第5回　アキュムレータマシン

前回、pc を取り付けて、命令を一つづつ取り出して実行する機構を作った。これでプログラム格納型コンピュータの基本的な部分ができた。このコンピュータは、アキュムレータのみを持つ最も簡単なものでありアキュムレータマシンと呼ばれる。

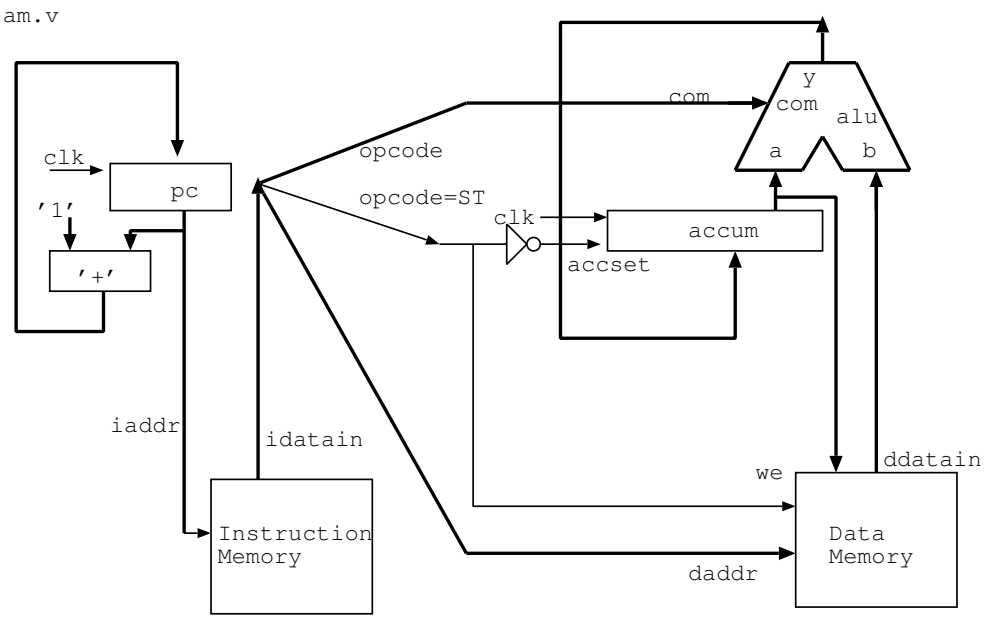

図 1: アキュムレータマシンの構成

図 1 にアキュムレータマシンの構造を示す。この動作を復習しよう。まず、Instruction Memory には pc がアドレスとして繋がっており、これによって指し示されたアドレスの命令が読み出さる。

読み出された命令が ST 命令以外ならば、Data memory から命令中の下位 8 ビットをアドレスとしてデータが読み出される。つまり LD 1 ならば 1 がアドレスに載る。そして読み出されたデータは ddatain から alu の b 入力に入る。alu の s 入力には、opcode の下位 3 ビットが入っている。LD 命令の場合、opcode は 0001 なので、com は 001 となり、THB が実行されて、読み出されたデータが、クロックの立ち上がリで、そのまま accum にセットされる。同時に pc には pc+1 がセットされ、次の番地の命令が指し示される。

読み出された命令が ADD 2 ならば、opcode は 0110 なので、com には 110 が与えられ、alu は a 入力と b 入力の加算を行う。a 入力には accum の値が常に入力され、b 入力は 2 番地から読み出したデータがやってくる。加算結果は EX 状態の終わりに accum に書き込まれる。

ちなみに、それぞれの命令で、accum に結果が設定されるのは、次のクロックの立ち上がりなので、この結果が利用可能になる（シミュレータ上で観測できる）のは、次の命令を実行するクロックサイクルになる。

