2012年度
計算機システム演習 第12回

計算機システムTA　福田圭祐（松岡研究室）

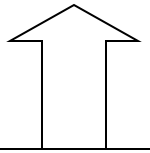
日程確認

- PC組み立て演習
  - 日時：7/20（金）15:05~　(2時間程度)
  - 場所：西7演習室
    - （福田は出張中につきTAは別の人です）
- 試験
  - 日時：8/6（月）3,4限
  - 場所：W833（授業と異なるので注意）

MIPSシミュレータ構築の流れ

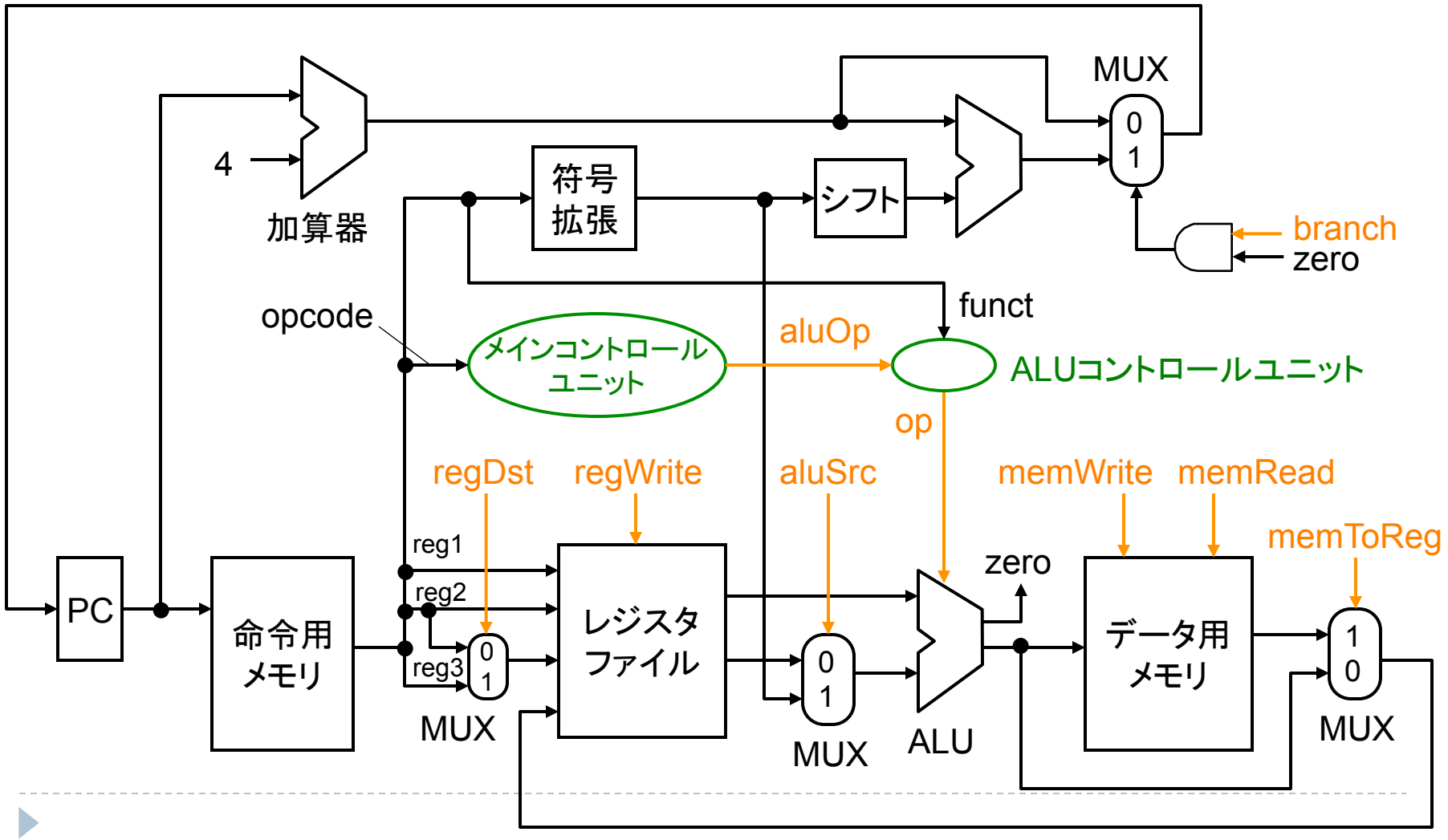
1. ALUの作成
2. レジスタファイル
3. メモリ領域
   - 命令用メモリ
   - データ用メモリ
4. PCの作成
5. メインコントロールユニット
6. ALUコントロールユニット

