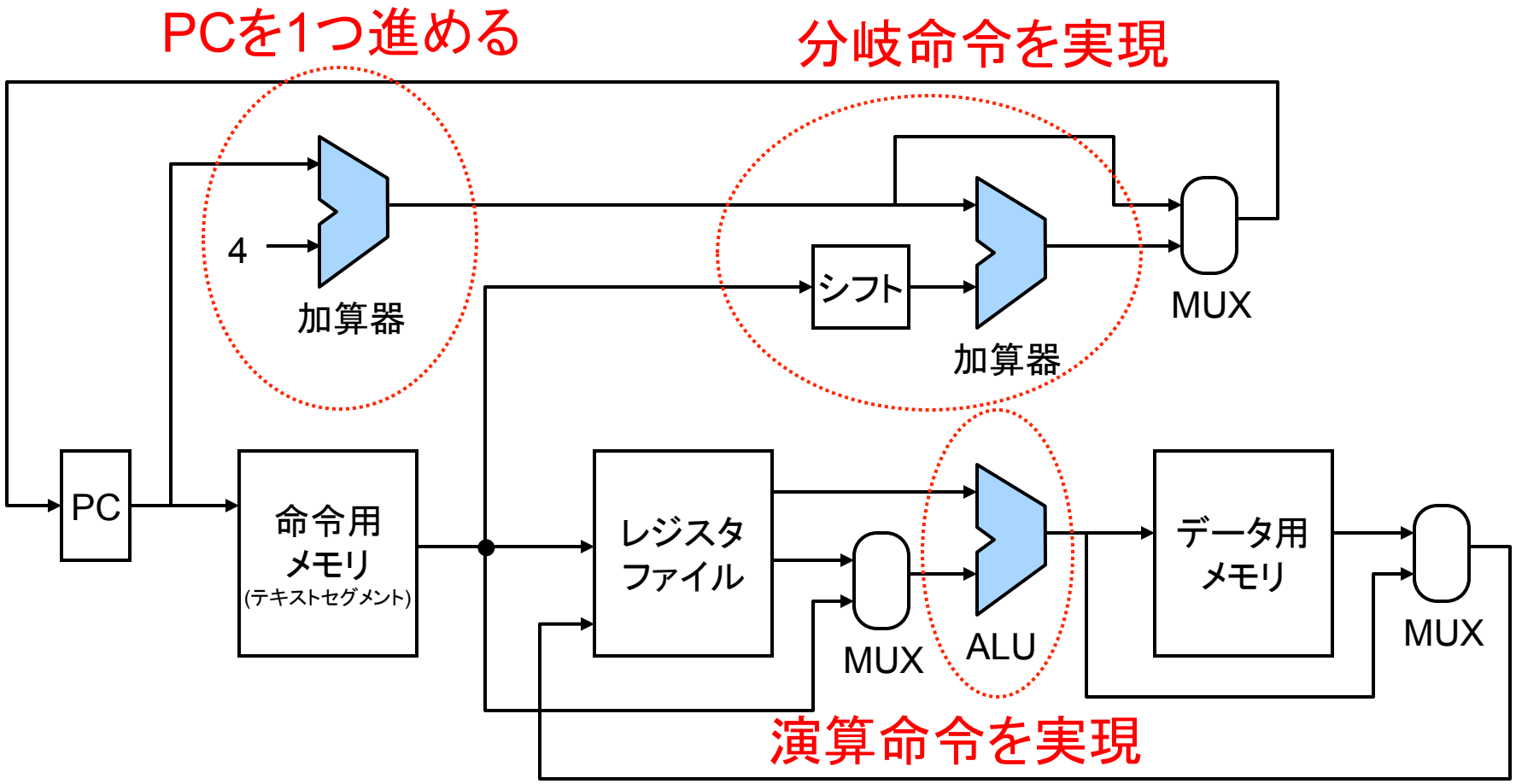
2012年度
計算機システム演習 第8回

計算機システムTA 福田圭祐（松岡研究室）

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
   - メモリアクセス命令
   - 分岐命令

MIPSシミュレータ　概略図



課題 1

- 32ビットの Ripple Carry Adder（RCA クラス）を完成させよ
  - RCADriver クラスを作ってテストすること

課題2

- 4ビットの Carry Lookahead Adder（CLA4 クラス）を完成させよ
  - CLA4Driver クラスを作ってテストすること

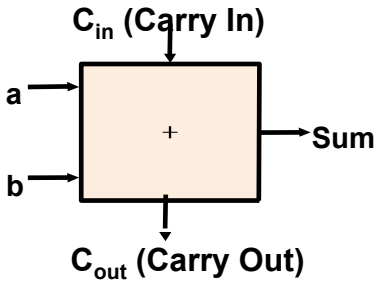
課題3 （オプション）

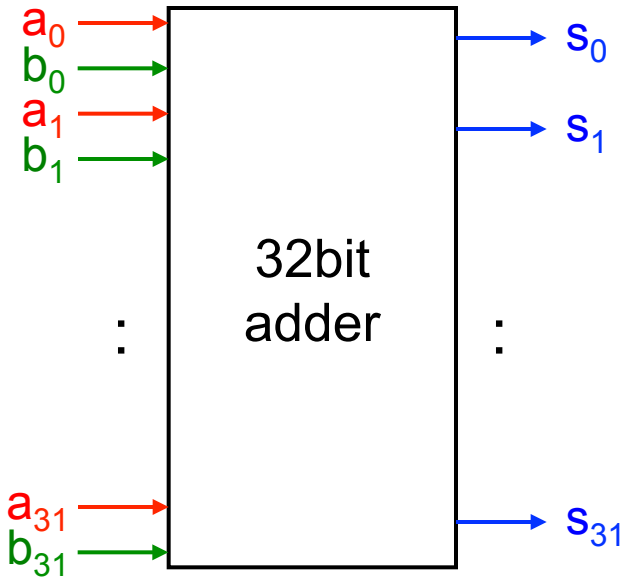
- 32ビットの Carry Lookahead Adder（CLA32 クラス）を作成せよ
  - CLA32Driver も作成し、テストすること
  - CLA(32bitのBUS) => CLA(16bitのBus) + CLA(16bitのBus)
    - バスの一部を取り出す Bus クラスのメソッドは以下の通り

```
// index番目からnum本の配線を取り出したバスを返す
public Bus getSubset(int index, int num) {
  Bus subBus = new Bus(num);
  for (int i = 0; i < num; i++)
    subBus.paths[i] = this.paths[index + i];
  return subBus;
}
```

