What is an FPGA?

Overview

An FPGA is a type of integrated circuit (IC) that can be programmed for different algorithms after fabrication. Modern FPGA devices consist of up to two million logic cells that can be configured to implement a variety of software algorithms. Although the traditional FPGA design flow is more similar to a regular IC than a processor, an FPGA provides significant cost advantages in comparison to an IC development effort and offers the same level of performance in most cases. Another advantage of the FPGA when compared to the IC is its ability to be dynamically reconfigured. This process, which is the same as loading a program in a processor, can affect part or all of the resources available in the FPGA fabric.

When using the Vivado® HLS compiler, it is important to have a basic understanding of the available resources in the FPGA fabric and how they interact to execute a target application. This chapter presents fundamental information about FPGAs, which is required to guide HLS to the best computational architecture for any algorithm.

FPGA Architecture

The basic structure of an FPGA is composed of the following elements:

- **Look-up table (LUT)**: This element performs logic operations.
- **Flip-Flop (FF)**: This register element stores the result of the LUT.
- **Wires**: These elements connect elements to one another.
- **Input/Output (I/O) pads**: These physically available ports get data in and out of the FPGA.

The combination of these elements results in the basic FPGA architecture shown in Figure 2-1. Although this structure is sufficient for the implementation of any algorithm, the efficiency of the resulting implementation is limited in terms of computational throughput, required resources, and achievable clock frequency.

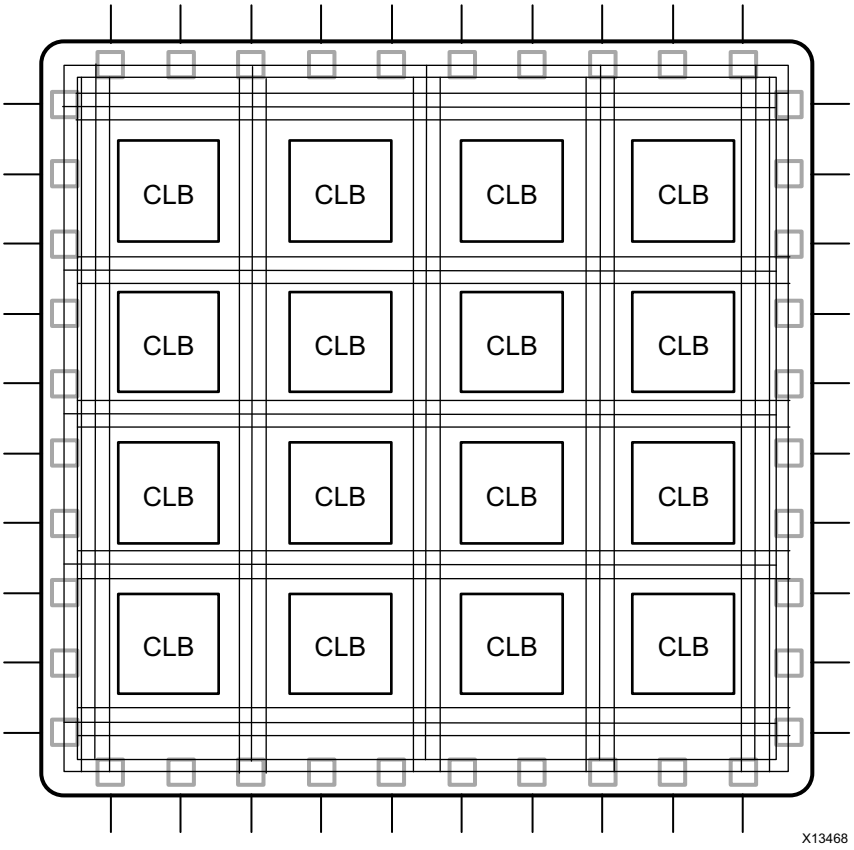

Figure 2-1: **Basic FPGA Architecture**

Contemporary FPGA architectures incorporate the basic elements along with additional computational and data storage blocks that increase the computational density and efficiency of the device. These additional elements, which are discussed in the following sections, are:

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

The combination of these elements provides the FPGA with the flexibility to implement any software algorithm running on a processor and results in the contemporary FPGA architecture shown in Figure 2-2.

