

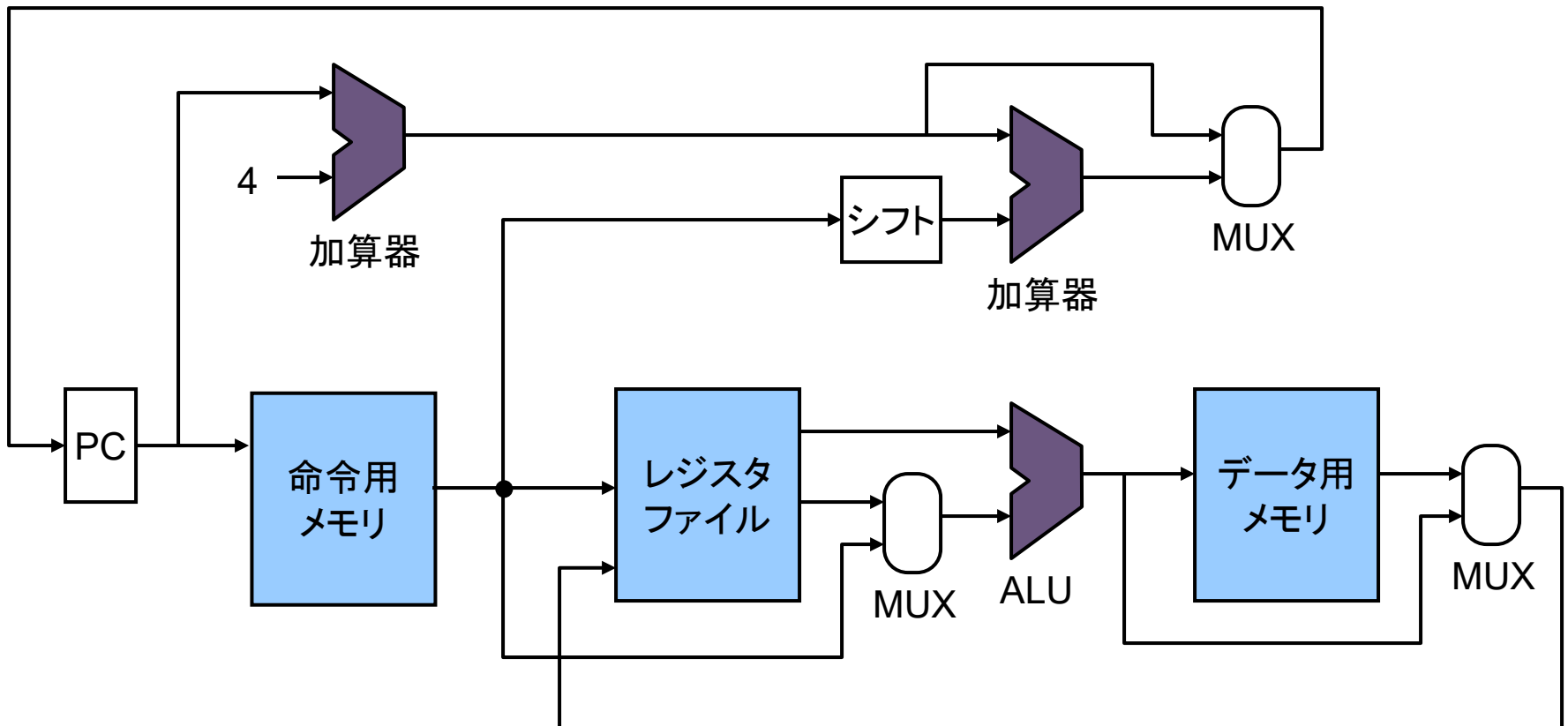
2018年度 計算機システム(演習)
第7回
2019.01.22

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)

