Available online at www.elixirpublishers.com (Elixir International Journal)

Computer Science and Engineering

Elixir Comp. Sci. & Engg. 85 (2015) 34671-34676

ISSN: 2229-712X

Outerloop pipelining in FFT using single dimension software pipelining

K. Immanuvel Arokia James[1] and M. J. Joyce Kiruba[2]

[1]Department of EEE VEL Tech Multi Tech Dr. RR Dr. SR Engg College, Chennai, India.

[2]Department of ECE, Dr. MGR University, Chennai, India.

ARTICLE INFO

Article history:
Received: 19 May 2012;
Received in revised form:
22 August 2015;
Accepted: 29 August 2015;

Keywords
FFT,
Pipelining, VHDL,
Nested loop,
FPGA,
Integer Linear Programming (ILP).

ABSTRACT

The aim of the paper is to produce faster results when pipelining above the inner most loop. The concept of outerloop pipelining is tested here with Fast Fourier Transform. Reduction in the number of cycles spent flushing and filling the pipeline and the potential for data reuse is another advantage in the outerloop pipelining. In this work we extend and adapt the existing SSP approach to better suit the generation of schedules for hardware, specifically FPGAs. We also introduce a search scheme to find the shortest schedule available within the pipelining framework to maximize the gains in pipelining above the innermost loop. The hardware compilers apply loop pipelining to increase the parallelism achieved, but pipelining is restricted to the only innermost level in nested loop. In this work we extend and adapt an existing outer loop pipelining approach known as Single Dimension Software Pipelining to generate schedules for FPGA hardware coprocessors. Each loop level in nine test loops is pipelined and the schedules are implemented in VHDL. Across the nine test loops we achieve acceleration over the innermost loop solution of up to 7 times, with a mean speedup of 3.2 times. The results suggest that inclusion of outer loop pipelining in future hardware compilers may be worthwhile as it can allow significantly improved results to be achieved at the cost of a small increase in compile time.

© 2015 Elixir All rights reserved.

Introduction

Perhaps the most widely used loop pipelining methods are based around modulo scheduling. In modulo scheduling the operations from a single iteration of the loop body are scheduled into *S* stages, with each stage requiring *T* clock cycles to execute. Each operation in the loop body is assigned to start on a single cycle in a single stage. The *S* stages run sequentially to execute a single iteration of the innermost loop, but may all run in parallel for different loop iterations without breaching the resource constraints of the target platform. A new iteration of the innermost loop is initiated every *T* clock cycles with the result that the executions of *S* loop iterations are overlapped. Standard modulo scheduling based methods are limited to pipelining (overlapping) the iterations of the innermost loop in a loop nest.
Single-dimension Software Pipelining (SSP) extends innermost loop pipelining methods, such as modulo scheduling, allowing them to be applied at any level in a rectangular loop nest. Under this methodology a single loop level is selected for pipelining based upon metrics such as the expected initiation interval for each level and/or the potential for data reuse.

The data dependence graph for the loop nest is then simplified according to the method presented in by assuming that the iterations from loop levels above and below the pipelined level execute sequentially, all dependence distance vectors are reduced to equivalent scalar values .This allows standard modulo scheduling techniques to be applied to the nested loop, regardless of which level is being pipelined. The final schedule is then constructed from the modulo schedule. A new iteration of the pipelined loop level is initiated every *T* clock cycles, but an extra delay must be added after each group of *S* consecutive iterations. The delay added between each group is the same and its value is defined. The extra delays are necessary to ensure that no more than *S* iterations are overlapped into a pipeline with *S* stages as this would cause resource conflicts.

PIPELINING

In computing, a pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements.
Instruction pipelines, such as the classic RISC pipeline, which are used in processors to allow overlapping execution of multiple instructions with the same circuitry. The circuitry is usually divided up into stages, including instruction decoding, arithmetic, and registers fetching stages, wherein each stage processes one instruction at a time.
Graphics pipelines, found in most graphics cards, which consist of multiple arithmetic units, or complete CPUs, that implement the various stages of common rendering operations (perspective projection, window clipping, color and light calculation, rendering, etc.).

Software pipelines, where commands can be written so that the output of one operation is automatically used as the input to the next, are following operation. The UNIX command pipe is a classic example of this concept; although other operating systems do support pipes as well.

