ComProcの進捗報告

割込サポート、UARTモジュール内製化、ほか

2023年12月3日　第2回 自作CPUを語る会

サイボウズ・ラボ　@uchan_nos

自己紹介

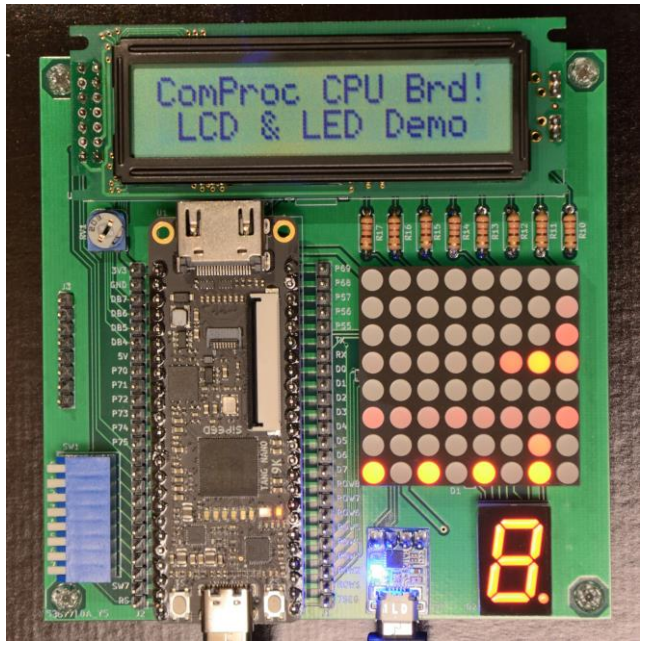
- 内田公太 @uchan_nos
- サイボウズ・ラボ株式会社
  - コンピュータ技術エバンジェリスト
  - 教育用OS・言語処理系・CPUの研究開発

- 代表著書「ゼロからのOS自作入門」
- 最近の寄稿記事
  - Software Design 2023年4月号 第1特集 第2章 コンピュータが計算できる理由
- 最近の同人誌
  - コンパイラとCPUどっちも作ってみた

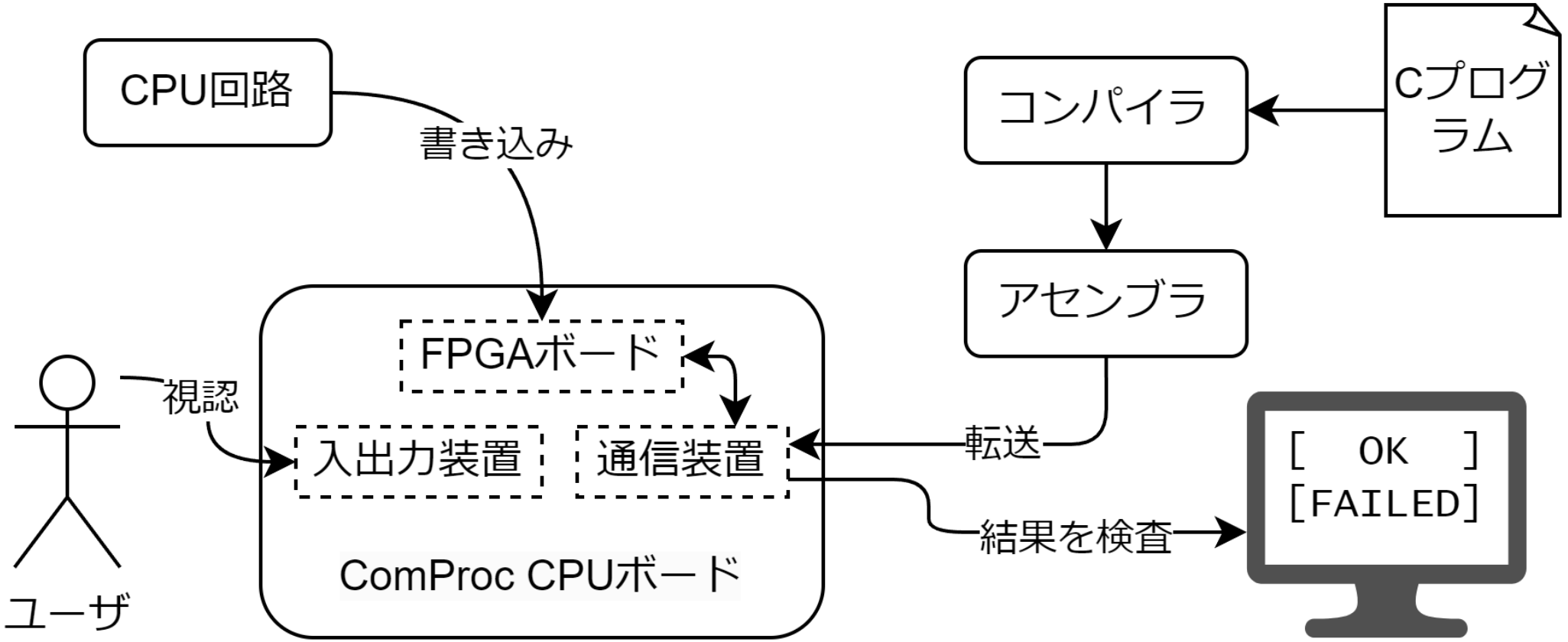
ComProcプロジェクトとは

- ComProc=Compiler+Processor
- CPUとコンパイラを自作する、uchan主導のプロジェクト
  - CPUとコンパイラを作るプロジェクトは珍しくない
  - ComProcプロジェクトは**CPUとコンパイラを同時並行に進化**させる点で、他のプロジェクトとは一線を画す

