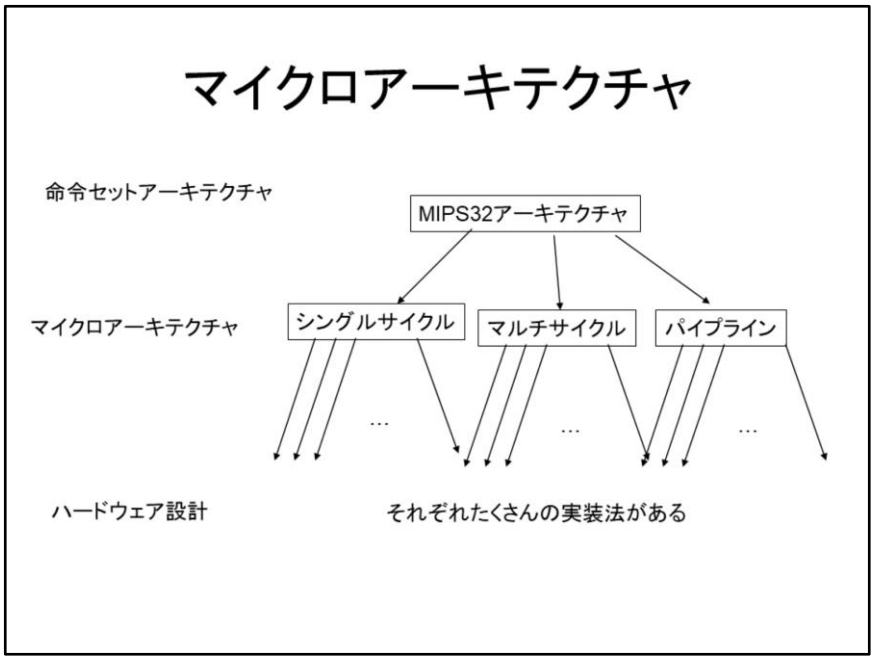
計算機構成　第6回
MIPSeのマイクロアーキテクチャ

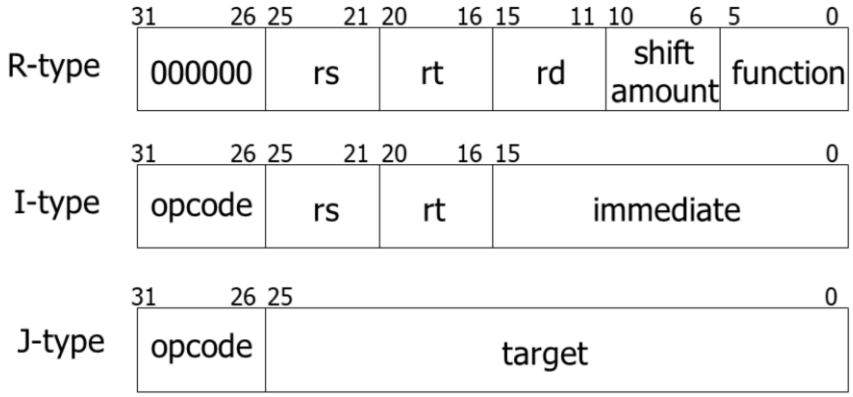
天野　hunga@am.ics.keio.ac.jp



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。ここでは、MIPSのマイクロアーキテクチャを紹介します。まずは、一番簡単なシングルサイクル実装を紹介しましょう。

命令フォーマット

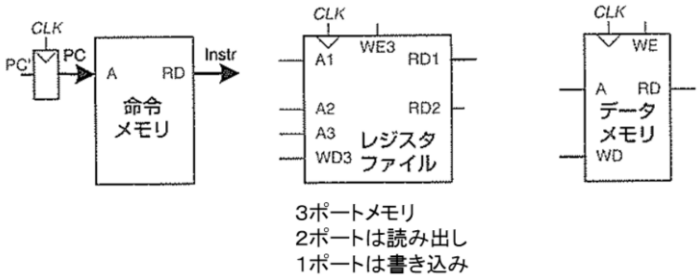
- 3種類の基本フォーマットを持つ



MIPSも3種類の命令フォーマットを持ちます。opcodeは6ビット、各レジスタは5ビットで示します。functionフィールドを使ってR型の命令を判別します。MIPSの場合さらにフィールドが余るので、これはシフト命令のシフトするビット数を指定するフィールドとして使っています。もう一つ混乱するのは、レジスタの位置がアセンブラで書く位置と逆になっている点です。これはrs,rtをR型とI型で統一するために必要です。rtはI型ではディスティネーション、R型ではソースレジスタとして使われますので注意が必要です。

