CybozuLabs

# ComProcアーキテクチャ解説

2023年6月11日　第1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

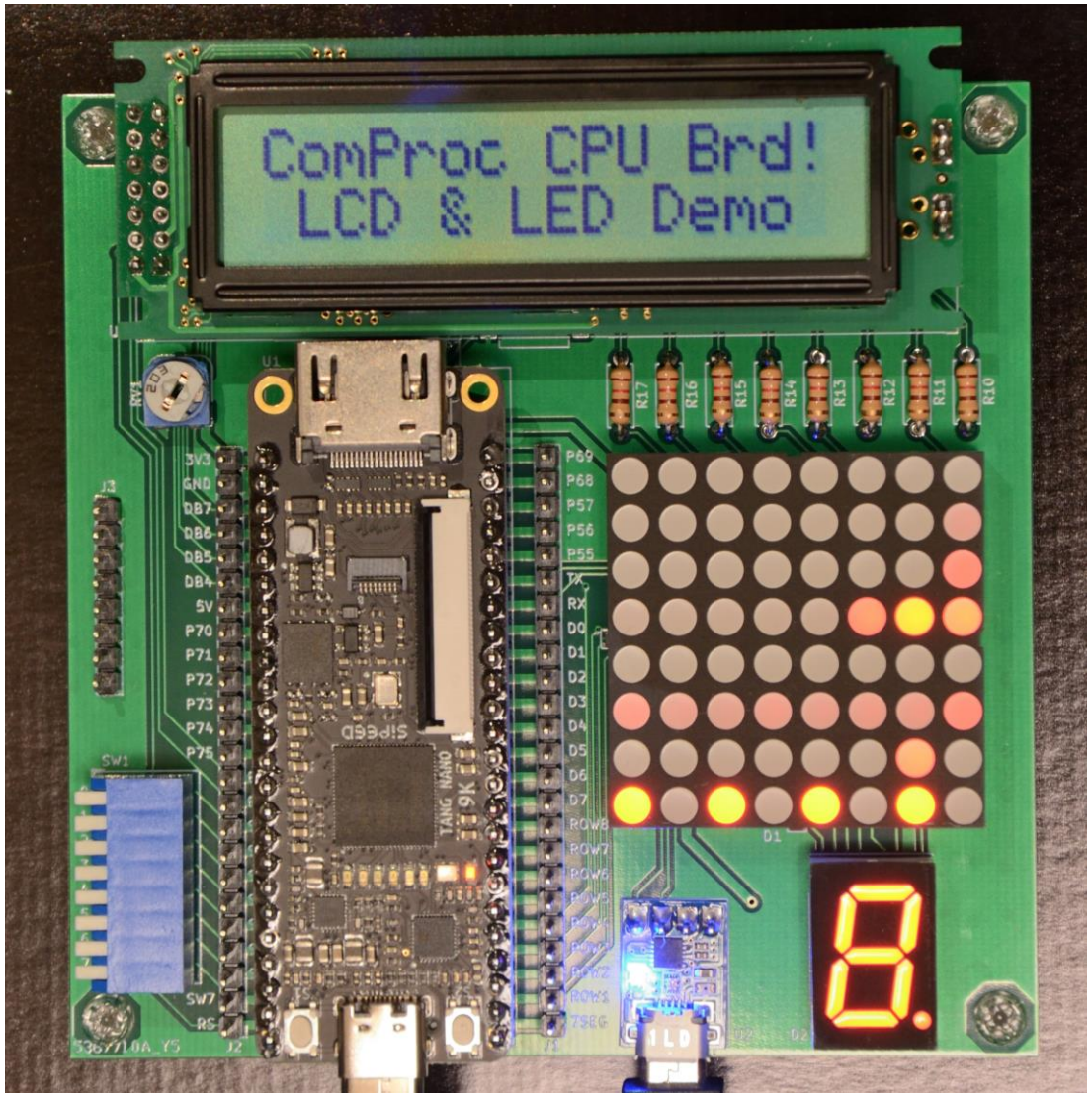


- ComProc=Compiler+Processor
- CPUとコンパイラを自作する、uchan主導のプロジェクト
  - CPUとコンパイラを作るプロジェクトは珍しくない
  - ComProcプロジェクトは**CPUとコンパイラを同時並行に進化**させる点で、他のプロジェクトとは一線を画す

- 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 CPU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 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
- 部品代（FPGA込み）は約5K円

- FPGAで自作CPU
  ＝超・低レイヤ！

# と思っていたけど、FPGAは高レイヤ！？

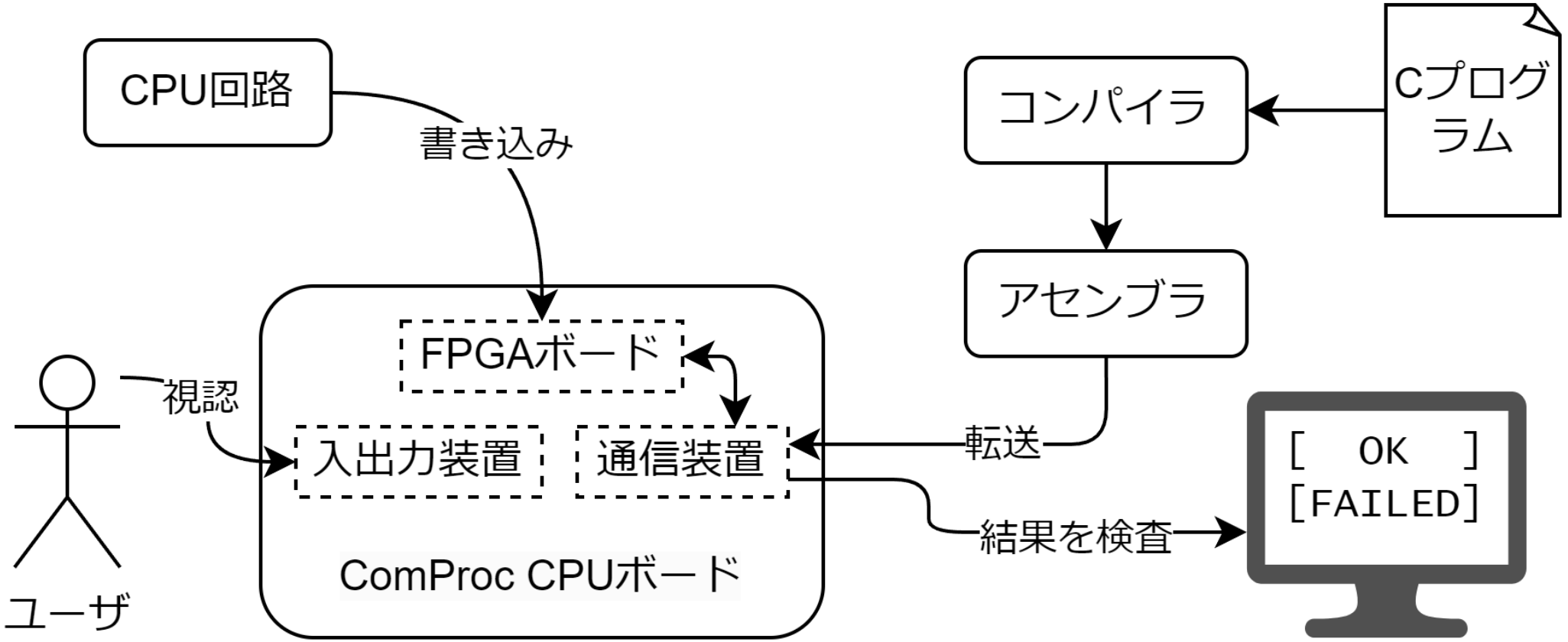
CybozuLabs

## タイムテーブル（仮）

| 時刻 | 内容 | 話者 | ライブ配信 |
|---|---|---|---|
| 13:00 - 13:10 | 集合 | | |
| 13:10 - 13:30 | リレーでコンピュータを作った話 | @kanade_k_1228 | ○ |
| 13:30 - 13:50 | NANDによるプロセッサ開発 | @cherry_takuan | ○ |
| 13:50 - 14:10 | リレーコンピュータを 作り始めるまで | @Kuon_Aoto | ○ |
| 14:10 - 14:30 | ComProcアーキテクチャ解説 | @uchan_nos | ○ |
| 14:30 - 17:00 | 交流会・LT会 | | × |

