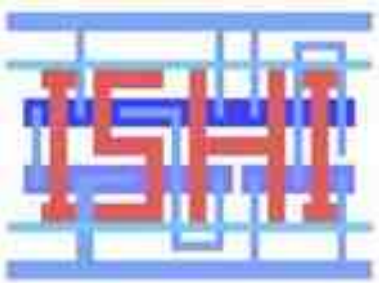
同人誌 Vol.3 予告編

ICチップ設計の醍醐味！

CPUの論理設計

2026年5月6日　Rev.02

ISHI会



https://ishi-kai.org

mmTech

https://github.com
/munetomo-maruyama

圓山 宗智（まるやま　むねとも）
えんざん そうち

e-mail　munetomo@ishi-kai.org
munetomo.maruyama@gmail.com
X　@Processing_Unitc

圓山 宗智（まるやま　むねとも）

えんざん そうち

- ペンネーム
- 仕事　半導体メーカ　マイコン・SoCの開発
- 趣味　マイコン・SoC・FPGA・回路設計・基板作り・C・Python・…
- ISHI会
  - 2024年　渋谷のイベント参加
  - 2025年　フェニテック・シャトル
    - インバータ・ハンズオン
    - 会社の若手エンジニア教育として
    - 個人的には、初のアナログ回路設計　R2R OPAMP
  - Tiny Tapeout　趣味で2発
  - 2026年　Open Silicon Magazine
    - ページ書きすぎて（140p）Vol.3に追いやられる
  - AMラジオを作ろうチーム

■picoCPUでCPU設計を体験しよう

（I/O誌2026年4月号掲載、ISHI会同人誌Vol.3掲載予定）

■世界最初のマイクロ・プロセサ4004の設計とシリコン化

（I/O誌2025年11月号掲載）

■シンプルなチューリング完全CPU bfCPUの設計とシリコン化

（I/O誌2026年4月号掲載、ISHI会同人誌Vol.3掲載予定）

■実用RISC-V CPUの自作と22nm 商用チップへの実装

■CPU設計とそのシリコン化の楽しさ

System Verilogの解説教材として

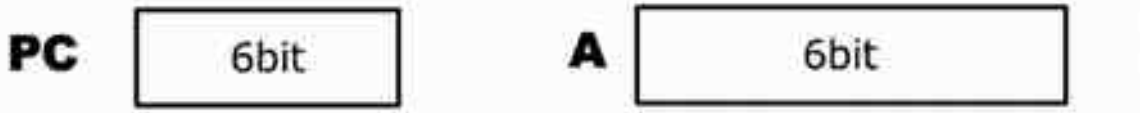
picoCPUで
CPU設計を体験しよう

https://github.com/munetomo-maruyama/picoCPU.git

（I/O誌2026年4月号掲載、ISHI会同人誌Vol.3掲載予定）

超簡易picoCPUの仕様を定義

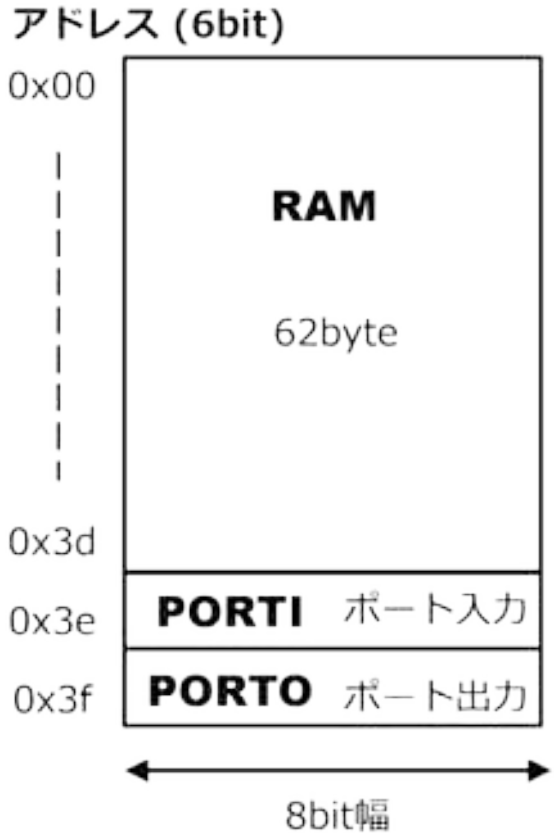
●最小限のCPU内リソース：
プログラム・カウンタPC(6bit)とアキュムレータA(8bit)



●命令セット：4命令のみ

種　類	オペコード	命　令	動　作
加算	00iiiiii	ADD #imm6	A ← A + imm6 (符号拡張)
条件分岐	01aaaaaa	JNZ @addr6	if(A!=0) PC ← addr6
ロード	10aaaaaa	LDA @addr6	A ← MEM[addr6]
ストア	11aaaaaa	STA @addr6	MEM[addr6] ← A

●メモリ空間：6bit空間
最後の2バイトはI/O



超簡易picoCPUのRTL設計

mmRISC

●CPU部のRTL cpu.sv：データ・パス部と制御部に分けて記述

制御部のステート定義

```
//---------------------------------
// Control State
//---------------------------------
`define STATE_INIT   3'h0
`define STATE_FETCH  3'h1
`define STATE_DECODE 3'h2
`define STATE_ADD    3'h3
`define STATE_JNZ    3'h4
`define STATE_LDA    3'h5
`define STATE_LDA2   3'h6
`define STATE_STA    3'h7
```

モジュールCPUの定義

```
//---------------------------------
// CPU
//---------------------------------
module CPU
(
    input  logic CLK,
    input  logic RES,
    //
    output logic [5:0] ADDR,
    output logic       RE,
    output logic       WE,
    output logic [7:0] WDATA,
    input  logic [7:0] RDATA
);
```

モジュール定義と入出力信号の定義

データパス部

```
//---------------------------------
// CPU Data Path : Instruction Code
//---------------------------------
logic [7:0] icode;     // Instruction Code
logic       icode_get; // Get icode
logic [1:0] opcode;    // Operation Code
//
assign opcode = RDATA[7:6];
//
always_ff @(posedge CLK, posedge RES)
begin
    if (RES)
        icode <= 8'h00;
    else if (icode_get)
        icode <= RDATA;
end

//---------------------------------
// CPU Data Path : Program Counter
//---------------------------------
logic [5:0] pc;       // Program Counter
logic [5:0] pc_next;  // Next PC
logic       pc_inc;   // Increment PC
logic       pc_jmp;   // Jump PC
//
assign pc_next = icode[5:0];
//
always_ff @(posedge CLK, posedge RES)
begin
    if (RES)
        pc <= 6'h00;
    else if (pc_inc)
        pc <= pc + 6'h01;
    else if (pc_jmp)
        pc <= pc_next;
end

//---------------------------------
// CPU Data Path : Register A
//---------------------------------
logic [7:0] regA;     // Register A
logic       regA_ld;  // Load to A
logic       regA_add; // Add to A
logic       regA_nz;  // A is Not Zero
logic [7:0] imm;      // Immediate Data
//
assign imm = {icode[5], icode[5], icode[5:0]};
//
always_ff @(posedge CLK, posedge RES)
begin
    if (RES)
        regA <= 8'h00;
    else if (regA_ld)
        regA <= RDATA;
    else if (regA_add)
        regA <= regA + imm;
end
//
assign regA_nz = |regA;

//---------------------------------
// CPU Data Path : Memory Access
//---------------------------------
logic addr_pc;
logic addr_rw;
//
assign ADDR = (addr_pc)? pc
            : (addr_rw)? icode[5:0]
            : 6'h00;
assign WDATA = regA;
```

メモリMEMからの命令コードのフェッチ

プログラム・カウンタPCの制御
ーインクリメント
ージャンプ

CPU内レジスタAの制御
ーMEMからのロード
ー定数値の加算

メモリMEMへのアクセス制御

制御部

```
//---------------------------------
// CPU Control Logic
//---------------------------------
logic [2:0] state;
logic [2:0] state_next;
//
always_ff @(posedge CLK, posedge RES)
begin
    if (RES)
        state <= `STATE_INIT;
    else
        state <= state_next;
end
//
always_comb
begin
    // Set Default Value
    icode_get = 1'b0;
    pc_inc    = 1'b0;
    pc_jmp    = 1'b0;
    regA_ld   = 1'b0;
    regA_add  = 1'b0;
    addr_pc   = 1'b0;
    addr_rw   = 1'b0;
    RE        = 1'b0;
    WE        = 1'b0;
    //
    case (state)
        //---------------------------------
        // Initial State
        `STATE_INIT:
        begin
            state_next = `STATE_FETCH;
        end
        //---------------------------------
        // Instruction Fetch
        `STATE_FETCH:
        begin
            RE = 1'b1;
            addr_pc = 1'b1;
            state_next = `STATE_DECODE;
        end
        //---------------------------------
        // Instruction Decode
        `STATE_DECODE:
        begin
            icode_get = 1'b1;
                if (opcode == 2'b00) state_next = `STATE_ADD;
            else if (opcode == 2'b01) state_next = `STATE_JNZ;
            else if (opcode == 2'b10) state_next = `STATE_LDA;
            else if (opcode == 2'b11) state_next = `STATE_STA;
            else state_next = `STATE_INIT; // never reach here
        end
        //---------------------------------
        // ADD
        `STATE_ADD:
        begin
            regA_add = 1'b1;
            pc_inc   = 1'b1;
            state_next = `STATE_FETCH;
        end
        //---------------------------------
        // JNZ
        `STATE_JNZ:
        begin
            pc_jmp =  regA_nz;
            pc_inc = ~regA_nz;
            state_next = `STATE_FETCH;
        end
        //---------------------------------
        // LDA
        `STATE_LDA:
        begin
            RE = 1'b1;
            addr_rw = 1'b1;
            state_next = `STATE_LDA2;
        end
        `STATE_LDA2:
        begin
            regA_ld = 1'b1;
            pc_inc  = 1'b1;
            state_next = `STATE_FETCH;
        end
        //---------------------------------
        // STA
        `STATE_STA:
        begin
            WE = 1'b1;
            addr_rw = 1'b1;
            pc_inc  = 1'b1;
            state_next = `STATE_FETCH;
        end
        //---------------------------------
        // Default (NOP)
        default:
        begin
            pc_inc  = 1'b1;
            state_next = `STATE_FETCH;
        end
    endcase
end

endmodule
```

次ステートstate_nextとデータ・パス部の制御信号を生成する組み合わせ回路

制御信号のうち後述で上書きしない信号のデフォルト値を設定

リセット後の初期状態からはすぐにフェッチ操作に移行

命令フェッチを実行し命令デコードに移行

命令デコード結果に応じて各命令の実行処理に移行

ADD命令を実行して命令フェッチに移行

JNZ命令を実行して命令フェッチに移行

次命令に行くか、ジャンプするかはregA_nzで判定

LDA命令を実行して命令フェッチに移行

memはアドレス出力の次サイクルにリードデータを出力するので2ステップかける

STA命令を実行して命令フェッチに移行

memはアドレス出力と同時にライト・データを出力するので1ステップで完了

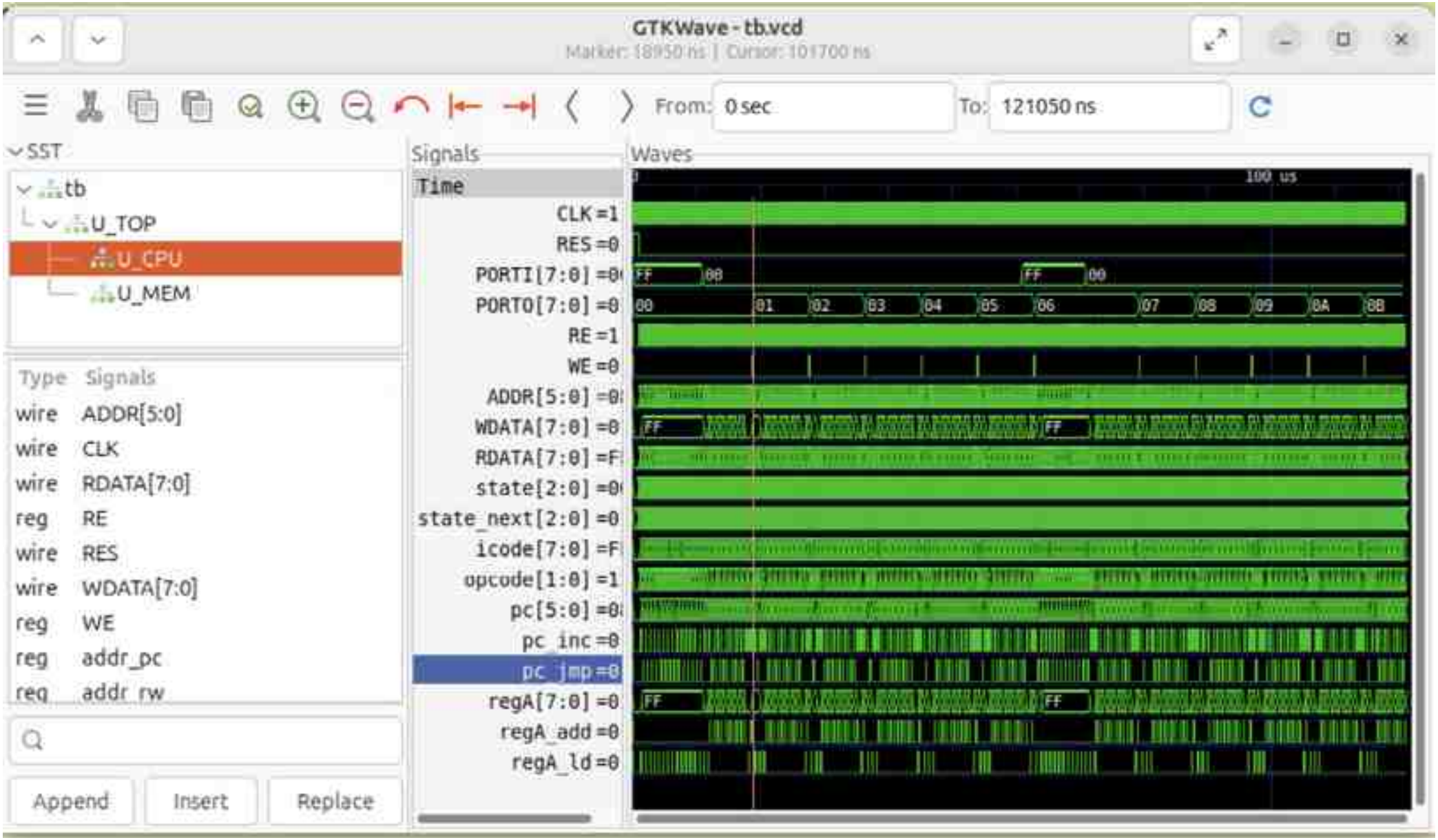
命令デコード結果に該当命令がなければノー・オペレーション

ただし、命令デコードは2ビットのみ見ているのでここには到達しない

モジュール定義の終了

超簡易picoCPUの論理シミュレーション

●PORTI=8'h00なら、PORTOをインクリメントするプログラムの検証（Lチカ）



世界最初のマイクロ・プロセサ 4004の設計とシリコン化

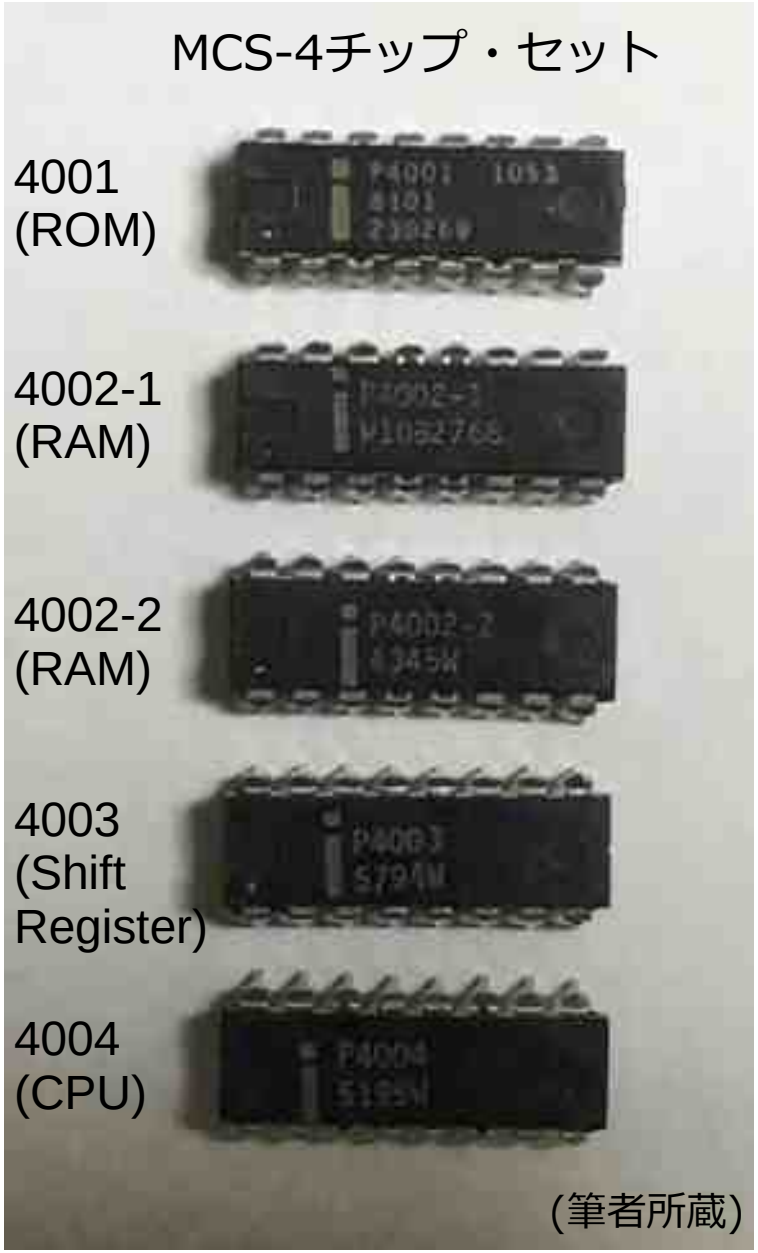
設計 **https://github.com/munetomo-maruyama/MCS4_SYSTEM.git**