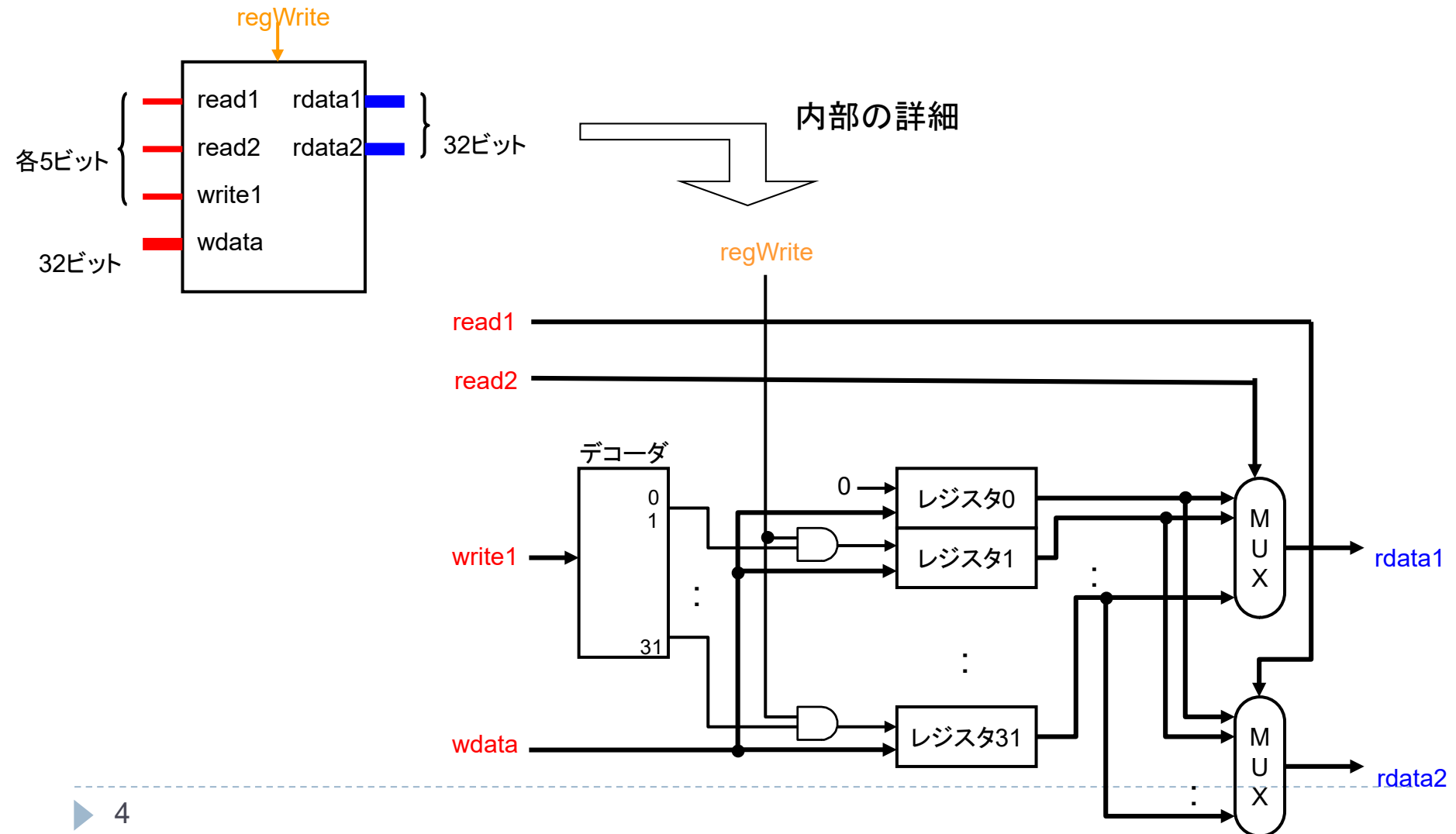
MIPSシミュレータ構築の流れ

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. 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シミュレータの全体像



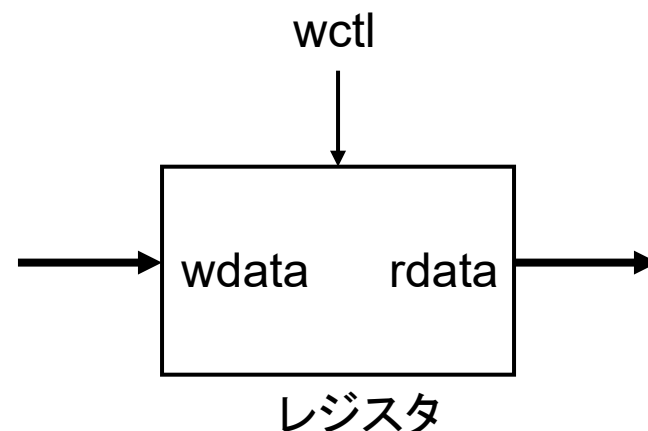
レジスタファイルの概要



レジスタ

- ▶ 状態回路で構成される
 - ▶ 内部状態を持ち、入力と状態により出力が決まる回路
 - ▶ 組み合わせ回路では入力のみから出力が決まった
- ▶ 入力
 - ▶ wctl: 書き込み制御フラグ (1bit)
 - ▶ wdata: 書き込むデータ (32bit)
- ▶ 出力
 - ▶ rdata: 読み出したデータ (32bit)
- ▶ 内部
 - ▶ val: 記憶されている値

```
typedef struct {  
    int val;  
} Register;
```



