TTEACHING APPLICATION IMPLEMENTATION ON FPGAS TO COMPUTER SCIENCE AND SOFTWARE ENGINEERING STUDENTS

Yoginder S. Dandass
Computer Science and Engineering, Box 9637
Mississippi State University, MS 39762

Abstract

Modern field programmable gate array (FPGA) devices enable the creation of hybrid hardware-software systems in which performance-critical portions of the application are implemented in hardware. However, the design and implementation of hardware modules requires considerable specialized skill that many Computer Science and Software Engineering students lack. This paper describes a finite impulse response (FIR) filter implementation for an FPGA platform using ImpulseC, a tool for automatically generating VHDL code from C code, in a course designed for students with minimal digital design experience.

Students create an initial software-only implementation of the FIR filter and are subsequently led through a series of incremental design optimizations, each one producing a better performance or consuming fewer resources than previous designs. The final implementation results in an implementation that is nearly 21 times faster than the software implementation. By the end of the course, students are able to complete FPGA implementations of systems that are considerably more complex than the FIR filter.

Introduction

Field programmable gate array (FPGA) devices are becoming an increasingly popular option for implementing embedded systems because time and performance critical portions of the application software can be implemented in optimized hardware. However, implementing complex processing elements in hardware using hardware description languages (HDLs) such as VHDL and Verilog requires specialized knowledge and skill in digital design concepts. Furthermore, because the implementation process is cumbersome, many important design decisions (*e.g.*, hardware-software partition boundaries and selection of hardware components) are made early. High implementation costs also prevent the exploration of alternative design choices.

A number of emerging C language-based tools (such as SystemC[1], Handel-C[2], and ImpulseC[3]) are addressing the difficulties associated with implementing applications in reconfigurable hardware. The central idea behind these languages is to enable application designers to leverage the features of the well-known C and C++ languages for describing the runtime behavior of applications. The specialized compilers generate the required low-level hardware implementations automatically. All of these tools require users to learn the supported subset of the languages or require learning an enhanced syntax and semantics.

The Department of Computer Science and Engineering at Mississippi State University (MSU), recently offered a split-level (*i.e.*, open to graduate and undergraduate students) Special Topics in Computer Science course focusing on application development techniques for reconfigurable and embedded computing. In this course, students are not required to have prior digital design coursework or experience. However, knowledge of C and a course in operating systems are essential prerequisites. We use ImpulseC and associated tools in the course because ImpulseC supports ANSI C syntax. Hints are provided to the C-to-VHDL converter using **`#pragma`** compiler directives. This means that students can use their preferred software development environment in order to develop, test, and debug the application before deploying the hardware portions on an FPGA.

In this course, we implement a finite impulse response (FIR) filter [4] in order to examine the advantages and pitfalls of using ImpulseC. The simplicity of the FIR filter kernel means that students can rapidly develop the source codes for the filter leaving more time for exploring a variety of design and implementation alternatives.

```
01: #define TAPS 64
02: void firfilter() {
03:   int   accum = 0; /*64 bit*/
04:   int   result;
05:   int   tap;
06:   int   coeff[TAPS];
07:   int   buffer[TAPS];
08:   … /* initialize coeff */
09:   … /* initialize buffer */
10:   for (;;) {/* do forever */
11:     read(instream,
             &(buffer[TAPS-1]));
12:     for (tap=0; tap<TAPS; tap++)
13:       accum += coeff[tap]*
                   buffer[tap];
14:     result = accum >> 2;
15:     write(outstream, &result);
16:     for (tap=1; tap<TAPS; tap++)
17:       buffer[tap-1]=buffer[tap];
18:   }
19: }
```

Figure 1: Fundamental FIR Filter Code.

