

2018年度 計算機システム(演習)  
第8回(最終回)  
2019.01.24

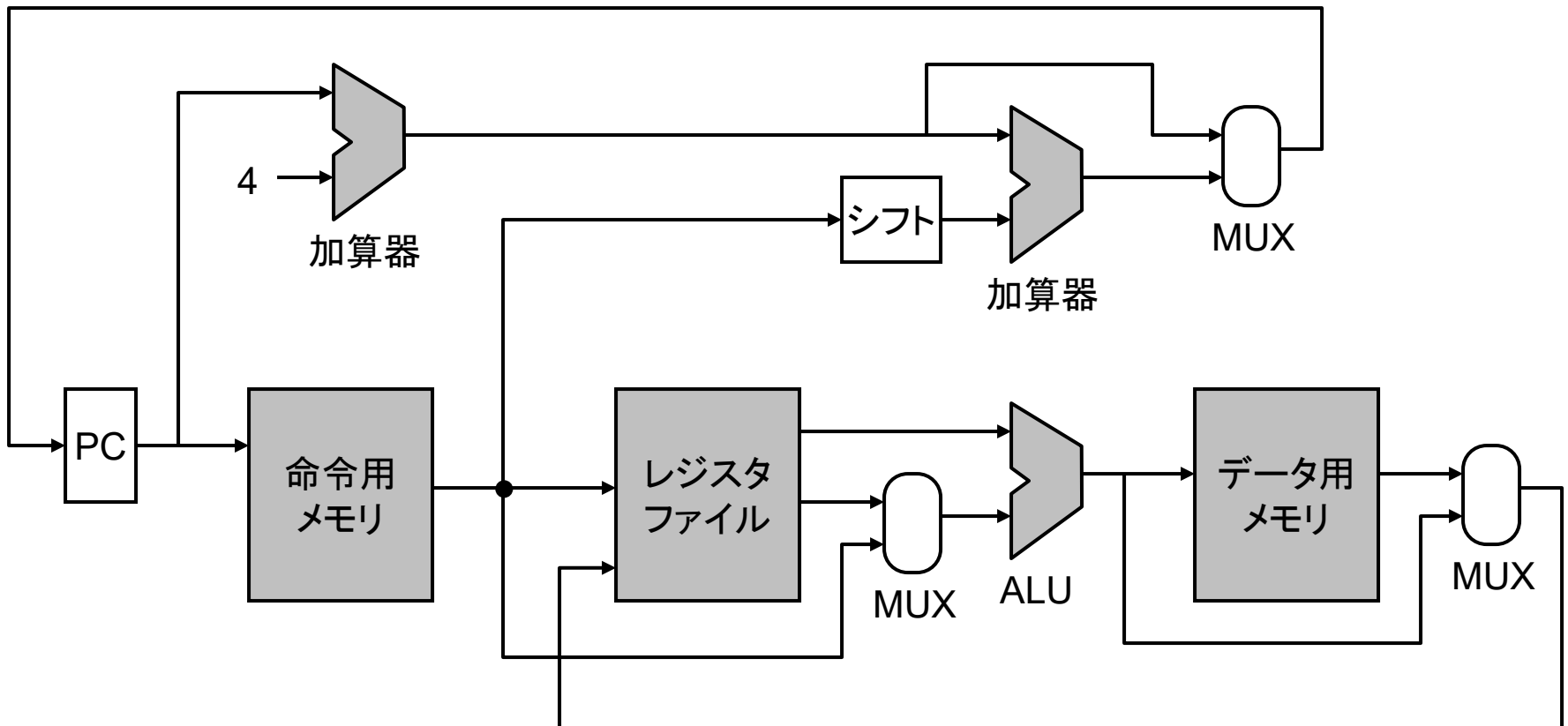
遠藤 敏夫(学術国際情報センター/数理・計算科学系 教授)  
野村 哲弘(学術国際情報センター/数理・計算科学系 助教)

# MIPSシミュレータ構築の流れ

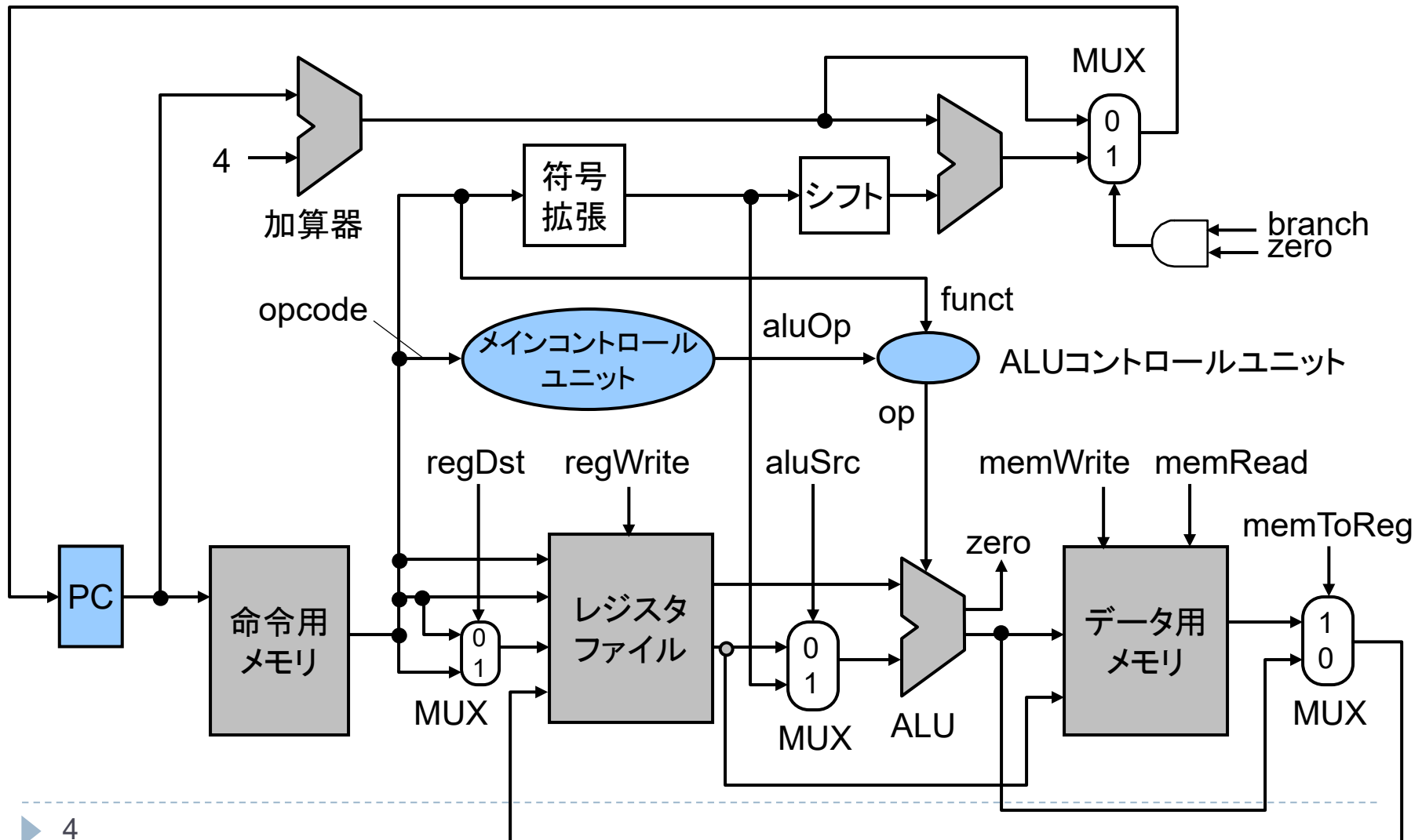
---

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. プログラムカウンタ
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. 連続実行
8. 機能拡張 (補足説明・課題の範囲外)
  - ▶ メモリアクセス命令
  - ▶ 分岐命令