Buffered, Synchronous pipelines

Conventional microprocessors are synchronous circuits that use buffered, synchronous pipelines. In these pipelines, "pipeline registers" are inserted in-between pipeline stages, and are clocked synchronously. The time between each clock signal is set to be greater than the longest delay between pipeline stages, so that when the registers are clocked, the data that is

Tele:
E-mail addresses: immanuel_james@yahoo.com

© 2015 Elixir All rights reserved

written to them is the final result of the previous stage. Another type is referred as buffered asynchronous pipelines.

Buffered, Asynchronous pipelines

Asynchronous pipelines are used in asynchronous circuits, and have their pipeline registers clocked asynchronously. Generally speaking, they use a request/acknowledge system, wherein each stage can detect when it's "finished". When a stage is finished and the next stage has sent it a "request" signal, the stage sends an "acknowledge" signal to the next stage, and a "request" signal to the previous stage. When a stage receives an "acknowledge" signal, it clocks its input registers, thus reading in the data from the previous stage. The AMULET microprocessor is an example of a microprocessor that uses buffered, asynchronous pipelines.

Unbuffered pipelines

Unbuffered pipelines, called "wave pipelines", do not have registers in-between pipeline stages. Instead, the delays in the pipeline are "balanced" so that, for each stage, the difference between the first stabilized output data and the last is minimized. Thus, data flows in "waves" through the pipeline, and each wave is kept as short (synchronous) as possible. The maximum rate that data can be fed into a wave pipeline is determined by the maximum difference in delay between the first piece of data coming out of the pipe and the last piece of data, for any given wave. If data is fed in faster than this, it is possible for waves of data to interfere with each other.

Parallelism is achieved by starting to execute one instruction before the previous one is finished.

• The simplest kind overlaps the execution of one instruction with the fetch of the next instruction, as on a RISC. Because two instructions can be processed simultaneously, we say that the pipeline has two *stages*.

RISC refers to Reduced Instruction Set Computer.

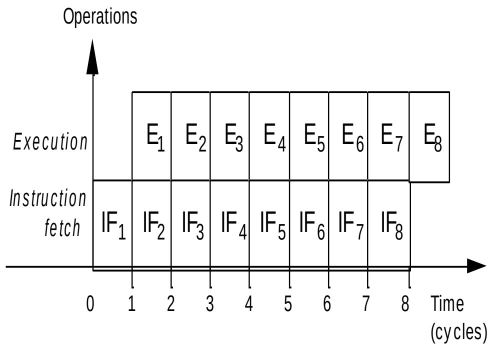


Figure 1. Three Stage pipeline

Load and store reference memory, so they take two cycles. A pipeline may have more than two stages. Suppose, for example, that an instruction consists of four phases:

- Instruction fetch
- Instruction decode
- Operand fetch
- Execute

In a non-pipelined processor, these must be executed sequentially, so that a result is only available each four pipeline cycles (subcycles):

In a pipelined processor, after a delay to load the pipeline, a result is available each pipeline cycle.

The type of pipelining described above achieves instruction-level parallelism—execution of multiple instructions in parallel.

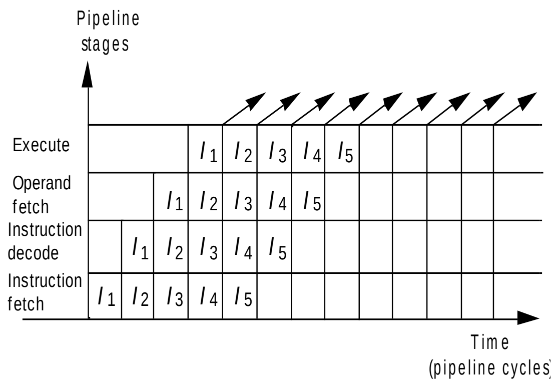


Figure2. Multiple instruction in parallel

Scheduling

The three stages are the different stages in a FFT diagram. The input data is getting loaded to the input buffer. The data gets processed to the three stages. Computation is done. The ready signal indicates that the data is ready to be sent out. Since the number of loops available in the calculation of FFT is greater, we adapt the outer loop pipelining of SSP approach.

