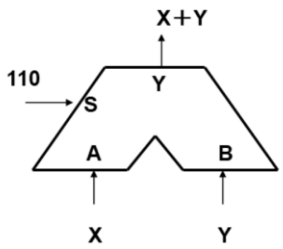
マイクロコンピュータ基礎　第3回
データパス：計算をするところ

情報工学科
天野英晴

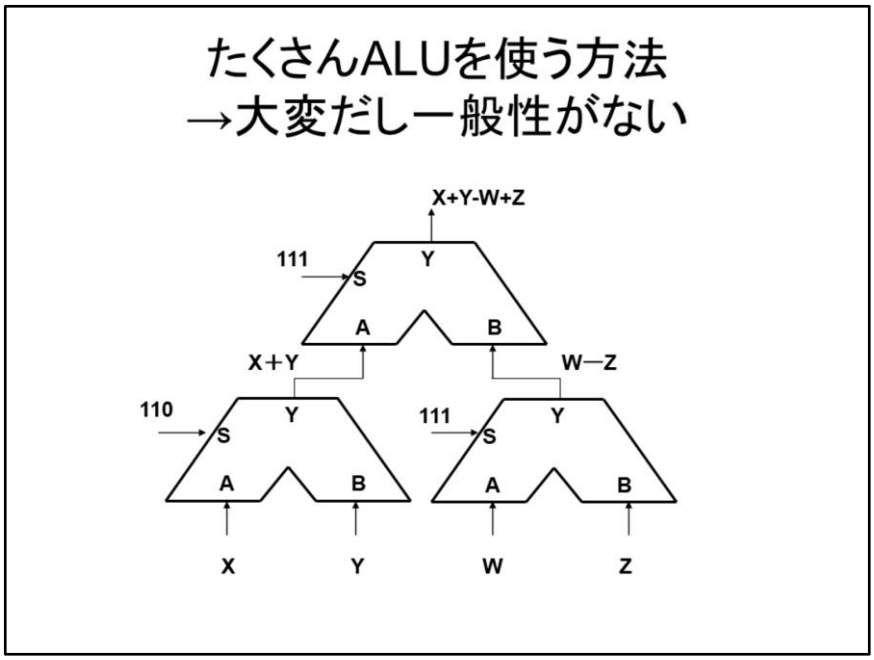
CPUはデータパスとコントローラから出来ています。今日は最も簡単なデータパスを設計します。

ALUで色々な演算ができる

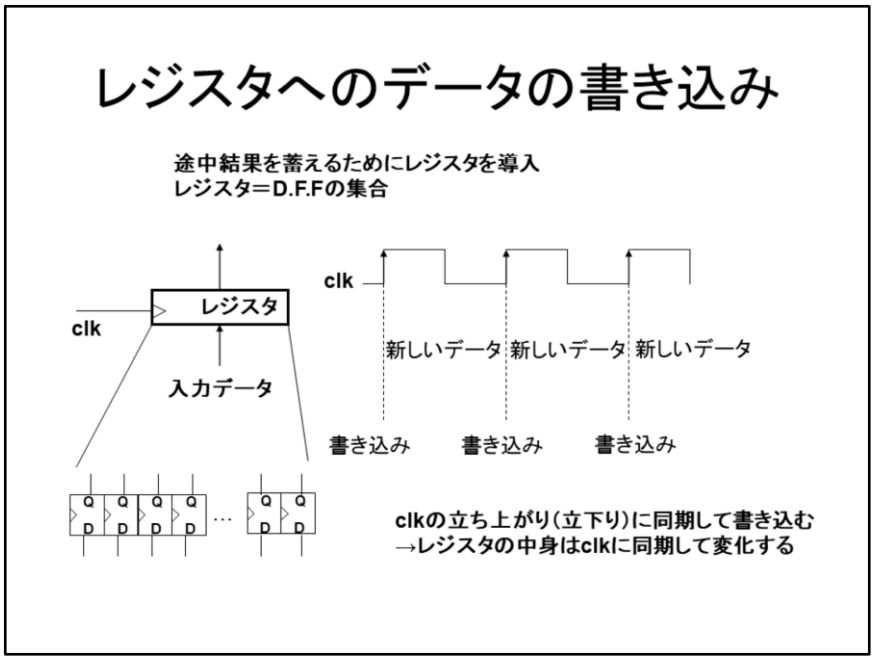
- しかし、2つの入力データに限定される



ALUは、S入力を変えることで様々な計算を行いました。しかし、ALU一つだけでは2つの入力に対して1回しか計算ができません。

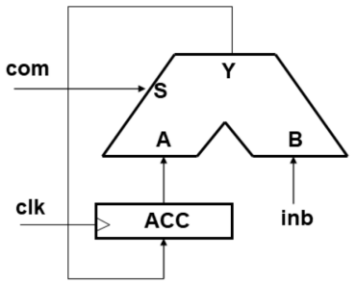


例えばX+Y-W+Zを計算するにはどうすれば良いでしょう？一つの手はALUを三つ使って1個目ではX+Y、2個目でW-Zを計算し、二つの答えを引き算すれば、結果を得ることができます。しかし、この方法では、式が変わるたびにALU同士の接続を変えなければならず、一般性がありません。（実は線を繋ぎ変えて計算する方式もあるのですが、ここではまず基本をやりましょう）ではどうすれば良いかというと、中間結果をどこかに蓄えておけばいいのです。



中間結果を蓄える役目をするのは、レジスタです。レジスタは、計算機基礎や電子回路基礎で習ったD-FFを記憶したいビット数分だけ集めてクロックを共通にしたものです。レジスタはクロックの立ち上がりでD入力の値を記憶してQに出力します。一定の周期のクロックを与えた場合、レジスタの内容はクロックに同期して変化することになります。