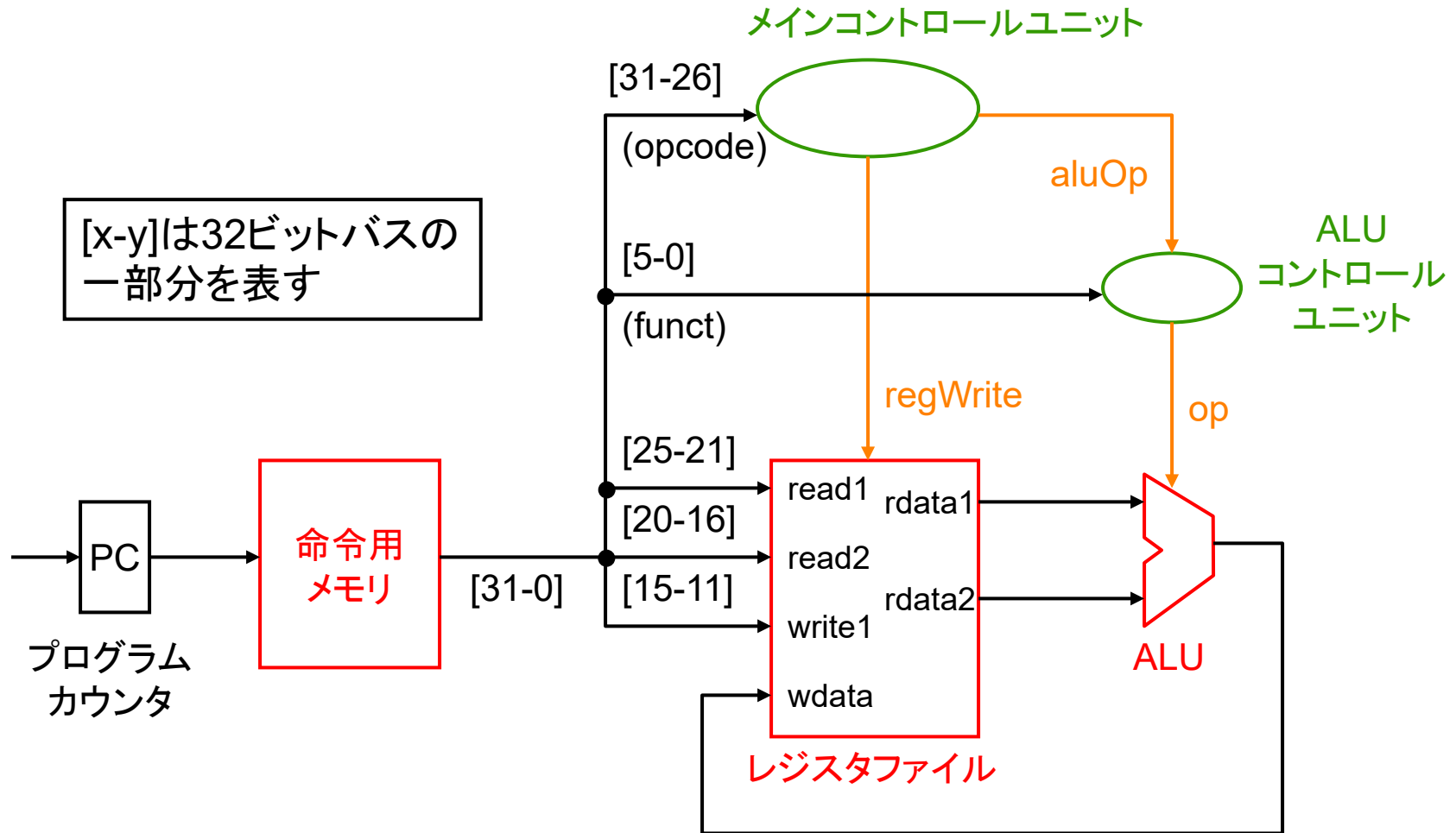
# MIPSシミュレータの概要



# MIPSシミュレータの完成図

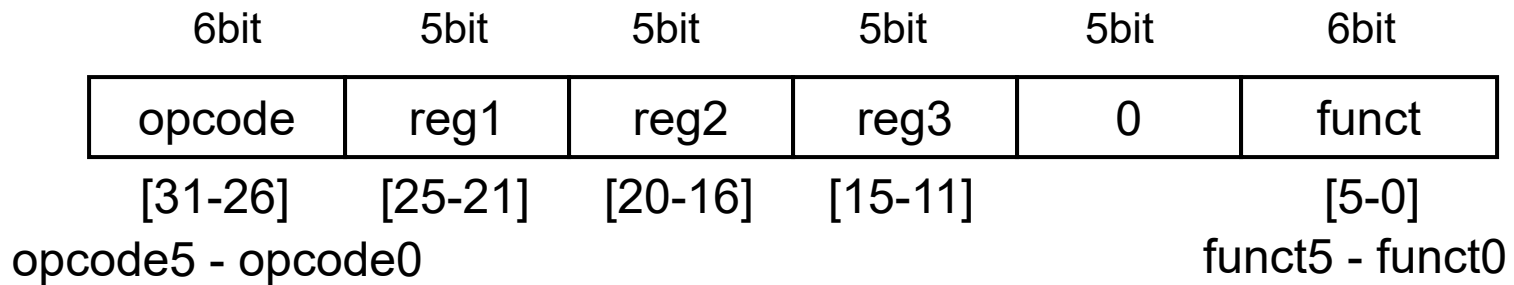


# 本日は「算術論理演算回路」を作る



# 算術論理演算命令のフォーマット

- ▶ 32ビットの命令の中身は以下のようにになっている



- ▶ opcodeで制御信号の値を決める(全体制御)
- ▶ funct が演算の内容を決める(ALU)
- ▶ 入力レジスタ番号: reg1, reg2
- ▶ 出力レジスタ番号: reg3

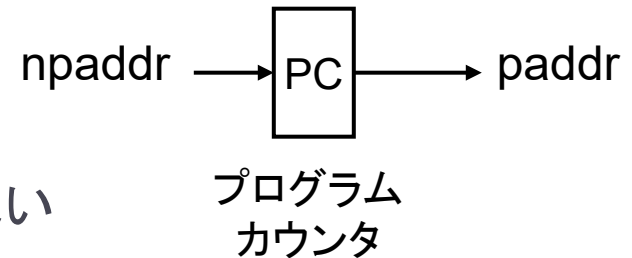
add    \$t0    \$t1    \$t2  
          ↑      ↑      ↑  
         reg3    reg1    reg2

funct	命令
100000	add
100010	sub
100100	and
100101	or
101010	slt

# プログラムカウンタ

---

- ▶ 実行しているアドレスを保持するレジスタ
  - ▶ 0x04000000 から実行を開始
  - ▶ 入力: 次の命令のアドレス
  - ▶ 出力: 実行する命令のアドレス
  - ▶ Register への wctl は常に true にしておけばよい
    - ▶ 実行毎(クロック毎)に実行アドレスを更新するため



# メインコントロールユニット

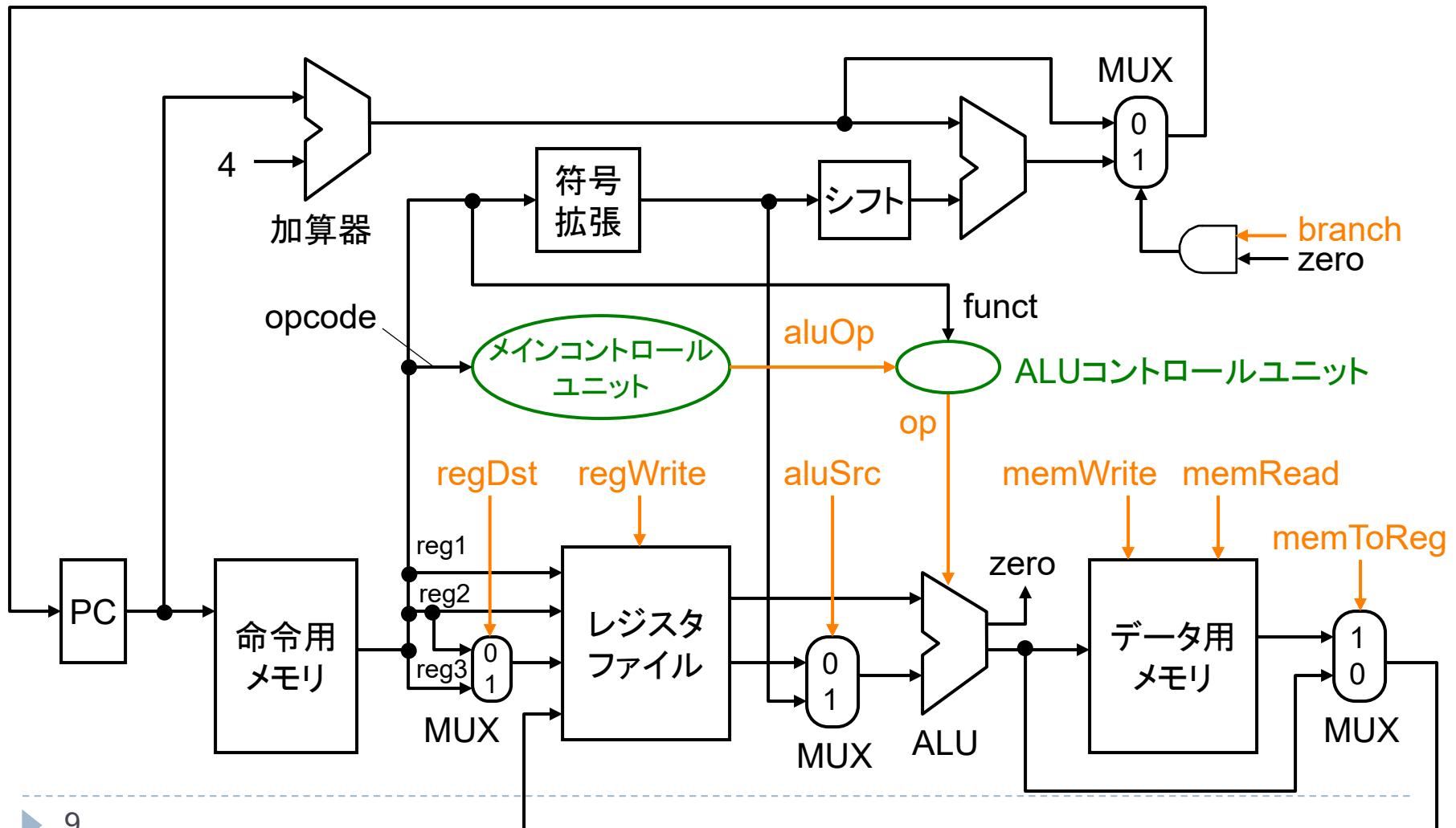
- ▶ 命令の内容(opcode)から制御信号の値を決める
  - ▶ 各回路に制御入力を接続する

