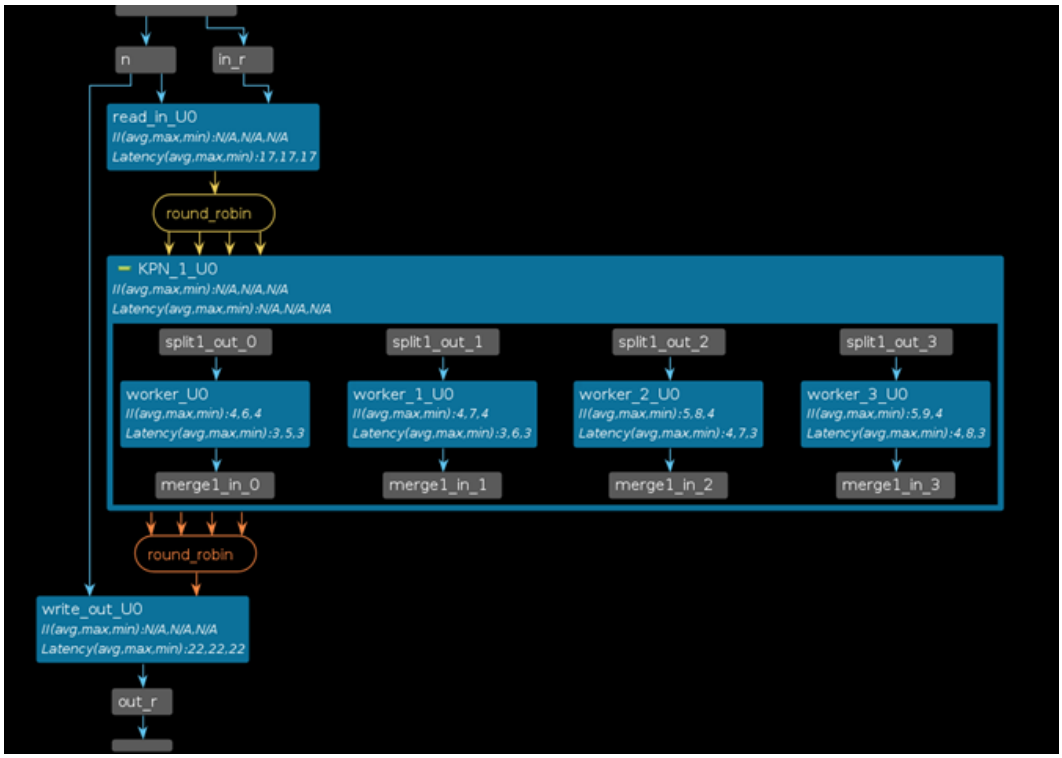


Summary

HLS designs that are purely data-driven and that do not require any interaction with the software application can be modeled using data-driven TLP models. Examples of such designs are:

- Simple rule-based “firewall” with “rules” compiled into the kernel
- Fast-Fourier Transforms with configuration data compiled into the kernel
- FIR filters with coefficients compiled into the kernel

If the design requires data transfer to/from external memory, then the control-driven TLP model can used. Examples of such designs include:

- Network router where the routing table must be updated entirely for kernel execution
- Load-balancer that uses a hash map to send data to a server, that must update server list, server map, and corresponding IP addresses simultaneously

However, most designs will be a mixed control-driven and data-driven model, requiring some access to external memory, and enabling streaming between parallel and pipelined tasks within the HLS design.
In summary, this chapter described some modeling choices to consider when designing your application written in C/C++. So far, this discussion talked about structuring your algorithm at a high level to make use of these special models such as the task-channel or dataflow optimizations. Another key aspect to achieving good throughput is to also consider instruction-level parallelism. Instruction-level parallelism in HLS refers to the ability to efficiently parallelize the operations inside loops, functions, and even arrays. The next few sections will walk you through these lower-level optimizations that work hand-in-hand with the macro-level optimizations.

