ComProcでDOSを動かす

2024年12月01日 第4回自作CPUを語る会

@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
- 第4回から当会の主催

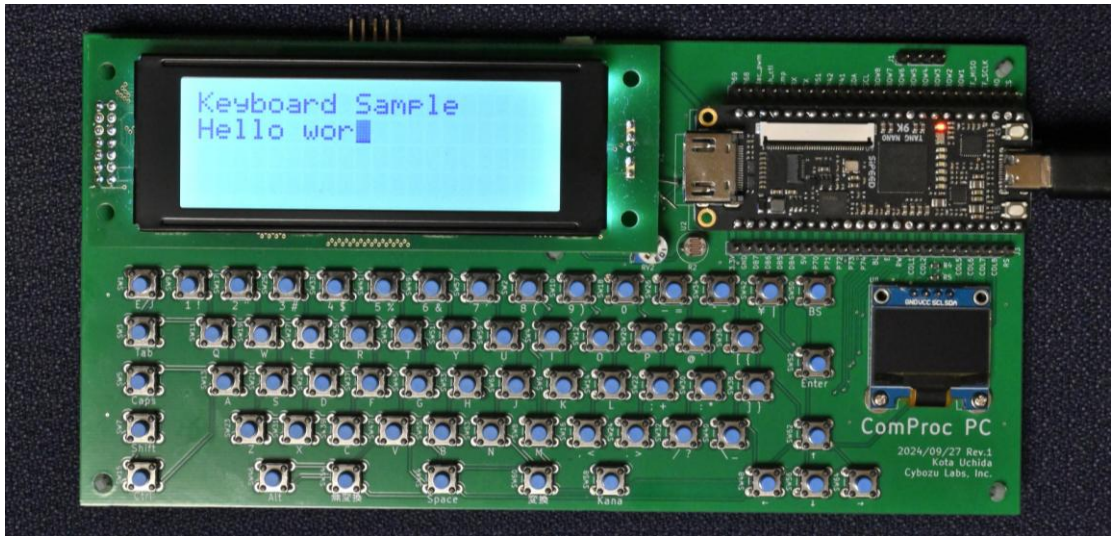
- 代表著作
  - 「ゼロからのOS自作入門」
  - 「自作エミュレータで学ぶx86アーキテクチャ」
  - 「コンパイラとCPUどっちも作ってみた」
  - Software Design 2023年4月号 第1特集 第2章 コンピュータが計算できる理由

ComProcプロジェクトとは

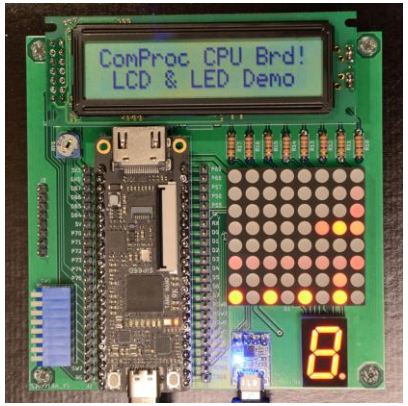
- ComProc=Compiler+Processor
- CPUとコンパイラを自作する、uchan主導のプロジェクト
  - CPUとコンパイラを作るプロジェクトは珍しくない
  - ComProcプロジェクトは**CPUとコンパイラを同時並行に進化**させる点で、他のプロジェクトとは一線を画す

- ComProcプロジェクトの構成要素
  - CPU回路：Verilogで記述されたCPUの実装
  - ComProcボード：FPGAボードを中核とした基板
  - コンパイラ実装：ComProcプロジェクトのもう一つの主役
  - アセンブラ実装：アセンブリ言語プログラムを受け取って機械語へ変換

ComProc PC



ComProc PC Rev.1で
キーボード入力を試している様子



←前作の基板
ComProc CPU Board Rev.4
はドットマトリクスLEDを
装備しており、CPU本体の
開発段階では便利だった。

- 自作CPU用の周辺機器を搭載したボード
  - 4行キャラクタLCD
  - 128×64ピクセルOLED
  - 64キーキーボード
  - 輝度センサー（CdS）
  - 圧電スピーカー
- FPGAボードはTang Nano 9K
  - 右上の黒い基板
  - microSDカードスロット有り

本日の話題一覧

- 自己紹介
- 作ったDOSの紹介
- やったこと
  - 大きなプログラムを動かせるように
  - SDカードのファイル読み取り
  - 位置独立実行ファイルのサポート
  - プログラムメモリへの命令転送
  - 関数ポインタの導入
  - byt信号のバグ修正