Figure 1 describes the basic FIR filter code for a 64 tap integer FIR filter. The number of taps in the filter is declared in line 01. In line 08 the 64 elements in *coeff* vector are initialized by reading the values from an external source (*e.g.*, the host processor). In line 09 the first 63 elements of *buffer* are read in from a data source (*e.g.*, a sensor). Lines 10–19 represent an infinite loop that processes a continuous stream of data from the external sensor, converting input samples into output samples. In the body of infinite loop, the input sample is acquired from a sensor in line 11 and processed in lines 12–13. The value of variable *accum* is the sum of the products of the corresponding elements of *coeff* and *buffer* computed as follows:

$$accum = \sum_{i=0}^{TAPS-1} coeff_i \times buffer_i \ . \qquad (1)$$

In lines 14 and 15, the result is scaled down and written to an output device (*i.e.*, the output sample is given to the consumer of the transformed sensor data). In lines 16 and 17, the elements of *buffer* are shifted down one position in order to make room for the next input sample value read in from the sensor (*i.e.*, the value that was read into vector *buffer* 63 iterations ago – and is now in *buffer*[0] – is discarded).

One of the pedagogical challenges posed to the students in the course is to optimize the performance of the FIR filter implemented on Digilent, Inc’s XUPV2P FPGA board [5]. This board contains a Virtex2 Pro FPGA and a variety of onboard peripherals and connectors that can be used for FPGA configuration, communication, memory expansion, and debugging. More importantly, the FPGA contains two PowerPC 405 (PPC405) hard IP cores that run at a frequency of 300MHz, 136 blocks of random access memory (for a total of 272 kilobytes of RAM), and 136 18-bit multipliers. The optimized FIR filter is required to run at a clock frequency of 100MHz and must fit completely on this board.

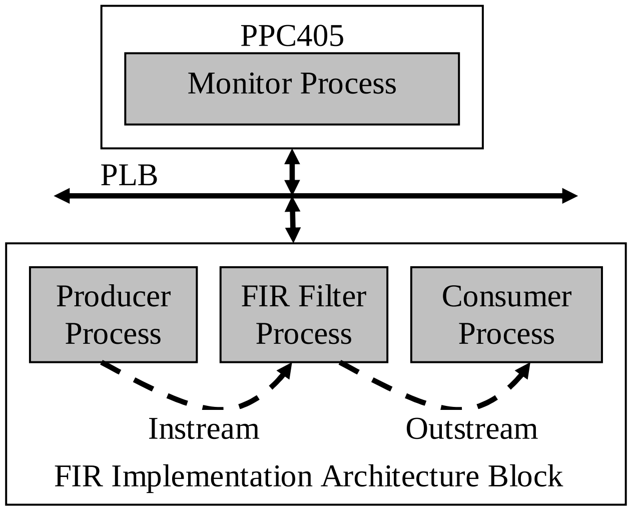

Figure 2: FIR Hardware Implementation.

The primary design of the expected solution is described in Figure 2. A software *monitor* process executes on the PPC405 and is responsible for signaling the hardware processes to start and for providing initial values for the *coeff* vector. The monitor process also counts the number of PPC405 clock cycles it takes for the hardware processes to produce a specified number of output samples.

The hardware portion of the implementation consists of three separate hardware processes. The *producer* process simulates a sensor device and generates raw values at a much faster rate than can be processed by the FIR filter process. The *consumer* process simulates the remainder of the system that uses the transformed output sample of the FIR filter. In our implementation, the consumer process simply discards the output sample and sends a message to the monitor process after the specified number of output samples have been received from the FIR filter process. The software monitor waits in a polling loop for the message from the consumer process, computes the number of clock cycles that have elapsed since the previous message, and sends this timing information to a terminal connected to the development machine via a RS232 link.

