自作CPUを語る会

2023/12/03

@kanade_k_1228

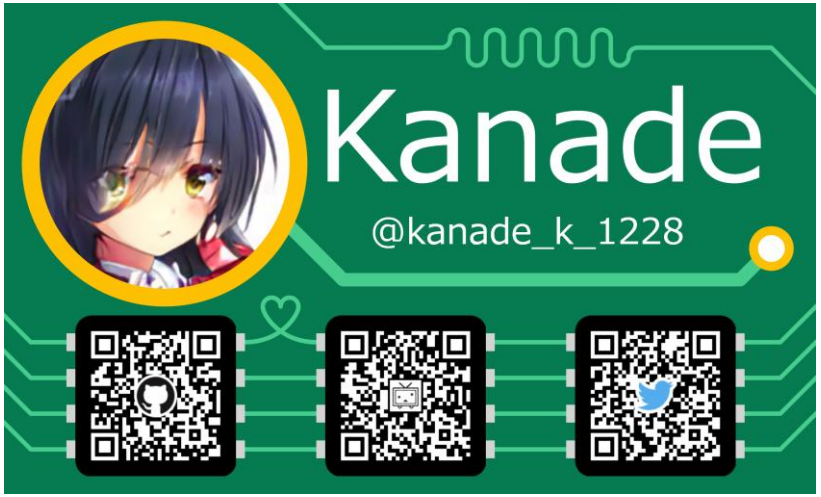
自作CPUを語る会：スライドはここにあります

https://making-cpu.github.io/archive/

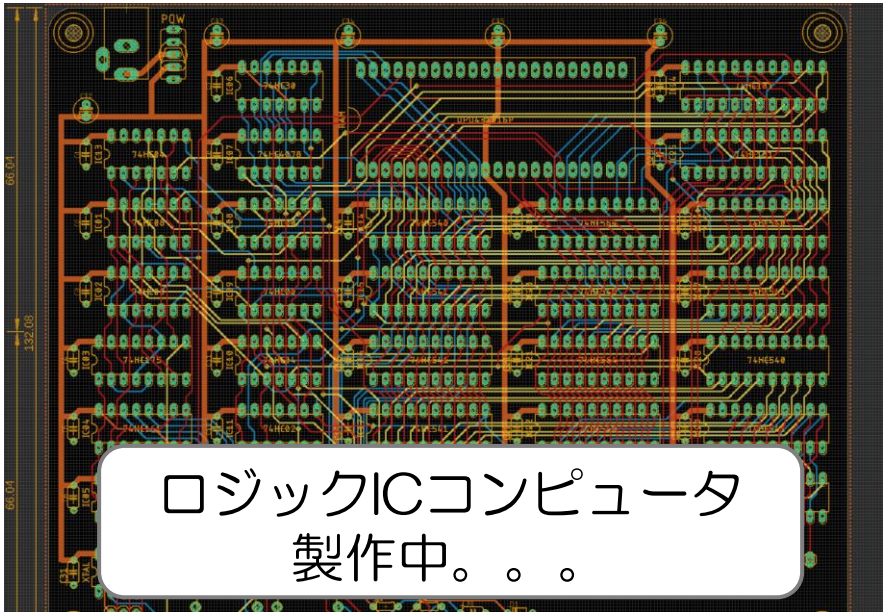
アンケート：CPUを

作ったことがある✋	作っている途中✋	作ったことがない✋

自己紹介



リレーコンピュータ
@ Maker Faire Tokyo 2022



ロジックICコンピュータ
製作中。。。

「自作CPUを語る会」開催の経緯

それぞれのCPUに、それぞれの設計思想がある

→　機械語が書けるレベルまで細かくアーキテクチャを理解したい

展示会では、そこまで語り合うのは難い

→

第１回

発表枠（発表15分+質問5分） 無料	先着順 3/3人
現地参加 無料	先着順 40/50人
オンライン聴講（試験的） 無料	参加者数 161人
現地参加（展示ブースあり） 無料	先着順 4/6人