本日の話題一覧

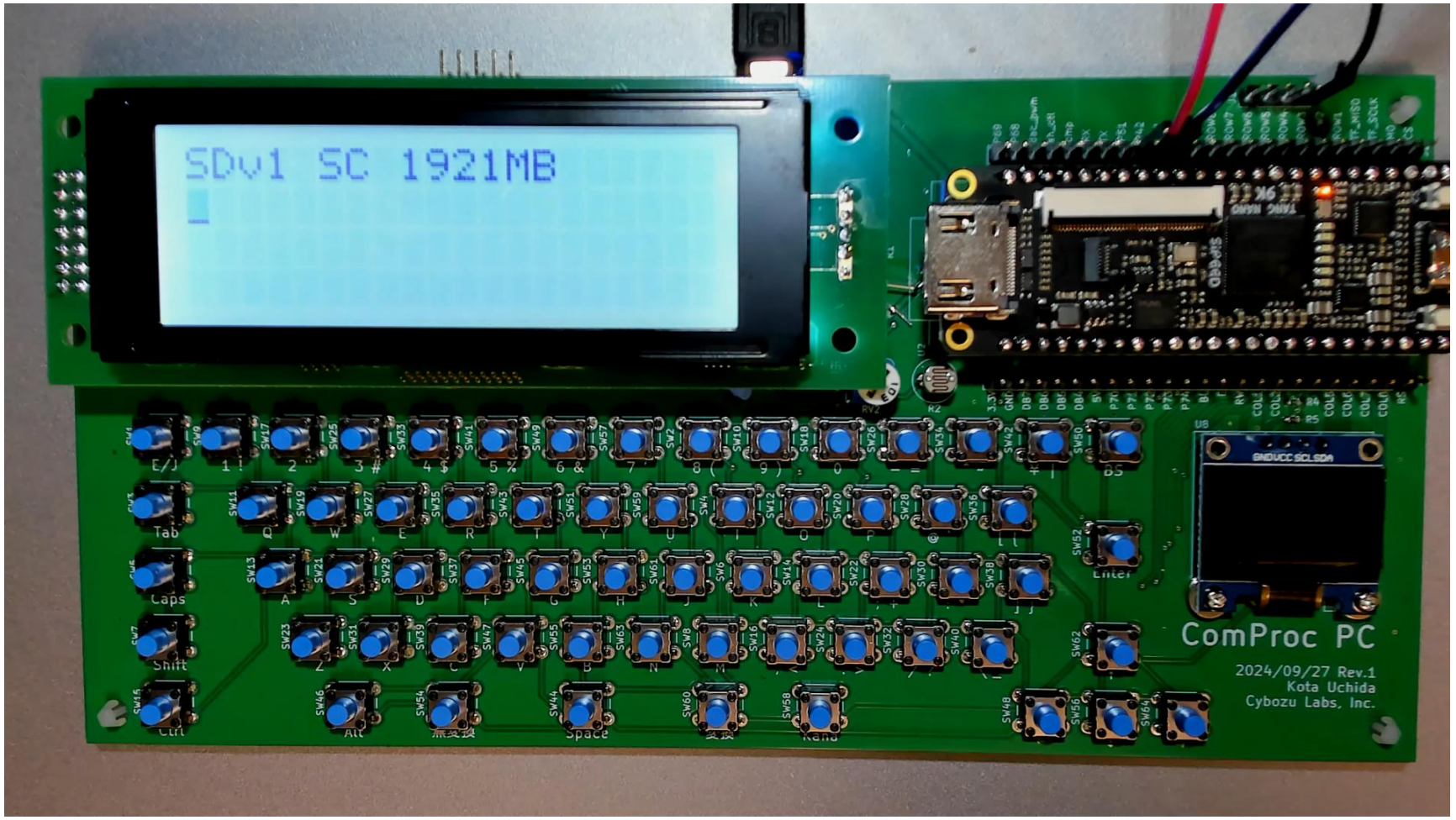
- ●自己紹介
- ●作ったDOSの紹介
- ●やったこと
  - ■大きなプログラムを動かせるように
  - ■SDカードのファイル読み取り
  - ■位置独立実行ファイルのサポート
  - ■プログラムメモリへの命令転送
  - ■関数ポインタの導入
  - ■byt信号のバグ修正

作ったDOSの紹介

- DOS: Disk Operating System
  - 「磁気ディスク装置を使用可能としたオペレーティングシステム」
    - https://ja.wikipedia.org/wiki/DOS_%28OS%29
  - SDカードからデータを読み取るプログラムを作ったので「DOS」と呼ぶことにした

- 作ったDOS
  - SDカードに保存されたプログラムを読み、実行する
    - ルートディレクトリに*.EXEとして保存
  - システムコールはまったく無く、単にプログラムへジャンプするだけ
  - 現状、プログラムローダーと呼ぶ方が正しい

