第4回　メモリの導入

データメモリの取り付け

前回設計したデータパスでは、アキュムレータを使うことで、演算した結果に対して次々に演算を施して行くことができた。しかし、以下のような演算はスムーズに行うことができない。

```
(X<<1)+(Y<<1)
```

上の式ではまず X をアキュムレータに入れて 1 ビット左シフトし、その結果をどこかにとっておいて、それから Y を左シフトして加えなければならない。つまり、結果をとっておく場所が必要であり、これを可能にするのがメモリである。

デバイスとしてのメモリはディジタル回路の時間に紹介するので、ここでは、メモリは単純に図 1 に示す表と考えて欲しい。この表は、各行に番号（アドレス）が振られており、アドレスの値によって、対応する行の中身（データ）を読み出すことができる。この場合、アドレスに 0 を設定すれば 1001 が、1 を設定すれば 1100 が読み出され、Dataout に出力される。

また、書き込みの場合、入力にデータを置き、アドレスを設定し、書き込み制御信号 we(write enable) を H にし、クロックを立ち上げると、そのアドレスにデータが書き込まれる。

この例は超簡単で、幅が 4 ビットで深さが 2^4=16 のメモリということになり、全容量は 64bit である。深さが 16=2^4 であるため、番地を識別するアドレス線も 4 本で良い。もちろん、実際のメモリはもっとずっとずっと大きく、アドレスが 32bit-48bit、データは 32bit-64bit で膨大な情報量を格納する。しかし原理は同じである。アドレスが m bit あれば、2^m 行分指定することができ、容量の大きいメモリほど、アドレスの本数も多い。

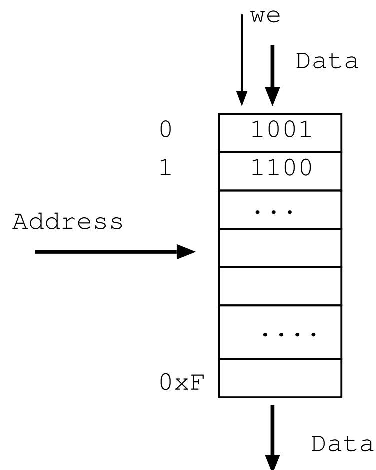

図 1: メモリの構成

それでは、深さ 2^8=256、幅 16bit のメモリを想定し、これを前回のデータパスにつないでみよう。この様子を図 2 に示す。

この構成では、データパスの入力にはメモリの出力、アキュムレータ出力にはメモリの入力が接続されている。すなわち、データを外から与えてやる代わりにメモリ中にあらかじめ格納してデータに対して演算を行い、途中結果もメモリに格納することができる。外部から与えるのは ALU の com とメモリのアドレスおよび we である。

例えば X が 0 番地、Y が 1 番地に格納されている場合、先の例は以下の値をクロックに同期して順に与えれば計算できる。

```
we com Address
0  001 00000000  : 0番地(X)を読み出し、ALUはスルー(LD)
```

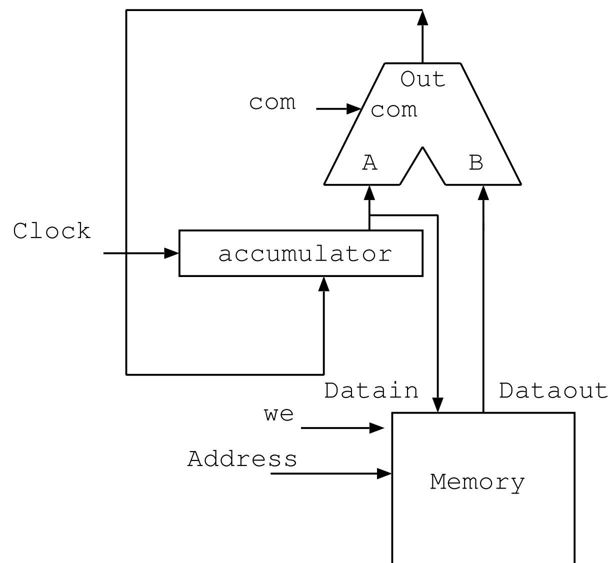

図 2: メモリ付きデータパス

```
0  100 00000000  : 1bit 左シフト (Address はここでは何でも良い)
1  000 00000010  : 2 番地に X<<1 を格納 (ST)
0  001 00000001  : 1 番地 (Y) を読み出し、ALU はスルー (LD)
0  100 00000000  : 1bit 左シフト (Address はここでは何でも良い)
0  110 00000010  : 2 番地の値 (X<<1) と加算
1  000 00000011  : 3 番地に答えを格納 (ST)
```

