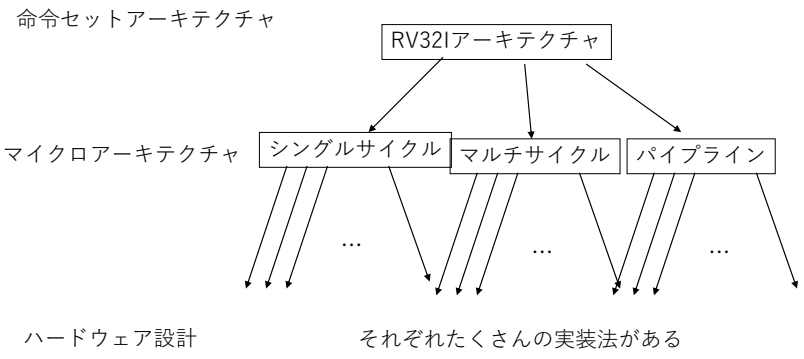
RV32Iのマルチサイクル マイクロアーキテクチャ

慶應義塾大学
天野

マイクロアーキテクチャ



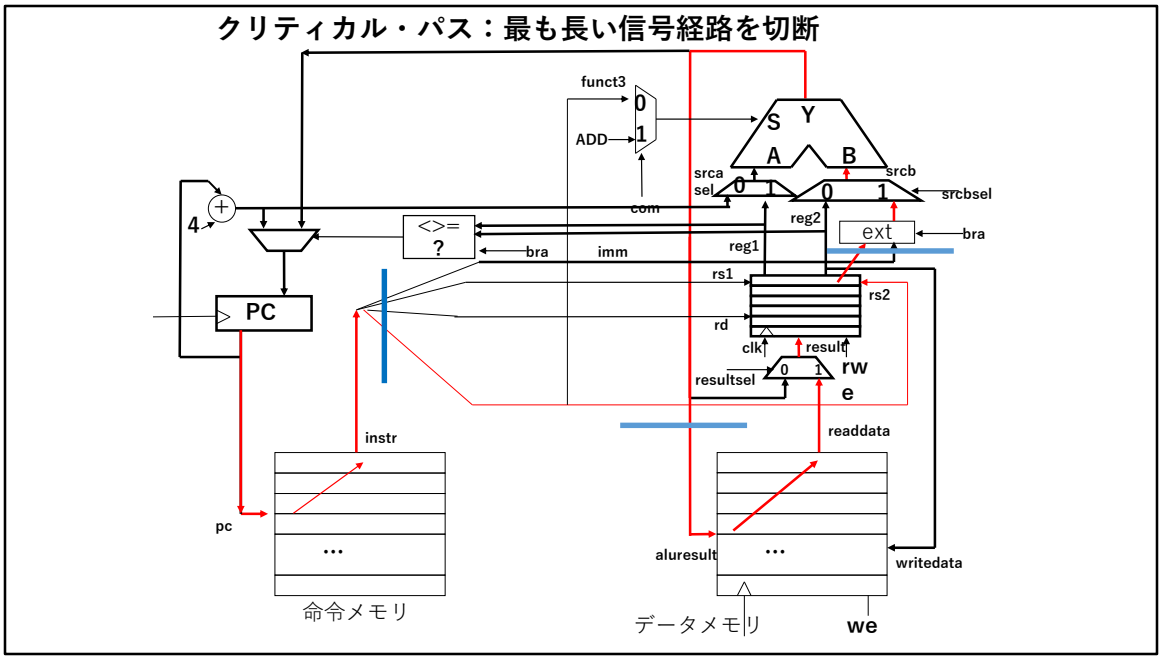
同じ命令セットでも様々な実装法があります。どのようにCPUを実現するかを決めるのがマイクロアーキテクチャです。

シングルサイクル版

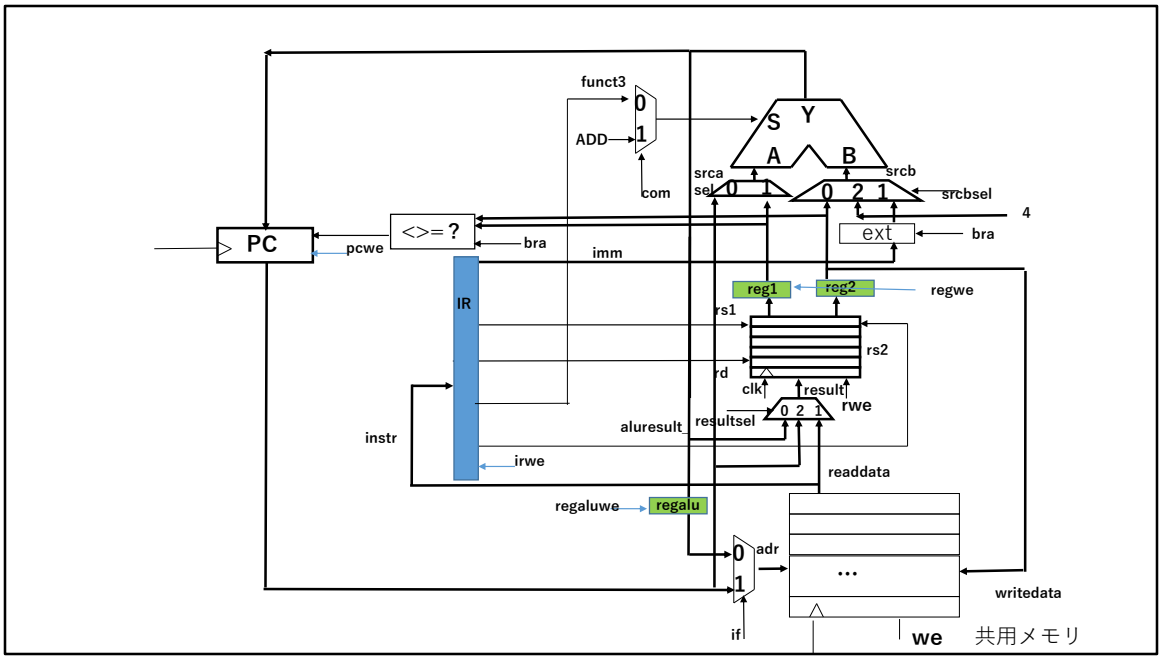
- 今までのRV32Iのシングルサイクル版
- 利点
  - 設計が簡単
  - 案外性能が高く、電力も小さい
- 欠点
  - 資源の共有ができない。特にメモリの分離が必要
  - 最も長いクリティカルパスにクロック周期が制約される

  →マルチサイクル版を作ってみる

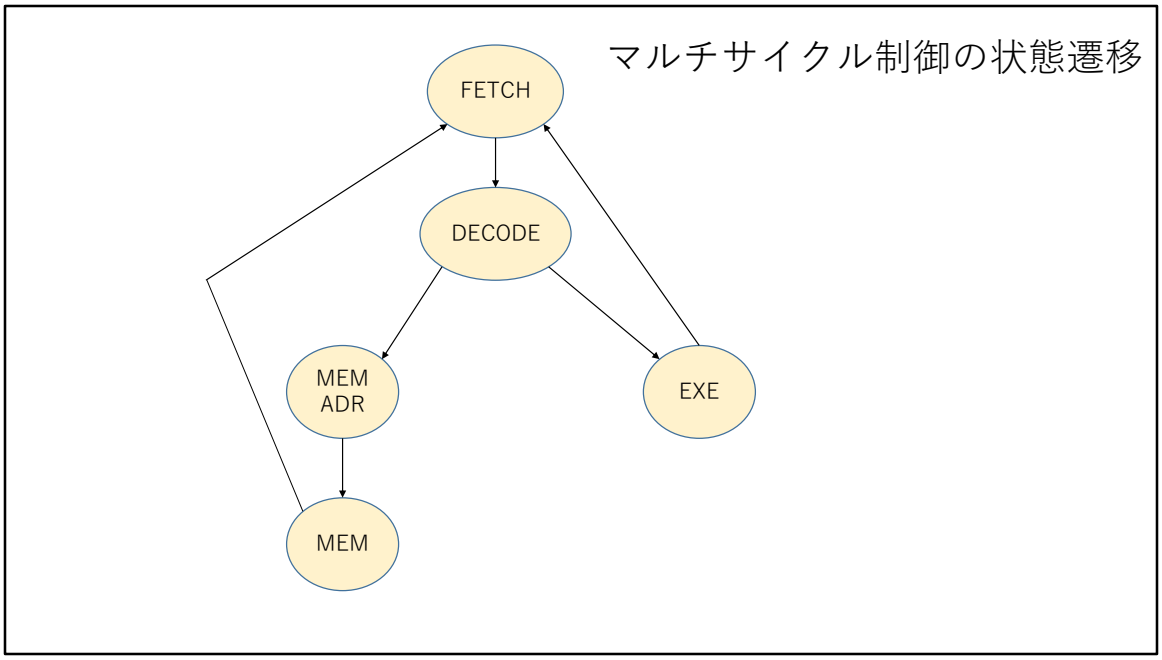
今まで紹介してきたのはシングルサイクル版のRV32Iです。シングルサイクル版は何といっても設計が簡単で理解しやすいです。また、後で評価を取ってみるとわかるのですが、案外性能が高く、消費電力も小さいです。一方で、すべての命令を単一サイクルで実行することから資源の共有ができず、特に命令メモリとデータメモリを分離しなければならない点が問題です。また、一番実行時間の長い命令に合わせたクロックを使わなければならない点では性能的に不利です。この問題はマルチサイクル版を使うことで解決されます。実際のCPUは歴史的にマルチサイクル版を使っていました。IntelのCPUも80486まではマルチサイクル版でした。



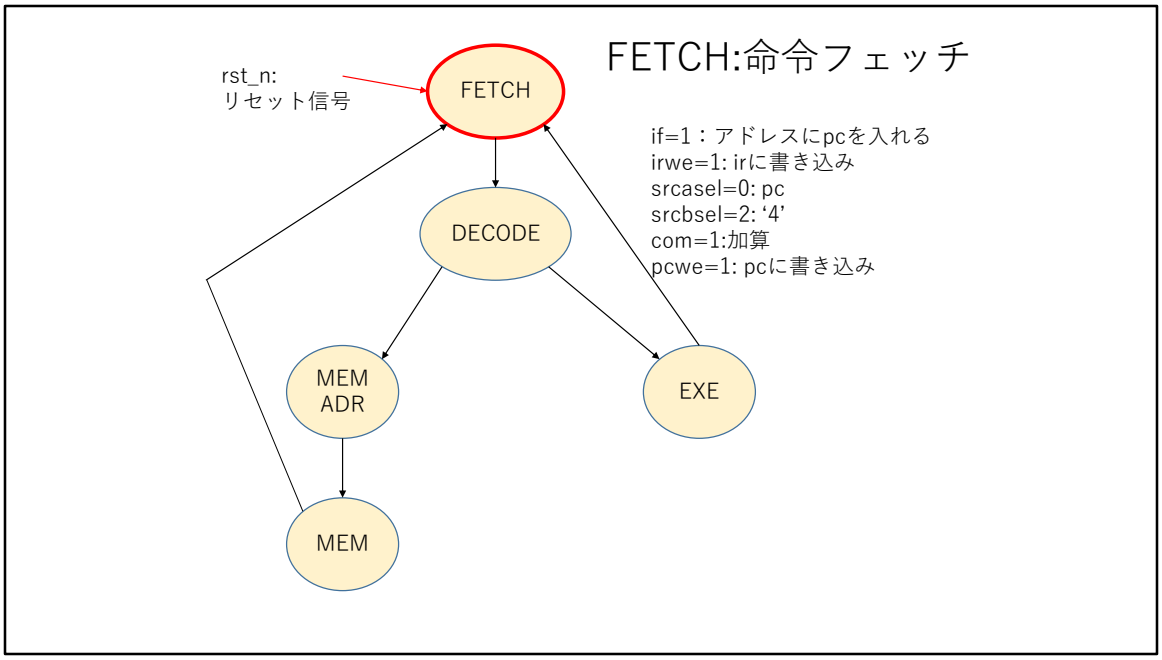
1サイクル版のクリティカルパスをなるべく均等に切る位置にレジスタを入れましょう。まず命令メモリから命令を読んでくる所、命令に従ってレジスタファイルから値を読む所、演算する所、データメモリからデータを読む所です。この場合、lw、sw命令では4クロック、他の命令では3クロック掛かりますが、クリティカルパスは短くなるため周波数は上がるはずです。またレジスタで中間結果を格納することにより、資源を共有することができます。ここでは命令メモリとデータメモリを共用させること、PCに 4 を加算する操作をALUにやらせて、加算器を節約することを考えます。



