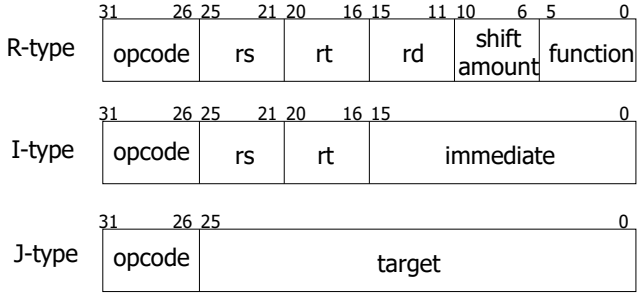
MIPSのマルチサイクル マイクロアーキテクチャ

慶應義塾大学
天野

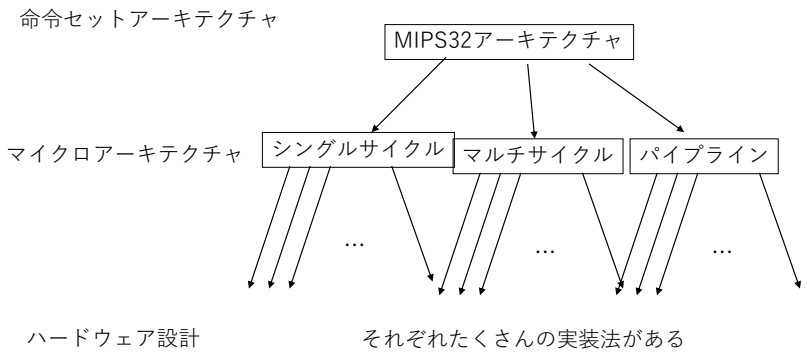
命令フォーマット

• 3種類の基本フォーマットを持つ



·まずMIPSの命令フォーマットを復習しておきます。
·MIPSの命令フィールドの決め方は以下の通りです。
·op(opcode):命令の種類を表すオペコードフィールド。
· FUNC(functional code):opフィールドで表現しきれない場合に補助オペコードとしてopフィールドを拡張する形で用いるフィールド。
· rs,rt,rs:5bitのレジスタ番号、rsの内容は命令実行後変化しない。rdには演算結果が格納される。rtは命令に応じて役目が切り替わる。
· Immediate:命令に直接値を埋め込むのに使用するフィールド。MIPSでは16ビット。
· target:immediateと同様に命令に直接値を埋め込むが、targetはjとjalで専門に使われ、なるべく遠くに飛ぶために26ビット分用意される。これの下に00が補われ(命令は4の倍数のアドレスに決まっているので)、命令のアドレスが指定される。

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

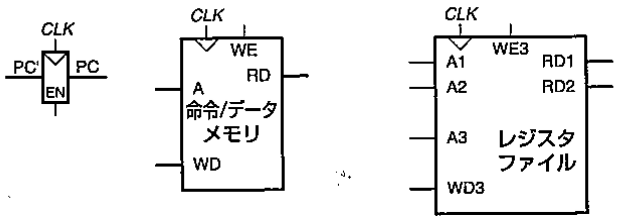
- 今までのMIPSeシングルサイクル版
- 利点
  - 設計が簡単
  - 案外性能が高く、電力も小さい
- 欠点
  - 資源の共有ができない。特にメモリの分離が必要
  - 最も長いクリティカルパスにクロック周期が制約される

  →マルチサイクル版

今まで紹介してきたのはシングルサイクル版のMIPSe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。

マルチサイクル
マイクロアーキテクチャ

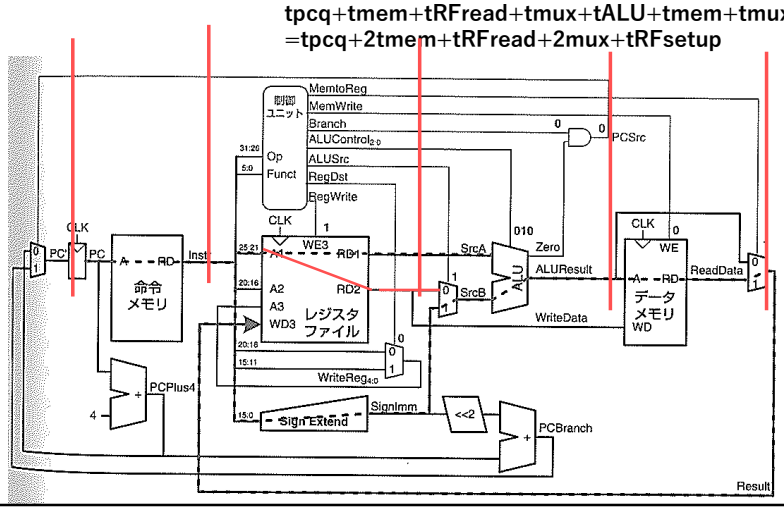
- 命令とデータメモリを兼用にする
- アーキテクチャ要素は以下の３つだけ



ではマルチサイクル版のCPUをシングルサイクル版同様に設計していきましょう。命令メモリとデータメモリを共用するため、メモリは単一ですみます。

クリティカルパスをどこで切るか

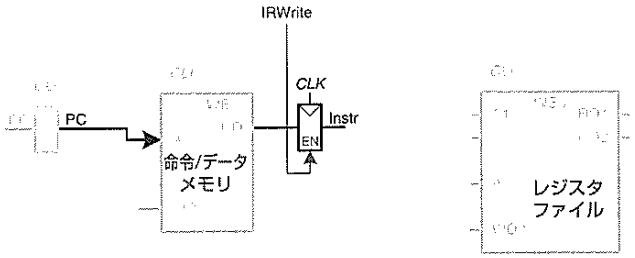
tpcq+tmem+tRFread+tmux+tALU+tmem+tmux+tRFsetup
=tpcq+2tmem+tRFread+2mux+tRFsetup