ここでは 2 番地を途中結果を格納する場所として使い、最終結果は 3 番地に格納している。ここで、we を 1、com を 000 とすると、アキュムレータの内容が Address に示す番地に格納される。この操作を Store（ストア）と呼ぶ。Store は Load の逆の操作であり、この言い方は様々なコンピュータで広く用いられる。

ここで、we と com をくっつけてしまって 4 ビットとし、命令コード (op-code) と呼ぶことにする。すると上の操作は以下のように記述することができる。

```
opcode Address
0001 00000000  : LD 0
0100 00000000  : SL
1000 00000010  : ST 2
0001 00000001  : LD 1
0100 00000000  : SL
0110 00000010  : ADD 2
1000 00000011  : ST 3
```

ここで、opcode に対応する ALU の操作および LD/ST を、対象とするアドレスと共に記述してやると、操作を見やすく示すことができる。これがコンピュータの命令 (Instruction) と、そのアセンブリ表記の第一歩である。

命令メモリの取り付け

上のデータパスでは、計算するデータはデータメモリにあらかじめ入れておけば良いのに、命令は外からクロックに同期して与えてやらなければならず、これは大変である。そこで、データ同様、命令もあらかじめメモリに入れておき、これを順番に読み出してデータパスに与えるようにすれば便利である。この構成を図 3 に示す。

ここでは、深さ 2^8=256、幅 12bit の命令メモリを想定し、ここに命令を入れておく。これを順に読み出すためのしかけとして、PC(Program Counter:プログラムカウンタ) を用意する。PC はスタート時に 0 に初期化しておき、クロックが立ち上がる毎に 1 ずつ増やしてやる。そうすると、命令メモリに格納されている命令が順にデータパスに与えられて、所定の処理が順に実行されていく。

読み出された来た命令の opcode に対応する部分は最上位ビットが 1 の場合は、ST 命令であることを示す。この際、we=1 としてデータメモリに accum の値を書き込む。また、ST 命令の時は accum に ALU の計算結果を書き込まない。このため、accum に accset というセット入力を与え、これが 1 の時のみ、結果を書き込むようにする。[1]

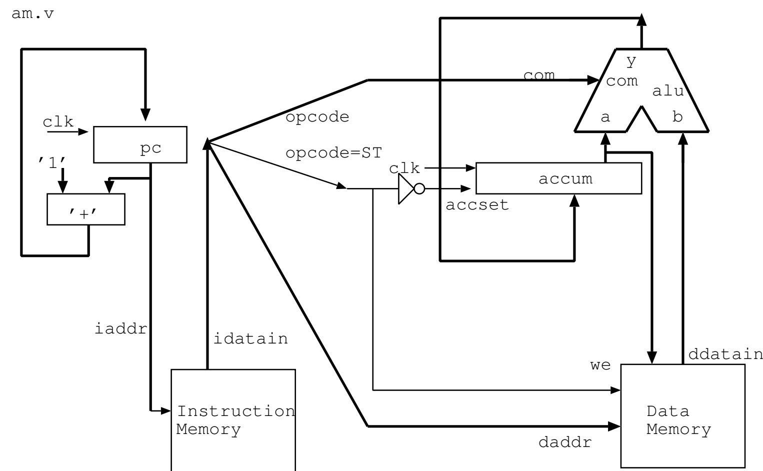

図 3: 命令メモリ付きデータパス

これが、プログラム格納型計算機の第一歩である。

メモリ付きデーパスの verilog 記述

メモリはレジスタの配列として考えて、以下のように reg 文で宣言する。

```
reg [MSB:LSB] 名前 [深さ]
```

例えば、深さ DEPTH でデータ幅 DATA_W のメモリ dmem を宣言するには

```
reg [`DATA_W-1:0] dmem[`DEPTH-1:0]
```