Loops Primer

When writing code intended for high-level synthesis (HLS), there is a frequent need to implement repetitive algorithms that process blocks of data — for example, signal or image processing. Typically, the C/C++ source code tends to include several loops or several nested loops.

When it comes to optimizing performance, loops are one of the best places to start exploring optimization. Each iteration of the loop takes at least one clock cycle to execute in hardware. Thinking from the hardware perspective, there is an implicit *wait until clock* for the loop body. The next iteration of a loop only starts when the previous iteration is finished. To improve performance loops can generally be either *pipelined* or *unrolled* to take advantage of the highly distributed and parallel FPGA architecture, as explained in the following sections.

Pipelining Loops

Pipelining loops permits starting the next iteration of a loop before the previous iteration finishes, enabling portions of the loop to overlap in execution. By default, every iteration of a loop only starts when the previous iteration has finished. In the loop example below, a single iteration of the loop adds two variables and stores the result in a third variable. Assume that in hardware this loop takes three cycles to finish one iteration. Also, assume that the loop variable `len` is 20, that is, the `vadd` loop runs for 20 iterations in the kernel. Therefore, it requires a total of 60 clock cycles (20 iterations * 3 cycles) to complete all the operations of this loop.

```
vadd: for(int i = 0; i < len; i++) {
   c[i] = a[i] + b[i];
}
```

★ **Tip:** It is good practice to always label a loop as shown in the example above (`vadd:…`). This practice helps with debugging the design in Vitis HLS. Sometimes the unused labels generate warnings during compilation, which can be safely ignored.

Pipelining the loop allows subsequent iterations of the loop to overlap and run concurrently. Pipelining a loop can be enabled by adding the PIPELINE pragma or directive inside the body of the loop as shown below:

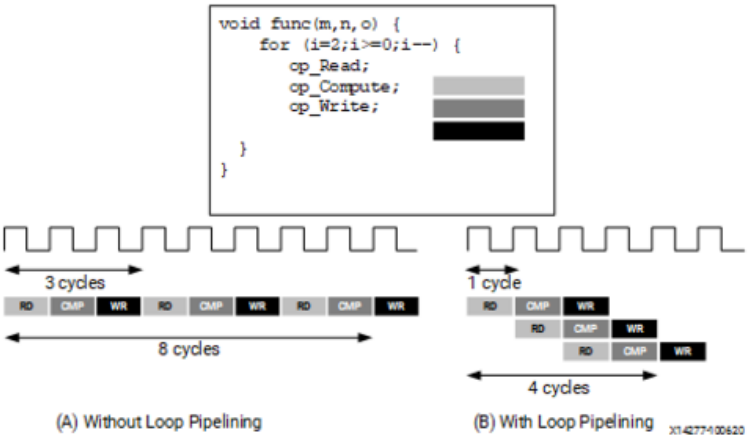
```
vadd: for(int i = 0; i < len; i++) {
#pragma HLS PIPELINE
c[i] = a[i] + b[i];
}
```

★ **Tip:** Vitis HLS automatically pipelines loops with more than 64 iterations. This feature can be changed or disabled using the `config_compile -pipeline_loops` command.

The number of cycles it takes to start the next iteration of a loop is called the Initiation Interval (II) of the pipelined loop. So II = 2 means the next iteration of a loop starts two cycles after the current iteration. An II = 1 is the ideal case, where each iteration of the loop starts in the very next cycle. When you use pragma HLS pipeline, you can specify the II for the compiler to achieve. If a target II is not specified, the compiler will try to achieve II=1 by default.

The following figure illustrates the difference in execution between pipelined and non-pipelined loops. In this figure, (A) shows the default sequential operation where there are three clock cycles between each input read (II = 3), and it requires eight clock cycles before the last output write is performed.

Figure: Loop Pipelining



In the pipelined version of the loop shown in (B), a new input sample is read every cycle (II = 1) and the final output is written after only four clock cycles: substantially improving both the II and latency while using the same hardware resources.

!! Important: Pipelining a loop causes any loops nested inside the pipelined loop to get automatically unrolled.