Tiny Tapeout **https://github.com/munetomo-maruyama/ttsky25a_MCS4_CPU.git**

セミナ **https://www.zep.co.jp/products/zep4004/**

（I/O誌2025年11月号掲載）

世界初のマイクロ・プロセサ4004



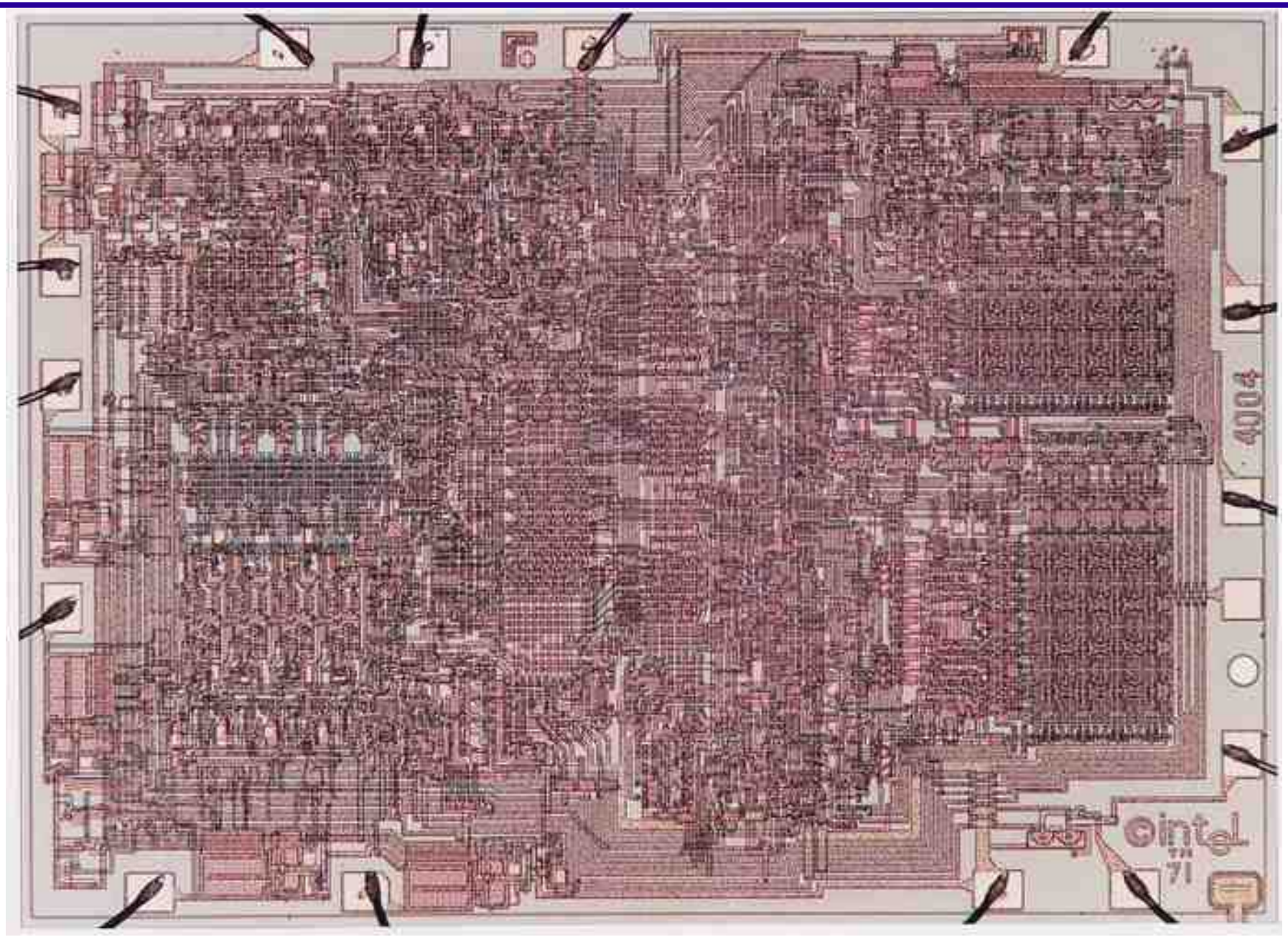
http://www.intel-vintage.info/intelmcs.htmより

4004の初期バージョン。
Gray Traceとよぶ。
eBayで20万円以上で
取引されている。

MCS-4のターゲット
アプリの電卓
Busicom社
141-PF

情報処理学会/コンピュータ博物館/情報処理技術遺産ホームページ
（http://museum.ipsj.or.jp/heritage/2011/Busicom_141-PF.html）より

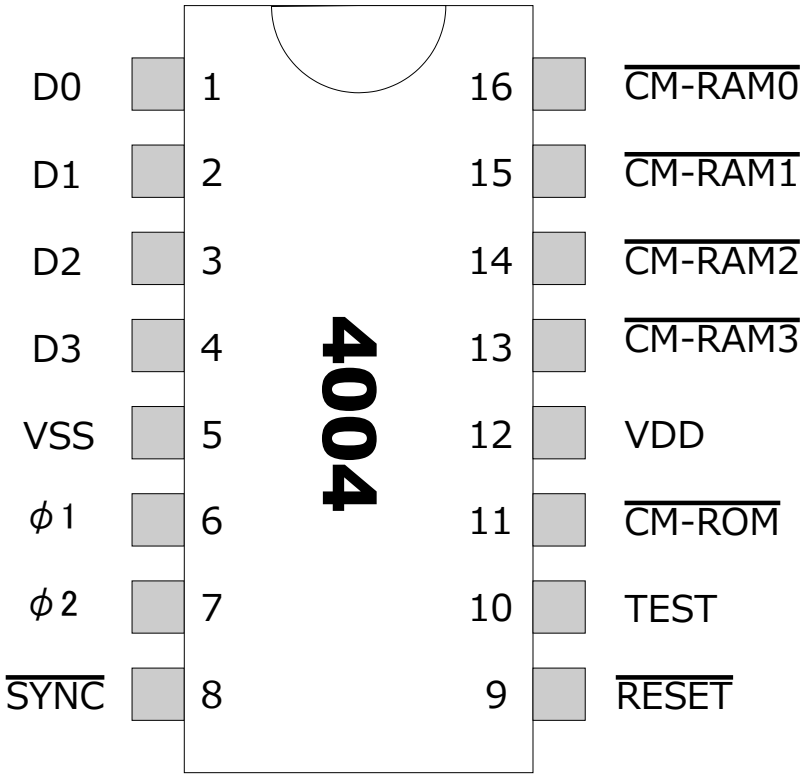
4004 (CPU)のレイアウト・パターン



10um PMOSプロセス、チップ・サイズ=2mm×3mm、2300トランジスタ（約600ゲート）

http://www.theregister.co.uk/Print/2011/11/15/the_first_forty_years_of_intel_microprocessors/より引用

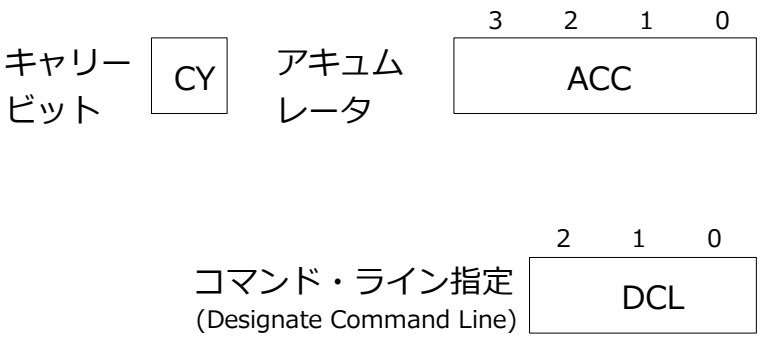
4004 (CPU)の端子配置と端子機能

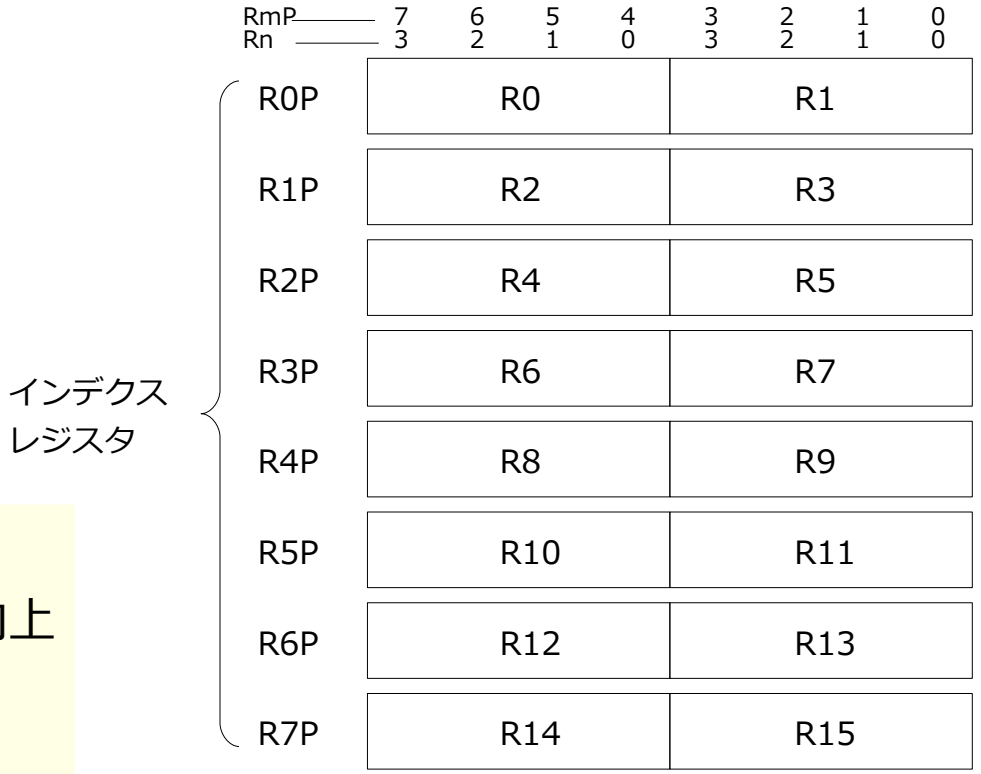


4ビットCPU

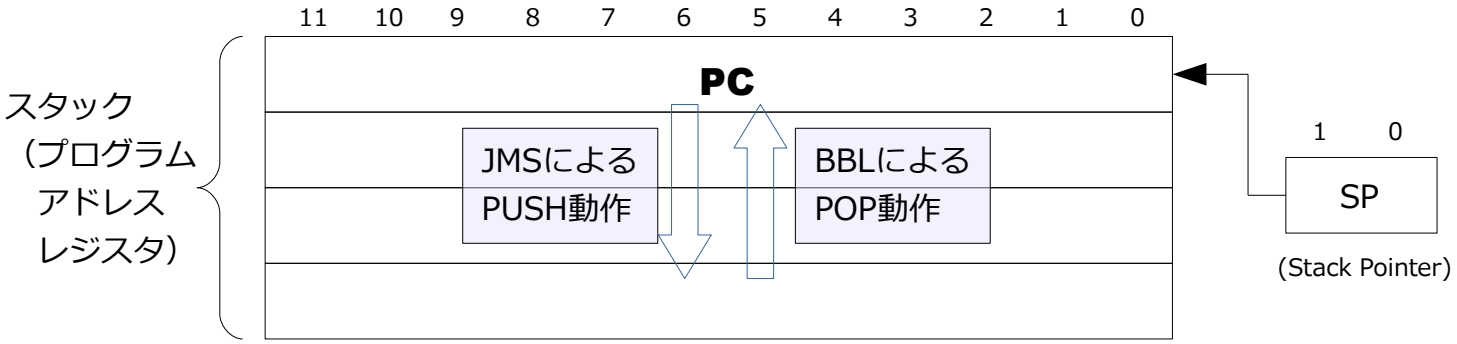
端子番号	端子名	入出力	端子機能	備考
1	D0	In/Out	データ・バス（最下位ビット）	
2	D1	In/Out	データ・バス	
3	D2	In/Out	データ・バス	
4	D3	In/Out	データ・バス（最上位ビット）	
5	VSS	Power	電源GND	
6	φ1	In	2相クロック入力	
7	φ2	In	2相クロック入力	
8	$\overline{\text{SYNC}}$	Out	同期信号出力	
9	$\overline{\text{RESET}}$	In	リセット入力	
10	TEST	In	条件分岐命令(JCN)への条件入力	
11	$\overline{\text{CM-ROM}}$	Out	ROM用コマンド制御出力	
12	VDD	Power	電源VDD（-15V）	
13	$\overline{\text{CM-RAM3}}$	Out	RAM用コマンド制御3出力	DCLで指定
14	$\overline{\text{CM-RAM2}}$	Out	RAM用コマンド制御2出力	DCLで指定
15	$\overline{\text{CM-RAM1}}$	Out	RAM用コマンド制御1出力	DCLで指定
16	$\overline{\text{CM-RAM0}}$	Out	RAM用コマンド制御0出力	DCLで指定

4004(CPU)プログラマーズ・モデル



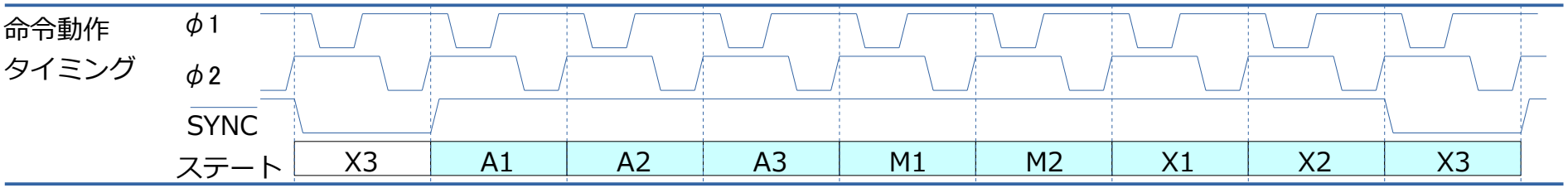


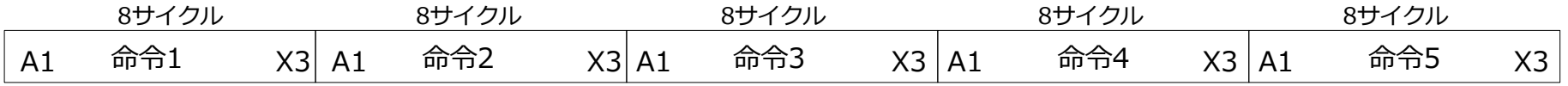
CPUに一時記憶用のレジスタが多い
→メモリ・アクセスを減らして性能向上
→現代のRISCアーキテクチャに近い
（命令長も1バイトか2バイトだし）



4004(CPU)の基本動作タイミング

●4004：1命令の実行単位は8サイクル（8ステート）



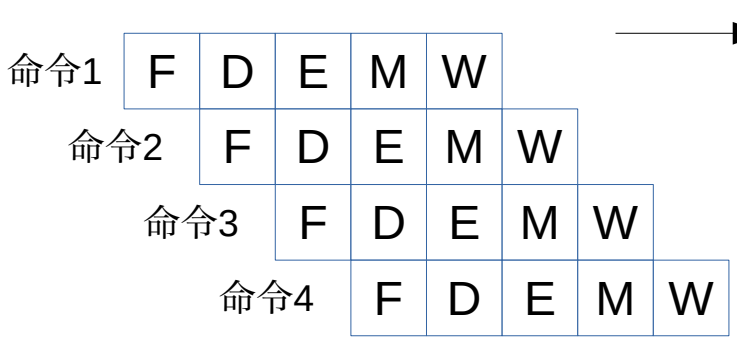


※4004の命令は、1バイト長または2バイト長。
※2バイト命令は8ステートを2回（16ステート）。
※1バイト命令でも16ステートかかるものがある。

→ 時間

●4004：命令数=46種類！

※今時のCPUだと：パイプライン動作による1命令1サイクル動作



F：命令フェッチ（IF：Instruction Fetch）
D：デコード（ID：Instruction Decode）
E：実行、アドレス計算（EX：Execution）
M：メモリ・アクセス（MA：Memory Access）
W：ライト・バック（WB：Write Back）