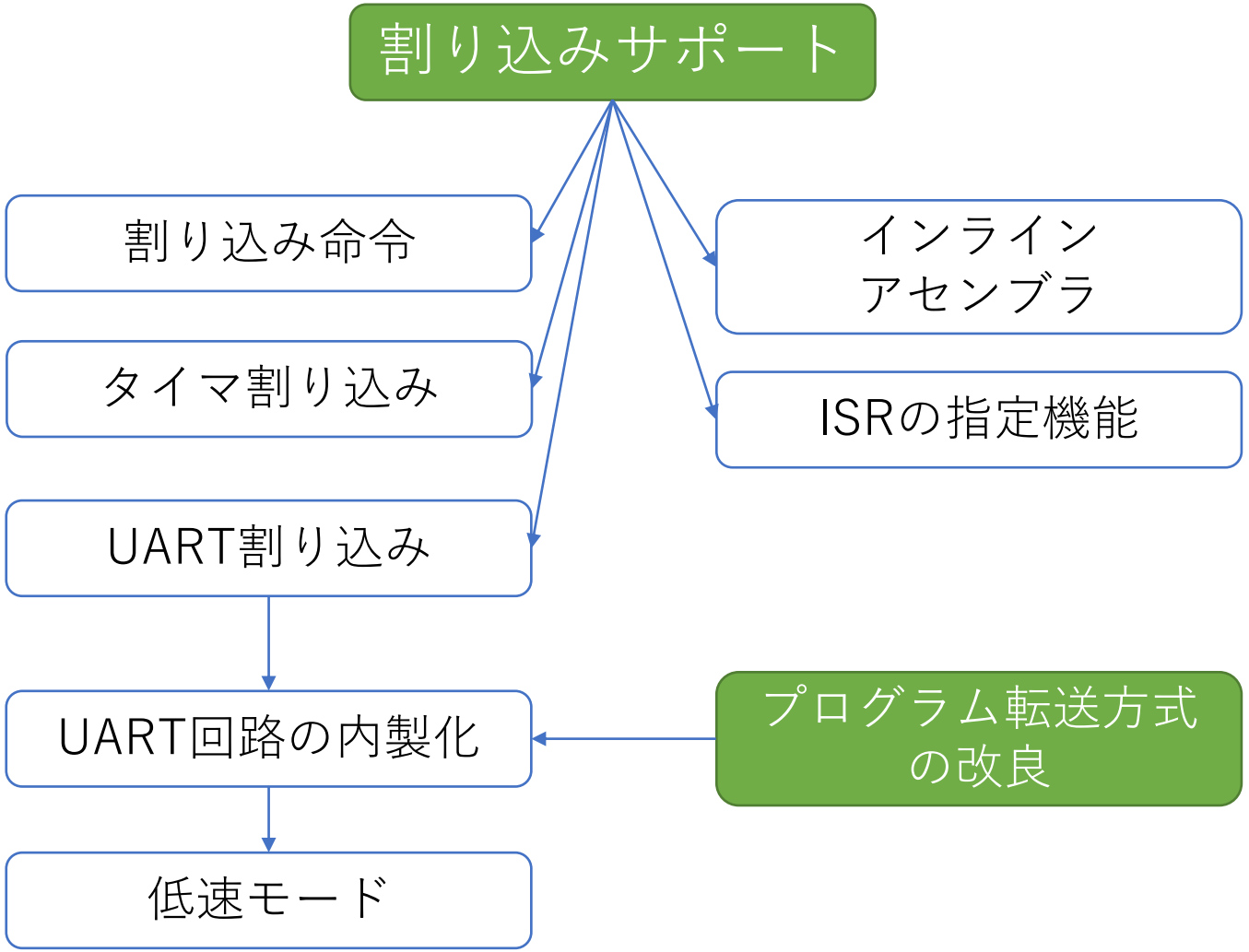
- ComProc CPU Board Rev.4
- FPGAボード「Tang Nano 9K」のI/Oを拡張するマザーボード
  - 出力：LED、キャラクタ液晶
  - 入力：DIPスイッチ
  - 入出力：UART



ComProcプロジェクトの全体像



目次



その他の
コンパイラの改良

グローバル変数の
アドレス指定

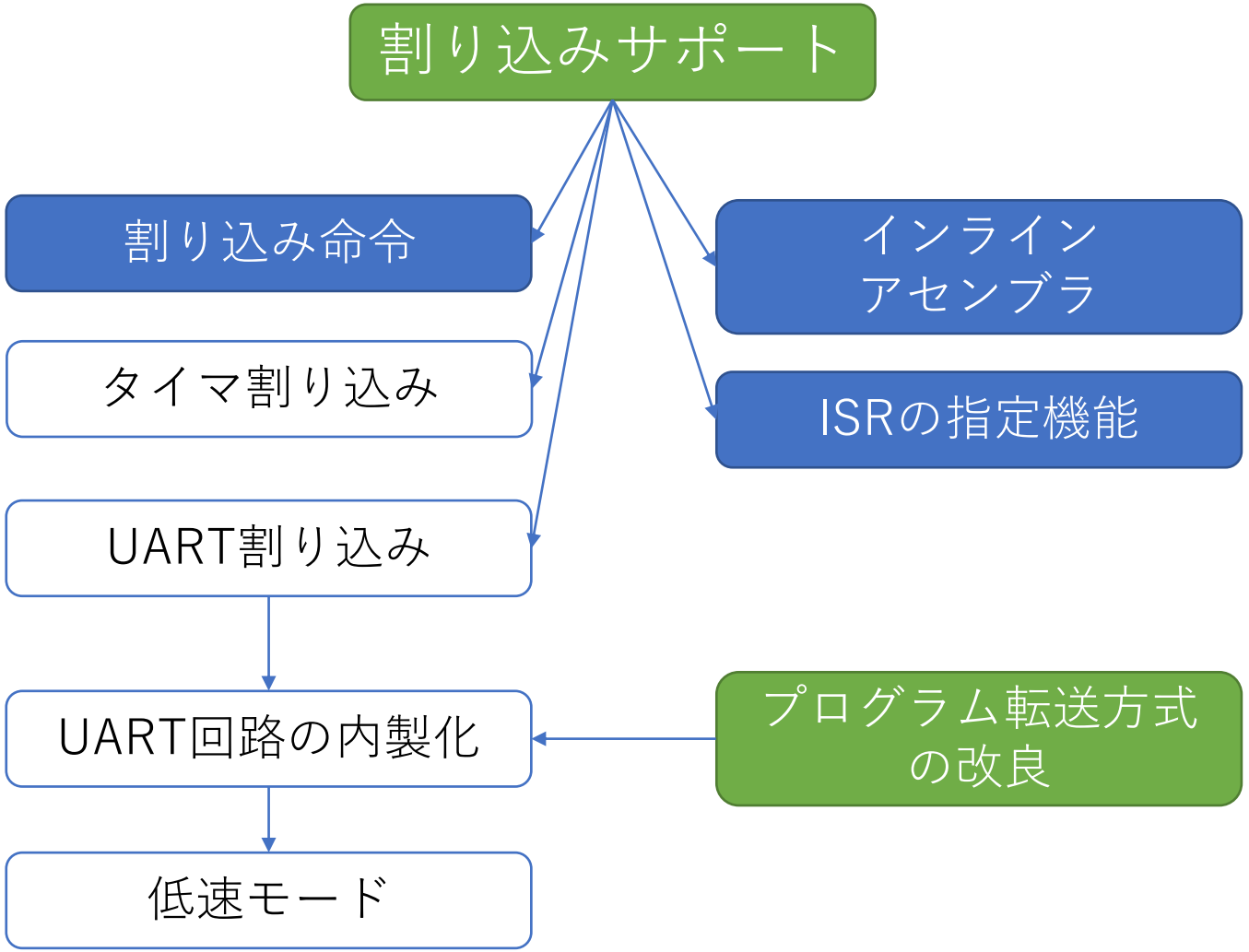
AST情報の印字

減算の構文解析バグ

ビジュアライザ

↑今回は割愛

目次



その他の
コンパイラの改良

グローバル変数の
アドレス指定

AST情報の印字

減算の構文解析バグ

ビジュアライザ

↑今回は割愛

割り込み命令

命令	名前の由来	動作
INT	Interrupt	割り込みハンドラにジャンプ & cstackに戻り先アドレスを保存
ISR	Interrupt Service Routine	スタックトップの値をisrレジスタに転送
IRET	Interrupt Return	cstackに保存されたアドレスにジャンプ & ienをセット

命令の使い方

```
void _ISR() {
  何らかの処理
}
int main() {
  asm("push _ISR¥n¥t"
      "isr¥n¥t"
      "int");
}
```