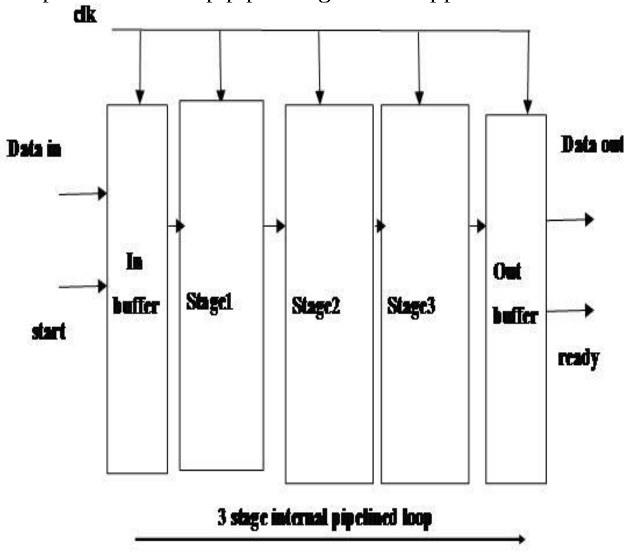


Figure3. Block Diagram for 3staged Pipelining

The three stages of the fast Fourier transform are described above.

The most profitable loop is selected for which the scheduling is to be done. The middle stage of FFT is chosen for scheduling. The technique is same for calculation of decimation in time as well as frequency.

Architectures of The FFT

There are many hardware implementations for both DIF and DIT algorithms. For instance, we can choose between digit-serial or bit-parallel arithmetic, or we can select between pipelined or iterative implementations. In the case of data-oriented applications presenting a continuous flow of samples, the best architectures are those that favor speed over area. The implementations that better fit these requirements are bit-parallel and pipelined architectures, where the processing is performed in several cascaded stages, We have chosen two main groups of FFT architectures, representing opposite points in the area-performance design space: feedback (FB) and feed forward (FF) architectures.

Architectures with FB provide the output flow at the clock frequency (one sample per clock cycle), because the feedback structure allows the reuse of some elements present in every stage. On the other hand, FF structures provide a higher throughput (R samples per clock cycle, R being the radix) because reuse is not applied and higher concurrency can be obtained, paying the price of a significant area overhead. Rotators are critical components in the FFT architecture because they require a significant percentage of the total area.

They are mainly composed of a first element that multiplies data by the twiddles and a memory that stores the twiddles. Here the rotator uses the CORDIC algorithm which allows performing the multiplication by the twiddles without multipliers. This algorithm performs the rotation of a complex vector by means of a series of shifts and additions.

Every shift rotates the vector components a given angle from a set of elemental angles. This algorithm presents an intrinsic gain of approximately 1.647 Therefore, to keep the dynamic range of the input samples, this element would have to increase the data bit width by one bit. As was explained for the butterfly, this extra bit can be truncated after rotation takes place or it can be kept. It is important to remark that overflow is avoided in any case.

Pipelining Above the Innermost loop

When pipelining above the innermost loop, a simple controller is needed to be implemented.

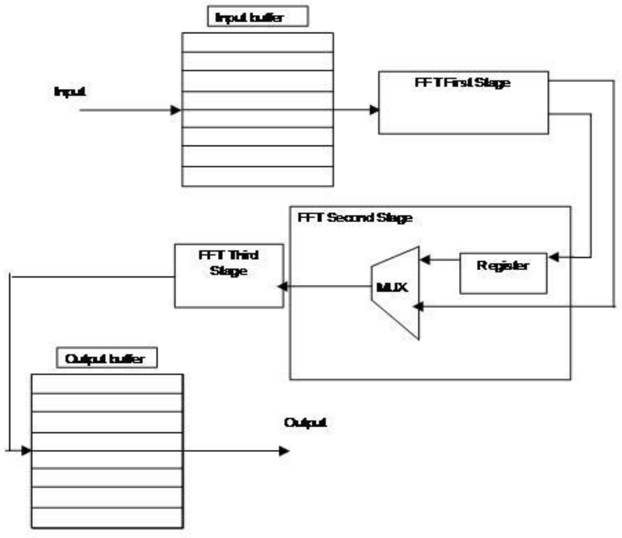


Figure 4. Block Diagram for Outer Loop Pipelining

- The first group of butterfly diagram is numbered from stag10, stag11……………………….stag16, stag17.
- The middle group of butterfly diagram is numbered from stag20, stag21……………………..stag26, stag27.
- The third group of butterfly diagram is numbered from stag30, stag31……………………….stag36, stag27.
- Since the middle group comprises a large number of computations, scheduling is done with the second stage.
- The most profitable loop is choosen.
- The second stage now contains registered inputs.
- A multiplexer is used to select the signal.
- When the mux selects 00 the outerloop pipelining is done. The registered inputs are being processed.
- When the mux selects 01, 10, 11 the innerloop pipelining is done. The registered inputs are not considered.
- First group of butterfly comprises of stag10,stag11…stag 17
- Middle group of butterfly comprises of stag20,stag21…stag 27
- In the middle stage of computation stage26,stage27 are given registered input, since the delay taken by these two stages are high
- When the mux selects 00 registered inputs are processed
- Last group of butterfly comprises of stag30,stag31…stag 37
- Output is taken from the output buffer

