A 64-Point FFT Design Example Using A|RT-Designer

Markus Rupp
Bell-Labs, Lucent Technologies, Wireless Research Lab.
791 Holmdel-Keyport Rd., Holmdel, NJ 07733-0400, USA
Fax: (732) 888 7074, Tel: (732) 888 7104
e-mail:rupp@lucent.com

Abstract

A new EDA tool called A|RT-Designer has been explored allowing to create VHDL code out of a C-code description. On the example of a 64-point FFT with rather hard requirements for an OFDM radio, a solution was found that is close to hand-optimized code available from the vendors. The experience in hardware exploration on a C-level obtained by the tool is reported in this paper.

1 Introduction

After the A|RT-Library and A|RT-Builder, two very useful tools for designing hardware structures using ANSI-C constructs, Frontier Design now offers with their new product A|RT-Designer[1] a tool that allows architectural exploration on a C-level. Again, the description is C-based using the same notation as before for A|RT-Library and A|RT-Builder, A|RT-Designer allows to define single hardware modules for specific tasks and build a complete Finite State Machine (FSM) around the specified C program. A|RT-Designer maps the procedure defined in C into a certain hardware structure that can largely be defined by the hardware designer. Starting with a basic structure: RAM, ROM, micro-controller, ALU, ACU, Multiplier, etc, the designer is free to instantiate as many of these units as required and at the same time define its own new hardware blocks to enlarge the tool's flexibility. New hardware blocks can be defined in C as well allowing also for defining specific timing constraints, like pipelining and multicycle properties.The tool thus allows to explore the architecture in terms of resources and required cycle time at a very early state. By adding PRAGMA comments into the C file or external PRAGMA-files, the hardware architecture and mapping properties are defined.

For a recent wireless LAN product development in the context of IEEE 802.11 HIPERLAN-II[2], it is required to perform a 64 point complex-valued FFT. The internal word length is set to 12bit accuracy throughout the designs. Due to already given signals, the final architecture has to run on either 20MHz, 40MHz or 60MHz. A complete transformation is expected every 2us, allowing to re-use the FFT for transmitter and receiver. Possible settings are given in the table 1. The data are expected to enter and leave the functional block serial, i.e., one complex value (24 bits) for each clock cycle. The final design is supposed to run on a XILINX 800 chip.

Clk in MHz	Cycle time in ns	# of cycles
20	50	40
40	25	80
60	16.6	120

Table 1: *Possible clocks and cycle counts for the design.*

2 Design Considerations

A favorable structure is taking advantage of the low complexity a radix-4 FFT can be build of. Utilizing a 4-point FFT, only 36+45=81 complex multiply operations are required. When implementing them with a very efficient CORDIC[3] structure, the signals become stretched by a fixed value. Such a stretching is not further harming the design but the "multiply by one" operations also need to be implemented now requiring 128 complex multiply operations. It is therefore an open question which designs are eventually better suited for implementation, those with more CORDIC operations or those with less complex multiplier operations.

0-7803-6514-3/00/$10.00©2000 IEEE

First designs for a 16-point FFT using A|RT-Builder show promising results but require a larger design part on the FSM while the actual operations are fairly straightforward in their designs. It is thus be of interest to learn how an automatic design tool like the A|RT-Designer can ease the design of such an FSM and how the tool helps for architectural exploration of such a design. It is furthermore of interest for future developments if the tools allows a simple way of extending a 16FFT to 64FFT to 256FFT, etc.

Applying a radix-4 FFT throughout the design, one recognizes that there are three columns with 16 FFT4 blocks each, and each column is maximally connected by 64 rotations. The 64-point FFT can therefore be split in three, almost identical parts consisting of 16 radix-4 FFTs and 64 rotations each. The rotations of the last row are not required, leaving 128 rotations all together. Only the addressing and the selection of the rotation constants (angles) are different. A possible solution with the A|RT-Designer is thus to solve the FFT for one column and concatenate three of such programs. The overhead of the controller circuitry might be relatively large though.

The computation of an FFT column requires not only that 64 complex values are present, but also that the resulting 64 complex values need to be stored. This is conceptionally different compared to the previous problem. The solution here is that a separate input- and output RAM of size 64 is assumed.