# ComProcプロジェクトの全体像





# 目次



- ComProcプロジェクトの概要
- CPUとコンパイラの同時並行進化とは
- ComProcの命令セット
- CPUのデータパス
- コンパイラへの最適化機能の追加

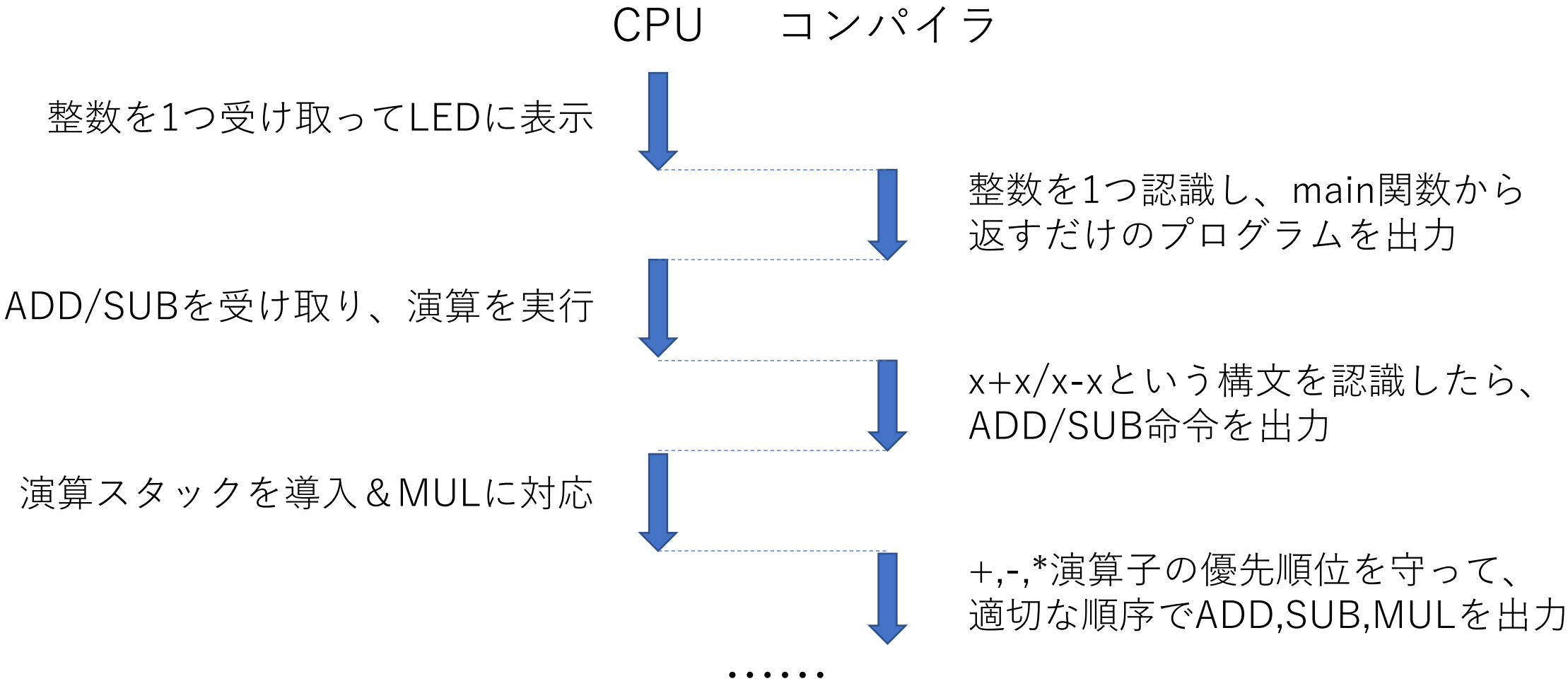
# 目次



- ComProcプロジェクトの概要
- CPUとコンパイラの同時並行進化とは
- ComProcの命令セット
- CPUのデータパス
- コンパイラへの最適化機能の追加

# CPUとコンパイラの同時並行進化とは






同時並行進化：CPUとコンパイラを、歩調を合わせて進化させていく

# 同時並行進化の特長



- 余計な命令を実装しなくて済む
  - コンパイラが使う命令だけを追加すればよい
- ずっとエンドツーエンドで動く状態を保てる
  - 「C言語コードをコンパイルしCPUで動く」状態をずっと維持できる
  - モチベーションを維持しやすい！
- CPUとコンパイラの役割分担を調整しやすい
  - CPUにこの仕組みがあるとコンパイラが書きやすい
  - この機能はコンパイラに持たせた方がCPUが楽できる
  - など調整しやすい

# CPUとコンパイラの役割分担



■コンパイラ開発を楽にする例

●CPUに演算スタックを用意
  ■コンパイラでのレジスタ割り付けを不要に

●演算スタックと別にコールスタックを用意
  ■引数の受け渡しが簡単に
  ■演算スタックが整合してなくてもRETが暴走しない

■ CPU開発を楽にする例

●条件分岐はスタックトップが0か1で判定
  ■下位1ビットだけ見る
  ■コンパイラは必要に応じ、0/非0を0/1に変換する命令を発行する

# 目次



- ComProcプロジェクトの概要
- CPUとコンパイラの同時並行進化とは
- ComProcの命令セット
- CPUのデータパス
- コンパイラへの最適化機能の追加

# 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]
ADD FP,simm10    |011000  simm10  | fp += simm10
                 |011001xxxxxxxxxx| 予約
                 |01101xxxxxxxxxx| 予約
                 |0111xxxxxxxxxx| 即値なし命令（別表）
```

# 命令セット（即値なし命令）

CybozuLabs

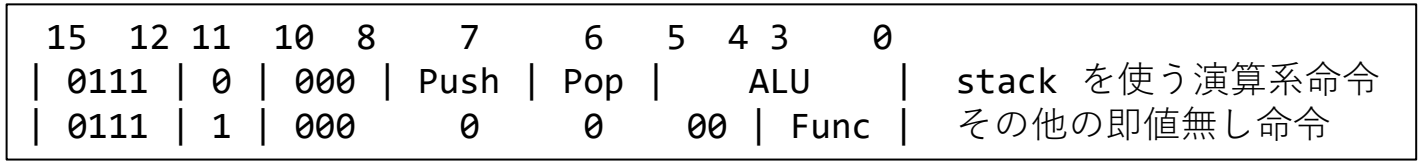
```
mnemonic    15     87      0  説明
-----------------------------------
NOP         |0111000000000000| stack[0] に ALU-A をロードするので、ALU=00h
POP         |0111000001001111| stack をポップ
                               stack[0] に ALU-B をロードするので、ALU=0fh
POP 1       |0111000001000000| stack[1] 以降をポップ（stack[0] 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
```

## 即値なし命令のビット構造



# 1+2*3を実行する様子



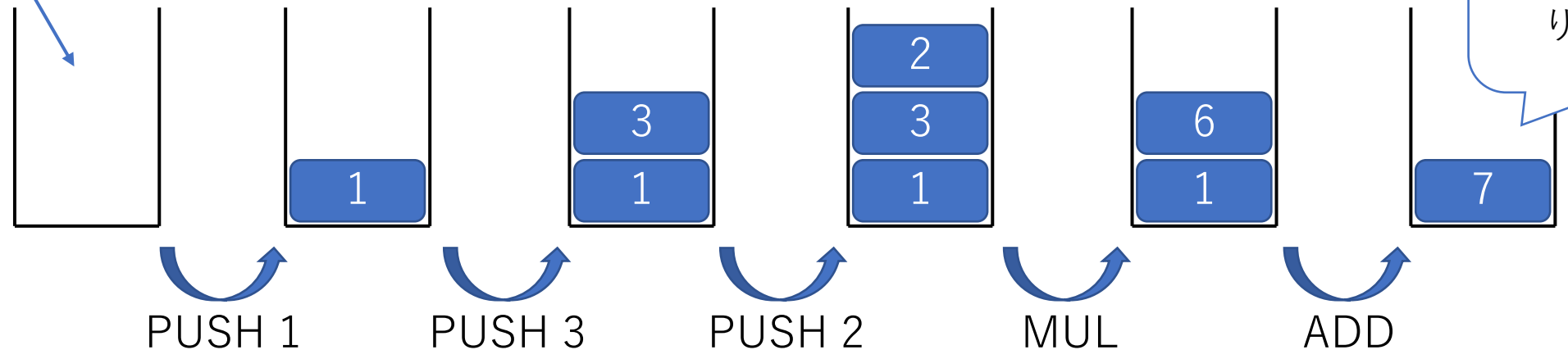
C言語コード

