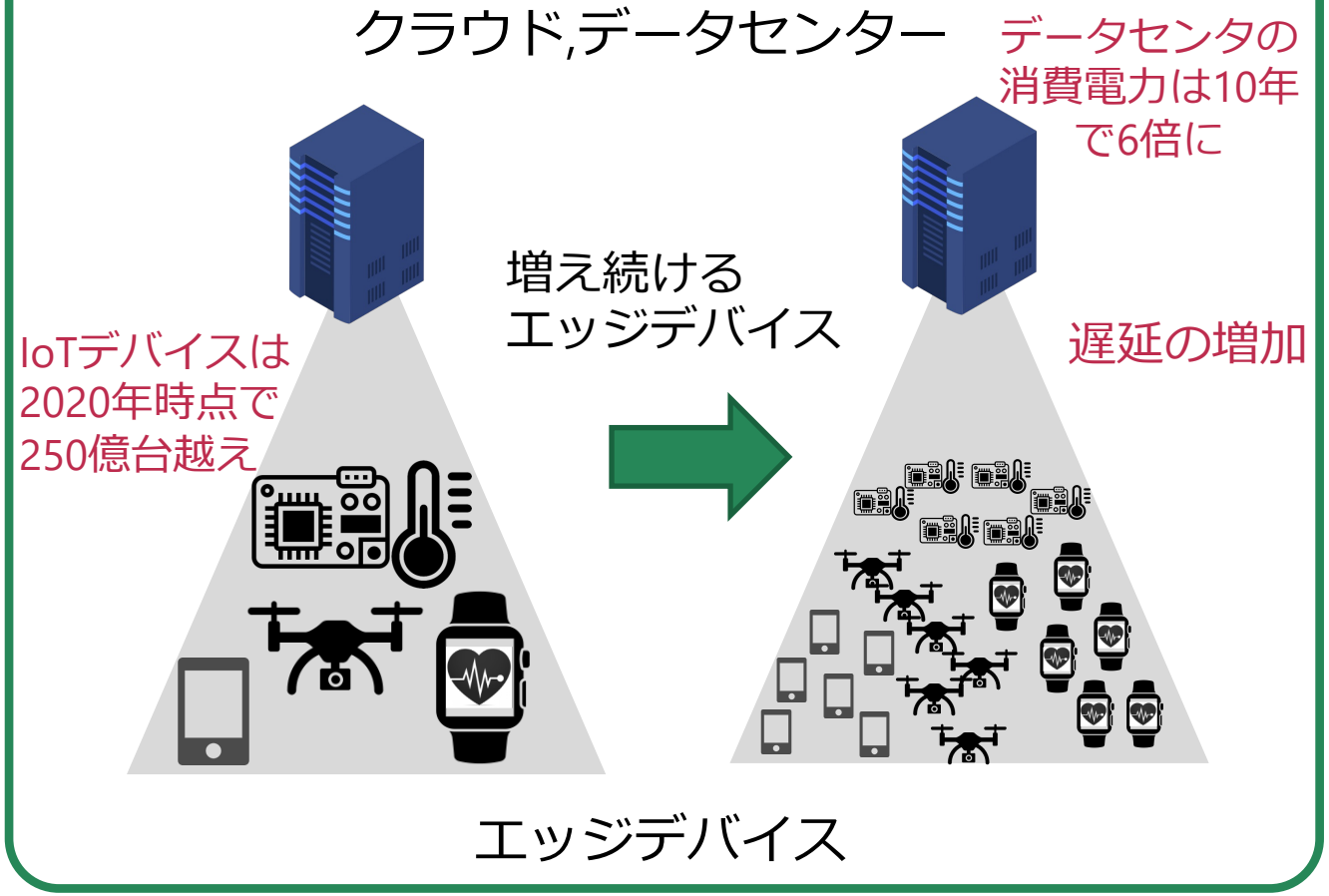
Jupyter Notebookを介したRISC-V SoC向け実機テスト環境の構築

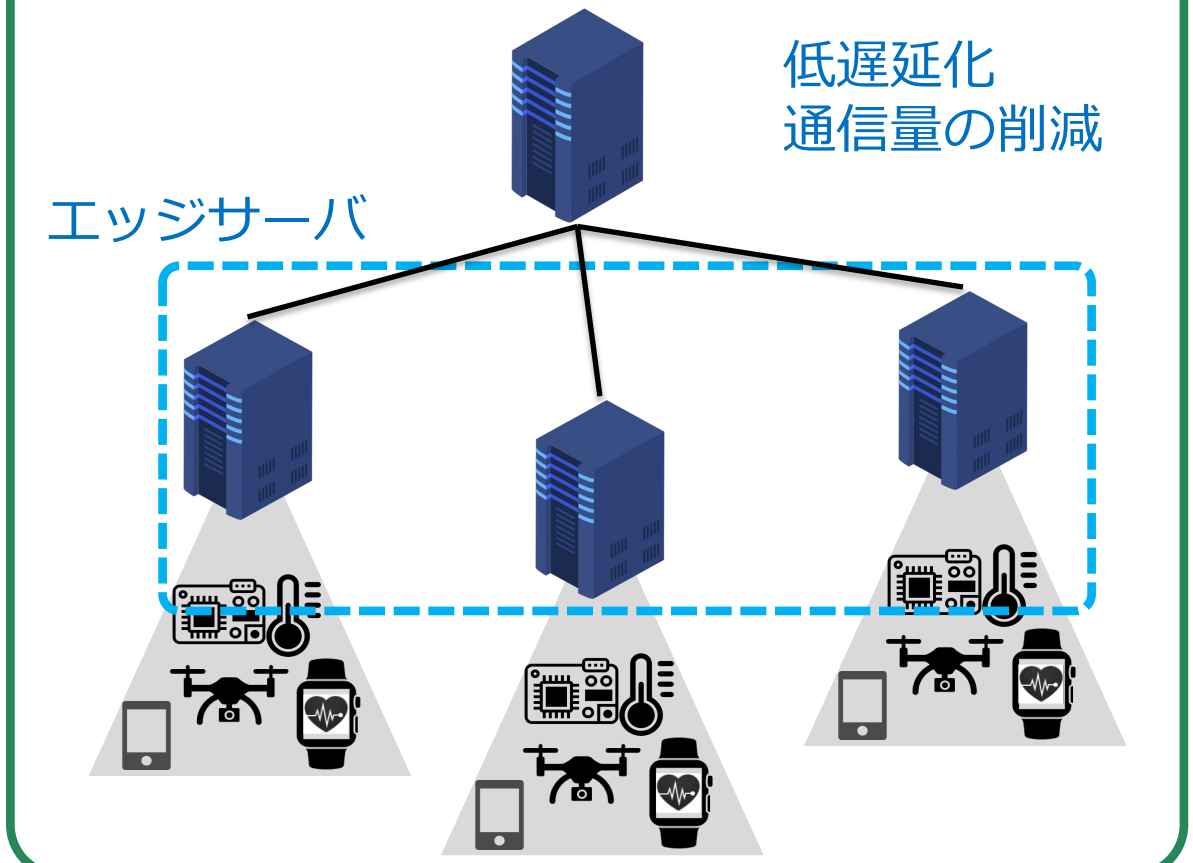
小島　拓也†‡, 亀井 愛佳§, 矢内 洋祐§, 天野 英晴§,
久我 守弘¶, 飯田 全広¶
†東京大学
‡JSTさきがけ
§慶應義塾大学
¶熊本大学

Society 5.0 時代における計算機システム

従来のクラウド集約型

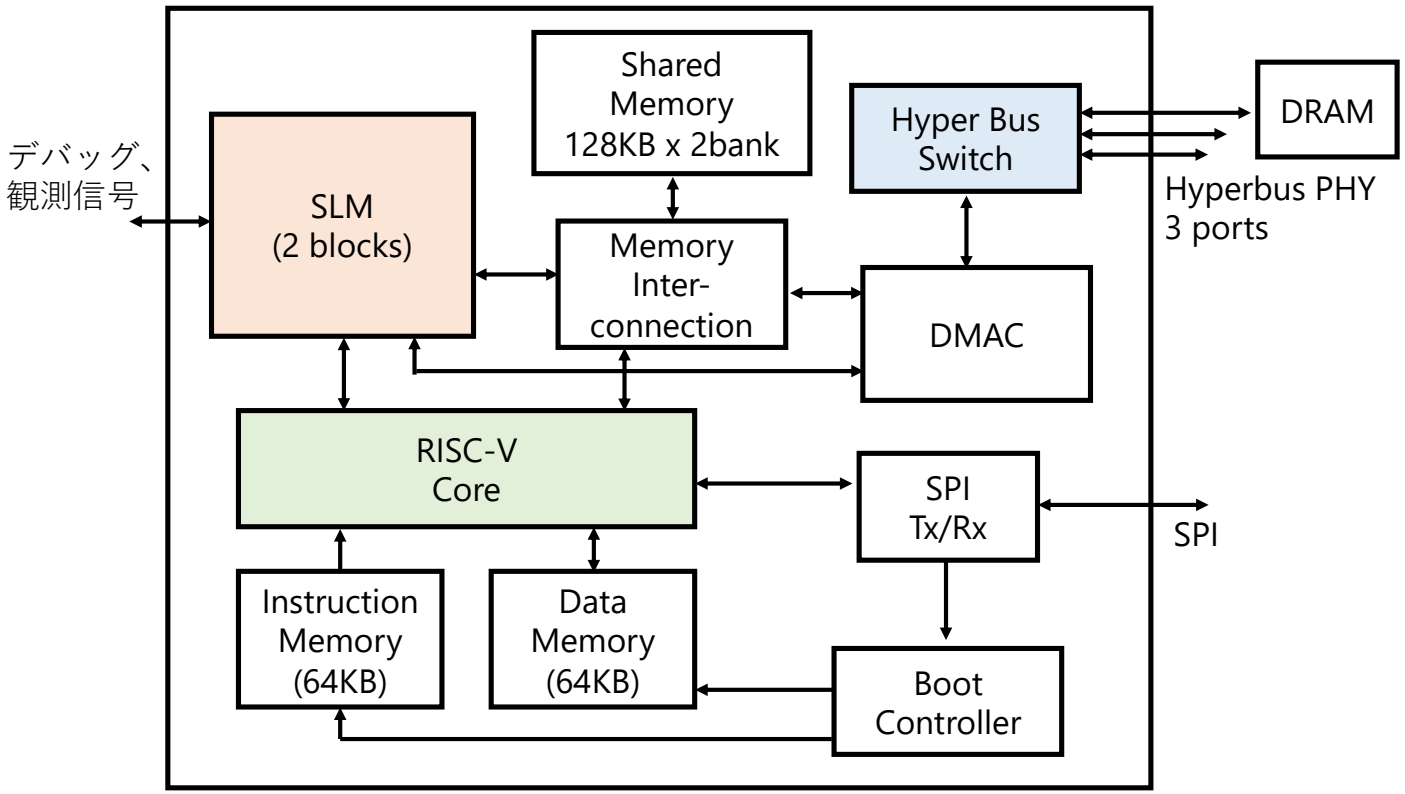


MEC (Multi-access Edge Computing)型



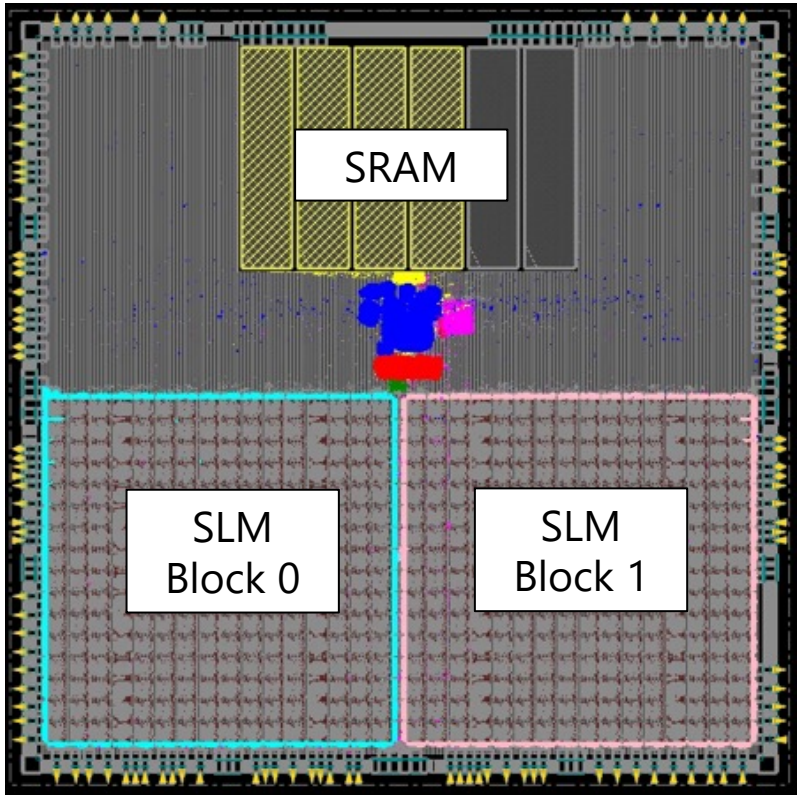
SLMLET: エッジ,MEC向けeFPGA・CPU混載SoC

- 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
- 特徴
  - 低コスト、スケーラブル
  - ハードIPが不要



