第6回　プログラム格納型への道 その2

分岐命令

命令の定義

前回、定義して実装した命令は以下の通りである。

```
0000XXXXXXXX   NOP
0001XXXXXXXX   LD X
0010XXXXXXXX   AND X
0011XXXXXXXX   OR X
0100--------   SL
0101--------   SR
0110XXXXXXXX   ADD X
0111XXXXXXXX   SUB X
1000XXXXXXXX   ST X
```

上記の命令を順に取ってきて（Fetch)、実行（Execution）することにより、一連の処理を自動的に実行できることがわかった。しかし、このアキュムレータマシンは、命令メモリに格納された命令を順番に実行するに過ぎない。前回の課題のプログラムとて、実行が終わったら、次の番地の命令の実行に移ってしまい停止することすらできない。すなわち、判断を行い、実行する命令を変更する機能がなければ、計算アルゴリズムを実行することができず、つまりコンピュータとは言えない。では、コンピュータとするためにはどうすれば良いだろう？

最も簡単な方法としては、アキュムレータの内容によってプログラムカウンタを変更する方法が考えられる。これが分岐 (Branch) 命令である。ここでは以下の命令を導入する。

```
機械語          命令
1010 XXXXXXXX     BEZ X (Branch Equal Zero) X
1011 XXXXXXXX     BNZ X (Branch Not equal Zero) X
```

BEZ 命令は、アキュムレータの内容が 0 ならば、PC の内容を XXXXXXXX に変更する。BNZ 命令は、アキュムレータの内容が 0 でなければ、PC の内容を XXXXXXXX に変更する。条件に合わない場合は何もせず、次の命令が実行される。

分岐命令の導入によって、アキュムレータの内容によって、動作する命令を変更して、反復計算をすることが可能である。

以下がその例である。この例は、2 番地の内容を m、3 番地の内容を n とし、m を n 回加算するプログラム、つまり乗算のプログラムである。答えは 0 番地に格納される。ただし、0 番地を 0 に、1 番地を 1 に初期化しておく必要がある。アキュムレータが 0 になるまで、6 番地の分岐命令によって 0 番地まで戻り、反復して実行する。その度に 3 番地の内容である n が 1 ずつ減っていき、0 になると 7 番地の内容が実行される。

```
番地  命令
0     LD 0
1     ADD 2
2     ST 0
3     LD 3
4     SUB 1
5     ST 3
6     BNZ 0
```

この場合、反復が終了した時には、アキュムレータの内容が 0 になっており、7 番地の分岐命令は無限ループに陥る。これをダイナミックストップと呼び、これによってプログラムは平和に停止する。

分岐命令を実行するためには、(1) フェッチした命令が分岐命令であり、(2) 条件が成立する場合に限り、PC に命令のオペランドの部分（飛び先）をセットできるようにすれば良い。