割り込みハンドラでは、RETではなくIRETで処理を抜ける

ISR命令で割り込みハンドラを登録

INT命令でソフトウェア割り込みを起こす

インラインアセンブラ


```
int main() {
  asm("HELLO");
}
```

コンパイル

```
        add fp,256
        call main
        st 130
fin:
        jmp fin
main:
        cpush fp
        HELLO
        cpop fp
        ret
```

- 指定した文字列が出力にそのまま埋め込まれる
- 引数を埋め込む機能は無い

ISRの指定機能

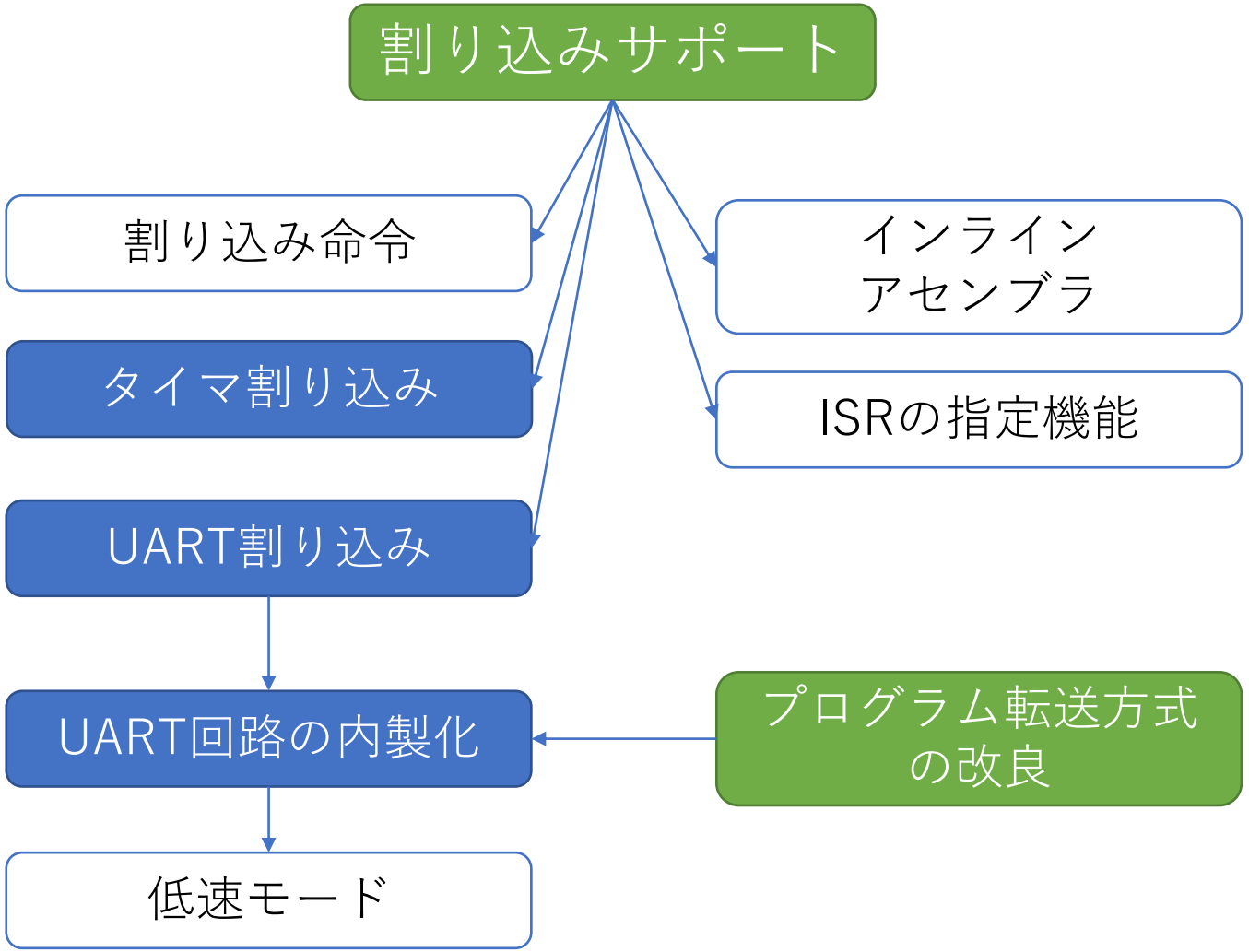
- 割り込みハンドラでは、RETの代わりにIRETを発行したい
- 「割り込みハンドラ」のマークが必要
  - GCCやClangの記法：__attribute__((interrupt))

- ComProcでの記法：_ISRxxx
  - _ISRを冠した名を持つ関数は割り込みハンドラになる

割り込みハンドラを判別するコンパイラのコード

```
if (strncmp(func_sym->name->raw, "_ISR", 4) == 0) {
  ctx->is_isr = 1;
}
…
Insn(ctx, ctx->is_isr ? "iret" : "ret");
```