メモリへの書き込みを行う ST 命令でのみ、データパスの動作はやや異なっている。ST 3 を実行する場合を図 2 に示す。ST 命令のアドレスは 00000011 であるので、最上位が 1 ならば ST 命令であるという判断ができる。ここで、Data memory の入力ポートには常に accum の出力が dataout を経由して接続されている。アドレスは他の命令同様、operand の値が与えられるため、この場合は 3 である。ここで we を H にすることで、data memory に accum の値を書き込む。

前回の設計で、以下の命令が利用可能である。X は番地の 2 進数を示し、-は使われないので何でも良いこと（ドントケアと呼ぶ）を示す。

```
ニーモニック     機械語          意味
NOP              0000 --------: No Operation: 何もしない
LD X             0001 XXXXXXXX: Load X: acc-m<- X 番地中の値
```

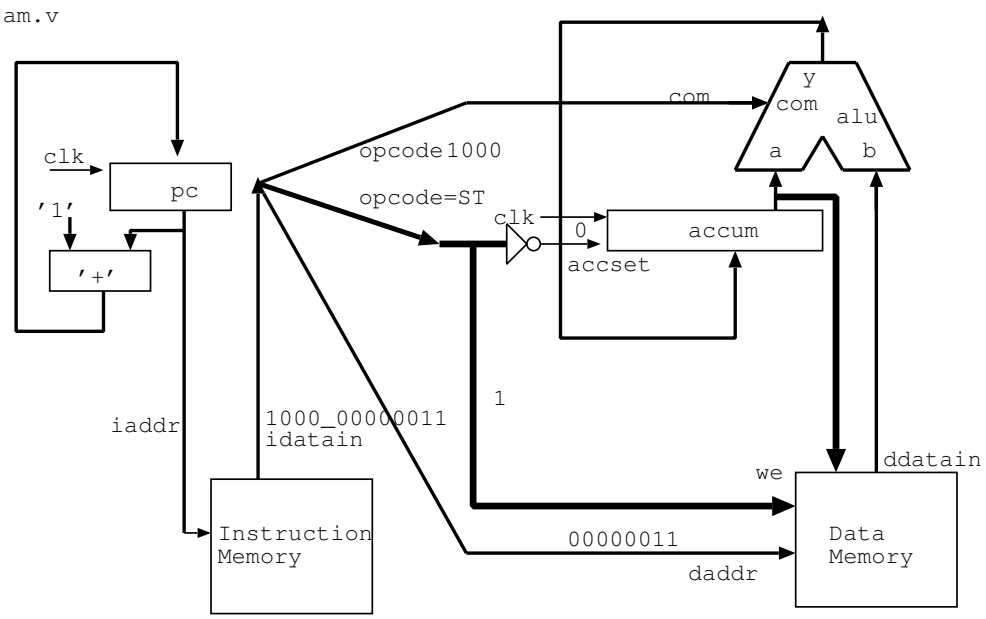

図 2: ST 命令の実行例

```
AND X          0010 XXXXXXXX: AND X: acc-m <- acc-m AND X 番地中の値
OR X           0011 XXXXXXXX: OR X:  acc-m <- acc-m OR X 番地中の値
SL             0100 --------: Shift Left: acc-m <- acc-m<<1
SR             0101 --------: Shift Right: acc-m <- acc-m>>1
ADD X          0110 XXXXXXXX: Add X: acc-m <- acc-m +  X 番地中の値
SUB X          0111 XXXXXXXX: S-b X: acc-m <- acc-m -  X 番地中の値
ST X           1000 XXXXXXXX: Store X: accum -> X 番地中
```

ここで、NOP はアキュムレータの値が自分自身に書き込まれるため、何もしない命令である。何もしない命令なんて意味があるんだろうか？と思われるかもしれないが、実はこの命令は時間稼ぎのために重要で、全てのコンピュータで装備されている。

さて、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う。完全な猪突猛進型で、演算処理の終了後に、停止することすらできない。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？