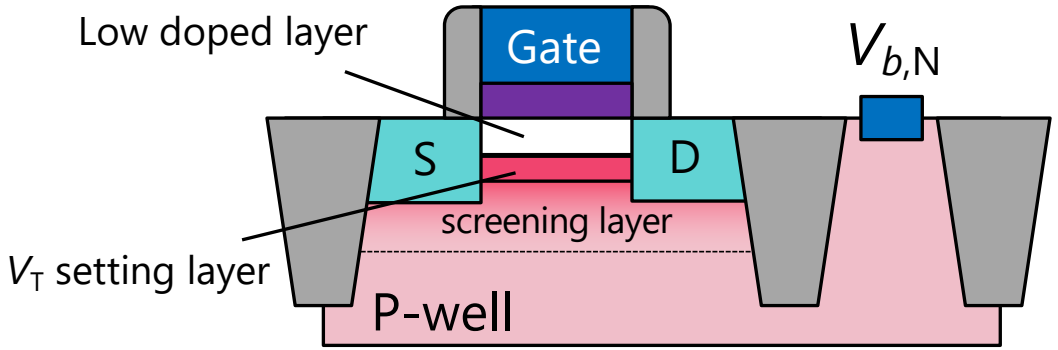
SLMLET SoCの構成

SLMLETのプロトタイプ



チップレイアウト 4.2mm角

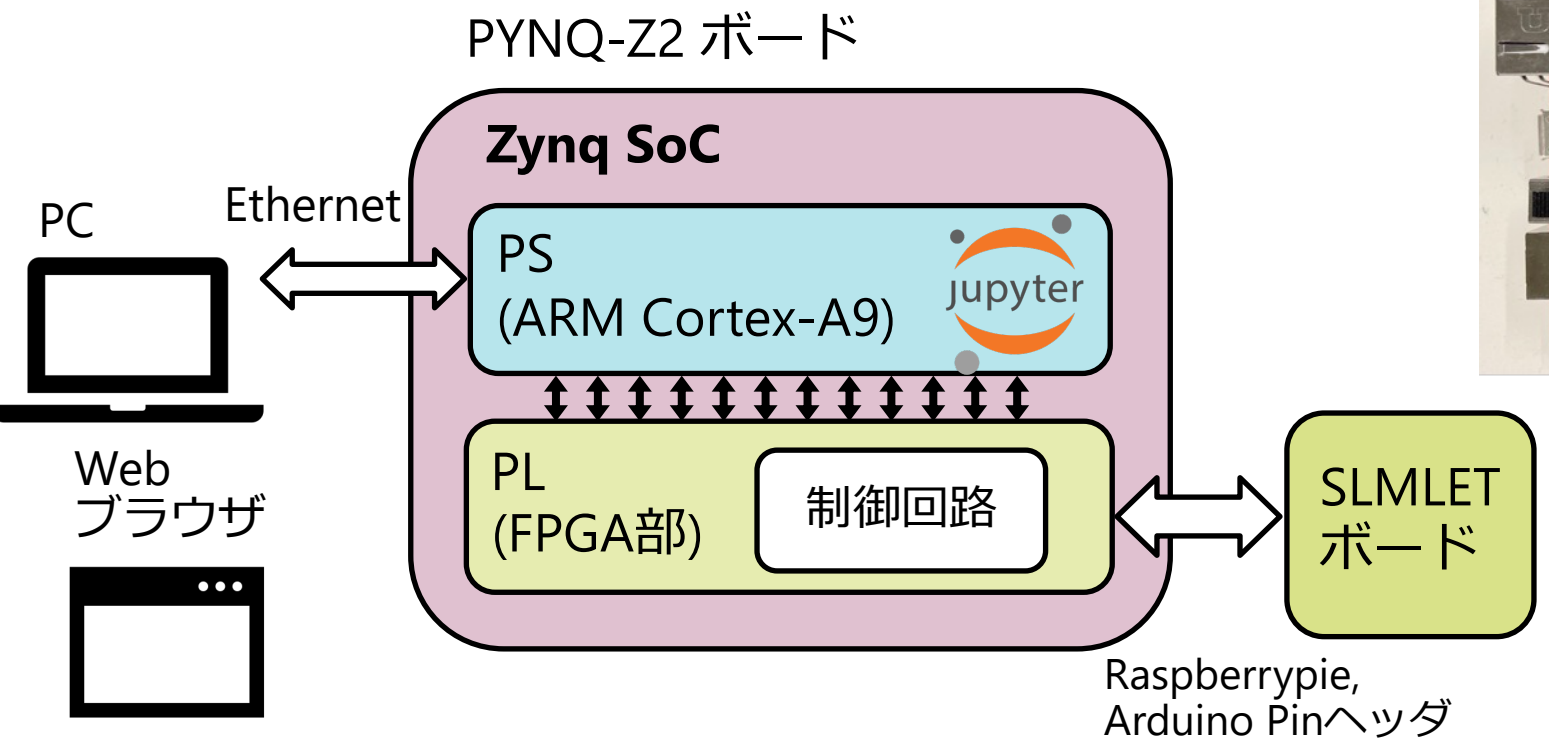
- USCJ 55nmを採用
  - 閾値電圧のばらつきが小さい
  - 240mV/V の高い基板効果係数
    → ボディバイアス制御によるリーク削減の期待
- スタンダードセル: C55DDCT07L60LVT

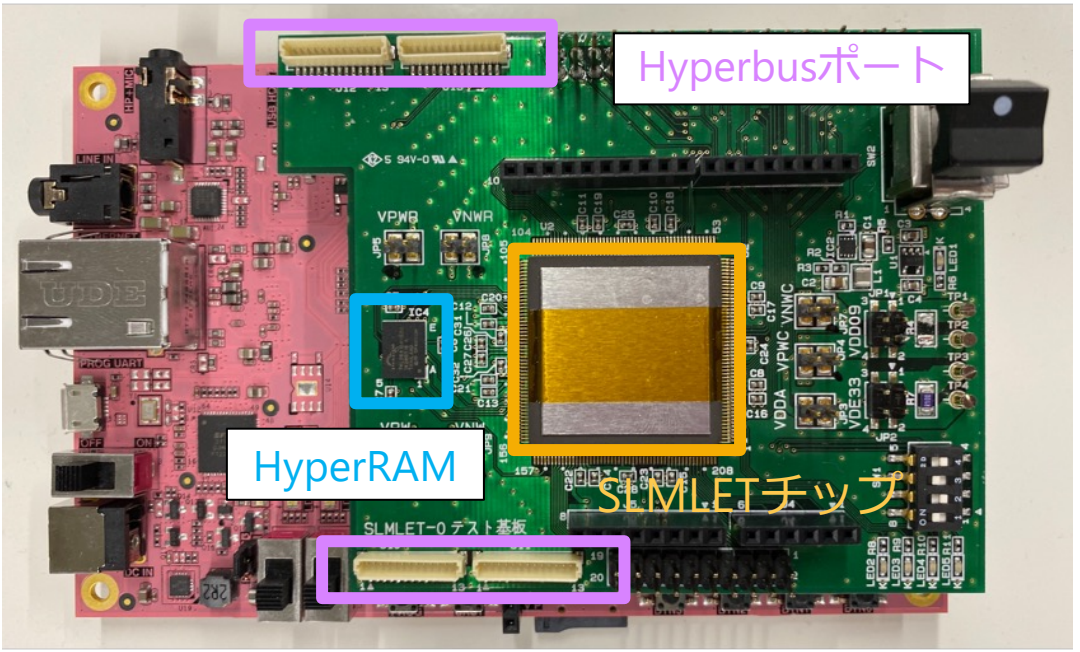


USCJプロセスのトランジスタ構造

テスト・評価システム

- PYNQ-Z2ボードに刺さるドーターボードを開発
  - PythonからPL部の回路を扱うことが可能
- PC上のWebブラウザを介してチップのテストおよび評価が可能





実際のボード写真

本研究の目的

- 効率的なテストおよび測定に向けた環境整備が急務
- 要件
  - PYNQ-Z2上のFPGA設計: 常に再利用可能な設計 (1度実装すれば良い)
  - ソフトウェア開発: ハードウェア実装を抽象化したAPI,ライブラリの整備
- 本研究による実装
  - SLMLET制御用FPGA回路設計
  - SLMLETソフトウェア開発キット (SDK)
  - Pythonライブラリ PySLMLET

