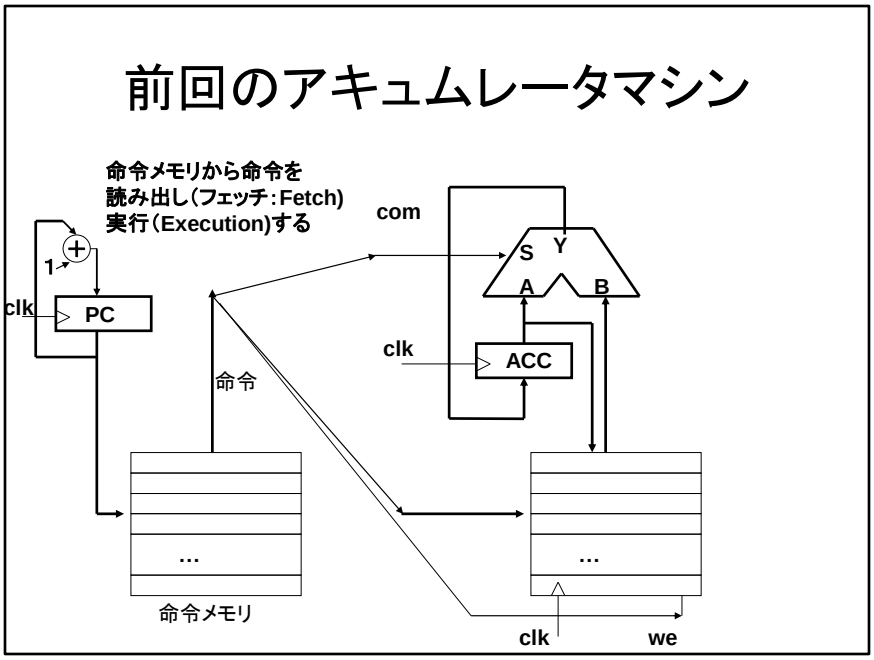
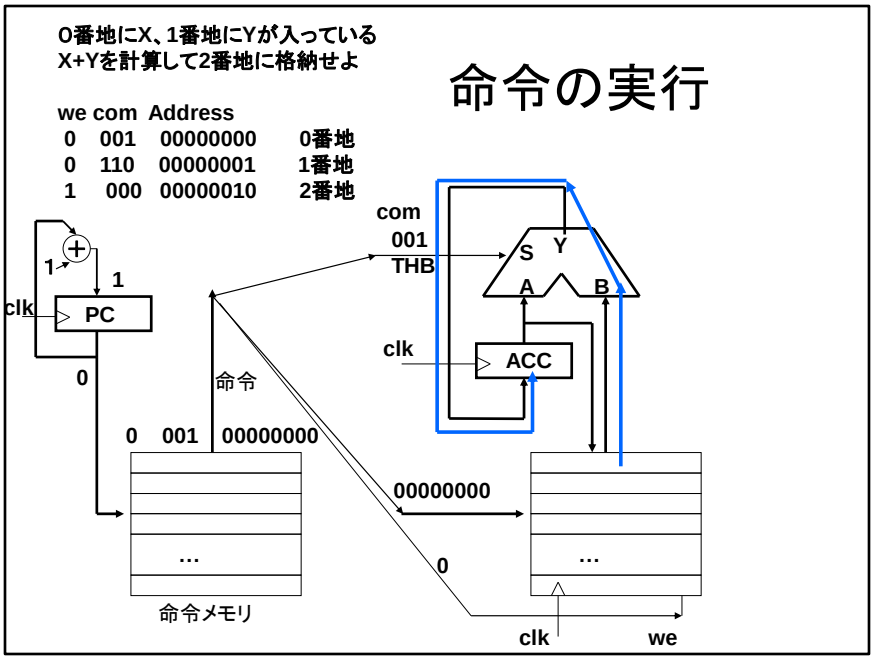
計算機構成　第4回
アキュムレータマシン

情報工学科
天野英晴

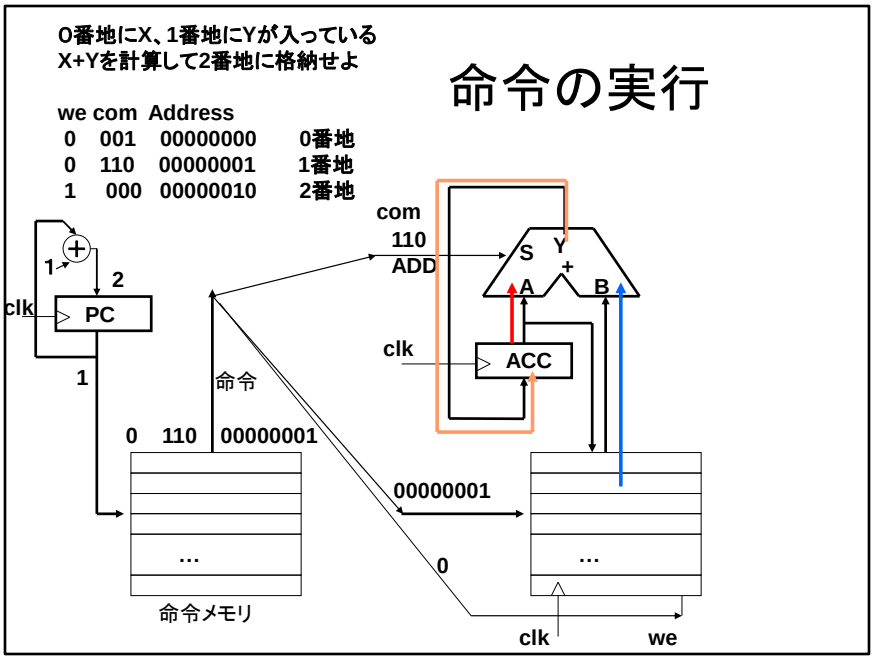
今回は、前回の構成に分岐命令を付けてプログラム格納型計算機のもっとも基本的な構成であるアキュムレータマシンを完成します。



