On Dataflow Computing With OpenSPL

(Draft v1.0-79-g838c34b)

John Winans
jwinans@niu.edu

October 20, 2015
# Contents

1 Introduction ......................................................... 1  
1.1 The FPGA ......................................................... 1  
1.1.1 Evolution .................................................... 1  
1.1.2 HDL Programming ........................................... 2  
1.1.3 The Machine ................................................ 3  
1.2 Some Words From The Marketing Department ..................... 4  

2 Dataflow Computing .................................................. 5  
2.1 An Example Problem ............................................. 6  
2.2 Problem Solving With a CPU .................................... 6  
2.3 Problem Solving With Pipelined Computing ..................... 9  
2.3.1 The Kernel Graph .......................................... 9  
2.3.2 Timing Diagrams for \( y = x^2 + z^2 \) ....................... 9  
2.4 Observations .................................................... 11  

3 OpenSPL Basics ...................................................... 13  
3.1 Introduction ..................................................... 13  
3.2 An Accelerator Architecture ................................... 13  
3.2.1 CPU Code .................................................. 14  
3.2.2 MaxJ Code ................................................ 15  
3.2.3 Kernel ..................................................... 16  
3.2.4 Manager .................................................... 16  

4 Maxeler IDE ............................................................ 19  
4.1 Accessing the OpenSPL Environment ............................ 19  
4.1.1 Installation of VM .......................................... 19  
4.1.2 \texttt{hermes.niu.edu} ..................................... 19  
4.2 Maxeler First Time Use ......................................... 20  
4.2.1 Set up Your Workspace .................................... 20
5 Your First OpenSPL Program

5.1 Introduction ................................................. 27
   5.1.1 Create a New Project ................................. 27
      5.1.1.1 Create New MaxCompiler Project ................. 28
      5.1.1.2 Name the Project ................................ 29
      5.1.1.3 Set the DFE Hardware Type .................... 30
      5.1.1.4 Select the SLiC Interface Type ................ 31
      5.1.1.5 Inspect the Project Template Stub Files ....... 32
      5.1.1.6 CPU Code Template ................................ 32
      5.1.1.7 Kernel Engine Code Template .................... 33
      5.1.1.8 Manager Code Template ........................... 33
      5.1.1.9 Build & Simulate Your Project .................. 36
      5.1.1.10 Build and Run Messages ......................... 37
      5.1.1.11 Original Kernel Graph ......................... 38
      5.1.1.12 Final Kernel Graph .............................. 39
   5.2 Convert Template Code to Desired Application .......... 39
      5.2.1 Add Display Logic .................................. 40
      5.2.2 Replace the Manager Template With a Simple Manager 41
      5.2.3 Change the Computation in the Kernel ............. 42
      5.2.4 Update the CPU Code ................................ 42
      5.2.5 Final Program Output ................................ 43

6 The Kernel ...................................................... 45
   6.1 Introduction ............................................. 45
   6.2 Widening the Pipeline ................................. 45
      6.2.1 Overloading a Kernel ............................... 45
      6.2.2 An N-fold kernel ................................... 46
   6.3 Temporal Alignment ..................................... 48
      6.3.1 \( y = x^2 + z^2 + z \) .............................. 48
      6.3.2 \( y = x^2 + z^2 + z - x \) .......................... 51
      6.3.3 Reality Check (There’s a Pipeline in my Pipeline!) 52

A Installing and Using NX ...................................... 55
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Configurable Logic Block</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>ICSA Block Diagram</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>RTL description of $y[i] = x[i]^2 + z[i]^2$</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Two unoptimized iterations of $y[i] = x[i]^2 + z[i]^2$</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Two RTL-parallel iterations of $y[i] = x[i]^2 + z[i]^2$</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Three iterations of RTL unrolled loop version of $y[i] = x[i]^2 + z[i]^2$</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Three iterations of out-of-order RTL-parallel execution of $y[i] = x[i]^2 + z[i]^2$</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>A Kernel graph for $y = x^2 + z^2$</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>A naive pipelined implementation of $y[i] = x[i]^2 + z[i]^2$ for $x = {1, 2, 3}$ and $z = {4, 5, 6}$.</td>
<td>10</td>
</tr>
<tr>
<td>2.8</td>
<td>A properly pipelined version of $y[i] = x[i]^2 + z[i]^2$ for $x = {1, 2, 3, 4, 5, 6, 7, 8, 9}$ and $z = {4, 5, 6, 7, 8, 9, 10, 11, 12}$</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Compiling an OpenSPL application.[1, p. 20]</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>VM Desktop Icons.</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Create a new workspace</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>The MaxIDE Welcome Window</td>
<td>21</td>
</tr>
<tr>
<td>4.4</td>
<td>Import MaxCompiler Projects</td>
<td>22</td>
</tr>
<tr>
<td>4.5</td>
<td>Select Project</td>
<td>22</td>
</tr>
<tr>
<td>4.6</td>
<td>Open Example Project</td>
<td>23</td>
</tr>
<tr>
<td>4.7</td>
<td>Build and Run Simulation console window messages</td>
<td>24</td>
</tr>
<tr>
<td>4.8</td>
<td>Simulation output messages</td>
<td>24</td>
</tr>
<tr>
<td>4.9</td>
<td>Example kernel Graph</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Creating a new MaxCompiler project</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Name the new project</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Set DFE hardware type</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>Select the SLiC interface type</td>
<td>31</td>
</tr>
<tr>
<td>5.5</td>
<td>Open the template stub files</td>
<td>32</td>
</tr>
<tr>
<td>5.6</td>
<td>Build &amp; Run your project in simulation.</td>
<td>36</td>
</tr>
</tbody>
</table>
5.7  Build and run messages in the console tab. ........................................ 37
5.8  The original kernel graph. ............................................................. 38
5.9  The final kernel graph. ............................................................... 39
5.10 Final kernel graph of $s = x^2 + 30a$. .......................................... 44
6.1  Multiple outputs from the same data stream. .................................. 46
6.2  An inefficient use of data (in contrast to Figure 6.1). ......................... 47
6.3  A primitive way to implement an N-fold kernel. .............................. 48
6.4  A (Broken) Kernel graph for $y = x^2 + z^2 + z$ ............................ 49
6.5  A Kernel graph for $y = x^2 + z^2 + z$ .......................................... 49
6.6  Six pipelined iterations of $y[i] = x[i]^2 + z[i]^2 + z$ for $x = \{1, 2, 3, 4, 5, 6\}$ and $z = \{4, 5, 6, 7, 8, 9\}$ .................................................... 50
6.7  A Kernel graph for $y = x^2 + z^2 + z - x$ .................................... 51
6.8  Six pipelined iterations of $y[i] = x[i]^2 + z[i]^2 + z - x$ for $x = \{1, 2, 3, 4, 5, 6\}$ and $z = \{4, 5, 6, 7, 8, 9\}$ .................................................... 52
6.9  Optimized kernel graph of $x^2 + x$. ............................................. 52
6.10 Timing diagram for $s = x^2 + x$ .................................................... 53

A.1 Set session and host names. ......................................................... 55
A.2 Accept the defaults. ................................................................. 56
A.3 Chose to place on desktop (or not). .............................................. 56

E.1 Right-click on your project and open Team->ShareProject. .................. 67
E.2 Choose SVN and click Next. ....................................................... 67
E.3 Select Create New... and click Next. ........................................... 68
E.4 Enter the URL for the repository and click Next. ............................. 68
E.5 Select Use Project Name... and click Next. ................................... 69
E.6 Enter a suitable commit comment and click Finish. .......................... 69
E.7 Right-click on your project and open Team->Commit. ....................... 70
E.8 Enter a suitable comment and click OK. ....................................... 70
E.9 Eclipse indicates the SVN version next to each file in the Project Explorer. 71
E.10 Eclipse indicates out of date files with a brown decoration on related Project Explorer icons. ............................................................. 71
E.11 Committing the project files changes the version number of \texttt{a1CpuCode.c} as seen in the Project Explorer. ........................................ 72
1

Introduction

Dataflow computing was popularized by a number of researchers in the 1980’s, especially J. B. Dennis. In the dataflow approach an application is considered as a dataflow graph of the executable actions; as soon as the operands for an action are valid, the action is executed and the result is forwarded to the next action in the graph. There are no load or store instructions as the operational node contains the relevant data. Creating a generalized interconnection among the action nodes proved to be a significant limitation to dataflow realizations in the 1980’s. Over recent years the extraordinary improvement in transistor array density allowed emulations of the application dataflow graph. The Maxeler dataflow implementations are a generalization of the earlier work employing static, synchronous dataflow with an emphasis on data streaming. Indeed “multiscale” dataflow incorporates vector and array processing to offer a multifaceted parallel compute platform.[1]

1.1 The FPGA

An FPGA (Field Programmable Gate Array) is a type of integrated circuit (as opposed to a computing system consisting of many parts) that, as its name implies, can be programmed to perform various functions.

The fact that it is field programmable means that it can be programmed after it has left the factory. Being a gate array, it is programmed by specifying the manner in which the its gates are to be interconnected.

1.1.1 Evolution

Over the years FPGA manufacturers have improved upon the operations that the so-called “gates” can perform to the point where the more advanced devices are far from containing just simple logic gates. In spite of the continued presence of the word “gate” in their name, an FPGA is an array of CLBs (Configurable Logic Blocks) that range from simple logic to complex truth tables (called LUTs) and
simple memories and a plurality of other types of **Hard IP**\(^1\) such as mathematical units of various types, specialized control units (for accessing large memories), communication units (for Ethernet links, PCIe, and others) and even whole multicore CPUs such as the Sparc or ARM.

A simplified CLB block diagram is shown in Figure 1.1. The LUT contains a **truth table** with as many rows as the quantity of input bits enumerate. The **clock** signal and associated D-Latch comprise a **register** that is used to store/remember the last value that was “looked up” in the truth table.\(^2\)

![Figure 1.1: Configurable Logic Block](image)

Programming an FPGA consists of specifying 1) the values of the bits in the LUTs and 2) a network map (referred to as a netlist) that describes which signals (the bits) that flow out of one block (a CLB or some Hard IP) and into another. Ultimately the signals originate on some of the pins of the FPGA chip and terminate at others which are, in turn, connected to other devices such as an Ethernet, large memory chip(s) and/or the PCIe signals in a PC so that the FPGA can interact with the rest of the world and perform useful work.

Implementing a function in an FPGA is therefore a task of expressing it in the form of a **netlist**. To create a netlist by hand would be an outrageous task as a modern FPGA would contain many millions of connections. Therefore a specialized high level language is used.

### 1.1.2 HDL Programming

Since the late ’80s, languages such as **Verilog** and **VHDL**, both known as HDLs (**Hardware Description Language**), have been used to program FPGAs. These languages are akin to using an assembly language to program a CPU. (If we continue this analogy downward then creating a netlist by hand would be akin to typing in CPU machine code in binary.) While assembly code is necessary for some specific functions and can often result in the most efficient execution of a program on some CPUs, the

\(^1\)Hard IP (**Intellectual Property**) refers to commonly used functions that might have been historically programmed into the FPGA by using multiple CLBs. But are more efficiently built by dedicating part of the silicon of the chip to a specific purpose.

