

2019年度 計算機システム(演習)
第5回
2020.01.07

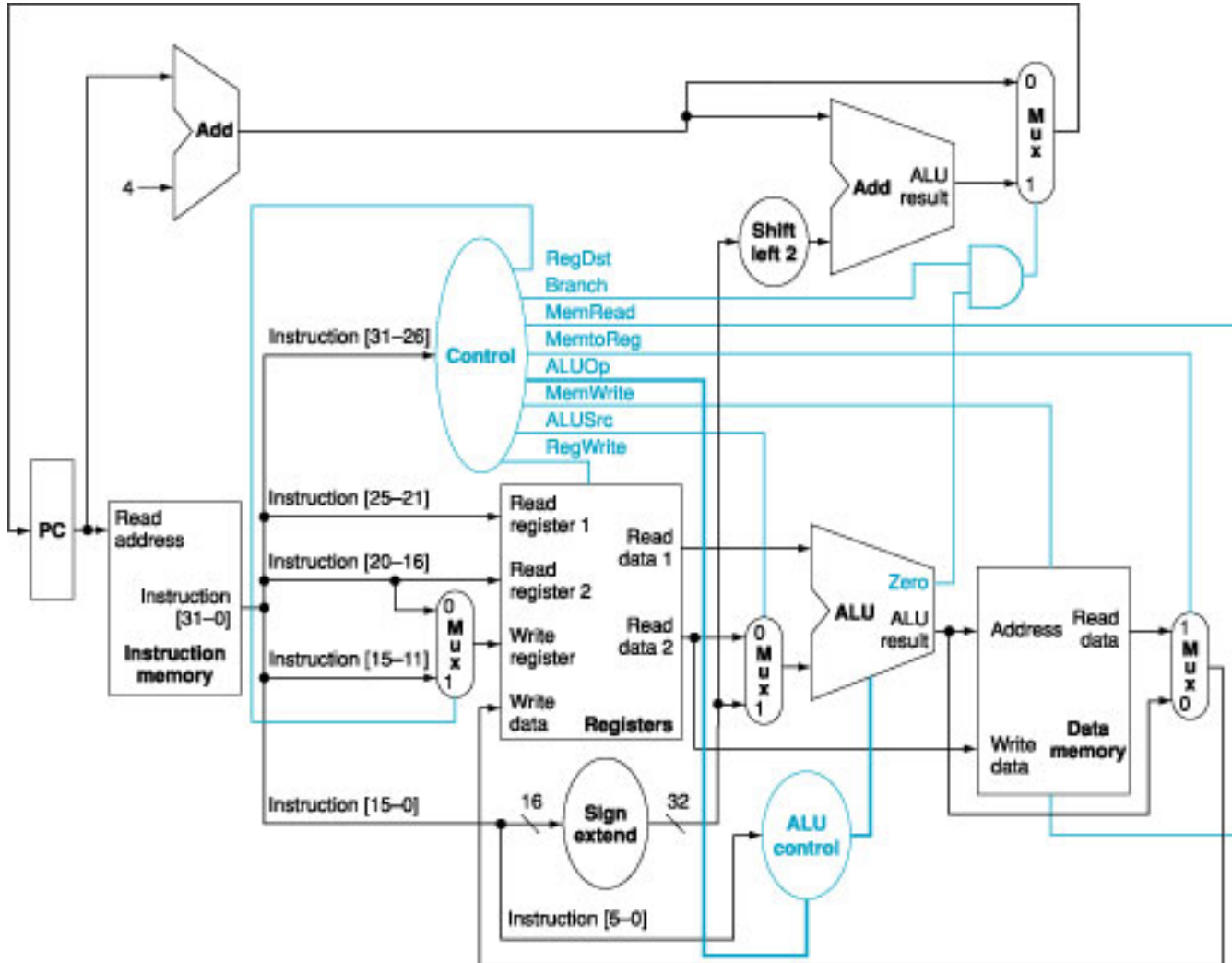
遠藤 敏夫(学術国際情報センター/数理・計算科学系 教授)
野村 哲弘(学術国際情報センター/数理・計算科学系 助教)

本日の内容 (Outline)

- ▶ MIPSシミュレータの概要
 - ▶ 今回から4回にかけてMIPSシミュレータを作ります
- ▶ ALUの作成
 - ▶ 1ビット加算器の作成
 - ▶ Signal, Pathを作成
 - ▶ 1ビット論理演算ユニットの作成
 - AND, OR, NOTゲート etc ...
 - ▶ 32ビット加算器の作成
 - ▶ 1ビットALU
 - ▶ 32ビットALU

MIPSシミュレータの作成

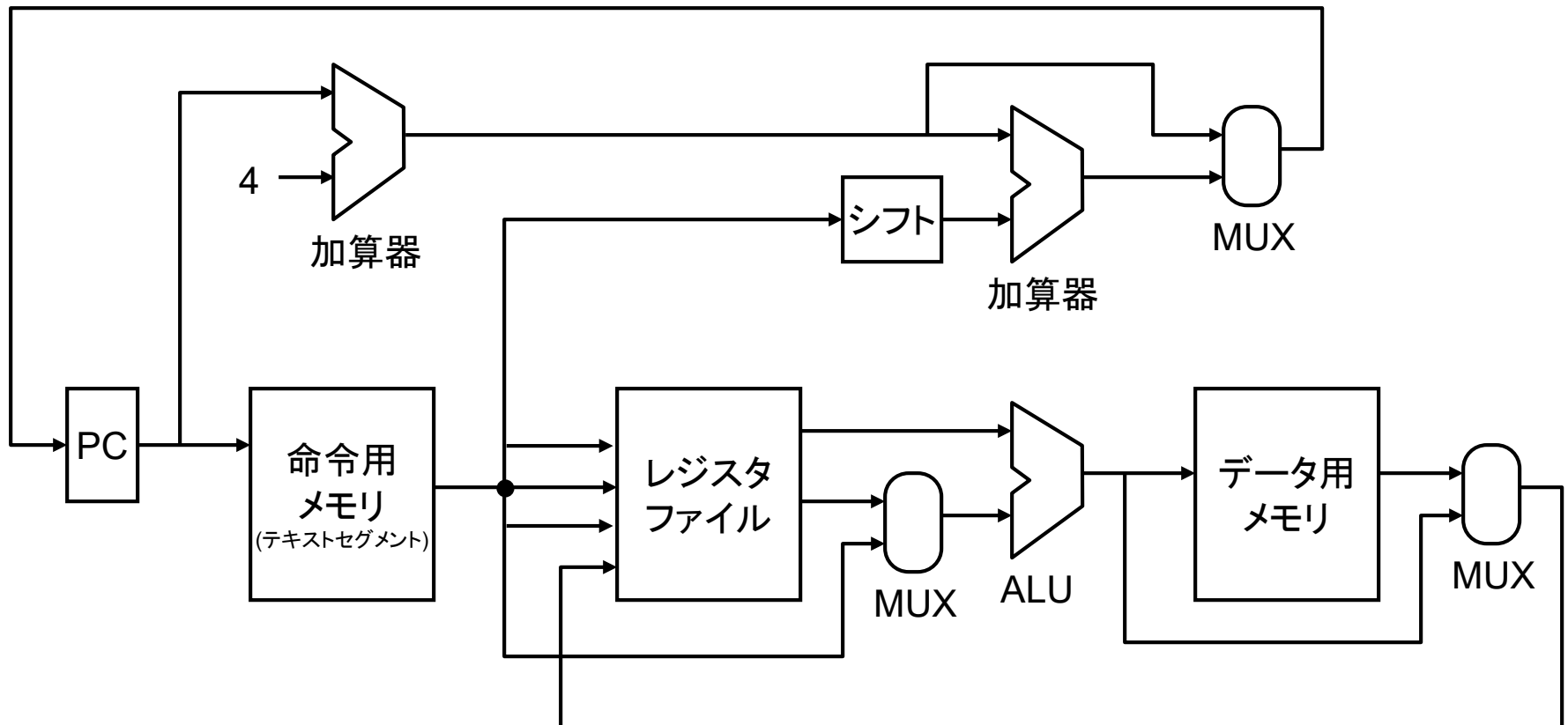
MIPSシミュレータ 完成図



MIPSシミュレータ構築の流れ

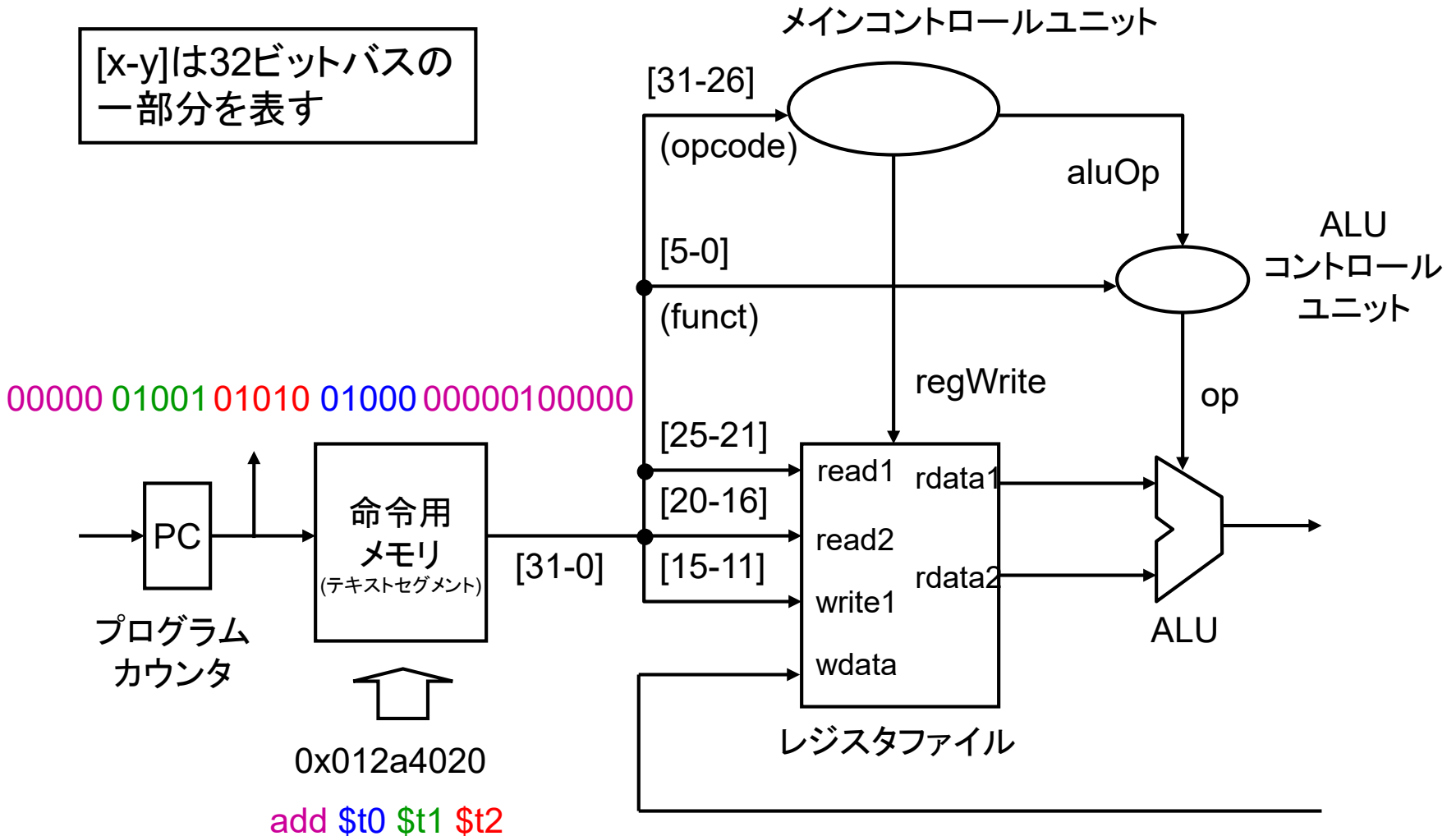
1. **ALUの作成**
2. レジスタファイル
3. メモリ領域
 - ▶ 命令用メモリ
 - ▶ データ用メモリ
4. PCの作成
5. メインコントロールユニット
6. ALUコントロールユニット
7. 機能拡張 (演習範囲外)
 - ▶ メモリアクセス命令
 - ▶ 分岐命令

