RTL GENERATION USING A HIGH PERFORMANCE DIF ALGORITHM USING 'C'

[1]Mr.Upendar Sapati, [2]Mrs. B.Indira Priyadarshini, [3]Dr. Pallavi Khare
[1]Associate professor, [2,3]Assistant professor
[1,2,3]Department of Electronics and Communication Engineering,
[1]Vjit, Aziz nagar, Hyd., [2,3]Matrusri Engg College, Hyd.

Abstract: Powerful high-level languages to RTL generators are now emerging. One of the use of these tools is to allow software engineers to implement algorithms quickly in a familiar language and target the design to a programmable device. The generators available today support syntaxes with varying degrees of fidelity to the original language. This paper focuses on the efficient use of C to RTL generators that have a high degree of fidelity to the original C language. However, coding algorithms without regard for the capabilities of the target programmable logic can lead to low-performance realizations of the algorithm that are several times slower than what could be achieved with a DSP. This paper presents the architecture of a high-performance radix-2 FFT written in ANSI C that is similar in composition to the classic C implementation that is familiar to most engineers. First, methods to organize memory elements and arrays for maximum data accesses per clock cycle are discussed. Next, the exploitation of the natural parallelism of a radix-2 decimation in frequency algorithm is discussed. Finally, the performance improvement by hiding the first and last of the $\log_2(n)$ butterfly stages is discussed. The resulting RTL outperforms hand optimized DSP assembly code by a factor of three while using less effective area than a DSP solution.

Index Terms: **C Language, Code Converters, Discrete Fourier Transforms, Field programmable gate arrays, Hardware Design Languages, Logic Design**

I. INTRODUCTION

The ability to describe digital logic in a high-level language (HLL) such as C is attractive for several reasons. First, the development time and source lines of code (SLOC) for the logic can be reduced if the level of abstraction is raised. This also results in lower maintenance costs and improves the reusability of the code. Secondly, the available developer pool for intellectual property (IP) blocks increases to include a large pool of embedded software engineers. Finally, if the language used has a high degree of fidelity to the original language, the code describing the IP may be used in both hardware and software targets with only small source code modifications required. Therefore, we attempted to develop an IP block with the following goals:

- Implement the code as it would be implemented in a software environment by a software engineer
- Show a dramatic improvement in development time
- For the chosen IP block (an FFT), beat the performance of a DSP while maintaining the power consumption.

The IP chosen was the complex FFT. The FFT is a standard DSP benchmark and has classic Radix-2 implementation in C [1].

II. C to RTL GENERATOR

A number of C to RTL tools are currently available. Some use extensions to the C language to allow control of the generated RTL down to the wire and register level. While use of a language with this level of control can result in performance that approaches that of an HDL [2], it also requires a larger time investment to learn the language and requires a larger number SLOC to implement the algorithm.

This IP development was done as part of a larger effort in a hardware/software runtime partitioning project [7]. Therefore, to allow reuse of our source code, we chose an RTL generator that has a high degree of fidelity to the C language. Impulse C [2] is largely a subset of the ANSI-C language with support for pointers and arrays. While function calls are not supported, methods for constructing complex applications that will be familiar to software engineers are available: processes and signals. Some control over the pipelining of the generated logic is available via a C #pragma directive. Since the compiler will try to implement large blocks of combinatorial logic, this #pragma can be used to tell the compiler the maximum allowable gate delay permitted in the generated logic. With this tool, the designer can trade latency in clock cycles for clock frequency.

ImpulseC comes with a development environment called CoDeveloper. This tool will compile the C code for execution on a desktop machine in order to debug and verify the algorithm. CoDeveloper allows simulation of the software and hardware components of a design and the development of test benches in C. This environment is very familiar to a software engineer, does not require learning an HDL simulation tool, and results in fast development and verification of new IP. The tool also has direct support for some common bus protocols that would enhance the portability of the IP blocks in embedded systems. For example, the tool can automatically place a CoreConnect PLB or OPB bus wrapper around the generated IP and also generate the required metadata required for incorporation into the Xilinx EDK environment [5].