\(^2\)Actual CLBs include additional components like a Full Adder because implementing them using LUTs would be notably slower and consume more power.
additional effort required and lack of portability often drive programmers to use simpler higher level languages like C or Java... at the expense of (possibly) ending up with slower-performing code.

Where FPGA programming is concerned, the next “higher-level” languages are being invented and discovered right now. One such language is called OpenSPL and is the subject of the rest of this book. OpenSPL is expressed as a combination of C and Java.

1.1.3 The Machine

The applications discussed in this book were designed to execute on a Maxeler DFE (Data Flow Engine) board. An DFE is what is generally known as a co-processor or application accelerator because it is connected to a traditional computer in the form of a peripheral device similar to a hard drive or audio interface as seen in Figure 1.2

As a PC peripheral device, the DFE board is connected to the PCIe (Peripheral Component Interconnect Express) bus. The PCIe bus is a set of high-speed serial lanes. The DFE upon which this text focuses is the Maxeler ICSA. The ICSA has eight lanes.

As an eight-lane PCIe device, the ICSA DFE can exchange up to eight simultaneous streams of data with the main memory of a host PC. These streams represent one of the types of I/O that an OpenSPL program can use. Other types of I/O include various types of memory and serial interfaces such as Ethernet that can be connected directly to an FPGA.

Note that from the perspective of an FPGA even memory starts to “look” and act like a peripheral device in that it requires the application to read from and write to it!

Each of the (on the order of) 1,000,000 CLBs operates independently, providing a great deal of fine-grained parallelism.

--
3For sake of completeness it is important to point out that FPGA (stand-alone) applications do not require connection to a computer. Other applications are implemented using FPGAs that include an entire CPU within the FPGA.
CHAPTER 1. INTRODUCTION

By connecting CLBs together to create complex functions and connecting the output of one function to the input of another, one or more chains or pipelines can be created that can receive/read one or more streams of data, process it in some way and then transmit/write one or more resulting streams of data. OpenSPL is well suited for implementing solutions to such a “streaming” application.

1.2 Some Words From The Marketing Department

On Monday, June 1, 2015 computer processor company Intel announced that it will buy Altera, one of the two primary vendors of field programmable gate arrays (FPGAs). Intel will spend nearly $17 billion in cash for Altera. The two companies already have a working relationship as Altera builds some of its FPGAs in Intel semiconductor fabrication facilities.

Generally, Intel’s processor chips and Altera’s special programmable (in a different way) circuits are quite different and are used in different, though possibly adjoining, spaces. Gate arrays can perform functions 10 times as fast as instruction sequences running through clocked processors, but the processors are far more flexible than FPGAs. FPGAs are a nice middle-ground between instruction-set processors and hard-wired circuitry, but they come at a cost of high price and high power, each with mitigating conditions [2]

From a June 2015 press release:

Stratix® 10 FPGAs and SoCs combine the industry’s highest performance (2X), and highest density (5.5MLEs) with advanced embedded processing capabilities (quad-core ARM® Cortex®-A53), GPU-class floating-point computation performance of up to 10 Tera floating-point operations per second (TFLOPS), heterogeneous 3D system-in-package (SiP) integration, and the most advanced security capabilities in a high-performance FPGA.

From a June 2014 white paper:

Single-precision floating-point performance on popular high-speed platforms [3]

- Texas Instruments’ TMS320C667x DSP = .16 TFLOPs.
- NVIDIA Tesla K20 GPU = 3.520 TFLOPs.
- Altera high-end Stratix 10 FPGA = 10 TFLOPs.

Searching the Internet for maximum performance numbers on Intel processors is tough since there are so many variations available. As of Q4 2014, it appears that the fastest Intel CPUs are capable of approximately 1 TFLOP.

Keep in mind that all of these “maximum speeds” are theoretical and are not likely to be achieved unless one is extremely careful about designing and writing code to suite the needs of each of the specific devices.
Dataflow Computing

This chapter is an introduction to the concepts of data flow programming.

A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.[4]

—Gene M. Amdahl, 1967
IBM

The rephrased version of the above statement is known as Amdahl’s law:

The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.¹

Slotnick’s Law:

The parallel approach to computing does require that some original thinking be done about numerical analysis and data management in order to secure efficient use.

In an environment which has represented the absence of the need to think as the highest virtue this is a decided disadvantage.

—Daniel Slotnick, 1967
Chief Architect
Illiac IV

It is the purpose of this text to discuss some original thinking about numerical analysis and data management while keeping an eye on the requirements of sequential processing in order to maximize the performance of an application.

¹See [5, Section 7.12] for a discussion of the pitfalls of improperly interpreting Amdahl’s Law.
2.1 An Example Problem

We are all familiar with writing computer programs for conventional computing systems that are executed in a serial manner on a general purpose CPU. With that in mind let us consider a simple C function that implements: \( y[i] = x[i]^2 + z[i]^2 \). (See [6] for a discussion of a Hadamard product and matrix multiplication.)

In C this can be implemented as shown in Listing 2.1

Listing 2.1: exampleFunction.c
A C implementation of \( y[i] = x[i]^2 + z[i]^2 \)

```
void f(float *x, float *y, float *z, int length)
{
    int i;
    for (i=0; i<length; ++i)
    {
        y[i] = x[i]*x[i] + z[i]*z[i];
    }
}
```

As a CPU iterates over the body of the loop, there are multiple operations that take place to compute the right side of the assignment statement, they all must complete before the assignment is made on line 6, and the assignment must be made before proceeding to the next iteration of the loop.

2.2 Problem Solving With a CPU

Focusing only on the body of the loop, we know that a CPU will perform following operations one at a time (described in Figure 2.1 using RTL (Register Transfer Language[7]) notation.)

\[
\begin{align*}
    t_1 & \leftarrow x[i] \\
    t_1 & \leftarrow t_1 \times t_1 \\
    t_2 & \leftarrow z[i] \\
    t_2 & \leftarrow t_2 \times t_2 \\
    t_2 & \leftarrow t_1 + t_2 \\
    y[i] & \leftarrow t_2 \\
    i & \leftarrow i + 1
\end{align*}
\]

Figure 2.1: RTL description of \( y[i] = x[i]^2 + z[i]^2 \)

Using a timing diagram\(^3\) we can see how and when the ALU (Arithmetic Logic Unit) and the memory interface units of the CPU are used over the course of time while the CPU executes the RTL in Figure 2.1. A timing diagram shows what operations take place in each functional unit over a continuum of time. When a unit is performing a useful task, such as squaring the number \( a \), that particular operation

---

\( ^2 \)RTL is commonly used as an intermediate language in compilers

\( ^3 \)For more information on timing diagrams see: http://en.wikipedia.org/wiki/Digital_timing_diagram
is indicated with \((a-a)\). Note that, as depicted here, any input(s) to an operation are read/sampled once at the beginning of the time period and the results of the operation are provided at the end and are held stable until the next output value is generated. An idle unit is indicated with a gray time period: \(\square\). The width of an item in a timing diagram is proportional the amount of time used to execute the specified operation. The position on the diagram’s horizontal axis represents the span of time over which the operation takes place.

The background colors in the timing diagram in this book have been chosen to indicate the type of operation being performed. ALU operations are displayed in blue. Memory transfers, in amber.

In the simplest case, where a CPU can only do one thing at a time. Let us assume that each of the 7 operations in our loop body take the same amount of time to complete. Each iteration of the loop body results in the execution of the same operations in the same order. Figure 2.2 is a timing diagram showing how the the first two iterations of our loop perform a total of 7 × 2 = 14 operations, each consuming one unit of time. That our example CPU can only do one thing at a time is made evident by that fact that only one of the units is not idle at any point in time.

![Figure 2.2: Two unoptimized iterations of \(y[i] = x[i]^2 + z[i]^2\)](image)

If a CPU is capable of exchanging data with memory at the same time that it is performing an operation with its ALU and it can “look ahead” in the instruction stream then it is possible for it to optimize the use of its functional units by scheduling more than one at the same time.\[^{[9]}\] As long as care is taken to ensure that the data required for any given operation is present when the operation starts, operating the units in parallel will reduce the time that it takes to execute the body of our loop from 7 to 5 units per-iteration. Figure 2.3 shows how a total of 5 × 2 = 10 units of time are used to complete two iterations of our loop when CPU schedules its functional units in parallel. The performance improvement is evident as the same 14 total units of time are allocated to the same 14 operations. The only difference is when they have been scheduled to take place.

![Figure 2.3: Two RTL-parallel iterations of \(y[i] = x[i]^2 + z[i]^2\)](image)

We can, however, change the RTL to better suit our needs. Using a technique called loop unrolling\[^{[10]}\] we can rewrite our program showing the iterations of our loop in the form of one long instruction stream. To better illustrate what is happening we will now consider three iterations of our loop body and introduce an additional variable \(k\) that we will use along with \(i\) as our index counter as shown in Figure 2.4.

\[^{[4]}\]The notion of any part of a machine being “idle” is a misnomer. Unless the power is removed, nothing actually stops per se. When used in the context of a timing diagram or pipeline, idle literally indicates that the specified unit’s activities are not consequential because its output will go unused during the indicated idle period. As an optimization, modern processors will dispatch specific instructions that are known to consume the least amount of power during such idle periods. It is easy to demonstrate the results of this by detecting the temperature (and fan speed) changes of a laptop when its activity changes from idle to busy.

\[^{[10]}\] We can rewrite our program showing the iterations of our loop in the form of one long instruction stream. To better illustrate what is happening we will now consider three iterations of our loop body and introduce an additional variable \(k\) that we will use along with \(i\) as our index counter as shown in Figure 2.4.

\[^{[9]}\] As long as care is taken to ensure that the data required for any given operation is present when the operation starts, operating the units in parallel will reduce the time that it takes to execute the body of our loop from 7 to 5 units per-iteration. Figure 2.3 shows how a total of 5 × 2 = 10 units of time are used to complete two iterations of our loop when CPU schedules its functional units in parallel. The performance improvement is evident as the same 14 total units of time are allocated to the same 14 operations. The only difference is when they have been scheduled to take place.