Register

```
void register_run(Register *reg, // ポインタで渡す
                  Signal wctl, Word wdata, Word *rdata)
{
    // 1. レジスタの値valをrdataに出力
    // 2. wctlの信号が1ならwdataの値を書き込む(if文を使ってよい)
    // 書き込む前の値を読み出せるように、読み出し、書き込みの
    // 順番で行う必要がある(同時に読み書きする場合への対策)
}
// 次の演習で使います
void register_set_value(Register *reg, int val)
{
    reg->val = val;
}
int register_get_value(Register *reg)
{
    return reg->val;
}
```

レジスタファイル

▶ レジスタの集合

- ▶ 機能: 同時に2つのレジスタを読み、1つのレジスタに書き込める
 - ▶ 例: `add $t0, $t1, $t2`

▶ 入力

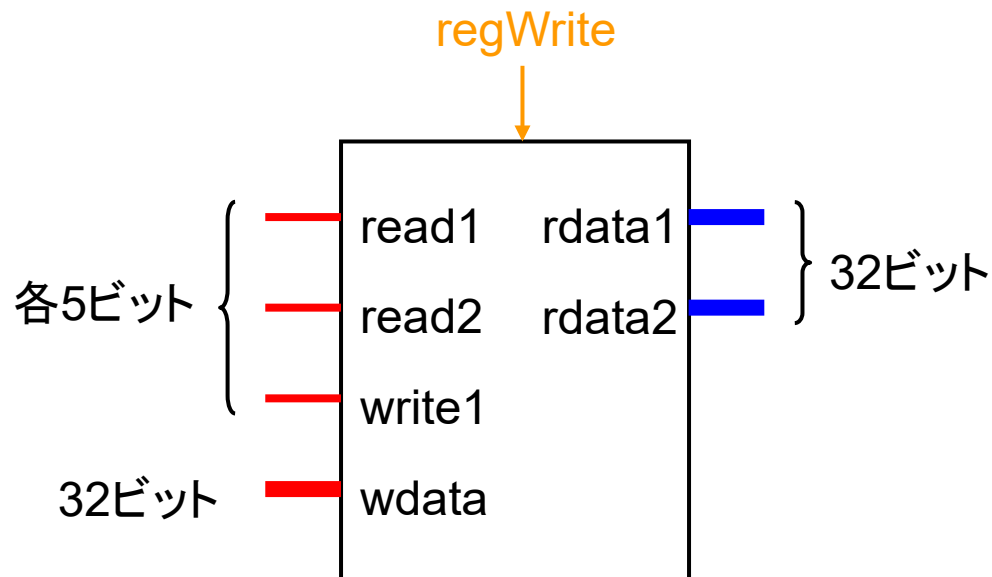
- ▶ `read1, read2, write1`
 - ▶ 読み書きするレジスタ番号
- ▶ `wdata`
 - ▶ 書き込むデータ
- ▶ `regWrite`
 - ▶ 書き込み制御信号