The various processes communicate with each other using ImpulseC *streams*. Streams are essentially unidirectional first-in first-out (FIFO) queues and are the preferred means for inter-process communication in the ImpulseC programming model. There are a total of six streams in the implementation. The *instream* stream connects the producer process to the to the FIR filter process. The *outstream* stream connects the FIR filter process to the consumer process. The remaining four streams connect the hardware processes to the monitor software process using the PPC405's processor local bus (PLB). The PLB is a bus architecture that is used for connecting peripherals implemented in FPGA fabric to the PPC405. The implementation of the PLB is provided by Xilinx for their FPGAs. Following is a list of the streams connected to the PLB in the FIR filter implementation:

- Monitor-to-producer: used for starting the producer,
- Monitor-to-FIR: used for providing the initial coefficient values to the FIR filter.
- Monitor-to-consumer: used for specifying the number of FIR filter output samples that must be received before the software monitor is notified, and
- Consumer-to-monitor: used to inform the monitor that the specified number of output samples have been received by the consumer process.

The streams connecting the hardware processes to the software processes are not critical for the performance of the FIR filter and are not shown in Figure 2. For this optimization exercise, the students focus on improving the performance of the code in lines 10 through 17.

Implementation Details

Students are lead through several different implementations of the FIR filter code in order to explore a variety of design scenarios. In ImpulseC, processes are simply functions with the **`void`** return type that take handles to ImpulseC inter-process communication objects as function arguments. For example, the FIR filter process has the following signature:

```
void filter(
co_stream coeffstream,
co_stream instream,
co_stream outstream);
```

where *co_stream* is the datatype for the ImpulsC stream handle. In order to inform ImpulseC that the `filter` function defines a process, we call the process creation function as follows:

```
fir_proc = co_process_create(
          "filter",
          (co_function)filter,
           3,
           coeffstream,
                instream,
                outstream);
```

Note that the steams are associated with the process when the `co_process_create()` function is called.

Also, ImpulseC must be informed of all processes that are to be executed in hardware by calling the `co_process_config()` function. For example the FIR filter process is placed into hardware with the following call:

```
  co_process_config(fir_proc,
                   co_loc,"pe0");
```

where `fir_proc` is the handle to the previously created FIR filter process object and the string "pe0" represents the FPGA hardware. If the `co_process_config()` function call is omitted for the the `fir_proc` handle, the FIR filter process is implemented in software.

The ImpulseC software development environment is collectively known as CoDeveloper and consists of tools such as a graphical integrated development environment (IDE), preprocessors, compilers, optimizers, and HDL generators. CoDeveloper also provides graphical tools such as StageMaster Explorer that enables users to examine, at a high level, the performance of the generated hardware configurations.

CoDeveloper exports the generated HDL components into an existing Embedded Development Kit (EDK) project. EDK is Xilinx, Inc's development environment for creating FPGA-based applications that require processors, processor busses, and software in addition to processing units implemented in the FPGA fabric [6]. Students are instructed to create a basic embedded system consisting of a single PPC405, 64 kilobytes of RAM (implemented using block RAM resource – *i.e.*, BRAMS – resident on the FPGA), a PLB, and an RS232 device. The RS232 port is used to communicate with a terminal program on the development workstation running windows XP. Standard I/O from the software executing on the PPC405 is directed to appear on the terminal window in the development workstation. Students with no prior digital design experience can use the EDK project wizard that handles most of the implementation details.

Once CoDeveloper exports the generated HDL modules to the EDK project, students simply instantiate the generated modules in their EDK projects, connect any ports that require special handling (typically, the default port connections are sufficient) and can use EDK's tools to synthesize the system's HDL and to download the FPGA configuration to the FPGA for performing live testing. Readers are encouraged to read Refs. [3] and [6] for additional details and tutorials.

Software Implementation

The first solution that students implement is a software module that combines the monitor and FIR filter processes. This software-based implementation serves as a benchmark for measuring the optimization achieved by the various hardware processes. The producer process is still implemented in hardware in order to simulate a hardware sensor that produces data and the *instream* stream connects the producer process to the software FIR filter process over the PLB. There is no consumer process (it is assumed that the software process will consume the transformed data itself).

A naïve implementation of the FIR filter code described in Figure 1 produces a new output sample after 16,359 PPC405 clock cycles (or 5,453 system clock cycles – the PPC405 core embedded in the FPGA operates at 300MHz while the FPGA fabric operates at 100MHz). This delay includes a significant penalty for reading the raw data from the *producer* process over the PLB. It also includes the time taken by the PPC405 to perform the required computation and buffer shifting operations.