MIPS (ver.1)
とする

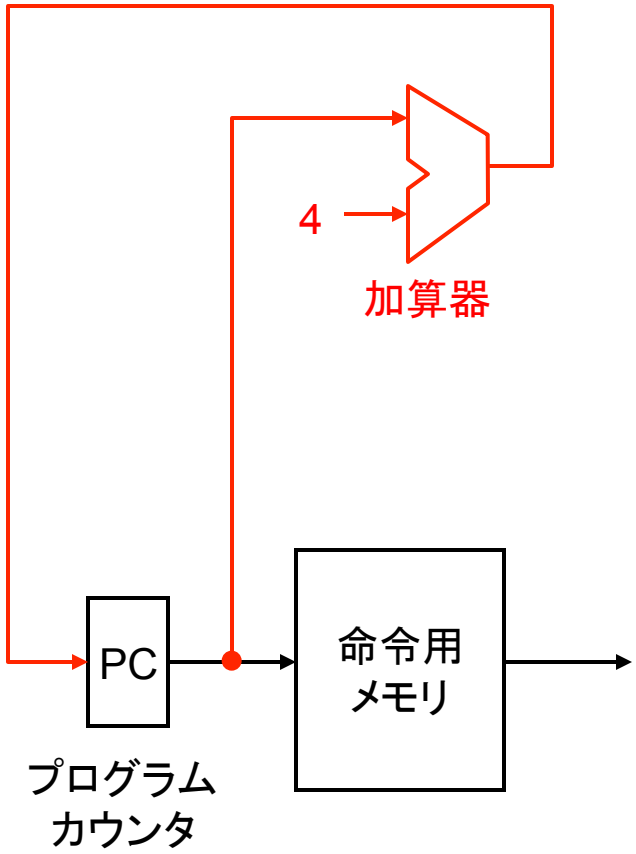


7. 機能拡張
   - メモリアクセス命令
   - 分岐命令

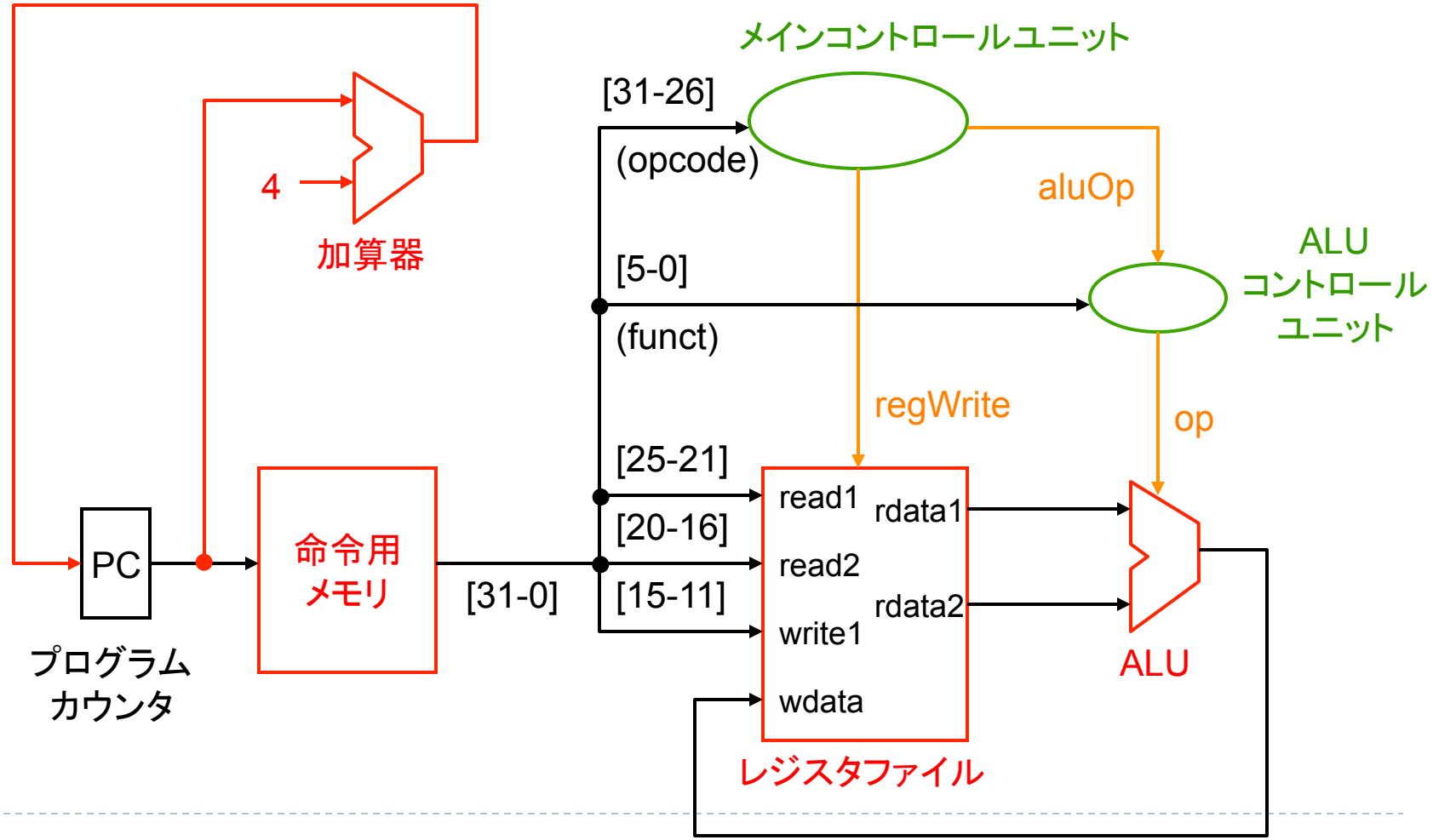
MIPSシミュレータの完成図



MIPS クラス（ver. 2）：
命令を連続実行するための回路

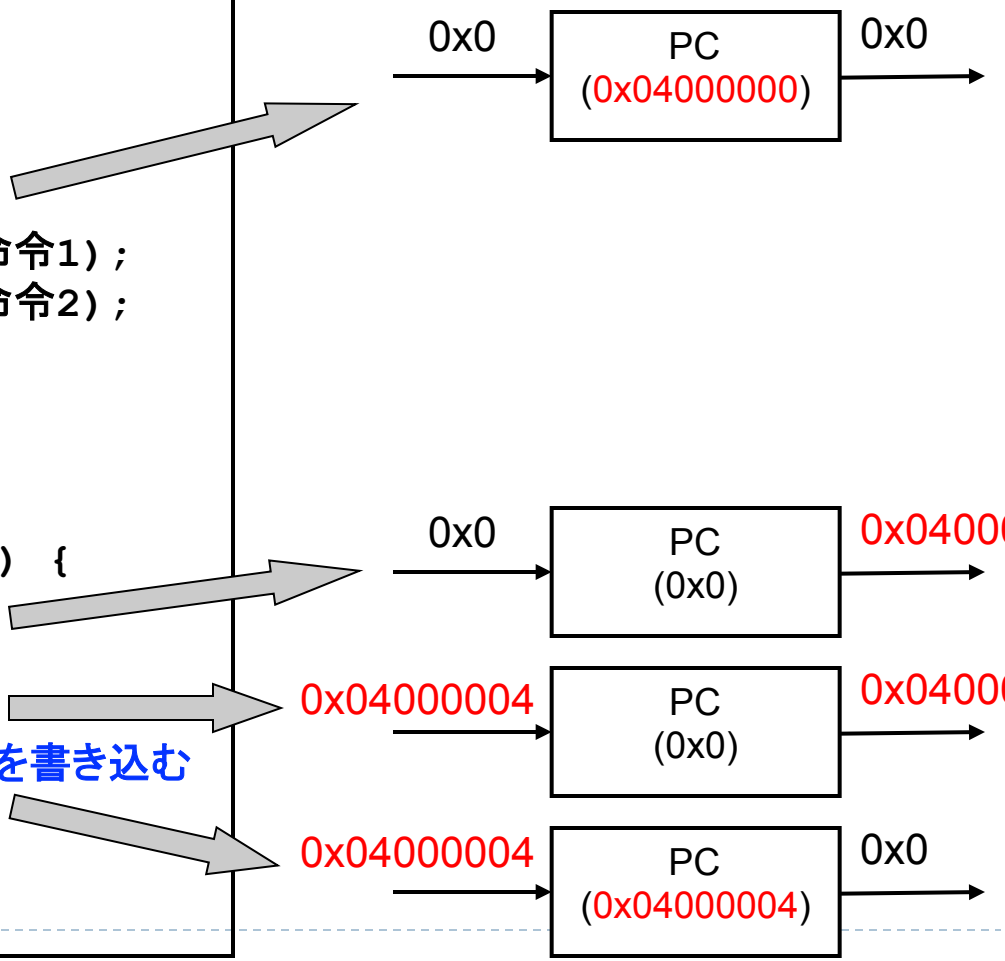


連続実行を追加した回路



MIPS クラス（ver. 2）

```
public class MIPS {
  // ALUを追加
  public MIPS() {
      :
    pc.setValue(0x04000000);
    iMem.setInst(0x04000000, 命令1);