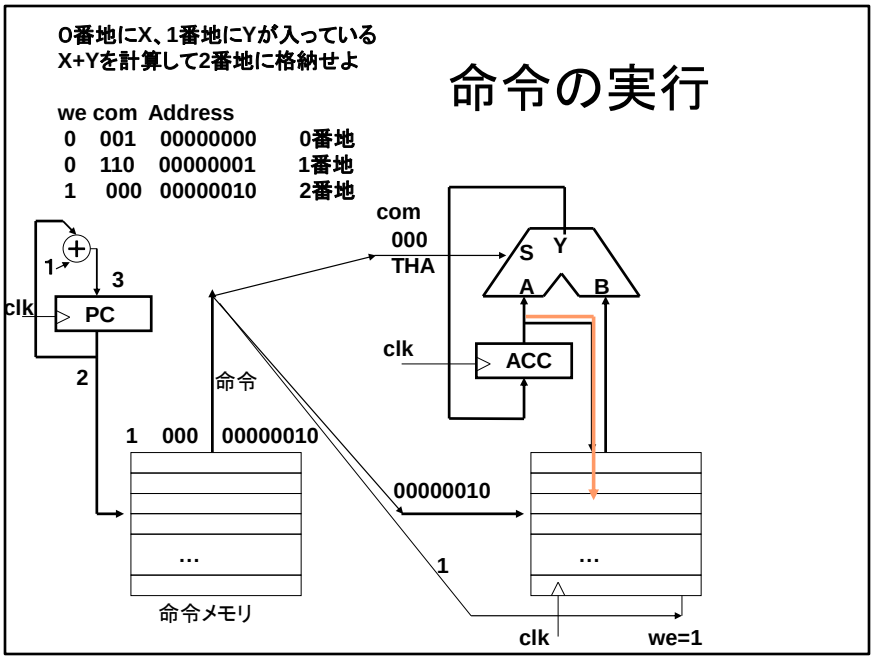
前回のアキュムレータマシンはPCに従って順番にメモリから命令を読み出し(フェッチし)、実行しました。PCは毎クロックカウントアップすることで次々と命令を実行することができます。



0番地にX、1番地にYが入っている状態で、X+Yを計算して2番地にしまうのは以下のように実行されます。最初のページは、PCが0でLD 0が実行されています。



次にPCが1になって、1番地の命令ADD 1が実行されます。この時ACCには前の命令の実行結果であるXが入っていますので、X+Yが計算されACCに保存されます。



次にPCは再びカウントアップされて2になり、ST　2が実行されます。ここではメモリのweが1になってACCの値が2番地に格納されます。

前回のマシンの問題点

- 命令メモリに入っている命令を一つずつ順番に実行する
- 判断と、それに基づいて処理を変えることができない
- 繰り返しができない

→ アルゴリズムが実行できない

- どうすればアルゴリズムが実行できるようになるのか？

→ 分岐命令

前回のマシンは、命令メモリに入っている命令を一つずつ順番に実行しました。すなわち猪突猛進で先に進むだけで、判断をしてそれに基づいてやることを変えることができません。これはすなわち繰り返しができないということです。繰り返しができないでやることを全部予め指定しておかなければならないと、これは不便です。自動機械として意味がないです。これはアルゴリズムが実行できないということになります。では、どうすればいいか、というと分岐命令をつけたし、計算結果について判断し、その結果に基づいてプログラムの実行を変えてやれば良いのです。