▶ 出力

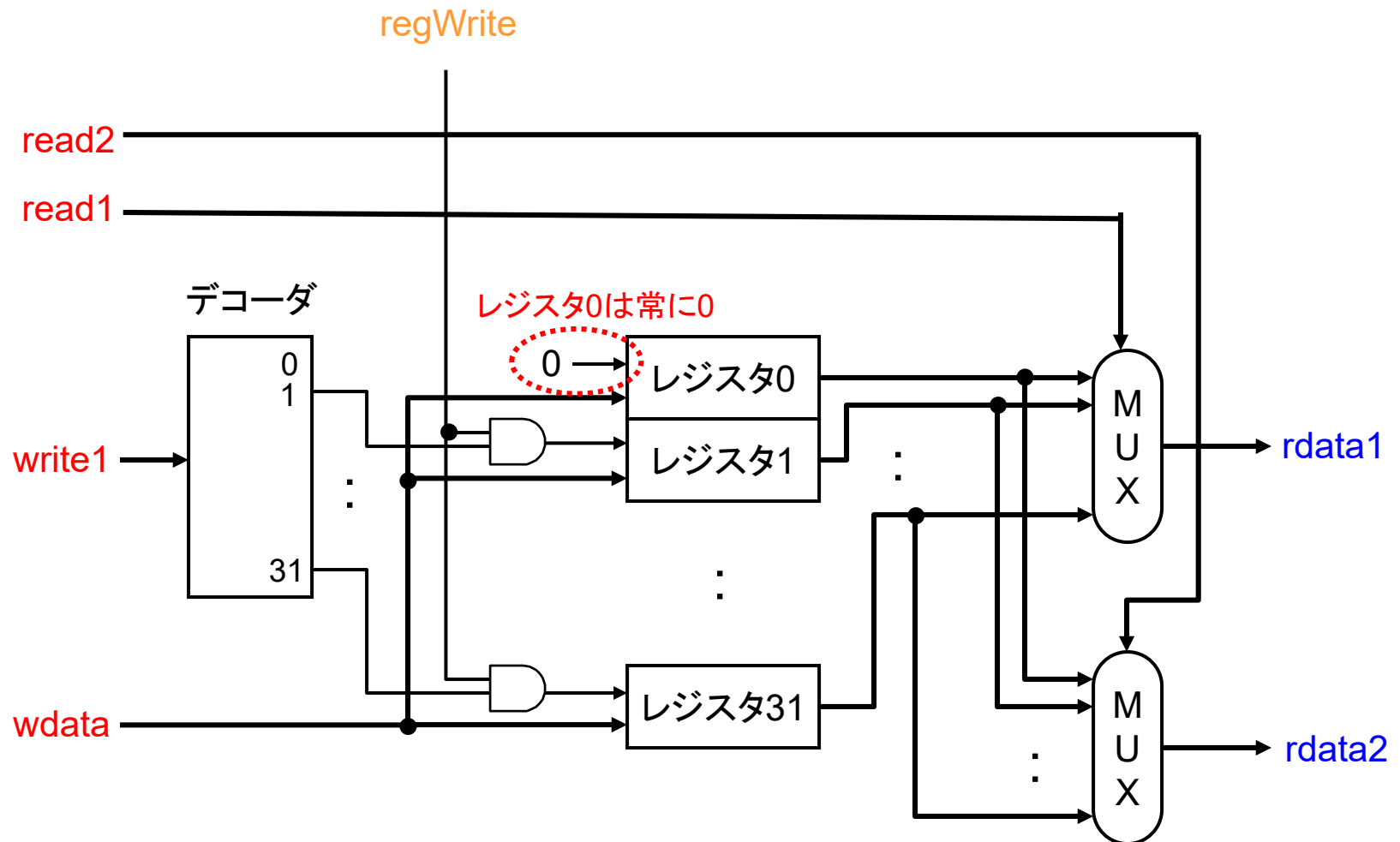
- ▶ `rdata1, rdata2`
 - ▶ 読んだデータ

▶ 内部

- ▶ 32個のレジスタ

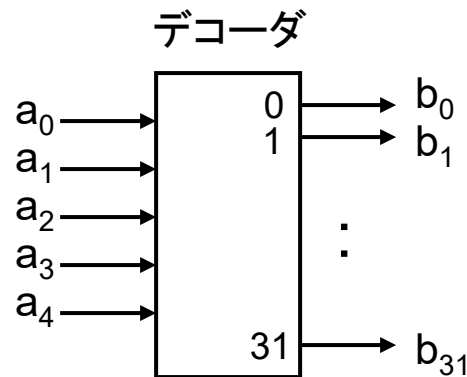


レジスタファイルの回路図



デコーダ

- ▶ 入力された数値に対応するビットを1にして出力
 - ▶ マルチプレクサの制御入力の部分に似ている
 - ▶ n 本の制御入力を使って 2^n 本の入力から1つ選ぶ
 - ▶ 例: $a = 00011$ (10進数で **3**) が入力されたら $b_3 = 1$ になる (他の b_i は 0)

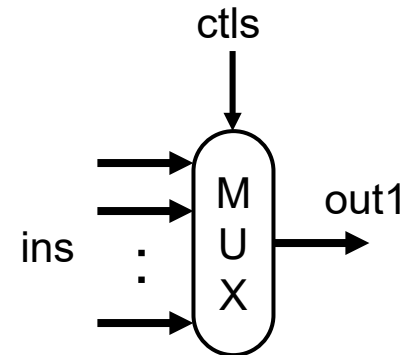


Decoder5

```
// 5ビット(a)から32ビット(b)へのデコーダ
void decoder5 (Signal a[5], Word *b)
{
    int i, val = 0;
    // 5ビットのaを10進法の値に変換
    for (i = 0; i < 5; ++i) {
        if (a[i]) {
            val += (1 << i);
        }
    }
    // word bの内、aの値によって選ばれる配線の値だけを1に設定する
    // (後の配線は0)
    // 回路を用いずに作成してよい(if文等使用)
}
```

MUX32

```
// 32入力のマルチプレクサ(word版)  
// 回路を用いずに作成してよい(if文等使用可)  
void mux32 (Word ins[32], Signal ctls[5], Word *out)  
{  
    // ctlsの値によって選択される入力wordの値を出力wordに設定する  
}
```