マルチサイクル版の設計には全体のクリティカルパスをできるだけ等しい遅延に分割します。分割した所にレジスタを入れて、データを一時的に蓄えるようにします。

命令フェッチステップ

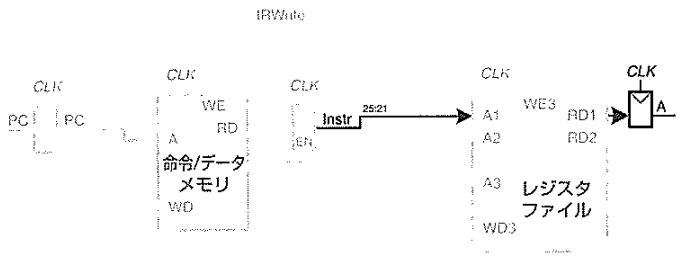
命令レジスタを設け、命令を保存



では、シングルサイクル版同様、lw命令から順番にデータパスを作って行きましょう。まず、プログラムカウンタの指示する命令をフェッチし、これをレジスタに入れます。このレジスタは命令レジスタ（Instruction Register: IR)と呼び、実行中の命令を保持します。

命令デコードステップ

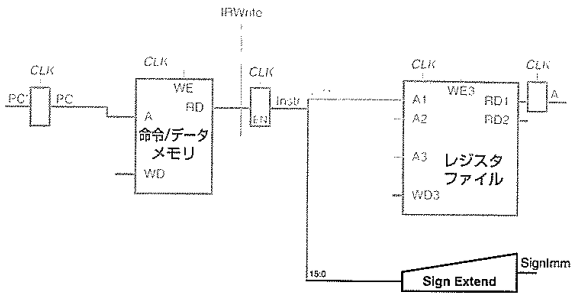
- レジスタファイルから読み出した所でデータを保存する



次に、この命令中のrs(レジスタファイル番号)に従ってディスプレースメントが入っているレジスタファイルを読み出します。ここまではシングルサイクル版と同じですが、読み出した命令はAレジスタにしまっておきます。レジスタファイルからレジスタを読み出している間に、読み出してきた命令のデコードを行うことから、この状態を命令デコードステップと呼びます。

命令デコードステップ

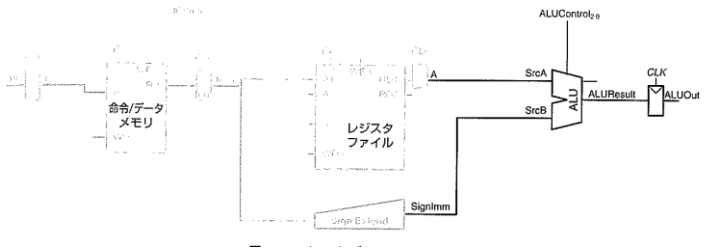
- 符号拡張は従来通り



符号拡張はシングルサイクル版と同じで命令の下16ビットを拡張します。

演算ステップ

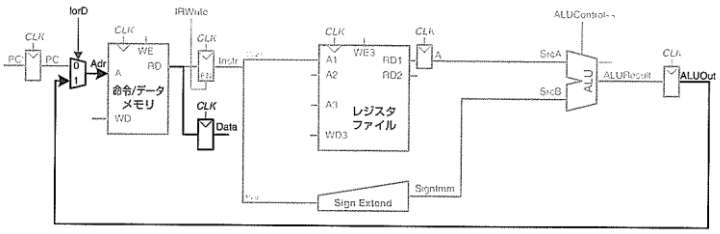
- ALUの演算結果をレジスタに保存



Aレジスタの値と符号拡張したディスプレースメントを加算し実効アドレスを計算して、レジスタALUOutに格納します。

データの読み出しステップ

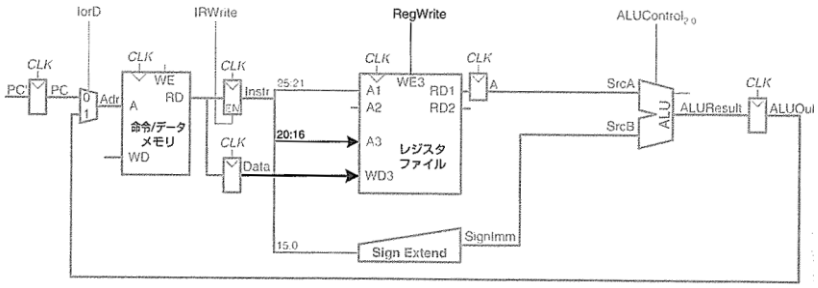
- データレジスタに読み出したデータを保存



この値でデータメモリを読み出します。読んだ値はレジスタDataに入れます。

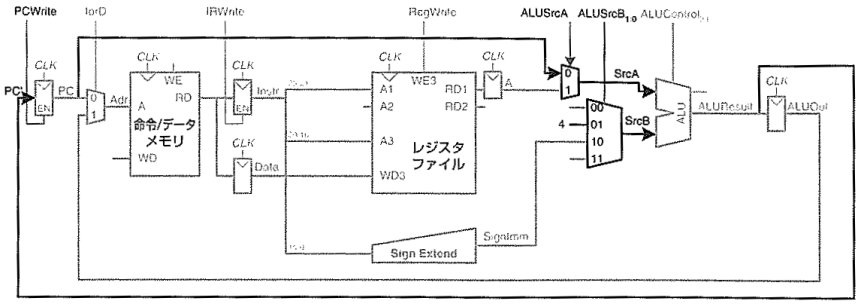
結果の書き込みステップ

- データレジスタの内容をレジスタファイルに書き込み



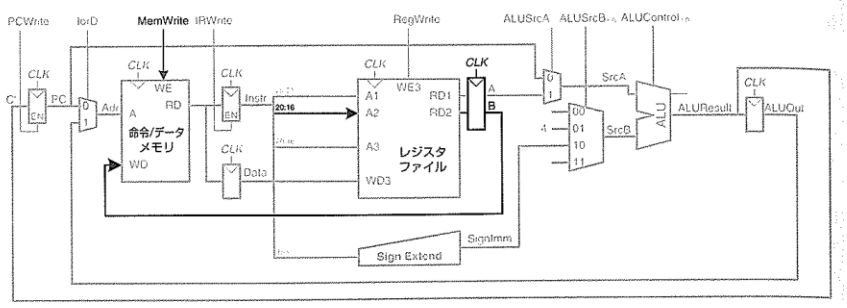
最後にDataレジスタの値をレジスタファイルに書き込みます。書き込む番号は20:16でrtに当たります。