The algorithm has access to all RAM locations and only needs to figure the scheduling of the various parts. The rotations will be implemented as CORDIC algorithms. CORDICs are quite efficient in hardware realization but relatively slow in computation. Thus, a pipelined structure may be necessary. One recognizes in Figures 1 and 2 every FFT4 block requiring four CORDIC operations. It is therefore of interest if only two or even only one instantiation of the CORDIC hardware also can result in low cycle counts.

3 The First Column

Since there are three architectural identical schemes required for the full 64-point FFT, only the first column is condidered for the moment. Following are two basic routines that will be reused several times: a CORDIC routine and a radix-4 FFT. A complex data structure (labeled compl) containing two 12-bit values for real and imaginary part is applied throughout the design.

```
void cordic(const compl in, const Int<12> co
            compl & out_p)
{
#pragma OUT out_p

Int<12> temp=(Int<12>)(co<<((Uint<1>)1));
t12 t_r=in.re;
t12 t_i=in.im;
t12 tin_r=in.re;
t12 tin_i=in.im;

loopr: for (Uint<4> i=1; i<12; i++)
          {
ifr:      if (temp[11]==1)
           {
add1:      t_r+=tin_i;
add2:      t_i-=tin_r;
           }
          else
           {
add3:      t_r-=tin_i;
add4:      t_i+=tin_r;
           }
sh1:      temp<<=(Uint<1>)1;
          tin_r=(t12)(t_r>>i);
          tin_i=(t12)(t_i>>i);
          }
if (((Uint<1>)co[11])==((Uint<1>)0))
   {
   out_p.re=t_r;
   out_p.im=t_i;
   }
   else
   {
   out_p.re=-t_r;
   out_p.im=-t_i;
   }
}

struct comp4 {
compl val_a;
compl val_b;
compl val_c;
compl val_d;
};

void fft4(const comp4 x, comp4 & y)
{
#pragma OUT y

t12 temp0_r,temp1_r,temp2_r,temp3_r;
t12 temp0_i,temp1_i,temp2_i,temp3_i;
```

```
temp0_r=x.val_a.re+x.val_c.re;
temp0_i=x.val_a.im+x.val_c.im;
temp1_r=x.val_a.re-x.val_c.re;
temp1_i=x.val_a.im-x.val_c.im;
temp2_r=x.val_b.re+x.val_d.re;
temp2_i=x.val_b.im+x.val_d.im;
temp3_r=x.val_b.re-x.val_d.re;
temp3_i=x.val_b.im-x.val_d.im;
y.val_a.re=temp0_r+temp2_r;
y.val_a.im=temp0_i+temp2_i;
y.val_c.re=temp0_r-temp2_r;
y.val_c.im=temp0_i-temp2_i;
y.val_b.re=temp1_r+temp3_i;
y.val_b.im=temp1_i-temp3_r;
y.val_d.re=temp1_r-temp3_i;
y.val_d.im=temp1_i+temp3_r;
}
```

Both routines use the fixed-point C-data types provided by A|RT-Library. By defining a new data structure comp4, the program becomes easier to read when the set of four complex valued variables is handled. The following program is required just to support the A|RT-Designer philosophy. Its implication will be explained later.

```
void dummy(const comp4 in1, compl & outa,
 compl & outb, compl & outc, compl & outd)
{
#pragma OUT outa outb outc outd
outa=in1.val_a;
outb=in1.val_b;
outc=in1.val_c;
outd=in1.val_d;
}
```

