FIR Filter Fits in an FPGA using a Bit Serial Approach

Raymond J. Andraka, Senior Engineer
Raytheon Company, Missile Systems Division, Tewksbury MA 01876

INTRODUCTION

Early digital processors almost exclusively used bit serial architectures because of the high cost of hardware. Bit serial machines have been supplanted by parallel architectures mostly due to the low cost of hardware today. As a result, bit serial solutions are often overlooked in applications where they may be the better choice. This is especially true when designing with Field Programmable Gate Arrays. Many of the elements (Eg. multipliers) used in parallel structures will not even fit in an FPGA. In those cases where an element fits, the routing resources often are insufficient or the resulting design is to slow to be an attractive replacement for dedicated function parts.

By returning to a bit-serial architecture, it is frequently possible to pack a relatively complex function into a single FPGA. A throughput improvement may even be realized over an equivalent parallel structure implemented in FPGAs. To illustrate the advantages of bit serial designs for FPGAs , I examine the implementation of an entire FIR filter in a single FPGA using only bit serial elements.

What is Bit Serial?

Bit parallel designs process all of the bits of an input simultaneously at a significant hardware cost. In contrast, a bit serial structure processes the input one bit at a time, generally using the results of the operations on the first bits to influence the processing of subsequent bits. The advantage enjoyed by the bit serial design is that all of the bits pass through the same logic, resulting in a huge reduction in the required hardware. Typically, the bit serial approach requires 1/n^{th} of the hardware required for the equivalent n-bit parallel design. The price of this logic reduction is that the serial hardware takes n clock cycles to execute, while the equivalent parallel structure executes in one. The time-hardware product, however, for the serial structure is often smaller than for equivalent parallel designs because the logic delays between registers are generally significantly smaller. This means that the serial machine can operate at a higher clock frequency. In the case of FPGAs, signal routing contributes significant propagation delays and often uses up logic cells. The serial structures tend to have very localized routing, often to only one destination. In contrast, the parallel machines usually need signals extended across the width of the processing element. The limited and slow routing resources in FPGAs make the serial processing elements even more attractive. In some cases, the overall throughput for a serial design implemented in an FPGA can actually exceed that of an equivalent parallel design in the same device.

FPGA selection and background

The techniques described in this paper apply to any FPGA, as well as to VLSI designs (which may benefit from the same advantages-especially where high data rates are not necessary). For the purpose of this project, the CLI 6000 series FPGAs made by Concurrent Logic Inc (CLI) were selected. These FPGAs are RAM based, and contain a 56 x 56 array of logic cells interconnected via busses and direct wires to nearest neighbors. Each cell basically consists of a half adder with a D flip flop on the sum output and some extra logic to allow other functions to be programmed. The cell is programmed to one of 64 configurations by a dedicated ram under the cell. The cells each have 2 (3 for certain special functions) inputs and 2 outputs, all of which are accessible from any side of the cell. Each input and output can be directly wired to/from any of the cell’s four nearest neighbors, or to any of 4 local busses which extend to other cells in the row or column. The local busses are broken into segments eight cells long and are connected across the breaks by programmable repeaters. The relatively simple cell in the CLI array is fairly well suited to the bit serial structures.

THE BASIC BUILDING BLOCKS

The most basic functions required for nearly any signal processor include addition, negation and delays. These blocks can then be used to construct the more complicated structures such as multipliers. In most cases, using a bit-serial architecture simplifies the hardware required since all of the bits pass through a single bit wide element. I will discuss the construction of these basic elements in the next paragraphs.

Bit Serial Adder

A bit-serial adder is constructed using a full adder with registers on both its carry and sum outputs. The registered carry output is wired back to the carry input of that full adder. In operation, the two words to be added together are simultaneously shifted least significant bit first into the remaining two inputs. The carry out from the addition of each bit is stored and then used in the summation of the next bit. In effect, the carries are held stationary while the inputs are rippled past. This adder is also known as a Carry Save Adder because of the nature of its operation. The output of the circuit is registered to allow bit pipelining. The resulting latency is one bit time. The serial adder must be cleared before each word to avoid errors. The input words and output word are always equal length. As with parallel addition, the radix points of the input words must be aligned.