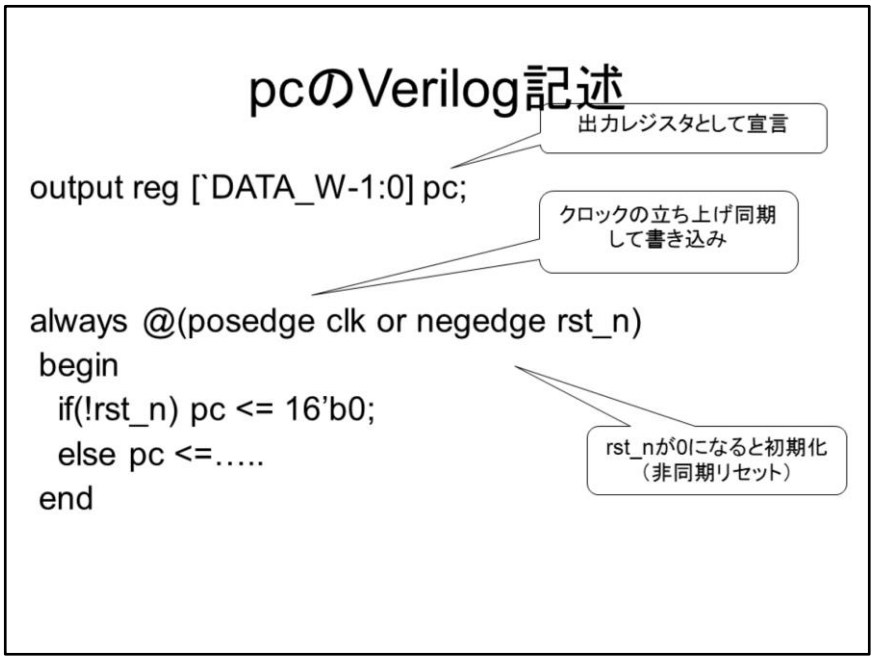
シングルサイクル
マイクロアーキテクチャ

- 状態要素は以下の3つ
- まずPCを命令メモリにつないで命令フェッチを実現

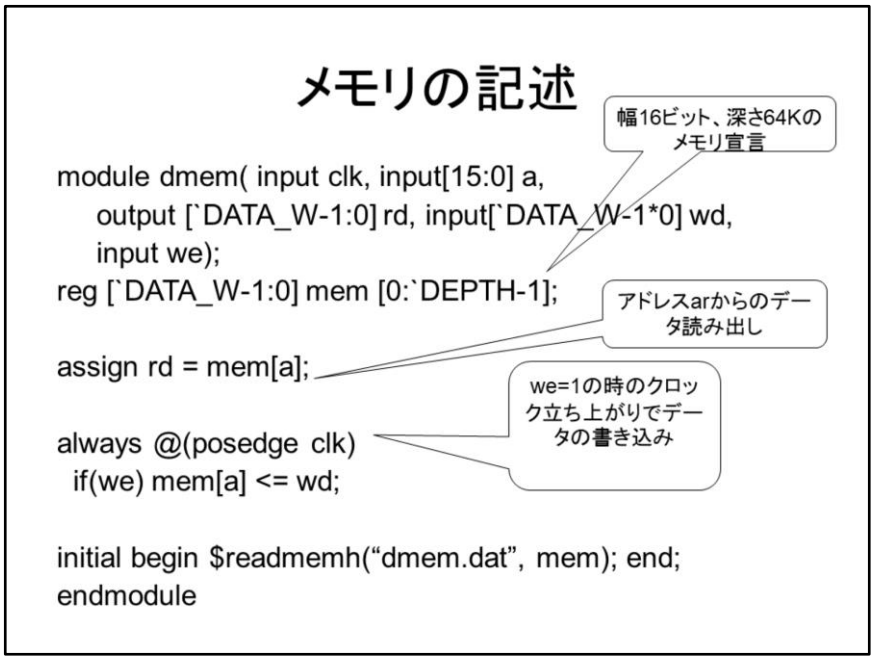


3ポートメモリ
2ポートは読み出し
1ポートは書き込み

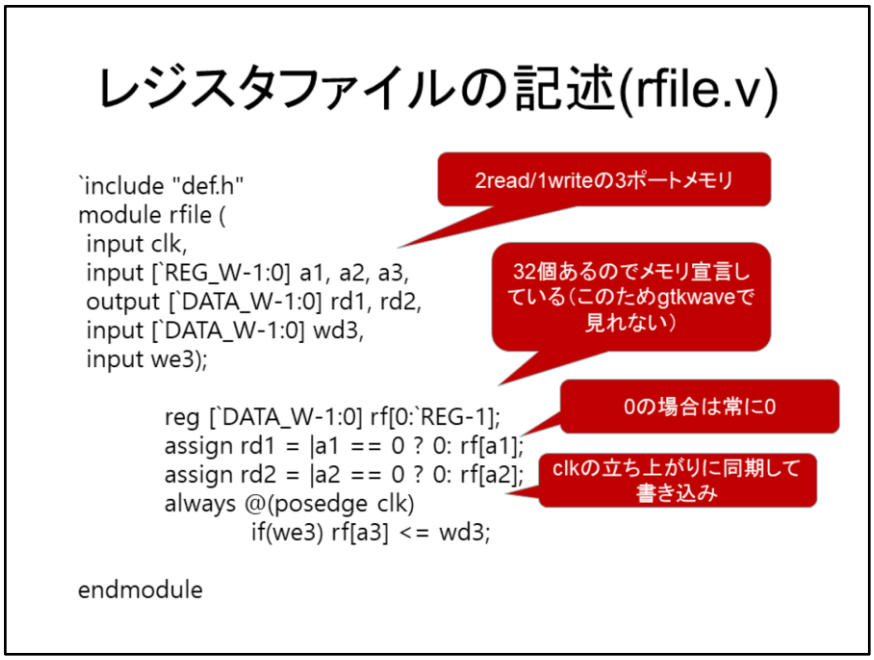
今回、Harris&Harris風に、状態要素を定義して、これを接続しながらデータパスを作って行きましょう。状態要素とは、本質的なCPUの状態を保持するハードウェアを指します。MIPSの場合、命令メモリ、はレジスタファイル、データメモリがそれに当たります。命令メモリ、データメモリはそれぞれ32ビットのアドレス空間を持ちます。空間のすべてを実際のメモリで埋める必要はないです。演習ではずっと小さいメモリを使います。データメモリはWE=1で、WDに与えたデータが次のレジスタファイルも2ポート読み出し、1ポート書き込みが同時に可能な3ポートメモリです。32ビットレジスタが32本入ります。A1から入れた番号に相当するレジスタはRD1から出力され、A2から入れた番号に相当するレジスタはRD2から出力されます。WE３＝１の時にA3から与えた番号のレジスタにWD3から入れたデータが書き込まれます。



まずpcの記述を解説しましょう。宣言は出力レジスタとして行っています。これは命令メモリに接続するためです。always文を使った普通のレジスタ記述で、クロックの立ち上がりrst_nが0になると0にクリアされます。すなわち非同期レジスタです。else以降は後程説明します。