```
int main() {
  return 1 + 2 * 3;
}
```

コンパイル
＆アセンブル

```
main:
        cpush fp
        push 1
        push 3
        push 2
        mul
        add
        cpop fp
        ret
```

演算スタック



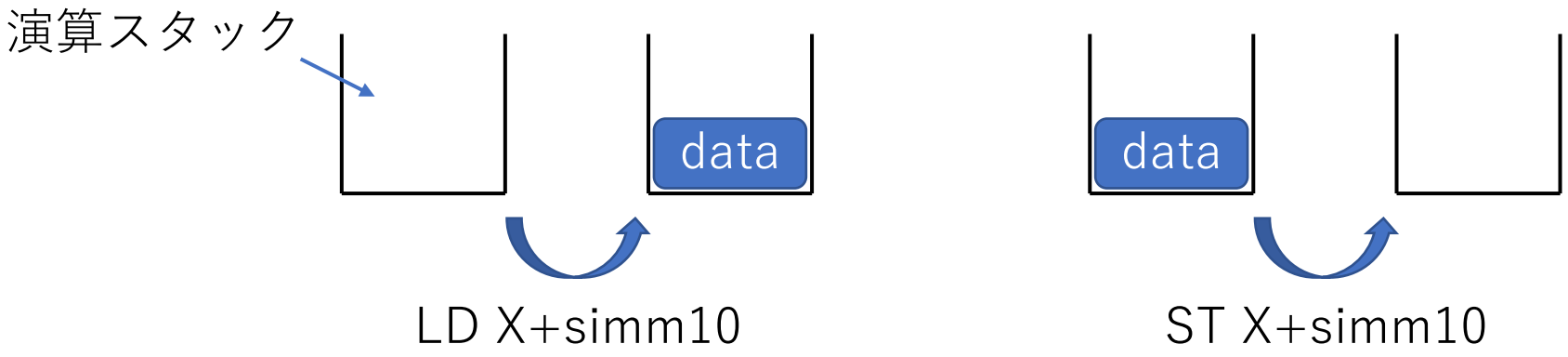

スタックトップの値が関数の戻り値となる

# ロード・ストア命令（即値あり）



- メモリ読み書きは専用の命令を使う
- LD X+simm10
  - 指定したアドレスから1ワード読み込む（LoaD）
  - 例：`LD cstack+2`
- ST X+simm10
  - 指定したアドレスにstack[0]を書き込む（STore）
- LD.1、ST.1はバイトバージョン

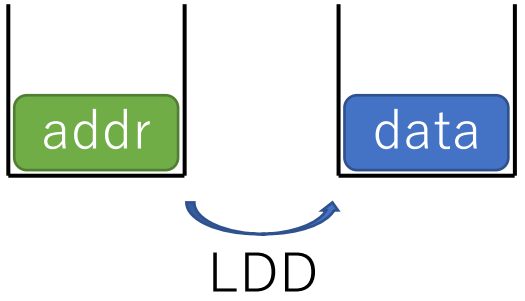
| X | X | ベース値 |
|---|---|---|
| 無表記 | 0 | 0 |
| fp | 1 | FP |
| ip | 2 | IP |
| cstack | 3 | cstack[0] |



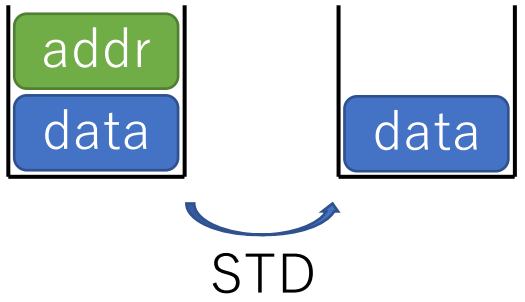
# ロード・ストア命令（即値なし）



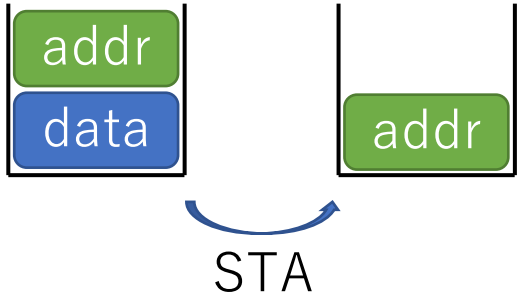
- LDD（LoaD Data）
  - stack[0]で指定したアドレスから1ワード読み込む




- STD（STore Data）
  - stack[0]で指定したアドレスにstack[1]を書き込む
  - スタックにはデータを残す




- STA（STore Address）
  - stack[0]で指定したアドレスにstack[1]を書き込む
  - スタックにはアドレスを残す



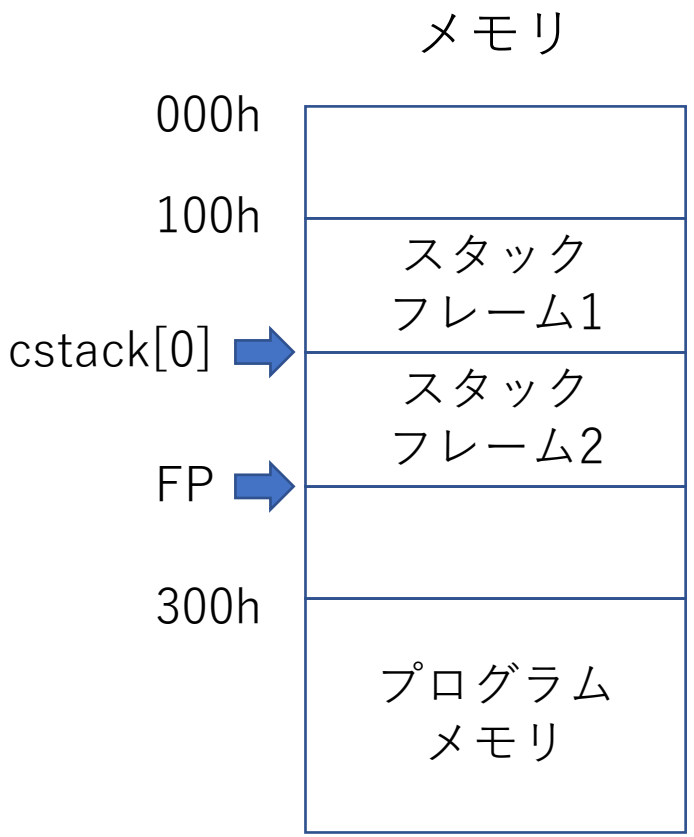
# スタックフレーム



- ローカル変数を格納する領域
- FPがスタックフレーム末尾を指す

- FP操作命令
  - `ADD FP, simm10`でFPを増減
  - `CPUSH FP`、`CPOP FP`で保存・復帰

- CALL/RETの戻り先アドレス
  - RISC-V、Arm、x86等はスタックフレームに戻り先アドレスを保存
  - ComProc CPUはPICと同様、専用のコールスタックを使う

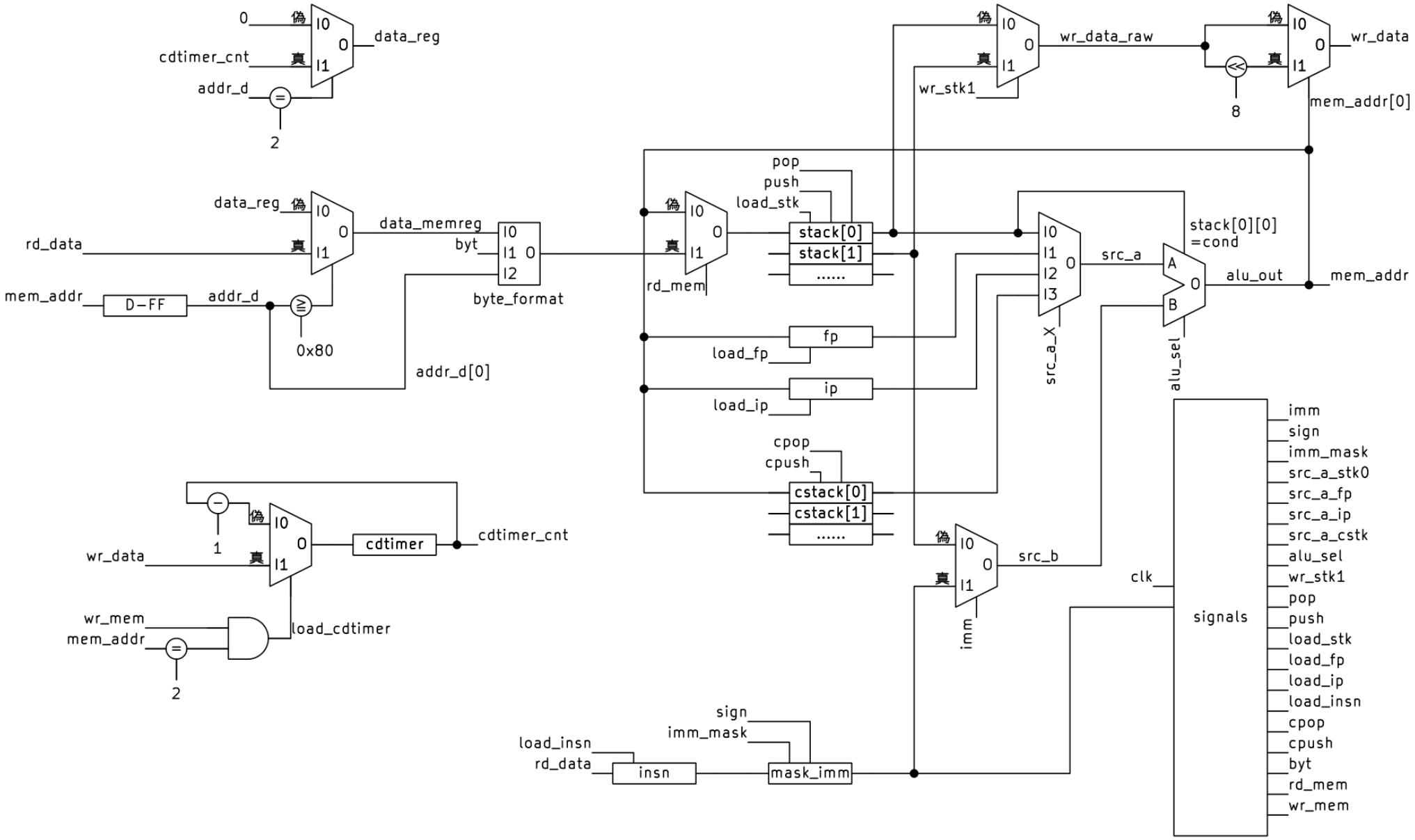


# 目次

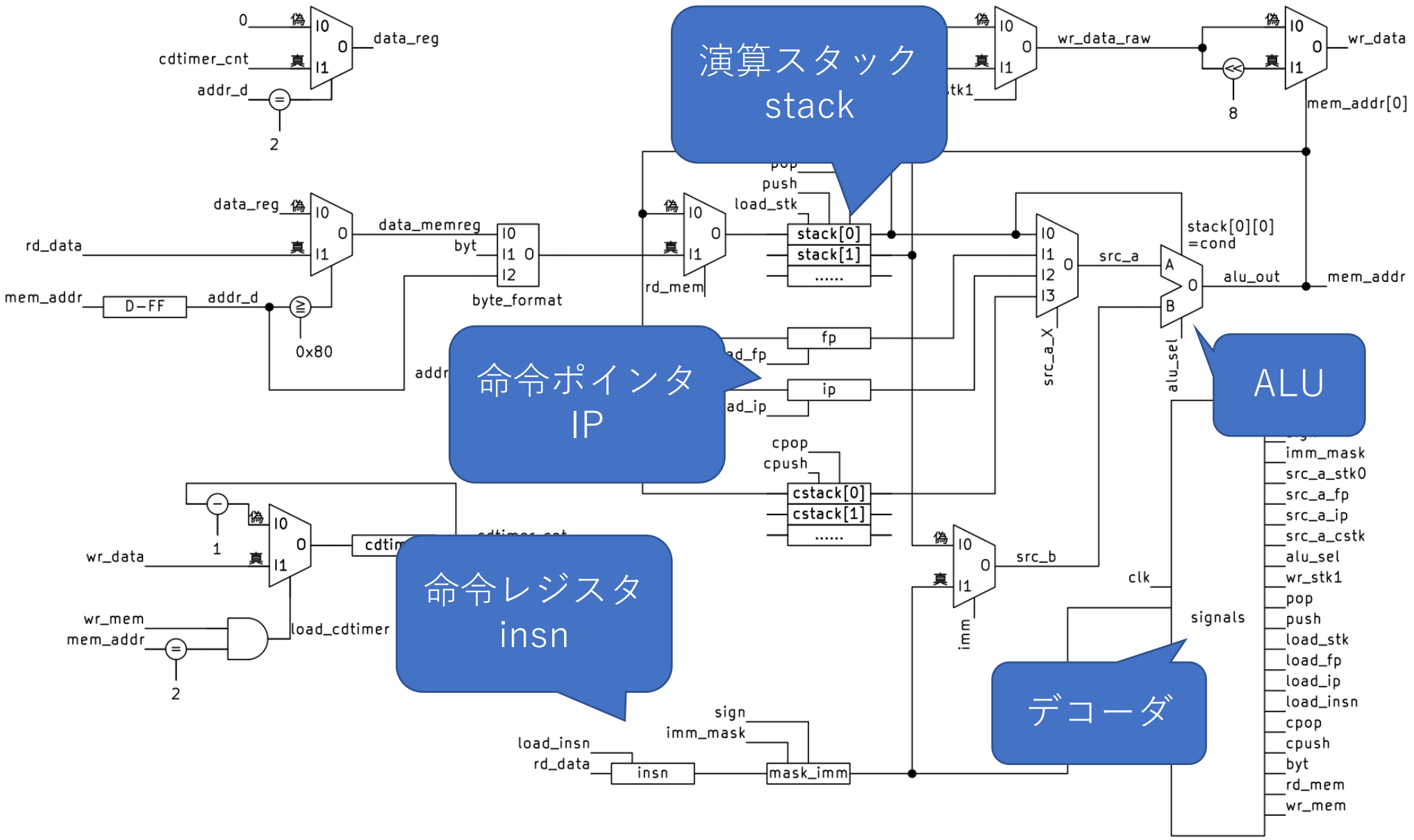


- ComProcプロジェクトの概要