RegisterFile

```
//RegisterFile構造体
typedef struct {
    Register r[32];
} RegisterFile;
```

```
// デコーダ、レジスタ、32ビットMUXを接続する
void register_file_run(RegisterFile *rf, // ポインタで渡す
    Signal register_write, // 入力
    Signal *read1, Signal *read2, // 入力
    Signal *write1, Word wdata, // 入力
    Word *rdata1, Word *rdata2) // 出力
{
    // デコーダ、各ANDゲートとレジスタ、32ビットMUXを実行する
}
```

RegisterFile の使い方の例

// 1番レジスタへの書き込みと読み出しを同時に行う

```
void test_register_file()
```

```
{
```

```
    Signal register_write;
```

```
    Signal read1[5] = {true, false, false, false, false}; // 1
```

```
    Signal read2[5] = {true, false, false, false, false}; // 1
```

```
    Signal write1[5] = {true, false, false, false, false}; // 1
```

```
    Word wdata, rdata1, rdata2;
```

```
    RegisterFile rf;
```

```
    register_write = true; // 書き込み
```

```
    word_set_value(&wdata, 100);
```

```
    register_file_run(&rf, register_write, read1, read2, write1, wdata,  
&rdata1, &rdata2); // 1番レジスタにwdataの値を書き込み
```

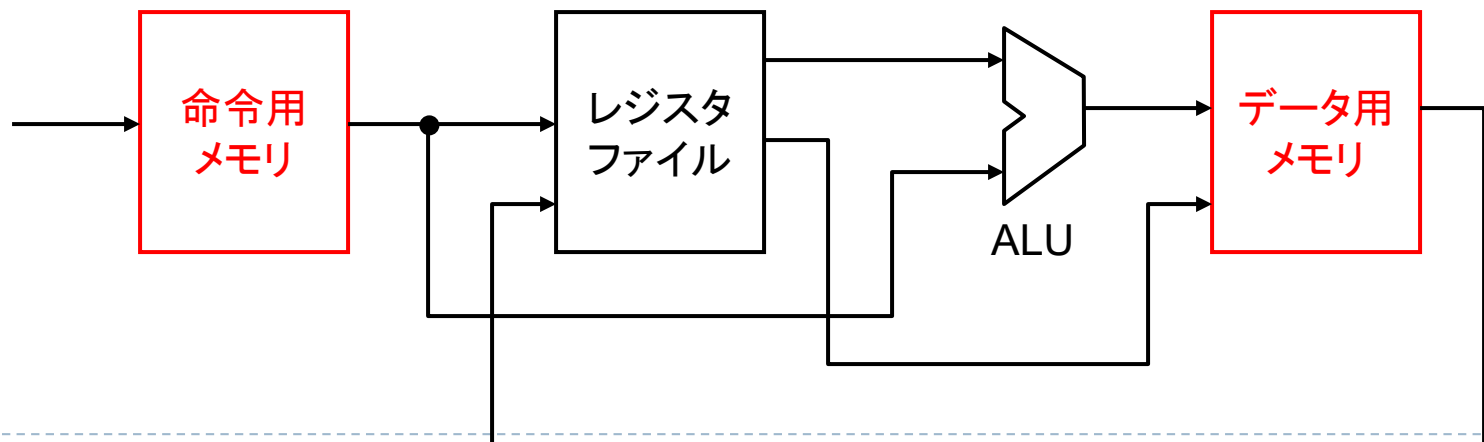
```
    register_file_run(&rf, register_write, read1, read2, write1, wdata,  
&rdata1, &rdata2); // 出力のrdata1に$1に書き込んだwdataが出力される
```

```
    printf("new data of rdata1: %d¥n", word_get_value(rdata1)); // 100
```

```
}
```

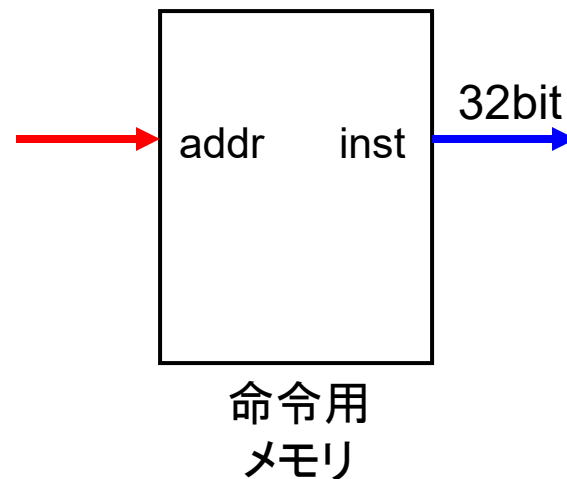
メモリ

- ▶ メモリを命令用とデータ用に分離し、個別に実装
 - ▶ 通常は区別はない
 - ▶ 制御を簡単にするため
- ▶ メモリは回路を用いずに実装する
 - ▶ 回路を用いると巨大になるため