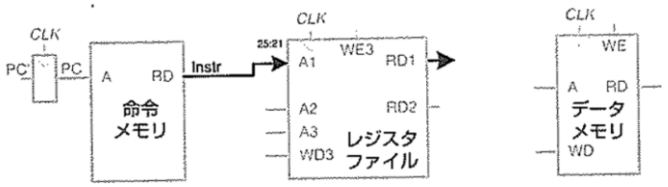


次はメモリの記述について復習します。今回は命令メモリはimem.vにデータメモリはdmem.vに記述してあり、テストベンチtest_mipse.vから呼ばれます。ここでは、書き込み機能を含んでいるdmemの方を解説します。メモリの宣言は、reg [MSB:LSB]「最小アドレス:最大アドレス]で行います。ここでは、16ビットで深さが65536のメモリが宣言されます。最小アドレスは0、最大アドレスは2のn乗にするのが普通です。MIPSは32ビットのアドレス空間(つまり3G)を持ちますが、演習ではそんなには使わないので、今回は命令、データ共に64Kとしました。このためアドレスは16ビットです。メモリはC言語の配列に似ているので、配列同様[ ]の中に番地を入れて値を取り出します。書き込む場合は、if(we)としてwe=Hの時だけclkに同期して入力を書き込みます。メモリにはファイルに書いてある初期値をあらかじめ設定することが出来ます。このための構文がreadmemh, readmembです。ここでは、dmem.datから16進数で初期値を読み込みます。同様にimem.vも定義されていますが、書き込み機能は持っていません。初期値はimem.datからやはり16進数で入力します。



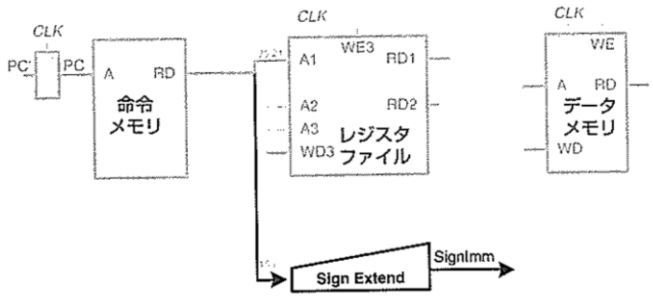
では次にレジスタファイルの記述を示します。レジスタファイルは、32本のレジスタを蓄えておくところで、2read/1writeの3ポートメモリとして記述してあります。ポート名は図に準拠してあります（ただし小文字です）。ポート1，2が読み出し、ポート3は書き込みです。rd1,rd2からはa1,a2で選んだ番号のレジスタが読み出されます。レジスタ0からは常に0が読まれるようになっています。ここではクロックと関係しないので、条件選択文が使われています。書き込みはwe3が1の時だけ、wd3の値をa3で選んだ番号のレジスタに書き込みます。

- まずlw命令を実現しよう
  lw rt,X(rs) ここでrsは25:21
- 命令の25:21をレジスタファイルのAポートのアドレスに繋ぐ



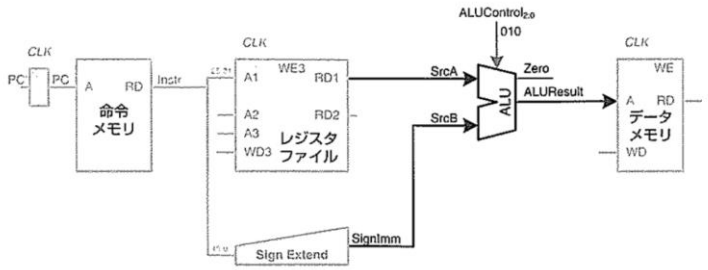
まずlw命令を実現できるデータパスを作って行きましょう。PCを命令メモリのアドレスに繋ぎ、出力のうち25:21、すなわちrsの入っているフィールドをレジスタファイルのAポートのアドレスにつなぎます。これによりrsの中身がRD1から出てきます。

- lw rt,X(rt) ここでXは15:0これを符号拡張する必要がある



命令の下位16ビット(15:0)がディスプレースメントXに当たります。これを符号拡張します。符号拡張はMSBを並べてくっつければよいのでハードウェアとしては簡単です。

- 実効アドレスはrsの内容＋Xの符号拡張
- これをデータメモリのアドレスに繋ぐ



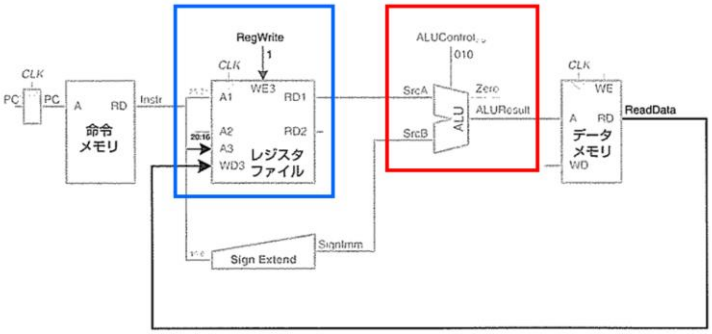
さてディスプレースメント付きレジスタ間接指定は、符号拡張したXと読み出したレジスタの中身を加算する必要があります。これをALUで行います。ALUControlには加算のコードを入れます。計算した実効アドレスをデータメモリのアドレスにつなぎます。