An important optimization can be made to the filter code. The buffer shift loop in lines 16 and

17 from Figure 1 can be eliminated by converting the linear buffer into a circular buffer. The code for implementing this optimization is shown in Figure 3 (note that ancillary code such as variable declaration and *coeff* and *buffer* initialization has been omitted).

```
tail = 0;
for (;;) {/* do forever */
 read(instream,
     &(buffer[(tail+TAPS-1)%TAPS]));
 for (tap=0; tap<TAPS; tap++)
  accum += coeff[(tap] *
           buffer[(tap+tail)%TAPS];
 result = accum >> 2;
 tail++;
}
```

Figure 3: Circular Buffer FIR Filter.

The variable *tail* in the code in Figure 3 maintains the index that corresponds to the current *buffer*$_0$ element in equation (1). The software implementation of the FIR filter using the circular buffer takes on average 12,633 PPC405 clock cycles (or 4,211 system clock cycles) to produce an output sample, a 22.78% improvement.

The Initial hardware Implementation

If students heed the design guidelines provided in Figure 2 and modularize their codes for the software FIR filter implementation appropriately, the effort required to create the initial hardware implementation is relatively trivial. The monitor process is split into the software monitor and the hardware FIR filter. The simple consumer process is also created and the required stream-based inter-process communication channels are established. The code is simulated in software using CoDeveloper for testing purposes. Once testing and verification is complete, the hardware is generated and exported into EDK.

In the EDK, students compile the software portions and generate the FPGA configuration *bitstream* from the hardware description. Before downloading the bitstream to the FPGA, students are encouraged to examine the summary reports produced by the synthesis tools in order to determine resource utilization and to ensure that the timing constraints are met. At this point, most students are surprised to learn that we only achieve a maximum frequency of 81.23MHz, well below our target of 100MHz. This is because ImpulseC generates a design with several logic levels that cannot be executed within the 10*ns* constraint. Table 1 shows an excerpt from the EDK report detailing the cause of the delay.

Students can readily determine from this timing report that the multiplication operation "MULT18X18:A9->P34" (in row three of Table 1) is the main source of the delay. Additional confirmation is obtained by examining the generated hardware using CoDeveloper's Stage Master tool. Figure 4 shows an excerpt from Stage Master identifying the number of operations that are performed in a single step. The subscripts immediately following arithmetic, assignment, and array indexing operations specify how these operations are grouped by ImpulseC's optimizer into stages. Operations with identical subscripts are executed in the same clock cycle. Clearly, the multiplication and summing of the product into *accum* is occurring in a single stage (*i.e.*, in stage 2). Therefore, we need to instruct ImpulseC's hardware generator to separate the multiplication and summation operations to occur in two, or more, separate stages, thereby shortening the time taken by each stage. This will result in the design with more stages that meets the specified timing constraint.

Table 1: Details for Failed Timing.

Cell (in->out)	Fanout	Gate Delay	Net Delay
FDP:C->Q	32	0.374	0.818
LUT3:I2->O	2	0.313	0.445
MULT18X18:A9->P34	2	4.541	0.561
LUT2:I1->O	1	0.313	0.000
MUXCY:S->O	0	0.377	0.000
XORCY:CI->O	1	0.868	0.533
LUT2:I0->O	0	0.313	0.000
XORCY:LI->O	1	0.535	0.506
LUT2:I1->O	0	0.313	0.000
XORCY:LI->O	1	0.535	0.418
LUT4:I2->O	1	0.313	0.000
FD:D		0.234	

accum $=_2$ accum $+_2$
(buffer[tail $+_1$ tap & $_1$ 63]$_2$)$_2$ $*_2$
(coeff[tap]$_1$)$_2$

Figure 4: Stage Master Excerpt from the Failing Code.