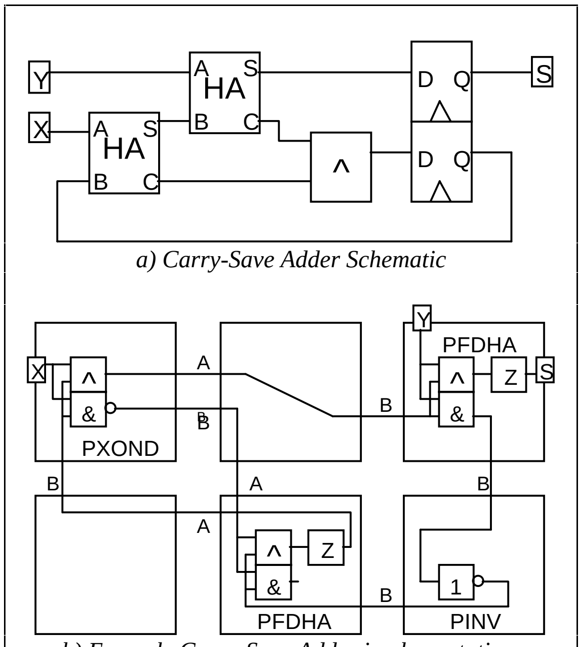

Y
X
A
S
HA
B
C
A
S
HA
B
C
^
D Q
D Q
S
a) Carry-Save Adder Schematic
X
^
&
PXOND
A
B
B
Y
PFDHA
^
Z
S
&
B
A
B
A
^
Z
&
B
1
PFDHA
PINV
b) Example Carry-Save Adder implementation.

Figure 1. Carry-Save Adder schematic and layout. Cells are: PFDHA= half adder with a register on sum output, PXOND = non-registered half adder , PINV = inverter (necessary to correct inverted half adder output).

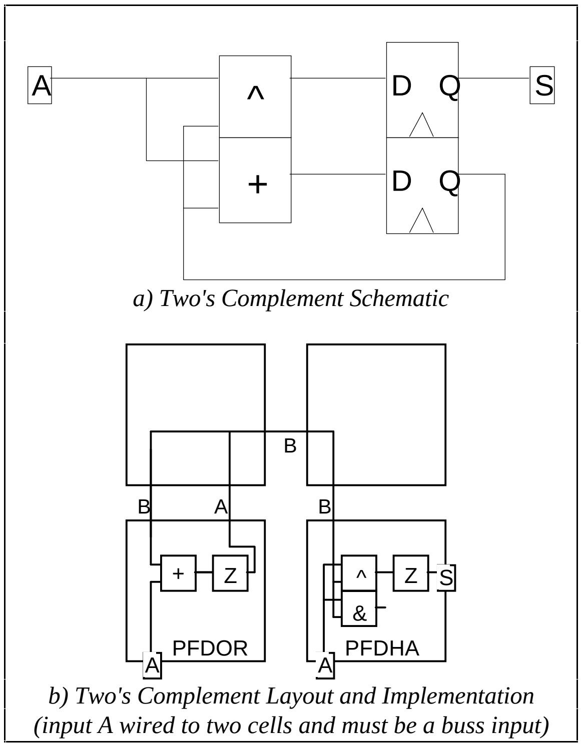

A
^
+
D Q
D Q
S
a) Two's Complement Schematic
B
B
A
B
+
Z
PFDOR
A
^
Z
S
&
PFDHA
A
b) Two's Complement Layout and Implementation
(input A wired to two cells and must be a buss input)

Figure 2. Two's Complement schematic and layout. FPGA cells are: PFDHA= half adder with a register on sum output, PFDOR = OR with register on output.

Bit Serial Two’s Complement

The second elementary function required is a part which will compute the two's complement of the input argument. Recalling the serial algorithm for two's complement (starting at least significant bit copy each bit until first one is encountered then invert remaining bits) yields a simple solution. A serial two's complement circuit is shown in figure 2. The input should be presented least significant bit first. The carry (detect)flip-flop must be reset before each input word, since it causes the XOR to invert the input continuously after the first logic one is detected. The output is registered, so the function has a one bit latency.

Delays

The remaining elementary function is the bit delay (a bit time is one clock cycle). The delay is useful for aligning words as well as for producing word delays required by some algorithms. The delay is simply a D flip-flop inserted into the data path for each bit of delay desired. Word delays are constructed from a string of bit delays equal in length to the number of bits in the word. A sample layout of a word delay for a filter is shown in figure 3.