ALUの記述

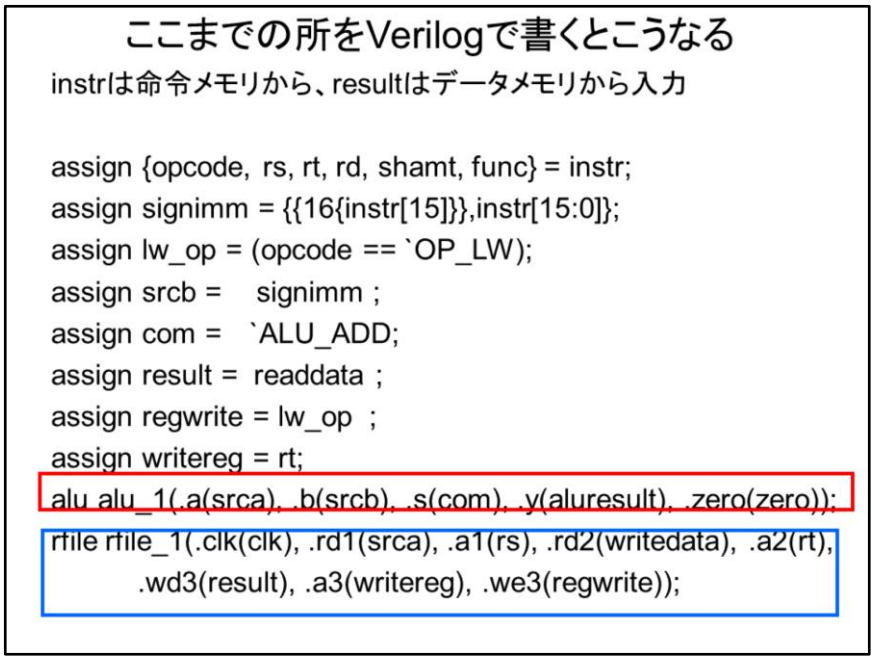
```
`include "def.h"
module alu (
  input [`DATA_W-1:0] a, b,
  input [`SEL_W-1:0] s,
  output [`DATA_W-1:0] y,
  output zero );
  assign y = s==`ALU_ADD ? a+b:
          s==`ALU_SUB ? a-b:
           s==`ALU_AND ? a & b:
           s==`ALU_OR ? a | b:
           s==`ALU_XOR ? a ^ b:
           s==`ALU_NOR ? ~ (a | b): b;
  assign zero = (y == 32'b0);
endmodule
```

ここでアドレス計算用にALUを導入します。この記述はアキュムレータマシン用に設計したものとあまり変わりません。計算結果がゼロになると、出力zeroが1にtなる点が違っています。オペレーションはMIPSに合わせたあります。

- lw rt,X(rs)　読み出した結果はrtに格納
- rtは20：16で指定
- we3を1にしてレジスタファイルに書き込み



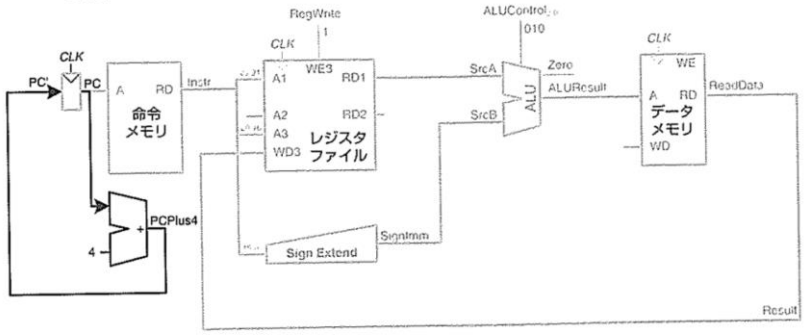
読み出したデータは、rtに格納する必要があるので、レジスタのWD3に送ります。rtは20：16で指定されているので、これをレジスタファイルの書き込みアドレスA3につなぎます。we3を1にしてこの結果をレジスタファイルに書き込みます。これでlwの機能は実現されます。



赤枠、青枠が全ページの赤枠、青枠に対応します。

- PCに4を足して次の命令をフェッチする

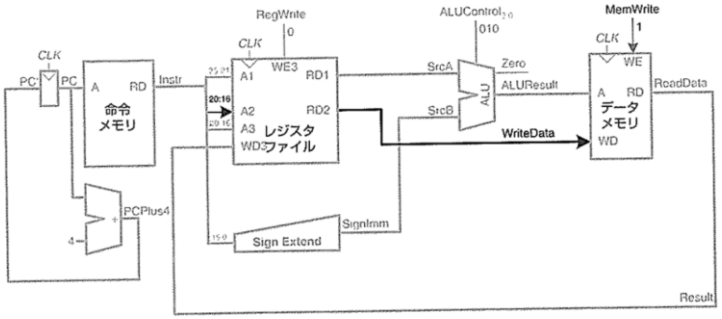
```
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else
    pc <= pcplus4;
end
```



実行したら、次の命令をフェッチしなければならないので、PCに4を足します。足した結果をPCに戻します。lwの結果をレジスタファイルに書き込んだのと同じクロックの立ち上がりでPC＋４がPCにセットされます。

swの実装

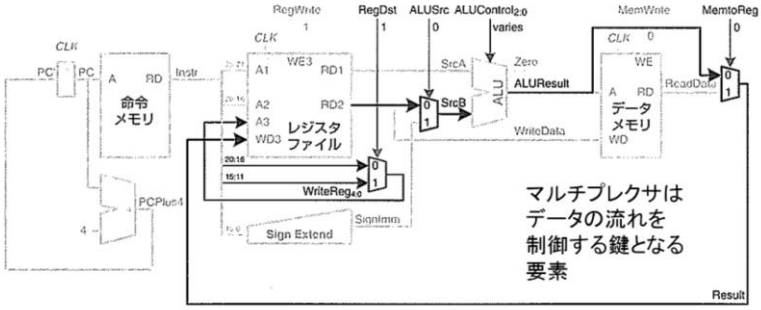
- sw rt,X(rs)　rtは20：16だが今度は書き込みデータの入ったレジスタを指定する。



では次はswを実装しましょう。実効アドレスの部分はlwと同じですが、swでは書き込むデータをしまったレジスタがrt,すなわち20:16のフィールドで指定されます。これをA2に入れて、書き込むデータをRD2から取り出します。このデータをデータメモリの書き込み入力WDにつないで出来上がりです。