本日の内容

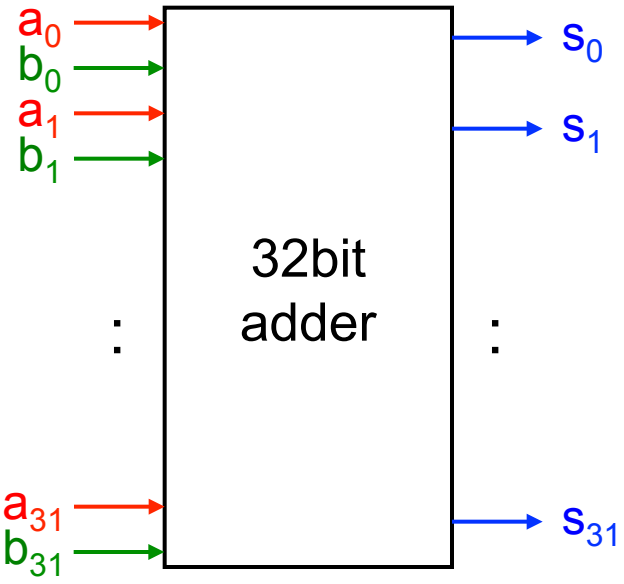
- ALUの作成
  - 1ビット加算器の作成
  - 32ビット加算器の作成
    - Ripple Carry Adder
    - Carry Look-ahead Adder
  - 1ビットALU
  - 32ビットALU





32ビット加算器

- 入力配線
  - 32本　2組
- 出力配線
  - 32本



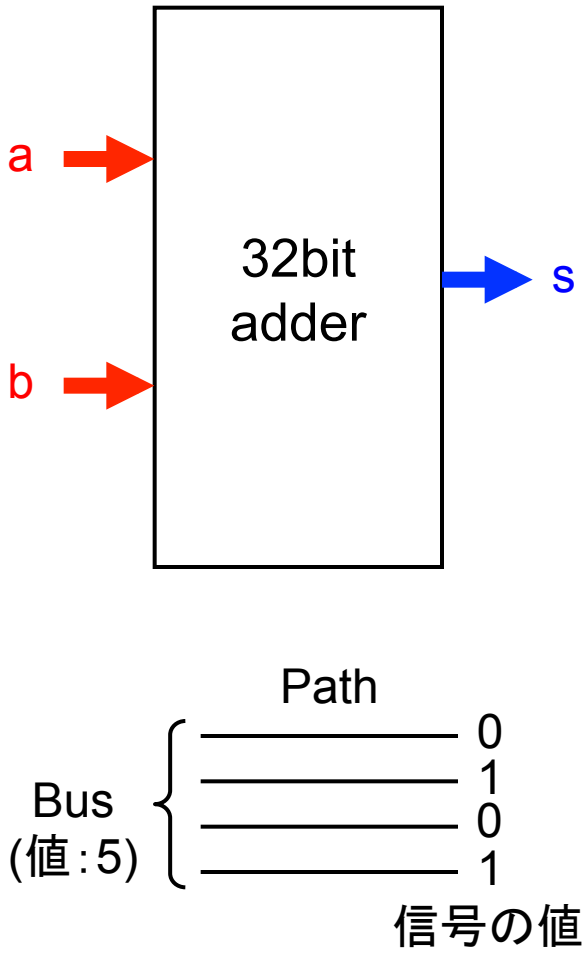
- 配線(Path)が多すぎて取り扱いにくい
  - 信号を1つ1つ設定するのは大変

32*3 = 96引数

コンストラクタ (Path inA0,…, Path inA31,Path inB0, … Path inB31, Path outS0, … Path outS31)

Bus クラス

- 配線 (Path) の集合をバスとして表現
  - Path の配列を持つ
- Busクラスの仕様
  - コンストラクタ： Bus (int n)
    - n: Pathの数
  - setValue(long val)
    - 指定された数値を二進数で表すように各配線の信号を設定
    - 例) val=5 => 101
  - getValue()
    - 各配線の信号が表す二進数の数値を返す
    - 例) 101 => 5
  - getPath(int i)
    - path[i]を返す