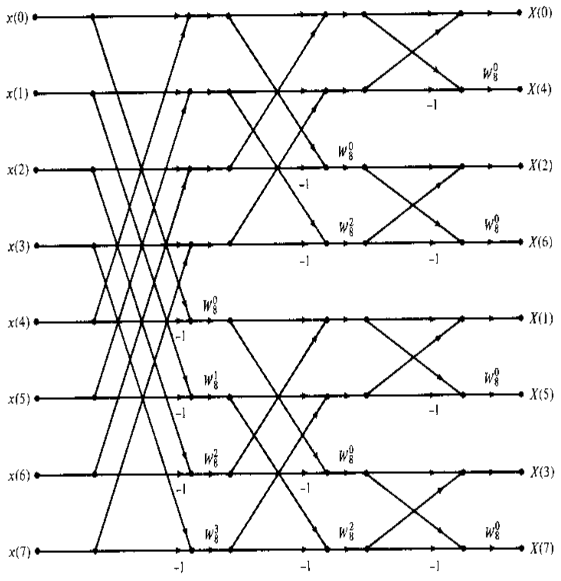


Figure5. N = 8-point decimation-in-frequency FFT algorithm

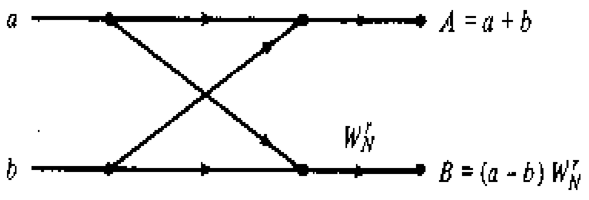


Figure6. Basic butterfly computation in the decimation-in-frequency

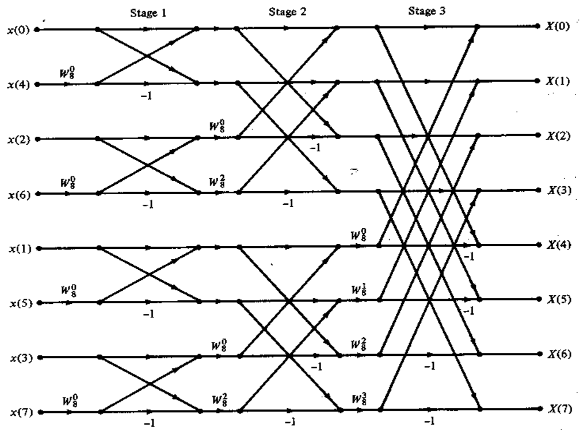


Figure 7. N = 8-point decimation-in-time FFT algorithm.

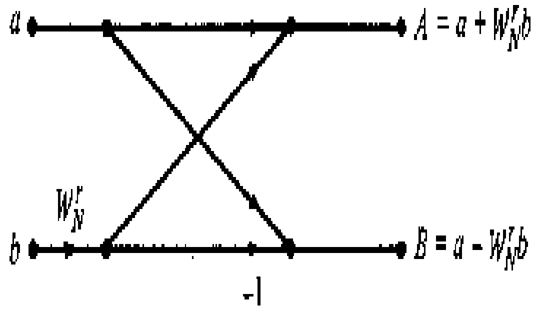


Figure 8. Basic butterfly computation in the decimation-in-time

- First group of butterfly comprises of stag10,stag11…stag 17
- Middle group of butterfly comprises of stag20,stag21…stag 27
- In the middle stage of computation stage23,stage27 are given registered input, since the delay taken by these two stages are high
- When the MUX selects 00 registered inputs are processed
- Last group of butterfly comprises of stag30,stag31…stag 37
- Output is taken from the output buffer