最も簡単な方法は、アキュムレータの内容を判断して、この結果によってプログラムカウンタを変更してしまうことである。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
1001 XXXXXXXX     BEZ X (Branch Equal Zero) X
1010 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
7     BEZ 7
```

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) 命令メモリから読み出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 に命令の下位 8 ビットの飛び先をセットできるようにすれば良い。

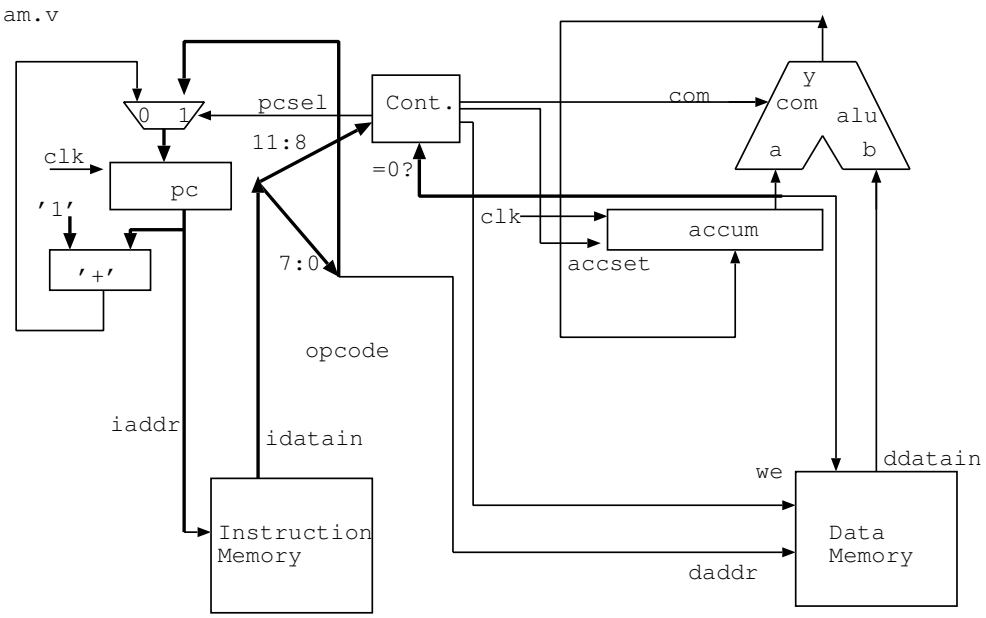

図 3: アキュムレータマシンの構成 (分岐命令付き)

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、accum には、accset 信号を L にして、分岐命令実行時にはデータをセットしないようにしてやる。この辺は ST 命令と同様である。太線に BEZ, BNZ 実行時のデータの流れを示す。まとめると、命令コードと accum の値に応じて pcsel, accset, we, com の値を以下のように制御する必要がある。

- ALU 演算命令 (LD,NOP を含む): accset=1, pcsel=0,we=0, com:opecode の下位 3 ビット
- ST 命令:accset=0, pcsel=0,we=1,com:ドントケア

- 分岐命令：accset=0, pcsel:条件が成立すれば 1、そうでなければ 0、we=0、com:ドントケア

この信号はコントローラ (Cont) で生成される。コントローラはここでは非常に簡単な組み合わせ回路で済むが、命令が増えるにつれて複雑になっていく。

イミーディエイト命令

今までの演算命令では、命令の下位 8 ビット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）という使い方が用意されている。この方法では、命令コードの下位 8 ビットはメモリのアドレスを示さず、そのまま数値として計算に使われる。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、アキュムレータに対して下位 8 ビットで示した数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。このような命令を直値命令あるいは即値命令と呼ぶが、どちらも発音しにくい上意味が取り難いせいか、英語読みでそのままイミーディエイト命令と呼ぶ人も多い。