4004プログラム開発ツールADS4004

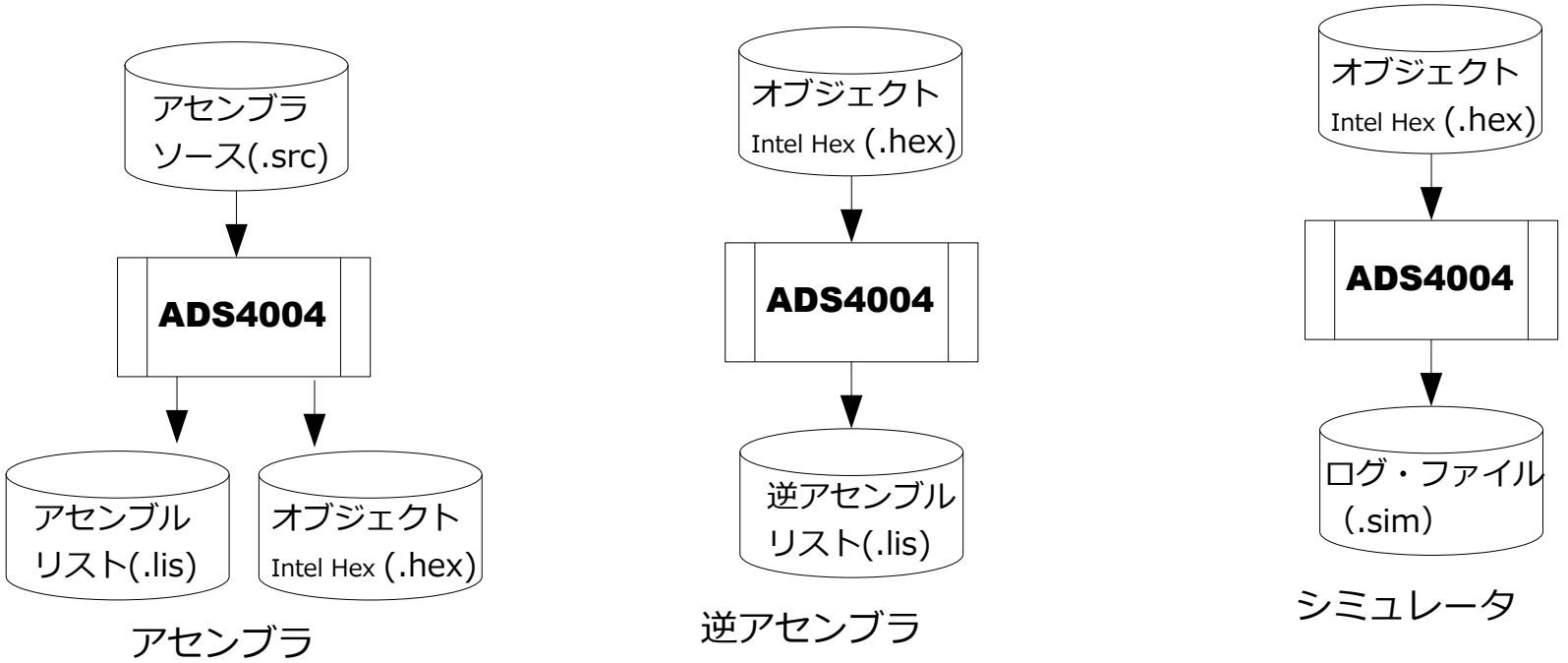
●現在は4004開発ツールは存在しない→自作するしかない！　「ADS4004」

・機能：2パス・アセンブラ、逆アセンブラ、CPUシミュレータ
・動作環境：標準Cで記述、任意OS環境で動作

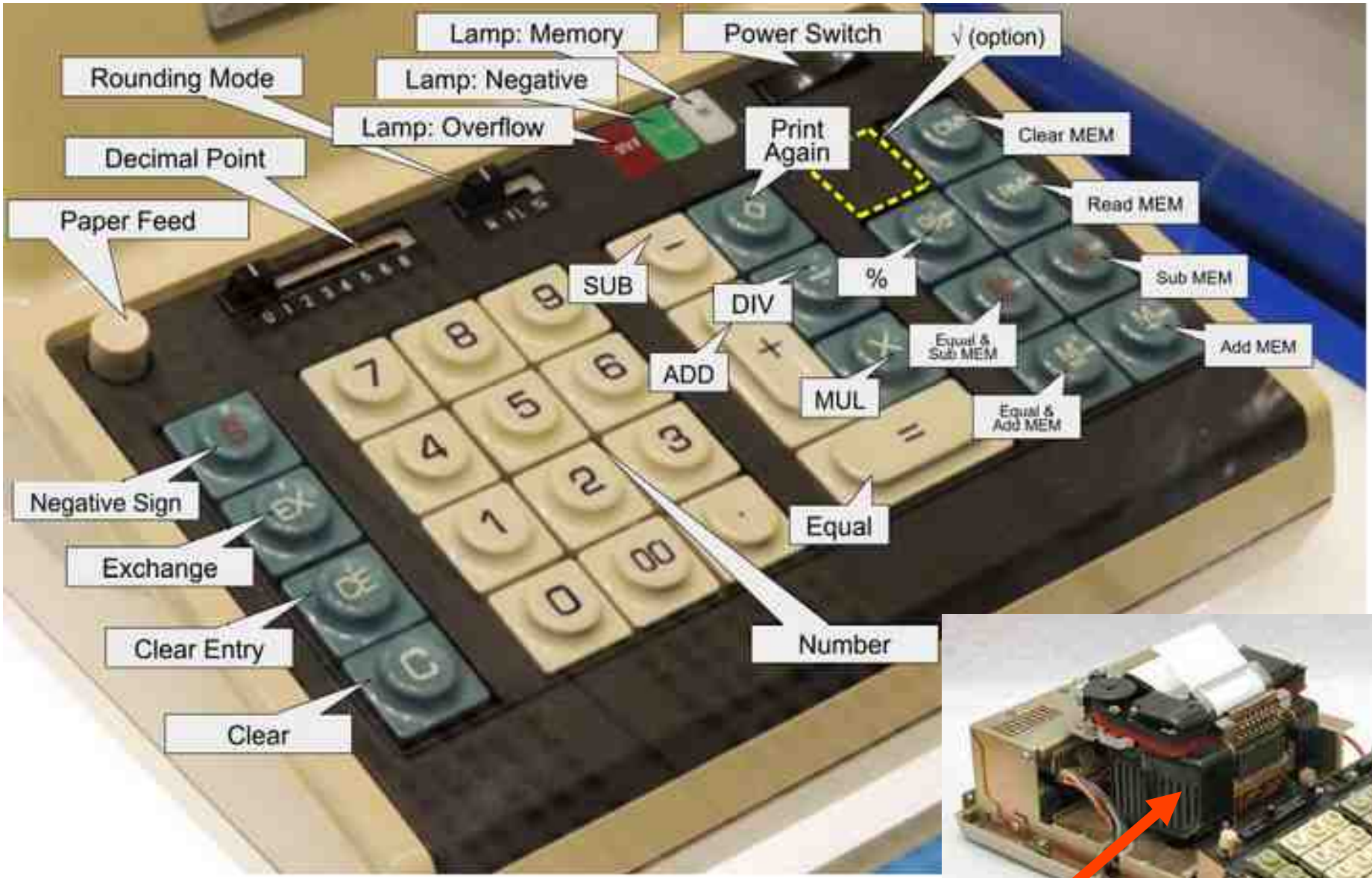
●プログラム本体「ads4004.c」

https://github.com/munetomo-maruyama/MCS4_SYSTEM
場所：MCS4_SYSTEM/SOFTWARE/ads4004/ads4004.c

●機能



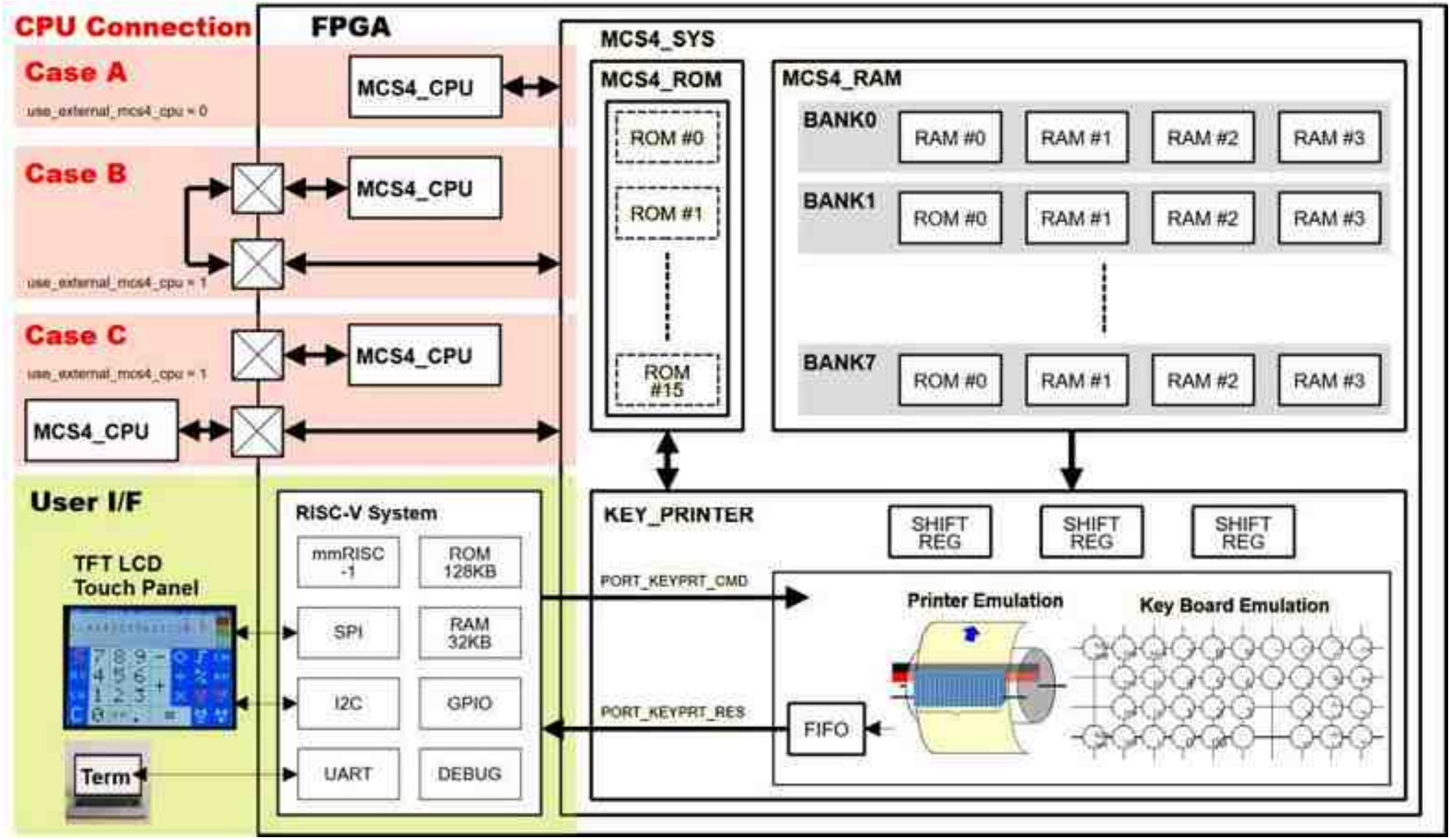
Busicom社電卓141-PF



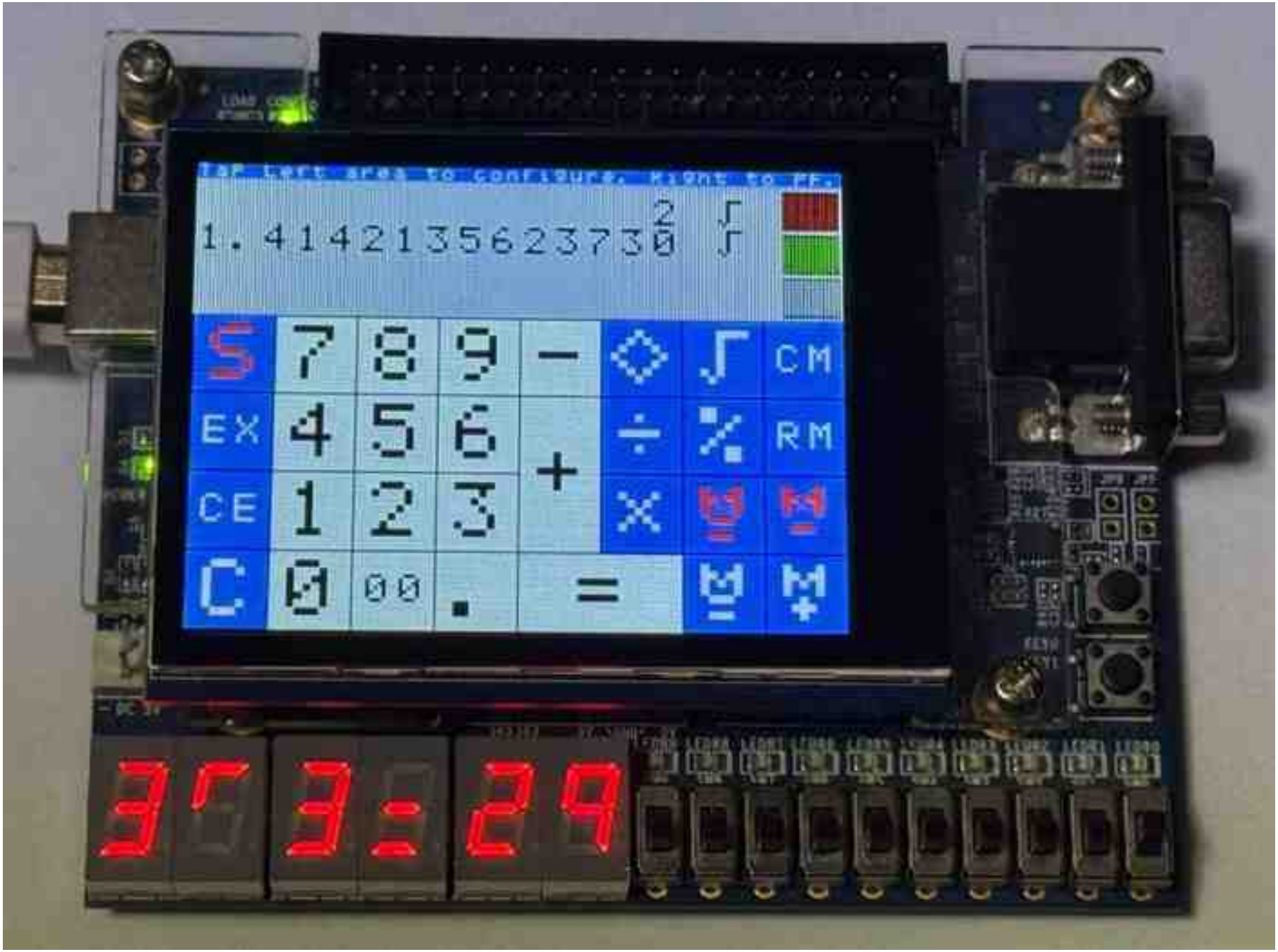
ドラム・プリンタ Model-102（信州精機）

設計したMCS-4 Systemのブロック図

- MCS-4システムのメモリ容量：最大構成（4001×16個、4002×32個）　※141-PFは4001×5個、4002×2個
- 電卓141-PFのキーボードとプリンタは、RISC-VとI/Fロジック（4003×3個）で再現(ソフトは完全コンパチ)
- 4004 CPUは、CaseA：内蔵、CaseB：内蔵するが外部端子で接続、CaseC：外部（Tiny Tapeoutチップ）



Busicom社電卓141-PFをFPGAで再現



4004(CPU)で円周率を500桁計算する

●4004で円周率を多桁計算
4004(CPU)で円周率を500桁計算するプログラムをアセンブラでコーディング

●実行結果
わずか17分！@750KHzで、500桁の計算結果が、141-PFのプリンタ（エミュレーション）に印字される。
LCDパネルだとどんどんスクロールして見えないので、シリアル・コンソールに出力も可能。

Machinの公式

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

```
===== pi4004
===== [CUI]
Loading...Done
     1415926535

     8979323846

     2643383279

     5028841971

     6939937510

     5820974944

     5923078164

     0628620899

     ..........
     0921861173

     8193261179
```

Tiny Tapeoutの登場

●個人でも設計・試作できるサービスTiny Tapeoutが登場！　https://tinytapeout.com/

・ICチップ開発の民主化。

・オープンソースの設計ツールと契約不要なPDK を使って自宅の PC でICチップを設計できる。

・対応する製造ファウンドリ：
　　SkyWater(米)130nm
　　IHP(独)130nm
　　GlobalFoundries(米)180nm

・複数のプロジェクトをタイル状にチップ内に並べるシャトル方式でウェハ試作

・試作費用を分担できて、一つのプロジェクトから見れば
　1 タイルあたり約$150～$300(2026 年時点の目安)