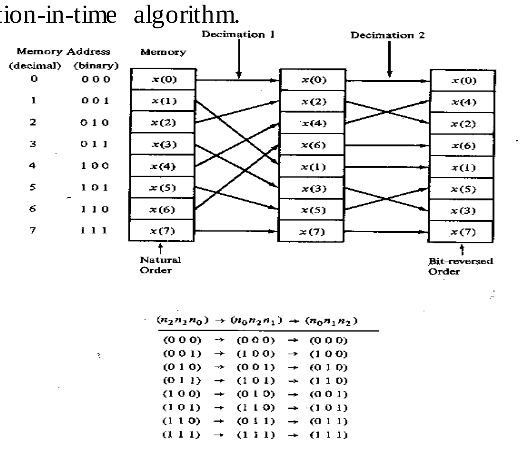
An important observation is concerned with the order of the input data sequence after it is decimated (v-1) times. For example, if we consider the case where N = 8, we know that the first decimation yields the sequence x(0), x(2), x(4), x(6), x(1), x(3), x(5), x(7), and the second decimation results in the sequence x(0), x(4), x(2), x(6), x(1), x(5), x(3), x(7). This shuffling of the input data sequence has a well-defined order as can be ascertained from observing Figure below, which illustrates the decimation of the eight-point sequence

Let us consider the computation of the $N = 2^v$ point DFT by the divide-and conquer approach. We split the N-point data sequence into two N/2-point data sequences $f_1(n)$ and $f_2(n)$, corresponding to the even-numbered and odd-numbered samples of x(n), respectively, that is,

$$f_1(n) = x(2n)$$

$$f_2(n) = x(2n+1), \qquad n = 0,1,\ldots,N/2-1$$

Thus $f_1(n)$ and $f_2(n)$ are obtained by decimating x(n) by a factor of 2, and hence the resulting FFT algorithm is called a decimation-in-time algorithm.



```
(n2n1n0) → (n0n2n1) → (n0n1n2)
(0 0 0)  → (0 0 0)  → (0 0 0)
(0 0 1)  → (1 0 0)  → (1 0 0)
(0 1 0)  → (0 0 1)  → (0 1 0)
(0 1 1)  → (1 0 1)  → (1 1 0)
(1 0 0)  → (0 1 0)  → (0 0 1)
(1 0 1)  → (1 1 0)  → (1 0 1)
(1 1 0)  → (0 1 1)  → (0 1 1)
(1 1 1)  → (1 1 1)  → (1 1 1)
```

Figure9. Shuffling of the data and bit reversal.

It is well known that the Fourier Transform (FT) has currently a key role in signal processing applications. The FT is useful for frequency domain analysis of a signal, i.e. it transposes a signal from time domain into frequency domain. Many applications ranging from telecommunication, electric energy distribution systems, fail prevention analysis and general signal processing use this transform as a tool for coding/decoding or spectrum analysis of a signal. Because of the complexity of the processing algorithm of FT and its importance in signal analysis, many people have been working on methods and application specific processor architecture for improving the computation performance. Butterfly highly parallel.

Simulation Results

- In the loop of a DIF FFT when the scheduling is not done
- When the MUX selects 01
- The inputs of stage 26, stage27 are not registered and the delay taken by these two stages are high.

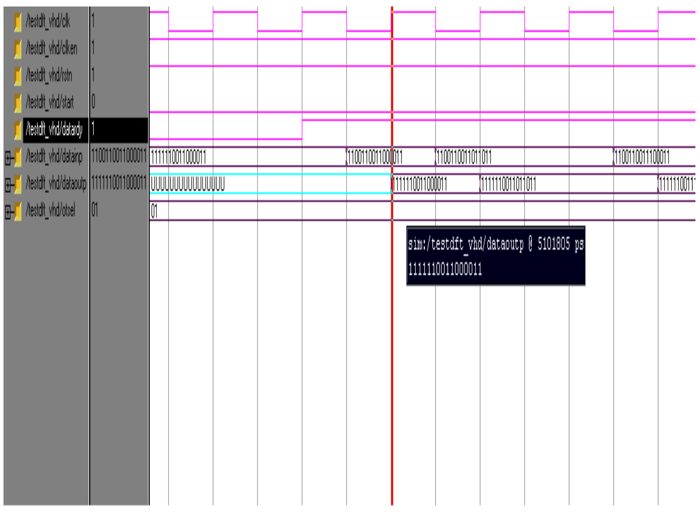


Figure10. When Scheduling Is Not Done In DIF

Time taken for the computation = 5101805ps

- In the loop of a DIF FFT after the scheduling while outerloop pipelining:
- When the MUX selects 00
- The inputs of stage 26, stage27 are registered and the delay taken by these two stages are controlled by outerloop pipelining.
- The simulation result is given below