MIPSシミュレータ 概略図

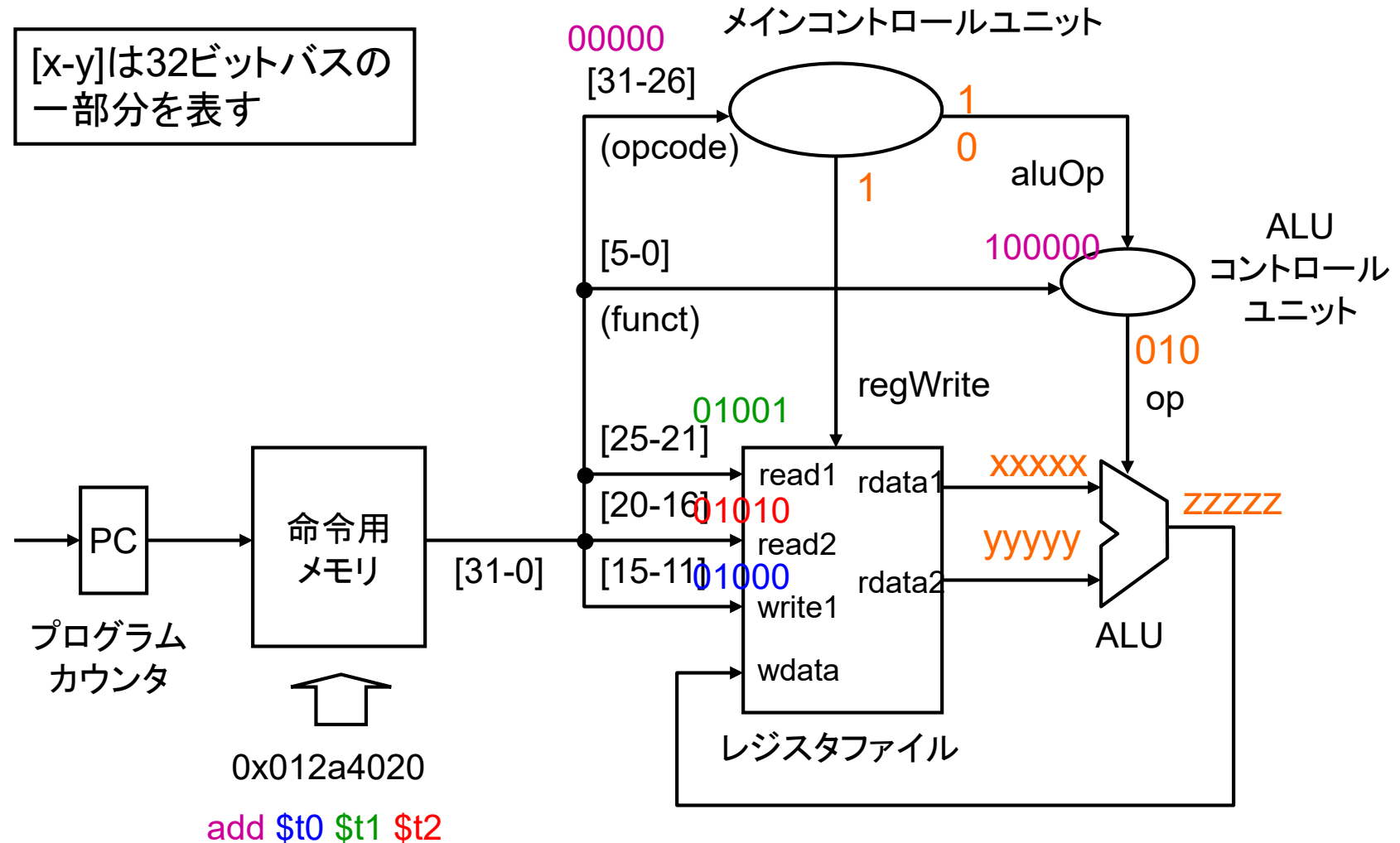


動作の流れ(一部)

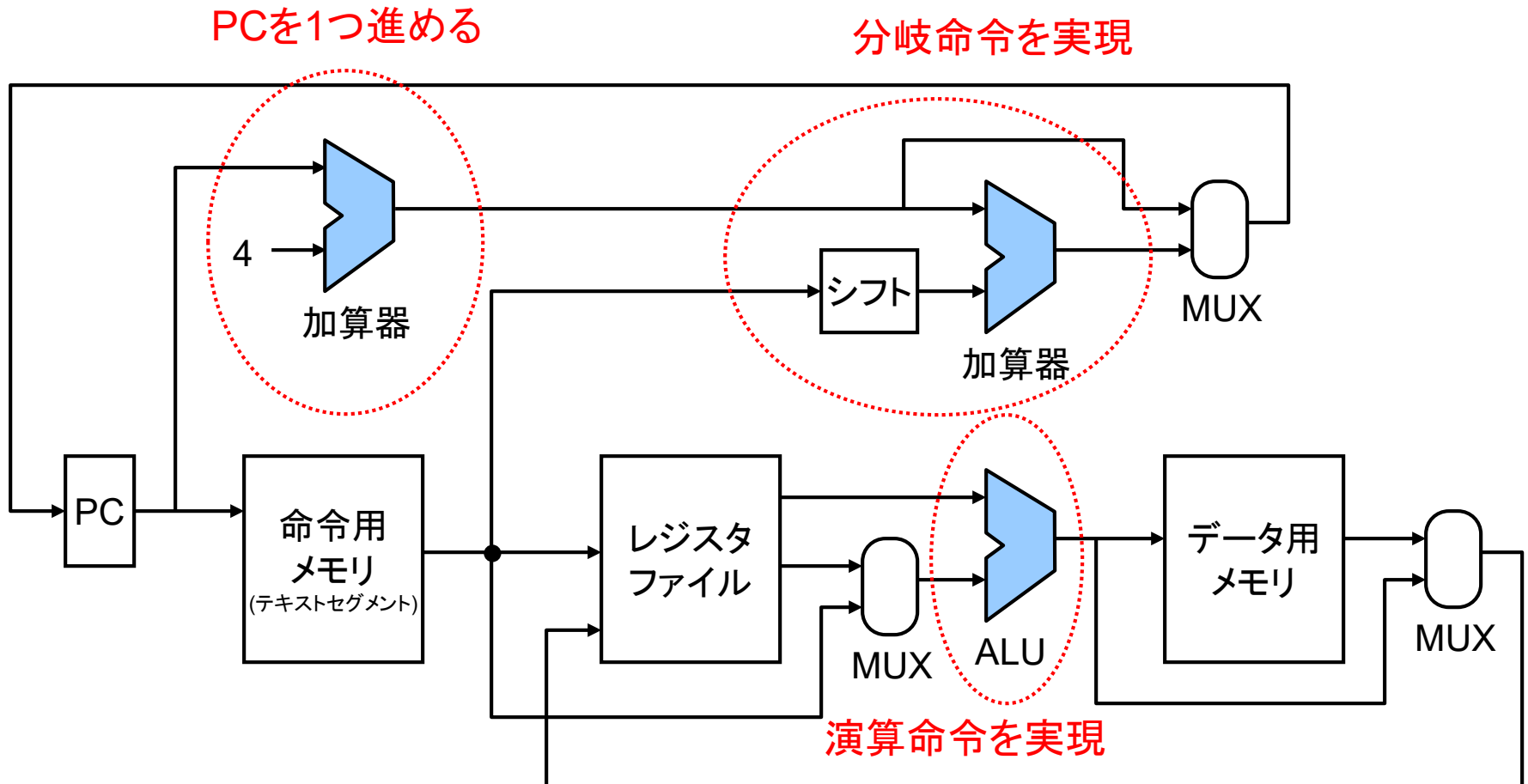
[x-y]は32ビットバスの
一部分を表す



動作の流れ(一部)



MIPSシミュレータ 概略図



本日の内容 (Outline)

- ▶ MIPSシミュレータの概要
- ▶ ALUの作成
 - ▶ 1ビット加算器の作成
 - ▶ Signal, Pathを作成
 - ▶ 1ビット論理演算ユニットの作成
 - AND, OR, NOTゲート etc ...
 - ▶ 32ビット加算器の作成
 - ▶ 1ビットALU
 - ▶ 32ビットALU

