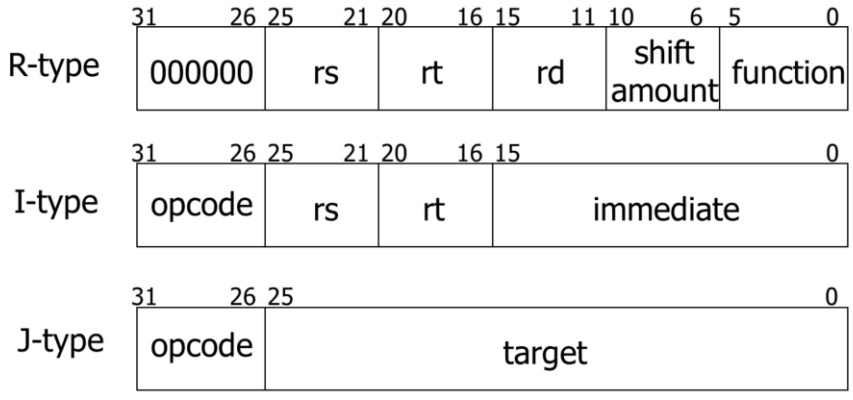
コンピュータアーキテクチャ B
MIPSのマイクロアーキテクチャ

情報工学科
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



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。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。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に注意してください。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。

現在使えるR型命令一覧

add rd,rs,rt	rd ←rs+rt	000000ssssstttttddddd100000
sub rd,rs,rt	rd ←rs-rt	000000ssssstttttddddd100010
and rd,rs,rt	rd ←rs&rt	000000ssssstttttddddd100101
or rd,rs,rt	rd ←rs|rt	000000ssssstttttddddd101010
slt rd,rs,rt	rs<rt rd←1 else rd←0	000000ssssstttttddddd101010
jr rs	pc ← rs	000000sssss0000000000001000

では現在使える命令の一覧を示します。

I型命令一覧

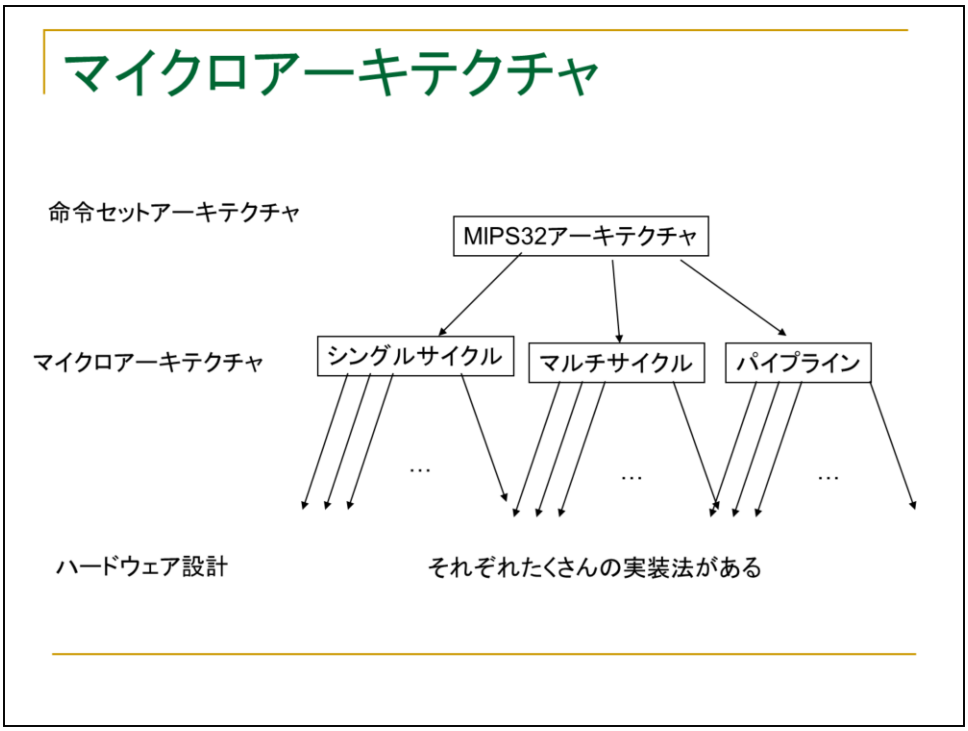
lw rt,offset(base)	ワードロード	100011tttttbbbbbb offset
sw rt,offset(base)	ワードストア	**101011tttttbbbbbb offset**
addi rt,rs,imm	rt←rs+(符号拡張)imm	001000ttttsssss imm
slti rt,rs,imm	rs<(符号拡張)imm rd←1else rd←0	**001010ttttsssss imm**
beq rs,rt, offset	rs=rtで分岐	**000100tttttssssss offset**
bne rs,rt, offset	rs≠rtで分岐	**000101tttttssssss offset**

lw, swはディスプレースメントを伴うため、I型になります。

J型命令一覧

j offset	不完全絶対分岐	**000010 offset**
jal offset	不完全絶対指定のサブルーチンコール $31←pc+4	**000011 offset**

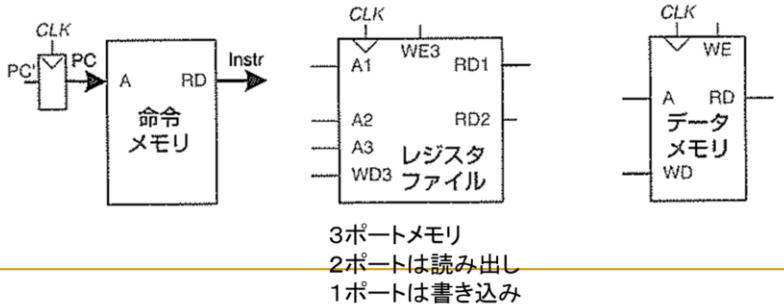
J型はjとjalのみです。



では次に実装法を復習をしましょう。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。計算機構成ではPOCOのシングルサイクル実装とマルチサイクル実装を紹介しました。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。

