Chapter 4
Fundamentals of High Level Synthesis

Introduction

One of the common misconceptions held by people is that synthesizing hardware from C++ provides users the freedom of expressing their algorithms using any style of C++ coding that they desire. When designing using high-level C++ synthesis, it is important to remember that we are still describing hardware using C++, and a "poor" description can lead to a sub-optimal RTL implementation. It is the responsibility of the user to code their C++ to not only account for the underlying memory architecture of a design, but to also adhere to the recommended coding style for C++ synthesis. Because of this it is important to have a solid understanding about what high level synthesis really does. This chapter attempts to cover the basics of high level synthesis, and to show what designers can expect from a given coding style. Where appropriate, the code examples are accompanied by hardware diagrams to hopefully allow RTL designers to relate C++ synthesis to concepts that are familiar to them.

The Top-level Design Module

Similar to RTL design, HLS requires that users specify where the "top" of their design is. This is where the design interfaces with the outside world and consists of port definitions, direction, and bit widths or in the case of C++, data types. Since we are still designing hardware, although using C++, it is helpful to see how this might relate to the world of RTL design. Consider the following simple Verilog RTL design that describes a d-type flip flow with asynchronous reset.
Example 4-1. Simple Verilog Design Module

```verilog
module top(clk, arst, din, dout);

input clk;
input arst;
input [31:0] din;
output [31:0] dout;
reg [31:0] dout;

always@(posedge clk or posedge arst)
begin
    if(arst == 1'b1)
        dout = 1'b0;
    else
        dout = din;
end
endmodule
```

The verilog module shown in Example 4-1 contains several input ports and a single output port. The inputs are for clock, reset, and 32-bit input data, the output is for the 32-bit output data. The port directions and widths are explicitly defined in the Verilog source.

In order for HLS to determine the top-level design from the C++, the user must either specify a pragma in the source code or set a user constraint in the synthesis tool.

So the C++ equivalent description of the Verilog design described above would look like:

Example 4-2. Setting the Top-level Design

```cpp
void top(int din, int& dout){
    dout = din;
}
```

Even this simple example illustrates how HLS can simplify the design process. The C++ description is very compact. Looking at the C++ description in Example 4-2 it is important to understand that there are several things that are implied in the code.

Registered Outputs

High-level synthesis by default builds synchronous designs. This means that all outputs of the top-level design are registered to guarantee that timing is met when connecting to another design. There are mechanisms to build smaller combinational blocks (automated component flows) but in general designs are synchronous.
Control Ports

The C++ code has no concept of timing so there are no clocks, enables, or resets described in the source. Instead these signals are added by the synthesis tool. Control over things like polarity, type of reset, etc, are taken care by setting design constraints.

Port Width

For the simple case, meaning minimal interface constraints in synthesis, the bit widths of the top-level ports, excluding clock and reset, are implied by the data type. In the design example shown above the data type is "int" which implies 32 bits. Designers can describe any arbitrary bit width using bit-accurate data types.

Port Direction

The port direction is implied by how an interface variable is used in the C++ code

Input ports

An input port is inferred when an interface variable is only read. In the C++ example shown above you can see that "din" is only ever read, so it is determined to be an input. If a variable is declared on the interface as "pass by value" it can only be an input. This is covered in more detail later.

Output ports

An output port is inferred in two cases. One is when the top-level function returns a value. The other is when the interface variable is only written in the C++ code. In the C++ example design it can be seen that "dout" is only ever written. It can also be seen that "dout" is declared as a reference. A variable must be declared as a reference or a pointer in order to infer an output. This is also covered in more detail later.

Inout Ports

Although this design does not contain any inout ports, these are inferred if an interface variable is both read and written in the same design. This requires that the variable is declared as a reference or a pointer.

High-level C++ Synthesis

Although this style guide is not intended to be a tutorial on the intricacies of high-level synthesis optimizations, it is useful to present a brief overview of the basic process of automatically transforming un-timed algorithmic C++ into hardware. Understanding these fundamental concepts goes a great way towards providing a solid foundation for the material covered in later sections.
Similar to the rest of the style guide, these concepts are best illustrated using simple examples consisting of both C++ code and hardware design concepts. Consider the following C++ example that accumulates four integer values shown in Example 4-3.

**Example 4-3. Simple C++ Accumulate**

```c++
#include "accum.h"
void accumulate(int a, int b, int c, int d, int &dout){
   int t1,t2;
   t1 = a + b;
   t2 = t1 + c;
   dout = t2 + d;
}
```

**Data Flow Graph Analysis**

The process of high-level synthesis starts by analyzing the data dependencies between the various steps in the algorithm shown above. This analysis leads to a Data Flow Graph (DFG) description shown in Figure 4-1.

![Data Flow Graph Description](image)

Each node of the DFG represents an operation defined in the C++ code, for this example all operations use the "add" operator. The connection between nodes represents data dependencies and indicates the order of operations. This example shows that t1 must be computed before t2 [1].