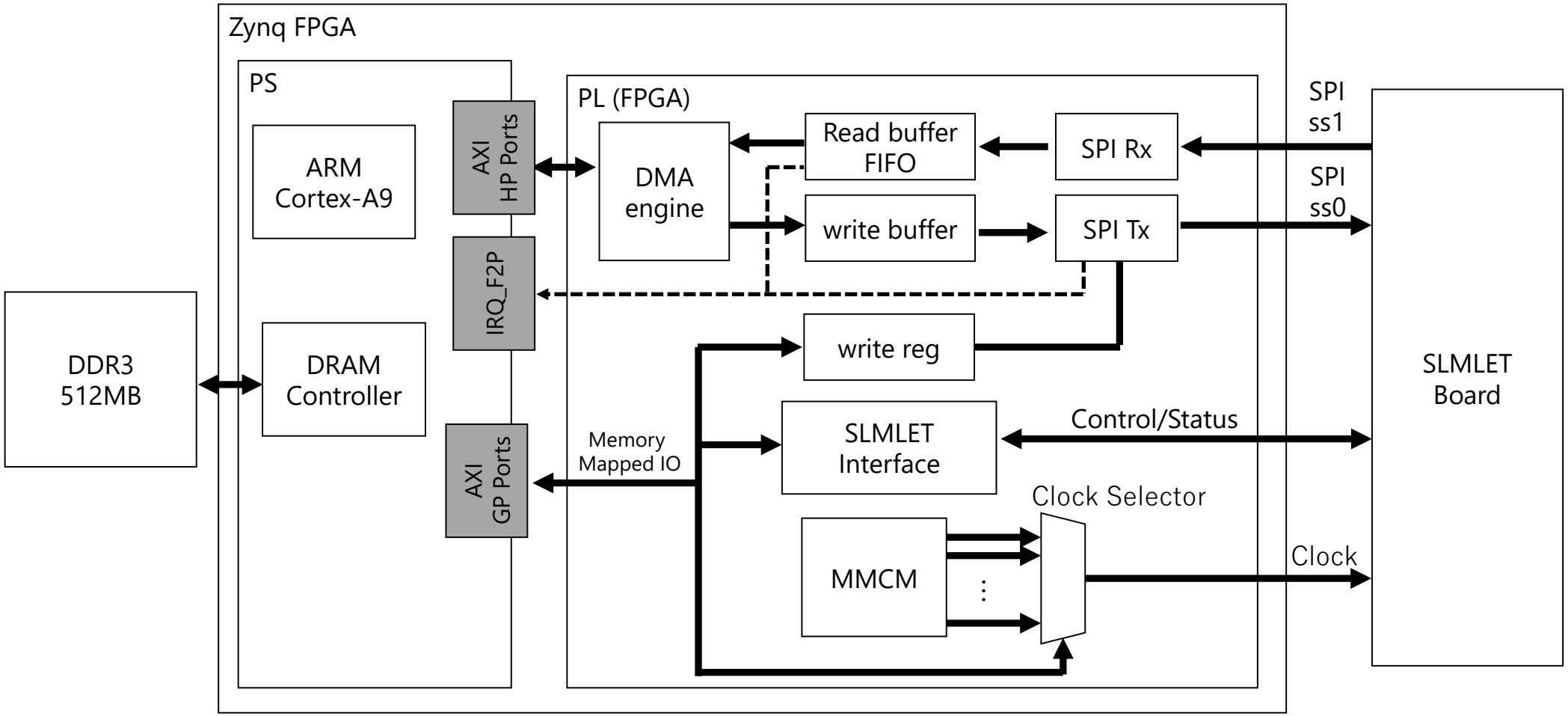
SLMLET制御用FPGA回路設計

制御用FPGA回路設計の設計方針

- 求められる機能

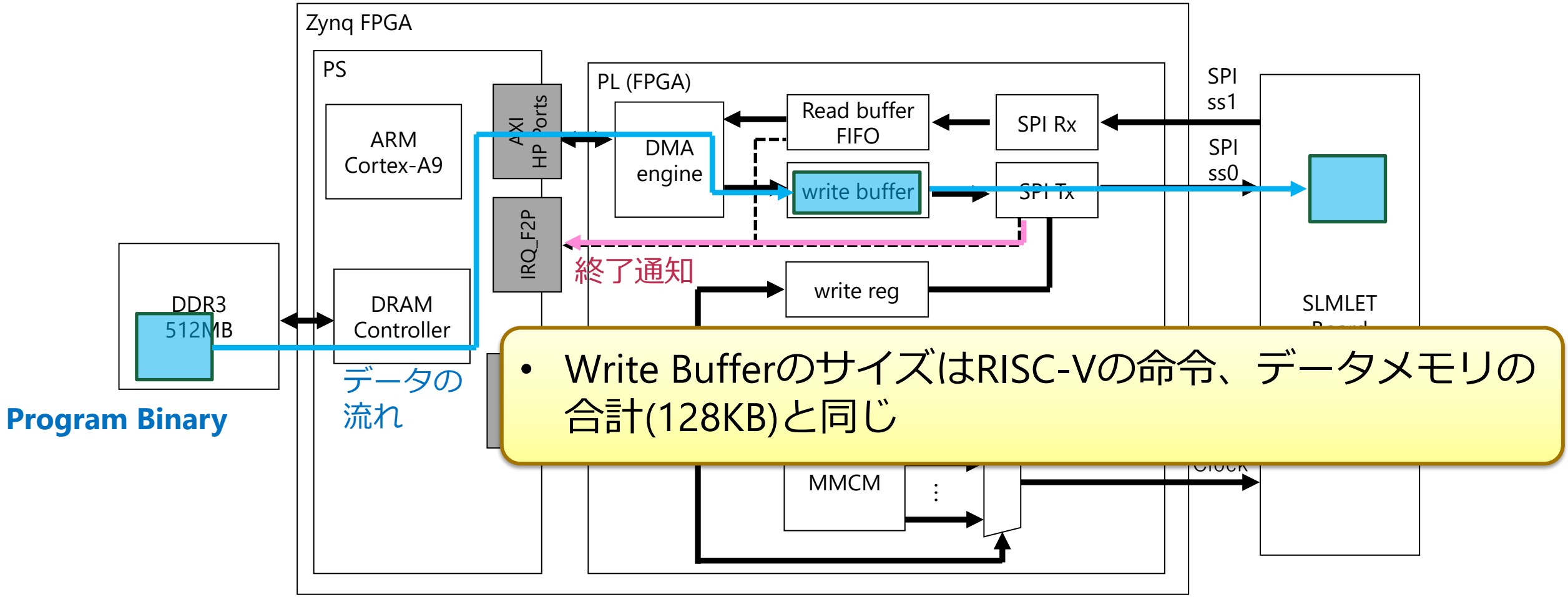
1. SPIを用いたデータの入出力
   - SLMLETボード側がマスタ、FPGA側がスレーブ
   - ブートを含む任意のデータ転送
   - プログラムデータはPS側のDRAMから送る
2. クロック信号の入力
   - FPGA上のMMCM(Mixed-Mode Clock Manager)で生成
   - PS側のソフトウェア制御でクロック周波数の切り替えが可能
3. 各種制御信号の入力、状態の観測
   - リセット信号、ブート開始信号, etc

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)

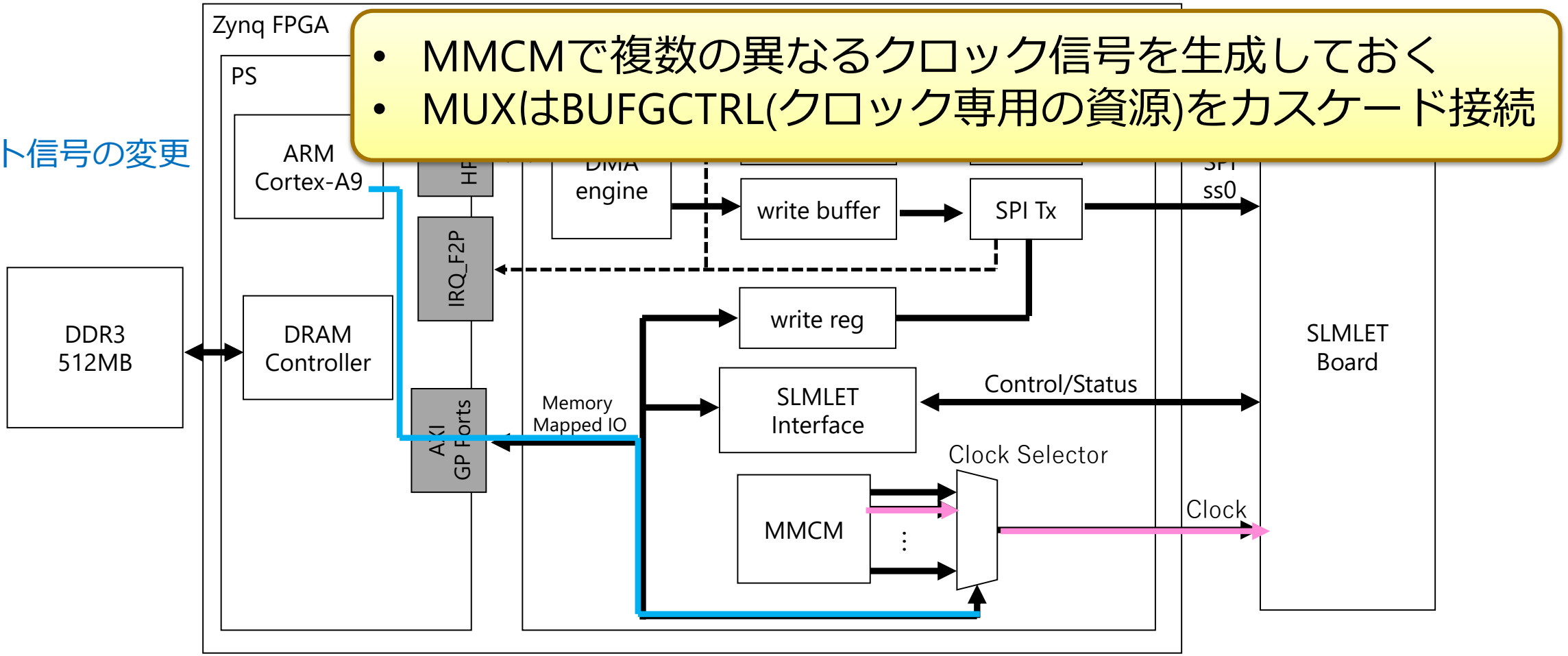


Zynq FPGA上に設計したシステムの構成

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

動作周波数と回路規模

- 130MHzまではタイミング違反なしで実装可能

資源	使用量	利用可能量	使用率 (%)
LUT	6128	53200	11.5
LUTRAM	1227	17400	7.05
FF	7925	106400	7.45
BRAM	68	140	**48.6**
IO	25	125	20
BUFG	27	32	**84.4**
MMCM	2	4	50

- BRAMはRead buffer(4KB), Write buffer(128KB)で主に消費
- BUFGの上限によりSLMLET入力用クロック信号は12種が最大

SLMLET SDK

SLMLET SDKの構成要素

- コンパイル環境
  - RISC-V GNU toolchain
  - SLMLET用のリンカスクリプト
  - SLM部のコンフィギュレーションデータの自動取り込み
- ライブラリ、API
  - SPIデータ入出力
  - printf, scanf (SPI経由)
  - HyperBusデータ転送
  - SLM制御

ライブラリを用いたHyperRAMの使用例

1. 共有メモリ領域を確保

```
1   int* src_mem = (int*)sharedMalloc(MEM_BANK0, N * sizeof(int)); // bank 0
2   int* dst_mem = (int*)sharedMalloc(MEM_BANK1, N * sizeof(int));; // bank 1
3
4   HBResult result;
5
6   // Core -> HyperRAM
7   printf("copy data from core to HyperRAM\n");
8   result = slmletMemcpy(ram_adder, src_mem, sizeof(int) * N, slmletMemcpySharedmemToHyperram);
9   if (result != HB_SUCCESS) {
10      printf("Fail in memcpy Core -> HyperRAM\n");
11      return -1;
12  }
13
14  //HyperRAM -> Core
15  printf("copy data from HyperRAM to core\n");
16  result = slmletMemcpy(dst_mem, ram_adder, sizeof(int) * N, slmletMemcpyHyperramToSharedmem);
17  if (result != HB_SUCCESS) {
18      printf("Fail in HyperRAM -> Core\n");
19      return -1;
20  }
```

2. Hyperbus I/Fを通じて共有メモリからHyperRAMへデータ転送

3. Hyperbus I/Fを通じてHyperRAMから共有メモリへデータ転送

ライブラリを用いたSLMブロックの使用例

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(MEM_BANK0);
3   if (!bitstream) {
4       printf("failed to load bitstream\n");
5       return 1;
6   }
```

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)

```
7   printf("start configuration\n");
8   configurationSLM(SLM_BLOCK0, bitstream);
9
```

3. リセット、開始信号の送信

```
10  printf("enable SLM\n");
11  resetSLM(SLM_BLOCK0);
12  startSLM(SLM_BLOCK0);
13
```

4.SLM Memory mapped領域への書き込み