![Figure 2.3: Two RTL-parallel iterations of \(y[i] = x[i]^2 + z[i]^2\)](image)

We can, however, change the RTL to better suit our needs. Using a technique called loop unrolling\[^{[10]}\] we can rewrite our program showing the iterations of our loop in the form of one long instruction stream. To better illustrate what is happening we will now consider three iterations of our loop body and introduce an additional variable \(k\) that we will use along with \(i\) as our index counter as shown in Figure 2.4.

\[^{[10]}\] We can rewrite our program showing the iterations of our loop in the form of one long instruction stream. To better illustrate what is happening we will now consider three iterations of our loop body and introduce an additional variable \(k\) that we will use along with \(i\) as our index counter as shown in Figure 2.4.

\[^{[9]}\] As long as care is taken to ensure that the data required for any given operation is present when the operation starts, operating the units in parallel will reduce the time that it takes to execute the body of our loop from 7 to 5 units per-iteration. Figure 2.3 shows how a total of 5 × 2 = 10 units of time are used to complete two iterations of our loop when CPU schedules its functional units in parallel. The performance improvement is evident as the same 14 total units of time are allocated to the same 14 operations. The only difference is when they have been scheduled to take place.

```
mem
<table>
<thead>
<tr>
<th>t₁ ← x[0]</th>
<th>t₁ ← x[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₂ ← z[0]</td>
<td>t₂ ← z[0]</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>y[0] ← t₁</td>
<td>y[0] ← t₁</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>t₂ ← z[0]</td>
<td>t₂ ← z[0]</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>t₁ ← x[1]</td>
<td>t₁ ← x[1]</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>y[1] ← t₂</td>
<td>y[1] ← t₂</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>t₂ ← z[1]</td>
<td>t₂ ← z[1]</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>i ← i + 1</td>
<td>i ← i + 1</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>t₂ ← t₂ × t₂</td>
<td>t₂ ← t₂ × t₂</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>t₁ ← t₁ × t₁</td>
<td>t₁ ← t₁ × t₁</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>y[1] ← t₂</td>
<td>y[1] ← t₂</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
</tbody>
</table>
```

```
ALU
<table>
<thead>
<tr>
<th>t₁ ← t₁ × t₁</th>
<th>t₁ ← t₁ × t₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₂ ← t₂ × t₂</td>
<td>t₂ ← t₂ × t₂</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>t₁ ← t₁ × t₁</td>
<td>t₁ ← t₁ × t₁</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>y[1] ← t₂</td>
<td>y[1] ← t₂</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>i ← i + 1</td>
<td>i ← i + 1</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
</tbody>
</table>
```

Figure 2.3: Two RTL-parallel iterations of \(y[i] = x[i]^2 + z[i]^2\)
CHAPTER 2. DATAFLOW COMPUTING

Out-of-order execution

\[
\begin{align*}
&k \leftarrow i + 1 & i \leftarrow k + 1 & k \leftarrow i + 1 \\
&t_1 \leftarrow x[i] & t_1 \leftarrow x[k] & t_1 \leftarrow x[i] \\
&t_1 \leftarrow t_1 \times t_1 & t_1 \leftarrow t_1 \times t_1 & t_1 \leftarrow t_1 \times t_1 \\
&t_2 \leftarrow z[i] & t_2 \leftarrow z[k] & t_2 \leftarrow z[i] \\
&t_2 \leftarrow t_2 \times t_2 & t_2 \leftarrow t_2 \times t_2 & t_2 \leftarrow t_2 \times t_2 \\
&t_2 \leftarrow t_1 + t_2 & t_2 \leftarrow t_1 + t_2 & t_2 \leftarrow t_1 + t_2 \\
&y[i] \leftarrow t_2 & y[k] \leftarrow t_2 & y[i] \leftarrow t_2
\end{align*}
\]

Figure 2.4: Three iterations of RTL unrolled loop version of \( y[i] = x[i]^2 + z[i]^2 \)

Using two counter registers (\( i \) and \( k \)) instead of one (\( i \)) and interleaving which is used during each of the original loop bodies we can now see how the operations can be reordered to better suit the capabilities of our two-unit CPU. Advanced CPUs are capable looking far enough ahead in the instruction stream to implement this type of optimization using what is called out-of-order execution.\[9\] The CPU now can use the ALU during the first time unit in each loop body to perform the counter increment for the next loop body and then it can relocate the \( t_1 \leftarrow x[i] \). As shown in Figure 2.4, out of order execution allows the first iteration of our loop to complete in 5 and the remaining iterations in 4 units of time completing 3 loops in less time that it took the original CPU design to do 2. The cost for this optimization is that the CPU has to allocate an additional register.

Using an extra register as a way to save time is a trade-off between space (silicon to make/use another register) and time (additional cycles required to perform operations that can not be scheduled to occur at other times).

The ALU is now saturated with work. Therefore we have gone as far as we can... with a CPU that has only two functional units.

Further optimization would require that we either eliminate some of the operations or add more functional units to further distribute the work. For example we could add an additional ALU and another path to access the memory in the system.

The extent to which adding functional units is helpful depends on how many of the operations must be completed before others can begin as well as the ability of the CPU to schedule the instructions across all of the units in an efficient manner.

While all of this is possible, some types of problems are more easily optimized by using a pipeline than a CPU.
2.3 Problem Solving With Pipelined Computing

Amdahl’s law states that gaining efficiency by performing operations in parallel is limited by the amount of time that is required to execute the longest single serial task. Serial, in this context, refers to the dependency path through the sequence of events that must performed in order.

2.3.1 The Kernel Graph

To better understand the dependency path in our application let us express it in the form of a network where edges will represent dependencies and nodes represent operations. Let us call our diagram a Kernel graph and draw it as shown in Figure 2.6. The multiplication units (the top two blue circles) have their input data flowing in from the \(x\) and \(z\) (inverted orange house) data-source units and their results flowing out to an adder unit (the bottom blue circle) that, in turn, has its output flowing to the \(y\) (orange house) data-sink unit. This notation was adopted from [1, p. 23].

![Kernel Graph Diagram]

Figure 2.6: A Kernel graph for \(y = x^2 + z^2\)

Expressed in this form we can see that the \(x\) and \(z\) data items can be (theoretically) fetched simultaneously because they have no direct or indirect dependencies shown in the graph. The two multiplication operations can also happen at the same time as long as they have data delivered from \(x\) and \(z\). The addition can not start until after both the multiplications have completed because the data items that the addition requires flows in from the multipliers. Finally, the delivery of the sum to the \(y\) data-sink can not start until the addition has completed.

2.3.2 Timing Diagrams for \(y = x^2 + z^2\)

At first glance it appears that 4 units of time is the best we can do. But, as Slotnick pointed out, “some original thinking” might offer additional opportunity.

Allocating and dedicating a functional unit for each node in our kernel graph and parallelizing them might
yield a machine that operates as shown in Figure 2.7 for the case when \( x = \{1, 2, 3\} \) and \( z = \{4, 5, 6\} \). For now let us continue to assume that \( i \) is (somehow) initialized to 0.

Our timing diagram now has a stair-step characteristic due to the chain of dependencies in the operations that can not be parallelized.

![Figure 2.7: A naive pipelined implementation of \( y[i] = x[i]^2 + z[i]^2 \) for \( x = \{1, 2, 3\} \) and \( z = \{4, 5, 6\} \).](image)

Note that our notation of labeling the timing diagram has changed. Since each “row” now represents a single-purpose dedicated functional unit, we can now indicate the one and only operation that each performs along the left edge of our diagram as opposed to within the time periods as was the case earlier when each of the units were used for different operations at different times. We can take advantage of this situation by now indicating the values of the data that are output by each functional unit at every point in time.

However, trading space for time by adding some more temporary registers we can see that the stair-steps can be collapsed as shown in Figure 2.8. This time we assume that \( i \) and \( k \) are initialized to 0 and that \( x = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \) and \( z = \{4, 5, 6, 7, 8, 9, 10, 11, 12\} \).

This time we use a separate counter for the \( x \) and \( y \) inputs (\( k \)) than we do for the \( y \) output (\( i \)) because, while they count the same things, they now have to do so at different times. We have also added enough temporary registers so that there is now one for every edge in our Kernel graph.

![Figure 2.8: A collapsed pipelined implementation of \( y[i] = x[i]^2 + z[i]^2 \) for \( x = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \) and \( z = \{4, 5, 6, 7, 8, 9, 10, 11, 12\} \).](image)

By not reusing any registers for more than one specific purpose, we have eliminated the need for any functional unit to wait on any other unless the two have a problem-specific data-dependency (represented by an edge in the Kernel graph) between them.

We can now see that the first “iteration” of our loop takes 4 time units and the rest each take 1. The first 3 time units in Figure 2.8 represent what is called filling the pipeline. The last 3 time units represent what is called flushing the pipeline.

What originally required \( 7 \times 3 = 21 \) time units now takes \( 3 + 1 \times 3 = 6 \).
2.4 Observations

The following observations may now be made:

1. The more data elements we process the greater the advantage gained by our pipelined implementation due to amortization of the pipeline fill and flush costs. Therefore: complexity \( n \to \infty = O(n) \).

2. The complexity of the function implemented will determine the number of stages required in the pipeline.

3. The number of stages in the pipeline will define the latency in our design. Latency is the amount of time between the arrival of data element(s) at the input unit(s) and the corresponding result leaving the output unit(s).

4. The duration of one time unit is equal to the latency divided by the number of stages in our pipeline.

5. Each stage in a pipeline runs at the same speed.

6. The period of time used for the tick rate of a pipeline is defined by the slowest stage.

7. For each tick of the pipeline one input value is consumed and one output value is produced.\[1, p. ?\]

8. In this particular case, once filled (and prior to flushing), every unit in the pipeline is doing useful work all the time.

Figure 2.8: A properly pipelined version of \( y[i] = x[i]^2 + z[i]^2 \) for \( x = \{1, 2, 3, 4, 5, 6, 7, 8\} \) and \( z = \{4, 5, 6, 7, 8, 9, 10, 11, 12\} \).
3

OpenSPL Basics

This chapter provides a brief overview of OpenSPL as a system for creating pipelines for dataflow computing.

With a basic understanding one can navigate the Maxeler IDE, run example programs, write a “Hello World!” application, simulate its execution and deploy applications on a DFE (Data Flow Engine).

3.1 Introduction

OpenSPL is an open specification for a Spatial Programming Language. The operations of a spatial program exists in space rather than as a sequence of operations over time. This means that all of the operations of program can happen at once and that the notion of what it means to execute a program is more about getting the data in and out of the system as opposed to the sequence of events that take place in typical procedural languages.

OpenSPL applications tend to manifest themselves in the form of one or more pipelines that are deployed using an FPGA (Field Programmable Gate Array) that, when acting in this fashion, is referred to as a DFE.

3.2 An Accelerator Architecture

An application that uses a DFE to improve its performance does so by using a it as an application accelerator. In doing so the code executes sequentially and that which executes spatially is written using two different styles and languages.

The code that runs sequentially can be written in a language like C and runs on a CPU in the manner that any C programmer is accustomed. The code that runs spatially is written in a variation of Java called MaxJ and ultimately runs on an FPGA on a DFE.

The coordination of compiling everything can be performed by the MaxIDE (Eclipse) and consists of...
executing the **MaxCompiler** to compile the MaxJ code into a `.max` file suitable for configuring an FPGA and a standard C compiler to create an executable program for the CPU. See Figure 3.1.

![Figure 3.1: Compiling an OpenSPL application.][1, p. 20]

When the C application runs it can implicitly or explicitly configure and use the DFE with one or more `.max` files to process data streams.

### 3.2.1 CPU Code

CPU code can call functions that are generated by the MaxCompiler. Supported languages include C, Python, Matlab and R. This text will focus on the use of C code on the CPU.