命令用メモリ

- ▶ 命令用メモリは読み出し専用
 - ▶ 入力: 16進数アドレス(バイトアドレッシング)
 - ▶ 命令サイズ: 4byte (32bit)
 - ▶ 4byte (32bit) 毎に命令を格納
 - ▶ 出力: 命令
- ▶ アドレス 0x04000000 から始める
 - ▶ spim と同じ開始アドレス
- ▶ Spimの「.textセグメント」に相当



InstMemory

```
#define INST_MEM_START 0x04000000 //mips.hの先頭に記述
```

```
typedef struct {  
    int mem[1024];  
} InstMemory;
```

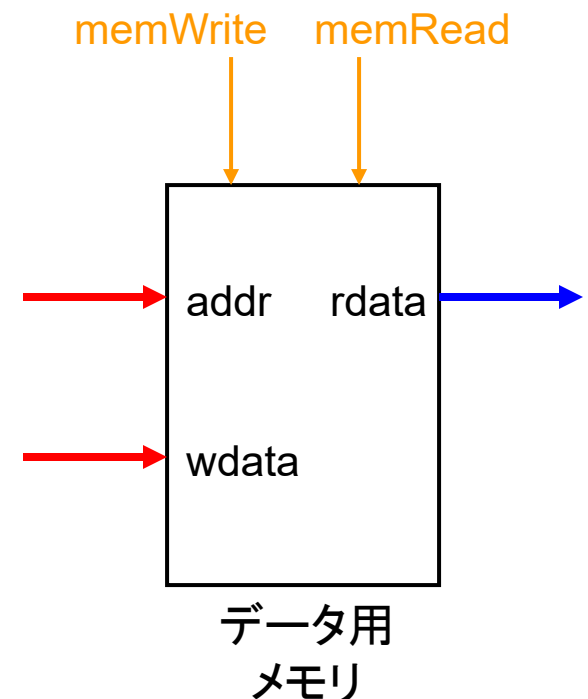
```
void inst_memory_run(InstMemory *im, Word addr, Word *inst)  
{  
    // instにデータを送る  
    // 命令アドレスは0x04000000から始まるので、  
    // 0x04000000をmem[0]に対応させると効率がよい  
    // memはワード単位なので4で割る  
    // mem[offset]の値をinstに設定  
}  
  
void inst_memory_set_inst(InstMemory *im, int addr, int inst){  
    // 命令instをメモリに書き込む処理を書く  
    // 0x04000000はmem[0]に対応  
}
```


InstMemoryの使い方例

```
void test_inst_memory()
{
    Word addr, inst;
    InstMemory im;
    word_set_value(&addr, 0x04000000);
    inst_memory_set_inst(&im, 0x04000000, 350);
    inst_memory_run(&im, addr, &inst);
    printf("InstMemory[0x04000000] = %d¥n",
word_get_value(inst)); // 350
}
```

データ用メモリ

- ▶ データ用メモリは読み書き可
 - ▶ 制御入力: memWrite、memRead
 - ▶ 読み書きを制御
 - ▶ 入力: アドレス、書き込むデータ
 - ▶ 出力: 読み出したデータ
- ▶ アドレス 0x10000000 から始める
 - ▶ spim と同じ開始アドレス
- ▶ Spimの「.dataセグメント」、スタック等に相当



DataMemory

```
#define DATA_MEM_START 0x10000000 //mips.hの先頭に記述  
  
typedef struct {  
    int mem[1024];  
} DataMemory;
```

DataMemory つづき

```
void data_memory_run(DataMemory *dm,
                    Signal mem_write, Signal mem_read,
                    Word addr, Word wdata,
                    Word *rdata)
{
    int val, offset;
    val = word_get_value(addr);
    offset = (val - DATA_MEM_START) / 4;
    // 本来ならmemReadとmemWriteが1かどうかはANDGateを
    // 使って判定するが、ここでは回路を用いないのでif文を使ってよい
    // 読み出し処理(メモリの値をrdataに設定)
    if (mem_read) {
        ...
    }
    // 書き込み処理(wdataの値をメモリに設定)
    else if (mem_write) {
        ...
    }
}
```