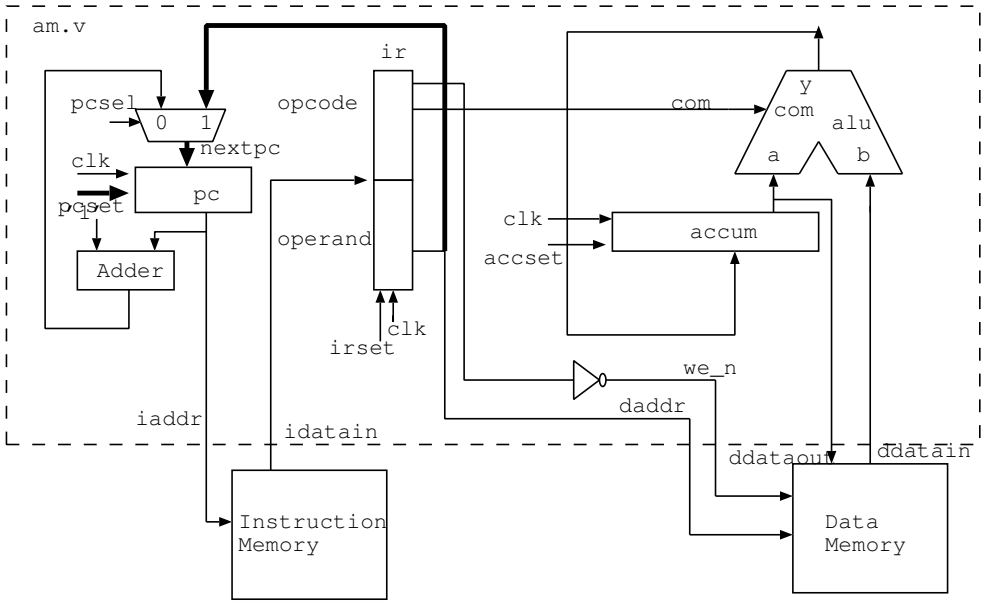

図 1: アキュムレータマシンの構成 (分岐命令付き)

図 refbranch にこの改造したアキュムレータマシンのハードウェア構成を示す。台形印は、マルチプレクサあるいはデータセレクタというハードウェアモジュールで、この場合、2 つの入力の片方を、選択信号 pcsel に応じて選んで出力する。ここでは、pcsel=1 ならば、pc に operand を送り、S=0 ならば、pc には以前通り pc+1 を送る。マルチプレクサは、今後データパスを複雑にしていく場合不可欠なものである。また、pc, ir, accum には、毎クロック値を書き込むのではなく、値を書き込む場合にはそれぞれの set 信号を H にする必要がある。すなわち、set 信号が H である場合に限り、クロックの立ち上がりでデータがセットされる。太線に BEZ, BNZ 実行時のデータの流れを示す。

さて、Verilog で回路を記述する場合、このマルチプレクサとセット信号を明示的に記述するかどうかは悩む所である。前回の Verilog 記述を見ると、pc、ir には IF 状態の時に、accum には EX 状態で ST 命令でない場合にのみデータが書き込まれている。制御が単純であるため、always 文中で set 信号を意識せずに記述していた。

図 2 を直接 Verilog で記述すると以下のようになる。nextpc を生成する文がマルチプレクサに相当する。分岐命令で pc がセットされるのは、当然、EX 状態のみである点に注意されたい。IF 状態では以前同様に pc はカウントアップされる。

```
wire pcsel, pcset;
wire [‘INST_W-1:0] nextpc;
assign pcsel = stat[‘EX] & (opcode == ‘OP_BEZ & acc == 16’b0 ) |
                              (opcode == ‘OP_BNZ & acc != 16’b0));
assign nextpc = pcsel ? operand : pc + 1;
assign pcset = pcsel | stat[’IF];

always @(posedge clk or negedge rst_n)
begin
```

```
   if(!rst_n) pc <= 0;
   else if(pcset)
     pc <= nextpc;
end
```

しかしこれは、様々な中間的な信号を定義しなくてはならず、見通しが悪い。

```
wire pcsel;
assign pcsel = stat[‘EX] & (opcode == ‘OP_BEZ & acc == 16’b0 ) |
                              (opcode == ‘OP_BNZ & acc != 16’b0));

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(pcsel)
     pc <= operand;
   else if(‘stat[‘IF])
     pc <= pc+1;
end
```

の方が見通しが良い。これは、マルチプレクサの機能を if 文中の条件で表してしまっている。しかし一方で、

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(stat[‘EX] && (opcode == ‘OP_BEZ && acc == 16’b0) |
                    (opcode == ‘OP_BNZ && acc != 16’b0)))
     pc <= operand;
   else if(‘stat[‘IF])
     pc <= pc+1;
end
```

とやると、さらに見通しが良いかというと、これは条件が複雑すぎて逆に読みにくくなっている。この授業では、if 文の条件があまり複雑な場合は、wire 文を設けて整理する方法を採用している。すなわち二番目の方法である。しかし、ここに挙げた三つの書き方はどれも正しいので、自分が見やすいと思う、好きなスタイルで記述されたい。

ここで、ST 命令同様、BEZ や BNZ では accum の値は変更されない。この点を忘れずに変更する必要がある。