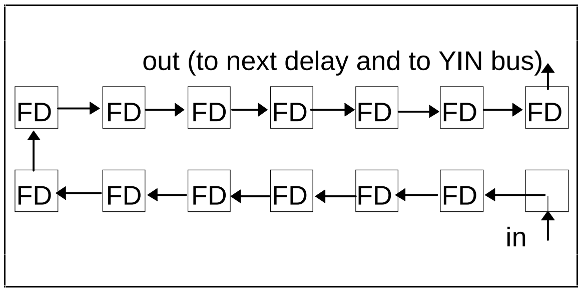


figure 3. Example Word delay

HIGHER FUNCTIONS

The repertoire of functions required to implement a FIR filter is rounded out with a multiplier and a column adder, both of which are constructed from the elements already discussed. Other DSP functions may require additional functions.

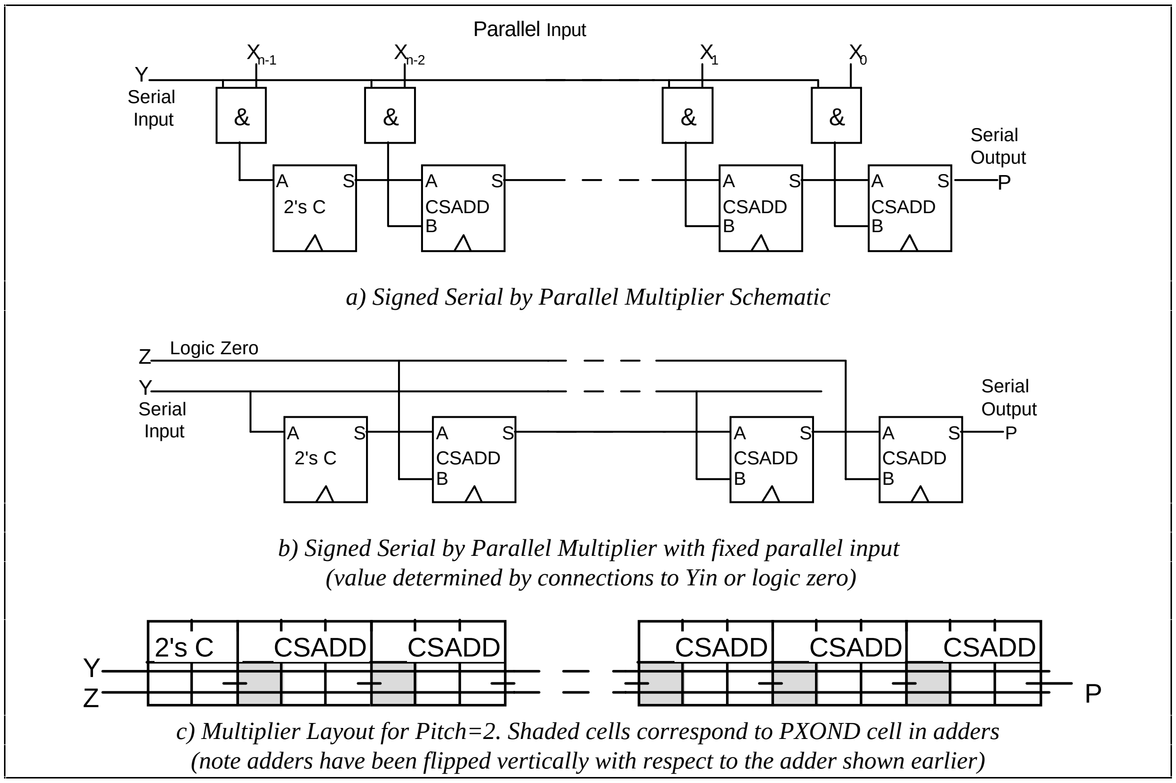
Multiplier

Multipliers are essential to most signal processing algorithms. The simple serial by parallel multiplier is particularly well suited for FPGA implementation because all of its routing is to nearest neighbors with the exception of the input. The number of cells is proportional to the number of bits in the parallel input. One input of this multiplier is parallel while the other is bit serial with the least significant bit presented first. The output is bit serial, also with the least significant bit first. The architecture of a general serial by parallel multiplier is shown in figure 4a. This multiplier performs the familiar shift-add algorithm: the parallel input is multiplied in turn by each bit of the serial input as it is presented, and each of those partial products is added to the shifted accumulation of the previous products. The bitwise products are simply the logical ANDs of the input bit with each of the parallel input bits. The shifting accumulator is easily constructed by chaining a series of carry-save adders together so that inputs to the accumulator are bit parallel and the sum is downshifted on each operation. The serial output is then taken from the output of the least significant bit adder. The output bit has the same weight as the previous serial input bit, yielding a latency of one bit. The number of bits in the output is equal to the sum of the number of bits in each of the inputs. Since the serial input has to be of the same length as the output, it is extended with sign bits.

Multiplication of negative (two's complement) numbers using an unmodified shift-add algorithm will yield an error in the upper half of the product. This error is the result of the inputs not being sign extended to account for the growth of the product (number of bits in the product is equal to the sum of the bits in the multiplicands). The serial input of the parallel by serial multiplier does not suffer from this error since it must be extended to account for the growth in the product. The parallel (X) input will suffer if not corrected. Fortunately, the correction can be made without adding bits to the multiplier hardware by recognizing that the sign extension of X, if taken alone, multiplies the serial input by either zero or negative one. This result is shifted into the lower bits of the multiplier during the course of the multiplication. The sign extension of the parallel input can be accomplished by replacing the most significant adder of the multiplier by a two's complement stage. The input of this stage is the bit product (AND) of the serial input and the sign bit of the parallel input.