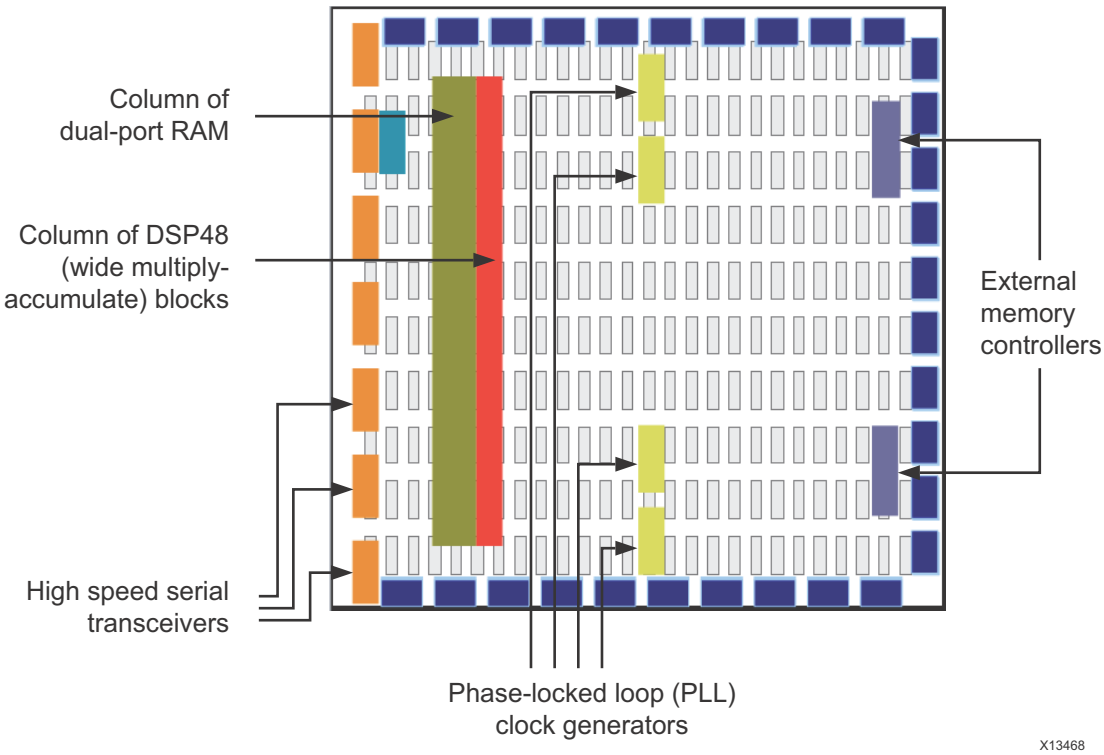

Figure 2-2: **Contemporary FPGA Architecture**

LUT

The LUT is the basic building block of an FPGA and is capable of implementing any logic function of *N* Boolean variables. Essentially, this element is a truth table in which different combinations of the inputs implement different functions to yield output values. The limit on the size of the truth table is *N*, where *N* represents the number of inputs to the LUT. For the general *N*-input LUT, the number of memory locations accessed by the table is:

$$2^N \qquad \textit{Equation 2-1}$$

which allows the table to implement the following number of functions:

$$2^{N^N} \qquad \textit{Equation 2-2}$$

Note: A typical value for *N* in Xilinx FPGA devices is 6.

The hardware implementation of a LUT can be thought of as a collection of memory cells connected to a set of multiplexers. The inputs to the LUT act as selector bits on the multiplexer to select the result at a given point in time. It is important to keep this representation in mind, because a LUT can be used as both a function compute engine and a data storage element. Figure 2-3 shows this functional representation of the LUT.

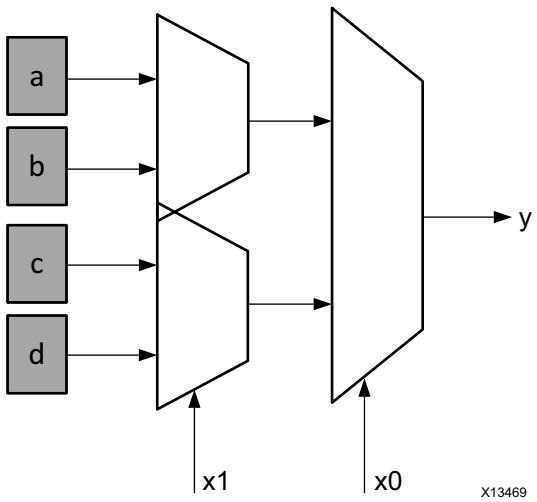

Figure 2-3: **Functional Representation of a LUT as Collection of Memory Cells**

Flip-Flop

The flip-flop is the basic storage unit within the FPGA fabric. This element is always paired with a LUT to assist in logic pipelining and data storage. The basic structure of a flip-flop includes a data input, clock input, clock enable, reset, and data output. During normal operation, any value at the data input port is latched and passed to the output on every pulse of the clock. The purpose of the clock enable pin is to allow the flip-flop to hold a specific value for more than one clock pulse. New data inputs are only latched and passed to the data output port when both clock and clock enable are equal to one. Figure 2-4 shows the structure of a flip-flop.

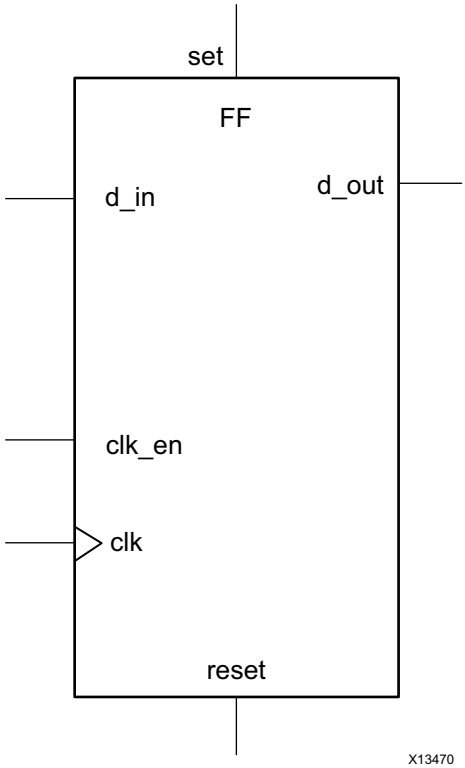

Figure 2-4: **Structure of a Flip-Flop**

DSP48 Block

The most complex computational block available in a Xilinx FPGA is the DSP48 block, which is shown in Figure 2-5. The DSP48 block is an arithmetic logic unit (ALU) embedded into the fabric of the FPGA, which is composed of a chain of three different blocks. The computational chain in the DSP48 is composed of an add/subtract unit connected to a multiplier connected to a final add/subtract/accumulate engine. This chain allows a single DSP48 unit to implement functions of the form:

$$p = a \times (b + d) + c \qquad \textit{Equation 2-3}$$

or

$$p += a \times (b + d) \qquad \textit{Equation 2-4}$$

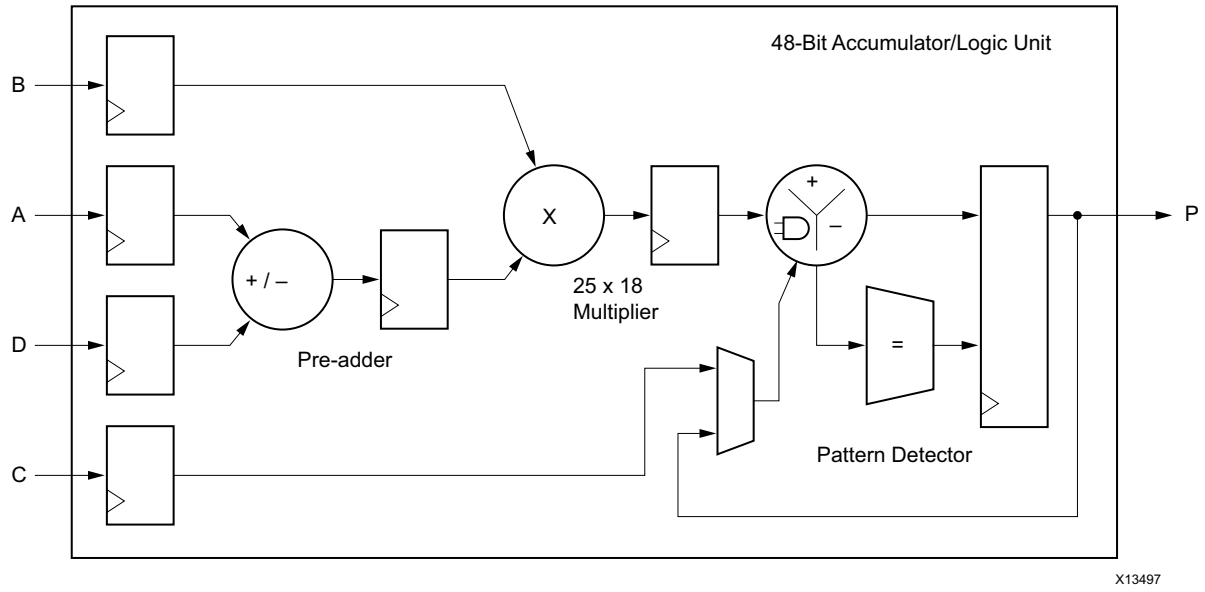

Figure 2-5: **Structure of a DSP48 Block**

BRAM and Other Memories

The FPGA fabric includes embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These elements are block RAMs (BRAMs), LUTs, and shift registers.

The BRAM is a dual-port RAM module instantiated into the FPGA fabric to provide on-chip storage for a relatively large set of data. The two types of BRAM memories available in a device can hold either 18 k or 36 k bits. The number of these memories available is device specific. The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations.

In terms of how arrays are represented in C/C++ code, BRAMs can implement either a RAM or a ROM. The only difference is when the data is written to the storage element. In a RAM

configuration, the data can be read and written at any time during the runtime of the circuit. In contrast, in a ROM configuration, data can only be read during the runtime of the circuit. The data of the ROM is written as part of the FPGA configuration and cannot be modified in any way.

As previously discussed, the LUT is a small memory in which the contents of a truth table are written during device configuration. Due to the flexibility of the LUT structure in Xilinx FPGAs, these blocks can be used as 64-bit memories and are commonly referred to as *distributed memories*. This is the fastest kind of memory available on the FPGA device, because it can be instantiated in any part of the fabric that improves the performance of the implemented circuit.

The shift register is a chain of registers connected to each other. The purpose of this structure is to provide data reuse along a computational path, such as with a filter. For example, a basic filter is composed of a chain of multipliers that multiply a data sample against a set of coefficients. By using a shift register to store the input data, a built-in data transport structure moves the data sample to the next multiplier in the chain on every clock cycle. Figure 2-6 shows an example shift register.

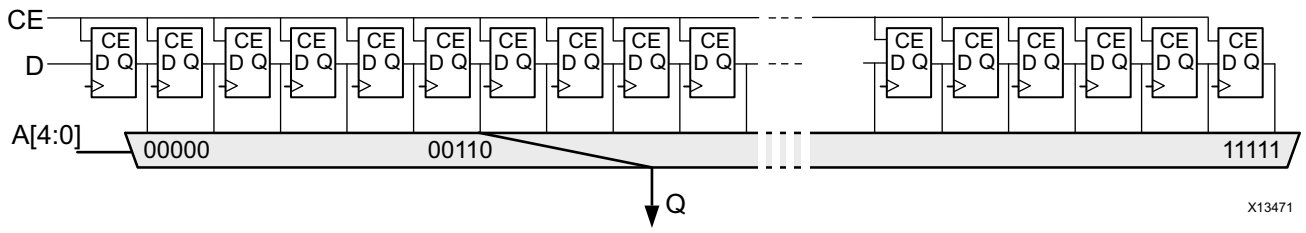

Figure 2-6: **Structure of an Addressable Shift Register**

FPGA Parallelism Versus Processor Architectures

When compared with processor architectures, the structures that comprise the FPGA fabric enable a high degree of parallelism in application execution. The custom processing architecture generated by the HLS compiler for a software program presents a different execution paradigm, which must be taken into account when deciding to port an application from a processor to an FPGA. To examine the benefits of the FPGA execution paradigm, this section provides a brief review of processor program execution.