**Resource Allocation**

Once the DFG has been assembled, each operation is mapped onto a hardware resource which is then used during scheduling. This is the process known as resource allocation. The resource corresponds to a physical implementation of the operator hardware. This implementation is annotated with both timing and area information which is used during scheduling. Any given operator may have multiple hardware resource implementations that each have different area/delay/latency trade-offs. The resources are selected from a technology specific pre-characterized library that contains sufficient data points to represent a wide range of bit widths and clock frequencies. Figure 4-2 shows the resource allocation of Figure 4-1. Each operation can potentially be allocated to a different resource. It is also typical that designers can explicitly control resource allocation to insert pipeline registers or limit the number of available resources.
Scheduling

High-level synthesis adds "time" to the design during the process known as "scheduling". Scheduling takes the operations described in the DFG and decides when (in which clock cycle) they are performed. This has the effect of adding registers between operations based on a target clock frequency. This is similar to what RTL designers would call pipelining, by which they mean inserting registers to reduce combinational delays. This is not the same as "loop pipelining" which is discussed later.

If we assume that the "add" operation for the DFG of Figure 4-1 takes 3 ns out of a 5 ns clock cycle, the resulting schedule would look something like the schedule shown in Figure 4-3. Each add operation is scheduled in its own clock cycle C1, C2, and C3. Thus registers are inserted automatically between each adder.

A data path state machine (DPFSM) is created to control the scheduled design. The FSM for this example requires four states that correspond to the four clock cycles needed to execute the schedule shown above. In HLS these state are also referred to as control steps or c-steps.
The state diagram of the DPFSM is shown in Figure 4-4 and illustrates that the scheduled design is capable of producing a new output every four clock cycles. Once C4 has completed a new C1 begins.

The resulting hardware that is generated from the schedule shown in Figure 4-3 varies depending on how the design was constrained in terms of resource allocation as well as the amount of loop pipelining used on the design. Loop pipelining is covered in detail next but for now let's assume that the design is unconstrained, which should allow the hardware to be realized with the minimum number of resources if sharing saves area. In this example that would be the minimum number of adders. The resulting hardware would look something like that shown in Figure 4-5.

The resulting hardware shown in Figure 4-5 uses a single adder to accumulate a, b, c, and d. It should be noted that the data path is 32-bits wide because the variables have been declared as integer types.
Classic RISC Pipelining

The HLS concept of "Loop Pipelining" is similar to the classic RISC pipeline covered in most introductory computer architecture classes.

The basic five stage pipeline in a RISC architecture typically consists of Instruction Fetch(IF), Instruction Decode(ID), Execute(EX), Memory access(MA), and Register write back(WB) stages.

![Figure 4-6. Five Stage RISC Pipeline](image)

Figure 4-6 illustrates how a new instruction can be fetched each clock cycle while the other pipeline stages are gradually activated. The time it takes for all pipeline stages to become active is known as the pipeline **"ramp up"**. Once all pipeline stages are active the pipeline **"ramp down"** is the time it takes for all pipeline stages to become inactive. The difference between the RISC pipeline and HLS loop pipelining is that the RISC pipeline is designed to fetch and execute every clock cycle. A design that does not need to run every clock cycle under-utilizes the pipeline, and a design that needs to fetch and execute multiple times per clock cycle is not possible. HLS removes these restrictions and allows the pipeline to be custom built to meet the design specification.

Loop Pipelining

Similar to the RISC pipelining example described in Figure 4-6, which allows new instructions to be read before the current instruction has finished, "Loop Pipelining" allows a new iteration of a loop to be started before the current iteration has finished. Although in Example 4-3 there are no explicit loops, the top-level function call has an implied loop, also known as the main loop. Each iteration of the implied loop corresponds to execution of the schedule shown in Figure 4-3 on page 39. "Loop pipelining" allows the execution of the loop iterations to be overlapped, increasing the design performance by running them in parallel. The amount of
overlap is controlled by the "Initiation Interval (II)". This also determines the number of pipeline stages

**Note**

The *Initiation Interval (II)* is how many clock cycles are taken before starting the next loop iteration. Thus an II=1 means a new loop iteration is started every clock cycle.

The initiation interval is set on a desired loop either as a design constraint in the HLS design environment, or alternatively can be set using a C++ compiler pragma.

**Note**

*Latency* refers to the time, in clock cycles, from the first input to the first output

**Note**

*Throughput*, not to be confused with IO throughput, refers to how often, in clock cycles, a function call can complete.

If the design of Example 4-3 on page 38 is left unconstrained there is only a single pipeline stage because there’s no overlap between execution of each iteration of the main loop. This results in data written every four clock cycles (Figure 4-7). The design has a latency of three and a throughput of four clock cycles. Because there is no overlap of any operation only a single adder is required if sharing reduces overall area.