```
14  printf("write data to SLM\n");
15  for (i = 0; i < FPGA_REG_COUNT; i++) {
16      writeSLM(SLM_BLOCK0, (int*)(4 * i), &write_data[i]);
17  }
18
```

5.SLM Memory mapped領域からの読み出し

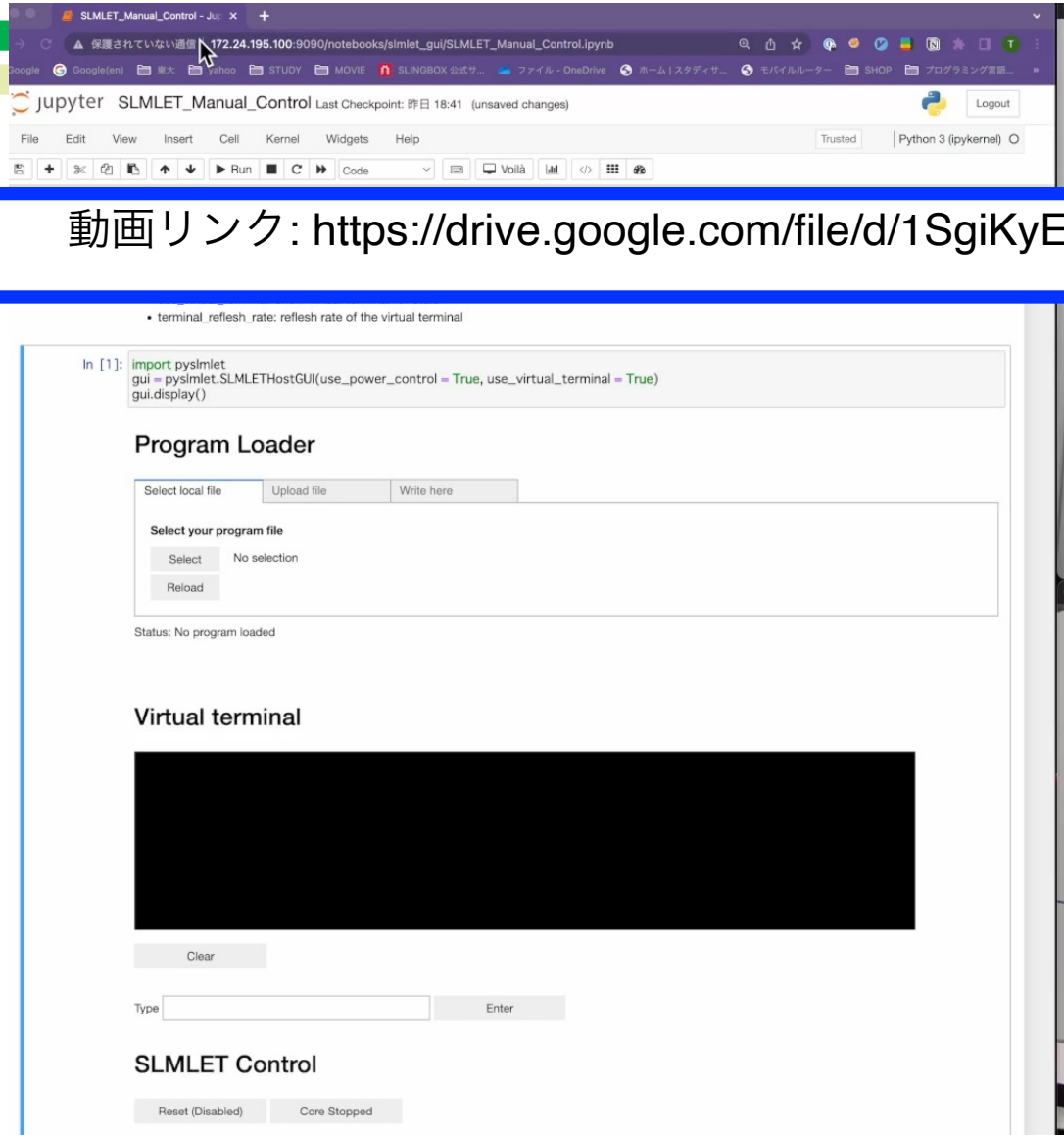
```
19  printf("read data from SLM:\n");
20  for (i = 0; i < FPGA_REG_COUNT; i++) {
21      printf("FPGA Reg %d: %08X\n", i, readSLM(SLM_BLOCK0, (int*)(4 * i)));
22  }
23
```

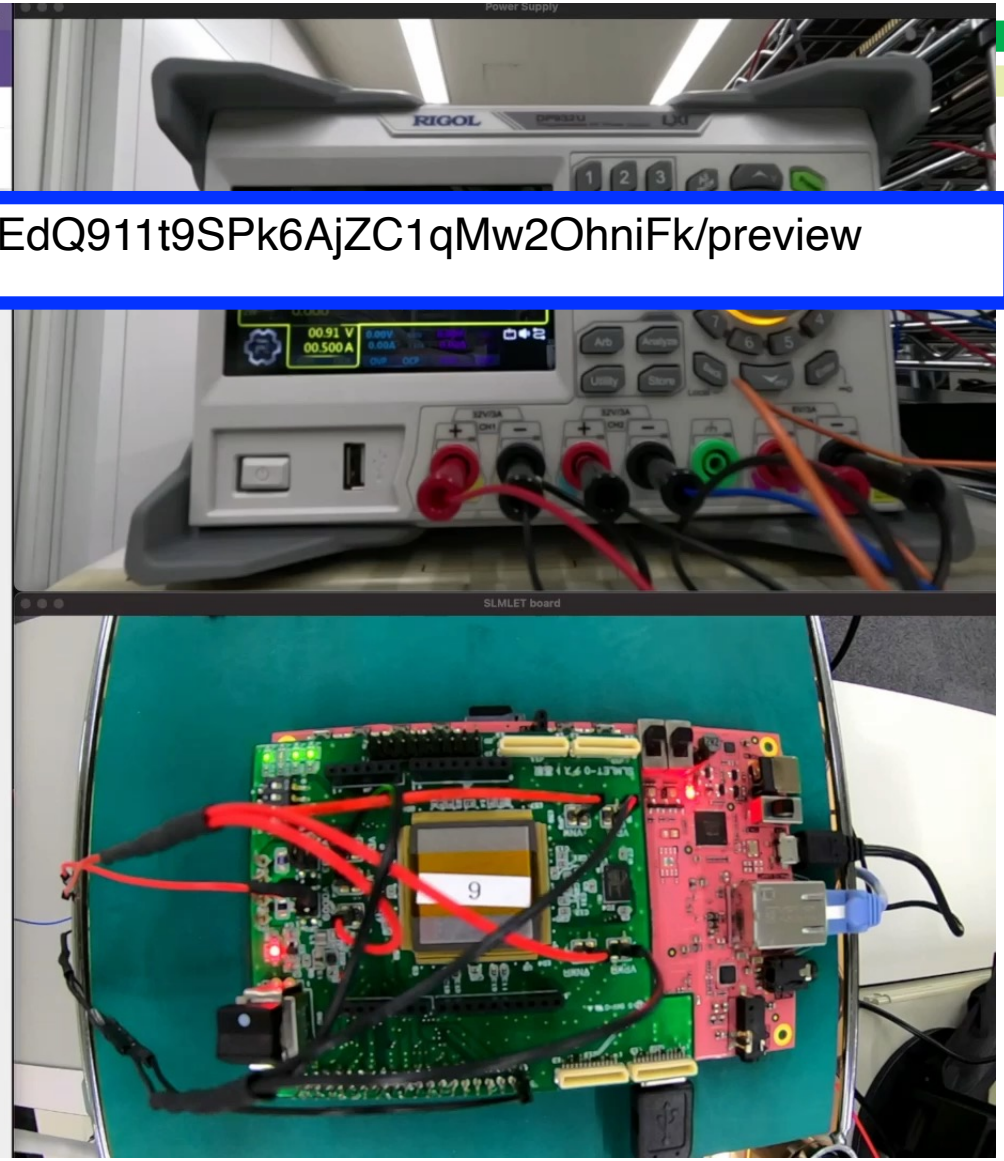
PySLMLET

PySLMLETの構成要素

- PL部の回路制御ドライバ
  - データ転送、制御信号の送信、クロック変更操作
  - データ転送はasyncioを用いた非同期処理
- Jupyter Notebook用GUI
  - 手動実行用GUI
  - 連続自動テスト用GUI
- 電源装置操作用ライブラリ
  - VISA/SCIPに対応したプログラマブル電源と連動

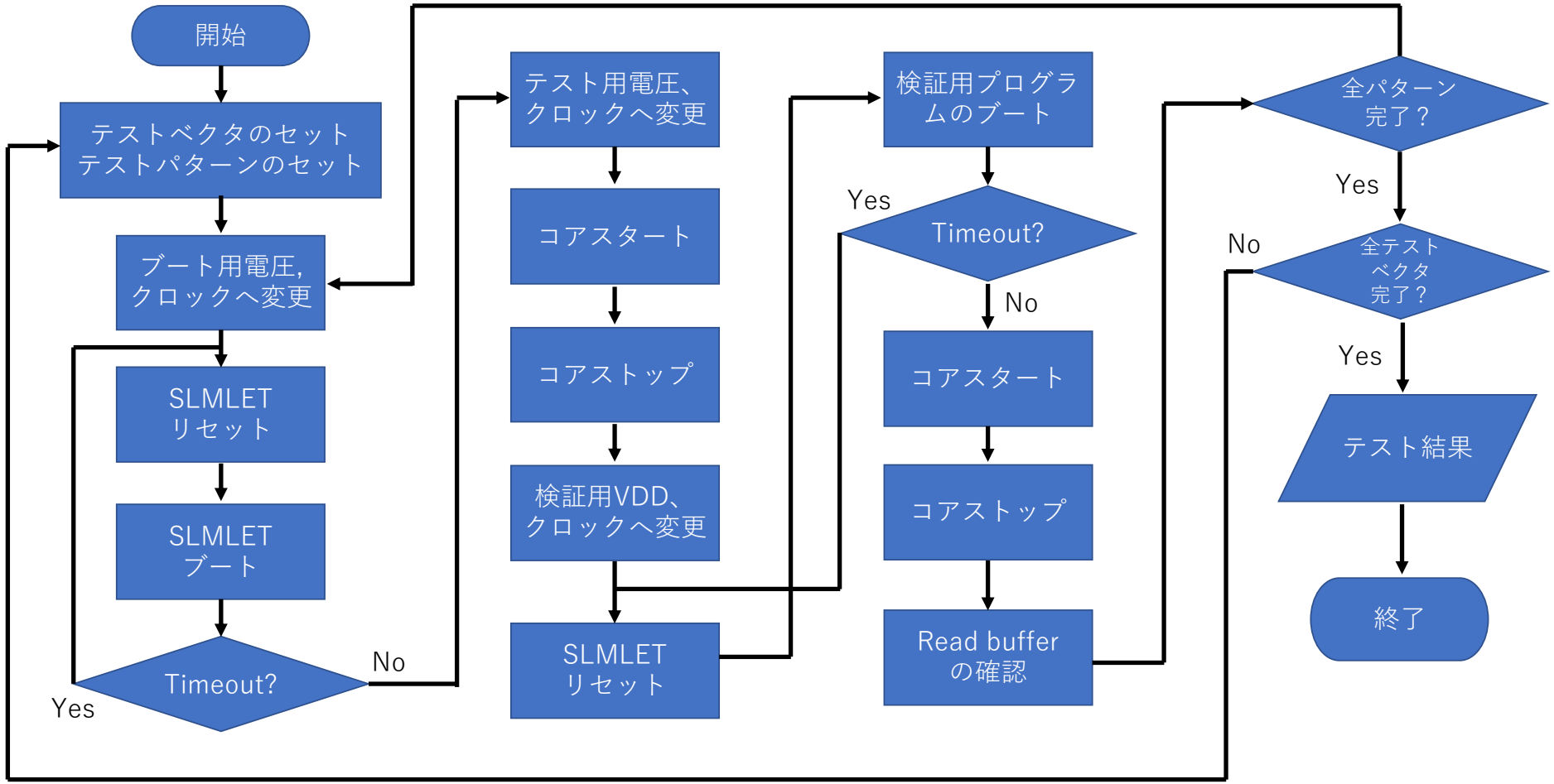
手動実行用GUI使用時の様子



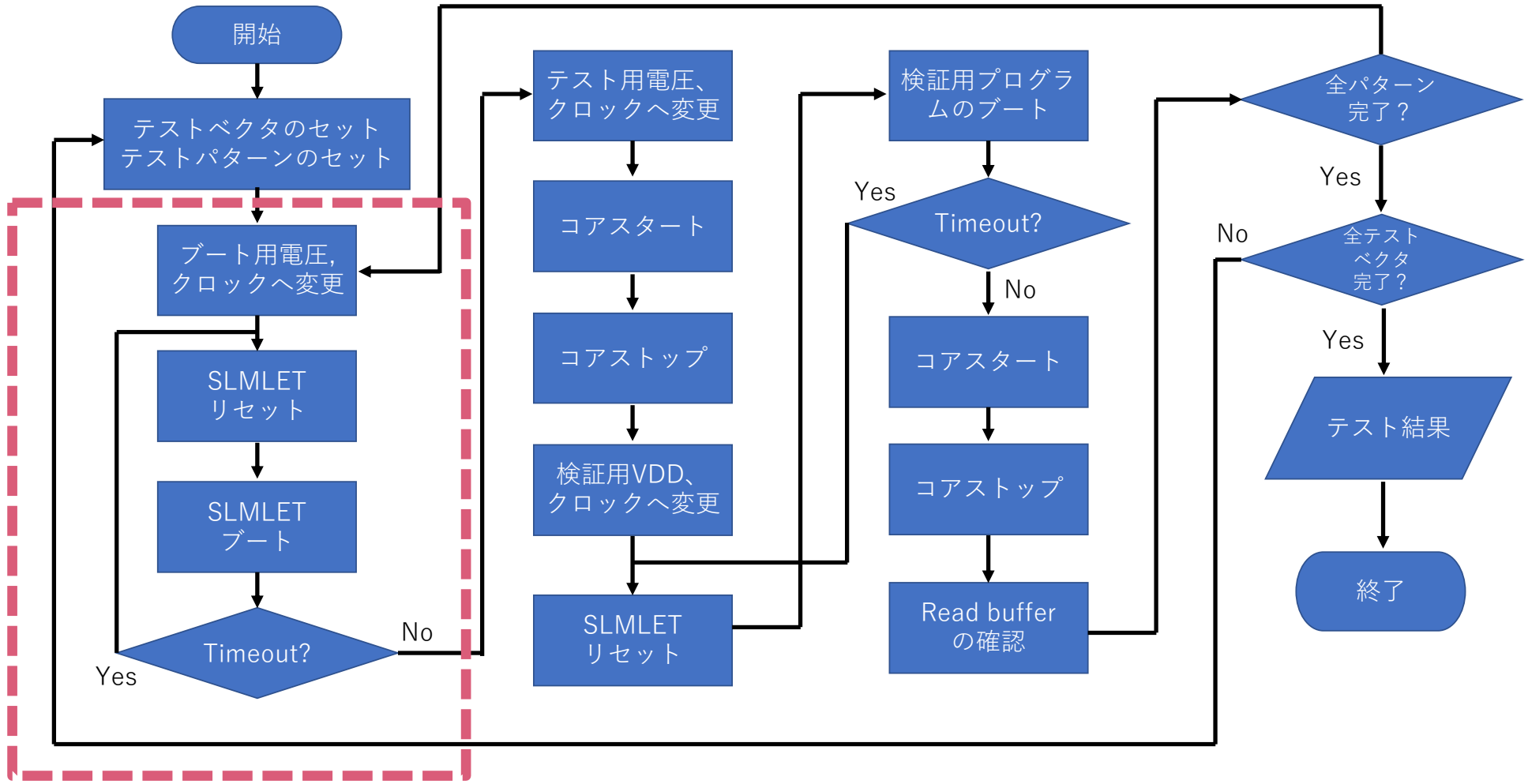


動画リンク: https://drive.google.com/file/d/1SgiKyEdQ911t9SPk6AjZC1qMw2OhniFk/preview

自動テストフロー

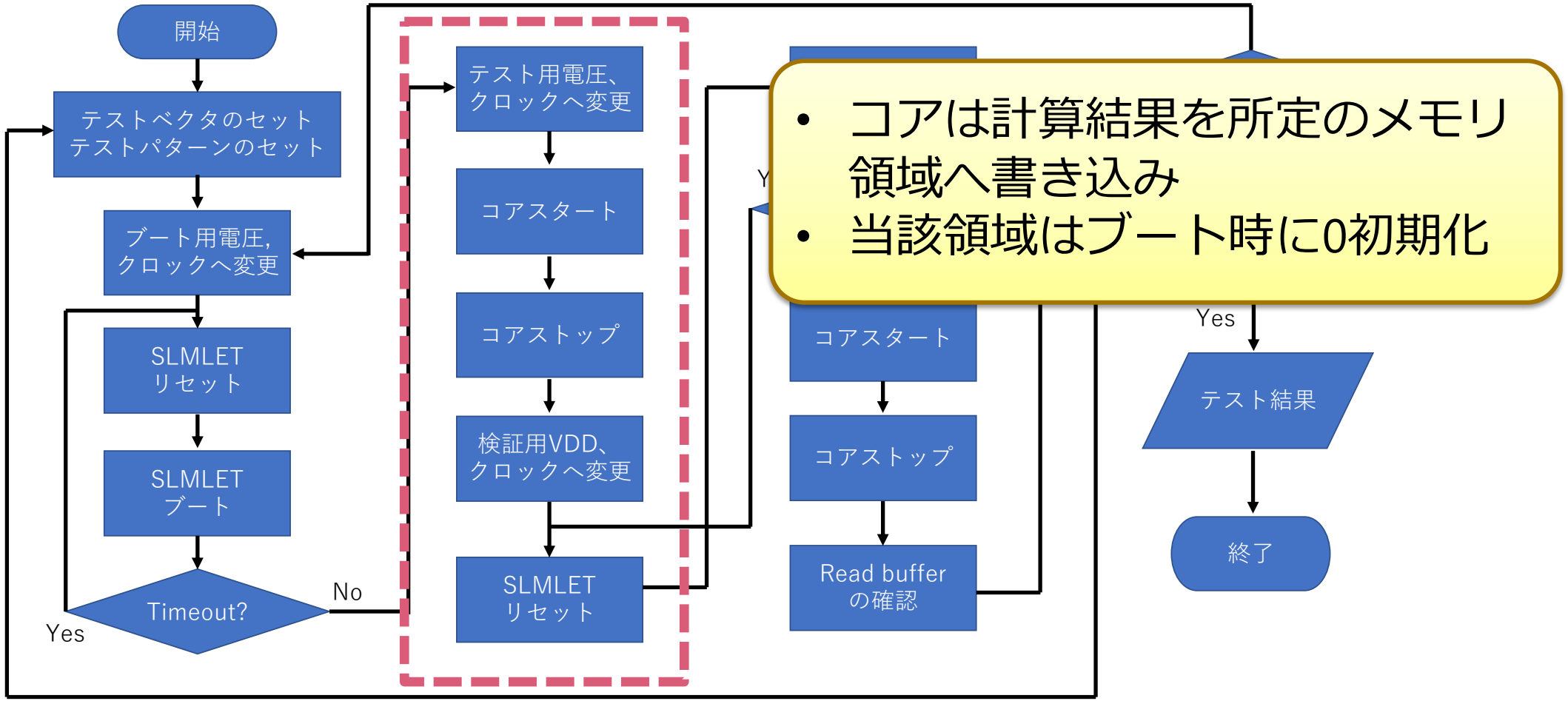

開始
テストベクタのセット