Program Execution on a Processor

A processor, regardless of its type, executes a program as a sequence of instructions that translate into useful computations for the software application. This sequence of instructions is generated by processor compiler tools, such as the GNU Compiler Collection (GCC), which transform an algorithm expressed in C/C++ into assembly language

constructs that are native to the processor. The job of a processor compiler is to take a C function of the form:

$$z = a + b; \qquad \textit{Equation 2-5}$$

and transform it into assembly code as follows:

```
ADD $R1,$R2,$R3
```

Figure 2-7: **Computation Expressed Assembly Code**

The assembly code in Figure 2-7 defines the addition operation to compute the value of *z* in terms of the internal registers of a processor. The code states that the input values for the computation are stored in registers `R1` and `R2`, and the result of the computation is stored in register `R3`. This code is simple, and it does not express all the instructions needed to compute the value of *z*. This code only handles the computation after the data has arrived at the processor. Therefore, the compiler must create additional assembly language instructions to load the registers of the processor with data from a central memory and to write back the result to memory. The complete assembly program to compute the value of *z* is as follows:

```
LD      a, $R1
LD      b, $R2
ADD     $R1,$R2,$R3
ST      $R3, c
```

Figure 2-8: **Complete Assembly Program to Compute Z**

The code in Figure 2-8 shows that even a simple operation, such as the addition of two values, results in multiple assembly instructions. The computational latency of each instruction is not equal across instruction types. For example, depending on the location of `a` and `b`, the `LD` operations take a different number of clock cycles to complete. If the values are in the processor cache, these load operations complete within a few tens of clock cycles. If the values are in the main, double data rate (DDR) memory, the operations take between hundreds and thousands of clock cycles to complete. If the values are in a hard drive, the load operations take even longer to complete. This is why software engineers with cache hit traces spend so much time restructuring their algorithms to increase the spatial locality of data in memory to increase the cache hit rate and decrease the processor time spent per instruction.

IMPORTANT: *The level of effort required by the software engineer in restructuring algorithms to better fit the available processor cache is not required when the same operation is implemented in an FPGA.*

Program Execution on an FPGA

The FPGA is an inherently parallel processing fabric capable of implementing any logical and arithmetic function that can run on a processor. The main difference is that the HLS compiler, which is used to transform software descriptions into RTL, is not hindered by the restrictions of a cache and a unified memory space.

The computation of *z* is compiled by HLS into several LUTs required to achieve the size of the output operand. For example, assume that in the original software program the variable *a*, *b*, and *z* are defined with the short data type. This type, which defines a 16-bit data container, gets implemented as 16 LUTs by HLS.

Note: As a general rule, 1 LUT is equivalent to 1 bit of computation.

The LUTs used for the computation of *z* are exclusive to this operation only. Unlike a processor, where all computations share the same ALU, an FPGA implementation instantiates independent sets of LUTs for each computation in the software algorithm.

In addition to assigning unique LUT resources per computation, the FPGA differs from a processor in both memory architecture and the cost of memory accesses. In an FPGA implementation, the HLS compiler arranges memories into multiple storage banks as close as possible to the point of use in the operation. This results in an instantaneous memory bandwidth, which far exceeds the capabilities of a processor. For example, the Xilinx Kintex®-7 410T device has a total of 1,590 18 k-bit BRAMs available. In terms of memory bandwidth, the memory layout of this device provides the software engineer with the capacity of 0.5M-bits per second at the register level and 23T-bits per second at the BRAM level.

With regard to computational throughput and memory bandwidth, the HLS compiler exercises the capabilities of the FPGA fabric through the processes of scheduling, pipelining, and dataflow. Although transparent to the user, these processes are integral stages of the software compilation process that extract the best possible circuit-level implementation of the software application.

Scheduling

Scheduling is the process of identifying the data and control dependencies between different operations to determine when each will execute. In traditional FPGA design, this is a manual process also referred to as parallelizing the software algorithm for a hardware implementation.

HLS analyzes dependencies between adjacent operations as well as across time. This allows the compiler to group operations to execute in the same clock cycle and to set up the hardware to allow the overlap of function calls. The overlap of function call executions removes the processor restriction that requires the current function call to fully complete before the next function call to the same set of operations can begin. This process is called *pipelining* and is covered in detail in the following section and remaining chapters.

Pipelining

Pipelining is a digital design technique that allows the designer to avoid data dependencies and increase the level of parallelism in an algorithm hardware implementation. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit is divided into a chain of independent stages. All stages in the chain run in parallel on the same clock cycle. The only difference is the source of data for each stage. Each stage in the computation receives its data values from the result computed by the preceding stage during the previous clock cycle. For example, to compute the following function the HLS compiler instantiates one multiplier and two adder blocks:

$$y = (a \times x) + b + c \qquad \textit{Equation 2-6}$$

Figure 2-9 shows this compute structure and the effects of pipelining. It shows two implementations of the example function. The top implementation is the datapath required to compute the result *y* without pipelining. This implementation behaves similarly to the corresponding C/C++ function in that all input values must be known at the start of the computation, and only one result *y* can be computed at a time. The bottom implementation shows the pipelined version of the same circuit.

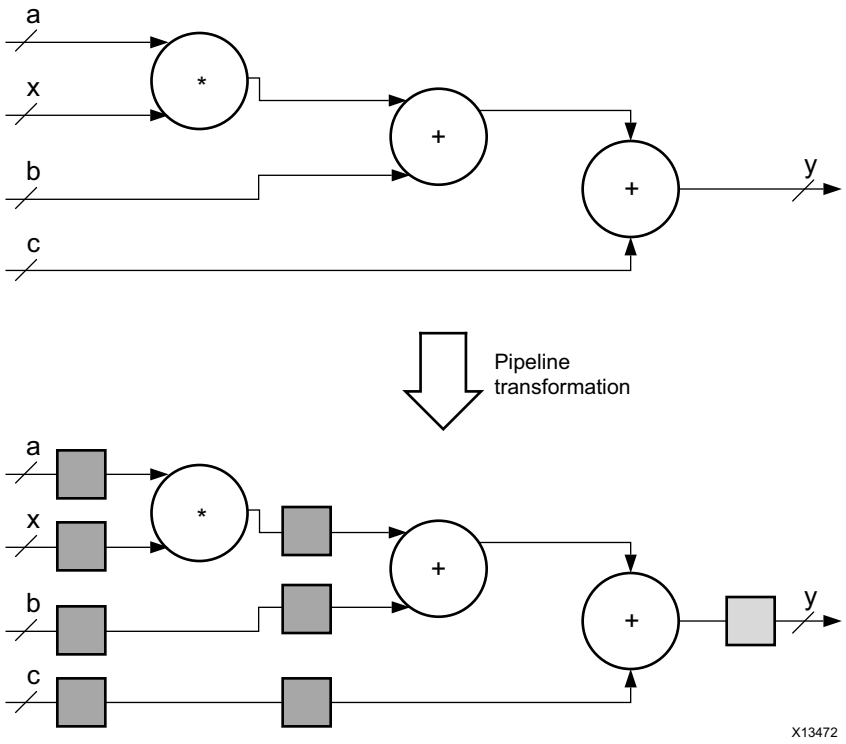

Figure 2-9: **FPGA Implementation of a Compute Function**

The boxes in the datapath in Figure 2-9 represent registers that are implemented by flip-flop blocks in the FPGA fabric. Each box can be counted as a single clock cycle. Therefore, in the pipelined version, the computation of each result *y* takes three clock

