Q&A\)](#)
商品選定・製作の参考にしてください。

1個 [117448]

1個 ¥3,780 (税込) 税別 ¥3,437

在庫あり

購入数量: 個

[かごに入れる](#)

[お気に入り追加](#)

通販購入可能数: 163個

[店舗在庫情報](#)

Tang nano 9Kは、Gowin GW1NR-9 FPGAチップをベースにした開発ボードです。HDMIコネクタ、RGBスクリーンインターフェイスコネクタ、SPIスクリーンコネクタ、32Mbit SPIフラッシュ、6個のLEDを装備しているため、FPGA検証、risc-vソフトコア検証、基本機能検証に簡単かつ迅速に使用できます。8640のLUT4ロジックユニットは、さまざまな複雑なロジック回路の設計に使用できるだけでなく、完全なPicoRVソフトコアの実行にも使用できます。また、FPGAの学習、ソフトコアの検証、さらなる設計など、ユーザーのさまざまなニーズにも対応可能です。

秋月電子通商

キーワードから探す

お知らせ 商品一覧 新商品一覧 全ジャンル一覧 詳細検索 一括注文

ホーム > 開発ツール > ボード > FPGA・CPLD・SPLDボード > FPGA・CPLD・SPLDボード本体 > Artix-7 35T FPGAモジュール Cmod A7-35T

[111552]Artix-7 35T FPGAモジュール Cmod A7-35T

在庫グレード: **AAA**

販売コード: 111552
型番: 410-328-35
発売日: 2017/01/31
メーカー: [National Instruments Corporation\(Digilent Inc.\)](#)

[よくある質問\(Q&A\)](#)
商品選定・製作の参考にしてください。

1個 [111552]

1個 ¥17,900 (税込) 税別 ¥16,273

在庫あり

購入数量: 個

[かごに入れる](#)

[お気に入り追加](#)

通販購入可能数: 11個

[店舗在庫情報](#)

Artix-7 35Tを使いやすいDIPパッケージにしたモジュールです。USB-マイクロBを使ってPCから直接FPGAの書き換えができます。

■搭載FPGAの主な仕様
Cmod : A7-35T
FPGA型番: XC7A35T-1CPG236C
1MSPS ADC : 有り
プログラムROM : Quad-SPI Flash/JTAG
LUT数 : 20,800
FF数 : 41,600
ブロックRAM : 225KB

マイコンボードとFPGAボードでマンデルブロー集合を描画した例

- 右はマイコンボード（160MHz）でのソフトウェア処理による描画（1フレームの描画に約33秒）
- 中央はFPGAボード（160MHz）にプロセッサとアクセラレータを実装（1フレームの描画に約1.43秒、**マイコンの23倍**）
- 左は、FPGA版のアクセラレータを12個に増やして（150MHz）描画（1フレームの描画に約0.152秒、**マイコンの217倍**）



右はマイコンボード XIAO ESP32C3

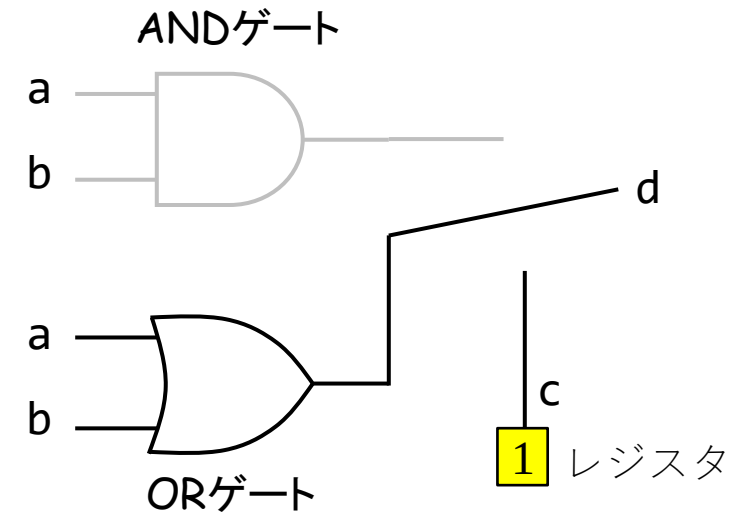
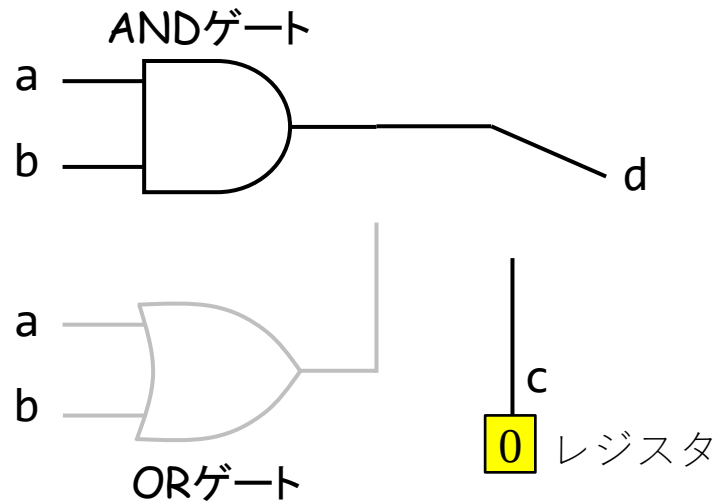
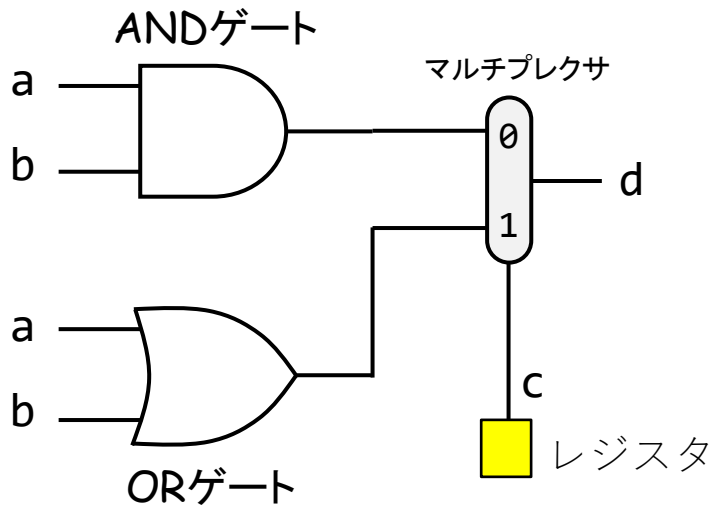
中央、左は Cmod A7

<https://www.youtube.com/watch?v=19PorSnw1c4>

やわらかいハードウェアという特徴を持つ**FPGA**に**RISC-V**プロセッサとアクセラレータを実装して、アプリケーションに特化する高性能**SoC**を簡単に構築できるフレームワークを紹介します。
実用的な**FPGA**開発の最初の一步に最適です。

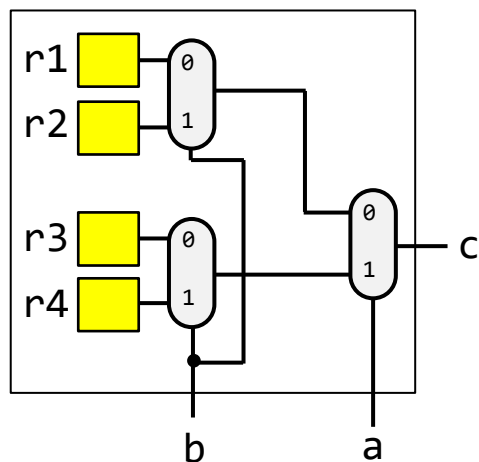
制御信号cによってANDゲートやORゲートになる回路

- ANDゲートとORゲートを切り替えることで性能が向上する例
- 黄色いレジスタの値で出力を選択する回路
- この例は、「やわらかいハードウェア」ではない！

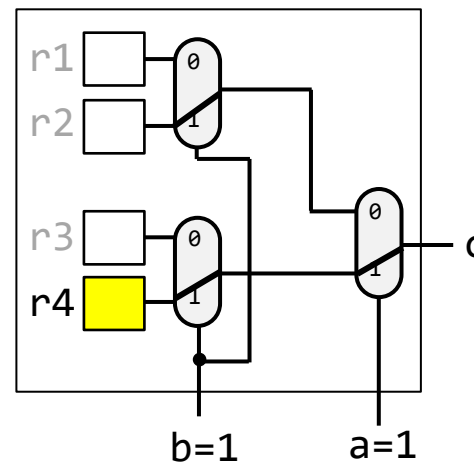
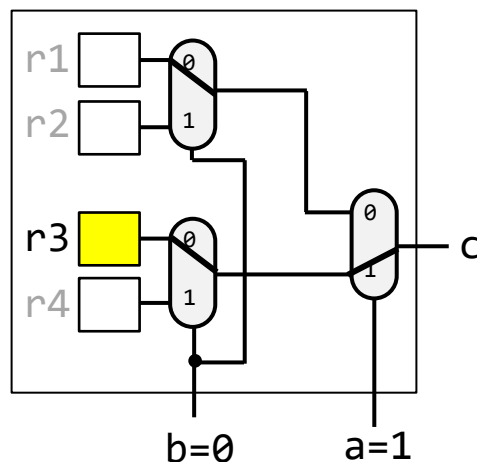
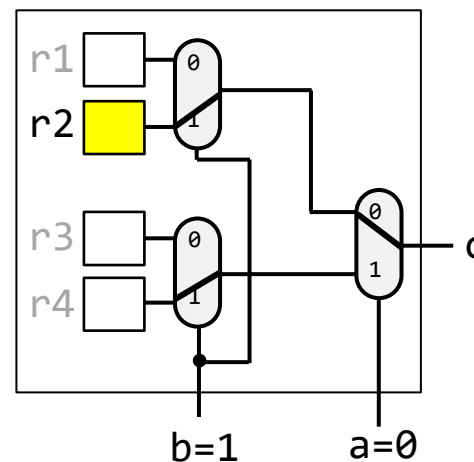
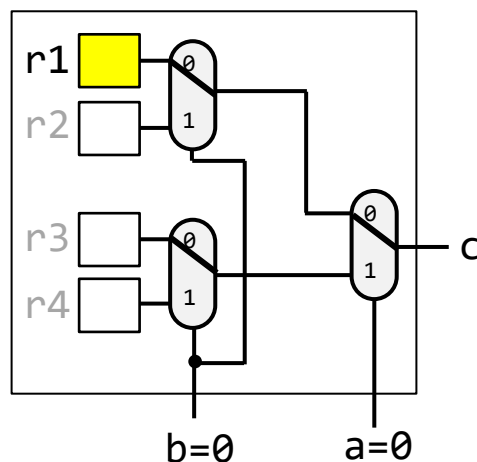


FPGAのルックアップテーブル (Lookup Table, LUT)

- a, b を入力として、c を出力とする LUT
- 値を保持する黄色いレジスタの値を選択する回路



2入力のLUTの構成



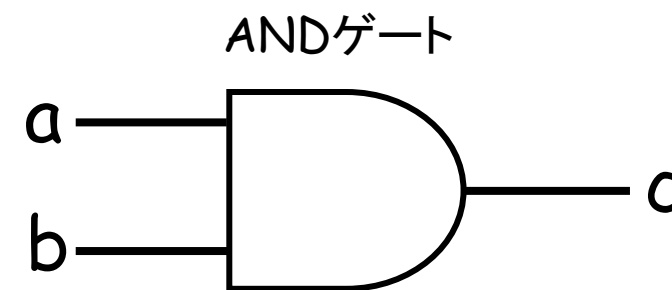
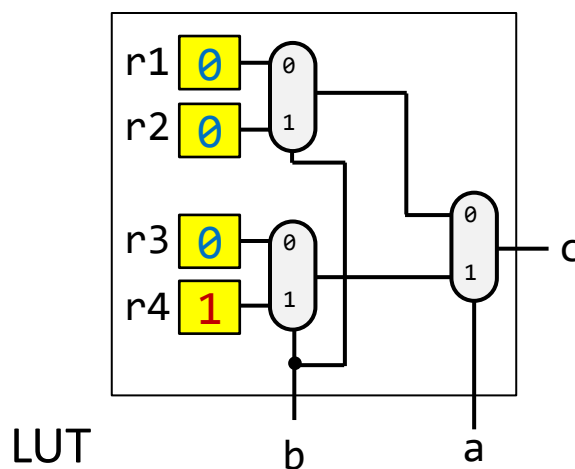
入力		出力
a	b	c
0	0	r1
0	1	r2
1	0	r3
1	1	r4

FPGAのルックアップテーブル (Lookup Table, LUT)

- 真理値表の黄色い部分の値に従ってLUTの黄色いレジスタの値を設定することで、任意の2入力で1出力の組み合わせ回路を実現できる。
- デバイスの電源を入れてから、黄色いレジスタの値を設定する作業をコンフィギュレーションと呼ぶ。これにより、そのLUTは「ユーザの必要とする回路」になる。
- 電源を入れてから、「コンフィギュレーションする内容」を変えると、違うハードウェアとして動く。コンフィギュレーションの時間は数秒から数十秒程度なので、あるハードウェアAが、すぐに、異なるハードウェアBになることができる。
- 「やわらかい」ハードウェア

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

真理値表



左の構成のLUTと等価な回路

Configuration Logic Block (CLB) of 7 Series FPGA

SLICEM

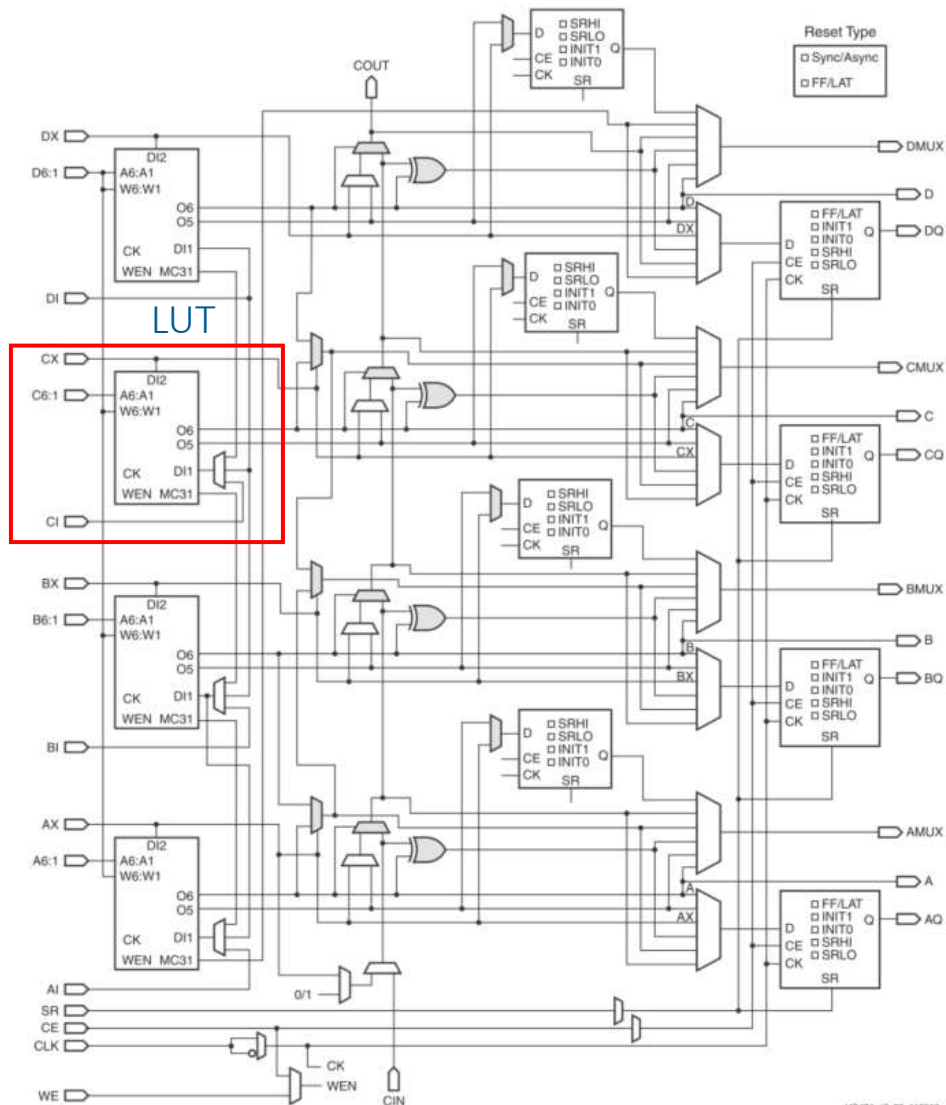


Figure 2-3: Diagram of SLICEM

SLICEL

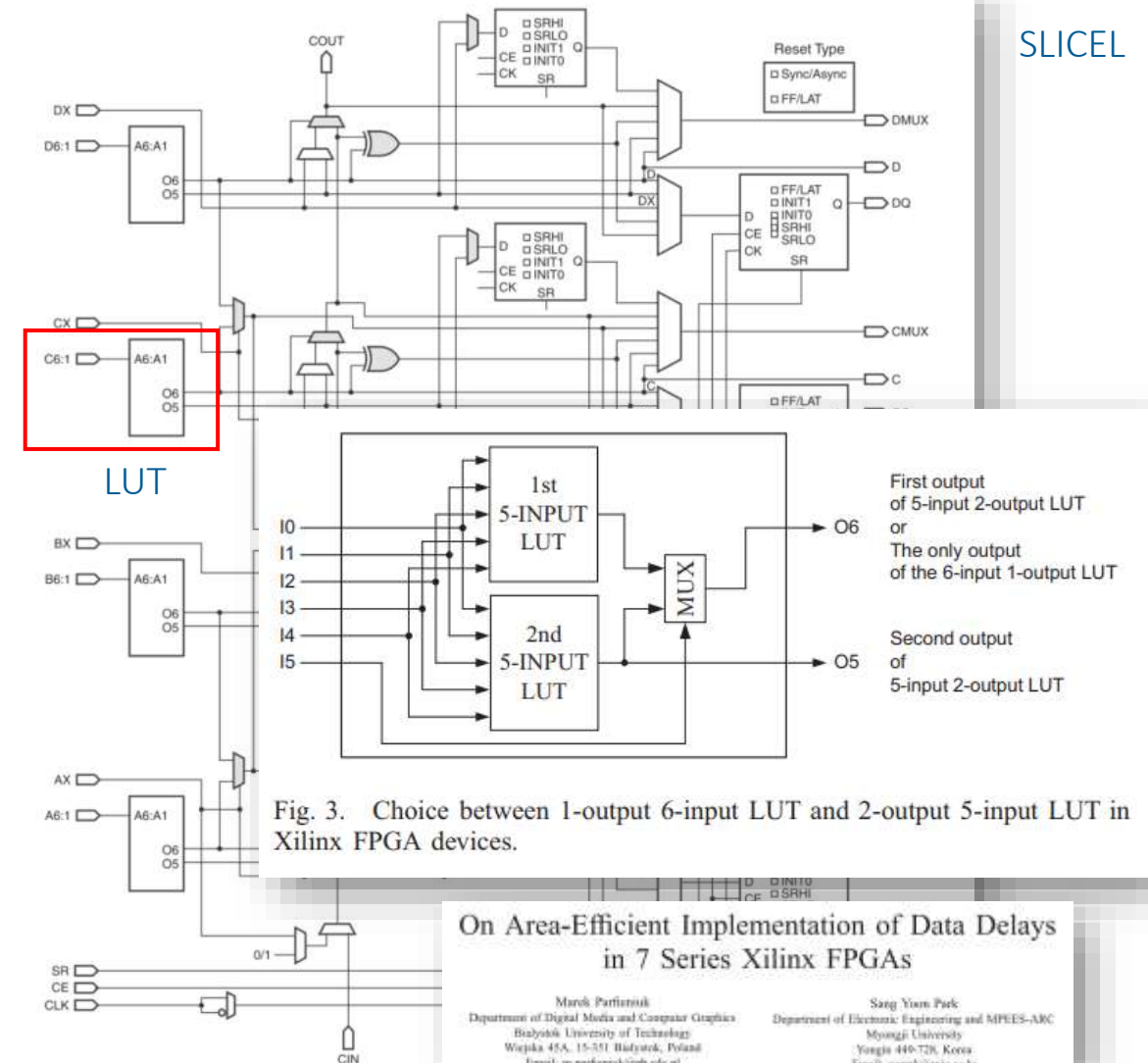


Fig. 3. Choice between 1-output 6-input LUT and 2-output 5-input LUT in Xilinx FPGA devices.

On Area-Efficient Implementation of Data Delays
in 7 Series Xilinx FPGAs

Marcel Parfeniuk
Department of Digital Media and Computer Graphics
Białystok University of Technology
Wiejska 45A, 15-551 Białystok, Poland
Email: m.parfeniuk@pwr.edu.pl

Sang Yoon Park
Department of Electronic Engineering and MPEES-ARC
Myongji University
Yongin 449-728, Korea
Email: syipark@mj.ac.kr

Figure 2-4: Diagram of SLICEL

Configuration Logic Block (CLB) of 7 Series FPGA

7 Series FPGAs Configurable Logic Block

User Guide

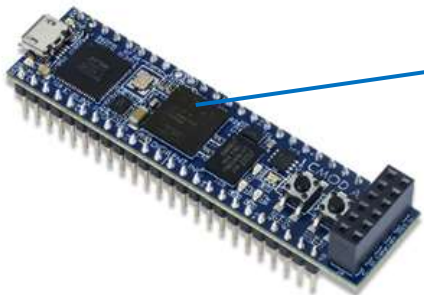
Slices = SLICEL + SLICEM

Distributed RAM (bit) = SLICEM x 256

UG474 (v1.8) September 27, 2016

Table 1-2: Artix-7 FPGA CLB Resources

Device	Slices ⁽¹⁾	SLICEL	SLICEM	6-input LUTs	Distributed RAM (Kb)	Shift Register (Kb)	Flip-Flops
7A12T	2,000 ⁽²⁾	1,316	684	8,000	171	86	16,000
7A15T	2,600 ⁽²⁾	1,800	800	10,400	200	100	20,800
7A25T	3,650	2,400	1,250	14,600	313	156	29,200
7A35T	5,200 ⁽²⁾	3,600	1,600	20,800	400	200	41,600
7A50T	8,150	5,750	2,400	32,600	600	300	65,200
7A75T	11,800 ⁽²⁾	8,232	3,568	47,200	892	446	94,400
7A100T	15,850	11,100	4,750	63,400	1,188	594	126,800
7A200T	33,650	22,100	11,550	134,600	2,888	1,444	269,200



- **CFU Playground[1]**

- ASIP (Application-Specific Instruction-set Processor) を用いるシステム開発では、独自の命令セットをサポートするプロセッサ設計とそのプロセッサを活用するソフトウェア設計という複雑な協調設計が必要になり、その開発時間の長期化が課題になっている。
- この課題に取り組むオープンソースのHW/SW協調設計フレームワーク

- **先行研究の課題**

- 複雑なファイル依存関係によりデバッグが困難
- 柔軟な設計のためにドメイン固有言語の習得が必要で学習コストが高い
- ライセンスが複雑で調査・管理にコストがかかり、違反のリスクがある

[1] S. Prakash et al ., “CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs,” 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp.157–167, 2023.

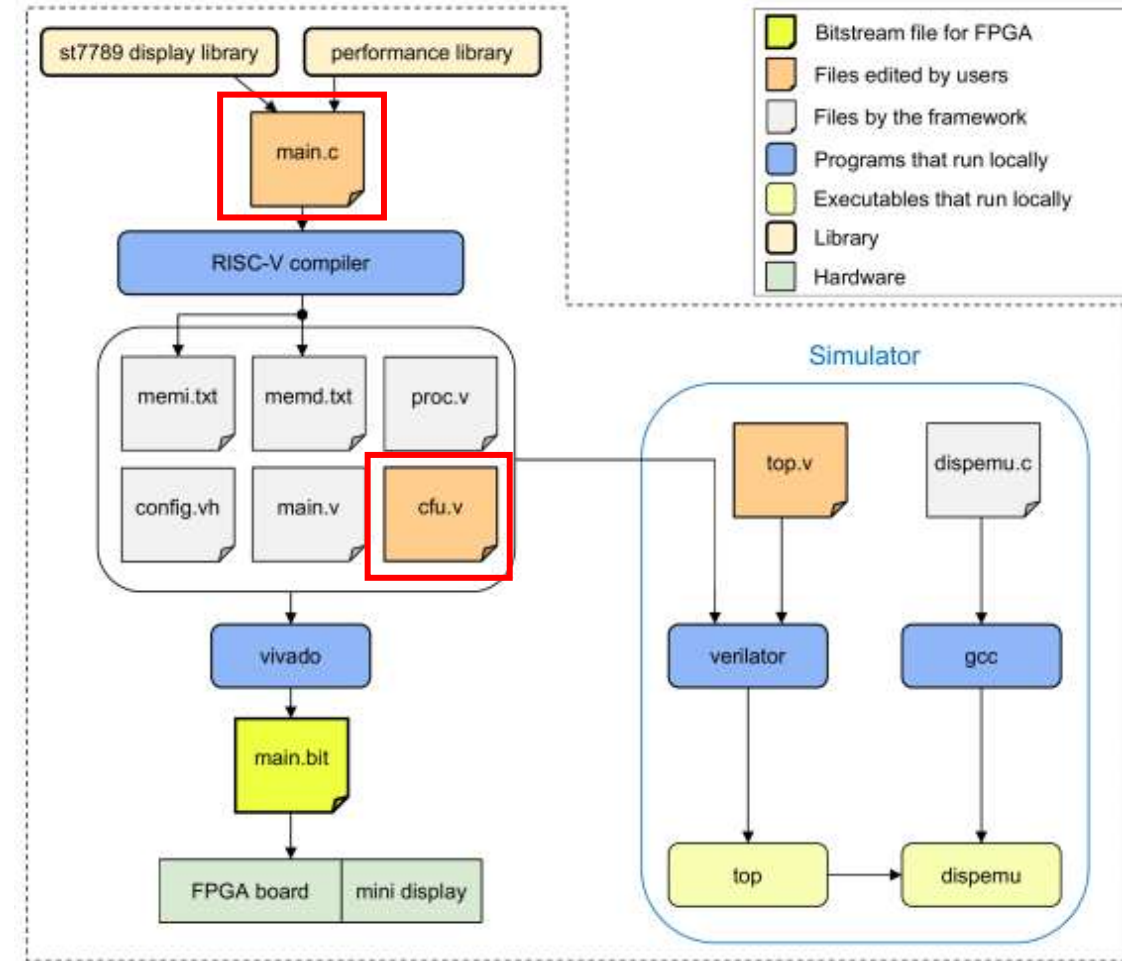
CFU Proving Ground (CFU PG) の目的と貢献

• 目的

- 先行研究（フレームワーク）の課題に対処する新しいHW/SW協調設計フレームワークとしてCFU Proving Groundを開発・提供する。

• 貢献

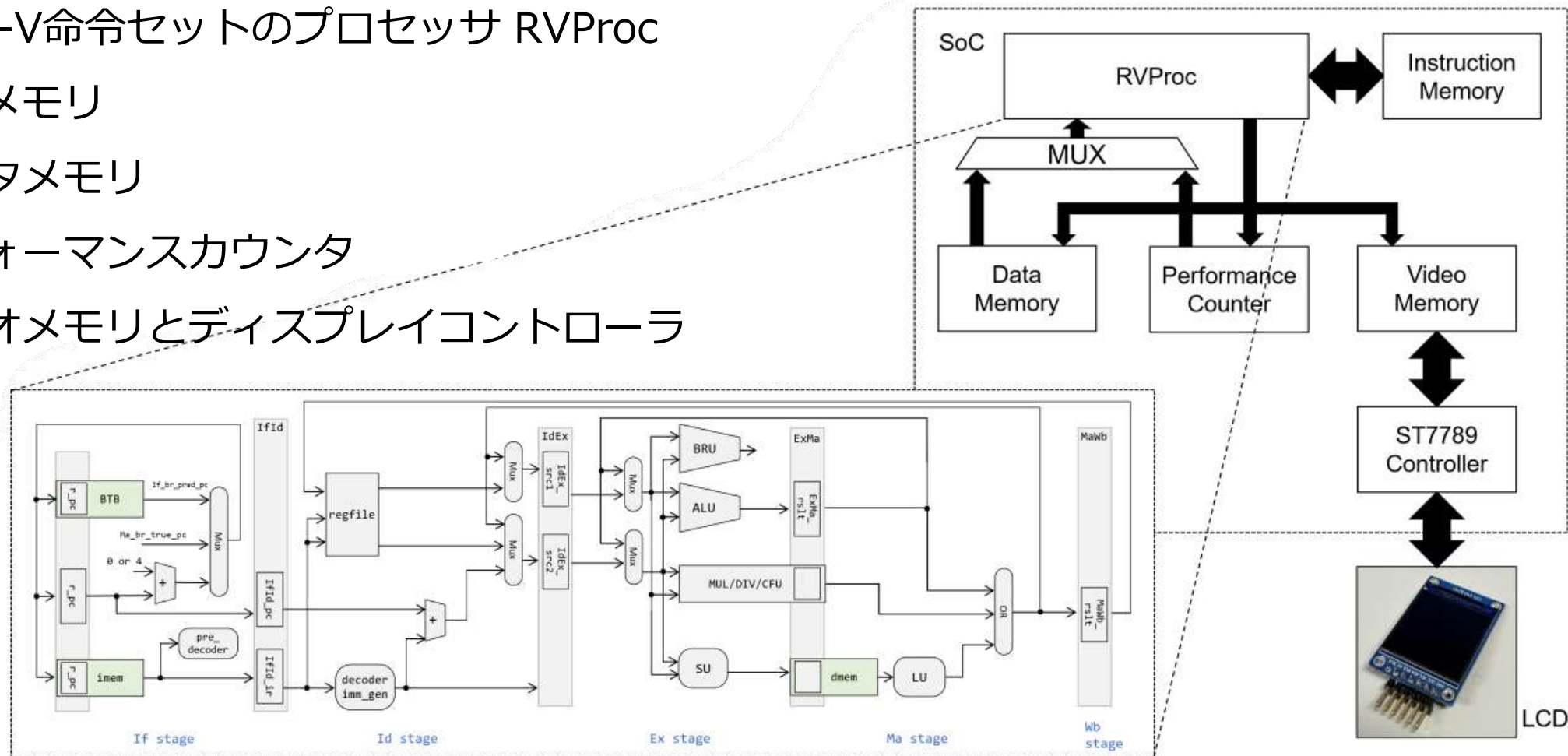
- 23個のファイル約2,700行のコードで構成され、この軽量な実装により、理解とデバッグが容易
- 少ない言語（C言語とVerilog HDL）による実装と主流のツールチェーンの使用により、操作方法の理解が容易
- すべてのファイルをMITライセンスでオープンソースとして公開することで、扱いが容易
- FPGAを活用するSoC開発の難しさを大幅に低減



CFU Proving Groundの設計フロー

CFU Proving Ground の SoC (System on Chip)