レジスタの利用

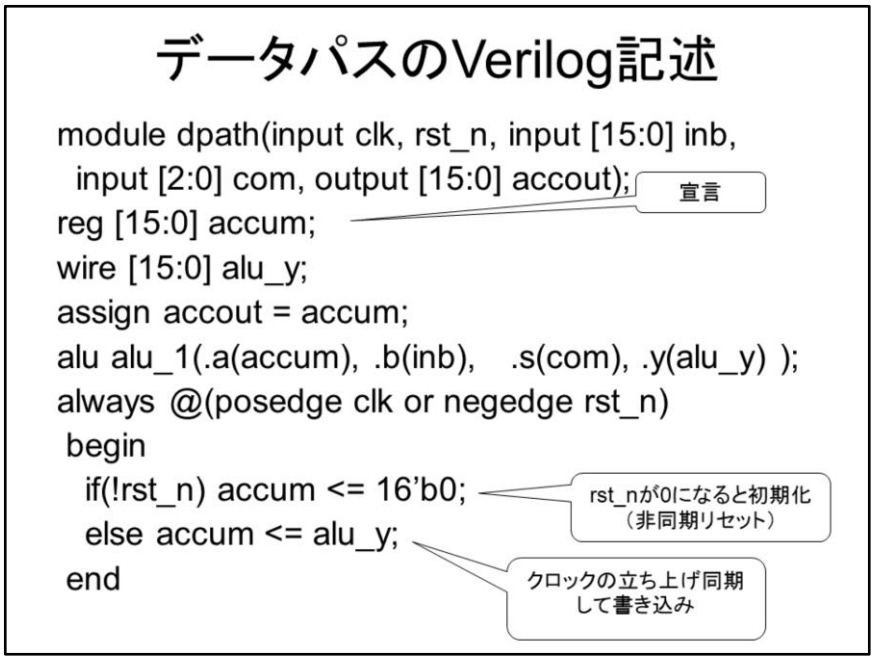


S	B	clk立上り後のACCの内容
001	X	X
110	Y	X+Y
111	W	X+Y-W
110	Z	X+Y-W+Z

ACC：アキュムレータ
結果を蓄えるレジスタ

レジスタ＋ALUでデータパス（計算をする場所）を形成する。

レジスタの出力をALUをA入力に接続し、ALUの出力をくるっと回してレジスタの入力に繋ぎます。この構造で、中間結果をレジスタにとっておけるようになります。このレジスタは結果が次々に積み重なっていくことからアキュムレータ（Accumulator)と呼ばれます。ここではACCと書きます。
B入力からのデータとS入力での演算の指示により、X+Y-W+Zは表のように実行されます。まずSを001にしてBにXを入れます。001はTHBなので、Y=Bとなり、これが次のクロックの立ち上がりでACCに入ります。すなわちACCはXになります。
次にSを110（加算）にして、足すべき数YをB入力に入れます。YにはX+Yが表れ、これが次のクロックの立ち上がりでACCに入ります。
次にSを111（減算）にして、WをB入力に入れます。Yには、今ACCにあるX+YからWが引かれた値が表れ、これが次のクロックの立ち上がりでACCに入ります。
さらにSを110（加算）にして、ZをB入力に入れます。Yには、今ACCにあるX+Y－WにZを足した値が表れ、これが次のクロックの立ち上がりでACCに入ります。
レジスタとALUを組み合わせた、計算のための基本構成をデータパスと呼びます。



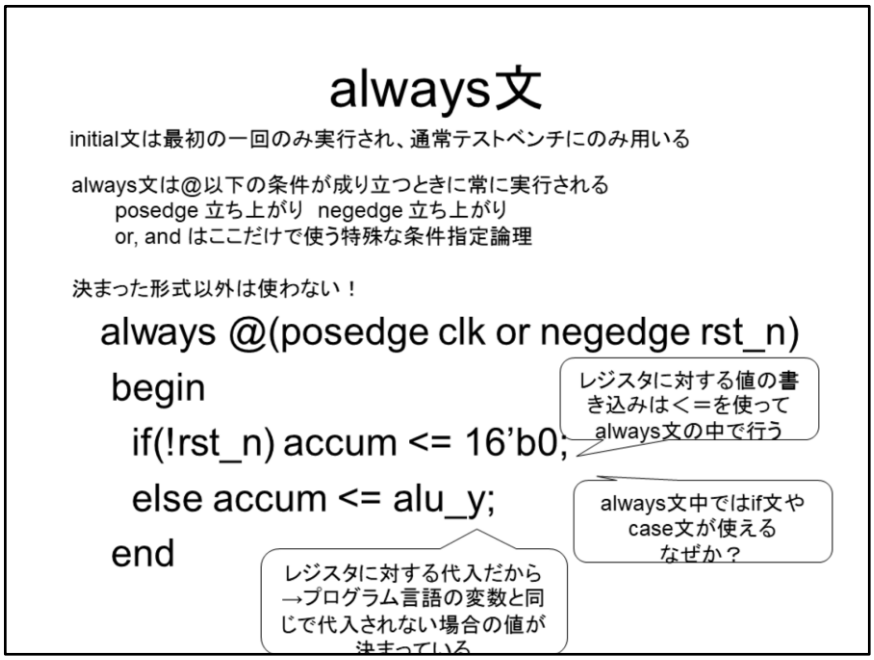
では、データパスのVerilog記述を解説します。図に従い、16ビットのinb入力、3ビットのcom入力を付け、ACCの出力をaccoutという名前で出力します。clkはクロック、rst_nはシステムリセット信号です。これはアクティブLの信号でLレベルでリセットします。歴史的な経緯でリセットはLで行う場合が多く、ここでも伝統に従っています。この授業ではアクティブLの信号はあまり使いませんが、使う場合は信号線名の後ろに_nを付けて示します。

まずアキュムレータaccumをreg文で宣言します。reg文は、D-FF、レジスタを宣言する文です。ここでは、バスの記述方法を用いて16ビットのレジスタを宣言しています。

次にALUの出力にalu_yという名前をつけます。これはregではなくwire文を使います。単に信号線に名前を付けるだけでレジスタができるわけではないからです。このregとwireの違いは段々に分かってくると思います。

今まで同様、ALUを実体化し、alu_1という名前にして、a, b, s,yにそれぞれ信号を繋ぎます。A入力にはレジスタ名をそのまま書くことにより、出力を繋いでいる点にご注意ください。

次にalways文を使ってレジスタに値がどのような条件で設定されるかを記述します。このalways文はVerilogの入門者が戸惑う奇妙な記法ですが、基本的に同じパターンでしか使わないので、それだけ覚えて使ってください。このalways文はパターンからはずれた書き方をすると、予期しない回路が合成されてトラブルの元です。



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。

テストベンチ 宣言部

```
module test;
  parameter STEP=10;
  reg clk, rst_n;
  reg [2:0] s;
  reg [15:0] b;
  wire [15:0] accum;
  dpath dpath0(.clk(clk), .rst_n(rst_n),
      .com(s), .datain(b), .accum(accum));
  initial begin
          $dumpfile(“dpath.vcd”);
          $dumpvars(0, test);
```

では次にこのデータパスをテストするテストベンチについて解説しましょう。宣言の部分は今までと同じです。ここではdpath.vcdというところにシミュレーションの結果をしまっておきます。

テストベンチ　続き

```
 clk <= 0;
 rst_n <=0;
#STEP  rst_n <= 1;
          s <= 3'b001; b<= 16'h2222;
#STEP  clk <=1; #STEP clk <=0;
         s<= 3'b110; b<= 16'h3333;
#STEP clk <=1; #STEP clk <=0;
        s<= 3'b111; b<=16'h1111;
#STEP clk <=1; #STEP clk <=0;
         s<=3'b110; b<=16'h4444;
#STEP clk <=1; #STEP clk <=0;
$finish;
end
```

実行して結果を見よう
display文は省略してある

次に、コマンド入力sとデータ入力bに値を入れ、時間を進めてクロックをL→H→Lに変化させて、データパスを動かします。ここでは、結果を表示するdisplay文は省略してありますが、実際のファイルには付いています。では、実行して結果を確認しましょう。sを変えてやれば様々な計算が可能です。では、iverilog test_dpath.v dpath.vでコンパイルし、vpp a.outで実行して結果を確認しましょう。

クロックをいちいち書くのは面倒

```
parameter STEP=10;
reg clk;
always #(STEP/2) begin
        clk <= ~clk;
end
```

周期STEPのクロックを発生する
initial文でclkを初期化してやる必要がある

さて、このtest.vでは一定時間を経過する毎にclkをLにしたり、Hにしたりするのをいちいち書いてやる必要がありました。これは面倒なので、always文を使ってclkが常に一定の時間でL->H-> Lを繰り返してもらう記述をここに示してあります。この記述では、5nsec毎にクロックが今までのレベルと反転し、周期はSTEP=10nsecになります。もちろん実際のクロックをこのように発振することはできず、テストベンチでしか使えない記法です。この構文では最初にclkがどうなっているかを指定する必要があり、これをinitial文中で行います。