分岐命令の導入

- ACCの内容によってPCの内容を変更する
  - 制御命令：分岐（Branch)と呼ぶ

BEZ X Branch Equal Zero if ACC==0 PC←X

1001XXXXXXXX →opcodeは適当に決めた

(例） 100100000001 ACCが0ならばPCは1になる→次は1番地の命令を実行→1番地に「飛ぶ」

BNZ X Branch Not equal Zero if ACC!=0 PC←X

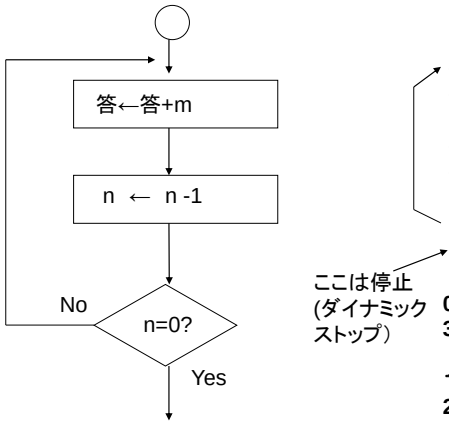
1010XXXXXXXX→opcodeは適当に決めた

(例）101000000001 ACCが0でなければPCは1になる→次は1番地の命令を実行→1番地に「飛ぶ」

オペランドは飛び先（命令メモリの番地）を示す：今までの命令と全く違うことに注意！

ここでは、最も簡単な分岐命令を導入します。これはACCの中身に応じてPCの内容を変更します。BEZ Xは、Branch Equal Zeroのニーモニック表現で、ACCが0ならばPCはXになります。つまり、次はX番地からプログラムを実行します。このことを分岐が成立（taken)したと呼びます。日本語では、Xに飛ぶと言うことが多いです。条件が成り立たなければ、分岐命令は無視され、普段通り次の命令を実行します。つまりこの命令ではなにもやらないことになります。
ここで、オプコードは1001にしました。これは1000がST命令なので単純にその次を割り当てたのです。BNZはBranch Not equal Zeroで、ACCが0でなければPCがXになり、プログラムはX番地に飛びます。オプコードは1010です。この分岐命令のオペランドは飛び先の命令メモリのアドレスです。分岐命令以外の命令は、この部分がデータメモリのアドレスに当たっていました。この点でも分岐命令はそれ以外の命令と全く違っていることがわかります。

分岐命令によるアルゴリズムの実行




```
0001 00000000    LD  0
0110 00000001    ADD 1
1000 00000010    ST 0
0001 00000010    LD 2
0111 00000011    SUB 3
1000 00000010    ST 2
1010 00000000    BNZ 0
1001 00000111    BEZ 7
```

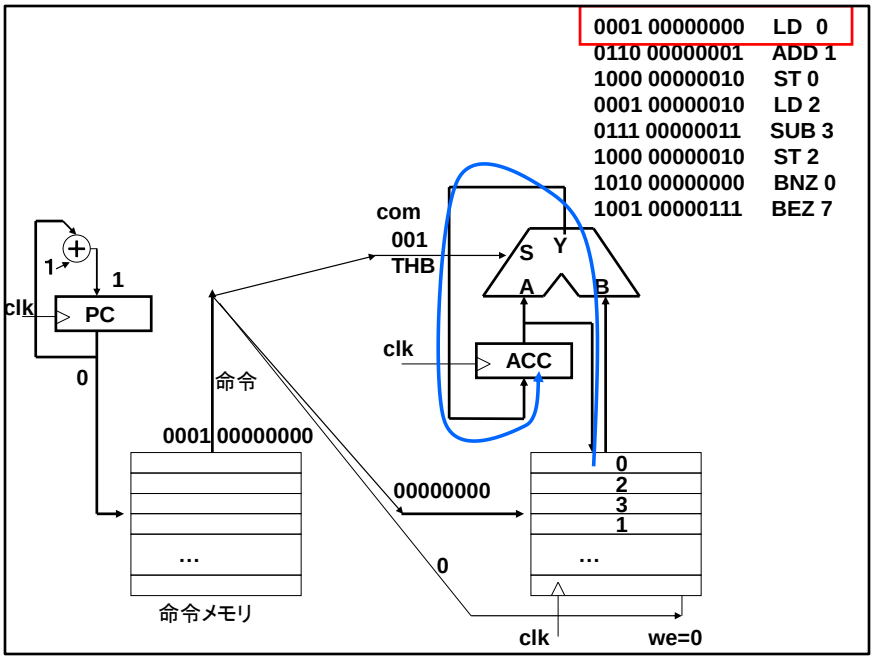
ここは停止
(ダイナミック
ストップ)