PCのカウントアップ、飛び先計算もALUでやらせる



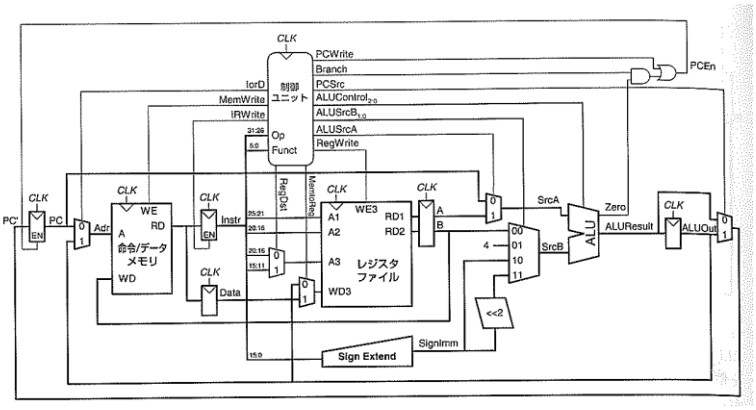
マルチサイクル版では、PCのカウントアップ、分岐命令の飛び先計算などをすべてALUにやらせます。このために、A,B両方の入力にマルチプレクサが必要で、B入力のマルチプレクサは拡張されています。PCと 4 を足した結果は、直接PCにフィードバックされます。これでlw命令の実行は終わりです。

sw命令の実装



Sw命令はlw命令とほとんど同じですが、レジスタファイルの2ポート目から読み出した値が書き込むべきデータになるので、これをメモリのデータ入力につなぐ必要があります。

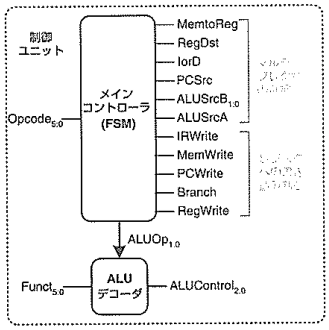
R型命令、beq命令、制御ユニットの付加



R型命令、beq命令を実行するために、さらに図のように拡張を行い、制御ユニットを付ければ基本的な部分はできあがりです。

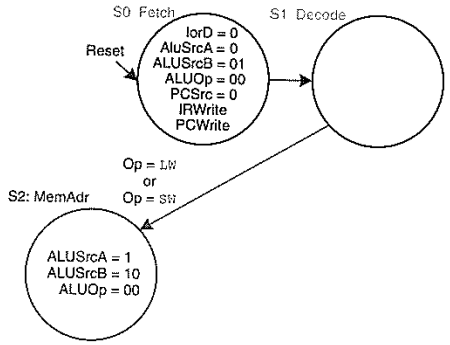
制御回路の内部構成

- メインコントローラは有限状態マシン（FSM）
- ALUデコーダは以前と同じ



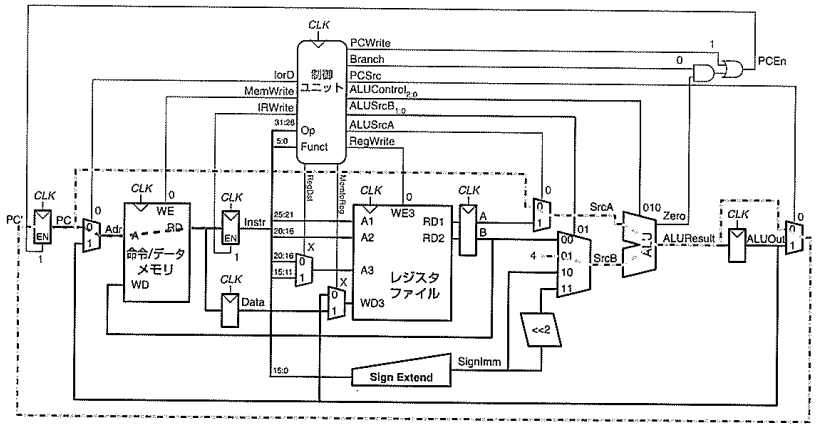
では、制御回路はどうなるでしょう？シングルサイクル版と違って、有限状態マシン（Finite State Machine)になります。これは同期式順序回路で、計算機基礎で設計法を勉強したもので、状態遷移によって制御を行っていきます。FSMは状態遷移図を描いてしまえば、システマチックに設計ができます。Verilog HDLで記述する場合、状態遷移図から回路までについては頭を悩ます必要はありません。要するにいかに状態遷移図を作るか、が問題になります。

フェッチ、デコード、メモリアドレス計算



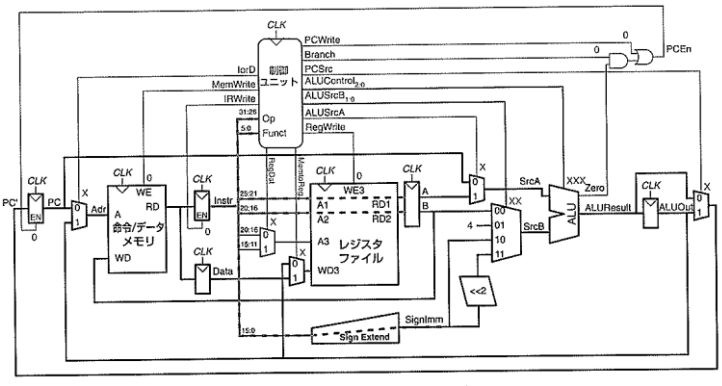
ではこのFSMの状態遷移図を作りましょう。データパス同様、lw命令をまず実装します。今回は状態に対応して出力が決まるMoore型を使います。最初の状態S0:Fetchでは、メモリから命令を読み出してこれを命令メモリに入れます。次にこれに従って、レジスタファイルからレジスタを読み出し、符号拡張をします。これがS1:Decodeです。ここで読んできた命令に依存して次の状態を決めることができます。lw命令の場合は、S2:MemAdrに遷移し、実効アドレスの計算を行います。この3つの状態でのデータの動きを見て行きましょう。各状態では○の中に示した信号線を制御します。ここで、IRWrite, PCWriteなどの書き込み制御線は、信号線名が書かれている状態で1になり、書き込みが行われます。その他の制御線はマルチプレクサを制御しますが、何も書いていなければDon’t careです。