**Figure 4-7. No Pipelining, L=3, TP=4**

If a pipelining constraint of II=3 is applied on the top-level design (main loop), then the next loop iteration can be started in C4 allowing writing of "dout" in C4 to be overlapped with the reading of the next values of "a" and "b". The output is now written every three clock cycles while still requiring only one adder to implement the hardware (Figure 4-8). Only one pipeline stage is required since C4 is only used to allow the completion of the write on “dout”. 
Note

The number of pipeline stages increases by one if other operations are scheduled in the last c-step.

**Figure 4-8. Pipeline II=3, L=3, TP=3**

Figure 4-9 shows that pipelining with an II=2 results in a new iteration started every two clock cycles. Iteration one is started in C3 while iteration 0 is computing \( t_3 = t_2 + d \). Since iteration one is computing \( t_1 = a + b \) it can be seen that two adders are required for the two pipeline stages.

**Figure 4-9. Pipeline II=2, L=3, TP=2**

Pipelining with an II=1 (Figure 4-10) results in a new iteration started every clock cycle. Iteration one is started in C2 and iteration 2 is started in C3. Looking at C3 in the Figure 4-10 shows that three adders are required in hardware since all three pipeline stages are active.
Loops

One of the most important features of HLS for tuning design performance is Loop Unrolling. However, it is necessary first to discuss what constitutes a “loop” in C++. Loops are the primary mechanism for applying high level synthesis constraints as well as moving data, or IO, into and out of an algorithm. The style in which loops are written can have a significant impact on the quality of results of the generated hardware. In order to talk about how to write loops it’s helpful to introduce a few definitions:

- **Interface synthesis** - the process of mapping top-level C++ variables to resources that implement an interface protocol (wire, handshake, memory).

- **Loop iterations** - the number of times the loop runs before it exits.

- **Loop iterator** - the variable used to compute the loop iteration.

- **Loop body** - the code between the start and the end of the loop.

- **Loop unrolling** - the number of times to copy the loop body.

- **Loop pipelining** - how often to start the next iteration of the loop.
What's in a Loop?

In HLS a design always has one loop which corresponds to the top-level function call. This is known as the “main loop” (See line 1 of Example 4-4).

Example 4-4. The Main Loop

```c++
void top(int din, int& dout){
    dout = din;
}
```

The "main loop" is a continuously running loop, which means that it runs for an infinite number of iterations. The analog to this can be seen from the equivalent Verilog module implementation that was shown in the top-level interface section. Once that Verilog module is reset it runs forever as long as the clock is supplied.

There are three ways to specify a loop in C++; using the “for” loop, “while” loop, and “do-while” loop. The syntax is as follows:

"for" Loop

Syntax:

```
LABEL: for( initialization; test-condition; increment ) {
    statement-list or loop body;
}
```

The “for” construct is a general looping mechanism consisting of 4 parts:

1. **initialization** - which consists of zero or more comma-delimited variable initialization statements
2. **test-condition** - which is evaluated to determine if the execution of the for loop continues
3. **increment** - which consists of zero or more comma-delimited statements that increment variables
4. **statement-list** - which consists of zero or more statements that execute each time the loop is executed.
Example 4-5. “for” Loop

```c
#include "simple_for_loop.h"
void simple_for_loop(int din[4], int dout[4]){
    FOR_LOOP:for(int i=0; i<4; i++){
        dout[i] = din[i];
    }
}
```

The example "for" loop shown in Example 4-5 copies four 32-bit values from din to dout. The for loop has initialization "int i=0", test condition "i<4", and increment "i++".

"while" Loop

The "while" keyword is used as a looping construct that executes the loop body as long as condition is tested as true. If the condition starts off as false, the loop body is never executed. (You can use a do loop to guarantee that the loop body executes at least once.)

Syntax:

```c
LABEL: while(test-condition) {
    statement-list or loop body;
}
```

Example 4-6. “while” Loop

```c
#include "simple_while_loop.h"
void simple_while_loop(int din[4], int dout[4]){
    int i=0;
    WHILE_LOOP:while(i<4){
        dout[i] = din[i];
        i++;
    }
}
```

The "while" loop shown in Example 4-6 has the same functionality of the previous "for" loop example.

"do" Loop

The "do" keyword is used as a looping construct that executes the loop body until the condition is tested as false. The loop body always executes at least once.

Syntax:

```c
LABEL: do{
    statement-list or loop body;
} while(test-condition);
```
Example 4-7. “do” Loop

```c
#include "simple_do_loop.h"
void simple_do_loop(int din[4], int dout[4]){
    int i=0;
    DO_LOOP: do{
        dout[i] = din[i];
        i++;
    }while(i<4);
}
```

The "do" loop shown in Example 4-7 has the same functionality as the previous "for" and "while" loop examples.

