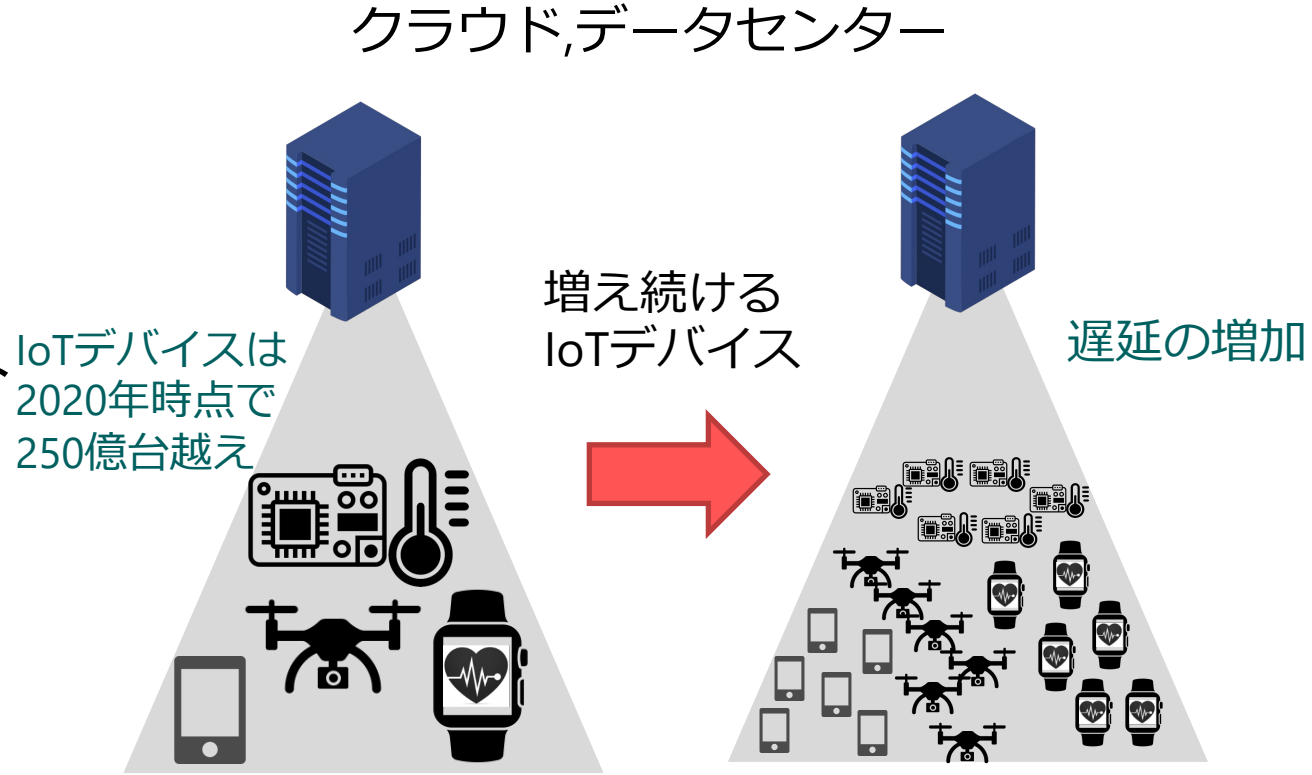
# FPGAとRISC-Vプロセッサを搭載したSoC向けHW/SW設計フローと実機評価

小島　拓也[1,2], 矢内 洋祐[3] , 天野 英晴[3], 奥原　颯[4]
久我 守弘[5] , 飯田 全広[5]

[1]東京大学, [2]JSTさきがけ, [3]慶應義塾大学,
[4]シンガポール国立大, [5]熊本大学

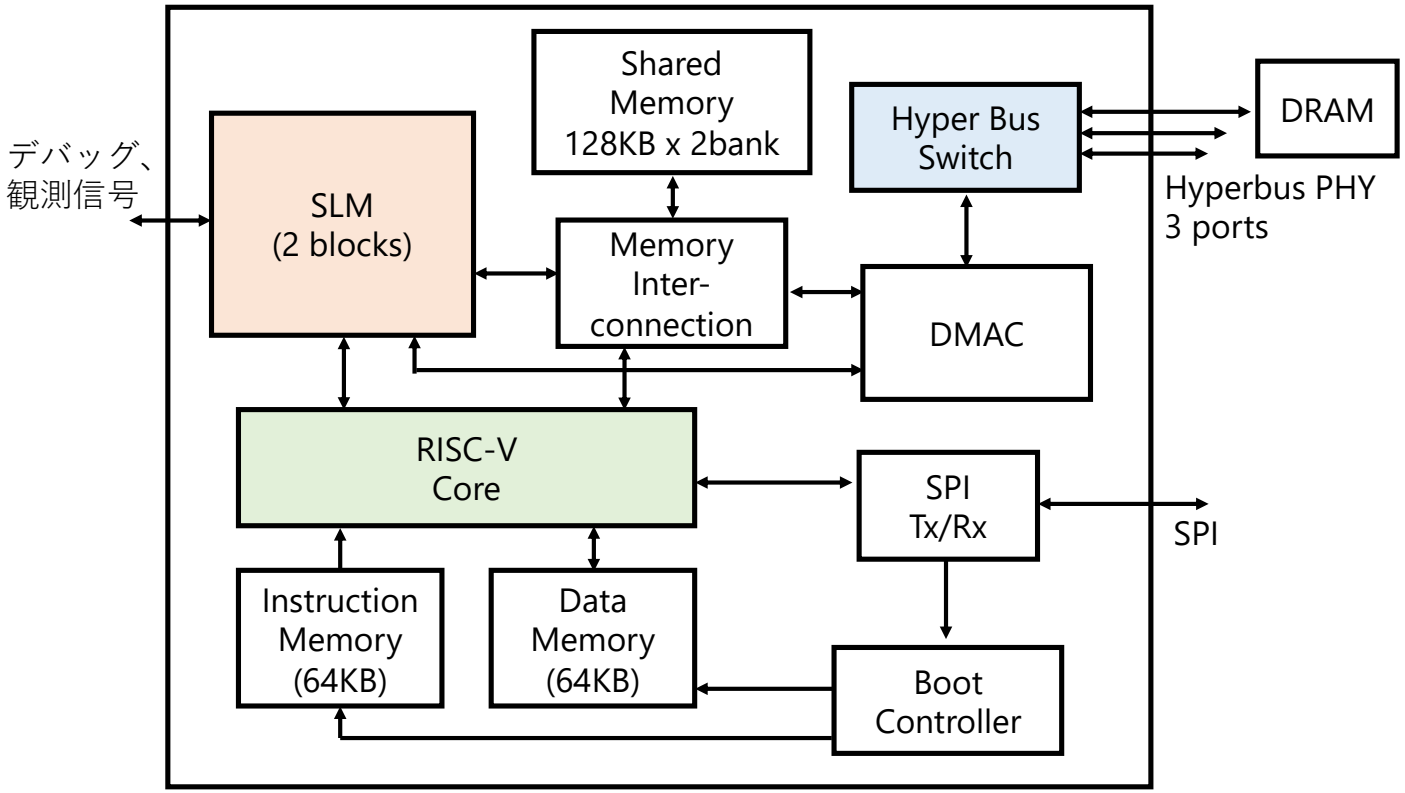
# エッジコンピューティングへのパラダイムシフト

- 従来のクラウド集約型コンピューティングの限界
  - データ転送量の増加
  - クラウド側の計算負荷が増加
  - 処理レイテンシーの増大
- エッジコンピューティング
  - クラウドの代わりにエンドポイントのデバイスで一部の処理を行う
- デバイスに対する要求
  - 高い処理スループット性
  - 低遅延性
  - 省電力性



# SLMLET: eFPGA・CPU混載SoC

- CPU: RISC-V RV32I
  - riscv-mini (ucbが公開, chisel実装)
  - コントローラとして利用
  - 命令、データ専用メモリ (各64KB)
- SLM (Scalable Logic Module)
  - 再構成可能なロジック
- HyperBus I/F
  - JEDEC xSPI 準拠
- 2バンクの共有メモリ
  - 各128KB
- SPI
  - ブートおよびペリフェラル用
  - 4チャネル



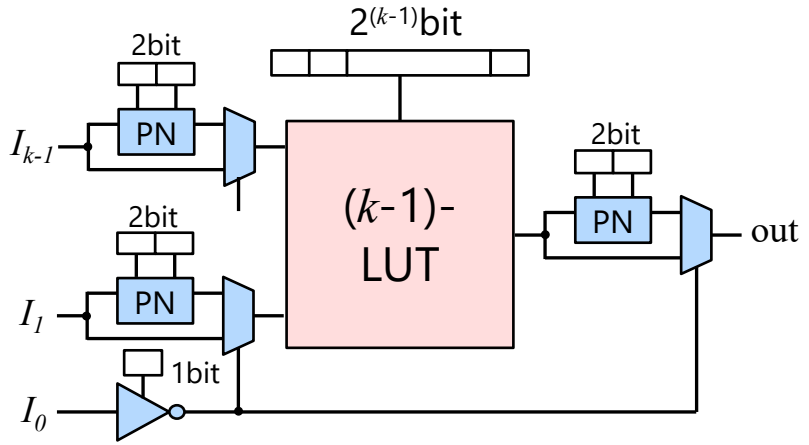

**SLMLET SoCの構成**

# 本発表の概略と流れ

- 概略
  - ソフトウェアライブラリの整備
  - FPGA CADとの連携したアプリケーション開発環境の実現
  - ケーススタディとしてCRC32を例に実機測定
    - v.s マイコン(ESP32, Raspberry Pi Pico): エネルギー削減49-78%, レイテンシ削減 39-46%
    - v.s ディスクリート型FPGA (Tang Nano 9K): 待機電力がおよそ50分の1
- 流れ
  1. SLM再構成ロジックと試作チップ
  2. HW/SW設計フロー
  3. 評価ボードと測定用システムの構築
  4. 実機評価の報告