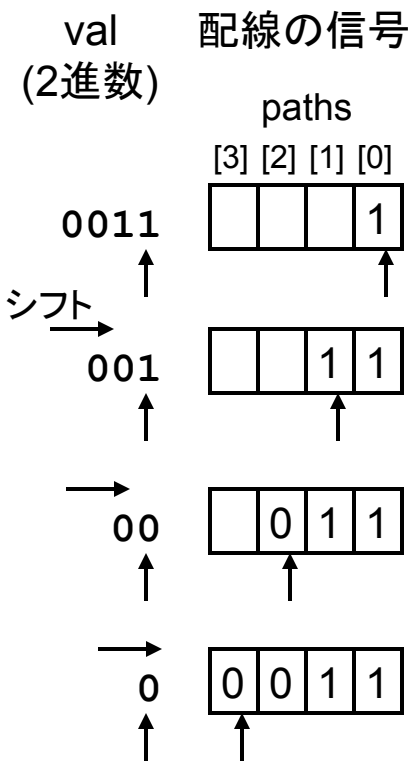


Bus クラスのコード （1/2）

0x: 16進数　　L: long型

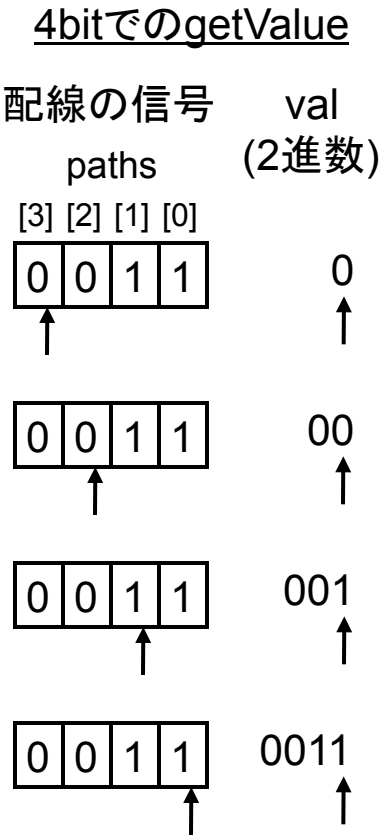
```
public class Bus {
  Path[] paths;
  // nビットのバスを作成
  public Bus(int n) {
    paths = new Path[n];
    for (int i = 0; i < n; i++)
      paths[i] = new Path();
  }
  // valの値を表すように配線に信号を設定
  public void setValue(long val) {
    for (int i = 0; i < paths.length; i++) {
      if ((val & 0x1L) != 0)   // ビットが立っていたら
        paths[i].setSignal(new Signal(true));
      else
        paths[i].setSignal(new Signal(false));
      val >>= 1;   // 1ビット右シフト
  }
```

4bitでのsetValue(3)



Bus クラスのコード　(2/2)

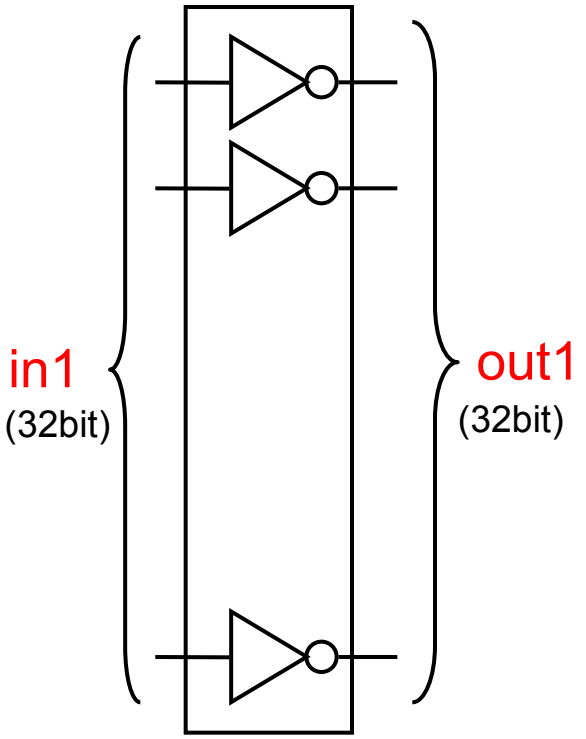
```
    :
  // 配線の信号によって表される整数値を返す
  public long getValue() {
    long val = 0;
    for (int i = paths.length-1; i >= 0; i--) {
      val <<= 1;  // 1ビット左シフト
      if (paths[i].readSignal().getValue())
        val += 1L;  // 信号が1ならビットを立てる
    }
    return val;
  }
  // i番目の配線を取り出す
  public Path getPath(int i) {
    return paths[i];
  }
}
```



Busクラスの使用例(1/2)

- Busクラスによって引数が2つに減る
  - Bus クラスを使う回路：Complement1 クラス

```
// 32ビットの1の補数表現を得る回路
public class Complement1 {
  NOTGate[] not1 = new NOTGate[32];
  public Complement1(Bus in1, Bus out1) {
    // i番目の配線の信号を反転する
    for (int i = 0; i < 32; i++)
      not1[i] = new
NOTGate(in1.getPath(i),
              out1.getPath(i));
  }
  public void run() {
    for (int i = 0; i < 32; i++)
      not1[i].run();
  }
}
```



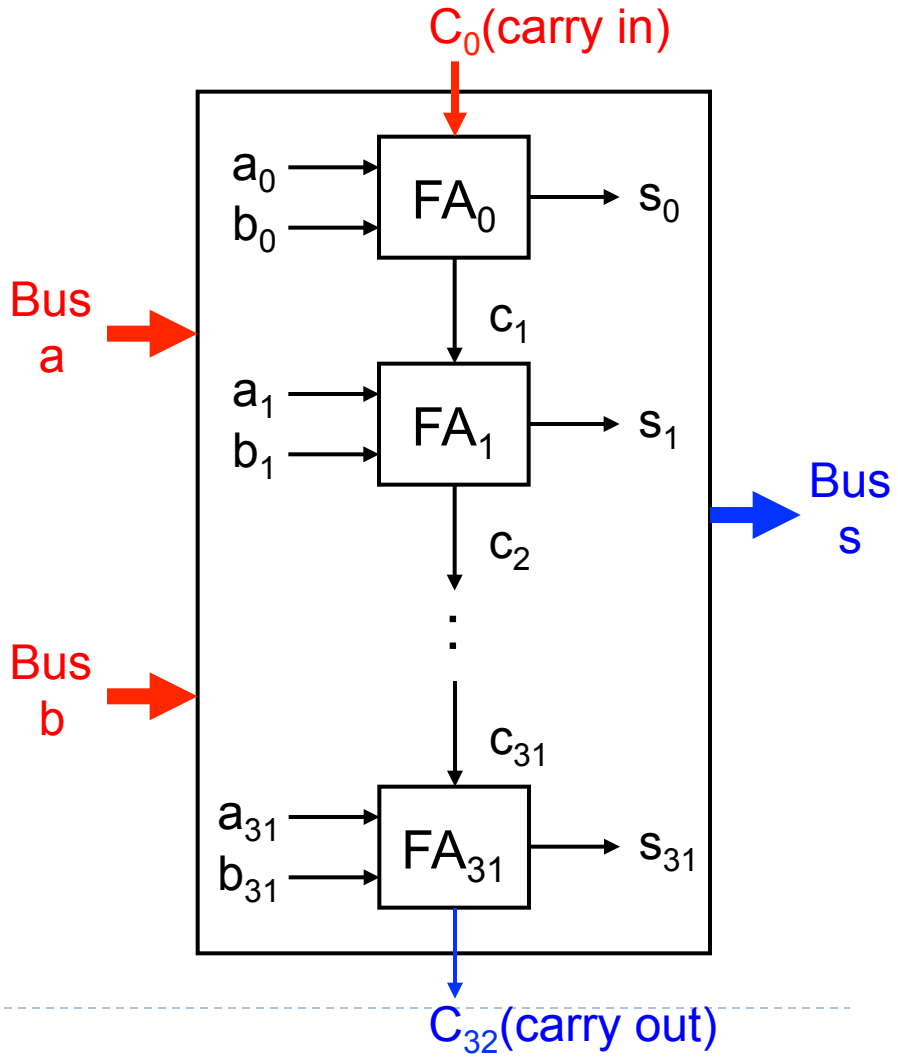
Bus クラスの使用例(2/2)

```
public class Complement1Driver {
  public static void main(String[] args) {
    Bus in1 = new Bus(32);  // 32ビットのバス
    Bus out1 = new Bus(32);
    Complement1 c1 = new Complement1(in1, out1);

    in1.setValue(100L);  // バスの値として100を設定
    c1.run();  // 回路を実行
    System.out.println(out1.getValue());  // バスの値を読み出す
  }
}
```