・完成すれば、パッケージに実装して評価ボードとともに届く。すぐに動かせる。

・設計できる回路：
　　RTL記述で設計した論理回路
　　ゲート・レベルのロジック設計
　　アナログ・チップの設計

Tiny Tapeoutで作れるチップ

mmRISC

● 1つのプロジェクトが占有できるタイル数
・1x1～8x2
・タイル1個の規模
　Skaywater 130nm プロセスの場合サイズが167um × 108um
　2000ゲートくらいに相当
・タイル数にかかわらず一つのプロジェクトが使える外部端子は下記のみ。

端子名	入出力	機能
rst_n	input	ユーザ・リセット
clk	input	ユーザ・クロック
ui[7:0]	input	ユーザ機能入力
uo[7:0]	output	ユーザ機能出力
uio[7:0]	in/out	ユーザ機能入出力
ena	input	プロジェクト選択イネーブル
sel_rst_n	input	プロジェクト選択 MUX リセット
sel_inc	input	プロジェクト選択 MUX インクリメント

ユーザ端子
　入力：8本
　出力：8本
　入出力：8本

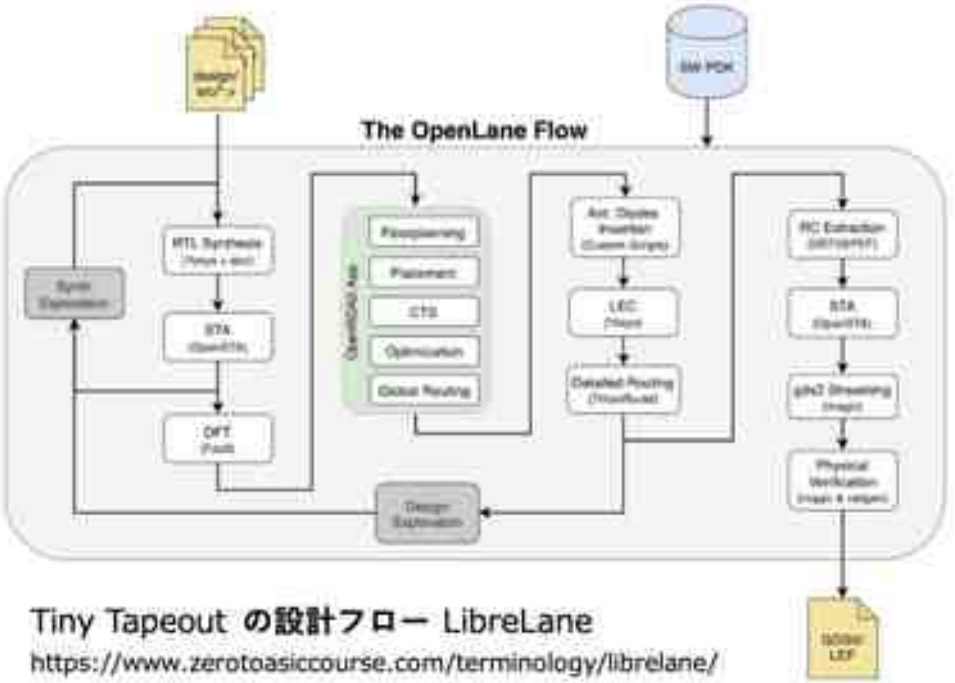
チップ内
プロジェクト
選択信号

※論理規模や端子数の制約はあるが、独自のロジックをシリコン化できるのは画期的
※さまざまな教育や人材育成、あるいは研究に活用することができる。

Tiny Tapeoutの設計フロー

●LibreLane：RTLからレイアウトまで

RTLベースのロジック・チップ設計には
LibreLaneを使用する。
以下の一連の工程を一気に実行する。
①RTL記述の論理合成
②仮負荷タイミング検証
③自動配置配線
④実負荷タイミング検証
⑤レイアウトGDSデータ生成
⑥DRC／LVS 検証



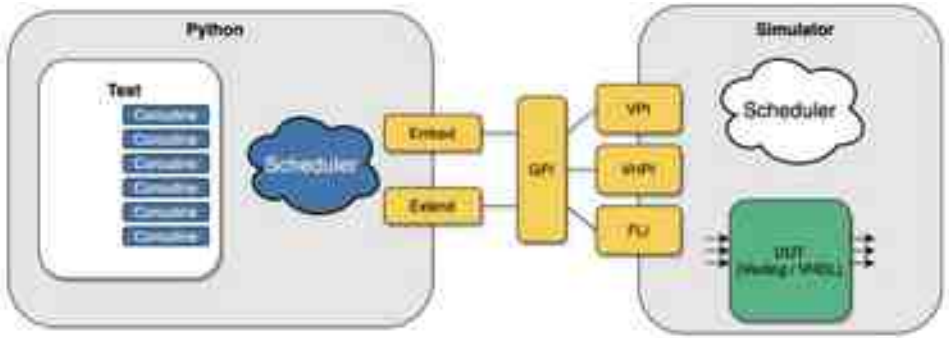
Tiny Tapeout の設計フロー LibreLane
https://www.zerotoasiccourse.com/terminology/librelane/

●論理検証のテストベンチはcocotb

・Tiny Tapeoutの論理検証環境では、
cocotb (coroutne cosimulation testbench)を使用

・Pythonにより高機能なテストベンチを記述可

・論理シミュレータはIcarus Verilogを使用し
PythonとはGPI(Generic Procedural Interface)でI/F
（商用の各種論理シミュレータにも対応）



cocotb のしくみ
https://docs.cocotb.org/en/stable/

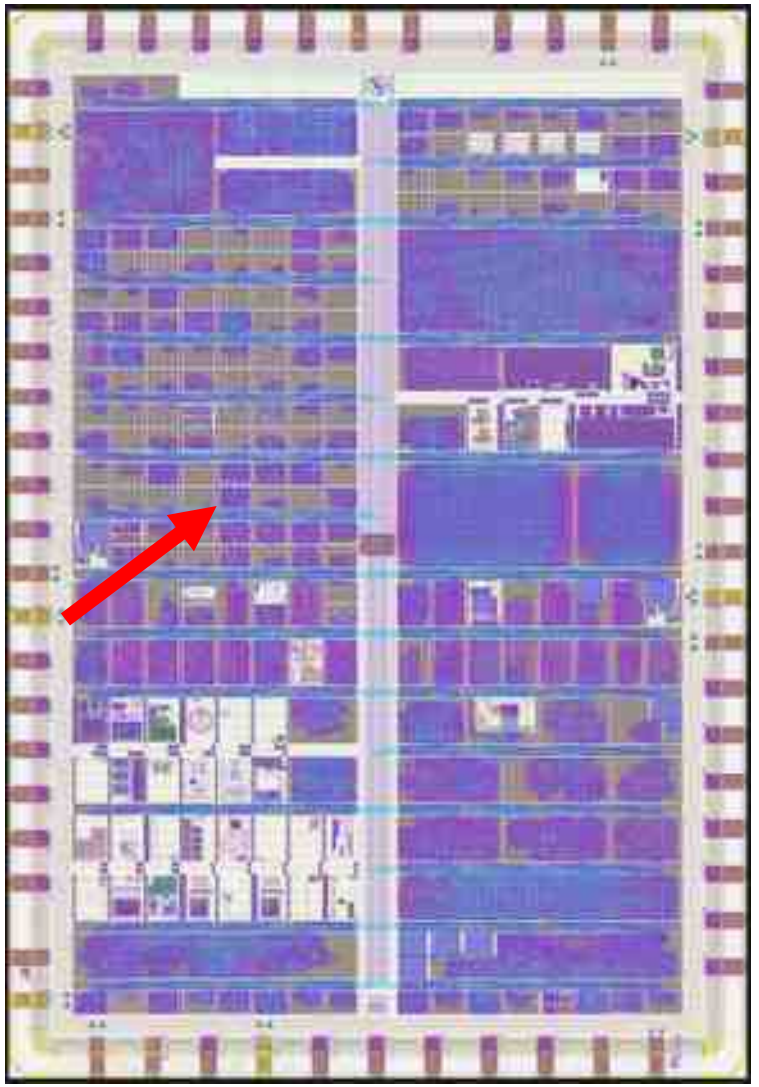
Tiny Tapeout TTSKy25a にテープアウト

mmRISC

●TTSKy25aに設計データを提出ずみ（TO締切2025年9月、ウェハ完成2026年5月、デリバリ2026年6月）

https://tinytapeout.com/chips/ttsky25a/tt_um_mcs4_cpu

https://tinytapeout.com/chips/ttsky25a/



シンプルなチューリング完全CPU bfCPUの設計とシリコン化

設計 **https://github.com/munetomo-maruyama/bfCPU.git**

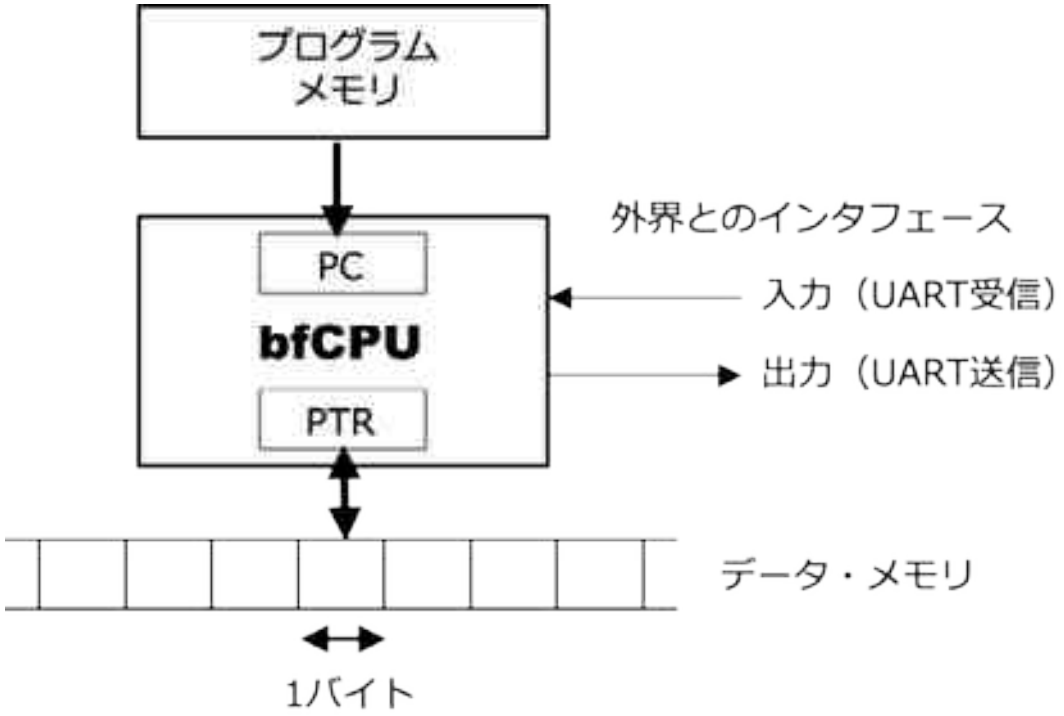
Tiny Tapeout **https://github.com/munetomo-maruyama/ttsky_bfCPU.git**

（I/O誌2026年4月号掲載、ISHI会同人誌Vol.3掲載予定）

チューリング完全で＋シンプルで＋文化のある＋楽しい　アーキテクチャ！

■ bfCPU

- 1993 年にスイスのUrban Müller が提唱したBrainf*ck(*=u) というアーキテクチャ
  ここではbfCPUと呼ぶ（少々品がない名称のため）
- シンプルなチューリング完全マシンとして有名
- わずか8 個の命令しか持たない8 ビットCPU
- これだけで任意のアルゴリズムを実現できる
- 外部とのインタフェースはUARTを使う
- **難解プログラミングとして有名　　でも、すごく面白い！**



bfCPUの深淵さ

●bfCPU のプログラムはその可読性・記述性がとても低く、難解プログラミング言語の一種に挙げられており、加算を実行するだけでもパズルのようだ。難解プログラミング言語とbfCPU について、Daniel Temkin は以下のように評している。

「 難解プログラミング言語(Esoteric programming languages：esolangs) は、プログラミングの世界から生まれた一種の表現芸術である。それは、言語の役割を単なる“命令と制御の道具” から、“文化的な表現や既存の枠組みへの拒絶” へと作り変える試みでもある。**なかでも群を抜いて悪名高いのがbfCPU の言語(命令) だ。**この言語は、コンピュータ本来のロジックをむき出しのままプログラマーに突きつける一方で、人間の言葉と機械語との距離を縮めることを徹底的に拒む。そうして私たちを滑稽なまでの論理の迷宮へと引きずり込むのだ。bfCPU は究極のミニマリズムを貫いており、命令として認識できるのはわずか8 つの記号のみしかない。マシンを動かすために“print” といった言葉を使うことは許されず、すべては記号の羅列で記述される。皮肉なことに、この“単純さ” こそが、扱う上での圧倒的な複雑さを生み出す。たとえば“32” という数字を直接表す記号すら存在しない。プログラマーはメモリの箱を一つずつ行ったり来たりしながら、値を保存する場所を探し、プラスとマイナスの記号を延々と書き連ねて、32 に到達するまでその場所の数値を増やしたり減らしたりしなければならない。こうなると、**32 という数字はもはや単なる“決まった定数” ではなく**、自らの知略で勝ち取るべき“リソース” へと変貌する。4 を8 回足すループを組んで32 を作るのか、あるいは、1 バイトの上限が256 であることを利用して、ゼロの壁を越えて逆算し目的の値へと着地させるのかなど、なんでもアリである。**32 という一つの数字をひねり出す行為そのものが、プログラマーにとっての“独自のスタイル” や“表現のこだわり” を披露する場**となる。」

Temkin Esolangs Daniel Temkin, Glitch && Human/Computer Interaction,2014, https://nooart.org/post/73353953758/temkin-glitchhumancomputerinteraction

bfCPUの命令体系

- オリジナルの命令は、**＞ ＜ ＋ － ． ， ［ ］**の８個のみ（命令コード＝１個のASCII文字）
- 今回の設計では、reset命令とnop命令を追加（命令コード＝4ビットのバイナリ）
- 命令コードは、オペコードのみ、オペランドなし

命令コード	記号	ニーモニック	等価C言語	意　味
0000	>	pinc / p++	ptr++	データ・ポインタPTRを１つインクリメントする（右隣のデータ・メモリを指す）
0001	<	pdec / p--	ptr--	データ・ポインタPTRを１つデクリメントする（左隣のデータ・メモリを指す）
0010	+	inc	(*ptr)++	データ・ポインタPTRが指すデータ・メモリ内のバイト・データを１つインクリメントする（１を加算）
0011	-	dec	(*ptr)--	データ・ポインタPTRが指すデータ・メモリ内のバイト・データを１つデクリメントする（１を減算）
0100	.	out	putchar(*ptr)	データ・ポインタPTRが指すデータ・メモリ内のバイト・データを出力する（UARTから送信する）
0101	,	in	*ptr=getchar()	バイト・データを入力して、データ・ポインタPTRが指すデータ・メモリにライトする（UARTが受信するまで待つ）
0110	[	begin	while(*ptr) {	データ・ポインタPTRが指すデータ・メモリ内のバイト・データがゼロなら、次の命令に進まずに、このbegin命令（[）の先にあるネスト深さが対応するend命令（]）の次の命令までジャンプする。非ゼロなら、このbegin命令（[）の次の命令に進む。
0111	]	end	}	データ・ポインタPTRが指すデータ・メモリ内のバイト・データが非ゼロなら、次の命令に進まずに、このend命令（]）の前にあるネスト深さが対応するbegin命令（[）の次の命令までジャンプする。ゼロなら、このend命令（]）の次の命令に進む。
1000	なし	reset	なし	PC=0、PTR=0に初期化し、データ・メモリ全体をゼロ・クリアして、プログラムの先頭から実行する
1001	なし	なし	なし	リザーブ
1010	なし	なし	なし	リザーブ
1011	なし	なし	なし	リザーブ
1100	なし	なし	なし	リザーブ
1101	なし	なし	なし	リザーブ
1110	なし	なし	なし	リザーブ
1111	なし	nop	なし	何も実行しない

bfCPUのプログラム例 (1)

bfCPUの難解プログラム集：https://brainfuck.org
この文化を楽しめるのがbfCPU

●PTRが指すデータ・メモリの中身をゼロクリアする
　　[+]　または　[-]

●加算プログラム（UARTが2つの数値を受信して加算結果を送信する）

```
,>,<[->+<]>.
```

●乗算プログラム（UARTが2つの数値を受信して乗算結果を送信する）

```
,>,<[->[->+>+<<]>>[-<<+>>]<<<]>>.
```

●Hello World! (UARTから文字列”Hello World!”+0x0aを出力)

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]
>>.>---.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