# SLM再構成ロジックによるFPGA

- Scalable Logic Module(SLM)
  - 省構成メモリなロジックセル
  - K入力ロジックを(K-1)-LUTとProgrammable NANDによって実現
  - シャノン展開とNPN同値変換を利用
- SLMを用いたFPGAのIP生成
  - SLMの入力サイズ、ロジックセル数などがカスタマイズ可能なフレームワーク[1]
  - 相互接続のスイッチブロックはWilton型




K入力ロジックのためのSLMブロック

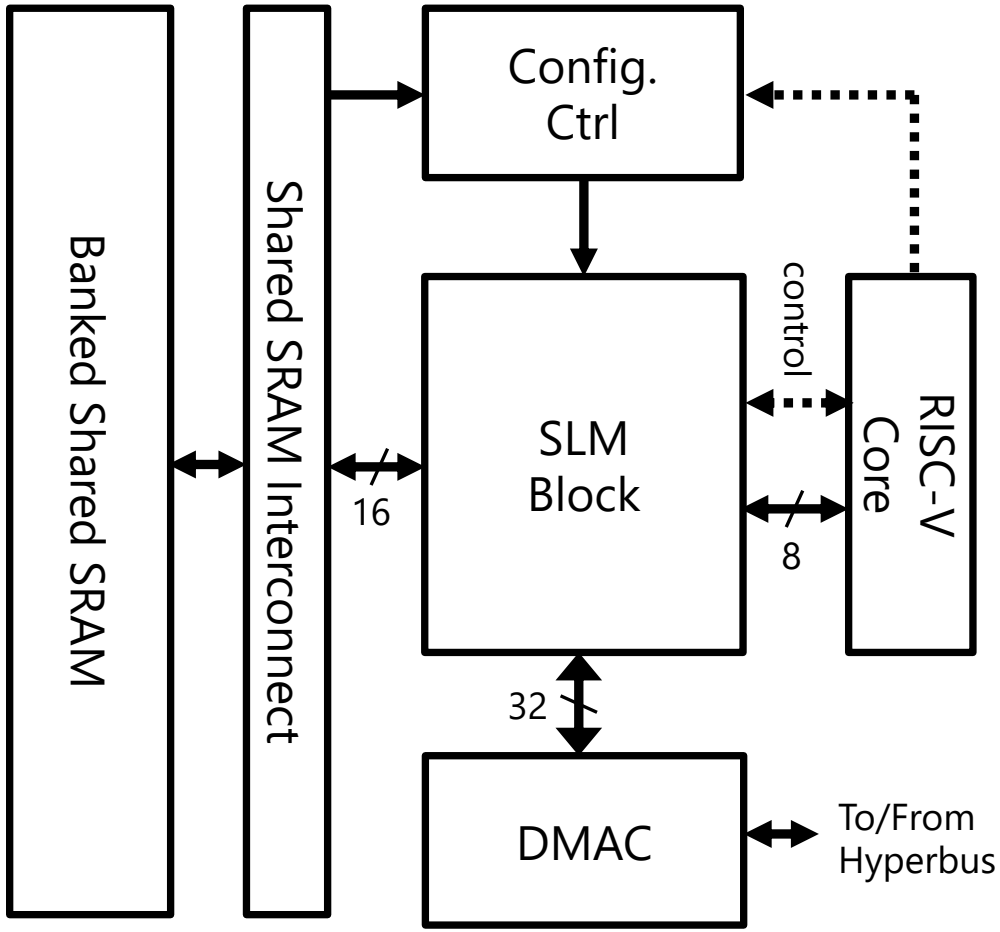
SLMLETに使用されたIPのパラメータ

| SLM構成 | セル数 / クラスタ | クラスタ数 | 総SLM数 | DSPブロック | 総FF数 |
|---|---|---|---|---|---|
| 5-SLM | 4 BLE (SLM) | 224 | 896 | 8 | 1024 bit |

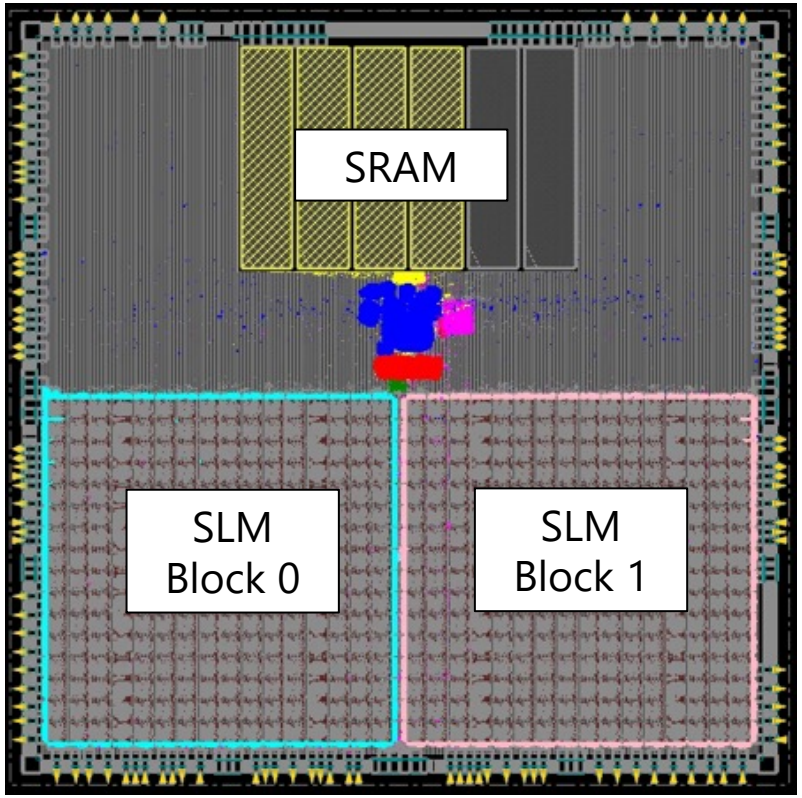

[1] Kuga, Morihiro, et al. "An eFPGA Generation Suite with Customizable Architecture and IDE." *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 106.3 (2023): 560-574.

# SLMブロックと周辺モジュール

- コンフィギュレーションコントローラ
  - RISC-Vコアからの指示に従い、共有メモリ上のビットストリームをSLMブロックに書き込み
  - ブロックごとに存在 → 並列再構成が可能
- 共有SRAM用インターコネクト
  - 各モジュールが排他的に共有SRAMへアクセス
- データ入出力用インターフェース
  - RISC-Vのメモリマップト IOとして8bit
  - 共有SRAMへ直接アクセスするための16bit
  - DMACコントローラと直結した32bit
    - 外部DRAMを使用可能

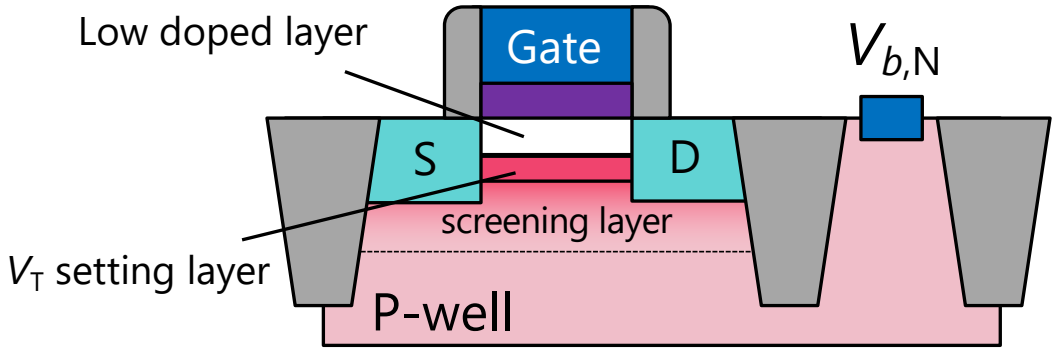


# SLMLETのプロトタイプ実装



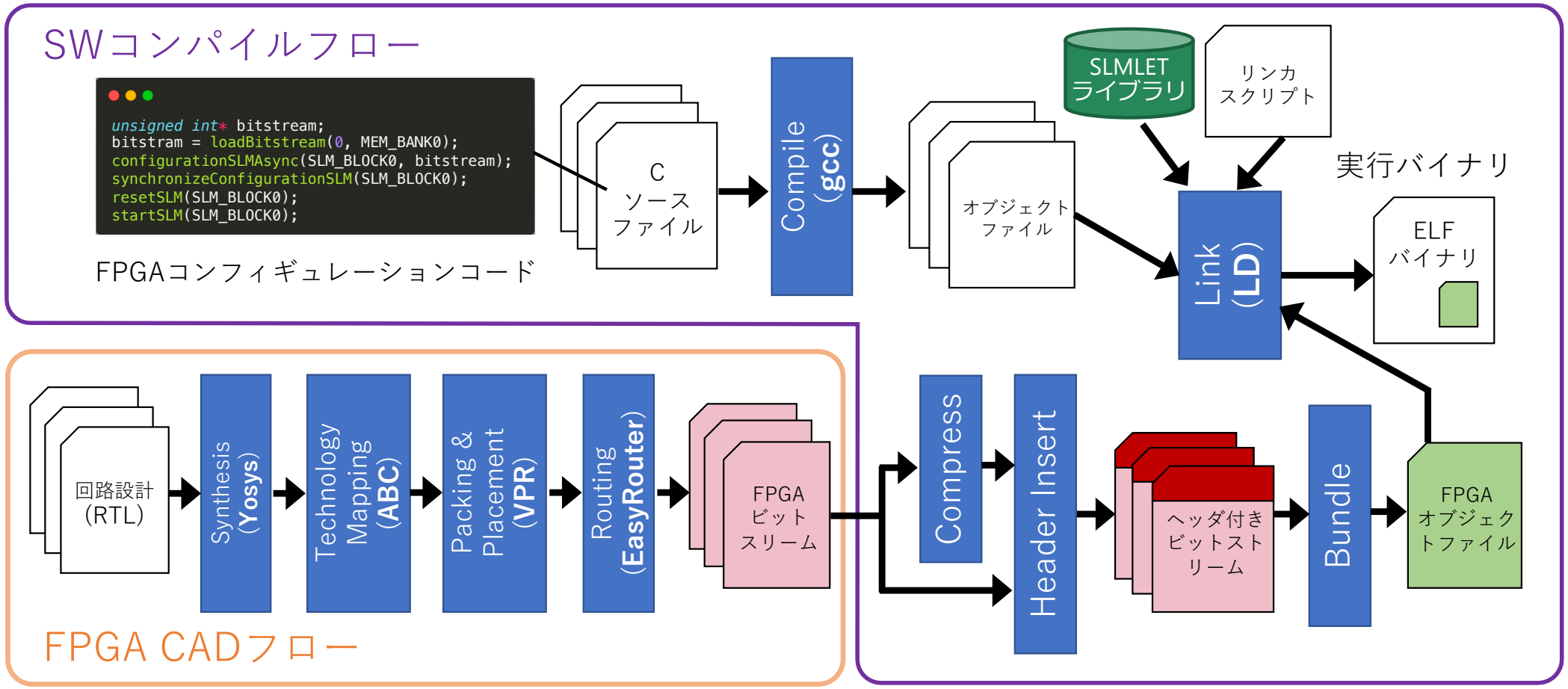

**チップレイアウト 4.2mm角**

- USCJ 55nmを採用
  - 閾値電圧のばらつきが小さい
  - 240mV/V の高い基板効果係数
    → ボディバイアス制御によるリーク削減の期待
- スタンダードセル: C55DDCT07L60LVT