目次



その他の
コンパイラの改良

グローバル変数の
アドレス指定

AST情報の印字

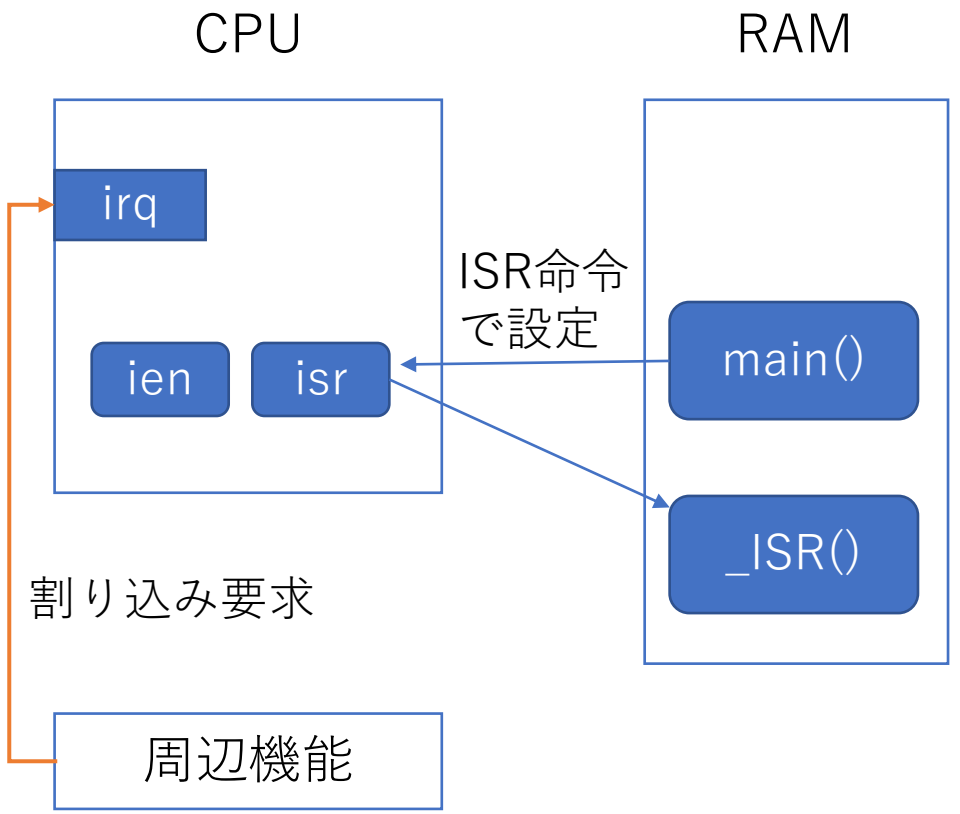
減算の構文解析バグ

ビジュアライザ

↑今回は割愛

ComProc CPUにおける割り込み処理

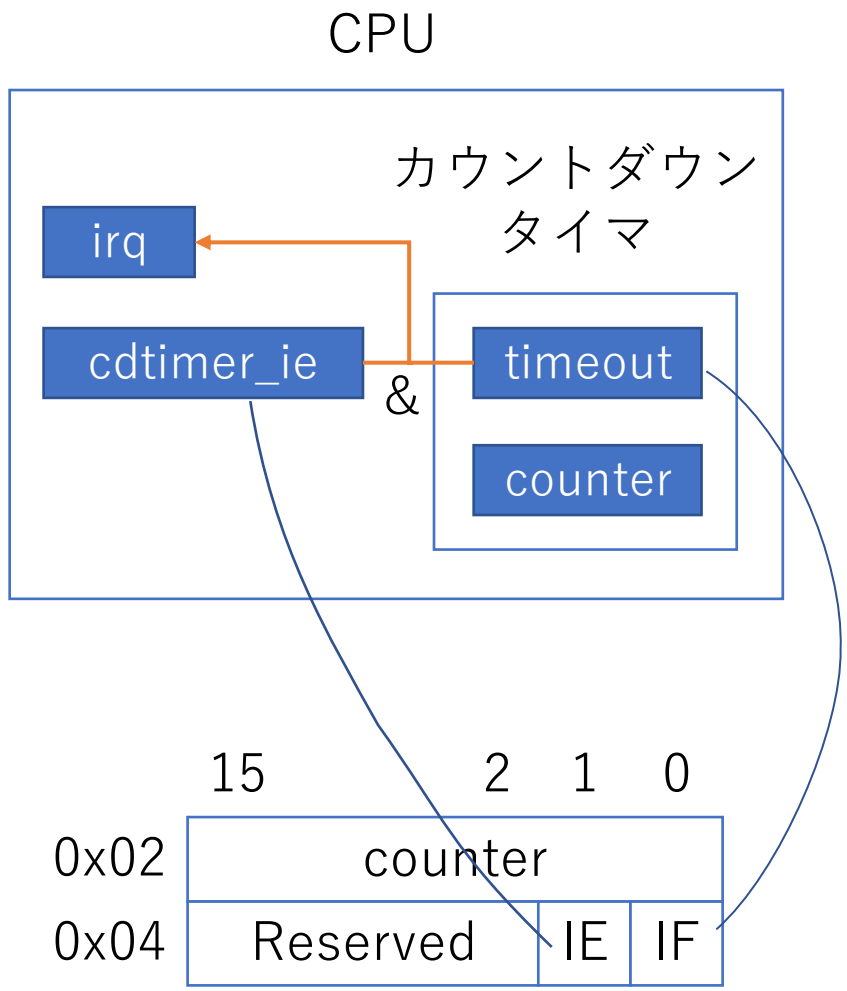
- CPUに追加した主な要素
  - irq (Interrupt Request):
    割込要求入力ポート
  - ien (Interrupt Enable):
    割込許可フラグ
  - isr (Interrupt Service Routine):
    割込ハンドラポインタ
- ISR命令によりハンドラ登録
  - 1つのハンドラのみ登録可
    =PICマイコンと同じ
- irq=1 & ien=1 で割り込み発生



タイマ割り込み

- カウントダウンタイマはカウンタを1ms毎に減らし、0で停止
- 0になるとtimeout=1を出力
- IE=1で割り込み許可

```
void _ISR() {
  char v = *(char*)0x80;
  *(char*)0x80 = (v << 1) | (v >> 7);
  *(int*)2 = 500;
}
int main() {
  asm("push _ISR¥n¥tisr");
  *(char*)0x80 = 0x37;  // LED
  *(int*)4 = 2;   // IE=1
  *(int*)2 = 500; // 500ms
  while (1);
}
```



