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

HotSPA2025 2025.06.10

藤野 蒼羽，吉瀬 謙二

東京科学大学 情報理工学院

背景

■ ASIP (Application-Specific Instruction-set Processor)

- 特定のアプリケーションの高速化を実現するプロセッサ
- アクセラレータをプロセッサに追加することで実現することが多い

■ ASIP開発における課題

- アプリケーション高速化に効果的なASIPの構成を見つけ出すのに時間がかかるケースがある
- 従来の設計手法では特定用途に対する最適化が不十分になる可能性がある

■ 本研究で焦点を当てる課題

- ASIP開発における課題に対処するためHW/SW協調フレームワークが数多く研究されている
- ファイルの依存関係が複雑でデバッグが難しい，ライセンス料が高く導入できない，使用方法を理解するまでのハードルの高さが課題

先行研究

■ CFU Playground[1]

- 先述の課題に取り組むオープンソースで最新のHW/SW協調設計フレームワーク

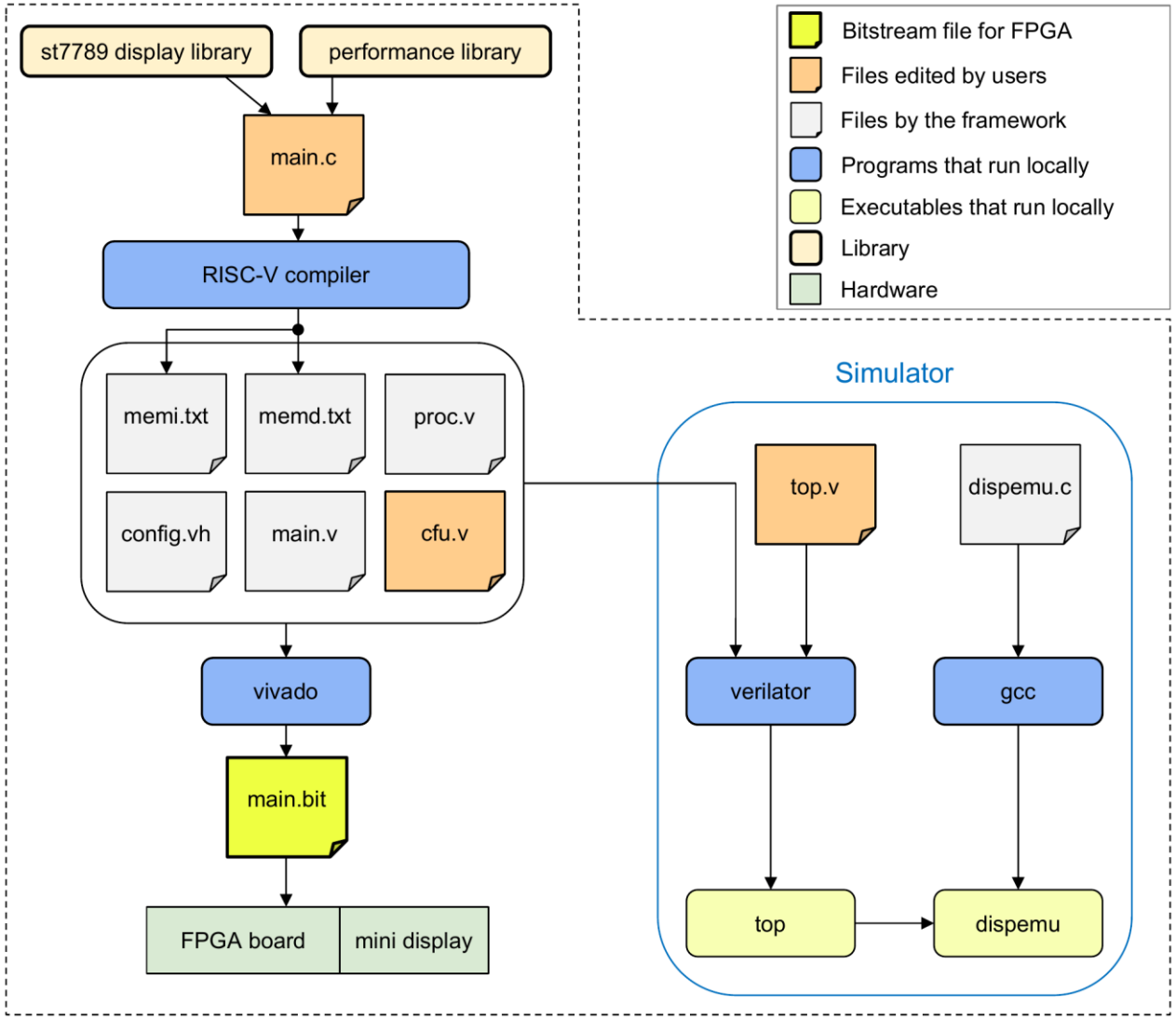
■ 先行研究の課題

- 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: 概要

- 本研究の位置づけ
  - CFU Playgroundの課題に対処するための新たなHW/SW協調設計フレームワーク，CFU Proving Groundを提案する
- 主な貢献
  - 計23ファイル2681行のコードで構成され，この軽量な実装によりデバッグが容易
  - 少ない言語による実装と主流のツールチェーンの使用によって操作方法の理解が容易
  - すべてのファイルをMITライセンスの下でオープンソースとして公開*