**USCJプロセスのトランジスタ構造**

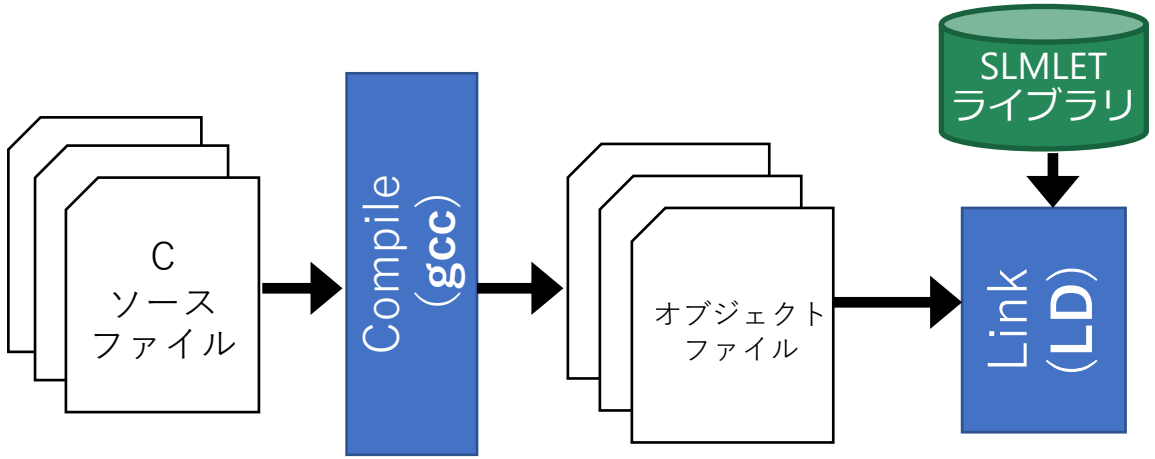
# HW/SW設計フロー: 全体像



# HW/SW設計フロー: SW記述

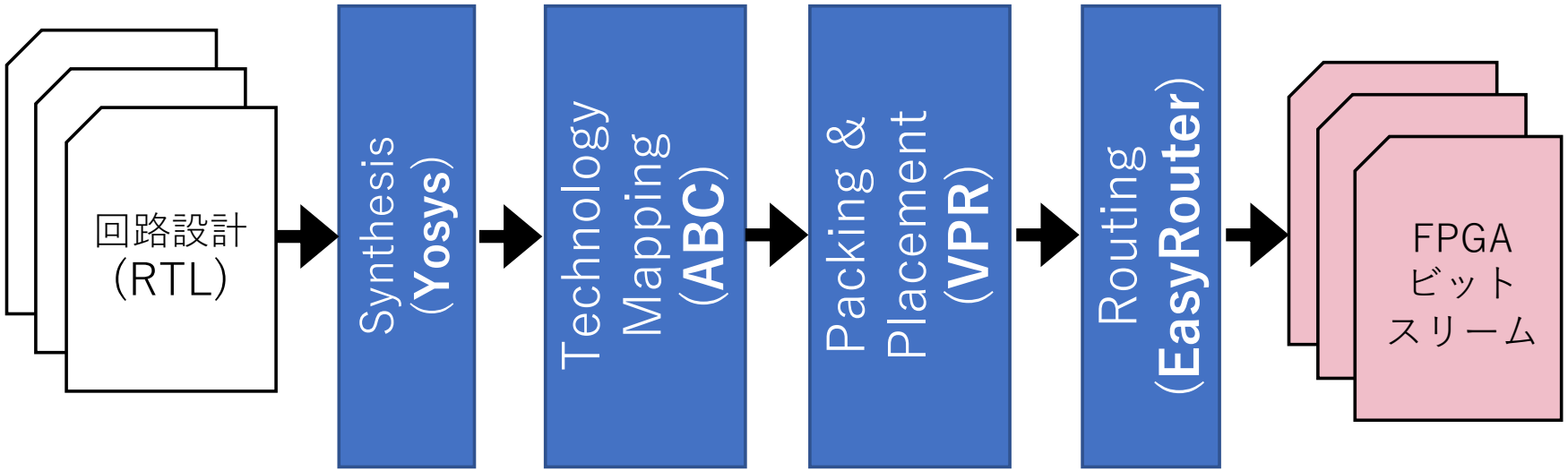
```
unsigned int* bitstream;
bitstram = loadBitstream(0, MEM_BANK0);
configurationSLMAsync(SLM_BLOCK0, bitstream);
synchronizeConfigurationSLM(SLM_BLOCK0);
resetSLM(SLM_BLOCK0);
startSLM(SLM_BLOCK0);
```

FPGAコンフィギュレーションコード




- RISC-Vコアで実行されるコードはC言語で記述
- SLMブロック制御
  - ライブラリが提供する関数(後述)を呼び出す

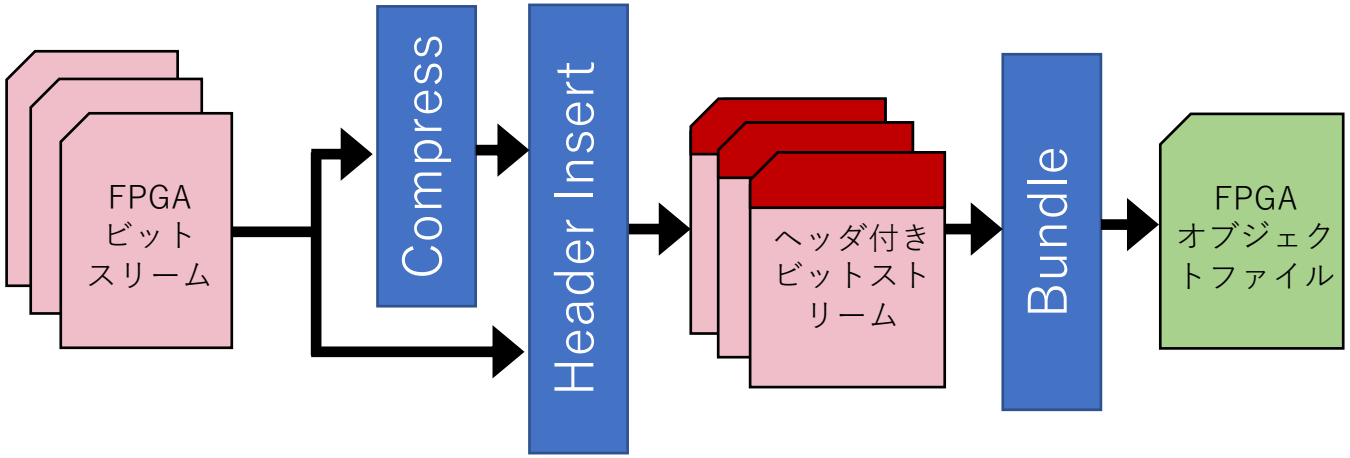
# HW/SW設計フロー: HW設計




- FPGA部に構成するハードウェアの設計
  - 先行研究[1]で整備されたFPGA CADを使用
  - OpenFPGAなど広く使用されるYosys, ABC, VPRにEasyRouter[2]を組み合わせ

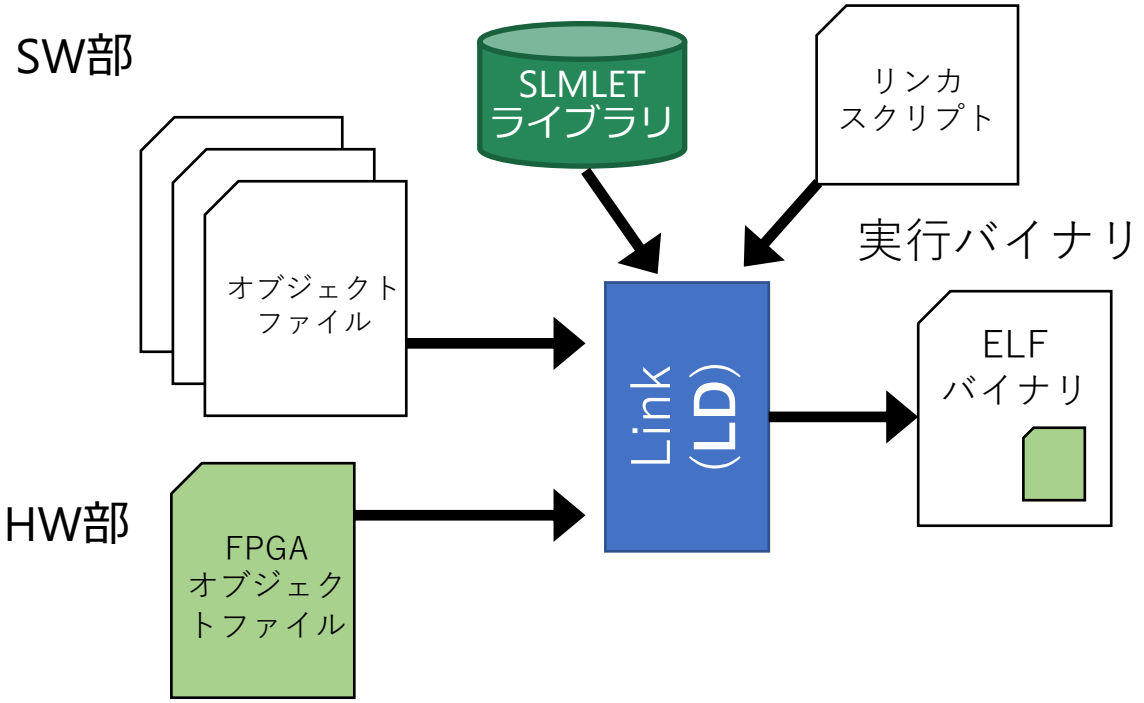
[2] Zhao, Qian, et al. "An automatic FPGA design and implementation framework." 2013 23rd International Conference on Field programmable Logic and Applications. IEEE, 2013.