制御パス	意味
aluOp (2bit)	命令の種類(演算、メモリアクセス、分岐)
regWrite	レジスタへの書き込みを制御
memRead	データ用メモリの読み出しを制御
memWrite	データ用メモリへの書き込みを制御
regDst	命令中の書き込みレジスタ番号の位置を制御
memToReg	計算結果とメモリ値のどちらをレジスタに書くかを制御
aluSrc	レジスタ値と命令中の値のどちらをALUに入力するかを制御
branch	分岐を制御

完成版のためにこれらも実装



# 回路全体を制御



# 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 : 使用されない (任意のXで動作は同じ)

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

これらも実装

これらも実装

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]

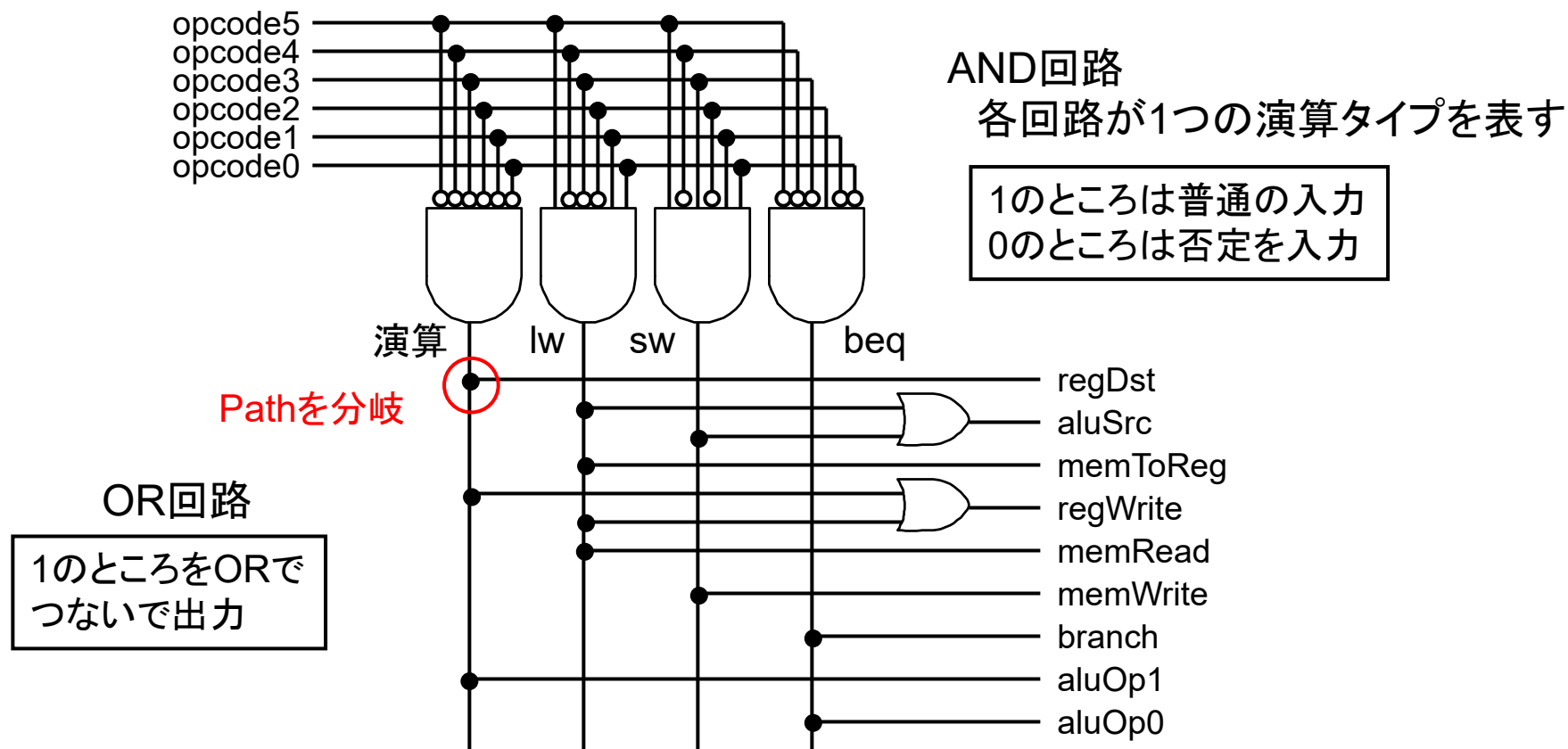
# メインコントロールユニットの真理値表

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)

# メインコントロールユニットの回路

## ▶ PLA (Programmable Logic Array) で作成



# メインコントロールユニットの回路 (演算のみ対応)

## ▶ PLA (Programmable Logic Array) で作成

opcode5  
opcode4  
opcode3  
opcode2  
opcode1  
opcode0

演算  
Pathを分岐

AND回路

各回路が1つの演算タイプを表す

1のところは普通の入力  
0のところは否定を入力

regDst  
aluSrc  
memToReg  
regWrite  
memRead  
memWrite  
branch  
aluOp1  
aluOp0

# Control Unit (演算のみ対応の簡略版)

---

```
void control_unit(Signal opcode[6], // 入力
                  Signal *register_dst, // 出力
                  Signal *register_write,
                  Signal *aluop1)
{
    // opcodeをNOTGateで反転させたものを用意
    // AND-N Gateに通す
    // 出力Pathに値を渡す
}
```

# ALUコントロールユニット

- ▶ ALU の使い方に関する制御
  - ▶ メインコントロールユニットから分離

命令	命令の種類 (メインコントロール ユニットから)		演算の種類 (命令のビット列([5-0])から)						[2][1][0]
	aluOp1	aluOp0	funct5	funct4	funct3	funct2	funct1	funct0	op
lw, sw	0	0	X	X	X	X	X	X	0 1 0
beq	X	1	X	X	X	X	X	X	1 1 0
add	1	X	X	X	0	0	0	0	0 1 0
sub	1	X	X	X	0	0	1	0	1 1 0
and	1	X	X	X	0	1	0	0	0 0 0
or	1	X	X	X	0	1	0	1	0 0 1
slt	1	X	X	X	1	0	1	0	1 1 1

# op毎の真理値表

op[2]: Binvert

op[1][0]: Operation(MUX)

op[2]=1

aluOp1	aluOp0	funct5	funct4	funct3	funct2	funct1	funct0
X	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

op[1]=1

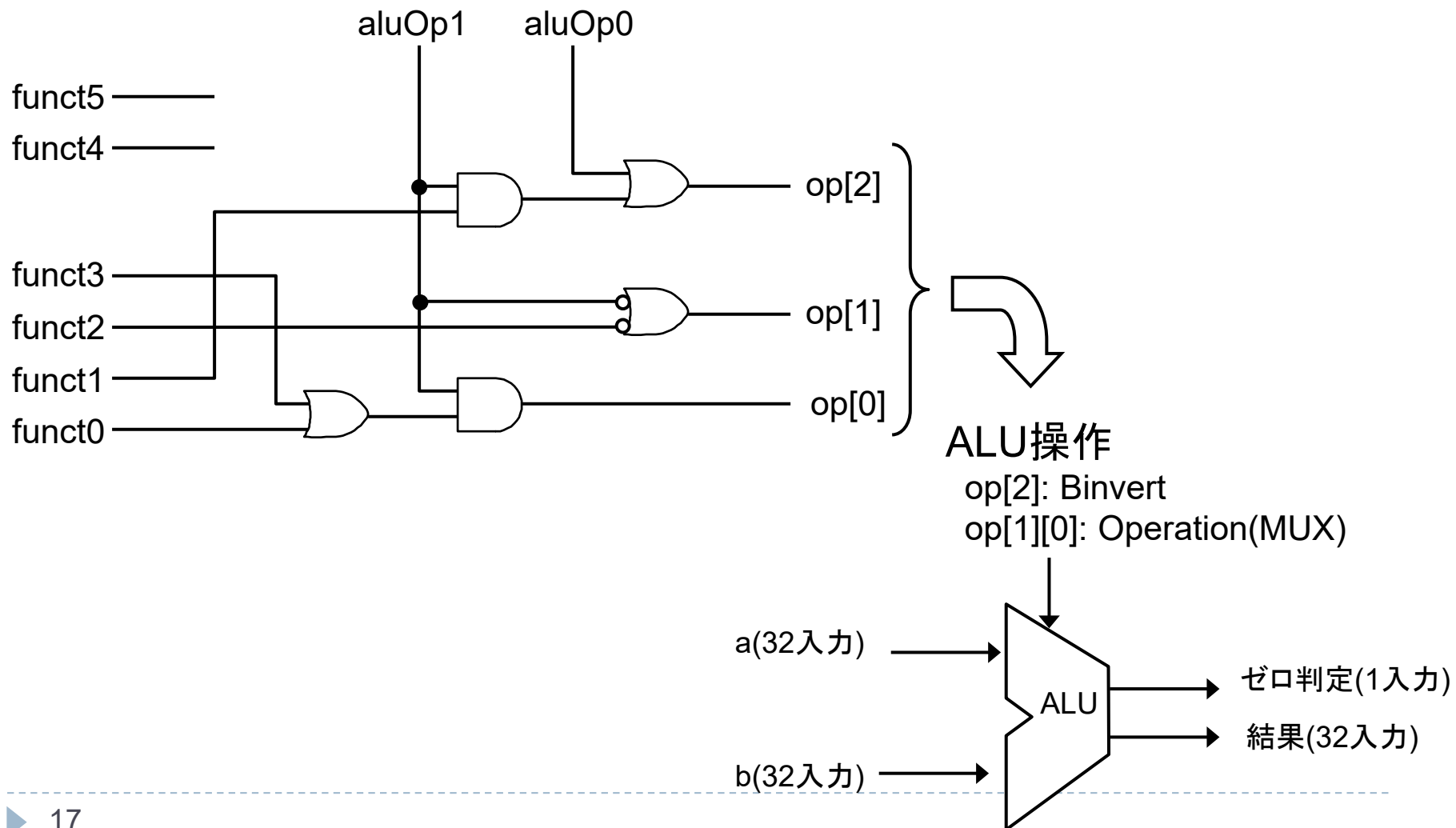
aluOp1	aluOp0	funct5	funct4	funct3	funct2	funct1	funct0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