- CPUとコンパイラの同時並行進化とは
- ComProcの命令セット
- CPUのデータパス
- コンパイラへの最適化機能の追加

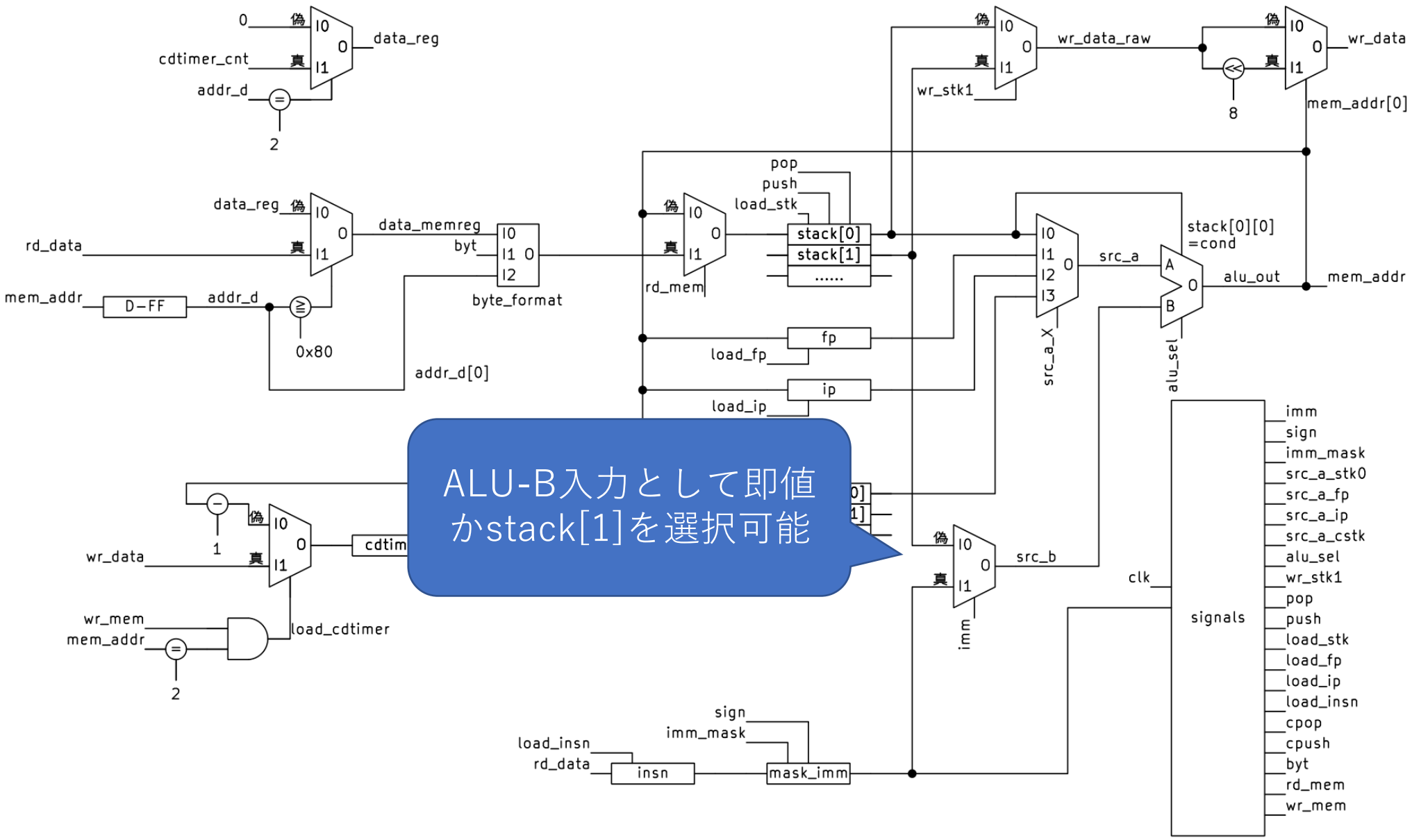
# データパス図


0
偽
I0
O
data_reg
cdtimer_cnt
真
I1
addr_d
=
2
data_reg
偽
I0
O
data_memreg
rd_data
真
I1
byt
I0
I1
O
I2
mem_addr
D−FF
addr_d
≧
0x80
byte_format
addr_d[0]
偽
I0
O
真
I1
rd_mem
pop
push
load_stk
stack[0]
stack[1]
.....
fp
load_fp
ip
load_ip
cpop
cpush
cstack[0]
cstack[1]
.....
偽
I0
O
真
I1
wr_stk1
wr_data_raw
偽
I0
O
真
I1
≪
8
wr_data
mem_addr[0]
I0
I1
I2
I3
O
src_a
src_a_X
stack[0][0]
=cond
A
O
B
alu_sel
alu_out
mem_addr
偽
I0
O
真
I1
imm
src_b
clk
signals
imm
sign
imm_mask
src_a_stk0
src_a_fp
src_a_ip
src_a_cstk
alu_sel
wr_stk1
pop
push
load_stk
load_fp
load_ip
load_insn
cpop
cpush
byt
rd_mem
wr_mem
−
1
偽
I0
O
真
I1
wr_data
cdtimer
cdtimer_cnt
wr_mem
mem_addr
=
2
load_cdtimer
sign
imm_mask
load_insn
rd_data
insn
mask_imm

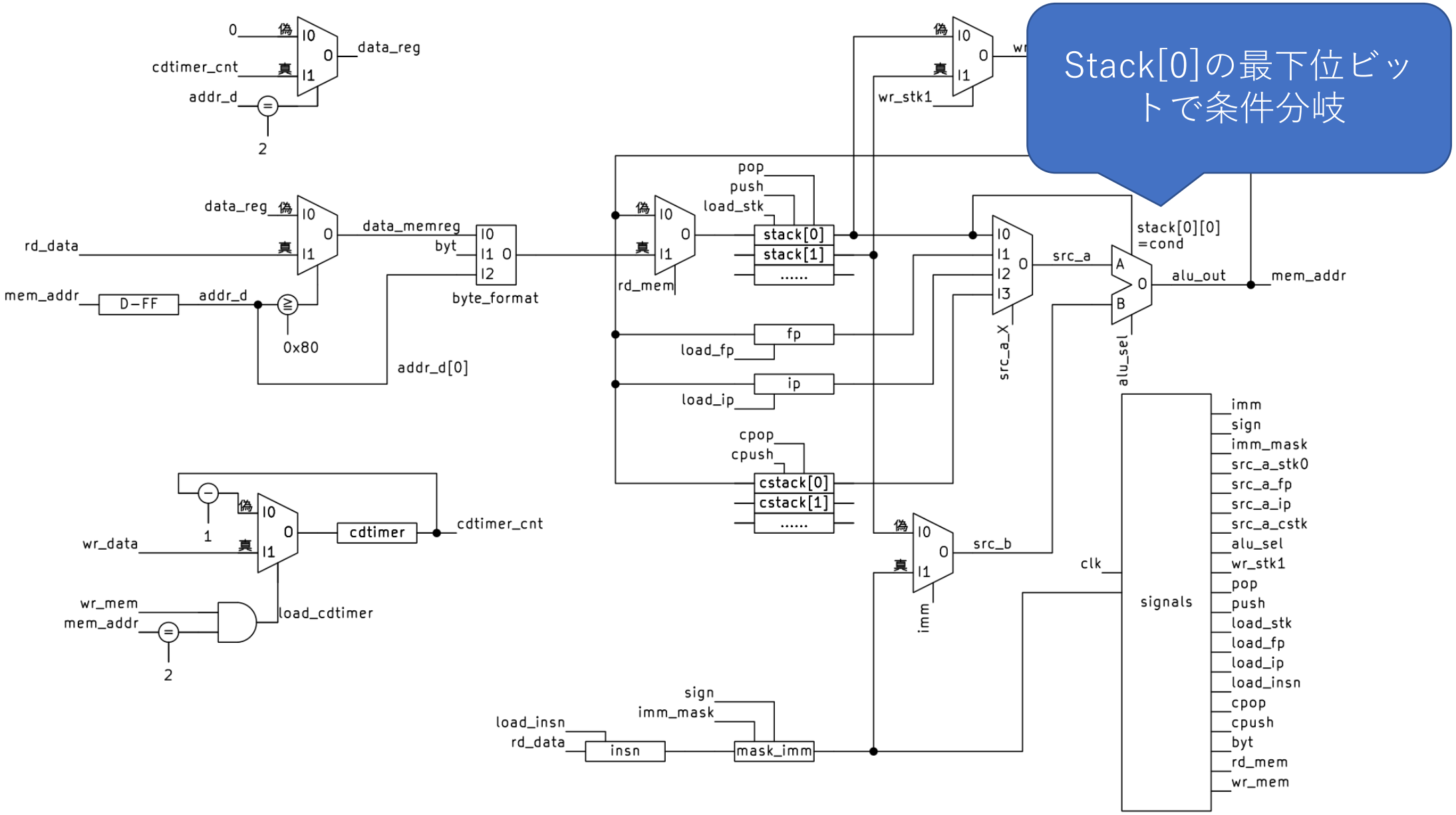
# データパス図


演算スタック
stack
命令ポインタ
IP
命令レジスタ
insn
ALU
デコーダ
data_reg
cdtimer_cnt
addr_d
wr_data_raw
wr_data
mem_addr[0]
mem_addr
rd_data
D-FF
data_memreg
byt
byte_format
rd_mem
push
load_stk
stack[0]
stack[1]
stack[0][0]
=cond
src_a
alu_out
fp
ip
cpop
cpush
cstack[0]
cstack[1]
src_b
imm
wr_mem
load_cdtimer
load_insn
sign
imm_mask
insn
mask_imm
clk
signals
imm_mask
src_a_stk0
src_a_fp
src_a_ip
src_a_cstk