R型命令の実装

- add rd,rt,rs rt,rsを指定するビットは今までと同じで25:21
- ALUのB入力にはrtを選択20:16で共通→符号拡張されたXとマルチプレクサで切り替え
- データメモリは飛越す→マルチプレクサ
- 計算結果はレジスタファイルに書き込む
- 書き込みのレジスタは15:11→マルチプレクサ



次にR型命令を実装してみます。add rd,rs,rtのうち、rsを指定するビットは今までと同じで25:21です。ここで指定されたデータはRD1から出てきます。また、rtは20:16で指定され、以前同様読み出されたデータはRD2から出てきます。この結果を加算しないといけないので、lw/swの際のディスプレースメントと切り替えるためにマルチプレクサを使います。これはALUSrcという信号で切り替えます。答はそのままレジスタファイルに書き戻すので、マルチプレクサを付けてlw命令で取ってきた値と切り替えます。このマルチプレクサはMemoryRegという名前の信号線で切り替えます。さらに、書き込み用のレジスタはrdで15:11のフィールドで示します。lwでは、20:16のrtを使っていたので、マルチプレクサで切り替えてやる必要があります。この切り替え信号がRegDstです。

ここまでの所をVerilogで書くとこうなる

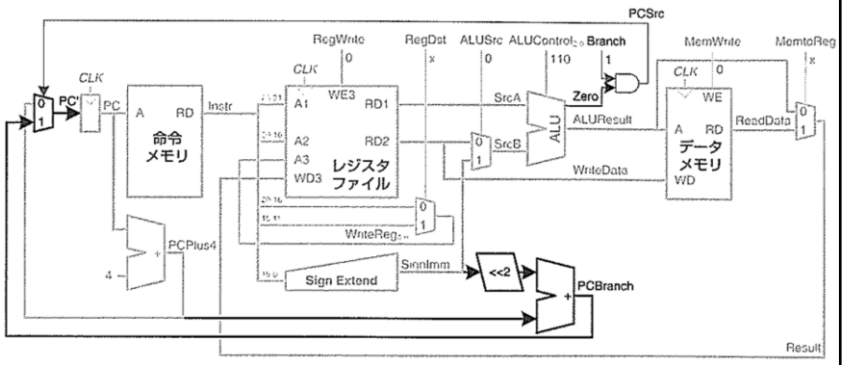
```
assign {opcode, rs, rt, rd, shamt, func} = instr;
assign signimm = {{16{instr[15]}},instr[15:0]};
assign sw_op = (opcode == `OP_SW);
assign lw_op = (opcode == `OP_LW);
assign alu_op = (opcode == `OP_REG) & (func[5:3] == 3'b100);
assign srcb =  (lw_op|sw_op) ? signimm:writedata ;
assign com =   (lw_op|sw_op) ? `ALU_ADD: func;
assign result = lw_op  ? readdata : aluresult;
assign regwrite = lw_op | alu_op ;
assign writereg = alu_op? rd: rt;
alu alu_1(.a(srca), .b(srcb), .s(com), .y(aluresult), .zero(zero));
rfile rfile_1(.clk(clk), .rd1(srca), .a1(rs), .rd2(writedata), .a2(rt),
        .wd3(result), .a3(writereg), .we3(regwrite));
```

マルチプレクサで拡張

ORで拡張

beq命令の実装

- beq rt,rs,飛び先
- 飛び先計算用の加算器が必要
- とぶかどうかはALUで引き算を行い出力で判断
- PCの入力をマルチプレクサで切り替える



では、次はbeq命令を実装します。ここでは、レジスタ演算命令と同じ指定で、ALUを使ってrt,rsを引き算して、Zeroが出るかどうかを調べます。Branch命令でZeroが出たことを検出し、PCSrcという信号を生成し、PC+4と飛び先（PCBranch)を切り替えます。Xを符号拡張したものを2ビット右シフトし、PC+4と加算します。これは専用の加算器を使い、飛び先番地PCBranchを生成します。

分岐命令の実装

```
assign beq_op = (opcode == `OP_BEQ);
assign bne_op = (opcode == `OP_BNE);
assign com = (lw_op|sw_op) ? `ALU_ADD:
                    (beq_op | bne_op) ? `ALU_SUB: func;
assign pcplus4 = pc+4;
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else if ((beq_op & zero) | (bne_op & !zero))
    pc <= pcplus4 +{signimm[29:0],2'b0} ;
  else
    pc <= pcplus4;
end
```

分岐命令の場合、ALUには引き算をやらせます。このためALU_SUBをコマンドに入れます。これで2つの入力が等しければzero=1となります。
その他の部分はpcの制御なので、pcのalways文の拡張で対処できます。飛び先の計算は符号拡張したものを2ビット左シフトして求めています。条件が成立した場合にこれをpcにセットします。そうでなければ今まで通り4を足して次に進めます。

例題　addi命令の付加

- まずデコード信号addi_opを定義する。
- 以下を条件選択文の条件に追加する。
  - ALUのB入力srcbにsignimm（イミーデエイトを拡張したもの）を入れる
  - ALUのcomは加算にする
- regwriteにORして追加する
- writereg, resultはデフォルトが使えるので修正なしでOK
- 命令追加の手口
  - 条件選択文の改造
  - regwriteにOR
  - 分岐命令ならば、pcのalways文の改造