III. DEVELOPMENT ENVIRONMENT:

A. IP Creation

A standard fixed-point 1024 point complex FFT was written in ImpulseC and debugged in the product's CoDeveloper environment. Once debugged, the RTL generator was run on the C code and VHDL output was collected.

A simple bus wrapper was placed around the generated VHDL to place the IP on to the local bus supported by the FPGA card. The IP was then synthesized using Synplicity's Synplify and implemented using the Xilinx tool chain. The bus-wrapper also included a counter to time the last-data-in to first-data-out latency of the algorithm in clock cycles.

I. Hardware

To allow fast evaluation of the developed IP and a rich programming and visualization environment for analyzing the performance of the algorithms, an Alpha-Data card with a VirtexII Pro FPGA on a PCI carrier card was chosen. In these experiments, the on-chip Power PC was not used. The card comes with supporting software library to handle the configuration of the FPGA. A standard PC running Windows XP was used for the IP test bench.

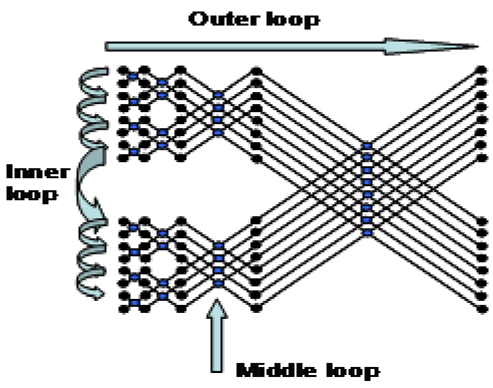
II. Software

The FPGA test bench was written on a PC using C in Visual Studio .NET. The core of the test bench was taken from the test bench used within the CoDeveloper environment. Additional code was added to perform the Alpha Data card initialization and FPGA configuration bit stream load.

B. Basic Radix-2FFT in c

The classic software implementation of an FFT is a radix-2 version. The calculations consist of a series of butterfly calculations performed by three nested loops. In a decimation in time implementation, the inputs are supplied to the input butterfly stage in "bit-reversed order" and the outputs are in "natural order." Conversely, in decimation in frequency implementation, the inputs are in natural order and the outputs are in bit-reversed order. The two implementations are very similar and require the same computational power.

In this paper, we adopt the graphical notation for a butterfly presented in [4].



shows a 16-point decimation in time algorithm with the three software loops highlighted. In the figure, the wing tips of a butterfly (black dots) represent data storage while the bodies of a butterfly (blue dots) represent the complex multiplies and adds that define an FFT butterfly.

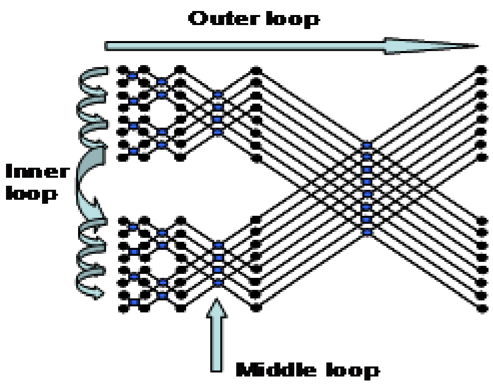


Figure 1. Classic decimation in time FFT

A radix-2 FFT requires log2(N) stages where N is the number of sample points. In Figure 1, the stages are indicated by columns of the butterfly bodies and for our 16-point illustration, there are four stages. In this paper, we'll say that each stage is made up of a number of branches and the number of branches in this case decreases with each stage. For example, the first stage has 8 branches and the second stage has four stages. The middle loop in our software implementation walks the butterflies in each of these branches. Finally, the innermost loop in our algorithm walks the branches themselves.

