TUNNING AND IMPLEMENTATION OF DSP ALGORITHMS ON FPGA

M.Licko, Z.Pohl, R.Matousek, A.Hermanek
UTIA Prague, Czech Republic

Introduction

Our department is equipped with a two sort of the hardware platforms based on FPGA. Strong, powerful and expensive ADC-RC1000 [1] and cheaper and less strong XSV-800 [2] with a wide range of peripheral interfaces. One of synthesis tools we are using is Handel-C [3]. We are using it for the C-to-hardware design.

Matlab is a widely used tool for rapid prototyping. Its strength based on the close syntax to C-language is enhanced with the Real Time Workshop [4].

This article describes an algorithm development process for FPGA. The process is shown on the example of an implementation of the QR RLS algorithm. To realize this algorithm means to perform operations such as multiplication, division and square root. When we will use the real data type representation we need to use the floating-point arithmetic unit. But there is a problem with the floating-point unit realisation on FPGA. The research indicates, that a logarithmic arithmetic unit can be used for the real data type processing in some cases such as QR RLS algorithm. We have developed a prototype of the logarithmic unit [5] and here we test it on DSP algorithms using Matlab.

Basic theory about the used algorithm

QRD based RLS algorithm for the signal reconstruction purpose as noise or echo cancellation has been chosen. Error signal U_{23} is the object of interest. Basic theory can be found in [6]. It shows the results on the so-called signal-flow graphs.

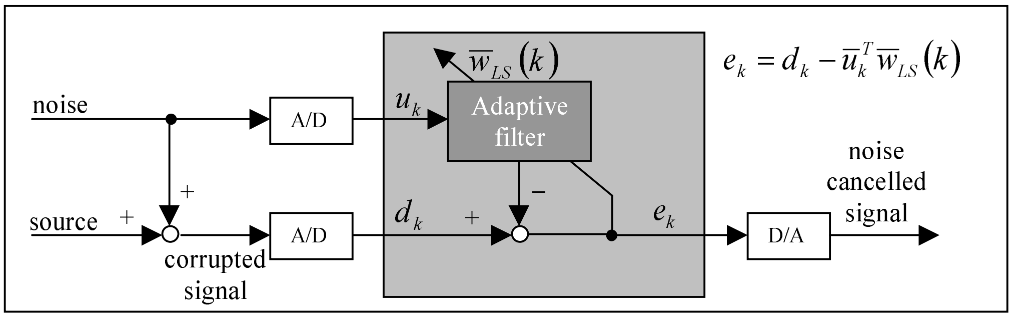

Fig. 1 Residual extraction schema

The adaptive filter on the Fig. 1 minimise the least-squares (LS) criterion (deterministic approach):

$$J_{LS}(w) = \frac{1}{L}\sum_{k=1}^{L}\left|d_k - \overline{w}_{LS}{}^T \overline{u}_k\right|^2 = \frac{1}{L}\sum_{k=1}^{L} e_k^2$$

The e_k is the object of an interest and the LS solution is:

$$\overline{w}_{LS}(k+1) = \overline{w}_{LS}(k) + \left[X_{uu}(k+1)\right]^{-1}\overline{u}_{k+1}\left(d_{k+1} - \overline{u}_{k+1}^T \overline{w}_{LS}(k)\right)$$

$$\left[X_{uu}(k+1)\right]^{-1} = \left[X_{uu}(k)\right]^{-1} - \frac{\left[X_{uu}(k)\right]^{-1}\overline{u}_{k+1}\overline{u}_{k+1}^T\left[X_{uu}(k)\right]^{-1}}{1 + \overline{u}_{k+1}^T\left[X_{uu}(k)\right]^{-1}\overline{u}_{k+1}}$$

It is transformed to the form:

$$\begin{bmatrix} R(k+1) & \overline{z}(k+1) \\ 0 & \varepsilon_{k+1} \end{bmatrix} = Q(k+1)^T \begin{bmatrix} R(k) & \overline{z}(k) \\ \overline{u}_{k+1}^T & d_{k+1} \end{bmatrix}$$

Q and R are the matrices of QRD decomposition and $\overline{z}$ is:

$$\overline{z}(k+1) = R(k+1)\overline{w}_{LS}(k+1)$$

There is no need for the weighting vector $\overline{w}_{LS}$ for the e_k evaluation (such reconstruction process is called a residual extraction):

$$e_k = d_{k+1} - \overline{u}_{k+1}^T \overline{w}_{LS}(k+1) = \varepsilon_{k+1}\prod_{i=1}^{N+1}\cos\varphi_i$$

QRD update is based on sequences of Givens transformations. A basic schema of the Moonen's signal-flow graph is on the next figure Fig. 2.