作ったDOSのデモ

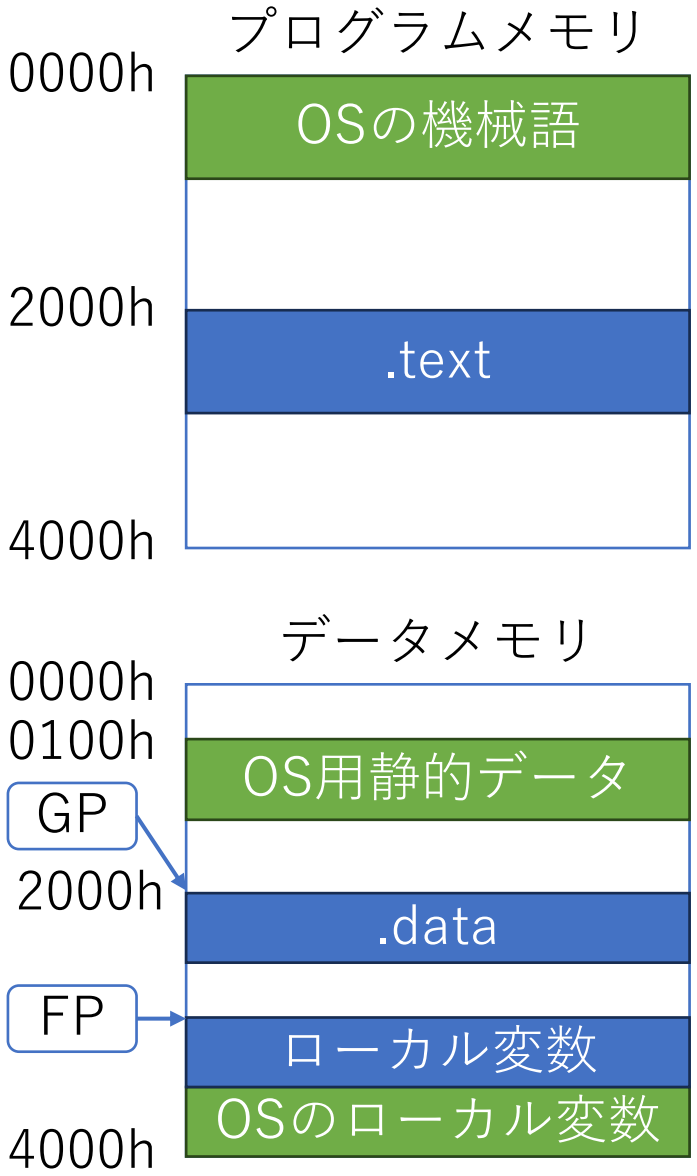


「SDv1…」はDOSの初期表示。

`ls`でSDカードのファイルを一覧する。
`ld <file>`で実行ファイルをメモリに読み出す。
`run`でメモリ上のプログラムを起動する。

「waiting enter...」はapp.exeの出力。Enterを押すとOSに処理が戻り、終了コードが表示される。

DOSと起動されるプログラムの関係



- APP.EXEは2つのセクション.dataと.textを含むファイル。
- .textはプログラムメモリへ置かれ、
- .dataはデータメモリへ置かれる。

- .textの先頭をCALLする。
- .textの先頭には「CALL main」が置かれていて、main関数が実行される。

- GPが.dataの先頭を指す。
- スタックフレームはOS用の領域と連続する。

本日の話題一覧

- 自己紹介
- 作ったDOSの紹介
- やったこと
  - ■大きなプログラムを動かせるように
  - ■SDカードのファイル読み取り
  - ■位置独立実行ファイルのサポート
  - ■プログラムメモリへの命令転送
  - ■関数ポインタの導入
  - ■byt信号のバグ修正

大きなプログラムを動かせるように

●各種の最適化による効率化

項目	最適化前	最適化後	変化率
符号無し整数	1641W	1634W	-0.42%
FPが変化するときだけCPUSH/CPOPを発行	1634W	1624W	-0.61%
引数が1個の場合にST/LDのペアを消去	1624W	1609W	-0.92%
なるべく即値付きCALLを使用	1451W	1341W	-7.6%

●ハーバードアーキテクチャ化によるビット幅拡張

■CALLの即値が14ビット=16Kワードに

■16ビット即値を一発で転送

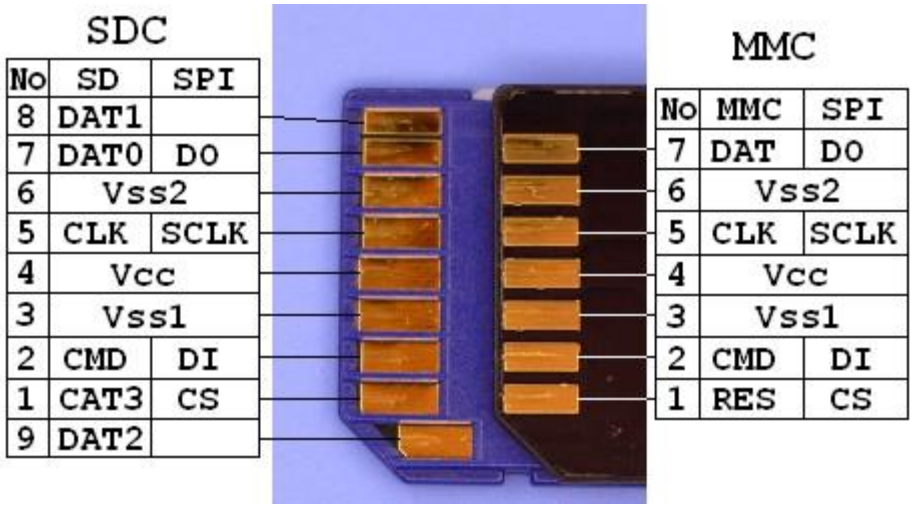
本日の話題一覧

- 自己紹介
- 作ったDOSの紹介
- やったこと
  - 大きなプログラムを動かせるように
  - SDカードのファイル読み取り
  - 位置独立実行ファイルのサポート
  - プログラムメモリへの命令転送
  - 関数ポインタの導入
  - byt信号のバグ修正

SDカードのファイル読み取り

CybozuLabs

- SPI通信回路の追加
  - CS、SCLK、MOSI、MISOの4ピン
- SDカード制御の各種コマンド実装
- ブロック単位での読み込み実験



http://elm-chan.org/docs/mmc/mmc.html

- FAT16ドライバの実装
  - 実験では2GBのmicroSDカードを使用
  - WindowsでFATを選びフォーマットしたらFAT16になった
  - FAT16は16ビットCPUで扱いやすくて助かる～

本日の話題一覧

- 自己紹介
- 作ったDOSの紹介
- やったこと
  - 大きなプログラムを動かせるように
  - SDカードのファイル読み取り
  - 位置独立実行ファイルのサポート
  - プログラムメモリへの命令転送
  - 関数ポインタの導入
  - byt信号のバグ修正