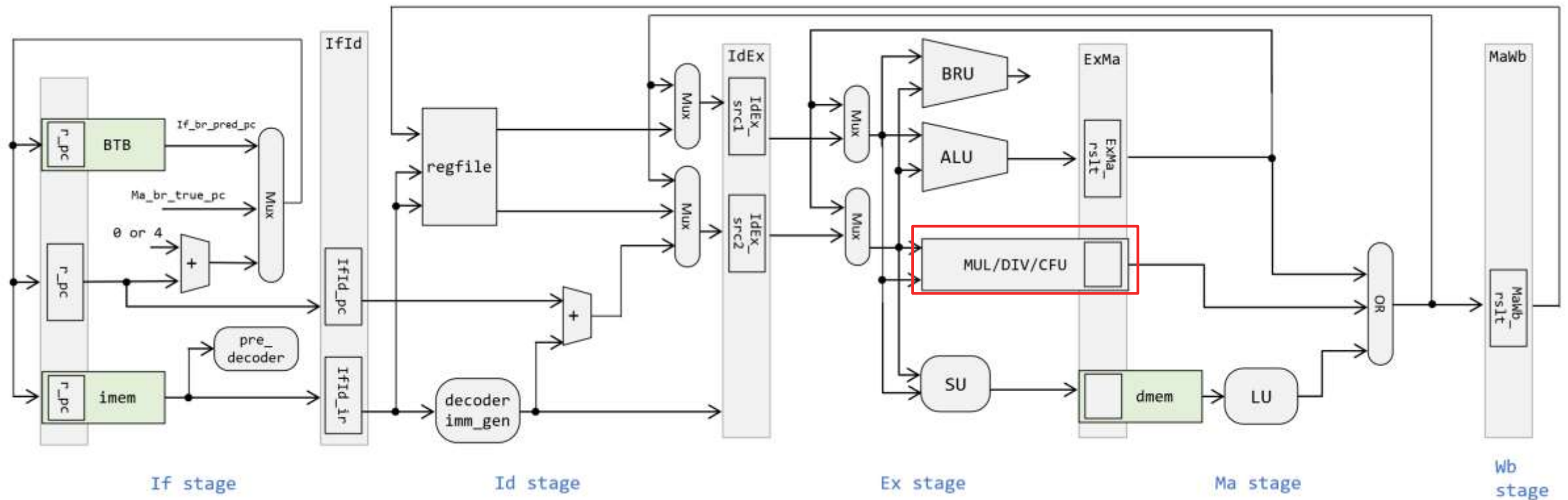
- FPGAに実装するSoCの構成要素
 - RISC-V命令セットのプロセッサ RVProc
 - 命令メモリ
 - データメモリ
 - パフォーマンスカウンタ
 - ビデオメモリとディスプレイコントローラ



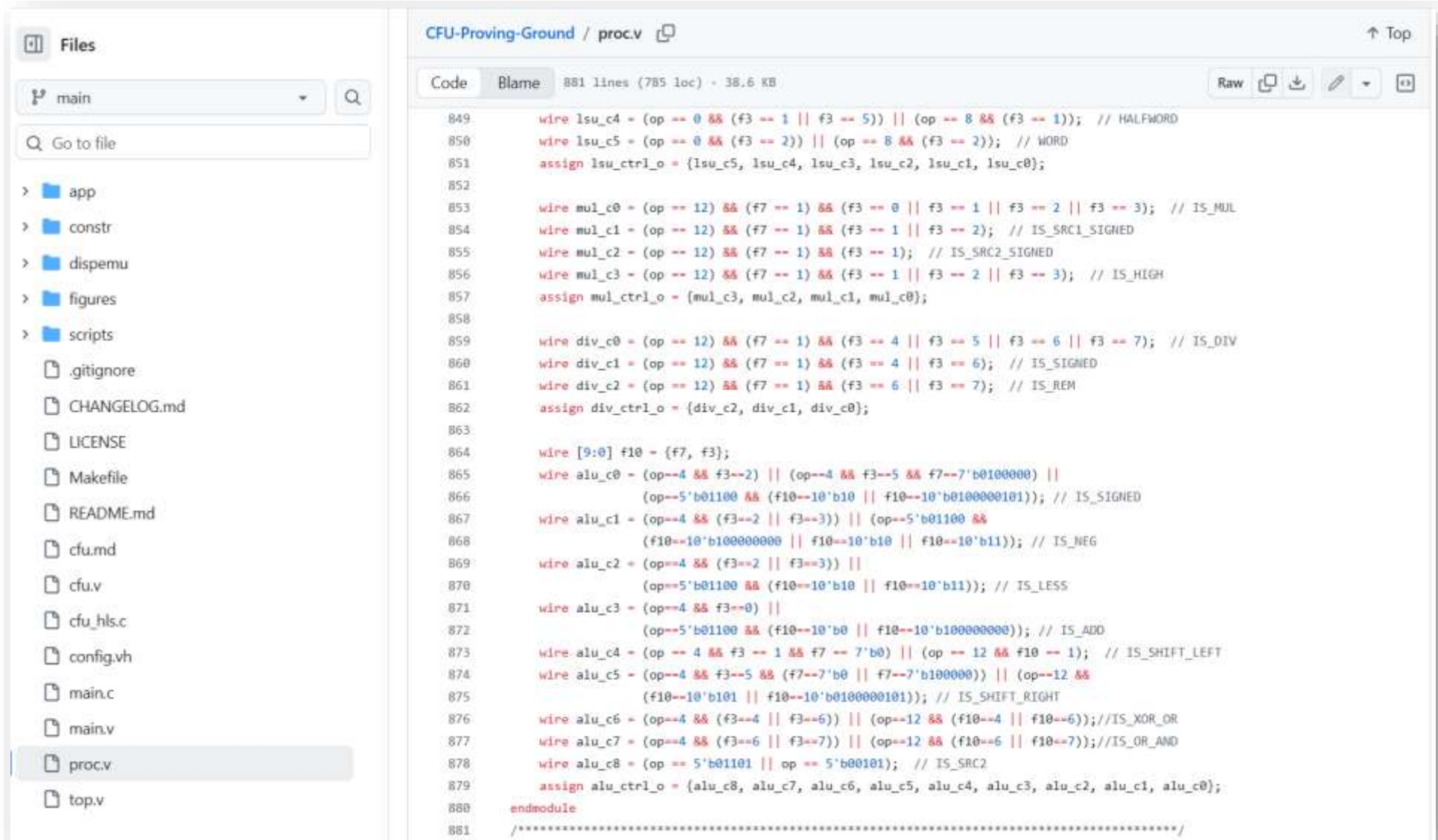
CFU Proving Ground のソフトプロセッサ RVProc

- RVProcの構成

- IF, ID, EX, MA, WBという5段のパイプライン処理のRV32IMのソフトプロセッサ
- キャッシュ無し、FPGAの内蔵メモリ(BRAM)を命令メモリ、データメモリとして活用
- CFU (Custom Function Unit)



CFU Proving Ground のソフトウェア RVProc



```
CFU-Proving-Ground / proc.v
Code Blame 881 lines (785 loc) • 38.6 KB
849 wire lsu_c4 = (op == 0 && (f3 == 1 || f3 == 5)) || (op == 8 && (f3 == 1)); // HALFWORD
850 wire lsu_c5 = (op == 0 && (f3 == 2)) || (op == 8 && (f3 == 2)); // WORD
851 assign lsu_ctrl_o = {lsu_c5, lsu_c4, lsu_c3, lsu_c2, lsu_c1, lsu_c0};
852
853 wire mul_c0 = (op == 12) && (f7 == 1) && (f3 == 0 || f3 == 1 || f3 == 2 || f3 == 3); // IS_MUL
854 wire mul_c1 = (op == 12) && (f7 == 1) && (f3 == 1 || f3 == 2); // IS_SRC1_SIGNED
855 wire mul_c2 = (op == 12) && (f7 == 1) && (f3 == 1); // IS_SRC2_SIGNED
856 wire mul_c3 = (op == 12) && (f7 == 1) && (f3 == 1 || f3 == 2 || f3 == 3); // IS_HIGH
857 assign mul_ctrl_o = {mul_c3, mul_c2, mul_c1, mul_c0};
858
859 wire div_c0 = (op == 12) && (f7 == 1) && (f3 == 4 || f3 == 5 || f3 == 6 || f3 == 7); // IS_DIV
860 wire div_c1 = (op == 12) && (f7 == 1) && (f3 == 4 || f3 == 6); // IS_SIGNED
861 wire div_c2 = (op == 12) && (f7 == 1) && (f3 == 6 || f3 == 7); // IS_REM
862 assign div_ctrl_o = {div_c2, div_c1, div_c0};
863
864 wire [9:0] f10 = {f7, f3};
865 wire alu_c0 = (op == 4 && f3 == 2) || (op == 4 && f3 == 5 && f7 == 7'b0100000) ||
866             (op == 5'b01100 && (f10 == 10'b10 || f10 == 10'b0100000101)); // IS_SIGNED
867 wire alu_c1 = (op == 4 && (f3 == 2 || f3 == 3)) || (op == 5'b01100 &&
868             (f10 == 10'b100000000 || f10 == 10'b10 || f10 == 10'b11)); // IS_NEG
869 wire alu_c2 = (op == 4 && (f3 == 2 || f3 == 3)) ||
870             (op == 5'b01100 && (f10 == 10'b10 || f10 == 10'b11)); // IS_LESS
871 wire alu_c3 = (op == 4 && f3 == 0) ||
872             (op == 5'b01100 && (f10 == 10'b0 || f10 == 10'b100000000)); // IS_ADD
873 wire alu_c4 = (op == 4 && f3 == 1 && f7 == 7'b0) || (op == 12 && f10 == 1); // IS_SHIFT_LEFT
874 wire alu_c5 = (op == 4 && f3 == 5 && (f7 == 7'b0 || f7 == 7'b100000)) || (op == 12 &&
875             (f10 == 10'b101 || f10 == 10'b0100000101)); // IS_SHIFT_RIGHT
876 wire alu_c6 = (op == 4 && (f3 == 4 || f3 == 6)) || (op == 12 && (f10 == 4 || f10 == 6)); // IS_XOR_OR
877 wire alu_c7 = (op == 4 && (f3 == 6 || f3 == 7)) || (op == 12 && (f10 == 6 || f10 == 7)); // IS_OR_AND
878 wire alu_c8 = (op == 5'b01101 || op == 5'b00101); // IS_SRC2
879 assign alu_ctrl_o = {alu_c8, alu_c7, alu_c6, alu_c5, alu_c4, alu_c3, alu_c2, alu_c1, alu_c0};
880 endmodule
881 /*****
```

RISC-VプロセッサをVerilog HDLで記述するための参考書

- RISC-Vで学ぶコンピュータアーキテクチャ 完全入門、吉瀬謙二、技術評論社



第1章 イントロダクション

- 1-1 コンピュータの基本構成
- 1-2 コンピュータの性能
- 1-3 特定用途向け半導体とFPGA

第2章 デジタル回路の基礎

- 2-1 組み合わせ回路
- 2-2 順序回路
- 2-3 やわらかいハードウェアとしてのFPGA

第3章 ハードウェア記述言語Verilog HDL

- 3-1 ANDゲートのモジュール記述
- 3-2 記述したモジュールのインスタンス化とシミュレーション
- 3-3 文字列を表示するシステムタスク\$display
- 3-4 ブロックの指定
- 3-5 指定した時間が経過するまで待たせる命令#
- 3-6 システムタスク\$finishと\$time
- 3-7 不定値xとハイインピーダンスz
- 3-8 複数本の信号線、数値の表現、default_nettype
- 3-9 三項演算子とマルチプレクサ
- 3-10 ビット選択
- 3-11 ビットの連結と複製
- 3-12 論理演算子と算術演算子
- 3-13 関係演算子
- 3-14 論理シフト、算術シフトの演算子
- 3-15 リダクション演算子
- 3-16 if文とcase文
- 3-17 クロック信号とカウンタ回路
- 3-18 シフトレジスタと波形ビューア
- 3-19 一定の間隔で生じる動作の記述とdefine文
- 3-20 ランダムアクセスメモリ (RAM) とinclude文
- 3-21 いくつかの代入の違い

第4章 RISC-V命令セットアーキテクチャ

- 4-1 RISC-V RV32I命令セットアーキテクチャの概要
- 4-2 データ形式、負の整数の表現
- 4-3 命令形式
- 4-4 R形式の算術演算命令、論理演算命令、シフト命令
- 4-5 I形式の算術演算命令、論理演算命令、シフト命令
- 4-6 ロード命令、ストア命令、エンディアンと整列
- 4-7 条件分岐命令とプログラムカウンタ
- 4-8 lui, auipc, jal, jalr命令とその他の命令

第5章 単一サイクルのプロセッサ

- 5-1 単一サイクルのプロセッサの設計方針
- 5-2 最初の版のプロセッサを設計するための構成要素
- 5-3 add命令を処理するx1のみの単一サイクルのプロセッサ
- 5-4 add命令を処理する単一サイクルのプロセッサ
- 5-5 addとaddi命令を処理する単一サイクルのプロセッサ
- 5-6 add, addi, lw, sw命令を処理する単一サイクルのプロセッサ
- 5-7 add, addi, lw, sw, bne命令を処理する単一サイクルのプロセッサ
- 5-8 シミュレーションの工夫と例題

第6章 プロセッサの高性能化の手法

- 6-1 回路の動作周波数とパイプライン処理
- 6-2 パイプライン処理 (2段) のプロセッサの設計と実装
- 6-3 パイプライン処理 (3段) のプロセッサの設計と実装
- 6-4 パイプライン処理 (4段) のプロセッサの設計と実装
- 6-5 パイプライン処理のプロセッサと同期式メモリ
- 6-6 パイプライン処理 (5段) のプロセッサの設計と実装
- 6-7 ここまでのプロセッサの性能

第7章 分岐予測

- 7-1 分岐予測の枠組み
- 7-2 分岐先バッファ
- 7-3 分岐の成立／不成立の予測
- 7-4 bimodal分岐予測
- 7-5 gshare分岐予測

第8章 キャッシュメモリ

- 8-1 メインメモリとキャッシュ
- 8-2 容量が大きくて遅いメモリ
- 8-3 プロセッサのストール
- 8-4 ダイレクトマップ方式のキャッシュメモリ
- 8-5 マルチワードのダイレクトマップ方式のキャッシュメモリ
- 8-6 セットアソシアティブ方式のキャッシュメモリ
- 8-7 データキャッシュ

第9章 FPGA評価ボードを利用した動作の確認

- 9-1 ファイルの準備
- 9-2 Vivadoで論理合成、配置・配線してFPGAで動作確認

CoreMark を用いたプロセッサの性能比較

- CoreMark
 - VexRiscv との比較では評価環境を合わせて speed grade -3 として測定
 - ESP32C3 との比較では、Cmod A7 が搭載する FPGA の speed grade -1 として測定
- 商用マイコン **ESP32-C3** [1] のスコア 483 に匹敵する
- オープンソースの代表的なソフトプロセッサ **VexRiscv** [2] より高性能

Processor	Frequency	CoreMark (Iteration/Second)	CoreMark/MHz
ESP32-C3[3]	160 MHz	483	3.02
RVProc (Speed Grade -1)	175 MHz	464	2.65
VexRiscv (Speed Grade -3)[4]	200 MHz	514	2.57
RVProc (Speed Grade -3)	235 MHz	623	2.65

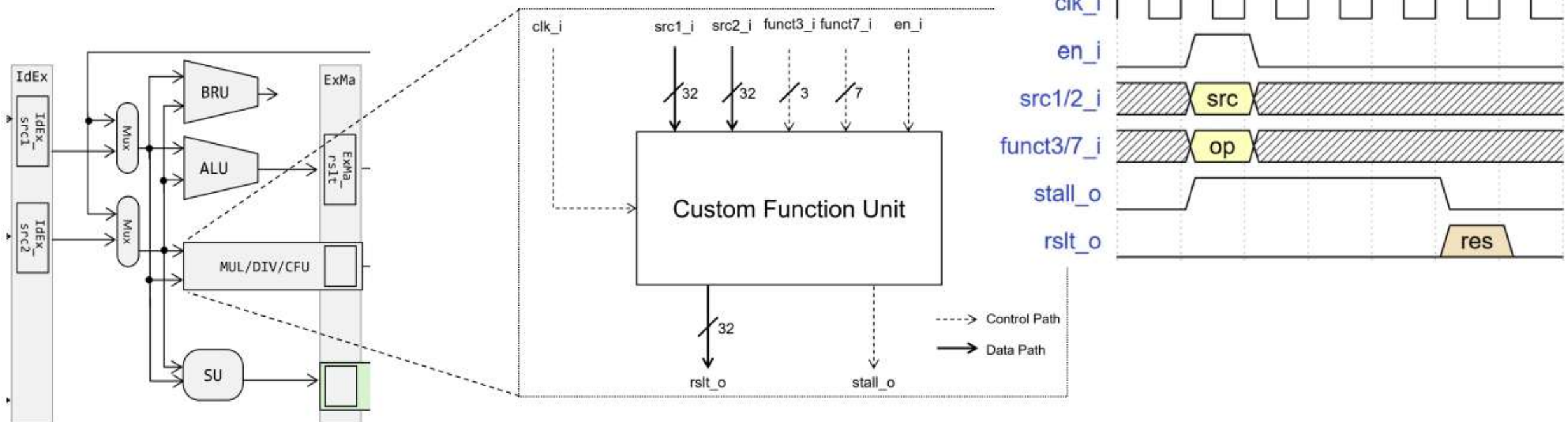


[1] ESPRESSIF, “ESP32-C3 Series Datasheet Version 2.0,” 2024.

[2] SpinalHDL, “VexRiscv,” <https://github.com/SpinalHDL/VexRiscv>, 2025. GitHub repository. Accessed: March 7, 2025.

CFU Proving Ground の CFU

- Custom Function Unit (CFU)
 - 本フレームワークにおけるアクセラレータ
 - 2入力、1出力というALUベースの演算を採用することで、プロセッサのデータパスを活かした実装が可能



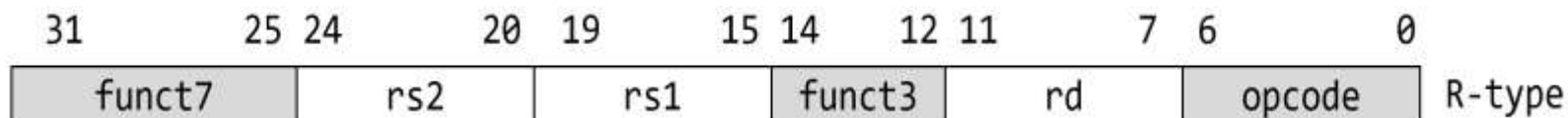
CFU Proving Ground のカスタム命令

- CFUを制御するためにカスタム命令を活用する
 - RISC-Vで定義されている拡張命令セットの1つ

Table 70. RISC-V base opcode map, $inst[1:0]=11$

$inst[4:2]$	$inst[6:5]$	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	OP-V	custom-2/rv128	48b
11		BRANCH	JALR	reserved	JAL	SYSTEM	OP-VE	custom-3/rv128	≥80b

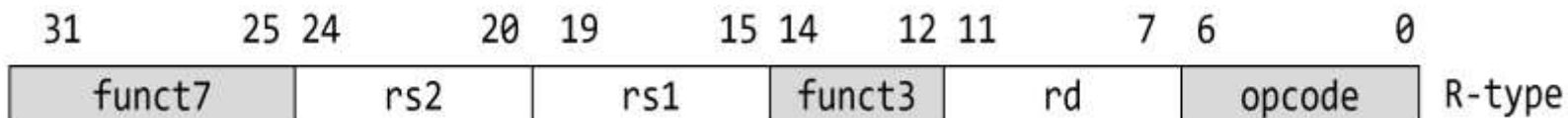
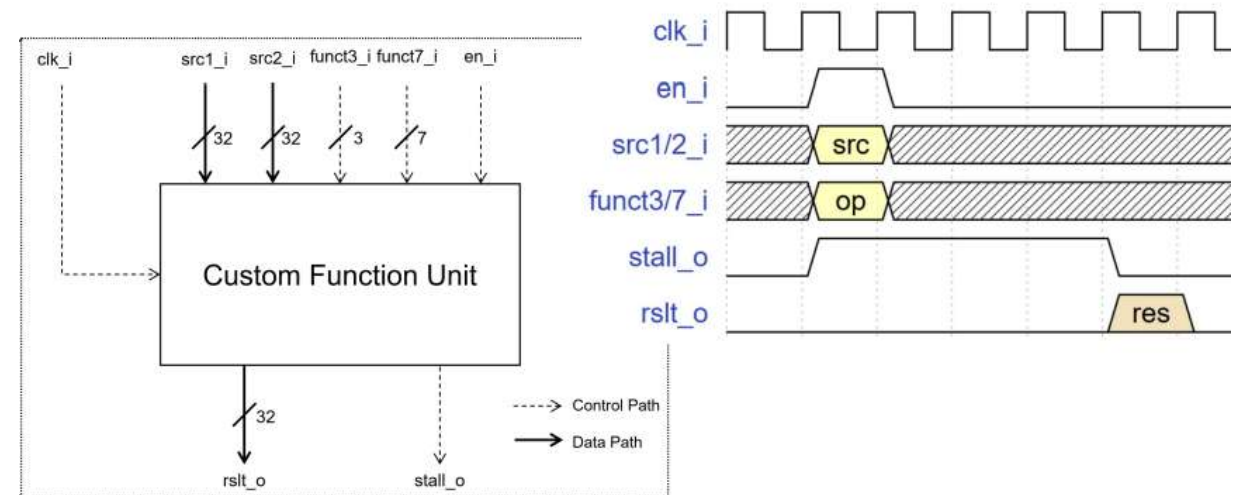
- ユーザーが独自にフィールドを定義して使用できる
- ALUベースというCFUの特徴から、R形式を採用する



- カスタム命令をC言語から使用するために、インラインアセンブラを使用する
 - GCCコンパイラに付属する機能の1つ
 - インラインアセンブラを利用したプログラム（ソフトウェア）の記述例

```
#define cfu_op(func7, funct3, rs1, rs2) ({  
    unsigned int __result;  
    asm volatile(  
        ".insn r CUSTOM_0, %3, %4, %0, %1, %2"  
        : "=r"(__result)  
        : "r"(rs1), "r"(rs2), "i"(funct3), "i"(func7)  
    );  
    __result;  
})
```

```
rslt = cfu_op(func7, funct3, in1, in2);
```



ソフトウェア: ボトルネック測定

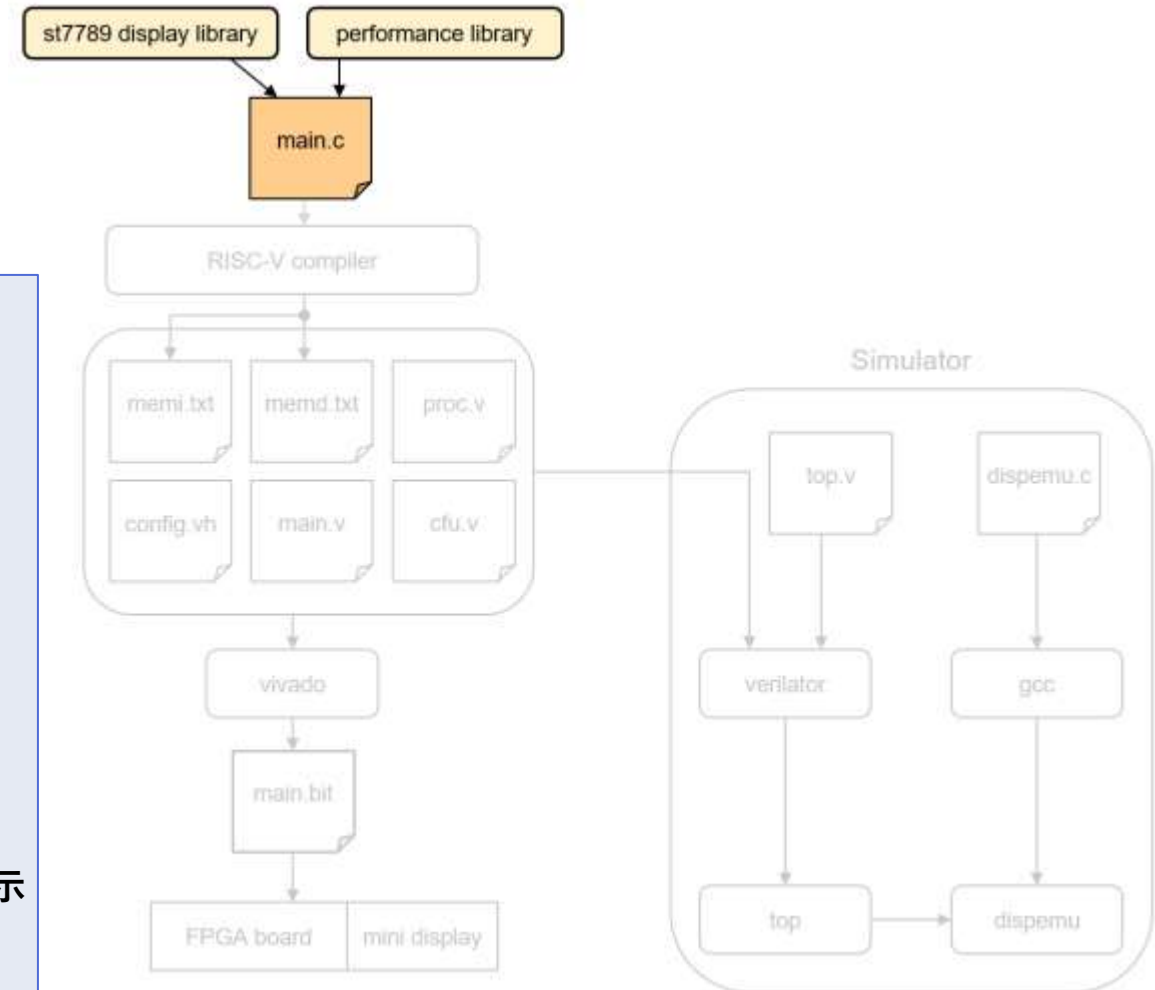
- 目的: CFUの実装で最大限の効果を得られるポイントを探す
 - パフォーマンスライブラリで実行クロックサイクルを測定できる

```
pg_perf_reset(); // カウンタを0に初期化
long long start = pg_perf_cycle(); // カウンタの値の読み取り
pg_perf_enable(); // カウント開始

for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        c[i] += a[i] * b[j];
    }
}

pg_perf_disable(); // カウント停止
long long end = pg_perf_cycle(); // カウントの値の読み取り
pg_lcd_prints("Elapsed Time:¥n"); // 文字列をミニディスプレイに表示
pg_lcd_printd(end - start); // 整数をミニディスプレイに表示
pg_lcd_prints("¥n");
```

測定するスコープ



ソフトウェア: プログラムのコンパイル

- 目的: RISC-V向けクロスコンパイルとメモリーイメージの作成

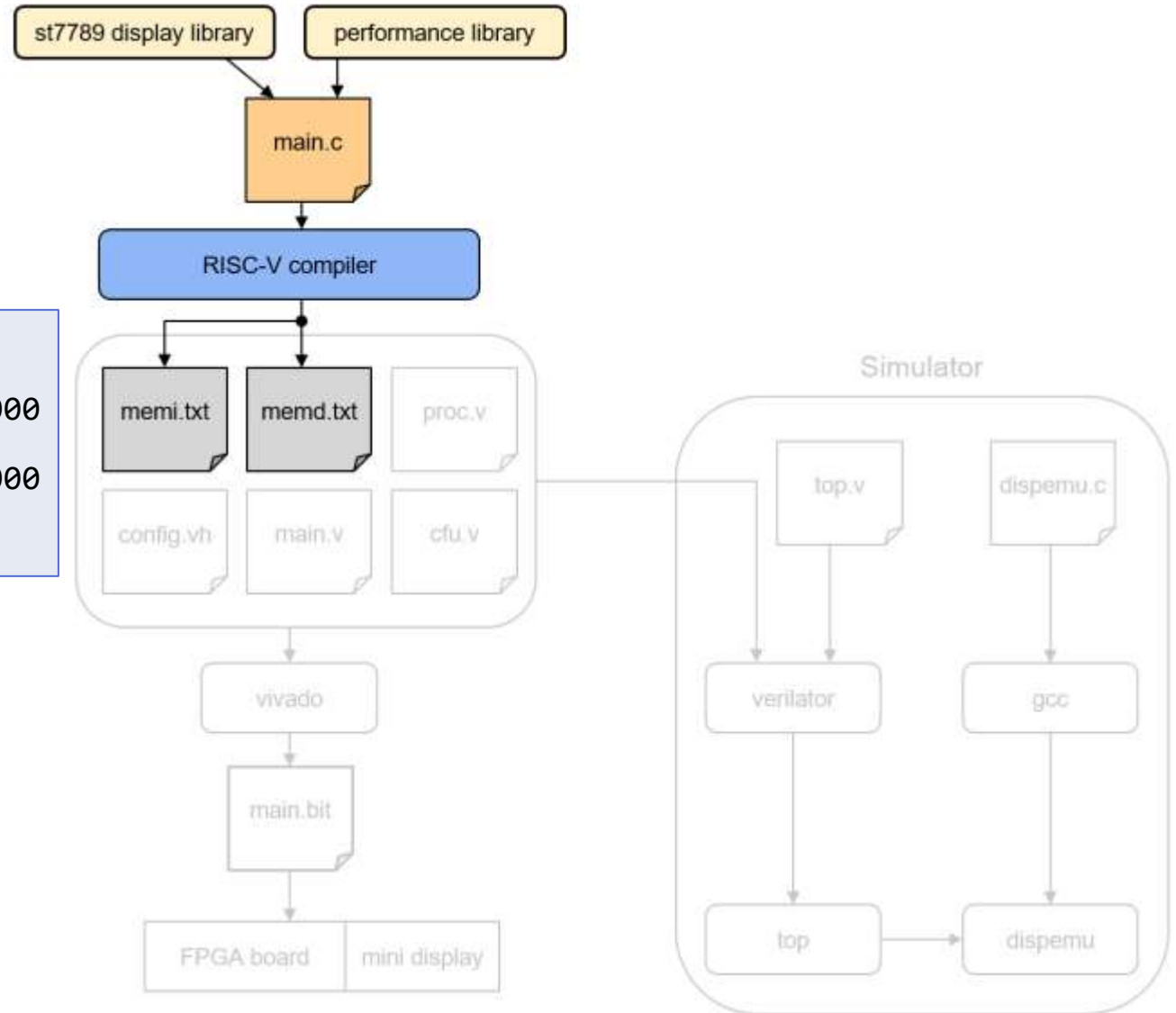
```
$ make prog
```

- link.ldのメモリーサイズの指定箇所

```
MEMORY {  
    imem : ORIGIN = 0x00000000, LENGTH = 0x00008000  
    dmem : ORIGIN = 0x10000000, LENGTH = 0x00004000  
}
```

- config.vhのメモリーサイズの指定箇所

```
// instruction memory size in byte  
`define IMEM_SIZE      (32*1024)  
  
// data memory size in byte  
`define DMEM_SIZE      (16*1024)
```