```
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、計算に使われる直値を示す部分に井桁印を付ける習慣がある。さて、命令の下位 8 ビットは、メモリのアドレスを表す場合、分岐の飛び先を示す場合、イミーディエイト命令の直接演算するデータを示す場合の三通りが考えられる。いずれも演算や処理の対象を示すことから、この部分をオペランド（operand) と呼ぶ。すなわち、一般的に、命令コードは処理内容を示す opcode とオペランドから構成される。

ここで一つ問題がある。今、アキュムレータに格納されているデータは 16 ビットであるのに対して、命令中のオペランドは 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。ところが、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 ビットに入れて全体として 16 ビットの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、命令のオペランド部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 4 に示す。太線は、ADDI 命令実行時のデータを動きを示す。

コントローラは ADDI 命令の場合に以下のように制御線に値を与えれば良い。accset=1, op_addi=1, pcsel=0, we=0, com=110(加算)。ちなみに新しいマルチプレクサの制御線である op_addi は、ADDI 命令の時だけ 1 にする。

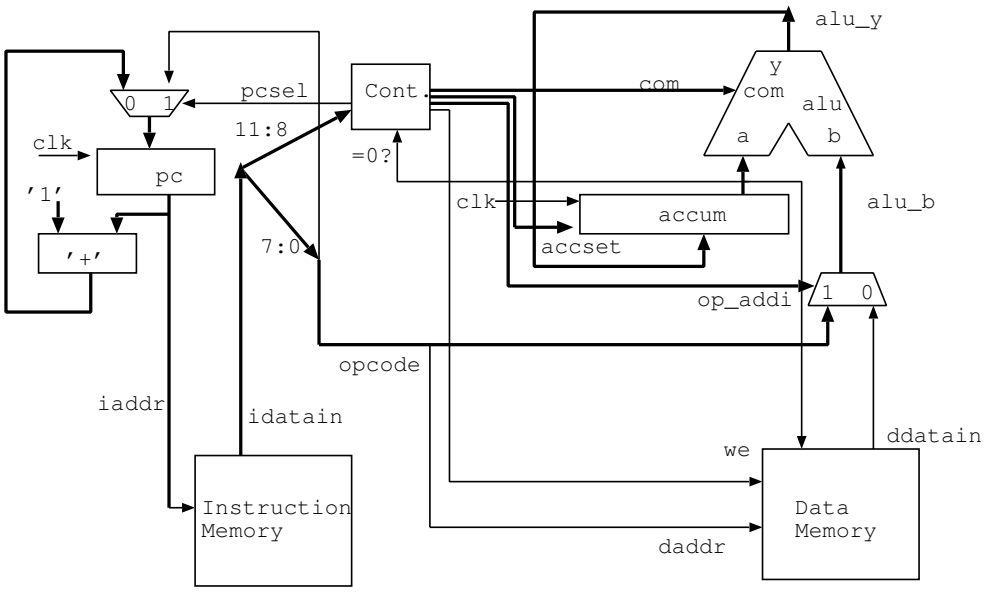

図 4: アキュムレータマシンの構成 (完成版)

Verilog 記述

それでは、アキュムレータマシンを Verilog で記述してみよう。

今までは、ALU 命令主体であったため、opcode はその下位 3 ビットを ALU のコマンドとして使って、あとは ST 命令を識別するだけで良かった。しかし、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。これが図中に示したコントローラである。コントローラは、まず opcode の部分を解読（デコード）する。

```
wire [‘OPCODE_W-1:0] opcode;
wire [‘ADDR_W-1:0] operand;
wire op_st, op_bez, op_bnz, op_addi ;

assign idatain = imem[pc];
assign {opcode, operand} = idatain;
assign op_st = opcode == ‘OP_ST;
assign op_bez = opcode == ‘OP_BEZ;
assign op_bnz = opcode == ‘OP_BNZ;
assign op_addi = opcode == ‘OP_ADDI;
```