「自作CPUを語る会」開催の経緯

それぞれのCPUに、それぞれの設計思想がある

→　機械語が書けるレベルまで細かくアーキテクチャを理解したい

展示会では、そこまで語り合うのは難い

→

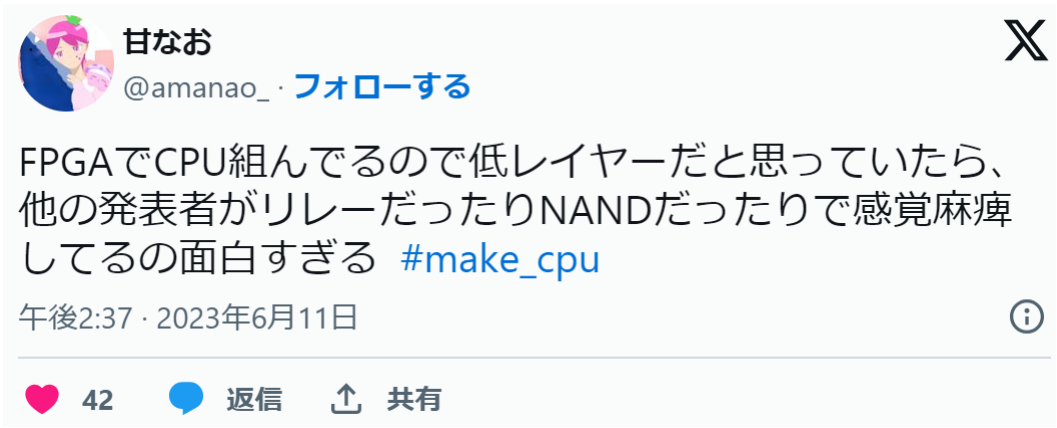
第１回

発表枠（発表15分+質問5分） 無料	先着順 3/3人
現地参加 無料	先着順 40/50人
オンライン聴講（試験的） 無料	参加者数 161人
現地参加（展示ブースあり） 無料	先着順 4/6人