In ImpulseC, programmers can specify the maximum delay that the generated hardware should have in a single stage by using the following **#pragma** directives:

- **#pragma** `CO SET StageDelay` *n*, and
- **#pragma** `CO SET DefaultDelay` *n*,

where *n* is the maximum acceptable delay. The DefaultDelay parameter specifies the maximum delay that ImpulseC should use unless the StageDelay parameter is used to override the default delay for a specific block of C statements. ImpulseC estimates the delay based on the widths of the operands.

Figure 5 illustrates the modifications needed to the FIR filter inner `for` loop in order to incorporate the stage delay specification of 64 for the multiplication and summation operations. Figure 6 shows the resulting StageMaster output showing that the array indexing, multiplication, and summation operations occur in separate steps.

```
for (tap=0; tap<TAPS; tap++) {
  #pragma CO SET StageDelay 64
  accum += coeff[(tap] *
               buffer[(tap+tail)%TAPS];
}
```

Figure 5: FIR Filter Inner Loop with a StageDelay Specification.

This initial hardware implementation produces a new FIR filter result every 972 CPU clock cycles (or 324 system clock cycles), a 92.31% improvement in performance compared to the software implementation. However, further improvements in performance are also possible.

accum $=_4$ accum $+_4$
(buffer[tail $+_1$ tap & $_1$ 63]$_2$)$_3$ $*_3$
(coeff[tap]$_1$)$_3$

Figure 6: Stage Master Excerpt from the Code with a StageDelay Setting of 32.

The pipelined hardware Implementation

The next optimization students are asked to make is to create a pipelined implementation of the FIR filter process. Pipelining results in an implementation that can perform several operations simultaneously. For example, the summation, multiplication, and array lookup operations could be occurring for three different iterations of the **`for`** loop in Figure 5 at the same time.

Figure 7 shows the inner loop with the StageDelay and PIPELINE **#pragma** directives. Recall that the StageDelay specification is required in order to meet timing constraints. Additionally, StageMaster shows that the pipeline has a *latency* of 5 and a *cycles/result* value of 2. The latency value of 5 means that every loop iteration takes 5 clock cycles to execute (*i.e.*, it takes 4 cycles for the operations shown in Figure 6 and one cycle for managing the loop). The cycles/result value of 2 indicates that the implementation completes a loop iteration every other clock cycle.

According to the software monitor process, the hardware implementation produces a new FIR filter output sample every 405 CPU clock cycles (or 135 system clock cycles), essentially doubling the performance of the non-pipelined implementation. Furthermore, the resource utilization of this design is relatively modest. The FIR filter process module uses 814 out of 13,696 slices, 2 BRAMs (one each for *buffer* and *coeff* arrays), and 3 out of 136 18-bit multipliers.

```
for (tap=0; tap<TAPS; tap++) {
 #pragma CO SET StageDelay 64
 #pragma CO PIPELINE
  accum += coeff[(tap] *
           buffer[(tap+tail)%TAPS];
}
```

Figure 7: FIR Filter Inner Loop with a Pipeline Specification.

Automatic loop unrolling

ImpulseC also provides a mechanism for unrolling **`for`** loops that have constant index lower and upper bounds. Programmers only need to place the **`#pragma CO UNROLL`** statement at the top of the **`for`** loop in order to cause that loop to be unrolled. Loop unrolling essentially results in the body of the loop being replicated as many times as specified by the loop index bounds. The loop index variable is replaced with the constant designating the appropriate loop index in each loop body copy. Unrolling has the potential for significantly improving performance because the optimizer will generate HDL that performs the loop body for all iterations simultaneously if possible. Note that simultaneous computation is only possible if the computation performed within an iteration is not dependent on the result of a previous iteration. However, unrolled loops can also result in significantly increased resource utilization because the logic required for implementing the loop body is replicated many times.