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



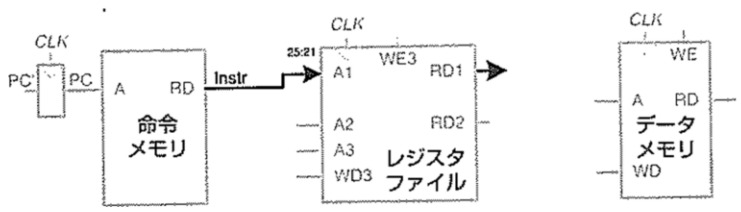
3ポートメモリ
2ポートは読み出し
1ポートは書き込み

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

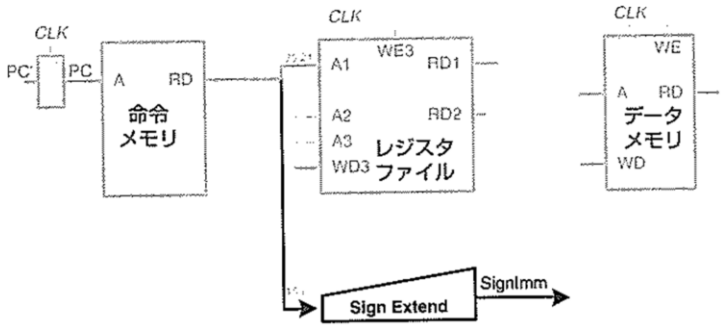
  lw　rt,X(rs)　ここでrsは25:21

- 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



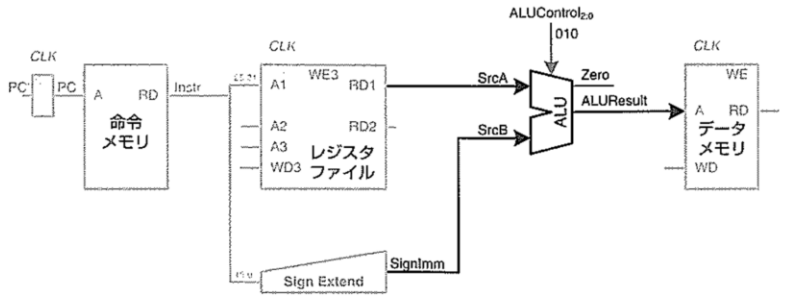
まずlw命令を実現できるデータパスを作って行きましょう。PCを命令メモリのアドレスに繋ぎ、出力のうち25:21、すなわちrsの入っているフィールドをレジスタファイルのAポートのアドレスにつなぎます。これによりrsの中身がRD1から出てきます。

- 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



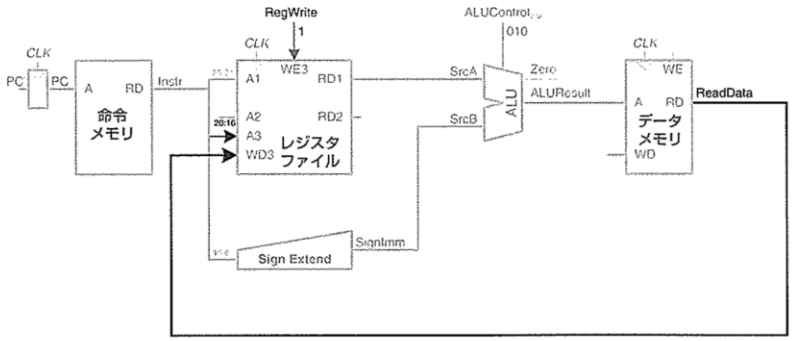
命令の下位16ビット(15:0)がディスプレースメントXに当たります。これを符号拡張します。符号拡張はMSBを並べてくっつければよいのでハードウェアとしては簡単です。

- 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



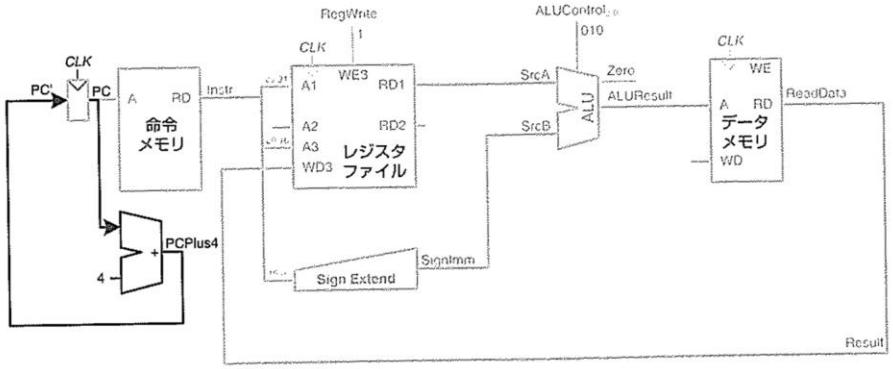
さてディスプレースメント付きレジスタ間接指定は、符号拡張したXと読み出したレジスタの中身を加算する必要があります。これをALUで行います。ALUControlには加算のコードを入れます。計算した実効アドレスをデータメモリのアドレスにつなぎます。

- 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。

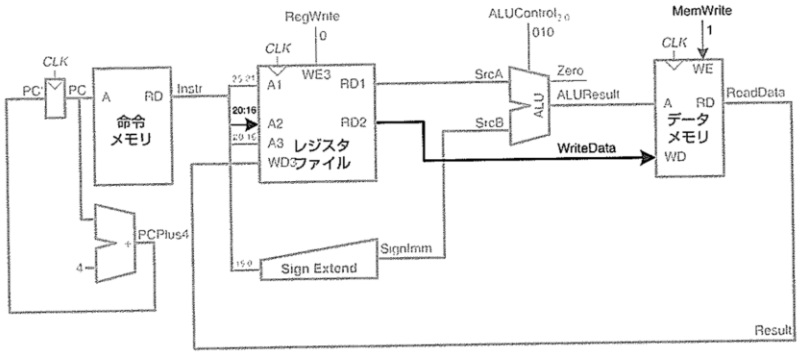
■ PCに4を足して次の命令をフェッチする



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

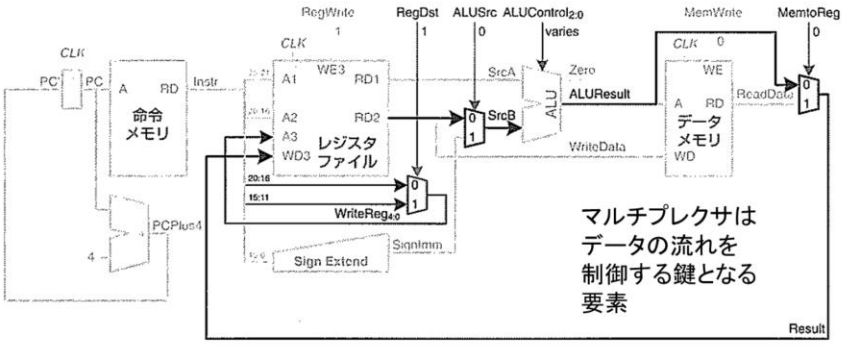
- sw rt,X(rs)　rtは20：16だが今度は書き込みデータの入ったレジスタを指定する。



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

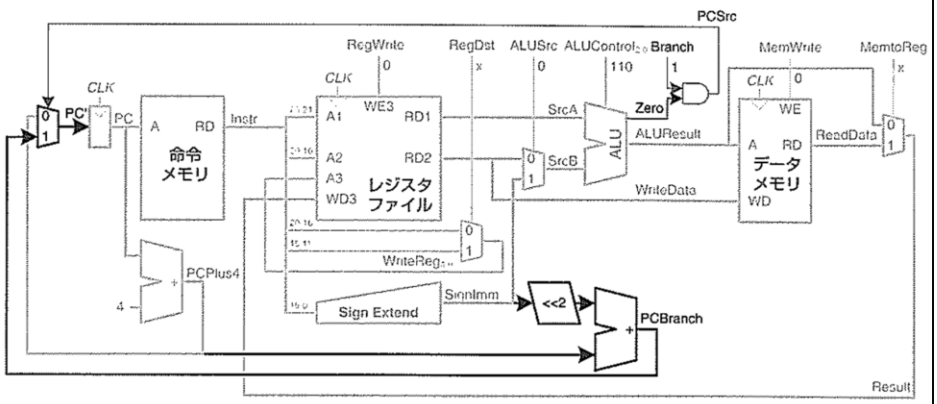
- add rd,rt,rs rt,rsを指定するビットは今までと同じで25:21
- ALUのB入力にはrtを選択20:16で共通→符号拡張されたXとマルチプレクサで切り替え
- データメモリは飛越す→マルチプレクサ
- 計算結果はレジスタファイルに書き込む
- 書き込みのレジスタは15:11→マルチプレクサ