The analysis of the correction for negative inputs reveals that the X and Y inputs do not have to have the same number of bits. The X input is essentially sign extended infinitely by the two's complement block, and the Y input is dependent only upon the length of the serial input. The hardware for the multiplier is independent of the precision of the serial input. It is therefore possible to save some hardware in cases where the parallel input does not need the precision of the serial input. This fact - could be advantageous in the FIR filter, since limited precision in the coefficients limits the placement of the filter's zeros but does not otherwise contribute to noise in the output.

The serial multiplier must be cleared before a new word is input to prevent errors. This is especially true if the X (parallel) input is negative since the two's complement circuit cannot self clear. No other controls are required. Secondly, the serial input has to be sign extended by the number of bits in the parallel input (ie to make the number of bits in the serial input equal to the number of bits in the output). The output is always a full precision output.



a) Signed Serial by Parallel Multiplier Schematic

b) Signed Serial by Parallel Multiplier with fixed parallel input (value determined by connections to Yin or logic zero)

c) Multiplier Layout for Pitch=2. Shaded cells correspond to PXOND cell in adders (note adders have been flipped vertically with respect to the adder shown earlier)

Figure 4 Serial by Parallel Multiplier Architectures and Layouts

The partial products presented to the shifting accumulator are generated by the logical AND of the input serial bit with each bit of the parallel input. If the parallel input is fixed, the AND gates can be eliminated by connecting the inputs to each adder directly to the Y (serial) input or to logic zero depending upon the value of the corresponding bit in the fixed parallel input. The FPGA is RAM based, so the value of the parallel input can still be changed by reprogramming if this is done. Since the coefficients of an FIR filter are normally changed infrequently with respect to the data rate, I chose to take advantage of this simplification. This eliminates the AND gates used to generate the product, and more significantly, the logic required for setting and holding the parallel input values. As a note, a further simplification of the fixed input multiplier is possible by noting that the adders associated with zeros in the parallel input reduce to a single delay flip-flop. I chose not to implement that reduction to minimize the changes required in reprogramming coefficients to the FIR filter. The simplified multiplier is shown in figure 4b.

The n bit multiplier is constructed in the FPGA by stringing n-1 of the carry-save adders (CSADD) and a two's complement together. One input of each CSADD is supplied by the previous stage output. The other inputs are supplied by one of two local busses (the Y input to the multiplier for ones, or logic zero for zeros): the parallel input to the multiplier is programmed by the local bus connections only. The layout used is shown in figure 4c. Alternatively, a multiplier with a 3 cell pitch may be created using a different layout for the CSADD adders to allow more parallel input bits to fit in the width of the FPGA.