とする。それでは、図 3 の verilog 記述を説明しよう。

ここでは、図の構造がほとんどそのままの形で記述されている。命令メモリから読み出された来た命令は、opcode とデータメモリに対するアドレス (daddr) に分離される。この opcode が ST 命令のときに書き込みが行われるようにしている。逆に accum には、ST 命令以外で alu で演算された値が格納されるようになっている。pc はリセット時に 0 になり、後は順に 1 ずつ増えて行く。すなわち、0 番地から順に命令が実行される。

```
`include "def.h"
module datapath2(
input clk, input rst_n,
output [`DATA_W-1:0] accout);

reg [`DATA_W-1:0] accum;
```

[1] com の下がすべて 0 ならば A 入力がスルーされて y から accum に書かれるので実は現時点では、この accset は必要ない。しかし後でどのみち必要になるので付けておくことにする。

```
reg [`ADDR_W-1:0] pc;
reg [`DATA_W-1:0] dmem [`DEPTH-1:0];
reg [`INST_W-1:0] imem [`DEPTH-1:0];

wire [`DATA_W-1:0] alu_y;
wire [`DATA_W-1:0] ddatain;
wire [`OPCODE_W-1:0] opcode;
wire [`INST_W-1:0] idatain;
wire [`ADDR_W-1:0] daddr;
wire we;

assign idatain = imem[pc];
assign {opcode, daddr} = idatain;

assign ddatain = dmem[daddr];
assign we = (opcode == `OP_ST);
always @(posedge clk)
     if(we) dmem[daddr] <= accum;

assign accout = accum;
alu alu_1(.a(accum), .b(ddatain), .s(opcode[`SEL_W-1:0]), .y(alu_y));

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(opcode !=`OP_ST) accum <= alu_y;
end

endmodule
```

では次に、これをテストするテストベンチ test.v を解説しよう。

```
`timescale 1ns/1ps
`include "def.h"
module test;
parameter STEP = 10;
   reg clk, rst_n;
   wire [`DATA_W-1:0] accout;

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

```
  end

  datapath2 dp2(.clk(clk), .rst_n(rst_n), .accout(accout));

  initial begin
     $dumpfile("datapath2.vcd");
     $dumpvars(0,test);
     $readmemh("dmem.dat", dp2.dmem);
     $readmemb("imem.dat", dp2.imem);
     clk <= `DISABLE;
     rst_n <= `ENABLE_N;
  #(STEP*1/4)
  #STEP
     rst_n <= `DISABLE_N;
  #(STEP*12)
  $finish;
  end

  always @(negedge clk) begin
     $display("pc:%h inst:%h acc:%h", dp2.pc, dp2.idatain, dp2.accum);
     $display("dmem:%h %h %h %h", dp2.dmem[0], dp2.dmem[1], dp2.dmem[2], dp2.dmem[3]);
  end
endmodule
```

ここでは、module datapath2 に dp2 という実体名を付け、以下の文で命令メモリとデータメモリの初期設定を行っている。

```
     $readmemh("dmem.dat", dp2.dmem);
     $readmemb("imem.dat", dp2.imem);
```

ここで readmemh は 16 進数で記述すること、readmemb は 2 進数で記述されたファイルから値を読み込んで、それぞれ dp2 中の dmem, imem に値を設定することを示す。

web 上のファイルでは imem.dat は

```
0001_00000000 // LD 0
0111_00000011 // SUB 3
1000_00000011 // ST 3
0001_00000001 // LD 1
0110_00000010 // ADD 2
0011_00000011 // OR 3
1000_00000011 // ST 3
```

になっている。opcode とアドレスを区切るのに_が使われているが、これは無視される。これに対して dmem.dat は 16 進数で、

```
0009
0005
0003
0002
```

となっている。実行して、結果を確認しよう。accum に答えが入るのは命令が実行された次のクロックである点に注意されたい。

演習 4:

X,Y,W,Z が 0,1,2,3 番地に格納されているとする。(X-Z) AND (Y-W) を演算する命令コードを書き、実行して答えを確認せよ。なお、答えは accum 上に出た時点で終わりにして良い。提出物は変更した imem.dat である。