Rolled Loops

**Note**

If a loop is left “rolled”, each iteration of the loop takes at least one clock cycle to execute in hardware. This is because there is an implied “wait until clock” for the loop body.

Consider the following C++ example that uses a “for” loop to accumulate four 32-bit integers from an array:

Example 4-8. C++ Accumulate Using Loops

```c
void accumulate4(int din[4], int &dout){
    int acc=0;
    ACCUM: for(int i=0;i<4;i++){
        acc += din[i];
    }
    dout = acc;
}
```

**Design Constraints**

Main loop pipelined with II=1
All loops left rolled

Although the loop is left rolled notice that the design has been pipelined with an II=1. This was done intentionally in order to ignore the effects of the extra clock cycle required for allowing the write of “dout” to complete, as it was discussed in “Loop Pipelining” on page 41. The effects of pipelining loops is covered in more detail in later sections. Figure 4-11 shows the schedule of the loop iterations for Example 4-8.
Figure 4-11. Schedule for Accumulate Using Loops

Figure 4-11 shows that each call to the “accumulate” function requires four clock cycles to accumulate the four 32-bit values in Example 4-8. This is because the loop has been left rolled and there is an implied “wait until clock” at the end of the loop body. The synthesized hardware would have the approximate structure shown in Figure 4-12.

Figure 4-12. Hardware Implementation - Accumulate Using Loops

Figure 4-12 has a structure similar to what one might expect from a hand-code RTL design. However, one important feature to note is that the control logic for this implementation is three bits wide. The reason for this is that the loop exit condition is “i<4”. This means that this loop only exits when “i>=4”, which requires at least three bits.

Note

The number of bits required for evaluating the loop exit condition is usually one bit larger than expected. This is because the loop iteration must increment before exiting.

Loop Unrolling

Loop unrolling is the primary mechanism to add parallelism into a design. This is done by automatically scheduling multiple loop iterations in parallel, when possible. The amount of parallelism is controlled by how many loop iterations are run in parallel. This is different than loop pipelining, which allows loop iterations to be started every II clock cycles. Loop unrolling
can theoretically execute all loop iterations within a single clock cycle as long as there are no dependencies between successive iterations.

Partial Loop Unrolling

If we take Example 4-8 and unroll the ACCUM loop by a factor of two, this has the equivalent effect of manually duplicating the loop body two times and running the ACCUM loop for half as many iterations. Example 4-9 illustrates the effects of loop unrolling by showing the ACCUM loop of Example 4-8 manually unrolled two times.

**Example 4-9. Manual Loop Unrolling - Unroll by 2**

```c
1  void accumulate(int din[4], int &dout){
2    int acc=0;
3    ACCUM:for(int i=0;i<4;i+=2){
4       acc += din[i];
5       acc += din[i+1];
6    }
7    dout = acc;
8 }
```

The details of Example 4-9 are:

- line 3 increments the ACCUM loop by two, which means that the “partially unrolled” loop now has two iterations.
- Lines 4 and 5 have duplicated the loop body two times, which shows that two accumulations are performed each iteration. It should be noted that the accumulate in Line 5 is dependent on the accumulate on line 4. For now it is assumed that there is still sufficient time to schedule both in the same clock cycle. Dependencies between loop iterations are discussed later.

Figure 4-13 shows the schedule of the loop iterations for Example 4-8 on page 47 when the ACCUM loop is unrolled by two. All four values are now accumulated in only two clock cycles.
Figure 4-14 shows the hardware implementation when unrolling by two. It can be seen that this design requires twice as many resources (adders) as the “rolled” version.

**Figure 4-14. Hardware Implementation - Accumulate Unroll by 2**

![Figure 4-14](image)

**Fully Unrolled Loops**

Taking Example 4-8 on page 47 and “fully” unrolling the ACCUM loop dissolves the loop and allows all iterations to be scheduled in the same clock cycle (Assuming that there is sufficient time to account for dependencies between iterations). The manual equivalent C++ of doing this is shown in Example 4-10.

**Example 4-10. Manual Loop Unrolling - Fully Unrolled**

```cpp
void accumulate(int din[4], int &dout){
    int acc=0;
    acc += din[0];
    acc += din[1];
    acc += din[2];
    acc += din[3];
    dout = acc;
}
```

Figure 4-15 shows the schedule of the fully unrolled ACCUM loop. All four values are now accumulated in a single clock cycle.

**Figure 4-15. Schedule for Accumulate - Fully Unrolled**

![Figure 4-15](image)
Figure 4-16 shows the approximate hardware when fully unrolling the ACCUM loop.

**Figure 4-16. Hardware Implementation - Accumulate with Fully Unrolled Loop**

![Diagram of hardware implementation](image)

**Dependencies Between Loop Iterations**

**Note**

Unrolling a loop does not necessarily guarantee that the loop iterations are scheduled in the same c-step. Dependencies between iterations can limit parallelism.