If there are data dependencies inside a loop, it might not be possible to achieve II = 1, and a larger initiation interval might be the result. Loop dependencies are data dependencies that can constrain the optimization of loops, typically pipelining. They can be within a single iteration of a loop and or between different iterations of a loop. The easiest way to understand loop dependencies is to examine an extreme example. In the following example, the result of the loop is used as the loop continuation or exit condition. Each iteration of the loop must finish before the next can start.

```
Minim_Loop: while (a != b) {
if (a > b)a -= b;
else b -= a;
}
```

The `Minim_Loop` loop in the example above cannot be pipelined because the next iteration of the loop cannot begin until the previous iteration ends. Not all loop dependencies are as extreme as this, but the example highlights that some operations cannot begin until some other operation has been completed. The solution is to try to ensure that the initial operation is performed as early as possible.
Loop dependencies can occur with any and all types of data. They are particularly common when using arrays.

Automatic Loop Pipelining

The `config_compile -pipeline_loops` command enables loops to be pipelined automatically based on the iteration count. All loops with a trip count greater than the specified limit are automatically pipelined. The default limit is 64.
Given the following example code:

```
loop1: for (i = 0; i < 5; i++) {
   // do something 5 times ...
}
```

If the `-pipeline_loops` option is set to 4, the `for` loop (`loop1`) will be pipelined. This is equivalent to adding the PIPELINE pragma as shown in the following code snippet:

```
loop1: for (i = 0; i < 5; i++) {
#pragma HLS PIPELINE II=1
```

```
      // do something 5 times ...
   }
```

However, for nested loops, the tool starts at the innermost loop, and determines if it can be pipelined, and then moves up the loop nest. In the example below, the `-pipeline_loops` threshold is still 4, and the inner most loop (`loop1`) is marked to be pipelined. Then the tool evaluates the parent loop of the loop pair (`loop2`) and finds that it can also be pipelined. In this case, the tool unrolls `loop1` into `loop2`, and marks `loop2` to be pipelined. Then it moves up to the parent loop (`loop3`) and repeats the analysis.

```
loop3: for (y = 0; y < 480; y++) {
   loop2: for (x = 0; x < 640; x++) {
      loop1: for (i = 0; i < 5; i++) {
         // do something 5 times ...
      }
   }
}
```

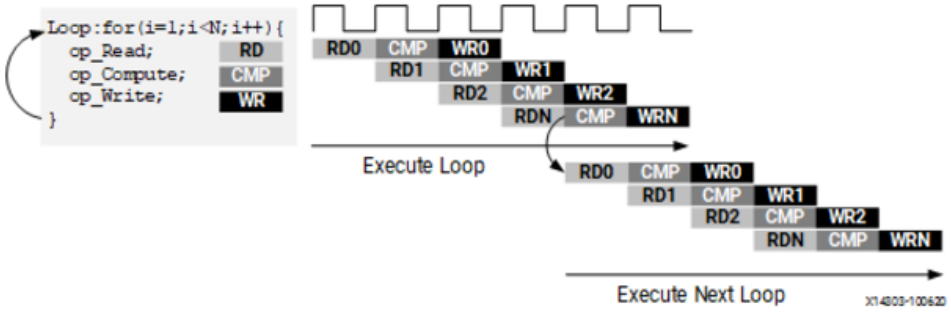
If there are loops in the design for which you do not want to use automatic pipelining, apply the PIPELINE directive with the `off` option to that loop. The `off` option prevents automatic loop pipelining.

!! Important: Vitis HLS applies the `config_compile -pipeline_loops` command after performing all user-specified directives. For example, if Vitis HLS applies a user-specified UNROLL directive to a loop, the loop is first unrolled, and automatic loop pipelining cannot be applied.

Rewinding Pipelined Loops for Performance

The PIPELINE pragma has an option called `rewind`. This option enables overlap of the execution of successive calls to the pipelined loop when this loop is the outermost construct of the top function, or of a dataflow region and the dataflow region is executed multiple times.
The following figure shows the operation when the `rewind` option is used when pipelining a loop. At the end of the loop iteration count, the loop starts to execute again. While it generally re-executes immediately, a delay is possible and is shown and described in the GUI.