alu_sel
wr_stk1
pop
push
load_stk
load_fp
load_ip
load_insn
cpop
cpush
byt
rd_mem
wr_mem

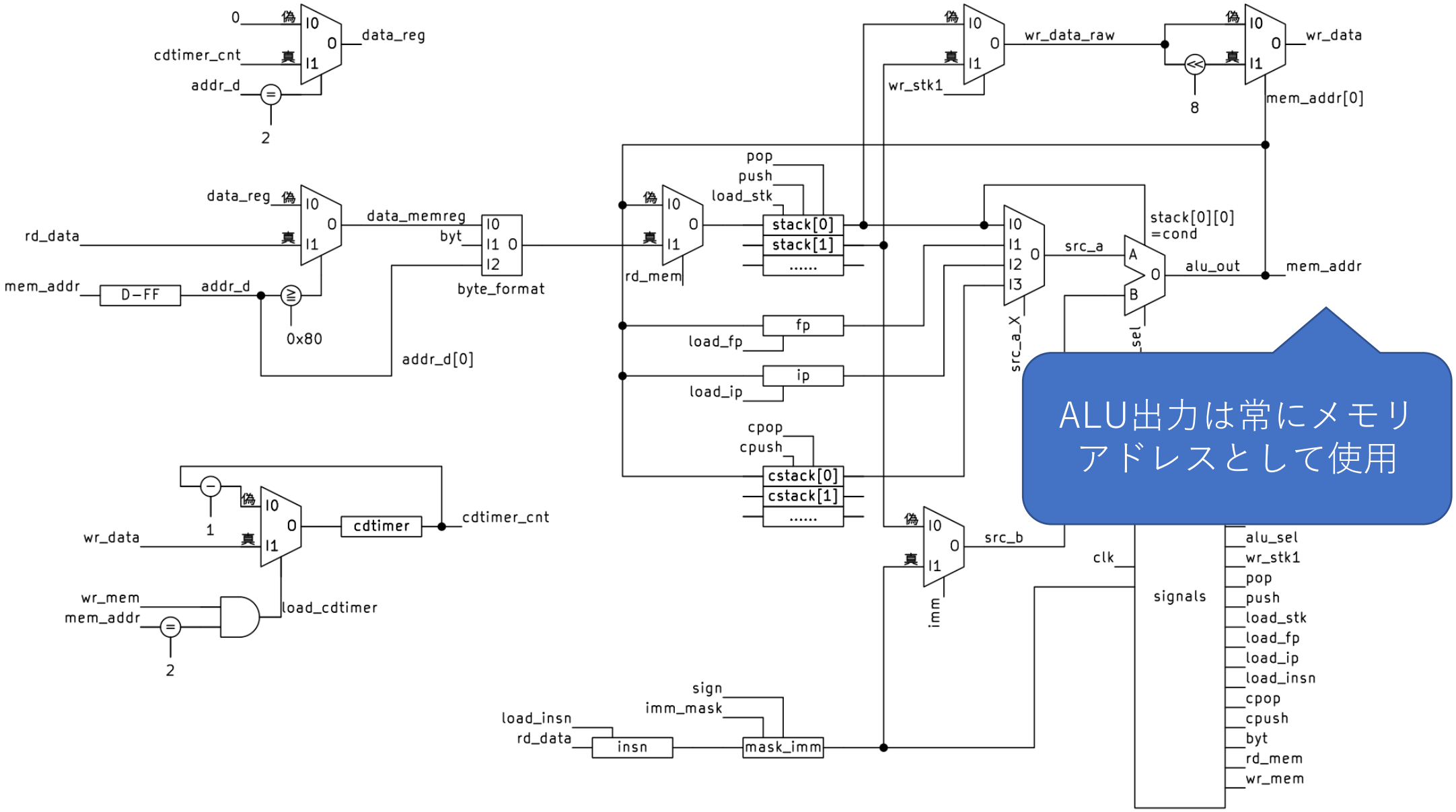
# データパス図


0
偽
I0
O
data_reg
cdtimer_cnt
真
I1
addr_d
=
2
data_reg
偽
I0
O
data_memreg
rd_data
真
I1
byt
I0
I1
O
I2
mem_addr
D−FF
addr_d
≧
0x80
byte_format
addr_d[0]
偽
I0
O
真
I1
rd_mem
pop
push
load_stk
stack[0]
stack[1]
.....
偽
I0
O
真
I1
wr_data_raw
wr_stk1
偽
I0
O
真
I1
wr_data
<<
8
mem_addr[0]
stack[0][0]
=cond
I0
I1
I2
I3
O
src_a
A
B
O
alu_out
mem_addr
src_a_X
alu_sel
fp
load_fp
ip
load_ip
ALU-B入力として即値
かstack[1]を選択可能
−
1
偽
I0
O
wr_data
真
I1
cdtim
wr_mem
mem_addr
=
2
load_cdtimer
偽
I0
O
真
I1
src_b
imm
clk
signals
imm
sign
imm_mask
src_a_stk0
src_a_fp
src_a_ip
src_a_cstk
alu_sel
wr_stk1
pop
push
load_stk
load_fp
load_ip
load_insn
cpop
cpush
byt
rd_mem
wr_mem
sign
imm_mask
load_insn
rd_data
insn
mask_imm

# データパス図


Stack[0]の最下位ビットで条件分岐
0
偽
真
data_reg
cdtimer_cnt
addr_d
=
2
data_reg
rd_data
mem_addr
D-FF
addr_d
≧
0x80
data_memreg
byt
byte_format
addr_d[0]
wr_data
1
cdtimer
cdtimer_cnt
wr_mem
mem_addr
load_cdtimer
pop
push
load_stk
stack[0]
stack[1]
.....
rd_mem
wr_stk1
fp
load_fp
ip
load_ip
cpop
cpush
cstack[0]
cstack[1]
.....
src_a
src_a_X
stack[0][0]
=cond
alu_sel
alu_out
mem_addr
src_b
imm
clk
signals
imm
sign
imm_mask