The previous examples have assumed that there is sufficient time to ignore the effects of any dependencies between loop iterations. Thus Figure 4-15 shows all four iterations scheduled in the same clock cycle, but it does not show the dependencies that exist between iterations. A more accurate depiction of the schedule that includes the dependencies and component delays is shown in Figure 4-17. If the adders in Figure 4-17 were sufficiently slow it would be likely that second stage of the adder tree would be scheduled in the next clock cycle, increasing the design latency. However, if the design is pipelined with II=1 it is still possible to achieve a throughput of accumulating four values per clock cycle. Thus some dependencies between loop iterations do not limit design performance. However in many cases the dependencies between iterations limit performance or prevent pipelining. This is covered in detail in “Data Feedback” on page 73.

**Figure 4-17. Schedule Dependencies**

![Diagram of schedule dependencies](image)

**Loops with Constant Bounds**

When writing loops for HLS it is important, when possible, to express them such that there is:

1. A constant initialization of the loop iterator
2. A test condition of the loop iterator against a constant value
3. A constant increment of the loop iterator

Writing the loop in this fashion allows HLS to optimize the design by reducing the bit widths of control and data path signals that are based on the loop iterator. This is because the three conditions listed above are sufficient for determining the maximum number of loop iterations. This is desirable to be able to get accurate information about latency and throughput of a design.

The four cycle accumulator in Example 4-8 on page 47 is a good example of writing loops with constant bounds. The corresponding hardware implementation shown in Figure 4-12 on page 48 shows that the control logic is optimally reduced to three bits. The main point to take away from this example is that even though the loop iterator "i" was declared as a 32-bit integer, HLS is able to reduce the bit widths to the fewest possible bits because the loop was written with constant bounds.

**Loops with Conditional Bounds**

The previous section showed that optimal hardware can be inferred if a loop is written with unconditional bounds. However, it is often the case that an algorithm or design requires that a loop terminate early base on some variable that has been defined outside of the loop, or on the design interface. This is a perfectly reasonable thing to do, but the way this is written in the C++ code can have a dramatic impact on the quality of results as well as accurate reporting of latency and throughput.

The accumulator design used in Example 4-8 can be modified illustrate the impact in quality-of-results when using a loop with conditional bounds. In order to make the accumulator more programmable the code is modified so that the accumulator can accumulate anywhere from one to four 32-bit values.

**Example 4-11. Conditional Accumulate**

```c++
1  #include "accum.h"
2  #include <ac_int.h>
3  void accumulate(int din[4], int &dout, unsigned int ctrl){
4    int acc=0;
5  ACCUM:for(int i=0;i<ctrl;i++){
6     acc += din[i];
7   }
8    dout = acc;
9  }  
10}
```

The modified accumulator design shown above now uses the interface variable "ctrl" on line 5 to select the number of loop iterations to be one through four. Synthesizing this design reveals that there are several inefficiencies with the resulting hardware.
Caution

Having a variable as the loop upper or lower bound often results in the loop counter hardware being larger than needed

Caution

Having a variable as the loop upper bound requires one extra clock cycle to test the loop condition

Caution

Having an unconstrained bit width on the loop exit condition results in control logic larger than needed

Let's first examine the hardware that would be synthesized from the conditional accumulator shown in Example 4-11:

![Figure 4-18. Loop With Conditional Bounds](image)

The resulting hardware synthesized from the C++ accumulator with conditional loop bounds results in a 32-bit loop counter and 33-bit logic for the loop exit condition (Figure 4-18). The reason for this is that the interface variable "ctrl" is a 32-bit integer. Because "ctrl" is on the design interface, HLS has no way of knowing, or more importantly proving, that it only should ever range from one to four.

Note

This is an important lesson about HLS in that it only automatically reduces bit-widths where it can symbolically prove that it can be done without changing the functionality between the C++ code and the generated RTL.

In this example, the C++ specifies a 32-bit interface variable which requires 33-bit control logic to be functionally equivalent. The solution to the problem shown above requires two minor C++ code changes, and can be split into two parts; fixing the loop counter, and fixing the loop exit condition.
Optimizing the Loop Counter

In order for HLS to reduce the bit width of the loop counter the loop upper bound should be set to a constant. However, since the execution of each loop iteration is determined by the variable, "ctrl", we need to add a mechanism for terminating the loop early. This is done by using a conditional break in the loop body shown in Example 4-12.

Example 4-12. Bounded Loop with Conditional Break

```c
#include "accum.h"
#include <ac_int.h>
void accumulate(int din[4], int &dout, int ctrl){
  int acc=0;
  ACCUM:for(int i=0;i<4;i++){
    acc += din[i];
    if(i>=ctrl-1)
      break;
  }
  dout = acc;
}
```

The conditional break is placed at the end of the loop because it is assumed that there is at least one loop iteration. Having the conditional break at the end of the loop should give the best quality of results in general. However, if “ctrl” can be zero, meaning that the loop can have zero iterations, the break must be placed at the beginning of the loop body. Figure 4-19 shows the resulting hardware from Example 4-12.