記述例：カウンタ

```
module counter (
        input clk, rst_n, output [3:0] c);
reg [3:0] count;
assign c = count;
always @(posedge clk or negedge rst_n) begin
        if(!rst_n)  count <=0;
        else count <=count+1;
end
endmodule
```

C言語じゃないので＋+は使えない！

少し、レジスタを使って順序回路を記述する方法の例題を示しましょう。ここでは、countをregで宣言し、rst_nがLの時には0に初期化し、そうでない時は1ずつ増やすという記述でカウンタを記述しています。countは4ビットで宣言されているので、1111を超えると次のクロックで0000に戻り、再びカウントアップを繰り返します。C言語ではないので、＋＋は使えない点にご注意ください。

記述例：カウンタ 簡易記述版

```
module counter (
        input clk, rst_n, output reg [3:0] c);
always @(posedge clk or negedge rst_n)
        if(!rst_n)  c <=0;
        else c <=c+1;
endmodule
```

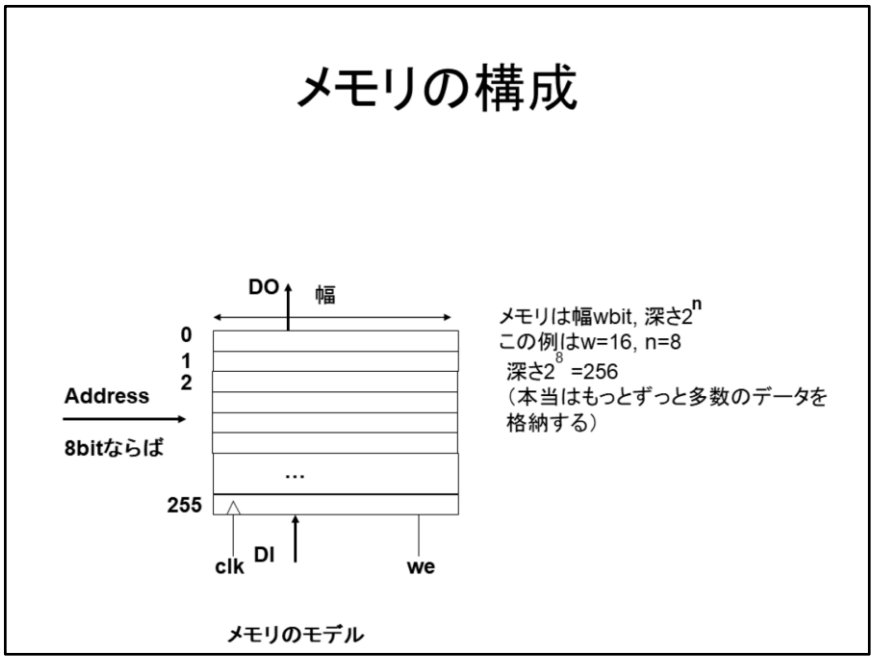
出力端子をレジスタとして使っている
不要なbegin endは省略

output regと書くことで、出力信号をレジスタにすることができます。このようにすると、出力に対して直接値を入れることが可能です。わざわざレジスタcountを宣言する必要がなくなります。また、この記述ではalwaysのbegin..endも省略してあります。これは、alywas中の文がif..else..の一文だけだからです。この辺はC言語の{　}と同じです。
このカウンタは、iverilog test_counter.v counter.vでコンパイル、vvp a.outで実行が可能です。実行して結果を確認しましょう。

このデータパスの問題点

- 途中結果を保存しておけない
- データやコマンドをいちいち入力しなければならない
- メモリを導入する
  - データメモリ：入力データの保存、途中結果の保存、出力結果の書き出し
  - 命令メモリ：コマンドとデータメモリのアドレスの保存
- 命令メモリを順番に読み出すことで計算を実行

ではこのデータパスで全ての計算ができるでしょうか？例えば、A>>1＋B<<1などができません。これはAを1ビット右シフトした結果をどこかにとって置いて、これにBを1ビット左シフトした結果を足さなければならないからです。電卓ではこのような場合、メモリ機能を使います。途中結果をメモリに保存しておき、これを取り出して計算を実行すればよいのです。これと同じように、複数のデータを記憶できるデータメモリを設けましょう。計算すべき入力データもメモリに最初に入れておくようにすれば、いちいち値を外から入力しなくても良くなります。また、別に命令メモリを設けてここにコマンドを入れておいて順番に読み出して実行させれば、いちいち計算しながら値を入力する必要がなくなります。

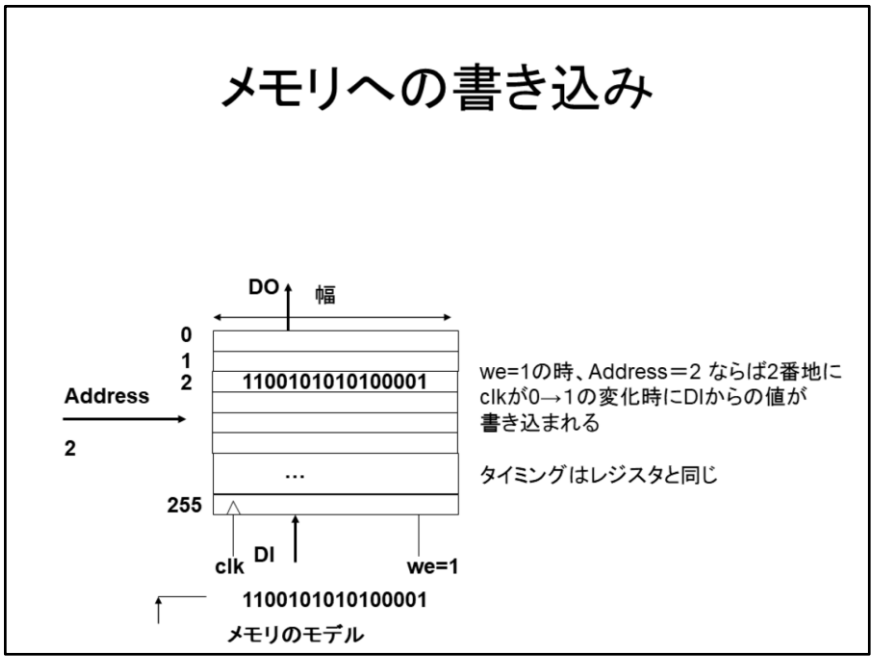


計算機基礎、電子回路基礎で習ったとおり、メモリは単純な表であり、アドレスで指定した番地のデータを読んだり書いたりできます。メモリの大きさは幅と深さで表すことができます。ここでは幅が16ビット、深さは2の8乗＝256のメモリを想定します。8ビットのアドレスによりデータを指定することができます。データは入力DIと出力DOで、それぞれ16ビットです。DOには、アドレスで指定した番地のデータが常に出力されています。一方、書き込みの際は、DIに値を与え、we(Write Enable)をHにした際にクロックがLからHに変化すると、そこでDIの値がアドレスで指定された番地に書き込まれます。