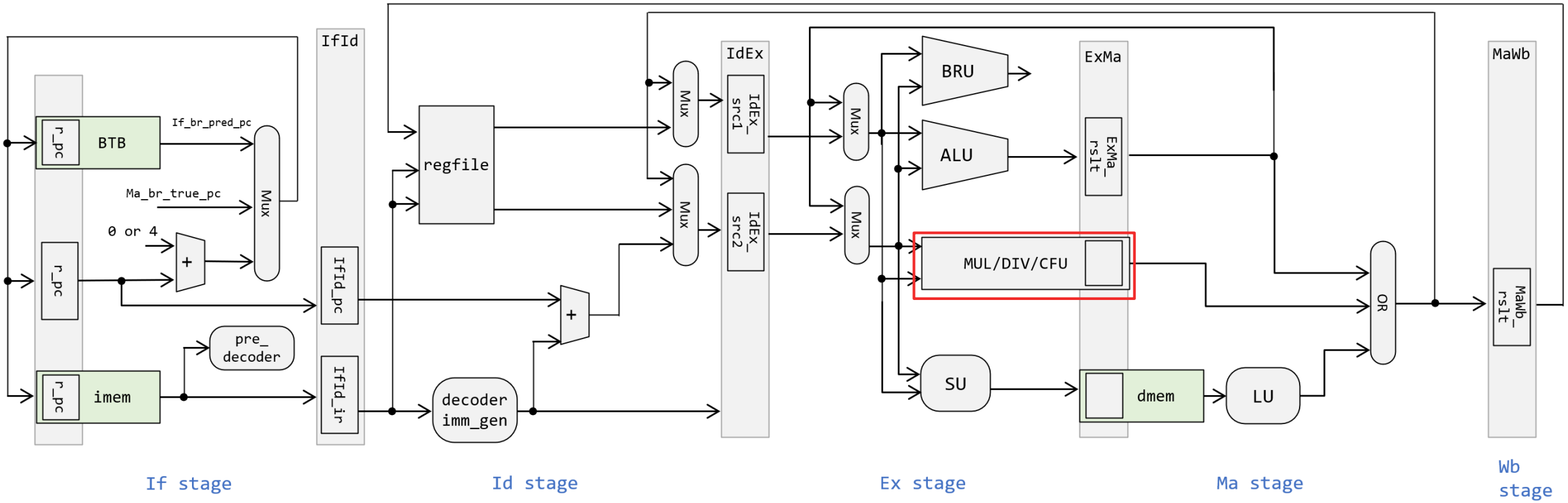
CFU Proving Groundの設計フロー

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

CFU Proving Ground: RVProc

■ RVProcの構成

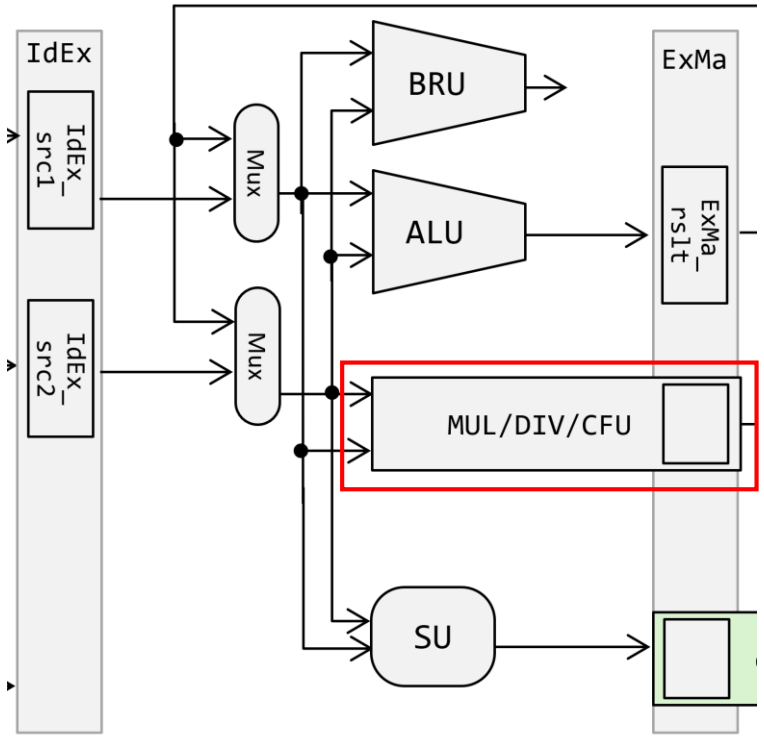
- 5段パイプランのRV32IMプロセッサ
- bimodal分岐予測器を搭載

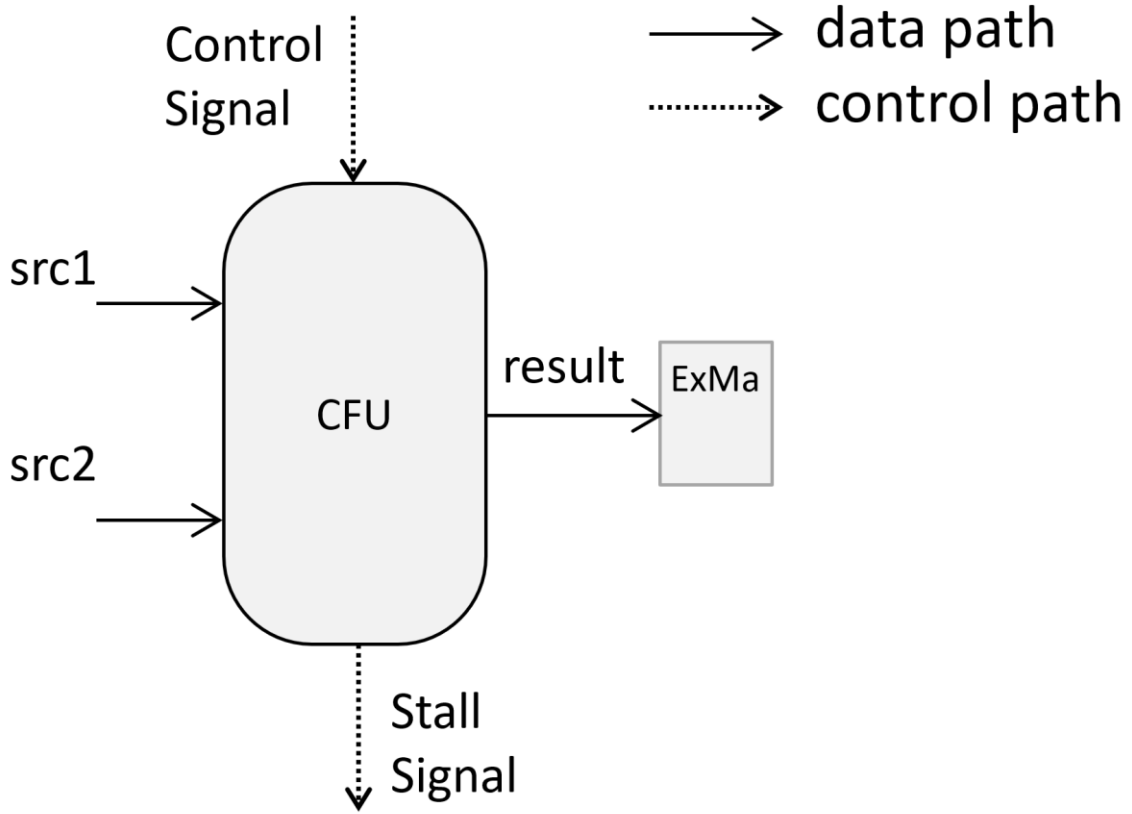


CFU Proving Ground: CFU

■ Custom Function Unit (CFU)

- 本フレームワークにおけるアクセラレータの呼称
- 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: カスタム命令

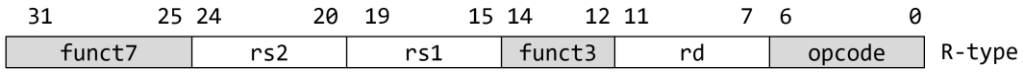
■ CFUを制御するためにカスタム命令を活用する

- 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形式のフィールドを採用する



CFU Proving Ground: プログラムの記述

■ カスタム命令をC言語から使用するためには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関数の記述例

```
static inline unsigned int cfu_op(unsigned int funct7,
    unsigned int funct3, unsigned int rs1, unsigned int rs2) {
    unsigned int result;
    asm volatile(
        ".insn r CUSTOM_0, %3, %4, %0, %1, %2"
        : "=r"(result)
        : "r"(rs1), "r"(rs2), "i"(funct3), "i"(funct7)
        :
    );
    return result;
}
```