テストパターンのセット
ブート用電圧,
クロックへ変更
SLMLET
リセット
SLMLET
ブート
Timeout?
Yes
No
テスト用電圧、
クロックへ変更
コアスタート
コアストップ
検証用VDD、
クロックへ変更
SLMLET
リセット
検証用プログラ
ムのブート
Yes
Timeout?
No
コアスタート
コアストップ
Read buffer
の確認
全パターン
完了？
Yes
No
全テスト
ベクタ
完了？
Yes
テスト結果
終了

自動テストフロー



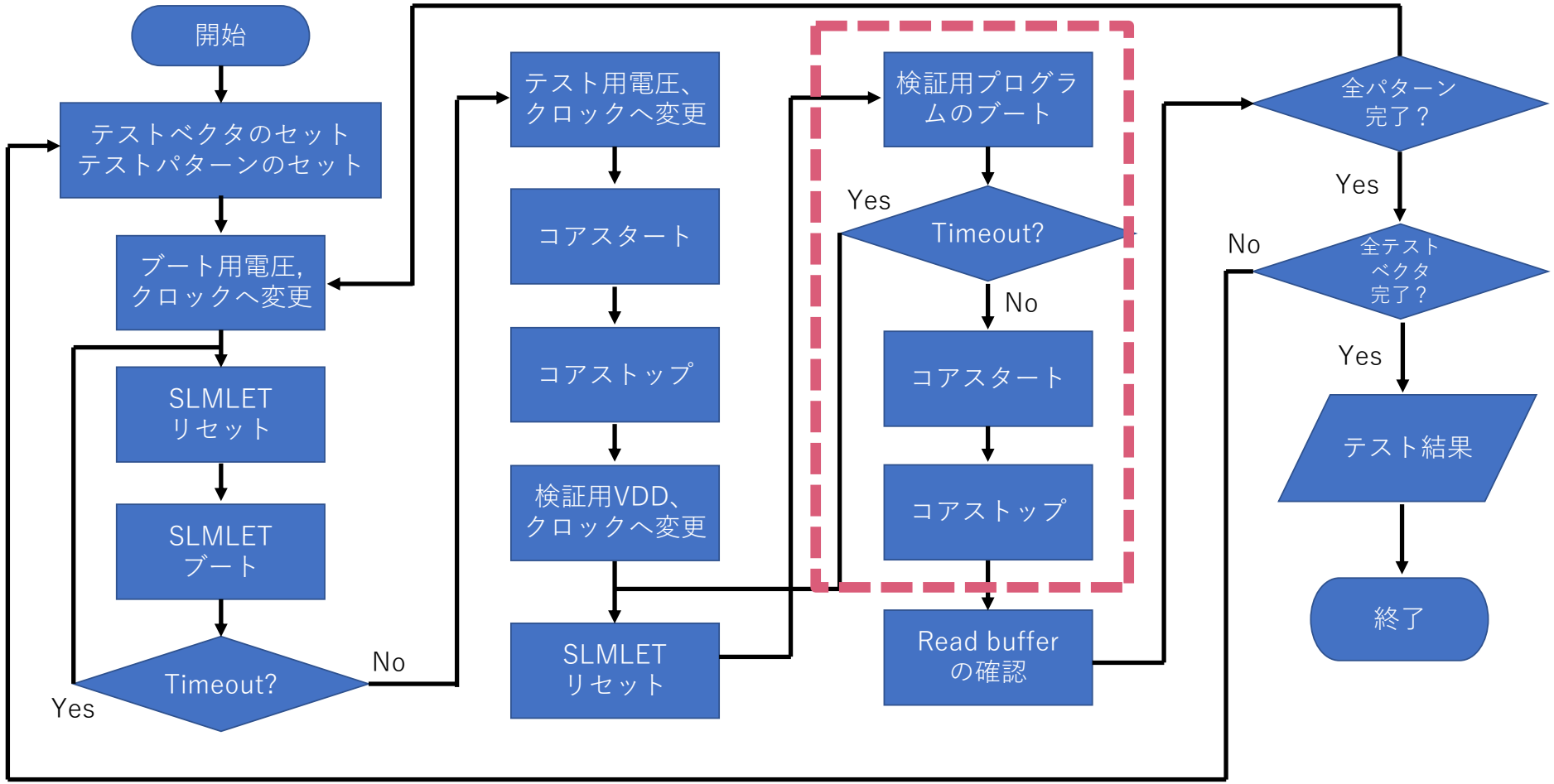
STEP1: プログラムのブートは信頼できる電圧、クロックを利用

自動テストフロー



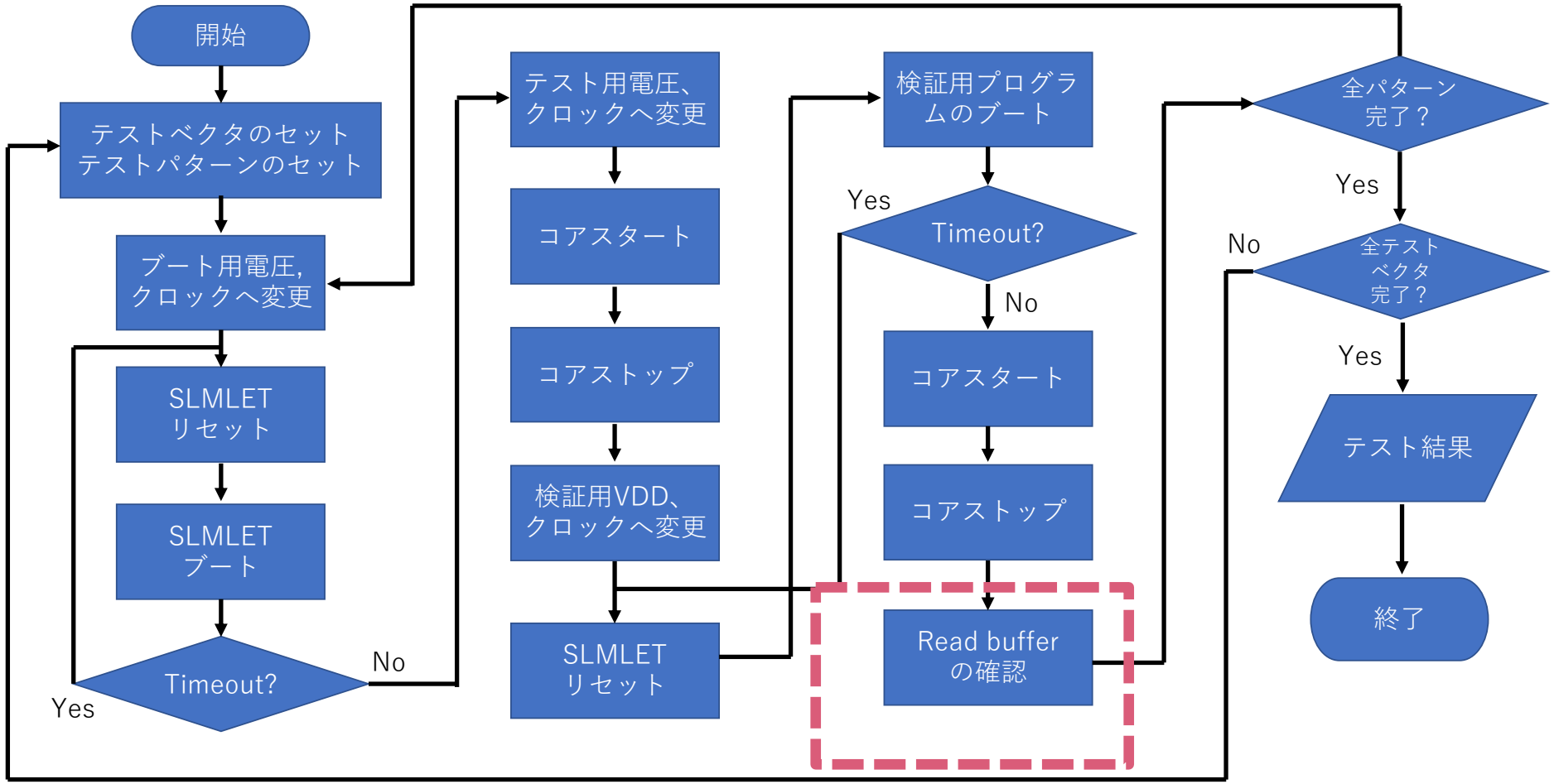
STEP2: テストしたい電圧、クロックに変更し、コアをスタート

自動テストフロー



STEP3: 結果をメモリから読み出す検証用プログラムをブート

自動テストフロー



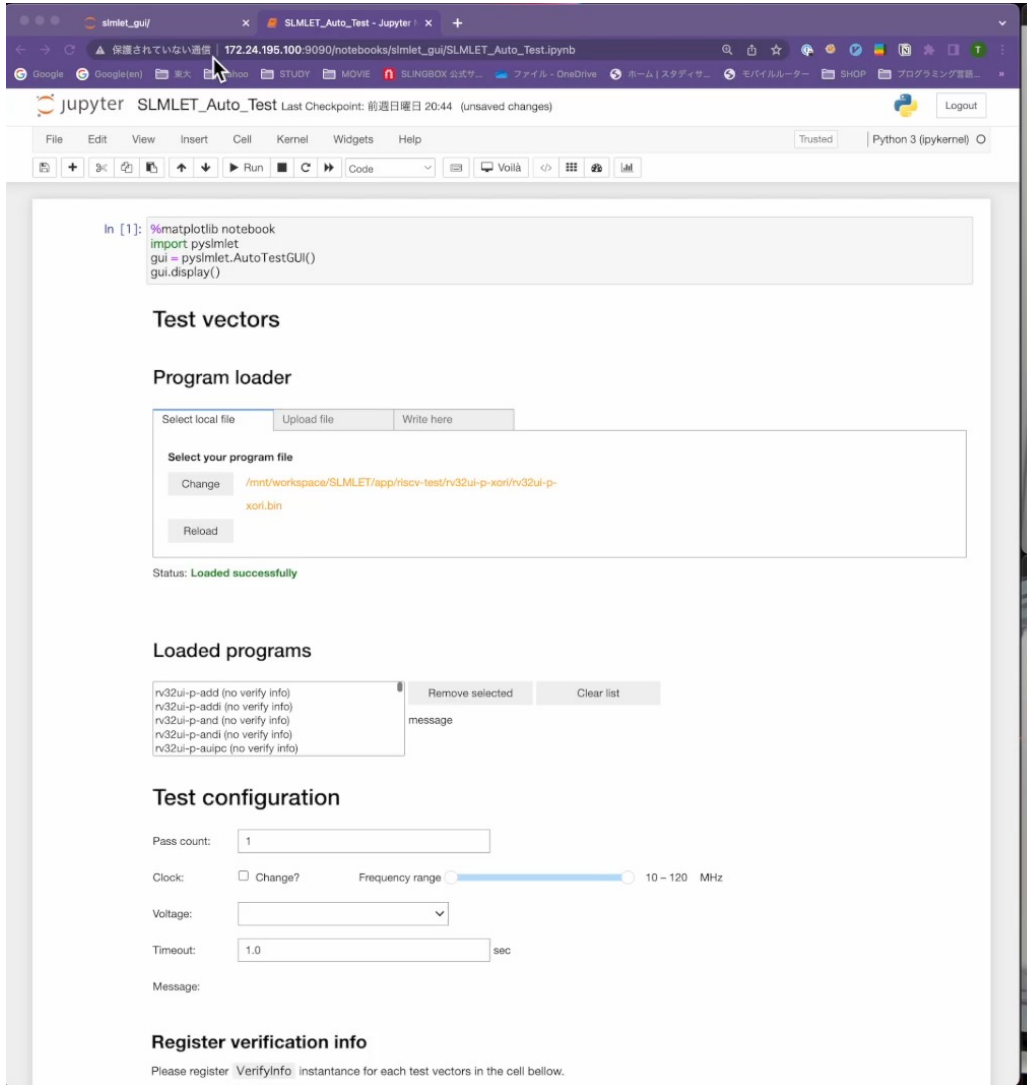
STEP4: 読み出した結果が想定される結果と一致するか確認

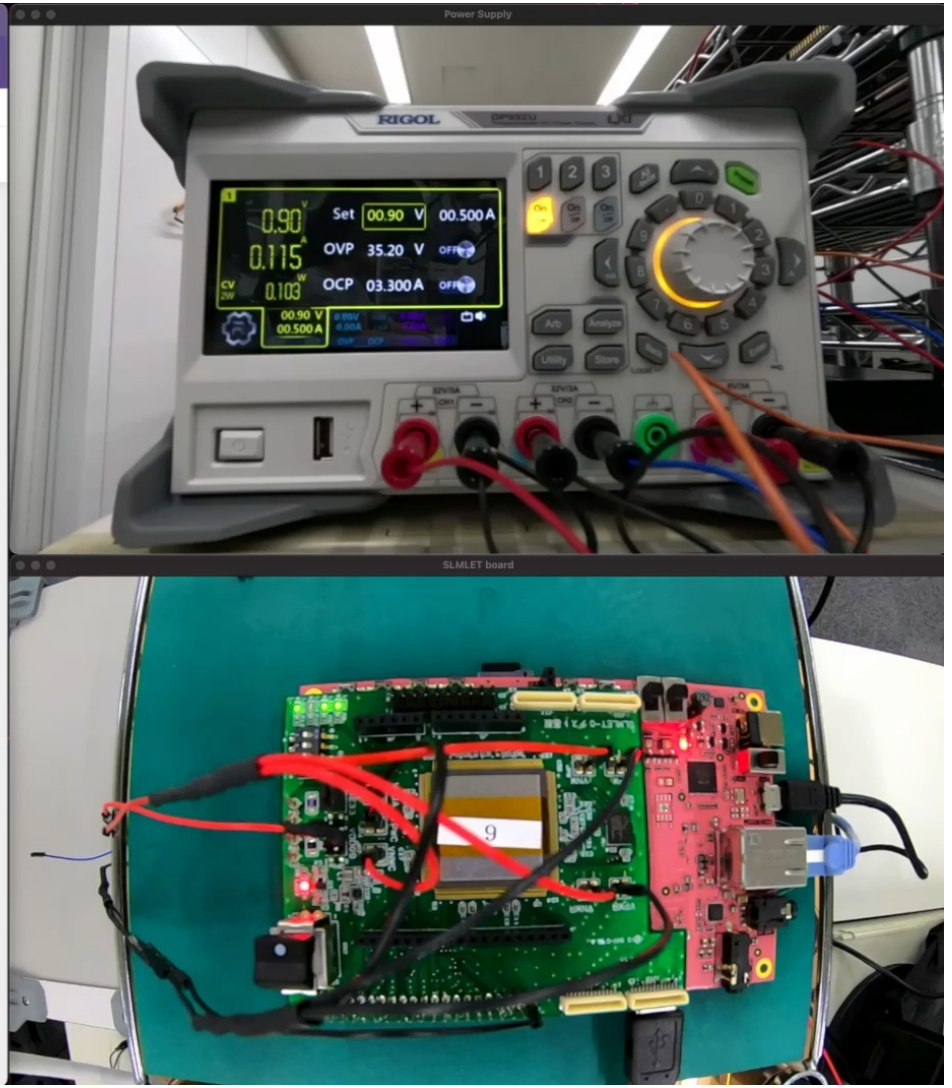
テスト・評価結果

機能テストの結果

- RISC-VコアのISAが正しく動作するか検証
- 公式のriscv-testsを利用
  - rv32ui-p-*のテストセット
  - 計39種のテストが含まれる
- 電源電圧は標準電圧0.90V, 動作周波数は20MHz
- パスしなかったテストベクタ
  - rv32ui-p-fence_i: riscv-miniではfench命令は未実装のため
  - rv32ui-p-lhu,lh,sh,sb: chiselのバグにより誤ったVerilogが生成されたことによる

機能テストの様子

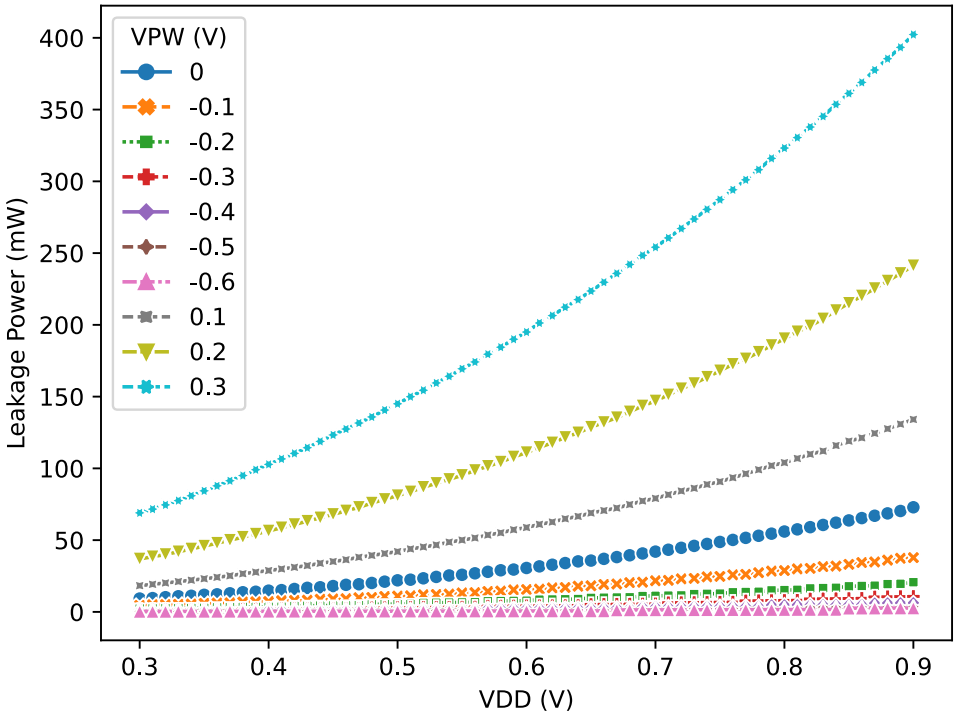

slmlet_gui/