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。

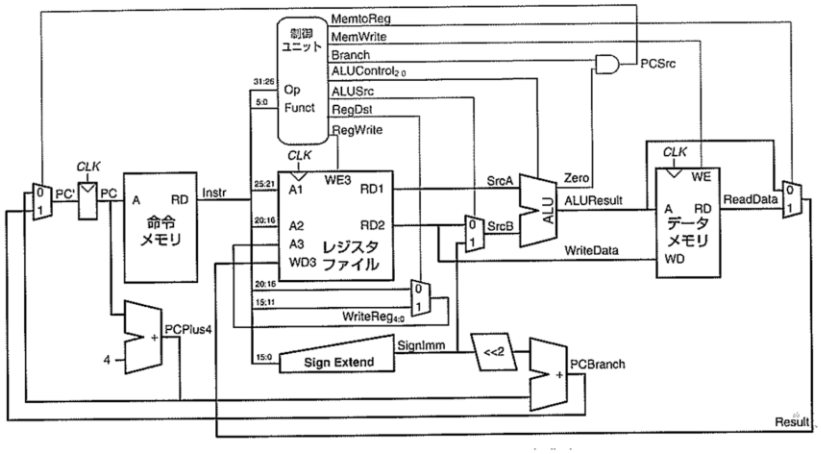
beq命令の実装

- 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



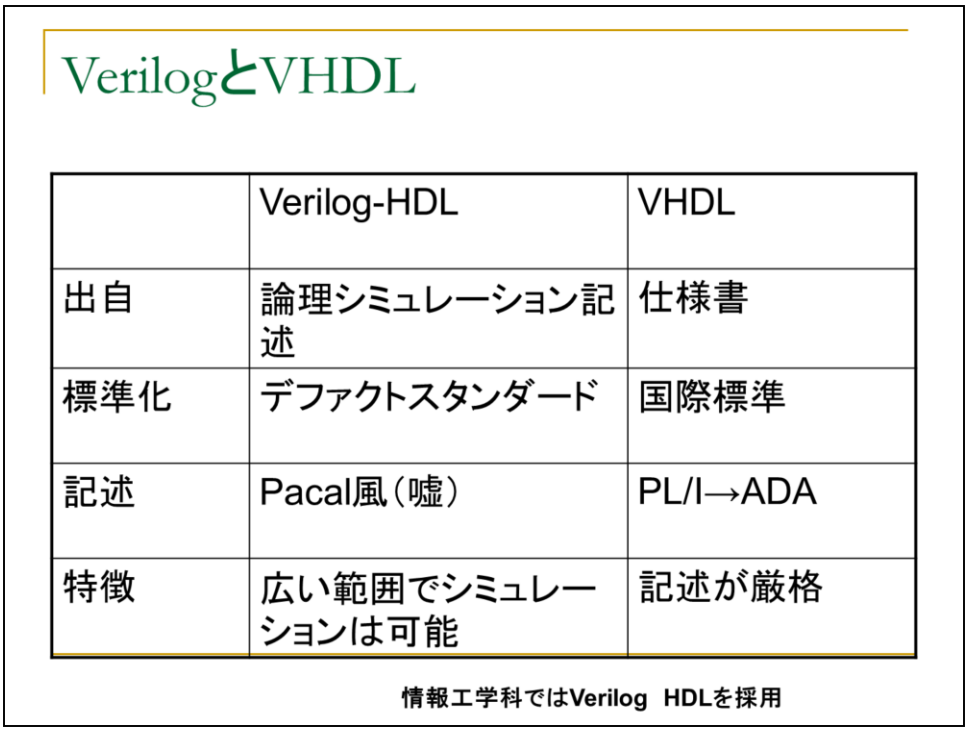
では、次はbeq命令を実装します。ここでは、レジスタ演算命令と同じ指定で、ALUを使ってrt,rsを引き算して、Zeroが出るかどうかを調べます。Branch命令でZeroが出たことを検出し、PCSrcという信号を生成し、PC+4と飛び先（PCBranch)を切り替えます。Xを符号拡張したものを2ビット右シフトし、PC+4と加算します。これは専用の加算器を使い、飛び先番地PCBranchを生成します。

制御ユニットをつけた全体図



さあ、これをVerilogHDLで記述しよう

では、今までの制御信号を制御する制御ユニットを付けた全体図を示します。さあこれをVerilog HDLで記述してみましょう。



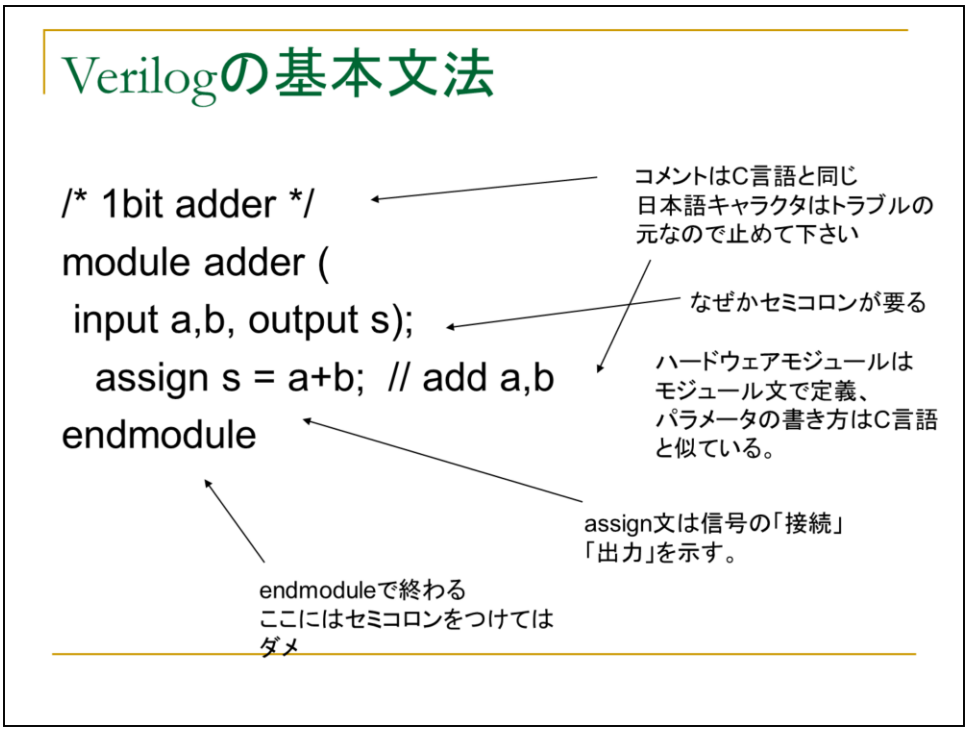
ここでVerilog HDLを復習します。Verilog HDLとVHDLは共に良く使われますが、その性格は全く違っています。