In the FIR filter code, summing the product into the *accum* variable is one factor that limits the effectiveness of the loop unrolling. Another limiting factor is that arrays are stored in BRAM blocks that are *dual ported* (*i.e.*, there can only be two simultaneous reads from different addresses). This means that only two iterations of the FIR filter inner loop can execute simultaneously. The primary factor that prevents us from utilizing ImpulseC's loop unrolling capability is that the resulting implementation's FPGA resource utilization is too high. When the inner loop in the FIR filter process is unrolled, the design's resource requirements are as follows:

- number of Slices: 15,933 out of 13,696 (116% of the device),
- number of Slice Flip Flops: 5,138 out of 27,392 (18% of the device)
- number of 4 input LUTs: 29,529 out of 27,392 (107% of the device),
- number of BRAMs: 2 out of 136 (1% of the device), and
- number of MULT18X18s: 136 out of 136 (100% of the device).

Clearly, this design will not fit in the FPGA we are using. There is no option in ImpulseC to perform a partial unroll. ImpulseC also does not have the ability to automatically limit the unroll operation when the BRAM dual access limitation occurs.

Manual Loop Unrolling

Loop unrolling, however, is an important optimization that can be achieved within resource utilization limits by restructuring the original program. The problem of dual port memory access can be solved by "scalarizing array variables," a feature supported by ImpulseC. "Scalarization" means that the ImpulseC compiler can move the implementation of an array variable from BRAM into FPGA registers provided that all accesses to the array elements is through constants (or an expression that can be completely resolved at compile time involving the index of an unrolled `for` loop). When the array elements are implemented as registers, they can be accessed simultaneously. The problem of excessive resource utilization can be addressed by manually unrolling the FIR filter inner loop. This way, the extent of the unrolling can be explicitly controlled.

There are two arrays in the FIR filter process, *coeff* and *buffer*. The loops that are used to initialize the 64 elements of *coeff* and 63 initial elements of *buffer* can be unrolled in order to ensure that array accesses occurs using constant array indices during initialization. In the main body of the FIR filter process, however, the need for using the variable `tail` in accessing *buffer* makes the task of using constant array indices complex. It is much simpler to use the linear buffer FIR filter processes shown in Figure 1. The primary reason for implementing the circular buffer version was to eliminate the cost of shifting *buffer* elements. However, if *buffer* is scalarized, its elements can be read from and written to simultaneously. This means that the shifting of the entire array can be performed in a single clock cycle.

In the manually unrolled implementation, we perform four sets of 16 loop iterations in our manually unrolled loop. In other words, we perform 16 multiplication and summations simultaneously. We also ensure that all access to the *coeff* and *buffer* arrays occur through constant indices. Sixteen accumulator variables *a_00*, *a_01*, …, and *a_15* are used in order to maximize parallelization of the multiplication and summation operations. Because the computation of an individual accumulator variable is not dependent on the value of another accumulator variable, all 16 accumulators can be computed independently from each other in a single iteration. If we were to use a single accumulator, then the summation of the individual products will result in sequential operations because the FPGA is not fast enough to sum all sixteen products.

Figure 8 shows the code for the modified FIR filter nested loops. In line 05, the individual accumulator variables are set to zero in anticipation of the upcoming accumulation operations. The partially unrolled inner loop is in lines 06 through 38. In lines 09 through 32, temporary variables *c_00* through *c_15* and *b_00* through *b_15* are assigned the appropriate values from the *coeff* and *buffer* arrays. A series of `if` statements is used to convert the *tap* variable's value into constant array indexing expressions in order to enable scalarization. A `case` statement may also be used; however, ImpulseC has problems with the implementation of `case` statements in particular situations. Therefore we generally avoid the use of `case` statements. ImpulseC "flattens" the if-else structure so that it executes in a single clock cycle. In lines 34 through 37, the sixteen products are computed from the corresponding c_00 through c_15 and b_00 through b_15 and summed with the corresponding accumulators a_00 through a_15.

The call to the `co_par_break()` function in line 33 informs ImpulseC's optimizer that we want the operations following line 33 to be performed in a separate clock cycle from any operations preceding line 33. This explicit control over the optimizer is required because otherwise the ImpulseC optimizer packs too many operations into one clock cycle resulting in a failure to meet timing constraints when the design is synthesized. The `co_par_break()` function has no other purpose and performs no computation.