cycles. By adding the register, each block is isolated into separate compute sections in time. This means that the section with the multiplier and the section with the two adders can run in parallel and reduce the overall computational latency of the function. By running both sections of the datapath in parallel, the block is essentially computing the values *y* and *y'* in parallel, where *y'* is the result of the next execution of Equation 2-6. The initial computation of *y*, which is also referred to as the *pipeline fill time*, takes three clock cycles. After this initial computation, a new value of *y* is available at the output on every clock cycle, because the computation pipeline contains overlapped data sets for the current and subsequent *y* computations.

Figure 2-10 shows a pipelined architecture in which raw data (dark gray), semi-computed data (white), and final data (light gray) exist simultaneously, and each stage result is captured in its own set of registers. Thus, although the latency for such computation is in multiple cycles, with every cycle a new result can be produced.

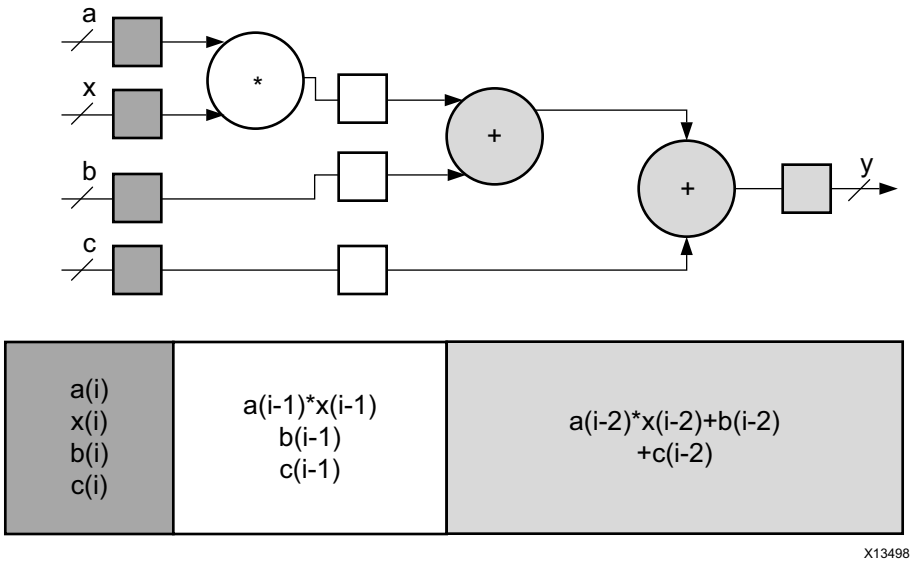

Figure 2-10: **Pipelined Architecture**

Dataflow

Dataflow is another digital design technique, which is similar in concept to pipelining. The goal of dataflow is to express parallelism at a coarse-grain level. In terms of software execution, this transformation applies to parallel execution of functions within a single program.

HLS extracts this level of parallelism by evaluating the interactions between different functions of a program based on their inputs and outputs. The simplest case of parallelism is when functions work on different data sets and do not communicate with each other. In this case, HLS allocates FPGA logic resources for each function and then runs the blocks in independently. The more complex case, which is typical in software programs, is when one

function provides results for another function. This case is referred to as the *consumer-producer scenario*.

HLS supports two use models for the consumer-producer scenario. In the first use model, the producer creates a complete data set before the consumer can start its operation. Parallelism is achieved by instantiating a pair of BRAM memories arranged as memory banks ping and pong. Each function can access only one memory bank, ping or pong, for the duration of a function call. When a new function call begins, the HLS-generated circuit switches the memory connections for both the producer and the consumer. This approach guarantees functional correctness but limits the level of achievable parallelism to across function calls.

In the second use model, the consumer can start working with partial results from the producer, and the achievable level of parallelism is extended to include execution within a function call. The HLS-generated modules for both functions are connected through the use of a first in, first out (FIFO) memory circuit. This memory circuit, which acts as a queue in software programming, provides data-level synchronization between the modules. At any point during a function call, both hardware modules are executing their programming. The only exception is that the consumer module waits for some data to be available from the producer before beginning computation. In HLS terminology, the wait time of the consumer module is referred to as the *interval* or *initiation interval (II)*.

Basic Concepts of Hardware Design

Overview

One of the key differences between a processor and an FPGA is whether the processing architecture is fixed. This difference directly affects how a compiler for each target works. With a processor, the computation architecture is fixed, and the job of the compiler is to determine how to best fit the software application in the available processing structures. Performance is a function of how well the application maps to the capabilities of the processor and the number of processor instructions needed for correct execution.

In contrast, an FPGA is similar to a blank slate with a box of building blocks. The job of the Vivado® HLS compiler is to create a processing architecture from the box of building blocks that best fits the software program. The process of guiding the HLS compiler to create the best processing architecture requires fundamental knowledge about hardware design concepts.

This chapter covers general design concepts that apply to both FPGA and processor-based designs and explains how these concepts are related. This chapter does not cover detailed aspects of FPGA design. As with processor compilers, the HLS compiler handles the low-level details of the algorithm implementation into the FPGA logic fabric.