Column Adder

An adder structure capable of simultaneously adding more than two inputs is a desirable function. This is easily accomplished by a tree of serial adders. Each serial adder (carry save adder) combines two input streams into one output, hence each level of the tree structure reduces the number of serial streams by half, adding one bit time of latency in the process. A column adder constructed in this manner allows an arbitrarily large number of inputs to be summed together without a sacrifice in the bit rate. If an odd number of inputs exist in a level, the odd input can be passed on to the next level via a register to keep the alignment of the bits. If overflow is to be avoided, one bit of growth must be allowed for each level in the adder. Since the input and output must have a similar number of bits, the input must include extra sign (guard) bits to prevent overflow. The number of levels and hence the latency and number of guard bits for an n input column adder is equal to $Log_2(n)$ rounded up to an integer. As with single serial adders, the inputs and output are presented least significant bit first. The column adder architecture is illustrated in figure 5. An FPGA implementation designed to match to a stack of 2 cell tall multipliers is shown in figure 6. Note that the CSADD layouts were optimized for the application.

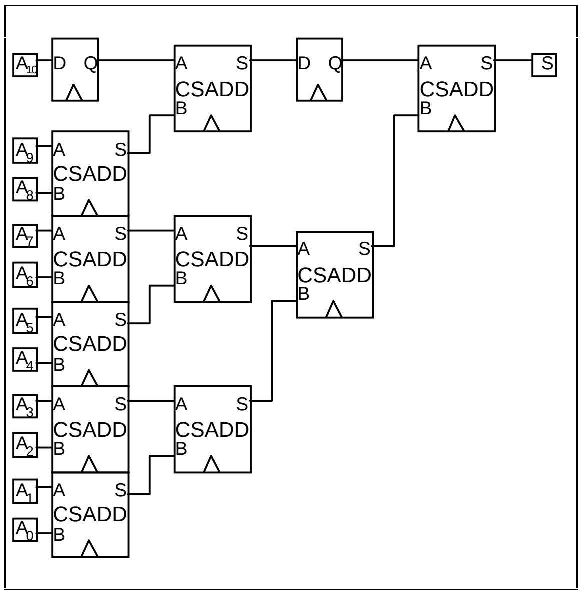


Figure 5. Example Serial Column Adder Architecture

PUTTING IT ALL TOGETHER: AN FIR FILTER

The FIR filter is essentially a discrete convolution of the input signal with a set of coefficients. Mathematically, the filter can be defined as:

$$Y_{[k]} = \overset{n}{S}\ C_i\, X_{[k-i]}$$

The signal flow diagram shown in figure 7 illustrates the algorithm and suggests an architecture using the elements created above. The word delays are inserted for the 'Z' delay blocks. The multiplications shown in the flow graph correspond to the serial by parallel multipliers with their parallel inputs programmed with the value of the associated coefficient. A separate word delay and multiplier are used for each tap in the filter. All of the summation blocks shown are combined and replaced by a column adder with as many inputs as there are taps. Figure 8 shows an example layout and interconnect for a seven tap filter using the functions developed earlier.