# HW/SW設計フロー: ビットストリームのオブジェクト化




- ビットストリームに対する前処理
  - Tag-Less Compressionによる圧縮 (オプション)
  - ヘッダー挿入
- バンドル化
  - 複数のビットストリームを一つのオブジェクトファイルにまとめる

# HW/SW設計フロー: 実行ファイルの生成

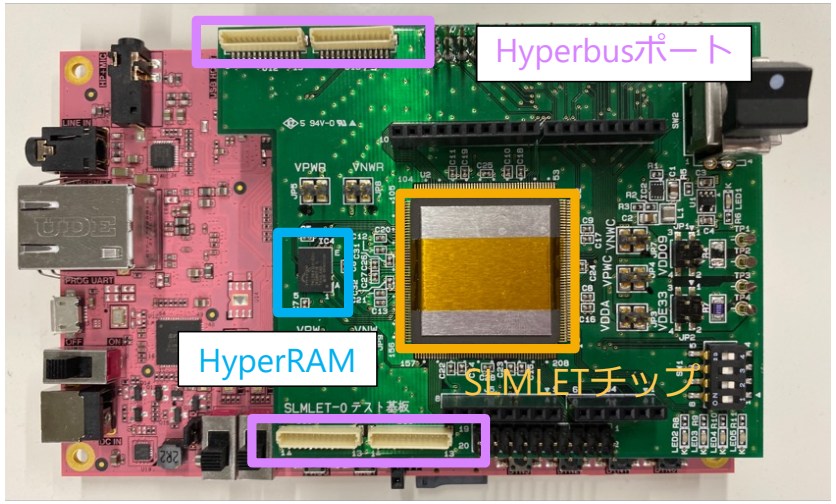



- リンカによって実行バイナリを生成
- カスタムなリンカスクリプト
  - 命令メモリとデータメモリの分離
  - メモリマップの記述
  - ライブラリのためのシンボル埋め込み

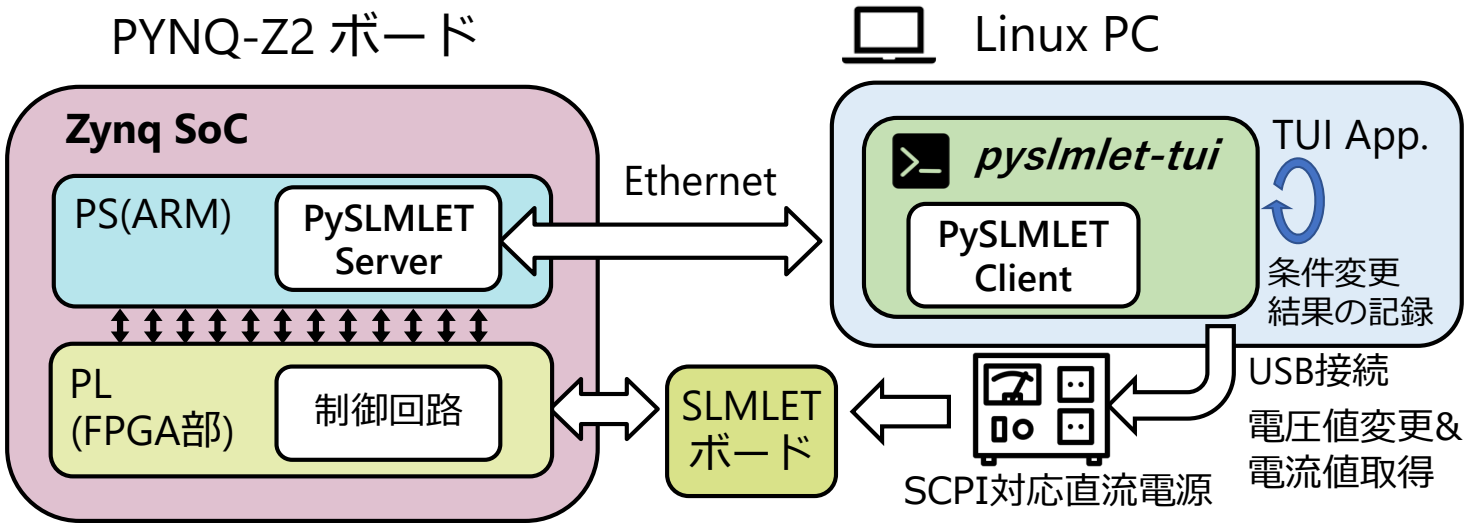
# プログラム例

```
1. コンフィギュレーションデータを共有メモリ領域に読み出し
 1 printf("load bitstream\n");
 2 unsigned int* bitstream = loadBitstream(0, USE_SRAM_BANK);
 3 if (!bitstream) {
 4     printf("failed to load bitstream\n");
 5     return 1;
 6 }
 7 2. コンフィギュレーションデータをSLMに書き込み (再構成)
 8 printf("start configuration\n");
 9 configurationSLMAsync(SLM_BLOCK0, bitstream);
10 // do something
11 synchronizeConfigurationSLM(SLM_BLOCK0);
12
13 printf("enable SLM\n");  3. リセット、開始信号の送信
14 resetSLM(USE_SLM_BLOCK);
15 startSLM(USE_SLM_BLOCK);
16                                  4.SLM  Memory mapped領域への書き込み
17 printf("write data to SLM\n");
18 for (i = 0; i < FPGA_REG_COUNT; i++) {
19     writeSLM(USE_SLM_BLOCK, (int*)(4 * i), &write_data[i]);
20 }
21
22 printf("read data from SLM:\n");      5.SLM  Memory mapped領域からの読み出し
23 for (i = 0; i < FPGA_REG_COUNT; i++) {
24     printf("FPGA Reg %d: %08X\n", i, readSLM(USE_SLM_BLOCK, (int*)(4 * i)));
25 }
```

# 評価環境: PySLMLET-TUI




**SLMLETボードとPYNQ-Z2**




- PySLMLET-TUI
  - 実行バイナリや電圧、周波数など条件を変更しテストを行うText-based UI
  - 簡単なシェルスクリプトで自動化測定を可能に
  - 先行研究でGUI版は実装済み [デモ動画]
- PySLMLET[3]
  - PYNQ FPGAはSLMLETのホストとして振る舞う回路が実装
  - Pythonのドライバでこれらを制御

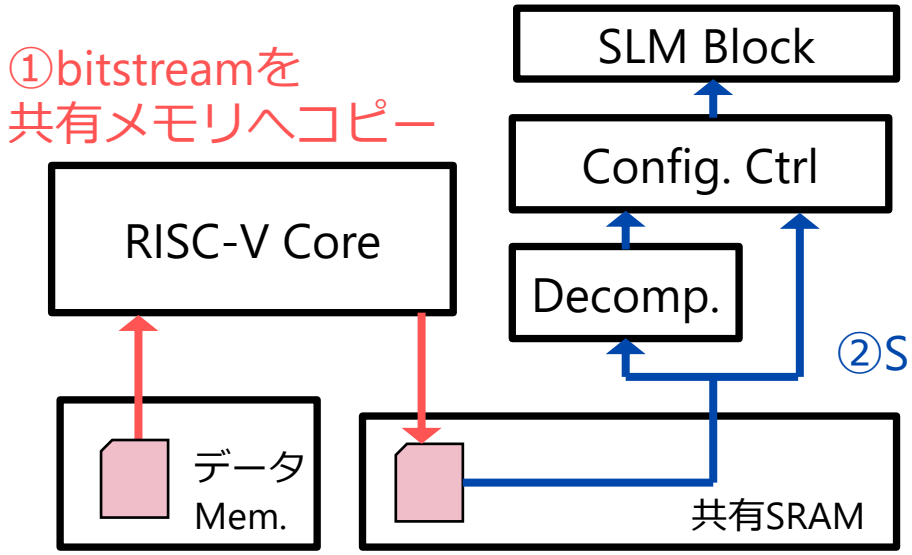
**評価環境の概要**

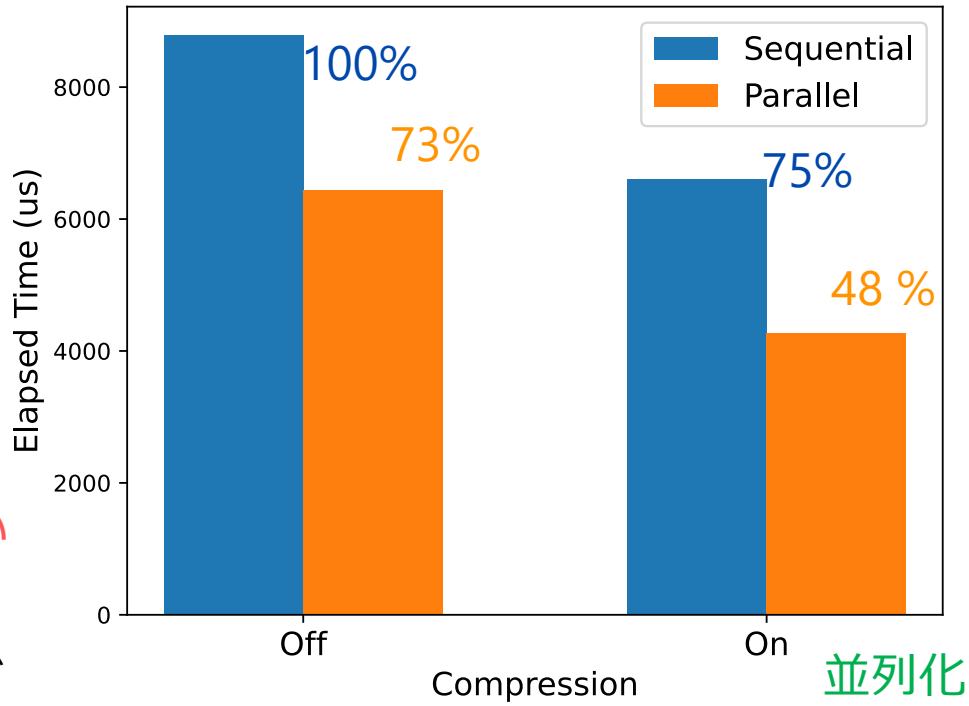
- pynqモジュールのimportに時間がかかりすぎる (20sec程度)
- サーバとクライアントに分離
- TUIはTCP/IPでサーバに接続
- TUI appの起動時間を短縮
- TUIを別マシンでも実行可能に