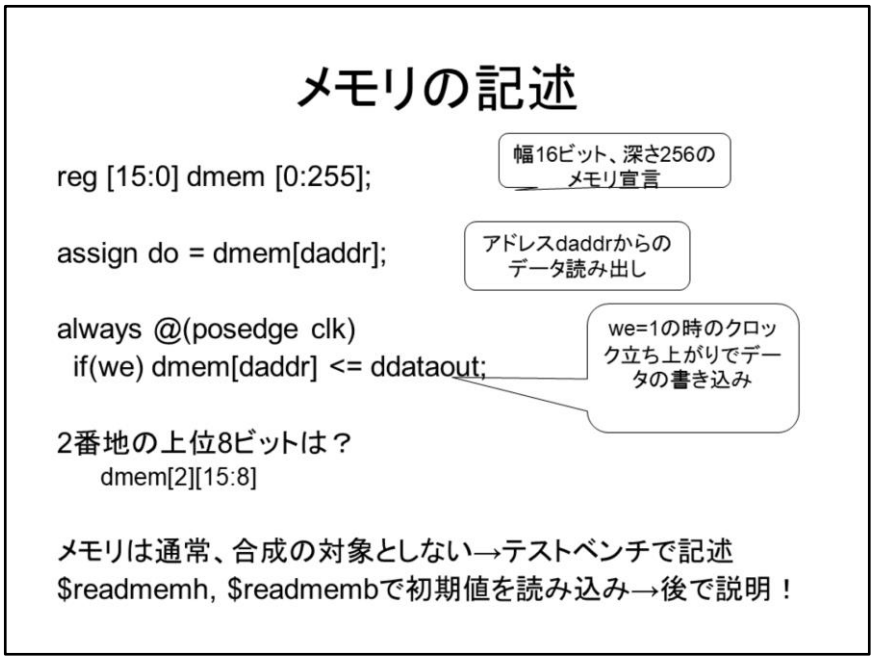
メモリからの読み出し

メモリのモデル

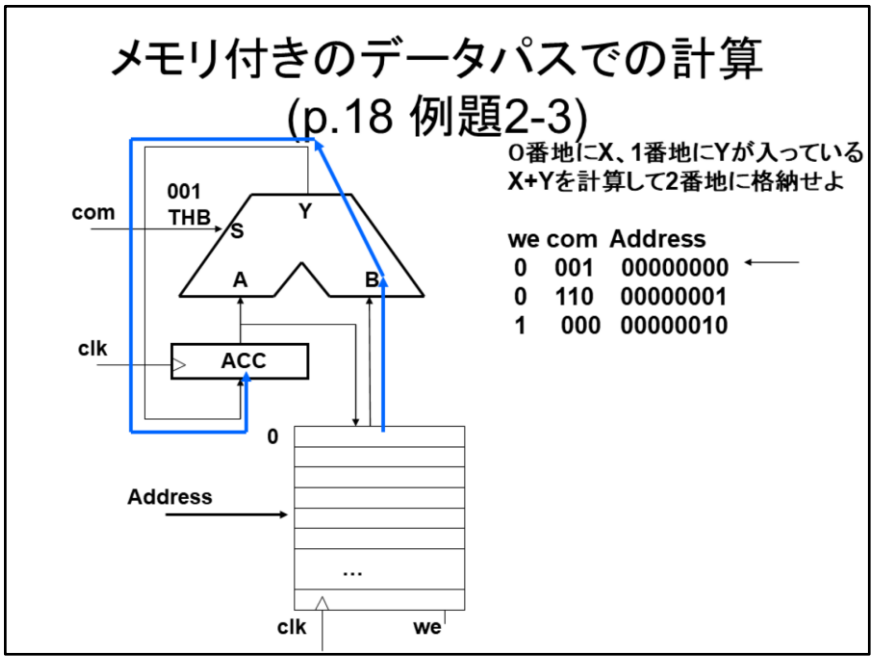
メモリの読み出しの例を示します。指定したアドレスの番地にしまってある入力はいつでもDOに出力されています。



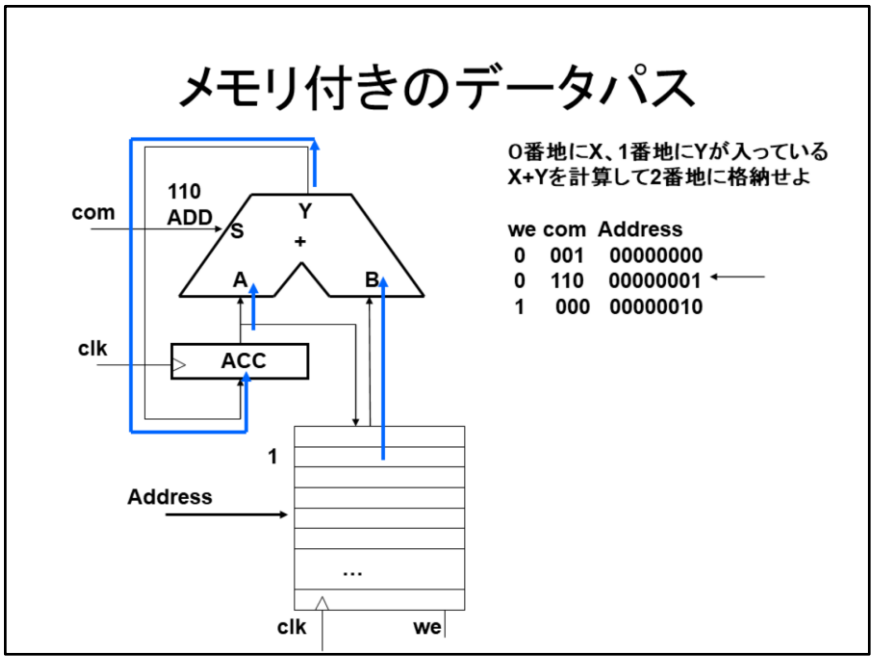
一方、書き込みの際は、DIに値を与え、weを1にします。ここでclkがLからHに変化するとDIの値が指定されたアドレスに書き込まれます。書き込みのタイミングはレジスタと同じである点にご注意ください。メモリには普通、リセット信号がなく、初期化はできません。メモリの値を確定するためには、あらかじめ値を書いておく必要があります。



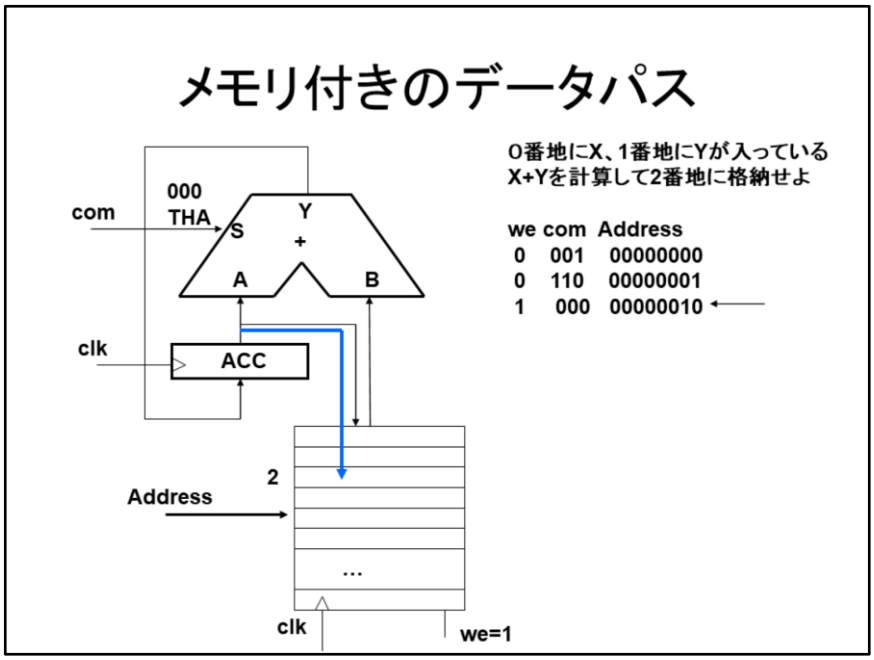
ではメモリの記述について解説します。メモリの宣言は、reg　[MSB:LSB] 「最小アドレス:最大アドレス]で行います。ここでは、16ビットで深さが256のメモリが宣言されます。最小アドレスは0、最大アドレスは2のn乗にするのが普通です。メモリはC言語の配列に似ているので、配列同様[　]の中に番地を入れて値を取り出します。書き込む場合は、if(we)としてwe=Hの時だけclkに同期して入力を書き込みます。では2番地の内容の上位8ビットを切り出す場合どう書けばよいでしょうか？この場合、[] []の形にして、最初の[]内にアドレスを書き、次の[]で取り出すビットを指定します。メモリは通常、特別な回路を使うため、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で後ほど紹介します。