Verilog-HDLは論理シミュレーションを記述する言語として誕生し、使われているうちに標準化されたデファクトスタンダードです。構文（シンタックス）はPascal風と称していますが、ウソで、Pascalとは結構違った独自の構文です。シミュレーションが動作すれば、そのハードウェアの様子が分かるので、とにかく動作することが重要です。このため、Verilog HDLは、宣言などをいい加減に書いてもシミュレーション上は動作してしまいます。これはありがたいことなんですが、合成の段階までに問題点に気づかないと、とんでもないバグを含んだハードウェアを生成することになります。

一方、VHDLはハードウェアの仕様書記述用の言語が発達したもので、厳格な構文を持ちエラーチェックをきっちり行います。米国国防省が制定したPL/I、ADAの流れを汲む構文で、国際標準としてトップダウンに制定されました（PL/IとADAは、米国国防省がこれ以外の言語で書いたソフトウェアの納入を認めなかったため、プログラミング言語として一時期相当使われました。しかし、あまりの融通の利かなさに腹を立てたプログラマたちは、長年の苦闘の末にこれを絶滅に追い込みました）。記述が厳格なので、シンタックスエラーを修正する段階でバグをかなり減らすことができます。一方、簡単なハードウェアの記述にも多数の行数を要するので不便です。

本大学の情報工学科ではVerilog HDLを使います。僕は最初はVHDLを使っていた（使っていたCADがこれしか受け付けなかった）のですが、この言語があまり好きになれず、Verilog HDLに切り替えました。VHDLでハードウェアを設計していると、新しいシステムを設計するんだ、というクリエイティブなことをやっているのではなくて、わ

かりきったシステムの仕様書を書いているような気分になってくるんです。もちろん、なんで書こうとクリエイティブな設計はできるのですが、ま、気分の問題です。幸いにして最近のCADは両方を受け付けてくれますので、どちらかで設計できれば問題ありません。



Verilog HDLで書いた1ビットの加算器adder.vを示します。C言語のプログラムの拡張し.cにしたのと同様に、Verilog HDLのファイルの拡張しは.vにします。また、ファイル名はトップモジュール名（最上位階層のモジュール名、ここではadder）にします。エディタは皆さんの好きなのを使ってください。僕はviを使いますが、皆さんはemacsがお好きな方が多いかと思います。

さて、まず最初の行はコメントです。コメントの付け方はCと同じで/* */で囲むか、//の後に書くかどちらかです。ここで日本語を使いたくなる人が居ると思いますが、日本語のフォントがトラブルの元となるので止めてください。英語でコメントを付けましょう。さて、Verilog HDLは（VHDLも）、ハードウェアをモジュールという単位で階層的に記述していきます。この場合1ビットの加算器が1つのモジュールになります。Verilogの記述はモジュールの定義から始まります。

module文の後にモジュール名を書き、これに続く（ ）内に入出力端子を定義してやることでモジュールが定義されます。ここではadderがモジュール名で、inputの後のa,bが入力端子名、outputの後のsが出力端子名です。後に紹介する方法で指定しない場合は1ビットの端子として宣言されます。

これらはカンマで区切っていくつでも並べて書くことができます。input文、output文自体が複数出てきても問題ありません。ここでC言語との違いはなぜか、最後の）の後にセミコロンが必要な点です。

モジュールを定義したら、次からの文はそのモジュールの構造あるいは動作を書きます。ここではassign s=a+b;でa入力とb入力の加算結果をsに出力する、あるいはaと

bを加算器に入れた出力をsに接続する、という意味があります。Verilogでは単純に＝を使うことはできず、必ずassignを先に付けます。（これについては後で詳しく解説します。）文の終わりはC言語と同様にセミコロンを付けます。

このモジュールはこれで終わりなので、endmodule文でモジュールの終わりを宣言します。この文の終わりにはセミコロンを付けてはいけません。

条件演算子（マルチプレクサ構文）

```
assign Y =  (条件1)? 式1:
            (条件2)? 式2:
              ……
            (条件n)? 式n: 式n+1;
```

- 成立した条件に対する式がYに出力
- どれも成立しなければ式n+1がYに出力
- 先に書いた条件に優先順位がある
- この授業の書き方のルール
  - 条件は可能な限り排他的（どれかが成り立てば他は成り立たない）に書く
  - 式中に選択構文を使って入れ子にしてはならない
- 上記を守れば選択構文で全ての組み合わせ回路は分かりやすく書ける

  → 他にもfunction文やalways文を使った書き方があるのだがこの授業ではやらない

条件演算子は良く使います。これはassign文の右辺の書き方で、条件1？式1：条件2？式2：…：式n+1;の形で、成立した条件に想定した式がYに出力されます。この構文はハードウェアとしてはマルチプレクサを生成するため、マルチプレクサ構文と呼ばれる場合もあります。C言語に慣れた方はswitch文に相当することがご理解できると思います。条件が複数成立した場合、先に書いた方の条件が優先されます。しかし、できる限り条件は排他的に書くことをお勧めします。排他的とは、ある条件が成立したら、他の条件は成り立たない、つまりただひとつだけ条件が成立するという意味です。また、式n+1はデフォルト、つまりどの条件も成り立たない場合に出力されます。条件演算子は式n+1を書かないとエラーになります。

もう一つ注意したいのは、式の書き方です。式の中にさらに条件演算子を使うこともできるのですが、これは絶対に止めてください。条件演算子の入れ子を使うと非常に読みにくくなります。このような場合、信号線を新たに定義して、別の条件演算子を使ってください。

define文の利用

- なるべくコード中に直接数を書かないようにする
- 変更が容易
- ＃ではなく、バックシングルコーテーションを用いる

```
`define DATA_W 16
`define SEL_W 3
`define ALU_THA `SEL_W'b000
`define ALU_THB `SEL_W'b001
`define ALU_AND `SEL_W'b010
`define ALU_OR `SEL_W'b011
```

VerilogではC言語同様define文で定義することができるようになっています。C言語との違いは＃ではなくて、バックシングルコーテーションを用いることと、定義するときだけでなく、この定義をコード中で引用するときもバックシングルコーテーションを付けること、の2点です。このバックシングルコーテーションはシングルコーテーション（数字の桁を指定するときに使う）と紛らわしいし、キーボードによって位置がはげしく異なるため、困ったもんです。このためこれを嫌ってparameter文を使う人も居ます。

define文の利用

- シングルバックコーテーションで引用

```
module alu (
  input [`DATA_W-1:0] a,b,
  input [`SEL_W-1:0] s,
  output [`DATA_W-1:0] y);
  assign y = s==`ALU_THA? a;
             s==`ALU_THB? b;
             s==`ALU_AND? a&b: a+b;
endmodule
```

引用する方でもバックシングルコーテーションで引用します。

比較演算子

- 成立すれば1、そうでなければ0を返す
- 大小比較：< <= > >=
- 等号： == != === !==

  == !=は、x（不定）、z（ハイインピーダンス）が入力にあれば結果はxやzになる

  === !==は、x、zを含めて比較する
  - この授業で== !=のみを利用する

条件を記述するのに比較演算子を使います。これはC言語の比較とほとんど同じなのであまり問題はないでしょう。ただし、大小比較については、全て符号無しの数を想定して比較が行われますので、この点を注意してください。符号付数同士の比較は符号ビットを判断して判断しなければなりません。（結構めんどくさい）