[3] 小島拓也, et al. "Jupyter Notebook を介した RISC-V SoC 向け実機テスト環境の構築." 研究報告組込みシステム (EMB) 2023.24 (2023): 1-7.

# 実験と評価

# 評価: 再構成にかかる時間



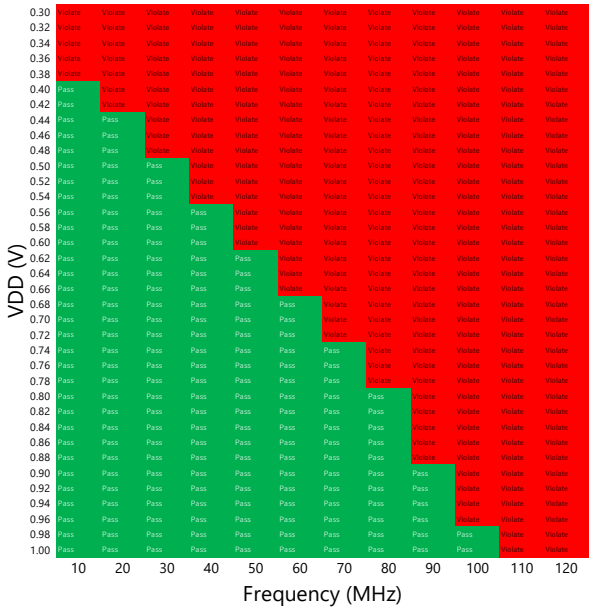




並列化&圧縮によりおよそ半分の時間に

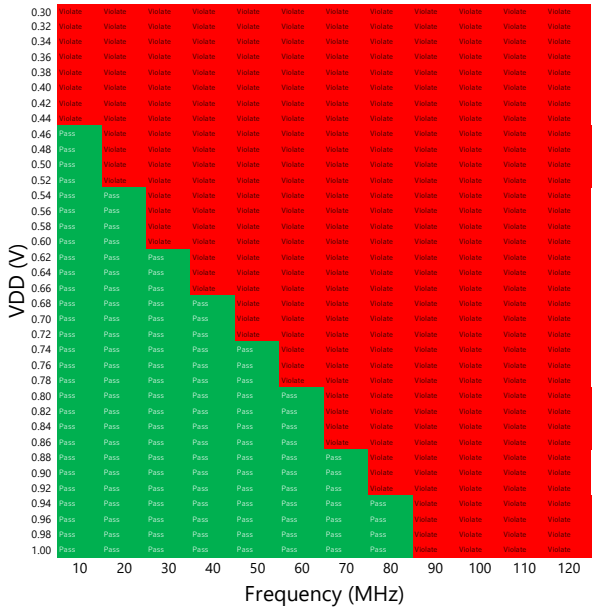
- 評価パターン
  - 二つのブロックを直列 or 並列に再構成
  - 再構成のライブラリ関数 ブロッキング or ノンブロッキング
  - TLCによる圧縮 or 非圧縮
- FPGA上の回路: RISC-Vから読み書き可能なメモリマップトなレジスタ 32bitx4本
  - SLM使用率: 29%, データ圧縮率49%
- 最新の類似研究[4]では6720 6-LUTの再構成に1msだがプロセッサは最大1.1GHz程度で動作
  → SLMLETの再構成時間は十分に高速といえる

[4] Chang, Ting-Jung, et al. "CIFER: A 12nm, 16mm 2, 22-Core SoC with a 1541 LUT6/mm 2 1.92 MOPS/LUT, Fully Synthesizable, CacheCoherent, Embedded FPGA." *2023 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 2023.

# 実機評価: SLMブロックの動作範囲

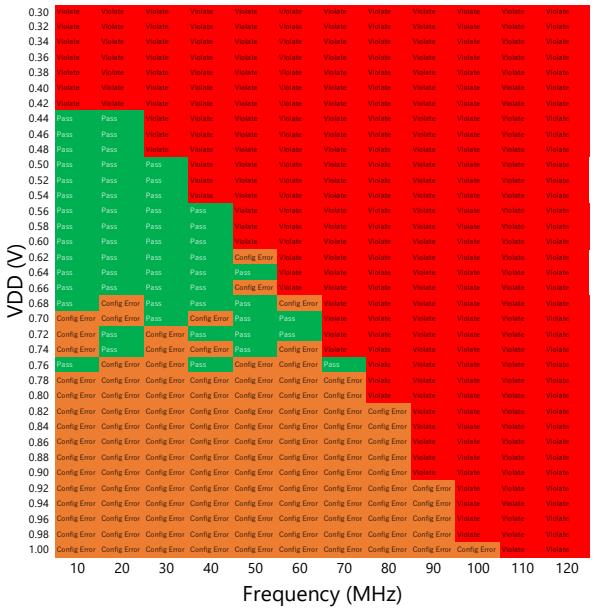



VPW=0 V ($f_{max}$ = 100MHz)

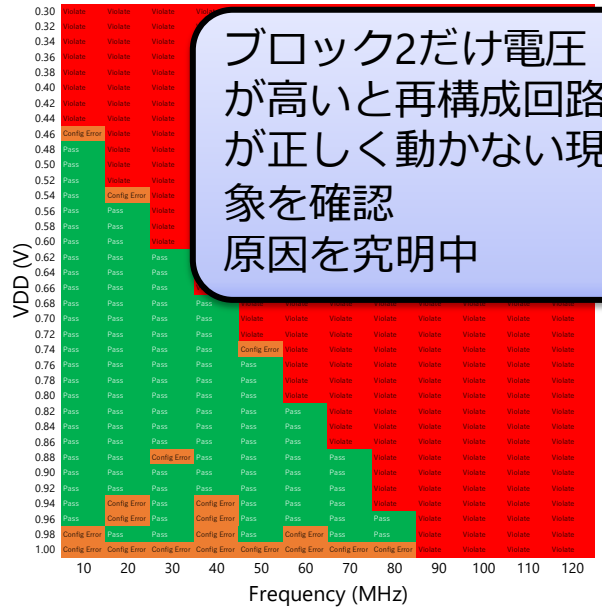



VPW=-0.4 V ($f_{max}$ = 80MHz)

ブロック1




VPW=0 V ($f_{max}$ = 100MHz)




VPW=-0.4 V ($f_{max}$ = 80MHz)

ブロック2

ブロック2だけ電圧が高いと再構成回路が正しく動かない現象を確認
原因を究明中

- 2つのブロックをそれぞれ正しく再構成および計算ができる電圧、周波数を測定
  - 先ほど同じメモリマップトレジスタの回路を使用 (最大SLM段数 6)
  - 電圧: 0.30-1.00 V (標準 0.90 V), 0.02 V刻み
  - 周波数: 10-120MHz, 10MHz刻み
  - ボディバイアス電圧: 0.0 V, -0.1 V, -0.2 V, -0.3 V, -0.4 V
- 動作特性は概ね同じ、想定よりも高い周波数でも動作した

失敗

失敗

再構成だけ失敗

# 他のシステムと比較

| | SLMLET (本研究) | ESP32 | Raspberry Pi Pico | Tang Nano 9K |
|---|---|---|---|---|
| 実装プロセス | USJC 55nm | TSMC 40nm | 40nm | TSMC 55nm |
| CPU | RV32I | Xtensa LX6 | ARM Cortex-M0+ | N/A* |
| 定格クロック | N/A | 240MHz | 133MHz | N/A |
| FPGA | 5-SLM 896 x2 | N/A | N/A | 4-LUT 8640 |
| 使用した開発環境 | SLMLET SDK | ESP IDF 5.0.2 | Pico SDK 1.5.1 | Gowin 1.9.8.11 IDE |
| GCC | 12.2.0 | 11.2.0 | 10.3.1 | N/A |
| 備考 | チップ電力を測定 | ボード電力を測定<br>通信モジュールはプログラムから明示的にOFF | ボード電力を測定<br>200MHzまでオーバークロックで動作を確認 | ボード電力を測定<br>FTDI D2XX Driver 1.4.27 |

*ボード上に別チップとしてBL702(RISC-V,144MHz)が搭載しているが基本はUART,JTAG用

- SLMLETはRISC-Vコアのみ&SLMブロックを併用のケースを実験
- ESP32, Raspberry Pi Picoはソフトウェア実装を使用
- Tang Nano 9K
  - 演算回路をRTLで実装,汎用PCとUART(FTDIのチップ)を介してデータ通信
  - UART IPを使用し、ボーレートは正しく動作した最大の921600に設定
  - UARTのインターフェースと演算回路を別のクロックドメインにし、境界(CDC)は4-phase handshake