カウントダウンタイマのレジスタ構成

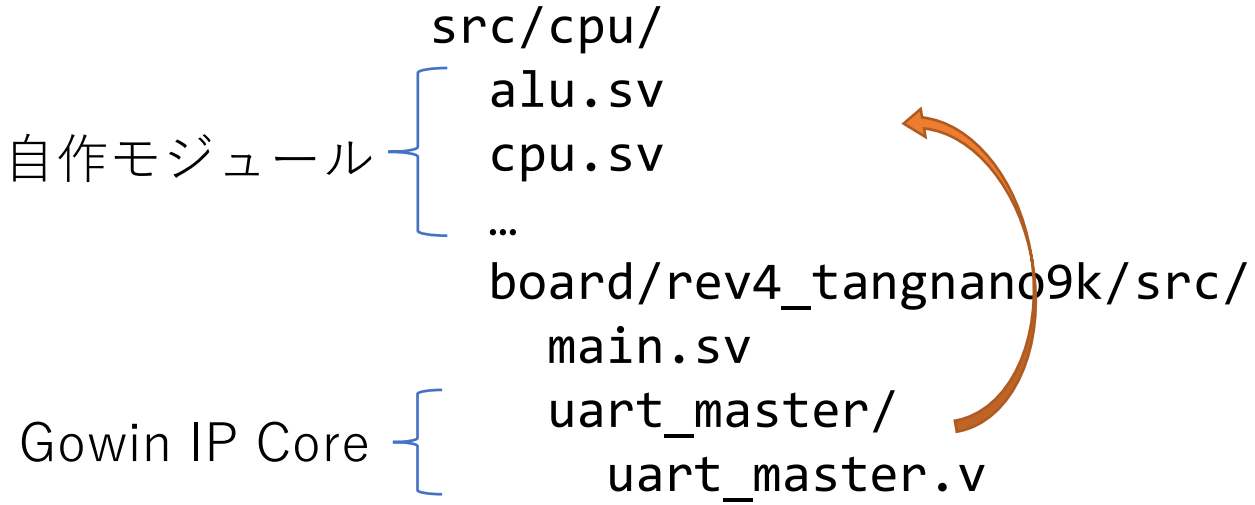
UART割り込みとUART回路の内製化

- UARTが受信を検知できるようにしたい
- 従来、UARTはGowin IPコアを用いていた

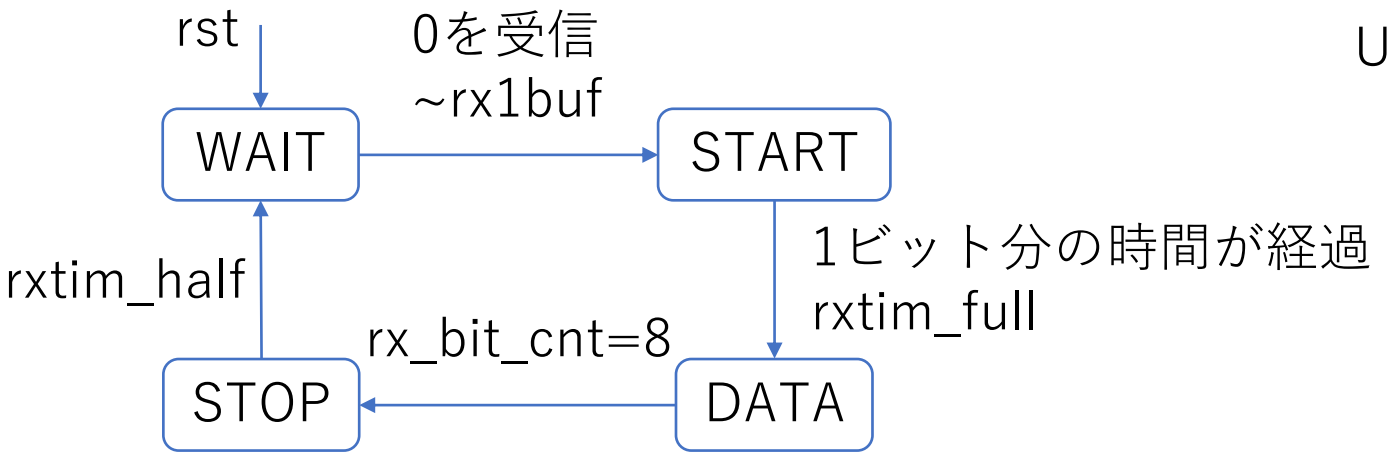
- Gowin IPが提供するUARTモジュールは、仕様が複雑で扱いが難しいし、
- cpuディレクトリに入れにくい

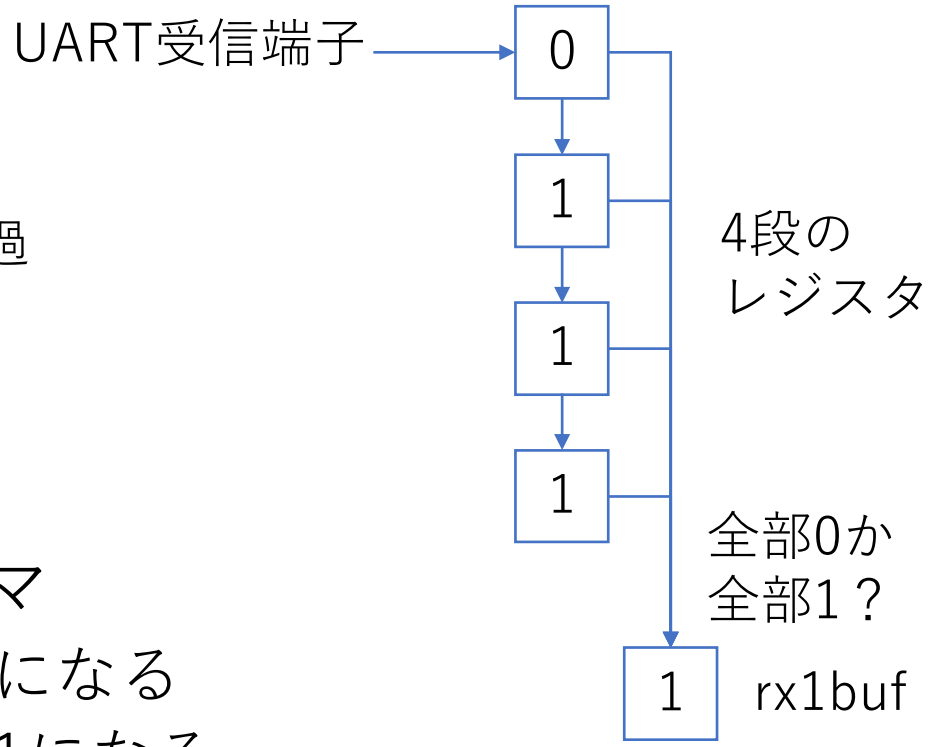
- 扱いやすい回路を自作する
- cpuディレクトリに移す

従来のファイル構成



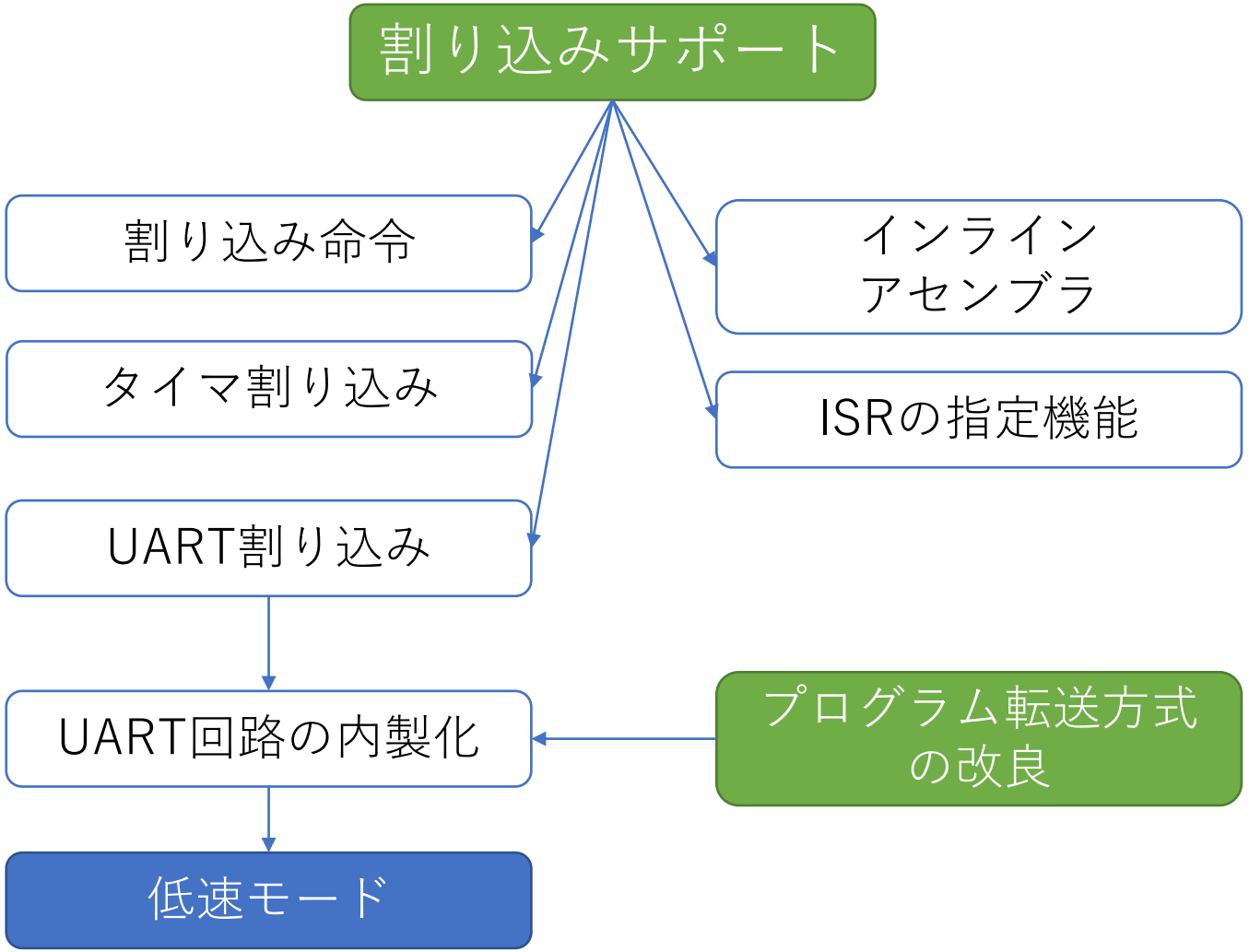
自作UART回路の受信動作





- rxtim：1ビット分の時間を数えるタイマ
  - rxtim_half：0.5ビット経過した瞬間だけ1になる
  - rmtim_half：カウンタが1周する瞬間だけ1になる
- rx1buf：フィルタを通した後の受信信号
  - ノイズ対策
  - 過去4つが全部0か全部1になったときだけ、rx1bufは更新される