●tic-tac-toeゲーム

```
//[tictactoe.b -- play tic-tac-toe
//(c) 2020 Daniel B. Cristofani
//http://brainfuck.org/
//This program is licensed under a Creative Commons Attribution-ShareAlike 4.0
//International License (http://creativecommons.org/licenses/by-sa/4.0/).]

--->--->>>>->->->>>>>-->>>>>>>>>>>>>>>>>>>>+>>++++++++++[
  <<++[
    --<+<<+<<+>>>>[
      >[<->>+++>>[-]+++<<<+[<++>>+<--]]+>+++++[>>++++++++++<<-]
      >>++++.[-]>>+[<<<<+>>+>>-]<<<<<<[>+<-]<<
    ]++++++++++.[-]>++
  ]-->>[-->[-]>]<<[
    >>--[
      -[
        -[
          -----[>+>+++++++<<+]-->>-.----->,[<->-]<[[<]+[->>]<-]<[<<,[-]]>>>>
        ]>
      ]<[
        >-[+<+++]+<+++[+[---->]+<<<<<<[>>]<[-]]
        >[<+[---->]++[<]<[>]>[[>]+>++++++++++<<-[<]]]>[>>>>]
      ]<[
        -[[>+>+<<-]>[<+>-]++>+>>]<[<<++[-->>[-]]>[[-]>[<<+>>-]>]]
      ]<[
        [[<<]-[>>]<+<-]>[-<+]<<[<<]-<[>[+>>]>[>]>[-]]
        >[[+>>]<-->>[>]+>>>]
      ]<[
        -[
          --[+<<<<--[+>[-]>[<<+>+>-]<<[>>+<<-]]++[>]]
          <<[>+>+<<-]>--[<+>-]++>>>
        ]<[<<<[-]+++>[-]>[<+>>>+<<-]+>>>]
      ]<[
        +[[<]<<[<<]-<->>+>[>>]>[>]<-]+[-<+]<++[[>+<-]++<[<<->>+]<++]<
        <<<<<<      +> > >+> > >+[
        <<<                 ->+>+>+[
        <<<<<<<   +>->+> > >->->+[
        <<<<<          ->+>+> >+>+[
        <<<<             ->->+>->+[
        <<<<<<<<<+>-> >+> > >->+>+[
        <<<<<          -> >+> >->+[
        <<<<             +>->+> >+]]]]]]]
        +++[[>+<-]<+++]--->>[-[<->-]<++>>]++[[<->-]>>]>[>]
      ]<
    ]
  ]<
]
reset // restart

//[This program plays tic-tac-toe. I've given it the first move. It needs
//interactive i/o, e.g. a command-line brainfuck interpreter or a brainfuck
//compiler that produces command-line executables. At the '>' prompt, enter
//the number of an empty space, followed by a linefeed, to play a move there.]
//}
```

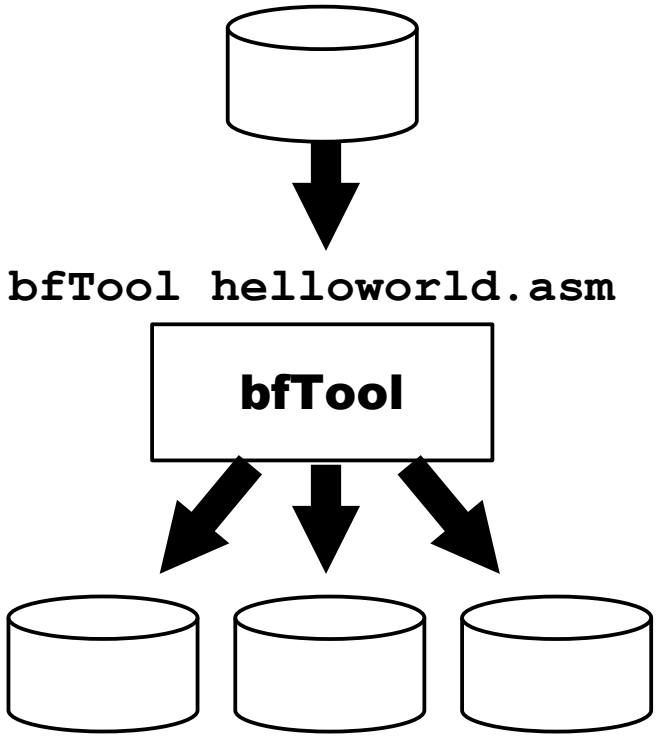
UART
ターミナル

```
X23
456
789
>5
X23
4O6
78X
>4
X23
OOX
78X
>2
XO3
OOX
7XX
>7
XOX
OOX
OXX
```

https://github.com/munetomo-maruyama/bfCPU/tree/main/bfTool

■アセンブル（コンパイル）

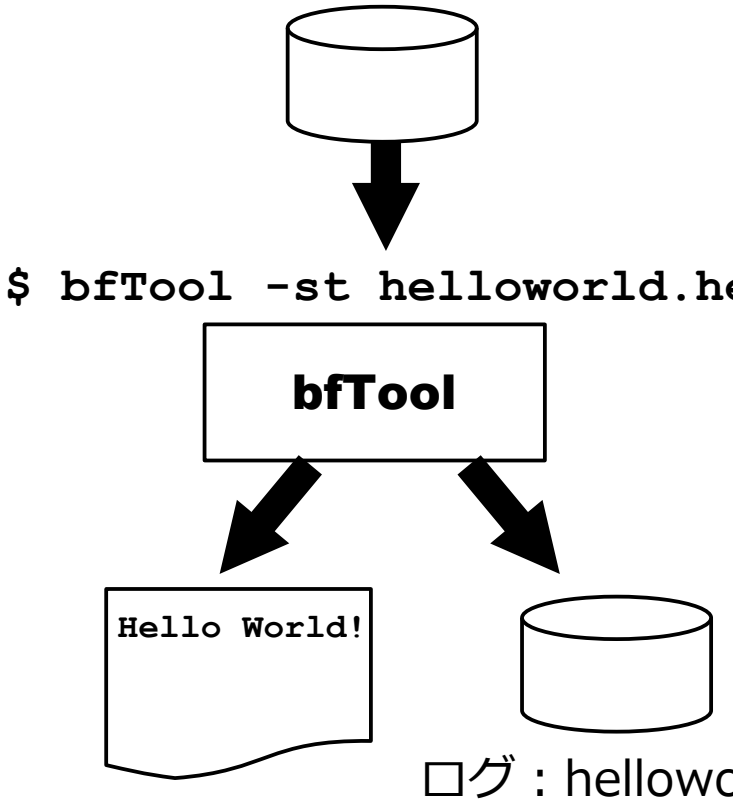
ソース：helloworld.asm



バイナリ：helloworld.hex
リスト　：helloworld.lis
Verilog　：helloworld.v
(論理SIMのメモリ初期化用)

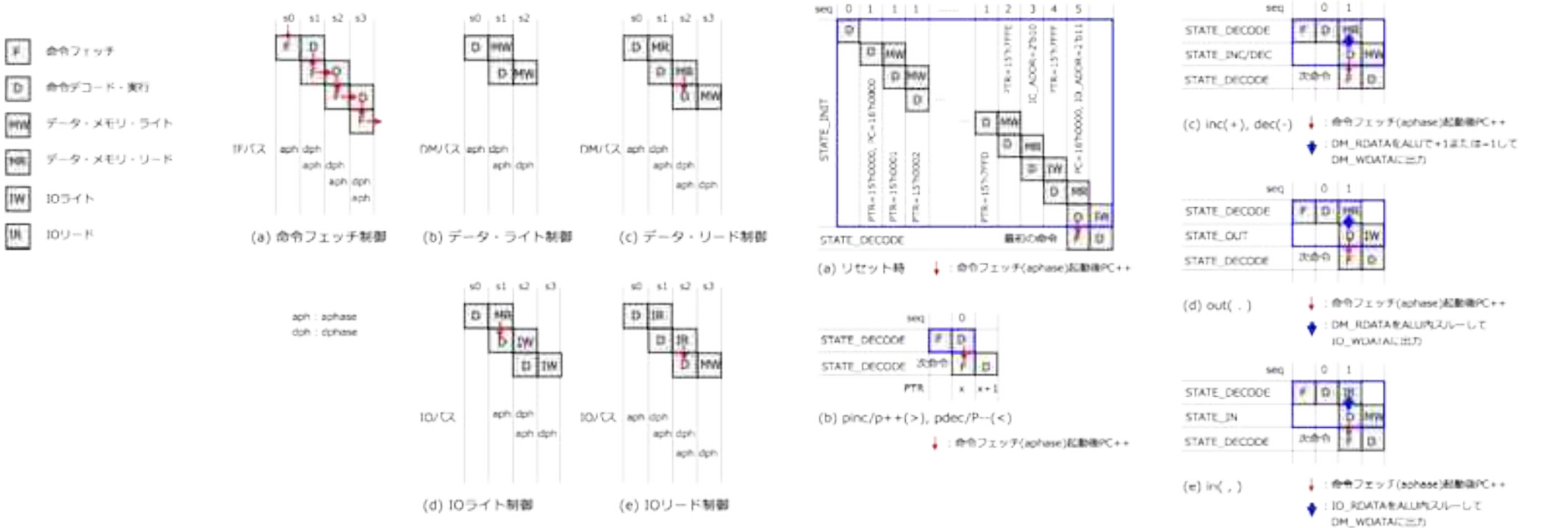
■シミュレーション

バイナリ：helloworld.hex



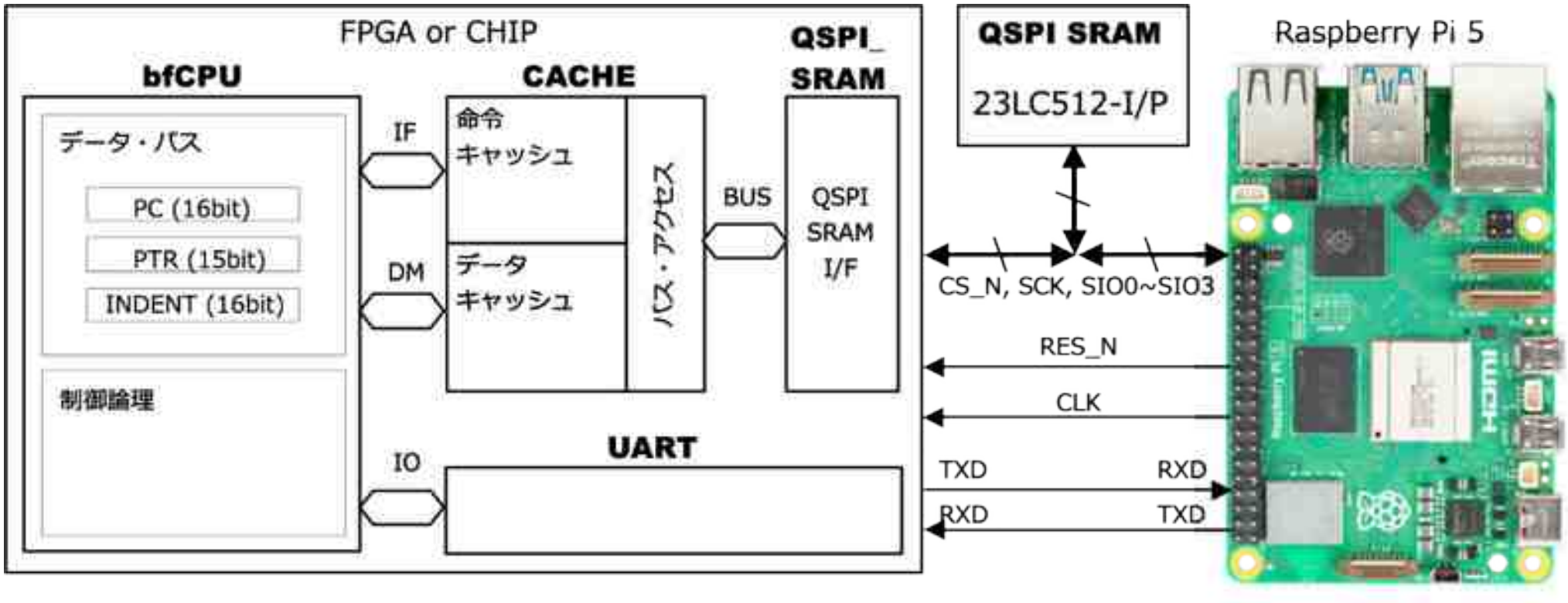
コンソール
ログ：helloworld.sim

bfCPUのパイプライン制御設計



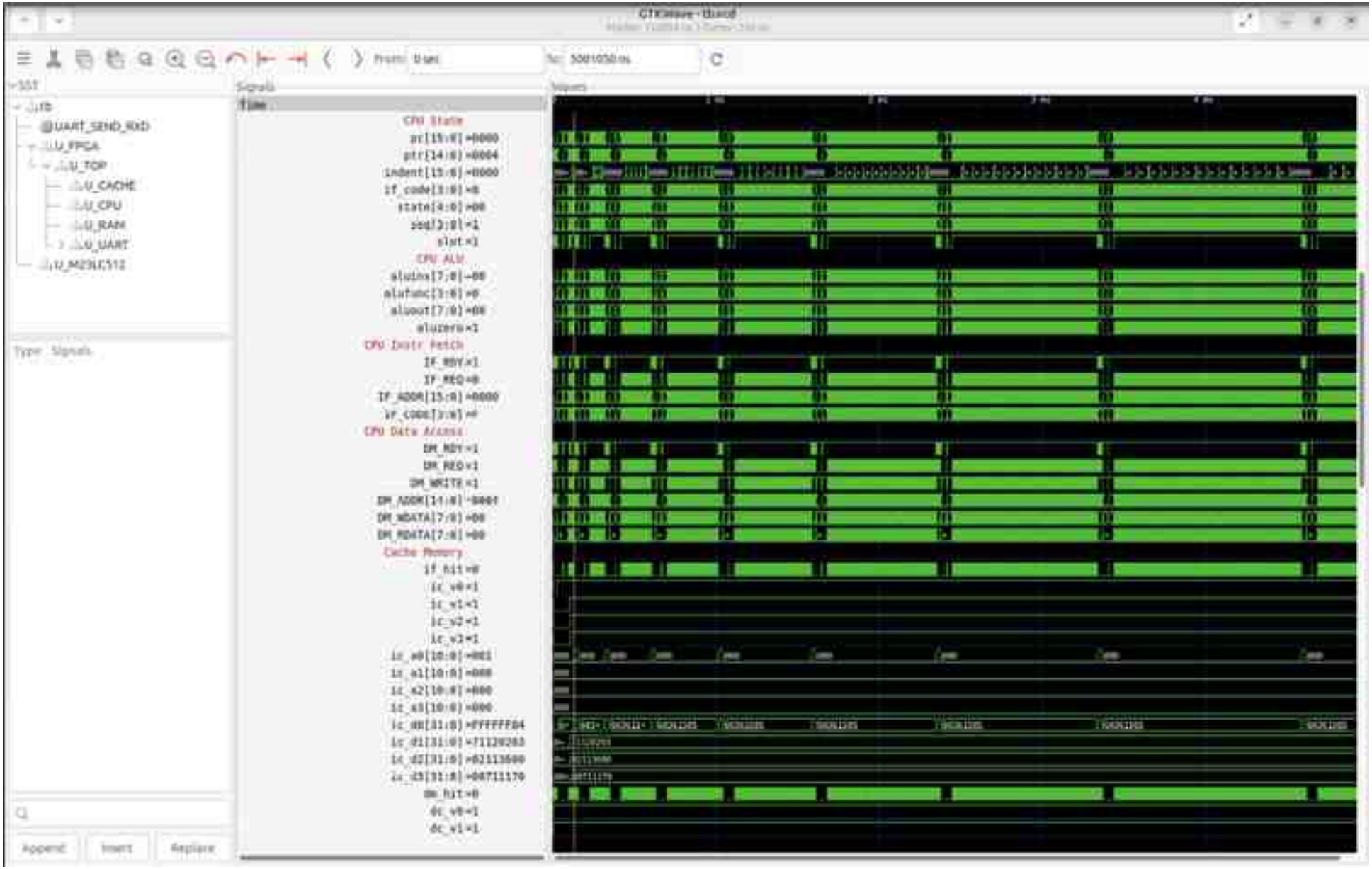
bfCPUシステムのブロック図

- bfCPU：2段～4段のパイプライン動作
- 命令キャッシュ：
  ダイレクト・マップ、4bit ×8word ×4 エントリ、ライン長: 4bit ×8word
- データ・キャッシュ：
  ダイレクト・マップ、ライト・バック、 8bit ×4word ×2 エントリ、ライン長: 8bit ×4word
- QSPI_SRAM：シーケンシャル・モード（Microchip 23LC512-I/P）
  QSPI_SRAMへのプログラム・ロードとbfCPUのリセット（実行開始司令）はRaspberry Pi 5が担う
  Raspberry Pi 5側の自作アプリ：**bfRun**