リダクション演算

- 論理演算子をバスの前に書くとリダクション演算子となる
- 全ビットを演算し、結果は1か0の1ビットの値になる

A=4'b1001ならば

AND &A=0
OR |A=1
NAND ~&A=1
NOR ~|A=0

リダクション演算は、バスで定義された信号について、全てのビットに対して同一の演算を行って一つの結果を得る演算子です。例えばA=4'b1001で、＆Aと書くと、全ビットのAND、すなわち1＆0＆0＆1が演算され、答えは0となります。
A[3]&A[2]&A[1]&A[0]と同じなのですが、信号の名前の前に記号を書けばよいので、スマートに記述ができます。同じ方法でORやNOTを付けることもできます。Aが0かどうかを判別するのに、A==4'b0000とやる代わりに、～｜Aとやってもいいのです。
（でも格好だけで同じですが、、）

演算子の優先順位

論理否定	! (条件に対する否定） ~
乗除算	＊ / %
加減算	+ -
シフト演算	<< >>
比較演算	< > <= >=
等号	== != === !==
論理積	&
排他的論理和	^ ~&
論理和	|
論理積（条件）	&&
論理和（条件）	||
条件	? :

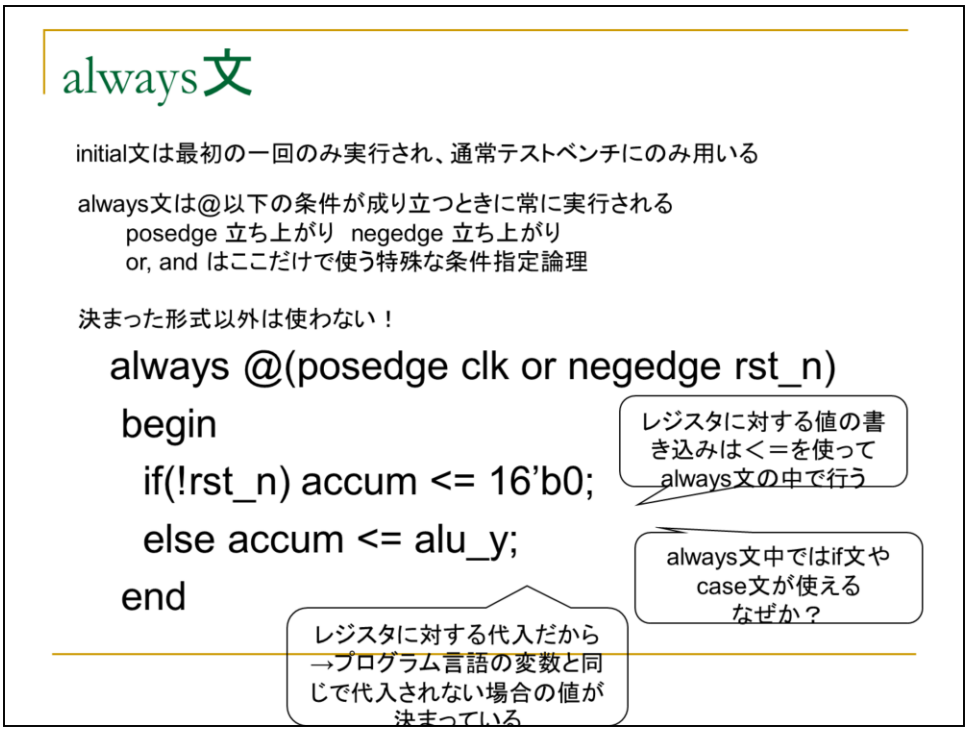
Verilogの演算子の優先順位を示します。割と普通の順序ですので自然に使えます。

ALUの記述(alu.v)

```
`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
```

では、MIPSeのALUのVerilog記述を見てみましょう。条件選択（マルチプレクサ）構文に注目してください。



ではalways文を復習しましょう。テストベンチにでてきたinitial文がシミュレーション時に一回のみに実行されるのに対して、always文は＠以下の条件が成り立つ際に常に実行されます。条件内のposedgeはLからHへの変化（立ち上がりエッジ）、negedgeはHからLへの変化（立下りエッジ）を示します。orはalways文の条件にだけ使う特殊な記法で、条件のどちらかが成り立ったときに（）内全体が成り立ったと見なします。この場合、clkがLからHに変化するか、rst_nがHからLに変化した時にalways文の条件が成立してbegin .. end内が有効になります。begin..endはC言語の{　}に相当し、複数の文をまとめて一つの構文とします。

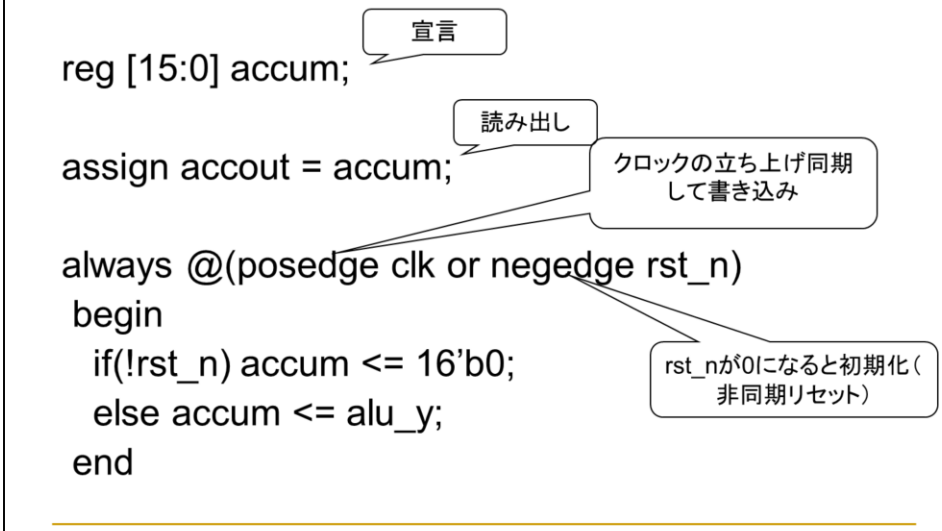
この構文中にはif文が使われています。これはC言語同様、(　)内の条件が満足されるとその後の文が実行され、そうでなければelse以下の文が実行されます。ここでは、単一の文しか書かれていませんが、begin..endを使って複数の文を書くことができます。

if(！rst_n)は、rst_nがLレベルである時、条件が成り立ちます。always文の条件より、これが成り立つのはrst_nがHからLに変化した時と考えられます。この場合は、accumに0が入ります。これはリセットが掛かったことになります。レジスタやD-F.F.は、システムがスタートしたときに原則として状態が決まっている必要があります。このため、システムには通常リセット入力があり、この記述ではrst_nがこれに当たります。rst_nはLになった時にクロックとは無関係に（条件がorなので）リセットが掛かります。これを非同期リセットと呼びます。最近、多くのディジタル回路では非同期リセットが使われ、この授業でもリセットは全部非同期にしています。