op[0]=1

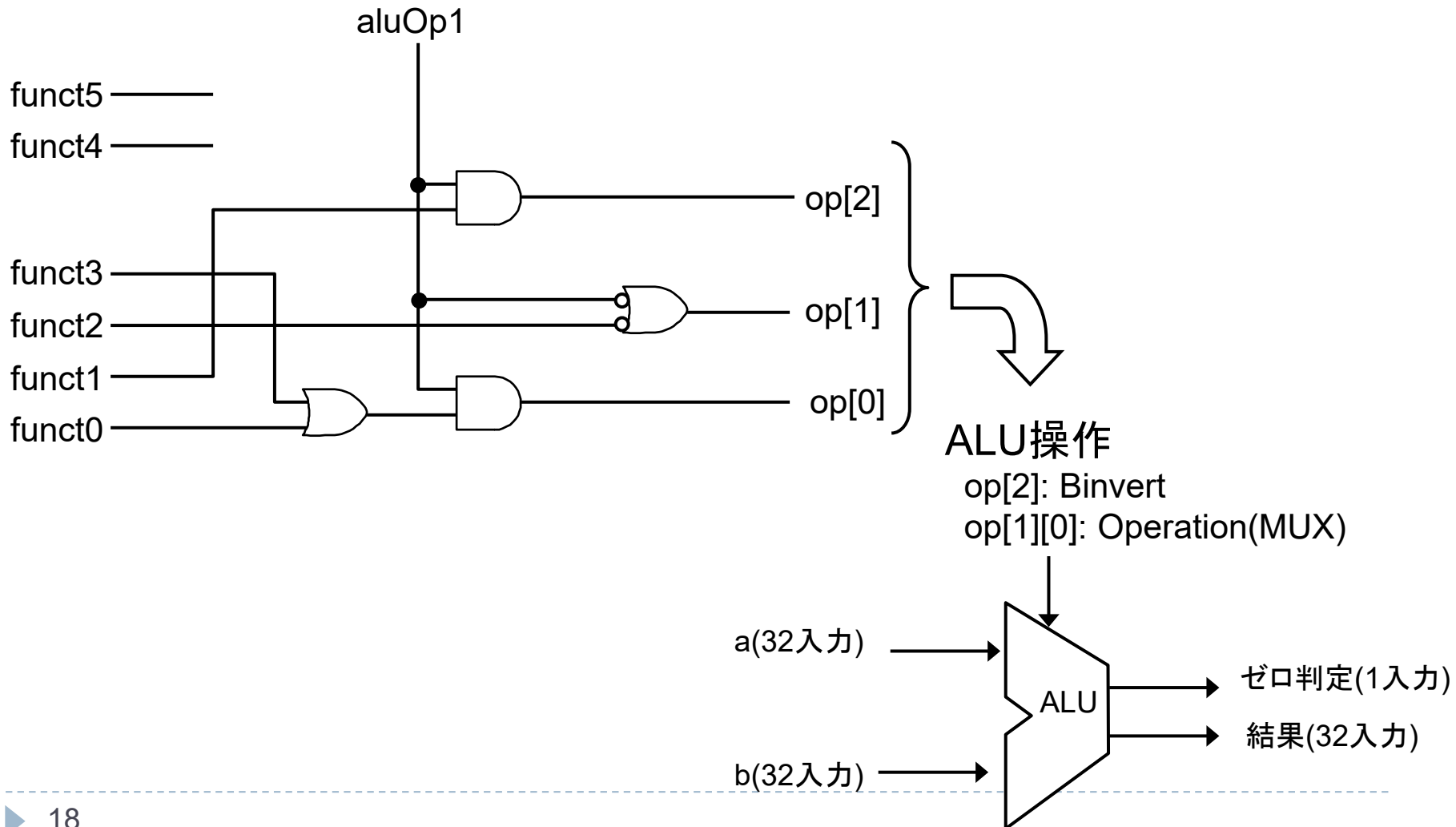
aluOp1	aluOp0	funct5	funct4	funct3	funct2	funct1	funct0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X



# ALUコントロールユニットの回路



# ALUコントロールユニットの回路 (簡略版)

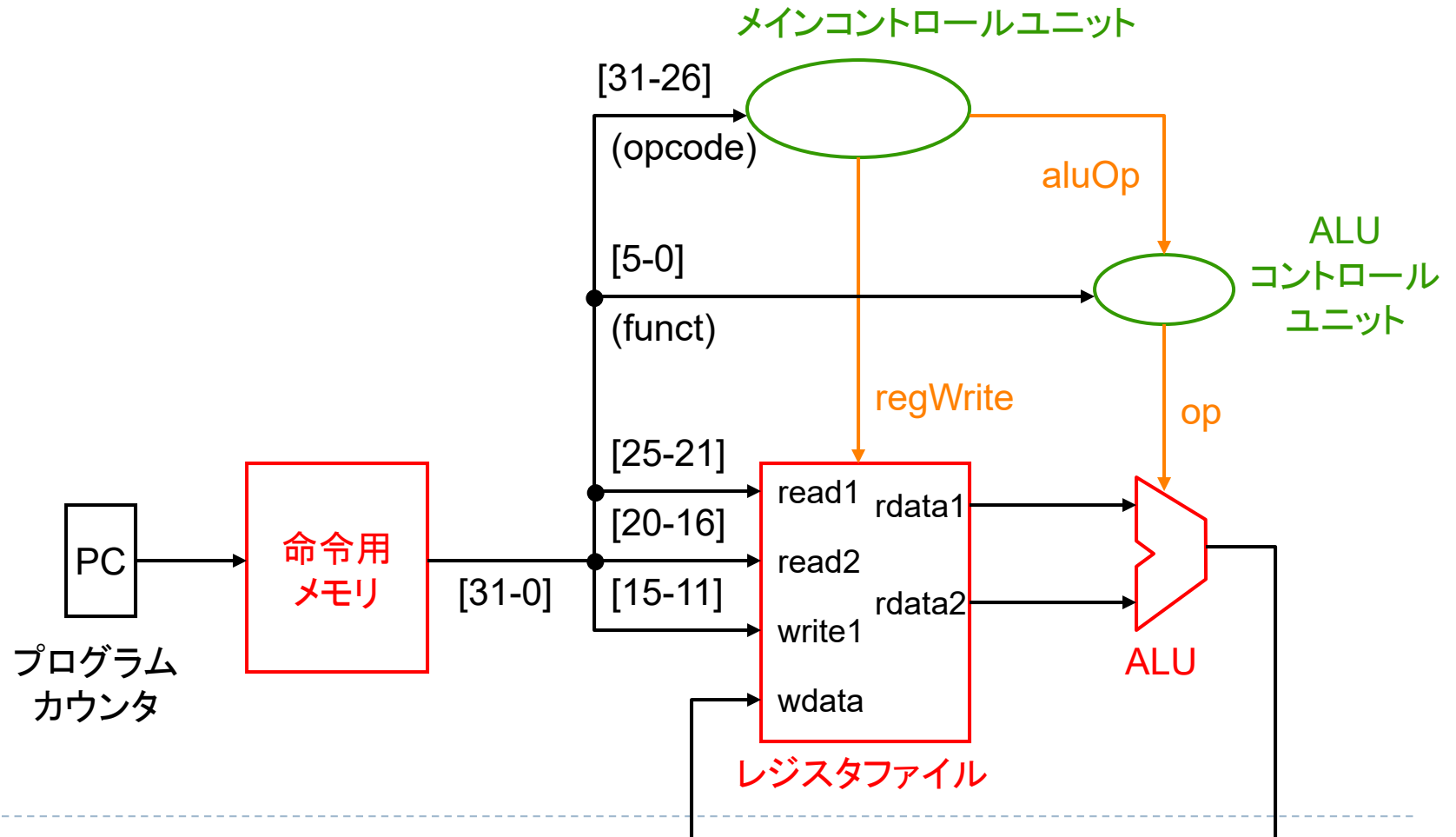


# ALUControlUnit (簡略版)

---

```
void alu_control_unit(Signal *funct, Signal aluop1, // 入力
                      Signal *ops) // 出力
{
    //前ページの回路図のように回路を作成する
}
```

# MIPS ver.1 完成イメージ



# MIPS 構造体

---

```
typedef struct {  
    Register pc;  
    RegisterFile rf;  
    InstMemory im;  
    DataMemory dm;  
} MIPS;
```

# MIPS ver.1 (1/3)

```
// 命令をInst Memoryにセットする
// 計算用にレジスタに予め値をセットしておく
void mips_init(MIPS *m, int inst)
{
    // PCに実行開始アドレスを設定
    register_set_value(&(m->pc), INST_MEM_START);
    // $t1 (9番レジスタ) に0x100を代入
    register_set_value((m->rf.r + 9), 0x100);
    // $t2 (10番レジスタ) に0x300を代入
    register_set_value((m->rf.r + 10), 0x300);
    // メモリに命令を格納 例) add $t0, $t1, $t2 => 0x012a4020
    inst_memory_set_inst(&(m->im), INST_MEM_START, inst);
}
```