# 比較実験: ベンチマーク

- ベンチマーク
  1. sram_memcpy (SLMLETのみ)
     - 共有SRAM上のデータ16KBをコピー
     - SLMブロックのSRAMインターフェースを使う回路
  2. CRC32
     - 1KBのバイナリデータからCRC32を計算(生成多項式0x04C11DB7)
     - ソフトウェア実装はMiBenchの実装を使用
  3. AES128
     - 128bit長の鍵を用いて1ブロックのAES暗号化を行う
     - ソフトウェア実装はMiBenchの実装を使用
- いずれもO3でコンパイル

# 結果: 処理時間の比較

- 最大動作周波数におけるレイテンシを計測

時間: us, 周波数: MHz

| | sram_memcpy | | CRC32 | | AES128 | |
|---|---|---|---|---|---|---|
| | 時間 | 周波数 | 時間 | 周波数 | 時間 | 周波数 |
| SLMLET (SW) | **95.60** (**28,679**) | 300 | 47.05 (14,117) | 300 | **6.217** (1,865) | 300 |
| SLMLET | 329.9 (32,994) | 100 | **30.73** (**2,151**) | 70 | 34.86 (**976**) | 28 |
| ESP32 | n/a | n/a | 77.58 | 240 | 7.814 | 240 |
| Raspberry Pi Pico | n/a | n/a | 66.59 | 200 | 14.00 | 200 |
| Tang Nano 9K | n/a | n/a | 885.7 | 50 | 1333 | 34 |

()書きはサイクル数

SLMブロックはSRAMインターフェースが16bitなので
ハードウェア化してもソフトウェア実装を上回ることができない

# 結果: 処理時間の比較

- 最大動作周波数におけるレイテンシを計測

時間: us, 周波数: MHz

| | sram_memcpy | | CRC32 | | AES128 | |
|---|---|---|---|---|---|---|
| | 時間 | 周波数 | 時間 | 周波数 | 時間 | 周波数 |
| SLMLET (SW) | **95.60** (**28,679**) | 300 | 47.05 (14,117) | 300 | **6.217** (1,865) | 300 |
| SLMLET | 329.9 (32,994) | 100 | **30.73** (**2,151**) | 70 | 34.86 (**976**) | 28 |
| ESP32 | n/a | n/a | 77.58 | 240 | 7.814 | 240 |
| Raspberry Pi Pico | n/a | n/a | 66.59 | 200 | 14.00 | 200 |
| Tang Nano 9K | n/a | n/a | 885.7 | 50 | 1333 | 34 |

()書きはサイクル数

1ワードのCRC32を計算を1サイクルで行うハードウェアを実装、サイクル数を大きく削減
周波数はソフトウェアより低いがそれでも時間を短縮
Tang Nano 9KはUART通信が支配的

# 結果: 処理時間の比較

- 最大動作周波数におけるレイテンシを計測

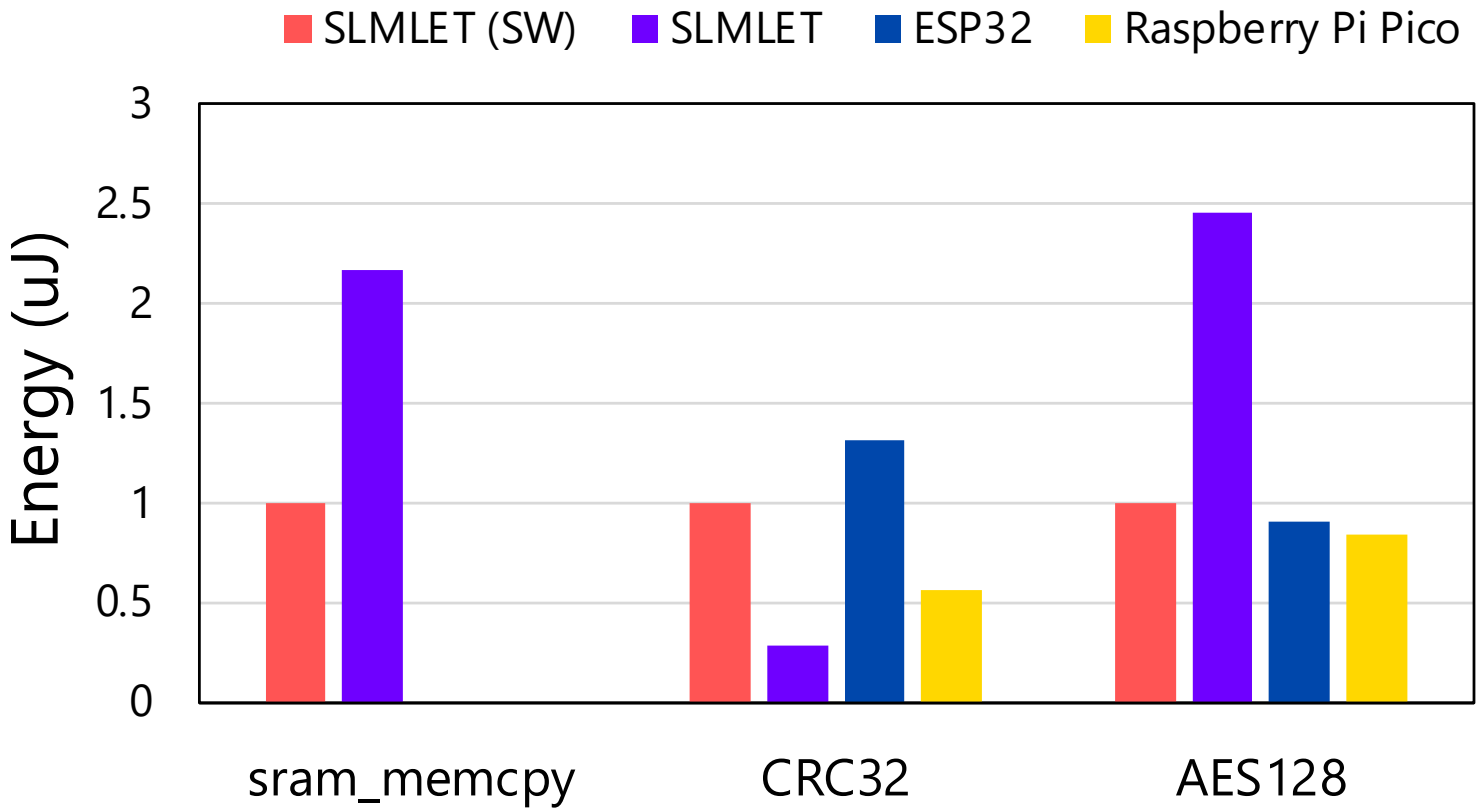
時間: us, 周波数: MHz

| | sram_memcpy | | CRC32 | | AES128 | |
|---|---|---|---|---|---|---|
| | 時間 | 周波数 | 時間 | 周波数 | 時間 | 周波数 |
| SLMLET (SW) | **95.60** (**28,679**) | 300 | 47.05 (14,117) | 300 | **6.217** (1,865) | 300 |
| SLMLET | 329.9 (32,994) | 100 | **30.73** (**2,151**) | 70 | 34.86 (**976**) | 28 |
| ESP32 | n/a | n/a | 77.58 | 240 | 7.814 | 240 |
| Raspberry Pi Pico | n/a | n/a | 66.59 | 200 | 14.00 | 200 |
| Tang Nano 9K | n/a | n/a | 885.7 | 50 | 1333 | 34 |

()書きはサイクル数

SLMブロックを2つ使用し、専用回路を実装
サイクル数は減少したが、二つのブロックにまたがる
クリティカルパスによって動作周波数が低い
結果的にはソフトウェア実装の方が早い

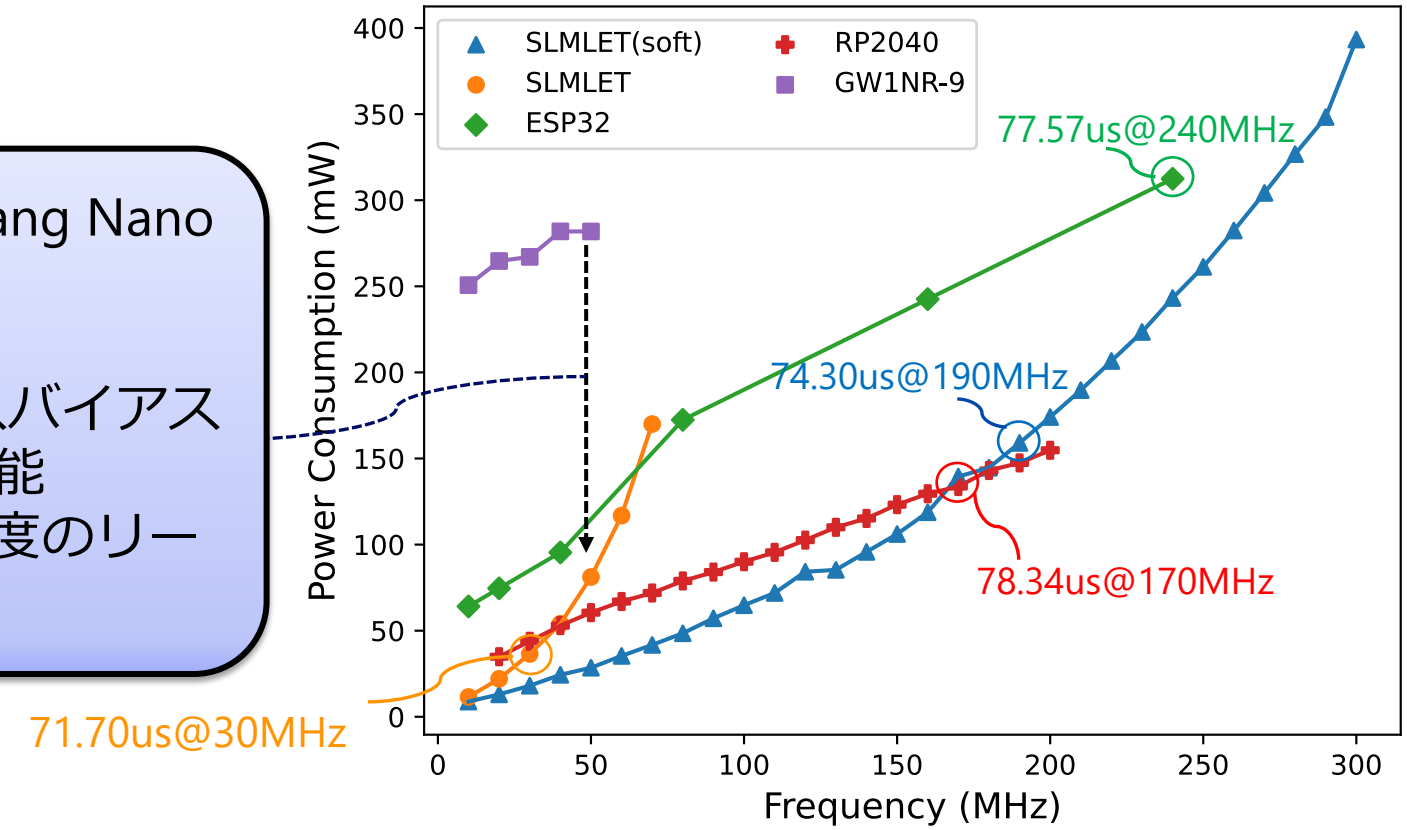
# 消費エネルギーの比較 (最高性能時)