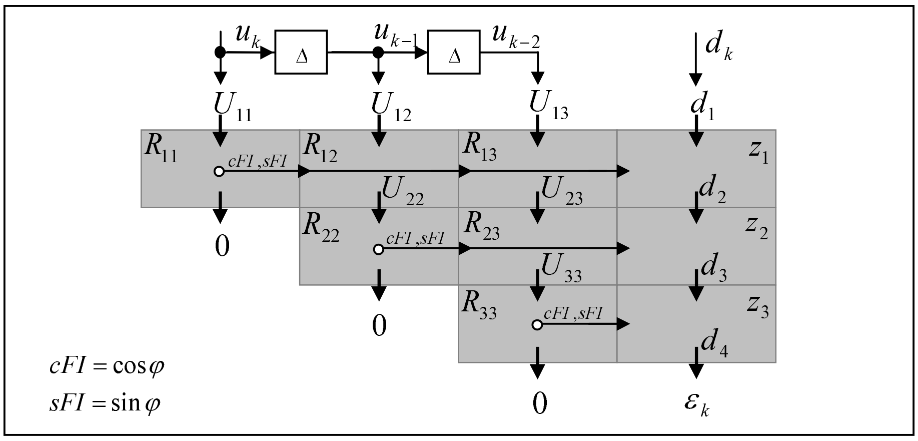

Fig. 2 A core of QRD based RLS adaptive filter

The transformation leads to the form where the vertical outgoing values are zeroed and the horizontal values are propagated throw the line from the leftmost side cell. Cells R_{ij} and z_i update locally its value.

Simple example of QRD based RLS implementation in Matlab

One iteration of the QR RLS than looks like:

```
for I=n:-1:2, U(1,i)=U(1,i-1); end;
U(1,1)=noise(iteration+1);
d=source(iteration+1)+noise(iteration+1);
for i=1:n
  for j=i:n
      if(i==j), den = 1/sqrt(U(i,i)^2 + R(i,i)^2);
                sFI = U(i,i) * den;
                cFI = R(i,i) * den;

                R(i,i) = (R(i,i)^2 + U(i,i)^2) * den;
                Z(i)   =  Z(i)*cFI + d(i)*sFI;
                d(i+1) = -Z(i)*sFI + d(i)*cFI;
                P(i+1) =  P(i)*cFI;
      else
                R(i,j)   =  R(i,j)*cFI + U(i,j)*sFI;
                U(i+1,j) = -R(i,j)*sFI + U(i,j)*cFI;
      end
  end
end
iteration = iteration + 1;
output(iteration+1) = d(n+1)*P(n+1);
```

Concatenation of arrays for a less memory consumption

$$UR = \begin{bmatrix} U_{33} & R_{11} & R_{12} & R_{13} \\ U_{23} & U_{22} & R_{22} & R_{23} \\ U_{13} & U_{12} & U_{11} & R_{33} \end{bmatrix}$$

Next step is the memory spending optimisation. It's possible to make compound matrix UR from the two arrays U and R :

```
for i=1:n-1, UR(n,i)=uR(n,i+1); end;
UR(n,n)=noise(iteration+1);
d(1,1) = source(iteration+1) + noise(iteration+1);
for i=1:n
   for j=i:n
       if(i==j), den = 1/sqrt( UR(n+1-i,n+1-i)^2 + UR(i,1+i)^2 );
                 sFI = UR(n+1-i,n+1-i) * den;
                 cFI = UR(ii,1+i)      * den;
                 UR(i,1+i) = (UR(i,1+i)^2 + UR(n+1-i,n+1-i)^2) * den;
                 Z(i)      =  Z(i)*cFI + d(i)*sFI;
                 d(i+1)    = -Z(i)*sFI + d(i)*cFI;
                 P(i+1)    =  P(i)*cFI;
       else
                 UR(i,j+1)    =  UR(i,j+1)*cFI + UR(n+1-i,n+1-j)*sFI;
                 UR(n-i,n+1-j)= -UR(i,j+1)*sFI + UR(n+1-i,n+1-j)*cFI;
       end
   end
end
iteration = iteration + 1;
output(iteration+1) = d(n+1)*P(n+1);
```

Conversion to process arithmetic operations with our logarithmic unit

As was mentioned in the introduction we developed the logarithmic ALU to process real data numbers on FPGA more effectively. So the next step is to change Matlab's arithmetic operations to our logarithmic ALU's function calls.