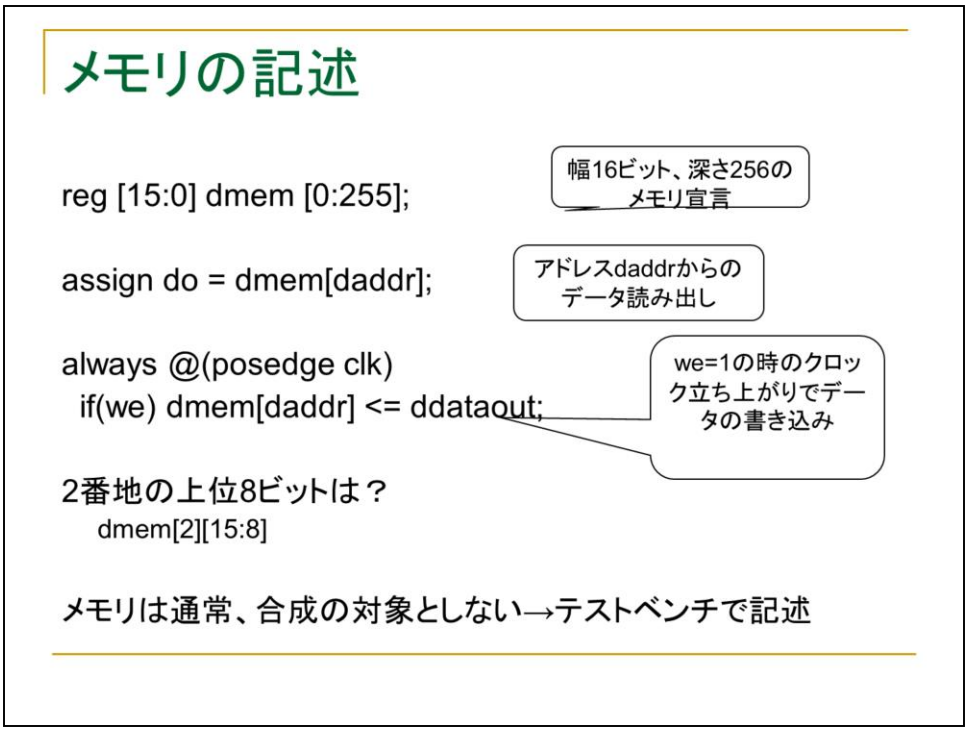
次にelseが成立した場合、つまりrst_n=Hの時には、クロックの立ち上がりで（

posedge clk)でalu_y、つまりALUの出力がaccumに格納されます。＜＝はブロッキング代入文と呼び、レジスタに代入するときにだけ使い、always文（initial文も可能）の中だけで使われます。if文が使えるのもalways文の中だけです。

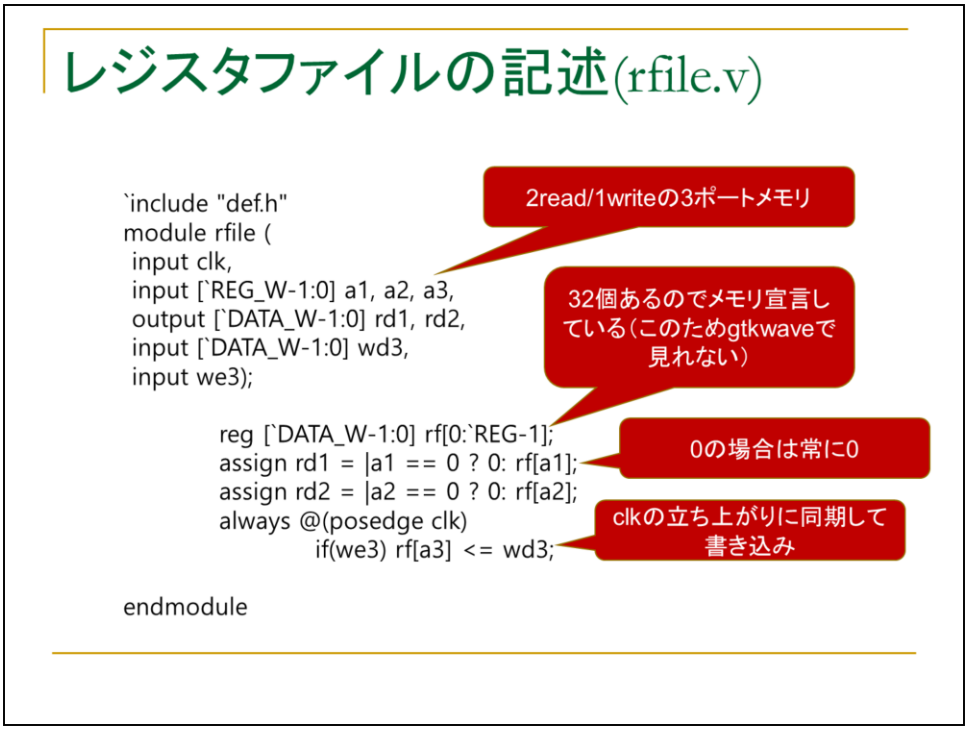
レジスタのVerilog記述



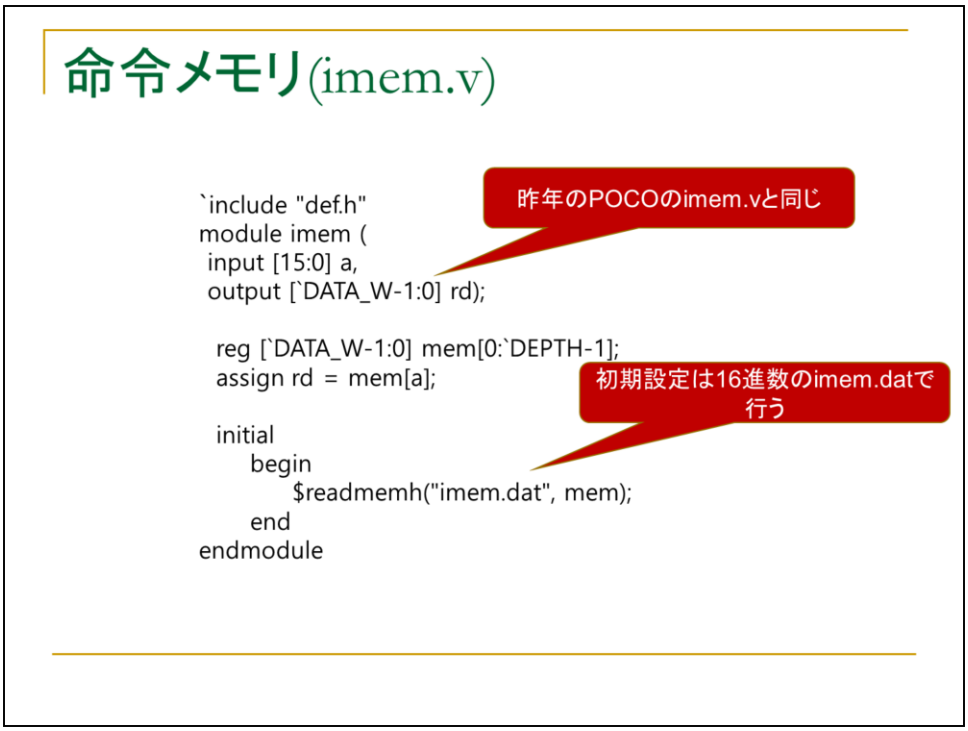
これは、rst_nが0になると0になるという非同期レジスタの記述です。それ以外の場合は、次のクロックの立ち上がりでaly_yがaccumに格納されます。