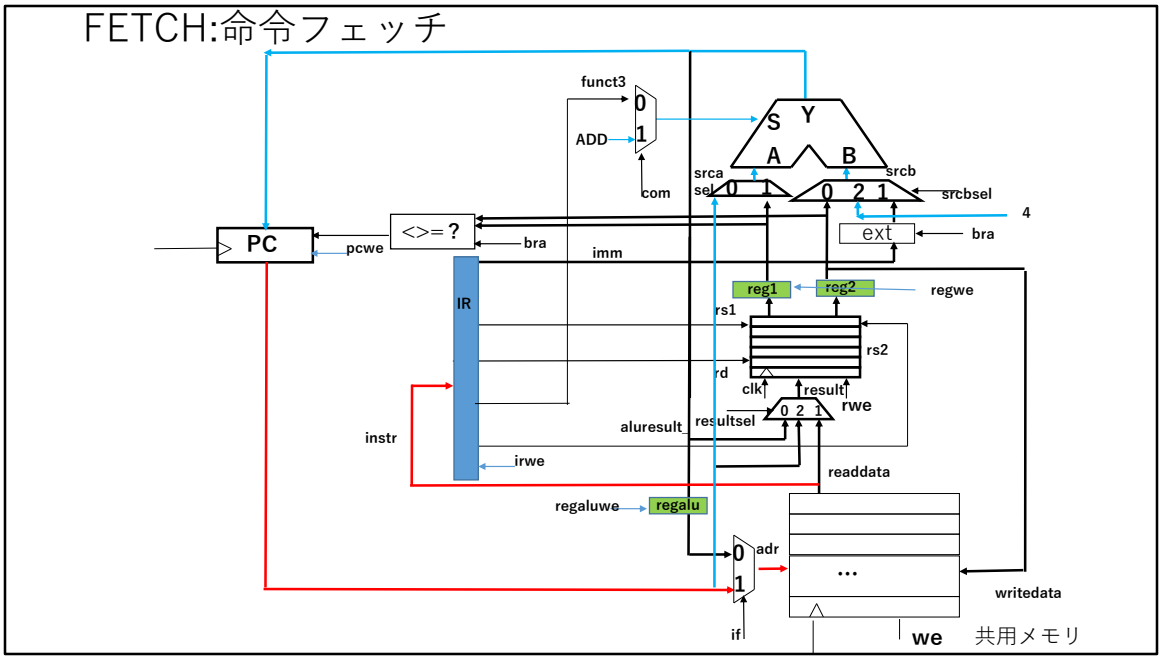
このように考えて作ったマルチサイクルのデータパスを図に示します。命令とデータは共用のメモリに入れておきます。取ってきた命令を蓄えておく命令レジスタ（Instruction Register: ir)を設けます。次にレジスタファイルから読んできたデータを蓄えるreg1, reg2, 計算結果を蓄えるregaluを入れます。メモリの共用のためにアドレスにマルチプレクサを入れます。また、PC＋4を計算するために、ALUのB入力のマルチプレクサを拡張します。



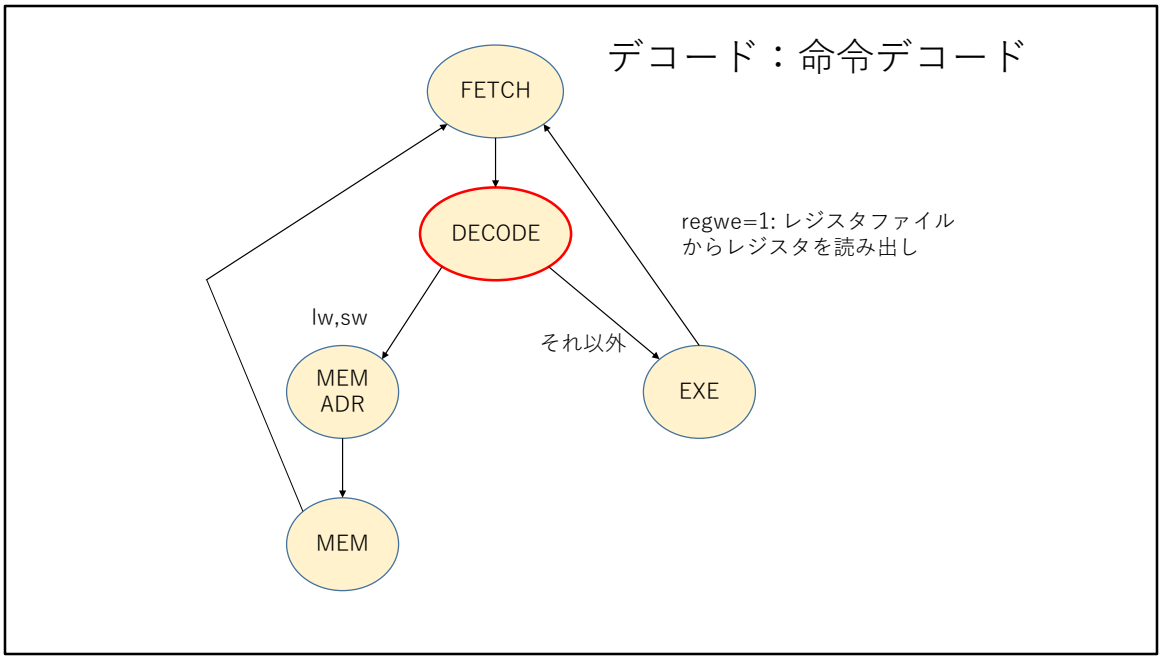
シングルサイクルと違って、マルチサイクルはステートマシン（有限状態マシン：Finite State Machine: FSM)を用いて制御します。このステートマシンによる制御はディジタル回路の制御の基本で、様々な用途に使います。



FETCH状態の役目は①pcが示す命令を読みだしてirにしまう。②pcに4を加えるの２つです。このために、この状態では①if=1としてアドレスをpcに入れ、irwe=1としてirに読んできたデータ（命令）をしまいます。次に②srcasel=0にしてALUのA入力からpcを入れてやり、srcbsel=2にしてB入力から4を与えます。ALUのcom=1として加算を指示します。pcwe=1として計算結果をpcに保存します。

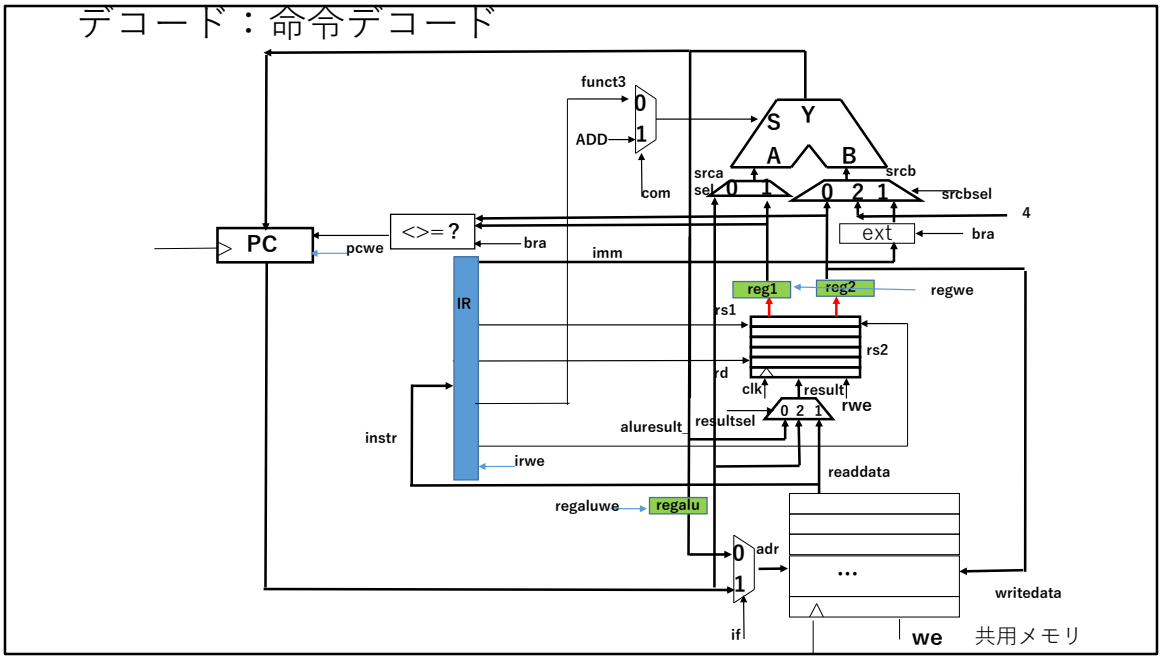


先に示した信号がどのようにデータの流れを制御するかを示します。赤が①命令を読みだしてirにしまう。青が②pcに 4 を加えてpcにしまう操作に対応します。

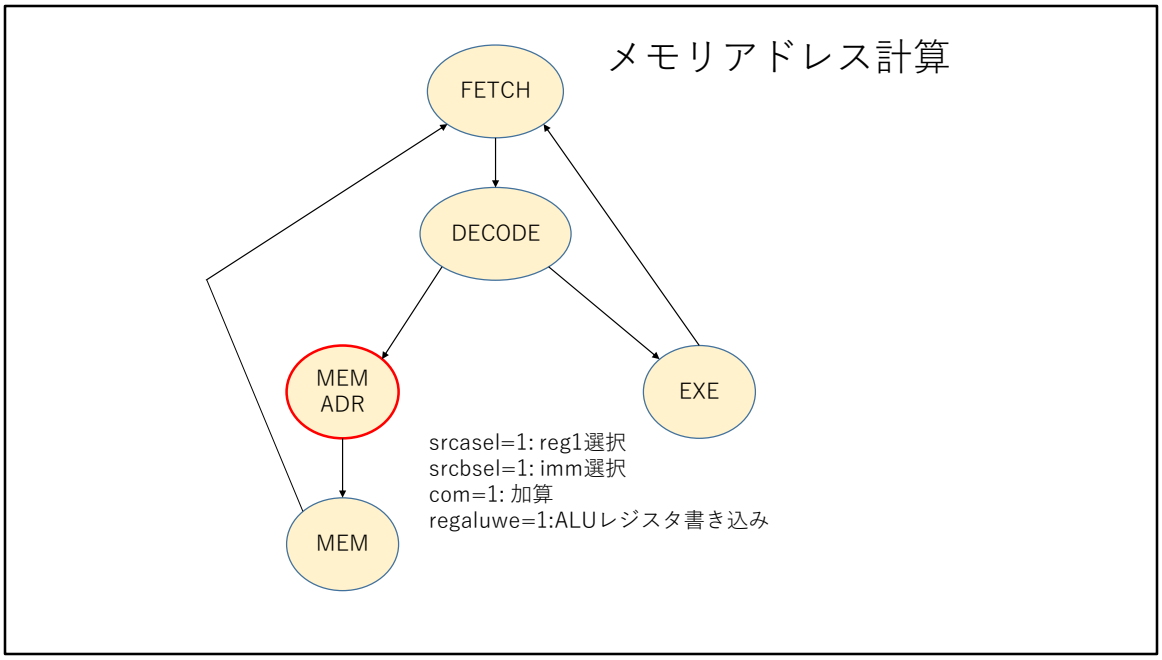


命令はFETCH状態からDECODE状態に移行するクロックの立ち上がりでirに格納されます。この結果、各デコード信号が確定するのがこの状態です。opcodeも確定するため、この値によってメモリアクセス状態とそれ以外に状態遷移します。この状態では、ir中の命令のrs1,rs2に対応するレジスタをレジスタファイルから読み出します。読み出した値をreg1, reg2に保存するため、regwe=1にします。

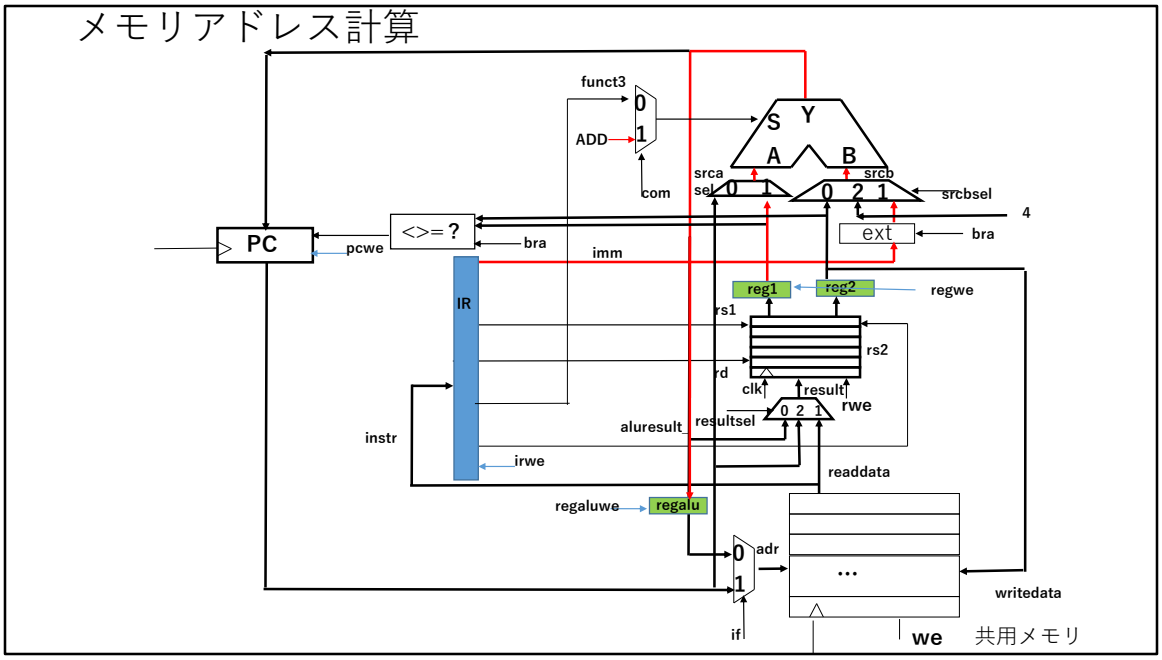
デコード：命令デコード



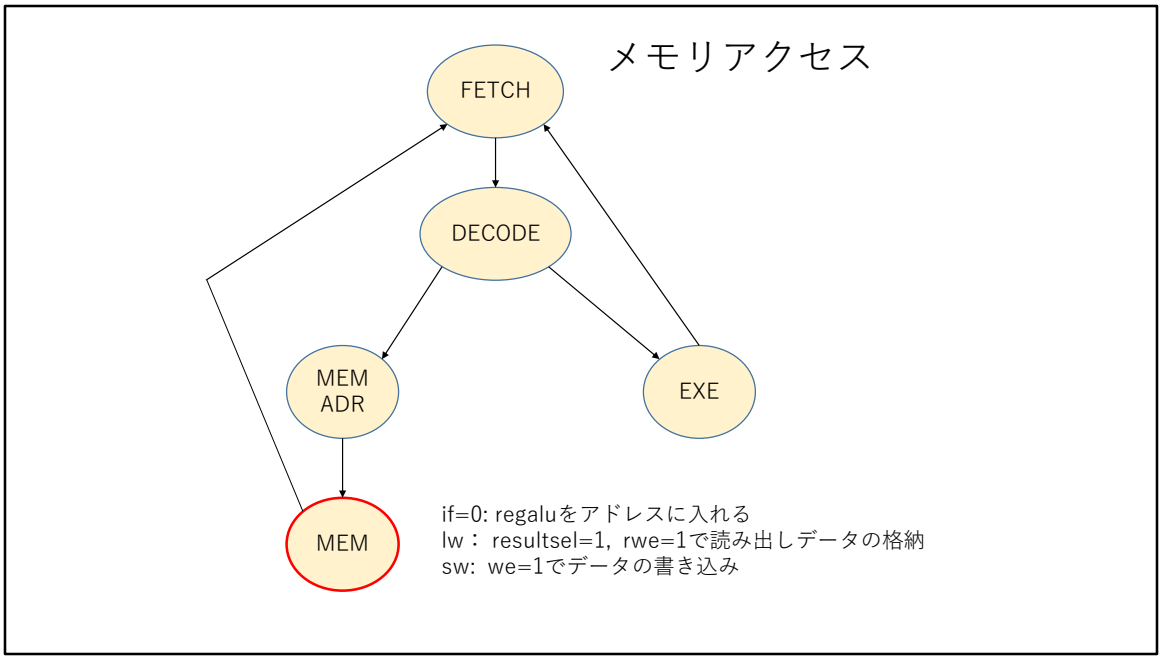
この状態は主として制御回路が働き、データパスからはレジスタファイルの読み出し以外には動きがないです。