Figure 4-19. Bounded Loop With Conditional Break

The code transformation has the effect of reducing the loop counter to three bits by fixing the upper loop bound to a constant. Unfortunately, the code change has actually made the design slightly larger. Putting the conditional break at the end of the loop has created a 33-bit subtractor to compute "ctrl-1" and a 34-bit subtractor to compute the ">=" operation. This is in part because “ctrl” is 32-bits and cannot be automatically reduced since it is on the design interface. The control logic can be further optimized.
Optimizing the Loop Control

There are two problems with the control logic for the loop exit condition of Example 4-12:

- Use of a 32-bit integer on the design interface
- Exit condition test requires a subtractor to compare against ctrl-1

HLS is not able to reduce the bit-widths on the top-level design interface for this design since it cannot prove that "ctrl" is always between one and four. In this case the designer must constrain the bit-width of "ctrl" to the desired number of bits. Native C++ data types do not give designers the ability to specify arbitrary bit widths on a variable so bit-accurate data types are required. A better way to write the code to optimize the loop control is shown in Example 4-13.

Example 4-13. Optimized Loop Control

```cpp
#include "accum.h"
#include <ac_int.h>
void accumulate(int din[4], int &dout, ac_int<3,false> ctrl){
  int acc=0;
  int i_old=0;
  ACCUM:for(int i=0;i<4;i++){
    acc += din[i];
    if(i_old==ctrl)
      break;
    i_old = i;
  }
  dout = acc;
}
```

The following code changes were made to optimize the loop control logic:

1. Line 3 - “ctrl” was constrained to three bits reducing the comparison logic to three bits
2. Line 10 - “i_old” stores the previous value of the loop iterator “i”
3. Line 8 - The exit condition test is made on the previous value of “i” eliminating the need for a subtractor.

The resulting hardware from Example 4-13 is shown in Figure 4-20.
### Nested Loops

Nested loops and the effects of pipelining nested loops is often one of the most misunderstood concepts of high-level C++ synthesis. Understanding the resulting hardware behavior from synthesizing non-pipelined and pipelined nested loops allows designers to more easily meet performance and area requirements. The simple accumulator that has been used in previous examples can be extended to illustrate the effects of nested loops.

#### Example 4-14. Nested Loop Accumulator

```cpp
#include "accum.h"
#include <ac_int.h>
#define MAX 100000
void accumulate(int din[2][4], int &dout){
    int acc=0;
    ROW:for(int i=0;i<2;i++){
        if(acc>MAY)
            acc = MAX;
        COL:for(int j=0;j<4;j++){
            acc += din[i][j];
        }
    }
    dout = acc;
}
```

The following enhancements to the C++ accumulator designer were made:

1. Line 4- The input data is a 2x4 array of type integer.
2. Lines 6 and 10 - Two loops, ROW and COL are nested to index the rows and columns of the 2x4 array.
3. Lines 7 and 8 - The accumulate variable "acc" is saturated to keep it from exceeding a maximum value at the beginning of the ROW loop. This is somewhat of an artificial example but helps illustrate the effects of nesting loops.

**Unconstrained Nested Loops**

If the nested loop accumulator in Example 4-14 is synthesized with both loops left rolled and no loop pipelining, the resulting hardware has a behavior similar to the state diagram shown in Figure 4-21. Note that for this example it is assumed that each iteration of the COL loop can execute in one clock cycle.

**Figure 4-21. State Diagram of Unconstrained Nested Loops**

The state diagram in Figure 4-21 shows that unconstrained nested loops have an overhead associated with the computation of the outer loop body and index (C1_ROW and C2_ROW). This overhead has the impact of increasing the latency of the design. This increased latency can be substantial compared to the number of clock cycles required to perform the main computation of the algorithm. Figure 4-21 shows that the execution of the design requires two c-steps (clock cycles) for the main loop, and two c-steps for the ROW loop, in addition to the eight cycles required to execute the COL loop twice. This means that the entire design takes 14 cycles to accumulate the eight values of din[2][4], which in turn means that 43% of the execution time is taken up by the main and ROW loop overhead in this example. Figure 4-22 shows the schedule for the unconstrained design.
Unconstrained nested loops can increase latency because of the overhead of computing the loop exit conditions and loop bodies separately.

**Pipelined Nested Loops**

Loop pipelining can be applied in order to improve design performance.

It is generally good practice to begin pipelining starting with the innermost loops and working up towards the top-most loops. This should in general give the best area/performance trade-off.

For this example the innermost loop is the COL loop. However since it was assumed that each iteration of the COL loop only requires one c-step to execute there is no benefit in pipelining this loop.