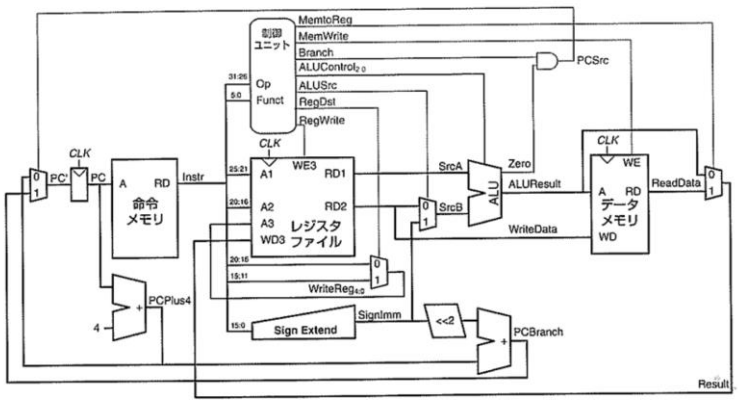
では、この設計にaddi命令を追加してみましょう。まず最初にこの命令をフェッチした時に1になる信号addi_opを定義します。addiはI型なので、opcodeを比較することで識別します。次にこのaddi_opを条件選択文に追加して、ALUのB入力に当たるsrcbにはイミーディエイトを拡張したsignimmが入るようにします。同様に、ALUのcomには加算を指定します。ALUのA入力にはレジスタrsの値が入りますので、これでレジスタとイミーディエイトの加算が実現します。答をレジスタファイルに書き込むために、regwriteのop_addiを論理和（OR)します。writereg, resultはデフォルトの入力であるrtとaluresultを使うので、変更は必要ないです。
このように命令を追加するには、演算系の命令ならば、ALUの入力や書き込みレジスタの条件選択文を改造し、答をレジスタファイルに書き込む命令の場合はregwriteにデコードした信号をORしてやります。分岐命令は、pcのalways文を修正します。

変更点

```
wire addi_op;
assign addi_op = (opcode == `OP_ADDI);
assign srcb = (addi_op | lw_op | sw_op ) ?
                                signimm : writedata;
assign regwrite = lw_op | alu_op | addi_op ;
```

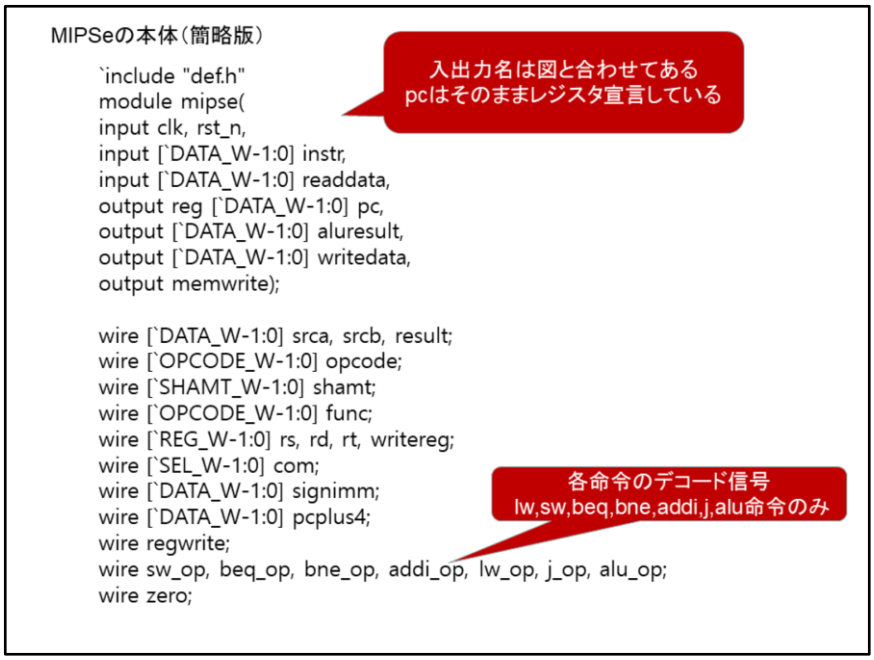
前のページの変更点を具体的にVerilogHDLの記述に直すとこのようになります。それぞれの行の意味を理解してください。