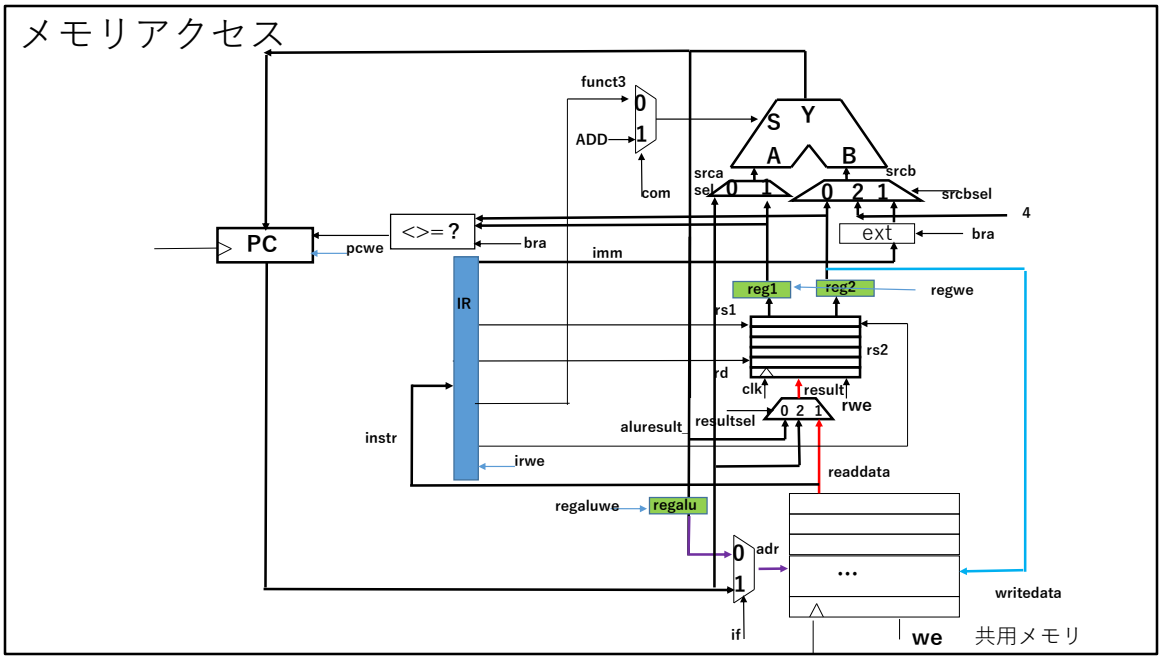
MEMADRは、実効アドレスの計算を行う状態です。このため、srcasel=1としてALUのA入力にはreg1を送ります。srcbsel=1として命令コード中のimm領域を送ります。これはlwとswで位置が違うため、この点をext内部で適切に入れ替える必要があります。com=1としてALUでは加算を行わせ、結果をreglauwe=1としてALU結果用のレジスタに書き込みます。



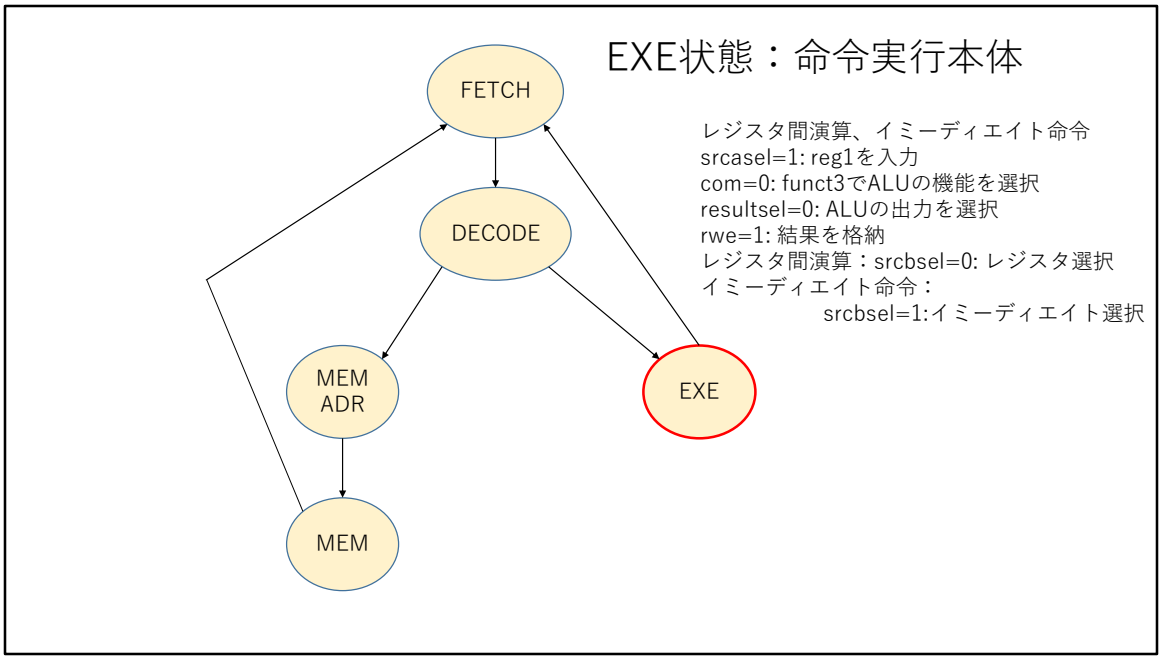
MEMADR状態での各部の動きを示します。計算されたアドレスはregaluに格納されます。



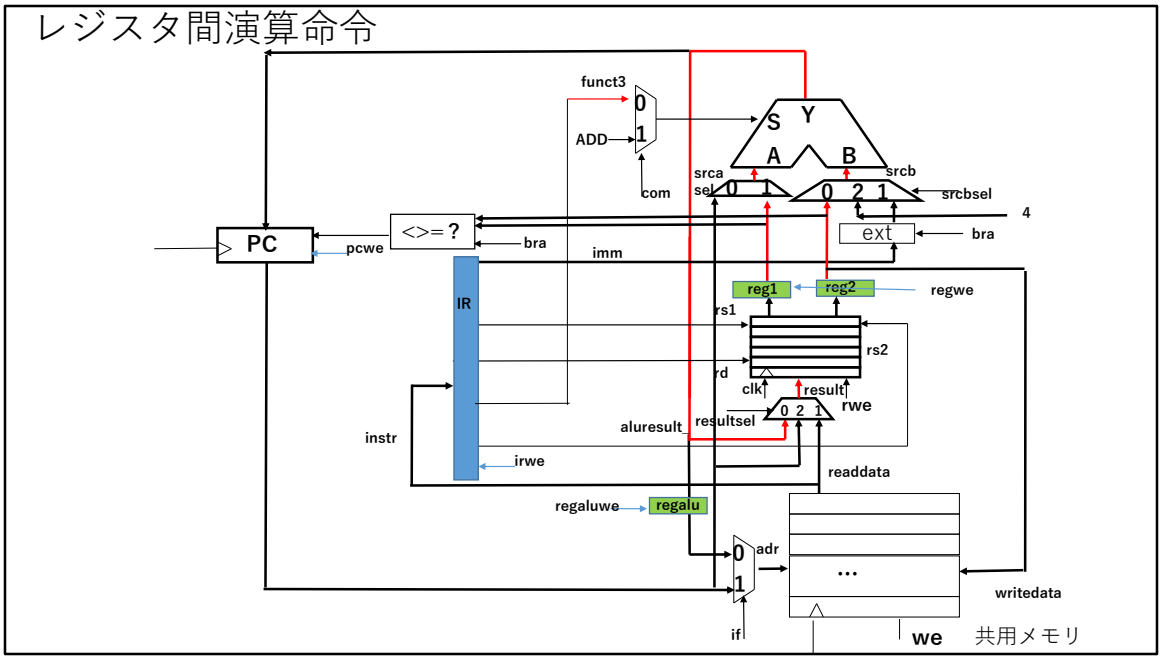
次のMEM状態では、lw命令ならば、resultsel=1,rwe=1として、読んできた値をレジスタファイルに格納します。sw命令の場合、we=1でデータを書き込みます。メモリのデータ入力にはあらかじめreg2が接続されているので、これだけでOKです。両者ともにif=1としてメモリのアドレスには先に計算したaluregの値を入れてやります。



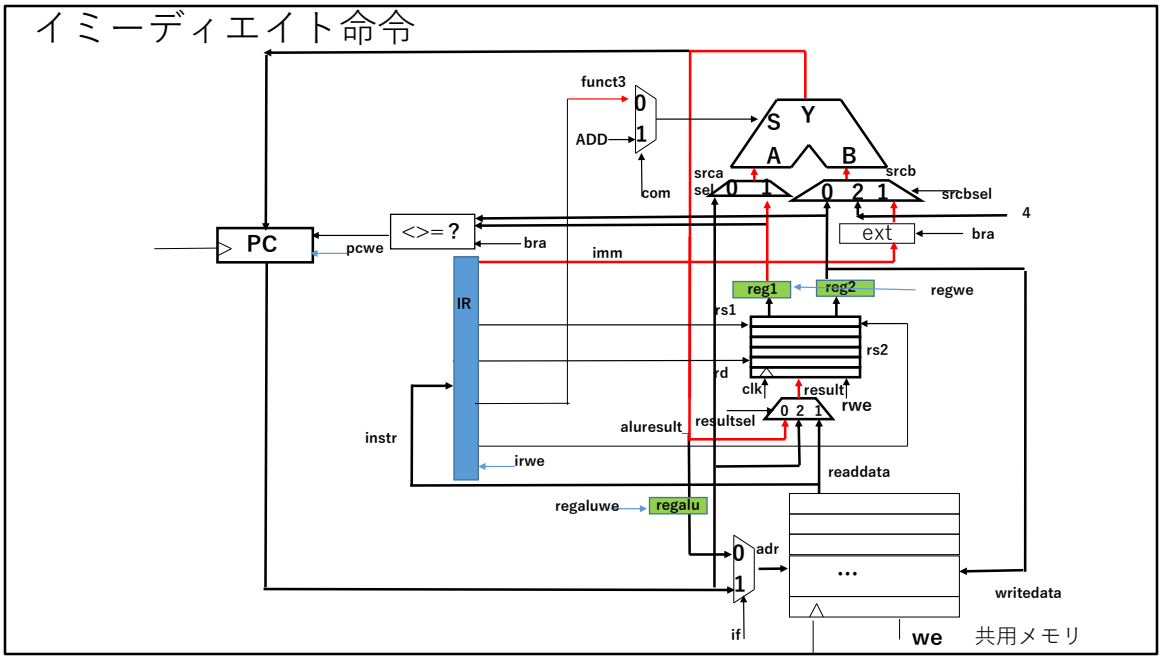
メモリアクセス時のデータの流れを示します。赤はlw, 青はsw,紫は共通です。



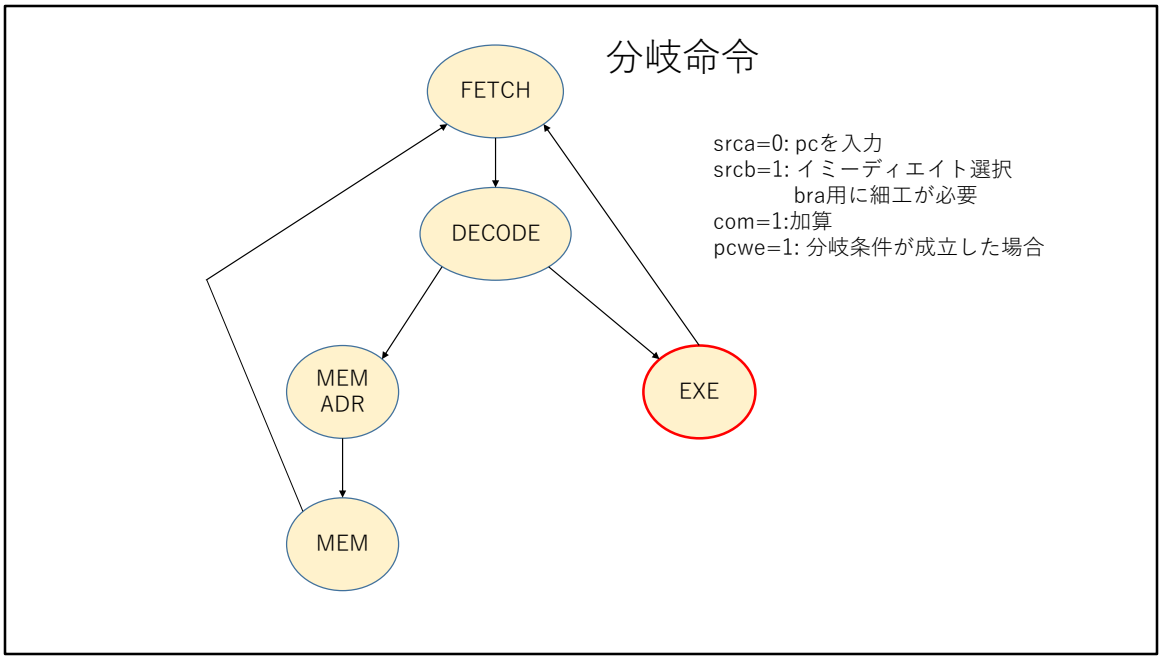
EXE状態は、命令実行状態で、メモリアクセス命令以外の命令を実行します。これは命令の種別に応じて動作が若干違ってきます。イミーディエイト命令とレジスタ間演算命令は、共に、com=0にして命令コードの一部func3をALUに入れてやって演算の種類を選び、演算結果はresultsel=0、rwe=1としてレジスタファイルに格納します。ALUのB入力についてのみ両者は異なっており、レジスタ間演算命令はsrcbsel=0としてレジスタを選択し、イミーディエイト命令はsrcbsel=1としてイミーディエイトを選択します。