第1目標：ALU()の作成


▶ ALU:Arithmetic Logic Unit

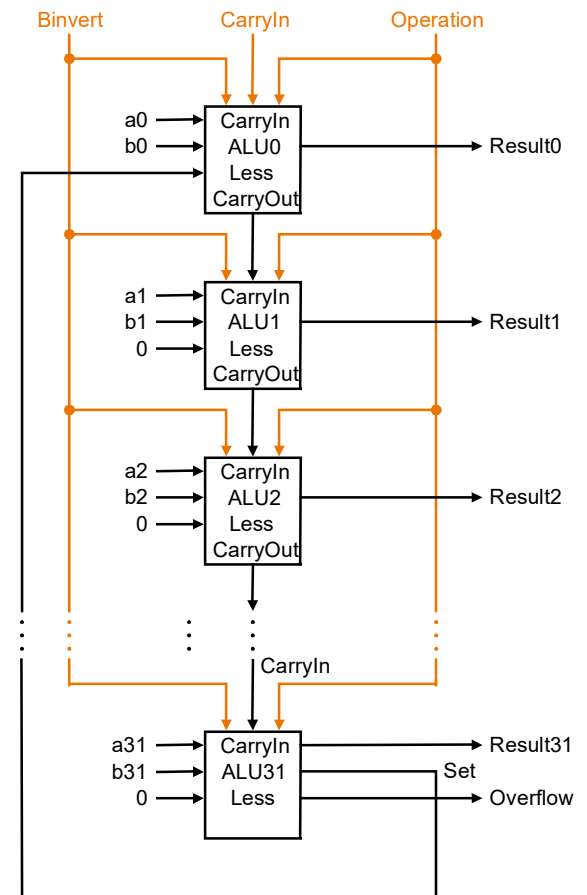
▶ 演算論理装置

▶ 組み合わせ回路

- ▶ AND, OR, NOT ゲートを使って作成

▶ 作成の順番

1. 1ビット加算器  本日の内容
2. 32ビット加算器 
3. 1ビットALU
4. 32ビットALU



Signal

▶ 回路内の信号を表現

- ▶ 信号は 0, 1 の値を持つ
 - ▶ 0 は false で表現
 - ▶ 1 は true で表現

▶ enumを用いて表現

- ▶ C言語の機能の一つ
- ▶ 定数のリストを定義
 - ▶ リストの頭のものから順番に整数0, 1, 2, ...と定義される
 - ▶ 自分で定義することも可能

mips.h

```
typedef enum {  
    false,    //false = 0  
    true      //true  = 1  
} Signal;
```

```
Signal s0 = true, s1 = false;  
printf("s0 = %d\n", s0); // 1  
printf("s1 = %d\n", s1); // 0
```

```
typedef enum {  
    ichiro = 1,    // = 1  
    saburo = 3,    // = 3  
    yonro         // = 4  
} Name;
```

ゲート

- ▶ 基本論理演算を実装した電子素子
- ▶ 3種類
 - ▶ AND, OR, NOT
- ▶ 入力から出力が決まる
 - ▶ 出力する信号を論理演算で決定する
 - ▶ 出力配線に新しい信号を設定する
- ▶ 関数にてゲートの機能を実現
 - ▶ 引数に入力信号 (signal) と出力信号のポインタ (signal *)

AND Gate

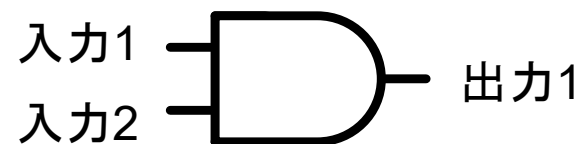
▶ AND ゲートを表現

▶ 論理積: $A \cdot B$

▶ 論理積は `&&` 演算子で実現

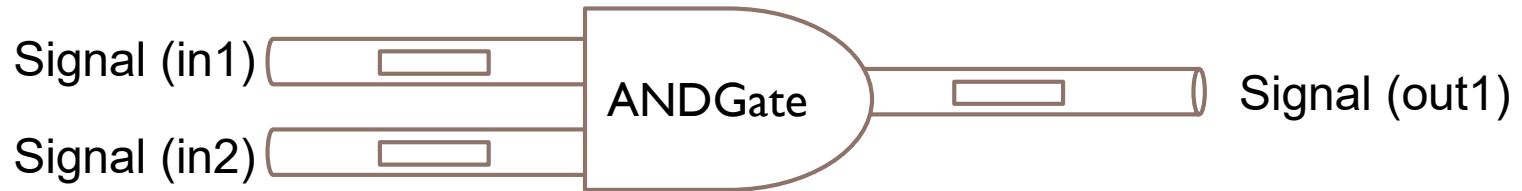
▶ 引数は入出力配線

▶ 入力信号から出力信号を決定



入力1	入力2	出力1
0	0	0
0	1	0
1	0	0
1	1	1

AND Gate のコード例



```
void and_gate(Signal in1, Signal in2, Signal *out1)
{
    *out1 = in1 && in2; // 論理積
}
```

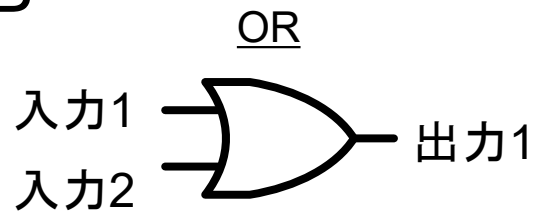
// テスト例

```
Signal a = true, b = false, out;
and_gate(a, b, &out);
printf("AND(%d, %d) => %d\n", a, b, out); //AND(1, 0) => 0
```

gate.c

OR Gate, NOT Gate

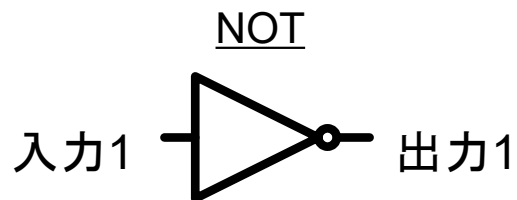
$A+B$



論理和
(||)

入力1	入力2	出力1
0	0	0
0	1	1
1	0	1
1	1	1

\overline{A}



否定
(!)

入力1	出力1
0	1
1	0

NAND Circuit

▶ 組み合わせ回路

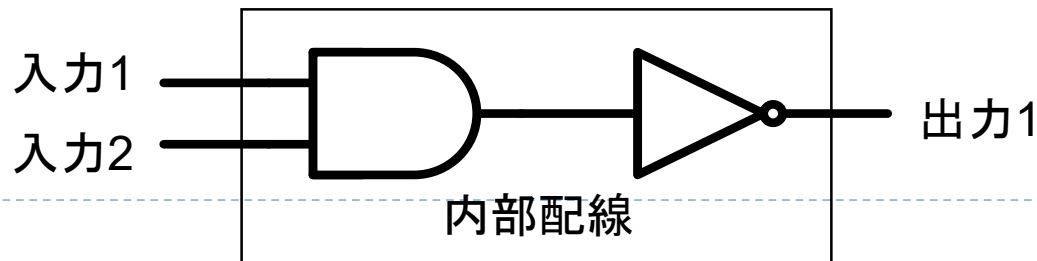
▶ AND Gate と NOT Gate の組み合わせ

- ▶ 内部状態を持たない
- ▶ 出力は入力だけに依存

▶ 回路を構成

- ▶ 内部配線でゲート同士を接続

入力1	入力2	出力1
0	0	1
0	1	1
1	0	1
1	1	0



NAND Circuit のコード例

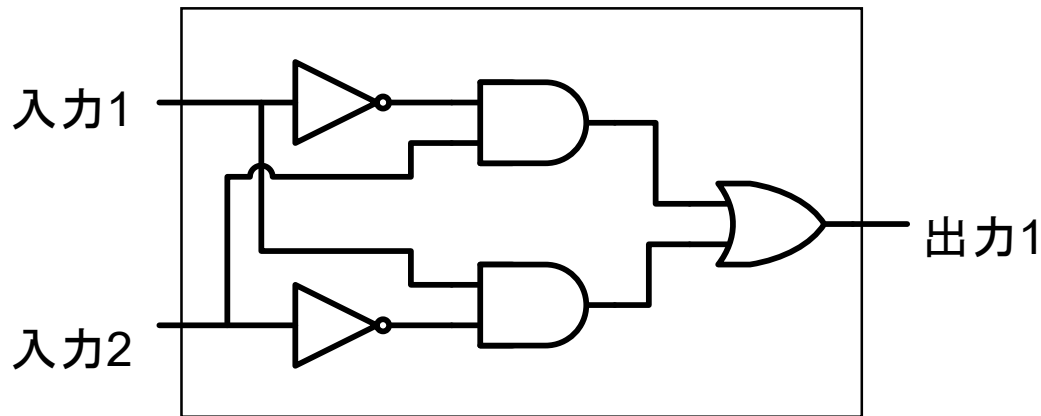
```
void nand_circuit (Signal in1, Signal in2, Signal *out1)
{
    Signal inner; //内部配線
    // 内部配線をつかってAND Gateの出力とNOT Gateの入力を接続
    and_gate(in1, in2, &inner);
    not_gate(inner, out1);
}
```

XOR Circuit

▶ XOR

- ▶ 排他的論理和 : exclusive or
- ▶ Not Equal
 - ▶ 2つの入力値が異なる場合に真
- ▶ $\overline{A} \cdot B + A \cdot \overline{B}$ と等価

入力1	入力2	出力1
0	0	0
0	1	1
1	0	1
1	1	0



1ビット全加算器 (FA) (1/2)

▶ 全加算器 (Full Adder)

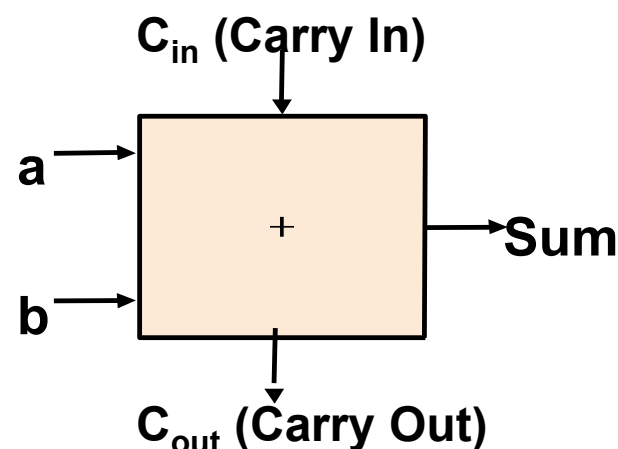
- ▶ 桁上がり (Carry In) を入力に受け付ける
 - ▶ c.f) 半加算器 (Half Adder) : 桁上がりを入力に取らない