    iMem.setInst(0x04000004, 命令2);
      :
  }
  public void run() {
    // for文で実行回数を指定
    for (int i = 0; i < 2; i++) {
      pc.run();
        :
      adder1.run();
      pc.run();  // 新しいPCの値を書き込む
    }
  }
}
```



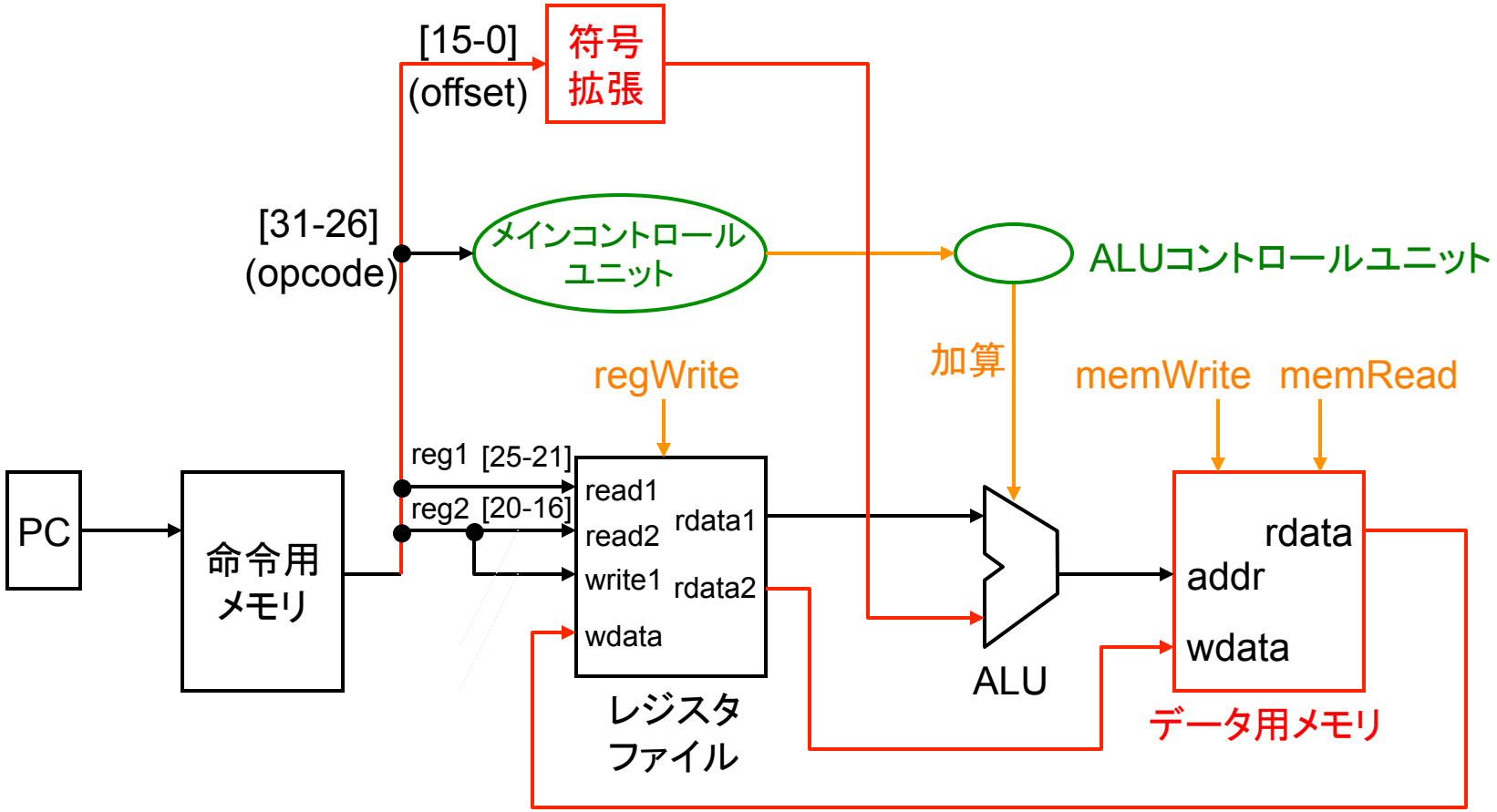
MIPS クラス(ver. 3): メモリアクセス命令のフォーマット

- **`lw/sw $x, offset($y)`**

opcode	reg1	reg2	offset
[31-26]	[25-21] $y	[20-16] $x	[15-0]