位置独立実行ファイルのサポート

- DOSというからにはSDカードに置いたプログラムを実行したい
- 今までは、0番地から実行開始する前提があった
- これからは、OSとアプリが両方0番地だと困る
  - ComProc MCUには「仮想アドレス」が無いので。

- 主な変更
- CALLをIP相対化
- グローバル変数領域を指すGPを導入

CALLをIP相対化

- 今まで即値付きCALLは絶対アドレス指定だった
- CALLをIP+simm14に改めた
- JMPやJZは元からIP相対
- これで、プログラムをどこに配置しても正常に分岐できる！

```
mnemonic         17   12   7      0  説明
-----------------------------------------
CALL simm14     |0000    simm14    | コールスタックに ip をプッシュし、ip+simm14 にジャンプ
JMP simm12      |000100   simm12   | ip+simm12 にジャンプ
ADD fp,simm12   |000101   simm12   | fp += simm12
JZ  simm12      |000110   simm12   | stack から値をポップし、0 なら ip+simm12 にジャンプ
```

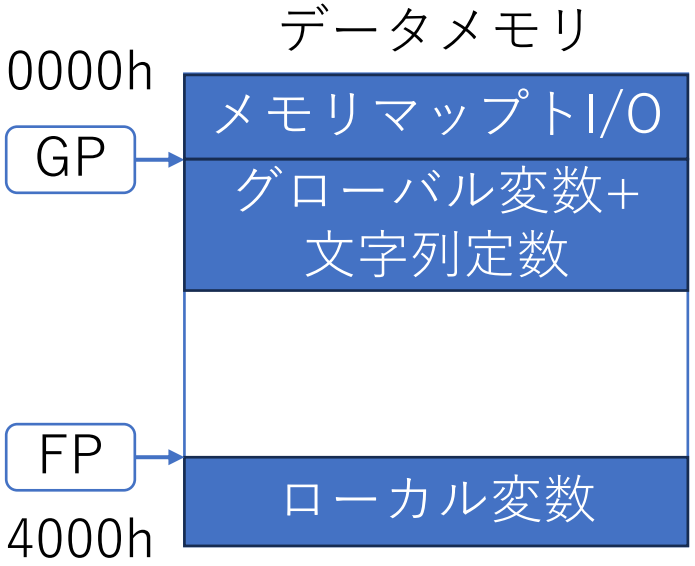
グローバル変数領域を指すGPを導入

- 今まで、メモリアクセス命令（LD/ST）のベースは4種
  - 0：絶対アドレス
  - FP：スタックフレーム先頭
  - IP：実行中の命令アドレス
  - CSTACK：Cstack[0]
    - 古い時代、Cstack[0]にはFPが保存されていた。
- IP相対は、ハーバードアーキテクチャ化により無意味に
- CSTACK相対は、コンパイラの最適化により未使用に
- ということで、ベースを以下3種に改めた
  - 0：絶対アドレス
  - FP：スタックフレーム先頭
  - GP：グローバル領域先頭

GPに対応するコンパイラの修正

CybozuLabs

- 今まで、グローバル変数や文字列定数は絶対アドレスでアクセスしていた
- GP相対でアクセスするように修正
  - 今まで：st zero+0x0142
  - これから：st gp+0x0042
- 絶対アドレスモードは0x0000～0x0100にあるメモリマップトI/O用に残してある



GPを設定するビルトイン関数

- GPを書き換えたくなるが、C言語の標準には機能がない
  - 一般に、インラインアセンブラかビルトイン関数として実現
  - 今回はビルトイン関数にしてみた
- `void __builtin_set_gp(unsigned int);`
  - 単に pop gp に置き換えられる
  - `__builtin_set_gp(41);`
    → push 41; pop gp

- ビルトイン関数とした理由：引数を受け取れるから
- 現状、インラインアセンブラは、C側から値を受け取れない

本日の話題一覧

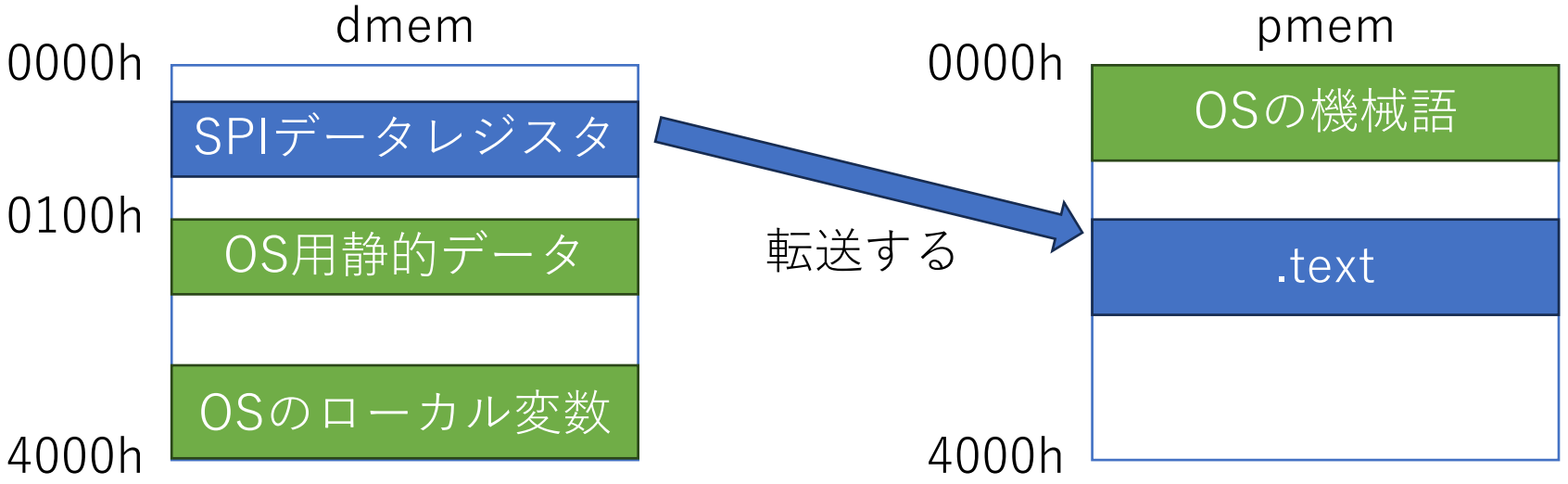
- 自己紹介
- 作ったDOSの紹介
- やったこと
  - 大きなプログラムを動かせるように
  - SDカードのファイル読み取り
  - 位置独立実行ファイルのサポート
  - プログラムメモリへの命令転送
  - 関数ポインタの導入
  - byt信号のバグ修正