制御ユニットをつけた全体図



分岐命令を入れた所までをとりあえず全体図にまとめてみます。Verilog HDLでは制御ユニットの制御線が条件選択文や論理和で表わされていることが分かります。

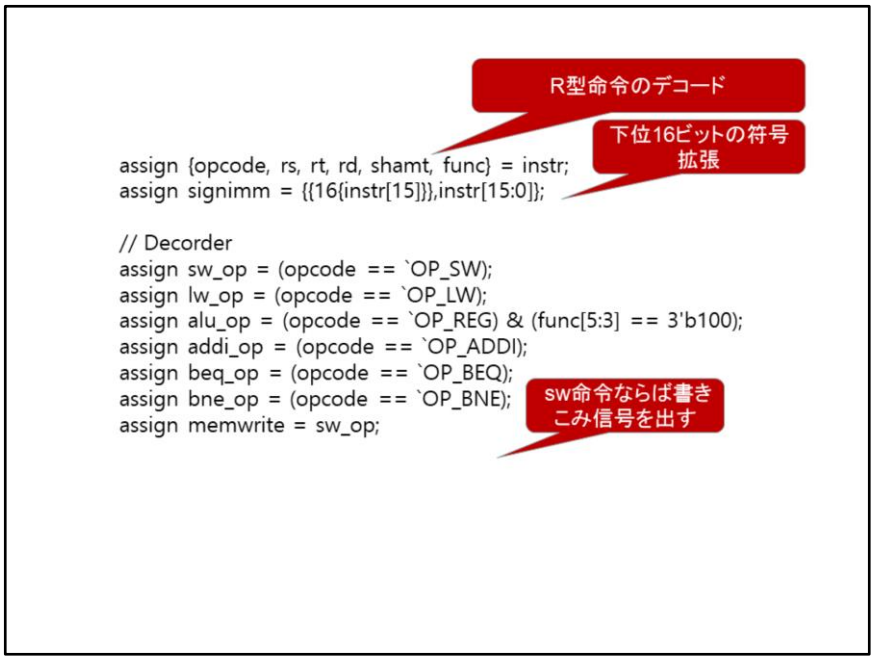
MIPSeの本体（簡略版）



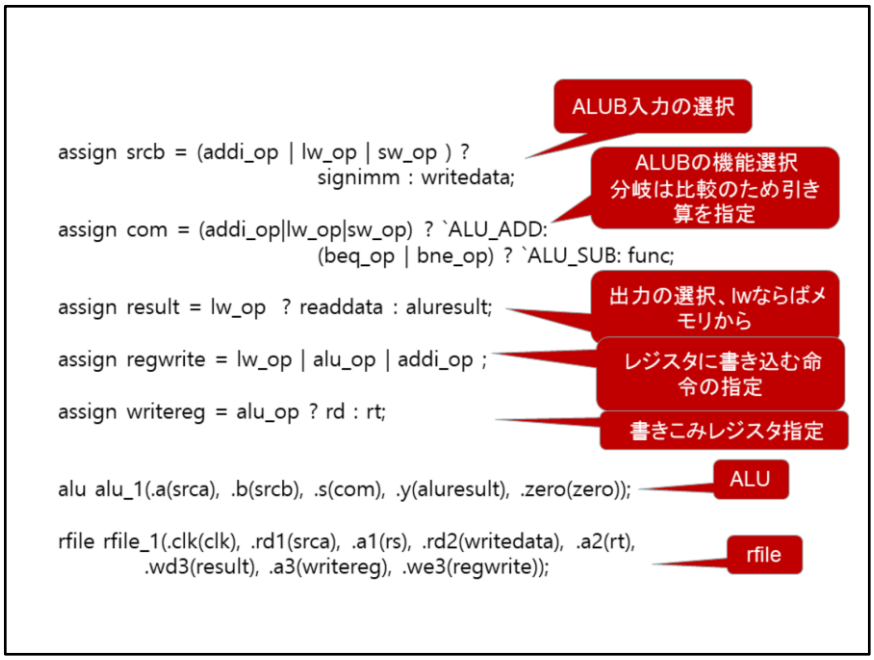

```
`include "def.h"
module mipse(
input clk, rst_n,
input [`DATA_W-1:0] instr,
input [`DATA_W-1:0] readdata,
output reg [`DATA_W-1:0] pc,
output [`DATA_W-1:0] aluresult,
output [`DATA_W-1:0] writedata,
output memwrite);

wire [`DATA_W-1:0] srca, srcb, result;
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;
wire sw_op, beq_op, bne_op, addi_op, lw_op, j_op, alu_op;
wire zero;
```

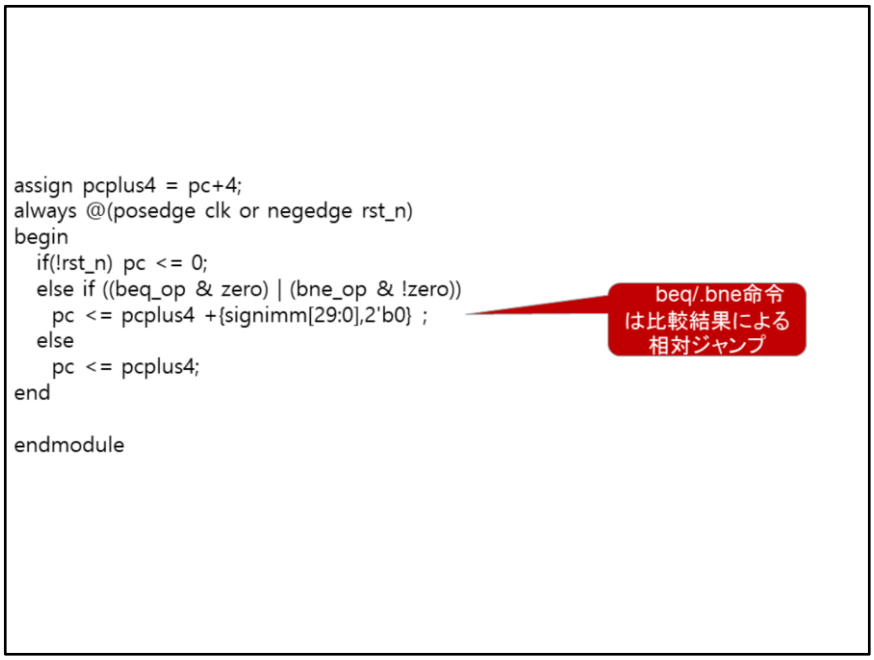
では、MIPSeの本体をまとめて説明しましょう。入出力名、信号名は図と合わせてありますが、すべて小文字に統一しています。プログラムカウンタPCは出力部でレジスタ宣言しています。図に出ていないのは各命令のデコード信号です。それぞれの記述を図と対応させて理解しましょう。



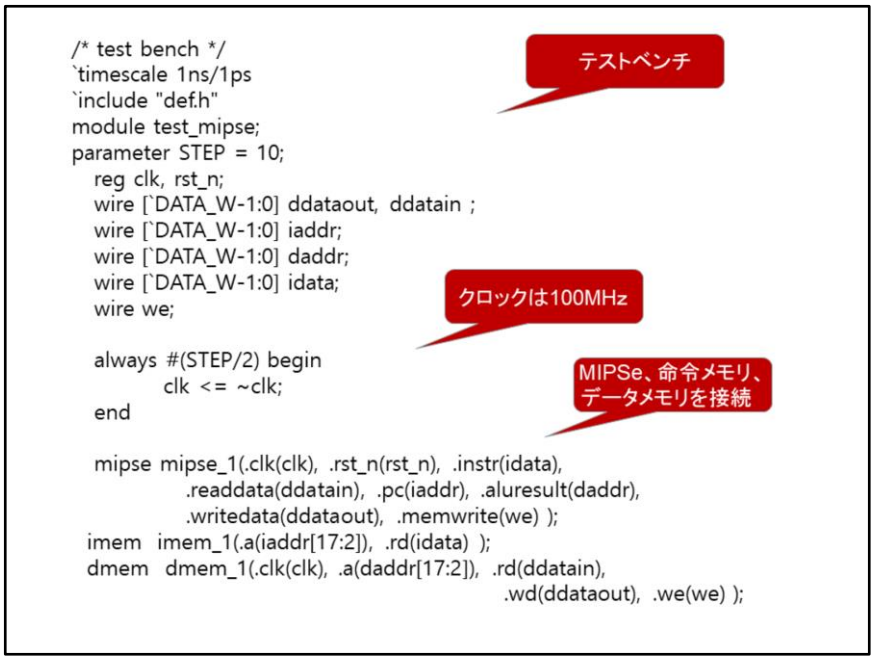
命令のデコード部分です。デコード信号は図にないので注意ください。連結構文{}の使い方、符号拡張の記述を思い出してください。