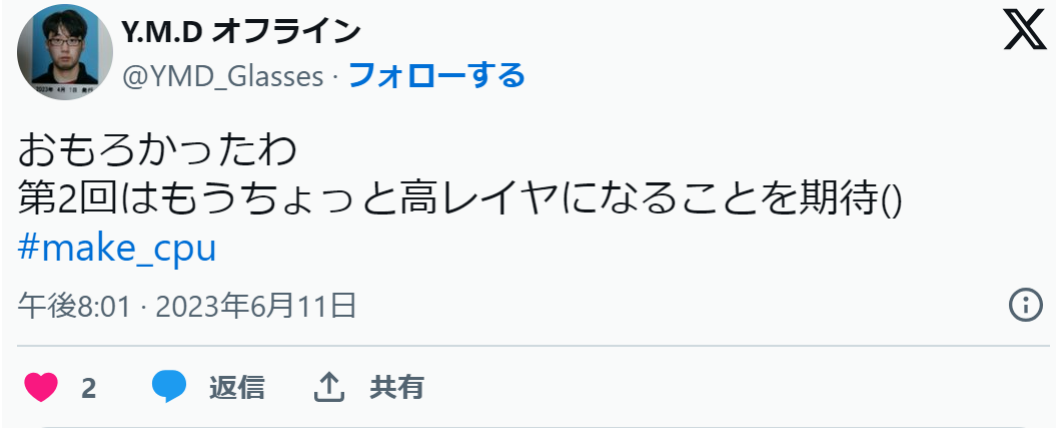
↓　あれ？減った？

現地 無料	先着順 28/50人
現地（展示ブースあり） 無料	先着順 6/8人
オンライン 無料	参加者数 66人

この会の特殊性



「FPGAが高レイヤ」



ごめんなさい
今回も低レイヤ成分が濃いです

自作CPUに対するよくある批判（妄想）

そんなの作って意味あるの？

作ってるとふと湧いてくる猜疑心でもある

うるさいな。
こっちは趣味でやってんだよ。

過去の技術と武術との対比

現代の戦で剣は使われない

剣道は無意味か？

根本的な部分は共通である

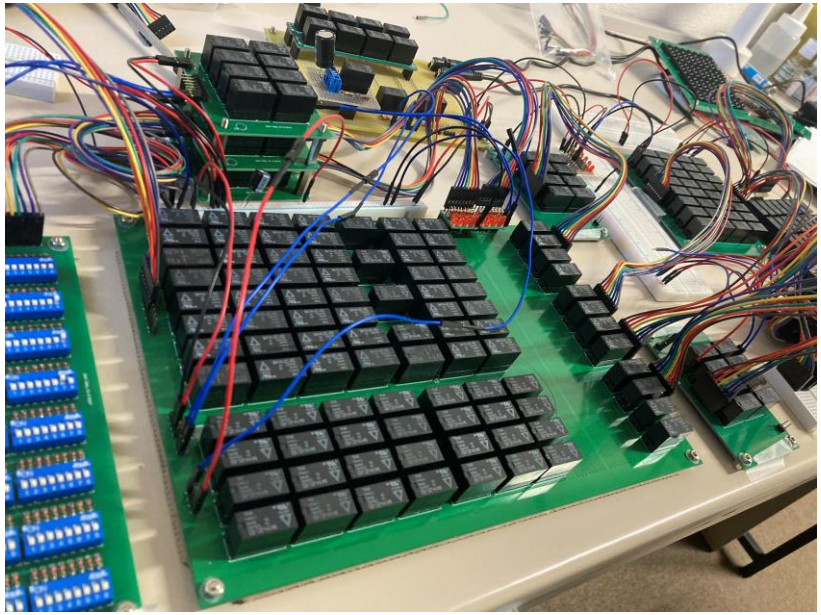
最先端の計算機はディスクリートなロジックICではできてない

過去の技術は無意味か？

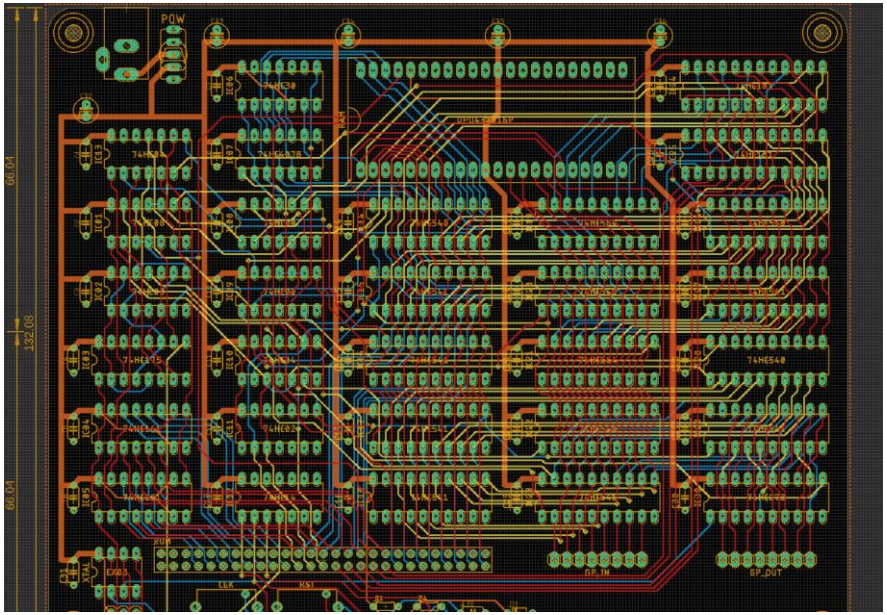
「非実用的コンピュータ科学」

非実用的コンピュータ科学

特殊で非実用的な制約条件を仮定したときに、
いかなるコンピュータシステムが良いのか？



半導体ではなくリレーで
コンピュータを作る
→リレーの再構成性を活かした
　アーキテクチャの発見



ロジックICでOSを走らせる
→何が起きる？

自作CPUに対するよくある批判（妄想）

そんなの作って意味あるの？

これは趣味でやっているのである。
性能という狭窄的な視点から
計算機を見ては面白くない！

今日はそんな「けったいな」計算機の話ができる集会です
楽しんでいってください！

ハードル上げてごめんなさい

と言いましたが…

学会でもないし、

一番気軽に発表できる場なので、気軽に発表してください～！

時刻	内容	話者
13:00 - 13:15	集合	
13:15 - 13:45	実用的ではないが実用的なコンピュータを考える	@kanade_k_1228
13:45 - 14:15	ロジックICでコンピュータ製作	@EN_gelou
14:15 - 14:30	休憩	
14:30 - 15:00	ComProcプロジェクト進捗報告：割込サポート、UARTモジュール内製化、ほか	@uchan_nos
15:00 - 15:30		@
15:30 - 17:00	交流会・LT会	