Note that this implementation stores complex data in the "data" array as sequential real/imaginary pairs. It should also be noted that this implementation is a floating point version; we will not have floating point support in our C to RTL tool or in the FPGA so we will implement a fixed-point version.

The outer loop, in addition to the logic to walk the stages also contains the initialization of FFT "twiddle" factors for each stage. Likewise, the middle loop contains logic to update the twiddle factors for each butterfly. Variations on this classic software approach may include table look-ups of twiddle factors to speed up the code.

C. C to RTL Implementations of the FFT:

To examine the performance of a flow mapping a software implementation of an algorithm to a hardware target, we begin with the classic FFT in C algorithm. Other options are available, e.g. Radix-4 and pipelined approaches, but these are left for future study. The general I/O flow of our implementation is to write the data input to the FPGA, wait until the first data out is available, and then read all the resulting data.

The DSP benchmark used for a comparison of our results is the data given for the TI TMS320C55x processor [6]. The data is given as the number of clock cycles from last-data-in to first-data-available with no overflow scaling and is reported to be 23,848 clock cycles.

- **Implementation A: Direct Mapping of the FFT**

The first implementation of the FFT algorithm is virtually a direct port of the classic algorithm to Impulse. The following is the code for the innermost loop.

```
j = i+mmax;
CMPLX_RD( i, cmplxI );
CMPLX_RD( j, cmplxJ );
tempr = (wr*cmplxJ[REAL] - wi*cmplxJ[IMAG]) >> FAC_SHIFT;
tempi = (wr*cmplxJ[IMAG] + wi*cmplxJ[REAL]) >> FAC_SHIFT;
cmplxJ[REAL] = cmplxI[REAL] - tempr;
cmplxJ[IMAG] = cmplxI[IMAG] - tempi;
cmplxI[REAL] += tempr;
cmplxI[IMAG] += tempi;
CMPLX_WR( i, cmplxI );
CMPLX_WR( j, cmplxJ );
```

Comparing the code to the original code in **Error! Reference source not found.**, we note the following differences:

- The direct mapped version reads the butterfly inputs into the temporary variables cmplxI and cmplxJ. These variables are two element arrays used to represent complex numbers.
- The direct mapped version implements a scale factor by right shifting the result of multiply operations

This implementation of the FFT required about 1 week to develop and test in the CoDeveloper environment and required about 100 SLOC. We note also that the size of the FFT can be changed by changing a single #define in the source module. Table 1 shows that it requires 48,162 clocks to complete; about twice the number required by our benchmark DSP.

- **Implementation B: Optimized complex number working variables**

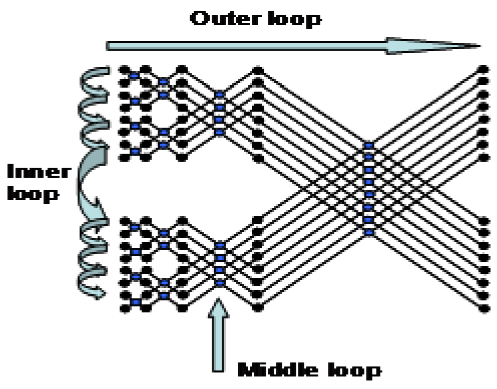
A simple first optimization to the algorithm is obtained when we realize that an FPGA and implements (within practical limits) as many memories as we need and bottlenecks in our code due to memory accesses can be reduced. Thus, in this implementation, we change the definition of cmplxI and cmplxJ from two element arrays to four individual integer variables: cmplxIReal, cmplxJImag, cmplxJReal, cmplxJImag. And, we redefine the CMPLX_RD and CMPLX_WR macros. This scheme allows simultaneous reading and writing of the working complex variables and, as Table 1 shows, reduces the inner-loop computation time from 9 to 6 clock cycles. These 3 clock cycles are saved for each of 5120 butterfly calculations, reducing the overall calculation time to 32802 clock cycles, about 1.4 times that of our benchmark DSP.