プログラムメモリへの命令転送

CybozuLabs

- SDカードから読んだデータは、まずdmemへ置かれる
  - SDカード上のデータ→SPIデータレジスタ（dmem上にマップ）
- 命令として実行するにはpmemに置く必要がある

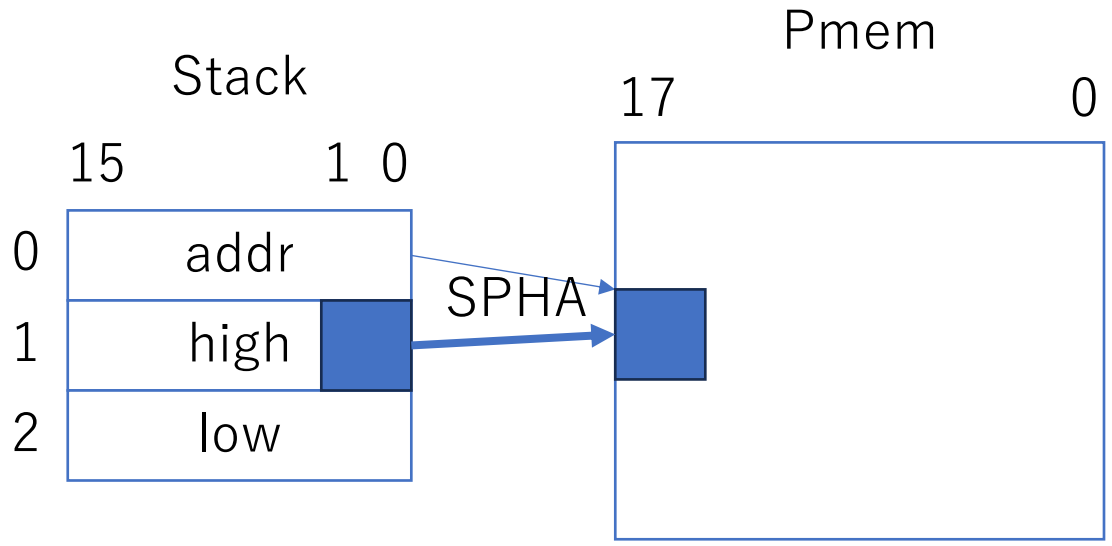
- 演算スタック→pmem転送のための命令を新設：SPHA、SPLA
  - Dmemから演算スタックを経由してpmemへ転送する設計