The CPU code is, no different than any other application you might write. The CPU portion of an OpenSPL application requires adding as little as one `#include` and one function-call statement to exchange data streams with the DFE.

To the C code, the data stream exchanges with the DFE are regions of memory like an array or a buffer created by calling the `malloc(3)` library function.

The above-mentioned header file to include and the function(s) to call to use the DFE are generated by the MaxCompiler when it builds the MaxJ files. The generated function(s) use the SLiC (**Simple Live CPU**) library interface that is part of the OpenSPL system. The SLiC interface provides the low-level services needed for the CPU to configure and exchange data with one or more DFEs.
3.2. AN ACCELERATOR ARCHITECTURE

3.2.2 MaxJ Code

The MaxJ part of an application is comprised of two components:

- One or more Kernels (pipelines) that are responsible for processing data streams.
- A single Manager that tends to the movement of data between the host system memory, Kernels, State Machines and memories on the DFE.

The MaxJ code is written in an extended version of Java which adds operator overloading to the base Java language. MaxJ source files have a .maxj file extension to differentiate them from pure Java.[1, p. 20] The operator overloading makes expressing mathematical operations used in Kernel pipelines easier to write.

There is a subtlety hiding in the box labeled Hardware Build or Simulation in Figure 3.1. As part of the build process, your MaxJ/Java application is actually executed. The output of the Java application is used to create the netlist that is ultimately deployed on the DFE.

The ultimate output of the MaxCompiler is a .max file that contains the executable DFE code and a .h file that contains generated function declarations and constants required to compile the CPU code. Thus the MaxJ code is compiled before the C code because the MaxJ code is the origin of that which defines the interface to the DFE code.¹

The life-cycle of the DFE code is similar to that of any executable program. . . as long as the responsibility of the operating system are taken into account: [1, p. 112]

1. Load - A .max file is loaded onto a DFE by the CPU code. The DFE card is now exclusively owned by the calling CPU process. Loading the .max file takes in the order of 100ms to 1s.
2. Run - The CPU code calls SLiC functions to execute actions on the DFE. A loaded .max file should be utilized for long enough to justify having waited up to a second to load it.
3. Unload - The DFE is released by the CPU process that returns it to the pool of DFEs managed by MaxelerOS for use by other applications.
4. Free - The .max file is deallocated.

The Basic Static ² SLiC interface implicitly loads the .max file onto the DFE when the first SLiC function is called, and then unloads the DFE and frees the .max file when the CPU code terminates.

This means that your application will stall if/when the DFE card(s) are in use by another application until that application terminates (or otherwise explicitly releases the DFE).

Note that a single application may serially reuse one DFE card by handling the loading and unloading of multiple .max files by using the SLiC API.

¹This creates a chicken-and-egg problem when it comes to writing the CPU application because the order of the arguments in the DFE-generated functions is not known until it has been compiled. To address that problem stub-in a call to the function and leave out its parameters, compile the application, it will fail on the call with incorrect arguments, look at the generated header-file (or use the IDE ‘insight’ to see what they are), add them and recompile. Empirical evidence suggests that the ordering is reliably reproducible and sorted alphabetically by type.

²One of three SLiC interfaces discussed in ??
Once the code for Kernels and the Manager are combined they form a complete dataflow program. The execution of this program results in either the generation of a dataflow engine configuration file (.max file), or the execution of a DFE simulation. In either case, the MaxCompiler always generates an include file to go with a .max file. [1, p. 31]

### 3.2.3 Kernel

An OpenSPL application will contain one or more Kernels. A Kernel that implements the Kernel graph shown in Figure 2.6 would contain logic like that shown in Listing 3.1.

Listing 3.1: KernelBody.maxj

The body of a simple kernel.

```jmax
DFEVar xs = io.input("x", dfeFloat(8, 24)); // A float stream called x
DFEVar zs = io.input("z", dfeFloat(8, 24)); // A float stream called z
DFEVar sum = xs*xs + zs*zs; // sum = xs^2 + zs^2
io.output("y", sum, dfeFloat(8, 24)); // A float stream called y
```

As can be seen the Kernel defines the name and type of data for each stream that it will process along with the operations it will perform on the data stream. In this case it will sum the squares of the elements in the \( x \) and \( z \) input streams and write the result to an output stream named \( y \).

The data types of each of the three streams is identical and set to `dfeFloat(8, 24)`. This is the OpenSPL way of defining what would appear in a C program as a `float`.

### 3.2.4 Manager

An OpenSPL application will contain one and only one Manager. The Manager coordinates the data flow between the CPU, Kernels, the DFE's memory and other devices depending on the particular type of DFE card(s) in the system. [1, p. 20] Each of these dataflows are are called a stream.

The simplest of all Managers is one that connects all of I/O defined in a single Kernel to the CPU and is shown in Listing 3.2.[11, p. 41]

Listing 3.2: SimpleManager.maxj

The simplest of Managers.

```java
public static void main(String[] args) {
    EngineParameters params = new EngineParameters(args);
    Manager manager = new Manager(params);
    Kernel kernel = new SimpleKernel(manager.makeKernelParameters());
    manager.setKernel(kernel);
    manager.setIO(IOType.ALL_CPU);
    manager.createSLiCinterface();
    manager.build();
}
```

This Manager makes boiler-plate calls to initialize the OpenSPL environment in lines 3 and 4.
The Kernel is created in line 5 and the default parameters are passed to the Kernel object’s constructor. The kernel is then linked to the manager in line 6.

All of the scalar and stream I/O variables are routed to the CPU application in line 7. This means that they will appear in the generated C-callable function in the generated .h file and will be named based on how the Kernel named them in the \texttt{io.input()} and \texttt{io.output()} calls such as those in Listing 3.1 on lines 1, 2 and 6.
4

Maxeler IDE

This chapter will present the Maxeler IDE to orient the reader before writing a first program.

4.1 Accessing the OpenSPL Environment

There are two ways to access the Maxeler IDE at NIU. Note that some documentation resources mention the availability of a web-based IDE. This is not available at NIU.

4.1.1 Installation of VM

Download and install the Maxeler VM from the University program web site and execute it using a lab PC or your own. In this configuration you will be limited to only development and testing applications using a simulation environment.

As you will see the simulated environment is where you will do the majority of your work. You will want to use this.

See Appendix B for details on installing and using VMware on Linux, Mac and Windows systems.

4.1.2 hermes.niu.edu

Using the software on hermes.niu.edu will allow for deployment of applications on real DFE hardware for final release testing and timing analysis.

Accessing hermes.niu.edu requires an ssh client and X windows server.

See Appendix A for details on using NX to improve performance of X windows.
4.2 Maxeler First Time Use

To start the Maxeler IDE:

```
[winans@hermes ~]$ maxide &
```

Alternatively, the MaxIDE icon on the desktop (on the VM) may be clicked.

![VM Desktop Icons](image)

Figure 4.1: VM Desktop Icons.

Note that the Maxeler IDE is based on Eclipse. See [http://www.eclipse.org/](http://www.eclipse.org/) for general information about the Eclipse IDE.

4.2.1 Set up Your Workspace

When started the first time the IDE will present a Workspace Launcher window (see Figure 4.2) asking you to where to put all of your files. Accepting the default of `~/workspace` should be suitable.

![Create a new workspace](image)

Figure 4.2: Create a new workspace

4.2.2 Documentation

When the IDE is started and there are no projects to display (as is the case when running it for the first time) a Welcome window is displayed (Figure 4.3).
4.2. MAXELER FIRST TIME USE

4.2.2.1 Documentation Available in the IDE

Note: When running the IDE on hermes.niu.edu you may not be able to view any of the help documents. The Welcome window presents a number of tutorials on how to use OpenSPL. These documents are very useful. It is recommended that they be skimmed early on in order to familiarize yourself with what is there so that help can be located down the road when it is needed.

![Figure 4.3: The MaxIDE Welcome Window](image)

4.2.3 Importing an Example Project

Also appearing on the Welcome Window is a link to a set of example projects that are discussed in the tutorial documents.

Select the Auto-import MaxCompiler tutorial projects link in the Welcome window (Figure 4.3).

Select MaxCompiler Dataflow Programming Tutorial from the menu box and check the examples box and then click finish in the Import MaxCompiler Projects window (Figure 4.4).
CHAPTER 4. MAXELER IDE

Figure 4.4: Import MaxCompiler Projects

This will import every project discussed in the Open MaxCompiler Dataflow Programming Tutorial.

Once completed the IDE will replace the Welcome window with the Project Explorer panel and list all of the imported projects.

Open a project by selecting tutorial-chap03-example1-movingaveragesimple from the Select Project panel (Figure 4.5) in the IDE menu bar.

Figure 4.5: Select Project

Once a project is open, navigate around to see what is in there.
4.2. MAXELER FIRST TIME USE

Click on the **Select Run/simulation** box over the project explorer and chose **simulation**.

Then click the **play** button icon to run it.

It will build and run your application.

If you receive a pop-up/warning about the simulator being started outside the IDE, select the **force reset** option.
Figure 4.7: Build and Run Simulation console window messages

Output from the run shows up in the terminal window below the build messages.

Figure 4.8: Simulation output messages

Once things have run, you can open the Run Rules > Simulation > Final Kernel Graph to see a diagram of the dataflow for the kernel and/or manager as seen in Figure 4.9.
4.3 MaxIDE Problems

If your connection to the server fails and/or the MaxIDE crashes, you may find that your workspace has become corrupted. If starting the MaxIDE results in displaying a broken workspace, terminate it immediately and start it again. Additionally, will want to backup copies of your source code files early and often.

See Appendix E for instructions on how to use SVN to backup and hand in your work.
5

Your First OpenSPL Program

5.1 Introduction

This chapter presents the details of creating a new project that will implement an application that calculates \( s_i = x_i^2 + 30a \).

The goals are to learn how to create new projects using the IDE, and alter the CPU, Manager and Kernel code to add/remove items from the SLiC interface.

5.1.1 Create a New Project

To create a new application from scratch we will use the IDE to create an application from a template based on the type of application we wish to develop and then change it to suit our needs.

In this example we will create an application with a Standard Manager using CPU Streams that will be compiled for the Icsa DFE hardware we will be using.

The stub application template created by the IDE will implement \( s_i = x_i + y_i + a \). The template code will be altered in order to implement \( s_i = x_i^2 + 30a \).
5.1.1.1 Create New MaxCompiler Project

Figure 5.1: Creating a new MaxCompiler project.

Begin by clicking the **New** icon in the upper-left of the IDE and select **MaxCompiler Project** from the menu as shown in Figure 5.1.

This will open a window that will prompt you for the details needed to create your project.
5.1.1.2 Name the Project

In the window that opens enter a suitable name for the new project, set the Manager Templates to CPU Stream (Vector Addition) and click Next (Figure 5.2).

The Project name field is used as the name that will appear in the Project Explorer tab on the left of the IDE.

Figure 5.2: Name the new project.

Fix Me:
Should we mention something about what the Vector Addition option means?
5.1.1.3 Set the DFE Hardware Type

Set the **DFE Model** to the type of hardware you are targeting with your application (hermes.niu.edu has a Icsa MAX4AB24B), chose a **Standard Manager**, provide a **Stem Name** for your manager and kernel and then click **Next** (Figure 5.3).