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_1をFA_1に伝播
  - FA_1を実行
  - ：
  - キャリーC_{31}をFA_{31}に伝播
  - FA_{31}を実行



RCA クラス

```
public class RCA {
  FA[] adder = new FA[32];  // 1ビット全加算器

  public RCA(Bus a, Bus b, Path carryIn,
             Bus s, Path carryOut) {
    Path[] c = new Path[?];
    for (int i = ?; i < ?; i++) {
      // 回路を作成
    }
  }
  public void run() {
    for (int i = ?; i < ?; i++) {
      // 回路を実行
    }
  }
}
```

RCADriver クラスの例
(RCAのテスト用プログラム)

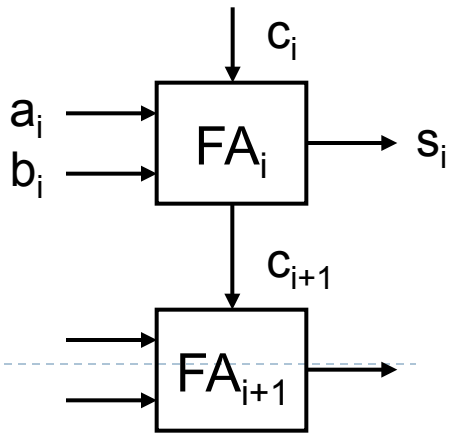
```
public class RCADriver {
  public static void main(String[] args) {
    Bus a = new Bus(32);
    Bus b = new Bus(32);
    Path carryIn = new Path();
    Bus s = new Bus(32);
    Path carryOut = new Path();

    RCA adder = new RCA(a, b, carryIn, s, carryOut);

    a.setValue(1000L);   // 例
    b.setValue(2000L);   // 例
    adder.run();
    System.out.println(s.getValue());   // 3000
  }
}
```

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～c_4が決まる

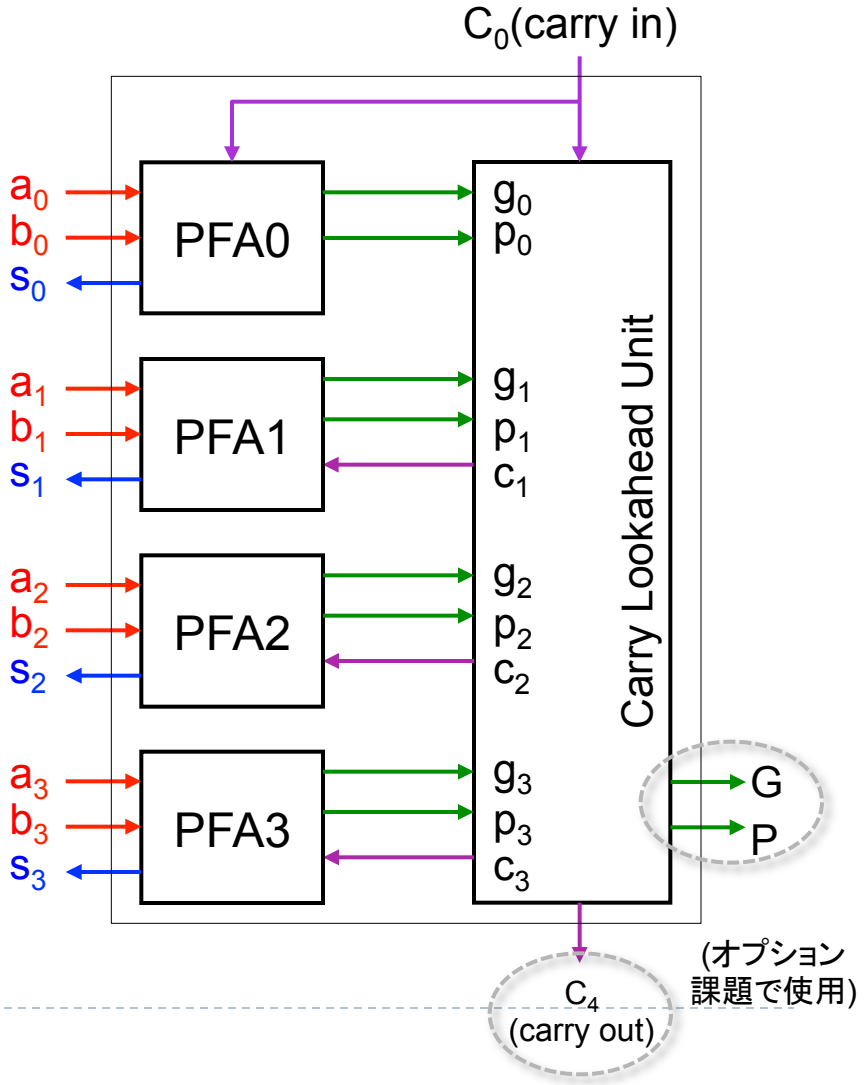
$g_i = a_i \cdot b_i$
$p_i = a_i + b_i$

(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～g_3, p_0～p_3, c_0
    - 出力: c_1～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 xor b_i xor c_i
- Carry Lookahead Unit (CLU)
  - c_1～c_4 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～c_4 を計算
  3. PFA: s_i を計算
     - ここで正しいs_iが出力