Finally the main routine to handle the first column of the 64-point FFT.

```
void fft64(const t12 real1[64],
   const t12 imag1[64], t12 real2[64],
                        t12 imag2[64])
{
comp4 in1;
static comp4 out1;
Uint<8> base_add0,base_add1;
Uint<8> base_add2,base_add3;
Uint<8> base_addw0,base_addw1;
Uint<8> base_addw2,base_addw3;
compl out2,outa,outb,outc,outd;

Uint<4> temp0_i=0;
Uint<4> temp1_i=1;
Uint<4> temp2_i=2;
Uint<4> temp3_i=3;
Uint<8> comp_add1=4;
Uint<8> comp_add2=8;
Uint<8> comp_add3=12;

loop1: for (Uint<6> i=0; i<16; i++)
   {
add10:    base_add0=add1[i];
add11:    base_add1=base_add0+comp_add1;
add12:    base_add2=base_add0+comp_add2;
add13:    base_add3=base_add0+comp_add3;

    in1.val_a.re=real1[base_add0];
    in1.val_a.im=imag1[base_add0];
    in1.val_b.re=real1[base_add1];
    in1.val_b.im=imag1[base_add1];
    in1.val_c.re=real1[base_add2];
    in1.val_c.im=imag1[base_add2];
    in1.val_d.re=real1[base_add3];
    in1.val_d.im=imag1[base_add3];
    fft4(in1,out1);
    dummy(out1,outa,outb,outc,outd);

addw10:  base_addw0=add1[i];
addw11:  base_addw1=base_addw0+comp_add1;
addw12:  base_addw2=base_addw0+comp_add2;
addw13:  base_addw3=base_addw0+comp_add3;

cord0: cordic(outa,cord_const[add2[temp0_i]]
                  ,out2);
         real2[base_addw0]=out2.re;
         imag2[base_addw0]=out2.im;
cord1: cordic(outb,cord_const[add2[temp1_i]]
                  ,out2);
         real2[base_addw1]=out2.re;
         imag2[base_addw1]=out2.im;
cord2: cordic(outc,cord_const[add2[temp2_i]]
                  ,out2);
         real2[base_addw2]=out2.re;
         imag2[base_addw2]=out2.im;
cord3: cordic(outd,cord_const[add2[temp3_i]]
                  ,out2);
         real2[base_addw3]=out2.re;
         imag2[base_addw3]=out2.im;
add15a:  temp0_i+=4;
add15b:  temp1_i+=4;
add15c:  temp2_i+=4;
add15d:  temp3_i+=4;
  }
}
```

While looking at the main routine fft64, one recognizes that several parts may look awkward for a C-program. They will be pointed out in the following.

- Many labels are used, a style that is very uncommon for programming. The labes are to identify operations and instantiate specific hardware for them. For example, the cordic operation will be handled by a separate CORDIC block. Since several of these blocks can be used to operate the four CORDICs in parallel one can also decide to run all four operations sequentially with only one hardware realization. Such a mapping is declared in a separate *architecture mapping* file.

- The address pointers for the two sets of four RAMs used to read and write the results have been instantiated twice. They are actually the same pointers and a good C-programmer would save some complexity by reusing them. Here, the situation is different. A|RT-Designer tries to fold the loop as much as possible. If the variables for the addresses exist from beginning to the end of the loop, loop-folding cannot be performed efficiently since now the new value cannot be computed before the last write operation has been performed. It is thus better to redo the computation of the four addresses. Note that the labels already indicate our intention to instantiate separate ACUs just for the index calculations. In the synthesis it turns out that ACUs that are used simply for adding are mapped very efficiently, i.e., with low Logic Cell (LC) count. Thus saving ACUs here is not a good strategy.

- The procedure dummy is still to explain. After the call of fft4, the outcome out1 is a set of four complex values. These values are written into the four CORDICs simultaneously. This method only works if there are four CORDICs instantiated. Otherwise, the A|RT-Designer will try to overwrite the values and it ends in a bus conflict. The separation in the routine helps here, however with introducing one further delay cycle but little or no additional LCs later on the synthesizing part.

The routines fft4 and cordic were declared at the local library, allowing to explore pipelining. The pipelining of the CORDIC routine then has been altered from 0 to 5 stages (0 stages mean that it is a one cycle operation, thus zero storage is required). The cycles, achieved with the *As-Soon-As-Possible* (ASAP) and *As-Last-As-Possible* (ALAP) scheduling selection, were filled in the following table 2. The last parameter shows how many cycles in the loop were obtained.

The whole architecture required only 7 ACUs and no multiplier. The following list shows again how the chosen clock frequencies and cycle times relate to the number of cycles that are possible. A CORDIC with two pipeline stages can be realized in 16ns (see Table3). Thus, the 95-cycle solution, utilizing CORDICs with two-stage pipelines, achieved with the ASAP scheduler is a possibility. However, four pipeline stages are also a solution; thus even less constraints can be put on the CORDIC realization.