Figure 5.3: Set DFE hardware type.
5.1.1.4 Select the SLiC Interface Type

Figure 5.4: Select the SLiC interface type.

Provide a suitable name for the file that will contain your C source code (the default is suitable), set the SLiC Interface type to **Basic Static** and click **Finish** (Figure 5.4).
5.1.1.5 Inspect the Project Template Stub Files

Figure 5.5: Open the template stub files.

Have a look at the template files by opening the project in the Project Explorer tab and navigate to your CPU Code and Engine Code Manager and Kernel files. Double-click on TestStreamCpu-Code.c, TestStreamKernel.maxj, and TestStreamManager.maxj files to see them in an editor tab (Figure 5.5).

5.1.1.6 CPU Code Template

Listing 5.1: workspace/FirstProject/CPUCode/TestStreamCpuCode.c

Generated CPUCode stub.

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "Maxfiles.h"
#include "MaxSLiCInterface.h"

int main(void)
{
    const int size = 384;
    int sizeBytes = size * sizeof(int32_t);
    int32_t *x = malloc(sizeBytes);
    int32_t *y = malloc(sizeBytes);
    int32_t *s = malloc(sizeBytes);

    // TODO Generate input data
    for(int i = 0; i < size; ++i) {
```

Note: At the moment we are not interested in the TestStreamEngineParameters file.
5.1.1.7 Kernel Engine Code Template

Listing 5.2: workspace/FirstProject/EngineCode/src/teststream/TestStreamKernel.maxj

Kernel Engine Code Template.

```java
package teststream;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class TestStreamKernel extends Kernel {

    private static final DFEType type = dfeInt(32);

    protected TestStreamKernel(KernelParameters parameters) {
        super(parameters);

        DFEVar x = io.input("x", type);
        DFEVar y = io.input("y", type);
        DFEVar a = io.scalarInput("a", type);

        // TODO replace with your computation
        DFEVar sum = x + y + a;

        io.output("s", sum, type);
    }

}
```

5.1.1.8 Manager Code Template

Listing 5.3: workspace/FirstProject/EngineCode/src/teststream/TestStreamManager.maxj

Manager Code Template.
package teststream;

import static com.maxeler.maxcompiler.v2.managers.standard.Manager.link;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;
import com.maxeler.maxcompiler.v2.managers.standard.IOLink.IODestination;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;

public class TestStreamManager {

    private static final String s_kernelName = "TestStreamKernel";

    public static void main(String[] args) {
        TestStreamEngineParameters params = new TestStreamEngineParameters(args);
        Manager manager = new Manager(params);
        Kernel kernel = new TestStreamKernel(manager.makeKernelParameters(s_kernelName));
        manager.setKernel(kernel);
        manager.setIO(
            link("x", IODestination.CPU),
            link("y", IODestination.CPU),
            link("s", IODestination.CPU));
        manager.createSLiCinterface(interfaceDefault());
        configBuild(manager, params);
        manager.build();
    }

    private static EngineInterface interfaceDefault() {
        EngineInterface engine_interface = new EngineInterface();
        CPUTypes type = CPUTypes.INT32;
        int size = type.sizeInBytes();
        InterfaceParam a = engine_interface.addParam("A", CPUTypes.INT);
        InterfaceParam N = engine_interface.addParam("N", CPUTypes.INT);
        engine_interface.setScalar(s_kernelName, "a", a);
        engine_interface.setTicks(s_kernelName, N);
        engine_interface.setStream("x", type, N * size);
        engine_interface.setStream("y", type, N * size);
        engine_interface.setStream("s", type, N * size);
        return engine_interface;
    }

    private static void configBuild(Manager manager, TestStreamEngineParameters params) {
        manager.setEnableStreamStatusBlocks(false);
        BuildConfig buildConfig = manager.getBuildConfig();
    }
}
54  buildConfig.setMPPRCostTableSearchRange(params.getMPPRStartCT(), params
55       .getMPPREndCT());
56  buildConfig.setMPPRParallelism(params.getMPPRThreads());
57  buildConfig.setMPPRRetryNearMissesThreshold(params.
58       getMPPRRetryThreshold());
5.1.1.9 Build & Simulate Your Project

To test your application you can quickly build it for simulation and run it without using the DFE. In simulation, your code will compile quickly and run slowly. But only a simulation can be debugged using watches in the IDE. (Compiling for the DFE takes at least 20 minutes and can not be easily debugged while running!)

Building a project for simulation can be done by right-clicking on its name in the Project Explorer tab.

Right-click on **FirstProject**. Then navigate the menu to **Run As** and click on **Simulation**. (Figure 5.6).

![Figure 5.6: Build & Run your project in simulation.](image)
5.1.1.10 Build and Run Messages

While a project is building or running, messages will appear in the Console tab in the IDE (Figure 5.7).

Recall that the C application prints “Running on DFE.” and “Done.” from lines 22 and 31, respectively, in Listing 5.1. We see those lines appearing timestamped at “Thu 19:17” in the Console tab thus verifying that the application has executed.

Listing 5.4: FirstProject.out
Output from IDE-generated skeleton project code.

Thu 19:17: ##########################################
Thu 19:17: Running
Thu 19:17: Executing command:
Thu 19:17: '/home/winans/max/workspace/FirstProject/RunRules/Simulation/binaries/TestStream'
Thu 19:17: Command output:
Thu 19:17: Running on DFE.
Thu 19:17: Done.
Thu 19:17: Process terminated with exit code 0.
5.1.11 Original Kernel Graph

The IDE generates a graphic version of the pipeline created by the kernel. You view it by clicking on **Original Kernel Graph** in the **Project Explorer** tab.

The original kernel graph represents the kernel as described by the java code.

Figure 5.8: The original kernel graph.
5.2. CONVERT TEMPLATE CODE TO DESIRED APPLICATION

5.1.12 Final Kernel Graph

The IDE generates a graphic version of the pipeline created by the kernel. You view it by clicking on Original Kernel Graph in the Project Explorer tab.

The final kernel graph shows how the DFE will actually process the dataflow. It shows the optimizations that are applied and indicates when and how buffering is used for temporal alignment.

In this trivial kernel, we can see that a three-input adder has been created to perform the kernel operation rather than a cascade of two-input adders. We do not see any temporal alignment because the kernel is trivial enough not to require any.

5.2 Convert Template Code to Desired Application

The skeleton application implements this:

\[ s_i = x_i + y_i + a \]  \hspace{1cm} (5.2.1)

But we want an application that does this:

\[ s_i = x_i^2 + 30a \]  \hspace{1cm} (5.2.2)

... and it would also be nice to see the input and output data streams so that we can hand-verify our code.
To accomplish these changes, we will add some printing logic, remove the y input stream from the SLiC interface, change the computation performed by the kernel and change the verification logic to match the new kernel computation.

### 5.2.1 Add Display Logic

To see what is going on we add printing logic to dump the input and output data streams. We need not use large data sets to test this application, so we will also reduce the number of elements in the I/O streams to 96 to minimize the noise-level.

Note the addition of `printInt32Vector()` on line 8, the calls to it on lines 34–41 in Listing 5.5.

Listing 5.5: `workspace/FirstProject2/CPUCode/TestStreamCpuCode.c`

Add print logic to the CPU code.

```c
c// #include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "Maxfiles.h"
#include "MaxSLiCInterface.h"

static void printInt32Vector(int count, int32_t *v)
{
    int i;
    for (i=0; i<count; ++i)
    {
        printf("%d ", v[i]);
    }
    printf("\n");
}

int main(void)
{
    const int size = 96;
    int sizeBytes = size * sizeof(int32_t);
    int32_t *x = malloc(sizeBytes);
    int32_t *y = malloc(sizeBytes);
    int32_t *s = malloc(sizeBytes);

    // TODO Generate input data
    for (int i = 0; i < size; ++i) {
        x[i] = random() % 100;
        y[i] = random() % 100;
    }

    printf("Running on DFE.\n");
    int scalar = 3;
    printf("a = %d\n", scalar);
    printf("Input x\n");
    printInt32Vector(size, x);
    printf("Input y\n");
    printInt32Vector(size, y);
    TestStream(scalar, size, x, y, s);
    printf("Output x\n");
    printInt32Vector(size, s);
}
```
5.2. CONverting TEMPLATE CODE TO DESIRED APPLICATION

```c
// TODO Use result data
for (int i = 0; i < size; ++i)
    if (s[i] != x[i] + y[i] + scalar)
        return 1;

printf("Done.\n");
return 0;
```

Running the new version of the application renders the output shown in Listing 5.6.

Listing 5.6: FirstProject2.out
Output with display logic to dump the streams.

```text
Tue 14:55: ##########################################
Tue 14:55: Running
Tue 14:55: Executing command:
Tue 14:55: '/home/winans/max/workspace/FirstProject2/RunRules/Simulation/binaries/TestStream'
Tue 14:55: Command output:
Tue 14:55: Running on DFE.
Tue 14:55: a = 3
Tue 14:55: Input x
Tue 14:55: 83 77 93 86 49 62 90 63 40 72 11 67 82 62 67 29 22 69 93 11 29 21 84
98 15 13 91 56 62 96 5 84 36 46 13 24 82 14 34 43 87 76 88 3 54 32 76 39 26
94 95 34 67 97 17 52 1 86 65 44 40 31 97 81 9 67 97 86 6 19 28 32 3 70 8 40
96 18 46 21 79 64 41 93 34 24 87 43 27 59 32 37 75 74 58 29
Tue 14:55: Input y
Tue 14:55: 86 15 35 92 21 27 59 26 26 36 68 29 30 23 35 2 58 67 56 42 73 19 37
24 70 26 80 73 70 81 25 27 5 29 57 95 45 67 64 50 8 78 84 51 99 60 68 12 86
39 70 78 1 2 92 56 80 41 89 19 29 17 71 75 27 56 53 65 83 24 71 29 19 68 15
49 23 45 51 55 88 28 50 0 64 14 56 91 65 36 51 28 7 21 95 37
Tue 14:55: Output x
Tue 14:55: 172 95 131 181 73 92 152 92 69 111 82 99 115 88 105 34 83 139 152 56
105 43 124 125 88 42 174 132 135 180 33 114 44 78 73 122 130 84 101 96 98
157 175 57 156 95 147 54 115 136 168 115 71 102 112 111 84 130 157 66 72 51
171 159 39 126 153 154 92 46 102 64 25 141 26 92 122 66 100 79 170 95 94 96
101 41 146 137 95 98 86 68 85 98 156 69
Tue 14:55: Done.
Tue 14:55: Process terminated with exit code 0.
```

5.2.2 Replace the Manager Template With a Simple Manager

Open the TestStreamManager.maxj file and delete line 22 though the end of the file with lines xx-yy shown in Listing 5.7 (which can be copied from the MovingAverageSimpleManager.maxj tutorial source) shown (thus replacing 38 lines of code with 5).

Listing 5.7: TestStreamManager.maxj
Simplified version of TestStreamManager.