# MIPS ver.1 (2/3)

```
void mips_run(MIPS *m, int inst_num)
{
    Word npaddr, paddr;
    Word instr;
    Word wdata, rdata1, rdata2;
    Signal register_dst, register_write, alu_op1, ops[3];
    Signal zero;
    // 順番を考えながら配線を行う
    // pc(register)を実行
    // 命令メモリから命令読み込み
    // Control Unitを実行
    // ALU Contorol Unitを実行
    // Register Fileの実行 (値の読み込み)
    // ALUで計算
    // 再度Register Fileの実行 (値の書き込み)
    // $t0 (8番レジスタ) の値をprintする
}
```

# MIPS ver.1 (3/3)

---

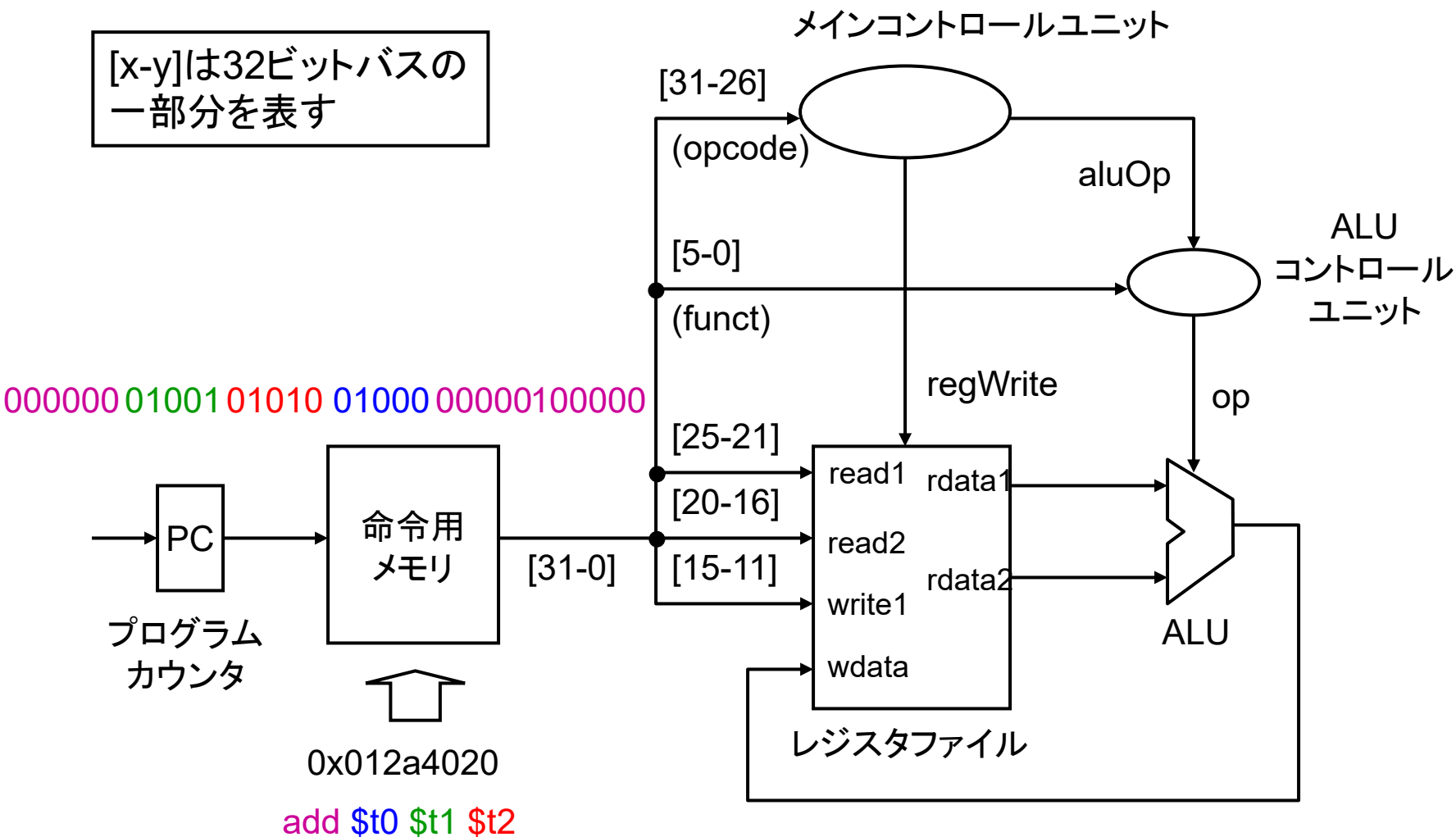
// テスト用の関数

```
void test_mips()
{
    MIPS m;
    int inst = 0x012a4020;
    mips_init(&m, inst);
    mips_run(&m);
}
```