- **Implementation C: Input buffer optimization**

The optimization implemented in the previous version can also be applied to the input data buffer; it too can be composed of two distinct memories in order to improve the parallelism of our memory accesses. Thus, our input data buffer is replaced with imaginary and real arrays and the input data is directed to the appropriate array during the data load period.

This implementation again drops three clock cycles from our inner loop computation time and results in a total calculation time of 17,442 clock cycles, or, about 0.7 times that of a DSP. With this optimization, we are now beating the DSP performance in terms of clock cycles to complete the job.

- **Implementation D: Further Parallelization**



Examination of, shows that the two main branches of the FFT could execute in parallel until the last stage where the results would be combined. Likewise, this could be achieved with a decimation in frequency approach where after the first stage; two engines could run in parallel to produce the output data.

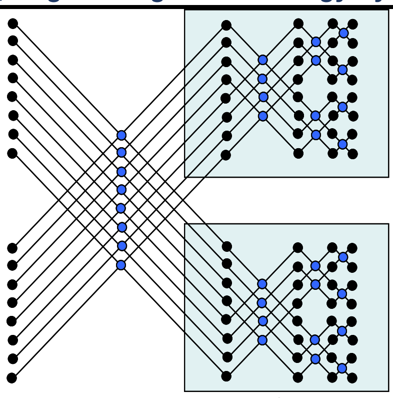


Figure 2. DIF FFT with two butterfly engines

Since ImpulseC will attempt to generate the fewest possible pipeline stages for our logic and create as much parallelism as possible, we only need to create two sets of variables and calculations within our main butterfly engine to effectively get two parallel machines. A more elegant solution here would have been to use the ImpulseC concept of processes, but the “cut and paste” code approach allows us to keep a structure for the code that resembles the original software implementation.

Further, since the data in the DIF arrives in order, we could be processing the first stage of butterflies as the second half of the input data arrives. The I/O for the output phase can also be overlapped with the butterfly calculations by indicating that data is ready to the host processor after completion of the penultimate butterfly stage. If we do this, we can complete each of the last butterfly stages as the host processor reads back the data. Effectively, we are “hiding” the first and last butterfly stages. Again, similar logic applies to a decimation in time implementation. But, here we used DIF because the multiplies on the output stage turn out to be “trivial” (by one) and make it more likely that we will get the data out as fast as the host can consume it. We took the tact here that there is more likely to be delay in the arrival of new data points than there is in the ability of the host to consume the output data. Figure shows the DIF algorithms with the input and output stages highlighted.

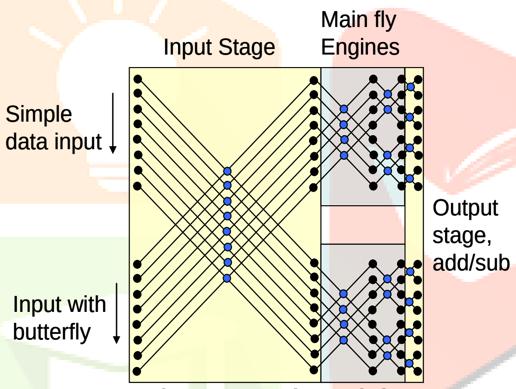


Figure 3. DIF implementation with stage hiding.

Since there is no way in ImpulseC to directly specify the reuse of multipliers within the FPGA and our input stage and second main butterfly engine result in four additional multipliers each, we looked for a way to reduce the total number of multipliers. This was achieved by moving the twiddle factor updates into ROM instead of calculating them in the middle loop using multiplies. The number of block rams on a device tends to equal the number of multipliers available and thus we traded block RAMs for multipliers in an attempt to keep their usage equal. Implementation D uses 12 multipliers and 8 block RAMs.