発表枠が埋まらなかった😳

コンピュータシステム全部作る

2023/12/03

@kanade_k_1228

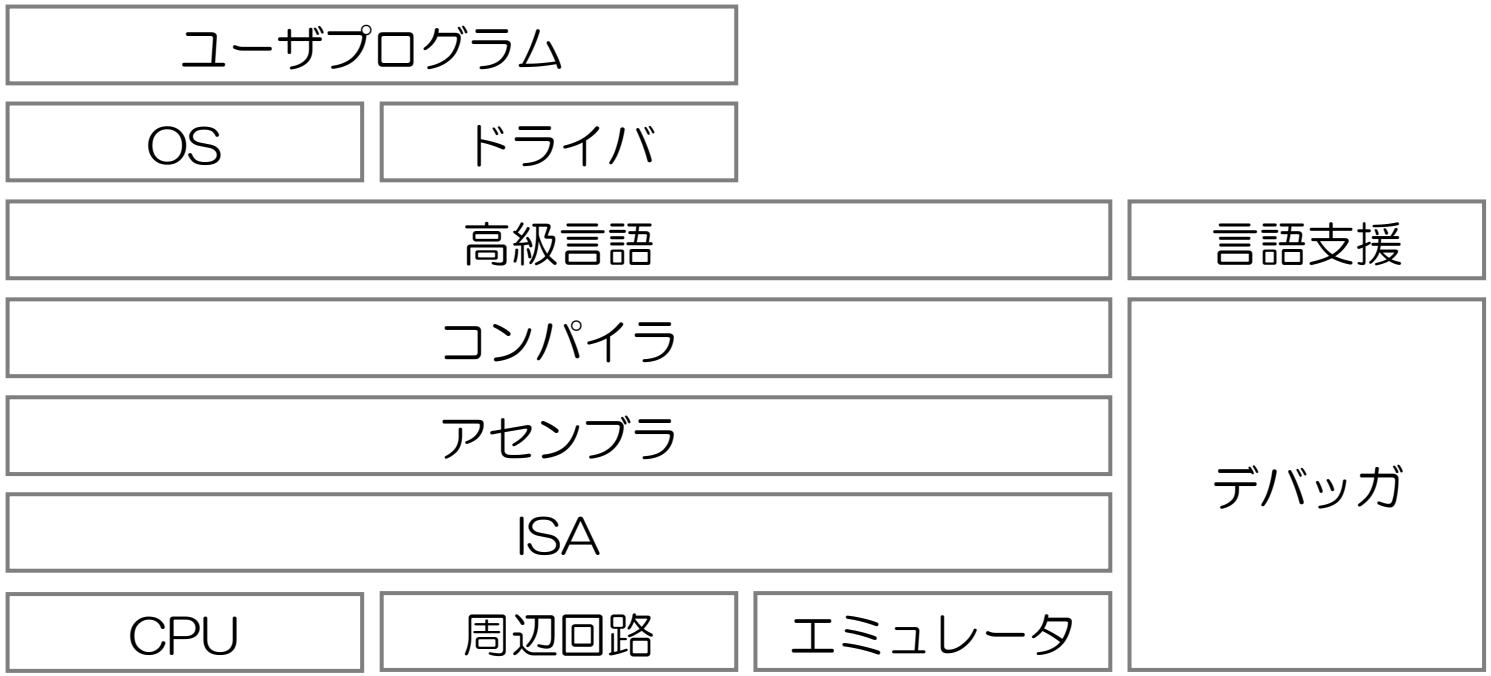
全部作って完全に理解しよう

What I cannot create,
I do not understand.