```
always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) accum <= 0;
    else if(stat[‘EX]) begin
      if((opcode !=‘OP_ST)&&(opcode != ‘OP_BEZ)&&(opcode!= ‘OP_BNZ))
                              accum <= alu_y;
    end
end
endmodule
```

すなわち、ST でも BEZ でも BNZ でもない命令では、alu で計算した値を accum にセットする。

イミーディエイト命令

今までの演算命令では、オペランドはメモリのアドレスであった。このため、例えば、1 を引くという操作を行う場合、先のプログラムで示したようにあらかじめ 1 番地に 1 という数字を初期化操作で入れておく必要がある。これでは不便であるため、コンピュータの命令セットにはイミーディエイトまたは直値（即値）というオペランドの使い方が用意されている。この方法では、オペランドがそのまま計算に使われる。例えば以下の命令である。

```
機械語          命令
1100 XXXXXXXX     ADDI #X (ADD Immediate) X: accum <- accum + X
```

ADDI は、アキュムレータの内容からオペランドに書いた値を直接加算する命令である。これを使うと乗算のプログラムは以下のようになる。この場合、1 番地を 1 に初期化する必要はない。

```
番地  命令
0     LD 0
1     ADD 2
2     ST 0
3     LD 3
4     ADDI #-1
5     ST 3
6     BNZ 0
7     BEZ 7
```

ちなみに、アセンブラ表現では、イミーディエイト命令であることを強調するために、計算に使われるオペランド部分に井桁印を付ける習慣がある。

ここで一つ問題がある。今、accum に格納されているデータは 16 ビットであるのに対して、オペランドは 8 ビットしかないので、データ幅が合わない。今、上記の例に従って、ADDI #-1 の機械語は、-1 を 2 の補数表現で表すため、1100 11111111 となる。ここで、上位 8 ビットを単純に 0 で埋めると、0000000011111111 となり、16 ビットの数値としては+127 となってしまう。すなわち、この場合、最上位ビットが 1 ならば 1 を、0 ならば 0 を上位 8 ビットに入れて値を作る必要がある。これを符号拡張 (sign extension) と呼ぶ。8bit の-1 を符号拡張すると、1111111111111111 となり、16 ビットでも-1 として扱うことができる。

イミーディエイト命令を実装するには、ALU の B 入力に対してマルチプレクサを付けてやり、メモリからのデータだけではなく、オペランドを入れてやることのできるようにする。次に、ALU の演算を ADDI 命令実行時に加算 (110) に設定してやる。これを付けた構成を図 2 に示す。太線は、ADDI 命令実行時のデータを動きを示す。

先に分岐命令の例を示したように、命令が増えてくると、命令に応じてデータパスのデータの流れを制御する必要がある。このため、opcode の部分を解読（デコード）して、状態と合わせて制御を行う必要がある。図 3 に制御信号を含めたアキュムレータマシンのコントローラの状態遷移図を示す。

さて、上の図を念頭に置いて、Verilog 記述を考える。まず、符号拡張だが連結の所でやったように以下のようにする。

```
{{8{operand[7]}},operand}
```

この記述で operand の最上位ビットが 8 個分 operand 自体に連結される。次に ALU の b 入力用のマルチプレクサは以下のように記述する。