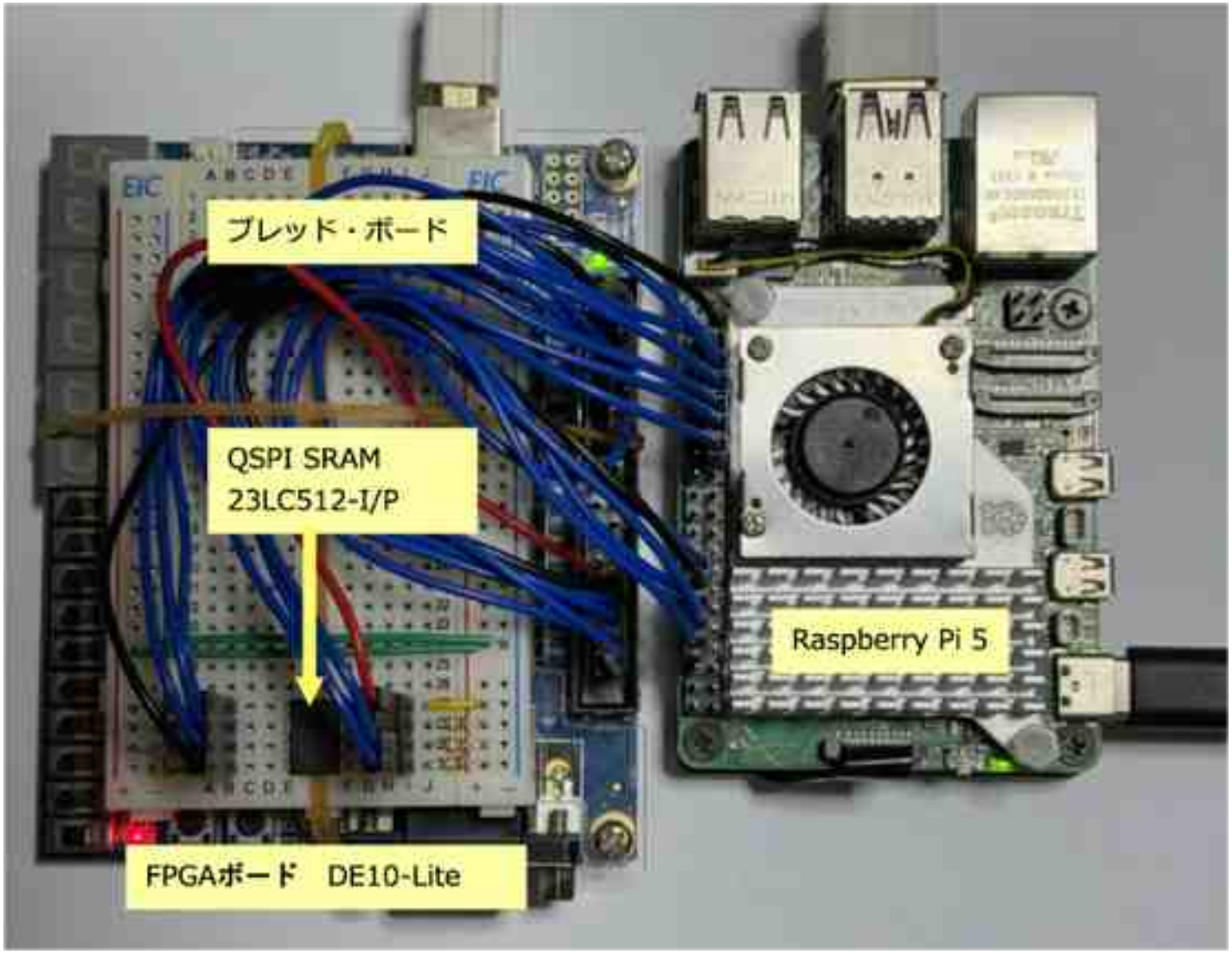
bfCPUシステムの論理検証

- 乗算プログラムでの検証結果：乗数・被乗数が大きくなるにつれ実行時間が長くなっていく。
- 乗算プログラム本体を実行中は命令バスやデータ・バスにウェイト入らず、キャッシュの効果が見える。



bfCPUシステムのFPGA実装とシステム検証

- FPGAボート(Terasic社DE10-Lite)＋QSPI_SRAM(23LC512-I/F)＋Raspberry Pi 5を結線
- ラズパイは、①システム・クロック生成、②リセット生成、③QSPI_SRAMへの書き込み、を担う
- ラズパイがbfCPUをリセット中にQSPI_SRAMにプログラムをライトし、リセット解除して実行開始
- ラズパイ上に上記操作のためのアプリ「**bfRun**」を用意

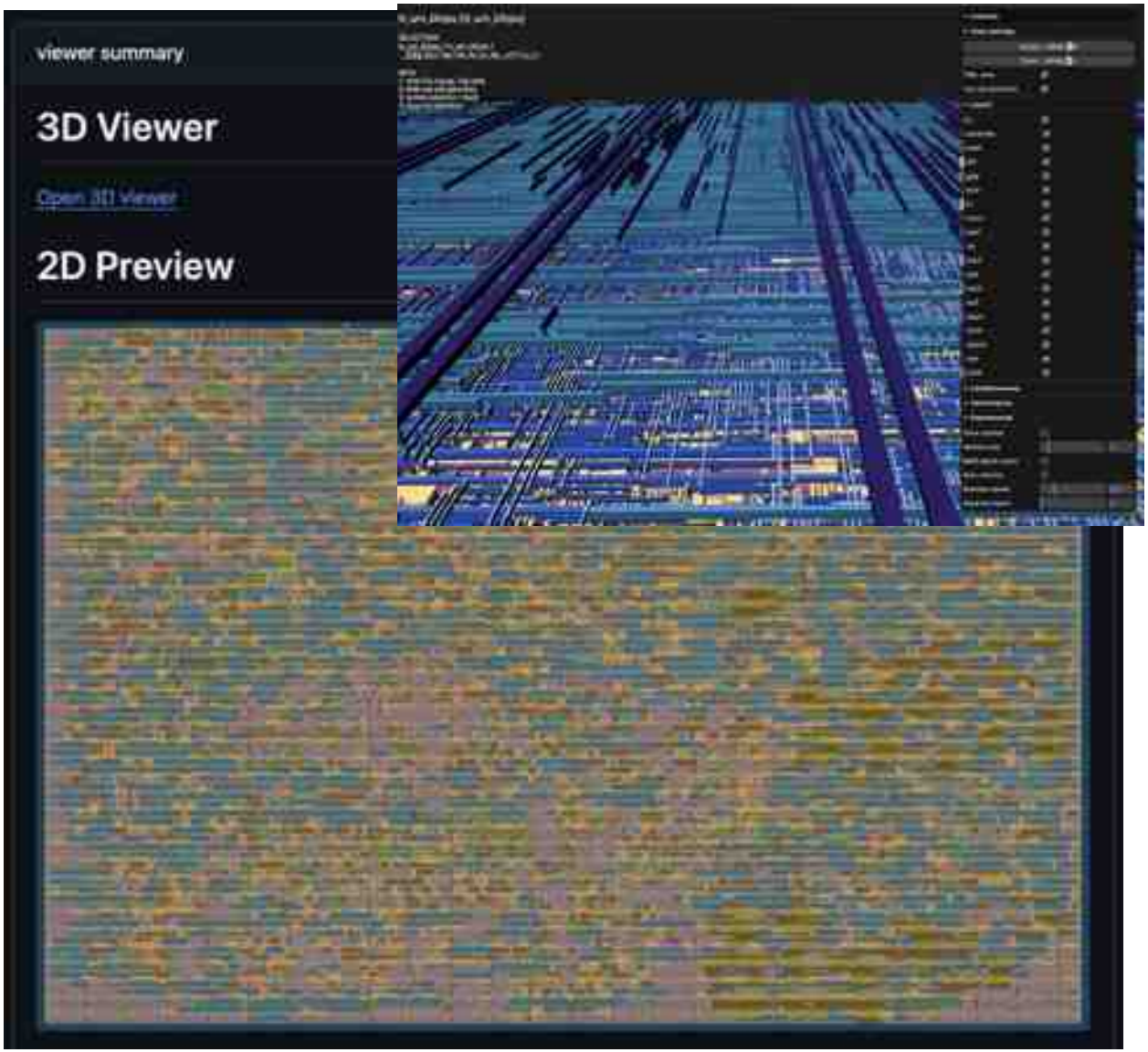


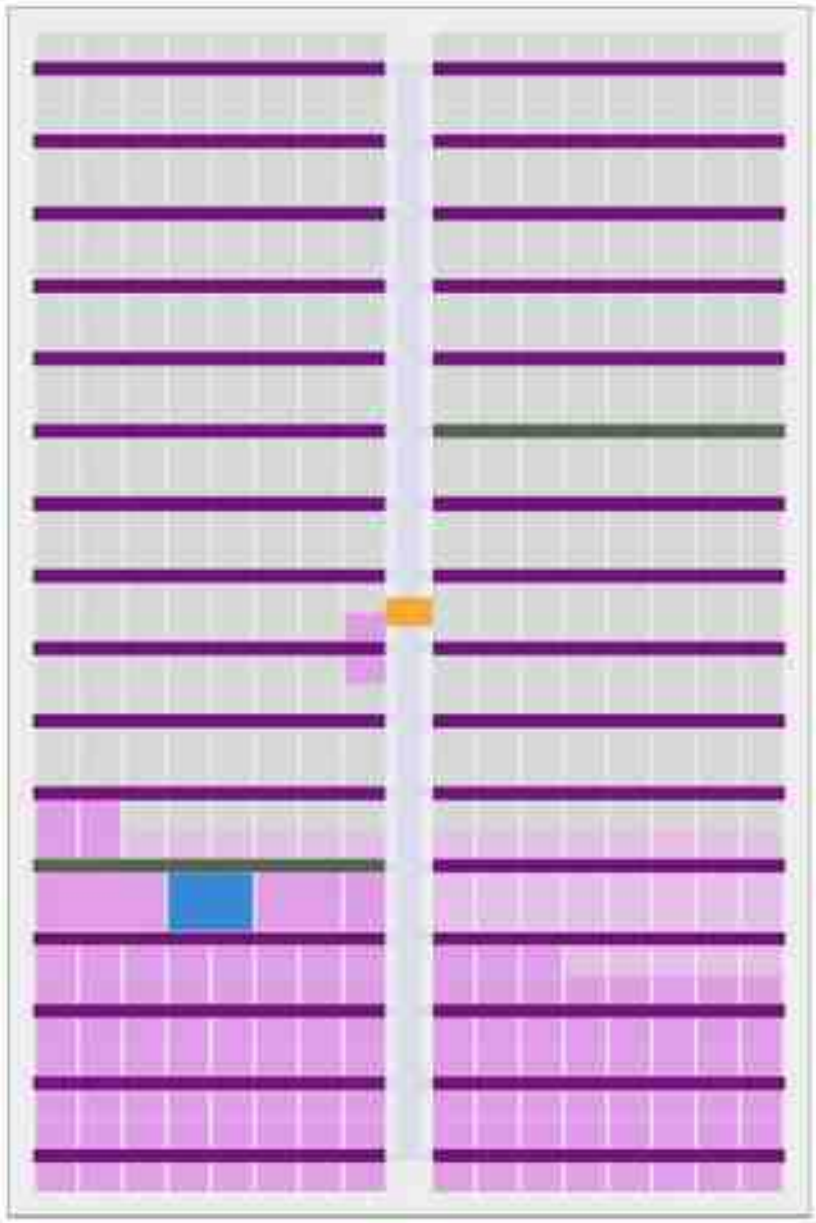
Terasic社DE10-Lite (Max 10)

Tiny Tapeout TTSKy26a にテープアウト

mmRISC

- TTSKy26aに設計データを提出ずみ（TO締切2026年5月、ウェハ完成2026年10月、デリバリ2026年12月）
- bfCPUのタイル数は2x2で実装（キャッシュ・レスにすると2x1）
- 配置密度は要調整　●アンテナエラー有りでも受付けてくれる





実用RISC-V CPUの自作と
22nm 商用シリコンへの実装

https://github.com/munetomo-maruyama/mmRISC-1.git

自作RISC-V 実用CPUコア「mmRISC-1」の仕様

項　目	内　容
CPUコア名称	mmRISC-1 (Much More RISC)
命令セット (ISA)	RV32IM[A][F]C　　[ ]は選択可能
Hart数（CPUコア数）	1 hart ～ 1,000,000 harts
特権モード	Machine Modeのみ
パイプライン段数	整数命令：3～5段、浮動小数点命令：5～6段
32ビット整数乗算性能	1-cycle
32ビット浮動小数点演算性能	1-cycle (FADD.S/FMUL.S/FMADD.S)
デバッグ・サポート機能 （実用CPUコアのために必須）	RISC-V規格の「External Debug Support Ver.0.13.2」に準拠 4-wire JTAGインタフェース／2-wire cJTAG インタフェース ハードウェア・トリガ x 4
割り込み	RISC-V標準：External + Machine Software + Machine Timer 独自拡張：User IRQ：64入力 全ての割り込み要因に独立ベクタをアサイン IRQ割り込み要求毎に16段階の優先レベル指定可、ネスティング可
バス・インタフェース	AHB-Lite（命令／データ／LR-SCモニタ／デバッグ・アクセス） AXI-4（オプション：対応容易性確保）
RTL	Verilog-2001, System Verilog
検証	論理シミュレーション(QuestaSim)　システム検証（FPGA + OpenOCD + Eclipse）
Github	https://github.com/munetomo-maruyama/mmRISC-1

各命令のパイプライン制御

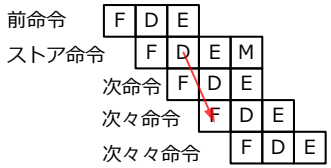
●ALU系（Reg-Reg/Reg-Imm）命令

LUI/AUIPC
ADDI/SLTI/SLTIU/XORI/ORI/ANDI
SLLI/SRLI/SRAI
ADD/SUB/SLL/SLT/SLTU/XOR/SRL/SRA/OR/AND
CSRRW/CSRRS/CSRRC
CSRRWI/CSRRSI/CSRRCI

・ALU系命令は、Eステージで、ソース・レジスタを読み出し、
演算を行い、その結果をディスティネーション・レジスタに書き込む。
前後の命令でディスティネーション・レジスタとソース・レジスタが
同一でも、干渉し合わず実行できる。
・Dステージで2命令先の命令フェッチを起動する。　　ALU：Arithmetic Logic Unit

●メモリ・ストア命令

SB/SH/SW

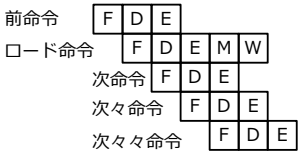


・ストア命令は、Eステージでメモリのアクセス先アドレスを計算して、
ライト・データを準備し、次のMステージでメモリにライトする。

●メモリ・ロード命令

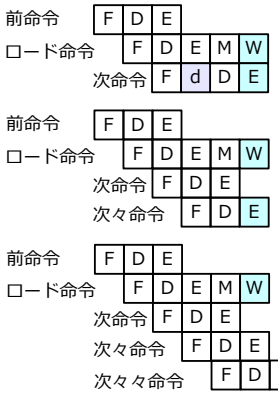
LB/LH/LW/LHB/LWU

(1) ロード命令のディスティネーション・レジスタを後続の命令が参照しない場合



・ロード命令は、Eステージでメモリのアクセス先アドレスを計算して、
Mステージでメモリをリードして、リード・データをWステージで
デスティネーション・レジスタに格納する。

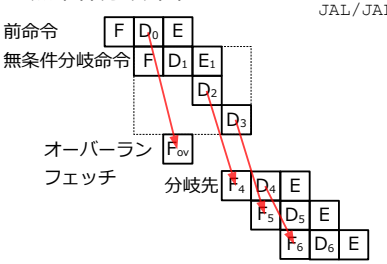
(2) ストール制御：ロード命令のディスティネーション・レジスタを後続の命令が参照する場合



・ロード命令のデスティネーション・レジスタを後続命令が参照する場合は、
ロード命令のWステージと、当該後続命令が参照するステージが重なるまで
その後続命令のDステージをストールさせる。
・Wステージではまだデスティネーション・レジスタへ書き戻す直前であり
後続命令でそのデータを参照するステージはレジスタ本体を参照せず、
Wステージで書き戻す直前の値を参照する（レジスタ・フォワーディング）。

●無条件分岐命令

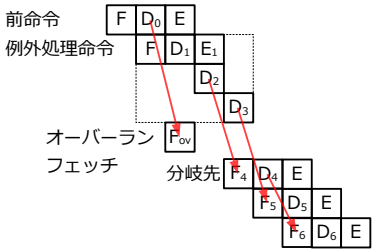
JAL/JALR



・無条件分岐命令直前の命令のD_0がその2命令先のフェッチF_{ov}を
起動しているが、それはオーバーラン・フェッチとなり無視。
・無条件分岐命令のIDステージは3ステップ構成。
・無条件分岐命令の最初のD_1は命令フェッチを起動しない。
ただしD_1後のE_1で分岐先アドレスを計算。
E_1で計算した分岐先アドレスからD_2が分岐先命令のフェッチF_4を起動。
D_3がその次の命令のフェッチF_5を起動する。
・分岐先の命令からは通常通り動作。

●例外処理関連命令

EBREAK/ECALL/MRET



・基本的に無条件分岐命令と同様。

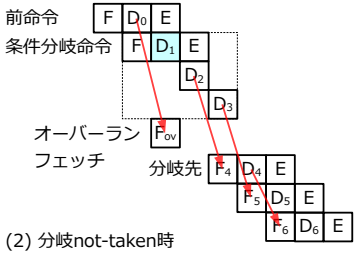
・EBREAK・ECALL命令
E_1：CSRのMSTATUS/MEPC/MCAUSE/MTVALを更新
分岐先：CSRのMTVECとMCAUSEにより決定