フェッチステップでのデータの流れ



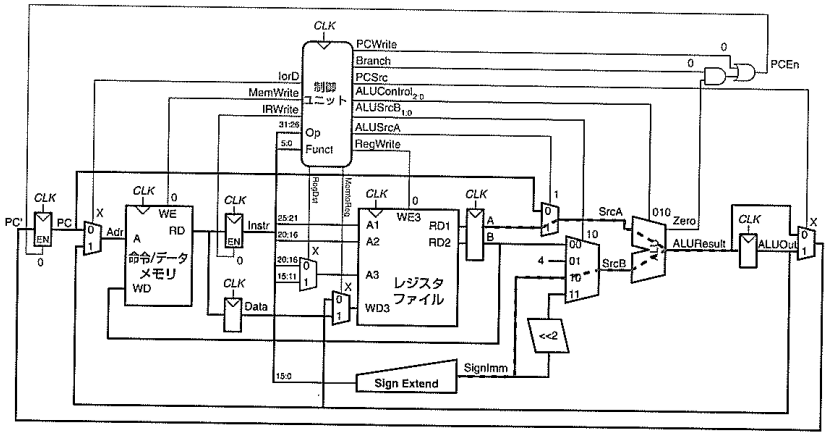
命令フェッチでは、①命令をフェッチしてIRに入れる、②PCをPC+4にする。の二つの仕事をします。①は、IorDを0にしてPCをメモリのアドレスに入れてやり、読み出したデータをIRWriteを1にしてIRに書き込みます。
②は、ALUSrcAを0、ALUSrcBを01にしてPCの値と4をALUに入れます。ALUControlを010にしてこれを加算して、PCSrcを0にしてこれをPCの入力に引っ張ってきます。そしてPCWriteを1にしてPCにPC+4を書き込みます。これでPCは更新されました。
この二つの作業を行うため、かなり多くの信号線を操作していることが分かります。

デコードステップでのデータの流れ



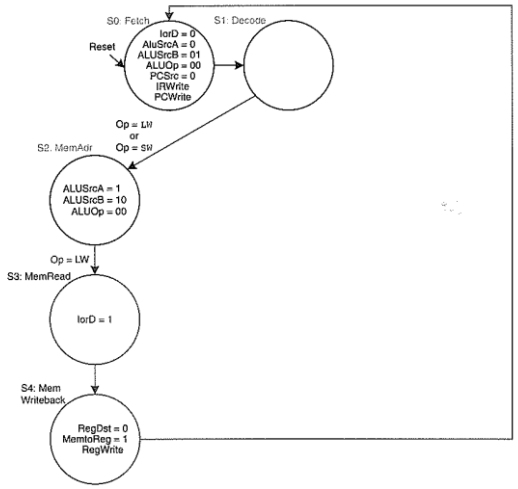
次のデコードステップですが、①読んできたレジスタ番号によってレジスタファイルを読み出してこれをA、Bレジスタに格納する。②符号拡張を行う、の2つの仕事をします。これは配線構造によって自動的に行われるので、制御を行う必要はありません。さらに③命令のopcode, functによって状態遷移を行います。これが命令デコードに当たります。lw命令の場合は、次にS2に遷移しますが、命令の種類に応じて様々な状態に遷移していきます。

アドレス計算でのデータの流れ



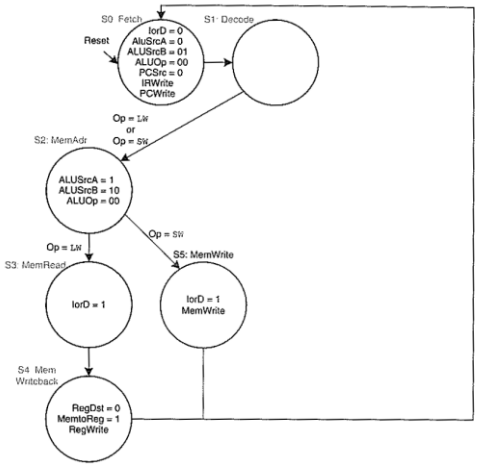
lw命令では、アドレス計算を行うため、ALUSrCAを1にしてレジスタを通し、ALUSrcBを10にしてSignImm（符号拡張したディスプレースメント）を通してやります。ALUControlを010にして加算を行い、結果はレジスタALUOutに格納します。

結果の書き込みと次の命令フェッチ



lw命令の場合、次のS3でメモリを読み出します。このため、IorDを1にしてPCに代わってALUOutをアドレスに入れてやります。読んだデータは自動的に次のクロックの立ち上がりで、レジスタDataに入ります。最後にS4でDataの値をレジスタファイルに書き戻してやります。このために、RegDstを0にしてrtを結果の書き込み用レジスタ番号として設定します。次にMemoryReg=1としてWD3にDataを入れてやり、RegWrite＝１でこの値をレジスタファイルに書き込みます。次の状態として再び命令フェッチに遷移します。PCは既に 4 カウントアップされているので、つぎの命令がフェッチされます。

メモリ書き込み命令の制御



swなどの書き込み命令は、S2までは実効アドレス計算なのでlwと同じです。書き込み用の状態S5を設けS3同様、IoD=1として、実効アドレスをメモリに送ってやります。ここで、MemWrite=1としてデータを書き込みます。書き込むデータはBレジスタに読み出されており、これは直接レジスタファイルのWDにつながっている点に着目してください。lw同様、次の状態はS0に遷移します。