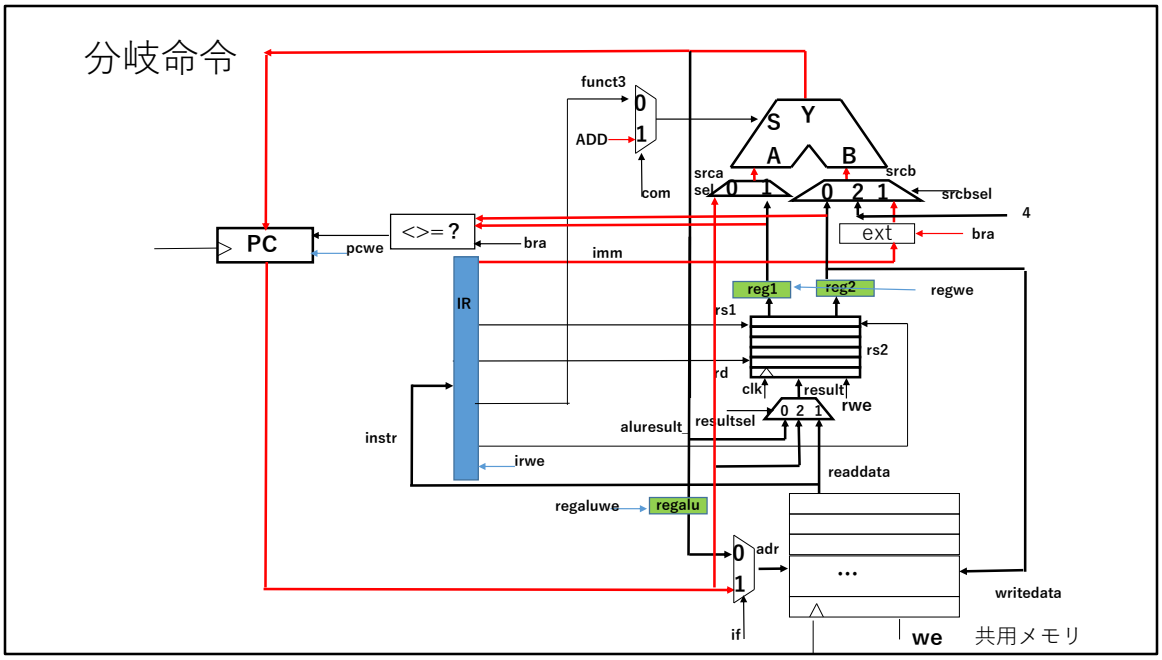
レジスタ間演算命令のデータの動きを示します。ALUのB入力はreg2が入ります。



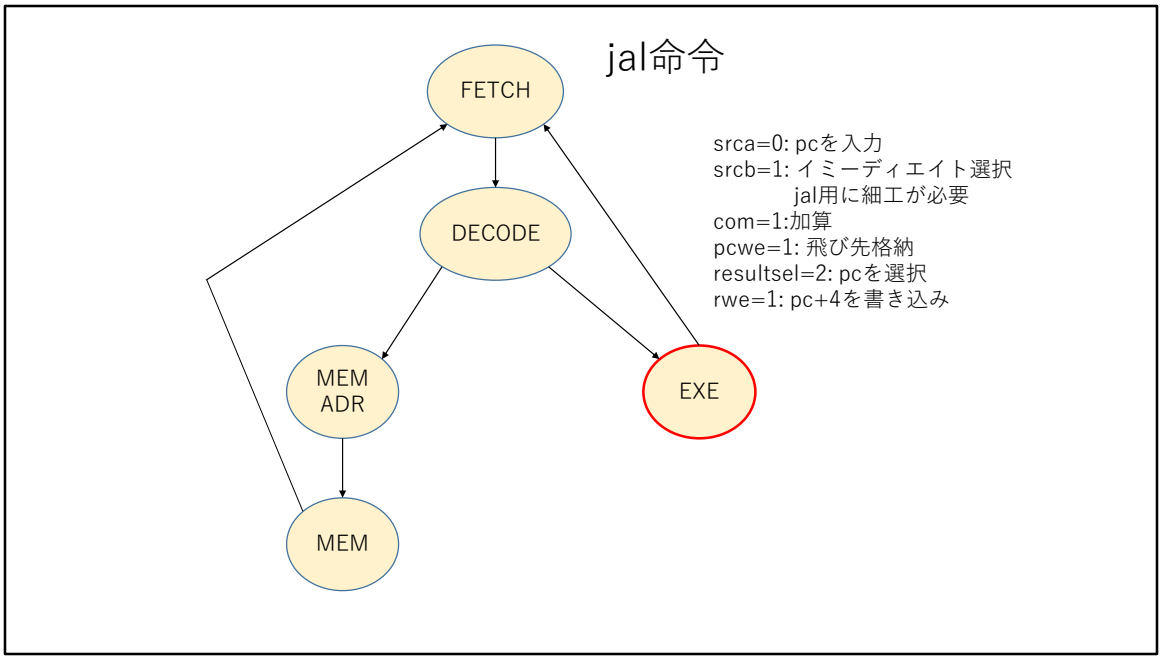
イミーディエイト命令は、ALU B入力に命令コードのimmを符号拡張して入れてやります。この点以外はレジスタ間演算命令と同じです。



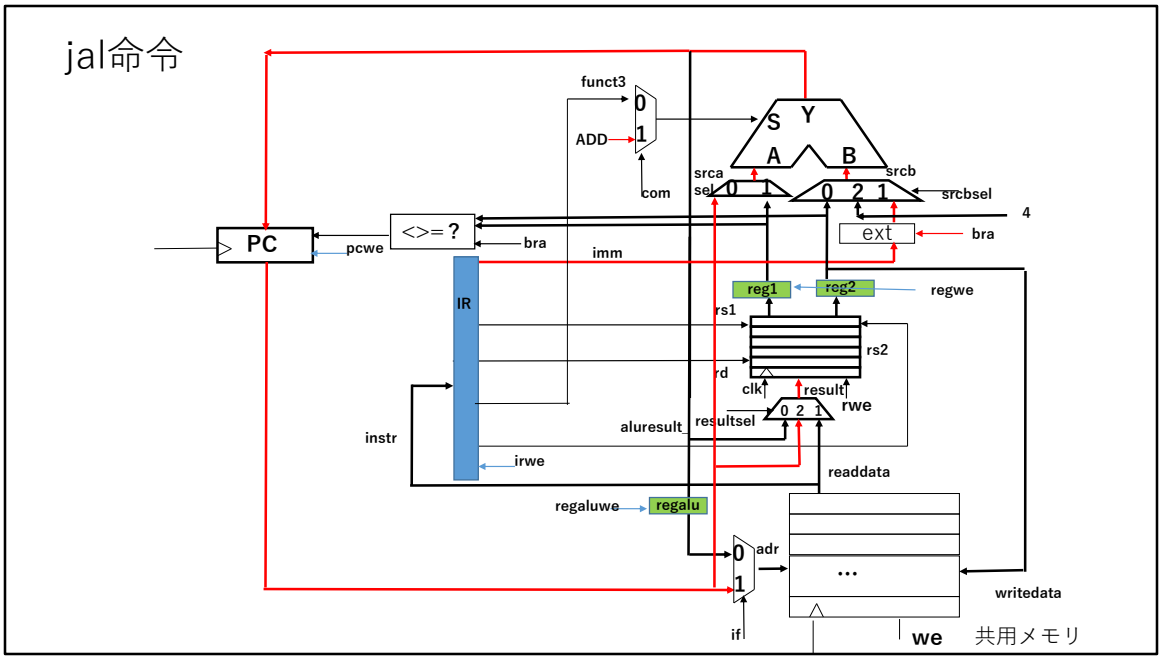
同じEXE状態で、分岐命令を実行する場合は、srca=0としてALU A入力にはpcを入力し、ALU B入力にはsrcb=1としてイミーディエイトを選択します。ただし、このイミーディエイトは、イミーディエイト命令やアドレス計算時に使ったものと並びが違うのでextで並び替えを行う必要があります。com=1としてALUには加算を指示します。同時にreg1, reg2の値の比較結果と分岐命令の種類から分岐が成立するかどうかを判別し、成立すればpcwe=1とします。



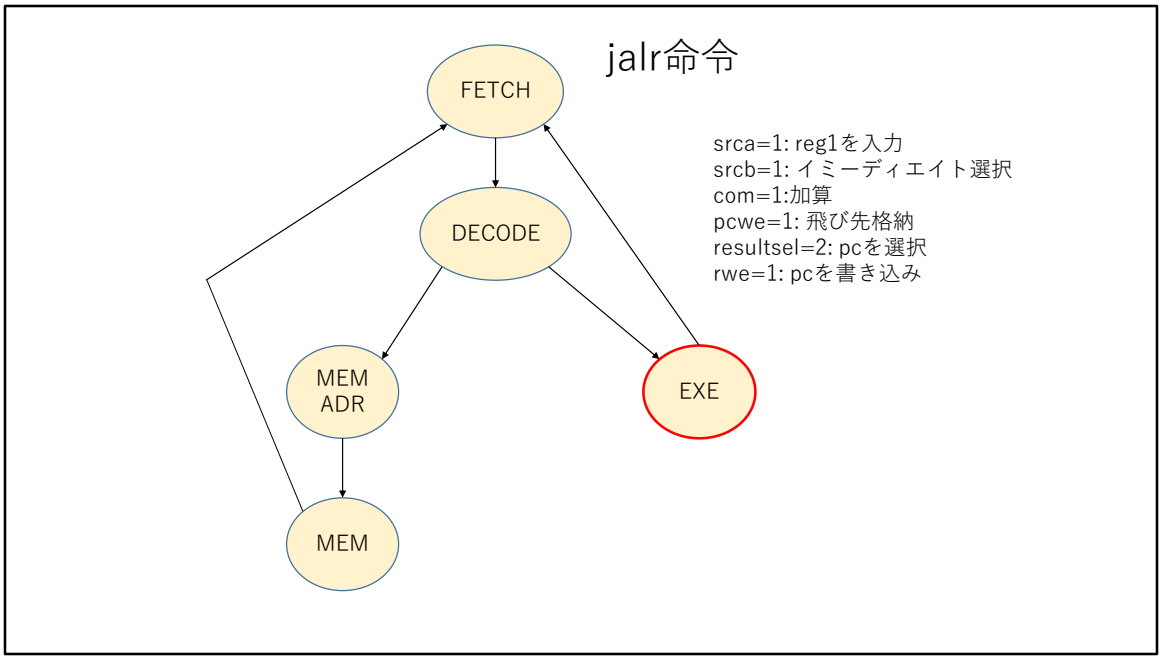
分岐命令のデータパスの流れを示します。pcの値はFETCH状態で既に4が加算されていることにご注意ください。このため飛び先は自然にpc+4が起点となります。



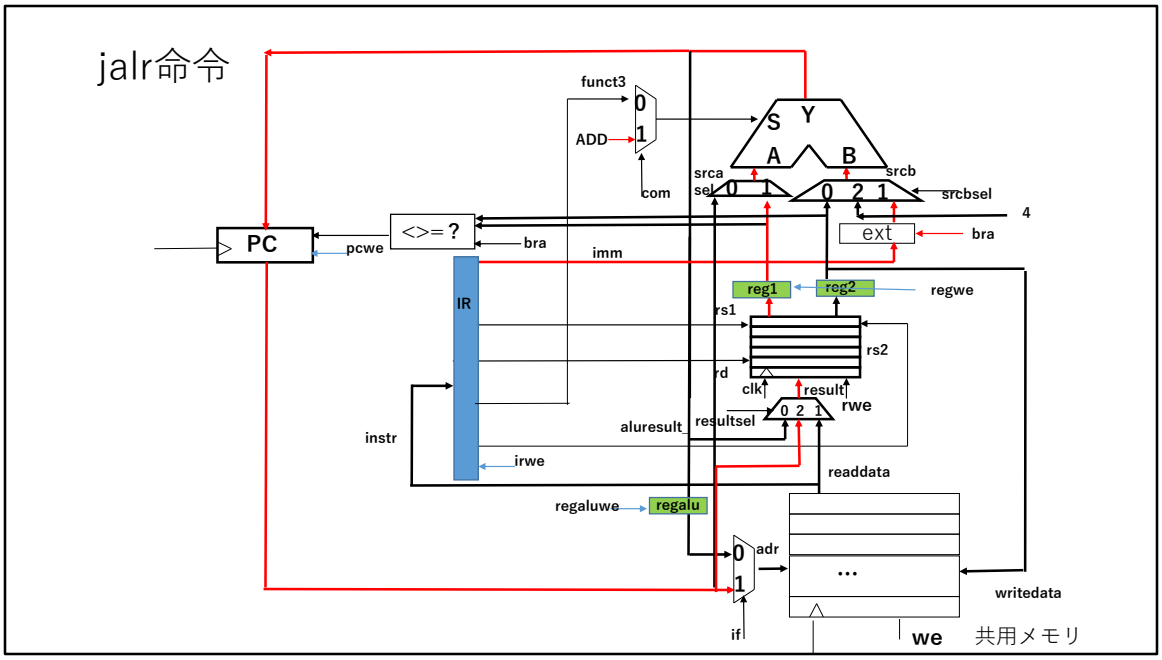
jal命令は、①飛び先を計算すると共に、②pcをレジスタファイルに保存する操作を行います。①飛び先の計算はsrca=0でA入力にpcを入れ、srcb=1としてイミーデエイトをB入力に入れてやります。ここで、イミーディエイトは、jalは他と違って20ビットあってビット入れ替えもあるので、extでこれを行います。com=1で加算してやります。分岐命令と違って無条件にpcwe=1として飛び先をpcに格納します。②pcをしまう操作は、resultselを 2 にしてrwe=1にして行います。なお、pcはFETCH状態で既に4加算されているので、戻ってくる時に同じ命令に戻る心配はないです。