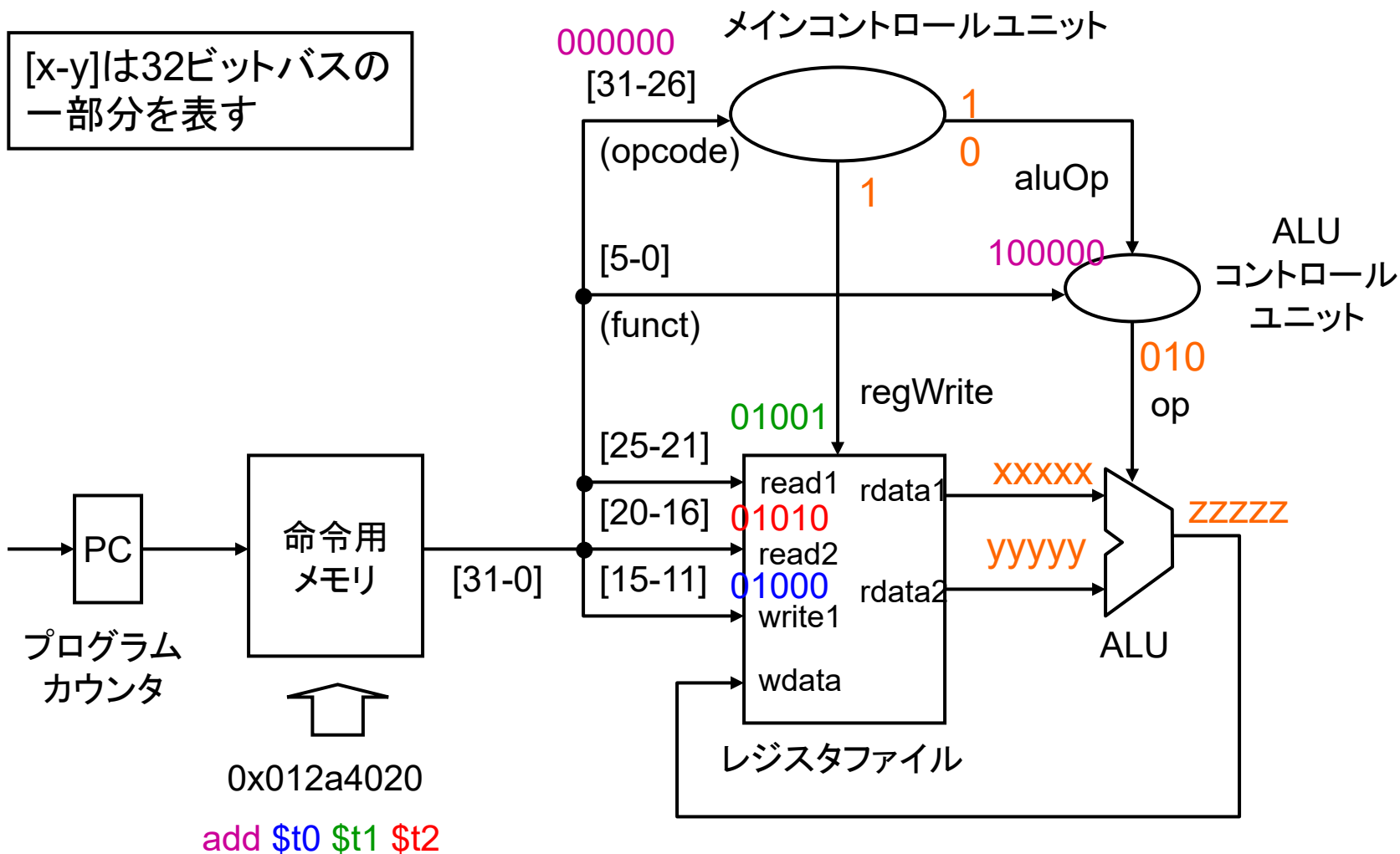


# Appendix: 動作の流れ

[x-y]は32ビットバスの一部を表す

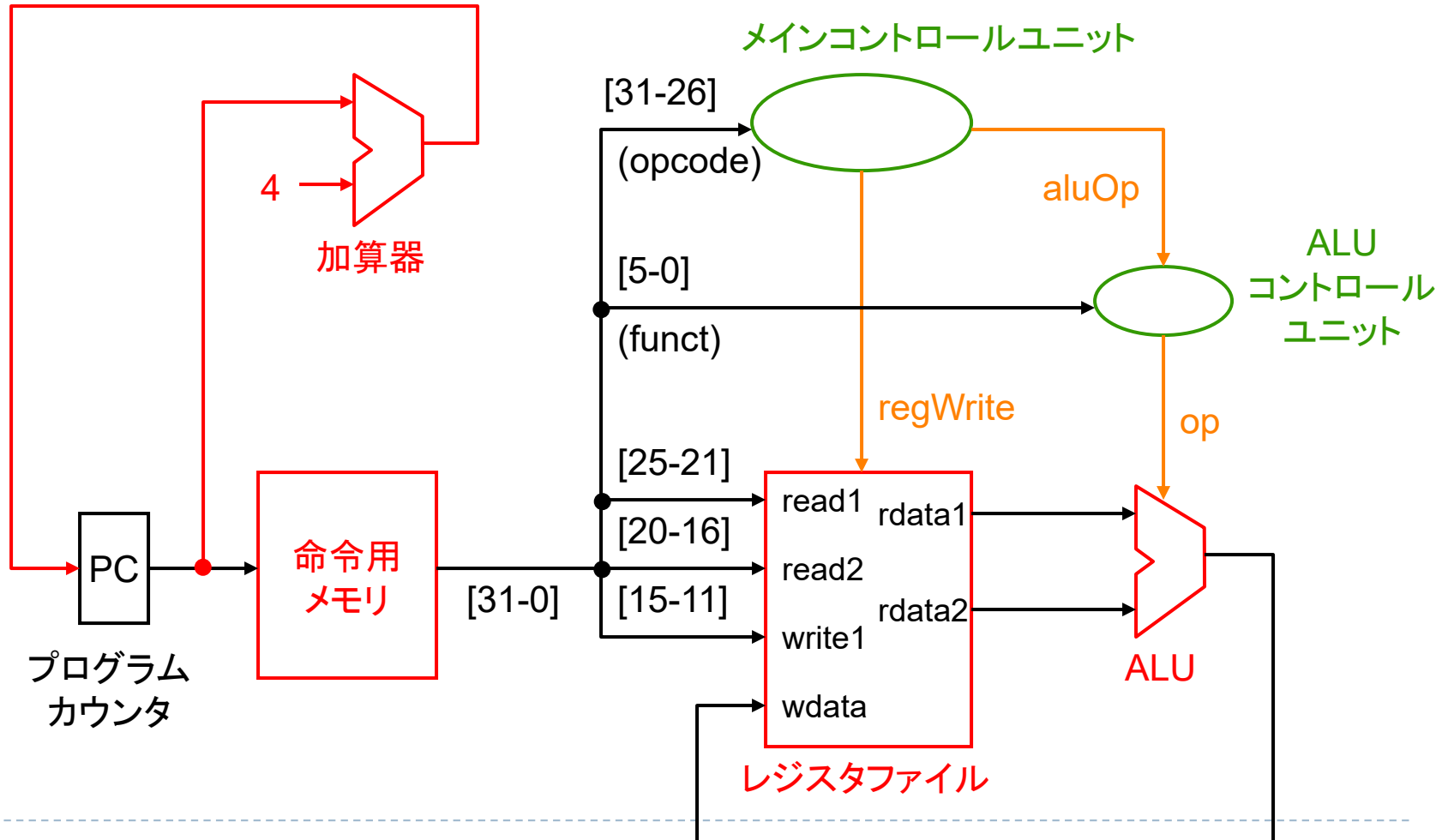


# Appendix: 動作の流れ



# MIPS (ver.1 + ver. 2)

## 連続実行を追加した回路



# MIPS (ver.1 + ver. 2)

## MIPS ver. 2 (1/3)

---

```
// 複数の命令をInst Memoryにセットする
// 計算用にレジスタに予め値をセットしておく
void mips_init(MIPS *m, int *inst, int inst_num)
{
    int i;
    // PCに実行開始アドレスを設定
    register_set_value(&(m->pc), INST_MEM_START);
    // $t1 (9番レジスタ) に0x100を代入
    register_set_value((m->rf.r + 9), 0x100);
    // $t2 (10番レジスタ) に0x300を代入
    register_set_value((m->rf.r + 10), 0x300);
    // メモリに命令を格納 例) add $t0, $t1, $t2 => 0x012a4020
    for (i = 0; i < inst_num; ++i) {
        inst_memory_set_inst(&(m->im),
                               INST_MEM_START + 4 * i, inst[i]);
    }
}
```

---

# MIPS (ver.1 + ver. 2)

## MIPS ver. 2 (2/3)

```
void mips_run(MIPS *m, int inst_num)
```

```
{
```

```
/* 省略 */
```

```
Signal zero, zero4;
```

```
for (i = 0; i < inst_num; ++i) {
```

```
    // 順番を考えながら配線を行う
```

```
    // pcを実行
```

```
    // 省略
```

```
    // +4の計算を行う
```

```
    Signal pcadd[3] = {false, true, false};
```

```
    Word four;
```

```
    word_set_value(&four, 4);
```

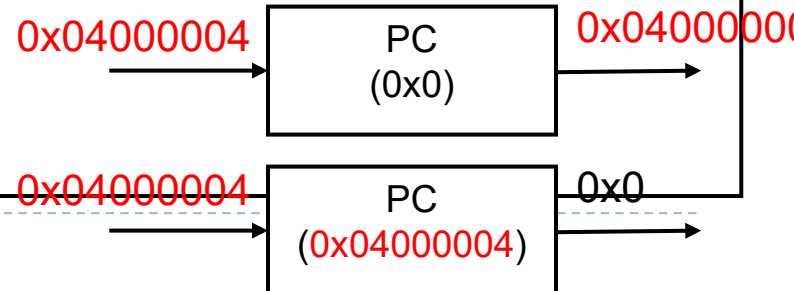
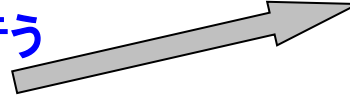
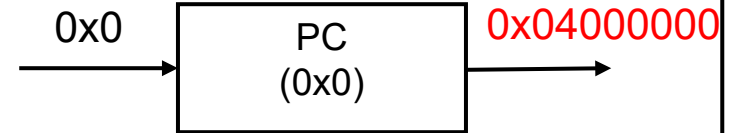
```
    alu32(pcadd, paddr, four, &npaddr, &zero4);
```

```
    // 新しいPCの値を書き込む
```

```
    register_run(&(m->pc), true, npaddr, &paddr);
```

```
}
```

```
}
```



# MIPS (ver.1 + ver. 2)

## MIPS ver. 2 (3/3)

---

// テスト用の関数

```
void test_mips()
{
    MIPS m;
    int inst[] = {0x012a4020, //add $t0, $t1, $t2
                  0x012a4022, //sub $t0, $t1, $t2
                  0x012a4024, //and $t0, $t1, $t2
                  0x012a4025, //or $t0, $t1, $t2
                  0x012a402a}; //slt $t0, $t1, $t2

    mips_init(&m, inst, 5);
    mips_run(&m, 5);
}
```



# 課題

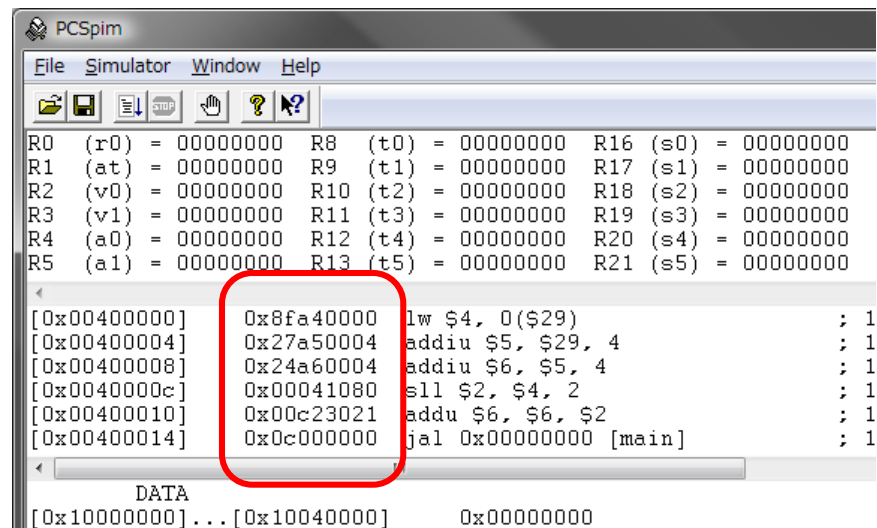


# 課題1 :

## MIPS (ver.1 + ver.2)

- ▶ 「算術論理演算回路」を作成せよ
  - ▶ add, sub, and, or, slt をそれぞれ実行し、結果をまとめる
    - ▶ 機械語命令は qtspim の Text Segments の2列目を見れば分かる
  - ▶ Control UnitやALU Control Unitは簡略版でOK

アセンブリ命令	機械語命令
add \$t0, \$t1, \$t2	0x012a4020
sub \$t0, \$t1, \$t2	0x012a4022
and \$t0, \$t1, \$t2	0x012a4024
or \$t0, \$t1, \$t2	0x012a4025
slt \$t0, \$t1, \$t2	0x012a402a





# 課題提出

---

- ▶ ✕ 切: **2/8 (金) 23:59**
- ▶ 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
    - ▶ 今回は第5回課題以降で作った**動くもの一式を提出**してください
  - ▶ ドキュメント
    - ▶ 感想等
    - ▶ 次ページ以降のアンケート
- ▶ 全課題の追加、再提出は2/12(火)までは受け付けます
  - ▶ それ以降もシステム上では提出できますが、成績に反映される保証はしません
- ▶ 質問等があれば [compsys18@el.gsic.titech.ac.jp](mailto:compsys18@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