In line 39, the 16 accumulator variables are summed into the final accumulator *accum*. Because the FPGA cannot add 16 32-bit variables in a single clock cycle, we set the DefaultDelay pragma directive to 128 which causes ImpulseC to split the summation to complete in four clock cycles.

```
for(;;) {
  /* Read values from the stream */
  co_stream_read(stmDataIn, &nSample, sizeof(nSample));
  firbuffer[TAPS - 1] = nSample;
  a_00 = a_01 = a_02 = a_03 = a_04 = a_05 = a_06 = a_07 =
  a_08 = a_09 = a_10 = a_11 = a_12 = a_13 = a_14 = a_15 = 0;
  for(tap=0; tap<TAPS/16; tap++) {
    #pragma CO SET StageDelay 64
    #pragma CO PIPELINE
    if (tap == 0){
      c_00 = coef[0*16 + 00]; b_00 = buffer[0*16 + 00];
      c_01 = coef[0*16 + 01]; b_02 = buffer[0*16 + 02];
      … /* similarly extract the others here*/
      c_15 = coef[0*16+ 15]; b_15 = buffer[0*16 + 15];
    }
    else if (tap == 1){
      c_00 = coef[1*16 + 00]; b_00 = buffer[1*16 + 00];
      c_01 = coef[1*16 + 01]; b_02 = buffer[1*16 + 02];
      … /*similarly extract the others here*/
      c_15 = coef[1*16 + 15]; b_15 = buffer[1*16 + 15];
    }
    else if (tap == 2){
      c_00 = coef[2*16 + 00]; b_00 = buffer[2*16 + 00];
      c_01 = coef[2*16 + 01]; b_02 = buffer[2*16 + 02];
      … /*similarly extract the others here*/
      c_15 = coef[2*16 + 15]; b_15 = buffer[2*16 + 15];
    }
    else if (tap == 3){
      c_00 = coef[3*16 + 00]; b_00 = buffer[3*16 + 00];
      c_01 = coef[3*16 + 01]; b_02 = buffer[3*16 + 02];
      … /*similarly extract the others here*/
      c_15 = coef[3*16 + 15]; b_15 = buffer[3*16 + 15];
    }
    co_par_break();
    a_00 += b_00 * c_00;
    a_01 += b_01 * c_01;
    … /* similarly compute a_02 to a_14 here */
    a_15 = b_15 * c_15;
  }
  accum = a_00 + a_01 + a_02 + a_03 + a_04 + a_05 + a_06 + a_07 +
          a_08 + a_09 + a_10 + a_11 + a_12 + a_13 + a_14 + a_15;
  nFiltered = accum >> 4;
  co_stream_write(stmDataOut, &nFiltered, sizeof(nFiltered));
  for (tap = 1; tap < TAPS; tap++){
     #pragma CO UNROLL
     firbuffer[tap-1] = firbuffer[tap];
  }
}
```

Figure 8: Code for the Manually Unrolled FIR Filter Computation.

The `for` loop in lines 42 through 55 performs the shifting of the buffer elements. Because this loop is unrolled (note the `#pragma CO UNROLL` directive in line 43), the shift operation is completed in a single clock cycle.

This pipelined and partially unrolled implementation of the FIR Filter consumes the following resources:

- number of Slices: 8,059 out of 13,696 (58% of the device),
- number of Slice Flip Flops: 9,702 out of 27,392 (35% of the device)
- number of 4 input LUTs: 14,821 out of 27,392 (54% of the device), and
- number of MULT18X18s: 64 out of 136 (47% of the device).

These statistics seem to suggest that there may sufficient resources to attempt a design that performs 32 simultaneous multiplication operations. However, when the resource consumption approaches 100%, it takes longer for the hardware "place and route" tools to complete.