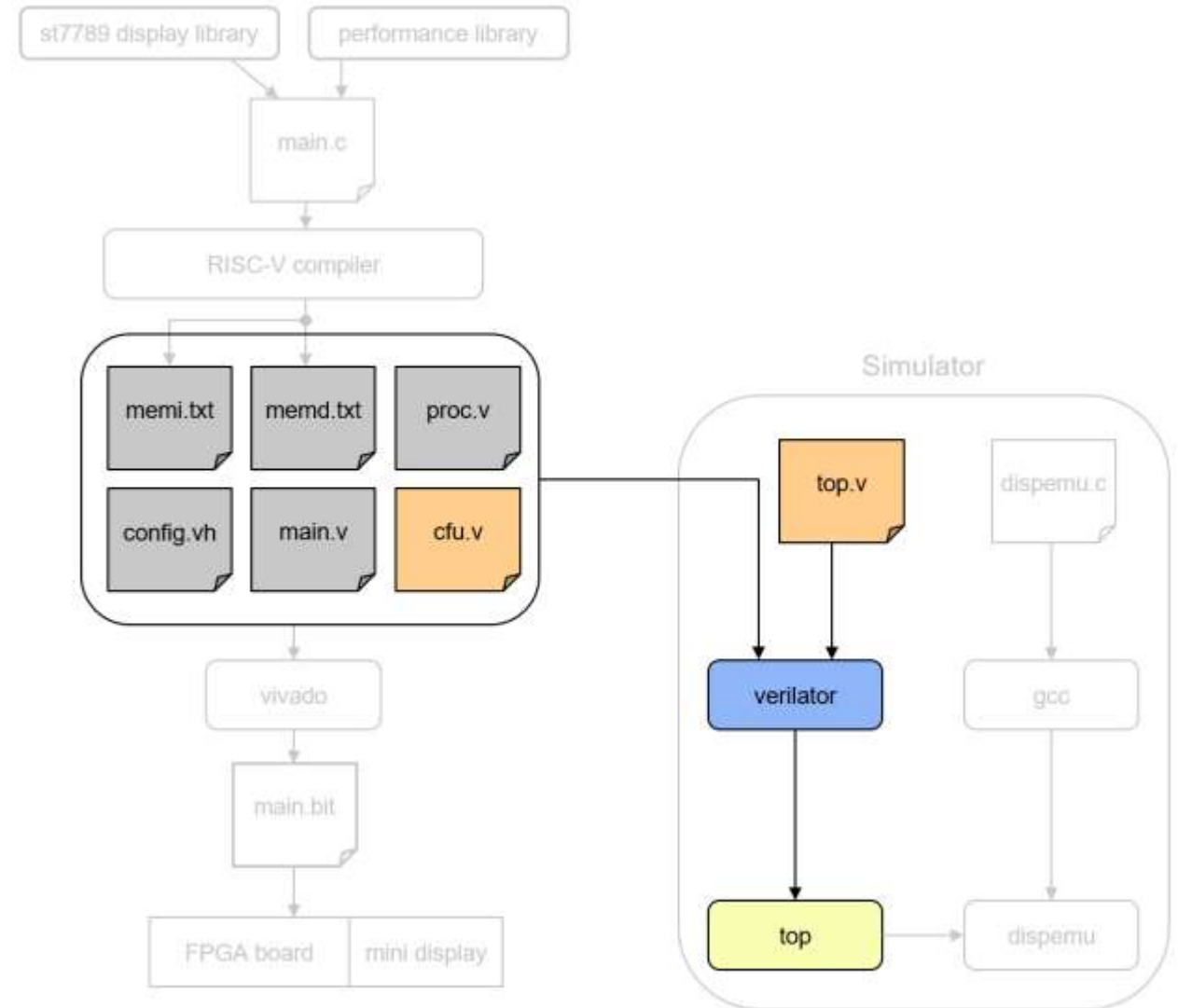
ソフトウェア: シミュレーションソースのコンパイル

- 目的: シミュレーションのためにプロセッサのVerilog記述をVerilator [1] でコンパイルする

```
$ verilator --binary --trace --top-module top ¥  
--Wno-WIDTHTRUNC --Wno-WIDTHEXPAND -o top *.v
```



```
$ make build
```



[1] <https://github.com/verilator/verilator>

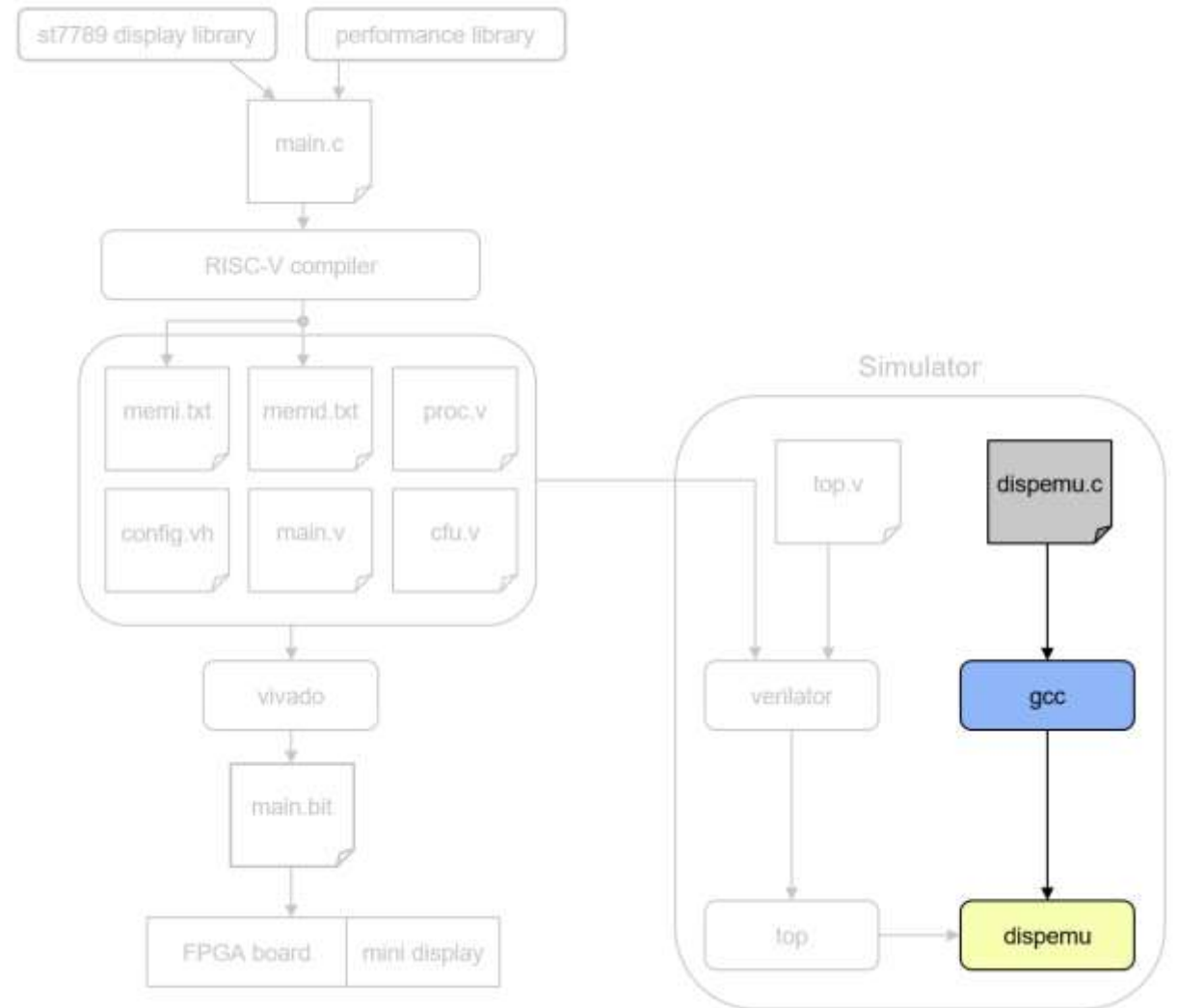
ソフトウェア: シミュレーションソースのコンパイル

- 目的: ディスプレイのシミュレータをコンパイルする

```
$ gcc -O2 dispemu/dispemu.c -o build/dispemu ¥  
-lcairo -lX11
```



```
$ make build
```



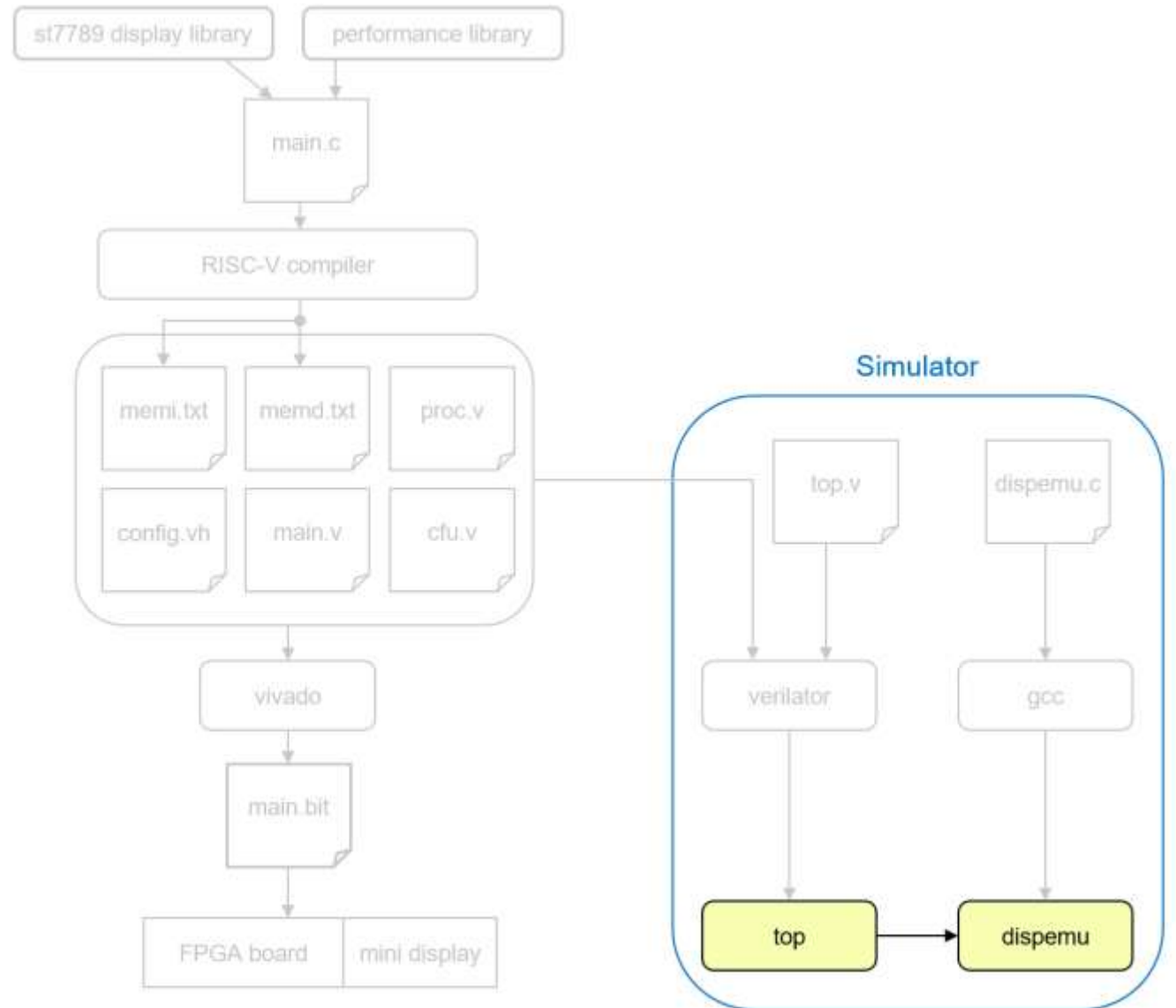
ソフトウェア: 実行クロックサイクルの評価

- 目的: 指定したスコープの実行に何サイクルかかっているか知る

```
$ ./obj_dir/top | build/dispemu
```



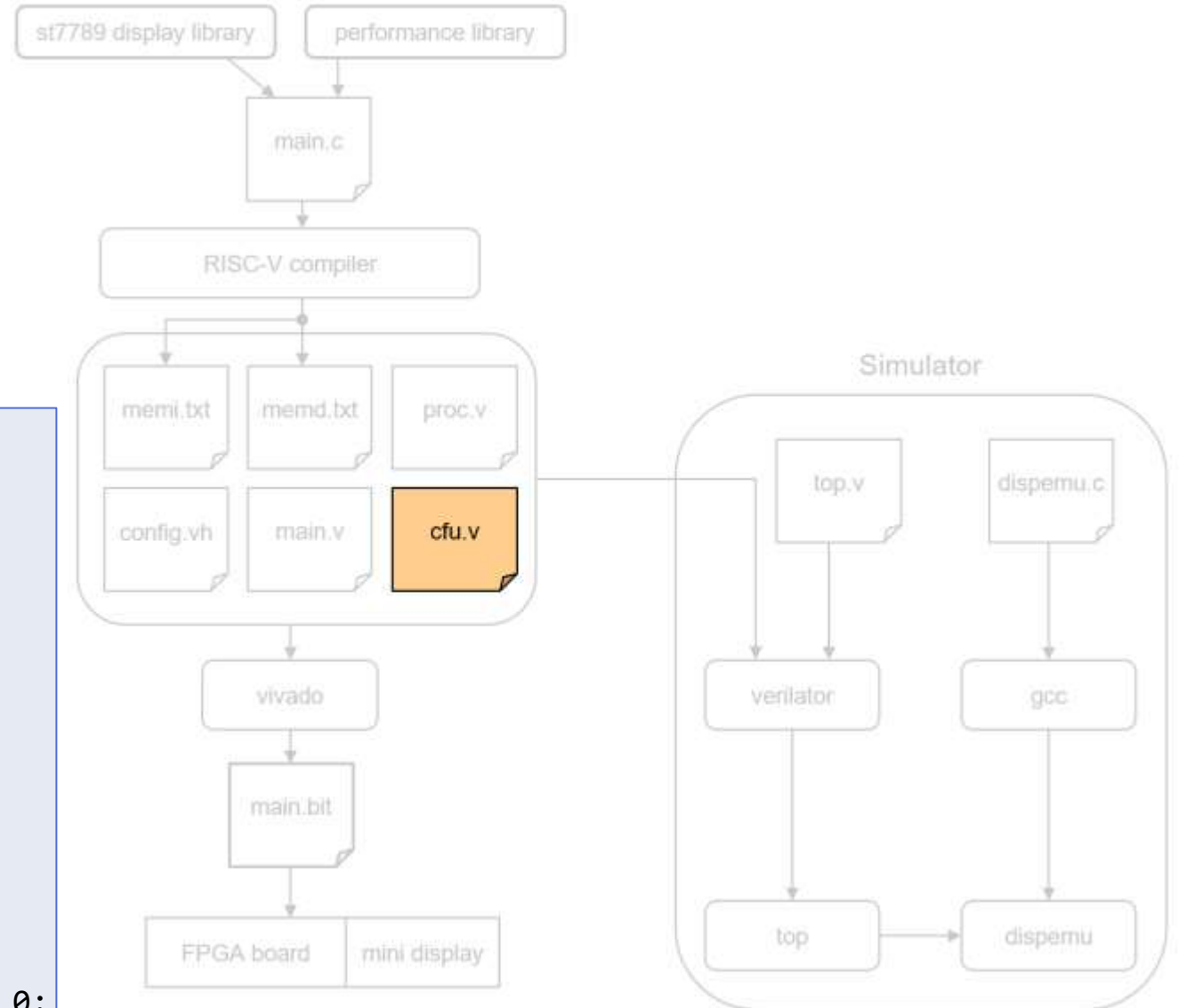
```
$ make drun
```



ハードウェア: アクセラレータの設計

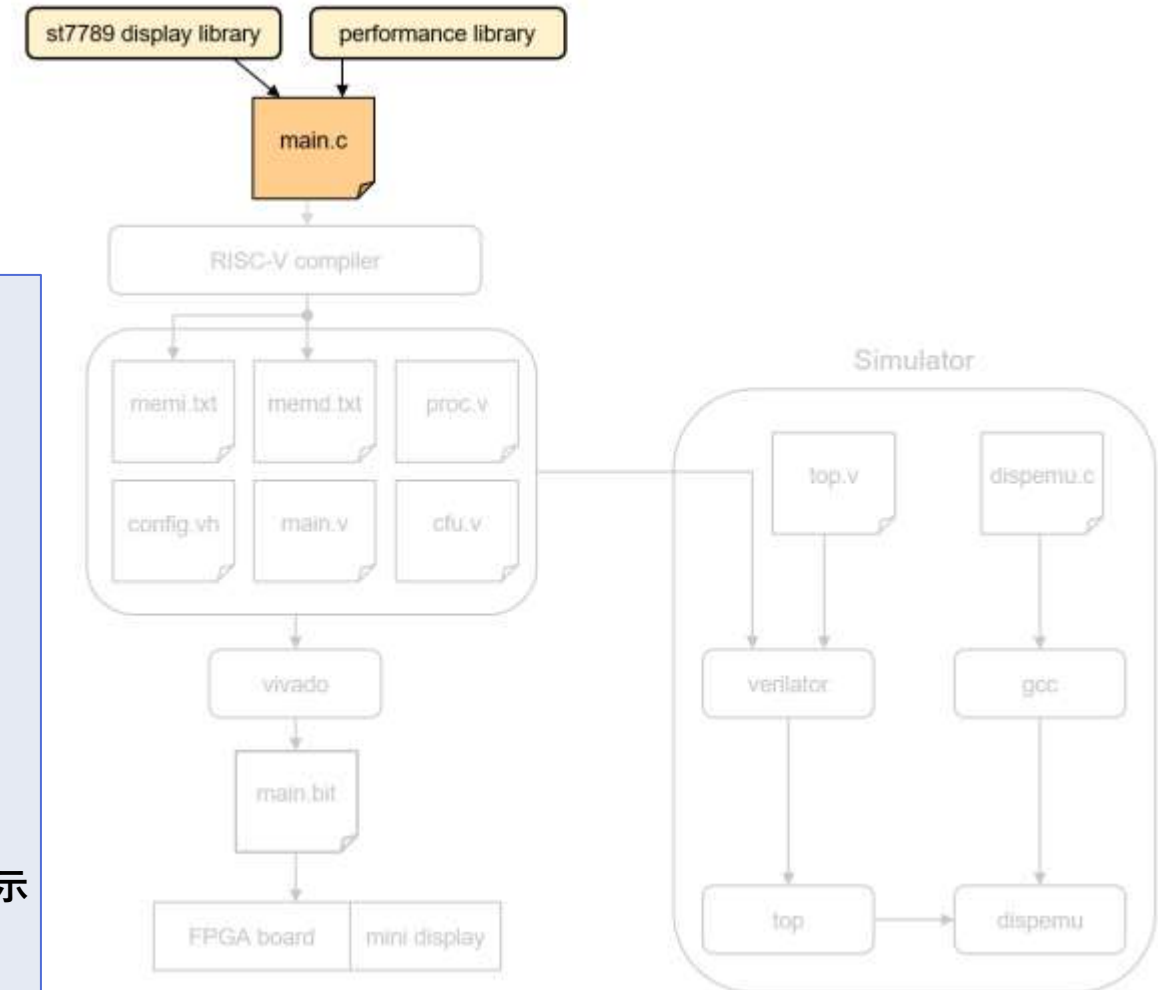
- 目的: ボトルネックとなっていた箇所を高速化するCFUを設計する
 - ユーザはここでカスタムロジックを記述する
 - 乗算を行うCFUのVerilog記述

```
module cfu (  
    input  wire      clk_i,  
    input  wire      en_i,  
    input  wire [ 2:0] funct3_i,  
    input  wire [ 6:0] funct7_i,  
    input  signed wire [31:0] src1_i,  
    input  signed wire [31:0] src2_i,  
    output wire      stall_o,  
    output wire [31:0] rslt_o  
);  
    assign stall_o = 0;  
    assign rslt_o = (en_i) ? src1_i * src2_i : 0;  
endmodule
```



- CFUを利用するように、カスタム命令を記述

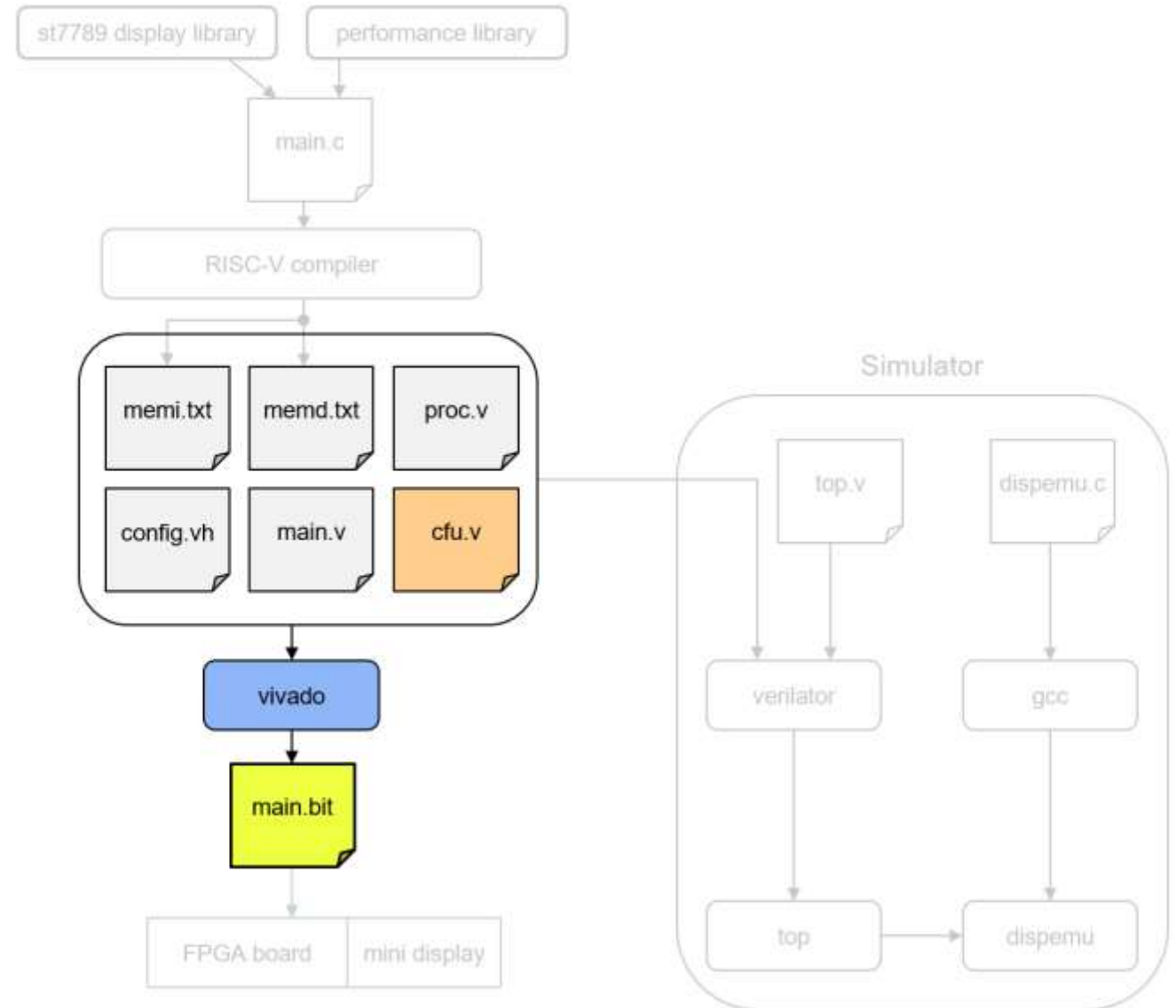
```
pg_perf_reset(); // カウンタを0に初期化
long long start = pg_perf_cycle(); // カウンタの値の読み取り
pg_perf_enable(); // カウント開始
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        c[i] += cfu_op(0, 0, a[i], b[j]);
    }
}
pg_perf_disable(); // カウント停止
long long end = pg_perf_cycle(); // カウントの値の読み取り
pg_lcd_prints("Elapsed Time:¥n"); // 文字列をミニディスプレイに表示
pg_lcd_printd(end - start); // 整数をミニディスプレイに表示
pg_lcd_prints("¥n");
```



ハードウェア: bitstreamの合成

- 目的：FPGAをコンフィギュレーションするためにVivadoでbitstreamを生成する
- config.vhから動作周波数を読み取り、Clocking Wizardを生成してbitstreamを合成する

```
$ make bit
```



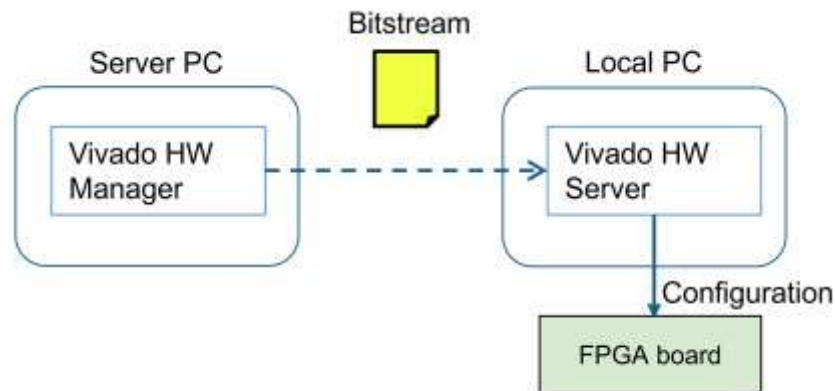
ハードウェア: FPGAへのロード

- 目的：合成したbitstreamをVivadoでFPGAにコンフィギュレーションする

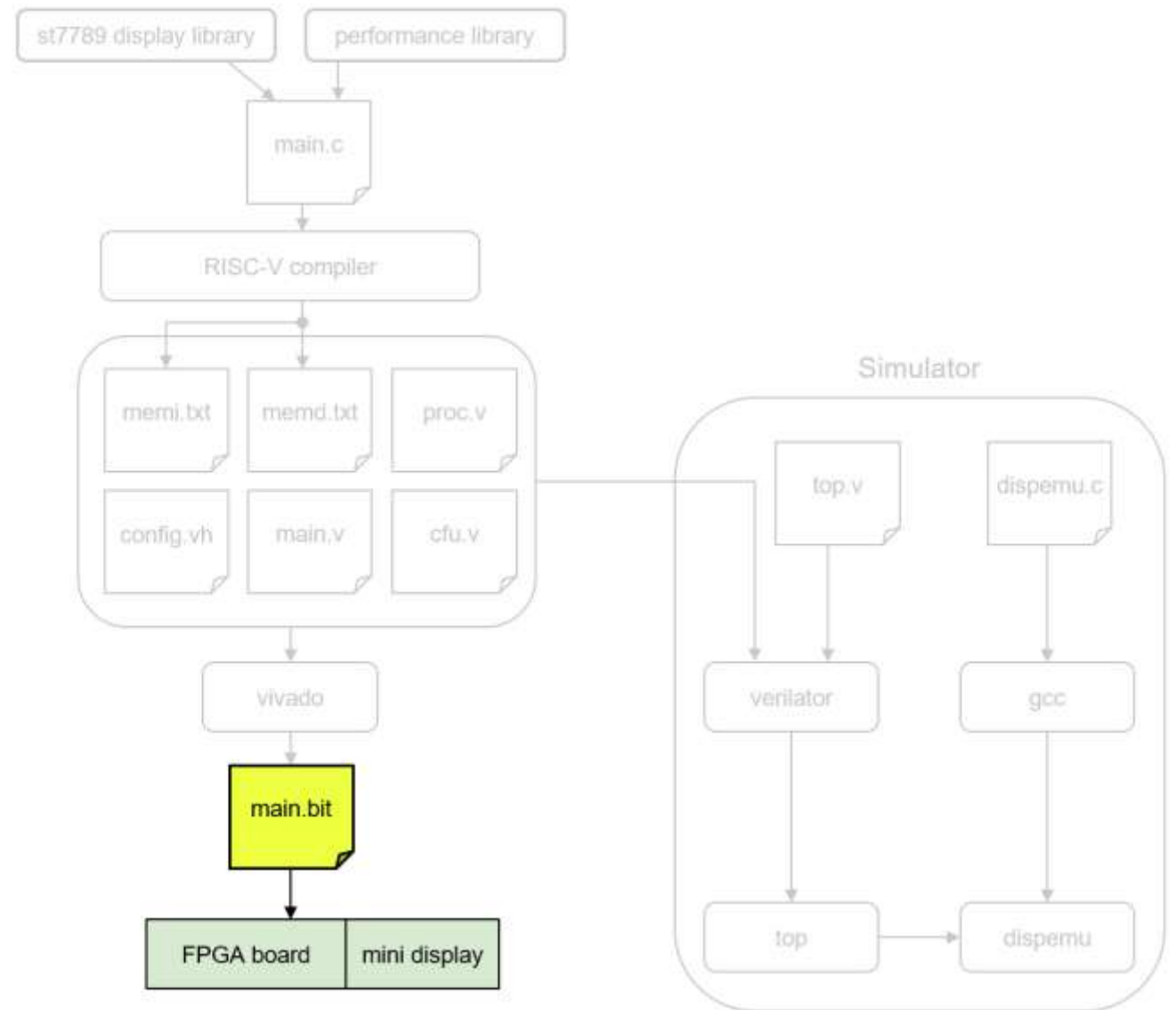
- scripts/prog_dev.tcl

```
set ip_addr 192.168.0.xxx  
set port 3121
```

- コンフィギュレーションの詳細



```
$ make conf
```



マンデルブロー集合の高速化の例

- C言語で記述したベースラインのコード
- 240 x 240 ピクセルのディスプレイを利用

```
/******  
**** 240x240 ST7789 mini display project Ver.2024-12-21a *****/  
**** Copyright (c) 2024 Archlab. Science Tokyo *****/  
**** Released under the MIT license https://opensource.org/licenses/mit *****/  
*****  
#include <stdio.h>  
#include <stdlib.h>  
#include <stdint.h>  
  
#define X_PIX    240 // display width  
#define Y_PIX    240 // display height  
#define ITER_MAX 256 // max iteration of mandelbrot  
  
float x_min = 0.270851;  
float x_max = 0.270900;  
float y_min = 0.004641;  
float y_max = 0.004713;  
  
void draw_pixel(int x, int y, int k)  
{  
    int color = ((k & 0x7f) << 11) ^ ((k & 0x7f) << 7) ^ (k & 0x7f);  
    int addr = ((x << 8) & 0xFF00) | (y & 0xFF);  
    printf("@%d_%d¥n", addr, color);  
    fflush(stdout);  
}
```

```
void mandelbrot()  
{  
    float dx = (x_max - x_min) / X_PIX;  
    float dy = (y_max - y_min) / Y_PIX;  
  
    for (int j = 1; j <= Y_PIX; j++) {  
        float y = y_min + j * dy;  
        for(int i = 1; i <= X_PIX; i++) {  
            int k;  
            float u = 0.0;  
            float v = 0.0;  
            float u2 = 0.0;  
            float v2 = 0.0;  
            float x = x_min + i * dx;  
            for (k = 1; k < ITER_MAX; k++) {  
                v = 2 * u * v + y;  
                u = u2 - v2 + x;  
                u2 = u * u;  
                v2 = v * v;  
                if (u2 + v2 >= 4.0) break;  
            };  
            draw_pixel(i, j, k);  
        }  
    }  
}  
  
int main()  
{  
    int cnt = 0;  
    float delta = 0.0000030;  
  
    while(1){  
        cnt++;  
        if(cnt%512==0) delta *= -1;  
        y_max += delta;  
        x_max += delta;  
        mandelbrot();  
  
        if(cnt%10==0) fprintf(stderr, "cnt = %d¥n", cnt);  
    }  
    return 0;  
}
```

マンデルブロー集合の高速化の例

- ベースラインの赤色をカスタム命令に
- CFUは、C言語で記述して High Level Synthesis (高位合成, HLS) を用いて Verilog HDL に変換

```
void mandelbrot()
{
    float dx = (x_max - x_min) / X_PIX;
    float dy = (y_max - y_min) / Y_PIX;

    for (int j = 1; j <= Y_PIX; j++) {
        float y = y_min + j * dy;
        for(int i = 1; i <= X_PIX; i++) {
            int k;
            float u = 0.0;
            float v = 0.0;
            float u2 = 0.0;
            float v2 = 0.0;
            float x = x_min + i * dx;
            for (k = 1; k < ITER_MAX; k++) {
                v = 2 * u * v + y;
                u = u2 - v2 + x;
                u2 = u * u;
                v2 = v * v;
                if (u2 + v2 >= 4.0) break;
            };
            draw_pixel(i, j, k);
        }
    }
}
```

```
#include <string.h>
#define ITER_MAX 256

void cfu_hls(char funct3_i, char funct7_i, int src1_i, int src2_i, int* rslt_o)
{
    float x; memcpy(&x, &src1_i, sizeof(float));
    float y; memcpy(&y, &src2_i, sizeof(float));
    float u = 0.0;
    float v = 0.0;
    float u2 = 0.0;
    float v2 = 0.0;
    int k;
    for (k = 1; k < ITER_MAX; k++) {
        v = 2 * u * v + y;
        u = u2 - v2 + x;
        u2 = u * u;
        v2 = v * v;
        if (u2 + v2 >= 4.0) break;
    };
    *rslt_o = k;
}
```

```
void mandelbrot()
{
    float dx = (x_max - x_min) / X_PIX;
    float dy = (y_max - y_min) / Y_PIX;

    for (int j = 1; j <= Y_PIX; j++) {
        float y = y_min + j * dy;
        for(int i = 1; i <= X_PIX; i++) {
            int k;
            float x = x_min + i * dx;

            int src1 = *((int *)&x);
            int src2 = *((int *)&y);
            cfu_op(0, 0, src1, src2, &k); // custom instruction
            draw_pixel(i, j, k);
        }
    }
}
```