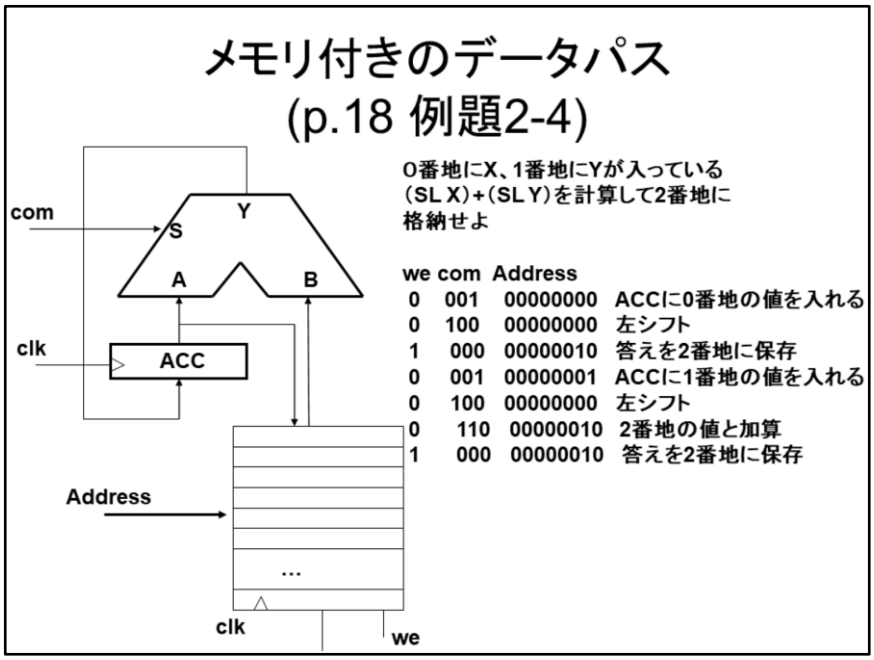
ではまずデータメモリだけ付けて、簡単な計算をやってみましょう。テキスト18ページ例題2.3で、対応するverilogファイルはdatapath1.v、test_datapath1.vです。0番地の中身と1番地の中身を足して2番地にしまうという至極簡単な操作です。
最初にwe=0,com=001,Address=0にして0番地の内容をACCに持ってきます。このようにメモリの中身をACCに持ってくることをロードと呼びます。



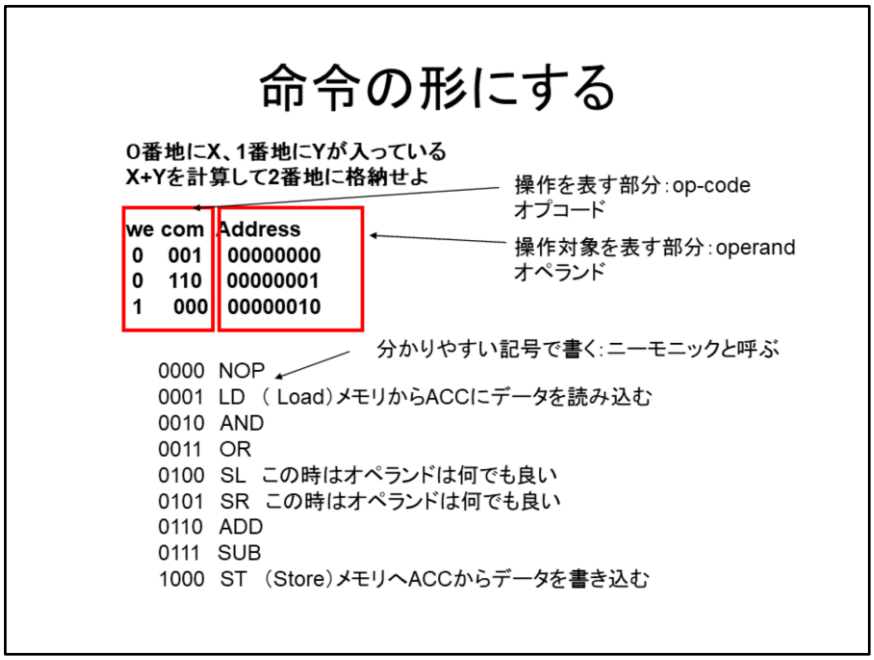
次にwe=0, com=110としてACCにロードした値とメモリの１番地から読んだ値を加算して、答えをACCに格納します。



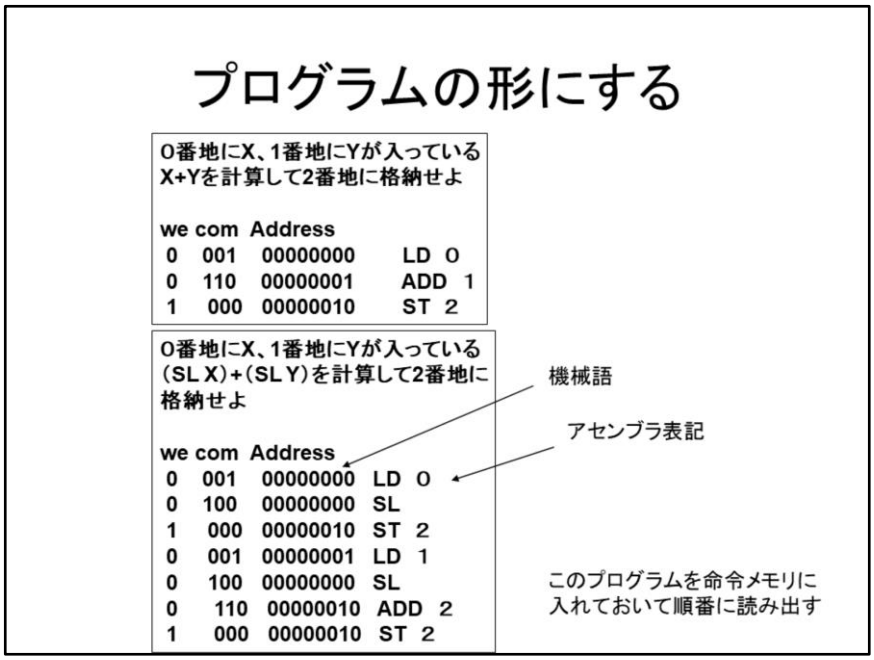
今まではメモリに書き込まないので、we=0にしていました。しかし、今度はwe=1としてACCの出力をメモリに書き込んでやります。この時、com=000とします。000はTHAなのでYにはACCの値がそのまま出てきて再びACCに格納されます。この操作でACCの内容は破壊されません。通常メモリにしまってもレジスタの値が壊れることはありません。レジスタの値をメモリに格納する操作をストアと呼びます。ストアはロードの反対の操作です。