上記のコードでは、命令メモリから読んできた命令コード (idatain) の上位 4 ビットを opcode、下位 4 ビットを operand として分離する。そして opcode を所定のビットパターンと比較して、それぞれの命令を識別する。

次に分岐命令を実装してみよう。分岐命令は、分岐命令実行時に条件が成立した時のみ、pc に飛び先の operand をセットし、そうでなければ 1 加算した結果をセットする。これは図中ではマルチプレクサで表されている。これを直接 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。

```
wire pcsel, pcset;
wire [‘INST_W-1:0] nextpc;
assign pcsel = (op_bez & acc == 0 )| (op_bnz & acc != 0);
assign nextpc = pcsel ? operand : pc + 1;

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

```
begin
   if(!rst_n) pc <= 0;
   else if (pcsel) pc <= nextpc;
   else pc <= pc + 1;
end
```

この方法だと、マルチプレクサで次の pc を選んで、これを書き込む様子が明示的に表現されている。しかし、これは中間的な信号である nextpc や pcsel が必要で面倒であるともいえる。そこで、直接 if 文中に条件を記述しても良い。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if (op_bez & (accum == 0) | op_bnz & (accum != 0))
        pc <= operand;
   else pc <= pc + 1;
end
```

この方法は、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。今回はこの方が見通しが良いだろう。さて、分岐命令の実行時には、ST 命令と同様に、accum は ALU の演算結果をセットしない。図で考えると accset を 0 にする必要がある。しかし、これも、上の記述と同様に、わざわざ accset を定義しないで、

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

と書けば良い。これで、ST でも BEZ でも BNZ でもない命令では、alu で計算した値を accum にセットしてくれる。

このように、レジスタに対する設定は、if 文が利用可能なので、条件をうまく書くと、マルチプレクサの存在を意識しないで見通しの良い記述ができる。しかし、if 文中の条件が複雑になりすぎると、かえって読み難いため、中間的な信号線を定義した方が良い場合もある。この辺をどう書くかは、個人の趣味の範囲に入り、読みやすさを心がけつつ、好きなように書いて欲しい。

次に、イミーディエイト命令の ADDI を実装しよう。この命令ではまず、オペランド部を符号拡張する必要がある。これは、Verilog では繰り返しと連結を利用して以下のように書く。

```
{{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を以下のように記述する。

```
wire [‘DATA_W-1:0] alu_b;
assign alu_b = op_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。

```
wire [‘SEL_W-1:0] com;
assign com = op_addi ? ‘ALU_ADD : opcode[‘SEL_W-1:0];
alu alu_1(.a(accum), .b(alu_b), .s(com), .y(alu_y));
```

全体をまとめて以下に示す。

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

reg [‘DATA_W-1:0] accum;
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] alu_b;
wire [‘DATA_W-1:0] ddatain;
wire [‘OPCODE_W-1:0] opcode;
wire [‘INST_W-1:0] idatain;
wire [‘ADDR_W-1:0] operand;
wire we;
wire op_st, op_bez, op_bnz, op_addi ;
wire [‘SEL_W-1:0] com;

assign idatain = imem[pc];
assign {opcode, operand} = idatain;
assign op_st = opcode == ‘OP_ST;
assign op_bez = opcode == ‘OP_BEZ;
assign op_bnz = opcode == ‘OP_BNZ;
assign op_addi = opcode == ‘OP_ADDI;

assign ddatain = dmem[operand];
assign we = op_st;
always @(posedge clk)
     if(we) dmem[operand] <= accum;

assign accout = accum;
assign com = op_addi ? ‘ALU_ADD : opcode[‘SEL_W-1:0];
assign alu_b = op_addi ? {{8{operand[7]}},operand} : ddatain;

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

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if (op_bez & (accum == 0) | op_bnz & (accum != 0))
pc <= operand;
   else pc <= pc + 1;
```

```
end

always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) accum <= 0;
    else if(!op_st & !op_bez & !op_bnz ) 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 accum.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。