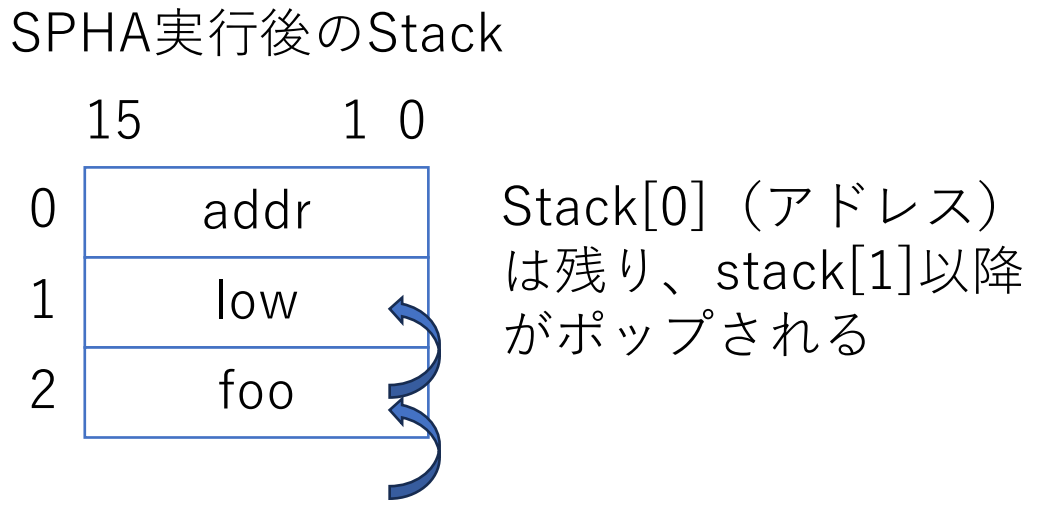


SPHA命令の動作

CybozuLabs

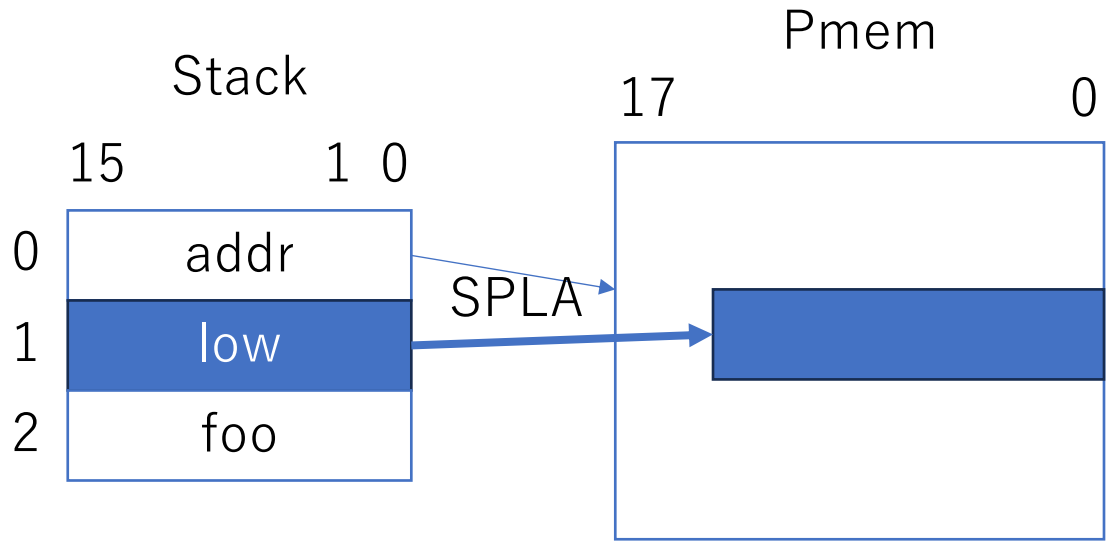
- SPHA：Store to Program memory High word, Address
  - プログラムメモリの上位ワードへ書く。アドレスを残す。
- Pmem[stack[0]] = stack[1] を行い、stack[1]以降をポップする。

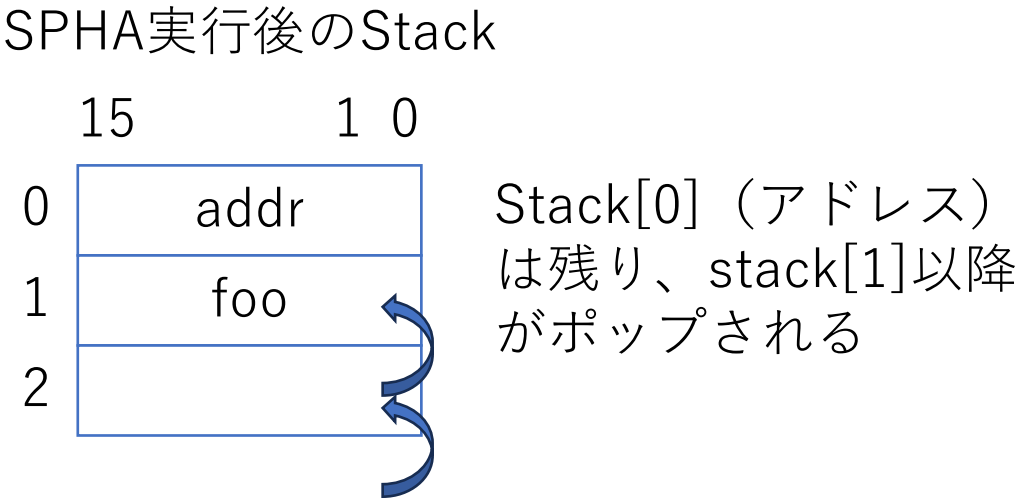




SPLA命令の動作

- SPLA：Store to Program memory Low word, Address
  - プログラムメモリの下位ワードへ書く。アドレスを残す。
- SPHAとSPLAは、転送先の上位、下位が違うだけ





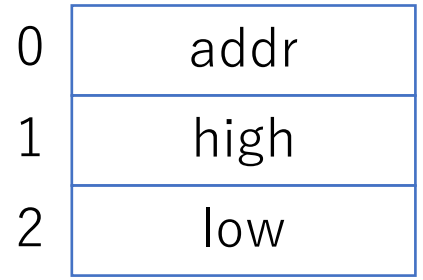
__builtin_write_pmem

- SPHA、SPLAを発行するビルトイン関数
- `int __builtin_write_pmem(int addr, int hi, int lo);`
  - spha; splaに置き換わる

- 関数の引数は右から順にスタックに積まれる
  - `__builtin_write_pmem(addr, high, low)`の場合

```
push low
push high
push addr
spha
spla
```

この時点のスタック



- SPHA、SPLAはアドレスを残すので、連続実行できる！

本日の話題一覧

- 自己紹介
- 作ったDOSの紹介
- やったこと
  - 大きなプログラムを動かせるように
  - SDカードのファイル読み取り
  - 位置独立実行ファイルのサポート
  - プログラムメモリへの命令転送
  - 関数ポインタの導入
  - byt信号のバグ修正