0番地 0
3番地 1

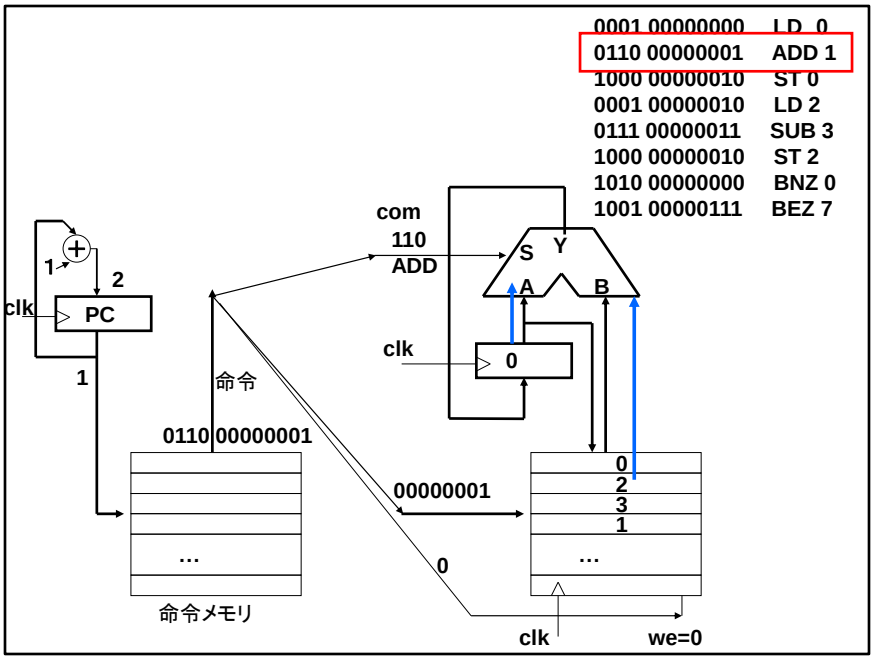
1番地 m
2番地 n

繰り返しによりアルゴリズムの実行が可能
→ プログラム格納型(Stored Program)方式

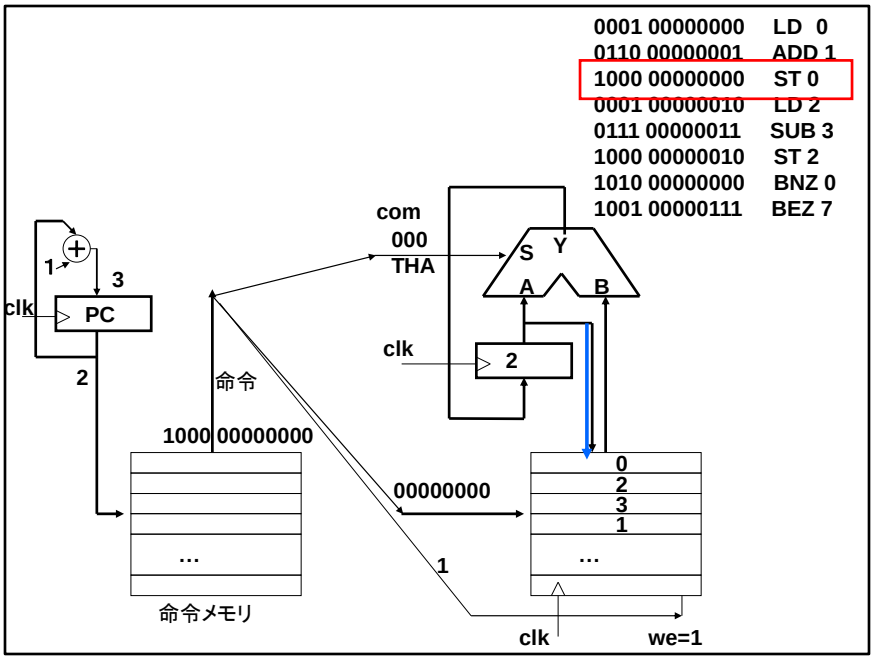
分岐命令を使うと繰り返しを使ったアルゴリズムが実行できます。この図は、1番地の値mと2番地の値nの値の掛け算を行うプログラムです。まず、0番地を0に初期化しておき、1番地にm、2番地にnを入れておきます。さらに3番地を1にしておきます。0番地からACCにLDし、1番地の内容を足して0番地にしまいます。これでmを1回足すことになります。次に2番地の値をLDし、1を引きます。このために3番地を1にしておくわけです。引いた値は再び2番地に戻します。これで、mを一回足すたびにnから1を引くことになります。ACCにはnが残っているので、ここでBNZ 0を実行すると、nが0でなければ、0番地にプログラムが戻り、以上の動作が繰り返されます。0になれば、プログラムは終了します。このままではプログラムカウンタが先に進んでしまうため、次にBEZ 7を入れておきます。ここまで来た時はACCの値は0なので、このBEZは必ず成立し、自分自身に飛ぶので、無限ループになります。これによりプログラムは停止します。このように無限ループを利用して停止させることをダイナミックストップと呼ぶことがあります。



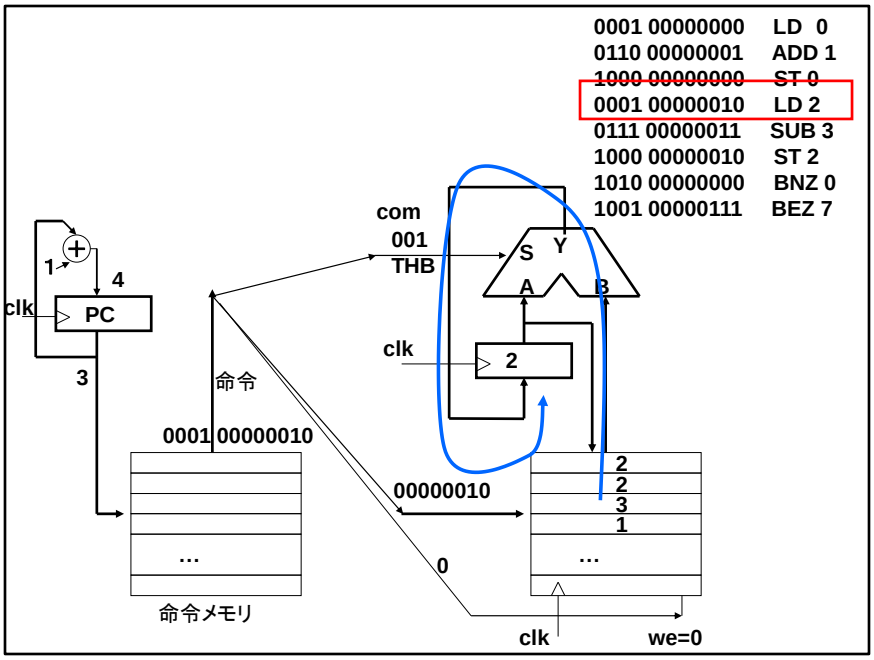
では、このアルゴリズムの実行の様子を示します。まず、LD 0で0番地のデータをACCにロードします。



