基礎から学ぶ Verilog HDL & FPGA 設計

第6回
スタックの設計

中野浩嗣，伊藤靖朗

デバイスの記事　ビギナーズ

数式の評価を行うために CPU ではスタックを用いる．今回は，このスタックを設計し，シミュレーションにより数式評価が正しく行えることを確認する．（筆者）

● 式の後置記法

数学や，C 言語などのプログラミング言語で幅広く用いられている式の表記法は，**中置記法**とよばれます．例えば，

1＋2＊4－3

(1＋2)＊(4－3)

のように，各二項演算子(＋，－，＊など)は，二つのオペランド(数字)の間に書かれます．CPU で中置記法の式を直接計算するのは困難なので，以下の**後置記法**を用います．

1 2 4＊ ＋ 3 －

1 2 ＋4 3 － ＊

後置記法では，二つのオペランドの後ろに二項演算子が書かれます．後置記法はそのまま日本語で正しく読むことができるので，**日本語記法**ともよばれます．例えば，上の後置記法を日本語で読むと，

1に，**2**と**4**を**かけた**ものを**たして**，**3**を**ひく**

1と**2**を**たした**ものと，**4**から**3**を**ひいた**ものを**かける**

となります．日本語では目的語の後に動詞がくるため，後置記法は日本語で自然に読むことができるのです．

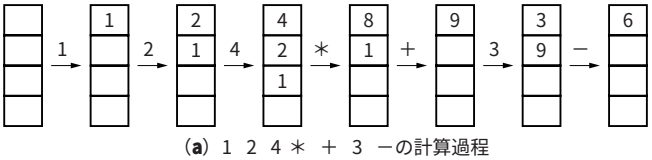
● 演算スタックによる式の評価

後置記法で書かれた式は，**演算スタック**と呼ばれる抽象データ構造を用いると，簡単に計算できます．演算スタックは，基本的にデータの配列です．演算スタックにはデータを追加(プッシュ)していきます．また，演算スタックのトップと 2 番目の間で二項演算を行い，演算結果をスタック・トップに格納します．

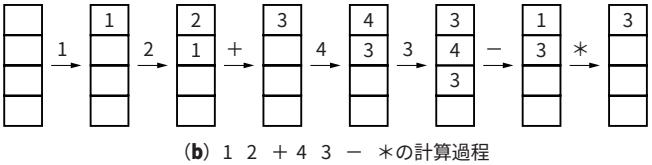
図1は演算スタック上の式の計算過程を表しています．データの追加はスタックのトップに対して行われ，すでにスタックにあるデータは一つずつ下に押し出されます．

二項演算は，スタック・トップと 2 番目の間で行われます．これら二つのデータは破棄され，3 番目以下のデータは一つずつ上に持ち上げられます．そして，演算結果がスタック・トップに格納されます．つまり，二項演算では二つのオペランドが一つの演算結果に置き換わるので，スタック上のデータは一つ減ることになります．

図1にはないが，単項演算の場合は，スタック・トップに対して演算が行われ，演算結果はスタック・トップに格納されるものとします．一つのオペランドが一つの演算結果に置き換わるだけなので，スタック上のデータの数は変わらず，2 番目以下のデータはそのままです．



(**a**) 1 2 4＊ ＋ 3 －の計算過程



(**b**) 1 2 ＋4 3 － ＊の計算過程

図1　演算スタック上の式の計算過程

Keyword　**中置記法，後置記法，日本語記法，演算スタック，演算，スタック・トップ**

● スタックの設計

さて，これから演算スタックを設計します．**演算**の部分は，連載第3回（2007年8月号）で設計した算術論理演算回路（ALU）を用いるものとします．ここでは，データを記憶する**スタック**の部分のみ設計します．あとで，算術論理演算回路とスタックを合わせて，演算スタックを設計します．

スタックは，LIFO（Last In Fast Out）のメモリであり，最後に入力されたデータが最初に取り出されます．ここでは，四つの16ビット・レジスタ（q[0]，q[1]，q[2]，q[3]とする）を用いて，スタックを構成することにします．

q[0]が**スタック・トップ**であり，**スタックの2番目**がq[1]です．スタックは，クロックclkとリセットresetに加えて，書き込みload，プッシュpush，ポップpopの制御用入力ポートを持ちます．