マイコンボードとFPGAボードでマンデルブロー集合を描画した例

- 右はマイコンボード（160MHz）でのソフトウェア処理による描画（1フレームの描画に約33秒）
- 中央はFPGAボード（160MHz）にプロセッサとアクセラレータを実装（1フレームの描画に約1.43秒、**マイコンの23倍**）
- 左は、FPGA版のアクセラレータを12個に増やして（150MHz）描画（1フレームの描画に約0.152秒、**マイコンの217倍**）



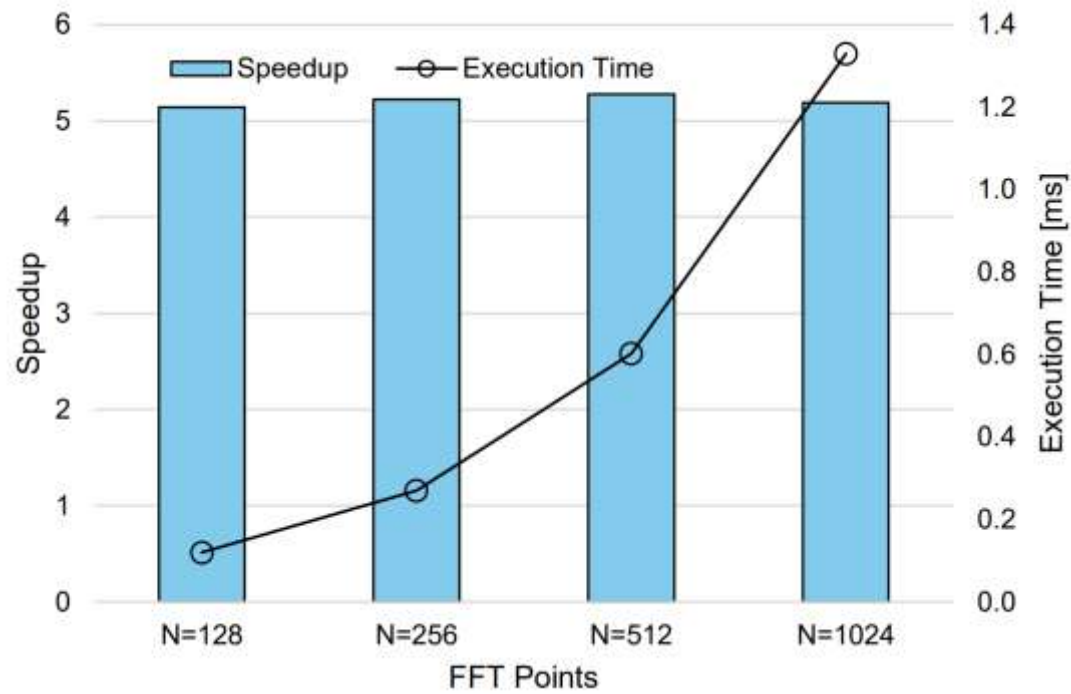
右はマイコンボード XIAO ESP32C3

中央、左は Cmod A7

<https://www.youtube.com/watch?v=19PorSnw1c4>

やわらかいハードウェアという特徴を持つ**FPGA**に**RISC-V**プロセッサとアクセラレータを実装して、アプリケーションに特化する高性能**SoC**を簡単に構築できるフレームワークを紹介します。
実用的な**FPGA**開発の最初の一步に最適です。

- Q16.16固定小数を用いた再帰を用いないバタフライ演算をCFUで実装
- 折れ線はCFUを用いる構成の実行時間、棒グラフは速度向上
- 本フレームワークのデフォルトの構成から、CFUを用いることで約5.2倍の高速化



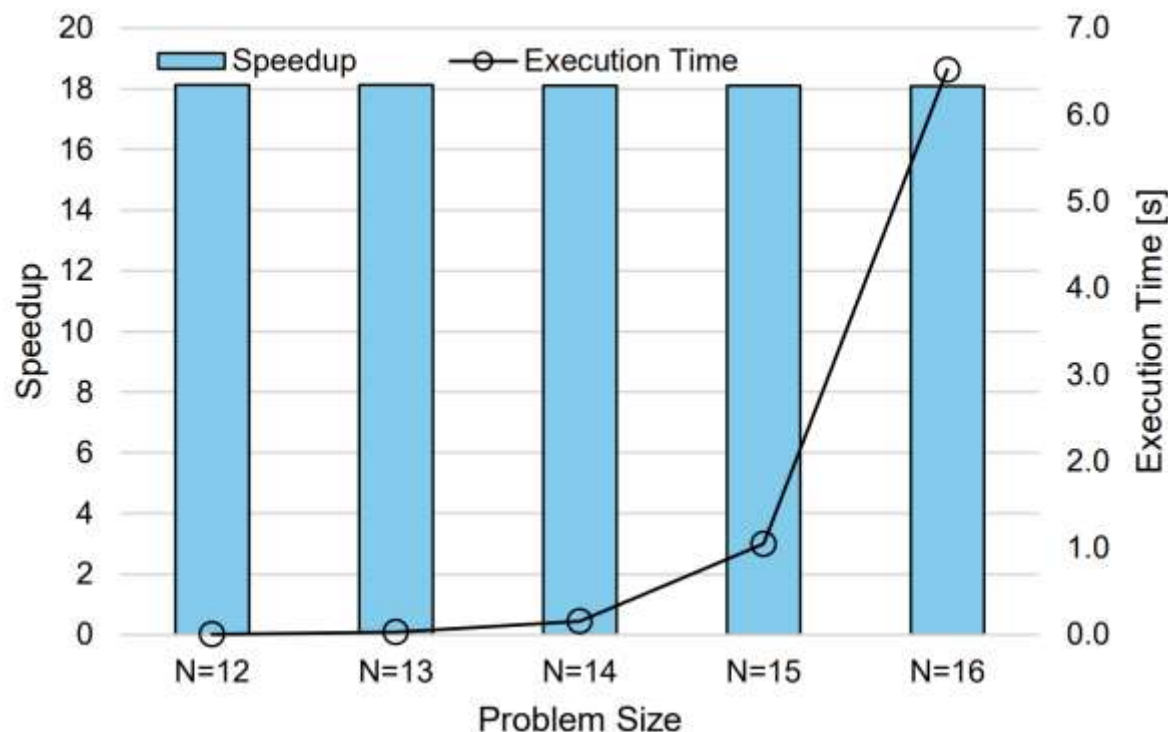
これが3命令に 😊

```
fix16_t temp_var1 = fix16_mul(f[2*idx_lower] , W_N[2*cnt_twiddle]);
fix16_t temp_var2 = fix16_mul(f[2*idx_lower+1], W_N[2*cnt_twiddle+1]);
fix16_t temp_var3 = fix16_mul(f[2*idx_lower] , W_N[2*cnt_twiddle+1]);
fix16_t temp_var4 = fix16_mul(f[2*idx_lower+1], W_N[2*cnt_twiddle]);
fix16_t temp_var1_2 = fix16_sub(temp_var1, temp_var2);
fix16_t temp_var3_4 = fix16_add(temp_var3, temp_var4);

fix16_t real = f[2*idx_upper];
fix16_t imag = f[2*idx_upper+1];

f[2*idx_upper] = fix16_add(real, temp_var1_2);
f[2*idx_upper+1] = fix16_add(imag, temp_var3_4);
f[2*idx_lower] = fix16_sub(real, temp_var1_2);
f[2*idx_lower+1] = fix16_sub(imag, temp_var3_4);
```

- 本フレームワークのデフォルトの構成から、カーネルをCFUとして実装することで約18倍の高速化



for(;;){
 if(r){
 int lsb = (-r) & r;
 a[h+1].cdt = (r & ~lsb);
 a[h+1].col = (a[h].col & ~lsb);
 a[h+1].pos = (a[h].pos | lsb) << 1;
 a[h+1].neg = (a[h].neg | lsb) >> 1;
 r = a[h+1].col & ~(a[h+1].pos | a[h+1].neg);
 h++;
 }
 else{
 r = a[h].cdt;
 h--;
 if(h==0) break;
 if(h==n) answers++;
 }
}
return answers;
}

これが1命令に 😊

Configuration Logic Block (CLB) of 7 Series FPGA

7 Series FPGAs Configurable Logic Block

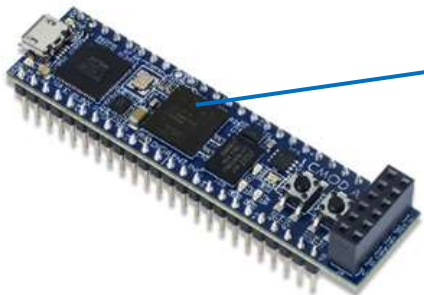
User Guide

Slices = SLICEL + SLICEM

Distributed RAM (bit) = SLICEM x 256

Table 1-2: Artix-7 FPGA CLB Resources

Device	Slices ⁽¹⁾	SLICEL	SLICEM	6-input LUTs	Distributed RAM (Kb)	Shift Register (Kb)	Flip-Flops
7A12T	2,000 ⁽²⁾	1,316	684	8,000	171	86	16,000
7A15T	2,600 ⁽²⁾	1,800	800	10,400	200	100	20,800
7A25T	3,650	2,400	1,250	14,600	313	156	29,200
7A35T	5,200 ⁽²⁾	3,600	1,600	20,800	400	200	41,600
7A50T	8,150	5,750	2,400	32,600	600	300	65,200
7A75T	11,800 ⁽²⁾	8,232	3,568	47,200	892	446	94,400
7A100T	15,850	11,100	4,750	63,400	1,188	594	126,800
7A200T	33,650	22,100	11,550	134,600	2,888	1,444	269,200



UG474 (v1.8) September 27, 2016

評価: FPGA内部のハードウェア資源の利用率

- デフォルト (32KiB, 16KiB)
 - 命令メモリ32KiB、データメモリ16KiBの構成

Resources	CFU Usage	Others	Total Usage	Available	Total Usage Rate
LUT	0	1771	1771	20800	8.51%
FF	0	872	872	41600	2.10%
BRAM	0	19	19	50	38.00%
DSP	0	4	4	90	4.44%

- N-Queens (8KiB, 4KiB)
 - 複雑な論理演算を行うため、LUTやFFが増加したと考えられる

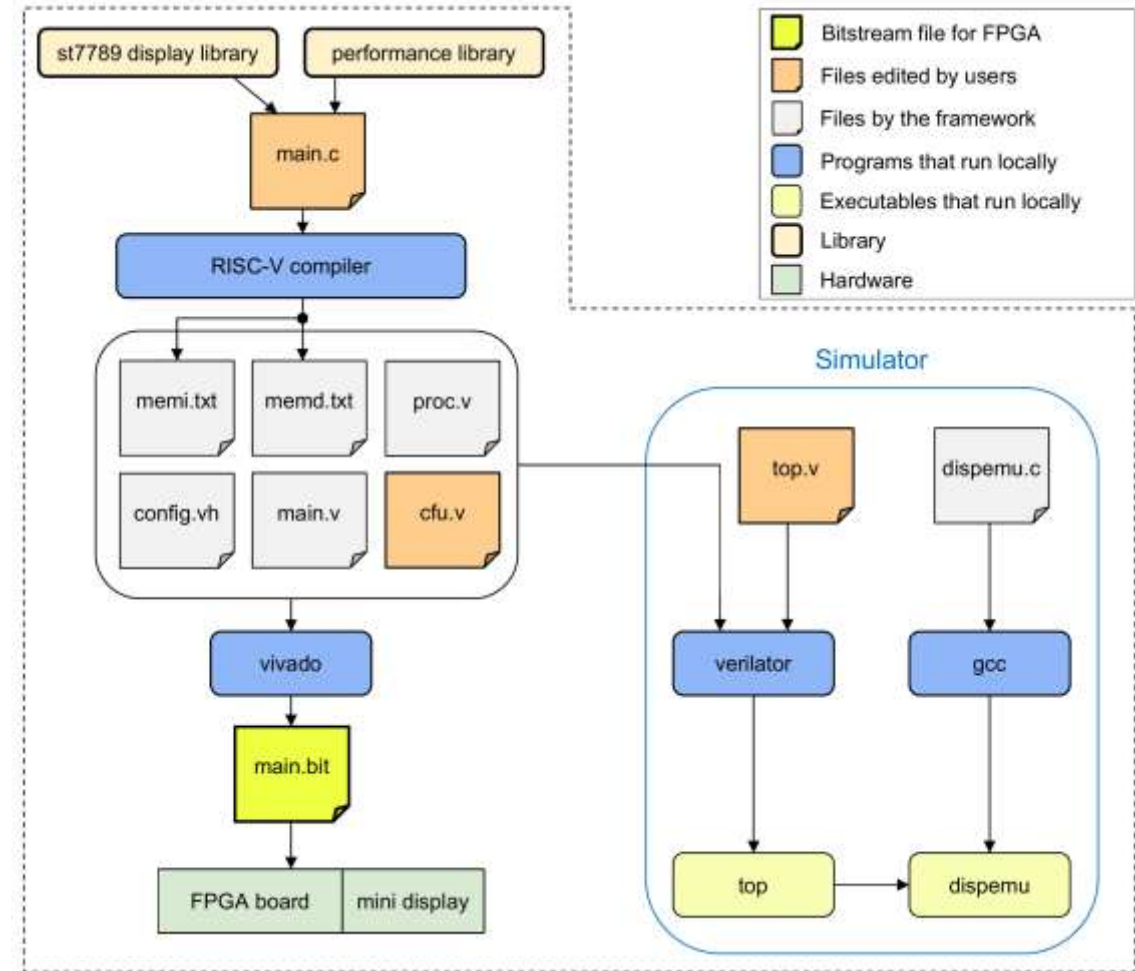
Resources	CFU Usage	Others	Total Usage	Available	Total Usage Rate
LUT	1153	1666	2819	20800	13.55%
FF	1279	1474	2753	41600	6.62%
BRAM	0	10	10	50	20%
DSP	0	4	4	90	4.44%

- FFT (8KiB, 32KiB)
 - 高速フーリエ変換では乗算を多用するため、DSPが増加している
 - 演算に必要な定数を保存するテーブルによりBRAMが増加

Resources	CFU Usage	Others	Total Usage	Available	Total Usage Rate
LUT	537	1699	2236	20800	10.75%
FF	598	910	1508	41600	3.63%
BRAM	0	23	23	50	46.00%
DSP	16	4	20	90	22.22%

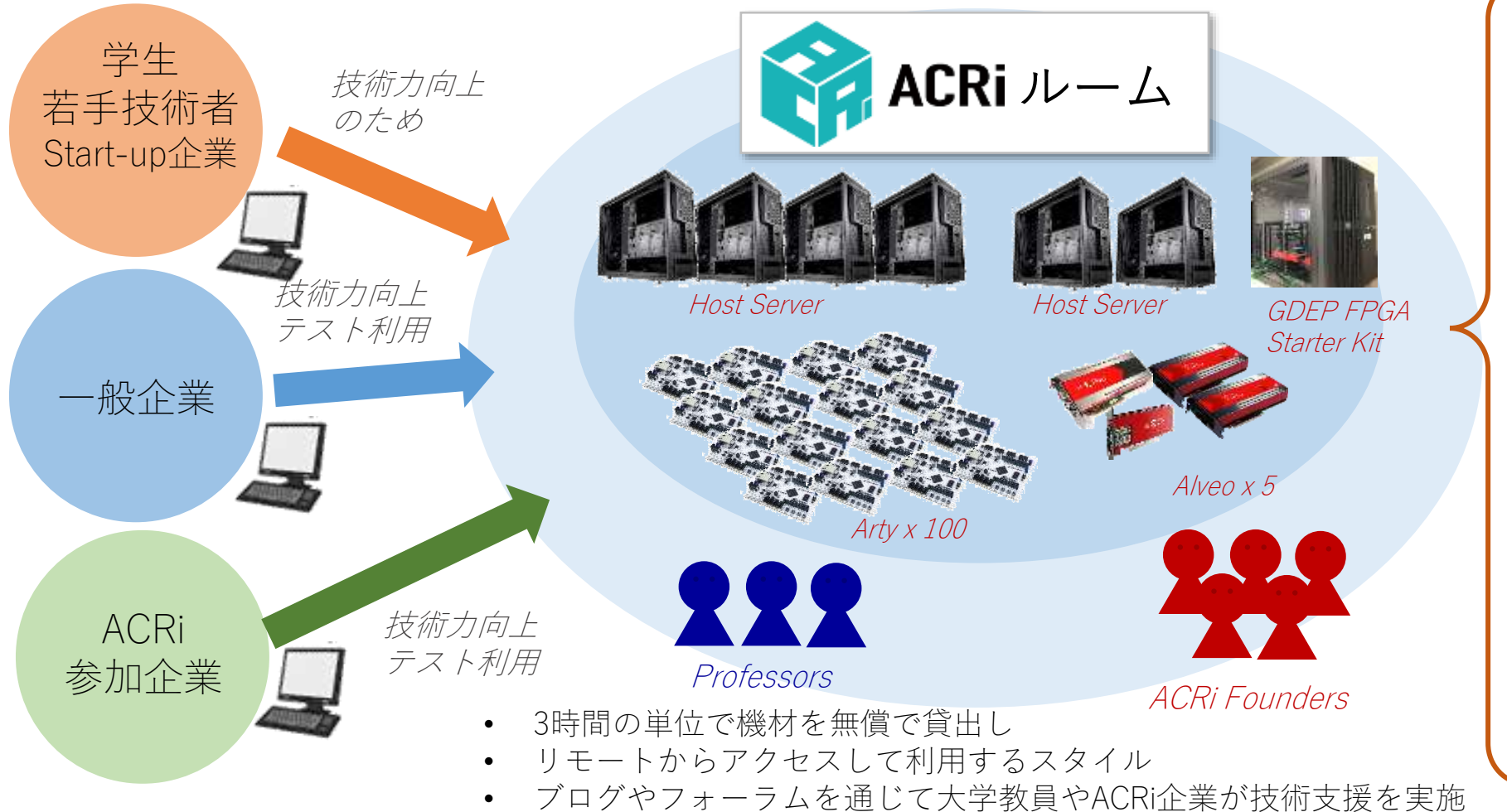
CFU Proving Ground の設計フローと設計ツール

- 必要となる設計ツール
 - RISC-V compiler (gcc)
 - クロスコンパイラの構築に約30分
 - Vivado
 - AMDが提供するフリーのソフトウェア
 - 約100GBのファイルのダウンロード、展開、インストールに約60分
 - Verilator
 - Ubuntu Linux などでは、apt で簡単にインストールできる
 - gcc, libcairo
 - Ubuntu Linux などでは、apt で簡単にインストールできる



日本初、産学連携でFPGA利用環境と学習機会を無償で提供

ACRiルームは、オンラインで無償利用できるFPGAの利用環境で、東京科学大学に設置された100枚を超えるFPGAボードと開発用ソフトウェアをリモートからアクセスして利用する。



FPGA Server



- CPU: Core i9 (8 core /16 thread)
- メモリ: DDR4 128GB (32GB x 4)
- ストレージ: SSD M.2 1TB x 2

Arty A7-35T



- 1サーバにArtyを15枚設置
- Digilent Arty A7-35T
- ユーザ毎にVMを割り当て
- 3時間でVMを再起動

Alveo



- 1サーバにAlveoを1枚設置
- Xilinx Alveo
U50 / U200 / U250 / U280

CFU Proving Ground の初期設定

Step (1) : Install the required software

If you are using [ACRi Room](#), these software are already installed and you can skip this step.

For RTL simulation, install Verilator version 5 if it is not installed. See the [Verilator website](#) for more information.

As an FPGA development tool, install Vivado 2024.2 if it is not installed. See the [Vivado site](#) for more information.

Our display simulator uses libcairo-deb package, please install it with the following command.

```
$ sudo apt -y install libcairo-dev
```

Please install the RISC-V compiler with the following commands. Installing the compiler can take a long time, so if you don't need it, just skip this.

```
$ sudo apt -y install autoconf automake autotools-dev curl python3 libmpc-dev
$ sudo apt -y install libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo
$ sudo apt -y install gperf libtool patchutils bc zlib1g-dev libexpat-dev
$ git clone https://github.com/riscv-collab/riscv-gnu-toolchain
$ cd riscv-gnu-toolchain
$ ./configure prefix=/tools/cad/riscv/rv32ima --with-arch=rv32ima --with-abi=ilp32
$ make
```

Step (2) : Clone the repository and edit the environment variables

Please clone this repository with the following command.

```
$ git clone https://github.com/archlab-sciencetokyo/CFU-Proving-Ground
```

Open and edit the `Makefile` on the CFU-Proving-Ground directory to specify proper absolute paths.

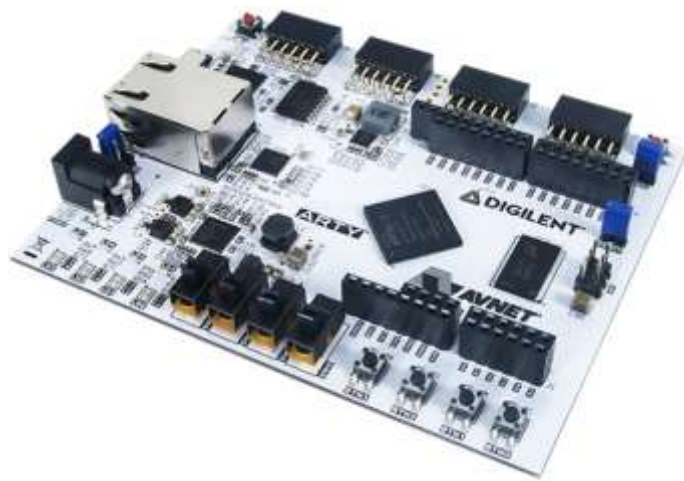
If you are using [ACRi Room](#), these paths are already set correctly and you do not need to change them.

variable	path to
GCC	riscv32-unknown-elf-gcc
GPP	riscv32-unknown-elf-g++
OBJCOPY	riscv32-unknown-elf-objcopy
OBJDUMP	riscv32-unknown-elf-objdump
VIVADO	vivado
RTL SIM	verilator

CFU Proving Ground の初期設定



Digilent Cmod A7



Digilent Arty A7



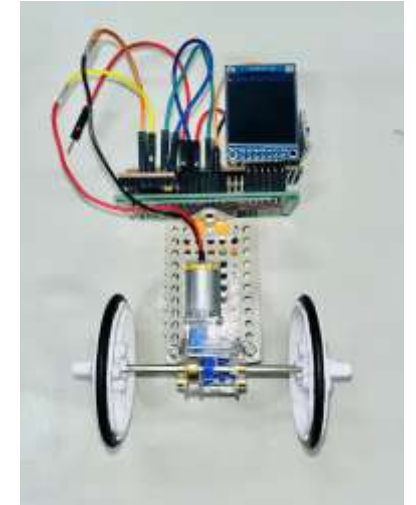
Digilent Nexys A7

```
1  # CFU Proving Ground since 2025-02   Copyright(c) 2025 Archlab. Science Tokyo
2  # Released under the MIT license https://opensource.org/licenses/mit
3
4  GCC      := /tools/cad/riscv/rv32ima/bin/riscv32-unknown-elf-gcc
5  GPP      := /tools/cad/riscv/rv32ima/bin/riscv32-unknown-elf-g++
6  OBJCOPY  := /tools/cad/riscv/rv32ima/bin/riscv32-unknown-elf-objcopy
7  OBJDUMP  := /tools/cad/riscv/rv32ima/bin/riscv32-unknown-elf-objdump
8  VIVADO    := /tools/Xilinx/Vivado/2024.1/bin/vivado
9  VPP      := /tools/Xilinx/Vitis/2024.1/bin/v++
10 RTLSIM    := /tools/cad/bin/verilator
11
12  TARGET := arty_a7
13  # TARGET := cmod_a7
14  # TARGET := nexys_a7
15
16  USE_HLS ?= 0
17
18  PART_arty_a7  := xc7a35tcsg324-1
19  PART_nexys_a7 := xc7a100tcsg324-1
20  PART_cmod_a7  := xc7a35ticpg236-1L
21
```

Makefile で利用するFPGAボードを指定

User-friendly two-wheeled self-balancing robot

- We propose **robbit**, a low-cost and user-friendly two wheeled self-balancing robot that uses an FPGA.
- We detail the design and implementation of robbit and demonstrate its stability and utility. Furthermore, we demonstrate the effectiveness of robbit by presenting its hardware costs, assembly time, and use cases.
- We release the source code of robbit **as open source**, and have a wide range of people using it as a learning material.



Hardware

- **FPGA board: Cmod A7-35T**
 - Size: 69.9mm × 18.8mm
 - Weight: 10.7g
 - **Sensor: MPU-6050(IMU)**
 - Accelerometer and gyroscope
 - **Motor: Mini motor standard gearbox 70188(TAMIYA)**
- **robbit**
- Price : **\$130.1**(FPGA board: \$104.9)
 - RISC-V processor
 - Cmod A7-35T(FPGA)



Cmod A7-35T



MPU-6050



Mini motor standard gearbox
70188(TAMIYA)

20

Hardware

- **Motor driver: TB6612FNG motor driver module**
- **Wheels: Slim tire set (55mm diameter) 70193**
- **Display: ST7789 IC 240 × 240 display**
- **Battery: EEMB Lithium-ion battery 653042**



TB6612FNG motor
driver module



Slim tire set
(55mm diameter)



ST7789 IC
240 × 240 display



Lithium-ion
battery 653042

21



robbit's photograph

- CFU Playground[1]
- Chipyard[2]
 - Rocket ChipやBOOMを用いたSoC自動生成フレームワーク
 - インターフェースに縛られないアクセラレータを扱うことが可能だが、複雑さが課題
- ASIP Designer[3]
 - ASIP設計用のソフトウェアツールチェーンの生成までをサポートするIP
 - オープンソースではないので機能に拡張性がない
- Open ASIP 2.0[4]
 - SDKの生成が可能な非常に高機能なフレームワーク
 - XMLベースのアーキテクチャ定義方式により習得が困難



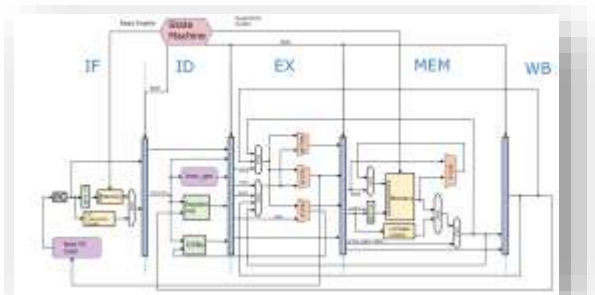
[1] S. Prakash et al., “CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs,” 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp.157–167, 2023.

[2] A. Amid et al., “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs,” IEEE Micro, vol.40, no.4, pp.10–21, 2020.

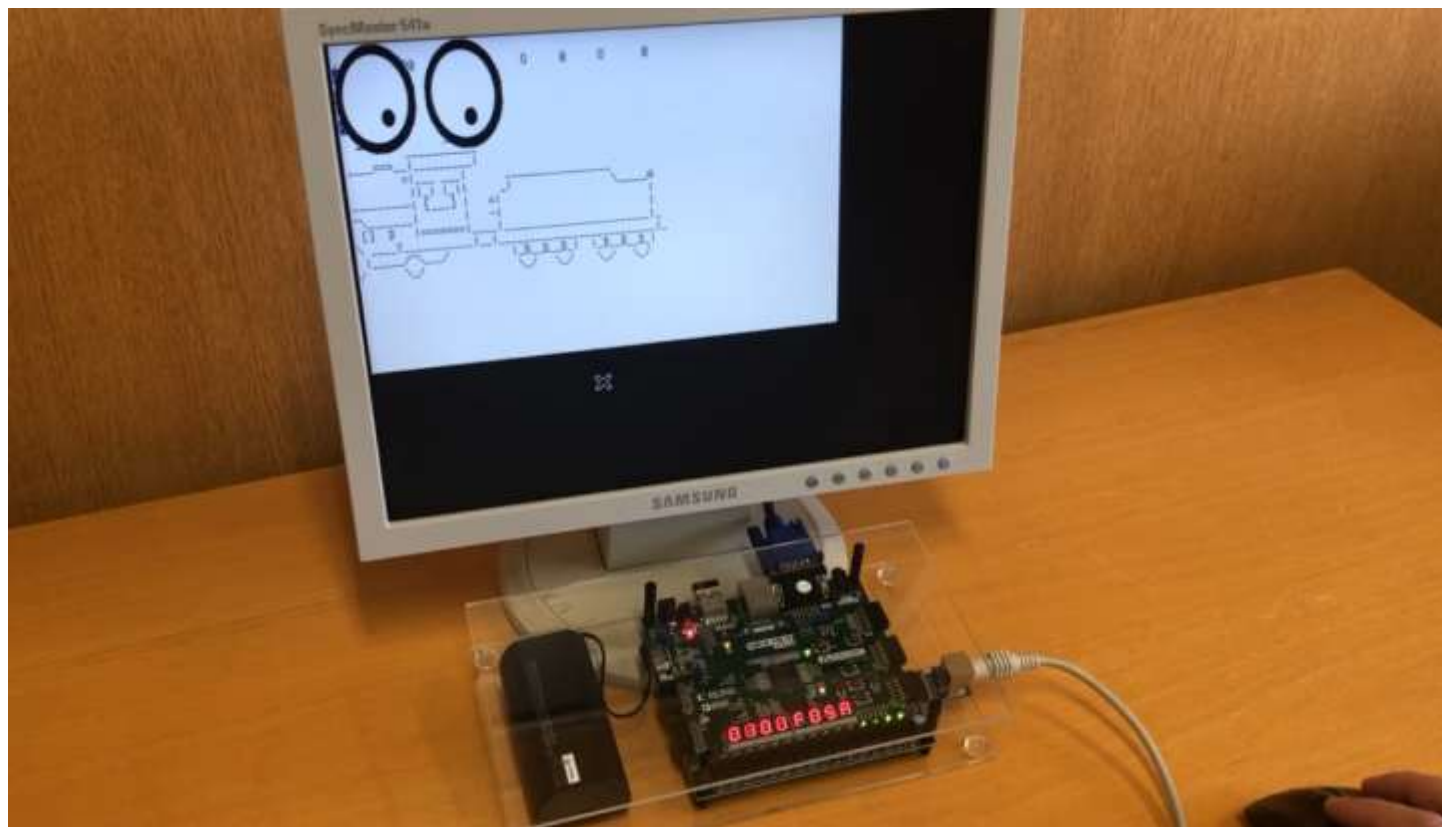
[3] Synopsys, Inc., “ASIP Designer,” <https://www.synopsys.com/dw/ipdir.php?ds=asip-designer>, 2024. Accessed: March 7, 2025.

[4] K. Hepola et al., “2022 IEEE 33rd International Conference on Applicationspecific Systems, Architectures and Processors (ASAP), pp.161–165, 2022.