DataMemoryの使い方例

```
void test_data_memory() {
    DataMemory dm;
    Signal mem_write, mem_read;
    Word addr, wdata, rdata;
    word_set_value(&rdata, 0);
    //アドレスと書き込むデータを指定して、書き込みフラグをtrueに
    mem_write = true;
    mem_read = false;
    word_set_value(&addr, 0x10000004);
    word_set_value(&wdata, 100);
    data_memory_run(&dm, mem_write, mem_read, addr, wdata,
&rdata);
    word_set_value(&addr, 0x10000008);
    word_set_value(&wdata, 200);
    data_memory_run(&dm, mem_write, mem_read, addr, wdata,
&rdata);
    printf("rdata : %d¥n", word_get_value(rdata)); // 0
}
```

DataMemoryの使い方例（つづき）

```
// 読み込みフラグをtrueにしてデータの読み込み
mem_write = false;
mem_read = true;
data_memory_run(&dm, mem_write, mem_read, addr, wdata,
&rdata);
printf("rdata : %d¥n", word_get_value(rdata)); // 200
word_set_value(&addr, 0x10000004);
data_memory_run(&dm, mem_write, mem_read, addr, wdata,
&rdata);
printf("rdata : %d¥n", word_get_value(rdata)); // 100
}
```

課題

課題1

- ▶ レジスタファイルを完成させよ
 - ▶ Register、Decoder5、MUX32、RegisterFile
 - ▶ 適宜自分で作成したRegisterFileをテストせよ
 - ▶ register_test関数を適宜作成せよ
 - ▶ 同じレジスタへの書き込みと読み出しを行うテストすること
 - run()を2回実行する必要がある

課題2

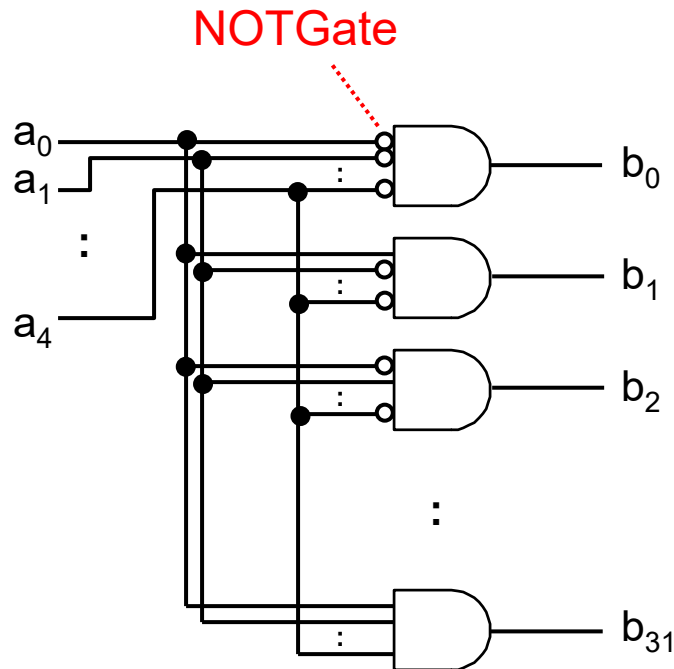
- ▶ メモリを完成させよ
 - ▶ InstMemory、DataMemory
 - ▶ 適宜テストを行うこと
 - ▶ データを書き込んだ後に正しく読み出せることを確認すること

課題提出

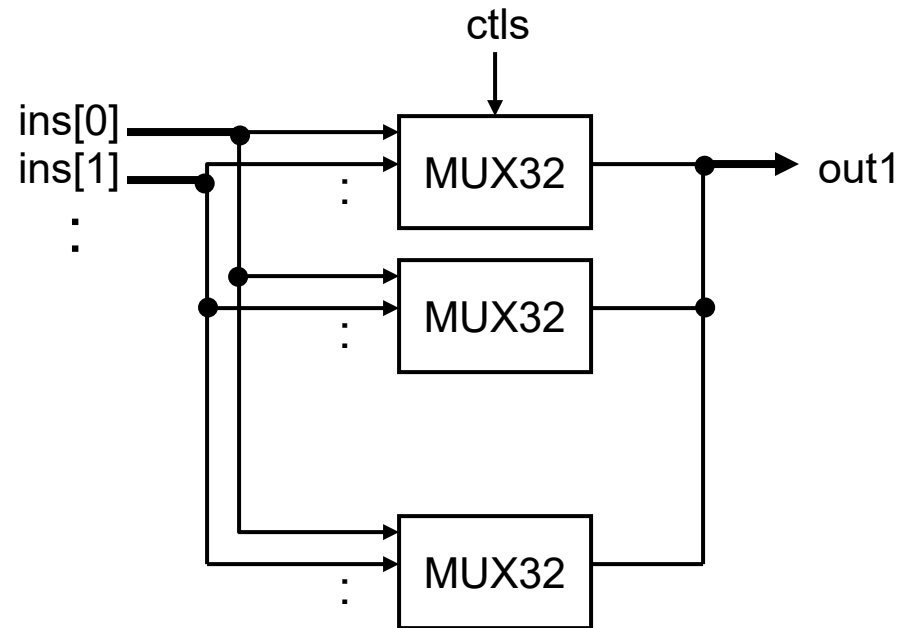
- ▶ ✕ 切: **2/5 (火) 23:59**
 - ▶ 締め切り日には講義・演習はないので質問があればメールで
- ▶ 提出物: 以下のファイルを1つのファイルに**圧縮したもの**
 - ▶ プログラムソース
 - ▶ シミュレータ全体ではなく、関係するコード(変更したファイル)のみ提出すること
 - ▶ 注意: 作成したプログラムは今後も使用するため、十分にテストすること
 - ▶ ドキュメント
 - ▶ テスト結果・工夫した点・感想等
- ▶ 質問等があれば compsys18@el.gsic.titech.ac.jp まで
 - ▶ 課題のレポートやコメントに書かれていると、返信が遅くなります