Figure: Loop Pipelining with Rewind Option



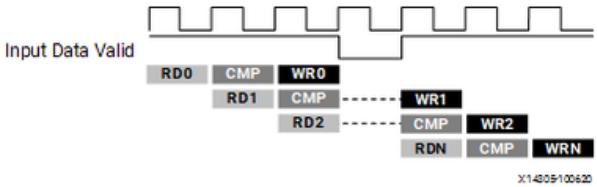
Note: If a loop is used around a DATAFLOW region, Vitis HLS automatically implements it to allow successive executions to overlap.

Flushing Pipelines and Pipeline Types

Flushing Pipelines

Pipelines continue to execute as long as data is available at the input of the pipeline. If there is no data available to process, the pipeline will stall. This is shown in the following figure, where the `Input Data Valid` signal goes low to indicate there is no more valid input data. Once the signal goes high, indicating there is new data available to process, the pipeline will continue operation.

Figure: Loop Pipelining with Stall



In some cases, it is desirable to have a pipeline that can be “emptied” or “flushed.” The `flush` option is provided to perform this. When a pipeline is “flushed” the pipeline stops reading new inputs when none are available (as determined by a data `valid` signal at the start of the pipeline) but continues processing, shutting down each successive pipeline stage, until the final input has been processed through to the output of the pipeline.
As described below, Vitis HLS will automatically select the right pipeline style to use for a given pipelined loop or function. However, you can override this default behavior by using the `config_compile -pipeline_style` command to specify the default pipeline style.
You can also specify the pipeline style for functions or loops with the PIPELINE pragma or directive. This option applies to the specific scope of the pragma or directive and does not change the global default assigned by `config_compile`.
Both `stp` and `flp` types of pipelines use the standard pipeline logic where the hardware pipeline created use various kinds of blocking signals to stall the pipeline. These blocking signals often become the driver of a high-fanout net, especially on pipelines that are deep in the number of physical stages and work on significant data sizes. Such high fanout nets, when they are created, are the prime cause of timing closure issues which cannot be fixed in RTL/Logic Synthesis or during Place-and-Route. To solve this issue, a new type of pipeline implementation called *free-running pipeline* (or `frp`) was created. The free-running pipeline is the most efficient architecture for handling a pipeline that operates with blocking signals. This is because

- It completely eliminates the blocking signal connections to the register enables
- It is a fully flushable pipeline, which allows bubbling invalid transactions
- Unlike the previous architectures which distribute fanouts (across flops), this reduces the fanouts
- It does not rely on the synthesis and/or place and route optimizations such as flop cloning
- This helps PnR by creating a structure where the wire length is reduced along with the reduction of the high fanouts

But there is a cost associated with this fanout reduction:

- the size of the FIFO buffers required for the blocking output ports causes additional resource usage.
- the mux delay at those blocking output ports
- the potential performance hit due to early validation of forward-pressure triggers

---

!! Important: The free-running pipeline (`frp`) can only be called from within a DATAFLOW region. The `frp` style cannot be applied to a loop that is called in a sequential or a pipelined region.

Pipeline Types

The three types of pipelines available in the tool are summarized in the following table. Vitis HLS will automatically select the right pipeline style to use for a given pipelined loop or function. If the pipeline is used with `hls::tasks`, the flushing pipeline (FLP) style is automatically selected to avoid deadlocks. If the pipeline control requires high fanout, and meets other free-running requirements, the tool will select the free-running pipeline (FRP) style to limit the high fanout. Finally, if neither of the above cases apply, then the standard pipeline (STP) style is selected.