- 入門用の FPGA ボードをターゲットに、スタンドアロンで動作するオープンソースで GUI（グラフィカル・ユーザーインターフェイス）対応の RISC-V システムの **RV-PC** という独自コンピューターを開発。
 - RV-PC** の多くのモジュールは **Verilog HDL** で記述され、メインモジュールは **RVCorePL** という5段パイプラインのソフトプロセッサ。RV-PC は、イーサネット、SD カード、VGA ディスプレイ、マウス、キーボードなど、幅広い I/O をサポートし、Linux や GUI アプリケーションを含む多数のソフトウェアを実行できる。



RVCorePL, RV-PCのための
5段パイプラインステージのプロセッサ

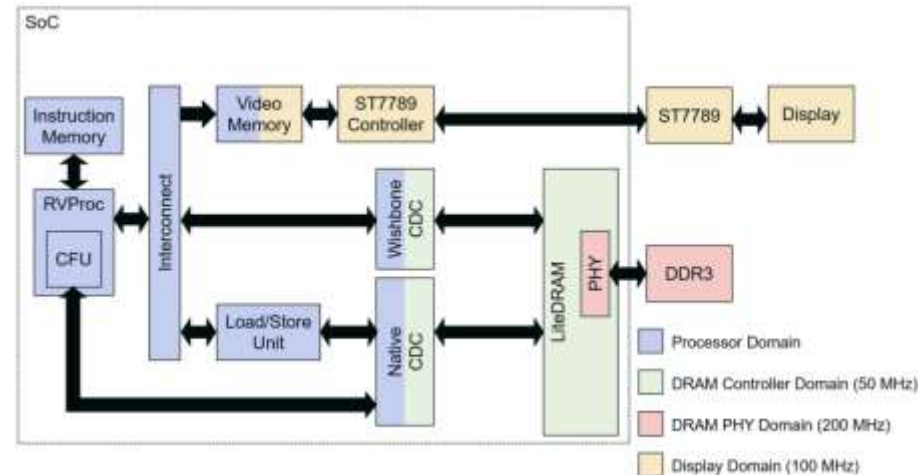


参考文献と今後の予定

- Aoba Fujino and Kenji Kise, "CFU Proving Ground: a Hardware/Software Co-Design Framework for Leveraging a Custom Function Unit and RISC-V Custom Instructions," 2025 IEEE 18th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc), pp. 199–206, Singapore, Dec. 2025.
- Daichi Kumagai, Komei Koderu, Yuya Iwata and Kenji Kise, "Robbit: A User-Friendly and Two-Wheeled Self-Balancing Robot Using an FPGA," 2025 IEEE 18th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc), pp. 780–787, Singapore, Dec. 2025.
- 藤野 蒼羽, 吉瀬 謙二, "RISC-V ソフトプロセッサでカスタム命令を簡単に扱うフレームワーク CFU Proving Ground の設計と実装", 電子情報通信学会技術研究報告, vol. 125, no. 66, RECONF2025-12, pp. 59–64, 2025 年 6 月.

imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

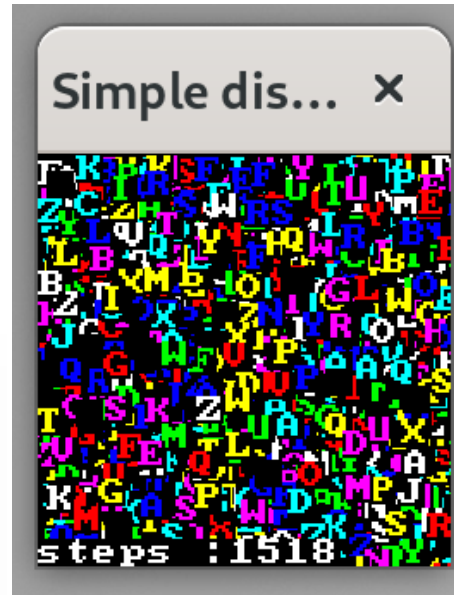
Fig 1.2 RV64I Base Instruction Set (Source: RV64I Base Instruction Set table in Volume I: Unprivileged ISA)



- ASIPを用いるシステム開発の開発時間の長期化が課題になっている。
- この課題の解決を目指して、CFU Proving Groundと呼ばれる、FPGAをターゲットするハードウェア／ソフトウェアの協調設計の開発環境を開発している。
- この環境では、CFU (Custom Function Unit) と呼ばれるアクセラレータを搭載するRISC-Vソフトプロセッサを用いて組み込みシステムの効率化を狙う。
- また、軽量なライブラリ、明確なファイル依存関係、柔軟なメモリマップを提供することでユーザービリティの向上を狙う。
- CFU Proving Groundの構成と使い方、ACRiルームでの動かし方、CFUを用いた高速化の事例を紹介した。

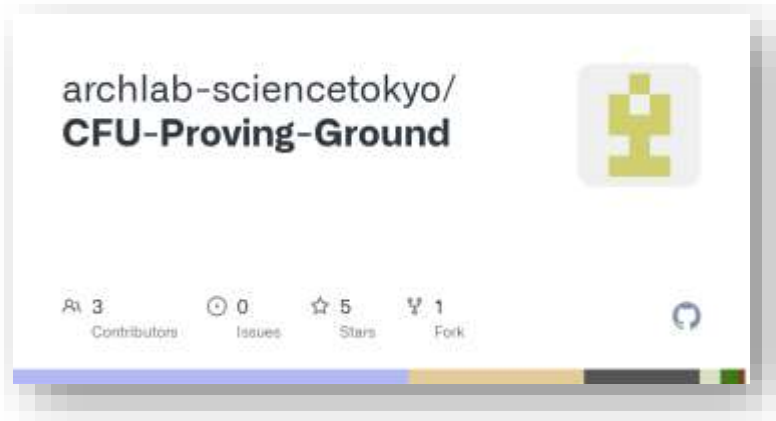
Try me, CFU PG 😊

```
$ git clone https://github.com/archlab-sciencetokyo/CFU-Proving-Ground
$ cd CFU-Proving-Ground
$ make
$ make drun
```



```
1  /* CFU Proving Ground since 2025-02    Copyright(c) 2025 Archlab. Science Tokyo /
2  / Released under the MIT license https://opensource.org/licenses/mit */
3
4  #include <stdlib.h>
5
6  #include "st7789.h"
7  #include "perf.h"
8  #include "util.h"
9
10 void RandomChar() {
11     int count = 0;
12     while (1) {
13         count++;
14         char c = 'A' + rand() % 26;
15         pg_lcd_draw_char(rand() % 240, rand() % 240, c, rand() & 0x7, 1);
16         pg_lcd_set_pos(0, 14);
17         pg_lcd_prints("steps :");
18         pg_lcd_printd(count);
19     }
20 }
21
22 int main () {
23     pg_lcd_reset();
24     RandomChar();
25     return 0;
26 }
```

main.c

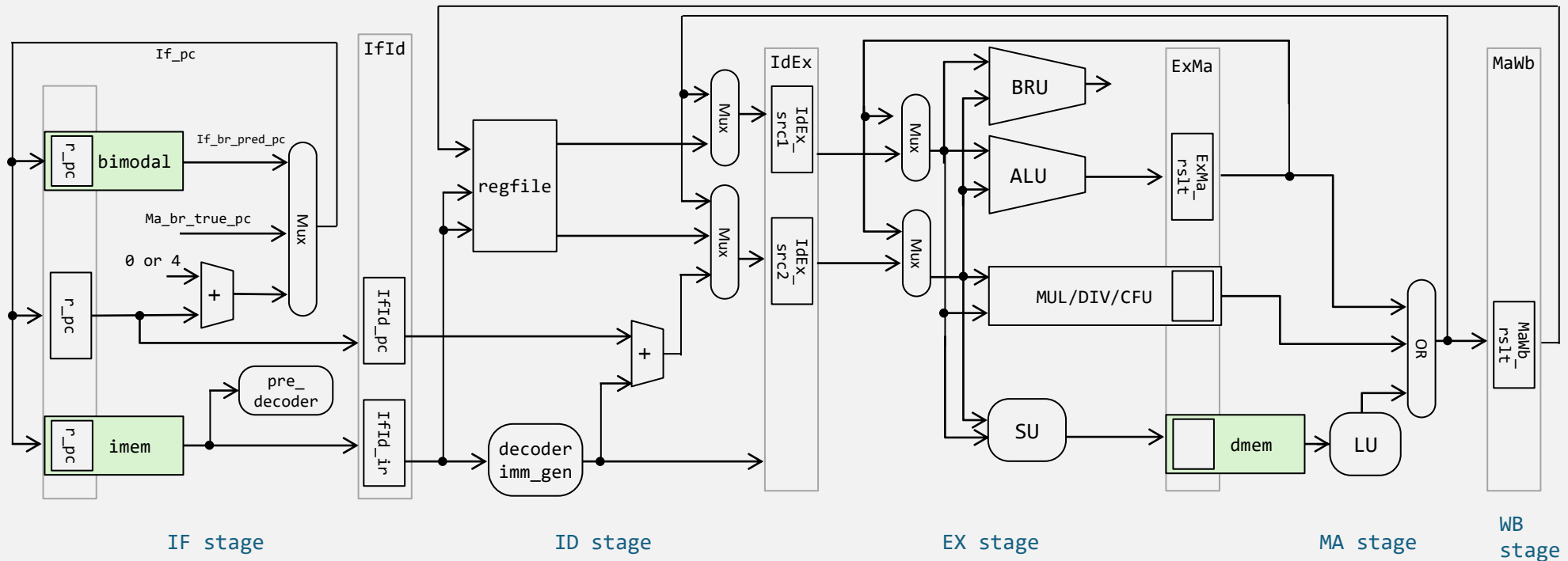


 <https://github.com/archlab-sciencetokyo/CFU-Proving-Ground>

ここからのスライドはプロセッサに関する補足資料です。

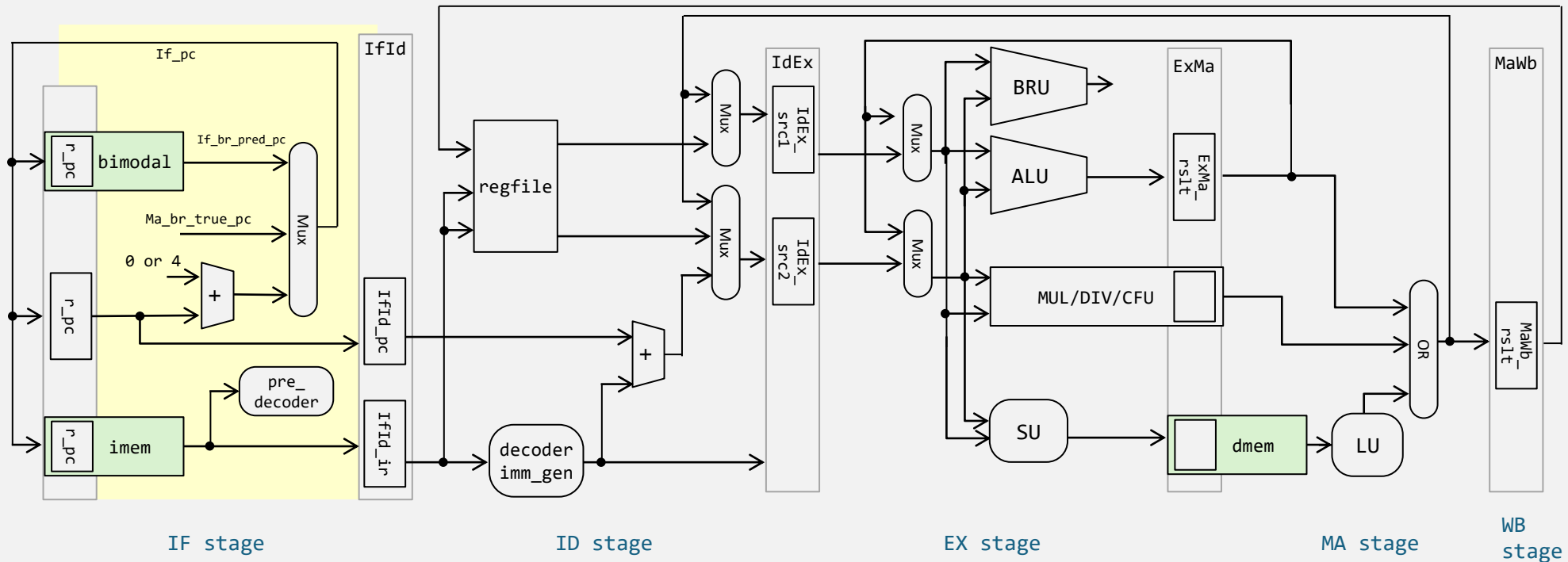
5-stage pipelining processor of CFU Proving Ground

- **IF (Instruction Fetch)** 命令メモリから命令をフェッチする。
- **ID (Instruction Decode)** 命令をデコード(解読)しながら、レジスタの値を読み出す。
- **EX (Execution)** 命令操作の実行またはアドレスの生成を行う。
- **MA (Memory Access)** 必要であれば、データメモリのオペランドにアクセスする。
- **WB (Write Back)** 必要であれば、結果をレジスタに書き込む。



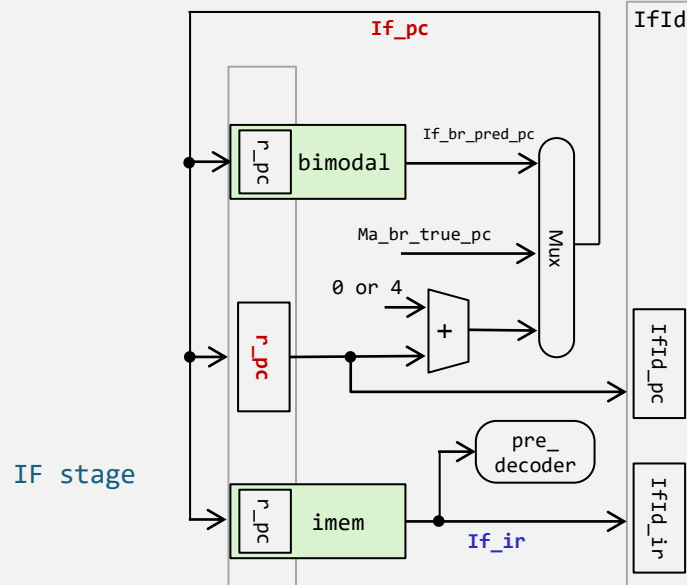
5-stage pipelining processor of CFU Proving Ground

- **IF (Instruction Fetch)** 命令メモリから命令をフェッチする。
- **ID (Instruction Decode)** 命令をデコード(解読)しながら、レジスタの値を読み出す。
- **EX (Execution)** 命令操作の実行またはアドレスの生成を行う。
- **MA (Memory Access)** 必要であれば、データメモリのオペランドにアクセスする。
- **WB (Write Back)** 必要であれば、結果をレジスタに書き込む。



IF (Instruction Fetch) stage

```
always @(posedge clk_i) if (!w_stall) begin
    r_pc <= If_pc; // update pc
    if (rst) begin
        IfId_v <= 0;
        IfId_pc <= 0;
        IfId_ir <= `NOP;
    end else if (!ExMa_stall) begin
        IfId_v <= If_v;
        IfId_load_muldiv_use <= If_load_muldiv_use;
        if (!IfId_load_muldiv_use) begin
            IfId_pc <= r_pc;
            IfId_ir <= If_ir;
            IfId_br_pred_tkn <= If_br_pred_tkn;
            IfId_pat_hist <= If_pat_hist;
            IfId_instr_type <= If_instr_type;
            IfId_rf_we <= If_rf_we;
            IfId_rd <= If_rd;
            IfId_rs1 <= If_rs1;
            IfId_rs2 <= If_rs2;
        end
    end
end
end
```



```
//-----
// pipeline registers
//-----
// IF: Instruction Fetch
reg [ `XLEN-1:0] r_pc; // program counter
```

```
//-----
// IF: Instruction Fetch
//-----
wire [ `PC_W-1:0] If_pc; // the program counter of the next clock cycle
wire [ `PC_W-1:0] If_pc_inc; //
wire If_pc_stall;
wire [1:0] If_pat_hist;
wire If_br_pred_tkn;
wire [31:0] If_br_pred_pc;
wire [ `ITYPE_W-1:0] If_instr_type;
wire If_rf_we;
wire [4:0] If_rd;
wire [4:0] If_rs1;
wire [4:0] If_rs2;
wire [31:0] If_ir; // instruction from imem
```

```
assign ibus_araddr_o = If_pc; // read address of imem
assign If_ir = ibus_rdata_i; // instruction from imem
```

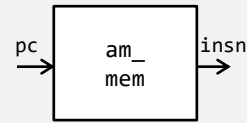
```
bimodal bimodal (
    .clk_i      (clk_i),           // input wire
    .rst_i      (rst),            // input wire
    .stall_i    (w_stall),        // input wire
    .raddr_i    (If_pc),          // input wire [ `XLEN-1:0]
    .pat_hist_o (If_pat_hist),    // output reg [1:0]
    .br_pred_tkn_o (If_br_pred_tkn), // output wire
    .br_pred_pc_o (If_br_pred_pc), // output reg [ `XLEN-1:0]
    .br_tkn_i   (Ma_br_tkn),      // input wire
    .br_tsfr_i  (ExMa_j_b_insn),  // input wire
    .waddr_i    (ExMa_pc),        // input wire [ `XLEN-1:0]
    .pat_hist_i (ExMa_pat_hist),  // input wire [1:0]
    .br_tkn_pc_i (ExMa_br_tkn_pc) // input wire [ `XLEN-1:0]
);
```

```
assign If_pc_stall = ExMa_stall || IfId_load_muldiv_use;
assign If_pc_inc = (If_pc_stall) ? 0 : 4;
assign If_pc = (w_stall) ? r_pc :
    (Ma_br_misp) ? Ma_br_true_pc :
    (!If_pc_stall & If_br_pred_tkn) ? If_br_pred_pc : r_pc + If_pc_inc;
```

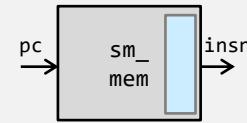
同期メモリ、非同期メモリ

```
module am_mem (  
    input wire clk,  
    input wire [`ADDRW+1:0] addr_i,  
    output wire [31:0] data_o  
);  
    reg [31:0] mem[0:`MEM_ENTRIES-1];  
  
    assign data_o = mem[addr_i];  
endmodule
```

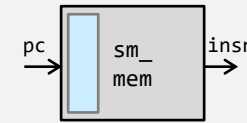
(a) 非同期メモリ



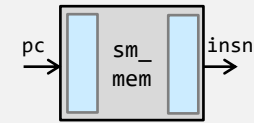
(a)



(b)



(c)



(d)

```
module sm_mem (  
    input wire clk,  
    input wire [`ADDRW+1:0] addr_i,  
    output wire [31:0] data_o  
);  
    reg [31:0] mem[0:`MEM_ENTRIES-1];  
  
    reg [31:0] rdata;  
    always @(posedge clk) begin  
        r_data <= mem[addr_i];  
    end  
    assign data_o = r_data;  
endmodule
```

(b) 同期メモリ

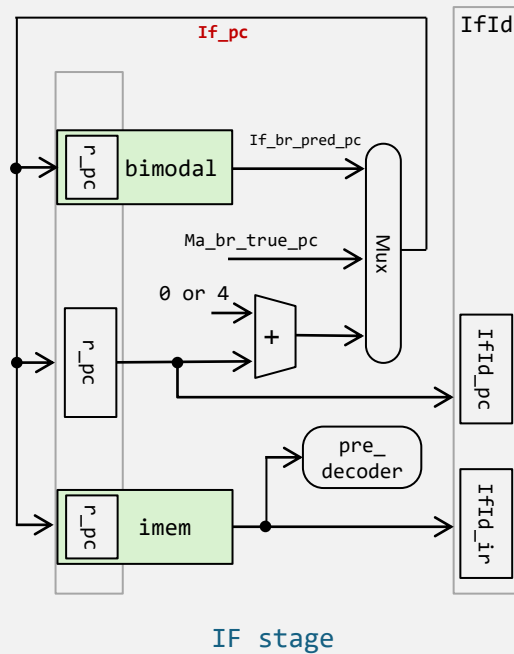
```
module sm_mem (  
    input wire clk,  
    input wire [`ADDRW+1:0] addr_i,  
    output wire [31:0] data_o  
);  
    reg [31:0] mem[0:`MEM_ENTRIES-1];  
  
    reg [`ADDRW+1:0] r_addr;  
    always @(posedge clk) begin  
        r_addr <= addr_i;  
    end  
    assign data_o = mem[r_addr];  
endmodule
```

(c) 同期メモリ

```
module sm_mem (  
    input wire clk,  
    input wire [`ADDRW+1:0] addr_i,  
    output wire [31:0] data_o  
);  
    reg [31:0] mem[0:`MEM_ENTRIES-1];  
  
    reg [`ADDRW+1:0] r_addr;  
    reg [31:0] r_data;  
    always @(posedge clk) begin  
        r_addr <= addr_i;  
        r_data <= mem[r_addr];  
    end  
    assign data_o = r_data;  
endmodule
```

(d) 同期メモリ

IF (Instruction Fetch) stage



```

cpu cpu (
    .clk_i      (clk),          // input wire
    .rst_i      (rst),          // input wire
    .stall_i     (0),           // input wire
    .ibus_araddr_o (imem_raddr), // output wire [`IBUS_ADDR_WIDTH-1:0]
    .ibus_rdata_i (imem_rdata), // input wire  [`IBUS_DATA_WIDTH-1:0]
    .dbus_addr_o  (dbus_addr),  // output wire  [`DBUS_ADDR_WIDTH-1:0]
    .dbus_wvalid_o (dbus_we),   // output wire
    .dbus_wdata_o (dbus_wdata), // output wire  [`DBUS_DATA_WIDTH-1:0]
    .dbus_wstrb_o (dbus_wstrb), // output wire  [`DBUS_STRB_WIDTH-1:0]
    .dbus_rdata_i (dbus_rdata)  // input wire  [`DBUS_DATA_WIDTH-1:0]
);

m_imem imem (
    .clk_i (clk),          // input wire
    .raddr_i (imem_raddr), // input wire [ADDR_WIDTH-1:0]
    .rdata_o (imem_rdata)  // output reg  [DATA_WIDTH-1:0]
);
    
```

```

module m_imem (
    input wire      clk_i,
    input wire [31:0] raddr_i,
    output wire [31:0] rdata_o
);

(* ram_style = "block" *) reg [31:0] imem[0:`IMEM_ENTRIES-1];
`include "memi.txt"

wire [`IMEM_ADDRW-1:0] valid_raddr = raddr_i[`IMEM_ADDRW+1:2];

reg [31:0] rdata = 0;
always @(posedge clk_i) begin
    rdata <= imem[valid_raddr];
end
assign rdata_o = rdata;
endmodule
    
```

シンプルなアセンブリ言語のコードのシミュレーション

```
reg r_rst = 0;
always @(posedge clk) r_rst <= !m0.rst;
always @(posedge clk) if (r_rst) begin
    $write(" %06d: %x ", minstret, m0.cpu.r_pc);
    if (!m0.cpu.IfId_v) $write("----- "); else if(m0.cpu.stall) $write("ssssssss "); else $write("%x ", m0.cpu.IfId_pc);
    if (!m0.cpu.IdEx_v) $write("----- "); else if(m0.cpu.stall) $write("ssssssss "); else $write("%x ", m0.cpu.IdEx_pc);
    if (!m0.cpu.ExMa_v) $write("----- "); else if(m0.cpu.stall) $write("ssssssss "); else $write("%x ", m0.cpu.ExMa_pc);
    if (!m0.cpu.MaWb_v) $write("-----"); else if(m0.cpu.stall) $write("ssssssss"); else $write("%x:", m0.cpu.MaWb_pc);
    if (m0.cpu.stall_i) $write(" stall");
    $write("\n");
end
```

- top.v を編集して、上図のコードを追加する。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行

- make
- make run

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00020000
.align 4
.section .text.init
.globl _start
_start:
    li    x1, 2
    li    x2, 0
    li    t0, TO_HOST
    li    t1, CMD_FINISH
    sw    t1, 0(t0)
L1: j     L1
```

```
./obj_dir/top
000000: 00000000 -----
000000: 00000004 00000000 -----
000000: 00000008 00000004 00000000 -----
000000: 0000000c 00000008 00000004 00000000 -----
000001: 00000010 0000000c 00000008 00000004 00000000:
000002: 00000014 00000010 0000000c 00000008 00000004:
000003: 00000018 00000014 00000010 0000000c 00000008:
000004: 0000001c 00000018 00000014 00000010 0000000c:
- top.v:47: Verilog $finish

===> mcycle           :      8
===> minstret         :      4
===> Total number of branch predictions :      0
===> Total number of branch mispredictions :      0
===> simulation finish!!

- Simulation Report: Verilator 5.026 2024-06-15
- Verilator: $finish at 105ps; walltime 0.000 s; speed 0.000 s/s
- Verilator: cpu 0.000 s on 1 threads; allocated 121 MB
```



シンプルなアセンブリ言語のコードのシミュレーション

- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
 - make
 - make run

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00020000
.align 4
.section .text.init
.globl _start
_start:
    li    x1, 2
    li    x2, 0
L2: addi   x2, x2, 1
    bne   x1, x2, L2
    li    t0, TO_HOST
    li    t1, CMD_FINISH
    sw    t1, 0(t0)
L1: j     L1
```

```
./obj_dir/top
000000: 00000000 -----
000000: 00000004 00000000 -----
000000: 00000008 00000004 00000000 -----
000000: 0000000c 00000008 00000004 00000000 -----
000001: 00000010 0000000c 00000008 00000004 00000000:
000002: 00000014 00000010 0000000c 00000008 00000004:
000003: 00000018 00000014 00000010 0000000c 00000008:
000004: 00000008 ----- 0000000c:
000004: 0000000c 00000008 -----
000004: 00000010 0000000c 00000008 -----
000004: 00000014 00000010 0000000c 00000008 -----
000005: 00000018 00000014 00000010 0000000c 00000008:
000006: 0000001c 00000018 00000014 00000010 0000000c:
000007: 00000020 0000001c 00000018 00000014 00000010:
000008: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

==> mcycle : 15
==> minstret : 8
==> Total number of branch predictions : 2
==> Total number of branch mispredictions : 1
==> simulation finish!!
```

```
assign If_pc_stall = ExMa_stall || IfId_load_muldiv_use;
assign If_pc_inc = (If_pc_stall) ? 0 : 4;
assign If_pc = (w_stall) ? r_pc :
    (Ma_br_misp) ? Ma_br_true_pc :
    (!If_pc_stall & If_br_pred_tkn) ? If_br_pred_pc :
    r_pc+If_pc_inc;
```



シンプルなアセンブリ言語のコードのシミュレーション

- `app/crt0.s` を下図のコードに編集し、実行する。

- `make; make run`

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00020000
.align 4
.section .text.init
.globl _start
_start:
    li    x1, 8
    li    x2, 0
L2: addi  x2, x2, 1
    bne   x1, x2, L2
    li    t0, TO_HOST
    li    t1, CMD_FINISH
    sw    t1, 0(t0)
L1: j     L1
```

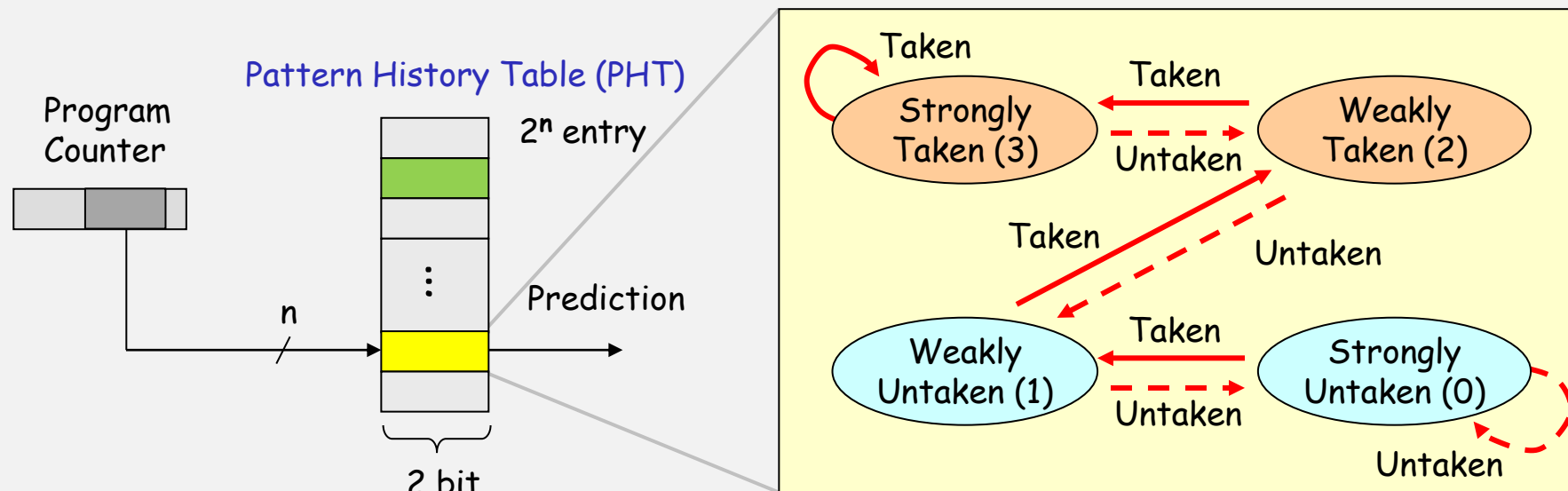
```
bimodal bimodal (
    .clk_i      (clk_i),          // input  wire
    .rst_i      (rst),            // input  wire
    .stall_i     (w_stall),        // input  wire
    .raddr_i     (If_pc),          // input  wire [XLEN-1:0]
    .pat_hist_o  (If_pat_hist),    // output  reg      [1:0]
    .br_pred_tkn_o (If_br_pred_tkn), // output  wire
    .br_pred_pc_o (If_br_pred_pc), // output  reg [XLEN-1:0]
    .br_tkn_i    (Ma_br_tkn),      // input  wire
    .br_tsfr_i   (ExMa_j_b_insn),  // input  wire
    .waddr_i     (ExMa_pc),        // input  wire [XLEN-1:0]
    .pat_hist_i  (ExMa_pat_hist),  // input  wire      [1:0]
    .br_tkn_pc_i (ExMa_br_tkn_pc) // input  wire [XLEN-1:0]
);
```

```
./obj_dir/top
000000: 00000000 -----
000000: 00000004 00000000 -----
000000: 00000008 00000004 00000000 -----
000000: 0000000c 00000008 00000004 00000000 -----
000001: 00000010 0000000c 00000008 00000004 00000000:
000002: 00000014 00000010 0000000c 00000008 00000004:
000003: 00000018 00000014 00000010 0000000c 00000008:
000004: 00000008 ----- 0000000c
000004: 00000010 0000000c 00000008 -----
000004: 00000014 00000010 0000000c 00000008 -----
000005: 00000018 00000014 00000010 0000000c 00000008:
000006: 00000008 ----- 0000000c
000006: 0000000c 00000008 -----
000006: 00000008 0000000c 00000008 -----
000006: 0000000c 00000008 0000000c 00000008 -----
000007: 00000008 0000000c 00000008 0000000c 00000008:
000008: 0000000c 00000008 0000000c 00000008 0000000c:
000009: 00000008 0000000c 00000008 0000000c 00000008:
000010: 0000000c 00000008 0000000c 00000008 0000000c:
000011: 00000008 0000000c 00000008 0000000c 00000008:
000012: 0000000c 00000008 0000000c 00000008 0000000c:
000013: 00000008 0000000c 00000008 0000000c 00000008:
000014: 0000000c 00000008 0000000c 00000008 0000000c:
000015: 00000008 0000000c 00000008 0000000c 00000008:
000016: 0000000c 00000008 0000000c 00000008 0000000c:
000017: 00000008 0000000c 00000008 0000000c 00000008:
000018: 00000010 ----- 0000000c
000018: 00000014 00000010 -----
000018: 00000018 00000014 00000010 -----
000018: 0000001c 00000018 00000014 00000010 -----
000019: 00000020 0000001c 00000018 00000014 00000010:
000020: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

==> mcycle : 33
==> minstret : 20
==> Total number of branch predictions : 8
==> Total number of branch mispredictions : 3
==> simulation finish!!
```

Simple branch predictor: **bimodal**

- Program has many **static** branch instructions. The behavior may depend on each branch. **Use plenty of counters (PHT) and assign a counter for a branch instruction.**
- How to predict
 - Select a 2-bit counter **using PC**, and it predicts 1 for taken if the MSB of the register is one; otherwise, it predicts 0 for untaken.
- How to update
 - Select a counter using PC, then update the counter in the same way as 2-bit counter.



CFU PG の bimodal branch predictor