▶ 講義スライドの論理式

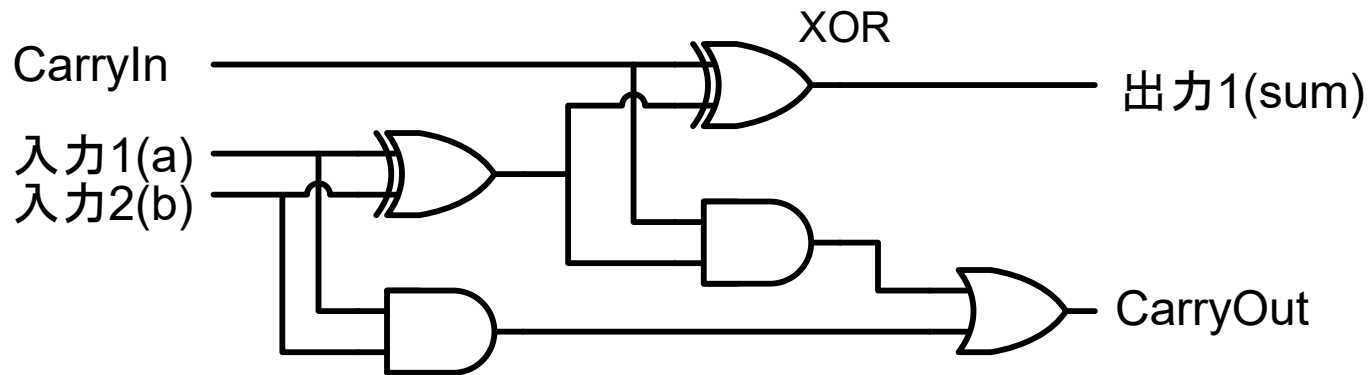
- ▶ $\text{sum} = A \text{ xor } B \text{ xor } C_{\text{in}}$
- ▶ $C_{\text{out}} = A \cdot B + A \cdot C_{\text{in}} + B \cdot C_{\text{in}}$

▶ 実装に使用する論理式

- ▶ $\text{sum} = A \text{ xor } B \text{ xor } C_{\text{in}}$
- ▶ $C_{\text{out}} = A \cdot B + (A \text{ xor } B) \cdot C_{\text{in}}$



1ビット全加算器 (FA) (2/2)



入力 1	入力 2	Carry In	出力 1	Carry Out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1

入力 1	入力 2	Carry In	出力 1	Carry Out
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

AND-N Gate, OR-N Gate

- ▶ n 入力の AND GateとOR Gate
 - ▶ 回路ではなくfor文で処理
- ▶ (ORN Gateは省略)

```
void andn_gate(Signal *in1, int n, Signal *out1) {  
    int i;  
    Signal s = true;  
    for (i = 0; i < n; ++i) {  
        if (in1[i] == false) {  
            s = false;  
            break;  
        }  
    }  
    *out1 = s;  
}
```

2進数（復習）

- ▶ 2進数を10進数に変換するには？
- ▶ 10進数を2進数に変換するには？
- ▶ 2進数の符号を逆転するには？（2の補数）

- ▶ (2007年院試問題より)
 - 1) 整数の20と-20のビット表現はそれぞれ00010100 と 11101100 となる. -40 のビット表現を答えよ.
 - 2) 整数を表現するのに,最上位ビットで符号を表し,その他のビットで数の絶対値を表す方法も考えられる.たとえば,この方法で-20を表すと10010100となる.この方法と前述の 2^8 を法とした方法を比較し, 2^8 を法とした方法が優れている点を論じよ.

2進数と16進数

- ▶ 10進数よりも、16進数の方が2進数と相性が良い

$$\begin{array}{ccc|c|c|c} 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & (2) \\ & & & 1 & & 4 & & (16) & \longrightarrow & 0x14 \end{array}$$

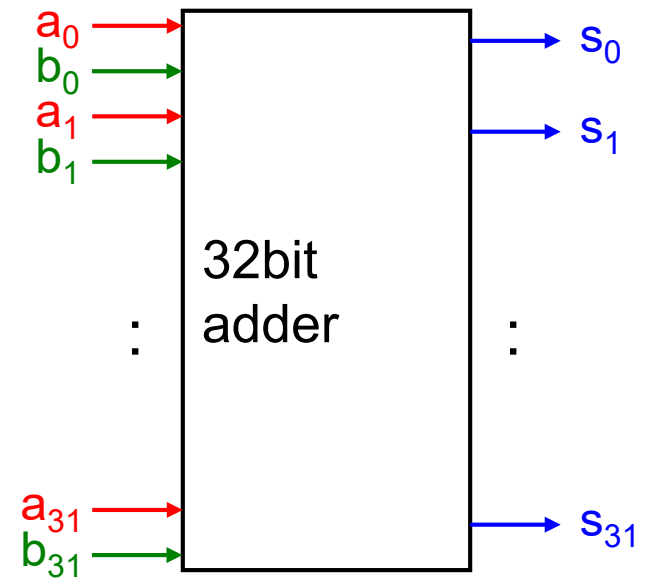
Java上での表現

- ▶ 問:

- 1) $F_{(16)}$ は、2進数でどう表現されるか？ 10進数では？
- 2) $-11_{(16)}$ は、32ビット2の補数ではどう表現されるか？
- 3) $1000\dots0000$ (32ビット2の補数表現) は、どのような数か？
16進数で書くと？

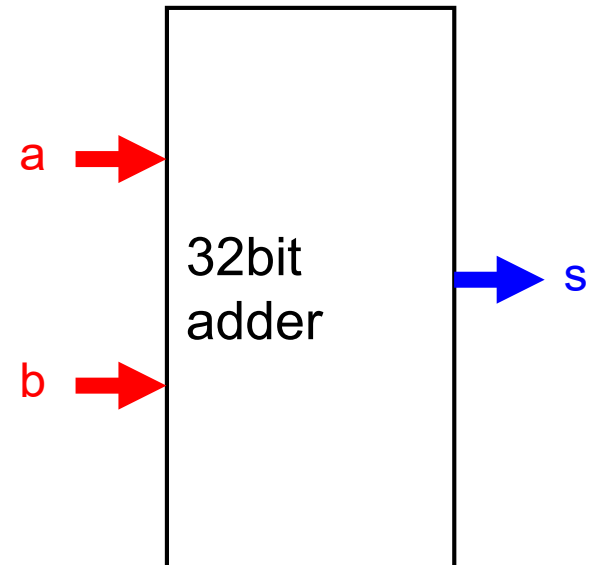
32ビット加算器

- ▶ 入力配線
 - ▶ 32本 2組
- ▶ 出力配線
 - ▶ 32本
- ▶ Signalが多すぎて取り扱いにくい
 - ▶ 信号を1つ1つ設定するのは大変



Word 構造体

- ▶ 信号 (Signal) の集合をバスとして表現
 - ▶ Signal の配列を持つ
- ▶ Word構造体と関数
 - ▶ `void word_set_value(Word *w, int val)`
 - ▶ 指定された数値を二進数で表すように各 Signalを設定
 - ▶ 例) `val=5 => 00...00101`
 - ▶ `int _word_get_value(Word w)`
 - ▶ 各配線の信号が表す二進数の数値を返す
 - ▶ 例) `00...00101 => 5`



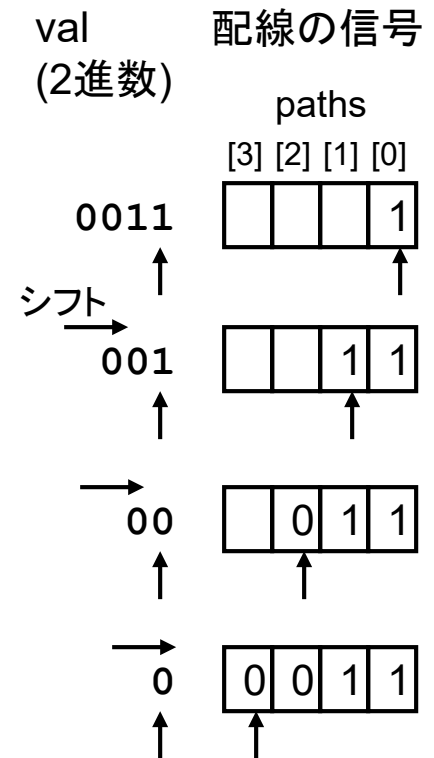
```
typedef struct {  
    Signal bit[32]  
} Word;
```

Word 構造体のコード (1/2)

0x: 16進数 L: long型

```
// valの値を表すように配線に信号を設定
void word_set_value(Word *w, int val)
{
    int i;
    for (i = 0; i < 32; ++i) {
        if ((val & 0x1) != 0) { // ビットが立っていたら
            w->bit[i] = true;
        }
        else {
            w->bit[i] = false;
        }
        val >>= 1; //1ビット右シフト
    }
}
```

4bitでのsetValue(3)

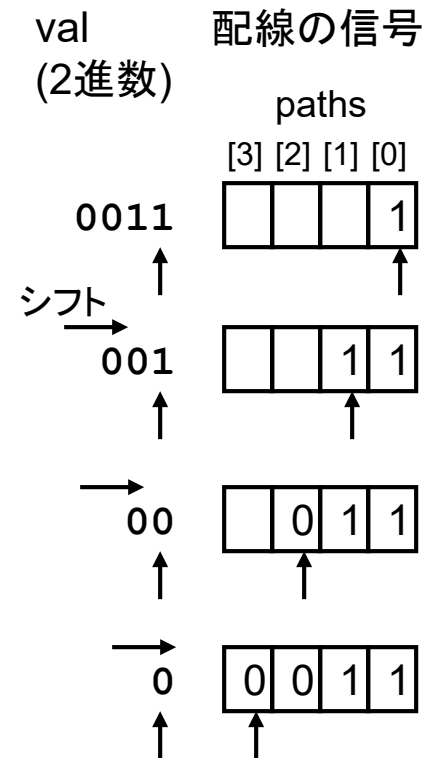


Word 構造体のコード (2/2)

0x: 16進数 L: long型

```
// 配線の信号によって表される整数値を返す
int word_get_value(Word w)
{
    int i, val = 0;
    for(i = 31; i >= 0; --i) {
        val <<= 1; // 1ビット左シフト
        if (w.bit[i]) {
            val++; // 信号が1ならビットを立てる
        }
    }
    return val;
}
```