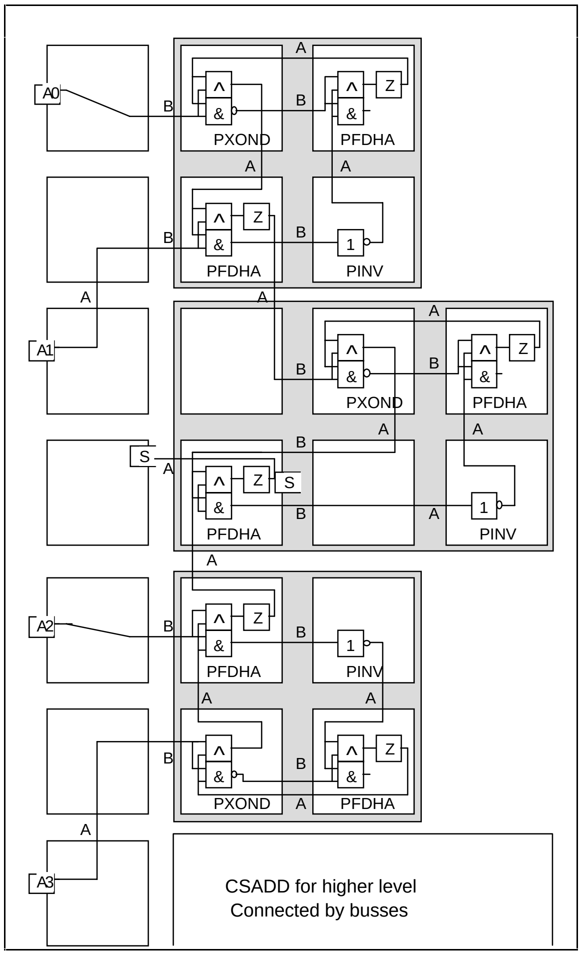


Figure 6. Portion of Serial Column Adder layout designed to match the outputs of a stack of 2 cell high multipliers . Each shaded cell group is a carry save adder. This pattern is repeated to create the first two levels of the column adder. The remaining levels are inserted in the 2 x 3 voids left in the resulting layout .

As each word is shifted into the filter (least significant bit first of course) it is fed to the first multiplier where it is multiplied by C_0. At the same time, The input is fed to the delay chain where it is delayed exactly one word time interval so it arrives at the second multiplier on the same clock that the second word is presented to the first multiplier. The succeeding words eventually fill the delay so that the each of the last n (n=number of taps) words received are simultaneously multiplied by the appropriate coefficients. The outputs of the multipliers are fed to the column adder to perform the summation. The latency for the entire filter is Log_2(# taps) +1 rounded up to the next integer. This reflects the latency of one for the multiplier added to the column adder latency. The delays do not contribute to the latency figure, as they are used to provide the past inputs to multipliers.

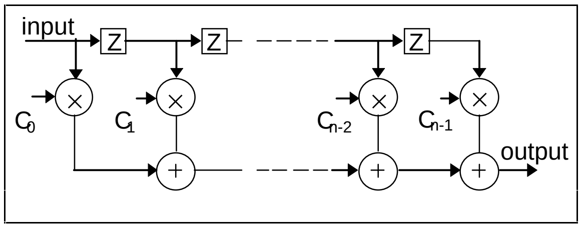


Figure 7. FIR filter signal flow diagram.

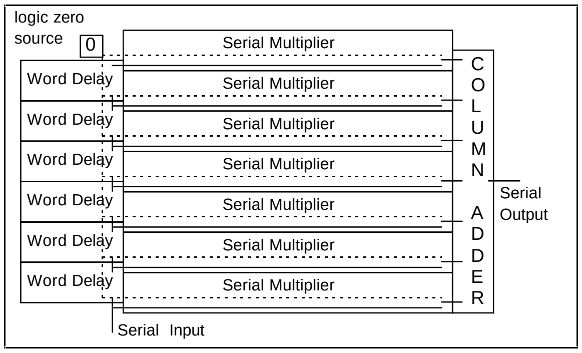


Figure 8. Serial FIR Filter as implemented in FPGA (example has 7 taps). The Coefficients are fixed inputs to the serial by parallel multipliers.

Input, Output and Control

Both the input and output of the filter are serial data streams with each word presented least significant bit first. The input words are of the same length as the output. The word length is equal to the sum of the word sizes of the input (number of bits excluding sign extension required for processor), the coefficients (length of multipliers) and the number of levels in the column adder. The word size of the input need not be the same as that of the coefficients. The input is sign extended to bring it to the same length as the output. This requirement is due to the nature of the serial processing; the inputs need to be as long as the outputs. The extra few bits due to the column adder allow the column sum to grow without overflow. The multiplier array must be reset before each new word begins to shift in. The delays, however, cannot be reset since they hold the old words. In the FPGA implementation, a local reset was wired to the array columns corresponding to to the multipliers and column adder instead of the global reset. That local reset was brought out as a control line in addition to the global reset which clears the filter.

More taps may be obtained by cascading two filters. This is done by adding an extra word delay to the end of the delay chain to feed the serial input of the second filter. The serial outputs of the two filters are summed using a serial adder to obtain the final output. This expansion scheme can be extended to create any number of taps by chaining chips together via the delay chain and summing their outputs with a column adder. That column adder could easily be included one of the devices.