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12状態＝12ビット
  - FETCH:　12'b0000_0000_0001
  - DECODE: 12'b0000_0000_0010
  - MEMADR:12'b0000_0000_0100
  …
- レジスタstatで状態を保持する
  - reg [11:0] 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12状態あるので12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 if….
          `MEMADR: if(lw_op) stat <= `MEMREAD;
                          else stat<= `MEMWR;
          `MEMREAD: stat <= `MEMWBACK;
          `MEMWBACK: stat<= `FETCH;
          `MEMWR: stat <= `FETCH;
        ….
```

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。

状態の判別

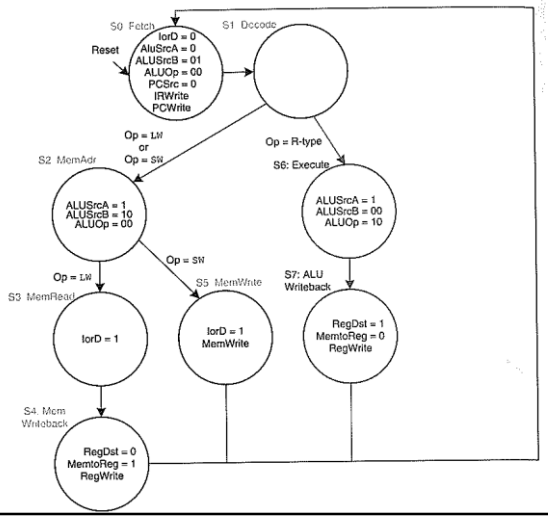
- 各状態の0の位置を__Bで定義する
  - FETCH: 12'b0000_0000_0001　→　FETCH_B: 4'b0000
  - DECODE: 12'b0000_0000_0010　→DECODE_B: 4'b0001
  - MEMADR: 12'b0000_0000_0100 →　MEMADR_B: 4'b0010
- statのビット位置を調べれば状態が分かる
  - stat[`FETCH_B]が1ならばFETCH状態
  - stat[`DECODE_B]が 1 ならばDECODE状態
  - stat[`MEMADR_B]が 1 ならばMEMADR状態
- 様々な記述でこの点を利用する

  - 例）命令レジスタ(instr)の記述

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

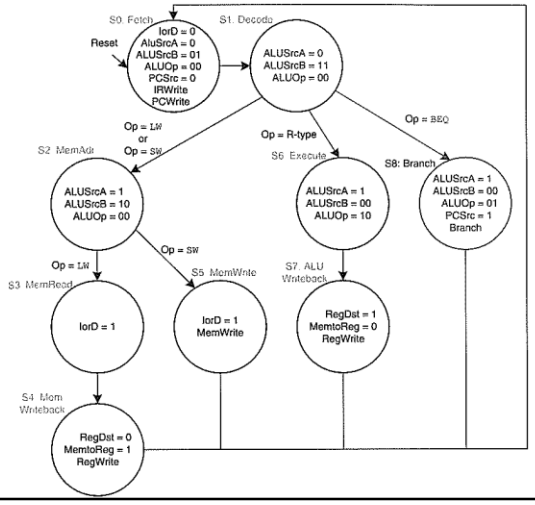
今回はMoore型なので、状態が決まれば、その状態により出力（あるいはデータパスでやること）が決まります。記述をするには、この状態マシンの現在の状態が何なのかを知る必要があります。One Hot Counterはこれが簡単にできます。今、それぞれの状態に対して状態__Bに対してそのビットの位置を定義します。例えばFETCHに対してはFETCH_B＝０、DECODE__B＝１、MEMADR__B＝２になります。このビット位置をstatの配列の中に入れてやれば、そのビットを切り出すことができます。One Hot Counterは、対応するビットが1ならば、状態マシンがその状態になっているので、判別が簡単にできます。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instrに蓄えます。

R型命令の制御



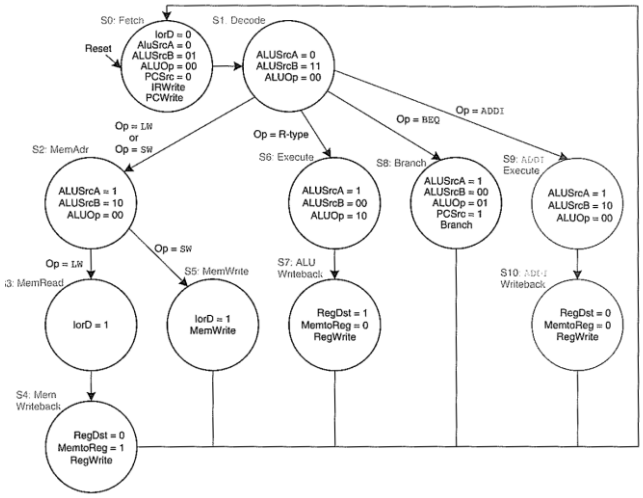
R型命令はIDの次に新しい状態S6に遷移します。ここから先は皆さんで状態の中の信号の変化を追ってください。

分岐命令の制御



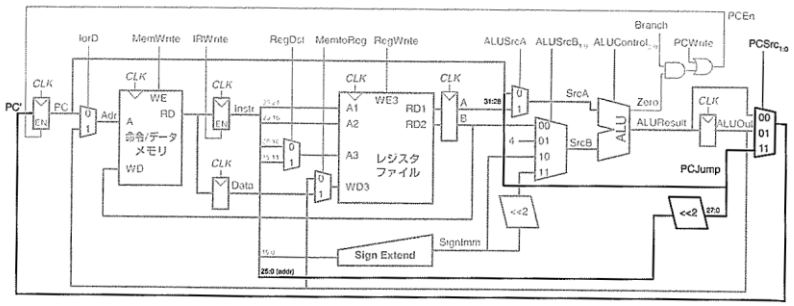
分岐命令では今までALUを使っていなかったS1:ID状態で、ALUに飛び先を計算させます。この飛び先をALUOutに入れておき、S8でALUで引き算を行って飛ぶかどうかを判定して、飛ぶ場合にはこの値をPCにセットします。この制御は、かなりトリッキーで、面白いです。

addi命令の制御



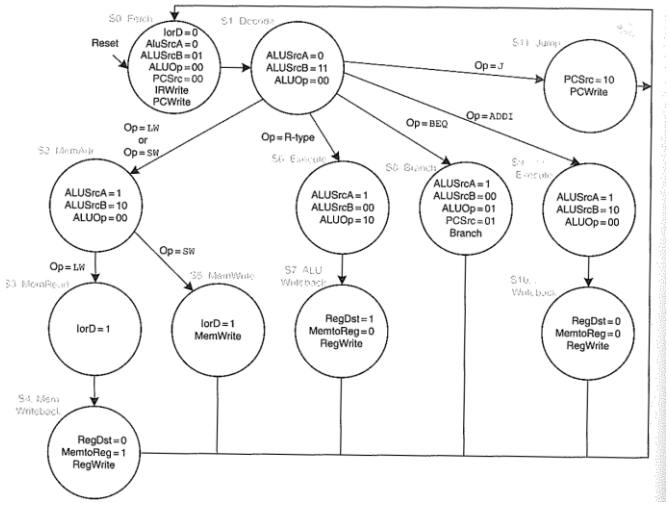
ADDIなどイミーディエイト命令は、lw,sw同様に符号拡張したイミーディエイトと、レジスタの値を加算します。

j命令用のデータパス強化



J型は、PCの上位4ビットと命令コード中の26ビットを2ビット左シフトした28ビットをくっつけるため、今までと違ったデータパスが必要になります。この図ではPCS r Cのマルチプレクサを拡張することで、これを実現しています。

j命令用の制御



J型命令は、飛び先のPCを構成してしまえば終わりなので、3クロックで実現することができます。この場合専用の状態を付け加えます。このように必要とされる機能ごとにデータパスをマルチプレクサを入れて拡張し、状態を増やすことで、様々な命令を実装できます。マルチサイクル版は、シングルサイクル版に比べて様々な命令が実装可能であることが分かります。ただし、複雑な命令、多様な命令を実装するとその分、状態遷移も複雑になり、データパスもごちゃごちゃします。

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mipse.asm