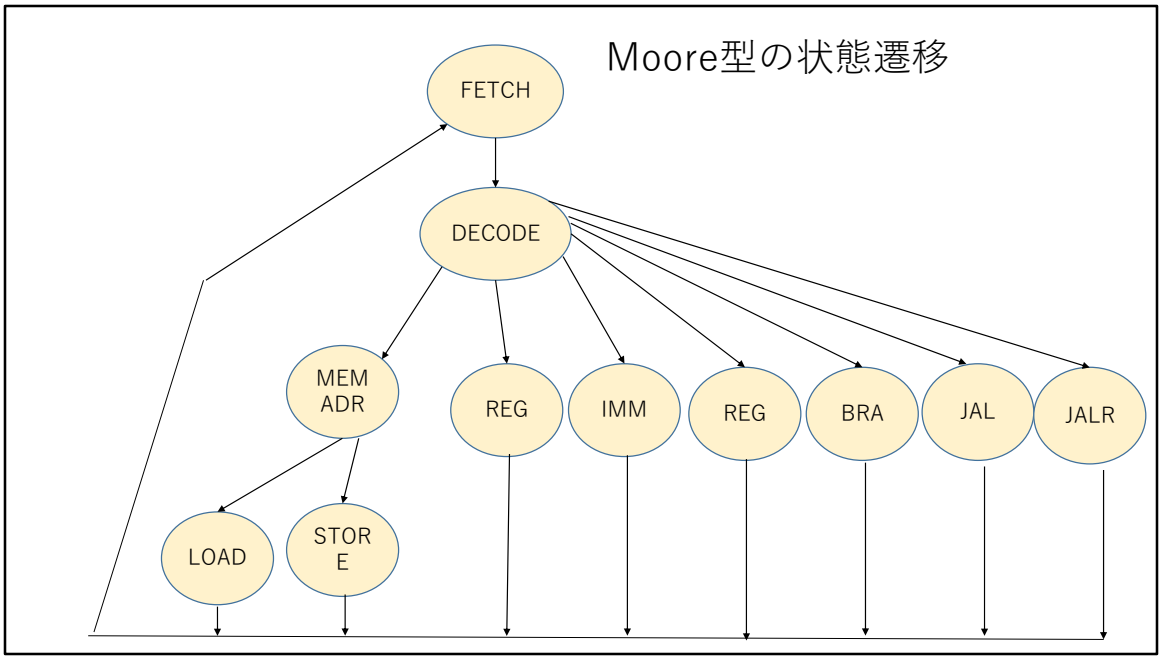
jal命令のデータの流れです。



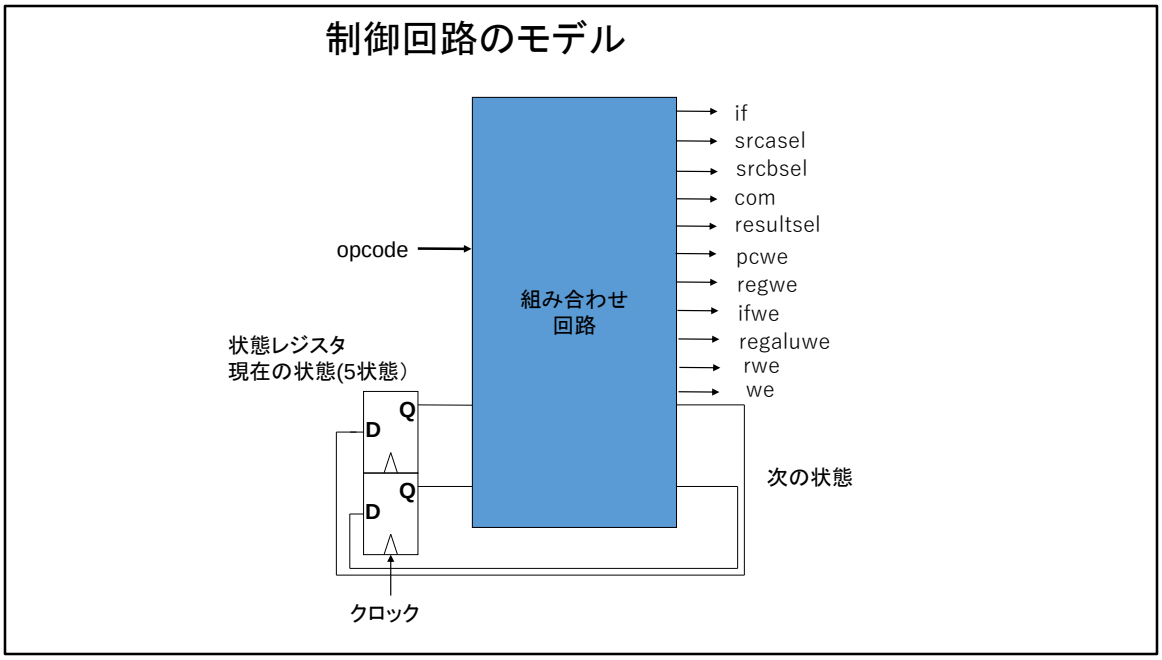
jalr命令は、①飛び先を計算する。②pcを保存する。２つの操作を行う点でjalと同じですが、飛び先はpcではなくてreg1とイミーディエイトの和なので、ALUのsrca=1, srcb=1とします。com=1として加算操作を行い、結果をpcwe=1としてpcに格納します。②はjal同様、resultsel=2、rwe=1としてpcをレジスタファイルに格納します。



jalr命令のデータパスの動きを示します。jalとの違いに注意してください。



今回利用した状態遷移は、状態内で、命令の種類に応じて出力信号を切り替えました。これはFSMとしては、状態と入力信号で出力を制御するMealy型になります。Mealy型を使うと状態数は減りますが、状態内で命令を判別する必要があるので、動作速度が落ちます。これを防ぐには、状態＝命令として出力信号が自動的に決まってしまうMoore型にすれば良いです。このようにすると動作速度は上がりますが、状態が増えて設計が大変になるので、ここでは簡単に書けるMearly型を採用しました。



Mearly型FSMによる制御回路のモデルを図に示します。組み合わせ回路は現在の状態とir中の命令のopcodeから、今までに定義した制御信号を発生します。この組み合わせ回路は大きなテーブルになりますが、Verilogなどのハードウェア記述言語の記述から自動的に生成されます。

状態遷移のVerilog記述

- One hot counterを用いる
  - 状態に対応するビットを設ける
  ○設計が簡単、状態遷移が2ビット変化で済む、状態の判別が高速
  ×必要フリップフロップ数が多い→しかし最近は気にならない
- ここでは5状態＝5ビット
  - FETCH:　5'b0_0001
  - DECODE: 5'b0_0010
  - MEMADR:5'b0_0100
  …
- レジスタstatで状態を保持する
  - reg [4:0] stat;

さて、ここで状態遷移をVerilog記述でどのように書くかを紹介します。状態の表現方法には色々あります。皆さんが計算機基礎でならったのは状態に普通の2進数を割り当てる方法でした。しかしここではHDL記述では一般的に用いられているOne hot counterを使います。この方法は状態一つにつき1ビットを割り当てる方法です。ここでは12状態に対して12ビットを割り当てます。FETCH状態は最下位ビットを割り当て、DECODE状態は下から2ビット目を割り当て、、、と順番に割り当てて行きます。
この方式は、全ての状態において必ずどこかの1bitのみが1となります。このため、設計が簡単で、状態遷移は2ビットのみで済みます。さらに状態の判別が簡単で済むという利点があります。欠点は、状態のビット数が増えるので、フリップフロップの数が増えてしまうことですが、最近のLSIは十分な面積をもっており、この程度は全く気にしなくても良いです。ここでは5状態あるので5ビットを用意し、レジスタstateに保持することにします（stateはVerilogの予約語で使えません）。

状態遷移のVerilogでの記述

```
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) stat <= `FETCH;
    else
        case(stat)
          `FETCH: stat <= `DECODE;
          `DECODE: if(lw_op|st_op) stat <= `MEMADR;
                          else stat<=`EXE;
          `MEMADR:  stat <= `MEM;
          `MEM: stat <= `FETCH;
          `EXE: stat<= `FETCH;
        end
```

case文とif elseを使って状態遷移図をそのまま記述

では、状態遷移をどのようにVerilogで書くかを紹介します。いつものalways文を使って、リセット時にはFETCH状態から始めるようにします。後は、case文を使って各状態の遷移を記述します。状態の分岐がある場合は、if文を使います。この方法で非常にスムーズに直接状態遷移が記述できます。

case文

• if else同様、always文中のみで利用可能

```
case (信号線)
   値１：文１
   値２：文２
   …
   default: 文
endcase
```

信号線と一致した値の文が有効になる（上の方が優先順位が高い）
defaultはなくてもいい
文中はレジスタ・メモリへの値の設定（<=)のみ

ここで久しぶりにVerilogの新文法がでてきます。このcase文は他の言語同様、一致した値に相当する文が有効になります。?:の条件選択文と違ってレジスタが対象で、always文中のみで使えます。if else if..で代替可能ですが、見やすいので特に状態遷移の記述で使われます。条件選択文と違ってdefaultはなくても大丈夫です。endcase後は;は付けてはいけません。

状態の判別

- 各状態の0の位置を__Bで定義する
  - FETCH: 5'b0_0001　→　FETCH_B: 3'b000
  - DECODE: 5'b0_0010　→DECODE_B: 3'b001
  - MEMADR: 5'b0_0100 →　MEMADR_B: 3'b010
- statのビット位置を調べれば状態が分かる
  - stat[`FETCH_B]が1ならばFETCH状態
  - stat[`DECODE_B]が 1 ならばDECODE状態
  - stat[`MEMADR_B]が 1 ならばMEMADR状態
- 様々な記述でこの点を利用する

  - 例）命令レジスタ(ir)の記述

```
reg [`DATA_W-1:0] instr;
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) ir<=0;
    else if (stat[`FETCH_B]) ir <= readdata;
end
```

今回はMearly型なので、状態と入力で出力（あるいはデータパスでやること）が決まります。記述をするには、この状態マシンの現在の状態が何なのかを知る必要があります。One Hot Counterはこれが簡単にできます。今、それぞれの状態に対して状態__Bに対してそのビットの位置を定義します。例えばFETCHに対してはFETCH_B＝０、DECODE__B＝１、MEMADR__B＝２になります。このビット位置をstatの配列の中に入れてやれば、そのビットを切り出すことができます。One Hot Counterは、対応するビットが1ならば、状態マシンがその状態になっているので、判別が簡単にできます。例えばstat[`FETCH_B]が1ならばFETCH状態、stat[`DECODE_B]が1ならばDECODE状態になっていることが分かります。これを利用して、それぞれの動作を書きます。例えば、FETCH状態の時に命令レジスタにフェッチしてきた命令を蓄えるという記述を示します。if(stat[`FETCH_B])が成立すれば、状態がFETCH状態になっていることがわかるので、この時に呼んできた命令をirに蓄えます。

ではここで動かしてみよう

- マルチサイクル版の掛け算プログラムmult.asm

```
        lw x3,x0,0x400
        lw x4,x0,0x404
        add x5,x0,x0
loop:   add x5,x5,x3
        addi x4,x4,-1
        bne x4,x0,loop
        sw x5,x0,0x408
        sw x5,x0,0x0
end: beq x0,x0,end
```

データメモリを0 x 400から置いた

0 x 0番地に答を書いたら終了（これは
シミュレーション上のお話し
命令領域に書いてはいけない）

- make: マルチサイクル版を作る
- make mult: mult.asmをアセンブルしてimem.datを作る
- 実行は./test（vpp test)で行う
- 0番地に値を書き込むとClock CountとCount(命令数）が出力される
  - 一命令あたりの平均クロック数Clock cycles Per Instruction (CPI)はいくつだろう？

では、ここで、マルチサイクル版のRV32Iを動かしてみましょう。ここでは今まで何度か出て来た掛け算のプログラムを実行します。ややファイルも増えて複雑になるので、Makefileを用意しておきましたので使ってください。今までと違って命令の実行に複数サイクル掛かることがわかります。状態遷移を観察してください。ここでは実行が終わると自動的に表示が停止して、実行に掛かったクロック数と実行した命令数を表示するようになっています。一命令当たり掛かったクロック数をＣＰＩ（Clock Cycles Per Instruction)と呼びます。ＣＰＩはいくつになるか計算してみてください。

マルチサイクル版のVerillog記述

```
module rv32i(
input clk, rst_n,
input [`DATA_W-1:0] readdata,
output [`DATA_W-1:0] adr,
output [`DATA_W-1:0] writedata,
output we);
```

ではマルチサイクル版のVerilog記述を紹介しましょう。clk, rst_nは今まで通りですが、1サイクル版と違ってメモリが一種類しかないので、インタフェースはむしろ簡単になっています。readdataはメモリからの入力、adrはメモリに対するアドレス、writedataはメモリへの書き込みデータです。これはレジスタファイルの出力レジスタreg2を直接接続します。weはメモリの書き込み信号でこれを 1 にするとメモリへの書き込みが行われます。

```
reg [4:0] stat;
reg [`DATA_W-1:0] reg1, reg2, regalu, ir, pc;
wire addcom;
wire [2:0] funct3;
wire [6:0] funct7;
wire [`REG_W-1:0] rs1, rs2, rd ;
wire [`DATA_W-1:0] srca, srcb, result, a, b, aluresult;
wire [`OPCODE_W-1:0] opcode;
wire [11:0] imm_i, imm_s;
wire [12:0] imm_b;
wire [20:0] imm_j;
wire rwe;
wire alu_op, imm_op, bra_op;
wire sw_op, beq_op, bne_op, blt_op, bge_op, bltu_op, bgeu_op, lw_op, jal_op;
wire slt_op;
wire ext;
wire [19:0] sext;
wire exec;
```