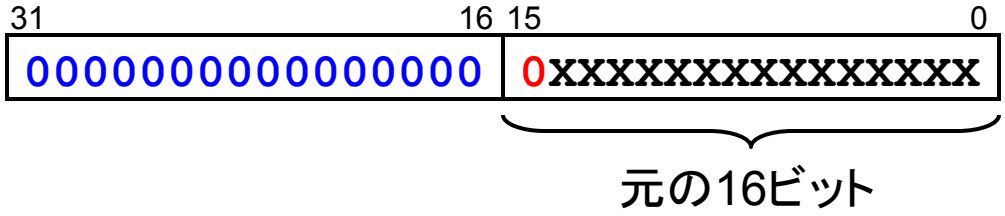
  - opcode
    - lw: 0x23(100011), sw: 0x2b(101011)
  - reg1 : ベースレジスタ($y)の番号
    - $y + offset のアドレスにアクセス
  - reg2: 入出力レジスタ($x)の番号
    - $x に読み込み、または、$x の値を書き込み
  - offset
    - 相対アドレス

メモリアクセス(のみ)のための回路

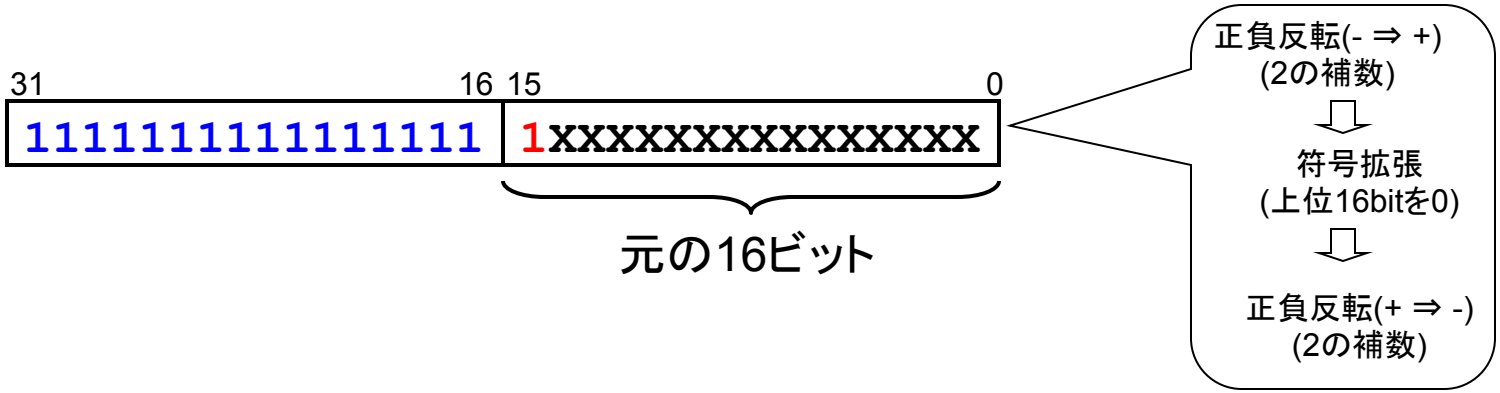


符号拡張

- 16ビット値を32ビット値に、符号が変わらないように変換する
  - 正の値（最上位ビットが0）なら上位16ビットを0に



  - 負の値（最上位ビットが1）なら上位16ビットを1に



メモリへの書き込み（sw）の挙動

1. レジスタファイルからベースレジスタ（$y）と入力レジスタ（$x）の値を読み出す
2. 命令中の offset の値を 32 ビットに符号拡張する
   - ALUの入力は32bit => アドレス計算をするためには32bitで有る必要がある
3. ALU で $y と offset の値を足す
4. 計算したアドレスに $x の値を書き込む

命令	regDst	aluSrc	memTo Reg	reg Write	mem Read	mem Write	branch	aluOp1	aluOp0
sw	X	1	X	0	0	1	0	0	0

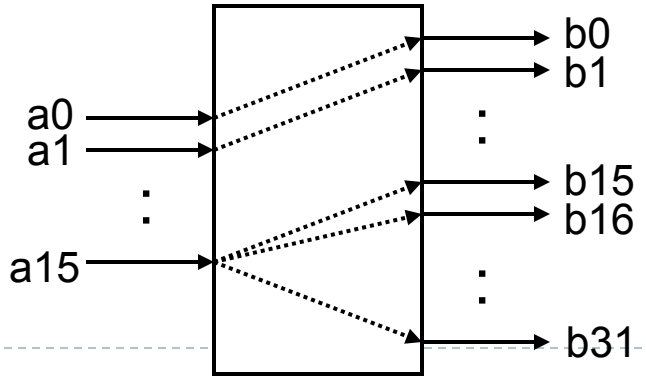
メモリからの読み出し（lw）の挙動

1. レジスタファイルからベースレジスタ（$y）の値を読み出す
2. offset の値を 32 ビットに符合拡張する
   - ALUの入力は32bit => アドレス計算をするためには32bitで有る必要がある
3. ALU で $y と offset の値を足す
4. 計算したアドレスの値をメモリから読み出す
5. その値を出力レジスタ（$x）に書き込む