SLMLET_Auto_Test - Jupyter
172.24.195.100:9090/notebooks/slmlet_gui/SLMLET_Auto_Test.ipynb
jupyter SLMLET_Auto_Test Last Checkpoint: 前週日曜日 20:44 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
Voilà
In [1]:
%matplotlib notebook
import pyslmlet
gui = pyslmlet.AutoTestGUI()
gui.display()
Test vectors
Program loader
Select local file
Upload file
Write here
Select your program file
Change
/mnt/workspace/SLMLET/app/riscv-test/rv32ui-p-xori/rv32ui-p-xori.bin
Reload
Status: Loaded successfully
Loaded programs
rv32ui-p-add (no verify info)
rv32ui-p-addi (no verify info)
rv32ui-p-and (no verify info)
rv32ui-p-andi (no verify info)
rv32ui-p-auipc (no verify info)
Remove selected
Clear list
message
Test configuration
Pass count:
1
Clock:
Change?
Frequency range
10 – 120 MHz
Voltage:
Timeout:
1.0
sec
Message:
Register verification info
Please register VerifyInfo instantance for each test vectors in the cell bellow.


Power Supply
RIGOL
Set 00.90 V 00.500 A
OVP 35.20 V
OCP 03.300 A
SLMLET board

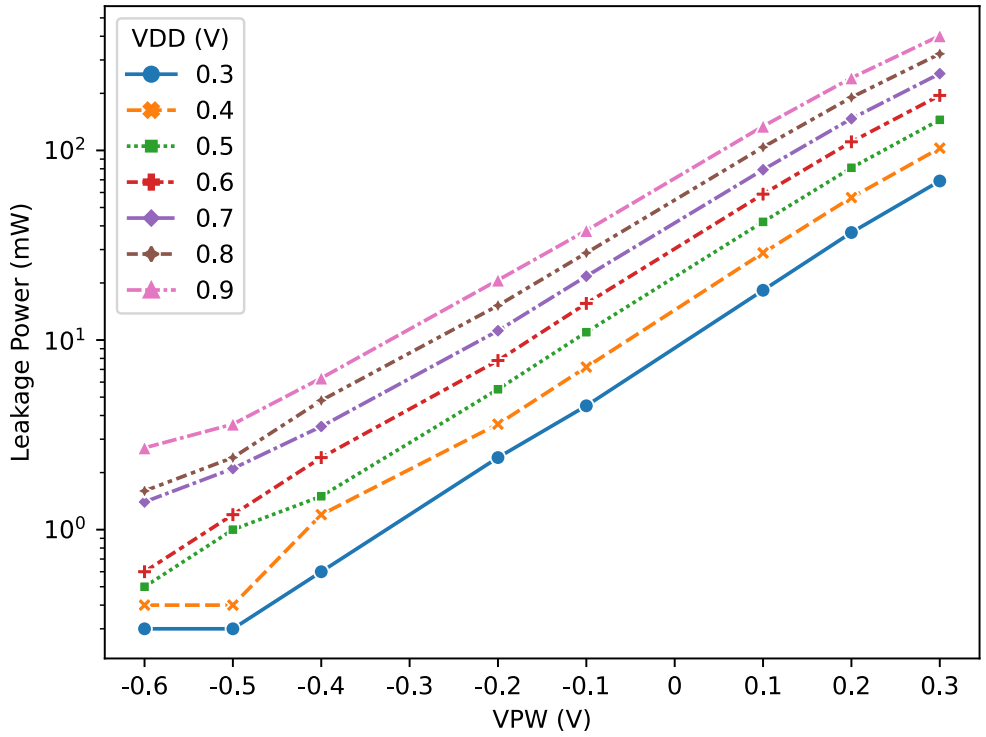
ボディバイアス制御によるリーク電力の増減

- 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

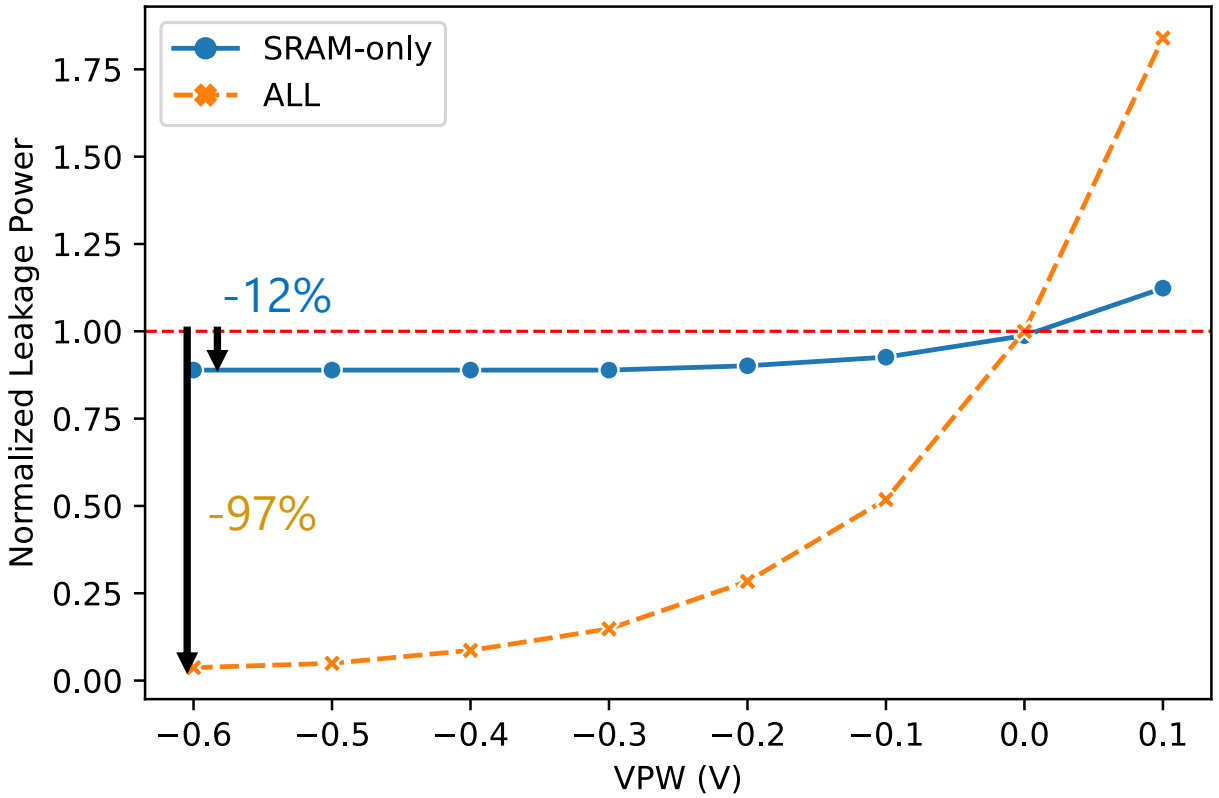


VPW-リーク電力の関係

リーク電力の内訳を推定

■ 試作チップではSRAMとロジック部で別々のボディバイアスドメイン

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



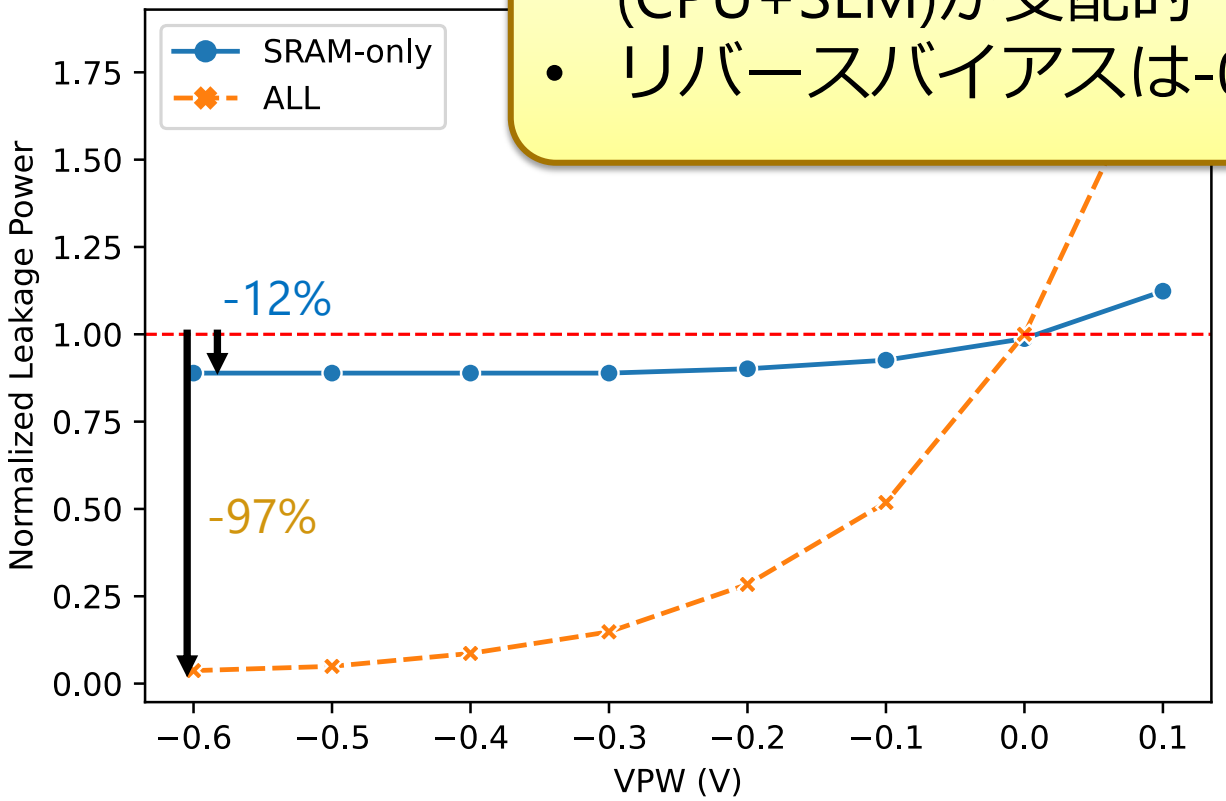
全体で共通のバイアス電圧を印加した時と、SRAMだけに印加した時の違い (ゼロバイアス時で正規化)

リーク電力の内訳を推定

■ 試作チップではSRAMとロジック部

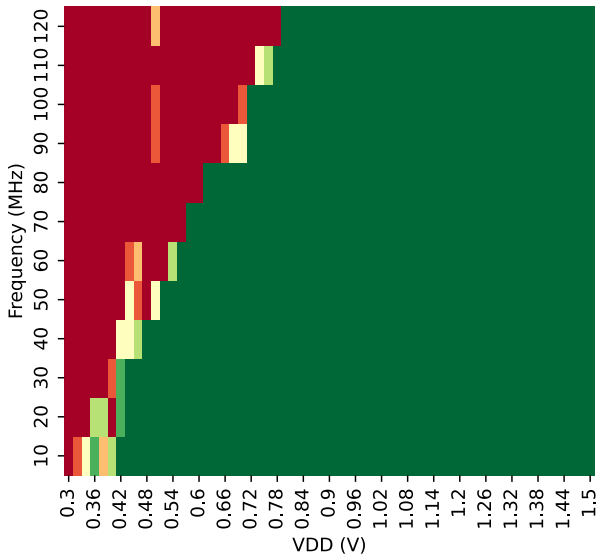
- SRAM部は全体の10%程度で、ロジック部(CPU+SLM)が支配的
- リバースバイアスは-0.3V程度までが妥当

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

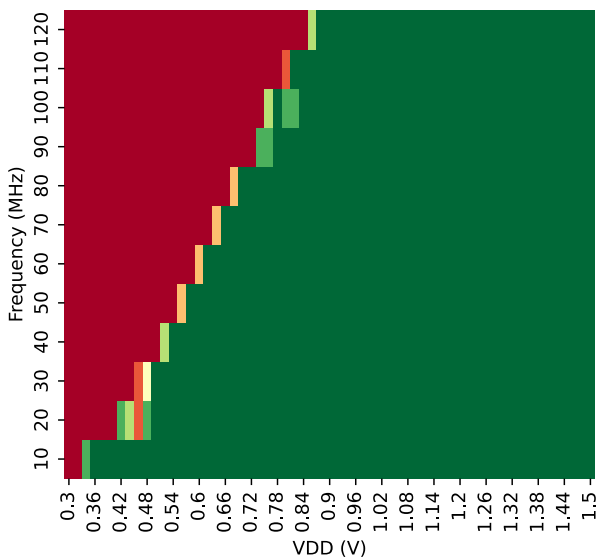


全体で共通のバイアス電圧を印加した時と、SRAMだけに印加した時の違い (ゼロバイアス時で正規化)

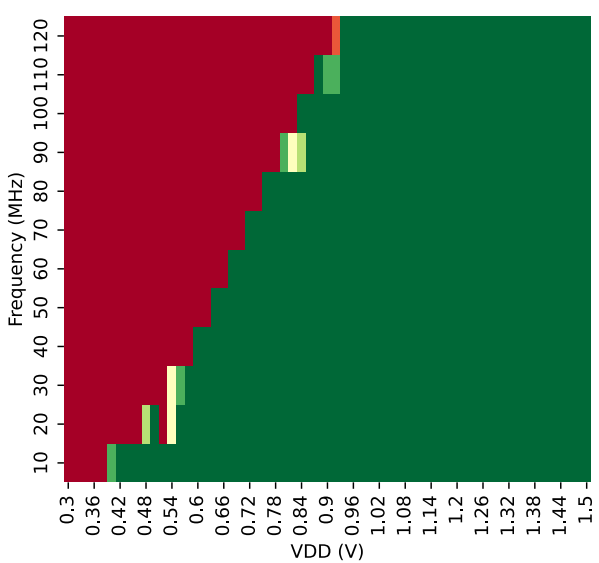
動作範囲とボディバイアス制御の効果



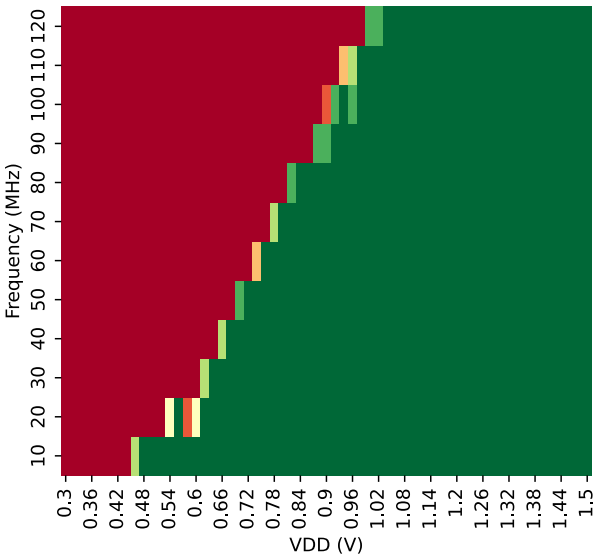
VPW=0.0 V