ではやや複雑な計算をやってみましょう。この例では、0番地の値Xと1番地の値Yについて X＜＜1 ＋ Y<<1を計算するものです。まず、Xを左シフトして、その結果をメモリに保存しておきます。ここでは2番地に保存しますが、他の番地でもOKです（0と1はX,Yを破壊するので止めておきましょう）。次に1番地の値を取り出して左シフトさせて、この結果と先ほど2番地にしまっておいた値を足します。



さて、データパスの制御信号weとcomの組み合わせて、どのような演算を行うか（あるいはメモリに書き込むか）を示します。また、アドレスはこの操作を行う対象を示します。このように操作を表す部分と操作対象を表す部分を組にしたものを命令と呼びます。命令は操作内容を示す部分（weとcomの組み合わせ）をオプコード（op-code)、操作対象を表す部分をオペランド（operand)と呼びます。オプコードは、ハードウェアに対してはこの場合4ビットのコードで表されますが、見やすくするため、人間に対しては記号で示します。これをニーモニックと呼びます。ここでは9種類の命令を用意します。ほとんどがALUの演算と同じ名前ですが、LD（ロード）はメモリからデータを取って来る操作、ST（ストア）はメモリにしまう操作です。なにもやらない命令NOP（ノップ）があるのは奇妙な気がしますが、後にこれもちゃんと役に立つことが分かります。

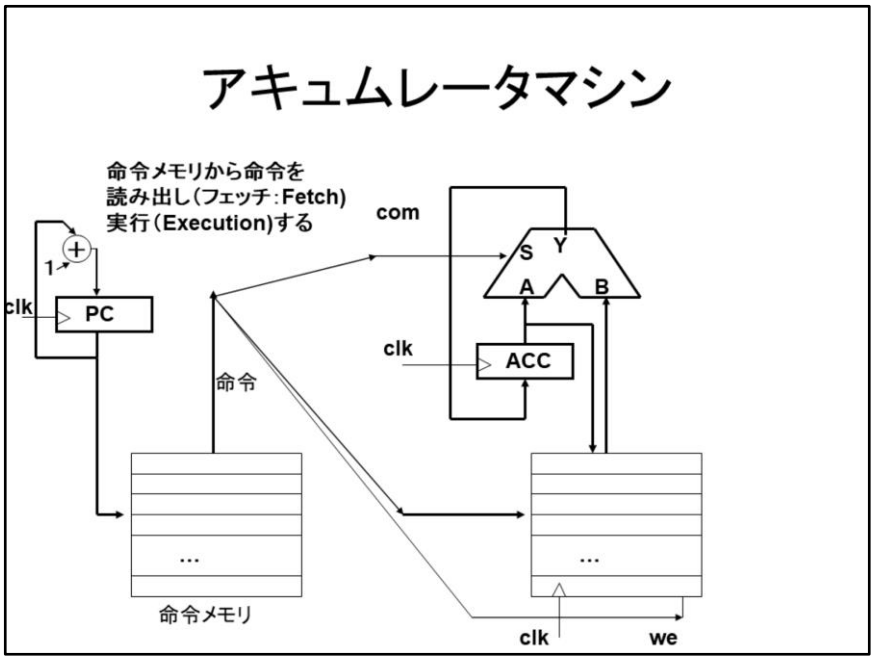


今まで、計算をやらせるために信号線に一定の1・0を与えてきましたが、これを命令の形で表し、プログラムの形にしてみたのがこのスライドです。LD 0は、0番地の値をACCに読み出すことを示し、ADD 1はACCの値に1番地の値を足すことを、ST 2はACCの値を2番地に格納することを示します。このように人間にわかりやすいようにニーモニックを使った命令の表記で表したプログラムをアセンブリ言語によるプログラム、アセンブラ表記と呼びます。元の1・0表記を機械語（マシンコード）と呼び、アセンブラ表記を機械語に変換するソフトウェアのことをアセンブラと呼びます。実は我々はアセンブリ言語による表記のこともアセンブラと呼ぶことが多いです。「アセンブラで書け」などのように使います。

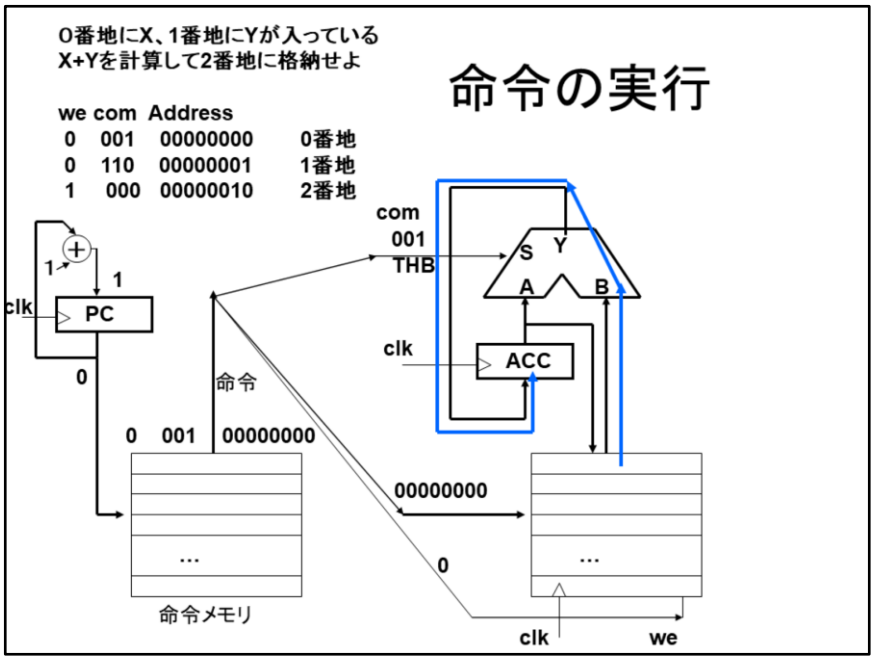
命令実行の仕組み

- 命令メモリ：命令の長さ12bitに対応する幅(w=12)、深さは256とする(n=8bit)
- PC (Program Counter): 現在実行する命令のアドレスを保持する
  - PCを1クロックに1つづつ増やしていくことで、命令メモリに入っている命令を順番に実行する
  - アキュムレータマシンの基礎
    - アキュムレータしかレジスタを持たないもっとも原始的なコンピュータ
    - EDSAC、EDVACなどの草創期のコンピュータ、6800、6502など草創期のマイクロプロセッサは一種のアキュムレータマシン