目次



その他の
コンパイラの改良

グローバル変数の
アドレス指定

AST情報の印字

減算の構文解析バグ

ビジュアライザ

↑今回は割愛

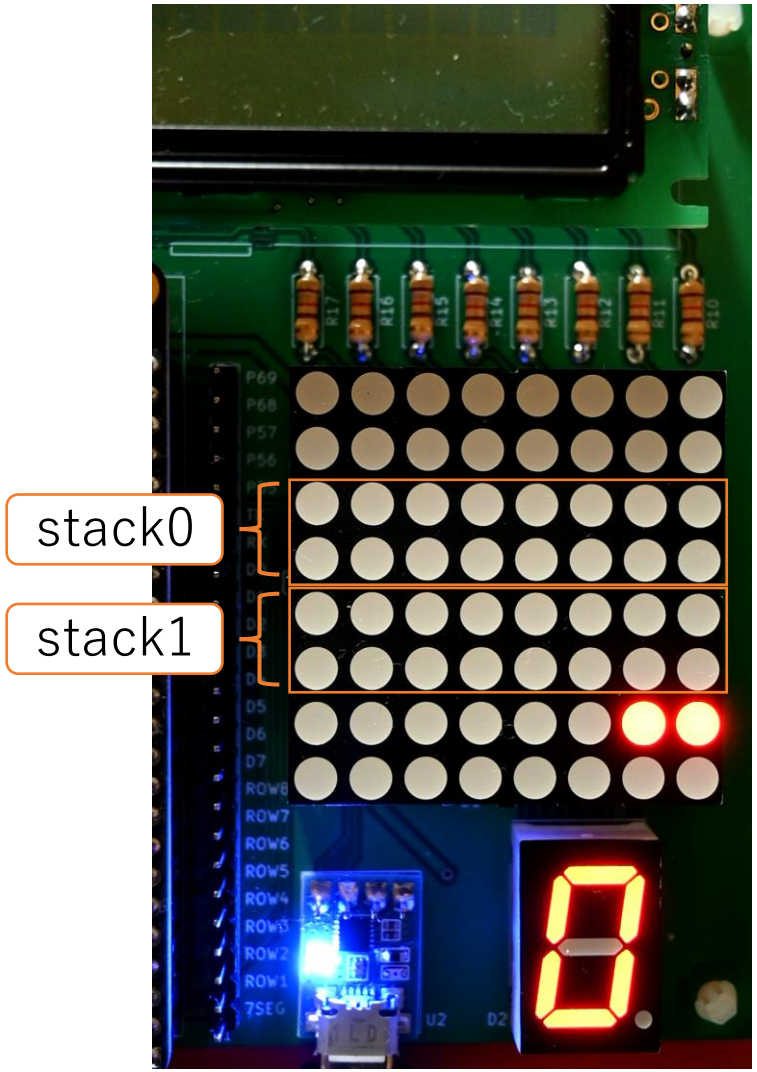
低速モードと謎挙動

- UART回路のデバッグのため、CPUをゆっくり動かしたい
- CPUのクロックを2Hzくらいにして動作実験
- ドットマトリクスLEDに表示する値を変えると、挙動が変わる！（泣）

- 動画は、stackの値をインクリメントし続けるプログラムを走らせている様子

```
L:
      PUSH 1
      ADD
      JMP  L
```