Richard Feynman
1918 - 1988

まだコンピュータを全部作っていないので
コンピュータのことを理解していないと
ファインマンに言われてしまった…

何を作りたいか

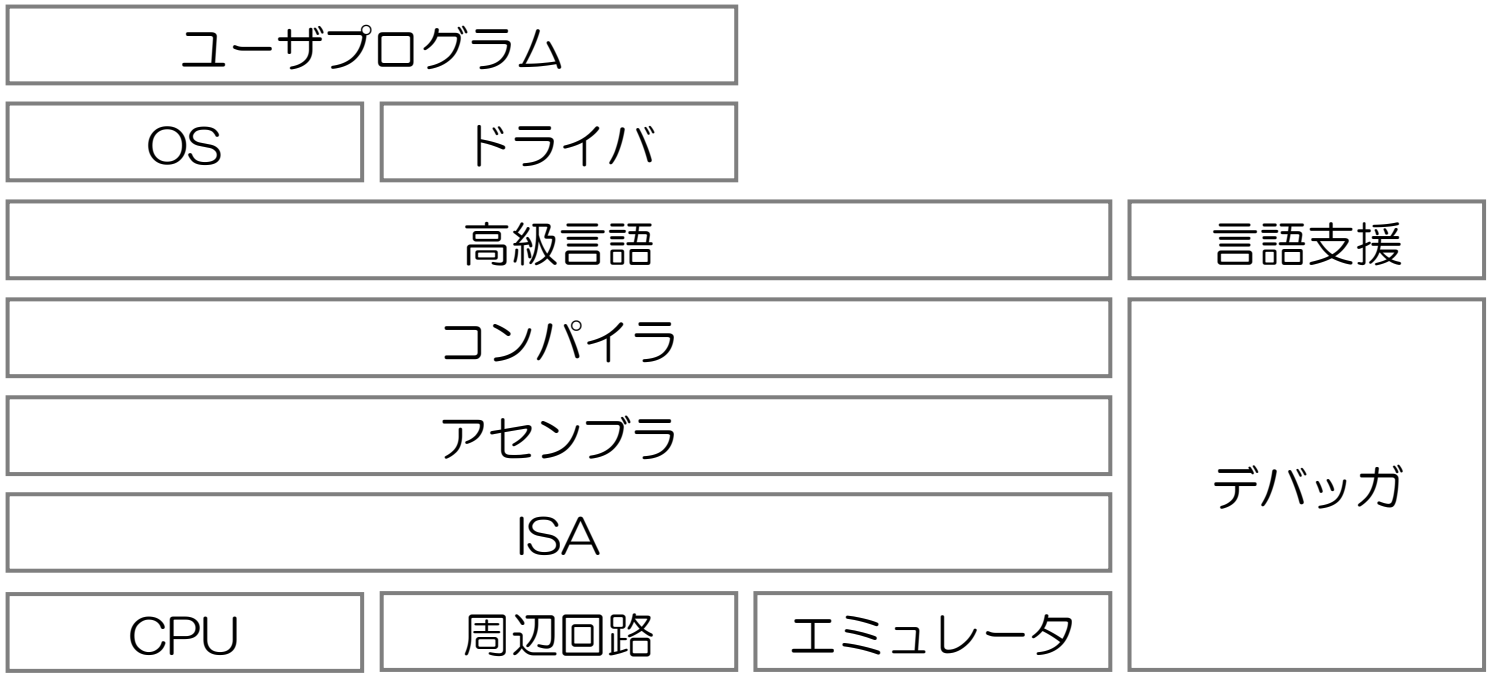


せっかく作るならロマンを求めたい

- ハードウェアは（できるだけスルーホールの）ICを組み合わせて作る
- 性能は悪くても、概念や機能は洗練されている

→　現代的な視点から過去のコンピュータを作る

開発の進め方



最初から仕様を定めることはせず、
いろいろなレイヤを開発しながら仕様を決めていく。

~~アジャイル開発~~「優柔不断開発」

開発の進め方：① ISA

①ISAを決める

- ハードウェアの実装が楽になるように
- ISAの使用者（ここではOS）にとっての UI が良くなるように

→　ISAが決まればアセンブラとエミュレータが作れる

開発の進め方：② OS

②小さなRTOSを書いてみる

- OSの肝となる「タスク管理」と「メモリ管理」をアセンブリで実装
- エミュレータ上で実行

→　割り込み仕様の詳細を詰める

開発の進め方：③ 高級言語

③高級言語を作る

- OSをアセンブリで書くのは非常に大変なのがわかった
- OSを書くのに必要な機能を持った高級言語を作る

→　ISAを変えたくなった

開発の進め方：今後

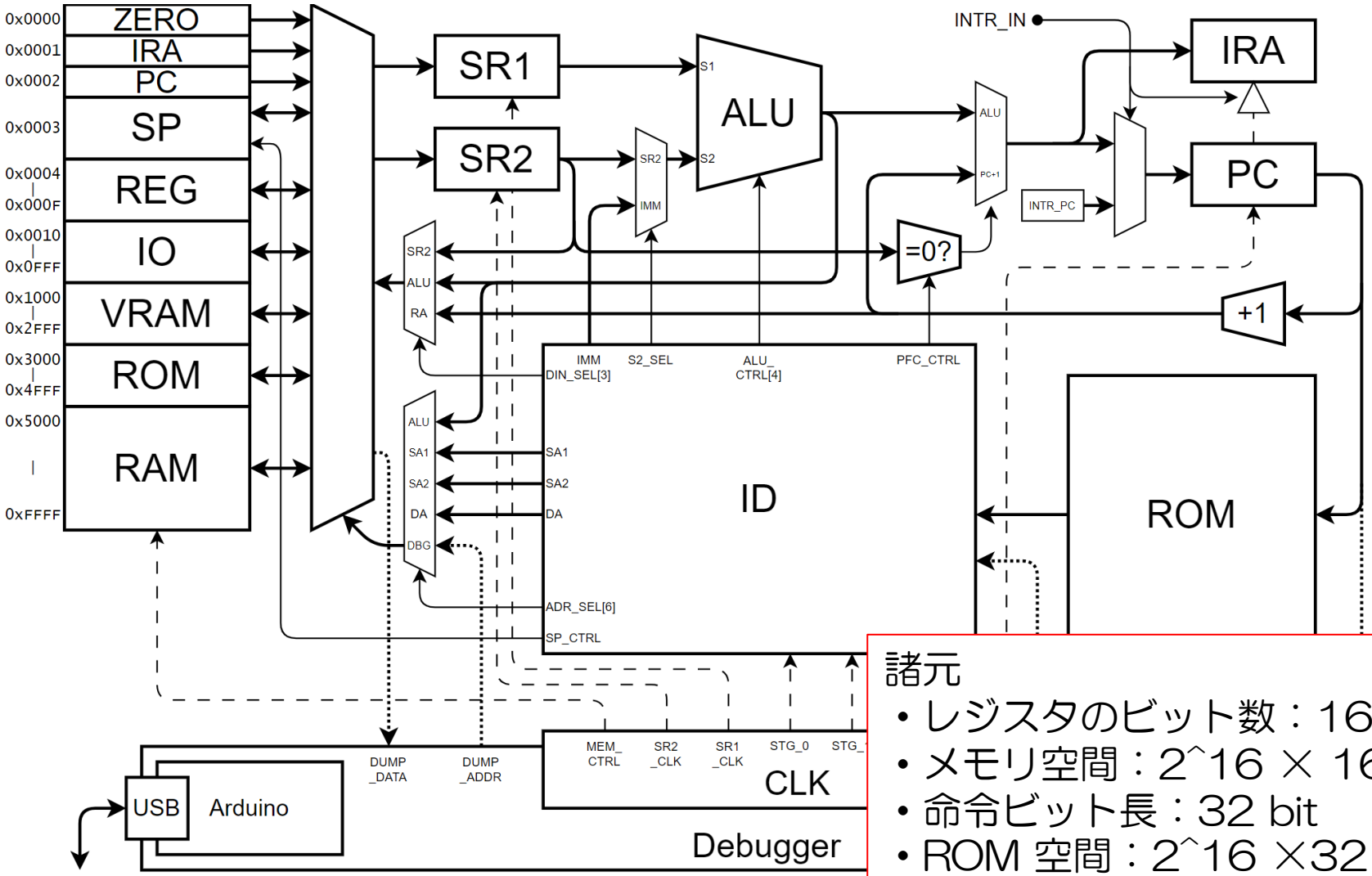
④ ISA を変え高級言語のコンパイラを作る
⑤ OS を高級言語で書いてみる
⑥ 気に入らなければ都度仕様を変更して上記のタスクを繰り返す
：
〇 高級言語の言語支援機能（Language Server）を作る
〇 OS を作りこむ