Clock Frequency

The processor clock frequency is one of the first items to consider when determining the execution platform of a specific algorithm. A commonly used guideline is that a high clock frequency translates into a higher performance execution rate of an algorithm. Although this might be a good first order rule for choosing between processors, it is actually misleading and can lead the designer to make the wrong choice when selecting between a processor and an FPGA.

The reason this general guideline is misleading is related to the nominal difference in clock frequency between a processor and an FPGA. For example, when comparing the clock frequencies of processors and FPGAs, it is not uncommon to face the comparison shown in Table 3-1.

Table 3-1: **Maximum Clock Frequency Examples**

Processor	FPGA
2 GHz	500 MHz

A simple analysis of the values in Table 3-1 can mislead a designer to assume the processor has four times the performance of the FPGA. This simple analysis incorrectly assumes that the only difference between the platforms is clock frequency. However, the platforms have additional differences.

The first major difference between a processor and an FPGA is how a software program is executed. A processor is able to execute any program on a common hardware platform. This common platform comprises the core of the processor and defines a fixed architecture onto which all software must be fitted. The compiler, which has a built-in understanding of the processor architecture, compiles the user software into a set of instructions. The resulting set of instructions is always executed in the same fundamental order, as shown in Figure 3-1.

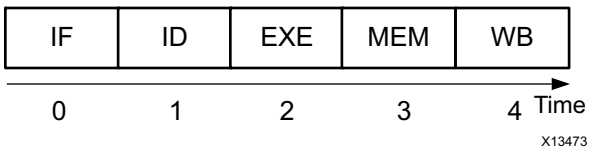

Figure 3-1: **Processor Instruction Execution Stages**

Regardless of the type of processor, standard versus specialized, the execution of an instruction is always the same. Each instruction of the user application must go through the following stages:

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Execute (EXE)
4. Memory operations (MEM)
5. Write back (WB)

The purpose of each stage is summarized in Table 3-2.

Table 3-2: **Instruction Processing Stages**

Stage	Description
IF	Get the instruction from program memory.
ID	Decode the instruction to determine the operation and the operators.
EXE	Execute the instruction on the available hardware. In a standard processor, this means the arithmetic logic unit (ALU) or floating point unit (FPU). A specialized processor adds on fixed function accelerators to the capabilities of the standard processor at this stage of instruction processing.
MEM	Fetch data for the next instruction using memory operations.
WB	Write the results of the instruction either to local registers or global memory.

Most modern processors include multiple copies of the instruction execution path and are capable of running instructions with some degree of overlap. Because instructions in a processor usually depend on each other, the overlap between copies of the instruction execution hardware is not perfect. In the best of cases, only the overhead stages introduced by using a processor can be overlapped. The EXE stages, which are responsible for application computation, execute sequentially. The reasons for this sequential execution are related to limited resources in the EXE stage and dependence between instructions.

Figure 3-2 shows a processor with multiple instructions executing in a semi-parallel order. This is the best case for a processor in which all instructions are executing as quickly as possible. Even in this best case, the processor is limited to only one EXE stage per clock cycle. This means that the user application moves forward by one operation per clock cycle. Even if the compiler determined that all five EXE stages could execute in parallel, the structure of the process would prevent it.

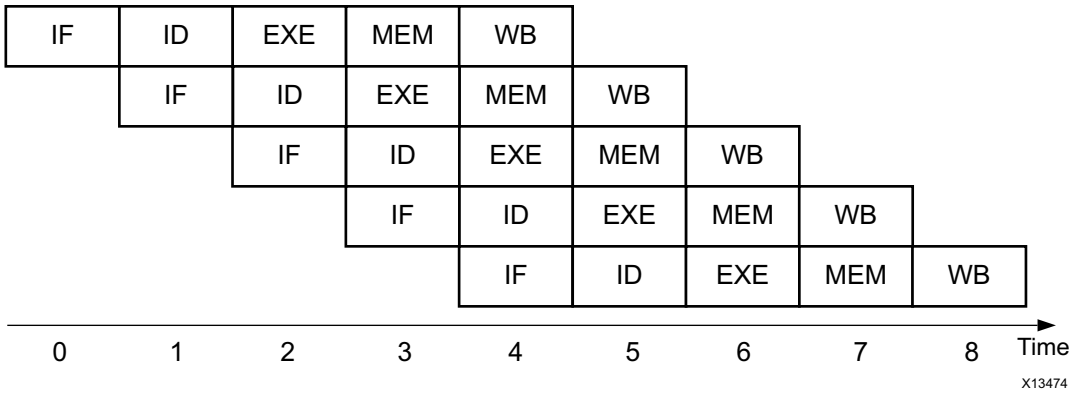

Figure 3-2: **Processor with Multiple Instruction Execution Units**

An FPGA does not execute all software on a common computation platform. It executes a single program at a time on a custom circuit for that program. Therefore, changing the user application changes the circuit in the FPGA. Unlike Figure 3-1, the EXE stage appears as shown in Figure 3-3 when processing in an FPGA. The presence of the MEM stage is application dependent.

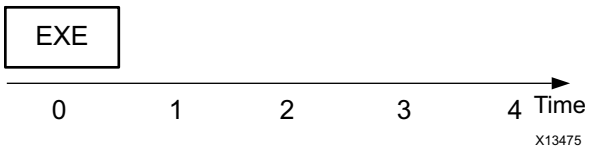

Figure 3-3: **FPGA Instruction Execution Stages**

Given this flexibility, the HLS compiler does not need to account for overhead stages in the platform and can find ways of maximizing instruction parallelism. Working with the same assumptions as in Figure 3-2, the execution profile of the same software in an FPGA is shown in Figure 3-4.