関数ポインタの導入

- OSからアプリを呼ぶために、特定の番地をCALLしたい
- 関数ポインタ！

- 簡易な関数ポインタ型を導入
- 「普通の型」の後に「(」が来たら関数ポインタとみなす
  - 普通の型：intとか、char*とか
- 関数ポインタの文法：*TYPE (* ID) ( )*
  - 今のところ、引数リストには何も書けない
  - 関数呼び出しでは引数のチェックをしていないので、問題無し

関数ポインタの使用例：
OSがアプリを呼ぶところ

```
int (*f)() = 0x1000;
__builtin_set_gp(block_buf);
int ret_code = f();
```

本日の話題一覧

- 自己紹介
- 作ったDOSの紹介
- やったこと
  - 大きなプログラムを動かせるように
  - SDカードのファイル読み取り
  - 位置独立実行ファイルのサポート
  - プログラムメモリへの命令転送
  - 関数ポインタの導入
  - byt信号のバグ修正

byt信号のバグ修正

- DOSを作っていたら不可解なバグに遭遇
- `lcd_puts("MBR ");`でMしか表示されない
- 2回実行すると「MMBR 」となる
- 何回実行しても再現性100%

- デバッグの時系列はXに
  https://x.com/uchan_nos/status/1854286101386780755
- 解決まで足かけ4日！

```
void lcd_out4(int rs, int val) {
  lcd_port = (val << 4) | rs | 1;
  delay_ms(2);
  lcd_port = lcd_port & 0xfe;
}
void lcd_out8(int rs, int val) {
  lcd_out4(rs, val >> 4);
  lcd_out4(rs, val & 0x0f);
}

void lcd_putc(int ch) {
  lcd_out8(4, ch);
}
void lcd_puts(char *s) {
  while (*s) {
    lcd_putc(*s++);
  }
}
```

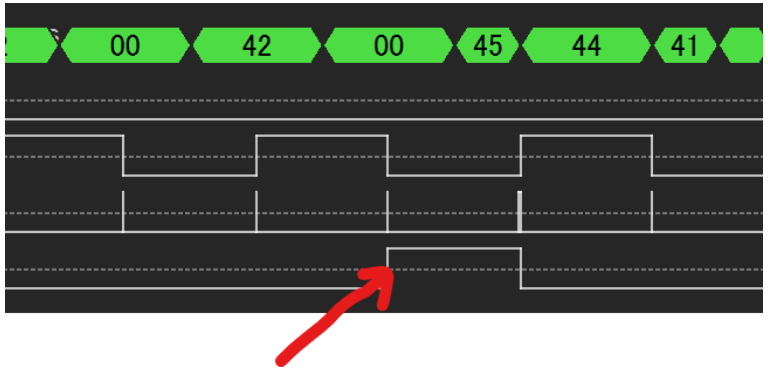
バグ特定の大雑把な流れ1/2

- MCU→LCDの信号を確認
  - 正常。正しく「バグった表示」になる信号が出ていた。
- 最小の再現コードを作る
  - 作る過程で、未使用関数を削除すると挙動が変わるなど、非常に不安定だった。
  - ただし、コードを変えなければ実行結果はいたって安定。
- lcd_putcの呼び出し回数を確認
  - Cコードにカウント処理を追加するとバグが出なくなるため、Verilogで回路的にカウント。
  - 正確にLCDに表示された文字数の分だけ呼ばれている。
- lcd_putsの動作を追う
  - CPU内部の値をデバッグ用のピンに出力し確認
  - 文字コードが読めるはずなのに0（NUL文字）が読めていることが判明

バグ特定の大雑把な流れ2/2

CybozuLabs

- UART受信データを確認
  - 受信データをデバッグ用ピンに出力し確認
  - データ化けは無かった
- メモリをダンプ
  - CPU実行直前のメモリ（dmem）をダンプ
  - 文字コードが書かれているべきところに0が！
  - 正しく書けていない or 途中で0になっちゃう
- メモリの制御信号を確認
  - byt信号が0であるべきなのに1になる瞬間があることが判明



byt信号のバグ修正

- UARTからプログラム受信中はbyt=0に強制することで解消
- `-assign dmem_byt = cpu_dmem_byt;`
- `+assign dmem_byt = img_recv_state == IMG_RECV_WAIT & cpu_dmem_byt;`

- bytはバイトアクセスを示す信号
  - dmemは16ビット幅
  - 一度に16ビットを読み書き可能
  - byt=1だと8ビット単位の読み書きとなる
- UARTから受信したデータは16ビット単位でdmemに書く
- 受信中にbyt=1となると8ビットが失われる
- byt=1になる理由も特定したが、長くなるので割愛