**Pipelined ROW Loop With II=1**

When nested loops are pipelined together the loops are “flattened” into a single loop. The initiation interval constraint is then applied to the flattened loop.

Figure 4-23 shows the state diagram that illustrates the effect of pipelining the ROW and COL loops for Example 4-14.
Figure 4-23. State Diagram of ROW Loop with II=1

Figure 4-23 shows how loop flattening removes the overhead of the C1_ROW and C2_ROW states by combining the saturation and ROW loop index logic into the same loop with the COL loop. Although pipelining nested loops improves performance in terms of latency and throughput, it is not without cost. The control logic become progressively more complex as more and more nested loops are pipelined. This can lead to larger area, or failure to schedule in some cases. So a good rule of thumb is to start pipelining the inner loops and work your way towards the outer loops until the performance target is met. Figure 4-24 shows the schedule when the ROW loop is pipelined with II=1.

Figure 4-24. Schedule of ROW Loop with II=1
Figure 4-24 shows that by pipelining the ROW and COL loops together, the two cycle overhead of the ROW loop has been absorbed into the flattened loop allowing the nested ROW and COL loops to execute in eight clock cycles. The only overhead remaining is caused by the main loop.

**Pipelined main Loop with II=1**

Similar to pipelining the ROW loop, pipelining the main loop causes the main, ROW, and COL loops to be flattened into a single loop. This has the effect of moving the loop iterator initialization and the write of the output “dout” into the ROW_COL loop and executing them conditionally, as shown in the state diagram of Figure 4-25. The net result is to increase the design performance at the expense of making the control logic more complicated. Figure 4-26 shows the schedule with the main loop pipelined. The two cycle overhead of the main loop has been flattened along with the ROW and COL loops allowing the design to achieve maximum performance.
Unrolling Nested Loops

Loop unrolling can be applied to nested loops to increase the design performance, often at the expense of larger area. Because of this, designers must be methodical in choosing how much to unroll a loop. For nested loops with a large number of iterations it is more commonplace to leave the outer loop(s) rolled and partially or fully unroll the inner loop when trying to increase design performance. This is also usually done in combination with loop pipelining.

Note

In general it is always better to pipeline loops first before using loop unrolling. This is because loop pipelining often gives a significant boost in performance with a smaller cost in terms of area.

Loop unrolling on the other hand usually has a greater impact on area when the loop body contains a large number of operations. This is because unrolling replicates the loop body leading to larger numbers of resources being scheduled in parallel.

Unrolling the Innermost Loop

Example 4-15 shows a C++ design that uses two nested loops to separately accumulate the rows of a two-dimensional array. This example is synthesized with the COL loop fully unrolled and the ROW loop pipelined with II=1.
Fundamentals of High Level Synthesis

Example 4-15. Unrolling the Inner Loop

```
#include "accum.h"
void accumulate(int din[2][4], int dout[2]){
    int acc[2];
    ROW:for(int i=0;i<2;i++){
        acc[i] = 0;
        COL:for(int j=0;j<4;j++){
            acc[i] += din[i][j];
        }
        dout[i] = acc[i];
    }
}
```

Fully unrolling the COL loop has the same effect as manually replicating the COL loop body, shown in Example 4-16.

Example 4-16. Unrolling the Inner Loop Manually

```
#include "accum.h"
void accumulate(int din[2][4], int dout[2]){
    int acc;
    ROW:for(int i=0;i<2;i++){
        acc=0;
        acc += din[i][0];
        acc += din[i][1];
        acc += din[i][2];
        acc += din[i][3];
        dout[i] = acc;
    }
}
```

Example 4-16, which shows the effects of duplicating the inner loop body is transformed during scheduling into something that more closely resembles the code shown in Example 4-17.

Example 4-17. Unrolling the Inner Loop Transformation

```
#include "accum.h"
void accumulate(int din[2][4], int dout[2]){
    int acc=0;
    ROW:for(int i=0;i<2;i++){
        dout[i] = din[i][0]+din[i][1]+din[i][2]+din[i][3];
    }
}
```

Example 4-17 shows that accumulating four values at a time requires three adders. Assuming that there is sufficient time to schedule the three adders in the same clock cycle, the design schedule looks like that shown in Figure 4-27. Each iteration of the ROW loop executes in one clock cycle, while there is still some overhead caused by not pipelining the main loop.
The hardware resulting from synthesizing Example 4-15 is shown in Figure 4-28. High-level synthesis automatically builds a balanced adder tree when unrolling accumulators inside a loop. There are some situations where the tree balancing does not happen automatically when the accumulate is conditional. This is discussed later.