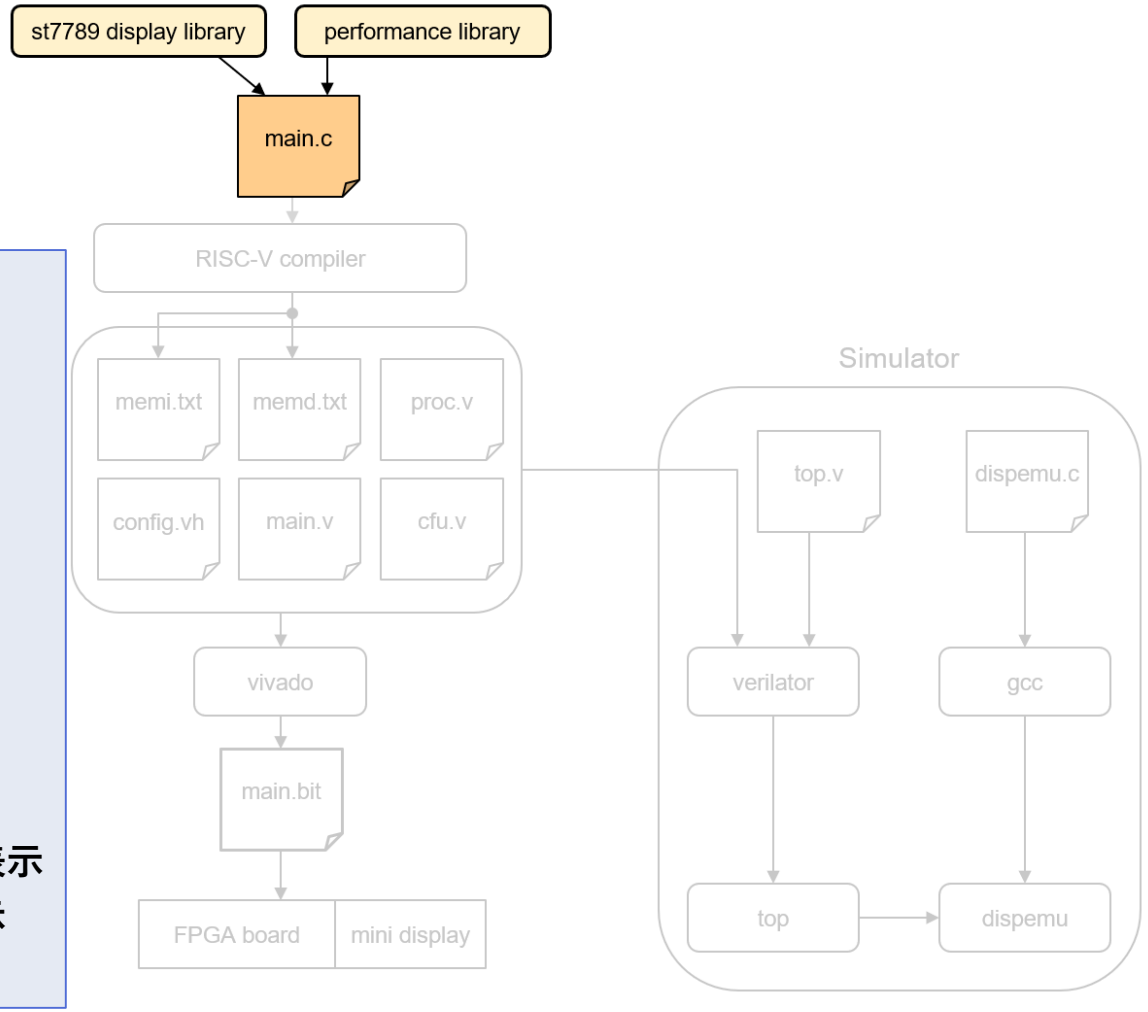
```
rslt = cfu_op(funct7, funct3, in1, in2);
```

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");
```

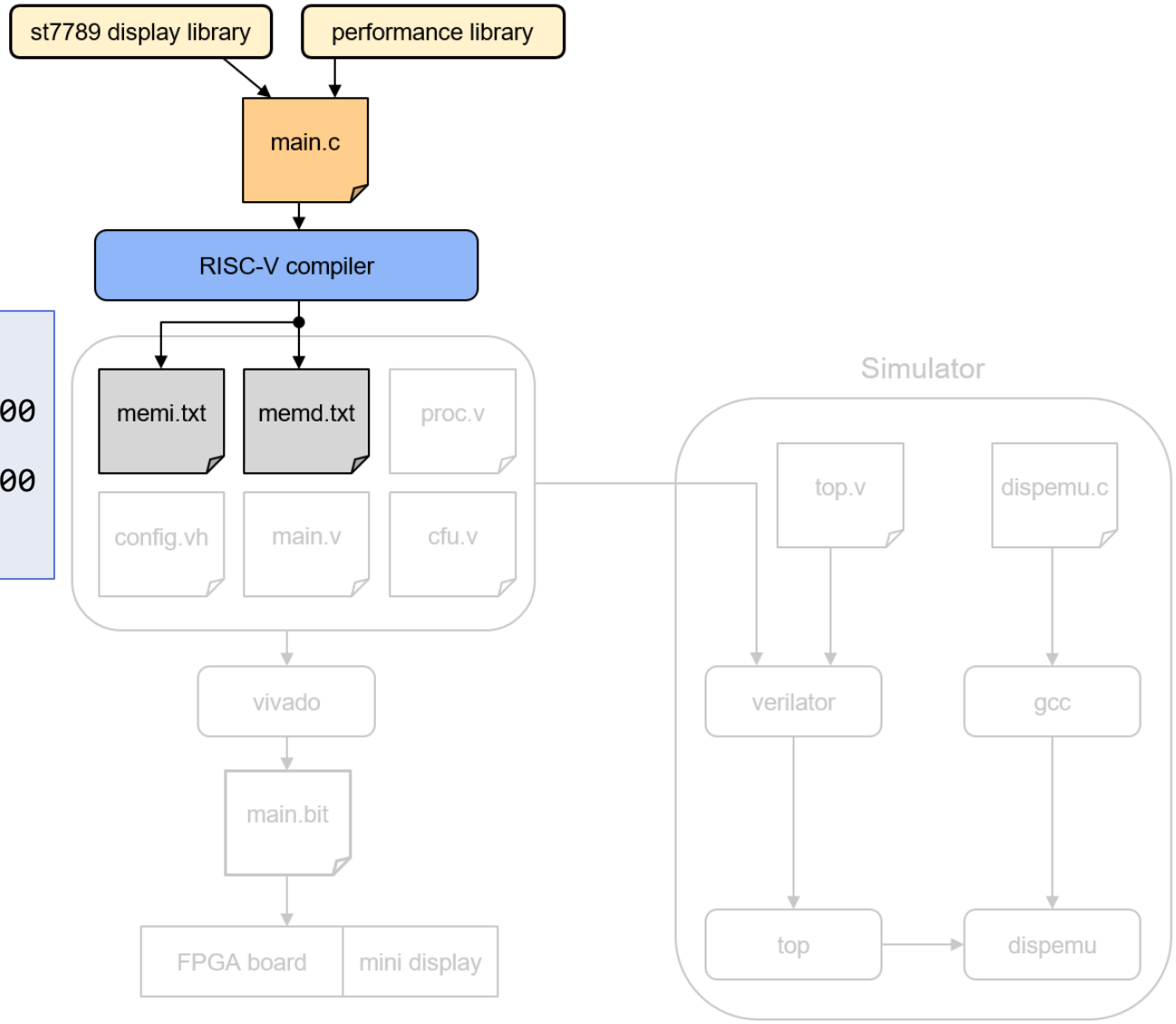
測定するスコープ



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イメージの作成

