第3回　データパスの基本

簡単なデータパス

計算を行うということ

前回設計した ALU で X+Y を計算するのは極めて容易である。両方の入力に X と Y を入れ、S に 110 をセットすれば良い。

`X+Y-W+Z`

という計算を ALU でどのように行えば良いのだろう。図 1 のように ALU を組み合わせる方法が考えられる。しかし、このようなことをしていくと、式が長くなると、どんどん利用しなければならない ALU 数が増えてしまう。

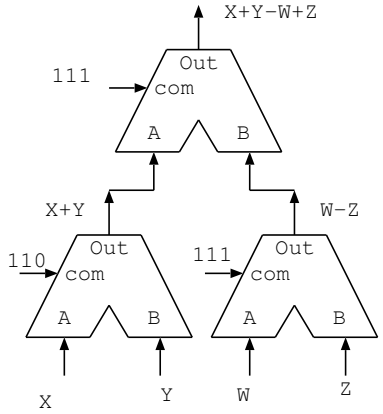

図 1: ALU の接続

そこで、数を記憶しておくレジスタというハードウェアを導入する。このレジスタには、D-FF という記憶素子を使うの。これについては計算機基礎でも習ったと思うが、ここでは、入力したクロックが L から H に変化した際に、入力データを記憶する装置だと考えてほしい。

図 2 に示すように、ALU の片方の入力にレジスタの出力を入れ、レジスタの入力に ALU の出力を接続する。すなわち、計算結果をレジスタに格納し、それを再び ALU の入力として計算に使えるようにするわけである。

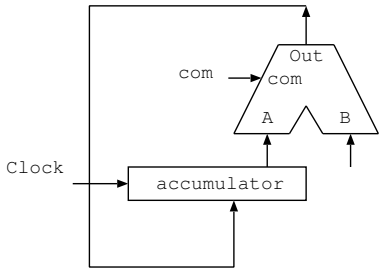

図 2: 最も簡単なデータパスの構成

X+Y-W+Z をこのデータパスで実行するには以下のようにすれば良い。ただし、一行実行した所でクロックを変化させてレジスタに値をセットすると考える。

`S=001 B=X`　レジスタ：`X`
`S=110 B=Y`　レジスタ：`X+Y`
`S=111 B=W`　レジスタ：`X+Y-W`

```
S=110 B=Z  レジスタ：X+Y-W+Z
```

このようにすると、ひとつの ALU を使い回すことが可能になる。計算結果はレジスタに置かれて、更新される。つまりこのレジスタは結果が積み重なる場所となり、アキュムレータ (accumlator) と呼ばれる。アキュムレータは電卓の表示された値に相当する。また、この一行めは、最初に計算すべき値をアキュムレータにセットする役割を果たす。このように外部からレジスタに値をセットする操作をロード (load) 操作と呼ぶ。このような計算のための装置をデータパス (datapath) と呼ぶ。

always 文を使ったレジスタの記述

今まで紹介した回路を verilog で記述する方法を示す。まず、レジスタであるアキュムレータを定義する必要があるが、これは reg 文を用いる。

```
reg [‘DATA_W-1:0] accum;
```

で、DATA_W ビットのレジスタ accum が定義される。前回通り DATA_W を 16 ビットとすれば 16 ビットのレジスタが定義されることになる。次に、クロックが立ち上がった時に、ALU の出力を accum に書き込ませるために always 文を用いる。always 文はその名前の通り、@の後の括弧内に示す waiting list 中の信号が変化すると「いつでも」、begin.. end 内の動作が行われる。

```
always @(waiting list...)
 begin
..
 end
```

ここで、clk が立ち上がった際の動作は以下のように書く。ここで、posedge は立ち上がりエッジでの動作、negedge は立ち下がりエッジでの動作を示す。

```
always @(posedge clk)
 begin
..
 end
```

さて、ここではリセット入力 (rst_n) を持たせて、この入力が L レベルになったら accum をクリアし、それ以外ではそれぞれの clk の立ち上がりエッジで ALU の出力の値をセットする記述を示そう。

```
module datapath(
input clk, input rst_n,
input [‘DATA_W-1:0] datain,
input [‘SEL_W-1:0] com,
output [‘DATA_W-1:0] accout);

reg [‘DATA_W-1:0] accum;
wire [‘DATA_W-1:0] alu_y;

assign accout = accum;
alu alu_1(.a(accum), .b(datain), .s(com), .y(alu_y));
```

```
always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) accum <= 'DATA_W'b0;
    else accum <= alu_y;
end
endmodule
```