src_a_stk0
src_a_fp
src_a_ip
src_a_cstk
alu_sel
wr_stk1
pop
push
load_stk
load_fp
load_ip
load_insn
cpop
cpush
byt
rd_mem
wr_mem
load_insn
rd_data
insn
sign
imm_mask
mask_imm

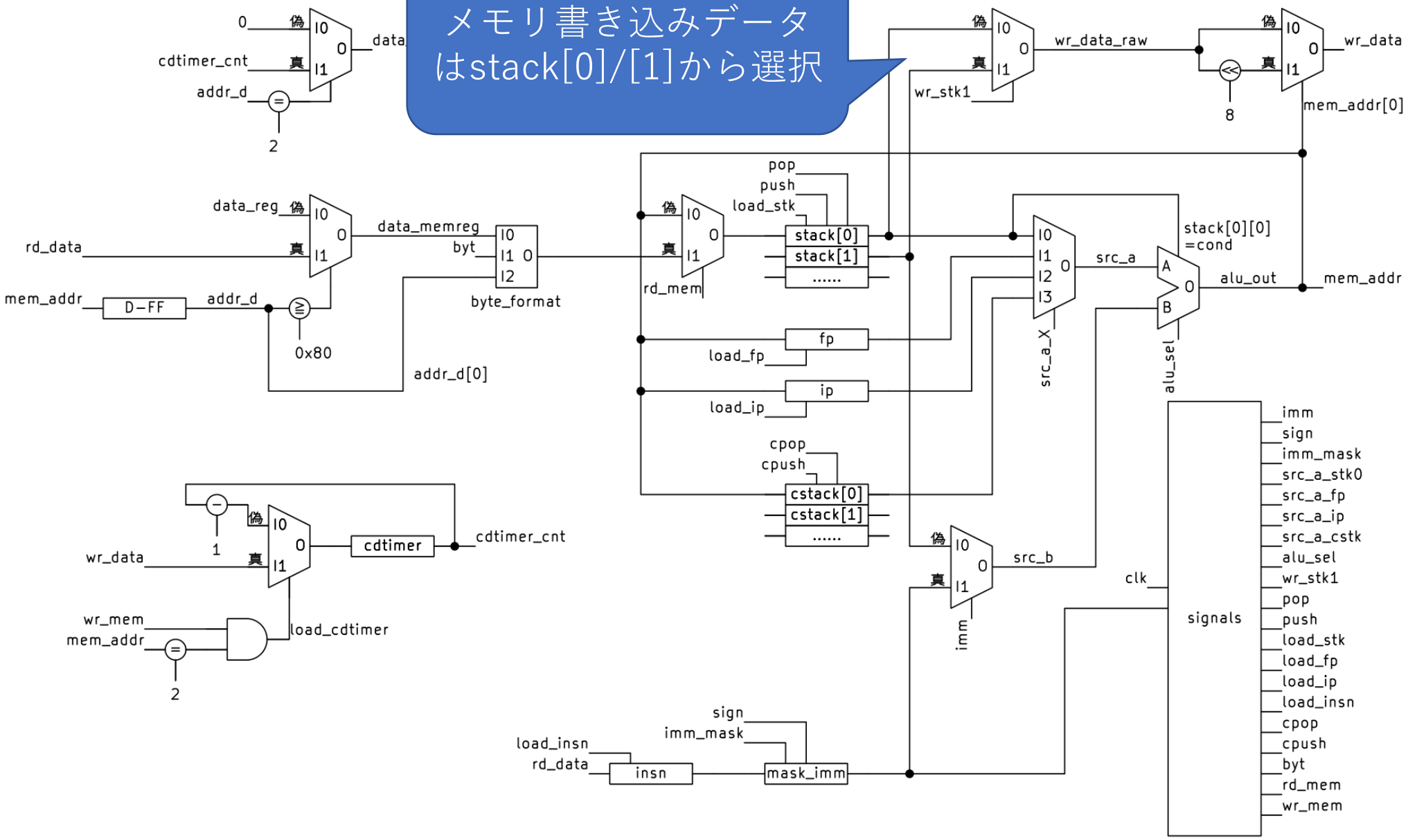
# データパス図


0
偽
真
data_reg
cdtimer_cnt
addr_d
2
data_reg
rd_data
mem_addr
D-FF
addr_d
0x80
data_memreg
byt
byte_format
addr_d[0]
wr_data_raw
wr_stk1
wr_data
8
mem_addr[0]
pop
push
load_stk
stack[0]
stack[1]
.....
rd_mem
fp
load_fp
ip
load_ip
cpop
cpush
cstack[0]
cstack[1]
.....
stack[0][0]
=cond
src_a
alu_out
mem_addr
src_a_X
sel
src_b
imm
clk
signals
alu_sel
wr_stk1
pop
push
load_stk
load_fp
load_ip
load_insn
cpop
cpush
byt
rd_mem
wr_mem
1
cdtimer
cdtimer_cnt
wr_data
wr_mem
mem_addr
load_cdtimer
2
sign
imm_mask
load_insn
rd_data
insn
mask_imm
ALU出力は常にメモリアドレスとして使用

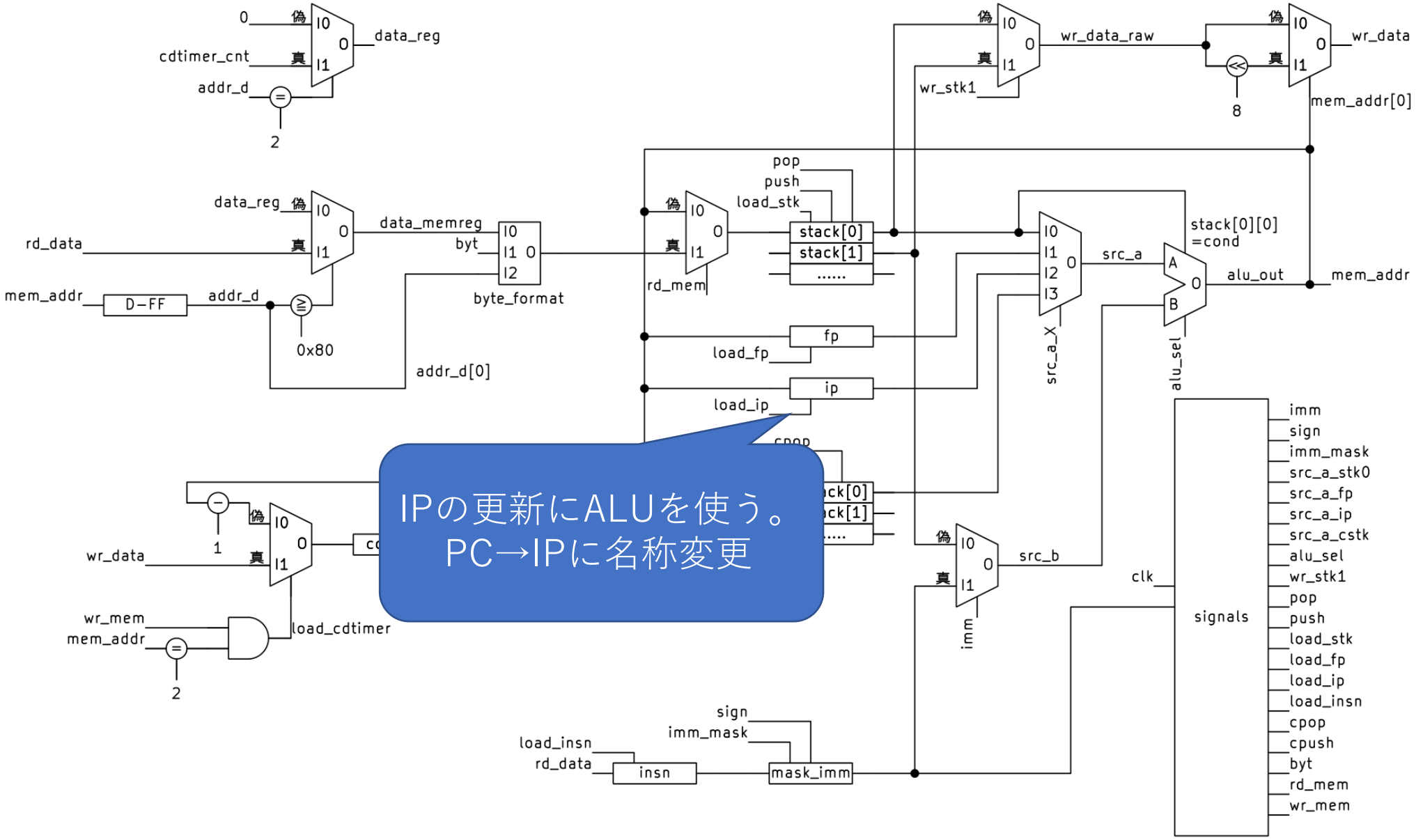
# データパス図



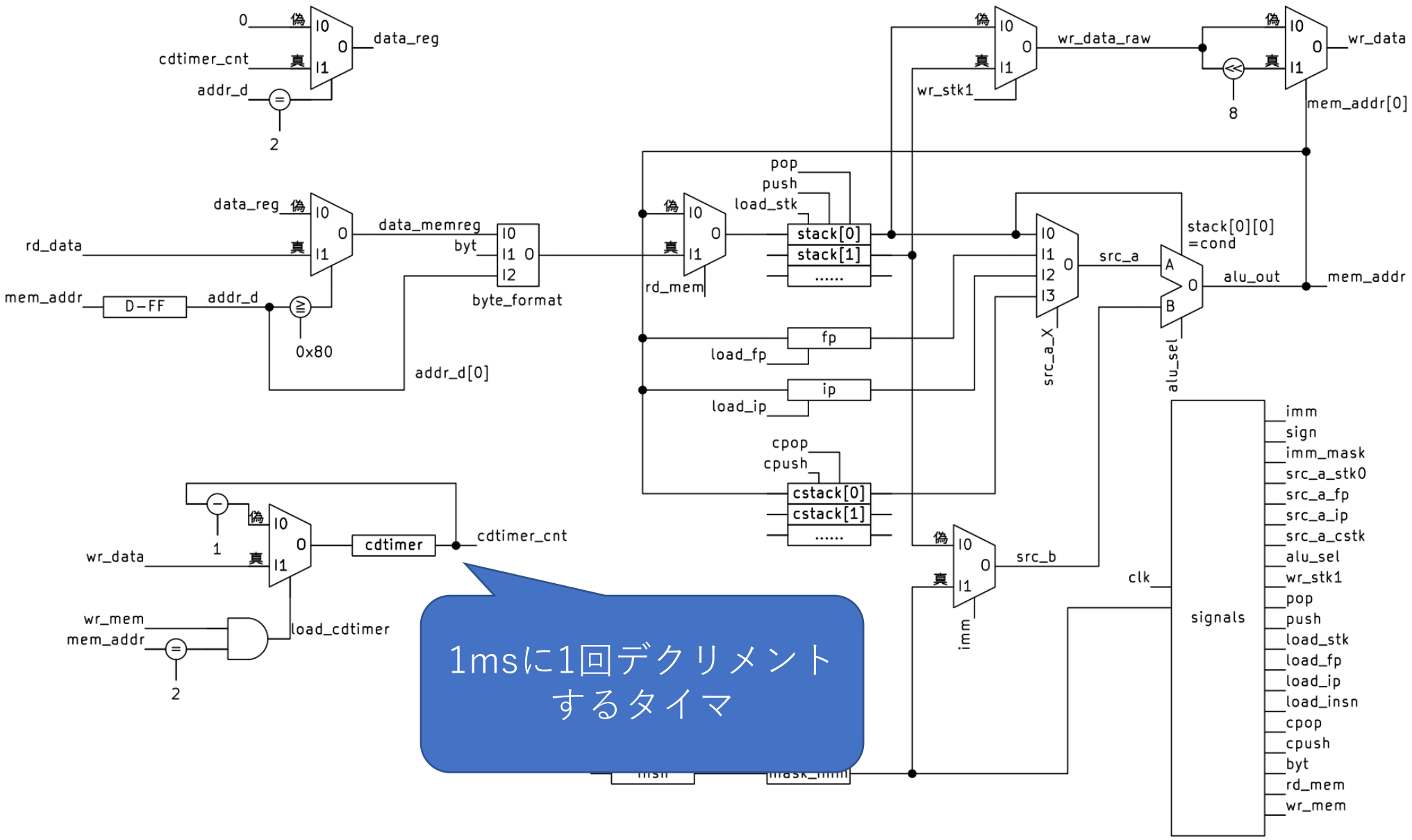
メモリ書き込みデータはstack[0]/[1]から選択



# データパス図


0
偽
I0
O
data_reg
cdtimer_cnt
真
I1
addr_d
=
2
data_reg
偽
I0
O
data_memreg
rd_data
真
I1
byt
I0
I1
O
I2
mem_addr
D−FF
addr_d
≧
0x80
byte_format
addr_d[0]
偽
I0
O
真
I1
rd_mem
pop
push
load_stk
stack[0]
stack[1]
.....
fp
load_fp
ip
load_ip
cpop
ack[0]
ack[1]
.....
偽
I0
O
真
I1
wr_data_raw
wr_stk1
偽
I0
O
真
I1
wr_data
≪
8
mem_addr[0]
I0
I1
I2
I3
O