```c
package teststream;
import static com.maxeler.maxcompiler.v2.managers.standard.Manager.link;
```
import com.maxeler.maxcompiler.v2.kernelfinker.Kernel;
import com.maxeler.maxcompiler.v2.managers.BuildConfig;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;
import com.maxeler.maxcompiler.v2.managers.standard.IOLink.IODestination;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;

public class TestStreamManager {
    private static final String s_kernelName = "TestStreamKernel";

    public static void main(String[] args) {
        TestStreamEngineParameters params = new TestStreamEngineParameters(args);
        Manager manager = new Manager(params);
        Kernel kernel = new TestStreamKernel(manager.makeKernelParameters(s_kernelName));
        manager.setKernel(kernel);
        manager.setIO(IOType.ALL_CPU); // Connect all kernel ports to the CPU
        manager.createSLiCinterface();
        manager.build();
    }
}

5.2.3 Change the Computation in the Kernel

Remove the extra input stream by deleting the DFEVar y definition on line 16 in TestStreamKernel.maxj and change the assignment on line 20 as shown in Listing 5.8.

Listing 5.8: workspace/FirstProject3/EngineCode/src/teststream/TestStreamKernel.maxj
Final version of TestStreamKernel.

protected TestStreamKernel(KernelParameters parameters) {
    super(parameters);
    DFEVar x = io.input("x", type);
    //DFEVar y = io.input("y", type);  // REMOVE THIS LINE
    DFEVar a = io.scalarInput("a", type);

    // TODO replace with your computation
    DFEVar sum = x*x + 30*a;  // CHANGE THIS LINE
    io.output("s", sum, type);
}

5.2.4 Update the CPU Code

Remove the y stream and change the verification logic in the CPU code to match the new kernel. See changes on lines 23, 29, 37-38 and 45 in Listing 5.5
5.2. CONVERT TEMPLATE CODE TO DESIRED APPLICATION

Listing 5.9: workspace/FirstProject3/CPUCode/TestStreamCpuCode.c
Final version of CPU code.

```c
int main ( void )
{
    const int size = 96;
    int sizeBytes = size * sizeof(int32_t);
    int32_t *x = malloc(sizeBytes);
    //int32_t *y = malloc(sizeBytes);  // REMOVE THIS
    int32_t *s = malloc(sizeBytes);
    // TODO Generate input data
    for ( int i = 0; i < size ; ++i) {
        x[i] = random() % 100;
        //y[i] = random() % 100;  // REMOVE THIS
    }

    printf("Running on DFE.\n");
    int scalar = 3;
    printf("a = %d\n", scalar);
    printf("Input x\n");
    printInt32Vector(size, x);
    //printf("Input y\n");  // REMOVE THIS
    //printf("Input int32Vector(size, y);  // REMOVE THIS
    TestStream(size, scalar, x, s);  // CHANGE THIS
    printf("Output x\n");
    printInt32Vector(size, s);

    // TODO Use result data
    for ( int i = 0; i < size; ++i)
        if ( s[i] != x[i]*x[i] + scalar*30)  // CHANGE THIS
            return 1;

    printf("Done.\n");
    return 0;
}
```

5.2.5 Final Program Output

Run your program one more time and see the final output in Listing 5.10

Listing 5.10: FirstProject3.out
Output of final application.

```
Tue 15:22: ##########################################
Tue 15:22: Running
Tue 15:22: ##########################################
Tue 15:22: Executing command :
Tue 15:22: '/home/winans/max/workspace/FirstProject3/RunRules/Simulation/binaries/TestStream'
Tue 15:22: Command output:
Tue 15:22: Running on DFE.
Tue 15:22: a = 3
Tue 15:22: Input x
Tue 15:22: 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67
```

"/KIU/courses/532/2015-fa/book/openspl/./FirstProject3.out 2015-10-20 15:01:52 -0500 v1.0-79-g838c34b
Figure 5.10: Final kernel graph of $s = x^2 + 30a$. 
6

The Kernel

The Kernels present in an OpenSPL application are responsible for the “data processing.”

6.1 Introduction

Maximum performance in a Maxeler solution is achieved through a combination of deep-pipelining and exploiting both inter- and intra-Kernel parallelism. The high I/O-bandwidth required by such parallelism is supported by flexible high-performance memory controllers and a highly parallel memory system.[1, p. 20]

The computation-to-data ratio, which describes how many mathematical operations are performed per item of data moved, is a key metric for estimating the performance of the final dataflow implementation. Code that requires large amounts of data to be moved and then performs only a few arithmetic operations poses higher balancing challenges than code with significant localized arithmetic activity.[1, p. 22]

6.2 Widening the Pipeline

A simple and straightforward method of improving the performance of an OpenSPL application is to do as much as can be done with the data that is present in one kernel.

Multiple streams (related or not) can flow through a kernel pipeline at the same time.

6.2.1 Overloading a Kernel

In some situations the same data streams are needed for multiple different calculations. For example, an application might require that the following calculations:
Putting all four of these functions into a single kernel will allow the $x$, $y$ and $z$ streams to be transferred into the DFE once and only once as opposed to the five times that it would otherwise take if each function were implemented in a separate kernel and invoked serially. See the graph for this kernel in Figure 6.1.

Figure 6.1: Multiple outputs from the same data stream.

This kernel will generate all five output streams in the same amount of time that would be required to execute to execute only one of the above equations as seen in Figure 6.2.

### 6.2.2 An N-fold kernel

Another variation on the theme of doing more data processing at the same time is to observe that some times it is not that different operations are performed on same data but, rather, the same operations are performed on different data.

For example, consider the following operations:

\[
\begin{align*}
a &= o + p + q + r \\
b &= s + t + u + v
\end{align*}
\]
Figure 6.2: An inefficient use of data (in contrast to Figure 6.1).

...that would produce the kernel graph in Figure 6.3. To implement such an application, one kernel with eight input streams can be created or a fancy manager could be used to create and connect two copies of a four-input stream kernel to eight input streams.
6.3 Temporal Alignment

So far we have only considered kernels that can be implemented without any internal delays. Some functions require the same data to be applied to different parts of the same pipeline... at different times.

6.3.1 $y = x^2 + z^2 + z$

Solving $y = x^2 + z^2$ was straight forward due to the clean symmetry of its kernel graph (see Figure 2.6.) If we add $z$ to the sum of the squares, something interesting happens.

Note the addition of the red edge in Figure 6.4 indicating the need to add $z$ to $x^2$ and $z^2$.

The problem with this graph is that when one considers the pipeline expressed in Figure 2.8, the $z$ values have to be made available to the adder at a different time than when they have to be delivered to the multiplier.

In order to address this issue, the $z$ can be delayed using a FIFO buffer as shown in Figure 6.5.

Notice that the FIFO is drawn as a green table with a number in the center that represents the number of elements (and therefore units of time) that the FIFO contains. When writing an element into a FIFO with length 1, it will come out one unit of time later.

A timing diagram for Figure 6.5 is shown in Figure 6.6. Note that this diagram is identical to Figure 2.8 except for those rows highlighted on the left in yellow.

The pipeline latency has not changed because we can implement it using an adder with three inputs. The extent to which we can add additional inputs to any type of operational unit depends on the type of FPGA and versions of the compilers we use.
6.3. TEMPORAL ALIGNMENT

Figure 6.4: A (Broken) Kernel graph for \( y = x^2 + z^2 + z \)

Figure 6.5: A Kernel graph for \( y = x^2 + z^2 + z \)
Figure 6.6: Six pipelined iterations of $y[i] = x[i]^2 + z[i]^2 + z$ for $x = \{1, 2, 3, 4, 5, 6\}$ and $z = \{4, 5, 6, 7, 8, 9\}$
6.3.2 \( y = x^2 + z^2 + z - x \)

The intention of this example is to illustrate a situation where a FIFO is required to provide a delay of more than one element as shown in Figure 6.7 with the corresponding timing diagram in Figure 6.8. The items added or changed from Figure 6.6 are highlighted on the left in yellow.

Figure 6.7: A Kernel graph for \( y = x^2 + z^2 + z - x \)

Note that along with adding a 2-element FIFO, the increased complexity of the mathematical function in this example has also changed the pipeline latency from 3 to 4.\(^1\)

The end-result still consumes one set of inputs per-tick and provides one output per-tick (once the pipeline has been filled). But we can start to see the impact of what it means to trade space for time as the six pipeline-stages that perform actual computation (shown in blue) are starting to be rivaled by the number of stages that just hold or move data (shown in yellow and green.)

This situation illustrates a limitation of dataflow computing.

\(^1\)Note that the problem presented here is to illustrate the use of FIFOs. Because we can create a 3-input adder, it would have been more efficient (lower latency) if the solution negated \( x \) and then fed it into a 3-input adder rather than through a FIFO and into a subtractor.
6.3.3 Reality Check (There’s a Pipeline in my Pipeline!)

There is more to temporal alignment that we have let on so far. Looking at an optimized kernel graph for $x^2 + x$ we see that a FIFO is created with a delay of two (rather than one!)

When calculating $x^2 + x$, the time it takes for the value of $x^2$ to appear at the output of the multiplier depends on the type of FPGA and data representation used. In this example, $x$ is a 32-bit integer on an Altera Stratix V FPGA (that is on a Maxeler Icsa board).
Note that the number ‘2’ in the FIFO in Figure 6.9. This implies that the multiplier operation is itself implemented in a two-stage pipeline that is abstracted out of the kernel graph but it shown in timing diagram in Figure 6.10.

Note that the actual times for any FIFOs required for temporal alignments only appear in optimized kernel graphs as their presence and size are dependent on the optimizations performed by the compiler.

Figure 6.10: Timing diagram for $s = x^2 + x$
Installing and Using NX

NX is a computer program that provides access to remote X Window Systems.

A.1 Download and Install the NX Client

Navigate to http://hermes.niu.edu/nxclient/ and click on your platform.

A.1.1 Linux

![Figure A.1: Set session and host names.]

Fix Me:
This screen-shot sequence is wrong and it never worked right for me.
A.2 Setting up NX

(From Kyle Gilgan Internship report 2014)

1. Run the installer
2. Enter hermes.niu.edu into the host field
3. Set the port to 22
4. Select your type of Internet connection by sliding the bar to the option that corresponds to your setup
5. Leave Unix in the first box and in the second box change KDE to custom
6. Click on the settings box and change "Run the console" to "Run the following command" and enter "MaxIDE" into the box
7. Leave the options as a "floating window"
8. Create a desktop shortcut if you would like one
9. Enter your username and password
10. MaxIDE will launch automatically
11. Once MaxIDE has launched, click on the window tab at the top of the screen in the toolbar
12. Click on preferences in the drop down
13. Expand the General tab
14. Expand the Appearance Tab
15. Click on the Colors and Fonts tab
16. This is where you can change the size of the MaxIDE font to a more readable size
17. Highlight "Text Font" in the box displayed and click on the bod that says "Edit..."
18. Select the font size that is most comfortable and click ok, then apply the results and click once more
19. Once you are ready to run your first program, expand the project in the left hand column
20. Expand the run rules
21. Expand the simulation
22. Right click on the simulation tab
23. Click "Edit Run Rules"
24. Towards the bottom of the page there are four tabs labeled "Common", "Max Files", "CPUCode", and "Simulator"
25. Click on the tab that says "Simulator"
26. Under Advanced Options change the socket name to your z-ID followed by another z (example z1234567z)
Running MaxIDE on VMware

B.1 Installing VMware

You can download a free VMware Player application for your operating system.

B.1.1 Linux

https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/7_0

B.1.2 Windows

https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/7_0

B.1.3 Mac

The Mac version of the VMware Player is called VMware Fusion.

http://www.vmware.com/products/fusion/fusion-evaluation.html

B.2 Loading the .vmx File

The .vmx is an image of an entire virtual machine that has been configured to run The Maxeler IDE in simulation mode on Centos 6.3.

Run the VM, click on the .vmx file. A Linux system will run contained in a window. Once started you can launch the Maxeler IDE as discussed in chapter 4.
Running MaxIDE on VirtualBox

C.1 Installing VirtualBox

You can download the free VirtualBox application for your operating system.\(^1\)

Installations and configuration instructions for Windows, OS X, Linux and Solaris are all available from Oracle here:

https://www.virtualbox.org/wiki/Downloads

C.2 Loading the .vmx File

The .vmx is an image of an entire virtual machine that has been configured to run The Maxeler IDE in simulation mode on Centos 6.3.

Run the VM, click on the .vmx file. A Linux system will run contained in a window. Once started you can launch the Maxeler IDE as discussed in chapter 4.

\(^{\text{**Fix Me:**}}\)

\(\) Install and verify how to start VirtualBox to run the .vmx file.

\(^1\) At time of writing this text, the Maxeler VM version 2015 is known to run on VirtualBox version 5.0.6.
Java Resources

XXX Add a lecture on Java

D.1 Web Resources

D.1.1 MIT OpenCourseware

MIT offers the lecture notes for a number of courses for free on the web. MIT Course Number 6.092 Introduction to Programming in Java may be of interest to the new Java programmer that already has some programming experience:

This course is an introduction to software engineering, using the Java(TM) programming language. It covers concepts useful to 6.005. Students will learn the fundamentals of Java. The focus is on developing high quality, working software that solves real problems.

The course is designed for students with some programming experience, but if you have none and are motivated you will do fine. Students who have taken 6.005 should not take this course. Each class is composed of one hour of lecture and one hour of assisted lab work.

This course is offered during the Independent Activities Period (IAP), which is a special 4-week term at MIT that runs from the first week of January until the end of the month.

http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-lecture-notes/

D.1.2 Introduction to Programming Using Java

The Seventh Edition of Introduction to Programming Using Java is a free on-line textbook on introductory programming, which uses Java as the language of instruction. This book is directed mainly towards
beginning programmers, although it might also be useful for experienced programmers who want to learn something about Java.

http://math.hws.edu/javanotes/

D.1.3 A Primer on Java

A few google searches turned up *A Primer on Java* that appears to be an easy read and may be of interest to anyone that hasn’t written programs in a while.

This book describes itself as:

> A gentle introduction to the basics of Java. No prior programming experience is necessary to read this text. My purpose was to write an easy but non-verbose introduction to the language.

> The book however is not intended as a reference work or for a full time Java programmer since it does not have an exhaustive topic coverage.

https://leanpub.com/aprimeronjava

D.2 References From Multiscale Dataflow Programming[1]

The following is a list of references from the Maxeler documentation. XXXXX

For further information on the Java language we recommend the following resources:

http://docs.oracle.com/javase/tutorial/java/index.html

An overview of Java and an introduction to its major syntactical features.

http://docs.oracle.com/javase/tutorial/collections/index.html

An overview of the Java “Collections” API which is used often in MaxCompiler interfaces.

http://docs.oracle.com/javase/6/docs/api/

API documentation for the standard Java libraries.


Introduction to using variable-argument methods in Java which are also common in MaxCompiler interfaces.
Managing Projects With Subversion

Subversion is a version control system is a database that records and tracks changes to files over time. When configured on a server machine one can use it to share files between multiple users and machines.

There are many resources on the Internet that discuss how to use Subversion. Here we will discuss accessing and using an existing repository with the MaxIDE.

### E.1 Introduction

Given the vintage of the Eclipse version used in the Maxeler IDE the only version control system supported out-of-the-box is “SVN”.²

We can use Subversion to store files for backup and to copy them between a virtual machine on a personal laptop for development, simulation and debugging, transferring them to hermes.niu.edu for building and running on the DFE card and for handing in homework.

In the simplest of terms Subversion manages database called a repository or repo that provides mechanisms for inserting and retrieving files. When one or more files are inserted into the repo the action is called a commit or a check-in. When a file is retrieved from the repo the action is called a checkout or an update.

The Subversion database contains a copy of every version of every file that was ever committed over the course of the file’s evolution. This means that the database has the ability to retrieve was committed to the repo one hour ago, four days ago, 14 months ago and so on.

---

¹For links to Subversion documentation see: [https://en.wikipedia.org/wiki/Apache_Subversion](https://en.wikipedia.org/wiki/Apache_Subversion)
²Technically, SVN is the name of a command that is part of the Subversion package. Subversion can be used as the back-end of a WebDAV server or in a stand-alone mode. Subversion provides the services of a version control system in a manner well suited to the WebDAV protocol. It is a common mistake to consider WebDAV, Subversion and SVN as synonymous with each other. Each provides a very different role.
Because the repo database has every version of every file, it can also be used to ask to show what has changed between two or more versions of the same file. This is very useful when you break some code and forgot what you did since the last working version!

Of course, you have to remember to commit your files every now and then or else the repo will not have copies of the evolving versions of your files to make this possible.

E.2 Creating a Subversion Repository

To use Subversion a repository must be present. You may create one named MyRepo in the current directory by running the following command:

```
$ svnadmin create MyRepo
```

Once created, the repository directory and any files within it should never be touched directly. They are only accessed using a client application like `svn` or an IDE such as Eclipse or MaxIDE.

For CSCI 532 a repo has already been created and assigned for your use on `hermes.niu.edu`. In order to reference your repo a URL is used that looks like this:

`svn+ssh://hermes.niu.edu/home/repos/z1234567`

Substitute your Z-number for the last portion of the path.

E.3 Checking a Project Into a Repository

An `svn` client capable of checking projects into and out of a Subversion repo is built into MaxIDE.

To check a project into a subversion repository:
E.3. CHECKING A PROJECT INTO A REPOSITORY

Figure E.1: Right-click on your project and open Team->ShareProject.

Figure E.2: Choose SVN and click Next.
APPENDIX E. MANAGING PROJECTS WITH SUBVERSION

Figure E.3: Select **Create New...** and click **Next**.

Figure E.4: Enter the URL for the repository and click **Next**.
E.3. CHECKING A PROJECT INTO A REPOSITORY

Figure E.5: Select Use Project Name... and click Next.

When checking into a shared repo the URL would have to include an additional directory for the student ID after the SvnRepo directory shown in Figure E.5.

Figure E.6: Enter a suitable commit comment and click Finish.

At this point you will have created a place for your project in the specified repository. You now have to proceed to commit the current version of your files.
APPENDIX E. MANAGING PROJECTS WITH SUBVERSION

Figure E.7: Right-click on your project and open Team->Commit.

Figure E.8: Enter a suitable comment and click OK.

At this point you should note that each of the files that are in your repository are displayed with a revision number indicating the when they were last committed to the repository. This number is simply incremented each time that anything is committed into the repository. As seen in Figure E.9 the version is 12.
At this point the project files are all in the repository. Any editing of the files will (as one would expect) make them out of date with the version in the repository. Eclipse will indicate that a file is out of date by placing a small brown indicator in the icon that represents the edited file(s) as well as in the parent directory icons up to the root project level. See Figure E.10.

After making changes to any files, repeat the commit process as described beginning in Figure E.7. Again, enter a suitable comment before clicking OK. The comments that are entered when committing files should accurately describe the changes that were made to the out of date files so that it is possible to identify and understand the various versions of the files as they are edited over time. Should a project stop working, an older version of the file(s) can be recovered from the repository. At times like this the easiest way to locate the desired version of a file is to plan ahead and properly annotate changes as they occur over time.
APPENDIX E. MANAGING PROJECTS WITH SUBVERSION

Figure E.11: Committing the project files changes the version number of `a1CpuCode.c` as seen in the Project Explorer.

After committing a new version of a file, the version number will change in the Project Explorer view. In the example shown in Figure E.11 committing changes to the file named `a1CpuCode.c` has changed the version from 12 to 14 at 6:31 PM on July 16th by ‘winans’.

E.4 Checking Files Out

Once files are committed/checked into a repository they can then be checked out elsewhere. This feature can be used to keep copies of a project’s files on multiple machines in sync and to hand in your homework.
IEEE-754 Floating Point Number Representation

This appendix discusses the 32-bit IEEE-754 Floating Point format.

- Note that the place values for integer binary numbers are:
  
  \[
  \ldots 128 \ 64 \ 32 \ 16 \ 8 \ 4 \ 2 \ 1 
  \]

- We can extend this to the right in binary similar to the way we do in decimal:
  
  \[
  \ldots 128 \ 64 \ 32 \ 16 \ 8 \ 4 \ 2 \ 1 \ . \ 1/2 \ 1/4 \ 1/8 \ 1/16 \ 1/32 \ 1/64 \ 1/128 \ \ldots
  \]
  
  Note that the ‘.’ in a binary number is a binary point, not a decimal point.

- Scientific notation as in \(27 \times 10^{-47}\) is used when either small fractions or large numbers when we are concerned with fewer digits than are necessary to represent the entire number.

- The format of a number in scientific notation is \(mantissa \times base^{exponent}\)

- In binary we have \(mantissa \times 2^{exponent}\)

- For simplicity sake, IEEE-754 format requires binary numbers to be normalized to \(1.mantissa \times 2^{exponent}\) where the significand is the part of the mantissa that is to the right of the binary–point.
  
  - The unnormalized binary value of \(-2.625\) is 10.101
  - The normalized value of \(-2.625\) is 1.0101 \(\times 2^1\)

- We need not store the ‘1.’ because all normalized floating point numbers will start that way. Thus we can save memory by simply remembering that that first bit is always there and that is supposed to be a 1.

\[
\begin{array}{ccccccccccccccccccccccc}
\end{array}
\]

- sign exponent significand

- The unnormalized binary value of \(-2.625\) is 10.101
- The normalized value of \(-2.625\) is 1.0101 \(\times 2^1\)

- We need not store the ‘1.’ because all normalized floating point numbers will start that way. Thus we can save memory by simply remembering that that first bit is always there and that is supposed to be a 1.
\begin{itemize}
  
  \item \(-((1 + \frac{1}{4} + \frac{1}{16}) \times 2^{128-127}) = -(\frac{5}{16} \times 2^1) = -(1.3125 \times 2^1) = -2.625\)

  \item \(-((1 + \frac{1}{4} + \frac{1}{16}) \times 2^{128-127}) = -((1 + \frac{1}{4} + \frac{1}{16}) \times 2^1) = -(2 + \frac{1}{2} + \frac{1}{8}) = -(2 + .5 + .125) = -2.625\)

  \item IEEE754 formats:

  \begin{tabular}{|c|c|c|}
  \hline
  & IEEE754 32–bit & IEEE754 64–bit \\
  \hline
  sign & 1 bit & 1 bit \\
  exponent & 8 bits (excess–127) & 11 bits (excess-1023) \\
  mantissa & 23 bits & 52 bits \\
  max exponent & 127 & 1023 \\
  min exponent & -126 & -1022 \\
  \hline
  \end{tabular}

  \item When the exponent is all ones, the mantissa is all zeros, and the sign is zero, the number represents positive infinity.

  \item When the exponent is all ones, the mantissa is all zeros, and the sign is one, the number represents negative infinity.

  \item Note that the binary representation of an IEEE754 number in memory can be compared for magnitude with another one using the same logic as for comparing sign–magnitude numbers because the magnitudes of the IEEE number’s magnitude grows upward and downward in the same fashion as sign–magnitude integers. This is why we use excess notation for the exponent… numbers with larger exponents look larger than numbers with smaller exponents even when incorrectly interpreted as sign–magnitude integers.

  \item Note that zero is a special case number. This is why the exponent of all–zeros is not used to represent the smallest possible exponent value. Zero is represented by an exponent of all–zeros and a mantissa of all–zeros. This allows for a positive and a negative zero if we observe that the sign can be either 1 or 0.

  \item On the numberline, numbers between zero and the smallest fraction in either direction are in the underflow areas.

  \item On the numberline, numbers greater than the mantissa of all–ones and the largest exponent allowed are in the overflow areas.

  \item Note that numbers have a higher resolution on the number–line when the exponent is smaller.

\end{itemize}
F.1 Floating Point Number Accuracy

Due to the finite number of bits used to store the value of a floating point number, it is not possible to represent every one of the infinite values on the real number line. The following C programs illustrate this point.

F.1.1 Powers Of Two

Just like the integer numbers, the powers of two that have bits to represent them can be represented perfectly... as can their sums:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

union floatbin
{
    unsigned int i;
    float f;
};

int main()
{
    union floatbin x, y;
    int i;

    x.f = 1.0;
    while (x.f > 1.0/1024.0)
    {
        y.f = -x.f;
        printf("%25.10 f = %08 x %25.10 f = %08 x
", x.f, x.i, y.f, y.i);
        x.f = x.f /2.0;
    }
}
```