まとめ

このスライドでは
- アプリを起動し
- 31を入力し
- 1F（=31）が表示される

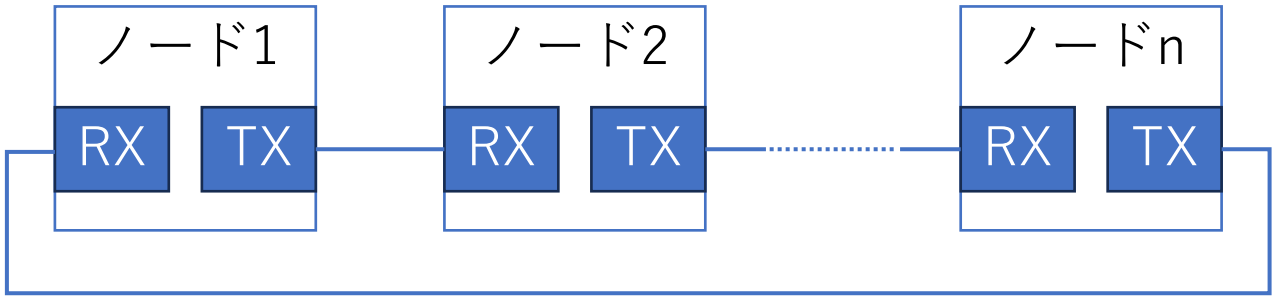
までの道のりを説明した

本日の話題一覧

- 自己紹介
- 作ったDOSの紹介
- やったこと
  - 大きなプログラムを動かせるように
  - SDカードのファイル読み取り
  - 位置独立実行ファイルのサポート
  - プログラムメモリへの命令転送
  - 関数ポインタの導入

おまけ：自作CPU同士を繋ぐ

- MSMP: Make-cpu Simple Messaging Protocol
- 11/12昼：私がNLP-16Aと通信したいと打診
  - ラボユース修了生が作っているNALD Only CPU
- 11/12夜：リングバス状に多数のCPUを繋ぎたいという話が出た



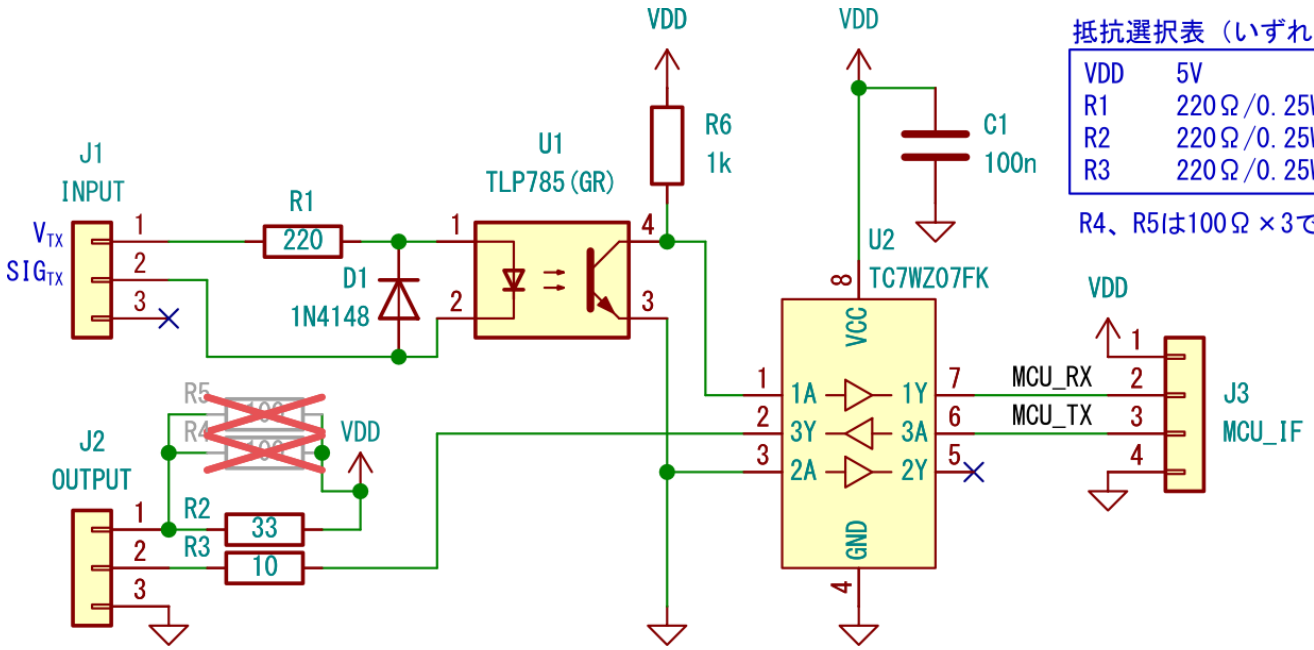
- 11/13：仕様検討を進め仕様を公開
- 11/13夜：インターフェース基板の設計を開始

考案したプロトコルの仕様

- 仕様書
  - MSMP: 自作CPU向けメッセージングプロトコル
    https://scrapbox.io/uchan/%E8%87%AA%E4%BD%9CCPU%E5%90%91%E3%81%91%E3%83%A1%E3%83%83%E3%82%BB%E3%83%BC%E3%82%B8%E3%83%B3%E3%82%B0%E3%83%97%E3%83%AD%E3%83%88%E3%82%B3%E3%83%AB
- 物理層
  - 9600bps UART
  - MIDIと同様のオプトアイソレータによる絶縁
- 論理層
  - 4ビットノードアドレス
  - メッセージ本文0〜63バイト可変長
  - バイト送信間隔の規定
    - デフォルトで20ms
    - 受信バッファが小さいCPUへの配慮

byte	7 4	3 0	意味
0	dddd	ssss	dddd=DST、ssss=SRC
1	ttll	llll	tt=TYPE、llllll=LEN
2	xxxx	xxxx	BODY

インターフェース回路



抵抗選択表（いずれも精度5%品でOK）

VDD	5V	3.3V
R1	220Ω/0.25W	220Ω/0.25W
R2	220Ω/0.25W	33Ω/0.5W
R3	220Ω/0.25W	10Ω/0.25W

R4、R5は100Ω×3で33Ωを代用する場合に実装