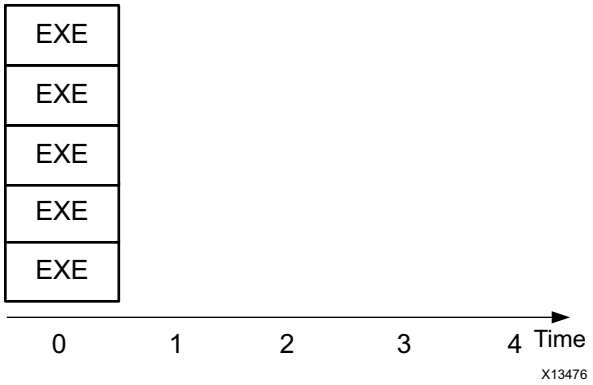

Figure 3-4: **FPGA with Multiple Instruction Execution Units**

Based on the comparison of Figure 3-2 and Figure 3-4, the FPGA has a nominal performance advantage of 9x compared to the processor. Actual numbers are always application specific, but FPGAs generally demonstrate at least 10x the performance of a processor for computationally intensive applications.

Another issue hidden by only focusing on the clock frequency is the power consumption of a software program. The approximation to power consumption is given by:

$$P = \frac{1}{2}cFV^2 \qquad \textit{Equation 3-1}$$

As shown in Equation 3-1, the relationship between power consumption and clock frequency is supported by empirical data, which shows higher power usage in a processor than an FPGA for the same computational workload. By creating a custom circuit per software program, an FPGA is able to run at a lower clock frequency with maximum parallelism between operations and without the instruction interpretation overhead found in a processor.

RECOMMENDED: *When selecting between a processor and an FPGA, it is recommended that application requirements and computational workload are analyzed based on throughput and latency instead of a maximum clock frequency.*

Latency and Pipelining

Latency is the number of clock cycles it takes to complete an instruction or set of instructions to generate an application result value. Using the basic processor architecture shown in Figure 3-1, the latency of an instruction is five clock cycles. If the application has a total of five instructions, the overall latency for this simple model is 25 clock cycles. That is, the result of the application is not available until 25 clock cycles expire.

Application latency is a key performance metric in both FPGAs and processors. In both cases, the problem of latency is resolved through the use of pipelining. In a processor, pipelining means that the next instruction can be launched into execution before the current instruction is complete. This allows the overlap of overhead stages required in instruction set processing. The best case result of pipelining for a processor is shown in Figure 3-2. By overlapping the execution of instructions, the processor achieves a latency of nine clock cycles for the five instruction application.

In an FPGA, the overhead cycles associated with instruction processing are not present. The latency is measured by how many clock cycles it takes to run the EXE stage of the original processor instruction. For the case in Figure 3-3, the latency is one clock cycle. Parallelism also plays an important role in latency. For the full five instruction application, the FPGA latency is also one clock cycle, as shown in Figure 3-4. With the one clock cycle latency of the FPGA, it might not be clear why pipelining is advantageous. However, the reason for pipelining in an FPGA is the same as in a processor, that is, to improve application performance.

As previously explained, the FPGA is a blank slate with building blocks that must be connected to implement an application. The HLS compiler can connect the blocks directly or through registers. Figure 3-5 shows an implementation of the EXE stage in Figure 3-3 that is implemented using five building blocks.

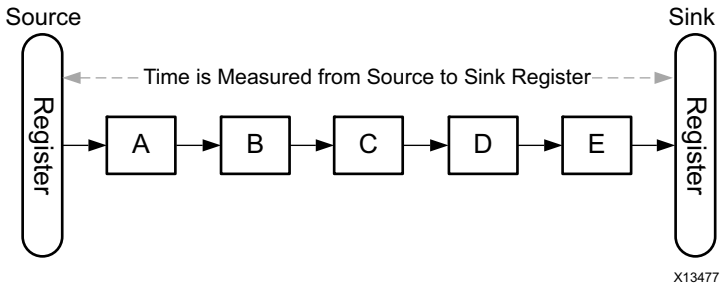

Figure 3-5: **FPGA Implementation without Pipelining**

Operation timing in an FPGA is the length of time it takes a signal to travel from a source register to a sink register. Assuming each building block in Figure 3-5 requires 2 ns to execute, the current design requires 10 ns to implement the functionality. The latency is still one clock cycle, but the clock frequency is limited to 100 MHz. The 100 MHz frequency limit is derived from the definition of clock frequency in an FPGA. For the case of an FPGA circuit, the clock frequency is defined as the longest signal travel time between source and sink registers.

Pipelining in an FPGA is the process of inserting more registers to break up large computation blocks into smaller segments. This partitioning of the computation increases the latency in absolute number of clock cycles but increases performance by allowing the custom circuit to run at a higher clock frequency.

Figure 3-6 shows the implementation of the processing architecture in Figure 3-5 after complete pipelining. Complete pipelining means that a register is inserted between each building block in the FPGA circuit. The addition of registers reduces the timing requirement of the circuit from 10 ns to 2 ns, which results in a maximum clock frequency of 500 MHz. In addition, by separating the computation into separate register-bounded regions, each block is allowed to always be busy, which positively impacts the application throughput.

One issue with pipelining is the latency of the circuit. The original circuit of Figure 3-5 has a latency of one clock cycle at the expense of a low clock frequency. In contrast, the circuit of Figure 3-6 has a latency of five clock cycles at a higher clock frequency.