VPW=-0.2 V



VPW=-0.2 V

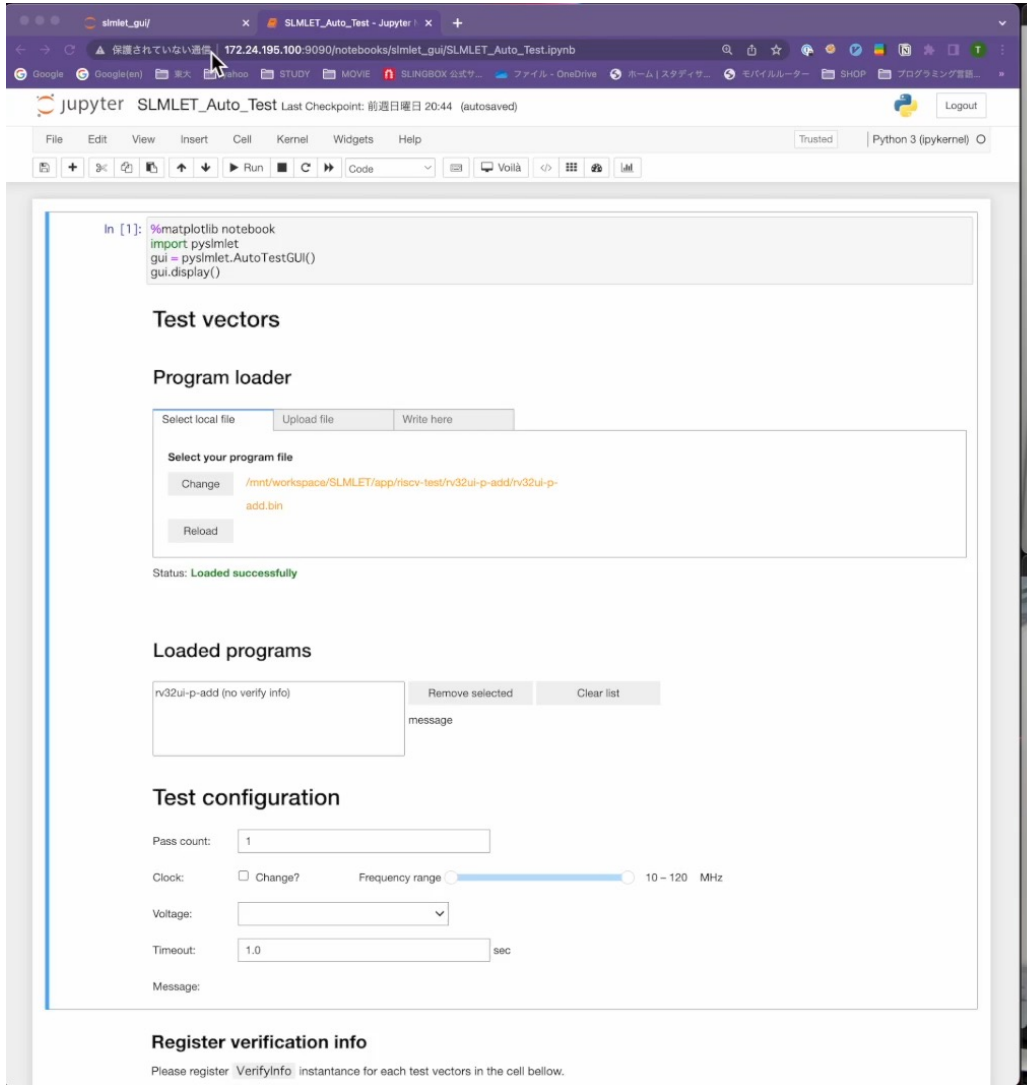


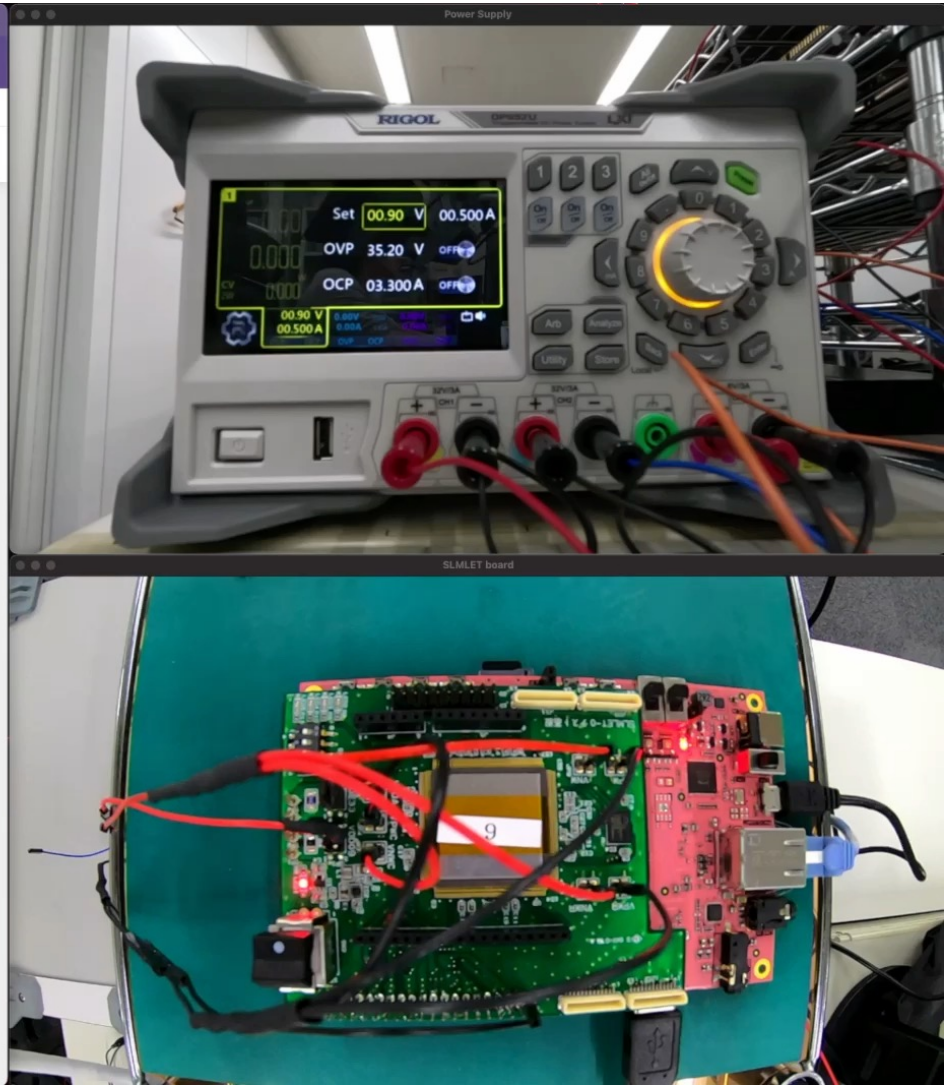
VPW=-0.3 V

各VDD,動作周波数におけるテストパス数

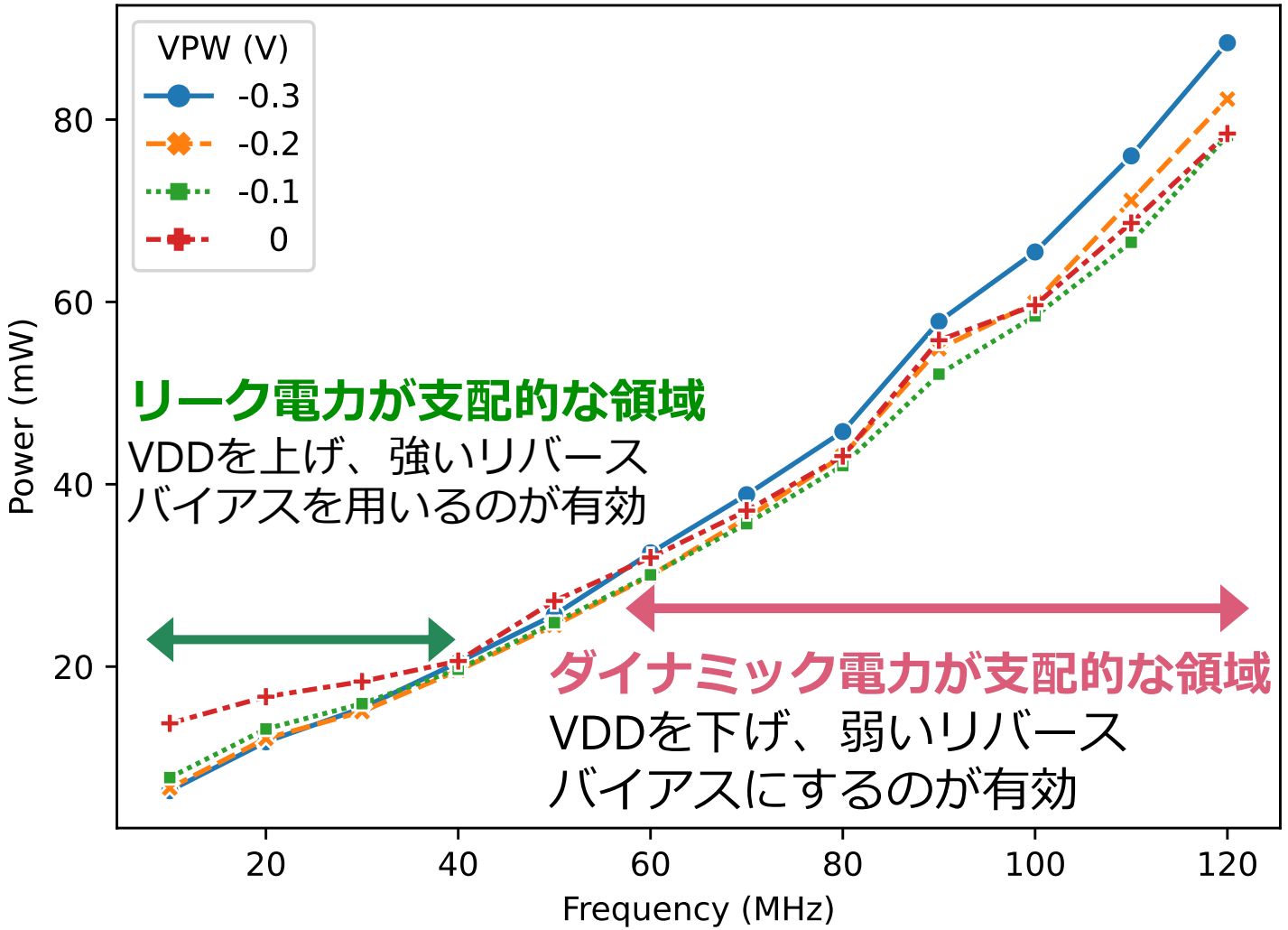
- 動作周波数に概ね比例して動作可能最小電圧が大きく
- リバースバイアスが強くなるにつれ動作可能最小電圧が大きく

電圧条件を変更した自動テストの様子





総消費電力の比較



各動作周波数における消費電力

【参考】

一般的なCMOS LSIの電力モデル式

$$P_{\text{total}} = P_{\text{leak}} + \alpha C V_{\text{DD}}^2 f$$

リーク電力　ダイナミック電力

サブスレッショルドリークのモデル式

$$I_{\text{sub}} = I_{\text{off}} 10^{\frac{V_{gs}+\eta(V_{ds}-V_{DD})-k_\gamma V_{sb}}{S}} \left(1 - e^{\frac{-V_{ds}}{v_T}}\right)$$

まとめ

- エッジ,MEC向けのSoC SLMLETの試作チップテスト環境を構築
  - PYNQ-Z2に実装したSLMLET制御用回路
  - ソフトウェア開発キット
  - Jupyuter Notebook,FPGA,測定装置を連携させるPythonライブラリ PySLMLET
- 構築したシステムによりテスト、評価を自動化
- 初期評価結果として見えてきたこと
  - リーク電力は90%弱をロジック部が占める
  - -0.3 V程度のリバースバイアスが特に動作周波数の低い場合に有効
  - 一方で、動作周波数が高い場合はゼロバイアスにし、VDDを下げるのが有効
- 今後の展望
  - HyperBus, SLM部を利用した際の評価を実施