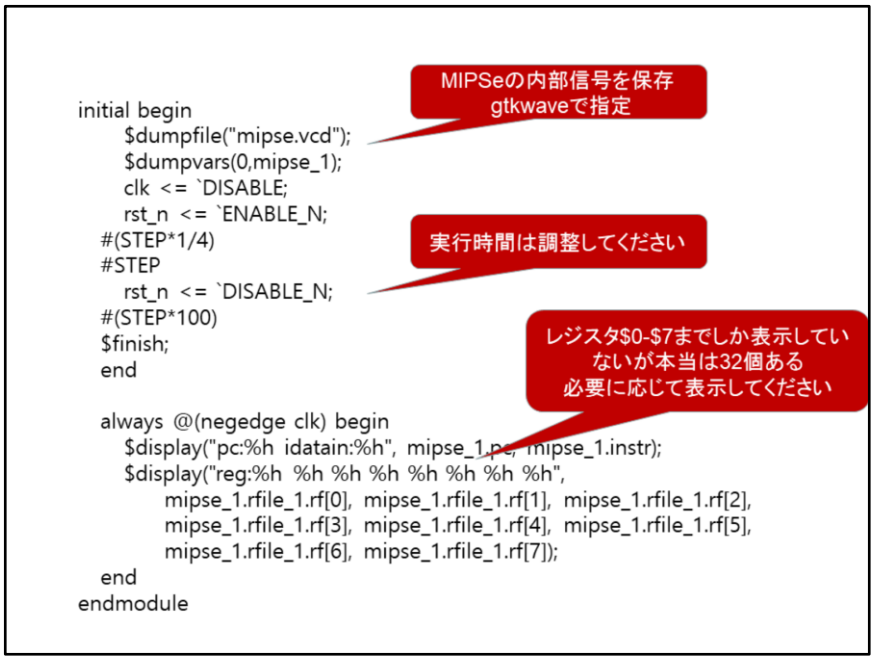
各部のマルチプレクサの制御、ALU、rfileの接続です。これも図と照らし合わせましょう。



分岐命令の記述です。分岐命令は、PC+4に対して符号拡張を行った値を2ビット右シフトして足してやります。



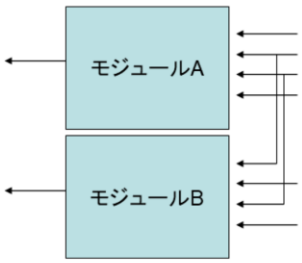
最後にテストベンチの解説です。クロックは１００MHzにしています。mipseと
imem,dmemを接続しています。アドレスは16ビット分を2ビットずらしてつなぎます。



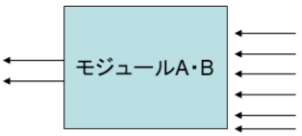
テストベンチの後半です。波形をgtkwaveで見れるように波形ファイル（vcdファイル）を作ります。命令が一つずつ実行されるのを見るためにプリントもしていますが、レジスタは実は
全部は表示していません。ここでの例ではメモリも表示していません。これは必要に応じて変えてください。

この授業の記述の特徴

- 出力信号依存の書き方
- 全ての出力を分離して記述している



AとBに共通性があっても
分離して書かなければならない



まとめて書ければ可読性が向上する
（かもしれない）
でもやらない

ここでの授業では、最も基本的な「入門スタイル」を使っています。この詳細はWebをご覧ください。この書き方は全ての出力を分けて書くのでやや見にくいですが、間違いが減ります。

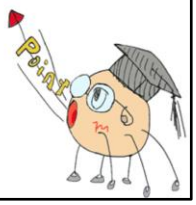
例題

- mult.asmをシミュレーションし、gtkwaveを用いて各信号の動きを確認しよう。

では、mult.asmの実行状況をシミュレーションして、gtkwaveを用いて各信号の動きを見てみましょう。

本日のまとめ

- Verilog HDLの記述を思い出そう！
- 信号線名は図と一致しているので、図を見ながら動作を追って行こう。
- gtkwaveを使って各部の信号を見て行こう。



インフォ丸が教えてくれる今日のまとめです。Verilog記述を思い出してくださいませ。

演習1

ori rt,rs,X 命令を付け加えよ

- simpleディレクトリの
- Xはゼロ拡張する

opcode 001101 （実はdef.hに付いている）

oritst.asmでテストして確認すること

$1が7になっていればOK

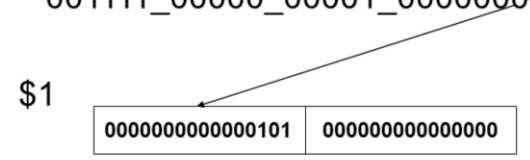
提出物：oriを付け加えたmipse.v

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
$1が00050009になっていればOK
提出物 luiの付いたmipse.v(oriが付いていてもOK)

前回軽く紹介したlui命令を実装してみましょう。