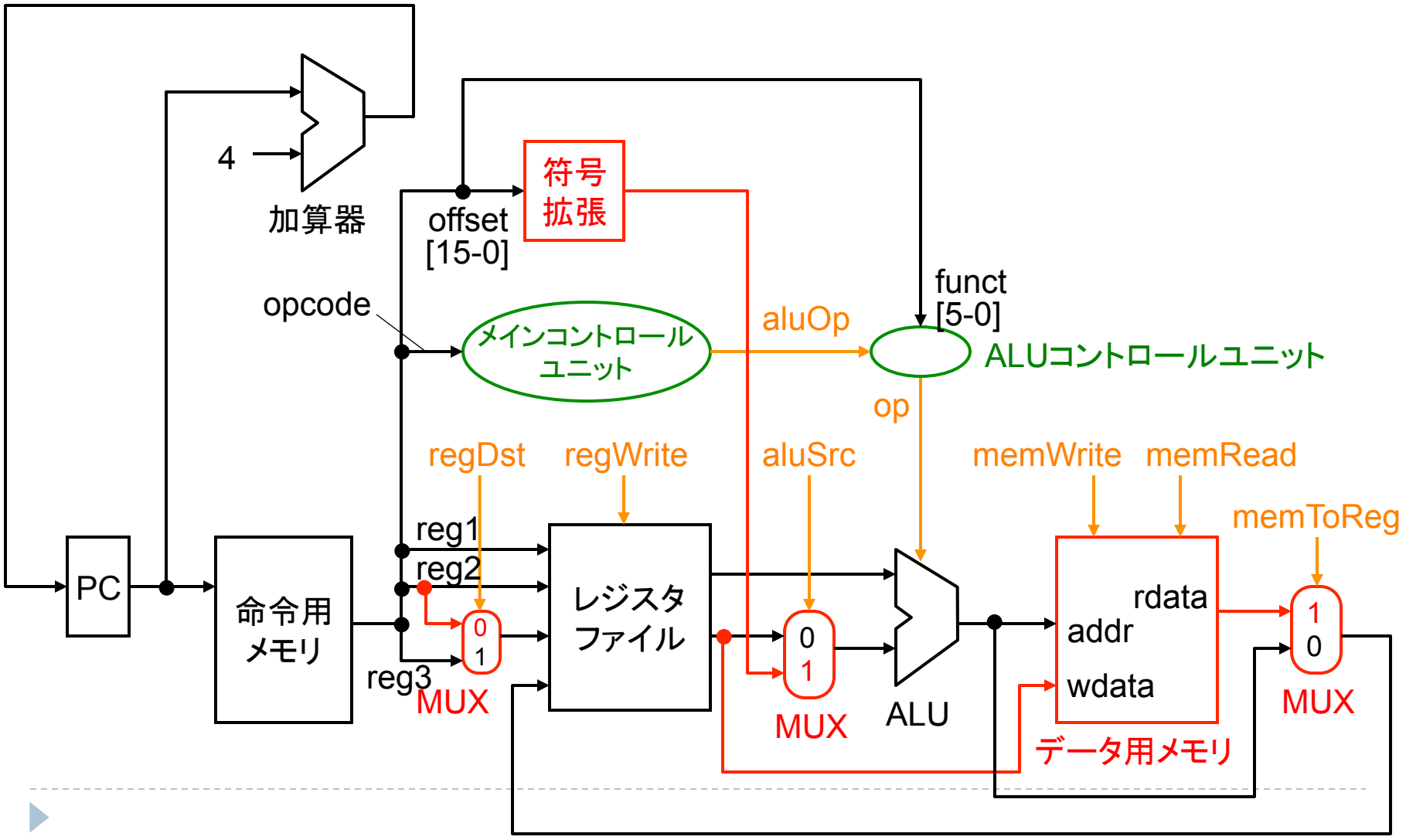
命令	regDst	aluSrc	memToReg	regWrite	memRead	memWrite	branch	aluOp1	aluOp0
lw	0	1	1	1	1	0	0	0	0

SignExtender クラス

```
public class SignExtender {
  public SignExtender(Bus a, Bus b) {
       :
  }
  public void run() {
    // バスaの信号をバスbの0～15本目に設定
    // バスaの15本目の信号(get)をバスbの16～31本目に設定(set)
  }
}
```

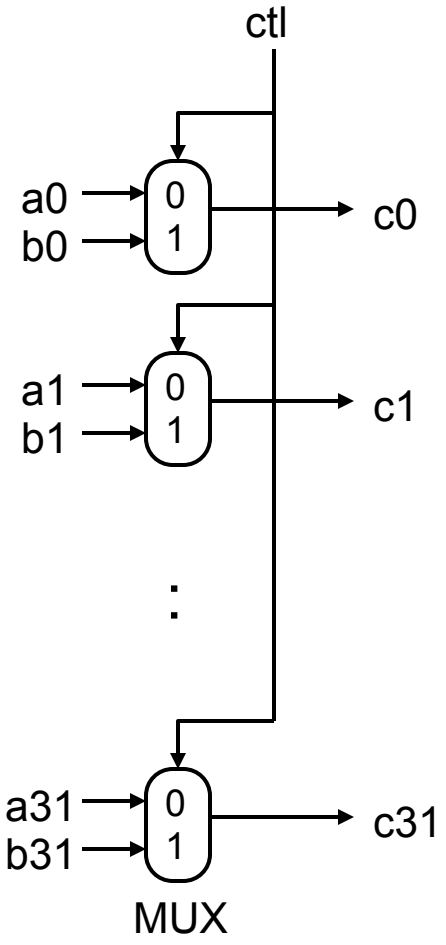


メモリアクセスを追加した回路



MUX_bus クラス

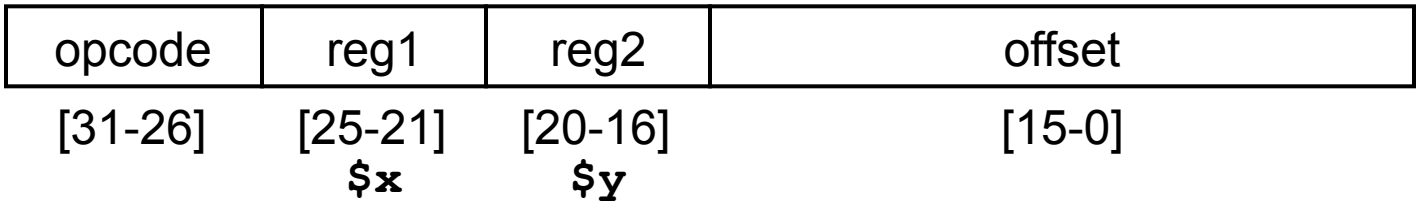
```
// バス版の2入力マルチプレクサ32個使用すればよいが、
// 今回は回路を用いずに実装してよい
public class MUX_bus {
  public MUX_bus(Path ctl,
                 Bus a, Bus b, Bus c) {

  }
  public void run() {
     // if ctl is 0: aのvalueをcに設定
     // else if ctl is 1: bのvalueをcに設定
  }
}
```