```
$ make prog
```



● link.ldのメモリサイズの指定箇所

```
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

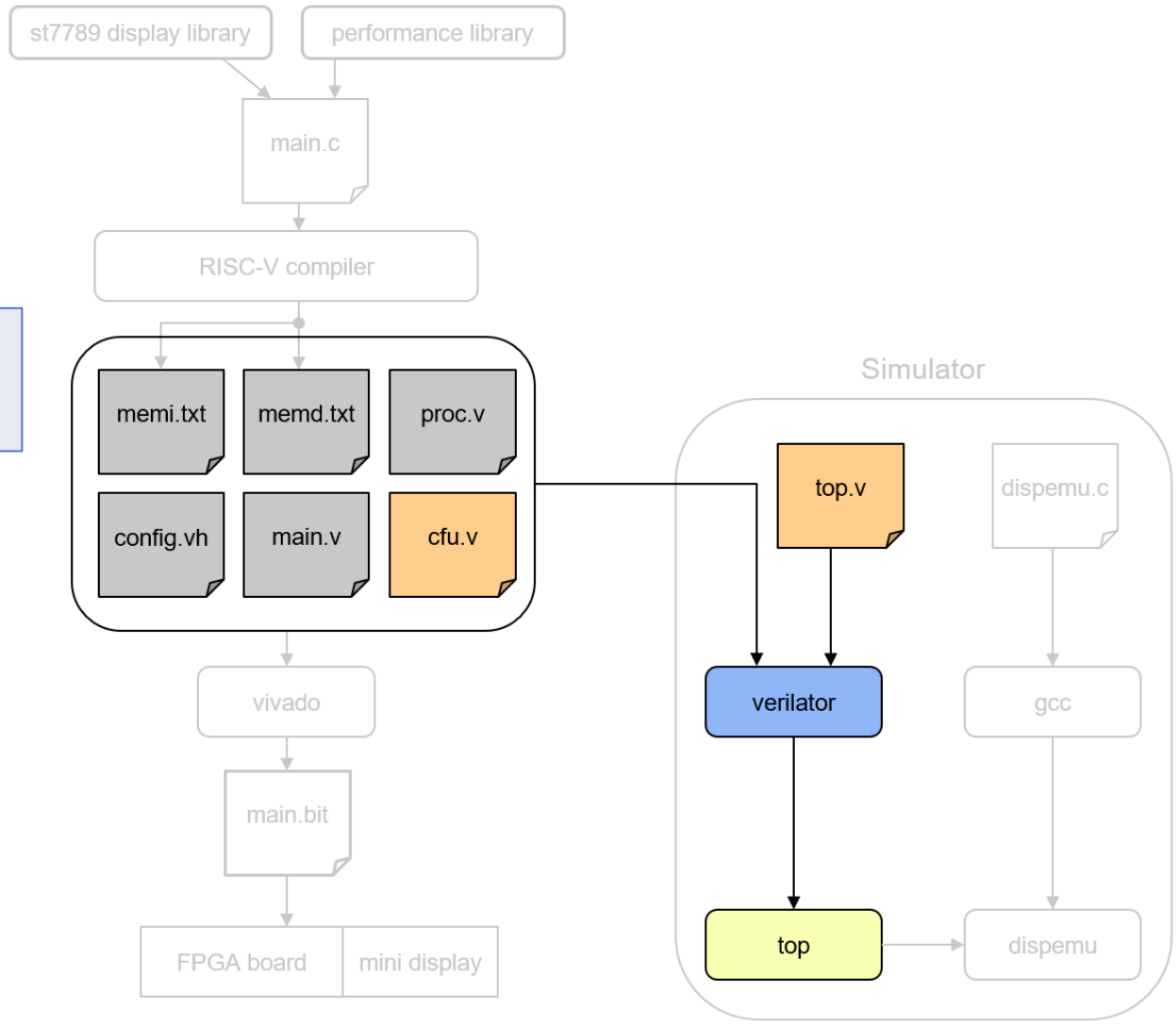
```
// instruction memory size in byte
`define IMEM_SIZE       (32*1024)

// 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*でコンパイルする

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

```
$ make build
```



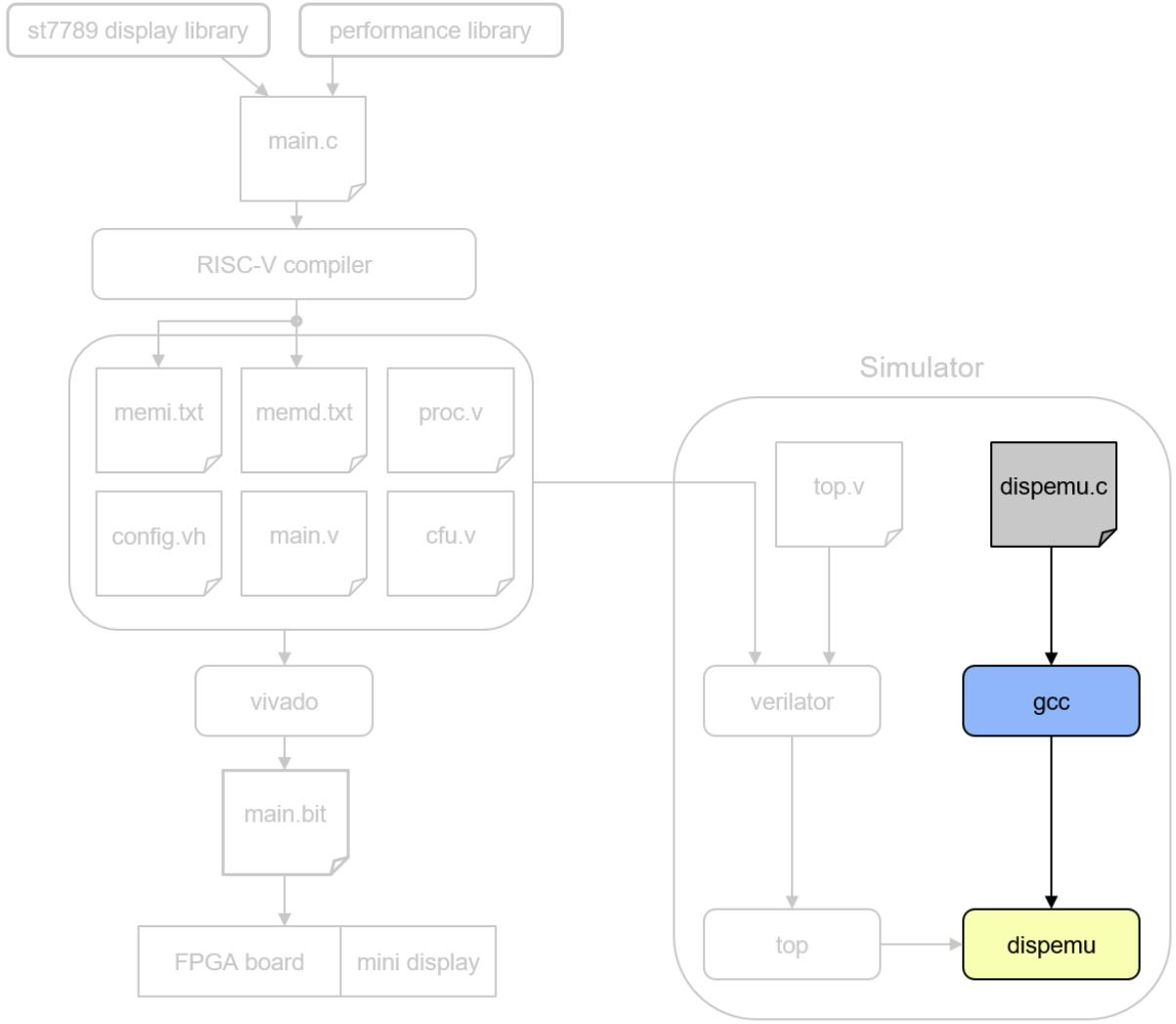
* https://github.com/verilator/verilator

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
```