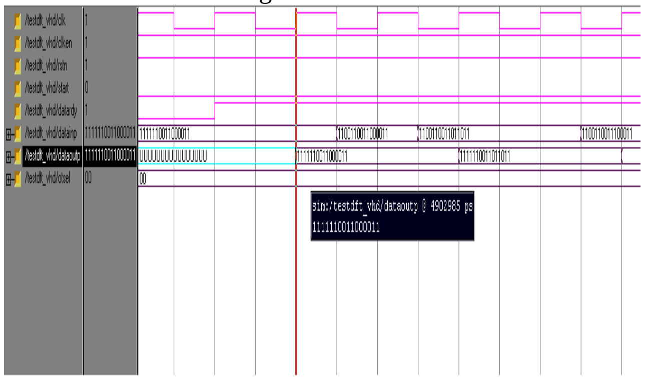


Figure11. When Scheduling Is Done In DIF

Time taken for the computation = 4902985ps

- In the loop of a DIT FFT when the scheduling is not done:
- When the MUX selects 01
- The inputs of stage 23, stage27 are not registered and the delay taken by these two stages are high.
- The simulation result is given below

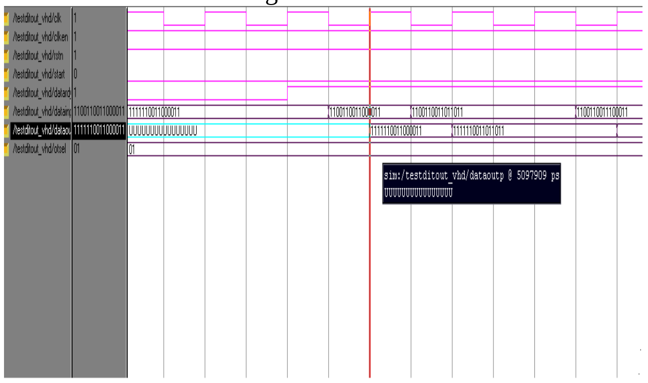
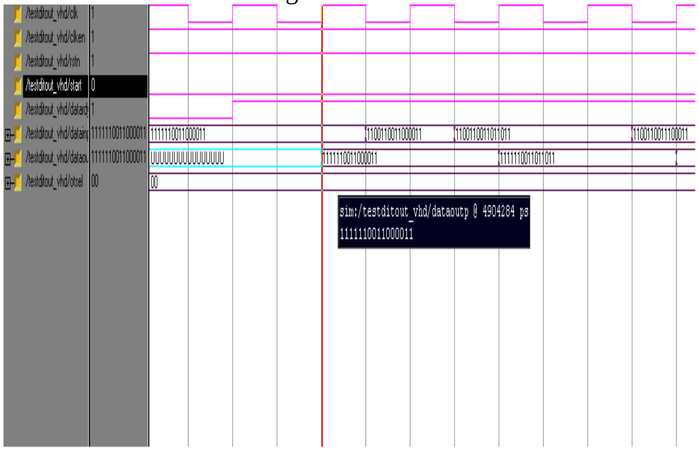


Figure12. When Scheduling Is Not Done In DIT

Time taken for the computation = 5097909ps

- In the loop of a DIT FFT after the scheduling while outerloop pipelining :

- When the MUX selects 00
- The inputs of stage 23, stage27 are registered and the delay taken by these two stages are controlled by outerloop.
- The simulation result is given below



```
sim:/testditout_vhd/dataoutp @ 4904284 ps
1111110011000011
```