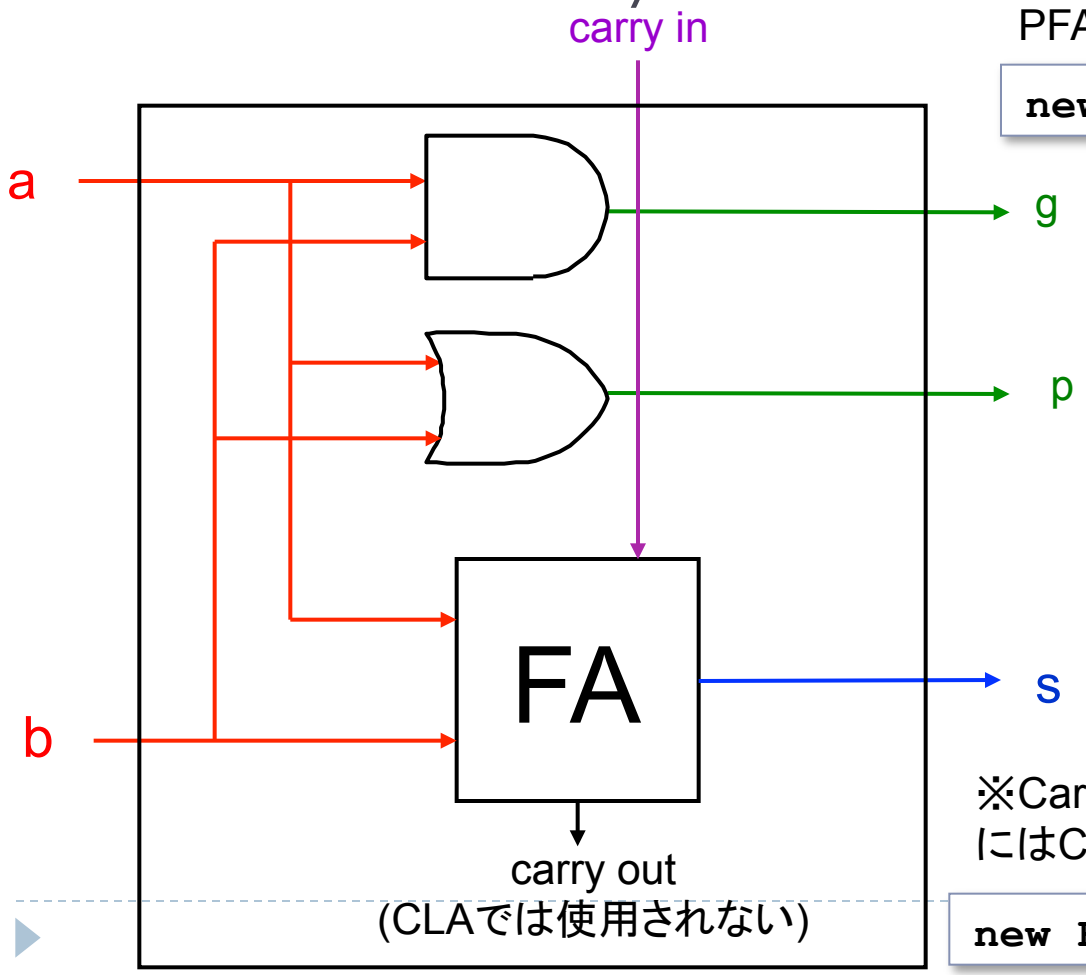


CLA4 クラス

```
public class CLA4 {
  // Bus a,b,s: 4ビット
  public CLA4(Bus a, Bus b, Path carryIn,
              Bus s, Path carryOut) { //Option課題ではG,Pも
    // g, pを計算する回路
    Path[] g = new Path[4]; //Busでも可
    Path[] p = new Path[4]; //Busでも可
    PFA[] pfa = new PFA[4];
    // cを計算する回路(Carry Lookahead Unit)
    Path[] innger_c = new Path[5];
    CLU clu = new CLU(g, p, c);
    // sを計算する回路
  }
  public void run() {
    // 回路を実行
  }
}
```

PFAクラス

- 以下のようにPFAクラスを実装
  - 注意： PFA1~3のcarry inはない



※PFA0用のcarry in.
PFA1~3は以下のように記述すればよい

```
new PFA(a,b,new Path(),g,p,s)
```

$g_i = a_i \cdot b_i$
$p_i = a_i + b_i$

※Carry outはCLUで計算するため、実際にはCLAでは使用されない

```
new FA(a,b,carryIn,s,new Path())
```

CLU クラス (Carry Lookahead Unit)

```
public class CLU {
  ANDGate and1, and2, …;
  ORGate or1, or2, …;
  public CLU(Path[] g, Path[] p, Path[] c) {
    // 入力:(g, p, c[0]) から出力:c[1]～c[4]を計算
    Path inner1 = new Path();
    and1 = new ANDGate(p[0], c[0], inner1);
    or1 = new ORGate(g[0], inner1, c[1]);  // c[1]を計算する回路
      : //以下 c[2]~c[4]を計算する回路
  }
  public void run() {
    and1.run();
    or1.run();
      :
  }
}
```