The total execution time for implementation D is 7186 clock cycles, about 0.3 times that of our DSP benchmark. Let’s examine where the savings come from (vs. Implementation C). First, we have implemented two main engines in parallel and each stage thus only has to run half as many butterflies, so the savings is:

$$8 \text{ stages}*(512/2 \text{ flies per stage})*(3 \text{ clocks per fly}) = 6144 \text{ clocks} \qquad (1)$$

Next, we note that the overhead of running the outer and middle loops is also saved by running the processes in parallel. The overhead for the outer loop has been further reduced by a single clock cycle by moving the twiddle factors to ROM, but the overall savings of the outer loop is negligible. The overhead of running the inner-loop is reported by the ImpulseC compiler output to be 2 clock cycles per execution of the loop, and the net savings is:

$$2 \text{ clocks}*(\Sigma 2^n, n=1,2,\ldots,8) \text{ loop iterations} = 1024 \text{ clocks} \qquad (2)$$

Finally, the savings due to the hiding of the first and last stages is:

$$2 \text{ stages}*512 \text{ flies}*3 \text{ clocks} = 3072 \text{ clocks} \qquad (3)$$

- **On Speed, Size, and Power**

This design is capable of running at 76MHz on a VirtexII Pro FPGA. This rate includes the penalty of moving data on and off the FPGA and no explorations were done into methods do improve the clock rate. The DSP for considered for our benchmark can operate at about twice our max frequency.

Power and energy estimates were performed for the FPGA and DSP options using the power estimation tools available from each vendor. The estimates were done separately for the data input/output phases and the butterfly calculation phases. The estimated power consumption for the DSP with 150MHz clock was about 165mW in both cases. Using an appropriate Virtex4 FPGA, the estimates came to about 400mW for the I/O phase and 800mW for the butterfly phase. The respective energy consumptions for the DSP and FPGA were 32 and 42 μJ.

IV. RESULTS AND CONCLUSIONS

Our DSP benchmark takes 23848 clock cycles to complete the FFT. While we can beat this by a factor of three, our maximum clock rate was only half that of the DSP and ways of improving the maximum clock rate of our design should be explored.

Table 1: implementation results

	A	B	C	D
Inner loop Clocks	9	6	3	3
Total clocks	48162	32802	17442	7186
Slices	536	398	425	859
Multipliers	8	8	8	12
Block RAMs	2	2	2	8

Our energy and size estimates show that the implementation offers an area improvement over a DSP solution but can not match the energy efficiency of the DSP solution.

This FFT began with a classic Radix2 software routine that took only a week to implement and we made simple variations on that design to explore the performance possibilities of implementations that maintain much of the structure of the original design. Parallelism was implemented here by “doubling-up” the software in our butterfly loop and overlapping the first and last butterfly stages with the I/O operations. Additional approaches to creating more parallelism that should be investigated include higher radix implementations, e.g. Radix4, and implementations that use Impulse C processes.

REFERENCES

[1] W. H. Press et al., “Numerical Recipes in C,” 2nd ed., Cambridge: Cambridge University Press, 1992, p. 507.
[2] Impulse Accelerated Technologies,
[3] S. Sukhsawas and K. Benkrid, “A High-level Implementation of a High Performance Pipeline FFT on Virtex-E FPGAs,” in Proc 2004 IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design.
[4] L. R. Rabiner and B. Gold, “Theory and Application of Digital Signal Processing,” Englewood Cliffs: Prentice-Hall, 1975, pp. 360-362.
[5] Xilinx Embedded Development Kit (EDK), http://www.xilinx.com.
[6] Texas Instruments TMS320C55x DSP core, http://www.ti.com.
[7] J. Ardini, “Demand and Penalty-Based Resource Allocation for Reconfigurable Systems with Runtime Partitioning,” to be presented at the *MAPLD 2005 International Conference*, Sept. 7-9 2005, Washington, D.C.