Table: Pipelining Types

Name	Stalled Pipeline	Free-Running/ Flushable Pipeline	Flushable Pipeline
Use cases	• When there is no timing issue due to high fanout on pipeline control • When flushable is not required (such as no performance or deadlock issue due to stall)	• When you need better timing due to fanout to register enables from pipeline control • When flushable is required for better performance or avoiding deadlock • Can only be called from a dataflow region.	When flushable is required for better performance or avoiding deadlock.
Pragma/Directive	`#pragma HLS pipeline style=stp`	`#pragma HLS pipeline style=frp`	`#pragma HLS pipeline style=flp`
Global Setting	`config_compile -pipeline_style stp` (default)	`config_compile -pipeline_style frp`	`config_compile -pipeline_style flp`

Name	Stalled Pipeline	Free-Running/ Flushable Pipeline	Flushable Pipeline

Disadvantages	• Not flushable, hence it can:
	  ◦ Cause more deadlocks in dataflow
	  ◦ Prevent already computed outputs from being delivered, if the inputs to the next iterations are missing
	• Timing issues due to high fanout on pipeline controls
		• Moderate resource increase due to FIFOs added on outputs
		• Requires at least one blocking I/O (stream or ap_hs).
		• Not all pipelining scenarios and I/O types are supported.
			• Can have larger II
			• Greater resource usage due to less sharing when II>1

Advantages	• Default pipeline. No usage constraints.
	• Typically the lowest overall resource usage.
		• Better timing due to
		  ◦ Less fanout
		  ◦ Simpler pipeline control logic
		• Flushable
			• Flushable
			• Avoids deadlock

!! Important: Flushing pipelines (`flp`) are compatible with the rewind option specified in the PIPELINE pragma or directive when the `config_compile -enable_auto_rewind` option is also enabled.

Managing Pipeline Dependencies

Vitis HLS constructs a hardware datapath that corresponds to the C/C++ source code. When there is no pipeline directive, the execution is always sequential and so there are no dependencies that the tool needs to take into account. But when features of the design has been pipelined, the tool needs to ensure that any possible dependencies are respected in the hardware that Vitis HLS generates.
Typical cases of *data dependencies* or *memory dependencies* are when a read or a write occurs after a previous read or write.

- A read-after-write (RAW), also called a true dependency, is when an instruction (and data it reads/uses) depends on the result of a previous operation.
  - I1: t = a * b;
  - I2: c = t + 1;

  The read in statement I2 depends on the write of t in statement I1. If the instructions are reordered, it uses the previous value of t.
- A write-after-read (WAR), also called an anti-dependence, is when an instruction cannot update a register or memory (by a write) before a previous instruction has read the data.
  - I1: b = t + a;
  - I2: t = 3;

  The write in statement I2 cannot execute before statement I1, otherwise the result of b is invalid.
- A write-after-write (WAW) is a dependence when a register or memory must be written in specific order otherwise other instructions might be corrupted.
  - I1: t = a * b;
  - I2: c = t + 1;
  - I3: t = 1;

  The write in statement I3 must happen after the write in statement I1. Otherwise, the statement I2 result is incorrect.
- A read-after-read has no dependency as instructions can be freely reordered if the variable is not declared as volatile. If it is, then the order of instructions has to be maintained.

For example, when a pipeline is generated, the tool needs to take care that a register or memory location read at a later stage has not been modified by a previous write. This is a true dependency or read-after-write (RAW) dependency. A specific example is:

```
int top(int a, int b) {
 int t,c;
I1: t = a * b;
I2: c = t + 1;
 return c;
}
```

Statement `I2` cannot be evaluated before statement `I1` completes because there is a dependency on variable `t`. In hardware, if the multiplication takes 3 clock cycles, then `I2` is delayed for that amount of time. If the above function is pipelined, then Vitis HLS detects this as a true dependency and schedules the operations accordingly. It uses data forwarding optimization to remove the RAW dependency, so that the function can operate at II =1.
Memory dependencies arise when the example applies to an array and not just variables.

```
int top(int a) {
 int r=1,rnext,m,i,out;
 static int mem[256];
L1: for(i=0;i<=254;i++) {
#pragma HLS PIPELINE II=1
I1:     m = r * a; mem[i+1] = m;    // line 7
I2:     rnext = mem[i]; r = rnext; // line 8
 }
 return r;
}
```