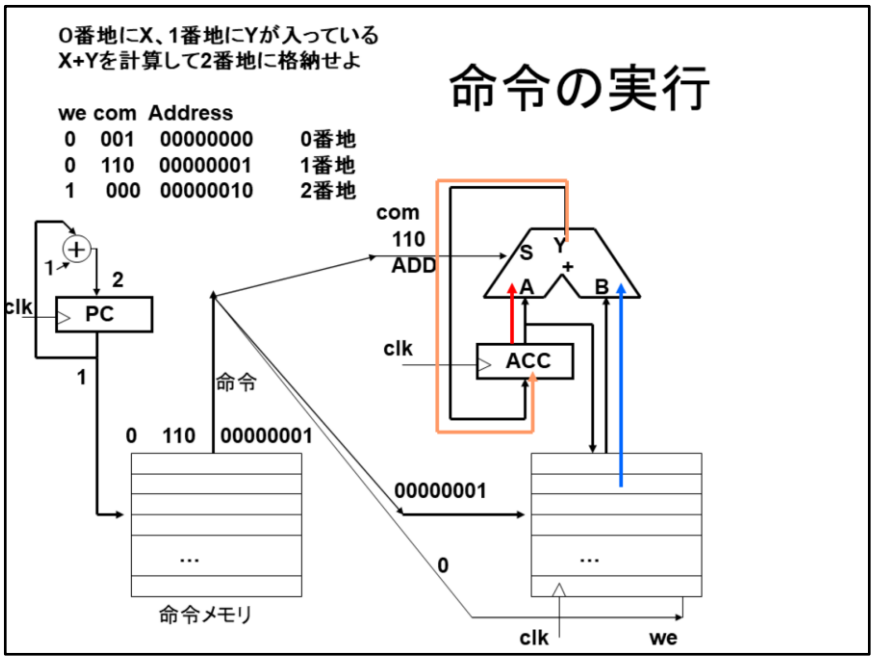
さて、この命令の並びでできたプログラムを命令メモリに入れておきます。今、命令はオプコード4ビット、オペランドが8ビットあるので全部で幅は12ビットになります。深さはデータメモリと同じ256にしましょう。この命令メモリの内容を順番に取り出してデータパスに信号として与えてやれば、いちいち手で信号とデータを与えなくても自動で計算を行ってくれます。これが最も原始的なコンピュータの原理です。このために、実行する命令の番地を持っているレジスタを設けます。これをプログラムカウンタ（PC)と呼びます。プログラムカウンタの指し示す番地の命令メモリを読み出し、これをデータパスに与え、計算が行われると同時にPCに1を足してやり、次の番地を指すようにすれば、順番に次々に命令メモリ中の命令を読み出して実行することができます。これがアキュムレータマシンの基本です。EDVAC,EDSACのような草創期のマシン、6800、6502など草創期のマイクロプロセッサはアキュムレータマシンでした。



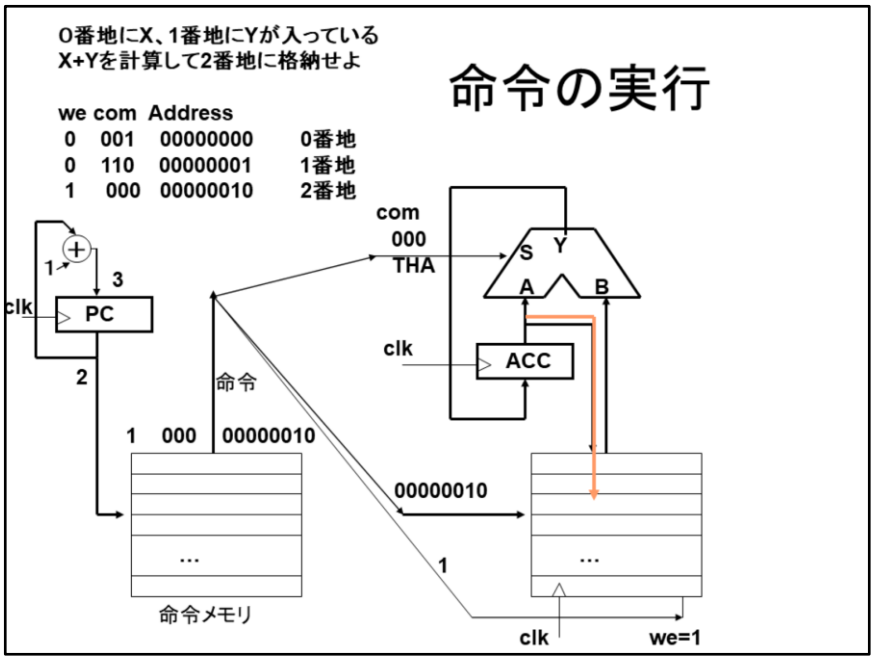
ではアキュムレータマシンの構造を示します。プログラムカウンタ(PC)の指し示す命令メモリの中身を命令として読み出し、オプコードをweとcomに、オペランドをデータメモリのアドレスとして与えてやります。



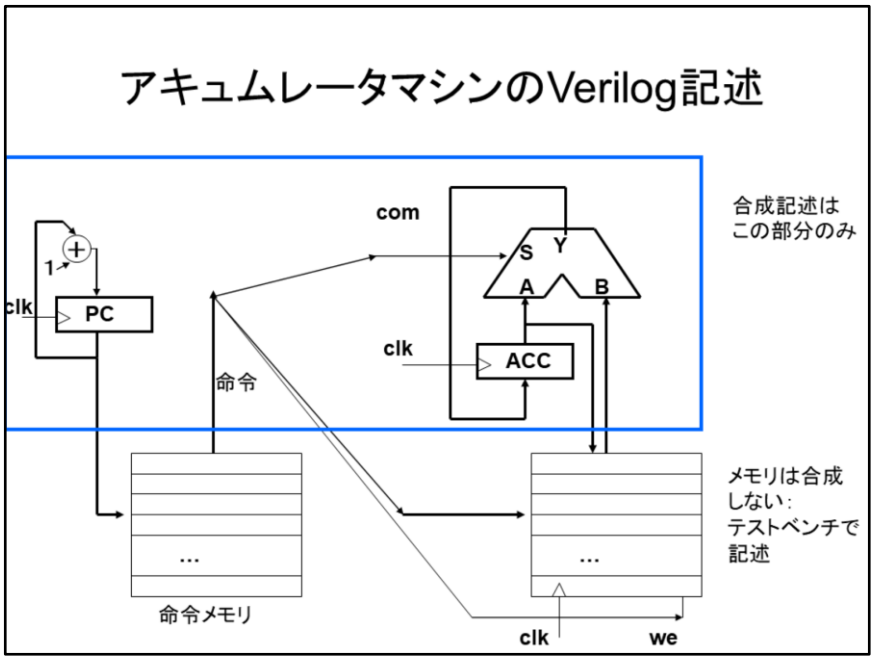
では、0番地の中身と1番地の中身を加算して2番地に格納する命令が実行される様子をアニメーションで示します。accum.vがアキュムレータの記述ですので、シミュレーションを動かしながら、スライドを見て、動きを掴んでください。最初の命令で0番地の内容をACCにロードします。この時PCは0から1になります。