1.0000000000 = 3f800000 -1.0000000000 = bf800000
0.5000000000 = 3f000000 0.5000000000 = bf000000
0.2500000000 = 3e800000 -0.2500000000 = be800000
0.1250000000 = 3e000000 -0.1250000000 = be000000
0.0625000000 = 3d800000 -0.0625000000 = bd800000
0.0312500000 = 3d000000 -0.0312500000 = bd000000
0.0156250000 = 3c800000 -0.0156250000 = bc800000
0.0078125000 = 3c000000 -0.0078125000 = bc000000
0.0039062500 = 3b800000 -0.0039062500 = bb800000
0.0019531250 = 3b000000 -0.0019531250 = bb000000
APPENDIX F. IEEE-754 FLOATING POINT NUMBER REPRESENTATION

F.1.2 Clean Decimal Numbers

When dealing with decimal values, you will find that they don’t map simply into binary floating point values (the same holds true for binary integer numbers).

Note how the decimal numbers are not accurately represented as they get larger. The decimal number 10 can be perfectly represented in IEEE format. The problem that arises after the 11th loop iteration is not because the prior number was not multiplied by 10. It is due to the fact that the prior number can not be represented accurately in IEEE format. Therefore its least significant bits were truncated in a best-effort attempt at rounding the value off. Once this happens, the value of \( x.f \) may not be what a programmer expects.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

union floatbin
{
    unsigned int   i;
    float          f;
};

int main()
{
    union floatbin x, y;
    int i;

    x.f = 10;
    while (x.f <= 10000000000000.0)
    {
        y.f = -x.f;
        printf("%25.10f = %08x %25.10f = %08x\n", x.f, x.i, y.f, y.i);
        x.f = x.f*10.0;
    }
}
```

```
10.0000000000 = 41200000  -10.0000000000 = c1200000
100.0000000000 = 42c80000  -100.0000000000 = c2c80000
1000.0000000000 = 447a0000  -1000.0000000000 = c47a0000
10000.0000000000 = 461c4000  -10000.0000000000 = c61c4000
100000.0000000000 = 47c35000  -100000.0000000000 = c7c35000
1000000.0000000000 = 49742400  -1000000.0000000000 = c9742400
10000000.0000000000 = 4b189680  -10000000.0000000000 = cb189680
100000000.0000000000 = 4cbebc20  -100000000.0000000000 = ccbebc20
1000000000.0000000000 = 4e6e6b28  -1000000000.0000000000 = ce6e6b28
10000000000.0000000000 = 501502f9  -10000000000.0000000000 = d01502f9
999999997952.0000000000 = 51ba43b7  -999999997952.0000000000 = d1ba43b7
999999995904.0000000000 = 5368d4a5  -999999995904.0000000000 = d368d4a5
9999999827968.0000000000 = 551184e7  -9999999827968.0000000000 = d51184e7
```
F.1.3 Accumulation of Error

This effect of rounding errors can be exaggerated if the number we multiply the \( x.f \) value by is itself something that can not be accurately represented in IEEE form.

If we multiply our \( x.f \) value by \( \frac{1}{10} \) each time, we can never be accurate and we start accumulating errors immediately.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

union floatbin
{
    unsigned int    i;
    float           f;
};

int main()
{
    union floatbin x, y;
    int    i;

    x.f = .1;
    while (x.f <= 2.0)
    {
        y.f = -x.f;
        printf("%25.10f = %08x  %25.10f = %08x\n", x.f, x.i, y.f, y.i);
        x.f += .1;
    }
}
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary Float Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1000000015</td>
<td>3dcccccd</td>
</tr>
<tr>
<td>0.2000000030</td>
<td>3e4cccccd</td>
</tr>
<tr>
<td>0.3000000119</td>
<td>3e99999a</td>
</tr>
<tr>
<td>0.4000000600</td>
<td>3ccccccd</td>
</tr>
<tr>
<td>0.5000000000</td>
<td>3f000000</td>
</tr>
<tr>
<td>0.6000000238</td>
<td>3f19999a</td>
</tr>
<tr>
<td>0.7000000477</td>
<td>3f333334</td>
</tr>
<tr>
<td>0.8000000715</td>
<td>3f4ccccce</td>
</tr>
<tr>
<td>0.9000000954</td>
<td>3f666668</td>
</tr>
<tr>
<td>1.0000001192</td>
<td>3f800001</td>
</tr>
<tr>
<td>1.1000001431</td>
<td>3f8ccccce</td>
</tr>
<tr>
<td>1.2000001669</td>
<td>3f99999b</td>
</tr>
<tr>
<td>1.3000001907</td>
<td>3fa66668</td>
</tr>
<tr>
<td>1.4000002146</td>
<td>3fb33335</td>
</tr>
<tr>
<td>1.5000002384</td>
<td>3fc00002</td>
</tr>
<tr>
<td>1.6000002623</td>
<td>3fcccccf</td>
</tr>
<tr>
<td>1.7000002861</td>
<td>3fd9999c</td>
</tr>
<tr>
<td>1.8000003099</td>
<td>3fe66669</td>
</tr>
<tr>
<td>1.9000003338</td>
<td>3ff33336</td>
</tr>
</tbody>
</table>
F.2 Reducing Accumulation of Errors

In order to use floating point numbers in a program without causing undesirable results, you can consider redesigning your algorithm so that an accumulation of errors is eliminated. This example is similar to the previous one, but this time we recalculate the desired value from known-accurate integer values. Thus we might see some rounding errors, but they can not accumulate.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

union floatbin
{
    unsigned int i;
    float f;
};

int main()
{
    union floatbin x, y;
    int i;

    i = 1;
    while (i <= 20)
    {
        x.f = i/10.0;
        y.f = -x.f;
        printf(" %25.10f = %08x %25.10f = %08x\n", x.f, x.i, y.f, y.i);
        i++;
    }
    return (0);
}
```

1.0000000015 = 3dcccccd -0.1000000015 = bdcccccd
0.2000000030 = 3e4ccccd -0.2000000030 = be4ccccd
0.3000000119 = 3e99999a -0.3000000119 = be99999a
0.4000000060 = 3ecccccd -0.4000000060 = becccccd
0.5000000000 = 3f000000 -0.5000000000 = bf000000
0.6000000238 = 3f19999a -0.6000000238 = bf19999a
0.6999999881 = 3f333333 -0.6999999881 = bf333333
0.8000000119 = 3f4ccccd -0.8000000119 = bf4ccccd
0.8999999523 = 3f666666 -0.8999999523 = bf666666
1.0000000000 = 3f800000 -1.0000000000 = bf800000
1.1000000238 = 3f8ccccd -1.1000000238 = bf8ccccd
1.2000000477 = 3f99999a -1.2000000477 = bf99999a
1.2999999523 = 3fa66666 -1.2999999523 = bfa66666
1.3999999762 = 3fb33333 -1.3999999762 = bfb33333
1.5000000000 = 3fc00000 -1.5000000000 = bfc00000
1.6000000238 = 3fc66666 -1.6000000238 = bfc66666
1.7000000477 = 3fd9999a -1.7000000477 = bfd9999a
1.7999999523 = 3fe66666 -1.7999999523 = bfe66666
1.8999999762 = 3ff33333 -1.8999999762 = bff33333
2.0000000000 = 40000000 -2.0000000000 = c0000000
Bibliography


## Index

### A
- ALU, 6
- Application Accelerator, 13
- Arithmetic Logic Unit, *see* ALU

### C
- CPUCode, 32

### D
- Data Flow Engine, *see* DFE
- DFE, 13

### E
- EngineCode, 33

### F
- Field Programmable Gate Array, *see* FPGA
- FPGA, 1, 13

### K
- Kernel, 33, 45
- Kernel Graph, 9

### L
- loop unrolling, 7

### M
- Manager, 33
- MaxCompiler, 14
- MaxJ, 13

### O
- OpenSPL, 13
- Out-of-order execution, 8

### P
- Pipeline, 9
  - Fill, 10
  - Flush, 10
  - pipeline, 4

### R
- register, 2
- Register Transfer Language, *see* RTL
- RTL, 6

### S
- Simple Live CPU, *see* SLiC
- SLiC, 14
- Stream, 16
- Subversion, 65
- SVN, *see* Subversion

### T
- Timing Diagram, 6

### V
- Virtual Machine, *see* VM
- VirtualBox, 61
- VM, 59, 61
- VMware, 59

### W
- Waveform Diagram, *see* Timing Diagram