The filter coefficients are determined when the device is programmed. The value of each bit of the coefficient is determined by connections to a single cell (except for the coefficient sign bits which have two connections), so to change a bit, only one cell needs to be updated in the device program (assuming the device was routed in a predictable manner). The bit values of the filter coefficients are defined by the connections of each stage of the multiplier to the corresponding Y bus (for a 1) or logic zero bus (for a 0). The coefficient bits are ordered on the multipliers so that the most significant bit corresponds to the input end of the multiplier. The coefficients are ordered in the FIR filter so that C_0 is at the input end of the filter. The CLI FPGA has a feature which allows partial reprogramming of the array while the remaining portion remains functional. This feature should allow changing coefficients on the fly. Even if this feature is not usable, FIR coefficients are usually changed as a result of a change of context, where the outputs are meaningless during the change anyway. According to the CLI data sheets, complete device reprogramming typically takes place in 8ms.

DESIGN ASSESSMENT

The real-estate occupied by the filter is determined mainly by the layout of the carry-save adders, the number of bits in the coefficients and the number of taps in the filter. If the multipliers are constructed of adders 3 cells wide and 2 cells tall (adders are chained horizontally), one device can contain a 27 tap filter with 12 bit coefficients and 17 bit inputs. Alternatively, if the multipliers are made of 3 cell tall by 2 cell wide adders, one device will support up to 18 taps with 18 bit coefficients and 25 bit inputs. These are respectable results for a single FPGA. Additional taps are easily had by expanding the filter to multiple devices as discussed above.

Using the maximum timing parameters from the CLI device specification, I found the maximum internal clock to clock delay in the FIR filter to be about 30 ns (attributed to the bus transit time on the Y inputs to the multipliers). This translates to a bit rate of about 33 Mhz. For the example 27 tap filter with 12 bit coefficients and 16 bit input data (33 bit output), this means a data rate approaching 1 million words per second.

Placement and Routing

The regular structure and compactness of the elements yields a remarkably high utilization of around 70%. This figure is arrived at by counting the number of cells used as logic (not wire cells) and dividing by the total number of cells in the rectangular area covered by the filter. As a comparison, a 25% utilization is considered good. The placement and routing of the cells is critical to obtaining the predicted data rate and logic density. Unfortunately, the automatic placement tool is incapable of producing satisfactory results, so hand placement is necessary. The automatic router does a decent job provided it has a good placement to start from. The routed solution, however, is not optimum. Additionally, the auto route does not connect the multiplier Y and Z busses in a predictable manner, so partial reprogramming would not be possible. These limitations could be overcome by writing a generator program which takes advantage of the regular structure of the filter to create a placed and routed data base. The generator would have the added benefit of drastically reducing design time and probability of errors.

CONCLUSIONS

In this paper, I have shown that it is possible to pack a relatively complex digital signal processing function into an FPGA by using bit serial structures. The cost of bit serial architectures in terms of more clock cycles can be offset to some degree by the shorter delay paths between pipeline registers. The resulting design is fast enough for many applications where a bit serial process may not have been considered. The bit serial design philosophy is extendable to other FPGAs, VLSI designs and other places, and is as applicable today as it was for the early processors.

REFERENCES

[1] P. Denyer and D. Renshaw, VLSI Signal Processing: A Bit Serial Approach. Selected Readings, Addison-Wesley, 1985.

[2] C.R. Rupp, Digital Functions and Processors, work in progress, University of Massachusetts at Lowell, 1992.

[3] P.J. Graumann and L.E. Turner, Implementing Digital Signal Processing Algorithms using Pipelined Bit-Serial Arithmetic and Field Programmable Gate Arrays, FPGA '92, February 1992.

[4] Concurrent Logic Inc., "CLI 6000 Series Field Programmable Gate Arrays", Data Sheet, Concurrent Logic Inc., December 1991.

AUTHOR'S BIOGRAPHY

Raymond J. Andraka is a senior engineer for Raytheon Company's Missile Systems Division, where he has been designing digital signal processors for radar systems and managing projects for the last 5 years. Ray graduated from Lehigh University in December 1983 with his BSEE. He then spent 4^{1}/2 years in the Air Force researching techniques for recovering data masked by noise and managing development contracts before joining Raytheon. He recently earned his MSEE from the University of Massachusetts at Lowell. Ray enjoys flying, tinkering and spending time with his wife and two boys.