```
wire [‘DATA_W-1:0] alu_b;
assign alu_b = (stat[‘EX] & opcode == ADDI) ? {{8{operand[7]}},operand} :
                                              ddatain;
```

次に、ALU の演算を ADDI 命令実行時に加算 (110) に設定する記述は以下の通りである。

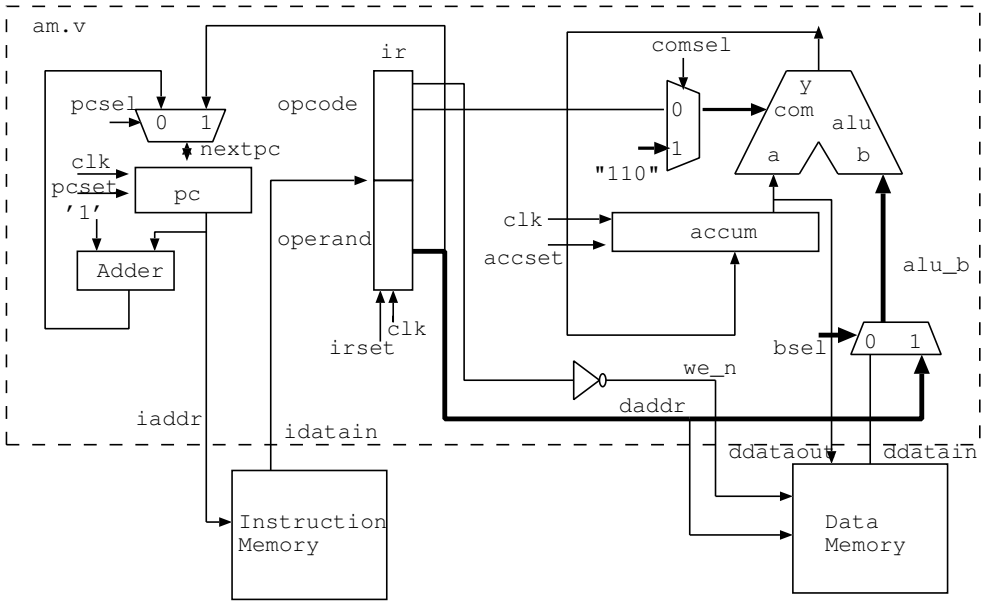

図 2: アキュムレータマシンの構成 (完成版)

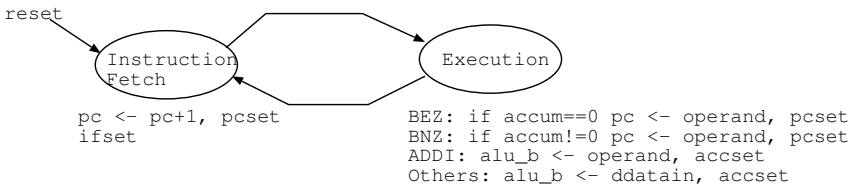

図 3: コントローラの状態遷移図

```
wire [‘SEL_W-1:0] com;
assign com = (stat[‘EX] & opcode == ADDI) ? 3’b110: opcode[‘SEL_W-1:0];
```

全体を書き直すと以下のようになる。

```
‘include "def.h"
module am(
input clk, input rst_n,
input [‘INST_W-1:0] idatain,
input [‘DATA_W-1:0] ddatain,
output [‘ADDR_W-1:0] iaddr, daddr,
output [‘DATA_W-1:0] ddataout,
output we_n);

reg [‘DATA_W-1:0] accum;
reg [‘ADDR_W-1:0] pc;
reg [‘INST_W-1:0] ir;
reg [‘STAT_W-1:0] stat;
wire [‘DATA_W-1:0] alu_y;
wire [‘OPCODE_W-1:0] opcode;
wire [‘ADDR_W-1:0] operand;
wire pcsel;

assign ddataout = accum;
assign daddr = operand;
assign {opcode, operand} = ir;
assign iaddr = pc;
assign we_n = ~(stat[‘EX] & (opcode == ‘OP_ST));
wire [‘DATA_W-1:0] alu_b;
wire [‘SEL_W-1:0] com;

assign alu_b = (stat[‘EX] & opcode == ADDI) ? {{8{operand[7]}},operand} :
                                              ddatain;
assign com = (stat[‘EX] & opcode == ADDI) ? 3’b110: opcode[‘SEL_W-1:0];
alu alu_1(.a(accum), .b(alu_b), .s(com), .y(alu_y));

assign pcsel = stat[‘EX] & (opcode == ‘OP_BEZ & accum == 16’b0 ) |
                              (opcode == ‘OP_BNZ & accum != 16’b0));