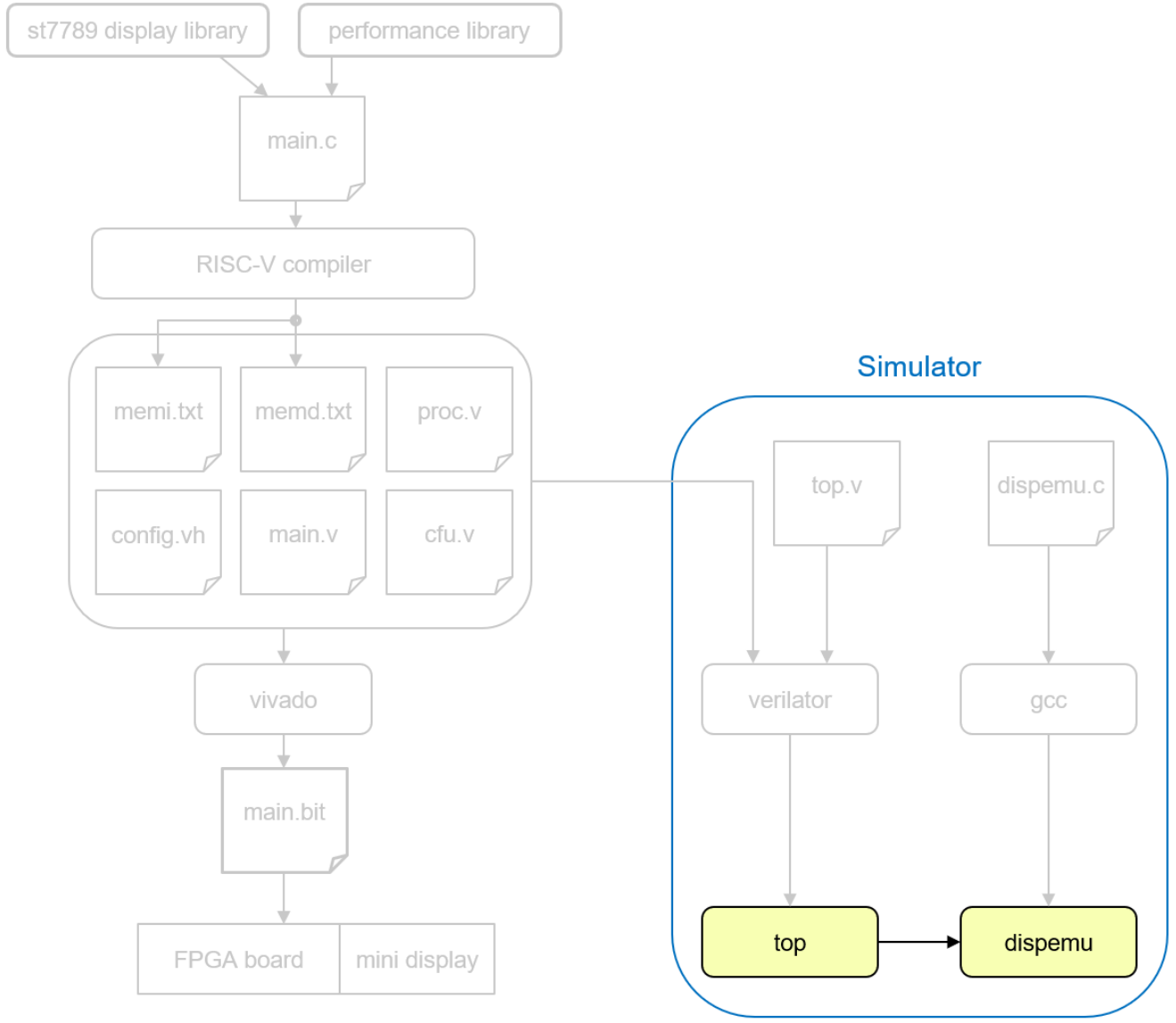


ソフトウェア: 実行クロックサイクルの評価

- 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
```

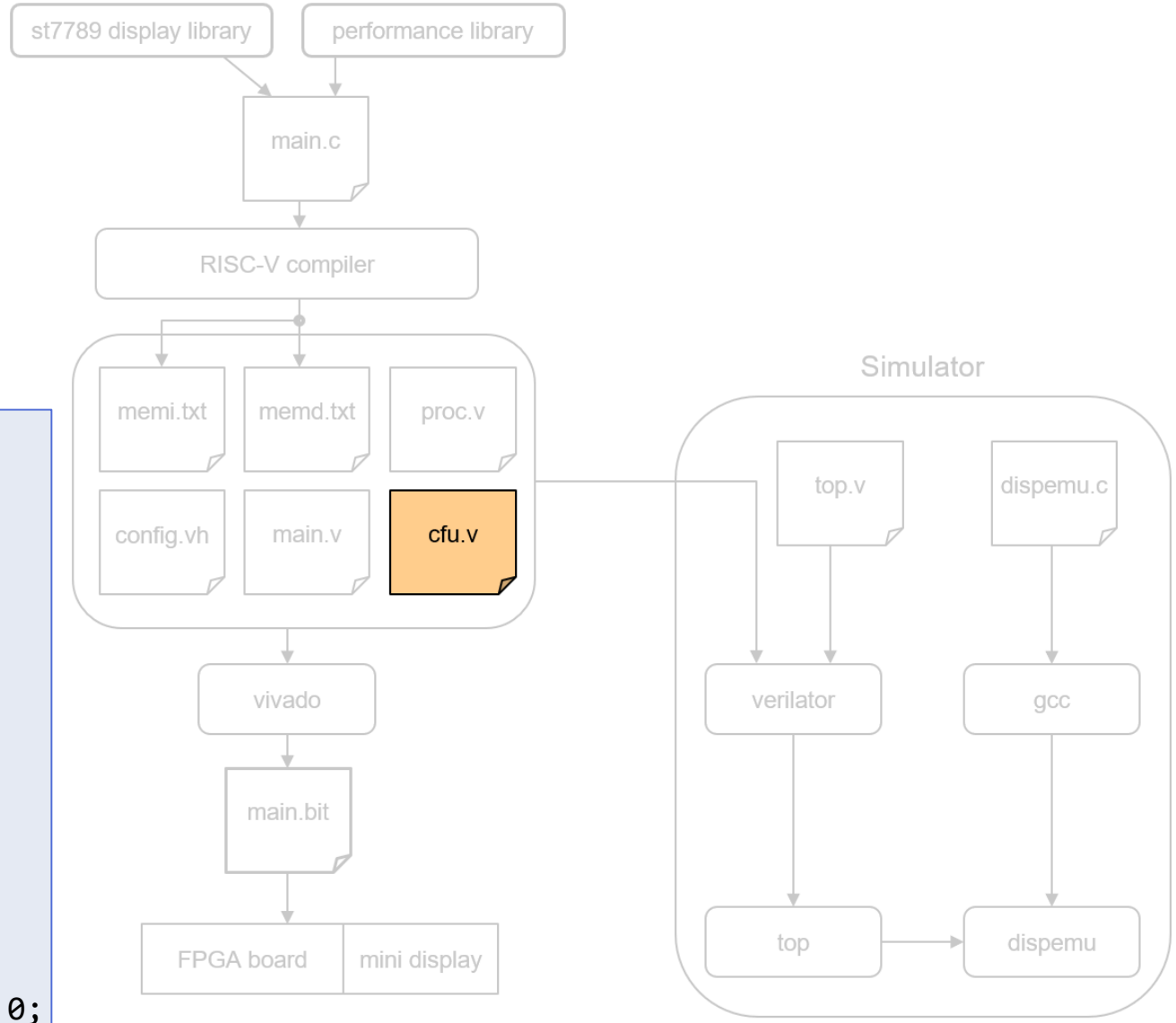
```
$ make drun
```



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
```