```
module bimodal (
    input wire      clk_i,
    input wire      rst_i,
    input wire      stall_i,
    input wire [31:0] raddr_i,
    output wire [1:0] pat_hist_o,
    output wire      br_pred_tkn_o,
    output wire [`PC_W-1:0] br_pred_pc_o,
    input wire      br_tkn_i,
    input wire      br_tsfr_i,
    input wire [31:0] waddr_i,
    input wire [1:0] pat_hist_i,
    input wire [31:0] br_tkn_pc_i
);

integer i;
(* ram_style = "block" *) reg [`PC_W-1:0] btb[0:`BTB_ENTRY-1]; // BTB, branch target buf
initial for (i = 0; i < `BTB_ENTRY; i = i + 1) btb[i] = 0; // init with weak untaken

wire [1:0] w_cnt = (br_tkn_i) ? pat_hist_i + (pat_hist_i < 3) : pat_hist_i - (pat_hist_i > 0);
wire [`BTB_IDXW-1:0] btb_ridx = raddr_i[`BTB_IDXW+`BTB_OSTW-1:`BTB_OSTW];
wire [`BTB_IDXW-1:0] btb_widx = waddr_i[`BTB_IDXW+`BTB_OSTW-1:`BTB_OSTW];

reg [31:0] r_btb_entry;
always @(posedge clk_i) if (!stall_i) begin
    r_btb_entry <= btb[btb_ridx];
    if (br_tsfr_i) begin
        btb[btb_widx] <= {br_tkn_pc_i[`PC_W-1:2], w_cnt}; // lower tow bits for counter
    end
end

assign pat_hist_o    = r_btb_entry[1:0];
assign br_pred_tkn_o = r_btb_entry[1];
assign br_pred_pc_o  = {r_btb_entry[`PC_W-1:2], 2'b0};
endmodule
```



IF (Instruction Fetch) stage

```
always @(posedge clk_i) if (!w_stall) begin
    r_pc <= If_pc; // update pc
    if (rst) begin
        IfId_v <= 0;
        IfId_pc <= 0;
        IfId_ir <= `NOP;
    end else if (!ExMa_stall) begin
        IfId_v <= If_v;
        IfId_load_muldiv_use <= If_load_muldiv_use;
        if (!IfId_load_muldiv_use) begin
            IfId_pc <= r_pc;
            IfId_ir <= If_ir;
            IfId_br_pred_tkn <= If_br_pred_tkn;
            IfId_pat_hist <= If_pat_hist;
            IfId_instr_type <= If_instr_type;
            IfId_rf_we <= If_rf_we;
            IfId_rd <= If_rd;
            IfId_rs1 <= If_rs1;
            IfId_rs2 <= If_rs2;
        end
    end
end
```

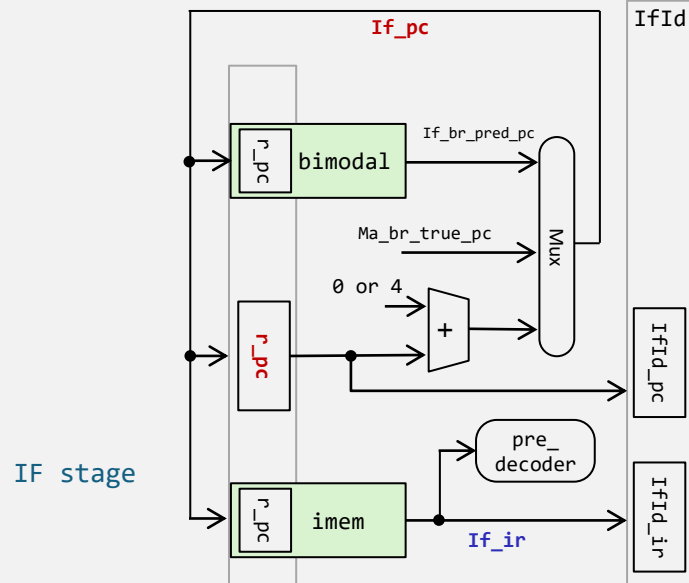
```
pre_decoder pre_decoder (
    .ir_i      (If_ir),           // input wire      [31:0]
    .instr_type_o(If_instr_type), // output wire [ `ITYPE_W-1:0]
    .rf_we_o   (If_rf_we),       // output wire
    .rd_o      (If_rd),          // output wire      [4:0]
    .rs1_o     (If_rs1),         // output wire      [4:0]
    .rs2_o     (If_rs2),         // output wire      [4:0]
);

wire If_load_muldiv_use = IfId_v && !Ma_br_misp && !IfId_load_muldiv_use
    && (Id_lsu_ctrl[ `LSU_CTRL_IS_LOAD ] ||
        Id_mul_ctrl[ `MUL_CTRL_IS_MUL ] ||
        Id_div_ctrl[ `DIV_CTRL_IS_DIV ] ||
        Id_cfu_ctrl[ `CFU_CTRL_IS_CFU ] )
    && IfId_rf_we && ((IfId_rd==If_rs1) || (IfId_rd==If_rs2));
```

```
module pre_decoder (
    input wire [31:0] ir_i,
    output wire [ 2:0] instr_type_o,
    output wire      rf_we_o,
    output wire [ 4:0] rd_o,
    output wire [ 4:0] rs1_o,
    output wire [ 4:0] rs2_o
);

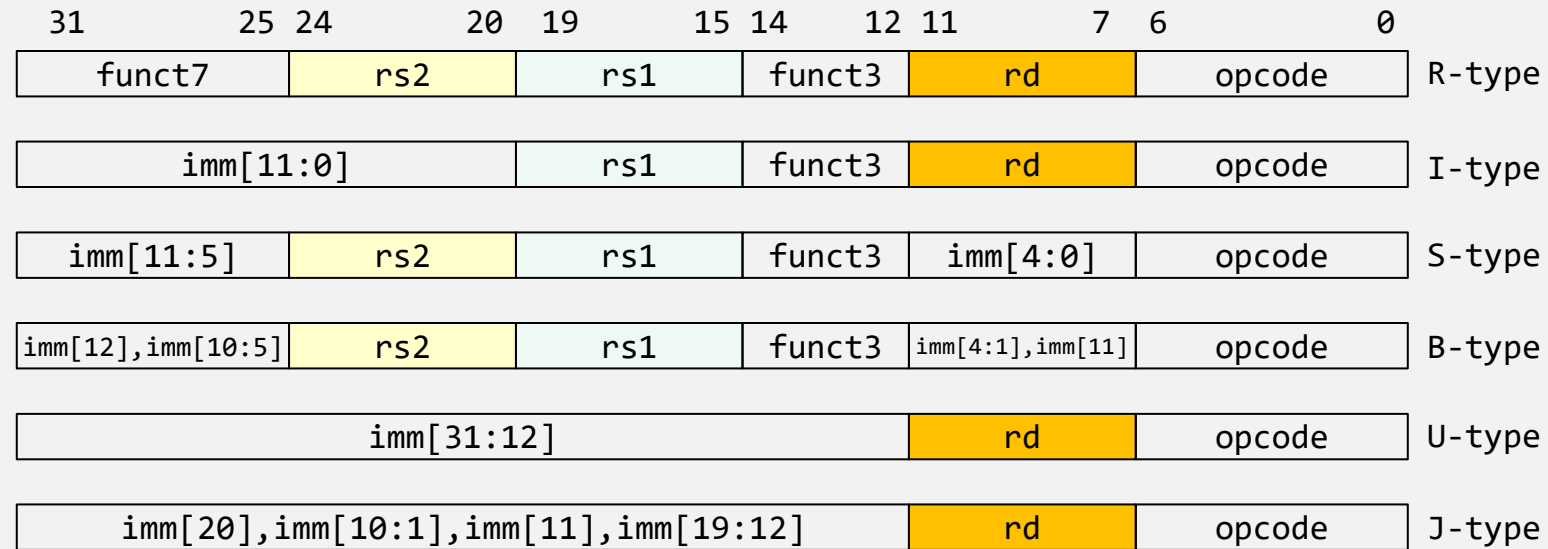
    wire [4:0] opcode = ir_i[6:2];
    assign instr_type_o = (opcode == 5'b01101) ? `U_TYPE : // LUI
        (opcode == 5'b00101) ? `U_TYPE : // AUIPC
        (opcode == 5'b11011) ? `J_TYPE : // JAL
        (opcode == 5'b11001) ? `I_TYPE : // JALR
        (opcode == 5'b11000) ? `B_TYPE : // BRANCH
        (opcode == 5'b00000) ? `I_TYPE : // LOAD
        (opcode == 5'b01000) ? `S_TYPE : // STORE
        (opcode == 5'b00100) ? `I_TYPE : // OP-IMM
        (opcode == 5'b01100) ? `R_TYPE : // OP
        (opcode == 5'b00010) ? `R_TYPE : `NONE_TYPE; // CUSTOM-0 : NONE

    assign rd_o = ((instr_type_o == `S_TYPE) | (instr_type_o == `B_TYPE)) ? 0 : ir_i[11:7];
    assign rs1_o = ((instr_type_o == `U_TYPE) | (instr_type_o == `J_TYPE)) ? 0 : ir_i[19:15];
    assign rs2_o = ((instr_type_o == `I_TYPE) |
        (instr_type_o == `U_TYPE) | (instr_type_o == `J_TYPE)) ? 0 : ir_i[24:20];
    assign rf_we_o = (rd_o != 0);
endmodule
```



IF stage

RISC-V RV32I instruction format



opcode5[2:0]	000	001	010	011	100	101	110
opcode5[4:3]							
00	LOAD (I-type)	LOAD-FP	custom-0	MISC-MEM (I-type)	OP-IMM (I-type)	AUIPC (U-type)	OP-IMM-32
01	STORE (S-type)	STORE-FP	custom-1	AMO	OP (R-type)	LUI (U-type)	OP-32
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	custom-2/ rv128
11	BRANCH (B-type)	JALR (I-type)	reserved	JAL (J-type)	SYSTEM (I-type)	reserved	custom-3/ rv128

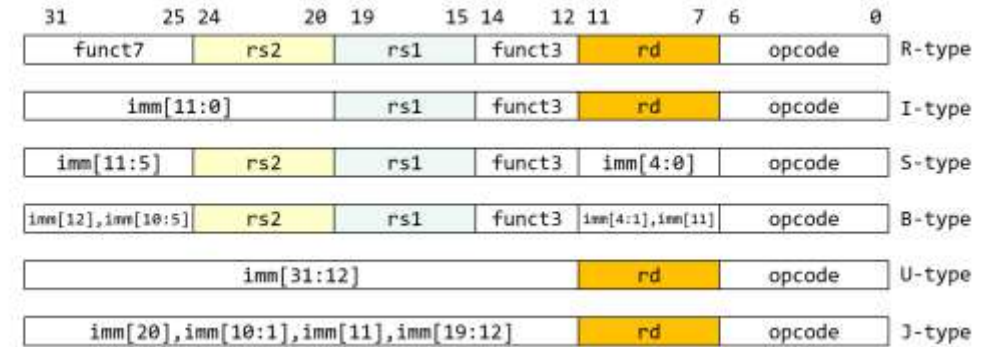
RISC-V RV32I instruction format



```
module pre_decoder (
    input wire [31:0] ir_i,
    output wire [ 2:0] instr_type_o,
    output wire      rf_we_o,
    output wire [ 4:0] rd_o,
    output wire [ 4:0] rs1_o,
    output wire [ 4:0] rs2_o
);

wire [4:0] opcode = ir_i[6:2];
assign instr_type_o = (opcode == 5'b01101) ? `U_TYPE : // LUI
    (opcode == 5'b00101) ? `U_TYPE : // AUIPC
    (opcode == 5'b11011) ? `J_TYPE : // JAL
    (opcode == 5'b11001) ? `I_TYPE : // JALR
    (opcode == 5'b11000) ? `B_TYPE : // BRANCH
    (opcode == 5'b00000) ? `I_TYPE : // LOAD
    (opcode == 5'b01000) ? `S_TYPE : // STORE
    (opcode == 5'b00100) ? `I_TYPE : // OP-IMM
    (opcode == 5'b01100) ? `R_TYPE : // OP
    (opcode == 5'b00010) ? `R_TYPE : `NONE_TYPE; // CUSTOM-0 : NONE

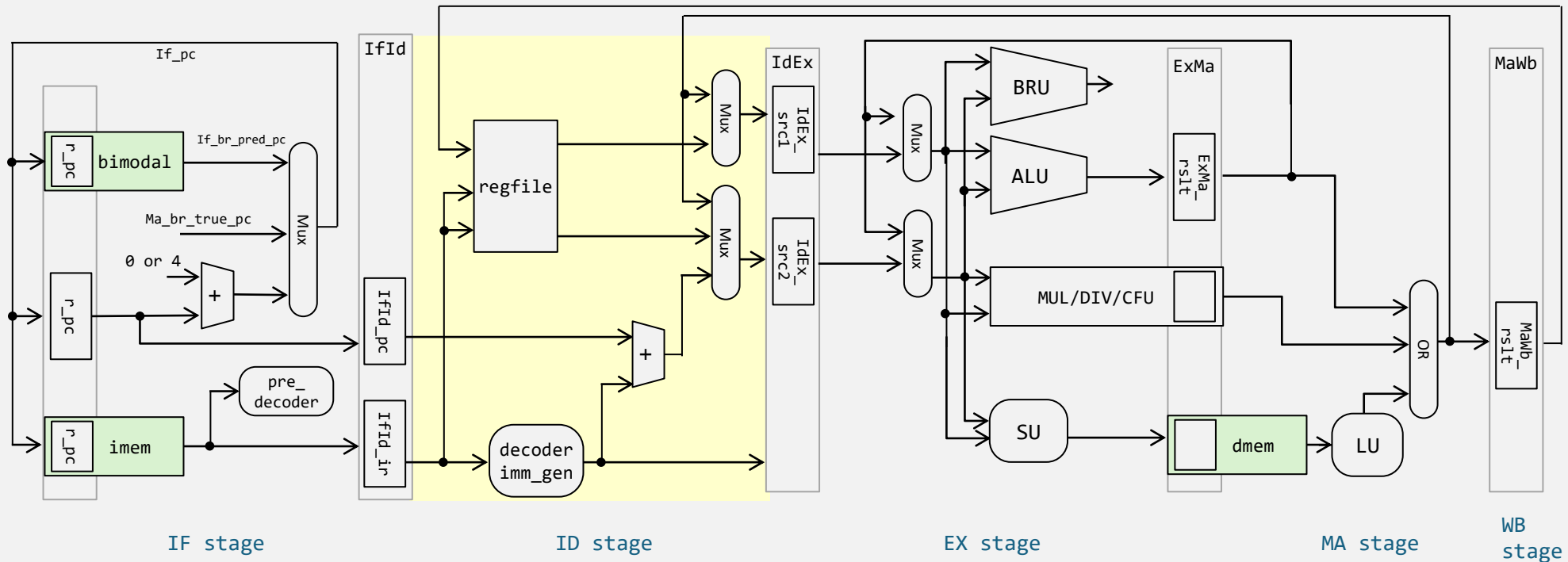
assign rd_o = ((instr_type_o == `S_TYPE) | (instr_type_o == `B_TYPE)) ? 0 : ir_i[11:7];
assign rs1_o = ((instr_type_o == `U_TYPE) | (instr_type_o == `J_TYPE)) ? 0 : ir_i[19:15];
assign rs2_o = ((instr_type_o == `I_TYPE) |
    (instr_type_o == `U_TYPE) | (instr_type_o == `J_TYPE)) ? 0 : ir_i[24:20];
assign rf_we_o = (rd_o != 0);
endmodule
```



opcode5[2:0]	000	001	010	011	100	101	110
opcode5[4:3]							
00	LOAD (I-type)	LOAD-FP	custom-0	MISC-MEM (I-type)	OP-IMM (I-type)	AUIPC (U-type)	OP-IMM-32
01	STORE (S-type)	STORE-FP	custom-1	AMO	OP (R-type)	LUI (U-type)	OP-32
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	custom-2/ rv128
11	BRANCH (B-type)	JALR (I-type)	reserved	JAL (J-type)	SYSTEM (I-type)	reserved	custom-3/ rv128

5-stage pipelining processor of CFU Proving Ground

- **IF (Instruction Fetch)** 命令メモリから命令をフェッチする。
- **ID (Instruction Decode)** 命令をデコード(解読)しながら、レジスタの値を読み出す。
- **EX (Execution)** 命令操作の実行またはアドレスの生成を行う。
- **MA (Memory Access)** 必要であれば、データメモリのオペランドにアクセスする。
- **WB (Write Back)** 必要であれば、結果をレジスタに書き込む。



ID (Instruction Decode) stage

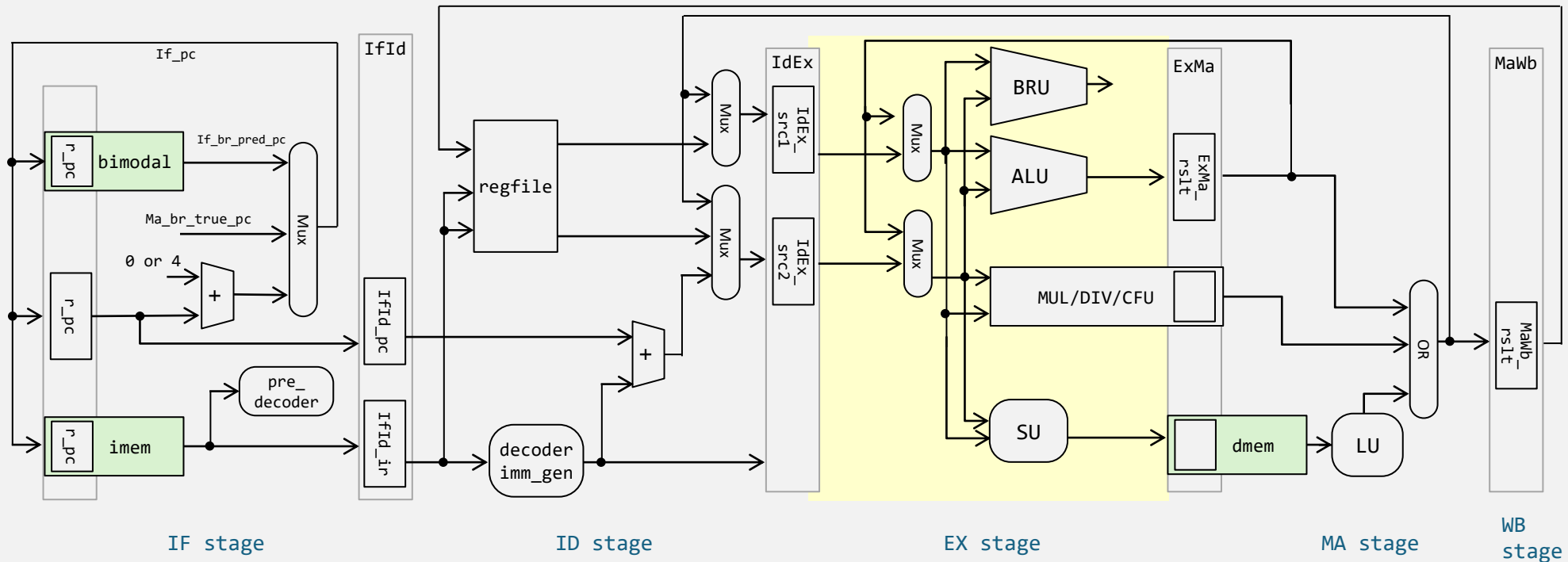
```
//-----  
// ID: Instruction Decode  
//-----  
// instruction decoder  
wire [ `SRC2_CTRL_WIDTH-1:0] Id_src2_ctrl;  
wire [ `ALU_CTRL_WIDTH-1:0] Id_alu_ctrl;  
wire [ `BRU_CTRL_WIDTH-1:0] Id_bru_ctrl;  
wire [ `LSU_CTRL_WIDTH-1:0] Id_lsu_ctrl;  
wire [ `MUL_CTRL_WIDTH-1:0] Id_mul_ctrl;  
wire [ `DIV_CTRL_WIDTH-1:0] Id_div_ctrl;  
wire [ `CFU_CTRL_WIDTH-1:0] Id_cfu_ctrl;  
decoder decoder (  
    .ir_i      (IfId_ir),          // input wire [31:0]  
    .src2_ctrl_o(Id_src2_ctrl),    // output wire [ `SRC2_CTRL_WIDTH-1:0]  
    .alu_ctrl_o (Id_alu_ctrl),     // output wire [ `ALU_CTRL_WIDTH-1:0]  
    .bru_ctrl_o (Id_bru_ctrl),     // output wire [ `BRU_CTRL_WIDTH-1:0]  
    .lsu_ctrl_o (Id_lsu_ctrl),     // output wire [ `LSU_CTRL_WIDTH-1:0]  
    .mul_ctrl_o (Id_mul_ctrl),     // output wire [ `MUL_CTRL_WIDTH-1:0]  
    .div_ctrl_o (Id_div_ctrl),     // output wire [ `DIV_CTRL_WIDTH-1:0]  
    .cfu_ctrl_o (Id_cfu_ctrl),     // output wire [ `CFU_CTRL_WIDTH-1:0]  
);  
  
// immediate value generator  
wire [ `XLEN-1:0] Id_imm;  
imm_gen imm_gen (  
    .ir_i      (IfId_ir),          // input wire [31:0]  
    .instr_type_i(IfId_instr_type), // input wire [ `ITYPE_W-1:0]  
    .imm_o      (Id_imm)           // output wire [ `XLEN-1:0]  
);  
  
// register file  
wire [ `XLEN-1:0] Id_xrs1;  
wire [ `XLEN-1:0] Id_xrs2;  
wire Wb_xreg_we = MaWb_v && MaWb_rf_we && !ExMa_stall;  
regfile xreg (  
    .clk_i      (clk_i),           // input wire  
    .rs1_i      (IfId_rs1),        // input wire [4:0]  
    .rs2_i      (IfId_rs2),        // input wire [4:0]  
    .xrs1_o      (Id_xrs1),         // output wire [ `XLEN-1:0]  
    .xrs2_o      (Id_xrs2),         // output wire [ `XLEN-1:0]  
    .we_i      (Wb_xreg_we && !w_stall), // input wire  
    .rd_i      (MaWb_rd),          // input wire [4:0]  
    .wdata_i      (MaWb_rslt)       // input wire [ `XLEN-1:0]  
);
```

```
// data forwarding  
wire Id_rs1_fwd_Ma_to_Ex = IdEx_v && IdEx_rf_we && (IdEx_rd == IfId_rs1);  
wire Id_rs2_fwd_Ma_to_Ex = IdEx_v && IdEx_rf_we && (IdEx_rd == IfId_rs2);  
wire Id_rs1_fwd_Wb_to_Ex = ExMa_v && ExMa_rf_we && (ExMa_rd == IfId_rs1);  
wire Id_rs2_fwd_Wb_to_Ex = ExMa_v && ExMa_rf_we && (ExMa_rd == IfId_rs2);  
  
wire [31:0] Id_pc_in = (Id_src2_ctrl[ `SRC2_CTRL_USE_AUIPC]) ? IfId_pc : 0;  
wire Id_use_imm = Id_src2_ctrl[ `SRC2_CTRL_USE_AUIPC] |  
    Id_src2_ctrl[ `SRC2_CTRL_USE_IMM];  
  
// source select  
wire [ `XLEN-1:0] Id_src1 = (Id_rs1_fwd_Wb_to_Ex) ? Ma_rslt : Id_xrs1;  
wire [ `XLEN-1:0] Id_src2 = (Id_rs2_fwd_Wb_to_Ex) ? Ma_rslt :  
    (Id_use_imm) ? Id_pc_in+Id_imm : Id_xrs2 ;  
  
wire [31:0] Id_j_pc4 = (Id_bru_ctrl[ `BRU_CTRL_IS_JAL_JALR]) ? IfId_pc + 4 : 0;  
  
always @(posedge clk_i) if (!w_stall) begin  
    if (rst) begin  
        IdEx_v <= 0;  
        IdEx_pc <= 0;  
        IdEx_ir <= `NOP;  
    end else if (!ExMa_stall) begin  
        IdEx_v <= Id_v;  
        IdEx_pc <= IfId_pc;  
        IdEx_j_pc4 <= Id_j_pc4;  
        IdEx_ir <= IfId_ir;  
        IdEx_br_pred_tkn <= IfId_br_pred_tkn;  
        IdEx_pat_hist <= IfId_pat_hist;  
        IdEx_alu_ctrl <= Id_alu_ctrl;  
        IdEx_bru_ctrl <= Id_bru_ctrl;  
        IdEx_lsu_ctrl <= Id_lsu_ctrl;  
        IdEx_mul_ctrl <= Id_mul_ctrl;  
        IdEx_div_ctrl <= Id_div_ctrl;  
        IdEx_rs1_fwd_Ma_to_Ex <= Id_rs1_fwd_Ma_to_Ex;  
        IdEx_rs2_fwd_Ma_to_Ex <= Id_rs2_fwd_Ma_to_Ex;  
        IdEx_src1 <= Id_src1;  
        IdEx_src2 <= Id_src2;  
        IdEx_imm <= Id_imm;  
        IdEx_rf_we <= IfId_rf_we;  
        IdEx_rd <= IfId_rd;  
        IdEx_cfu_ctrl <= Id_cfu_ctrl; // Note  
    end  
end
```



5-stage pipelining processor of CFU Proving Ground

- **IF (Instruction Fetch)** 命令メモリから命令をフェッチする。
- **ID (Instruction Decode)** 命令をデコード(解読)しながら、レジスタの値を読み出す。
- **EX (Execution)** 命令操作の実行またはアドレスの生成を行う。
- **MA (Memory Access)** 必要であれば、データメモリのオペランドにアクセスする。
- **WB (Write Back)** 必要であれば、結果をレジスタに書き込む。



EX (Execution) stage

```
//-----
// EX: Execution
//-----

wire Ex_valid = IdEx_v && !Ma_br_misp && !ExMa_stall;

///// data forwarding
wire [`XLEN-1:0] Ex_src1 = (IdEx_rs1_fwd_Ma_to_Ex) ? ExMa_rslt : IdEx_src1;
wire [`XLEN-1:0] Ex_src2 = (IdEx_rs2_fwd_Ma_to_Ex) ? ExMa_rslt : IdEx_src2;

///// arithmetic logic unit
wire [`XLEN-1:0] Ex_alu_rslt;
alu alu (
    .alu_ctrl_i(IdEx_alu_ctrl), // input wire [`ALU_CTRL_WIDTH-1:0]
    .src1_i (Ex_src1),          // input wire      [`XLEN-1:0]
    .src2_i (Ex_src2),          // input wire      [`XLEN-1:0]
    .j_pc4_i (IdEx_j_pc4),      // input wire      [`XLEN-1:0]
    .rslt_o (Ex_alu_rslt)       // output wire     [`XLEN-1:0]
);

///// branch resolution unit
wire      Ex_is_ctrl_tsfr;
wire      Ex_br_tkn;
wire      Ex_br_misp_rslt1;
wire      Ex_br_misp_rslt2;
wire [`XLEN-1:0] Ex_br_tkn_pc;
bru bru (
    .bru_ctrl_i (IdEx_bru_ctrl), // input wire [`BRU_CTRL_WIDTH-1:0]
    .src1_i      (Ex_src1),       // input wire      [`XLEN-1:0]
    .src2_i      (Ex_src2),       // input wire      [`XLEN-1:0]
    .pc_i        (IdEx_pc),       // input wire      [`XLEN-1:0]
    .imm_i       (IdEx_imm),      // input wire      [`XLEN-1:0]
    .npc_i       (IfId_pc),       // input wire      [`XLEN-1:0]
    .br_pred_tkn_i (IdEx_br_pred_tkn), // input wire
    .is_ctrl_tsfr_o (Ex_is_ctrl_tsfr), // output wire
    .br_tkn_o      (Ex_br_tkn),    // output wire
    .br_misp_rslt1_o (Ex_br_misp_rslt1), // output wire
    .br_misp_rslt2_o (Ex_br_misp_rslt2), // output wire
    .br_tkn_pc_o   (Ex_br_tkn_pc)  // output wire     [`XLEN-1:0]
);

///// store unit
wire [      `XLEN-1:0] dbus_addr = dbus_addr_o; // for simulation
wire [      `XLEN-1:0] dbus_wdata = dbus_wdata_o; // for simulation
wire [`DBUS_OFFSET_W-1:0] dbus_offset; // Note
store_unit store_unit (
    .valid_i (Ex_valid && !w_stall), // input wire
    .lsu_ctrl_i (IdEx_lsu_ctrl), // input wire [`LSU_CTRL_WIDTH-1:0]
    .src1_i (Ex_src1), // input wire      [`XLEN-1:0]
    .src2_i (Ex_src2), // input wire      [`XLEN-1:0]
    .imm_i (IdEx_imm), // input wire      [`XLEN-1:0]
    .dbus_addr_o (dbus_addr_o), // output wire     [`XLEN-1:0]
    .dbus_offset_o (dbus_offset), // output wire     [OFFSET_WIDTH-1:0]
    .dbus_wvalid_o (dbus_wvalid_o), // output wire
    .dbus_wdata_o (dbus_wdata_o), // output wire     [`XLEN-1:0]
    .dbus_wstrb_o (dbus_wstrb_o) // output wire     [`XBYTES-1:0]
);
```

```
///// multiplier unit
wire      Ex_mul_stall;
wire [`XLEN-1:0] Ex_mul_rslt;
multiplier multiplier (
    .clk_i (clk_i), // input wire
    .rst_i (rst),   // input wire // Note
    .stall_i (w_stall), // input wire
    .valid_i (Ex_valid), // input wire
    .mul_ctrl_i (IdEx_mul_ctrl), // input wire [`MUL_CTRL_WIDTH-1:0]
    .src1_i (Ex_src1), // input wire      [`XLEN-1:0]
    .src2_i (Ex_src2), // input wire      [`XLEN-1:0]
    .stall_o (Ex_mul_stall), // output wire
    .rslt_o (Ex_mul_rslt) // output wire     [`XLEN-1:0]
);

///// divider unit
wire      Ex_div_stall;
wire [`XLEN-1:0] Ex_div_rslt;
divider divider (
    .clk_i (clk_i), // input wire
    .rst_i (rst),   // input wire
    .stall_i (w_stall), // input wire // Note
    .valid_i (Ex_valid), // input wire
    .div_ctrl_i (IdEx_div_ctrl), // input wire [`DIV_CTRL_WIDTH-1:0]
    .src1_i (Ex_src1), // input wire      [`XLEN-1:0]
    .src2_i (Ex_src2), // input wire      [`XLEN-1:0]
    .stall_o (Ex_div_stall), // output wire
    .rslt_o (Ex_div_rslt) // output wire     [`XLEN-1:0]
);

always @(posedge clk_i) if (!w_stall) begin
    ExMa_mul_stall <= Ex_mul_stall;
    ExMa_div_stall <= Ex_div_stall;
    ExMa_stall <= Ex_mul_stall | Ex_div_stall | Ex_cfu_stall;
    ExMa_mdc_rslt <= Ex_mul_rslt | Ex_div_rslt | Ex_cfu_rslt;
    if (rst) begin
        ExMa_v <= 0;
        ExMa_pc <= 0;
        ExMa_ir <= `NOP;
    end else if (!ExMa_stall) begin
        ExMa_v <= Ex_v;
        ExMa_pc <= IdEx_pc;
        ExMa_ir <= IdEx_ir;
        ExMa_pat_hist <= IdEx_pat_hist;
        ExMa_is_ctrl_tsfr <= Ex_is_ctrl_tsfr;
        ExMa_br_tkn <= Ex_br_tkn;
        ExMa_br_misp_rslt1 <= Ex_br_misp_rslt1;
        ExMa_br_misp_rslt2 <= Ex_br_misp_rslt2;
        ExMa_br_tkn_pc <= Ex_br_tkn_pc;
        ExMa_lsu_ctrl <= IdEx_lsu_ctrl;
        ExMa_dbus_offset <= dbus_offset;
        ExMa_rf_we <= IdEx_rf_we;
        ExMa_rd <= IdEx_rd;
        ExMa_rslt <= Ex_alu_rslt;
        ExMa_j_b_insn <= IdEx_bru_ctrl[0] & Ex_v;
    end
end
```



EX (Execution) stage