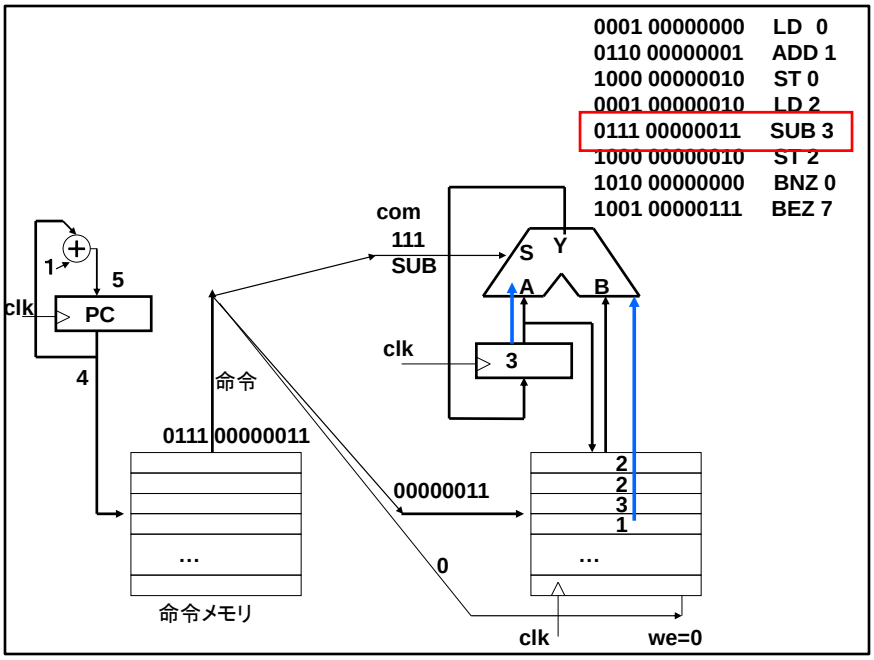
次に1番地の値mを加算します。ここでは2が入っています。



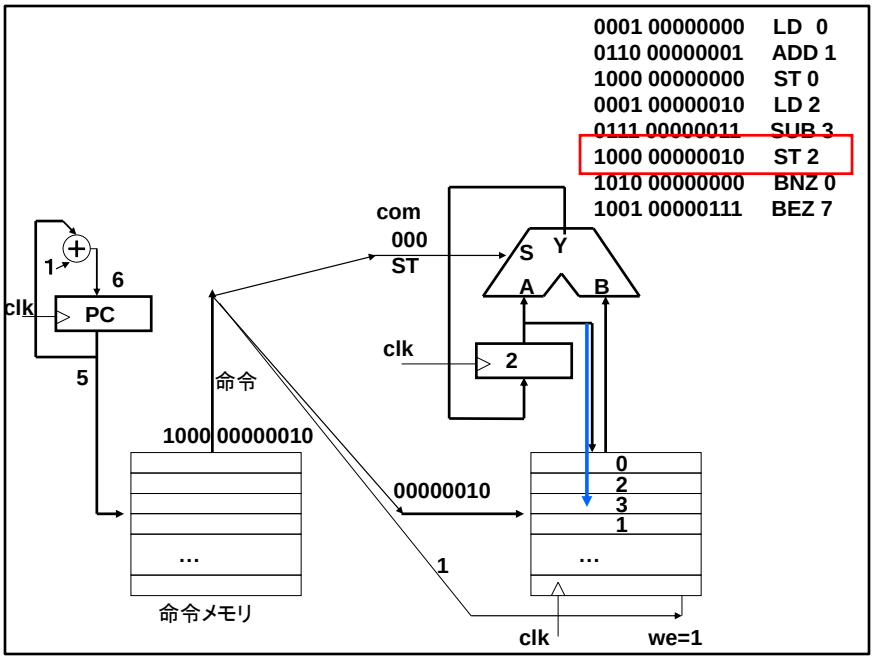
この結果を0番地に格納します。今まで0だったのが2になりました。



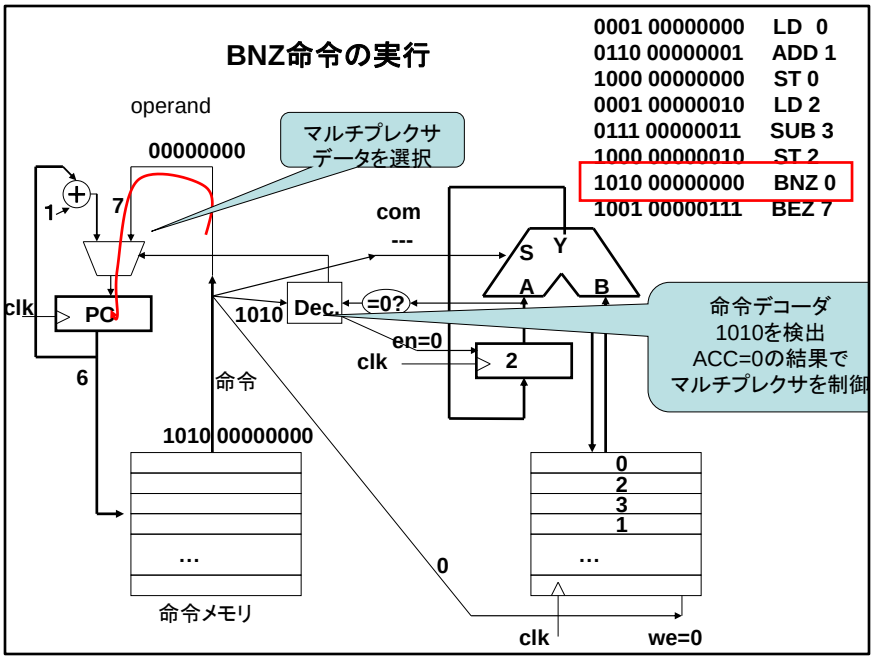
LD 2で、今度はmをACCに持ってきます。mは3でした。



mから1を引くためにSUB　3を実行します。3番地には1があるので、これでｍ－１が実行されたことになります。



この値を2番地に格納します。



ここで、BNZ　0を実行します。このためには、命令デコーダで1010を検出します。同時にACCの中身が0かどうかを調べます。両方共比較的簡単な回路で検出可能です。1010でACCの値が0でなければ、命令中の下位8ビットをPCにセットします。この場合PCは0となり、プログラムは0番地から再び実行されます。ACCの中身が0でならば、通常通りPC+1がPCにセットされます。

アキュムレータマシンのVerilog記述 入出力とレジスタ、ワイヤの宣言

```
`include “def.h”
module amb(
  input clk, input rst_n,
  input [`OPCODE_W-1:0] opcode,
  input[`ADDR_W-1:0] operand,
  input[`DATA_W-1:0] ddatain,
  output we,
  output  reg [`ADDR_W-1:0] pc,
  output reg [`DATA_W-1:0] accum);

wire [`DATA_W-1:0] alu_y;
wire op_st, op_bez, op_bnz;
```

命令メモリ

データメモリ

命令メモリのアドレス

命令のデコード信号

では、分岐命令のついたアキュムレータマシンのVerilog記述を示します。入出力は前回のアキュムレータマシンと同じですが、ストア命令を検出する信号のほかに、BEZ命令、BNZ命令を検出する信号であるop_bez,op_bnzを用意しています。この信号はオプコードが1001と1010でそれぞれHレベルになります。

アキュムレータマシンのVerilog記述
デコードと入出力、ALUの接続

```
assign op_st = opcode== `OP_ST;
assign op_bez = opcode== `OP_BEZ;
assign op_bnz = opcode==`OP_BNZ;

assign we = op_st;
alu alu_1(.a(accum), .b(datain),
      .s(opcode[`SEL_W-1:0], .y(alu_y));
```

def.h

```
`define OP_ST 4'b1000
`define OP_BEZ  4'b1001
`define OP_BNZ 4'b1010