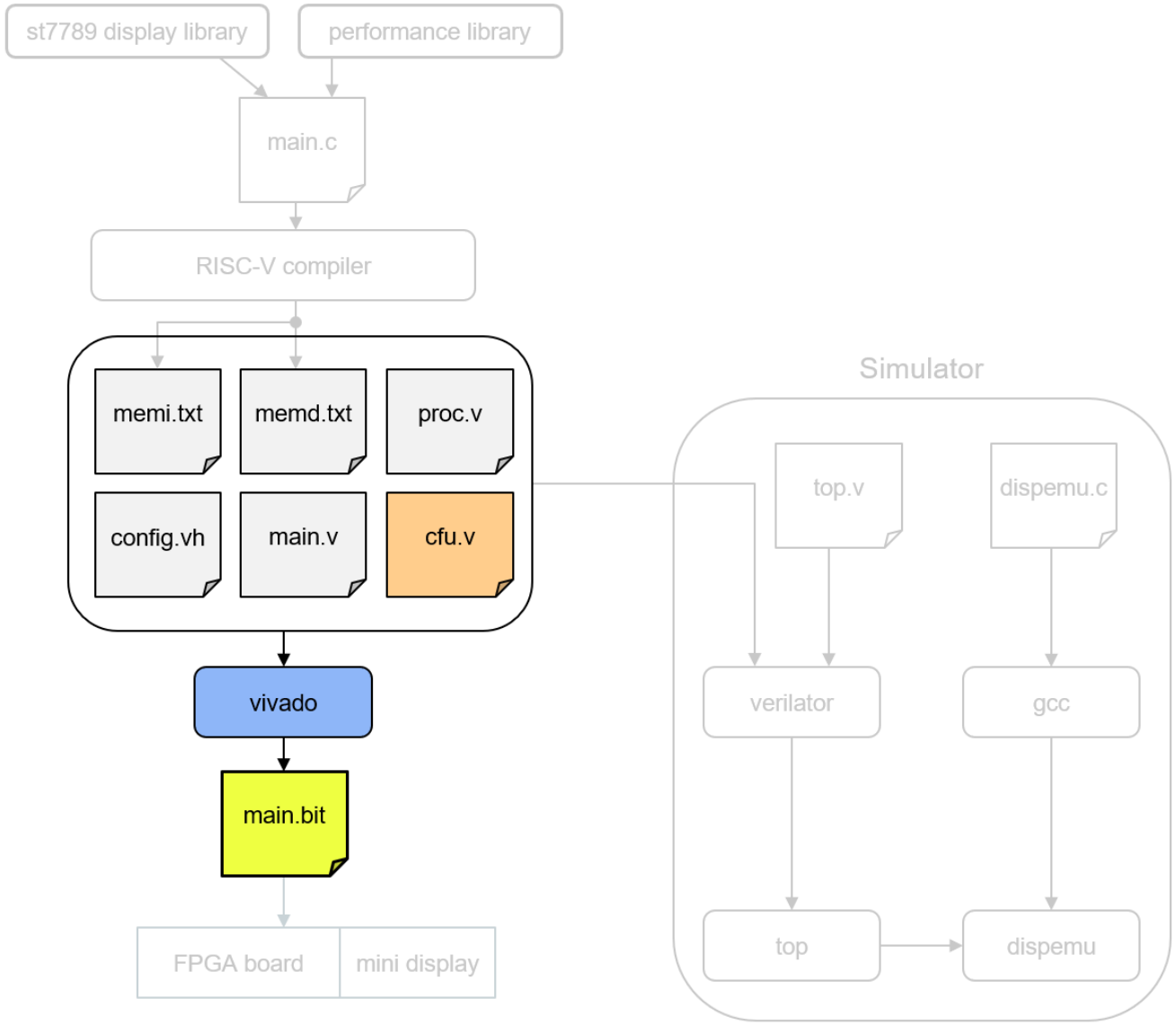


ハードウェア: bitstreamの合成

- 目的：FPGAをコンフィギュレーションするためにVivadoでbitstreamを合成する

- 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
```

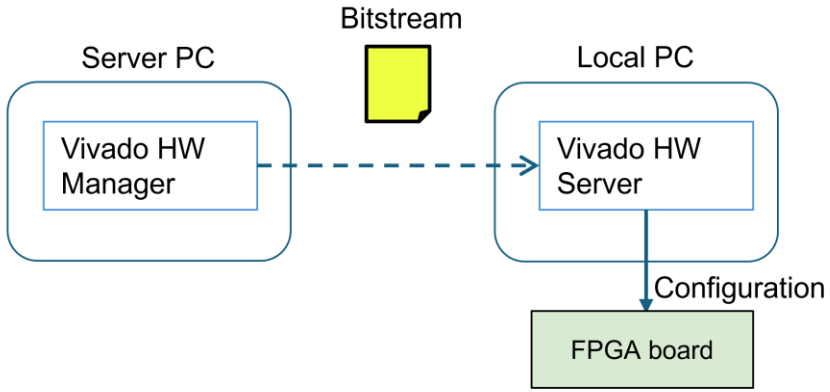


ハードウェア: FPGAへのロード

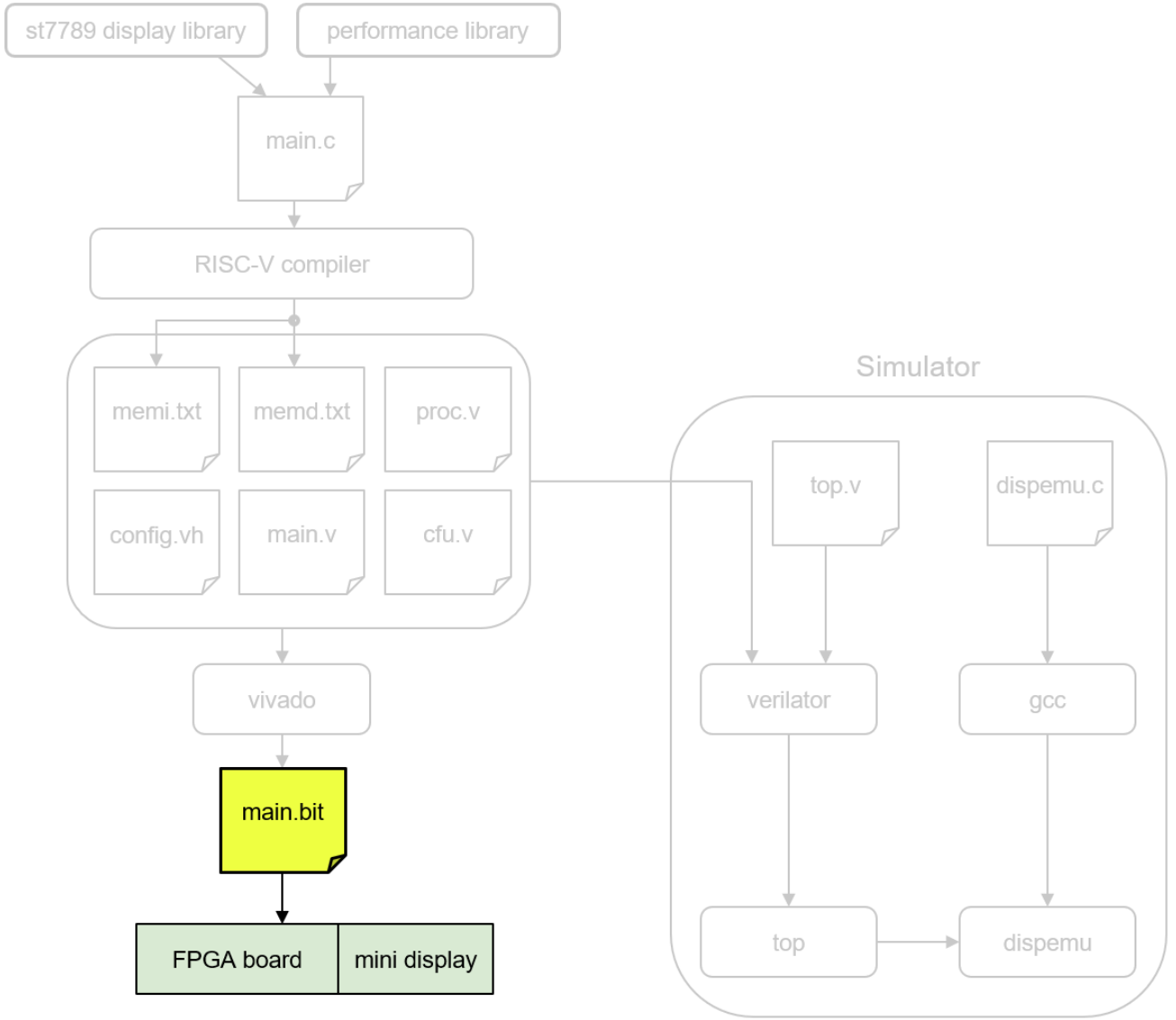
- 目的：合成したbitstreamをVivadoでFPGAにコンフィギュレーションする
  - scripts/prog_dev.tcl