・MRET命令
E_1：CSRのMSTATUSを更新
分岐先：CSRのMEPCが示すアドレス

●条件分岐命令

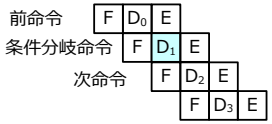
BEQ/BNE/BLT/BGE/BLTU/BGEU

(1) 分岐taken時



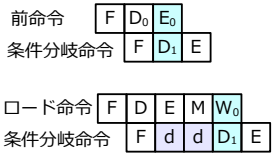
・条件分岐命令は、前命令までの実行結果が反映された汎用レジスタの
大小判定で分岐するかどうかをD_1で判断する。
・条件分岐命令のD_1で分岐する判断をしたら、その後の動作は
無条件分岐命令と同様である。分岐taken時の条件分岐命令は
3ステップで実行される。

(2) 分岐not-taken時



・条件分岐命令のD_1で分岐条件不成立の場合は、通常命令と同様に
D_1で2命令先の命令フェッチを起動し、1ステップで終了する。

(3) ストール制御：前の命令のデスティネーション・レジスタを条件分岐命令が参照する場合

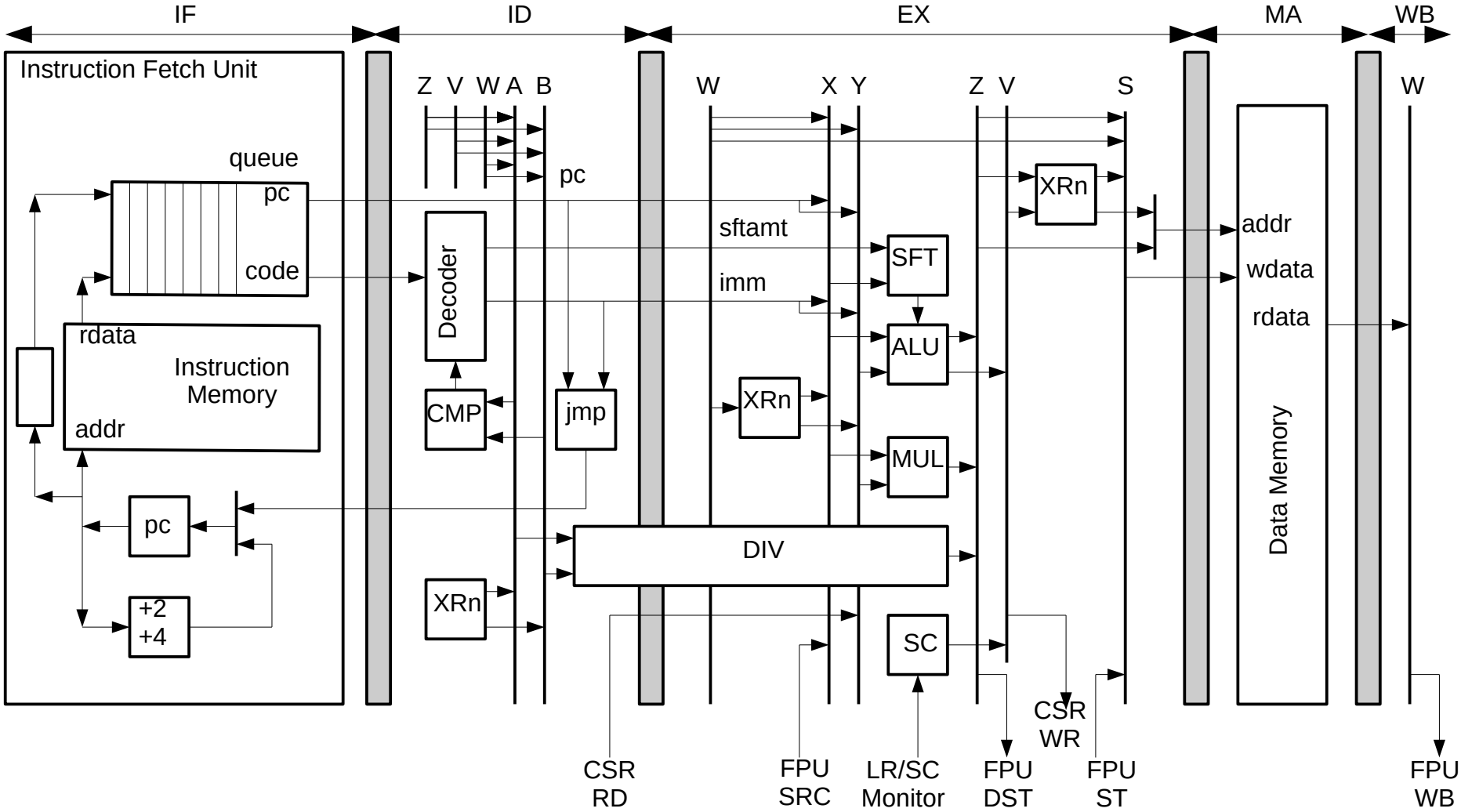


・直前命令のE_0のデスティネーション・レジスタを条件分岐命令のD_1が参照する場合
は、D_1時点ではデスティネーションレジスタに書き込まれていないので、書き込まれ
る前のALU出力を参照する（レジスタ・フォワーディング）。

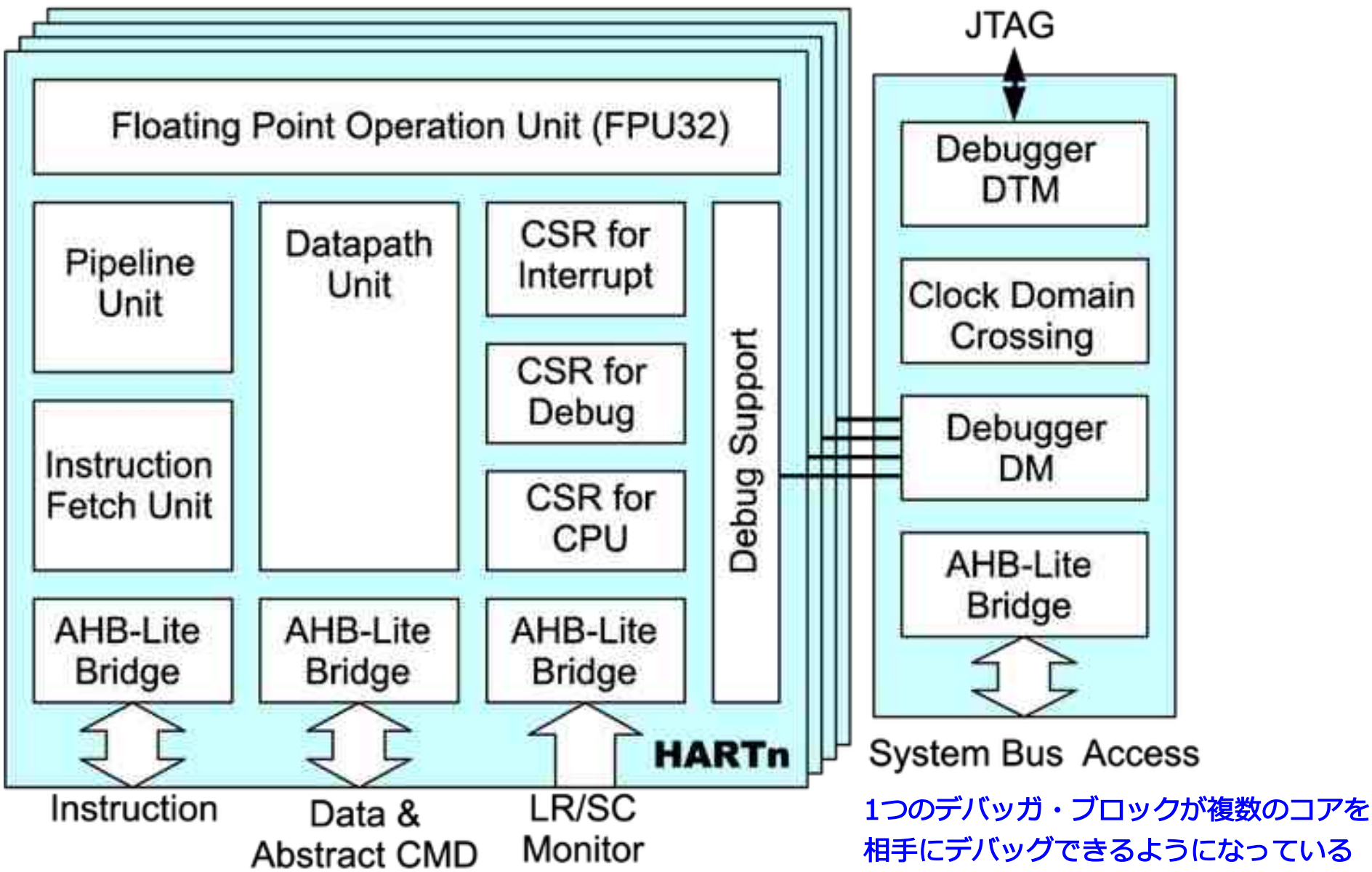
・ロード命令のデスティネーション・レジスタを条件分岐命令のD_1が参照する場合
は、D_1をストールしてW_0ステージにおけるレジスタに書き戻す直前の値を参照する。

整数演算データ・パスの構造

●各命令のデータの流れから、パイプライン動作とデータ・パスを描く



CPU部のブロック図

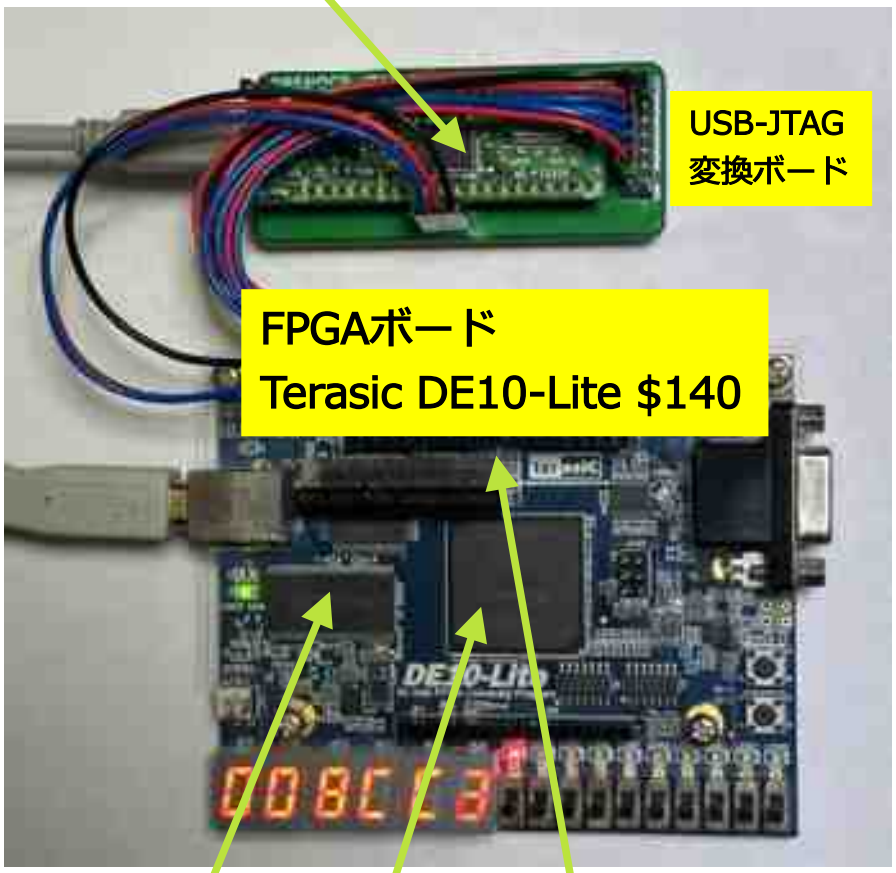


1つのデバッガ・ブロックが複数のコアを相手にデバッグできるようになっている

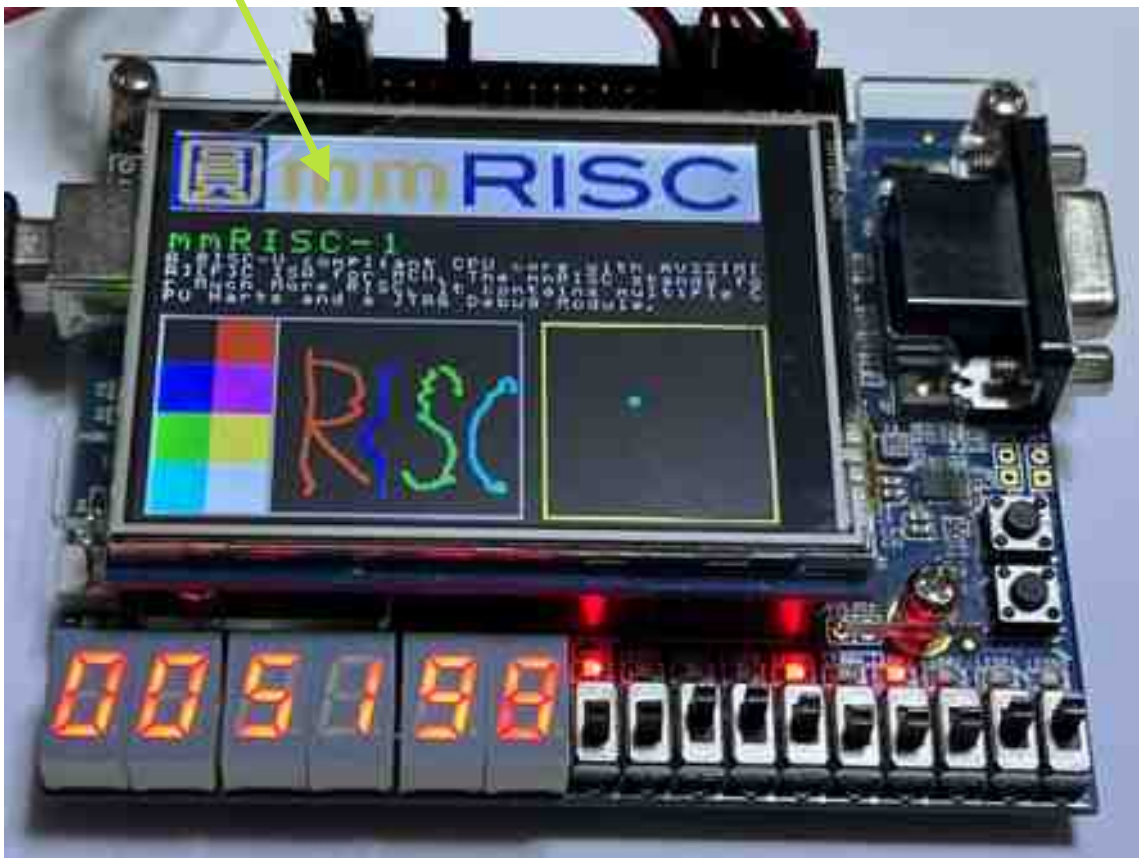
システム検証のためのFPGAボード

mmRISC

FTDI FT2232D
秋月電子 USBシリアル2ch変換モジュール AE-FT2232



Adafruit 2.8" TFT Touch Shield for Arduino with
Capacitive Touch Screen (Product ID 1947)
LCD Controller=ILI9341 (SPI I/F)
Capacitive Touch Controller=FT6206 (I2C I/F)



LCDタッチパネルを
載せるとカッコいい！

mmRISC-1の標準デバッガへの対応

mmRISC

- Eclipse + OpenOCD

4-wire JTAG

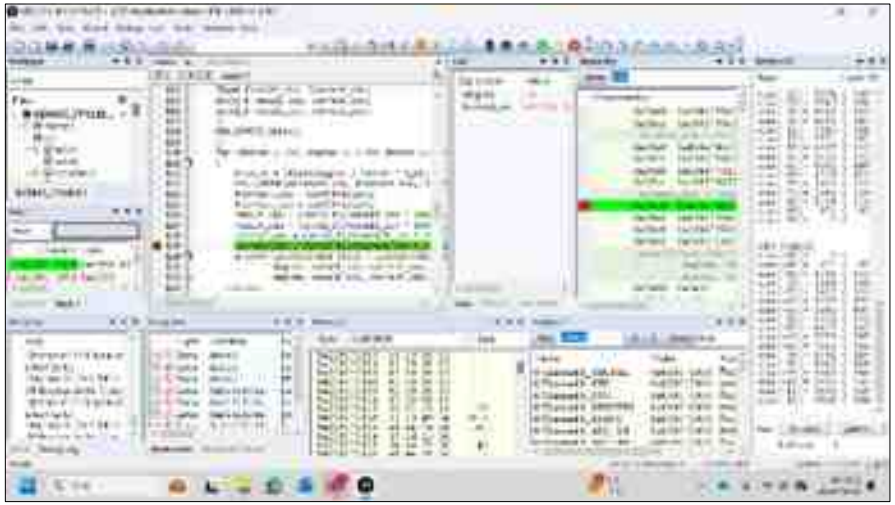
2-wire cJTAG

USB-JTAG/cJTAG I/F

- 某 商用 RISC-V統合化開発環境（非公式・動作確認済）

4-wire JTAG

2-wire cJTAG



USB-JTAG/cJTAG I/F

mmRISC-1をシリコン化

● mmRISC-1を**某社MCU**向けにご採用いただきシリコン化

ターゲット・プロセス：22nm mixed signal CMOS（プレーナ・プロセスの最終世代）

リアルな半導体設計は、手間と工数が膨大

RTL記述	DFTのSCANテスト・モード時は、JTAG/cJTAG関連論理のF/Fの JTAG/cJTAGクロックをメイン・クロックに切り替える対応が必要
機能検証	CHIP全体でクローズできているかの視点で（メモリ、デバッグ、モジュール間信号）
システム検証	CHIP全体を大規模FPGAに実装して、ターゲット・アプリケーションを載せて検証
論理合成	DFT(SCAN F/F)挿入、タイミング制約SDC（FPGAからはだいぶ変わる）
等価検証	論理合成前後で、RTLとゲート・レベル・ネットの間の等価検証
テスト設計	SCANテスト・パターンの自動生成と検出率確認
機能検証	ゲート・レベルでの機能検証（Verilogの曖昧さの保険）
タイミング検証	仮負荷ベースでタイミング制約SDCによるスタティック・タイミング検証
レイアウト	クロック分配とタイミング収束させながらの物理レイアウト実装
タイミング検証	実負荷ベースでタイミング制約SDCによるスタティック・タイミング検証
等価検証	レイアウト前後で、ゲート・レベル・ネットの間の等価検証
機能検証	物理レイアウト後のゲート・レベルでの機能検証
テープアウト	DRC/ERC/LVS検証後、テープアウト