状態はstatに記憶

新しいレジスタとpc

信号線名はシングルサイクル版とほぼ同じ、図と対応のこと

デコード信号名の定義はシングルサイクル版と同じ

マルチサイクル記述では、レジスタはプログラムカウンタpc,命令レジスタir,レジスタファイルの値を一時記憶するreg1,reg2、ALUの出力を一時記憶するregaluを定義します。これは図と同じ名前ですので対応を見てください。また、それぞれの信号線に名前を付けています。これも図とVerilog記述を一致しておきましたので、対応してください。

```
assign writedata = reg2;

assign sreg1 = $signed(reg1);
assign sreg2 = $signed(reg2);
assign {funct7, rs2, rs1, funct3, rd, opcode} = ir;
assign sext = {20{ir[31]}};
assign imm_i = {funct7,rs2};
assign imm_s = {funct7,rd};
assign imm_b = {funct7[6],rd[0],funct7[5:0],rd[4:1],1'b0};
assign imm_j = {ir[31], ir[19:12],ir[20],ir[30:21],1'b0};
```

比較命令用

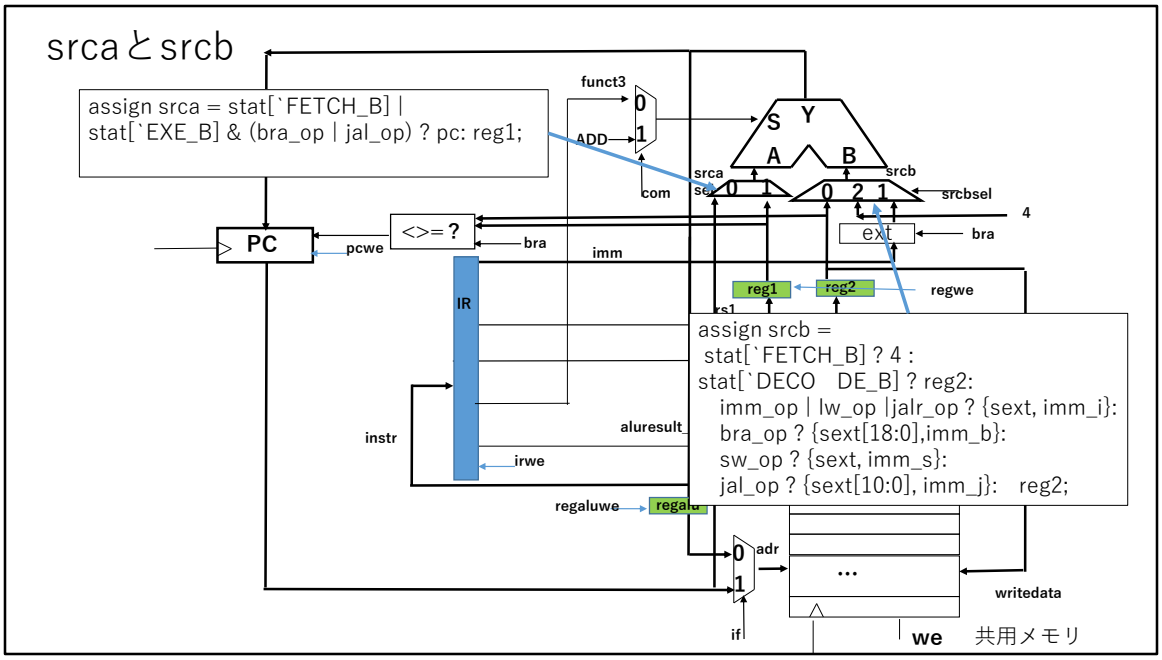
命令レジスタの値をデコード

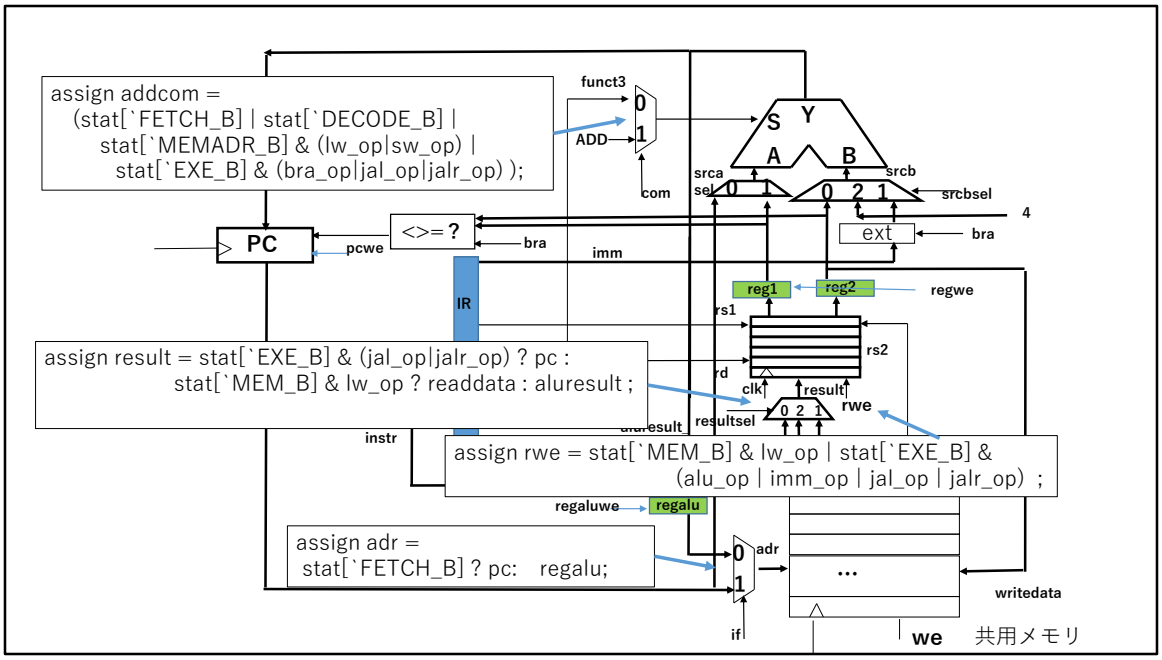
命令のデコードの部分です。これは今までとほとんど同じでしたが、命令は命令レジスタirに入っているので、そこからopcode,レジスタ、func3,funct7,immを切り出します。これもシングルサイクル版と同じです。

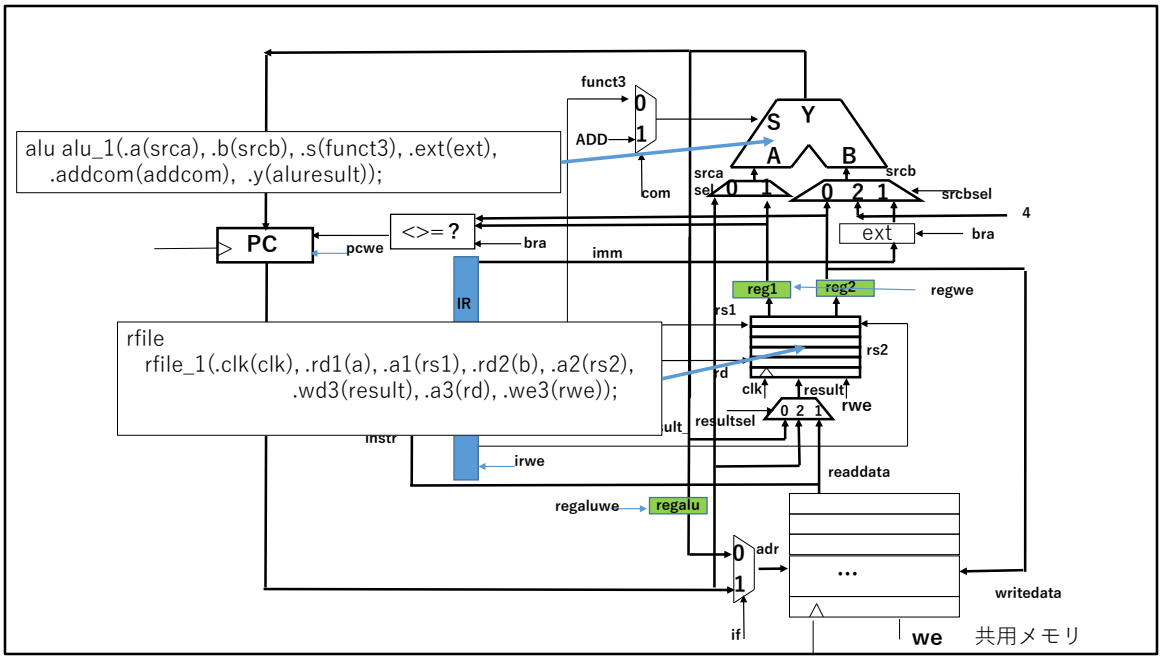
```
// Decorder
assign sw_op = (opcode == `OP_STORE) & (funct3 == 3'b010);
assign lw_op = (opcode == `OP_LOAD) & (funct3 == 3'b010);
assign alu_op = (opcode == `OP_REG) ;
assign imm_op = (opcode == `OP_IMM) ;
assign bra_op = (opcode == `OP_BRA) ;
assign jal_op = (opcode == `OP_JAL);
assign jalr_op = (opcode == `OP_JALR)& (funct3== 3'b000);
assign beq_op = bra_op & (funct3 == 3'b000);
assign bne_op = bra_op & (funct3 == 3'b001);
assign blt_op = bra_op & (funct3 == 3'b100);
assign bge_op = bra_op & (funct3 == 3'b101);
assign bltu_op = bra_op & (funct3 == 3'b110);
assign bgeu_op = bra_op & (funct3 == 3'b111);

assign ext = stat[`EXE_B] & alu_op  & funct7[5];
assign we = stat[`MEM_B] & sw_op;
```

各種命令デコード信号

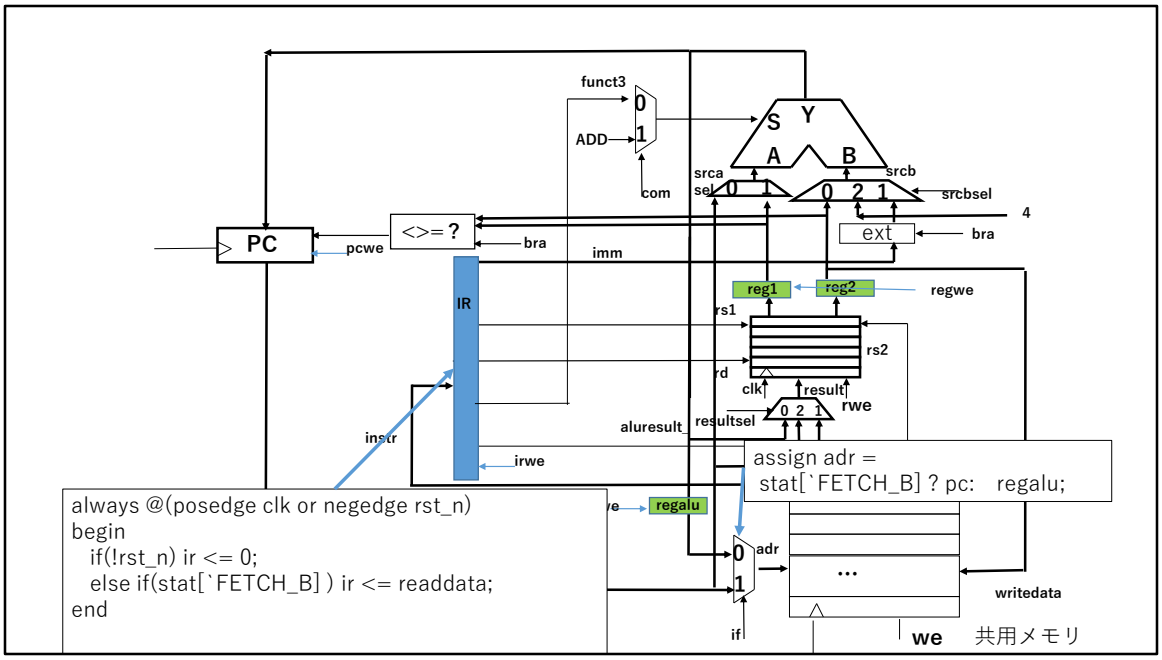
各種デコード信号もシングルサイクルと同じです。ここで注意点としては、これらの信号はir中の命令をデコードしたため、そこに命令が留まる限り有効になってしまいます。実際にこれを利用する場合、原則として状態を示す信号とANDする必要があります。ext, weがこの例です。


srcaとsrcb
assign srca = stat[`FETCH_B] |
stat[`EXE_B] & (bra_op | jal_op) ? pc: reg1;
funct3
ADD
com
S
Y
A
B
srca
srcb
srcbsel
4
ext
bra
PC
pcwe
<>= ?
bra
imm
IR
reg1
reg2
regwe
rs1
assign srcb =
stat[`FETCH_B] ? 4 :
stat[`DECO DE_B] ? reg2:
imm_op | lw_op |jalr_op ? {sext, imm_i}:
bra_op ? {sext[18:0],imm_b}:
sw_op ? {sext, imm_s}:
jal_op ? {sext[10:0], imm_j}: reg2;
instr
aluresult
irwe
regaluwe
adr
writedata
if
we
共用メモリ