The core of the QR RLS algorithm will look like:

```
if(i==j), %den = 1/sqrt( uR(n+1-i,n+1-i)^2 + uR(i,1+i)^2 );
      denL = ld2(1,...
                 lsq2( la2(...
                             lm2(uRL(n+1-i,n+1-i),uRL(n+1-i,n+1-i)),...
                             lm2(uRL(i,1+i),uRL(i,1+i))...
                          )...
                     )...
                );
      sFIL = lm2(uRL(n+1-i,n+1-i), denL);
      cFIL = lm2(uRL(i,1+i)      , denL);

      uRL(i,1+i) = lm2( ...
                         la2(...
                             lm2(uRL(i,1+i),uRL(i,1+i)),...
                             lm2(uRL(n+1-i,n+1-i),uRL(n+1-i,n+1-i))...
                         ),denL...
                     );
      ZL(i)   =   la2(...
                             lm2(ZL(i),cFIL),...
                             lm2(dL(i),sFIL)...
                     );
      dL(i+1) =   la2(...
                             lm2( lm2(d2log(-1), ZL(i)) ,sFIL),...
                             lm2(dL(i),cFIL)...
                     );
      PL(i+1) = lm2(PL(i),cFIL);
else
      uRL(i,j+1)    =  la2(...
                             lm2(uRL(i,j+1),cFIL),...
                             lm2(uRL(n+1-i,n+1-j),sFIL)...
                         );
      uRL(n-i,n+1-j)=  la2(...
                             lm2(lm2(d2log(-1),uRL(i,j+1)),sFIL),...
                             lm2(uRL(n+1-i,n+1-j),cFIL)...
                         );
end
```

QRD based RLS computed with the floating point ALU and the our logarithmic ALU

There are two versions of QR RLS compared on the next figure Fig. 3 where the composed sinus signal is reconstructed. A variance of the noise is changed at the time 4s.

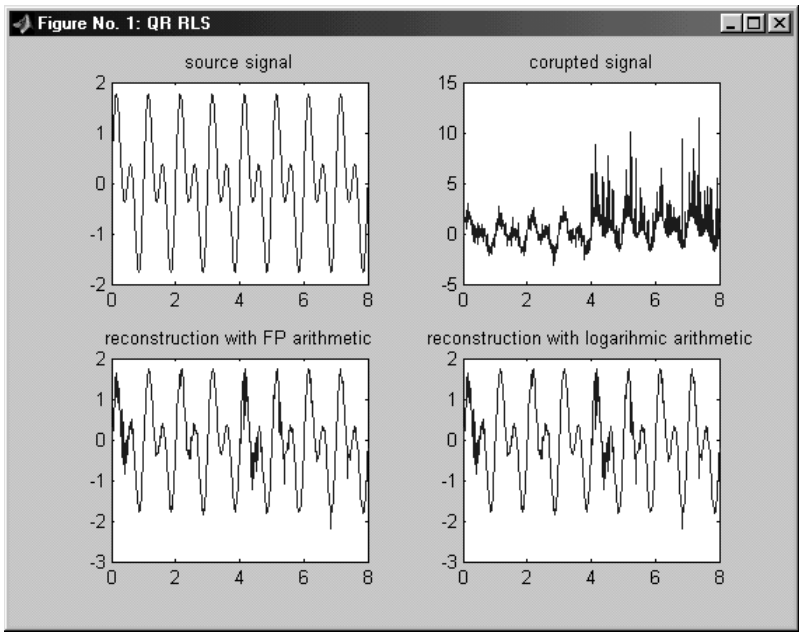

Fig. 3 QR RLS reconstruction with the floating point and the logarithmic arithmetic

Simulink's S-function implementation

The basic M-functions were changed to the S-function form to simulate the results under the Simulink. A reason was the possibility to implement better user interface for the algorithm's tuning purposes and better parameters manipulation. The code is closer to the form of the target platform too.