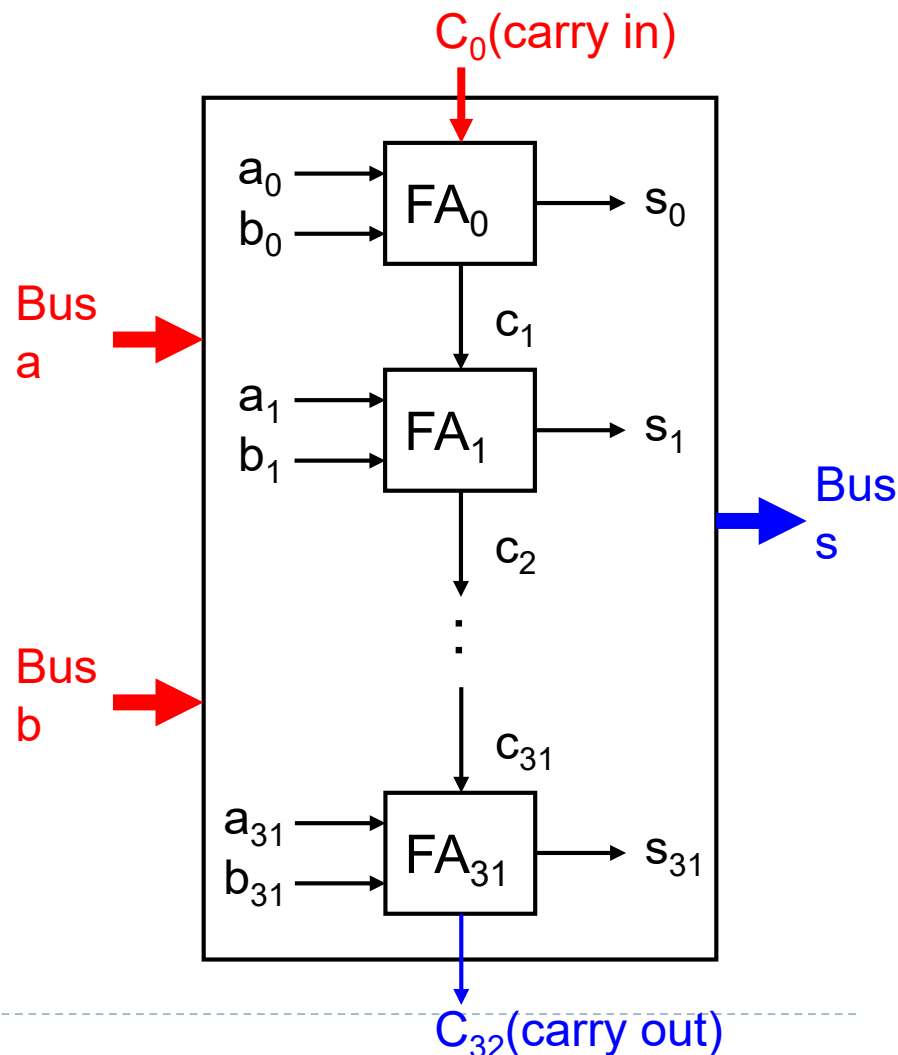
4bitでのsetValue(3)



32ビット Ripple Carry Adder

▶ 1ビット全加算器 (FA) を32個つなげた加算器

- ▶ FA_0 を実行
- ▶ キャリー C_i を FA_i に伝播
- ▶ FA_i を実行
- ▶ :
- ▶ キャリー C_{31} を FA_{31} に伝播
- ▶ FA_{31} を実行



RCA のコード

```
void rca(Word in1, Word in2, Signal carry_in,  
         Word *out1, Signal *carry_out){  
    // FAをつなげる (for文を使ってよい)  
}
```

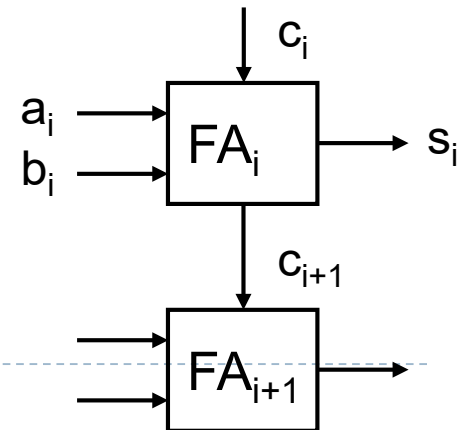
Carry Lookahead Adder (1/2)

▶ Ripple Carry Adder の問題

- ▶ キャリーの伝播のために FA を一つずつしか実行できないため、**遅い**

▶ 同時により多くのビットを計算できないか？

- ▶ あらかじめキャリーが計算できればよい
- ▶ キャリーは $c_{i+1} = g_i + p_i \cdot c_i$ の関係を満たす
 - ▶ $g_i = a_i \cdot b_i$: Generator
 - ▶ $p_i = a_i + b_i$: Propagator
- ▶ 前回は $c_{i+1} = a_i \cdot b_i + (a_i \text{ xor } b_i) \cdot c_i$ とした



Carry Lookahead Adder (2/2)

▶ この式を展開していくと

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + g_0 p_1 + p_0 p_1 c_0$$

$$c_3 = g_2 + g_1 p_2 + g_0 p_1 p_2 + p_0 p_1 p_2 c_0$$

$$c_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + p_0 p_1 p_2 p_3 c_0$$

c_0 が決まれば
 $c_1 \sim c_4$ が決まる

$$g_i = a_i \cdot b_i$$

$$p_i = a_i + b_i$$

(オプション
課題で使用)

▶ この計算を行うPFA (Partial FA) と CLA (Carry Lookahead Unit) 導入

▶ PFA

- ▶ 入力: Bus a, b, Path carry
- ▶ 出力: g, p, s

▶ CLA

- ▶ 入力: $g_0 \sim g_3, p_0 \sim p_3, c_0$
- ▶ 出力: $c_1 \sim c_4$

4ビット Carry Lookahead Adder

▶ PFA (Partial FA)

- ▶ g_i, p_i を計算
- ▶ s_i (sum)を計算
 - ▶ $a_i \text{ xor } b_i \text{ xor } c_i$

▶ Carry Lookahead Unit (CLU)