```
        lw $1,0x1000($0)
        lw $2,0x1004($0)
        add $3,$0,$0
loop:   add $3,$3,$2
        addi $1,$1,-1
        bne $1,$0,loop
        sw $3,0x2000($0)
end:    j,end
```

データメモリを0 x 1000から置いた

0 x 2000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）

- make mipse: 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./mipse（vpp mipse)で行う
- 2000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mipse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 mipse(
input clk, rst_n,
input [`DATA_W-1:0] readdata,
output [`DATA_W-1:0] adr,
output reg [`DATA_W-1:0] b,
output memwrite);
```

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、bはメモリへの書き込みデータです。なんでこれがbなの？と思うかもしれませんが、図を見るとわかるようにsw命令での書き込みデータはbレジスタから出てくるので、これを直接繋いでやっても大丈夫です。memwrite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 [`DATA_W-1:0] pc;
reg [`DATA_W-1:0] instr;
reg [`DATA_W-1:0] a;
reg [`DATA_W-1:0] data;
reg [`DATA_W-1:0] aluout;
wire [`DATA_W-1:0] rd1,rd2,wd3;
reg [11:0] stat;

wire [`DATA_W-1:0] srca, srcb, aluresult;
wire [`OPCODE_W-1:0] opcode;
wire [`SHAMT_W-1:0] shamt;
wire [`OPCODE_W-1:0] func;
wire [`REG_W-1:0] rs, rd, rt, writereg;
wire [`SEL_W-1:0] com;
wire [`DATA_W-1:0] signimm;
wire [`DATA_W-1:0] pcplus4;
wire regwrite;
```

命令メモリ

a,data,aluoutは図と対応のこと

状態はstatに記憶

信号線名も図と対応のこと

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instr, レジスタファイルの値を一時記憶するa(bは出力レジスタで定義してしまったのでここにはないです）、データメモリから読んできた値を蓄えるデータレジスタdata、ALUの出力を一時記憶するaluout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。

```
wire sw_op, beq_op, bne_op, addi_op, lw_op, j_op, alu_op;
wire zero;

assign {opcode, rs, rt, rd, shamt, func} = instr;
assign signimm = {{16{instr[15]}},instr[15:0]};

// Decorder
assign sw_op = (opcode == `OP_SW);
assign lw_op = (opcode == `OP_LW);
assign alu_op = (opcode == `OP_REG) & (func[5:3] == 3'b100);
assign addi_op = (opcode == `OP_ADDI);
assign beq_op = (opcode == `OP_BEQ);
assign bne_op = (opcode == `OP_BNE);
assign j_op = (opcode == `OP_J);
```

命令はinstrに保存されている

デコードはシングルサイクルと同じ

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instrに入っているので、そこからopcode,レジスタ、func, immを切り出します。デコーダでそれぞれの命令をデコードしてやります。

```
// State Machine
always @(posedge clk or negedge rst_n)    begin
        if(!rst_n) stat <= `FETCH;
        else
          case (stat)
           `FETCH: stat <= `DECODE;
           `DECODE: if(lw_op | sw_op) stat <= `MEMADR;
                                      else if (alu_op) stat <= `EXECUTE;
                                      else if (bne_op | beq_op)  stat <= `BRANCH;
                                      else if (addi_op) stat <= `ADDIEX;
                                      else if (j_op) stat <= `JUMP;
           `MEMADR: if (lw_op) stat <= `MEMREAD;
                                      else stat <= `MEMWR;
           `MEMREAD: stat <= `MEMWBACK;
           `MEMWBACK: stat <= `FETCH;
           `MEMWR: stat <= `FETCH;
           `EXECUTE: stat <= `ALUWBACK;
           `ALUWBACK: stat <= `FETCH;
           `BRANCH: stat <= `FETCH;
           `ADDIEX: stat <= `ADDIWB;
           `ADDIWB: stat <= `FETCH;
           `JUMP: stat <= `FETCH;
           endcase    end
```

状態遷移：図と対応のこと

では先の状態遷移図がVerilogでどのように記述されるかを見ましょう。基本的に記述は図と1対１対応しています。状態名も同じにしてあります。これはmipse.vでは最後の部分に書いてありますが、ここでは、状態遷移が図と同じであることを理解してから全体の記述を見て行くことにします。

```
// MemWrite
assign adr =  stat[`FETCH_B] ? pc : aluout;
// MemWrite
assign memwrite = stat[`MEMWR_B];

// ALU op
assign com = stat[`FETCH_B] | stat[`DECODE_B] |
                    stat[`MEMADR_B] | stat[`ADDIEX_B] ? `ALU_ADD :
                    stat[`BRANCH_B] ? `ALU_SUB : func;

// ALU srcb
assign srcb = stat[`FETCH_B] ? 4 :
                        stat[`DECODE_B] ? signimm << 2:
                        stat[`MEMADR_B] | stat[`ADDIEX_B] ?
                                            signimm: b;
// ALU srca
assign srca = stat[`FETCH_B] | stat[`DECODE_B] ? pc : a;
```

メモリのアドレスと書き込み

ALUのコマンドは状態で決まる

ALUの入力も状態で決まる

次にメモリ周辺とALU周辺です。全ての信号線は状態によって決まります。例えばadrはフェッチではpcそれ以外ではディスプレースメントとレジスタを加算した値が出てくるaluoutになります。メモリの書き込みはMEMWR状態のみで行われます（st_opを入れてはダメなことに気を付けましょう）。ALUのコマンドはFETCH,DECODE,MEMADR,ADDIEXでは加算、BRANCHでは引き算、それ以外はfuncで決めます。ALUのB入力はFETCHではpcに加えるための4、DECODEでは飛び先、MEMADRとADDIEXではイミーディエイト、それ以外では b レジスタを入れます。A入力はFETCHとDECODEではpcを入れ、それ以外はaレジスタを入れます。FETCHではpcに4を足すためで、DECODEでは分岐命令ならば飛び先を計算するためです。状態によって入力を選択しているのに注意してください。

レジスタファイル周辺