また，入力ポートに16ビットのdと，出力ポートに16ビットのqtopとqnextの二つを持ちます．出力ポートqtopとqnextには，それぞれq[0]とq[1]の保持する値が常時出力されます．これまで設計したモジュールと同様に，リセットresetは非同期リセットに用います．各レジスタの値はクロックclkの立ち上がりで更新されます．**図2**はその動作を表しています．

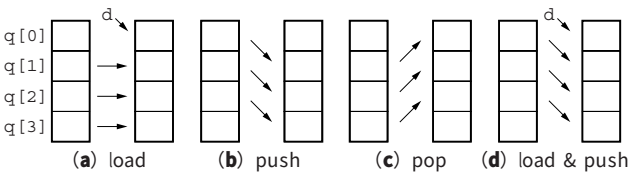


図2　各レジスタにおけるスタックの動作

リスト1　スタックのVerilog HDL記述 stack.v（その1）

```
1     module stack(clk, reset, load, push, pop, d,
                                        qtop, qnext);
2
3       input clk, reset, load, push, pop;
4       input [15:0] d;
5       output [15:0] qtop, qnext;
6       reg [15:0] q [3:0];
7
8       assign qtop = q[0];
9       assign qnext = q[1];
10
11      always @(posedge clk or negedge reset)
12        if(!reset) q[0] <= 0;
13        else if(load) q[0] <= d;
14        else if(pop) q[0] <= q[1];
15
16      always @(posedge clk or negedge reset)
17        if(!reset) q[1] <= 0;
18        else if(push) q[1] <= q[0];
19        else if(pop) q[1] <= q[2];
20
21      always @(posedge clk or negedge reset)
22        if(!reset) q[2] <= 0;
23        else if(push) q[2] <= q[1];
24        else if(pop) q[2] <= q[3];
25
26      always @(posedge clk or negedge reset)
27        if(!reset) q[3] <= 0;
28        else if(push) q[3] <= q[2];
29
30    endmodule
```

入力ポートloadが1のときに，q[0]は，入力ポートdの値をラッチします．入力ポートpushが1のときは，全データが一つ下に移動します．したがって，pushとloadが両方とも1のときは，全データが一つ下に移動し同時に，空いたスタック・トップが，dの値をラッチします．入力ポートpopが1のときは，データが一つ上にポップされます．

● スタックのVerilog HDL記述

リスト1は，以上の動作を実装するスタックのVerilog HDL記述です．6行目のreg文で，16ビットのレジスタの配列を定義しています．前の[15：0]は各レジスタのビットの範囲であり，[3：0]は配列のインデックスの範囲です．この場合，q[0]，q[1]，q[2]，q[3]がそれぞれ16ビットのレジスタになります．

8行目と9行目のassign文で，q[0]とq[1]をそれぞれqtopとqnextに直結しています．四つのalways文を使って，四つのレジスタの動作を定義しています．

11行目のalways文はq[0]，16行目はq[1]，21行目はq[2]，26行目はq[3]の値を定めています．

各always文では，load，push，popの値に依存して，正しく値を決めているのが容易に確認できるでしょう．

● スタックの拡張

評価すべき式が長くなると，スタックのレジスタが4個では足りないことがあります．そこで，parameter文を用いてレジスタ数を指定できるようにします．

リスト2はそのVerilog HDL記述です．2行目のparameter文でレジスタの数Nを指定します．ここでは，8としましょう．Nをparameter文で指定しておくと，このモジュールをインスタンス化するときに，Nの値を指定することにより，レジスタ数を変更することができます．

7行目で，N個の16ビットのレジスタqを用いることを宣言，12～15行目でq[0]への値の代入，24～26行目でq[N－1]への値の代入をalways文で定義しています．また，17～22行目で，q[1]からq[N－2]への値の代入を定義しています．同じ記述が繰り返されるため，整数変数iを17行目で宣言し，19行目のfor文でiを1からN－2まで繰り返します．繰り返されるのは，20～22行目のif文であり，q[i]への値の代入を定義しています．