```
module alu (
  input wire [ALU_CTRL_WIDTH-1:0] alu_ctrl_i,
  input wire [31:0] src1_i,
  input wire [31:0] src2_i,
  input wire [31:0] j_pc4_i,
  output wire [31:0] rslt_o
);

wire w_signed = alu_ctrl_i[ALU_CTRL_IS_SIGNED];
wire w_neg = alu_ctrl_i[ALU_CTRL_IS_NEG];
wire w_less = alu_ctrl_i[ALU_CTRL_IS_LESS];

wire [33:0] adder_src1 = {w_signed && src1_i[31], src1_i, 1'b1};
wire [33:0] adder_src2 = {w_signed && src2_i[31], src2_i, 1'b0} ^ {34{w_neg}};
wire [33:0] adder_rslt_t = adder_src1+adder_src2;
wire less_rslt = w_less && adder_rslt_t[33];
wire [31:0] adder_rslt = (alu_ctrl_i[ALU_CTRL_IS_ADD]) ? adder_rslt_t[32:1] : 0;

wire signed [32:0] right_shifter_src1 = {w_signed && src1_i[31], src1_i};
wire [4:0] shamt = src2_i[4:0];
wire [31:0] left_shifter_rslt = (alu_ctrl_i[ALU_CTRL_IS_SHIFT_LEFT] ) ?
  src1_i << shamt : 0;
wire [31:0] right_shifter_rslt = (alu_ctrl_i[ALU_CTRL_IS_SHIFT_RIGHT]) ?
  right_shifter_src1 >> shamt : 0;

wire [31:0] bitwise_rslt = ((alu_ctrl_i[ALU_CTRL_IS_XOR_OR]) ?
  (src1_i ^ src2_i) : 0) |
  ((alu_ctrl_i[ALU_CTRL_IS_OR_AND])
  ? (src1_i & src2_i) : 0);
wire [31:0] lui_auiipc_rslt = (alu_ctrl_i[ALU_CTRL_IS_SRC2]) ? src2_i : 0;

assign rslt_o = less_rslt | adder_rslt | left_shifter_rslt | right_shifter_rslt |
  bitwise_rslt | lui_auiipc_rslt | j_pc4_i;
endmodule
```

```
module store_unit (
  input wire valid_i,
  input wire [5:0] lsu_ctrl_i,
  input wire [31:0] src1_i,
  input wire [31:0] src2_i,
  input wire [31:0] imm_i,
  output wire [31:0] dbus_addr_o,
  output wire [1:0] dbus_offset_o,
  output wire dbus_wvalid_o,
  output wire [31:0] dbus_wdata_o,
  output wire [3:0] dbus_wstrb_o
);

assign dbus_addr_o = (valid_i && (lsu_ctrl_i[LSU_CTRL_IS_STORE] ||
  lsu_ctrl_i[LSU_CTRL_IS_LOAD])) ? src1_i + imm_i : 0;
assign dbus_offset_o = dbus_addr_o[1:0];
assign dbus_wvalid_o = valid_i && lsu_ctrl_i[LSU_CTRL_IS_STORE];

wire w_sb = lsu_ctrl_i[LSU_CTRL_IS_BYTE];
wire w_sh = lsu_ctrl_i[LSU_CTRL_IS_HALFWORD];
wire w_sw = lsu_ctrl_i[LSU_CTRL_IS_WORD];

assign dbus_wdata_o[7:0] = src2_i[7:0];
assign dbus_wdata_o[15:8] = (w_sb) ? src2_i[7:0] : src2_i[15:8];
assign dbus_wdata_o[23:16] = (w_sw) ? src2_i[23:16] : src2_i[7:0];
assign dbus_wdata_o[31:24] = (w_sb) ? src2_i[7:0] : (w_sh) ? src2_i[15:8] : src2_i[31:24];

assign dbus_wstrb_o[0] = (w_sb && dbus_offset_o==0) || (w_sh && dbus_offset_o[1]==0) || w_sw;
assign dbus_wstrb_o[1] = (w_sb && dbus_offset_o==1) || (w_sh && dbus_offset_o[1]==0) || w_sw;
assign dbus_wstrb_o[2] = (w_sb && dbus_offset_o==2) || (w_sh && dbus_offset_o[1]==1) || w_sw;
assign dbus_wstrb_o[3] = (w_sb && dbus_offset_o==3) || (w_sh && dbus_offset_o[1]==1) || w_sw;
endmodule
```



EX (Execution) stage

```
module bru (
  input wire [`BRU_CTRL_WIDTH-1:0] bru_ctrl_i,
  input wire [31:0] src1_i,
  input wire [31:0] src2_i,
  input wire [31:0] pc_i,
  input wire [31:0] imm_i,
  input wire [31:0] npc_i,
  input wire br_pred_tkn_i,
  output wire is_ctrl_tsfr_o,
  output wire br_tkn_o,
  output wire br_misp_rslt1_o,
  output wire br_misp_rslt2_o,
  output wire [31:0] br_tkn_pc_o
);

  wire signed [32:0] sext_src1 = {bru_ctrl_i[`BRU_CTRL_IS_SIGNED] && src1_i[31], src1_i};
  wire signed [32:0] sext_src2 = {bru_ctrl_i[`BRU_CTRL_IS_SIGNED] && src2_i[31], src2_i};

  wire w_eq = (src1_i == src2_i); // equal
  wire w_lt = (sext_src1 < sext_src2); // less than

  assign br_tkn_o = bru_ctrl_i[`BRU_CTRL_IS_JAL_JALR] |
    (bru_ctrl_i[`BRU_CTRL_IS_BEQ] & w_eq) |
    (bru_ctrl_i[`BRU_CTRL_IS_BNE] & !w_eq) |
    (bru_ctrl_i[`BRU_CTRL_IS_BLT] & w_lt) |
    (bru_ctrl_i[`BRU_CTRL_IS_BGE] & !w_lt);

  wire [31:0] br_tkn_pc_t;
  assign br_tkn_pc_t = ((bru_ctrl_i[`BRU_CTRL_IS_JALR]) ? src1_i : pc_i) + imm_i;
  assign br_tkn_pc_o = {br_tkn_pc_t[31:1], 1'b0};

  assign is_ctrl_tsfr_o = (bru_ctrl_i[`BRU_CTRL_IS_CTRL_TSFR] || br_pred_tkn_i);

  assign br_misp_rslt1_o = (npc_i != br_tkn_pc_o);
  assign br_misp_rslt2_o = (npc_i != (pc_i + 'h4));
endmodule
```

```
module multiplier (
  input wire clk_i,
  input wire rst_i,
  input wire stall_i,
  input wire valid_i,
  input wire [3:0] mul_ctrl_i,
  input wire [31:0] src1_i,
  input wire [31:0] src2_i,
  output wire stall_o,
  output wire [31:0] rslt_o
);

  reg [1:0] state = `MUL_IDLE;
  reg signed [32:0] r_multiplicand; // 33bit
  reg signed [32:0] r_multiplier; // 33bit
  reg [63:0] product; // 64bit
  reg is_high; //

  assign rslt_o = (state != `MUL_RET) ? 0 : (is_high) ? product[63:32] : product[31:0];

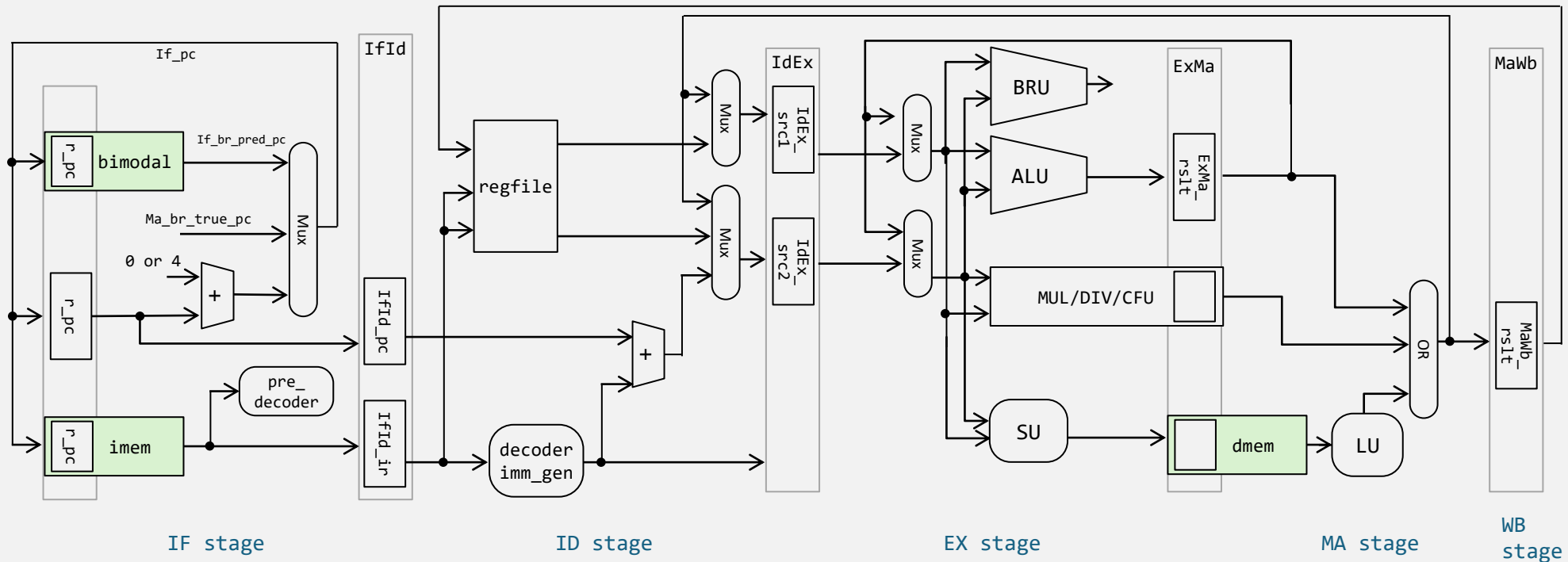
  wire w_mul = mul_ctrl_i[`MUL_CTRL_IS_MUL];
  wire w_src1_signed = mul_ctrl_i[`MUL_CTRL_IS_SRC1_SIGNED];
  wire w_src2_signed = mul_ctrl_i[`MUL_CTRL_IS_SRC2_SIGNED];
  wire w_is_high = mul_ctrl_i[`MUL_CTRL_IS_HIGH];
  wire [1:0] w_state = (state == `MUL_IDLE && valid_i && w_mul) ? `MUL_EXEC :
    (state == `MUL_EXEC) ? `MUL_RET : `MUL_IDLE;

  always @(posedge clk_i) if (!stall_i) begin
    if (rst_i) begin
      state <= `MUL_IDLE;
    end else if (!stall_i) begin
      if (state == `MUL_IDLE) r_multiplicand <= {w_src1_signed && src1_i[31], src1_i};
      if (state == `MUL_IDLE) r_multiplier <= {w_src2_signed && src2_i[31], src2_i};
      if (state == `MUL_IDLE) is_high <= w_is_high;
      if (state == `MUL_EXEC) product <= r_multiplicand * r_multiplier;
      state <= w_state;
    end
  end
  assign stall_o = (w_state != `MUL_IDLE);
endmodule
```

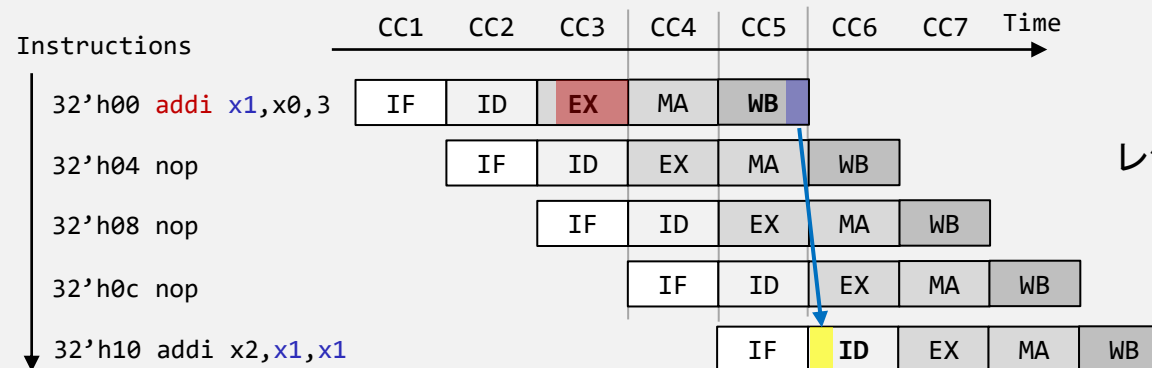
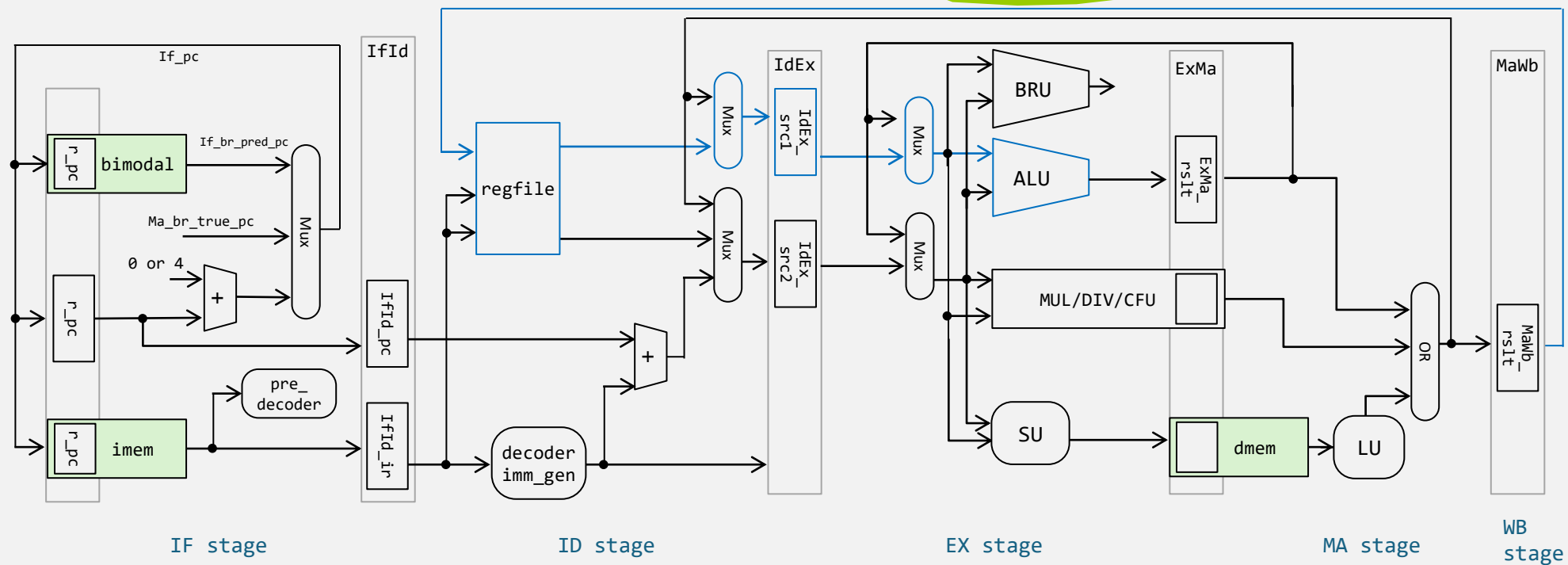


5-stage pipelining processor of CFU Proving Ground

- **IF (Instruction Fetch)** 命令メモリから命令をフェッチする。
- **ID (Instruction Decode)** 命令をデコード(解読)しながら、レジスタの値を読み出す。
- **EX (Execution)** 命令操作の実行またはアドレスの生成を行う。
- **MA (Memory Access)** 必要であれば、データメモリのオペランドにアクセスする。
- **WB (Write Back)** 必要であれば、結果をレジスタに書き込む。



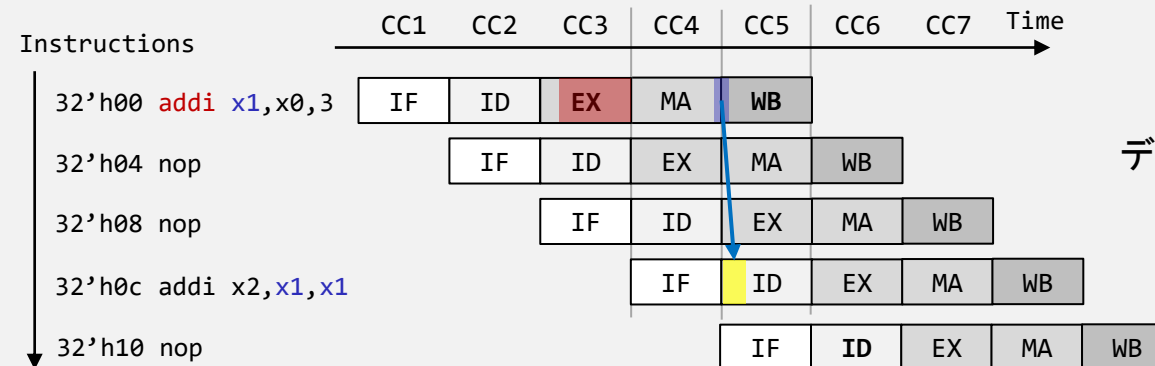
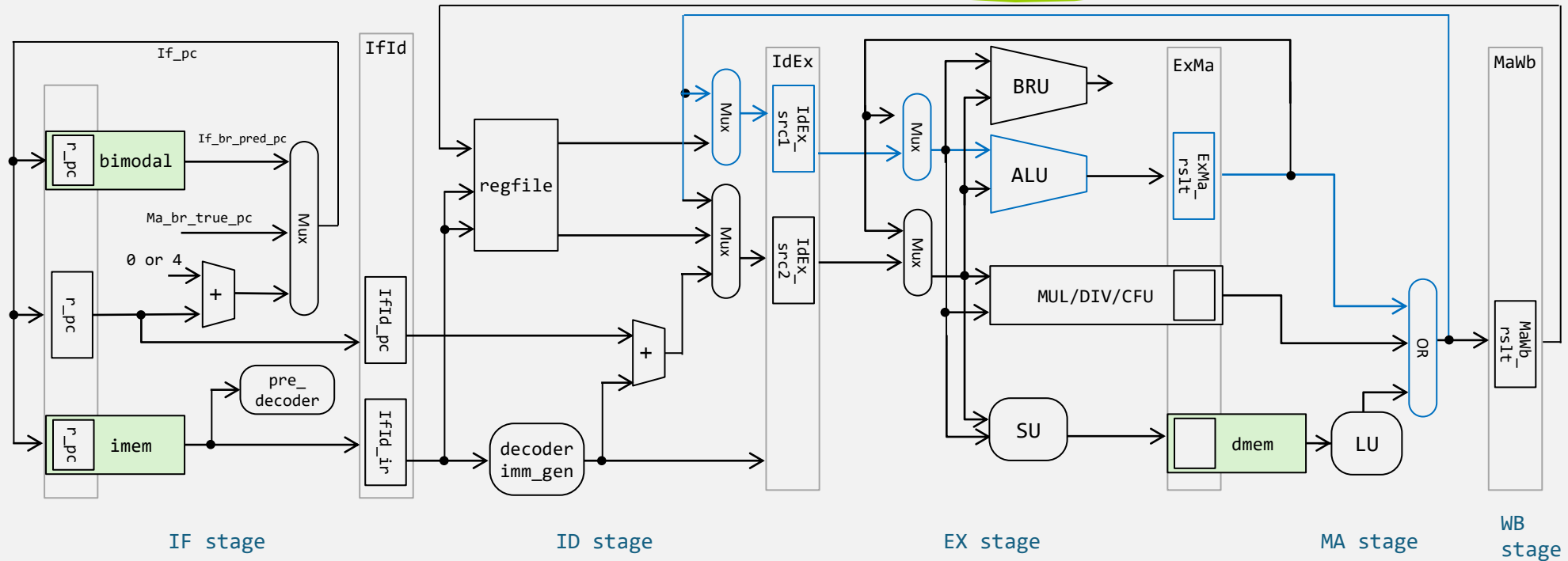
パイプラインプロセッサでのデータ供給 (1)



レジスタファイルから

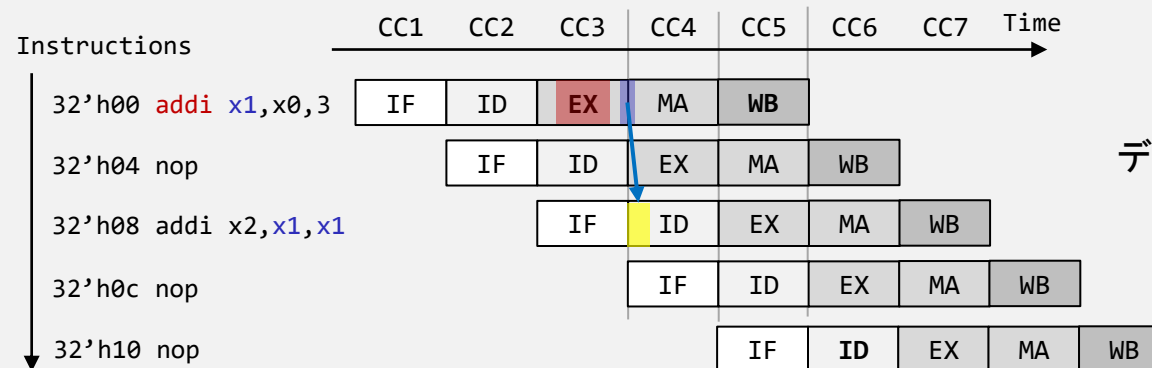
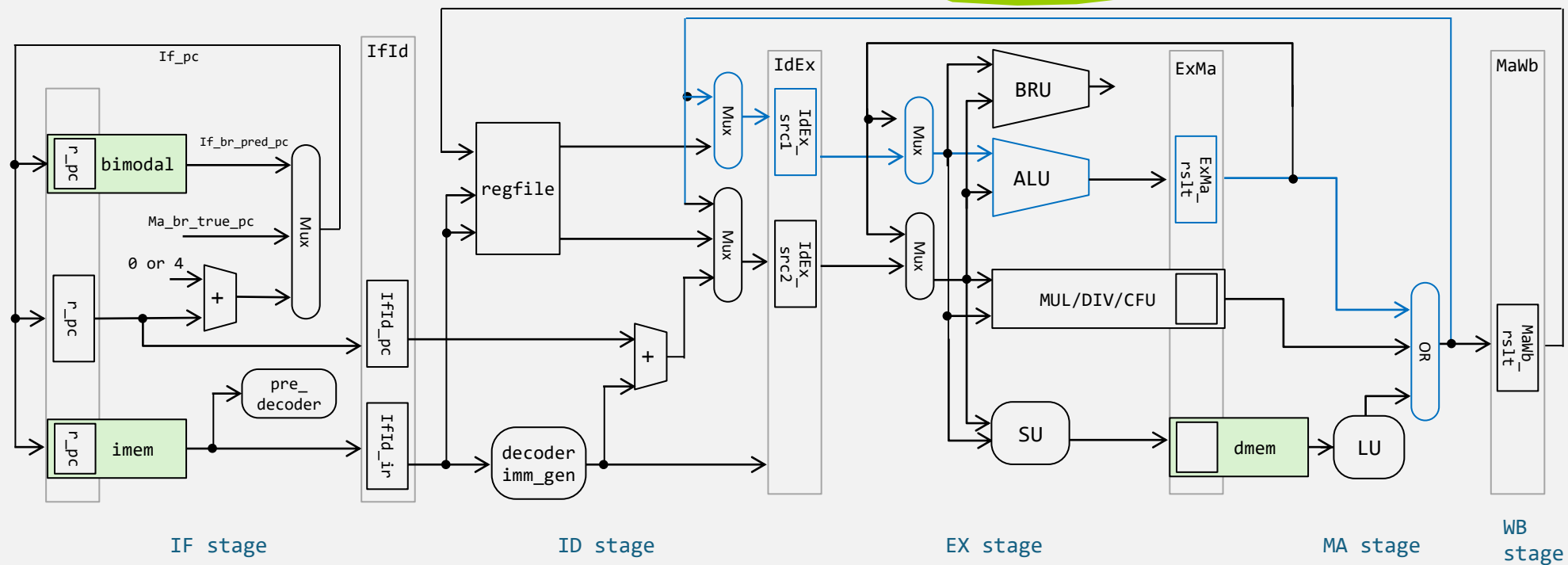


パイプラインプロセッサでのデータ供給 (2)



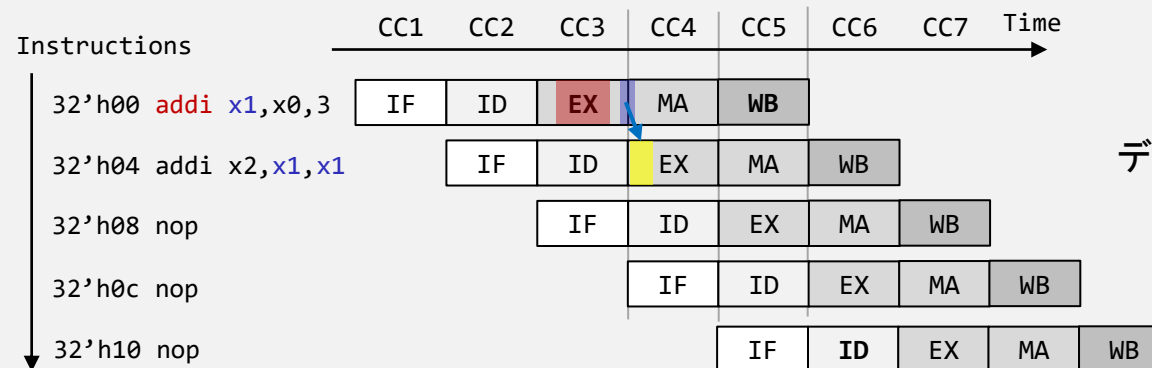
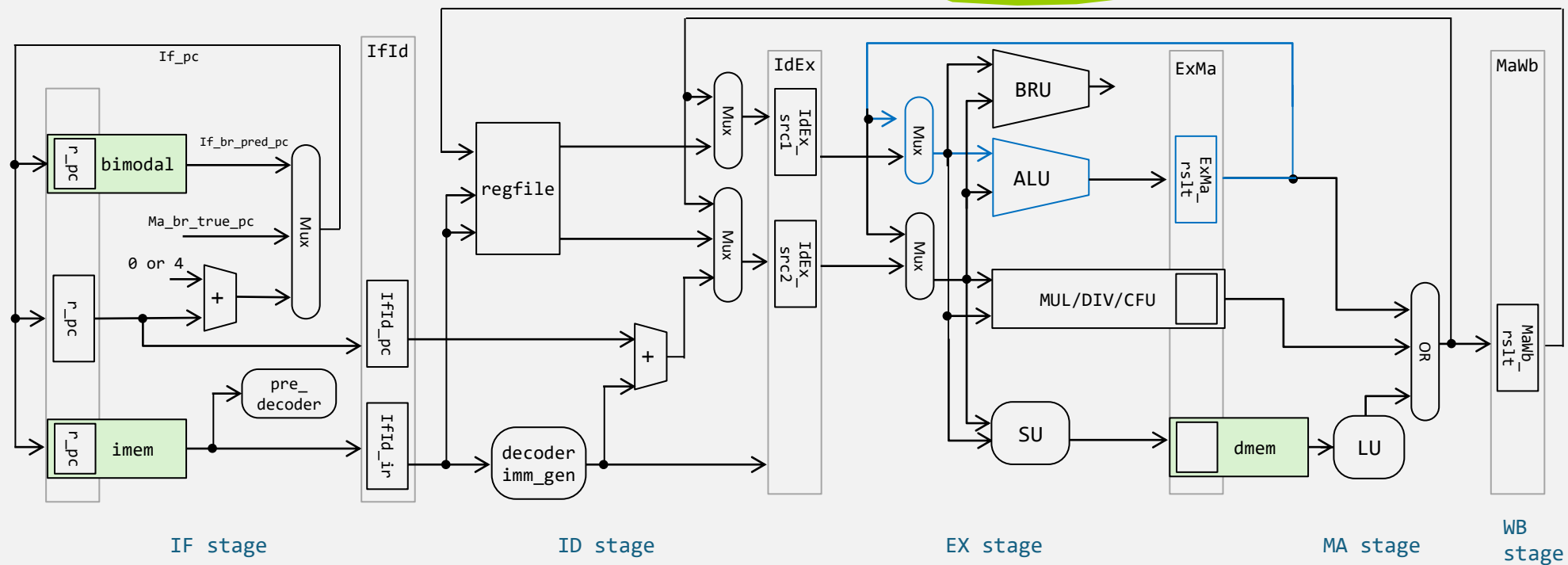
データフォワーディング

パイプラインプロセッサでのデータ供給 (3)



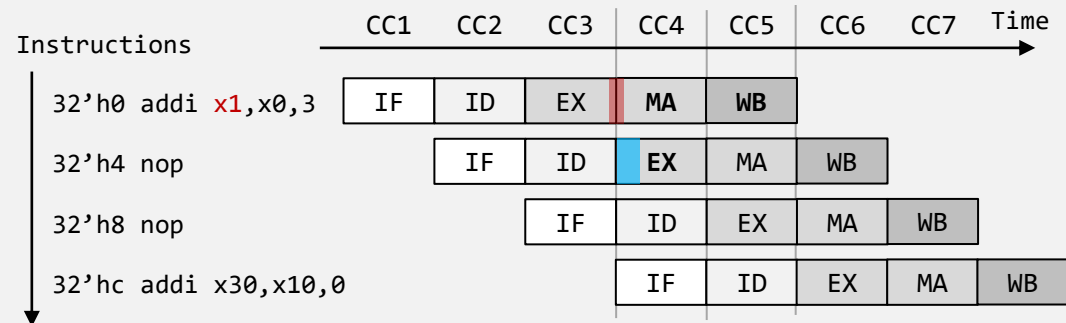
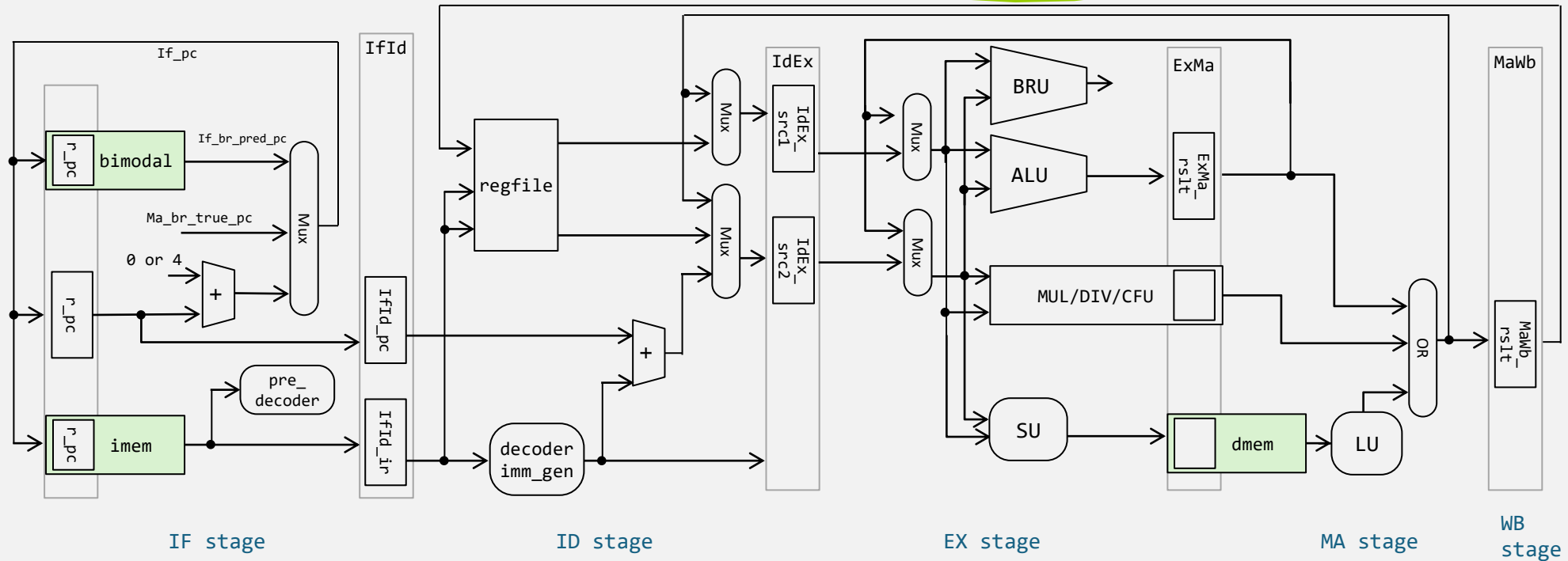
データフォワーディング

パイプラインプロセッサでのデータ供給 (4)



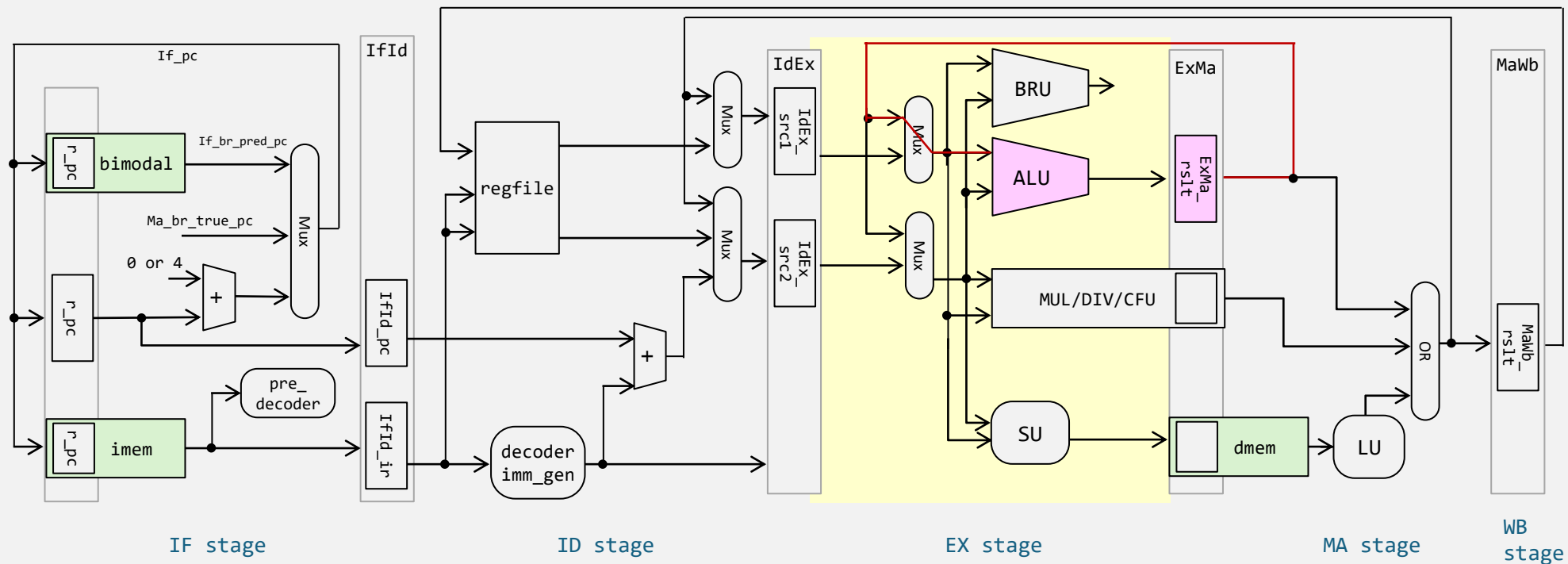
データフォワーディング

5-stage pipelining processor of CFU Proving Ground



データフォワーディング

- **IF (Instruction Fetch)** 命令メモリから命令をフェッチする。
- **ID (Instruction Decode)** 命令をデコード(解読)しながら、レジスタの値を読み出す。
- **EX (Execution)** 命令操作の実行またはアドレスの生成を行う。
- **MA (Memory Access)** 必要であれば、データメモリのオペランドにアクセスする。
- **WB (Write Back)** 必要であれば、結果をレジスタに書き込む。



データフォワーディング

- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
 - make
 - make run

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00020000
.align 4
.section .text.init
.globl _start
_start:
    li    x1, 1
    li    x2, 2
    add   x3, x2, x2
    li    t0, TO_HOST
    li    t1, CMD_FINISH
    sw    t1, 0(t0)
L1: j     L1
```

```
./obj_dir/top
000000: 00000000 -----
000000: 00000004 00000000 -----
000000: 00000008 00000004 00000000 -----
000000: 0000000c 00000008 00000004 00000000 -----
000001: 00000010 0000000c 00000008 00000004 00000000:
000002: 00000014 00000010 0000000c 00000008 00000004:
000003: 00000018 00000014 00000010 0000000c 00000008:
000004: 0000001c 00000018 00000014 00000010 0000000c:
000005: 00000020 0000001c 00000018 00000014 00000010:
- top.v:47: Verilog $finish

==> mcycle : 9
==> minstret : 510
==> Total number of branch predictions : 0
==> Total number of branch mispredictions : 0
==> simulation finish!!
```