ANDGateN、ORGateN クラス

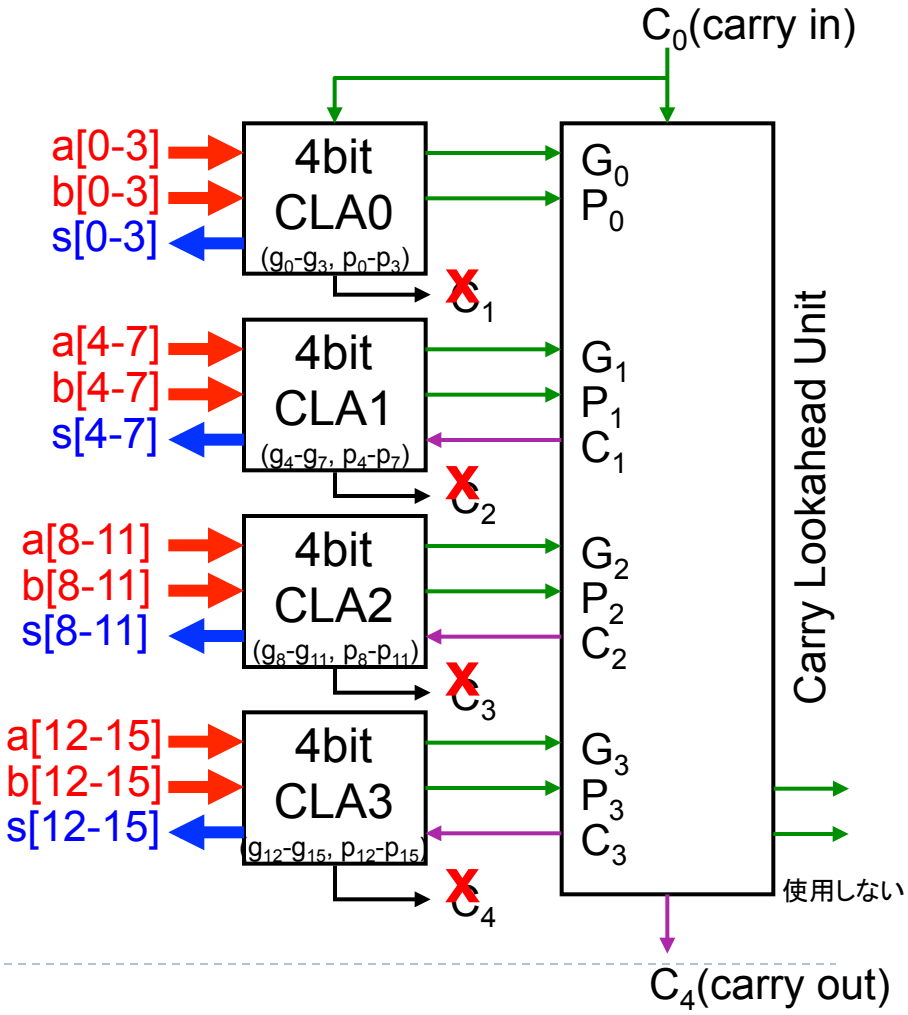
- n 入力の ANDGate（CLU クラスで利用）

```
public class ANDGateN {
  Path[] ins; Path out1;
  public ANDGateN(Path[] ins, Path out1) {
    this.ins = ins;
    this.out1 = out1;
  }
  public void run() {
    boolean val = true;
    for (int i = 0; i < ins.length; i++)
      if (!ins[i].readSignal().getValue()) {
        val = false;
        break;
      }
    out1.setSignal(new Signal(val));
  }
}
```

- ORGateNは省略

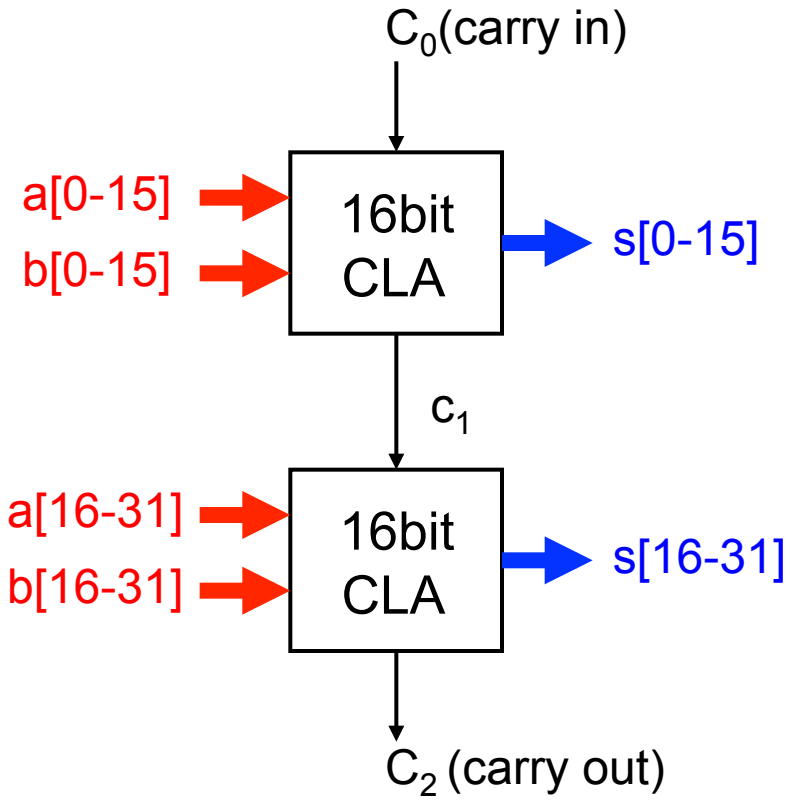
16ビット Carry Lookahead Adder

- 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
    - $C_1 = G_0 + P_0 C_0$
    - $C_2 = G_1 + P_1 C_1 = G_1 + G_0 P_1 + P_0 P_1 C_0$
    - $C_3 = \ldots, C_4 = \ldots$:
- 16bit CLAにおけるCLU
  - 4ビットの場合と全く同じ回路
  - 各 4ビット CLA の carry in を計算