In the above example, scheduling of loop `L1` leads to a scheduling warning message:

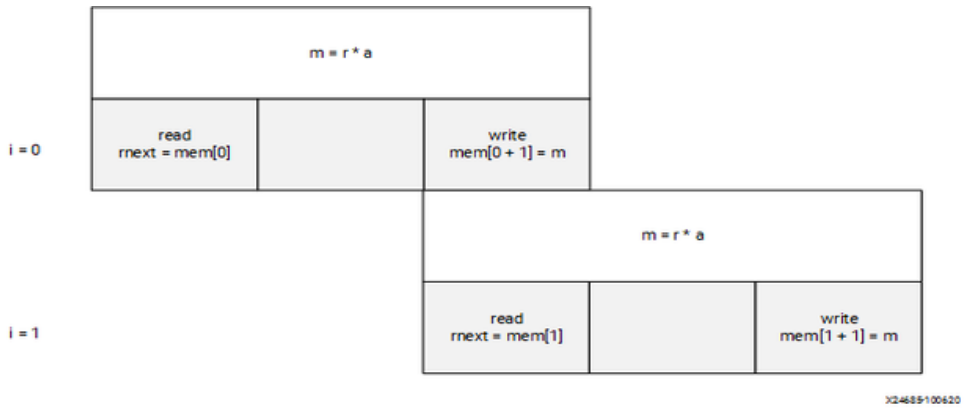
```
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
(II = 1, distance = 1) between 'store' operation (top.cpp:7) of variable 'm', top.cpp:7
on array 'mem' and 'load' operation ('rnext', top.cpp:8) on array 'mem'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

There are no issues within the same iteration of the loop as you write an index and read another one. The two instructions could execute at the same time, concurrently. However, observe the read and writes over a few iterations:

```
// Iteration for i=0
I1:     m = r * a; mem[1] = m;      // line 7
I2:     rnext = mem[0]; r = rnext; // line 8
// Iteration for i=1
I1:     m = r * a; mem[2] = m;      // line 7
I2:     rnext = mem[1]; r = rnext; // line 8
// Iteration for i=2
I1:     m = r * a; mem[3] = m;      // line 7
I2:     rnext = mem[2]; r = rnext; // line 8
```

When considering two successive iterations, the multiplication result `m` (with a latency = 2) from statement `I1` is written to a location that is read by statement `I2` of the next iteration of the loop into `rnext`. In this situation, there is a RAW dependence as the next loop iteration cannot start reading `mem[i]` before the previous computation's write completes.

Figure: Dependency Example



Note that if the clock frequency is increased, then the multiplier needs more pipeline stages and increased latency. This will force II to increase as well.
Consider the following code, where the operations have been swapped, changing the functionality:

```
int top(int a) {
 int r,m,i;
 static int mem[256];
L1: for(i=0;i<=254;i++) {
#pragma HLS PIPELINE II=1
I1:     r = mem[i];             // line 7
I2:     m = r * a , mem[i+1]=m; // line 8
 }
 return r;
}
```

The scheduling warning is:

```
INFO: [SCHED 204-61] Pipelining loop 'L1'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint (II = 1,
distance = 1)
 between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array 'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint (II = 2,
distance = 1)
 between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array 'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint (II = 3,
distance = 1)
 between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array 'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 4, Depth: 4.
```

Observe the continued read and writes over a few iterations:

```
Iteration with i=0
I1:     r = mem[0];           // line 7
I2:     m = r * a , mem[1]=m; // line 8
Iteration with i=1
I1:     r = mem[1];           // line 7
I2:     m = r * a , mem[2]=m; // line 8
Iteration with i=2
I1:     r = mem[2];           // line 7
I2:     m = r * a , mem[3]=m; // line 8
```

A longer II is needed because the RAW dependence is via reading `r` from `mem[i]`, performing multiplication, and writing to `mem[i+1]`.

Removing False Dependencies to Improve Loop Pipelining

False dependencies are dependencies that arise when the compiler is too conservative. These dependencies do not exist in the real code, but cannot be determined by the compiler. These dependencies can prevent loop pipelining.
The following example illustrates false dependencies. In this example, the read and write accesses are to two different addresses in the same loop iteration. Both of these addresses are dependent on the input data, and can point to any individual element of the `hist` array. Because of this, Vitis HLS assumes that both of these accesses can access the same location. As a result, it schedules the read and write operations to the array in alternating cycles, resulting in a loop II of 2. However, the code shows that `hist[old]` and `hist[val]` can never access the same location because they are in the else branch of the conditional `if(old == val)`.

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) f
  int acc = 0;
  int i, val;
  int old = in[0];
  for(i = 0; i < INPUT SIZE; i++)
  {
    #pragma HLS PIPELINE II=1
    val = in[i];
```

```
    if(old == val)
    {
        acc = acc + 1;
    }
    else
    {
        hist[old] = acc;
        acc = hist[val] + 1;
    }

    old = val;
  }

  hist[old] = acc;
```

To overcome this deficiency, you can use the DEPENDENCE directive to provide Vitis HLS with additional information about the dependencies.

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) {
  int acc = 0;
  int i, val;
  int old = in[0];
  #pragma HLS DEPENDENCE variable=hist type=intra direction=RAW dependent=false
  for(i = 0; i < INPUT SIZE; i++)
  {
    #pragma HLS PIPELINE II=1
    val = in[i];
    if(old == val)
    {
        acc = acc + 1;
    }
    else
    {
        hist[old] = acc;
        acc = hist[val] + 1;
    }

    old = val;
  }

  hist[old] = acc;
```

!! Important: Specifying a FALSE dependency, when in fact the dependency is not FALSE, can result in incorrect hardware. Be sure dependencies are correct (TRUE or FALSE) before specifying them.

When specifying dependencies there are two main types:

- Inter - Specifies the dependency is between different iterations of the same loop.If this is specified as FALSE it allows Vitis HLS to perform operations in parallel if the pipelined or loop is unrolled or partially unrolled and prevents such concurrent operation when specified as TRUE.
- Intra - Specifies dependence within the same iteration of a loop, for example an array being accessed at the start and end of the same iteration. When intra dependencies are specified as FALSE, Vitis HLS may move operations freely within the loop, increasing their mobility and potentially improving performance or area. When the dependency is specified as TRUE, the operations must be performed in the order specified.

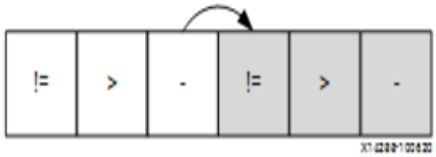
Scalar Dependencies

Some scalar dependencies are much harder to resolve and often require changes to the source code. A scalar data dependency could look like the following:

```
while (a != b) {
   if (a > b) a -= b;
   else b -= a;
 }
```

The next iteration of this loop cannot start until the current iteration has calculated the updated the values of `a` and `b`, as shown in the following figure.

Figure: Scalar Dependency



If the result of the previous loop iteration must be available before the current iteration can begin, loop pipelining is not possible. If Vitis HLS cannot pipeline with the specified initiation interval, it increases the initiation internal. If it cannot pipeline at all, as shown by the above example, it halts pipelining and proceeds to output a non-pipelined design.

Unrolling Loops

A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations might also be impacted by logic inside the loop body (for example, `break` conditions or modifications to a loop exit variable). You can unroll loops to create multiple copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel. Using the UNROLL pragma you can unroll loops to increase data access and throughput.
By default, HLS loops are kept rolled. This means that each iteration of the loop uses the same hardware. Unrolling the loop means that each iteration of the loop has its own hardware to perform the loop function. This means that the performance for unrolled loops can be significantly better than for rolled loops. However, the added performance comes at the expense of added area and resource utilization.
Consider the basic_loops_primer example from GitHub, as shown below:

```
#include "test.h"

dout_t test(din_t A[N]) {
  dout_t out_accum=0;
```