```
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
```



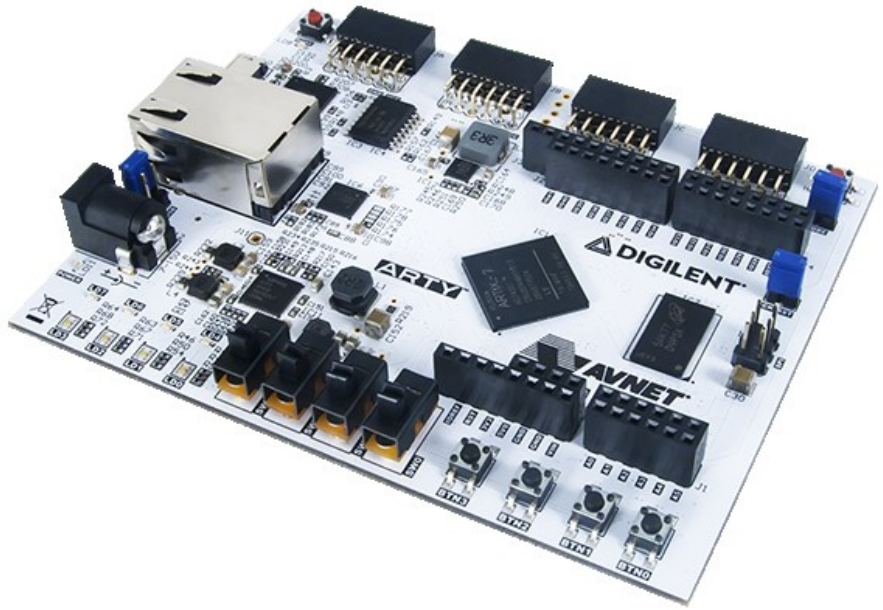
評価: 概要

■ 開発環境

- Intel Core i9-12900KF / 128 GiB DDR4
- Ubuntu22.04.2 LTS
- AMD Xilinx Vivado 2024.2

■ 評価環境

- Arty A7-35T



評価: CoreMark

■ CoreMark

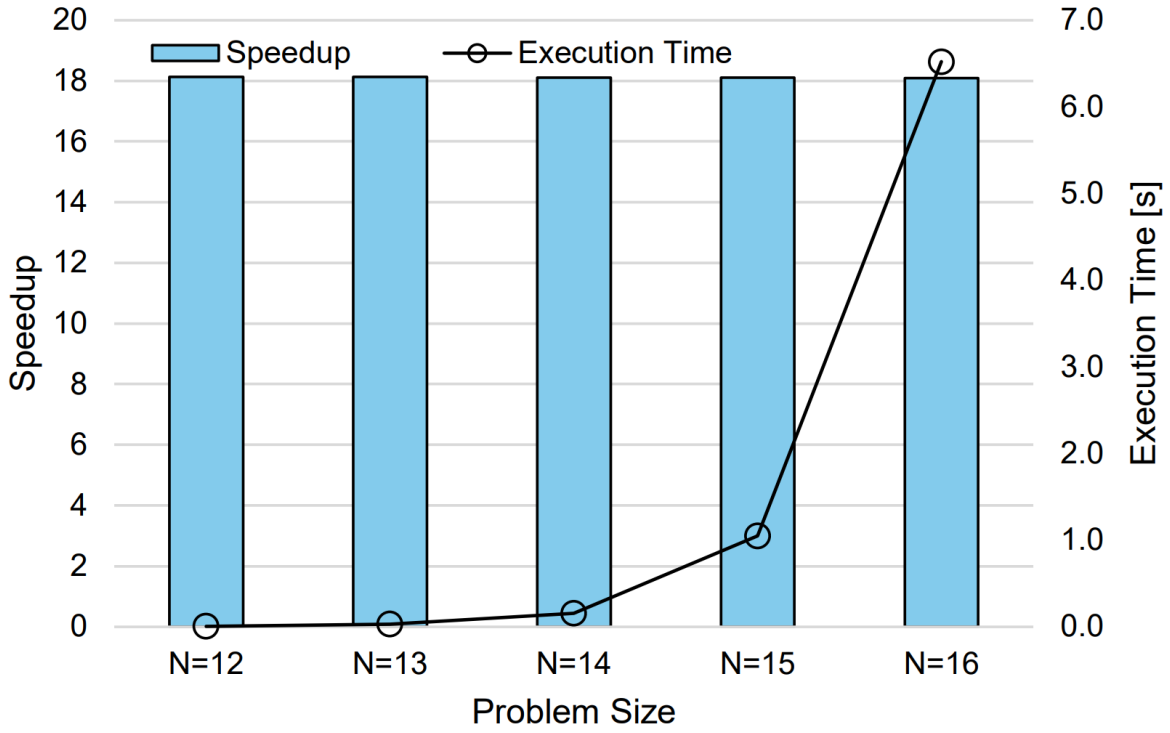
- VexRiscvの評価環境に合わせて測定

Processer	F_{max} [MHz]	CoreMark/MHz	CoreMark
ESP32-C3[2]	160	2.55	408
VexRiscv full mex perf[3]	200	2.57	514
RVProc (proposal)	235	2.65	622

[2] ESPRESSIF, “ESP32-C3 Series Datasheet Version 2.0,” 2024.
[3] SpinalHDL, “VexRiscv,” https://github.com/SpinalHDL/ VexRiscv, 2025. GitHub repository. Accessed: March 7, 2025.

評価: N-Queens

■ カーネルをCFUとして実装し，本フレームワークのデフォルトの構成から実行クロックサイクルに関して平均18倍の高速化



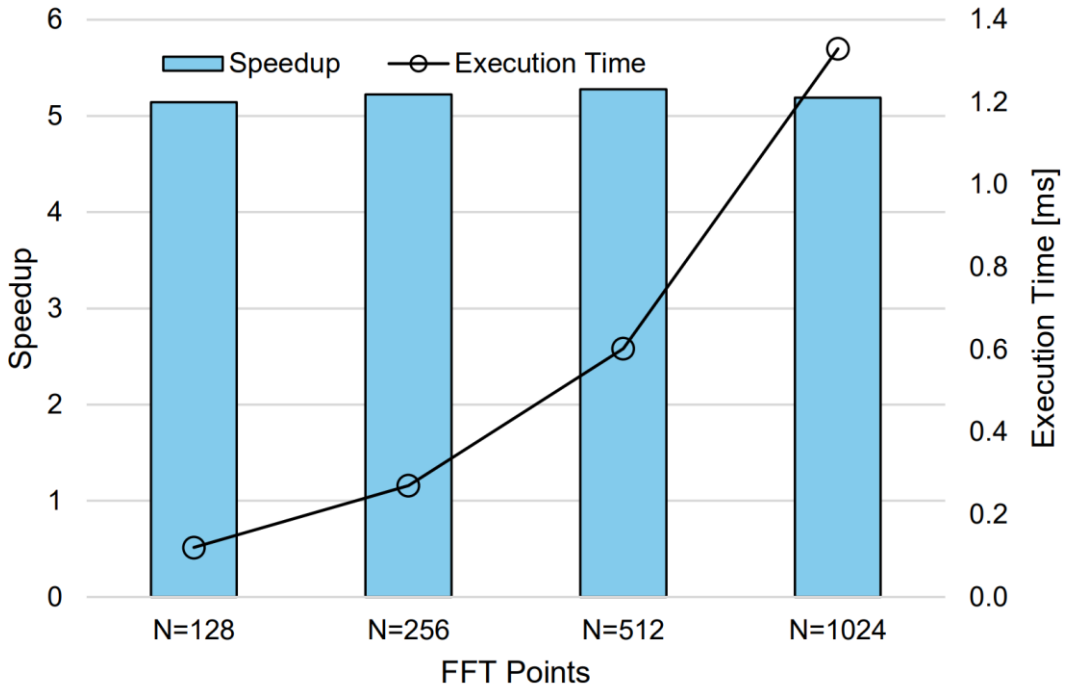
これが1命令に😊

```
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;
}
```

CFUにオフロードするベースラインのCプログラム

評価: 高速フーリエ変換

- 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で実装し，本フレームワークのデフォルトの構成から実行クロックサイクルに関して平均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);
```

CFUにオフロードするベースラインのCプログラム

評価: ハードウェア使用量