# of stages	ASAP	ALAP	# in loop
0	78	77	4
1	92	91	5
2	95	104	5/6
3	124	106	7/6
4	110	108	6
5	124	123	7

Table 2: *Achieved cycles for 16-point FFT.*

Clk	Cycle	# of cyc.	# p. stage	# stages
80MHz	12.5	160ns	3	3
60MHz	16.6	120ns	4	2
40MHz	25	80ns	6	1
20MHz	50	40ns	6	1

Table 3: *Piplining and timing for CORDICS.*

A|RT-Designer allows also exploring how many CORDIC instantiations are required. By instantiating two or even only one, it is possible to measure the required cycles again and find solutions with less complexity. In this case, for a pipeline of two stages, a solution with 104 cycles was found for two and for one CORDIC instantiation.

4 Complete Design

The previous section described only the first row of four 16-point FFTs. In order to achieve the full 64-point FFT, one can tripple the effort and concatenate the blocks, or squeeze all operations in one big loop construct. Since the first solution would lead to a large overhead in the micro- controller, the second solution was followed. Table 4 displays the results. Table 5 lists the LC count for the various blocks using Altera Flex10K and Virtex 800. The micro-controller is remarkably small. The RAM size can be moved out into Embedded Array Blocks (EAB).

Table 6 finally compares the achieved timing and

#stages	ASAP	ALAP	ALAP gr.	in loop	Remark
2	123	123	108	7/7/6	ALAP greedy possible
3	112	138	123	6/8/7	ASAP possible
4	128	154	137	7/9/8	

Table 4: *Final cycle counts for full 64-point FFT.*

Block	Altera LC	Xilinx Slices
FFT4	184	96
CORDIC 1 stage pip.	502	173
Micro_ROM	471	
RAM-Size	16,524	

Table 5: *Comparison of Altera and Xilinx implementation.*

LC count and compares them to comercially available solutions. The results achieved with A|RT-Designer are very comparable with hand-optimized code available from Altera and Xilinx. Compared to the very fast C62 from TI, an FPGA implementation is much more efficient.

	#LC	best possible time
A|RT-Designer	1,388	1.8μ
Xilinx	1,161	1.9μ
A|RT-Designer	2,200	3μ
Altera	3,960	1.2μ
A|RT-Builder	6,200	2μ
TI-C62x,300MHz		2μ

Table 6: *Comparing results.*

5 Conclusion

After a few weeks of getting to understand the little hooks in harware design and the underlying philosophy, A|RT-Designer becomes a tool to get used to. After such a learning period, it becomes a tool with great potential. In particular the exploration with various pipelining stages and repetitive instantiations of hardware blocks is extremely useful.

The particularities in hardware designs for wireless baseband processing typically require block operations. This includes that a block of data is available at a certain time to process, and somehow a block of output data can be written somewhere after the processing stage. This kind of processing is not supported and a support would be of greatest help. Compared to the TI optimizing C-compiler, the tool should also offer a deeper exploration of hardware blocks, i.e., it should not be necessary to dedicate operations to particular hardware blocks. Instead, the tool should be able to find the optimal setting itself and possibly give suggestions what a good architecture setup for a certain number of execution cycles is.

The particular example under consideration of building a 64-point FFT, revealed many of the tool's shortcomings, the already mentioned lack of block-processing being one of them. Although many complex rotations can be saved, the tool did not show any reasonable results when mapping such an inhomogeneous problem. Only if a homogeneous problem-formulation is available, loop folding becomes an efficient method to decrease cycles.

A completely parameterized version of the C-program that could handle FFTs of different sizes does not seem practical. However, to modify the existing code for a 256-point FFT, for example, is a very straight-forward example. Only the hardware mapping and dedicate pragmas need to be added before compilation.

References

[1] A|RT-Designer USER reference guide, Nov. 1999.

[2] R. Van Nee, R Prasad, "OFDM for Wireless Multimedia Communications," Artech House Publishers, 2000.

[3] Behrooz and Parhami , "Computer Arithmetic," Oxford University Press, 2000.