32ビット Carry Lookahead Adder

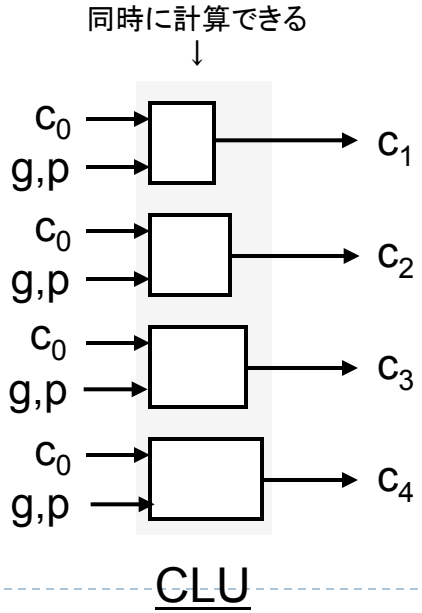
- 16ビット Carry Lookahead Adder を2つつなげる



注意：Carry Lookahead Adder

- ハードウェアで作った場合に速くなる回路
  - C_1 ～ 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
- Java で作った場合にはむしろ遅くなる
  - 並列計算していないため
    - 結局、逐次計算
  - 回路がより複雑になるため
    - 計算量が増える

ハードウェアを作っているつもりで実装する



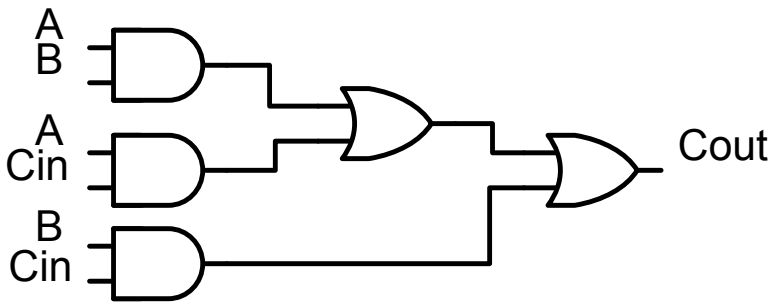
Ripple Carry Adder と Carry Lookahead Adderの速度の違い