MIPS クラス（ver.4）：
分岐命令のフォーマット

- **`beq $x, $y, label`**

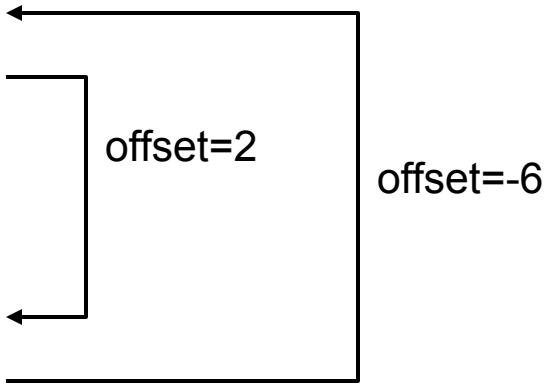
※SPIMがlabel⇒相対命令数へ変換している



  - opcode
    - 0x4 (000100)
  - reg1, reg2：比較するレジスタの番号
  - offset
    - PC+4 の位置からジャンプする命令数（相対ジャンプ）

オフセットの計算

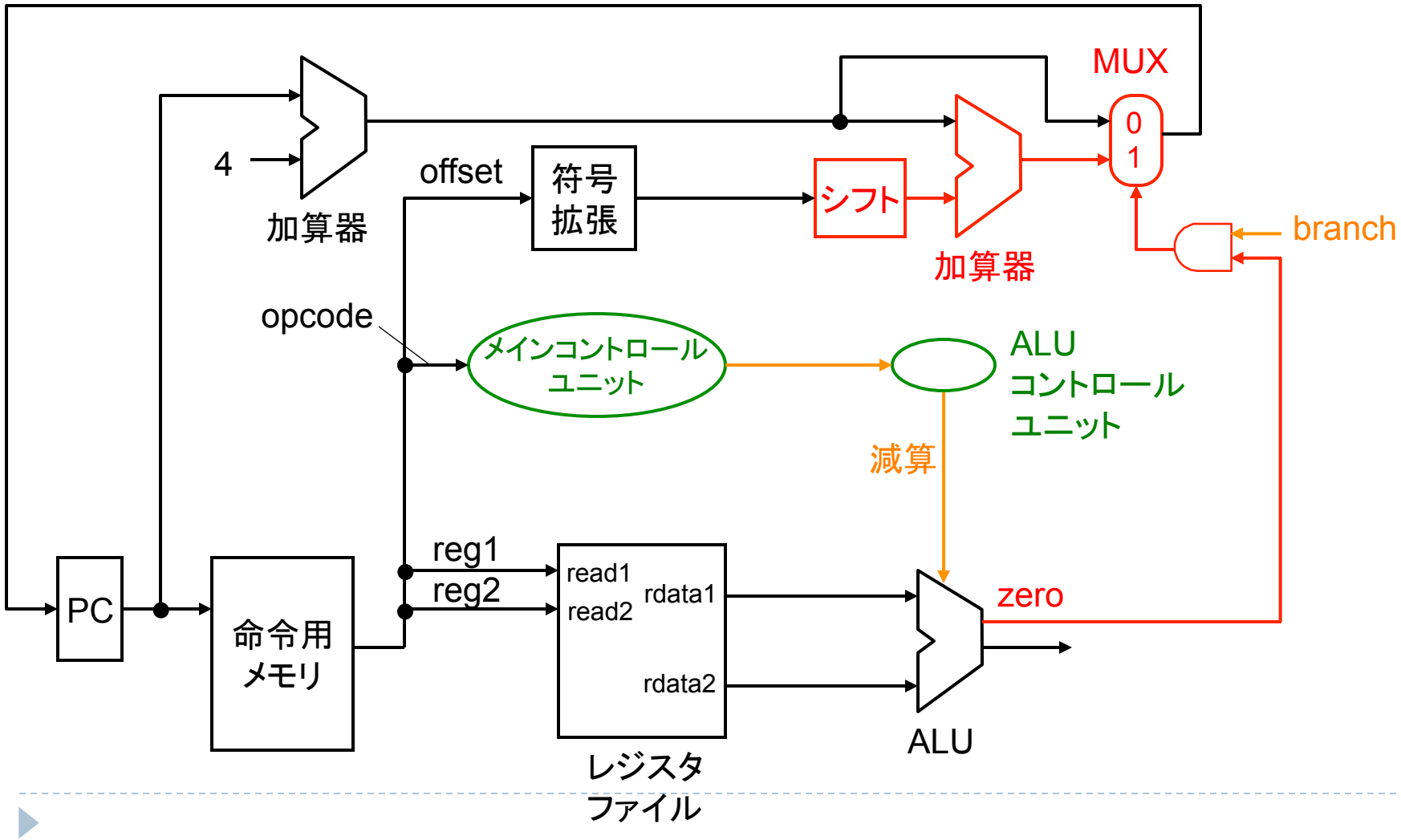
```
            loop:
0x04000000          xxx
0x04000004          beq $t0, $t1, next
0x04000008          xxx
0x0400000c          xxx
            next:
0x04000010          xxx
0x04000014          beq $t2, $t3, loop
0x04000018          xxx
```



spim のオフセット計算は PC+4の位置から
ジャンプする命令数(ワード数)になっているので注意

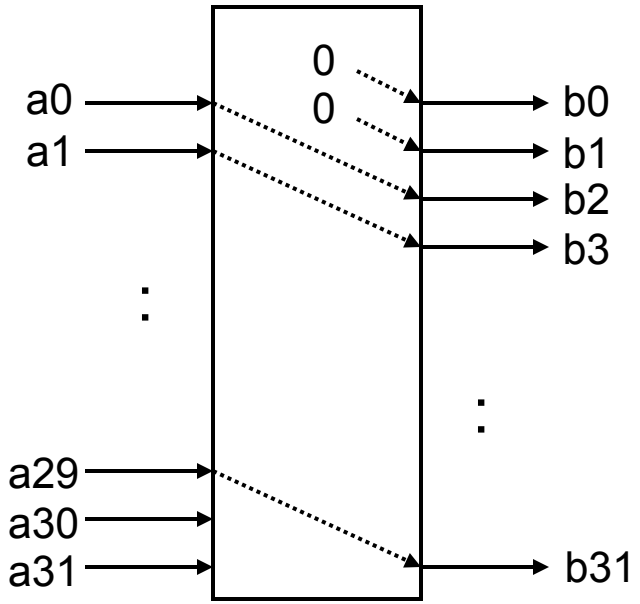
⇒ バイトアドレスに変換する必要がある

分岐（のみ）のための回路


MUX
0
1
4
加算器
offset
符号
拡張
シフト
加算器
branch
opcode
メインコントロール
ユニット
ALU
コントロール
ユニット
減算
PC
命令用
メモリ
reg1
reg2
read1
read2
rdata1
rdata2
レジスタ
ファイル
zero
ALU