```
//RegDst
assign wd3 = stat[`MEMWBACK_B]  ? data : aluout;

//RegWrite
assign regwrite =
stat[`MEMWBACK_B] | stat[`ALUWBACK_B] | stat[`ADDIWB_B];

//MemtoReg
assign writereg = stat[`ALUWBACK_B] ? rd : rt;

alu alu_1(.a(srca), .b(srcb), .s(com), .y(aluresult), .zero(zero));

rfile rfile_1(.clk(clk), .rd1(rd1), .a1(rs), .rd2(rd2), .a2(rt),
        .wd3(wd3), .a3(writereg), .we3(regwrite));
```

次にレジスタファイル周辺を記述します。書き込みデータのwd3にはMEMWBACK、つまり読んできた結果を書き込む時はdata,それ以外はALUの出力のレジスタaluoutを入れてやります。レジスタの書き込みは、それぞれのレジスタ書き込み状態MEMWBACK, ALUWBACK, ADDIWBで１になるようにします。書き込むレジスタ番号ですが、ALUWBACKの時はrd,それ以外はrtです。この辺、操作と状態が一対一対応しているのでわかり易いと思います。ALU、レジスタファイルとの入出力は以前とほとんど同じです。

```
// Instr
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) instr <= 0;
  else if (stat[`FETCH_B]) instr <= readdata;
end

// ALUOUT
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) aluout <= 0;
  else aluout <= aluresult;
end
```

命令レジスタはFETCH状態のみ

ALUの出力レジスタは毎クロック入れる

ではそれぞれのレジスタの記述をしましょう。命令レジスタにはFETCH状態の時のみメモリからの値を入れてやります。ALUの出力は命令に依らず、毎クロック値を入れます。

```
// DATA
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) data <= 0;
  else data <= readdata;
end

// A,B
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) begin a <= 0; b<= 0; end
  else if (stat[`DECODE_B]) begin a <= rd1; b<= rd2; end
end
```

データレジスタは毎クロック

レジスタファイルからのA,Bレジスタは DECODE状態のみで値を格納

データレジスタは、毎クロックメモリからのデータを格納します。もちろんMEMREADの時だけ格納させてもいいのですが、ま、毎クロックやっても害がないのでそうなっています。これは実は状態遷移図もそうなっていて、これに合わせてあります。ハードウェアを簡単にするためにはこのようにした方が有利です。A,BレジスタはDECODE状態の時のみ値を蓄え、後の状態ではこれを保持しているようにしています。A,Bは全く同じ動作をするので、同じalways文を使って書いています。

```
// PC
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else if (stat[`JUMP_B])
        pc     <= {pc[31:28],instr[25:0],2'b0};
  else if (stat[`BRANCH_B] & ((beq_op & zero) | (bne_op & !zero)))
    pc <= aluout;
  else if(stat[`FETCH_B])
    pc <= aluresult;
end
```

j命令

beq, bne命令

pcに4を足す

PCの動作はやや面倒です。ここでは状態をif-elseで記述していますが、本来それぞれの状態は排他的なので、順番はどうでもいいです。まずJUMP状態では28ビットの飛び先を上位4ビットのPCとくっつけて飛び先とします。BRANCH状態では分岐が成立するかを調べて、成立した時のみ、aluoutに格納された飛び先をpcに入れてやります。この飛び先は一つ前のDECODE状態で計算された結果を使っており、BRANCH状態ではALUの出力は引き算をやってA,Bの両レジスタを比較しているのです。この辺、ちょっとトリッキーなんですが、これは状態遷移に合わせた結果です。（僕のせいではなくパターソン＆ヘネシーのテキスト、Harris&Harrisのテキストも両方ともこれを使っているので止められないのです）FETCH状態ではALUの結果をそのまま書き込んでいますが、これはpc+4が計算されているのです。

恰好を付けた版：mipsek.v

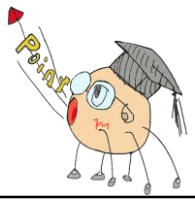
```
`define SN 12
`define FETCH_B    0
`define DECODE_B   1
`define MEMADR_B   2
….
`define BRANCH_B   8
`define ADDIEX_B   9
`define ADDIWB_B   10
`define JUMP_B     11

`define FETCH    `SN'b1<<`FETCH_B
`define DECODE   `SN'b1<<`DECODE_B
`define MEMADR   `SN'b1<<`MEMADR_B
…
`define BRANCH   `SN'b1<<`BRANCH_B
`define ADDIEX   `SN'b1<<`ADDIEX_B
`define ADDIWB   `SN'b1<<`ADDIWB_B
`define JUMP     `SN'b1<<`JUMP_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mipsek.vを使った方が楽にできます。make mipsekとやってから./mipsekで実行可能です。

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

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、mipse1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。

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

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

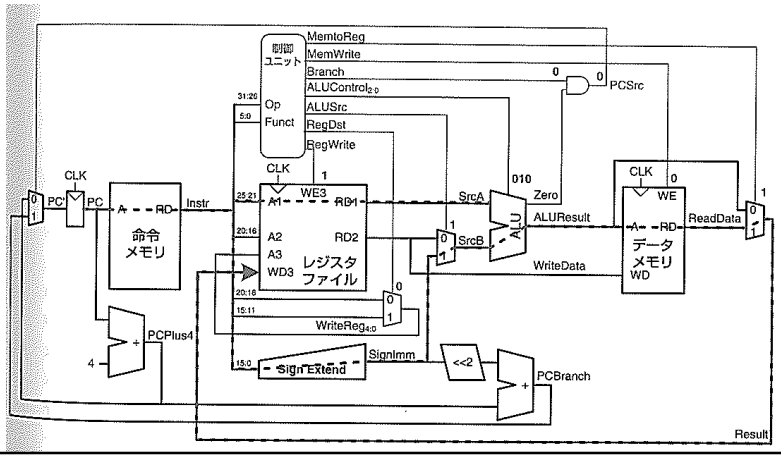
×BはAの1.2倍遅い　この言い方は避ける

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。

シングルサイクル版のクリティカルパス

- lw命令が最も長い

tpcq+tmem+tRFread+tALU+tmem+tmux+tRFsetup
=tpcq+2tmem+tRFread+mux+tRFsetup



では、このCPUの性能を見積もってみましょう。シングルサイクルマイクロアーキテクチャは、全ての命令を1クロックサイクルで終わらせるので、最も遅延時間の長い命令の遅延を調べれば動作周波数が分かります。CPI=1なので、性能は動作周波数で決まります。ではどの命令の遅延パスが一番長いでしょうか？それはALUで実効アドレスを計算して、これでデータメモリを読み出すlw命令です。最も長い遅延パスをクリティカルパスと呼びます。これは図に示すようになります。