- ▶ $c_1 \sim c_3$, carry out を計算

▶ 動作

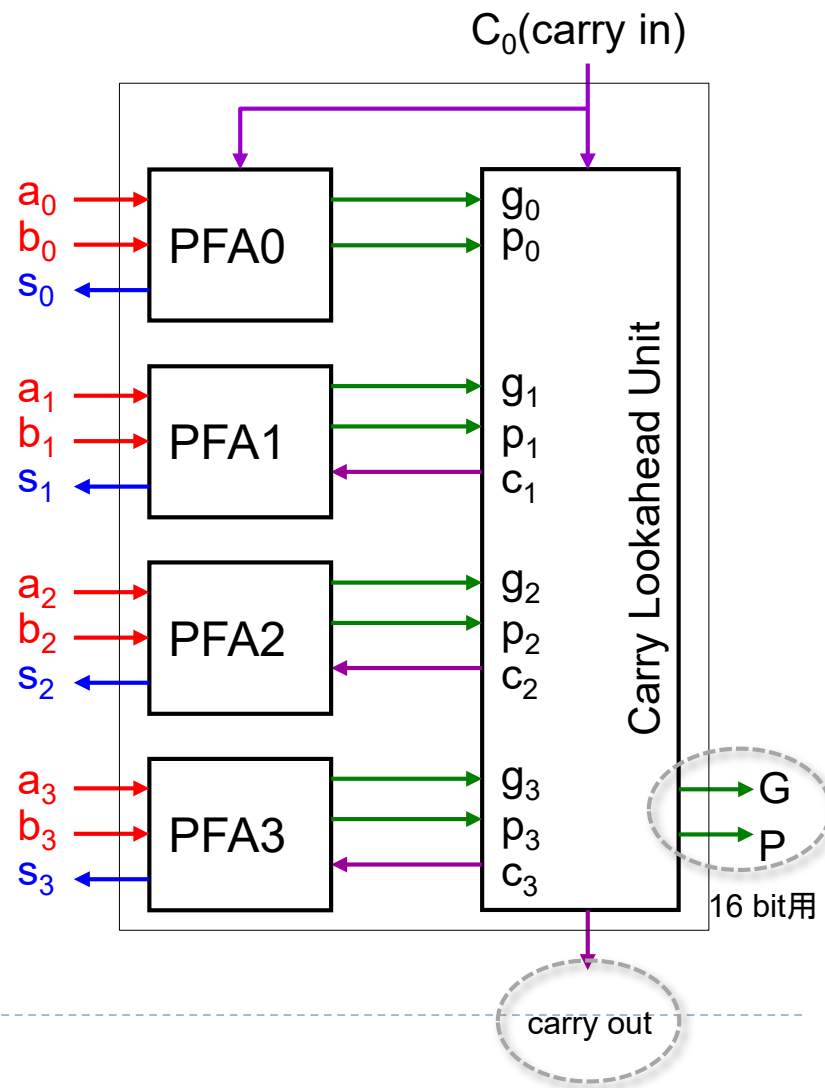
1. PFA: g_i, p_i を計算

- ▶ ここで出力される s_i は $c_i = 0$ の場合

2. CLU: $c_1 \sim c_3$, carry out を計算

3. PFA: s_i を計算

- ▶ ここで正しい s_i が出力

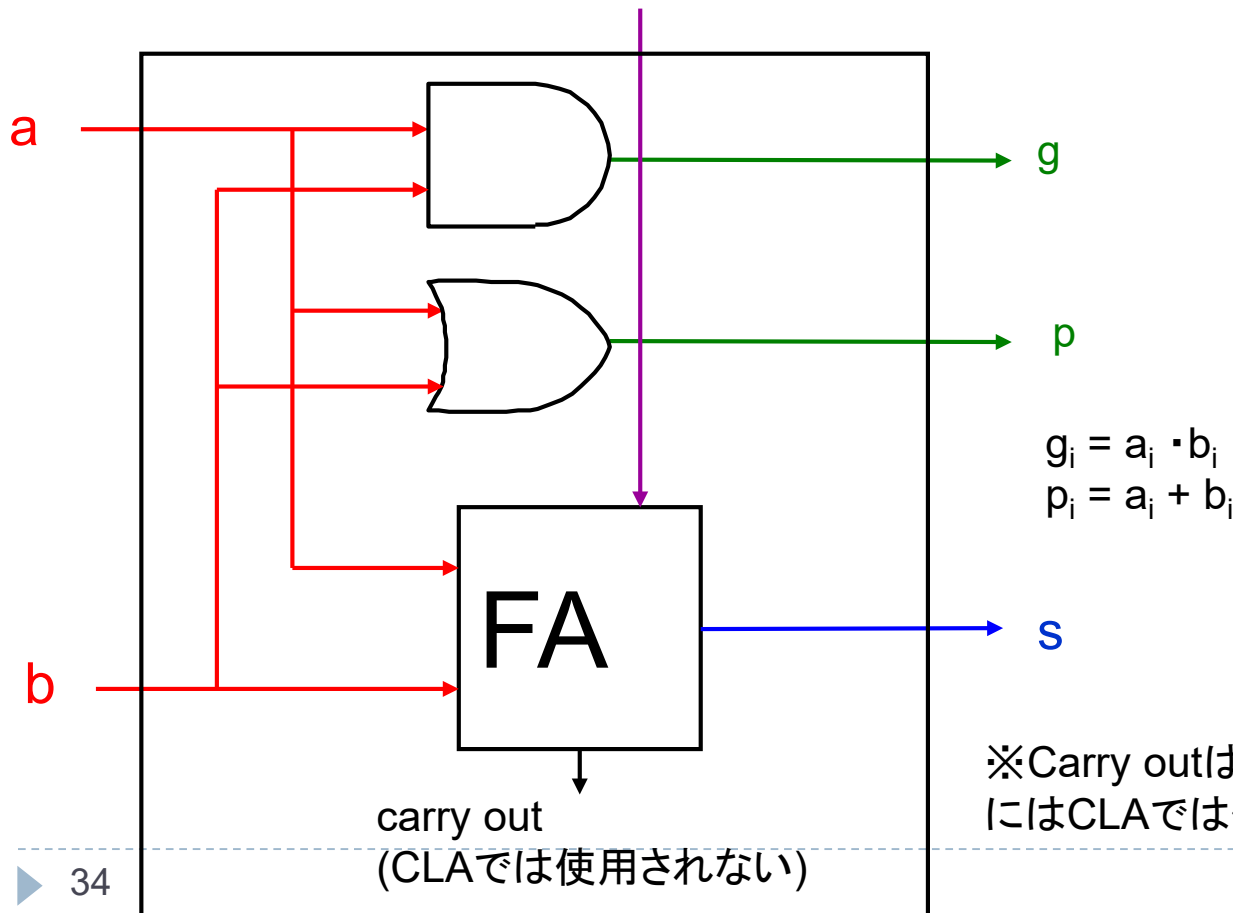


PFA

▶ 以下のようにPFAを実装

- ▶ 注意: PFA1~3のcarry inはない

※PFA0用のcarry in.



※Carry outはCLUで計算するため、実際にはCLAでは使用されない

16ビット Carry Lookahead Adder

▶ 4bit CLAのCarry-inの計算

▶ Cはどのように記述されるか？

▶ C_1 = “4bit CLA0のcarry-out”

$$C_1 = g_3 + g_2 P_3 + g_1 P_2 P_3 + g_0 P_1 P_2 P_3 + P_0 P_1 P_2 P_3 C_0$$

▶ C_2 = “4bit CLA1のcarry-out”

$$C_2 = g_7 + g_6 P_7 + g_5 P_6 P_7 + g_4 P_5 P_6 P_7 + P_4 P_5 P_6 P_7 C_1$$

▶ C_3 = “4bit CLA2～”, C_4 = “4bit CLA3～”

▶ 以下のように一般化すると

$$G_i = g_{3+4xi} + P_{3+4xi} g_{2+4xi} + P_{3+4xi} P_{2+4xi} g_{1+4xi} + P_{3+4xi} P_{2+4xi} P_{1+4xi} g_{0+4xi}$$

$$P_i = P_{0+4xi} P_{1+4xi} P_{2+4xi} P_{3+4xi}$$

▶ Cは以下のように記述できる

$$C_1 = G_0 + P_0 C_0$$

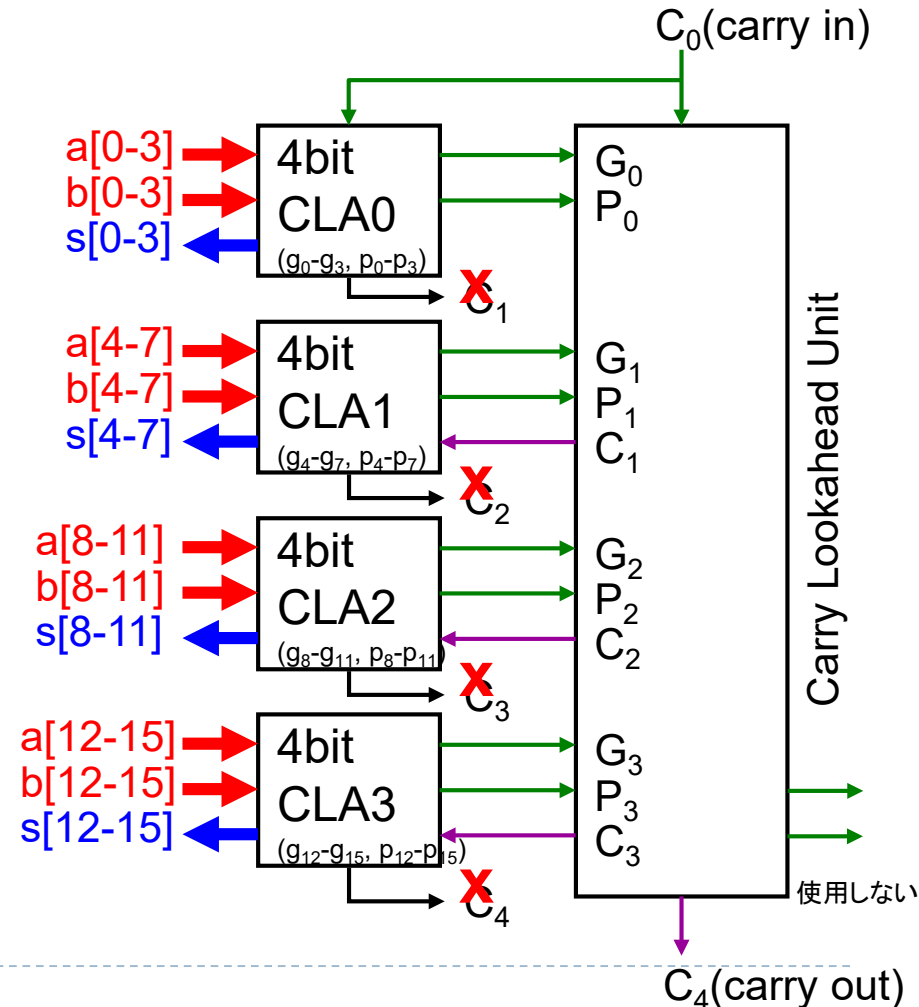
$$C_2 = G_1 + P_1 C_1 = G_1 + G_0 P_1 + P_0 P_1 C_0$$

$$C_3 = \dots, C_4 = \dots$$

▶ 16bit CLAにおけるCLU

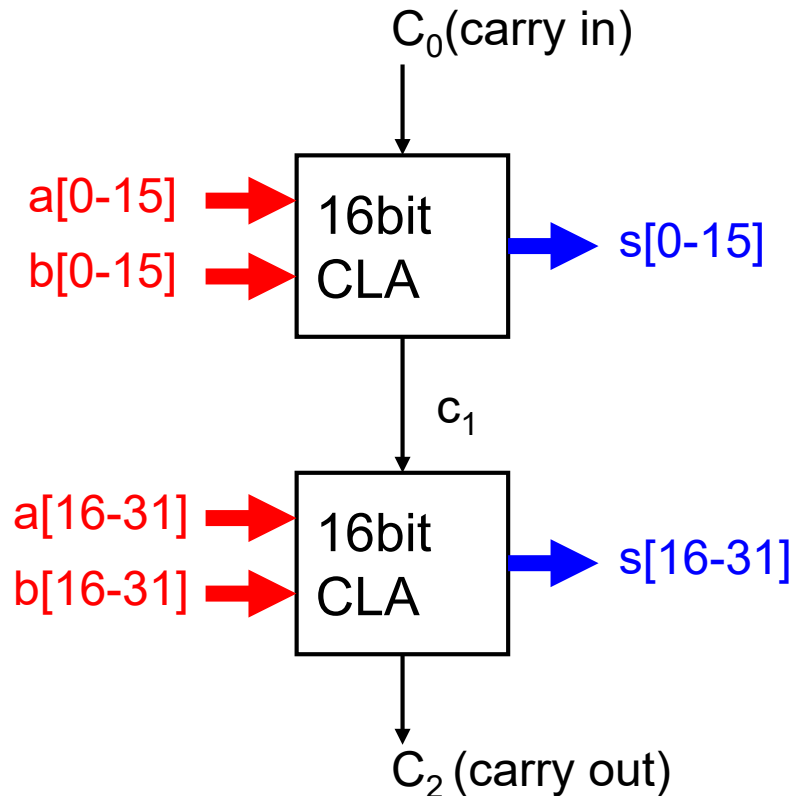
▶ 4ビットの場合と全く同じ回路

▶ 4ビット CLA の carry in を計算



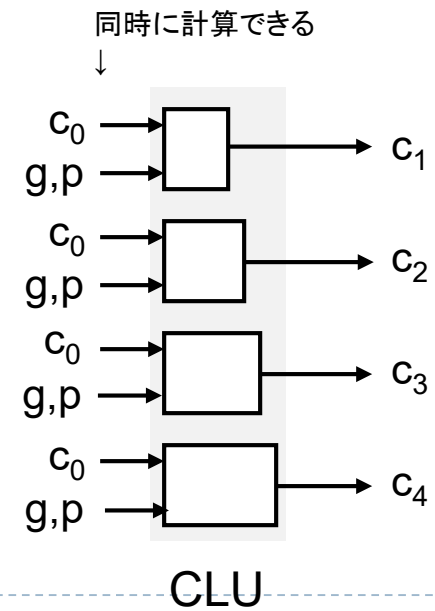
32ビット Carry Lookahead Adder

- ▶ 16ビット Carry Lookahead Adder を2つつなげる



注意：Carry Lookahead Adder の性能

- ▶ **ハードウェアで作った場合に速くなる回路**
 - ▶ $C_1 \sim C_4$ が並列に(同時に)計算できるため
 - ▶ C_i から C_{i+1} を逐次計算するより速い
- ▶ **シミュレータとして作った場合にはむしろ遅くなる**
 - ▶ プログラムを並列化していない
 - ▶ 回路がより複雑になるため
 - ▶ 計算量が増える