Shifter2 クラス

```
// 2ビット左シフト
public class Shifter2 {
  public Shifter2(Bus a, Bus b) {
    :
  }
  public void run() {
    // a[i]信号をb[i+2]に設定
    // bの下位2ビットは0にする
  }
}
```

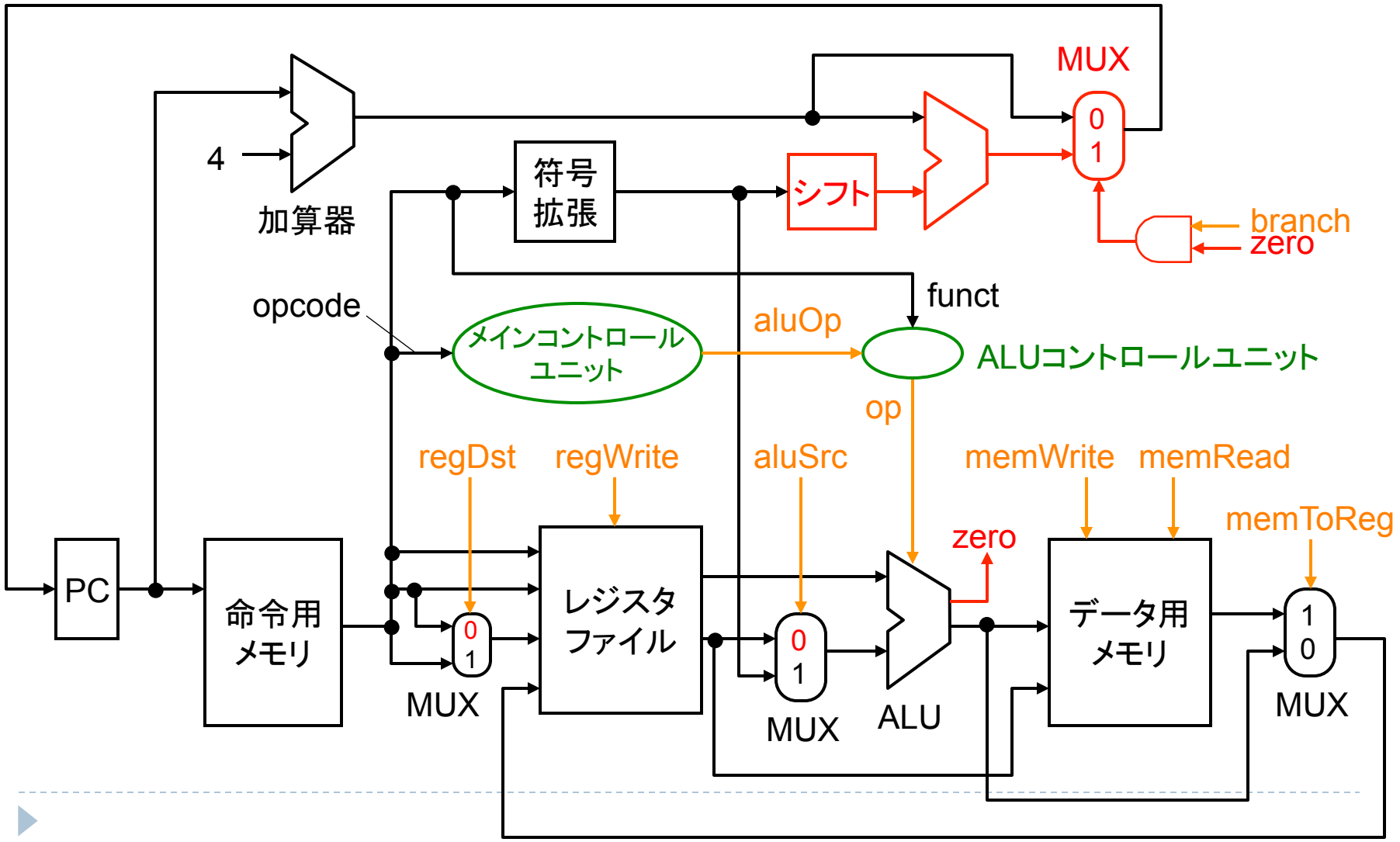


分岐命令の実行手順

1. ジャンプ先のアドレス PC + 4 + offset × 4 を計算する
   - 4倍は2ビット左シフトで実現（Shifter2）
2. レジスタファイルから比較する2つのレジスタ（$x, $y）を読み出す
3. ALU で $x - $y を計算する
4. zero フラグが 1 になれば、PC をジャンプ先のアドレスに変更する

命令	regDst	aluSrc	memToReg	regWrite	memRead	memWrite	branch	aluOp1	aluOp0
beq	X	0	X	0	0	0	1	0	1

分岐を追加した回路


MUX
0
1
4
加算器
符号
拡張
シフト
branch
zero
funct
opcode
aluOp
メインコントロール
ユニット
ALUコントロールユニット
op
regDst
regWrite
aluSrc
memWrite
memRead
memToReg
zero
PC
命令用
メモリ
レジスタ
ファイル
データ用
メモリ
MUX
MUX
ALU
MUX
1
0

Appendix: 制御信号の値

- 命令の種類によって以下のように決定される
  - X は使われないので、気にしなくてよい

命令	regDst	aluSrc	memTo Reg	reg Write	mem Read	mem Write	branch	aluOp1	aluOp0
演算	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