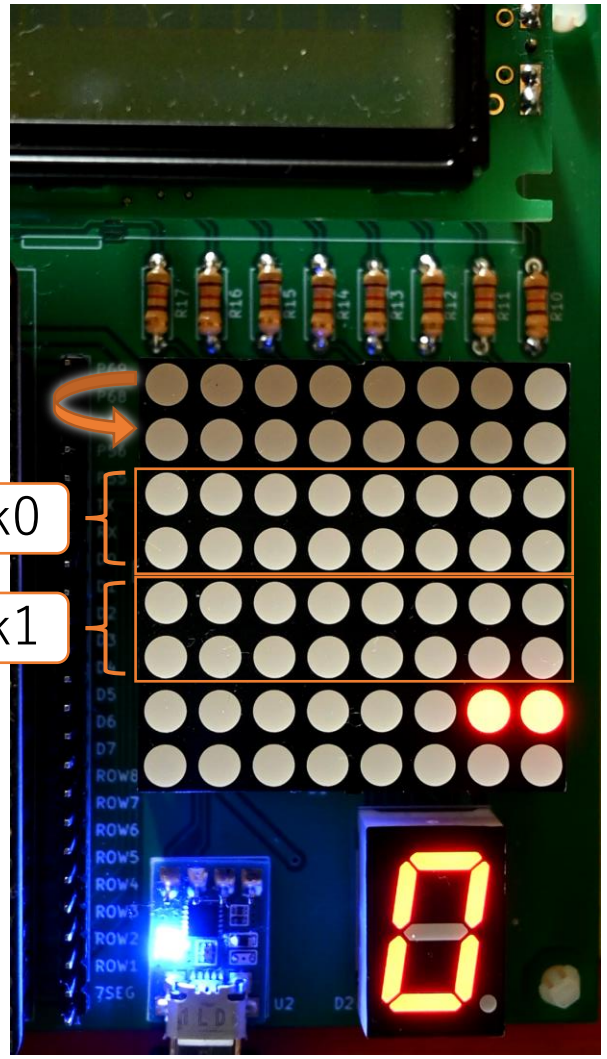
※これは正常な表示



表示値を変えると、挙動不審に

- LEDの1行目に表示していた値を、2行目に表示するようにしただけで、結果が不正に

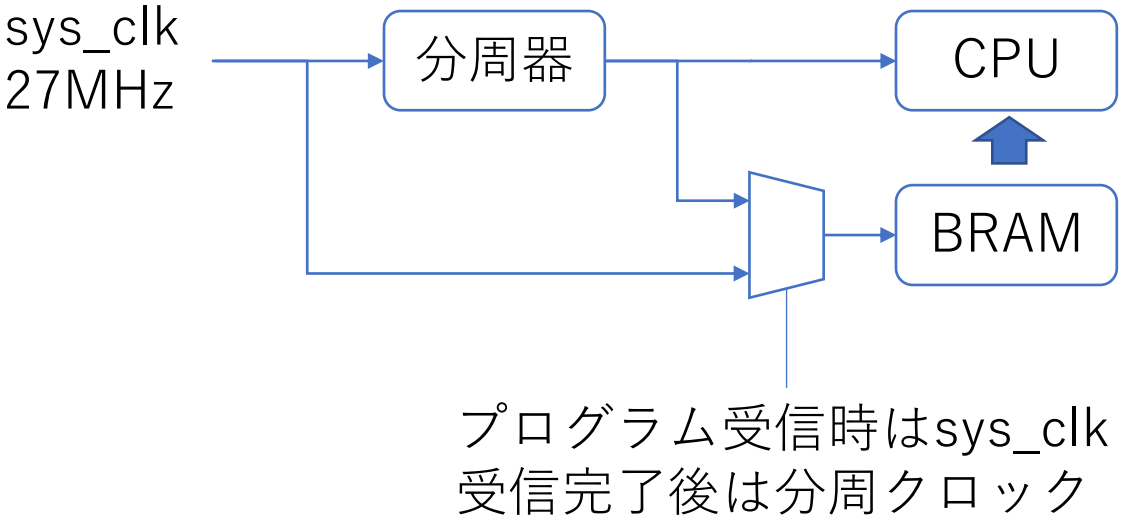
```
// LED の各行に情報を表示
function [7:0] led_pattern(input [3:0] row_index);
  case (row_index)
    4'd0:    led_pattern = wr_data[15:8];
    4'd1:    led_pattern = wr_data[7:0];
    4'd2:    led_pattern = cpu_stack0[15:8];
    4'd3:    led_pattern = cpu_stack0[7:0];
    4'd4:    led_pattern = cpu_stack1[15:8];
    4'd5:    led_pattern = cpu_stack1[7:0];
    4'd6:    led_pattern = {cpu_load_insn, 3'd0, mem_addr[11:8]};
    4'd7:    led_pattern = mem_addr[7:0];
    4'd8:    led_pattern = encode_7seg(mem_addr[4:0]);
    default: led_pattern = 8'b00000000;
  endcase
endfunction
```



苦労した調査と原因

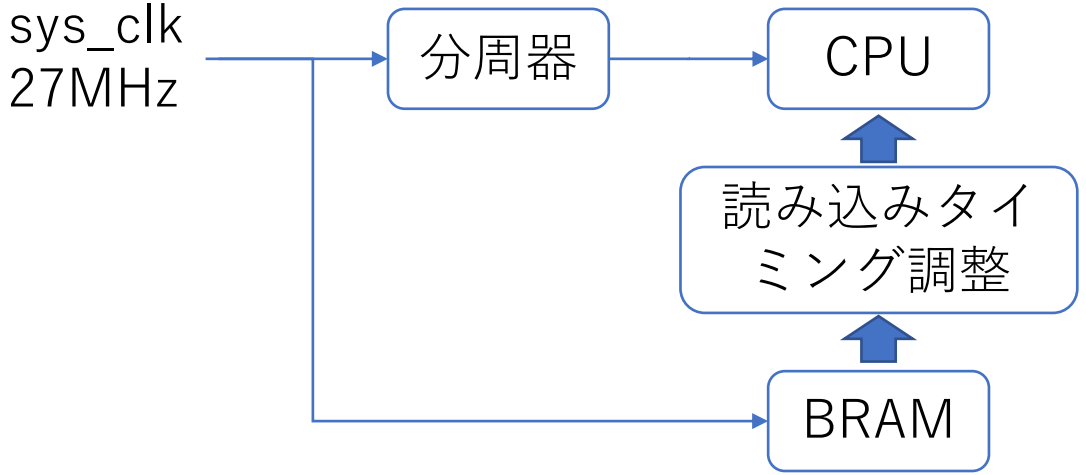
- UART受信のデータは正常で、cpu.insnに取り込まれた命令列が壊れていることが判明
  - 通信ノイズやUART受信回路の問題ではなさそう