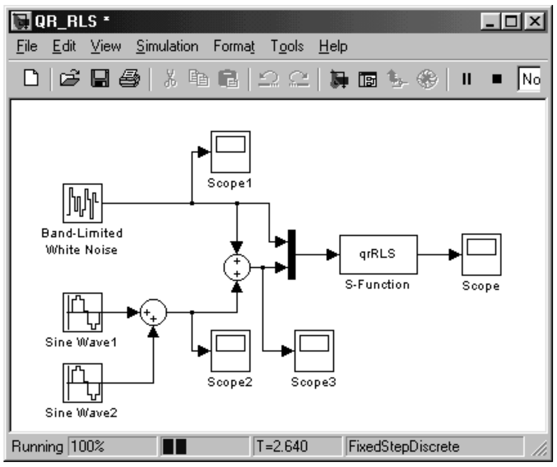

Fig. 4 Simulink's block diagram with the qrRLS S-function

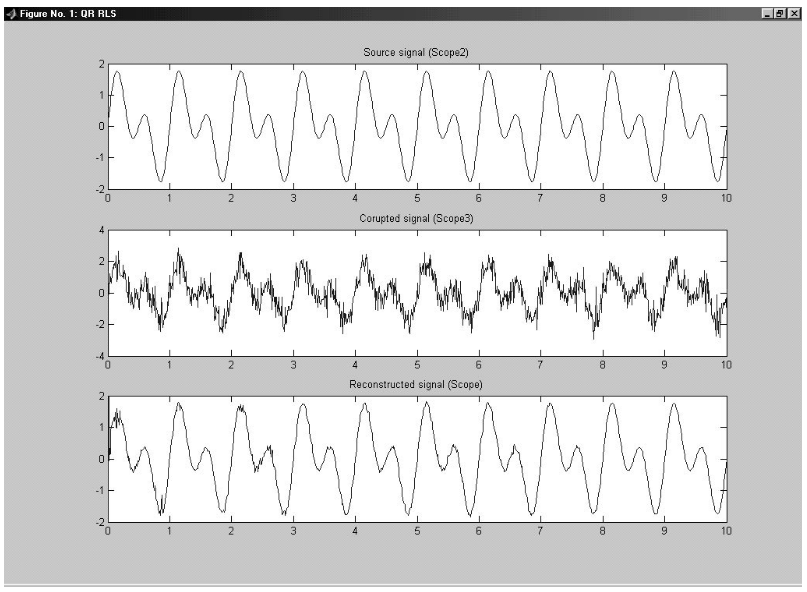

Fig. 5 Results of the reconstruction under Simulink

Notice for FPGA implementation

There is an example of the code in HandelC language of I/O conversion:

```
int 32 a, aL;    // A width of the data type must be declared explicitly
unsigned 20 ai; // There is a 20bit A/D, D/A converter

 XSV800_IN(ai);             // get the value from the I/O interface
 A = (int)( copy(sign(ai),8) @  ai  @ (unsigned 4)0 ); //20bit to 32bit
 int2log( a, aL );          // conversion from integer to logarithm
 // There is the place to put an algorithm for the logarithmic ALU
 log2int( aL,a );           // back conversion from log. to int.
 ai = (unsigned)(a[23:4]); // conversion from 32bit to 20bit
 XSV800_OUT ( ai );         // put the value to the I/O interface
```

There is a '`par`' statement in the HandelC to process parts of the code concurrently. The multiplication, division and square root operations are possible to process in parallel with our logarithmic ALU. The addition must be serialized.

Conclusion

We can recommend Matlab for the rapid prototyping of the DSP algorithms for FPGA hardware platforms. Especially when the C-to-hardware synthesis design suite can be used. Using of such tools shortens our development time. And consequently we are able to concentrate on the appropriate theoretical problem.

The goal of this contribution was to briefly present our application area. With a point of view on the alternative real type data number crunching for FPGA. For this purpose we have developed the logarithmic ALU. All the source codes are implemented as MEX-files too.

References

[1] http://www.alphadata.co.uk

[2] http://www.xess.com

[3] http://www.celoxica.com

[4] http://www.mathworks.com/access/helpdesk/help/toolbox/rtw/rtw.shtml

[5] http://napier.ncl.ac.uk/HSLA

[6] http://www.esat.kuleuven.ac.be/~moonen/publications.html

Acknowledgments

This work was supported by the Ministry of Education of the Czech Republic under Project LN00B096.

Contact

Centre for Applied Cybernetics
Department of Signal Processing, Institute of Theory and Automation
Academy of Sciences of the Czech Republic (UTIA AV CR)
Pod vodarenskou vezi 4, 182 08, Prague 8, Czech Republic
Email: Licko@utia.cas.cz
www: www.utia.cas.cz/ZS
dce.felk.cvut.cz/cak
www.c-a-k.cz