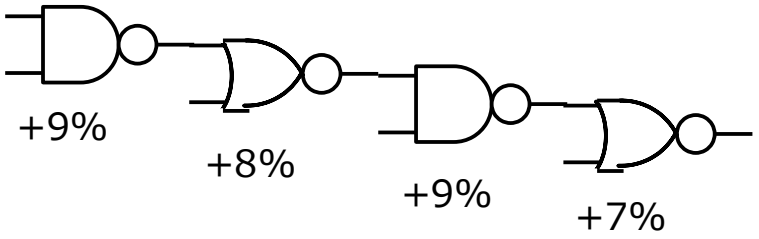
試作 → 機能・特性の評価 → テスト立ち上げ → 信頼性試験 → 量産

mmRISC-1をシリコン化

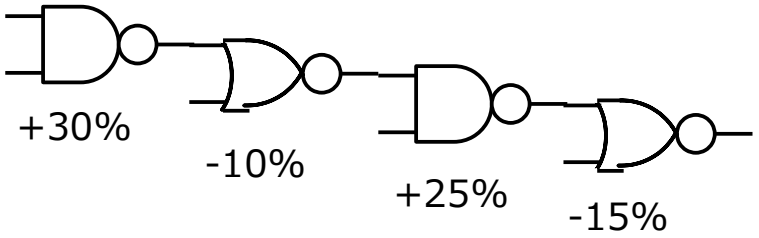
●苦労した点：OCV (On Chip Variation)

OCV：同一チップ内のゲートの遅延ばらつき ⇨ 静的タイミング解析（STA：Static Timing Analysis）

■レガシー 180nm世代：フラットOCV
・チップ内のゲートのばらつきは同程度の傾向
・大きくても±10%のばらつき
・STAはOCVをワースト・マージンとして加えるだけで簡単



■微細化 22nm世代：AOCV・SOCV・POCV
・チップ内のゲートのばらつきはランダム、しかもでかい
・OCVの最大値±40%をワースト・マージンとして加えるとタイミングが全く閉じない（悲観的になりすぎる）



・AOCV (Advanced OCV)：短いパスは悲観的に、長いパスは楽観的にみる
・SOCV (Statistical OCV)／POCV (Parametric OCV)：セル遅延を平均値と標準偏差で統計的に扱う

・STA用EDAツールが**高級化**！　要するに**金**！

■FinFET 7nm世代は、フラットOCVだと±50%～±100%のばらつきになるらしい。
AOCV/SOCV/POCVを駆使しても設計が大変になる

mmRISC-1をシリコン化

●苦労した点：温度に対する遅延特性

■レガシー 180nm世代：

・温度が高いと電子や正孔が散乱されてキャリア移動度が低くなるため、温度が高いと遅延が大きい

・ワースト条件がどこにあるか把握しやすい（Tj=-40℃はベスト、Tj=+125℃はワースト）

・検証条件すなわちサインオフ条件を絞りやすい

■微細化 22nm世代：

・電源電圧の低下とともに、閾値電圧（Vth）の影響が大きくなる

・温度が低いとVthが上がる⇨電流を減らそうとする⇨遅延が増える

・移動度とVthの凌ぎ合いで、どこがワースト温度条件かがわかりにくい

・検証条件すなわちサインオフ条件を絞れず、多くの条件での検証が必須！

※Tj=0℃くらいがワースト遅延になる条件もある。

●出来上がった22nmシリコンチップ

●組み込みフィードバック制御アプリ向けMCU

・複数のエンド・ユーザでシステム試作済/試作中

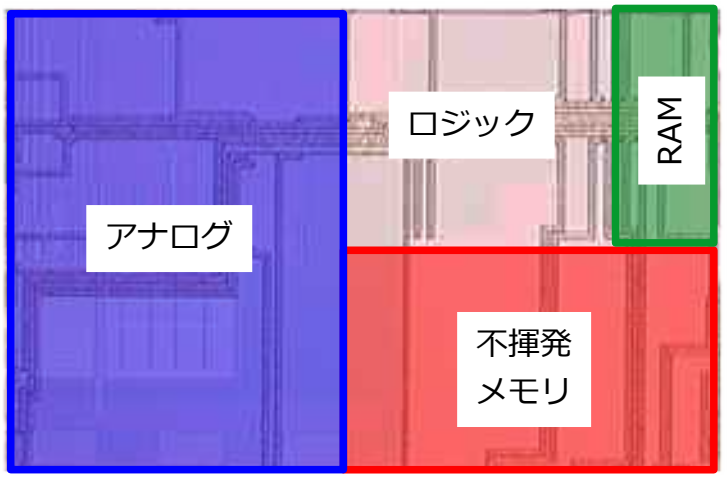
・CP歩留り ＞＞ 95%

●mmRISC-1コア

・RV32IMAFC（符号小数点演算命令含む）

・2-wire cJTAGデバッガ

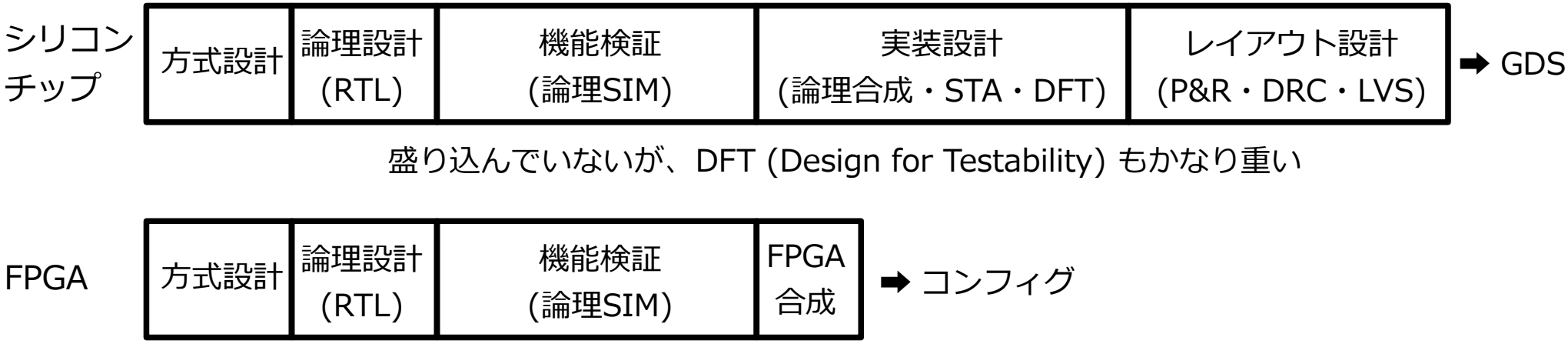
ロジックはゴミ



●苦労が多く緊張もするが、何千万・何億個も旅立ち、世の中への貢献度は高い

mmRISC-1をシリコン化

●製品としてのシリコン化は、設計工数・EDA費用がとにかく大変



●シリコン・チップ化を誰でもできるのがオープンソース・シリコン！

FPGAとの違い：
- どこにもないこの世で唯一のレイアウトされたチップ
- それが物理的・電気的に動作している不思議さ
- 眺めて楽しい

CPU設計と
そのシリコン化の楽しさ

CPU設計とそのシリコン化の楽しさ

●CPU設計の楽しさ

ーCentral Processing Unitだけに、システムの中心を担える

ー取り巻くエコシステム（コンパイラ、OS、デバッガ）との一体感

ー本当の意味で、ロジカルな思考を楽しめる

ーバグれない責務の重い仕事　→　やりがいと達成感

ー「ソフト＝世の中」を動かす土台なので、やっぱり、なんだか楽しい

●CPUのシリコン化のやりがい

ー本物のCPU（GPU含むプロセサ全般）はシリコンの上で実現可

ー自分の設計が、何千万、何億個と生産されて世のシステムを動かす

ー自分の思考が唯一無二なチップ・レイアウトとなって動く

オープンソース半導体

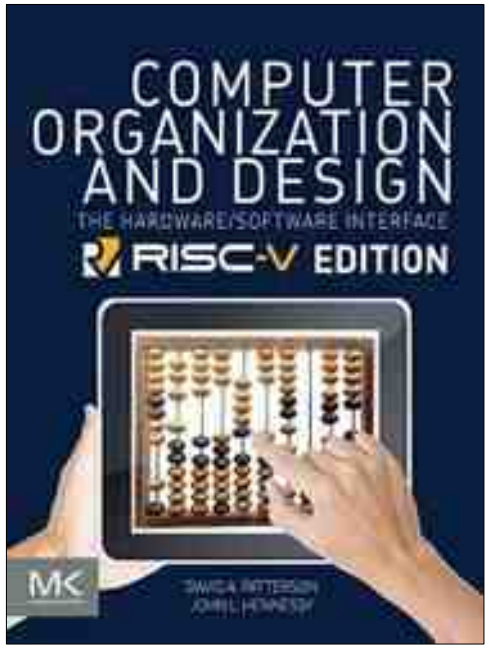
■半導体業界の課題
- ●人材不足（特に日本は、米台中と比較して極端に少ない）
- **●開発費用（ツール、試作）が高い**
- **●開発機会が少ない**
- **●教育機会も不足（子供、学生、若手）**
- **●人材が育たない**

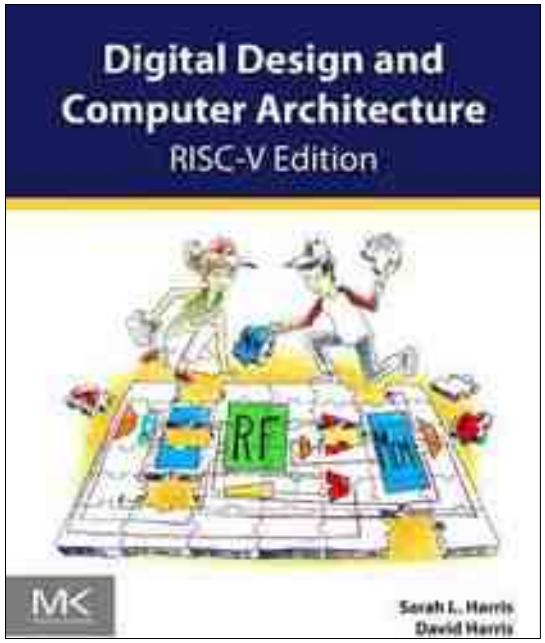
■半導体開発のオープン化（民主化）
- **●オープンソースのEDAツール（無償）**
- **●オープン化されたPDK（無償＋契約不要）**
- **●安価なシャトル試作サービス**

●CPU設計を勉強し、オープンソース・シリコンで実践鍛錬しよう！

●就職なら、ぜひ、半導体メーカまたは半導体設計を行うシステム・メーカへ！

CPU設計の参考書





(左) David Patterson, John Hennessy, Computer Organization and Design RISC-V Edition, 2017, Morgan Kaufmann
(中) Sarah L. Harris, David Harris, Digital Design and Computer Architecture, RISC-V Edition, 2021, Morgan Kaufmann
(右) David Patterson, Andrew Waterman, RISC-V原典 オープンアーキテクチャのススメ, 2018, 日経BP

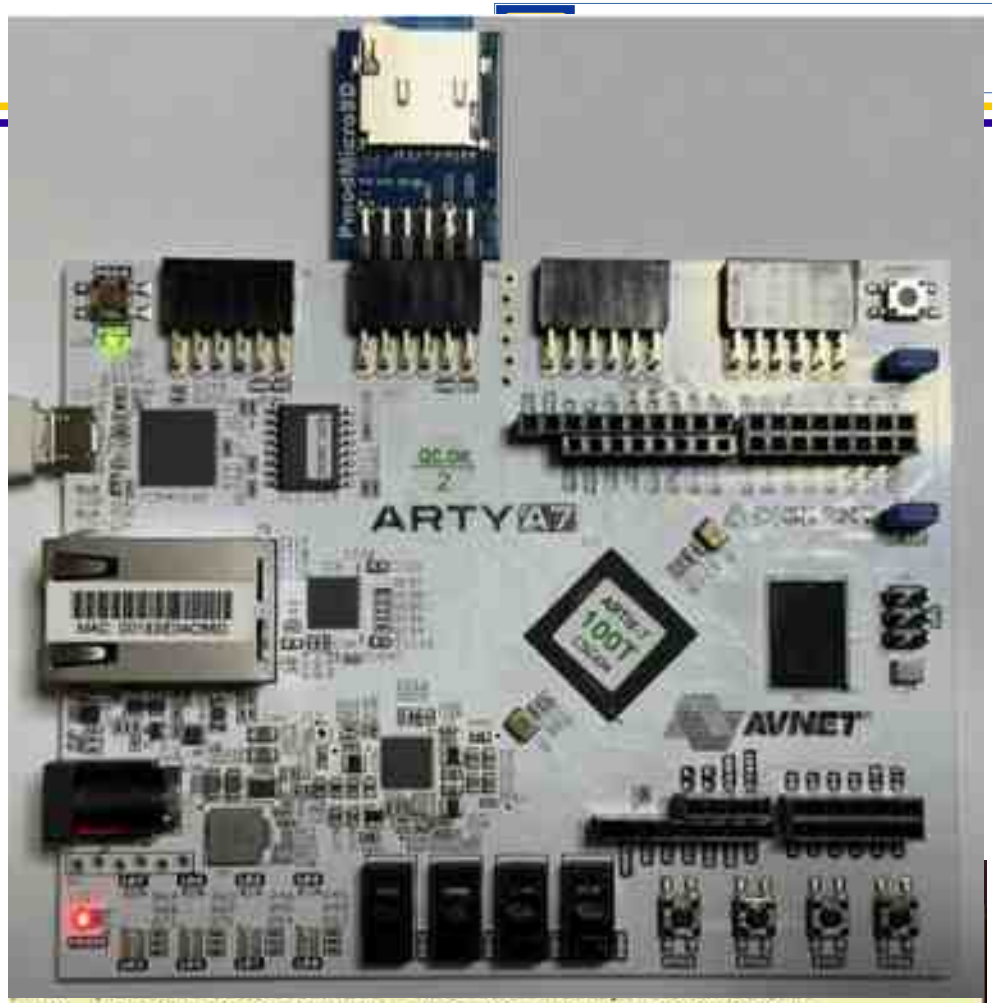
パイプライン
分岐予測
スーパスカラ
アウト・オブ・オーダ
マルチ・スレッド
マルチ・プロセサ
キャッシュ・メモリ
仮想記憶

今、学び、遊んでいること

●mmRISC-2のゆっくり開発
ーRV64GC
ー$L1, $L2, MMU
ーLinuxブート
ー組み込み用途に切り取り易い構造

●お手本：vivado-risc-v
ーRocket Chipベース
ーlitex使わないシンプルSoC
ーこのお手本でDebianの立ち上げはOK

●シリコン化：ChipFoundry?
ーCPUコア部をシリコン化？
ーシステム側はFPGAで？
ー$14,950なので勇気はない




```
[  OK  ] Finished systemd-user-sessions.service - Permit User Sessions.
[  OK  ] Started serial-getty@ttyAU0.service - Serial Getty on ttyAU0.
[  OK  ] Reached target getty.target - Login Prompts.
[  OK  ] Started ssh.service - OpenBSD Secure Shell server.
[  OK  ] Reached target multi-user.target - Multi-User System.
[  OK  ] Reached target graphical.target - Graphical Interface.
pl04p104
Debian GNU/Linux 13 debian ttyAU0

debian login: debian
Password:
Linux debian 6.19.12-dirty #1 SMP Sun May  3 13:45:01 JST 2026 riscv64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
debian@debian:~$
```

まとめ

- picoCPUでCPU設計を体験しよう
- 世界最初のマイクロ・プロセサ4004の設計とシリコン化
- シンプルなチューリング完全CPU bfCPUの設計とシリコン化
- 実用RISC-V CPUの自作と22nm 商用チップへの実装

- CPU作りは、FPGAでも実シリコンでも楽しい
- Tiny Tapeoutは、自由な創造を拡げるプラットフォーム
- 新たなアイディアで独自チップを設計して試作しよう！
- 半導体は買う時代から自分で創る時代
- **半導体で遊び、半導体を知り、そして次世代半導体の創造を！**