IMPORTANT: *The latency caused by pipelining is one of the trade-offs to consider during FPGA design.*

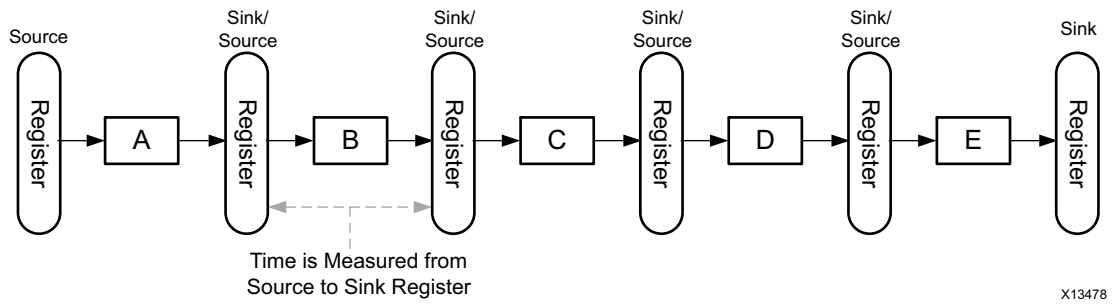

Figure 3-6: **FPGA Implementation with Pipelining**

Throughput

Throughput is another metric used to determine overall performance of an implementation. It is the number of clock cycles it takes for the processing logic to accept the next input data sample. With this value, it is important to remember that the clock frequency of the circuit changes the meaning of the throughput number.

For example, both Figure 3-5 and Figure 3-6 show implementations that require one clock cycle between input data samples. The key difference is that the implementation in Figure 3-5 requires 10 ns between input samples, whereas the circuit in Figure 3-6 only requires 2 ns between input data samples. After the time base is known, it is clear that the second implementation has higher performance, because it can accept a higher input data rate.

Note: The definition of throughput described in this section can also be used when analyzing applications executing on a processor.

Memory Architecture and Layout

The memory architecture of the selected implementation platform is one of the physical elements that can affect the performance of a software application. Memory architecture determines the upper bound on achievable performance. At some performance point, all applications on either a processor or an FPGA become memory bound regardless of the type and number of available computational resources. One strategy in FPGA design is understanding where the memory bound is and how it can be affected by data layout and memory organization.

In a processor-based system, the software engineer must fit the application on essentially the same memory architecture regardless of the specific type of processor. This commonality simplifies the process of application migration at the expense of performance. Common memory architecture familiar to software engineers consists of memories that are slow, medium, or fast based on the number of clock cycles it takes to get the data to the processor. These memory classifications are defined in Table 3-3.

Table 3-3: **Memory Type Definitions**

Memory Type	Definition
Slow	Mass storage devices, such as hard drives
Medium	DDR memories
Fast	On-chip cache memories of different sizes depending on the specific processor

The memory architecture shown in this table assumes that the user is presented with a single large memory space. Within this memory space, the user allocates and deallocates regions to store program data. The physical location of data and how it moves between the different levels in the hierarchy is handled by the computation platform and is transparent to the user. In this kind of system, the only way to boost performance is to reuse data in the cache as much as possible.

To achieve this goal, the software engineer must spend large amounts of time looking at cache traces, restructuring the software algorithm to increase data locality, and managing memory allocation to minimize the instantaneous memory footprint of the program. Although all of these techniques are portable across processors, the results are not. A software program must be tuned for each processor it runs on to maximize performance.

With experience in working with processor-based memory, the first difference a software engineer encounters when working with memory in an FPGA is the lack of fixed on-chip memory architecture. FPGA-based systems can be attached to slow and medium memories but exhibit the greatest degree of differentiation in terms of available fast memories. That is, instead of restructuring the software to best use an existing cache, the HLS compiler builds a fast memory architecture to best fit the data layout in the algorithm. The resulting FPGA implementation can have one or more internal banks of different sizes that can be accessed independently from one another.

The code examples in Figure 3-7 show best practice recommendations for addressing the memory requirements of a program.

Processor Code

```
void foo(......)
{
   int *A = (int *)malloc(10 * sizeof(int));
   ....
   free(A);
}
```

FPGA Code

```
void foo(......)
{
   int A[10];
   ....

}
```

Figure 3-7: **Processor and FPGA Code Examples**

The FPGA code might surprise a seasoned software engineer with its lack of dynamic memory allocation. The use of dynamic memory allocation has long been part of the best practice guidelines for processor-based systems due to the underlying fixed memory architecture.

In contrast to this approach, the HLS compiler builds a memory architecture that is tailored to the application. This tailored memory architecture is shaped both by the size of the memory blocks in the program as well as by how the data is used throughout program execution. Current state-of-the-art compilers for FPGAs, such as HLS, require that the memory requirements of an application are fully analyzable at compile time.

The benefit of static memory allocation is that HLS can implement the memory for array `A` in different ways. Depending on the computation in the algorithm, the HLS compiler can implement the memory for `A` as registers, shift registers, FIFOs, or BRAMs.

Note: Despite the restriction on dynamic memory allocation, pointers are fully supported by the HLS compiler. For details on pointer support, see Pointers in Chapter 4.

Registers

A register implementation of a memory is the fastest possible memory structure. In this implementation style, each entry of `A` becomes an independent entity. Each independent entity is embedded into the computation where it is used without the need to address logic or additional delays.

Shift Register

In processor programming terms, a shift register can be thought of as a special case of a queue. In this implementation, each element of `A` is used multiple times in different parts of the computation. The key characteristic of a shift register is that every element of `A` can be accessed on every clock cycle. In addition, moving all data items to the next adjacent storage container requires only one clock cycle.

FIFO

A FIFO can be thought of as a queue with a single point of entry and a single point of exit. This kind of structure is typically used to transmit data between program loops or functions. There is no addressing logic involved, and the implementation details are completely handled by the HLS compiler.

BRAM

A BRAM is a random-access memory that is embedded into the FPGA fabric. A Xilinx FPGA device includes many of these embedded memories. The exact number of memories is device specific. In processor programming terms, this kind of memory can be thought of as a cache with the following limitations:

- Does not implement cache coherency, collision, and cache miss tracking logic typically found in a processor cache.
- Holds its values only as long as the device is powered on.
- Supports parallel same cycle access to two different memory locations.