- 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

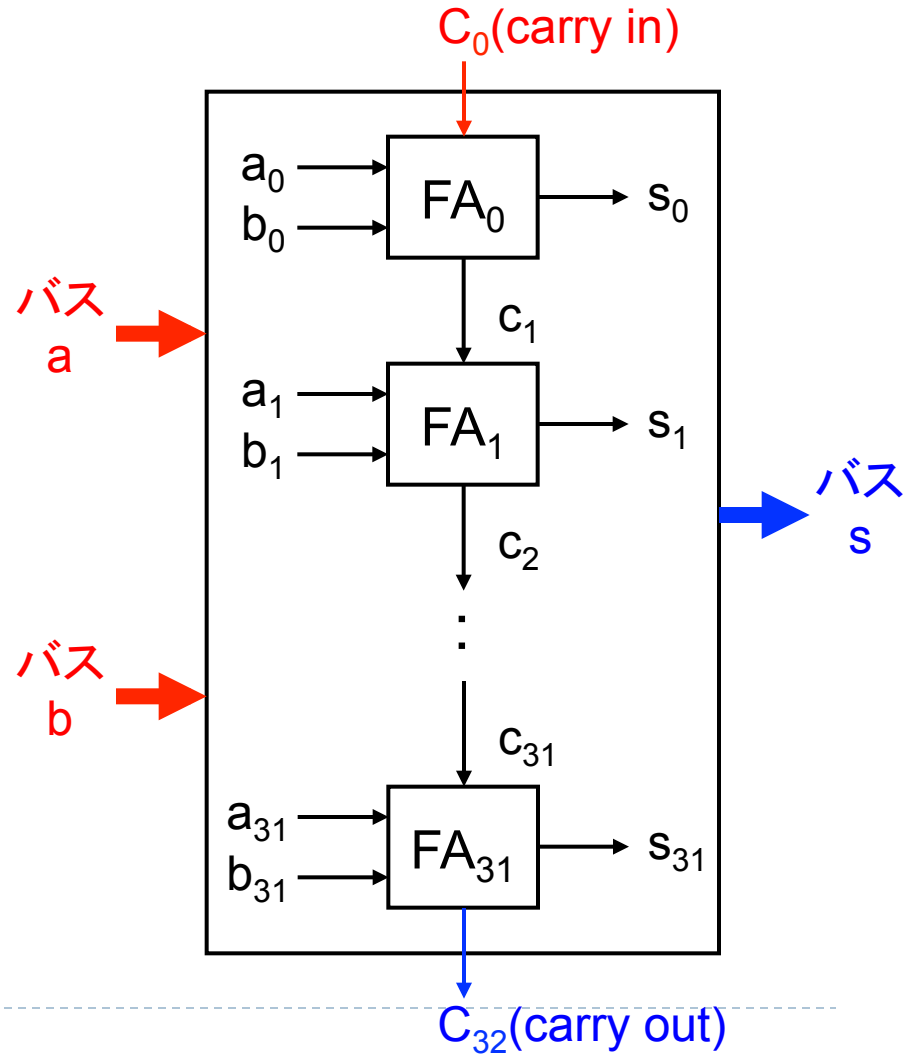
- 加算器では、このCarryの伝播がクリティカルパスになる

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

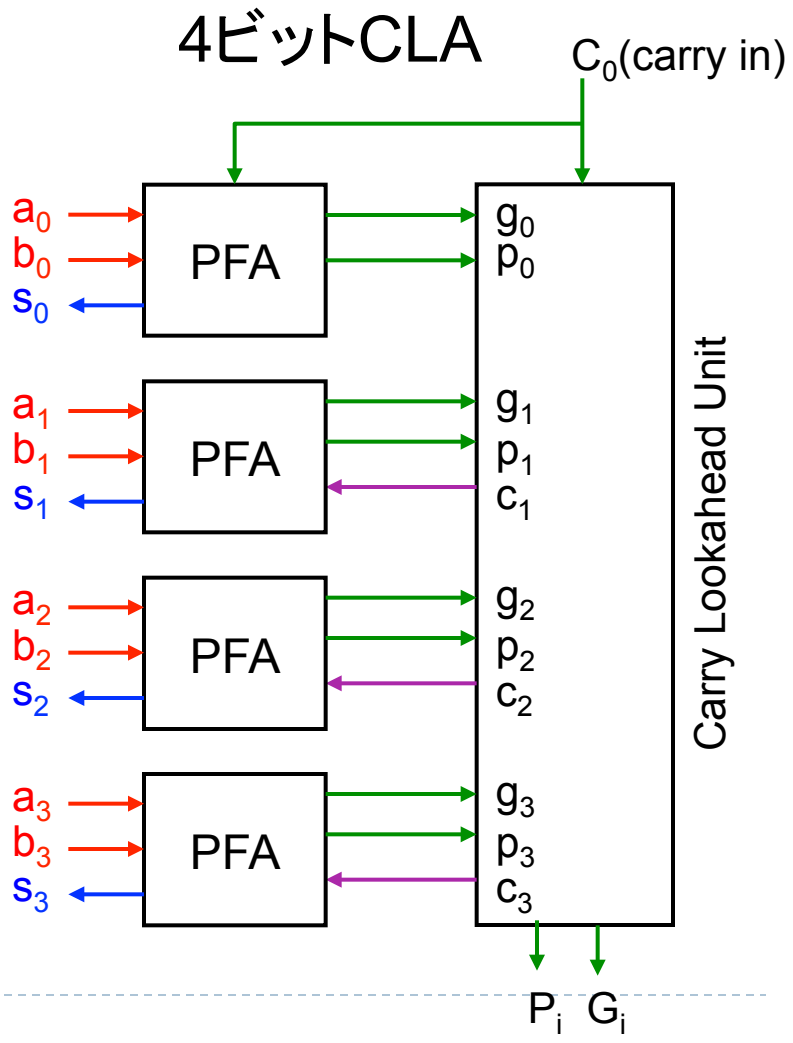


- 32ビット伝搬ゲート数
  - およそ32x3=96ゲート



Carry Lookahead Adderの場合:1

1. PFAでgi, piを計算
   - gi = ai･bi, pi = ai+bi （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Gi,Piを計算
   - $G_i = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$
   - $P_i = p_0 p_1 p_2 p_3$

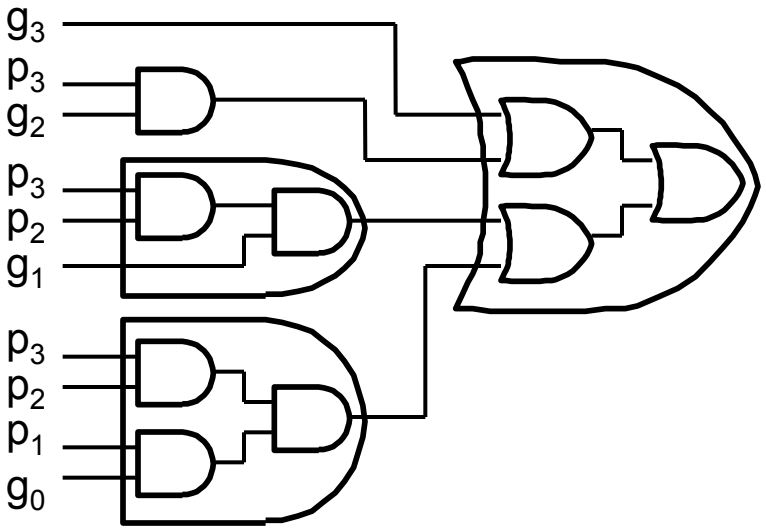


Carry Lookahead Adderの場合: 2

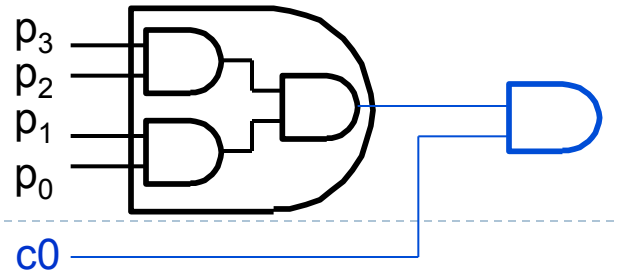
2. (つづき)
   - 図のように回路を構成すれば、最長パスはGi計算の 4ゲート
   - 全8個の4ビットCLAで並列計算可能

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 + P0 P1 P2 P3 c0
   - パスは 4+1ゲート=5ゲート

Gi 計算回路



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
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 5ゲート

- 以上で、32ビット各桁へCarryがいきわたる
  - 合計： 1+ 4 + 5 + 4 + 5 = およそ19ゲート

結果

- RCA : CLA = 96 : 19 ≒ 5 : 1
  - ゲート数はCarry Lookahead Adderの方が少ない
- なぜ、Carry Lookahead Adder は早いか
  - クロックサイクルは最低でもゲート時間分は必要
    - RCA：96ゲート時間、CLA：19ゲート時間
    - RCAの場合、96ゲート時間待たないと、回路が安定しない（正確な演算結果を出せない）
  - CLAのほうが、クロックサイクルを短く設定できる

課題

課題1（再掲）

- 32ビットの Ripple Carry Adder（RCA クラス）を完成させよ
  - RCADriver クラスを作ってテストすること

課題2（再掲）

- 4ビットの Carry Lookahead Adder（CLA4 クラス）を完成させよ
  - CLA4Driver クラスを作ってテストすること

課題3 （オプション） （再掲）

- 32ビットの Carry Lookahead Adder（CLA32 クラス）を作成せよ
  - CLA32Driver も作成し、テストすること
  - CLA(32bitのBUS) => CLA(16bitのBus) + CLA(16bitのBus)
    - バスの一部を取り出す Bus クラスのメソッドは以下の通り

```
// index番目からnum本の配線を取り出したバスを返す
public Bus getSubset(int index, int num) {
  Bus subBus = new Bus(num);
  for (int i = 0; i < num; i++)
    subBus.paths[i] = this.paths[index + i];
  return subBus;
}
```

課題提出

- 〆切：6/29 (金) 23:59
- 提出物：以下のファイルを圧縮したもの
  - ドキュメント(pdf,plain txt,wordなんでも可)
    - 製作方針、説明など
    - テスト結果　(テスト内容とその結果)
    - 感想等
  - プログラムソース一式 (ソースコードのみ)
    - 必ず…Driver.javaクラスも含める
    - .classファイルは含めないこと