src_a
src_a_X
stack[0][0]
=cond
A
O
B
alu_out
mem_addr
alu_sel
偽
I0
O
真
I1
src_b
imm
clk
signals
imm
sign
imm_mask
src_a_stk0
src_a_fp
src_a_ip
src_a_cstk
alu_sel
wr_stk1
pop
push
load_stk
load_fp
load_ip
load_insn
cpop
cpush
byt
rd_mem
wr_mem
−
1
偽
I0
O
真
I1
wr_data
wr_mem
mem_addr
=
2
load_cdtimer
sign
imm_mask
load_insn
rd_data
insn
mask_imm
IPの更新にALUを使う。
PC→IPに名称変更

# データパス図


0
偽
I0
O
data_reg
cdtimer_cnt
真
I1
addr_d
=
2
data_reg
偽
I0
O
data_memreg
rd_data
真
I1
byt
I0
I1
O
I2
mem_addr
D−FF
addr_d
≧
0x80
byte_format
addr_d[0]
偽
I0
O
真
I1
rd_mem
pop
push
load_stk
stack[0]
stack[1]
.....
fp
load_fp
ip
load_ip
cpop
cpush
cstack[0]
cstack[1]
.....
偽
I0
O
真
I1
wr_data_raw
wr_stk1
偽
I0
O
真
I1
wr_data
≪
8
mem_addr[0]
I0
I1
I2
I3
O
src_a
src_a_X
stack[0][0]
=cond
A
O
B
alu_sel
alu_out
mem_addr
偽
I0
O
真
I1
src_b
imm
clk
signals
imm
sign
imm_mask
src_a_stk0
src_a_fp
src_a_ip
src_a_cstk
alu_sel
wr_stk1
pop
push
load_stk
load_fp
load_ip
load_insn
cpop
cpush
byt
rd_mem
wr_mem
−
1
偽
I0
O
真
I1
wr_data
cdtimer
cdtimer_cnt
wr_mem
mem_addr
=
load_cdtimer
2
1msに1回デクリメントするタイマ

# 目次



- ComProcプロジェクトの概要
- CPUとコンパイラの同時並行進化とは
- ComProcの命令セット
- CPUのデータパス
- コンパイラへの最適化機能の追加
  - 時間がないので、興味ある人は展示ブースへどうぞ

# コンパイラの出力を眺める



最適化の話をするには、
ComProcのアセンブリコード
に慣れてもらう必要がある

対象Cコード