- SLMLET (SW実装)のケースで正規化
- CRC32ではSLMブロックを使うことで以下のエネルギー削減を達成
  - SLMLET (SW)に対し 71%, ESP32に対し 78%, Raspberry Pi Picoに対し 49%

# 消費電力の比較 (CRC32)

FPGA上の演算回路はTang Nano 9Kは50MHz
SLMブロックは70MHz
→ 50MHzではリバースバイアスによるリーク削減が可能
VPW=-0.3 Vで**5mW**程度のリーク電力



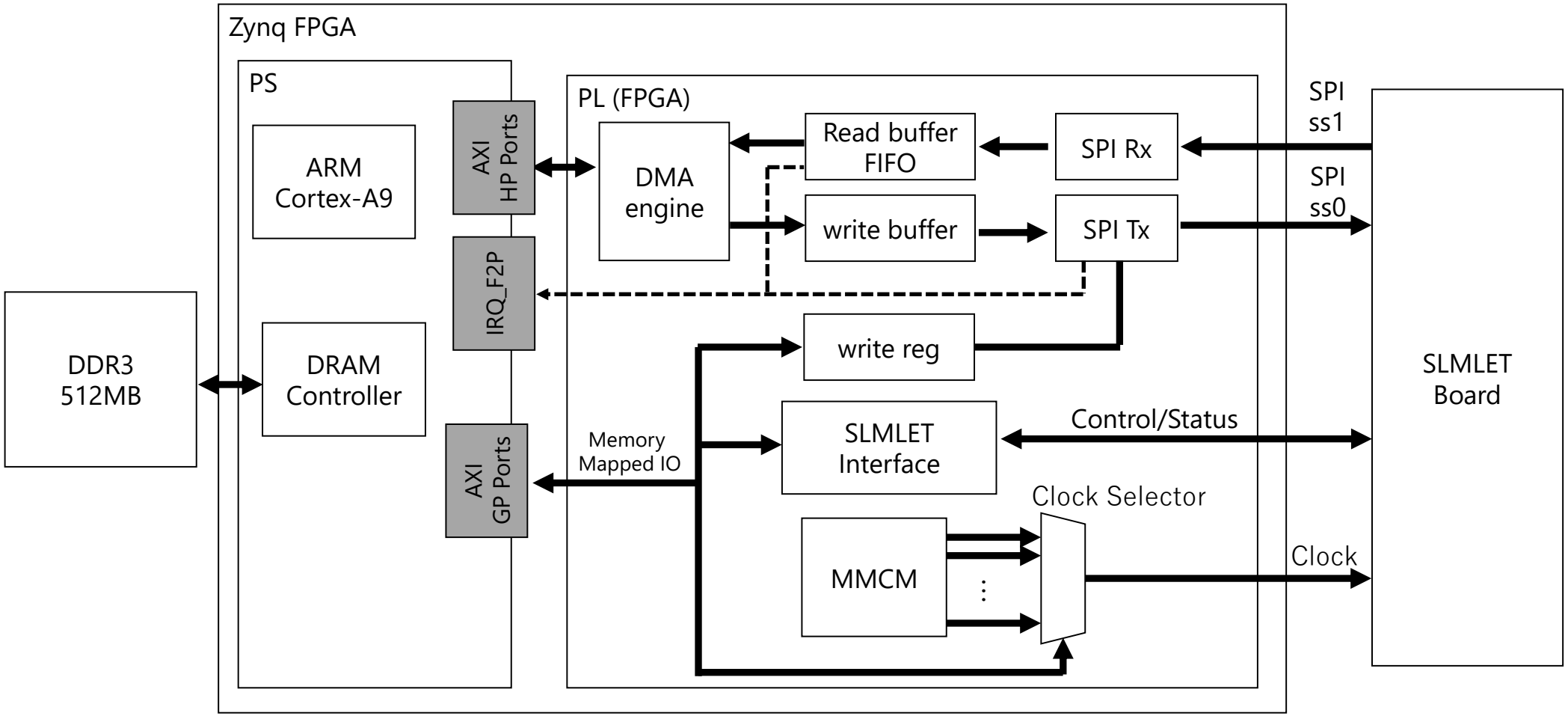

- 70us程度の処理時間となる動作点で比較した時の電力削減
  - SLMLET (SW)に対し 77%, ESP32に対し 88%, Raspberry Pi Picoに対し 72%

# まとめ

- SLMLETはエッジ向けのRISC-VコアとSLM再構成ロジックによるFPGAを搭載したSoC
- 本研究ではSLMLET向けのSW/HW協調開発フローおよびライブラリを実装
- 実機による実験により
  - 再構成処理の並列化、ビットストリームの圧縮を行うことで先行研究と比べても十分に高速な回路切り替えが可能
  - 単純な回路であれば100MHz程度の動作が可能
  - ブロック2の再構成コントローラ部で未解決の不具合が発見
- エッジ向けの他のデバイスと比較
  - CRC32のようなハードウェア化の恩恵が大きいベンチマークではレイテンシの短縮とエネルギーの削減を確認
- 今後
  - Hyperbus経由のDRAMを用いた再構成処理やアプリケーション開発
  - より具体的なユースケースを用いた評価