# アンケート

---

- ▶ 最終課題のドキュメントの末尾に記述してもらえればと思います
  - ▶ 来年度以降に役立てます
  - ▶ 回答の有無や回答内容は点数に影響を与えません
  - ▶ 選択肢で「その他」となっているものに関しては、具体的に記述する必要はありません。書いてくれてもいいです。

# アンケート

---

## ▶ 演習課題のプログラム作成をどこで行ったか

1. 演習室
2. その他

## ▶ 演習課題のプログラム作成環境について

### ▶ OS

1. Windows系
2. MacOS系
3. その他

### ▶ エディタ

1. Sublime textなど具体的な名称を書いてもらえればと思います

### ▶ コンパイル等の環境

1. Macの場合ターミナルなど具体的な名称を書いてもらえればと思います

# アンケート

---

## ▶ 演習課題について

### ▶ 難易度 (5段階)

1. 簡単であれば1、難しければ5、適量であれば3

### ▶ 分量 (5段階)

1. 少なければ1、多ければ5、適量であれば3

## ▶ 授業全体の感想や要望

## 2/1 授業予定

---

- ▶ 5-6限 (13:20-14:50) 演習 (出席は任意)
  - ▶ 講義室での説明は行いません
  - ▶ W7計算機室で演習・講義に関する質問を受け付けます
- ▶ 7-8限 (15:05-16:35) 講義
  - ▶ W621で行います

# 補足説明

## (MIPSシミュレータをさらに拡張)

# 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)**

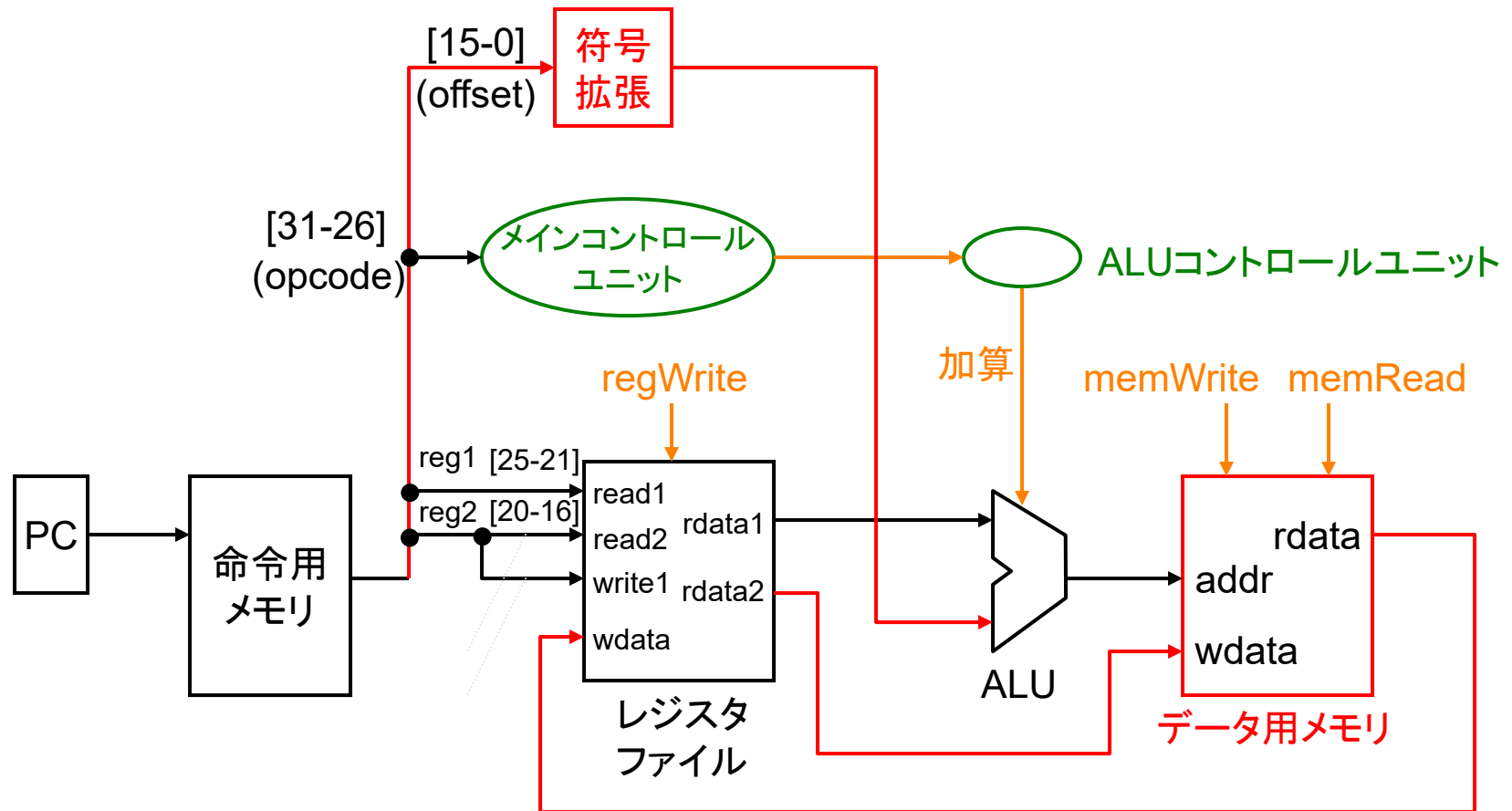
opcode	reg1	reg2	offset
[31-26]	[25-21] \$y	[20-16] \$x	[15-0]

- ▶ 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**



# MIPS クラス(ver. 3)

## メモリアクセス(のみ)のための回路

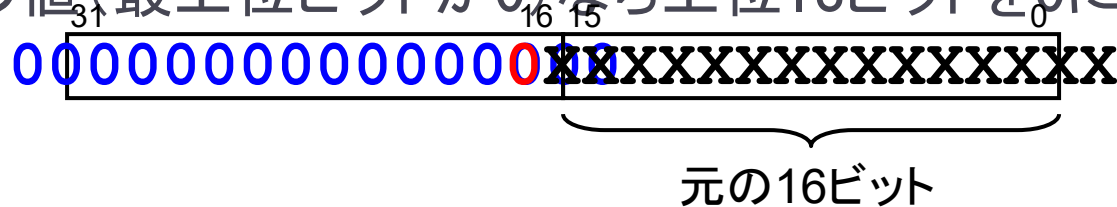




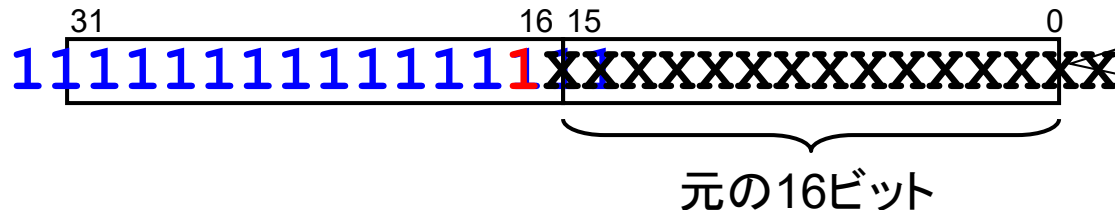
## 符号拡張

- ▶ 符号拡張＝「同じ値のまま、2進数のビット数を増やす」
- ▶ 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



正負反転(- ⇒ +)  
(2の補数)



符号拡張  
(上位16bitを0)



正負反転(+ ⇒ -)  
(2の補数)

## 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である必要がある
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	memToReg	regWrite	memRead	memWrite	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