R 形式

opcode	reg1	reg2	reg3	0	funct
[31-26]	[25-21]	[20-16]	[15-11]		[5-0]

I 形式

opcode	reg1	reg2	offset
[31-26]	[25-21]	[20-16]	[15-0]

Appendix:
メインコントロールユニットの真理値表

入出力	配線名	演算	lw	sw	beq
入力	opcode5	0	1	1	0
	opcode4	0	0	0	0
	opcode3	0	0	1	0
	opcode2	0	0	0	1
	opcode1	0	1	1	0
	opcode0	0	1	1	0
出力	regDst	1	0	X	X
	aluSrc	0	1	1	0
	memToReg	0	1	X	X
	regWrite	1	1	0	0
	memRead	0	1	0	0
	memWrite	0	0	1	0
	branch	0	0	0	1
	aluOp1	1	0	0	0
	aluOp0	0	0	0	1

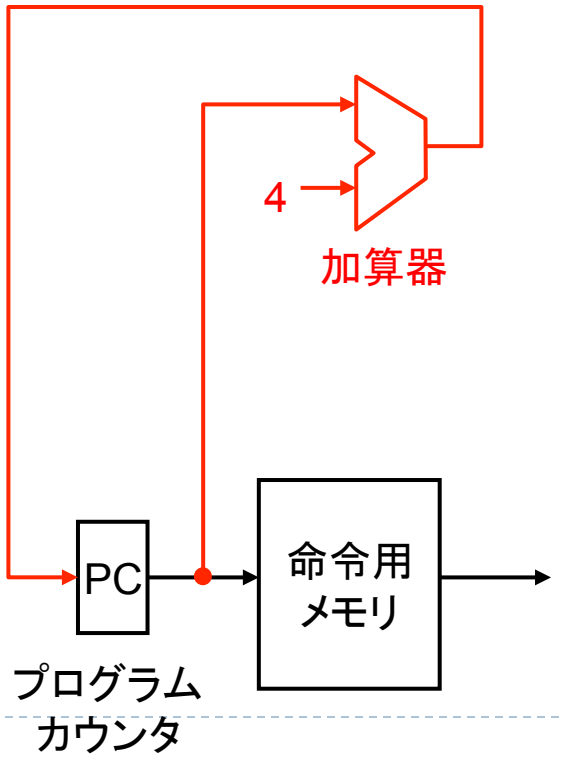
命令の種類を
決めるビット列
（命令に含まれる）

課題

最終締め切りに注意！

課題1： MIPSクラス(ver.1 + ver.2)

- 連続実行をサポートせよ
  - 以下の回路を追加するのみ



課題2 (オプション)：
MIPSクラス(ver.1 + ver.2+ver.3+ver.4)

- メモリアクセス命令(ver.3)と分岐命令(ver.4)をサポートせよ
  - メモリアドレスは setValue メソッドを使ってレジスタに設定
    - **`regFile.setValue(11, 0x10000000);`**
      - 11番レジスタ($t3)にセット

アセンブリ命令	機械語命令
`sw $t1, 0($t3)`	`0xad690000`
`lw $t0, 0($t3)`	`0x8d680000`
`beq $t0, $t2, loop`	`0x110afff8`

(beqの下位16ビット(fff8)は適宜変更してテスト)

課題3 (オプション)：
授業でカバーしていない機能の実装

- プログラムファイルから命令をロード
- 自作のアセンブラを実装
  - 例) add $t0, $t1, $t2 => 0x012a4020
- qtspimのようなGUIの作成
- その他追加したい命令
  http://www.cs.wisc.edu/~larus/HP_AppA.pdf の A.10（p. 44以降）の命令一覧を参考に
    - 命令フォーマットも載ってる
  - immediate命令は比較的容易に実装できる

課題3(オプション)：補足

- 実装しやすいように、多少仕様を変えてもかまいません
  - 複数の命令をサポートすると、制御ユニットの処理が複雑になるので、制御ユニットは回路を使わずに実装するとか
  - 変えた場合は、変更点も簡単にレポートに書くこと
- 最終的な説明、回路図、実行結果、実行方法などをまとめたレポートを書くこと
  - GUIを実装した場合はGUIの画面の説明もつけてください

課題提出

- 〆切： 8/6(Mon) 23:59 （最終締め切り）
  - **未提出の課題も上記の締め切りまでに提出すること**
- 提出物：以下のファイルを圧縮したもの
  - (必要ならば) ドキュメント(pdf,plain txt,wordなんでも可)
    - 演習の授業に対する感想等
  - 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
- 提出方法：Webから提出

まとめ

- C言語
  - 値渡し & 参照渡し
  - 配列 & ポインター
  - スタック領域 & ヒープ領域
- アセンブリ言語
  - 擬似命令(blt)
  - callee-save & caller-save
- アーキテクチャ
  - 組み合わせ回路、積和標準形、真理値表
  - 負の整数の表現法
  - Ripple Carry Adder & Carry Look-ahead Adder
  - 各命令(R形式、lw,sw,branch,jump)における実行時間
  - シングルサイクル実装、マルチサイクル実装、パイプライン実装
    - それぞれのクロック周波数
  - クリティカルパス
  - パイプライン
    - パイプラインハザード
      - 構造、制御(分岐)、データハザード
      - フォワーディング(バイパシング)、パイプラインストール、コンパイラによる最適化

- コンピュータの性能評価
  - 用語
    - サイクルタイム、
    - クロックレート(クロック周波数)、
    - CPI (cycles per instruction,1命令あたりの平均クロックサイクル)
    - MIPS (Million Instructions Per Second)
  - 性能評価の計算
  - アムダールの法則

- メモリとI/O
  - メモリ階層と局所性
  - キャッシュミス率
  - キャッシュ実装
    - Direct Mapped Cache (1-way set associative)
    - N-way set associative
    - Fully associative
  - 実行時間の計算
    - 実行時間＝(実行サイクル数＋ストールサイクル数)×サイクルタイム
    - ストールサイクル数＝命令数×ミスの割合×ミスのペナルティ
  - 仮想記憶
    - ページフォルト
    - LRU