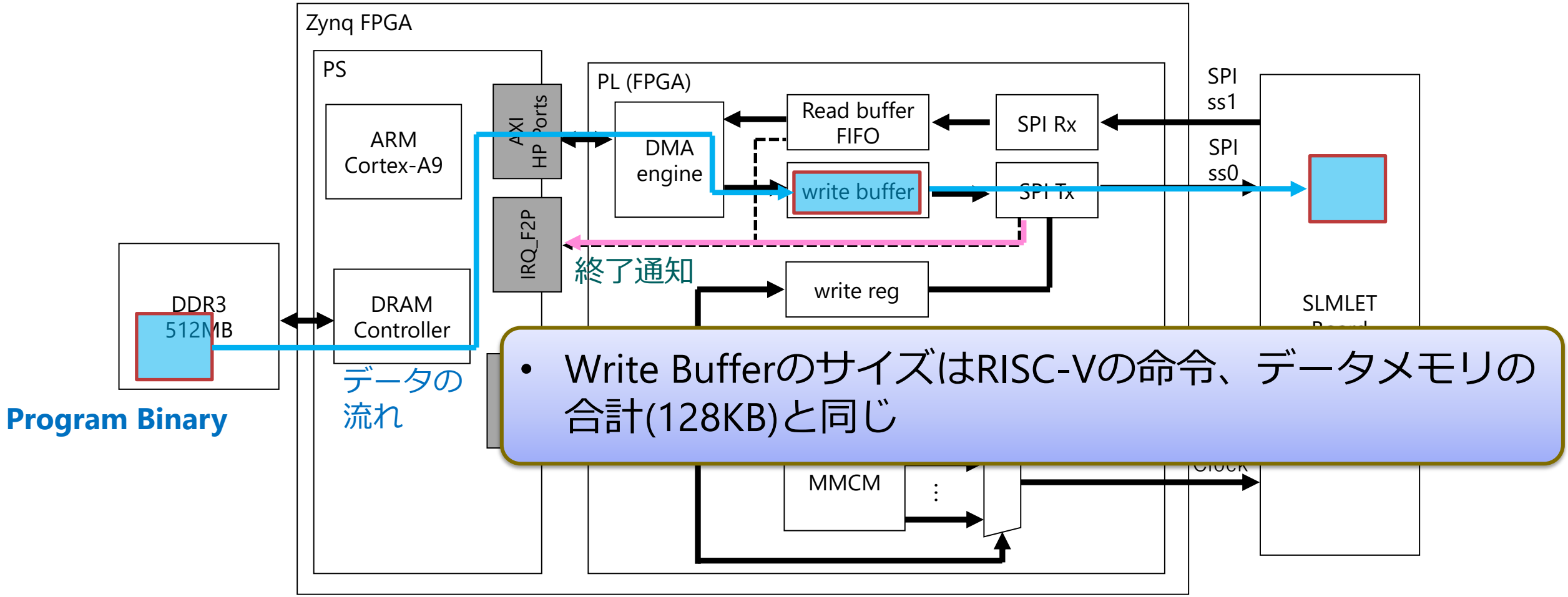
# 予備スライド

# 制御用FPGA回路設計




**Zynq FPGA上に設計したシステムの構成**

# データ転送 (ブートを例に)

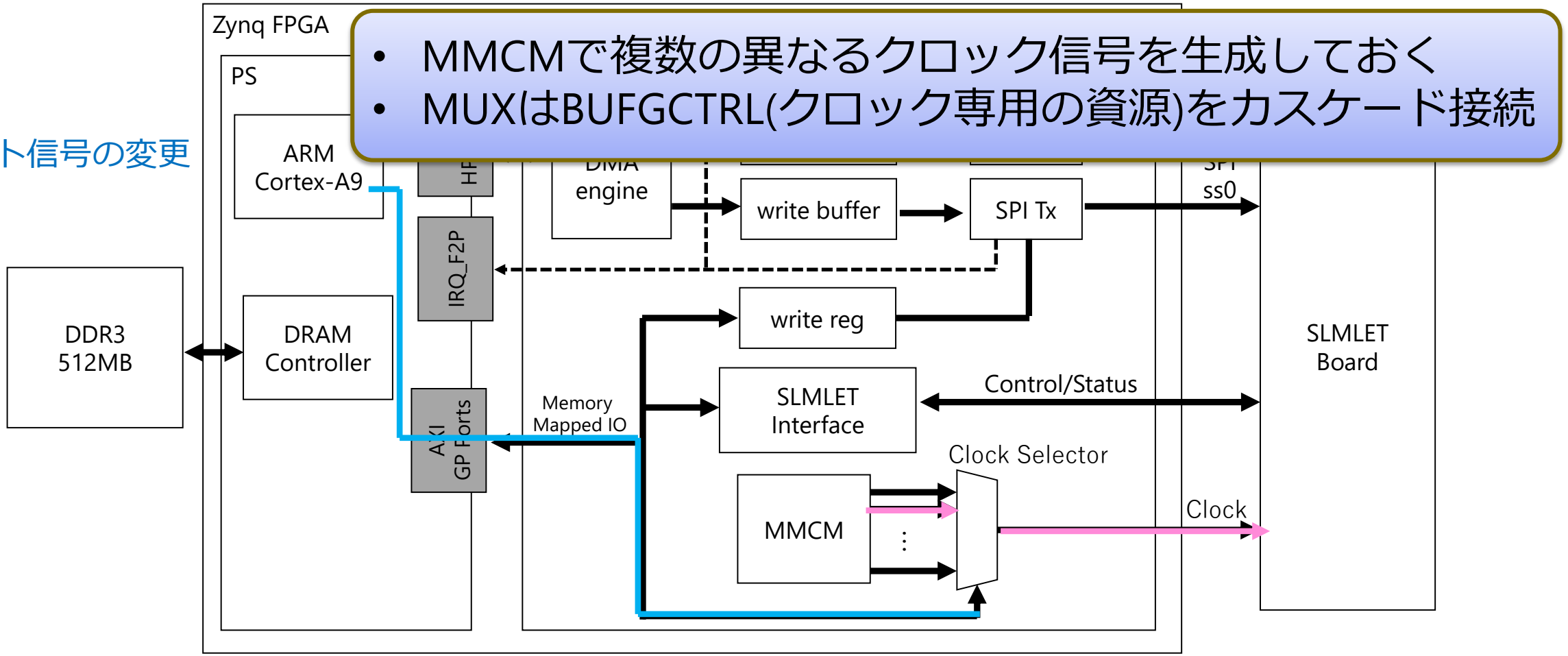



**Zynq FPGA上に設計したシステムの構成**

# クロック信号の切り替え

- MMCMで複数の異なるクロック信号を生成しておく
- MUXはBUFGCTRL(クロック専用の資源)をカスケード接続

セレクト信号の変更

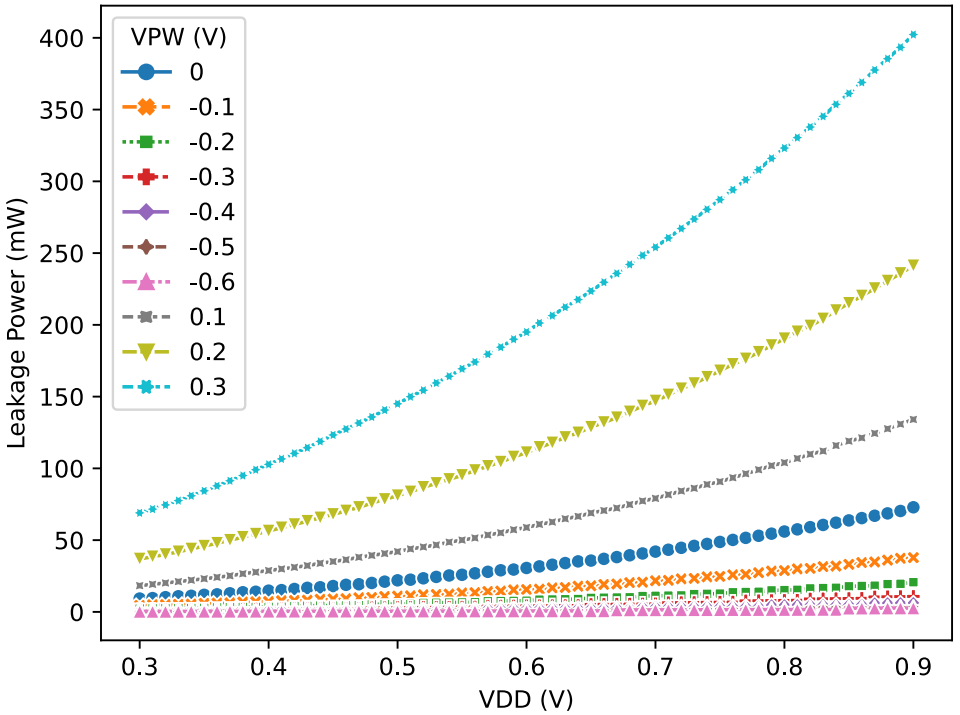



**Zynq FPGA上に設計したシステムの構成**

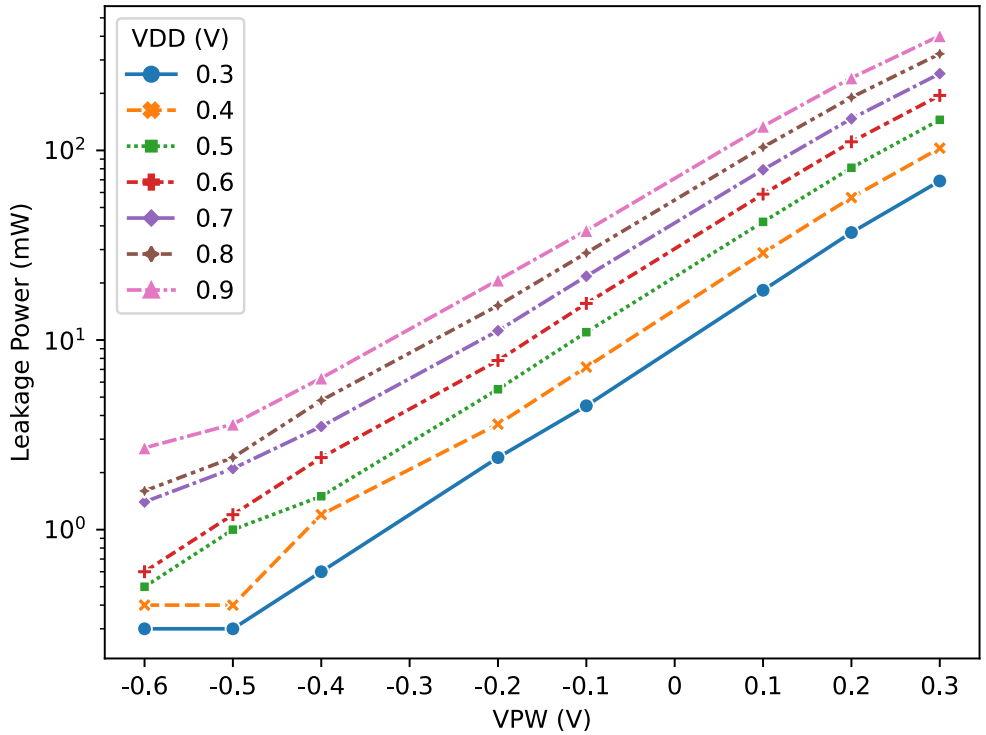
# ボディバイアス制御によるリーク電力の増減

- 電源電圧VDDとボディバイアス電圧 VPW (P-well側), VNW (N-well側)を変えて測定
  - VDD = VPW + VNWとなるように設定
  - リバースバイアス (VPW < 0 V)時はリークが削減されるが遅延が増加 (後述)

【測定条件】
- VDD: 0.30-0.90 V
- VPW: -0.6-+0.3 V

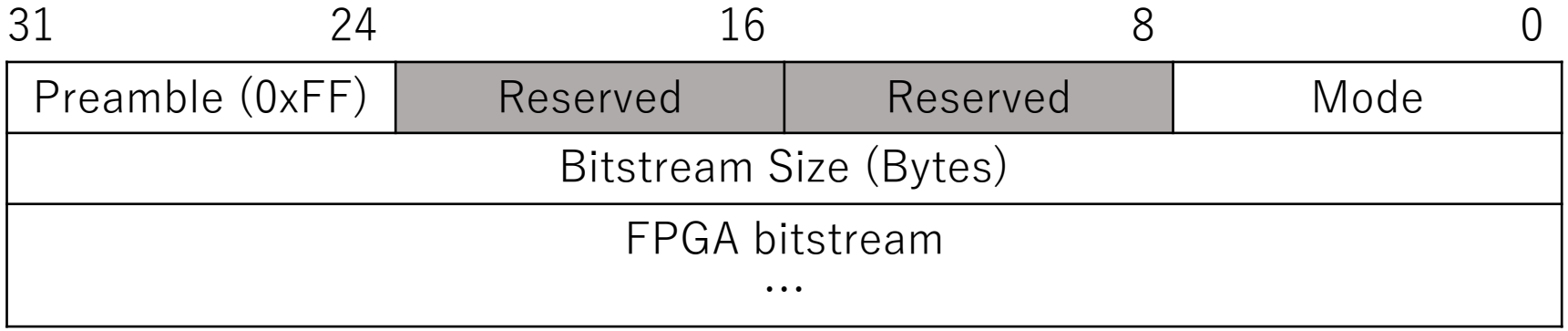



**VDD-リーク電力の関係**




**VPW-リーク電力の関係**

# 可変長ビットストリームへの対応




- 各ビットストリームに挿入されるヘッダ
  - Mode: 圧縮 or 非圧縮
  - Bitstream Size: ビットストリームのサイズ
- ライブラリがヘッダを見て判断