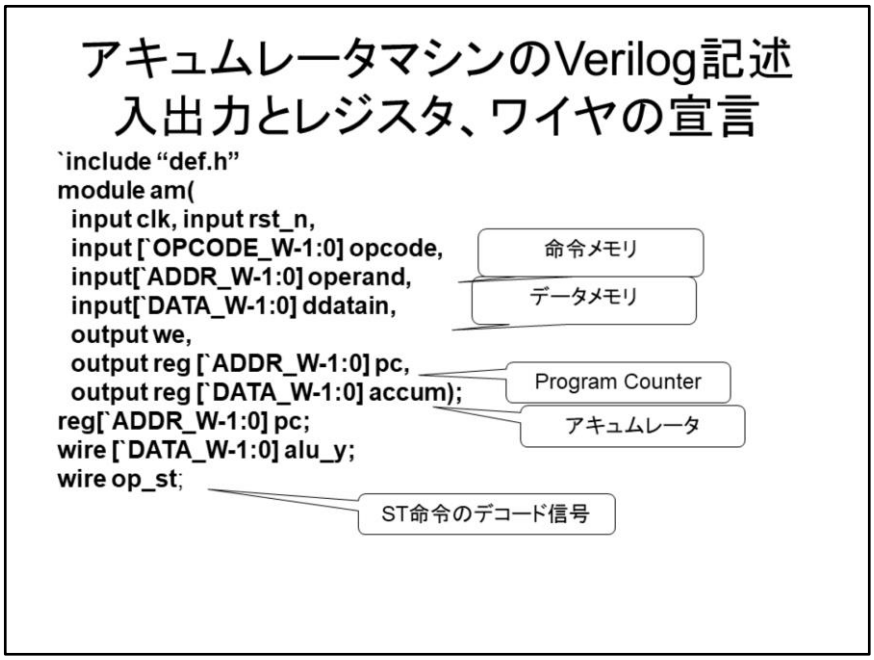
次にPCに従って1番地の命令ADD 1を読み出します。これによりACCの値とデータメモリの1番地の値が加算されます。同時にPCは一つ増えて2になります。



次にPCに従って2番地の命令が読み出されます。この命令は最初のビットが1なのでwe=1となり、ACCの内容が2番地に格納されます。



ではアキュムレータマシンのVerilog記述を解説しましょう。まず、CPUの記述からメモリを分離します。先に紹介しましたように、メモリは普通のディジタル回路とやや違っています。電子回路基礎で紹介したように特殊なチップや特殊な回路を使います。そこで、合成の対象とするのは、この四角で囲った部分だけにします。



ではアキュムレータマシンのVerilog記述を説明しましょう。まず入出力は、clk, rst_n, メモリに対する入出力になります。命令メモリからの入力はopcodeとoperandに分けて入力することにします。アドレスにはpcを繋ぎます。データメモリからの入力はddatain、書き込みはaccumを使います。これに書き込み制御のweが加わります。内部信号としてレジスタでpcを宣言します。これは出力名と同じにしてやると繋がなくて済みます。alu_yは以前説明したALUの出力です。このアキュムレータマシンはST命令だけがやや特殊な動きをするので、これを検出してやります。今の命令がST命令のときop_stがHになります。

opcodeがST命令に相当するときに、op_stをHにします。これはdef.h中にST命令のパターンを入れておいて比較します。次にALUを接続します。これは今までと同じなのですが、comにはopcodeの下位3ビットを入れてやります。一番上のビットはST命令用なのでALUとはとりあえず関係ないからです。

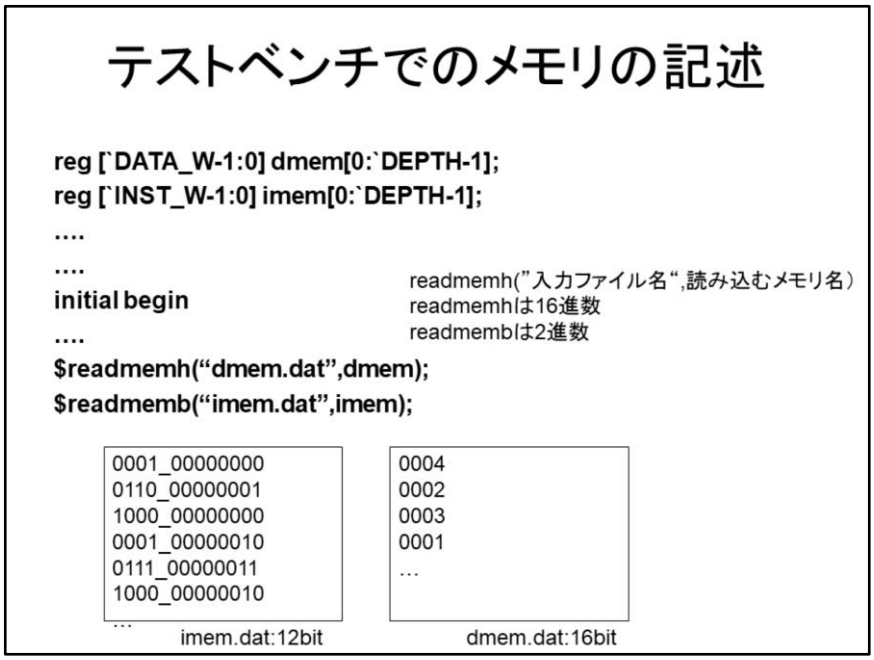
アキュムレータマシンのVerilog記述 レジスタの制御

```
always @(posedge clk or negedge rst_n)
 begin
  if(!rst_n) pc <=0;
  else pc <= pc+1;
 end

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

endmodule
```

pcの制御

accの制御

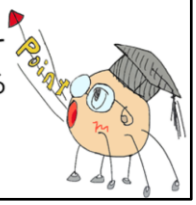
次にPCの制御ですが、今回リセット信号rst_nがLの時に初期化をし、それ以外は毎クロック1ずつ増やします。このことにより、次々に命令メモリを読み出します。アキュムレータの記述は以前と同じですが、ST命令では、accumにALUの出力を入れないようにしています（本当はTHAでaccumの値がalu_yに出てくるのでこれは不要ですが、後のためです）



では、テストベンチ中のメモリの記述を見てみましょう。今回はデータメモリ、命令メモリ共に初期設定が必要です。命令メモリには実行する命令、データメモリには計算対象のデータを入れておきます。ここで、$readmemhは16進数で指定したファイル（dmem.dat)から、データメモリに対してシミュレーション実行時にデータを読み込みます。一方、命令メモリに対しては、imem.datというファイルから$readmembを使って2進数で命令を設定します。imem.dat, dmem.datはあらかじめ設定しておく必要があります。imem.datを書き換えることで命令を、dmem.datを書き換えることで計算対象のデータを書き換えます。では、iverilog test_am.v am.vでコンパイル、vvp a.outで実行して結果を確かめましょう。

本日のまとめ

- ALUとアキュムレータを組み合わせてデータパスを作る
- 中間データ、入力データ、結果を入れておくためにデータメモリを使う
- 制御信号、データメモリのアドレスを命令の形にまとめることができる
  - 命令は、操作の内容を示すオプコードと操作対象のオペランドからできている
  - 命令を1・0パターンで表したものを機械語と呼ぶ
  - ニーモニックを使って人に読める形にしたのはアセンブリ表記と呼ぶ
- 命令メモリに命令を入れておき、PCを使って読み出す
  - 自動的に順番に命令メモリから命令を読み出して実行できる
  - プログラム格納型計算機まであと一歩だ！



インフォ丸が教えてくれる今日のまとめです。

今日のVerilog 構文

- レジスタ構文
  - reg [MSB:0] 信号名
- always @(posedge clk or negedge rst_n)
  - この授業ではこの形しか使わない
- ブロッキング代入文

  accum <= alu_y; レジスタの値を代入する
- if文
  - C言語とほとんど同じ
  - always文の中でのみ使える
- メモリ構文
  - reg [MSB:0][0:最大番地]
  - $readmemh, $readmembでファイルから値を初期化

今回も新しい構文をたくさん紹介しました。しかし大体終わりが近づいています。

演習3.1

– Aを0番地、Bを1番地のデータとして(A+B) OR (A-B)のデータを2番地にしまう命令の実行をプログラムしてミュレーションせよ　提出物は imem.dat

演習は二つあります。片方はコンピュータとは関係ないもので、Veirlogの練習です。
もう片方はimem.datを書き直します。これを提出してください。