最終的に…

自作コンピュータハードウェアの上で、
自作言語で書かれた、自作OSが走る。
それらが自作の言語と開発環境で作られている。

何年かかるかな

RK16：現在のアーキテクチャ



諸元
- レジスタのビット数：16 bit
- メモリ空間：2^16 × 16 bit
- 命令ビット長：32 bit
- ROM 空間：2^16 ×32 bit

RK16：ISA

命令							31 ~ 20	19 ~ 16	15 ~ 12	11 ~ 8	7 ~ 4	3 ~ 0
	nop						0	0	0	0	0	0
レジスタ演算	add	da	sa2	sa1	-	[da] = [sa1] + [sa2]	-	func	da	sa2	sa1	calc
移値	mov	da	zero	sa1	-	[da] = [sa1]	-	add	da	zero	sa1	
即値演算	addi	da	-	sa1	imm	[da] = [sa1] + imm	imm		da	func	sa1	calci
即値ロード	loadi	da	-	zero	imm	[da] = imm			da	add	zero	
ロード	load	da	-	sa1	imm	[da] = [[sa1]+imm]	imm		da	0	sa1	load
ポップ	pop	da	-	sp	1	[da] = [[sp]+1], SP++	1		da	inc	sp	
ストア	store	-	sa2	sa1	imm	[[sa1]+imm] = [sa2]	imm		0	sa2	sa1	store
プッシュ	push	-	sa2	sp	0	[[sp]] = [sa2], SP--	0		dec	sa2	sp	
条件分岐	if	zero	sa2	sa1	imm	if([sa2]==0) PC = [sa1] + imm	imm		zero	sa2	sa1	calif
移動	jump	zero	zero	zero	imm	PC = imm	imm		zero	zero	zero	
関数呼出	call	ra	zero	zero	imm	[ra] = PC+1, PC = imm	imm		ra	zero	zero	
関数復帰	ret	zero	zero	ra	0	PC = [ra]	0		zero	zero	ra	
割込復帰	iret	zero	zero	ira	0	PC = [ira]	0		zero	zero	ira	

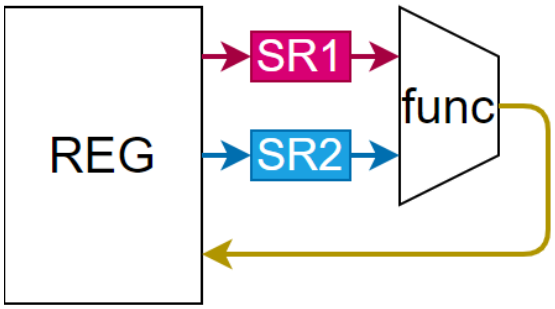
- ロジックICでの実装が簡単になるような冗長なISA
- リレーコンピュータのコンセプトを継承して機能を追加
  - LD／ST命令の追加

諸元
- レジスタのビット数：16 bit
- メモリ空間：2^16 × 16 bit
- 命令ビット長：32 bit
- ROM 空間：2^16 ×32 bit

RK16：①演算命令　②即値演算命令

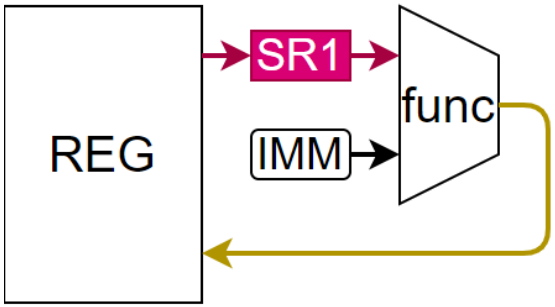
命令							31 ~ 20	19 ~ 16	15 ~ 12	11 ~ 8	7 ~ 4	3 ~ 0
	nop						0	0	0	0	0	0
レジスタ演算	add	da	sa2	sa1	-	[da] = [sa1] + [sa2]	-	func	da	sa2	sa1	calc
移値	mov	da	zero	sa1	-	[da] = [sa1]	-	add	da	zero	sa1	
即値演算	addi	da	-	sa1	imm	[da] = [sa1] + imm	imm		da	func	sa1	calci
即値ロード	loadi	da	-	zero	imm	[da] = imm			da	add	zero	