Figure13. When Scheduling Is Done In DIT

Time taken for the computation = 4904284ps

The FFT is a fast implementation of the Discrete Fourier Transform (DFT), indicated by. It is based on a divide-and-conquer model, by which the discrete transform is divided into smaller and simpler transforms, and from these simpler transforms, the whole transform is obtained .The divide-and-conquer model is based on the idea that a *N*-point DFT computation can be divided into two *N/2*-point DFT computation. These *N/2*-point DFT computations can be divided into two *N/4*-point DFT computation, and so on. Actually, the division occurs after a reorganization of the points, so that each point corresponds to a two points DFT in each position when using a radix-2 method, for example. After the division and the DFT computation, a merging process is performed, in which the transforms are reassembled Nowadays semiconductor technology is able to create very complex devices that can enclose a complete system in a single chip (SoC). If the system is created from scratch, achieving the desired performance is costly and time consuming. To meet the tight time-to-market requirement, the electronic design uses pre-designed intellectual property (IP) cores as a common practice. These cores may be parametrizable and customizable to be synthesized in a large application specification. They are available to the designer from heterogeneous sources, design team, CAD tool libraries, CAD tool independent libraries, etc. One of the areas that major demands of application specific circuits design is digital signal processing (DSP). Fast Fourier Transform is a computationally intensive DSP function, widely used in many applications

HDL Synthesis Report

```
Macro Statistics
# Adders/Subtractors                    : 1
 4-bit adder                            : 1
# Registers                             : 47
 1-bit register                         : 2
 16-bit register                        : 43
 4-bit register                         : 2
# Xors                                  : 74
 1-bit xor2                             : 60
 16-bit xor2                            : 14
=======================================
*      Advanced HDL Synthesis           *
=======================================
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <FSM_0> on signal <nxt_state[1:3]> with sequential encoding.
-------------------
 State | Encoding
-------------------
 st0   | 000
 st1   | 001
 st2   | 011
 st3   | 100
 st31  | 010
 st4   | 101
 st5   | 110
-------------------
Advanced HDL Synthesis Report
Macro Statistics
# Adders/Subtractors          : 1
 4-bit adder                  : 1
# Registers                   : 42
 Flip-Flops                   : 42
# Xors                        : 14
 16-bit xor2                  : 14
```

Performance analysis

- For both the computations of DIF FFT and DIT FFT we get faster results when pipelining above the innermost loop.
- SSP combines both the techniques of inner as well as outerloop pipelining.
- Our extended Single Dimension Software Pipelining algorithm has been used to pipeline each level in nine nested loops.
- The pipelined data path for each loop level is implemented manually in VHDL based on the schedule produced by our tool.
- The results are faster with outerloop pipelining.

The VHDL for the pipeline controller for each case is generated automatically by our scheduling tool from the set of parameterized component blocks described in the previous section

Table I

Computation	time taken by inner loop	Time taken by outer loop
DIF FFT	5101805ps	4902985ps
DIT FFT	5097909ps	4904284ps

Advantages

- Reduction in the number of cycles spent flushing and filling the pipeline and the potential for data reuse.
- Computation takes place at a faster rate when compared with inner loop pipelining.
- Time consumption is minimum.
- Since SSP combines both loops, the user can select any one which suits for that particular application.

Conclusion

In this work an existing methodology for pipelining software loops above the innermost loop level has been adapted for use in generating FPGA based hardware co-processors. The Single-dimension Software Pipelining (SSP) method for a multi-dimensional loop nest chooses the most profitable loop level in the loop nest and software pipelines it. Our scheduling tool has been applied to test loops of FFT. The fastest solution is found when the loop is pipelined above the innermost loop. The results suggest that inclusion of outer loop pipelining in future hardware compilers may be worthwhile as it can allow significantly improved results to be achieved at the cost of a small increase in compile time.

Future work

Our experimental results were based upon the test loops of FFT. In many typical DSP applications, loops comprise a majority of the number of cycles, or MIPS. Because of this, performance of loops can greatly affect the performance of the entire application. Therefore one possible future work is to investigate with the loops of FIR, IIR filters and DCT.

References

[1] Kieron Turkington, George A. Constantinides, Konstantinos Massselos, and Peter, Outer Loop pipeling for Application Specific Datapaths in FPGAs, pp 1- 13,2007.

[2] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. Proceedings in SIGPLAN' 88 Conference on Programming Language Design and Implementation (PLDI), pp. 318-328, 1988.

[3] D. Petkov. Efficient Pipelining of Nested Loops: Unrolland-Squash, M.Eng. Thesis, Massachusetts Institute of Technology, pp, 1-5 January 2001.

[4] Hongbo Rongy, Alban Douillety, R. Govindarajan z, Guang R. Gaoy Code Generation for Single-Dimension Software Pipelining of Multi-Dimensional Loops. University of Delaware, New York pp 1-12, 2004.