ここでは、rst_n はアクティブ L であるため、_n を末尾につけてある。このように verilog ではアクティブ L の信号線には_n などを付けてこれをはっきりさせる習慣があり、この授業でもお勧めする。さて、waiting list を見ると、clk の立ち上がりと rst_n の立ち下がりのどちらかで begin end 内の文が動作するように or となっている。この or は、waiting list 内の信号名あるいはその動作のみに利用する特殊な記号である点に注意されたい。この場合、レジスタは、rst_n が L になるときはいつでも、即座に 0 になる。この方式を非同期クリア（リセット）と呼ぶ。rst_n が H の時が通常動作であり、この時はそれぞれのクロックの立ち上がりで、ALU の出力が格納される。

```
always @(posedge clk)
begin
    if(!rst_n) accum <= 'DATA_W'b0;
    else accum <= alu_y;
end
```

rst_n を waiting list に入れなければ、レジスタは、rst_n が L レベルの時にクロックが立ち上がったらクリアされる。これを同期クリア（リセット）と呼ぶ。レジスタやフリップフロップのクリアを同期で行うか、非同期で行うかは、利用するチップに依存するが、多くの場合非同期リセットの方が、リセット線の遅延差によるトラブルを避けることができ、有利である。

クロックの発生

次にこれをテストするテスト用のシミュレーション用記述を作ろう。このように、テストしたい回路にどのような入力を与えるか、どこで出力するかを記述するファイルをテストベンチと呼ぶ。

まず、今までと違って、クロック信号を生成してやる必要がある。ここでは clk という名前でテストベンチ内で reg 文で宣言する。

```
parameter STEP = 10;
   reg clk;

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

この記述では、STEP/2 つまり 5nsec 経過する度に、clk が反転する。すなわち、全体で 100MHz のクロックが発生される。ただし、clk は initial 文で、0 か 1 に初期化する必要がある。

入力信号の発生

次に initial 文を使って、datapath への入力を制御する。最初は、シミュレーション開始時の初期化である。最初の 2 行は、gtkwave で波形を見るための記述である。$dumpfile でファイル名 (ここでは datapath.vcd) を指定する。次に$dumpvars で、時間 0 から module test、つまりこのテストベンチのモジュール以下全体を記録して見られるようにする。

```
initial begin
   $dumpfile("datapath.vcd");
   $dumpvars(0,test);
   clk <= `DISABLE;
   rst_n <= `ENABLE_N;
   datain <= `DATA_W'h0003;
   com <= `ALU_THB;
#STEP
   rst_n <= `DISABLE_N;
```

次に、clk, rst_n を指定する。1'b1, 1'b0 などと書いても良いのだが、ここでは、分かりやすさを重視して、あらかじめ define しておいた記号を使う。L レベルで稼働 (enable) か H レベルで稼働かは、信号毎に異なるので下のように区別しよう。

```
`DISABLE   0   H レベルで稼働時
`ENABLE    1   H レベルで稼働時
`DISABLE_N 1   L レベルで稼働時
`ENABLE_N  0   L レベルで稼働時
```

同じ 0 や 1 でも意味が異なることを明確に示すことができる。このような define はすべてのファイルで共通なので、まとめて一つのファイルに入れておき、include 文で指定する。ここでは def.h というファイルに入れておき、test.v, datapath.v alu.v の 3 つの記述で指定している。この def.h は、シミュレーション時に同じディレクトリに置いておく必要がある。

次に datapath に与える datain と com の値も定義しておく。最初はデータとしては 3、コマンドはスルー B を設定する。リセットは、最初 0 にしておき、1 クロック分時間を進めてから 1 にする。これで、データパスは動作を始めるはずだ。

```
#(STEP*1/2)
   $display("com:%h datain:%h accout:%h", com, datain, accout);
```

では、次にクロックが立ち上がるまでの半クロック分の時間を進めてから表示してみる。これで最初の結果が確認できる。スルー B はロード操作に相当するので、3 がアキュムレータにセットされているはずである。

以下、同様に datain と com を与えて、演算を進める。ここでは、

- 5 を足す
- 2 を足す
- 3 を AND

```
#(STEP*1/2)
   datain <= `DATA_W'h0005;
   com <= `ALU_ADD;
#(STEP*1/2)
   $display("com:%h datain:%h accout:%h", com, datain, accout);
#(STEP*1/2)
  datain <= `DATA_W'h0002;
#(STEP*1/2)
  $display("com:%h datain:%h accout:%h", com, datain, accout);
```

```
#(STEP*1/2)
    com <= ‘ALU_AND;
  datain <= ‘DATA_W’h0003;
#(STEP*1/2)
  $display("com:%h datain:%h accout:%h", com, datain, accout);
#(STEP*1/2)
  $finish;
end
```

演習 3

前回改造した ALU を使って、以下の処理を行うテストベンチを作れ。

- 3 をロード
- 5 を足す
- 2 を引く
- 1bit 左シフト