次はメモリの記述について復習します。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256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。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。では2番地の内容の上位8ビットを切り出す場合どう書けばよいでしょうか？この場合、[] []の形にして、最初の[]内にアドレスを書き、次の[]で取り出すビットを指定します。メモリは通常、特別な回路を使うため、論理合成の対象とはしません。このためテストベンチに入れるか、独立のモジュールとして別に記述します。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。



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、2read/1writeの3ポートメモリとして記述してあります。図で示したのと同じ名前を使います(ただし小文字です)。



命令メモリは昨年のPOCOのimem.vと同じです。ポート名は図と合わせてあります。初期化はimem.datで行うので、アセンブルした結果はここに入れる必要があります。結果は32ビットで長いので16ビットで表します。メモリは32ビット分のアドレスがあるのですが、演習でそんなに大きいのは使わないので16ビットで済ませています。

データメモリ(dmem.v)

```
`include "def.h"
module dmem (
 input clk,
 input [15:0] a,
 output [`DATA_W-1:0] rd,
 input [`DATA_W-1:0] wd,
 input  we);
   reg [`DATA_W-1:0] mem[0:`DEPTH-1];
   assign rd = mem[a];
   always @(posedge clk)  begin
                    if(we) mem[a] <= wd;
   end
   initial begin
          $readmemh("dmem.dat", mem);
    end
endmodule
```

昨年のPOCOのdmem.vと同じ

初期設定は16進数のdmem.datで行う

データメモリは、読み出しと書き込みが可能です。ポート名は図と合わせてあります。we=1の時にクロックの立ち上がりに同期してwdが書き込まれます。こちらも16ビットアドレスにしています。

MIPSeの本体（簡略版）

入出力名は図と合わせてある
pcはそのままレジスタ宣言している

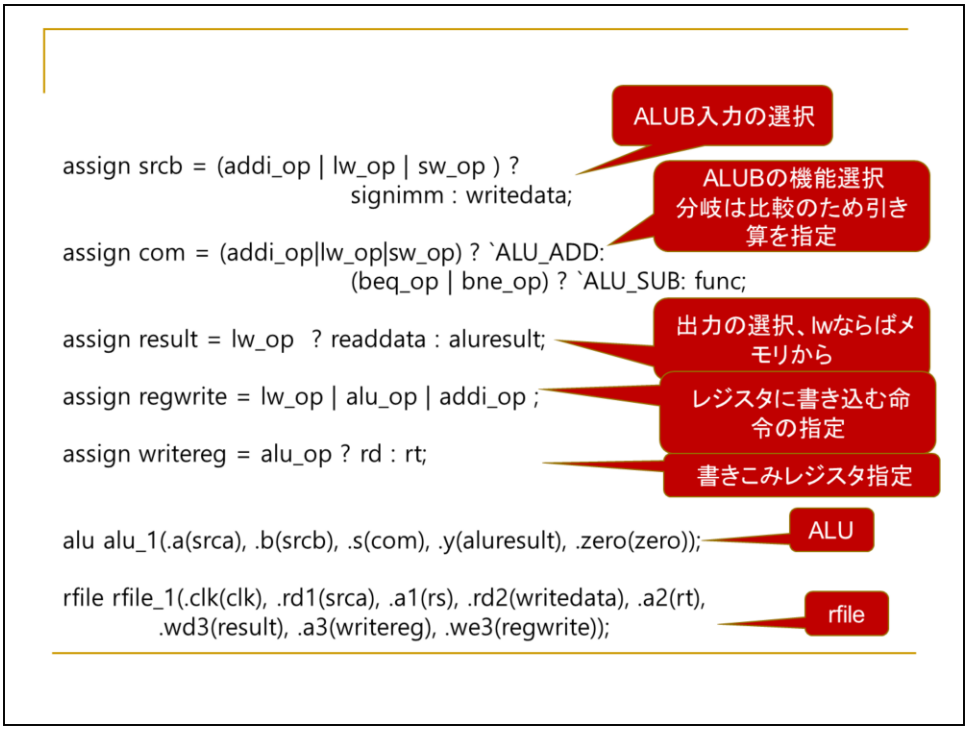
```
`include "def.h"
module mipse(
input clk, rst_n,
input [`DATA_W-1:0] instr,
input [`DATA_W-1:0] readdata,
output reg [`DATA_W-1:0] pc,
output [`DATA_W-1:0] aluresult,
output [`DATA_W-1:0] writedata,
output memwrite);

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;
```