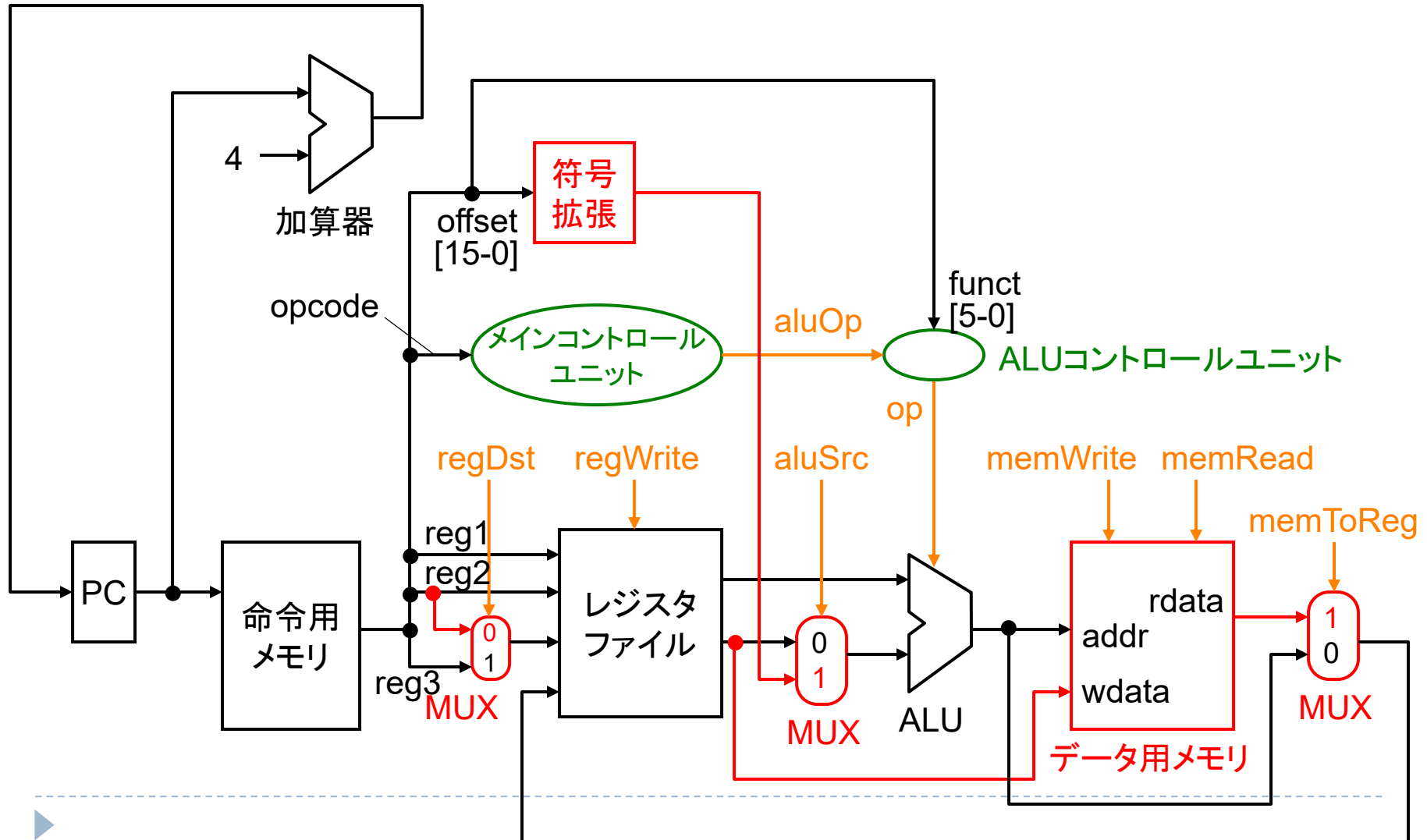
1. レジスタファイルからベースレジスタ(\$y)の値を読み出す
2. offset の値を 32 ビットに符合拡張する
  - ▶ ALUの入力は32bit => アドレス計算をするためには32bitである必要がある
3. ALU で \$y と offset の値を足す
4. 計算したアドレスの値をメモリから読み出す
5. その値を出力レジスタ(\$x)に書き込む

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



# MIPS クラス(ver.1 + ver.2 + ver. 3)

## メモリアクセスを追加した回路



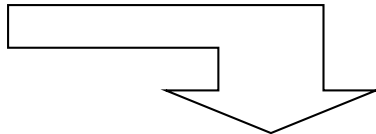
# MIPS クラス(ver.4):

## 分岐命令のフォーマット

---

▶ **beq \$x, \$y, label**

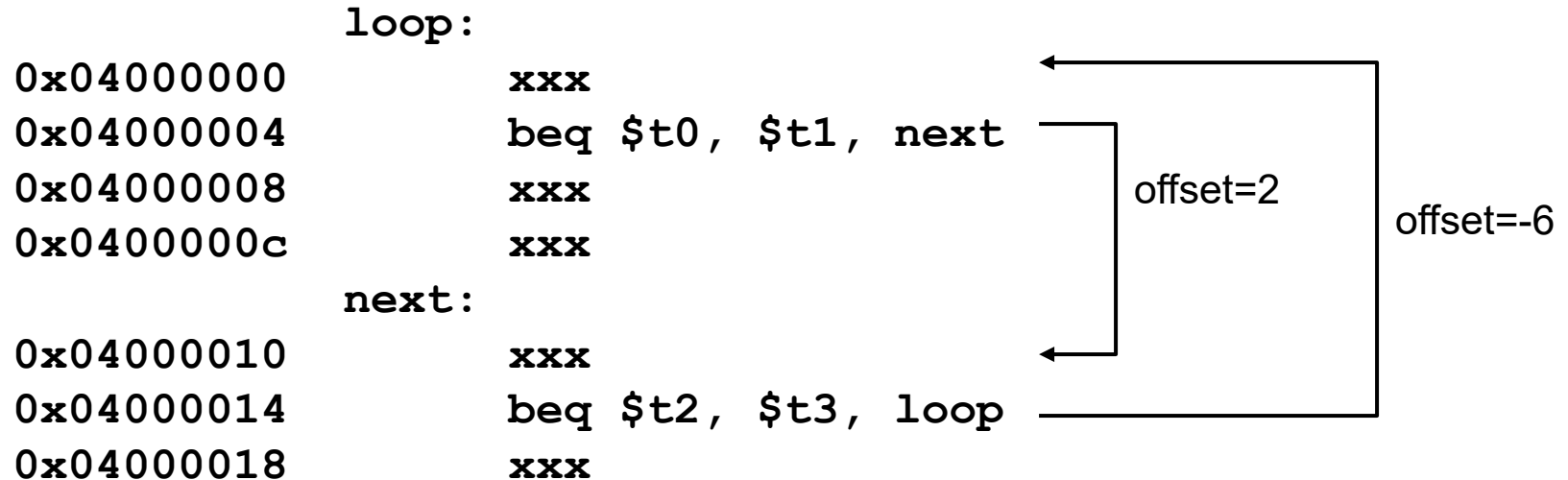
※SPIMがlabel⇒相対命令数へ変換している



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

- ▶ 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

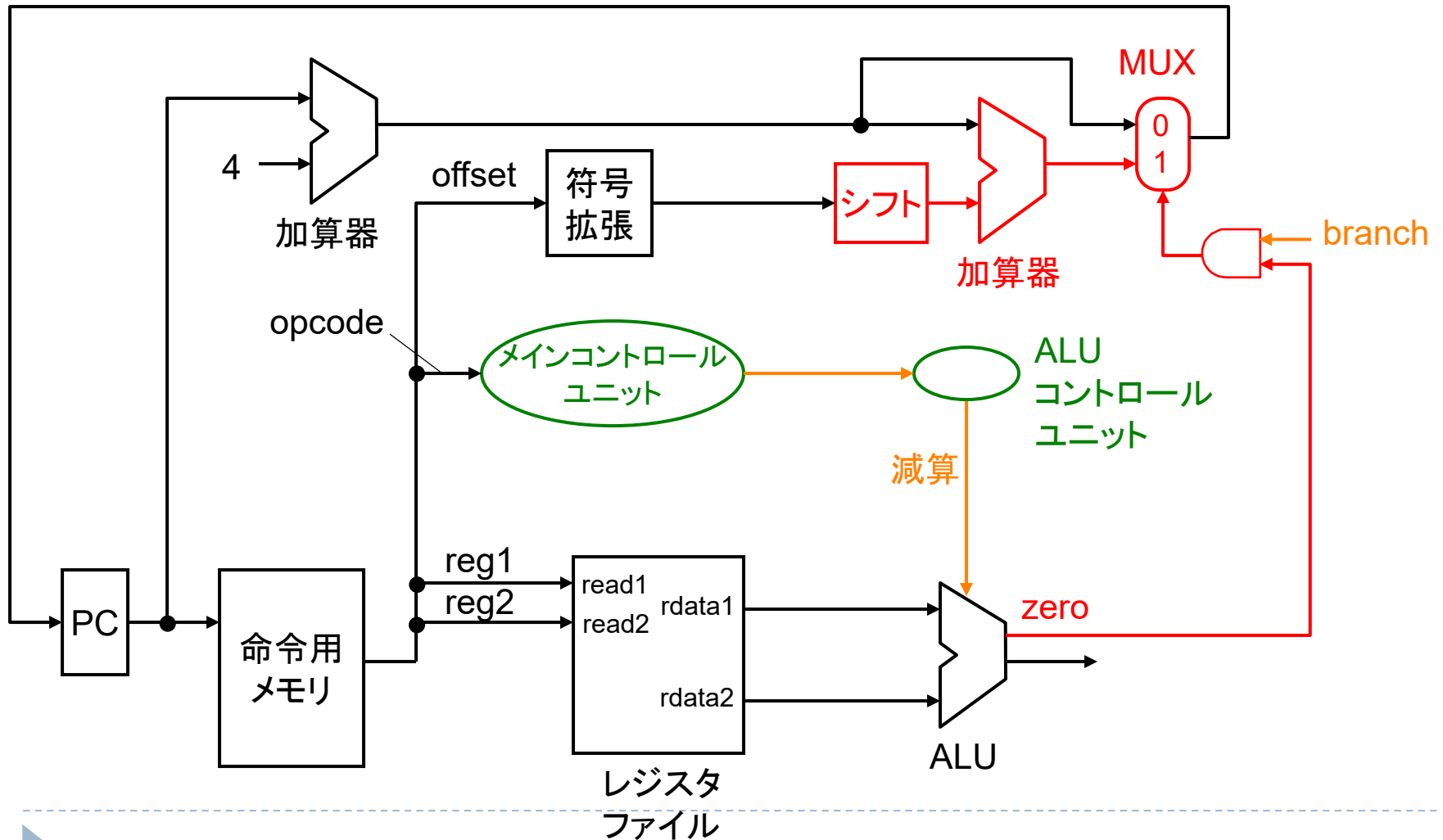


spim のオフセット計算は **PC+4の位置から**  
ジャンプする**命令数(ワード数)**になっているので注意

⇒ バイトアドレスに変換する必要がある

# MIPS クラス(ver.4)

## 分岐（のみ）のための回路



## 分岐命令の実行手順

---

1. ジャンプ先のアドレス  $PC + 4 + \text{offset} \times 4$  を計算する
  - ◆ 4倍は2ビット左シフトで実現
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





# MIPS クラス(ver.1 + ver.2 + ver.3 + ver.4)

## 分岐を追加した回路