The synthesized design takes 63 CPU clock cycles (or 21 system clock cycles) to produce a new output sample when executed on the FPGA. This is an 87.5% improvement over the previous pipelined implementation. A 16-fold improvement in performance is not achieved because there are a number of operations that must be performed sequentially. This is a good opportunity to introduce students to Amdahl's law [7] which states that the speedup of a problem with size W that has a serial component with size W_s, is bounded from above by W/W_s regardless of the number of processors utilized.

Conclusions

This paper describes a series of hands-on projects that can be used to introduce hardware-software codesign to Computer Science, Software Engineering, and beginning Computer Engineering students who have had little (or no) exposure to digital design. By using ImpulseC, students can express the expected behavior of the application in C, a language most students are comfortable with. However, in order to facilitate experimentation, students must make an effort to organize the code in a manner that allows easy movement of functionality between software and hardware platforms. Furthermore, in order to enhance performance, students need to take advantage of the optimization features offered by ImpulseC.

The processes of starting from a software-only implementation and making incremental modifications with increasing performance improvements holds the students' interest while keeping the workload on the students manageable. Many Computer Science and Software Engineering students are wary of the course initially. However, they quickly learn that hardware-software codesign can be performed rapidly using modern programming tools. The ability to quickly make changes to the C code and compiler directives in order to test alternative designs is also appealing to the students.

The most valuable advantage of using ImpulseC is the ease with which students can explore the design space. This encourages students to try a variety of alternatives in order to come up with a faster performing implementation than their peers. In this course students were organized into pairs and several pairs had friendly competitions trying to outdo each other.

This simple FIR filter implementation also provides the foundations on which the students worked towards other advanced term projects in the class (*e.g.*, parallel AES modules, bioinformatics peptide processing pipeline, and extension of ImpulseC streams for use over network links).

While ImpulseC provides a number of advantages when implementing applications for a FPGA platform, it is not perfect. ImpulseC and the CodDeveloper tools suffer from a number of minor bugs and the development team puts out new releases on a nearly monthly basis. The language has limited support for pointers. Furthermore, it is likely that existing C programs will require extensive reorganization in order to be successfully ported for execution in hardware. However, with some effort, students quickly learn to workaround the issues that come with ImpulseC programming and become effective users of the language, the streaming programming model, and the CoDeveloper toolset. The source codes for all of these implementations are available upon request.

References

1. D. C. Black and J. Donovan, *SystemC: From the Ground Up*, Springer, New York, 2005.

2. I. Page, “Constructing Hardware-Software Systems from a Aingle Description,” *Journal of VLSI Signal Processing*, 12(1), pp. 87-107, 1996.

3. D. Pellerin and S. Thibault, *Practical FPGA Programming in C*, Prentice Hall, New Jersey, 2005.

4. R. G. Lyons, *Understanding Digital Signal Processing*, Prentice Hall, new Jersey, 1996.

5. Xilinx, Inc., *Xilinx University Program Virtex-II Pro Development System*, 2005, http://www.digilentinc.com/Data/Products/XUPV2P/XUPV2P_User_Guide.pdf, (accessed may 2007).

6. Xilinx, Inc., EDK Concepts, Tools, and Techniques, http://www.xilinx.com/ise/embedded/edk91i_docs/edk_ctt.pdf, (accessed May 2007).

7. G. M. Amdahl, “Validity of the Single Processor Approach to Achieving large Scale Computing Capabilities,” in *Proceedings of the AFIPS Conference*, 1967.

Biographical Information

Yoginder S. Dandass is an Assistant Professor in the Department of Computer Science and Engineering at Mississippi State University (MSU). His research interests include high performance computing and reconfigurable computing with applications in computer security, digital forensics, and bioinformatics. He obtained his PhD from MSU in 2003 his MS degree from Shippensburg University of Pennsylvania in 1996. From 1997 until 2003, Dr. Dandass was employed as a researcher at MSU’s NSF Engineering Research Center for Computational Fluid Dynamics and in the Department of Computer Science. He also has over eight years of experience as a consultant in the information technology industry prior to 1997.