…
```

これを検出するVerilog記述は簡単で、オプコード部分を比較すればよいです。ALUの実体化の部分は前回と同じです。オプコードの下位3ビットがALUのsに入れている点にご注意ください。

命令のデコード

assign op_st = opcode== `OP_ST;

op_stはST命令がフェッチされたときだけ1になる

assign op_bez = opcode== `OP_BEZ;

op_bezはBEZ命令がフェッチされたときだけ1になる

assign op_bnz = opcode==`OP_BNZ;

op_bnzはBNZ命令がフェッチされたときだけ1になる

これらの信号線を使って、CPUの動作を制御する

この信号の生成を命令デコードと呼ぶ

今回は、ST命令、BEZ命令、BNZ命令だけをデコードし、他は同じパターンの命令と考える

→ メモリとアキュムレータの中身を演算して答えをアキュムレータに入れる

LD命令もこの一種として考える

一般的にオプコードを調べてどの命令がフェッチされかを調べる操作がコンピュータには必要になり、これを命令デコードと呼びます。今回のアキュムレータマシンは、ST命令、BEZ命令、BNZ命令だけをデコードします。他の命令は「メモリとアキュムレータの中身をALUで演算して答えをアキュムレータに入れる」という共通の処理であり、命令のオプコードの下位3ビットをALUのコマンドに入れてALUで行う演算の種類を変えて実現します。

アキュムレータマシンのVerilog記述 レジスタの制御

```
always @(posedge clk or negedge rst_n)
  begin
    if(!rst_n) pc <=0;
    else if (op_bez & (accum==0) | op_bnz & (accum!=0))
      pc <= oprand;
    else pc <= pc+1;
  end

always @(posedge clk or negedge rst_n)
  begin
    if(!rst_n)  accum <=0;
    else if(!op_st & !op_bez & !op_bnz)
              accum <= alu_y;
  end

endmodule
```

pcの制御

その他の命令ではアキュムレータにALUの出力を保存

次にpc周辺の記述です。今まではリセット以外ではpc+1を入れることで毎クロックカウントアップしていました。これに分岐命令の実装を付け加えます。分岐命令が成立するかどうかを調べます。これは、op_bezつまりBEZ命令でaccumが0の時、op_bnzつまりBNZ命令でaccumが0でない時にオペランドつまり飛び先がPCに入ります。
アキュムレータaccumの記述も変更します。op_st, op_bez, op_bnzの時はaccumに何をいれず、そうでない時にALUの出力をaccumに入れるように変更します。