Ripple Carry Adder と Carry Lookahead Adderの速度の違い

- ▶ Carryが最下位(0)ビットから最上位(31)ビットまで伝播されるときに通過するゲート数を数える
 - ▶ ANDGateとORGateのおよその通過数を数える
 - ▶ 2つのゲートの実行時間は同じ
 - ▶ 並列に計算できる計算も考慮
- ▶ 加算器では、このCarryの伝播がクリティカルパスになる

Ripple Carry Adder の場合

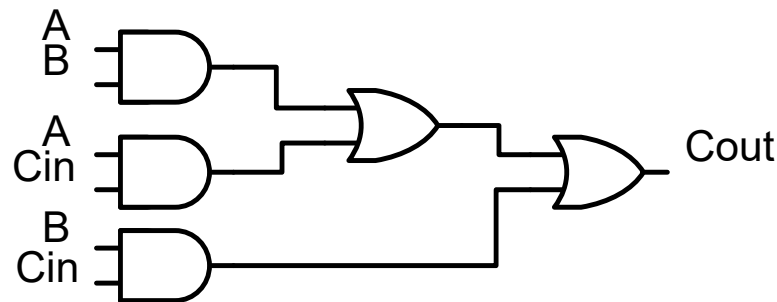
▶ 1ビット全加算器の C_{out} 計算

▶ $C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in}$

▶ ANDGate 並列3個 → 1回

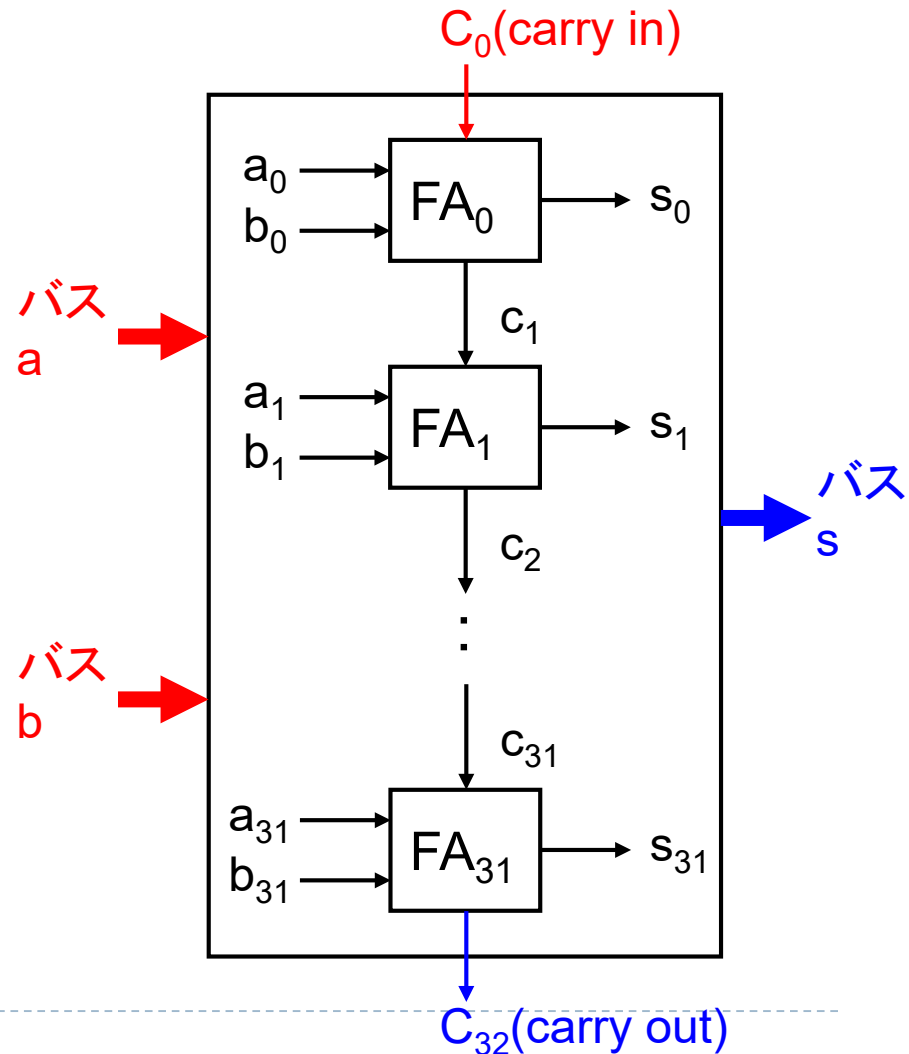
▶ ORGate 逐次2個 → 2回

▶ 計: **3ゲート**



▶ 32ビット伝搬ゲート数

▶ およそ $32 \times 3 =$ **96ゲート**



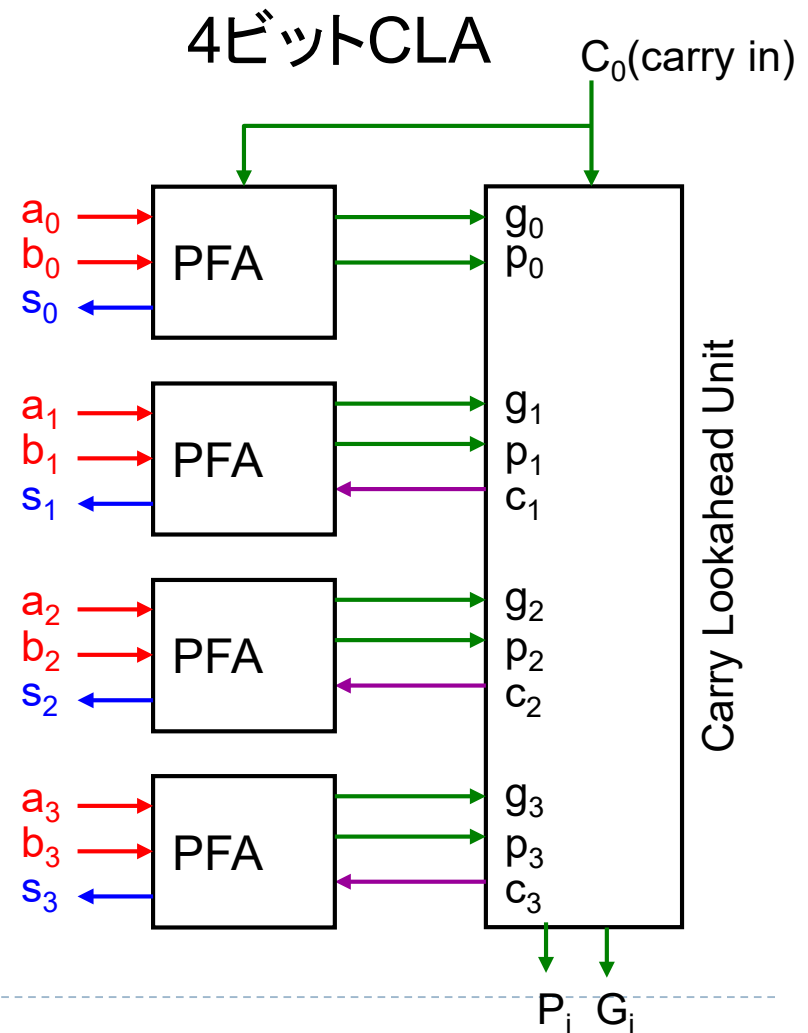
Carry Lookahead Adderの場合:1

1. PFAで g_i, p_i を計算

- ▶ $g_i = a_i \cdot b_i$, $p_i = a_i + b_i$ (並列)
- ▶ 32ビットで並列計算
- ▶ 計: 1 ゲート

2. 4ビットCLA内のCLUで G_i, P_i を計算

- ▶ $G_i = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$
- ▶ $P_i = p_0 p_1 p_2 p_3$

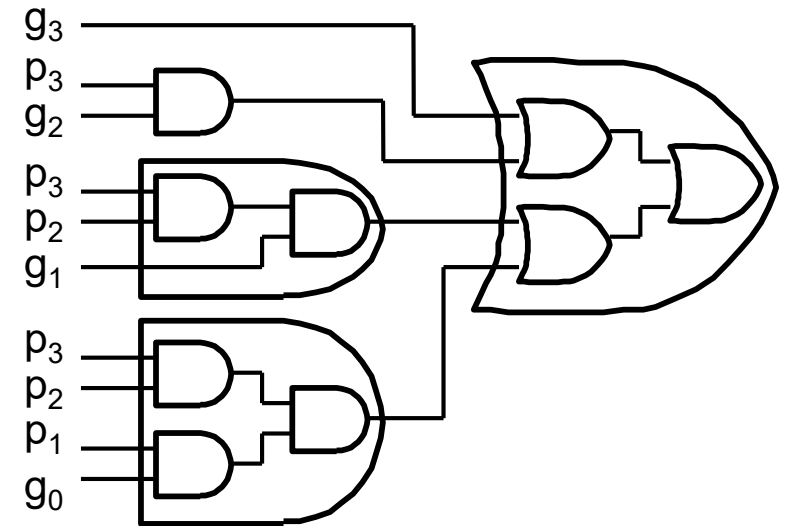


Carry Lookahead Adderの場合: 2

2. (つづき)

- ▶ 図のように回路を構成すれば、最長パスはGi計算の4ゲート
- ▶ 全8個の4ビットCLAで並列計算可能

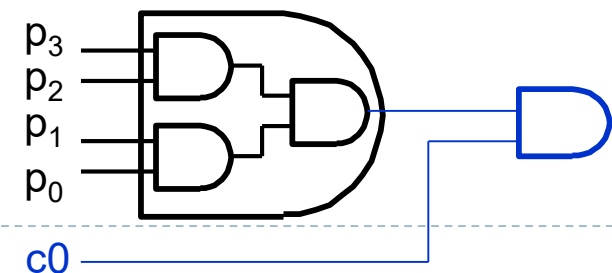
Gi 計算回路



3. 下位16ビットCLAで CarryOutを計算

- ▶
$$C_{out} = G3 + G2 P3 + G1 P2 P3 + G0 P1 P2 P3 + \underline{P0 P1 P2 P3} \ c0$$
- ▶ パスは5ゲート