calc



①演算命令
- ３オペランド命令
- レジスタ <= F (レジスタ, レジスタ)
- funcフィールドで、演算を指定する

calci

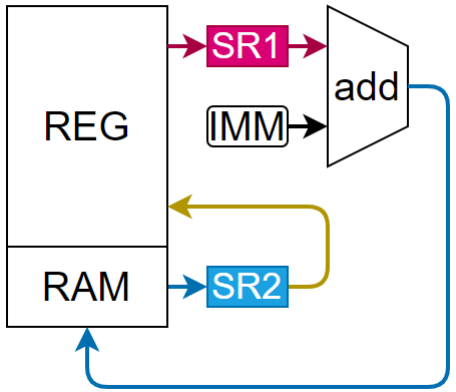


②即値演算命令
- ３オペランド命令
- レジスタ <= F (レジスタ, 即値)
- funcフィールドで、演算を指定する

RK16：③ロード命令　④ストア命令

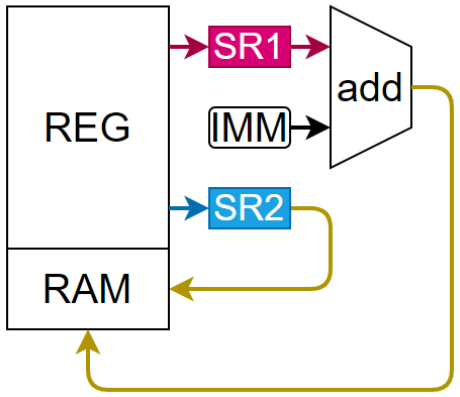
命令							31 ~ 20	19 ~ 16	15 ~ 12	11 ~ 8	7 ~ 4	3 ~ 0
ロード	load	da	-	sa1	imm	[da] = [[sa1]+imm]	imm		da	0	sa1	load
ポップ	pop	da	-	sp	1	[da] = [[sp]+1], SP++	1		da	inc	sp	
ストア	store	-	sa2	sa1	imm	[[sa1]+imm] = [sa2]	imm		0	sa2	sa1	store
プッシュ	push	-	sa2	sp	0	[[sp]] = [sa2], SP--	0		dec	sa2	sp	

load



③ロード命令
・３オペランド命令
1．レジスタと即値からアドレスを計算
2．↑のアドレスから値を読み出す
3．レジスタに書き込む

store

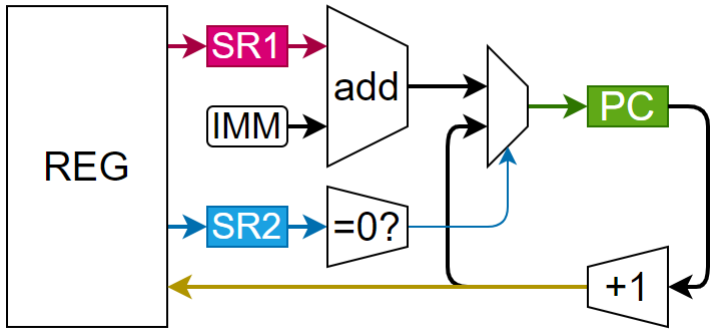


④ストア命令
・３オペランド命令
1．レジスタと即値からアドレスを計算
2．レジスタから値を読み出す
3．↑のアドレスに書き込む

RK16：⑤制御命令

命令							31 ~ 20	19 ~ 16	15 ~ 12	11 ~ 8	7 ~ 4	3 ~ 0
条件分岐	if	zero	sa2	sa1	imm	if([sa2]==0) PC = [sa1] + imm	imm		zero	sa2	sa1	calif
移動	jump	zero	zero	zero	imm	PC = imm	imm		zero	zero	zero	
関数呼出	call	ra	zero	zero	imm	[ra] = PC+1, PC = imm	imm		ra	zero	zero	
関数復帰	ret	zero	zero	ra	0	PC = [ra]	0		zero	zero	ra	
割込復帰	iret	zero	zero	ira	0	PC = [ira]	0		zero	zero	ira	