イミーディエイト命令

- アキュムレータから1引きたい！
  - 3番地に1をあらかじめ入れておき
  - SUB 3
  - 直接1を足したり、引いたりできれば便利！

→ イミーディエイト命令（直値、即値命令）

ADDI #1 ACC←ACC+1

ADDI #-1 ACC←ACC-1

命令コード中の数字をそのまま計算に使う

便利なのでどのマシンでも持っている

次にもう一点改良を行います。アキュムレータから1を引く際に、3番地に1を入れておき、SUB 3を実行しました。しかし、直接1を足したり、引いたりできると便利です。これを実現するのがイミーディエイト命令（Immediate命令）といいます。日本語では直値、即値と呼びます。ADDI #1、ADDI #-1のように命令コード中の数字をそのまま計算に使うことができます。便利なのでどのマシンもこの命令を持っています。コード中の数字が直接演算に使われることを強調するために数字の頭に＃を付けています。本来これは必要ないですが、この授業では間違いを防ぐために、この記法を使います。

ADDI命令の符号拡張

- ADDI #X　1110　XXXXXXXX

  ADDI #1　1110_00000001

  ADDI #-1　1110_11111111

  opcodeの下位3ビット110をADDと共通にしておく

  ここで困ったことに気づく

  命令コード中の数字は8ビット分しか存在しない

  しかしデータは16ビット幅だ

  負の数も扱う必要がある

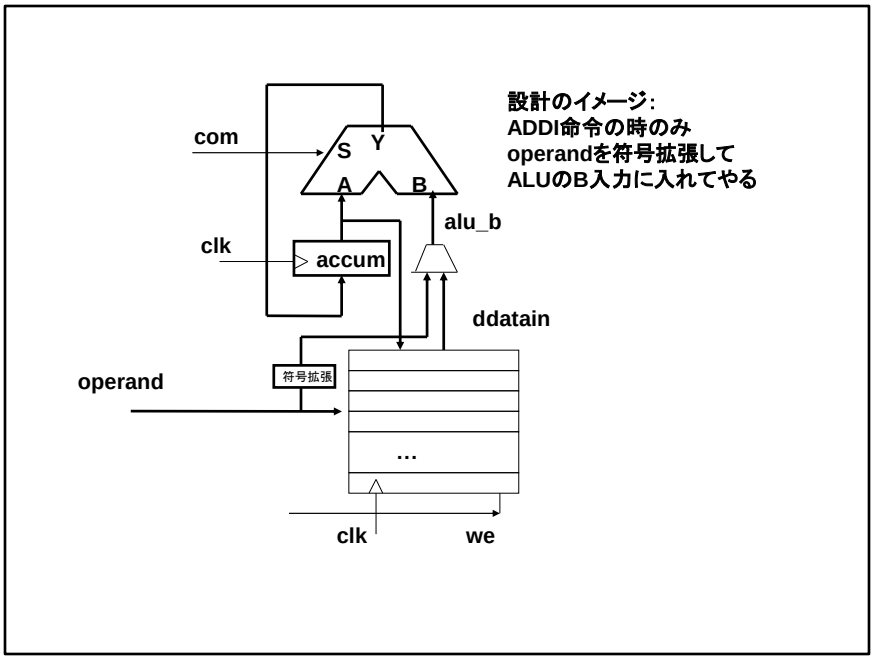
  → 符号拡張（sign extension)

  8bitの最上位の符号ビットを8ビット補って16ビットに数を引き伸ばしてやる

  2：00000010 → 0000000000000010

  −2：11111110 → 1111111111111110

ここではADDI命令を実行します。これには1110を割り当てます。下位8ビットを計算する値に割り当てます。ここで困ったことに気づきます。命令コード中に含むことができる数字は8ビットなのに、このアキュムレータマシンのデータ幅は16ビットあります。しかも1引いたりするので基本的に符号付の数を扱う必要があります。そこで、符号拡張（Sign Extension)を行います。これは8ビットの最上位の符号ビットを8ビット分上位に補い、数を16ビットに引き伸ばしてやる方法です。これは、コンピュータのあらゆる場所で使います。符号を考えない場合、0を埋めればよく、これをゼロ拡張と呼びます。



このイミーディエイト命令は、メモリからデータを読んでくる代わりに、オペランドを符号拡張したものをALUのB入力に入れることで実装します。これにはマルチプレクサを使います。

符号拡張とゼロ拡張のVerilog記述

{n{x}}はxをn回繰り返して並べることを意味する

- 同じ数の繰り返しは{繰り返し回数{数}}

例　{8{1'b1}} →　11111111　{3{16'habcd}} → abcdabcdabcd

{ {8{operand[7]}},operand} →
符号ビットを8ビット並べ、operandと連結→　符号拡張

{8'b0,operand} →
0を8個とoperandを連結→ゼロ拡張

ここで、符号拡張とゼロ拡張をVerilog記述を紹介します。{n{x}}はxをn回繰り返して並べる表現です。例をいくつかスライド中に示します。符号拡張はこれを利用します。operandは8ビットなので、符号ビットはoperand[7]です。これを8ビット並べてoperandと連結すれば符号拡張になります。ゼロ拡張は、この代わりに0を8つ並べます。この記述は{}が3重になるので、見難いですが、これは、ま、しょうがないと思ってくださいませ。