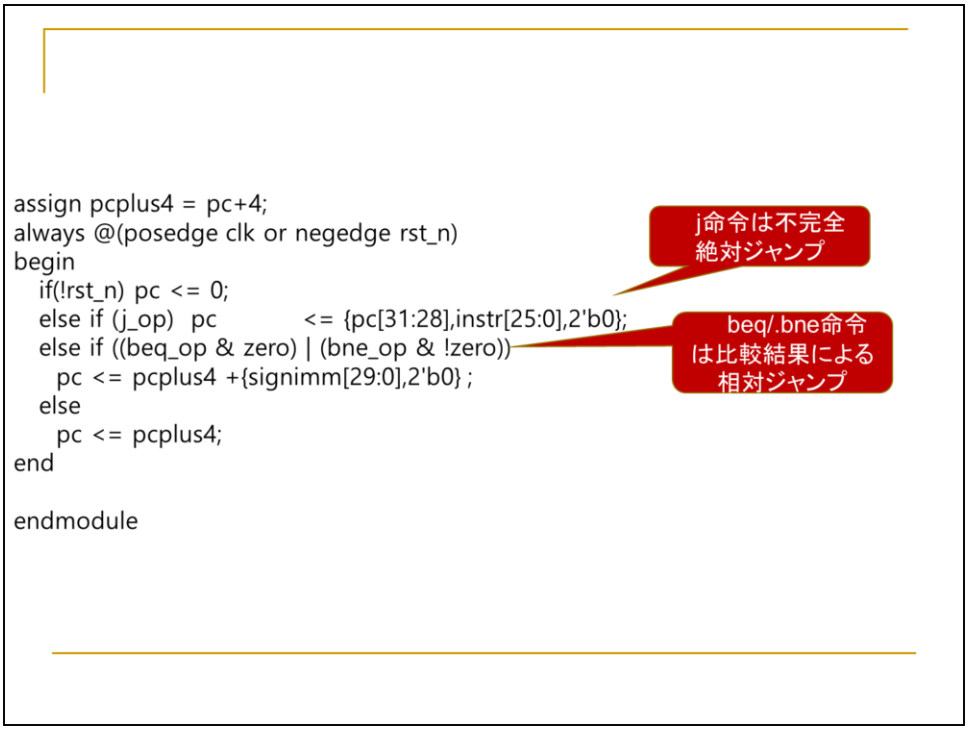
各命令のデコード信号
lw,sw,beq,bne,addi,j,alu命令のみ

では、MIPSeの本体を説明しましょう。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。

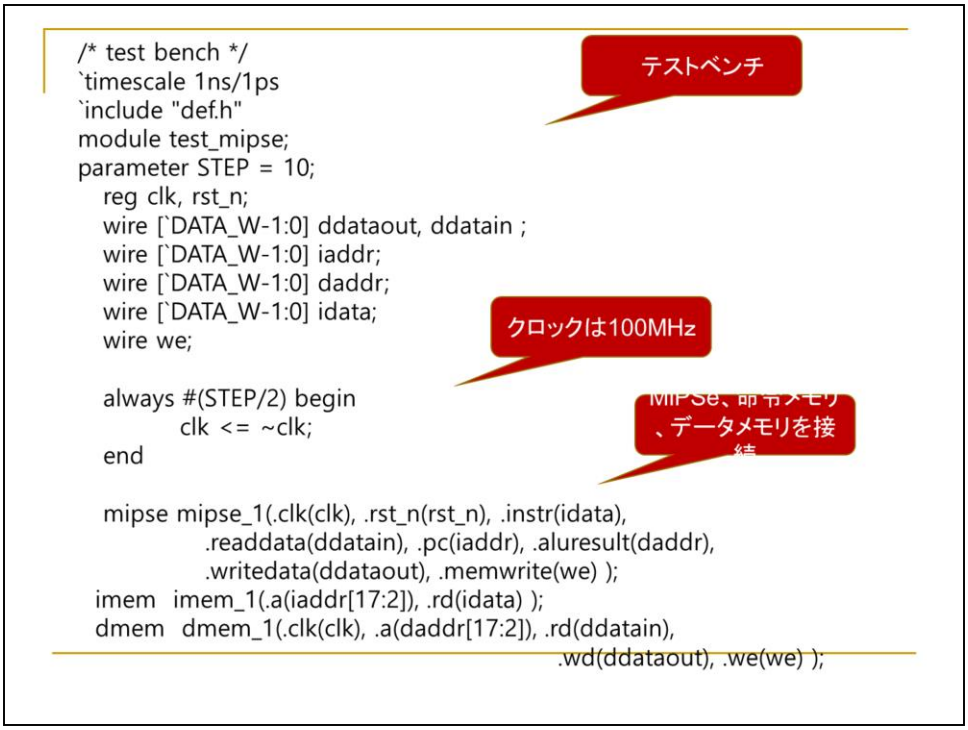
命令のデコード部分です。デコード信号は図にないので注意ください。連結構文{}の使い方、符号拡張の記述を思い出してください。



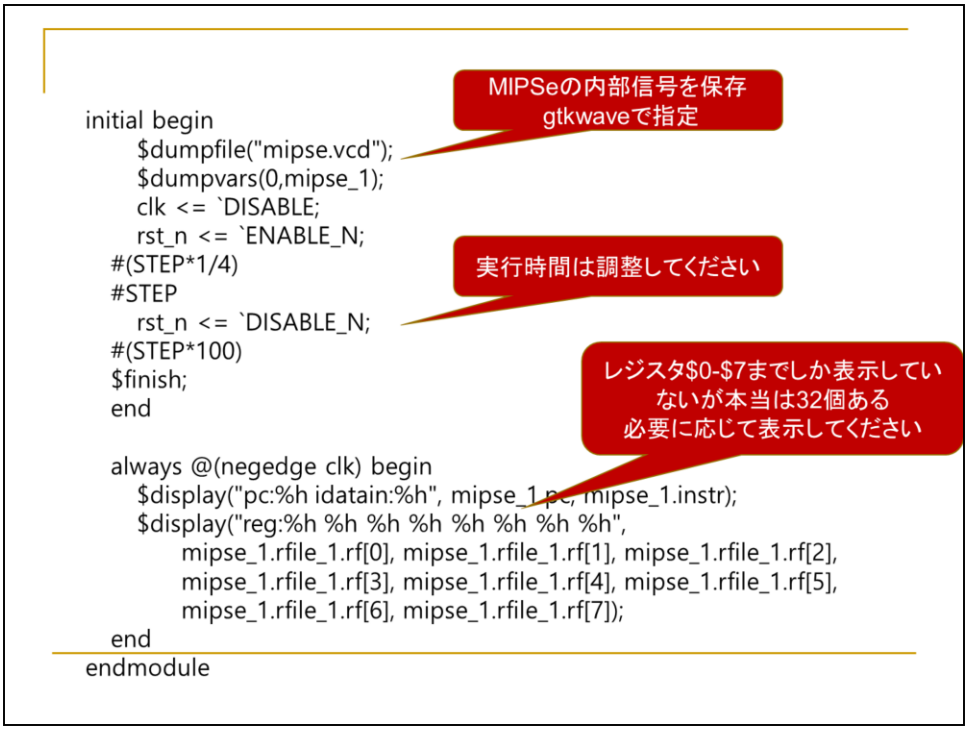
各部のマルチプレクサの制御、ALU、rfileの接続です。これも図と照らし合わせましょう。



分岐命令とj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。一方、j命令は上位4ビットはPCのままです。



最後にテストベンチの開設です。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。

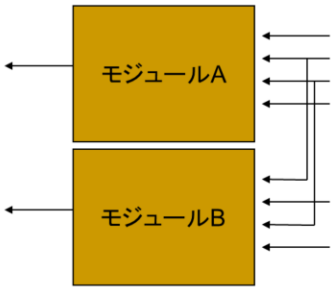


テストベンチの後半です。波形をgtkwaveで見れるように波形ファイル（vcdファイル）を作ります。命令が一つずつ実行されるのを見るためにプリントもしていますが、レジスタは実は

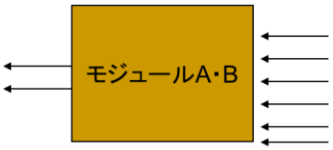
全部は表示していません。ここでの例ではメモリも表示していません。これは必要に応じて変えてください。

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。

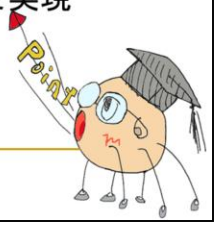
例題

- 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

- MIPSは32ビットのRISC、アドレス、データ共に32ビット
- アドレスはバイトアドレッシング、32ビット命令、データは4の倍数のみ
- 32ビットのレジスタを32本持つ。レジスタ0は常に0
- ディスプレースメント付きレジスタ間接指定でメモリのアドレスを指定
- 3オペランド
- 条件分岐はレジスタ二つを比較、PC相対指定
- 大小比較は比較命令と分岐命令の組み合わせで実現
- jとjalは制限付きの絶対指定



インフォ丸が教えてくれる今日のまとめです。

演習3

- reviewを用いる
- andi rt,rs,X　(ANDI命令)を付け加えよ
- Xはゼロ拡張する

opcode 001100 （実はdef.hに付いている）

anditst.asmでテストして確認すること

$5555が1515になっていればOK

提出物：andiを付け加えたmipse.v

覚えておくと便利

tarの解凍
tar xvf file.tar

アセンブラ
./asm.pl file.asm –o imem.dat

論理シミュレーションiverilog
iverilog *.v
vvp a.out (./a.out | more)

波形ビューアgtkwave
gtkwave mipse.vcd

資料
http://www.am.ics.keio.ac.jp

レポート提出
keio.jp経由で提出のこと