```
void delay_ms(int ms) {
  int *t = 2;
  *t = ms;
  while (*t > 0) {}
}
```

```
delay_ms:
    cpush fp
    st cstack+0x0
    add fp,0x2   ; 引数の領域
    add fp,0x2   ; t の領域
    push 2
    push cstack+0x2
    sta   ; t に 2 を書く
    pop
    push cstack+0x0
    ldd   ; ms を読む
    push cstack+0x2
    ldd   ; t を読む
    std   ; *t に ms を書く
    pop
```

```
L_0:
    push cstack+0x2
    ldd   ; t の値を読む
    ldd   ; *t の値を読む
    push 0
    lt    ; *t > 0
    push 0
    neq   ; (*t > 0) != 0
    jz L_1
    jmp L_0
L_1:
    cpop fp
    ret
```

# 今回追加した最適化



- 変数定義に出会うたびに`add fp,0x2`を繰り返すことを止めて、一気にFPを調整する
- `push cstack+N`と`ldd`が連続する部分は`ld cstack+N`に短縮する
  - 同様にPUSH+STA/STDも短縮する
- スタックのゴミを掃除するための`pop`を省略する

- これをすべて適用したら、
  led.cのコンパイル結果が67命令→47命令に

# 最適化の効果



最適化前：25命令

```
delay_ms:
    cpush fp
    st cstack+0x0
    add fp,0x2
    add fp,0x2
    push 2
    push cstack+0x2
    sta
    pop
    push cstack+0x0
    ldd
    push cstack+0x2
    ldd
    std
    pop
```

```
L_0:
    push cstack+0x2
    ldd
    ldd
    push 0
    lt
    push 0
    neq
    jz L_1
    jmp L_0
L_1:
    cpop fp
    ret
```

最適化後：18命令

```
delay_ms:
    cpush fp
    st cstack+0
    add fp,4  ; add fp の削減
    push 2
    st cstack+2  ; push+sta
    ld cstack+0  ; push+ldd
    ld cstack+2  ; push+ldd
    std  ; pop の削減
L_1:
    ld cstack+2  ; push+ldd
    ldd
    push 0
    lt
    push 0
    neq
    jz L_2
    jmp L_1
L_2:
    cpop fp
    ret
```

# この後やりたい最適化



- push 0; neq の省略
  - 0/非0を0/1に変換するために挿入される命令
  - 条件ジャンプJZ/JNZがstack[0]の最下位ビットしか見ないから
  - LTはもともと0/1を出力する

無駄

```
……
push 0
lt   ; *t > 0
push 0
neq  ; (*t > 0) != 0
jz L_1
……
```

- 定数埋め込み
  - 変数の値が変わらない場合
  - 毎回メモリから読む必要がない
  - 定数伝播解析はちょっと面倒

定数2

```
void delay_ms(int ms) {
  int *t = 2;
  *t = ms;
  while (*t > 0) {}
}
```

無駄

push 2 で十分

```
push 2
st cstack+2
ld cstack+0
ld cstack+2  ; 定数 2
std
```

# 今後やりたいこと



- CPU
  - 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
  - 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