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(pcsel)
     pc <= operand;
   else if(stat[‘IF])
     pc <= pc+1;
end
```

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) ir <= 0;
   else if(stat[`IF])
     ir <= idatain;
end

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) stat <= `STAT_IF;
   else
    case (stat)
      `STAT_IF: stat <= `STAT_EX;
      `STAT_EX: stat <= `STAT_IF;
    endcase
end

always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) accum <= 0;
    else if(stat[`EX]) begin
      if((opcode !=`OP_ST)&&(opcode != `OP_BEZ)&&(opcode!= `OP_BNZ))
                              accum <= alu_y;
    end
end
endmodule
```

上記の記述は、それぞれの命令を解読 (デコード) している部分が分散している。そこで、少し格好を付けてデコード部分をまとめて見たのが以下の記述である。こちらの方が中間信号の名前は増えるが見やすいかもしれない。

```
`include "def.h"
module am(
input clk, input rst_n,
input [`INST_W-1:0] idatain,
input [`DATA_W-1:0] ddatain,
output [`ADDR_W-1:0] iaddr, daddr,
output [`DATA_W-1:0] ddataout,
output we_n);

reg [`DATA_W-1:0] accum;
reg [`ADDR_W-1:0] pc;
reg [`INST_W-1:0] ir;
reg [`STAT_W-1:0] stat;
wire [`DATA_W-1:0] alu_y;
wire [`OPCODE_W-1:0] opcode;
wire [`ADDR_W-1:0] operand;
```

```
wire pcsel;
wire st_op, bez_op, bnz_op, addi_op;

assign ddataout = accum;
assign daddr = operand;
assign {opcode, operand} = ir;
assign iaddr = pc;

// Decorder
assign st_op = stat[`EX] & (opcode == `OP_ST);
assign bez_op = stat[`EX] & (opcode == `OP_BEZ);
assign bnz_op = stat[`EX] & (opcode == `OP_BNZ);
assign addi_op = stat[`EX] & (opcode == `OP_ADDI);

assign we_n = ~st_op;
wire [`DATA_W-1:0] alu_b;
wire [`SEL_W-1:0] com;

assign alu_b = addi_op ? {{8{operand[7]}},operand} :ddatain;
assign com = addi_op ? 3'b110: opcode[`SEL_W-1:0];

alu alu_1(.a(accum), .b(alu_b), .s(com), .y(alu_y));

assign pcsel = (bez_op & accum == 16'b0 ) | (bnz_op & accum != 16'b0);

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(pcsel)
     pc <= operand;
   else if(stat[`IF])
     pc <= pc+1;
end

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) ir <= 0;
   else if(stat[`IF])
     ir <= idatain;
end
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) stat <= `STAT_IF;
   else
    case (stat)
      `STAT_IF: stat <= `STAT_EX;
```

```
      `STAT_EX: stat <= `STAT_IF;
    endcase
end

always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) accum <= 0;
    else if(stat[`EX])
      if (!(st_op|bez_op|bnz_op)) accum <= alu_y;
end
endmodule
```

シミュレーションに利用する。テストベンチは、前回とほとんど同じで良い。ただ、最後のシミュレーションの実行クロック数 (finish 文の前) のみを、場合に応じて変更する必要がある。例えば、

```
   #STEP
      rst_n <= `DISABLE_N;
   #(STEP*50)
   $finish;
```

とすれば、大方の場合は大丈夫かと思う。実装した am2.v で、掛け算のプログラムを実行してみよう。imem.dat を下のように設定する。

```
0001_00000000 // LD 0
0110_00000010 // ADD 2
1000_00000000 // ST 0
0001_00000011 // LD 3
1100_11111111 // ADDI #-1
1000_00000011 // ST 3
1010_00000000 // BNZ 0
1001_00000111 // BEZ 7
```

このプログラムでは 2 番地と 3 番地の掛け算を行って 0 番地に書き込むため、dmem.dat は、例えば以下のようにしておけば、2 × 3 が計算されるはずだ。

```
0000
0001
0002
0003
```

演習 6

1 番地に X が格納されている。X+(X-1)+...+1 を計算するプログラムを書け。