バスの連結 { , }

```
wire [3:0] a,b,c;
wire d;
wire [7:0] x,y;
assign x = {a,b};   4bitのバスを二つ連結して8bitにする。
assign y = {c,d,d,d,d}; 4bitに1bitを4つ連結して8bitにする。
```

{ }を使っていくつでもくっつけて一つのバスにできる。

連結を使ってバスの分割も可能

```
assign {a,b} = x;   assign a=x[7:4]; assign b=x[3:0];と同じ
assign {a,d,d,d,d,b,c} = {x,y}; などと書くこともできる
```

読みやすいので良く使う→左右の幅の違いに注意

ついでにバスの連結の記法を説明します。Verilogは{ , }の形で簡単に連結してバスの形にすることができます。右辺にも左辺にも使うことができます。読みやすいので良く使いますが、左右の幅の違いに注意してください。

{}で格好良く書ける

- 前回のテストベンチ
  - assign opcode = imem[pcout][11:8];
  - assign operand=imem[pcout][7:0];
- 今回のテストベンチ
  - assign {opcode, operand} = imem[pcout];
- 同じことを書いているが、下の方が分かりやすい

例えば、前回は分けて書いた記述も、{　}を使って分かり易く書くことができます。下の方が分かり易いと思います。この書き方は結構良く使うのですが、左辺と右辺の幅が必ず同じになるようにご注意ください。

イミーディエイト命令のVerilog記述

```
wire op_addi;
wire [`DATA_W-1:0] alu_b;
wire [`SEL_W-1:0] com;
assign op_addi = opcode == `OP_ADDI;
…
assign com = op_addi ? `ALU_ADD: opcode[`SEL_W-1:0];
assign alu_b = op_addi ? {{8{operand[7]}},operand}:
                                    ddatain;
alu alu_1(.a(accum), .b(alu_b), .s(com),  .y(alu_y) );
```

コマンドはcomに加算を入れてやる。
iverilog test_ambi.v ambi.v alu.v
./a.out > | tmpで結果を確認してみよう！
これは、test_ambi.datを使っている

イミーディエイト命令を付加するためにVerilog記述を修正します。オプコードがADDIの時を検出し、符号拡張をした値をALUのB入力に入れてやります。このためにop_addiとalu_bという信号を宣言します。alu_bはALUのB入力で、今まではメモリからの入力を直接入れてやったのですが、op_addiがHの時は、符号拡張したオペランドを入れてやるようにします。ここは、条件演算子を使います。
ADDIは加算を行いますが、ADDI命令は1110で定義してあり、下3ビットはALUのADDコマンドの110になっています。なので、以前通りオプコードの下3ビットを入れてやれば良いです。これは若干セコいテクニックで一般性はないです。では、イミーディエイト命令を使ったプログラムを動かしてみましょう。

演算命令

オプコード	ニーモニック	意味
0000	NOP	No Operation　何もしない
0001	LD　X	Load　ACC←　（Xの中身）
0010	AND	論理積 ACC←　ACC&（Xの中身）
0011	OR	論理和 ACC←　ACC｜（Xの中身）
0100	SL	左シフト　ACC←ACC<<1
0101	SR	右シフト　ACC←ACC>>1
0110	ADD	加算　ACC←ACC+（Xの中身）
0111	SUB	減算　ACC←ACC-（Xの中身）
1000	ST　X	Store　（X）　←　ACC

今まで出てきた演算、ロード、ストア命令をまとめます。

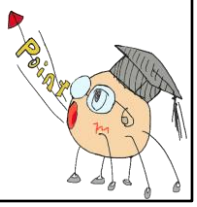
分岐命令、ADDI

オプコード	ニーモニック	意味
1001	BEZ　X	ACCが0ならばXに飛ぶ
1010	BNZ　X	ACCが0でなければXに飛ぶ
1110	ADDI　＃X	ACC←ACC＋X（符号拡張）

今回付け加えた分岐命令、ADDI命令をまとめます。

本日のまとめ

- 分岐命令はACCの中身を判断してPCの中身を書き換える
  - BEZ　ACCが0ならば成立
  - BNZ　ACCが0でなければ成立
- 分岐命令を使うとアルゴリズムが実行できる
- イミーディエイト命令は、コード中の数字を直接足すことができる
  - ADDI命令



インフォ丸が教えてくれる今日のまとめです。

今日のVerilog 構文

- **{n{x}}はxをn回繰り返して並べることを意味する**
- { , }で信号線を連結できる
  - 左右の幅の違いに注意！

だんだん新しい構文が減ってきました。今回はビット操作に関連するものです。

演習課題

演習1

- 1番地にXが格納されている。X+(X-1)+(X-2)+…2+1を計算するプログラムを実行せよ

- 提出物はimem.dat

演習2

- オペランドが符号拡張されてアキュムレータに入る

LDI #X　1011　XXXXXXXX

命令を実装せよ

ヒント:2つ方法がある

①ADDIと同じ方法で、ALUのcomを001にする。

②アキュムレータの入力に直接入れてやる

いずれも、op_ldiを定義せよ

test_ambi.vのimemの設定フィアルをimem_ldi.datに入れ替えてテストに用いよ

提出物はambi.v