■ デフォルト（32KiB, 16KiB）

- 命令メモリ32KiB，データメモリ16KiBの構成

Resouces	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

Resouces	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

Resouces	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%

関連研究

- Chipyard[4]
  - 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[5]
  - 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[6]
  - 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

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

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

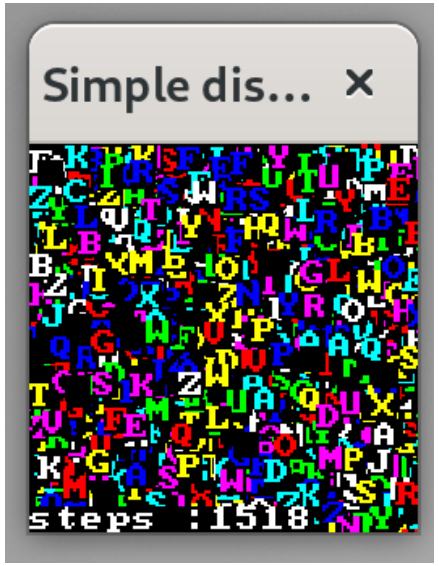
[6] K. Hepola et al., ” 2022 IEEE 33rd International Conference on Applicationspecific Systems, Architectures and Processors (ASAP), pp.161–165, 2022.

まとめ

- 本研究では新しいHW/SW協調設計フレームワークCFU Proving Groundを紹介した
- 本フレームワークを利用することで，N-queensのケースで平均18倍，高速フーリエ変換のケースで平均5.2倍の高速化を達成した
- 今後は，高位合成のサポートや本フレームワークの応用方法の模索が主な研究方針となる
- ファイル依存の複雑性，ライセンス依存の少なさ，使用方法を理解するまでのハードルについて定量的に評価する指標を考察する必要がある

Try Me😊

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



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