- 推測される原因：BRAMに分周したクロックを供給したこと

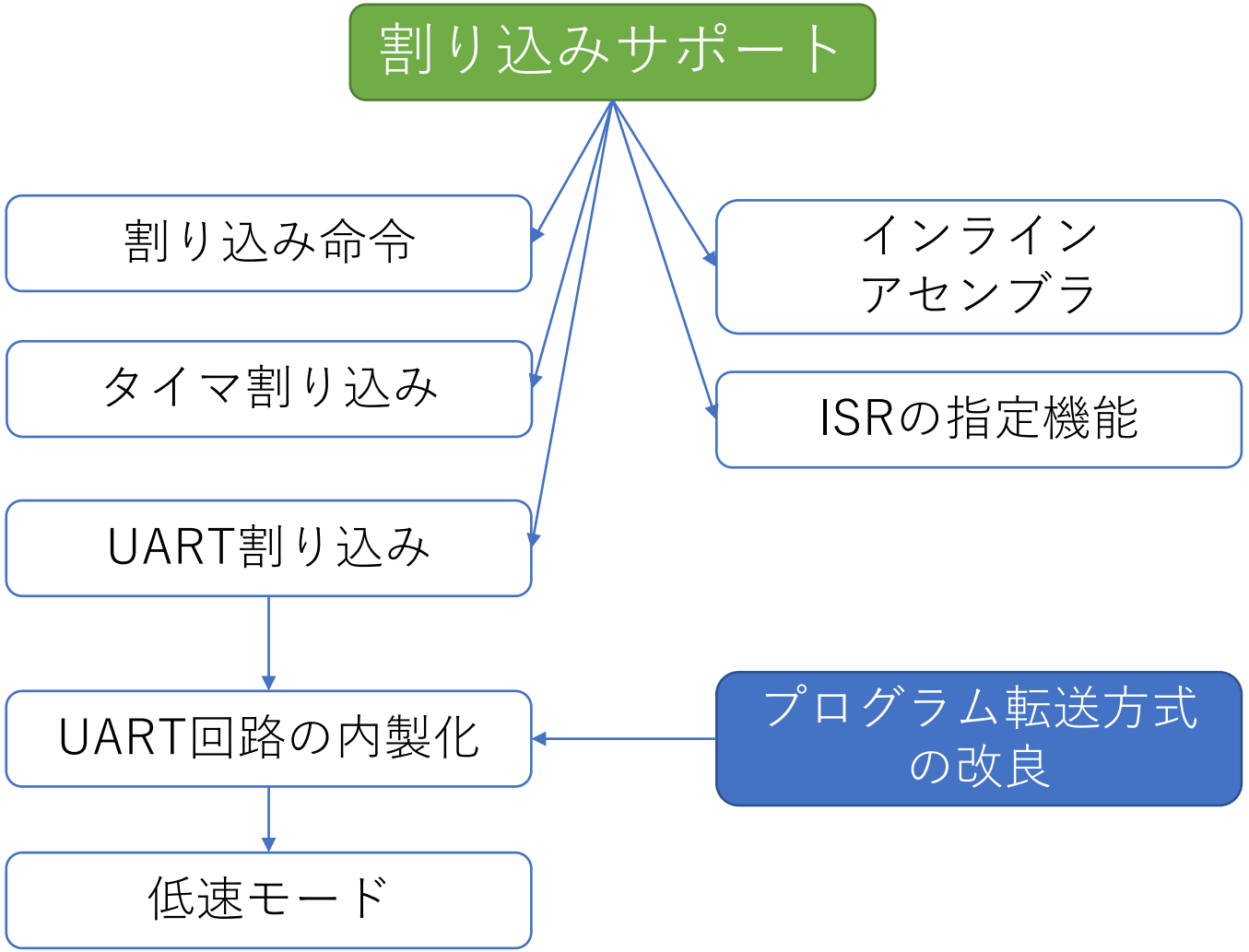


クロック供給を修正

- BRAMにはシステムクロックを入れつつ、タイミング調整回路を介してCPUに接続するようにした



目次



その他の
コンパイラの改良

グローバル変数の
アドレス指定

AST情報の印字

減算の構文解析バグ

ビジュアライザ

↑今回は割愛

プログラム転送方式の改良

- 従来、プログラム転送が主、データ転送が従だった
  - プログラムはそのまま送信
  - CPU上のソフトで読むためのデータは7E XXとして送信
- 本格的に自作CPU用プログラムを開発する上で障害となる
  - データを送る際に7Eを付加しなければならないが、
    そんな処理をやってくれるUART通信ソフトは存在しない

- データ転送を主、プログラム転送を従にする
- プログラム転送のための特別な「マーク」を定義
  - 2ms～20msの間隔で「55 AA」を受信→プログラム転送モードに移行
  - プログラム転送モードで「7F FF」を受信→通常モードに復帰

なぜ2ms～20msにしたのか

55 AAを**データとして**送るとき、
誤って「マーク」と認識しないため

12 34 55 AA BE EF …

ツールを使って連続で送る

- 1文字は10ビット
  - スタート+データ+ストップ
- 115200bps→1文字あたり0.087ms
- 9600bps→1文字あたり1.04ms

- 下限：2ms

手動で1文字ずつ送る

- UART通信ソフトで1文字ずつ送るという想定
  - 「55」と入力し、Enter
- 人間の操作なので、1文字20ms以下では送れないだろう

- 上限：20ms

マークの送出

uart.pyの実装（抜粋）

```
if args.delim:
    ser.write(b'¥x55')
    ser.flush()
    time.sleep(0.005)
    ser.write(b'¥xAA')
    ser.flush()
```

2ms～20msは、
Windowsなどの汎用OS上で
現実的な待ち時間である

10ms±0.1msとかだと
必要な精度が出ない恐れ

今後やりたいこと

- CPU
  - 【完】割り込み機能
  - マイクロマウス製作に必要な周辺回路の設計実装
  - GDB（OpenOCD）と接続できるように、JTAGのサポート
    https://www.besttechnology.co.jp/modules/knowledge/?OpenOCD
- 可視化
  - 【完】 CPUが動作する様子のアニメ生成
    https://twitter.com/cherry_takuan/status/1647263687307317248?s=20
  - MieruCompilerのような可視化
    http://www.sde.cs.titech.ac.jp/~gondow/MieruCompiler/
- 最適化
  - コンパイラにさらなる最適化機能を追加

命令セット（即値あり命令）

CybozuLabs

mnemonic	15 87 0	説明
PUSH uimm15	1 uimm15	uimm15 を stack にプッシュ
JMP simm12	0000 simm11 0	pc+simm12 にジャンプ
CALL simm12	0000 simm11 1	コールスタックに pc+2 をプッシュ、pc+simm12 にジャンプ
JZ simm12	0001 simm11 0	stack から値をポップし、0 なら pc+simm12 にジャンプ
JNZ simm12	0001 simm11 1	stack から値をポップし、1 なら pc+simm12 にジャンプ
LD.1 X+simm10	0010xx simm10	バイトバージョン
ST.1 X+simm10	0011xx simm10	バイトバージョン
LD X+simm10	0100xx simm9 0	mem[X+simm10] から読んだ値を stack にプッシュ
ST X+simm10	0100xx simm9 1	stack からポップした値を mem[X+simm10] に書く
PUSH X+simm10	0101xx simm10	X+simm10 を stack にプッシュ
		X の選択: 0=0, 1=fp, 2=ip, 3=cstack[0]
	011000xxxxxxxxxx	予約
ADD FP,simm10	011001 simm10	fp += simm10
	01101xxxxxxxxxxx	予約
	0111xxxxxxxxxxx	即値なし命令（別表）

命令セット（即値なし命令）

CybozuLabs

```
mnemonic    15     87      0  説明
-----------------------------------
NOP         |0111000000000000| stack[0] に ALU-A をロードするので、ALU=00h
POP         |0111000001001111| stack をポップ
                               stack[0] に ALU-B をロードするので、ALU=0fh
POP 1       |0111000001000000| stack[1] 以降をポップ（stack[0] を保持）
                               stack[0] に ALU-A をロードするので、ALU=00h
INC         |0111000000000001| stack[0]++
INC2        |0111000000000010| stack[0] += 2
NOT         |0111000000000100| stack[0] = ~stack[0]

AND         |0111000001010000| stack[0] &= stack[1]
OR          |0111000001010001| stack[0] |= stack[1]
XOR         |0111000001010010| stack[0] ^= stack[1]
SHR         |0111000001010100| stack[0] >>= stack[1]（符号なしシフト）
SAR         |0111000001010101| stack[0] >>= stack[1]（符号付きシフト）
SHL         |0111000001010110| stack[0] <<= stack[1]
JOIN        |0111000001010111| stack[0] |= (stack[1] << 8)

ADD         |0111000001100000| stack[0] += stack[1]
SUB         |0111000001100001| stack[0] -= stack[1]
MUL         |0111000001100010| stack[0] *= stack[1]
LT          |0111000001101000| stack[0] = stack[0] < stack[1]
EQ          |0111000001101001| stack[0] = stack[0] == stack[1]
NEQ         |0111000001101010| stack[0] = stack[0] != stack[1]

DUP         |0111000010000000| stack[0] を stack にプッシュ
DUP 1       |0111000010001111| stack[1] を stack にプッシュ
RET         |0111100000000000| コールスタックからアドレスをポップし、ジャンプ
CPOP FP     |0111100000000010| コールスタックから値をポップし FP に書く
CPUSH FP    |0111100000000011| コールスタックに FP をプッシュ
LDD         |0111100000001000| stack からアドレスをポップし、mem[addr] を stack にプッシュ
STA         |0111100000001100| stack から値とアドレスをポップしメモリに書き、アドレスをプッシュ
STD         |0111100000001110| stack から値とアドレスをポップしメモリに書き、値をプッシュ
                               stack[1] = data, stack[0] = addr
LDD.1       |0111100000001001| byte version
STA.1       |0111100000001101| byte version
STD.1       |0111100000001111| byte version
INT         |0111100000010000| ソフトウェア割り込みを発生
ISR         |0111100000010001| stack から値を取り出し、ISR レジスタに書く
IRET        |0111100000010010| 割り込みハンドラから戻る
```

即値なし命令のビット構造

```
 15  12 11  10  8    7     6   5  4 3    0
| 0111 | 0 | 000 | Push | Pop |    ALU    |  stack を使う演算系命令
| 0111 | 1 | 000     0     0    00 | Func |  その他の即値無し命令
```