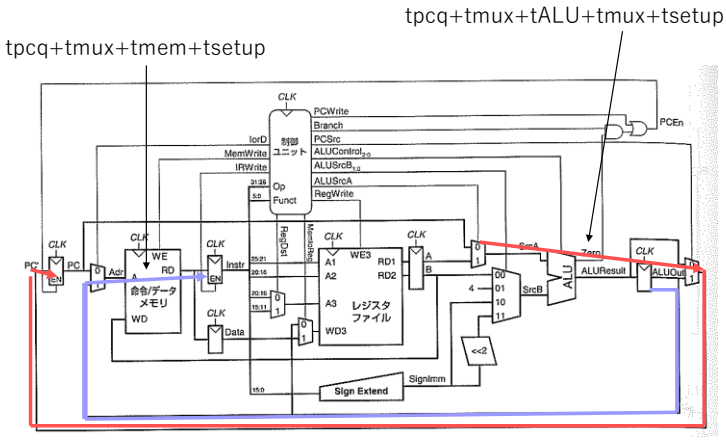
遅延の例

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+20=925psec

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925psecとなり、1.08GH z で動作することがわかります。

マルチサイクル版の
クリティカルパスの検討



マルチサイクル版のマイクロアーキテクチャでも、最も長いパスがクリティカルパスになります。これはシングルサイクル版よりも短くなります。データパスを検討すると、この2つのパスの辺になりそうです。

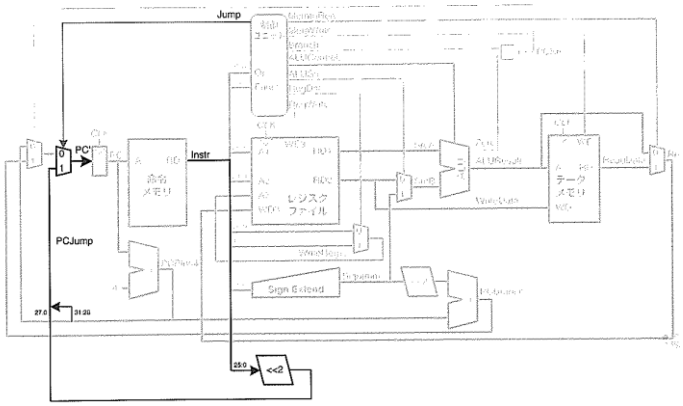
性能解析

- 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+20=325ps
- 平均CPI
  - 25%ロード、10%ストア、11%分岐、2%ジャンプ、52%R型命令とすると
  - 0.25×5＋（0.51+0.1)×4＋(0.11＋0.02)×3＝4.12
- 325×4.12＝1339
- これはシングルサイクルの925に比べて完敗である
- 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。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。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。その代わり、単一のメモリを命令とデータの両方に使える分、ハードウェア量は少なくて済みます。

コストの計算：シングルサイクル版

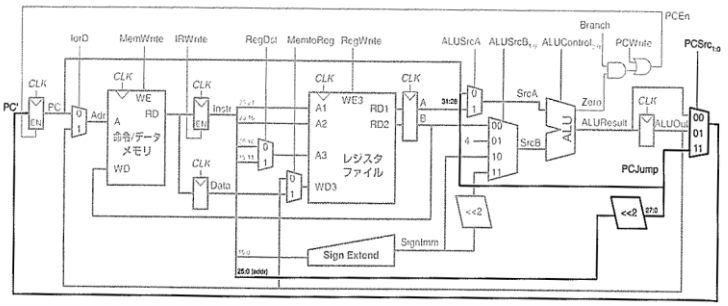
モジュール	個数
メモリ	2
レジスタファイル	1
ALU	1
加算器	2
マルチプレクサ	5
レジスタ	1



ではコスト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。

コストの計算：マルチサイクル版

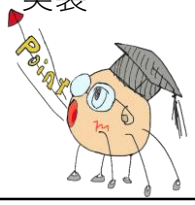
モジュール	個数
メモリ	1
レジスタファイル	1
ALU	1
加算器	0
マルチプレクサ	6
レジスタ	6



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、マルチプレクサが増え（入力数も増えています）、レジスタも増えています。とはいえ、マルチプレクサ、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　性能評価

- 0x1000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
- この値と授業中のスライドの数値を利用して、シングルサイクル版mipseとマルチサイクル版のmipseの性能を比較せよ。

XXがYYのZZ倍速い、という言い方で示せ。

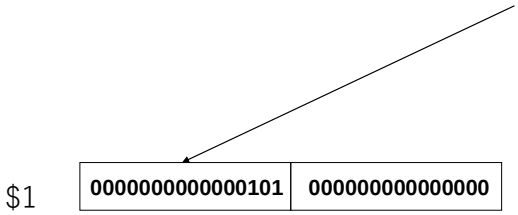
最初は楽勝です。

演習２　luiを実装せよ

上位16bitに直値を設定する命令

lui（opcode: 001111）

- 下位は0にする、rsの位置は0になる
- lui $1,5　001111_00000_00001_0000000000000101



luitst.asmを実行して結果を確認せよ　make luitstでOK
$1が55550000になっていればOK
提出物 luiの付いたmipse.v (mipsek.v)

次はluiの実装です。状態を増やす人は、mipsek.vを利用した方がいいかもしれません。

演習3　jr命令を実装せよ。

- jr rs　　000000_sssss_000000000000000_001000
- def.h中に定義はできている
- jrtstを使ってテスト　$2が0x5555になればOK
- make jrtstでアセンブルできる
- 提出物はjrの付いたmipse.v(mipsek.v)

最後はjrを実装する課題です。