Pi 計算回路



Carry Lookahead Adderの場合:3

4. 上位16ビットCLAで各4ビットCLAへのCarryInを計算
 - ▶ 最長パスは3と同じ 5ゲート
 - ▶ ただし最上位4ビットCLAへのCarryIn計算には 4ゲート
 - ▶ 合計の方には下の方を用いる
5. 4ビットCLAで各桁へのCarryInを計算
 - ▶ 最長パスは最上位ビットの4ゲート
- ▶ 以上で、32ビット各桁へCarryがいきわたる
 - ▶ 合計: およそ14ゲート

MIPSシミュレータの プログラムについて

MIPSシミュータプログラム

- ▶ こちらから雛形を提供します
 - ▶ (本日の資料としてOCW-iに掲載)
 - ▶ mips_simulator
 - ▶ src
 - プログラムのソースファイルを置くフォルダ
 - ▶ bin
 - バイナリファイル(実行ファイル)が生成されるフォルダ
 - ▶ obj
 - 各ソースファイルがコンパイルされた状態のものが保管されているフォルダ
 - ▶ mips.h
 - ヘッダファイル
 - ▶ Makefile
 - コンパイルを簡単にするためのもの
 - ▶ ディレクトリ構造は変更しないことを推奨

```
mips_student
├── Makefile
├── bin
├── mips.h
├── obj
└── src
    ├── adder.c
    ├── alu.c
    ├── control_unit.c
    ├── gate.c
    ├── memory.c
    ├── mips.c
    ├── reg_file.c
    ├── test.c
    └── word.c
```

ヘッダファイル

- ▶ 規模が大きい場合、もしくは管理を行いやすくするために一つのプログラムを複数のファイルに分ける
 - ▶ 通常、別の.cファイルで定義された変数や関数を使うことはできない
- ▶ 複数のファイル内で共有されるべき定数や変数、関数を定義する
 - ▶ 対応するcファイルではヘッダファイルをincludeする

```
#include "inc.h"

int inc(int a) {
    return a + 1;
}
```

inc.c

```
int inc(int a);
```

inc.h

```
#include <stdio.h>
#include "inc.h"
int main() {
    int a, b;
    a = 2;
    b = inc(a);
    printf("%d\n", b);
    return 0;
}
```

main.c

ヘッダファイル（続き）

▶ includeする時の記述方法について

▶ #include <stdio.h>

- ▶ C言語の標準ライブラリや“-I(ディレクトリ名)”で指定した場所にあるヘッダファイル

▶ #include “inc.h”

- ▶ コンパイル時のディレクトリにあるヘッダファイル

オブジェクトファイル

- ▶ ソースファイルをバイナリ(0,1の列)にしたもの
 - ▶ 各ファイルごとにオブジェクトファイルを生成し、結合することで実行ファイルが生成される
 - ▶ オブジェクトファイル単体では実行できず、他の必要なライブラリ等と結合する必要がある
 - ▶ コンパイル時に-cオプションをつけることで生成される

```
gcc -c inc.c //オブジェクトファイルinc.oを生成
gcc -c main.c //オブジェクトファイルmain.oを生成
gcc -o inc inc.o main.o //実行ファイルincを生成
./inc //実行ファイルincを実行
```

Makefile

▶ コンパイル作業の自動化

- ▶ 大規模なプログラムの実装において、一部のプログラムを変更しただけなのにすべてのプログラムをコンパイルし直すのは無駄
- ▶ コンパイルのたびに長いコマンドを書くのは面倒
- ▶ => Makefileに記述することで、makeコマンドで簡単にコンパイルが可能

ターゲット名

タブを入れる

コンパイル対象の
ファイル名

やりたいことを書く

ターゲット名を表す
(ここではinc)

```
inc : inc.o main.o
    gcc -o $@ inc.o main.o
%.o : %.c
    gcc -c -o $@ $<
clean :
    rm -rf *.o inc
```


Makefileの例 (MIPSシミュレータ)

```
CC = gcc #コンパイラ
CFLAGS = -Wall -O3 -g #コンパイルオプション
INCLUDE = -I. #ヘッダファイルがあるディレクトリを指定
BIN = ./bin
SRC = ./src
OBJ = ./obj
OBJ_TEST = $(OBJ)/test.o
OBJS = $(OBJ)/word.o $(OBJ)/gate.o $(OBJ_TEST)

all: # make もしくは make all と書いた時に実行される
    make test
test: $(OBJS)
    $(CC) $(CFLAGS) $(INCLUDE) -o $(BIN)/$@
$(OBJS) $(OBJ)/%.o : $(SRC)/%.c
    mkdir -p $(dir $@)
    $(CC) -c $(CFLAGS) $(INCLUDE) -o $@ $<
clean : # 生成されたものを削除
    rm -rf $(BIN)/*
    rm -rf $(OBJ)/*
```

変数の宣言と代入が可能
変数を参照するときには変数名の頭に\$をつける
#でコメントが書ける

Makefileの使用例 (MIPSシミュレータ)

```
#Makefileのあるディレクトリに移動
#コンパイル
$ make test
#生成された実行ファイルtestを実行する
$ ./bin/test
#実行結果が表示される

#生成された実行ファイルやオブジェクトファイルを削除する
#こちらは必要に応じて実行する
$ make clean
```

Makefileの編集 (MIPSシミュレータ)

- ▶ 新たに adder.c をコンパイルの対象として追加する

```
...  
OBJ = ./obj  
OBJ_TEST = $(OBJ)/test.o  
OBJS = $(OBJ)/word.o $(OBJ)/gate.o $(OBJ)/adder.o $(OBJ_TEST)  
// make testのところでは拡張子.oとしてファイルを見つけたい  
// 拡張子を.oとして、$(OBJ_TEST)の前に追加
```

課題の解き方

- ▶ 各課題に対応するファイルを編集します
 - ▶ 第5回: gate.c, adder.c
 - ▶ 第6回: alu.c
 - ▶ 第7回: reg_file.c, memory.c
 - ▶ 第8回: control_unit.c
 - ▶ 必要に応じてMakefileも
- ▶ main関数はtest.cにあります
 - ▶ 各課題でテスト関数を作り、main()から呼び出す
 - ▶ 最初は呼び出し部分が殆どコメントアウトされているはず

コンパイル・実行の方法

▶ コンパイル

- ▶ 下記の通りコマンドを実行、Makefileに従って自動的にコンパイルされる

```
$ make test
```

- ▶ エラーがなければ最終的に /bin/test に出力される)

▶ 実行

- ▶ 下記の通りコマンドを実行

```
$ ./bin/test
```

- ▶ コンパイルエラーを起こしていると、古いtestを実行することがあるので注意

演習室以外での環境について

- ▶ MacやLinux系はコマンドでmakeが使えると思います
 - ▶ 入っていない場合は各自で調べてインストールしてください
- ▶ Windowsの場合
 - ▶ cygwin: インストーラでmakeを追加
- ▶ どうしてもmakeが使えない場合
 - ▶ 逐一丁寧に全部書いてコンパイルしてください

```
gcc -Wall -O3 -g -o bin/test -I. src/word.c src/gate.c src/test.c
```

課題

課題1

- ▶ OR Gate, XOR Circuitの各関数を作成せよ
 - ▶ OR GateはAND Gateと同様にC言語の演算子を使ってよい
 - ▶ XOR Circuitは、各種Gateを組み合わせて作ること
 - ▶ すべての入力についてそれぞれが正しい結果を返すことを確認すること
 - ▶ n入力の論理回路に対して、出力は 2^n パターンある
 - ▶ gate.c の test_gate に追記する
 - AND Gate等のものを参考に

課題2

- ▶ 1ビット全加算器を作成せよ (FA)
 - ▶ テストケースを作成し、レポートにはその実行結果も含める
 - ▶ 8入力パターンすべて試すこと

課題3

- ▶ 32ビットの Ripple Carry Adder (RCA)を完成させよ
 - ▶ `adder.c`の`test_rca`にてテストをすること

課題4

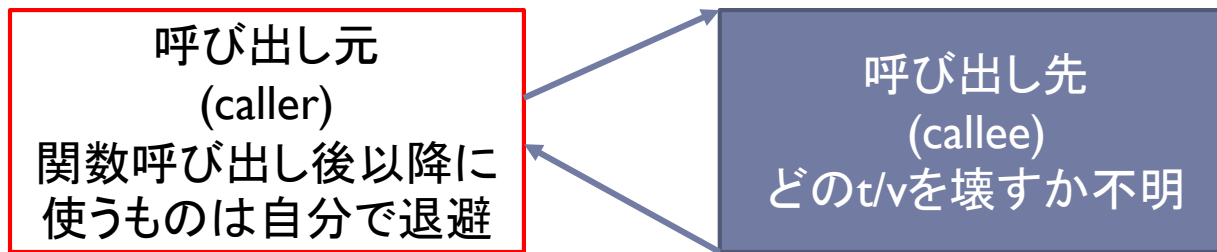
- ▶ 32ビットでのRCAとCLAを比較せよ
 - ▶ AND GateとOR Gateの数をもとにして、クリティカルパスを考えよ
 - ▶ 先のページの空白部分を適宜補完して、レポートにまとめる
 - ▶ RCAと比較しつつ、CLAの利点を述べよ

課題提出

- ▶ ✕ 切: 1/17 (金) 23:59 (この日は授業ありません)
- ▶ 提出物: 以下のファイルを1つのファイルに圧縮したもの
 - ▶ プログラムソース (gate.c と adder.c)
 - ▶ シミュレータ全体ではなく、関係するコード(変更したファイル)のみ提出すること
 - ▶ 作成したプログラムは今後も使用するため、十分にテストすること
 - ▶ ドキュメント
 - ▶ 実行結果
 - ▶ 課題4について
 - ▶ 感想等
- ▶ 質問等があれば compsys19@el.gsic.titech.ac.jp まで
 - ▶ 課題のレポートやコメントに書かれていると、返信が遅くなります

第4回課題のヒント

- ▶ どのレジスタを保存しなければならないか?
 - ▶ 呼び出し元・呼び出し先の中身がブラックボックスでも動く
 - ▶ caller-save (t/v/a)



- ▶ callee-save (s/ra)