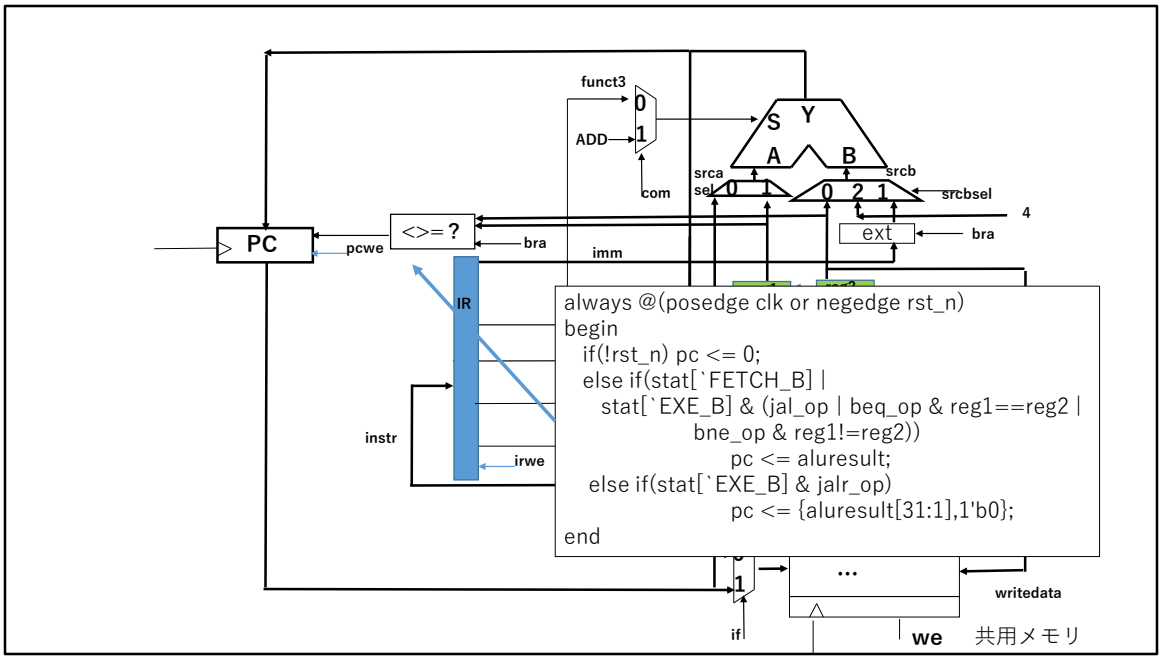

assign addcom =
(stat[`FETCH_B] | stat[`DECODE_B] |
stat[`MEMADR_B] & (lw_op|sw_op) |
stat[`EXE_B] & (bra_op|jal_op|jalr_op));
funct3
ADD
com
S
Y
A
B
srca
sel
srcb
srcbsel
4
ext
bra
imm
PC
pcwe
<>= ?
bra
IR
reg1
reg2
regwe
rs1
rs2
rd
clk
result
rwe
resultsel
assign result = stat[`EXE_B] & (jal_op|jalr_op) ? pc :
stat[`MEM_B] & lw_op ? readdata : aluresult ;
instr
assign rwe = stat[`MEM_B] & lw_op | stat[`EXE_B] &
(alu_op | imm_op | jal_op | jalr_op) ;
regaluwe
regalu
assign adr =
stat[`FETCH_B] ? pc: regalu;
adr
writedata
if
we
共用メモリ



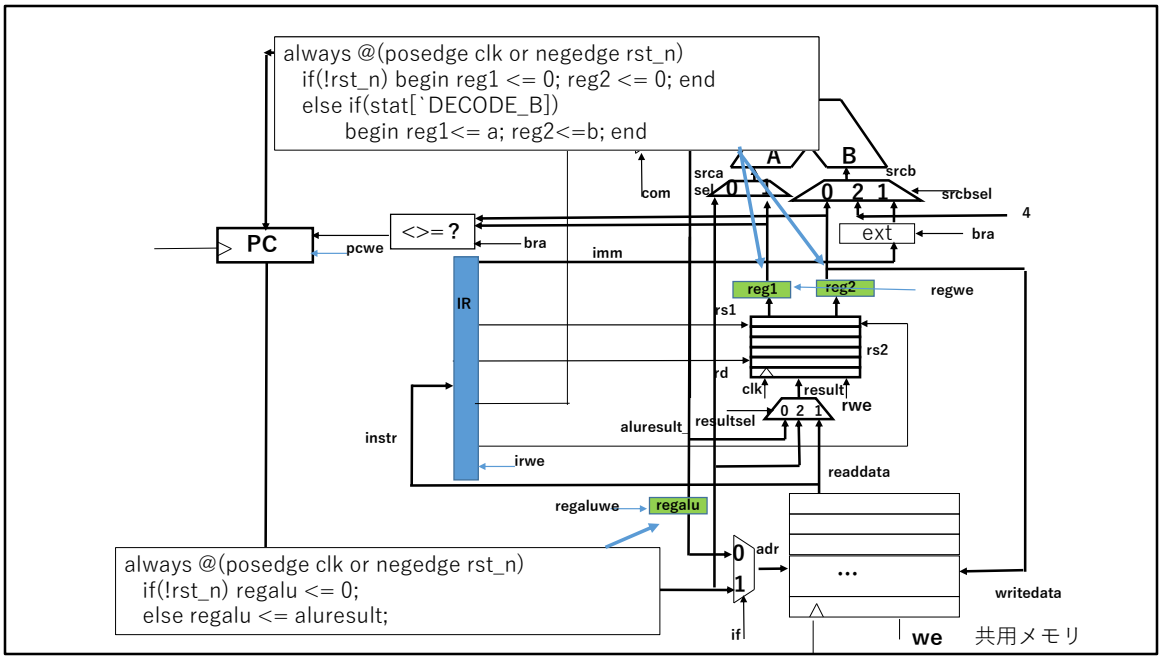
このように考えて作ったマルチサイクルのデータパスを図に示します。命令とデータは共用のメモリに入れておきます。取ってきた命令を蓄えておく命令レジスタ（Instruction Register: ir)を設けます。次にレジスタファイルから読んできたデータを蓄えるreg1, reg2, 計算結果を蓄えるregaluを入れます。メモリの共用のためにアドレスにマルチプレクサを入れます。また、PC＋4を計算するために、ALUのB入力のマルチプレクサを拡張します。



このように考えて作ったマルチサイクルのデータパスを図に示します。命令とデータは共用のメモリに入れておきます。取ってきた命令を蓄えておく命令レジスタ（Instruction Register: ir)を設けます。次にレジスタファイルから読んできたデータを蓄えるreg1, reg2, 計算結果を蓄えるregaluを入れます。メモリの共用のためにアドレスにマルチプレクサを入れます。また、PC＋4を計算するために、ALUのB入力のマルチプレクサを拡張します。



このように考えて作ったマルチサイクルのデータパスを図に示します。命令とデータは共用のメモリに入れておきます。取ってきた命令を蓄えておく命令レジスタ（Instruction Register: ir)を設けます。次にレジスタファイルから読んできたデータを蓄えるreg1, reg2, 計算結果を蓄えるregaluを入れます。メモリの共用のためにアドレスにマルチプレクサを入れます。また、PC＋4を計算するために、ALUのB入力のマルチプレクサを拡張します。



このように考えて作ったマルチサイクルのデータパスを図に示します。命令とデータは共用のメモリに入れておきます。取ってきた命令を蓄えておく命令レジスタ（Instruction Register: ir)を設けます。次にレジスタファイルから読んできたデータを蓄えるreg1, reg2, 計算結果を蓄えるregaluを入れます。メモリの共用のためにアドレスにマルチプレクサを入れます。また、PC＋4を計算するために、ALUのB入力のマルチプレクサを拡張します。

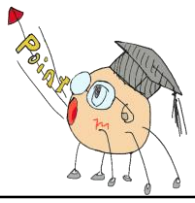
恰好を付ける方法

```
`define SN 5
`define FETCH_B   0
`define DECODE_B  1
`define MEMADR_B  2
`define MEM_B  3
`define EXE_B  4

`define FETCH   `SN'b1<<`FETCH_B
`define DECODE  `SN'b1<<`DECODE_B
`define MEMADR  `SN'b1<<`MEMADR_B
`define MEM   `SN'b1<<`MEM_B
`define EXE  `SN'b1<<`EXE_B
…
reg [`SN-1:0] stat;
```

さて、今まで状態遷移の定義をする場合に生のデータを書いてきましたが、これだと状態を一つ増やす度に多数の行の変更が必要です。このため、普通One hot counterを使う場合は、まずビットの位置に相当する定義をしてしまい、それからその分ビットをシフトする、という定義の方法を使います。このようにすれば、状態数の変更が簡単にできます。まずSNを修正し、その数にあった状態を定義・削除してやれば良いです。演習問題をやる時に状態を付け加える必要がある場合、この方法を取ると便利です。

マルチサイクル マイクロアーキテクチャまとめ

- データパス中にレジスタを入れて途中結果を格納することで、資源の再利用を可能とする
  - 命令・データメモリは兼用
  - PC演算用、分岐演算用の加算器が不要になる
  - しかしレジスタ分の資源は増加する
  - マルチプレクサも増える
  - 1命令実行に複数クロック掛かる
  - クロック数は命令毎に違う
- 制御は有限状態マシン（FSM)で行う。
  - 状態を増やすことで柔軟な制御が可能



ではこの辺でマルチサイクル版をまとめておきます。

シングルサイクル版vs.
マルチサイクル版

- CPUのマイクロアーキテクチャは
  性能、コスト（面積）、消費電力で評価する。
- ここでは性能とコスト（ハードウェア量）を簡単に評価する。
- 本格的な評価は論理合成をやった後

では、次にシングルサイクル版とマルチサイクル版のどちらのマイクロアーキテクチャが有利なのかを評価しましょう。ここでは性能とハードウェア量を簡単に見積もって比較しましょう。本格的な評価は論理合成をやった後で、多分来年のコンピュータアーキテクチャになると思います。

CPUの性能評価式

- CPUの性能はプログラム実行時間の逆数

CPU Time＝プログラム実行時のサイクル数×クロック周期
＝命令数×平均CPI×クロック周期

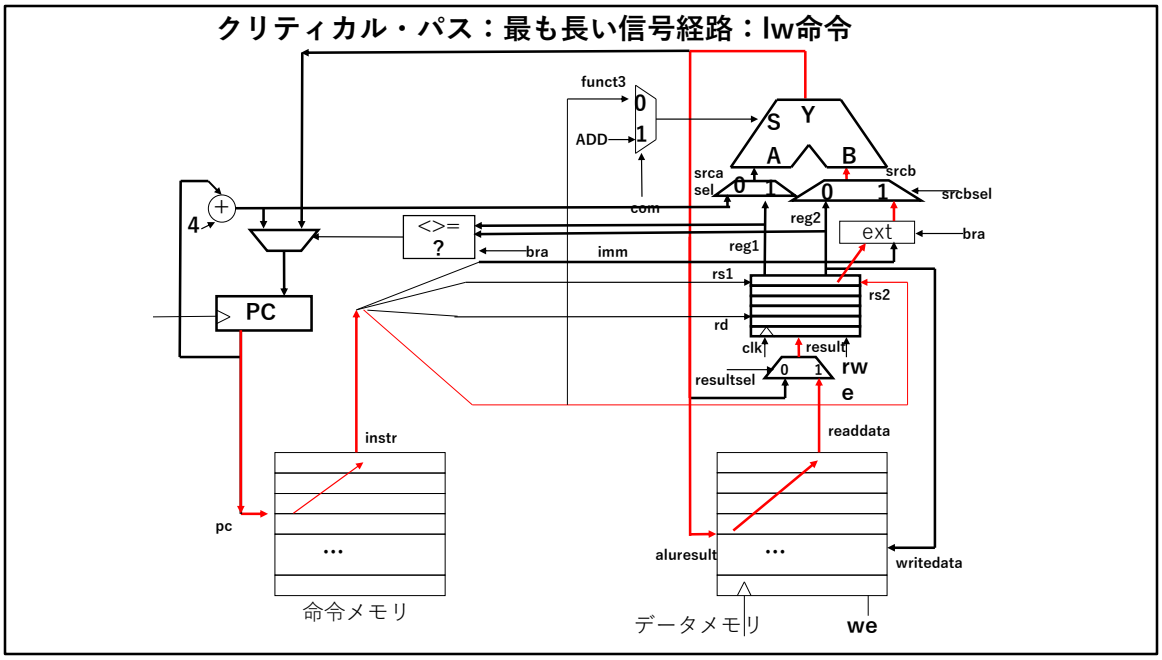
CPI （Clock cycles Per Instruction） 命令当たりのクロック数

→ 1サイクル版では1だがマルチサイクル版では命令によって違ってくる

命令数は実行するプログラム、コンパイラ、命令セットに依存

ここで問題になるのはクロック周期→クリティカルパスによって決まる。

では、次に性能の評価についての一般的な方法を学びます。CPUの性能は、CPUがあるプログラムを実行した際の実行時間の逆数です。実行時間が短い方が性能が高いのでこれは当たり前かと思います。実際のコンピュータではOperating System（OS)が走って実行中にもジョブが切り替わりますが、この影響が入ると困るので、CPUが単一のジョブをOSの介入なしに実行した場合の実行時間（CPU実行時間：CPUTime)を測ります。今まで紹介してきたように、CPUは単一のシステムクロックに同期して動くと考えて良いので、CPU Timeはプログラム実行時のサイクル数×クロック周期で表されます。クロック周期とはクロックが立ち上がってから次に立ち上がるまでの時間で、この逆数がクロック周波数です。プログラム実行時のサイクル数は、実行した命令数×平均CPI（Clock cycles Per Instruction)に分解されます。CPIは一命令が実行するのに要するクロック数で、1サイクル版では全部1ですが、マルチサイクル版では命令毎に違っています。このため、一つのプログラムを動かした場合の平均CPIは、プログラムの種類によって変わります。つまり実行時間の長い命令を多数含んでいるプログラムでは平均CPIは長くなります。もちろんコンパイラにも依存します。