データフォワーディング

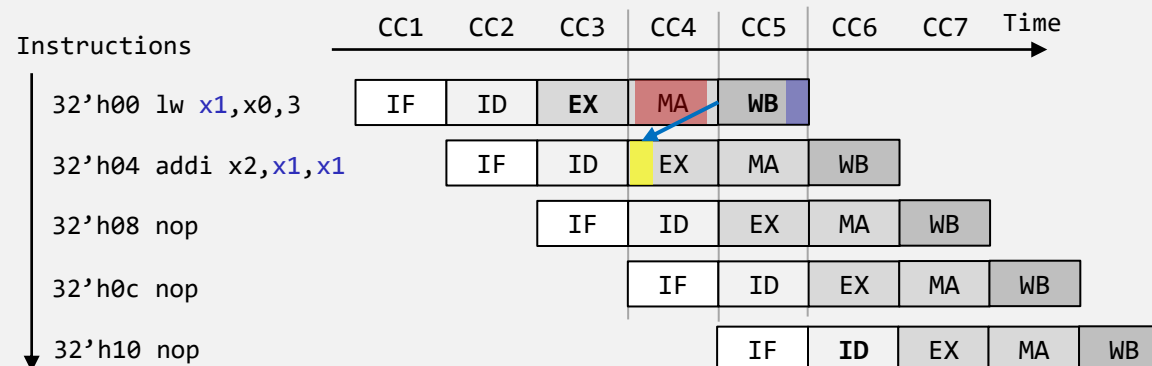
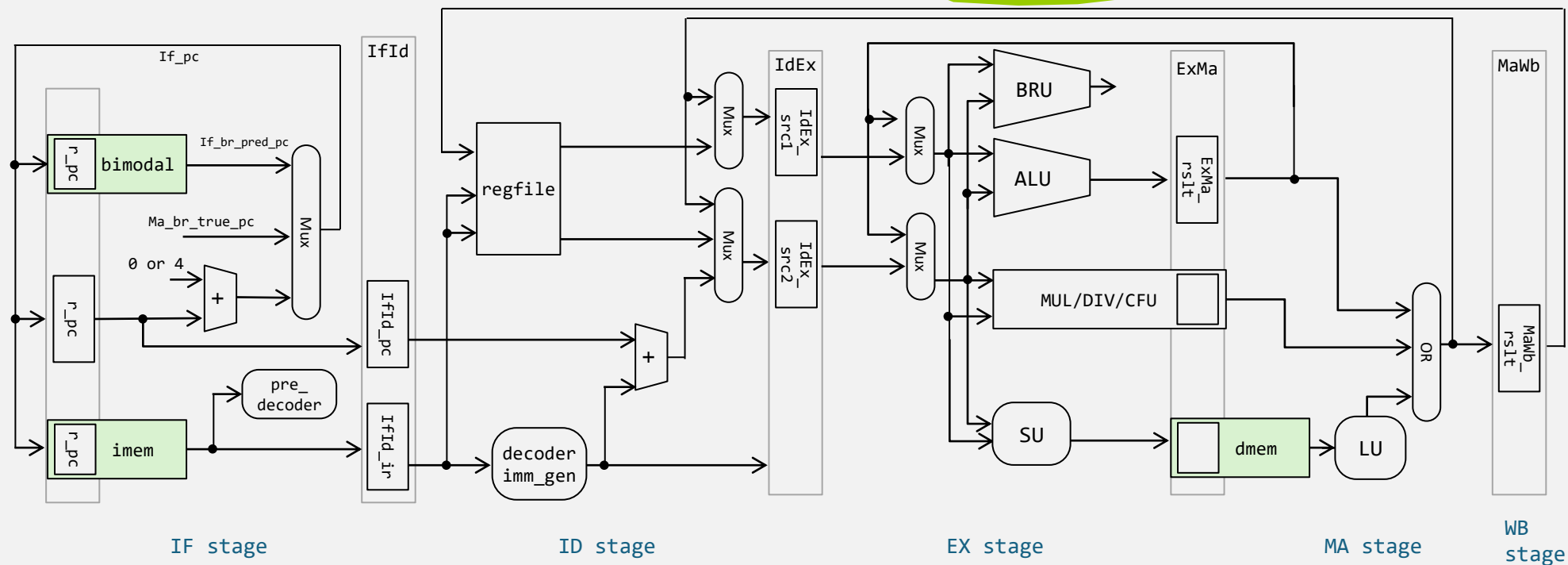
- ロード命令がロードしたデータを**直後の命令**が利用する場合には、データフォワーディングでオペランドを供給できない。そのため、**直後の命令**のデコードステージから実行ステージへの移動を1サイクル遅らせる(バブルをひとつ挿入する)。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
 - make
 - make run

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00020000
.equ DMEM_ADDR,  0x10000000
.align 4
.section .text.init
.globl _start
_start:
    li    x1, 1
    li    x2, DMEM_ADDR
    lw    x2, 0(x2)
    add   x3, x2, x2
    li    t0, TO_HOST
    li    t1, CMD_FINISH
    sw    t1, 0(t0)
L1: j     L1
```

```
./obj_dir/top
000000: 00000000 -----
000000: 00000004 00000000 -----
000000: 00000008 00000004 00000000 -----
000000: 0000000c 00000008 00000004 00000000 -----
000001: 00000010 0000000c 00000008 00000004 00000000:
000002: 00000010 0000000c ----- 00000008 00000004:
000003: 00000014 00000010 0000000c ----- 00000008:
000003: 00000018 00000014 00000010 0000000c -----
000004: 0000001c 00000018 00000014 00000010 0000000c:
000005: 00000020 0000001c 00000018 00000014 00000010:
000006: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

==> mcycle : 11
==> minstret : 6
==> Total number of branch predictions : 0
==> Total number of branch mispredictions : 0
==> simulation finish!!
```

パイプラインプロセッサでのデータ供給

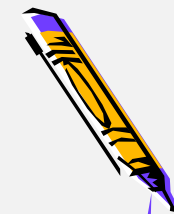


マルチサイクルを必要とする命令の処理

- 除算の実行には35サイクルを要する。そのため、34サイクルでストールさせる。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
 - make
 - make run

```
.equ TO_HOST,      0x80000000
.equ CMD_FINISH,   0x00020000
.align 4
.section .text.init
.globl _start

_start:
    li      x1, 1
    li      x2, 2
    div     x3, x2, x2
    li      t0, TO_HOST
    li      t1, CMD_FINISH
    sw      t1, 0(t0)
L1: j      L1
```

[illegible]

マルチサイクルを必要とする命令の処理

- 乗算の実行には3サイクルを要する。そのため、2サイクルでストールさせる。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
 - make
 - make run

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00020000
.align 4
.section .text.init
.globl _start
_start:
    li    x1, 1
    li    x2, 2
    mul   x3, x2, x2
    li    t0, TO_HOST
    li    t1, CMD_FINISH
    sw    t1, 0(t0)
L1: j     L1
```

```
./obj_dir/top
000000: 00000000 -----
000000: 00000004 00000000 -----
000000: 00000008 00000004 00000000 -----
000000: 0000000c 00000008 00000004 00000000 -----
000001: 00000010 0000000c 00000008 00000004 00000000:
000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
000002: 00000014 ssss0010 ssss000c ssss0008 ssss0004:
000002: 00000014 00000010 0000000c 00000008 00000004:
000003: 00000018 00000014 00000010 0000000c 00000008:
000004: 0000001c 00000018 00000014 00000010 0000000c:
000005: 00000020 0000001c 00000018 00000014 00000010:
- top.v:47: Verilog $finish

==> mcycle           : 11
==> minstret          : 5
==> Total number of branch predictions : 0
==> Total number of branch mispredictions : 0
==> simulation finish!!
```



マルチサイクルを必要とする命令の処理

- 乗算の実行には3サイクルを要する。そのため、2サイクルでストールさせる。
- ロード命令と同様に、直後の命令が除算の結果を利用する場合には、データフォワードリングでオペランドを供給できない。そのため、バブルをひとつ挿入する。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
 - make
 - make run

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00020000
.align 4
.section .text.init
.globl _start
_start:
    li    x1, 1
    li    x2, 2
    mul   x3, x2, x2
    add   x4, x3, x3
    li    t0, TO_HOST
    li    t1, CMD_FINISH
    sw    t1, 0(t0)
L1: j     L1
```

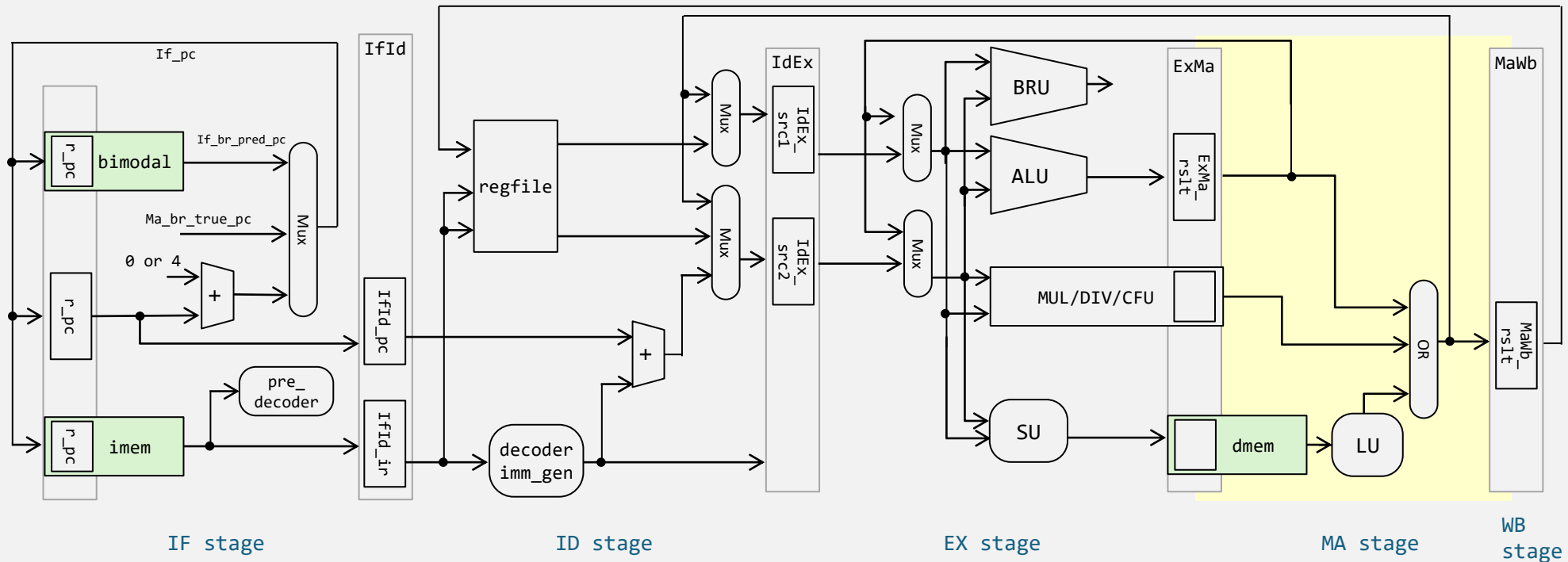
```
./obj_dir/top
000000: 00000000 -----
000000: 00000004 00000000 -----
000000: 00000008 00000004 00000000 -----
000000: 0000000c 00000008 00000004 00000000 -----
000001: 00000010 0000000c 00000008 00000004 00000000:
000002: 00000010 ssss000c ----- ssss0008 ssss0004:
000002: 00000010 ssss000c ----- ssss0008 ssss0004:
000002: 00000010 0000000c ----- 00000008 00000004:
000003: 00000014 00000010 0000000c ----- 00000008:
000003: 00000018 00000014 00000010 0000000c -----
000004: 0000001c 00000018 00000014 00000010 0000000c:
000005: 00000020 0000001c 00000018 00000014 00000010:
000006: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

==> mcycle : 13
==> minstret : 6
==> Total number of branch predictions : 0
==> Total number of branch mispredictions : 0
==> simulation finish!!
```



5-stage pipelining processor of CFU Proving Ground

- **IF (Instruction Fetch)** 命令メモリから命令をフェッチする。
- **ID (Instruction Decode)** 命令をデコード(解読)しながら、レジスタの値を読み出す。
- **EX (Execution)** 命令操作の実行またはアドレスの生成を行う。
- **MA (Memory Access)** 必要であれば、データメモリのオペランドにアクセスする。
- **WB (Write Back)** 必要であれば、結果をレジスタに書き込む。



MA (Memory Access) stage

```
//-----  
// MA: Memory Access  
//-----  
// load unit  
wire [`XLEN-1:0] Ma_load_rslt;  
load_unit load_unit (  
    .lsu_ctrl_i (ExMa_lsu_ctrl), // input wire [`LSU_CTRL_WIDTH-1:0]  
    .dbus_offset_i(ExMa_dbus_offset), // input wire [OFFSET_WIDTH-1:0]  
    .dbus_rdata_i (dbus_rdata_i), // input wire [`XLEN-1:0]  
    .rslt_o (Ma_load_rslt) // output wire [`XLEN-1:0]  
);  
  
wire [`XLEN-1:0] Ma_rslt = ExMa_rslt | ExMa_mdc_rslt | Ma_load_rslt;  
  
always @(posedge clk_i) if (!w_stall) begin  
    if (rst) begin  
        MaWb_v <= 0;  
        MaWb_pc <= 0;  
        MaWb_ir <= `NOP;  
    end else if (!ExMa_stall) begin  
        MaWb_v <= Ma_v;  
        MaWb_pc <= ExMa_pc;  
        MaWb_ir <= ExMa_ir;  
        MaWb_rf_we <= ExMa_rf_we;  
        MaWb_rd <= ExMa_rd;  
        MaWb_rslt <= Ma_rslt;  
    end  
end
```

```
module load_unit (  
    input wire [ 5:0] lsu_ctrl_i,  
    input wire [ 1:0] dbus_offset_i,  
    input wire [31:0] dbus_rdata_i,  
    output wire [31:0] rslt_o  
);  
  
wire w_lb = lsu_ctrl_i[`LSU_CTRL_IS_BYTE];  
wire w_lh = lsu_ctrl_i[`LSU_CTRL_IS_HALFWORD];  
wire w_lw = lsu_ctrl_i[`LSU_CTRL_IS_WORD];  
wire w_signed = lsu_ctrl_i[`LSU_CTRL_IS_SIGNED];  
wire w_load = lsu_ctrl_i[`LSU_CTRL_IS_LOAD];  
wire [1:0] ost = dbus_offset_i; // offset  
wire [31:0] d = dbus_rdata_i; // data  
  
wire w_lb_sign = w_lb & ((ost==0) ? d[7] : (ost==1) ? d[15] : (ost==2) ? d[23] : d[31]) & w_signed;  
wire w_lh_sign = w_lh & ((ost[1] == 0) ? d[15] : d[31]) & w_signed;  
  
wire [7:0] w1, w2, w3, w4;  
assign w1 = (w_load==0) ? 0 : (w_lw || (w_lh & ost[1]==0) || (w_lb & ost==0)) ? d[7:0] :  
    (w_lb && ost==1) ? d[15:8] : ((w_lb && ost==2) ||  
    (w_lh && ost[1]==1)) ? d[23:16] : d[31:24];  
assign w2 = (w_load==0) ? 0 : (w_lw || (w_lh && ost[1]==0)) ? d[15:8] :  
    (w_lh && ost[1]==1) ? d[31:24] : (w_lb_sign) ? 8'hff : 0;  
assign w3 = (w_load == 0) ? 0 : (w_lw) ? d[23:16] : ((w_lb_sign) || (w_lh_sign)) ? 8'hff : 0;  
assign w4 = (w_load == 0) ? 0 : (w_lw) ? d[31:24] : ((w_lb_sign) || (w_lh_sign)) ? 8'hff : 0;  
assign rslt_o = {w4, w3, w2, w1};  
endmodule
```



MA (Memory Access) stage

- **ロード命令**がロードしたデータを**直後の命令**が利用する場合には、データフォワーディングでオペランドを供給できない。そのため、**直後の命令**のデコードステージから実行ステージへの移動を1サイクル遅らせる(バブルをひとつ挿入する)。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
 - make
 - make run

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00020000
.equ DMEM_ADDR,  0x10000000
.align 4
.section .text.init
.globl _start
_start:
    li    x1, 1
    li    x2, DMEM_ADDR
    lw    x2, 0(x2)
    add   x3, x2, x2
    li    t0, TO_HOST
    li    t1, CMD_FINISH
    sw    t1, 0(t0)
L1: j     L1
```

```
./obj_dir/top
000000: 00000000 -----
000000: 00000004 00000000 -----
000000: 00000008 00000004 00000000 -----
000000: 0000000c 00000008 00000004 00000000 -----
000001: 00000010 0000000c 00000008 00000004 00000000:
000002: 00000010 0000000c ----- 00000008 00000004:
000003: 00000014 00000010 0000000c ----- 00000008:
000003: 00000018 00000014 00000010 0000000c -----
000004: 0000001c 00000018 00000014 00000010 0000000c:
000005: 00000020 0000001c 00000018 00000014 00000010:
000006: 00000024 00000020 0000001c 00000018 00000014:
- top.v:47: Verilog $finish

==> mcycle : 11
==> minstret : 6
==> Total number of branch predictions : 0
==> Total number of branch mispredictions : 0
==> simulation finish!!
```



MA (Memory Access) stage

- ロード命令がロードしたデータを直後の命令が利用する場合には、データフォワーディングでオペランドを供給できない。そのため、直後の命令のデコードステージから実行ステージへの移動を1サイクル遅らせる(バブルをひとつ挿入する)。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
 - make
 - make run
 - cat mtrace.txt

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00020000
.equ DMEM_ADDR,  0x10000000
.align 4
.section .text.init
.globl _start
_start:
    li    x1, 9
    li    x2, DMEM_ADDR
    sw    x1, 0(x2)
    lw    x2, 0(x2)
    add   x3, x2, x2
    li    t0, TO_HOST
    li    t1, CMD_FINISH
    sw    t1, 0(t0)
L1: j     L1
```

```
./obj_dir/top
000000: 00000000 -----: 00000000:
000000: 00000004 00000000 -----: 00000000:
000000: 00000008 00000004 00000000 -----: 00000000:
000000: 0000000c 00000008 00000004 00000000 -----: 00000000:
000001: 00000010 0000000c 00000008 00000004 00000000: 00000000:
000002: 00000014 00000010 0000000c 00000008 00000004: 00000000:
000003: 00000014 00000010 ----- 0000000c 00000008: 00000000: stall
000003: 00000014 00000010 ----- 0000000c 00000008: 00000000: stall
000003: 00000014 00000010 ----- 0000000c 00000008: 00000000: stall
000003: 00000014 00000010 ----- 0000000c 00000008: 00000000: stall
000003: 00000014 00000010 ----- 0000000c 00000008: 00000009: stall
000004: 00000018 00000014 00000010 ----- 0000000c: 00000009:
000004: 0000001c 00000018 00000014 00000010 -----: 00000009:
000005: 00000020 0000001c 00000018 00000014 00000010: 00000009:
000006: 00000024 00000020 0000001c 00000018 00000014: 00000009:
000007: 00000028 00000024 00000020 0000001c 00000018: 00000009:
- top.v:47: Verilog $finish

==> mcycle : 18
==> minstret : 7
==> Total number of branch predictions : 0
==> Total number of branch mispredictions : 0
==> simulation finish!!
```

```
0000001: st 10000000 00000009 1111
0000003: ld 10000000 00000009
```



MA (Memory Access) stage

- ロード命令がロードしたデータを直後の命令が利用する場合には、データフォワーディングでオペランドを供給できない。そのため、直後の命令のデコードステージから実行ステージへの移動を1サイクル遅らせる(バブルをひとつ挿入する)。
- app/crt0.s を下図のコードに編集する。
- 次のコマンドで実行
 - make
 - make run
 - cat mtrace.txt

```
.equ TO_HOST,    0x80000000
.equ CMD_FINISH, 0x00200000
.equ DMEM_ADDR,  0x10000000
.align 4
.section .text.init
.globl _start
_start:
li    x1, 9
li    x2, DMEM_ADDR
sw    x1, 0(x2)
lw    x2, 0(x2)
add   x3, x2, x2
li    t0, TO_HOST
li    t1, CMD_FINISH
sw    t1, 0(t0)
L1: j    L1
```

```
./obj_dir/top
000000: 00000000 -----: 0 0 00000000:
000000: 00000004 -----: 0 0 00000000:
000000: 00000008 00000004 -----: 0 0 00000000:
000000: 0000000c 00000008 00000004 00000000 -----: 0 0 00000000:
000001: 00000010 0000000c 00000008 00000004 00000000: 1 0 00000000:
000002: 00000014 00000010 0000000c 00000008 00000004: 0 1 00000000:
000003: 00000014 00000010 -----: 0000000c 00000008: 0 0 00000000: stall
000003: 00000014 00000010 -----: 0000000c 00000008: 0 0 00000000: stall
000003: 00000014 00000010 -----: 0000000c 00000008: 0 0 00000000: stall
000003: 00000014 00000010 -----: 0000000c 00000008: 0 0 00000000: stall
000003: 00000014 00000010 -----: 0000000c 00000008: 0 0 00000009: stall
000004: 00000018 00000014 00000010 -----: 0000000c 00000000: 0 0 00000009:
000004: 0000001c 00000018 00000014 00000010 -----: 0 0 00000009:
000005: 00000020 0000001c 00000018 00000014 00000010: 0 0 00000009:
000006: 00000024 00000020 0000001c 00000018 00000014: 0 0 00000009:
000007: 00000028 00000024 00000020 0000001c 00000018: 0 0 00000009:
- top.v:47: Verilog $finish

==> mcycle : 18
==> minstret : 7
==> Total number of branch predictions : 0
==> Total number of branch mispredictions : 0
==> simulation finish!!
```

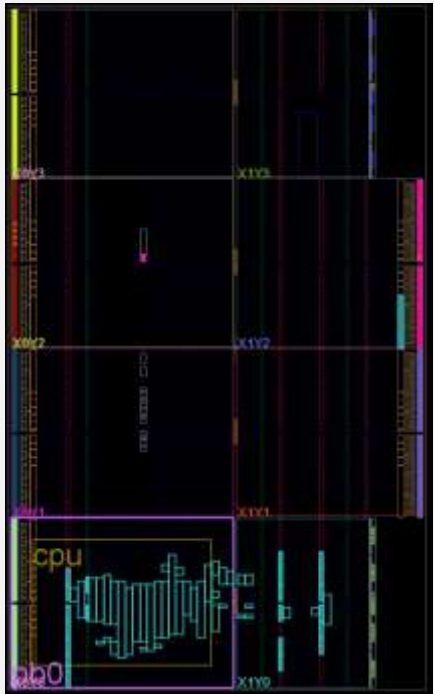
```
0000001: st 10000000 00000009 1111
0000003: ld 10000000 00000009
```



論理合成、配置・配線の結果

- Vivado 2024.1, xc7a100tcsg324-1
- `define CLK_FREQ_MHZ 120 // operating clock frequency in MHz
- `define IMEM_SIZE (32*1024) // instruction memory size in byte
- `define DMEM_SIZE (16*1024) // data memory size in byte

Resource	Utilization	Available	Utilization %
LUT	1652	63400	2.61
LUTRAM	44	19000	0.23
FF	912	126800	0.72
BRAM	19	135	14.07
DSP	4	240	1.67
IO	5	210	2.38
BUFG	2	32	6.25
MMCM	1	6	16.67

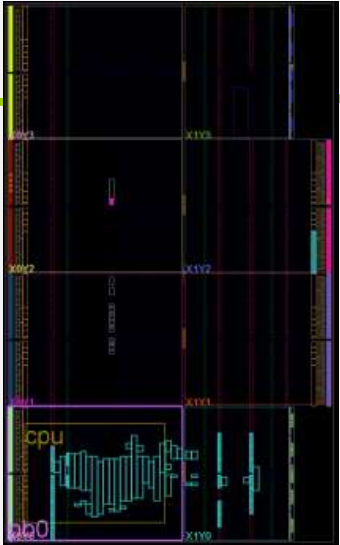


Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	0.661	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_3/ADDRARDADDR[14]	6.993	3.054	3.939	8.3
Path 2	0.677	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_7/ADDRARDADDR[14]	6.967	3.054	3.913	8.3
Path 3	0.696	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_5/ADDRARDADDR[9]	6.876	3.054	3.822	8.3
Path 4	0.696	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_7/ADDRARDADDR[12]	6.947	3.054	3.893	8.3
Path 5	0.706	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_5/ADDRARDADDR[12]	6.866	3.054	3.812	8.3
Path 6	0.722	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_3/ADDRARDADDR[12]	6.932	3.054	3.878	8.3
Path 7	0.745	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_3/ADDRARDADDR[6]	6.908	3.054	3.854	8.3
Path 8	0.763	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_4/ADDRARDADDR[12]	6.805	3.054	3.751	8.3
Path 9	0.781	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_5/ADDRARDADDR[6]	6.791	3.054	3.737	8.3
Path 10	0.784	3	10	cpu/bimodal/btb_reg/CLKBWRCLK	imem/rdata_reg_7/ADDRARDADDR[9]	6.859	3.054	3.805	8.3



論理合成、配置・配線の結果

- Vivado 2024.1, xc7a100tcsg324-1
- `define CLK_FREQ_MHZ 120 // operating clock frequency in MHz
- `define IMEM_SIZE (32*1024) // instruction memory size in byte
- `define DMEM_SIZE (16*1024) // data memory size in byte



Name	^1	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)	MMCME2_ADV (6)
main		1652	912	572	1608	44	19	4	5	2	1
clk_wiz_0 (clk_wiz_0)		0	0	0	0	0	0	0	0	2	1
cpu (cpu)		1383	550	449	1383	0	1	4	0	0	0
alu (alu)		122	0	54	122	0	0	0	0	0	0
bimodal (bimodal)		28	0	19	28	0	1	0	0	0	0
bru (bru)		117	0	51	117	0	0	0	0	0	0
divider (divider)		379	108	112	379	0	0	0	0	0	0
multiplier (multiplier)		95	38	48	95	0	0	4	0	0	0
store_unit (store_unit)		79	0	35	79	0	0	0	0	0	0
dmem (m_dmem)		43	45	19	43	0	4	0	0	0	0
imem (m_imem)		27	0	10	27	0	8	0	0	0	0
perf (perf_cntr)		31	98	40	31	0	0	0	0	0	0
st7789_disp (m_st7789_disp)		123	165	68	123	0	0	0	0	0	0
vmem (vmem)		2	52	15	2	0	6	0	0	0	0