補足

- ▶ Decoder5 クラスおよび MUX32_bus クラスを回路を使って実現する



Decoder5



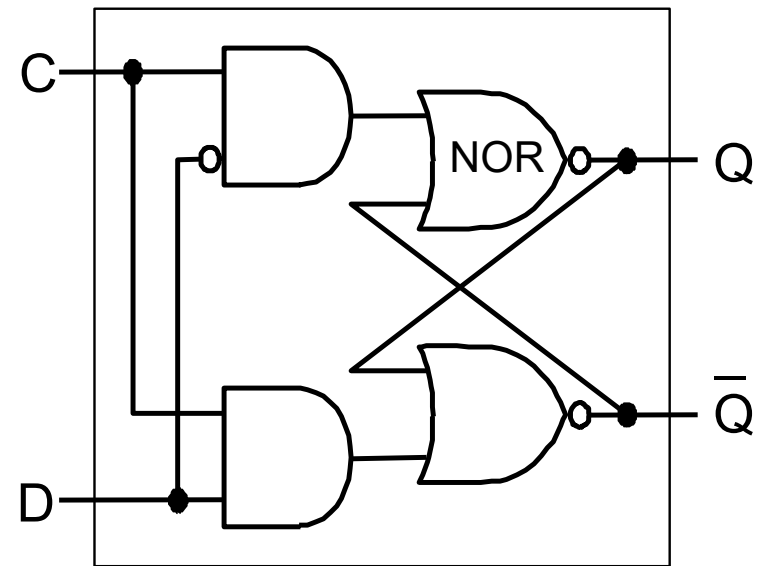
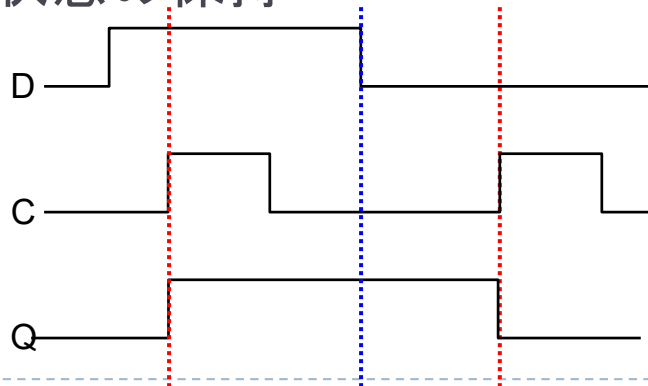
MUX32_bus

補足

- ▶ レジスタをフリップフロップを使って作成
 - ▶ D-フリップフロップを配列として作成
 - ▶ 1ビットの情報を保持する
 - ▶ D-フリップフロップはD-ラッチの組み合わせ回路
 - ▶ 制御入力としてクロックを入れる
- ▶ 演習のプログラムのように手続き型言語でシミュレートする場合、1クロックの間にも回路全体が安定するまで複数回ゲートを評価する必要がある

補足:D-ラッチ

- ▶ クロック(C)が 1 の間、入力 D の値が出力 Q に反映される
 - ▶ メモリへの書き込み
- ▶ クロックが 0 の間、D の値が変化しても Q は変化しない
 - ▶ メモリからの読み出し
 - ▶ 状態の保持



NOR: ORの出力を反転したゲート

補足：D-フリップフロップ

- ▶ 下降クロックエッジでのみ
入力が出力に反映される
 - ▶ C=1 になった時に1つ目のD-ラッチに反映
 - ▶ C=0 になった時に2つ目のD-ラッチに反映