分岐命令用に拡張したRV32Iの構成を示します。RV32Iの分岐命令にはやることが二つあります。レジスタの比較（大小も含め）と、飛び先アドレスの計算です。どちらかにALUを使い、どちらかに専用の回路を設ける必要があります。ここでは、ALUで飛び先を計算することにします。飛び先アドレスを計算するために、PC＋４をALUのAポートに入れるために、マルチプレクサを付けます。B入力のextも、分岐命令のイミーディエイトデータをまとめるために、構造を変更する必要がありますが、この図には表れていません。符号拡張され0を補ったデータがB入力から入ると考えてください。読み出してきた二つのレジスタは、専用の比較器に入力し、大小、等値関係を判定します。分岐命令の成立条件に適合したら、PCの直前のマルチプレクサを切り替えて、飛び先を設定します。

遅延の例

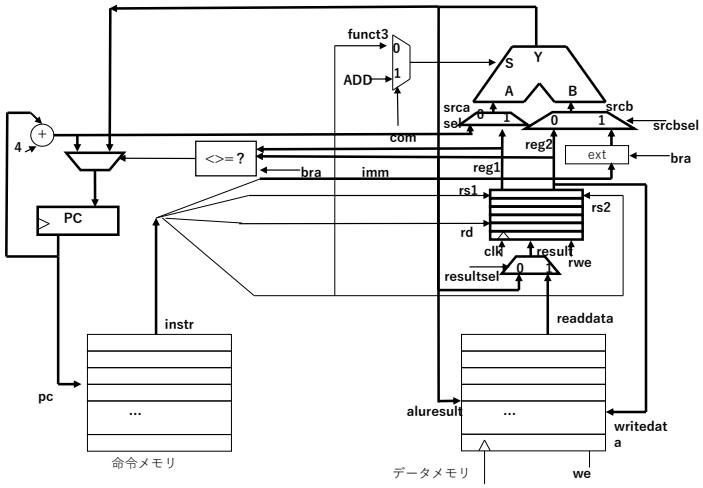
遅延要因	記号	遅延(psec)
レジスタclk→Q	tpcq	30
レジスタセットアップ	tsetup	20
マルチプレクサ	tmux	25
ALU	tALU	200
メモリ読み出し	tmem	250
レジスタファイル読み出し	tRFread	150
レジスタファイルセットアップ	tRFsetup	20

この数値を使うと30+2(250)+150+25+200+25+20=950psec
1.05GHzとなる。

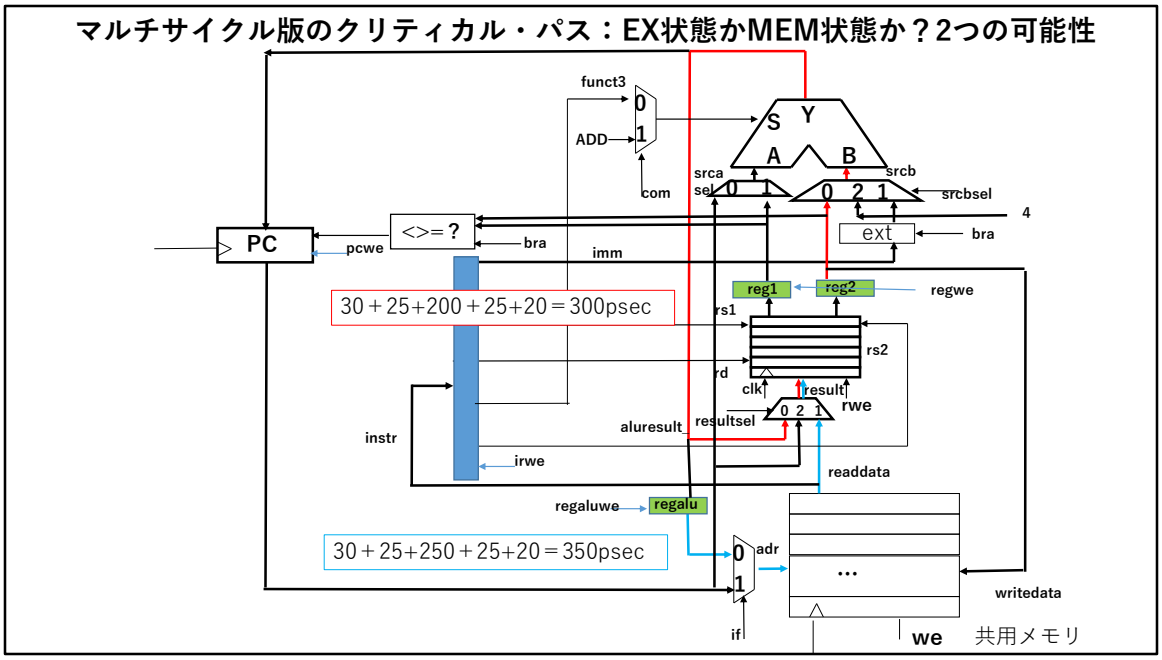
この表は各部の遅延時間の例です。遅延時間はCPUを実装するプロセスによって決まりますが、この値は最近のプロセスとしてリーゾナブルなものです。やはり、メモリの読み出し時間が長いです。ALUは演算機の作り方によりますが、これに次ぐ長さになります。この数値を使うとクリティカルパスは950psecとなり、1.05GH z で動作することがわかります。

コストの計算：シングルサイクル版

モジュール	個数
メモリ	2
レジスタファイル	1
ALU	1
加算器	1
マルチプレクサ	5
レジスタ	1
比較器	1



今回は半導体のコストまでは見積もれないので、その前段階となるモジュール数を見積もって見ましょう。シングルサイクル版ではj命令を実装した段階でのデータパスのリソース使用量は表のようになっています。



マルチサイクル版のクリティカルパスには二つ可能性があります。EXEC状態で、ALUを利用する場合（赤いパス）と、MEM状態でメモリから読み出しを行う場合（青いパス）です。ここではメモリの遅延が大きいため、後者の方（青い線）が大きくなり、350psecになります。

マルチサイクル版性能解析

- クリティカルパス：今回の仮定では
  - ALU：tpcq+tmux+tALU+tmux+tsetup
    30+25+200+25+20＝300ps
  - メモリ：tpcq+tmux+tmem+tsetup
    30+25+250+25+20=350ps　（3.07GHｚ）
- 平均CPI
  - multの実行結果から3.29
- 325×3.29＝1069
- これはシングルサイクルの950より長い(つまり遅い)
- なぜだろう？

この二つのパスを先ほどの値を入れて検討するとこのようになります。マルチサイクル版は、メモリアクセス命令ではCPI=4、それ以外ではCPI=3となります。multの評価結果から平均CPUは3.29になりました。シングルサイクルと性能を比較すると完敗です。これは、一命令あたりのクロックサイクル数が増えた割には、遅延時間が減っていないためです。クリティカルパスをきっちり4等分するのは不可能に近く、今回の実装はそこそこがんばっている方です。マルチサイクル版は、性能面ではシングルサイクルに勝てない場合が多いです。

性能の比較

- CPU　A　10秒で実行
- CPU　B　12秒で実行
- Aの性能はBの性能の1.2倍
  遅い方の性能（速い方の実行時間）を基準にする

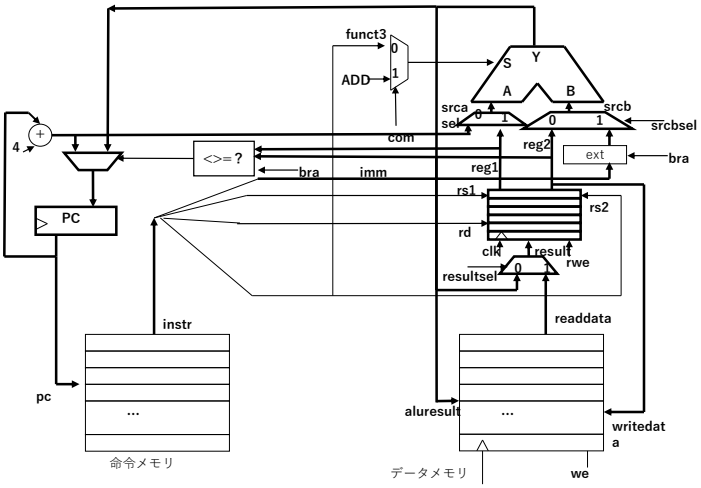
$$\frac{\text{CPU　Aの性能}}{\text{CPU　Bの性能}} = \frac{\text{CPU　Bの実行時間}}{\text{CPU　Aの実行時間}}$$

×BはAの1.2倍遅い　この言い方は避ける
今回は、1069/950＝1.125
シングルサイクル版が1.125倍速い（12.5%速い）

では次に性能の比較方法について検討します。CPU Aはあるプログラムを10秒で実行し、Bは同じプログラムを12秒で実行します。AはBの何倍速いでしょう？この場合、Bの性能を基準とします。Bの性能はBの実行時間の逆数、Aの性能はAの実行時間の逆数なんで分子と分母が入れ替わり、Bの実行時間をAの実行時間で割った値となります。これは12／10で1.2倍になります。ではBはAの何倍遅いのでしょうか？この考え方は基準が入れ替わってしまうため混乱を招きます。このため、コンピュータの性能比較では常に遅い方の性能（つまり速い方の実行時間）を基準に取ってで、（速い方）は（遅い方）のX倍という言い方をします。

コストの計算：シングルサイクル版

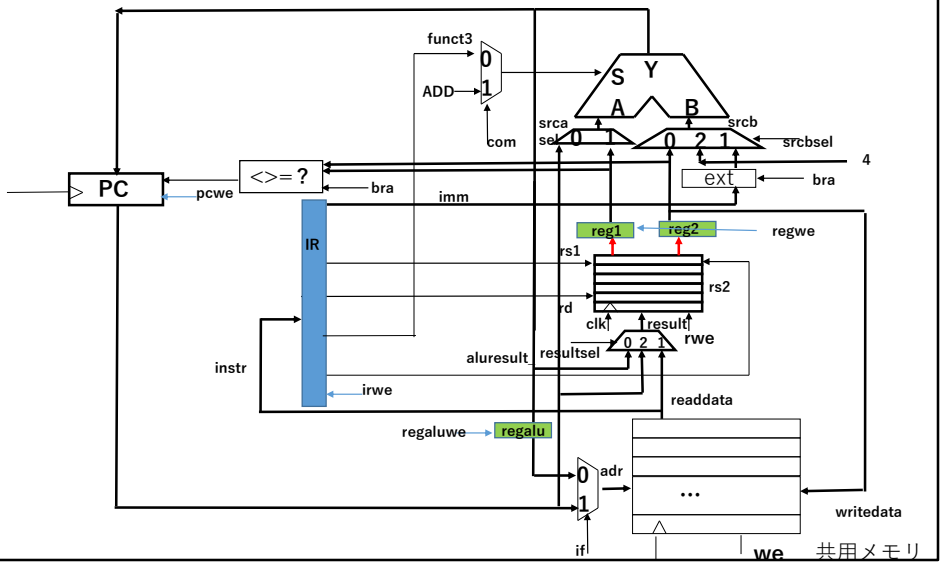
モジュール	個数
メモリ	2
レジスタファイル	1
ALU	1
加算器	1
マルチプレクサ	5
レジスタ	1
比較器	1



シングルサイクル版のコストを表に示します。

コストの計算：マルチサイクル版

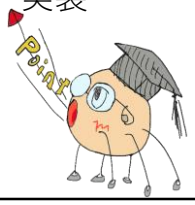
モジュール	個数
メモリ	1
レジスタファイル	1
ALU	1
加算器	0
マルチプレクサ	5
レジスタ	5
比較器	1



一方、マルチサイクル版は、メモリが一つで済み、加算器がなくなった一方、レジスタが増えています。とはいえ、レジスタのハードウェア量はさほど大きくないことを考えると、コスト的にはかなり有利と言えると思います。ただし、このコストには制御回路のFSMのは含まれていないので注意は必要です。

性能とコストの比較のまとめ

- ISAが同じ場合、性能は、クロック周期とCPIで決まる。
  - クロック周期はクリティカルパスで決まる。
  - CPI（Clock cycles Per Instruction)は、シングルサイクル版は常に１だがマルチサイクル版は動作させるプログラムに依存
  - 性能比較は、遅い方の性能（速い方の実行時間）を基準にする。
- コストは必要モジュール数で評価したが、実装の状況により異なる。



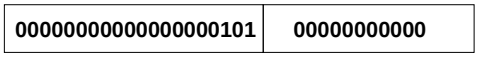
性能とコストの比較の部分をまとめます。

演習１　lui（Load Upper Immediate)の実装

上位20bitに直値を設定する命令

lui rd, imm

- 下位は0にする
- lui x1,5



- x1を0x12345678に設定せよ
- lui x1, 0x12345
- ori x1, 0x678

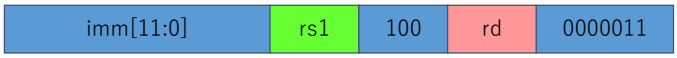
- 前回同様にlui.asmを利用する
- 提出物はluiの付いたrv32i.v

これは、シングルサイクル版でやった演習と同じです。RISC-Vのイミーディエイト命令は12ビットなので、この範囲を超えると値を入れにくいです。このため、レジスタの上位20ビットにデータを入れる命令が用意されています。この命令はセコイ感じもしますが、便利なので、全てのRISCが持っています。

演習2

符号無の比較命令

bltu rs1,rs2,X 命令を付け加えよ



正規の命令なのでアセンブラが利用可能

make bltuでimem.datを生成

8番地でループしたら成功

8番地以降まで進んでしまったら失敗

提出物はbltuの付いたrv32i.v(luiが付いていてもOK)

演習3　性能評価

- 0x400番地から並んでいる8個のデータの総和を求めるプログラムmsum.asmを実行し、マルチサイクル版のCPIを求めよ。
- 授業中のスライドの数値より、やや高速なメモリを利用したことで、tmem=180nsecになった。この時、シングルサイクル版とマルチサイクル版の性能を比較せよ。

- 両者のmsumの実行時間を示し、XXがYYのZZ倍速いという言い方で示せ。
- テキストファイルを提出のこと