calif



⑤制御命令
・４オペランド命令

1．ジャンプ先アドレスを計算
2．レジスタの値で条件分岐
3．戻りアドレスをレジスタに保存

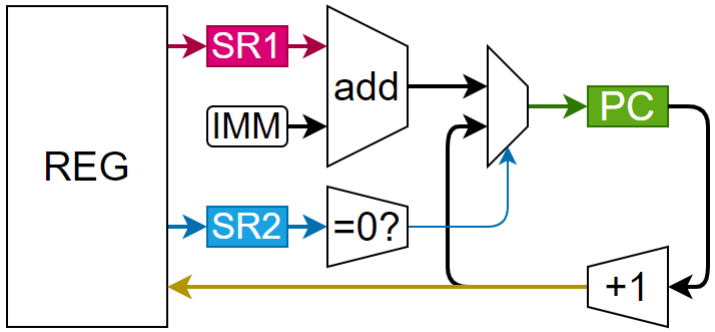
→　制御命令（分岐・ジャンプ・関数）を１命令で実現

が、この命令は疑似命令を介してしか使用できない

RK16：⑤疑似命令

命令							31 ~ 20	19 ~ 16	15 ~ 12	11 ~ 8	7 ~ 4	3 ~ 0
条件分岐	if	zero	sa2	sa1	imm	if([sa2]==0) PC = [sa1] + imm	imm		zero	sa2	sa1	calif
移動	jump	zero	zero	zero	imm	PC = imm	imm		zero	zero	zero	
関数呼出	call	ra	zero	zero	imm	[ra] = PC+1, PC = imm	imm		ra	zero	zero	
関数復帰	ret	zero	zero	ra	0	PC = [ra]	0		zero	zero	ra	
割込復帰	iret	zero	zero	ira	0	PC = [ira]	0		zero	zero	ira	

calif



①条件分岐
戻りアドレスは使用しないので
zeroレジスタに捨てる

②ジャンプ
条件文をTaken（zero）にするだけ

②関数呼び出し
戻りアドレスをRAレジスタに保存

③リターン
PCに戻りアドレスをセットする

読みやすいアセンブリコードになる！

RK16：わかりやすいアセンブリ言語

アセンブリ言語は機械語と一対一の言語
→　その中で、いかにプログラマの認知負荷を下げられるか？

```
@0x0200 hoge
@0x0123 fuga
@0x4567 piyo
@0x89AB foo
@0xcdef bar
@0x0010 exit
```

```
#0x0100 const
#0x0030 '0'
#0x0031 '1'
#0x0032 '2'
#0x0033 '3'
#0x0034 '4'
#0x0035 '5'
```

変数ラベル
　メモリのアドレスに
　名前を付ける

定数ラベル
　定数を #define する

```
 18 |           |           | calc_test:
 19 | 0000      | 0000_0000 |     nop   ; no operation
 20 | 0001      | 0000_c0d0 |     mov   s0    s1    ; s0 <= s1
 21 | 0002      | 0000_a980 |     add   t2    t0    t1    ; t2 <= t
 22 | 0003      | 0007_a980 |     sub   t2    t0    t1
 23 | 0004      | 0004_a980 |     and   t2    t0    t1
 24 | 0005      | 0006_a980 |     or    t2    t0    t1
 25 | 0006      | 0005_a980 |     xor   t2    t0    t1
 26 | 0007      | 0001_a080 |     not   t2    t0
 27 | 0008      | 000d_a080 |     srs   t2    t0
 28 | 0009      | 000c_a080 |     sru   t2    t0
 29 | 000a      | 0002_a080 |     sl    t2    t0
 30 | 000b      | 0008_a980 |     eq    t2    t0    t1
 31 | 000c      | 000b_a980 |     lts   t2    t0    t1
 32 | 000d      | 000a_a980 |     ltu   t2    t0    t1
 33 |           |           | calci_test:
 34 | 000e      | 0100_a081 |     addi  t2    t0    0100 = const
 35 | 000f      | 0200_a781 |     subi  t2    t0    0200 = hoge
 36 | 0010      | 0123_a481 |     andi  t2    t0    0123 = fuga
 37 | 0011      | 0000_a681 |     ori   t2    t0    0000
 38 | 0012      | 0110_a581 |     xori  t2    t0    0110
 39 | 0013      | 1100_a881 |     eqi   t2    t0    1100
 40 | 0014      | f000_ab81 |     ltsi  t2    t0    f000
 41 | 0015      | f000_aa81 |     ltui  t2    t0    f000
 42 |           |           |
 43 |           |           | a:
 44 | 0016      | 001d_000f |     jump              001d : e
 45 | 0017      | 0001_002f |     jumpr             0001
 46 | 0018      | 001a_000f |     if    zero        001a : b
 47 | 0019      | 001a_020f |     ifr   zero        001a : b
 48 |           |           | b:
 49 | 001a      | 001b_400f |     call              001b : c
 50 |           |           | c:
 51 | 001b      | 0000_004f |     ret
 52 |           |           | d:
 53 | 001c      | 0000_001f |     iret
 54 |           |           | e:
 55 | 001d      | 0001_8001 |     loadi t0          0001
 56 | 001e      | 0010_0807 |     store t0    zero  0010 = exit
```

リッチなCLIのアセンブラ

ありがとうございました！