**Rampup/Rampdown of Pipelined Nested Loops**

Increasing the clock frequency when synthesizing Example 4-15 at some point requires that the adder tree shown in Figure 4-28 be scheduled over multiple clock cycles or c-steps. Figure 4-29 shows the design schedule where the first two adders are scheduled together in the same c-step, with the second adder stage scheduled in the next c-step. Two pipeline stages are created when the ROW loop is pipelined with II=1, and the design latency and throughput is affected due to pipeline rampup and rampdown, initially discussed in “Classic RISC Pipelining” on page 41. For loops with large number of iterations, the effect of rampup/rampdown may be negligible, and allowing the pipeline to rampdown has the added benefit of allowing all data to be “flushed” from the pipeline stages. In this example the cost of rampup/rampdown is significant compared to the number of iterations for the ROW loop.
Figure 4-29. Schedule of Ramp-up/down with Inner Loop Fully Unrolled

Figure 4-30 shows the hardware generated for the schedule shown in Figure 4-29. The adder tree has been separated into two pipeline stages.

Figure 4-30. Hardware of Ramp-up/down with Inner Loop Fully Unrolled

Rampup Only of Nested Loops with Pipelined Main Loop

A possible solution for increasing the performance for designs that have both rampup and rampdown of the pipeline would be to pipeline the “main” loop with II=1. When this is done the pipeline only ramps up and then runs forever, removing the throughput cost of pipeline rampdown. This is shown in Figure 4-31.
Caution

There are side effects associated with pipelining the main loop when the design has rolled loops. If IO is mapped to a handshaking interface and is accessed inside of the pipelined loop it can cause the pipeline to stall. This is covered in “Conditional IO” on page 90.

Unrolling the Outer Loop

The previous section illustrated how unrolling the innermost loop replicates the loop body resulting in higher performance. The core architectural feature resulting from unrolling the innermost loop was a balanced adder tree, Figure 4-28. If the inner loop is left rolled and the outer loop is unrolled the inner loop is replicated as many times as the loop is unrolled. Example 4-18 shows the effects of manually unrolling the outer loop where there are now two copies of the inner loop, COL_0 and COL_1.
Example 4-18. Manually Unrolling the Outer Loop

```
#include "accum.h"
void accumulate(int din[2][4], int dout[2]){
    int acc[2];
    acc[0] = 0;
    COL_0:for(int j=0;j<4;j++){
        acc[0] += din[0][j];
    }
    dout[0] = acc[0];
    acc[1] = 0;
    COL_1:for(int j=0;j<4;j++){
        acc[1] += din[1][j];
    }
    dout[1] = acc[1];
}
```

When possible, high-level synthesis automatically merges all of the replicated loops into a single loop, leading to a number of accumulators running in parallel. Example 4-19 shows the effects of manually merging the two COL loops.

Example 4-19. Manual Merging

```
#include "accum.h"
void accumulate(int din[2][4], int dout[2]){
    int acc[2];
    acc[0] = 0;
    acc[1] = 0;
    COL_0_1:for(int j=0;j<4;j++){
        acc[0] += din[0][j];
        acc[1] += din[1][j];
    }
    dout[0] = acc[0];
    dout[1] = acc[1];
}
```

Figure 4-32 shows the schedule when the ROW is fully unrolled and all copies of the COL loop are merged.
Figure 4-32. Unrolling the Outer Loop with Loop Merging

Figure 4-33 shows the synthesized hardware resulting from unrolling the outer loop which has had the effect of creating two accumulators running in parallel.

Reversing the Loop Order

The previous section illustrated how unrolling the outer loop cause the inner loop to be replicated and merged automatically during synthesis. However, there are situations that prevent automatic merging, and this leads to sub-optimal performance. Example 4-20 shows the accumulator design used in the previous section that has been modified to conditionally assign the index for the “acc” array. This conditional index assignment breaks automatic loop merging.
Example 4-20. Conditional Index Breaks Loop Merging

#include "accum.h"
void accumulate(int din[2][4], int dout[2], bool flag){
    int acc[2];
    int idx;
    ROW:for(int i=0;i<2;i++){
        idx = flag ? i: 1-i;
        acc[idx] = 0;
    COL:for(int j=0;j<4;j++){
            acc[idx] += din[i][j];
        }
        dout[i] = acc[i];
    }
}

Not merging the two copies of the COL loop that result from unrolling the ROW loop causes the loops to be scheduled sequentially (See “Sequential Loops” on page 69). Pipelining the main loop with II=1 causes the two copies of the COL loop, COL_0 and COL_1, to be flattened into the main loop, but they are still be executed sequentially as shown in Figure 4-34.

Figure 4-34. Schedule with Conditional Index and ROW Loop Unrolled

One possible solution to achieve the desired behavior of two accumulators running in parallel is to reverse the order of the ROW and COL loops. However, this must be done carefully since it usually requires that the outer loop body must be moved to the inner loop and executed conditionally. Example 4-21 shows how to manually reverse the loop order.
Example 4-21. Reversing the Loop Order

```cpp
#include "accum.h"
void accumulate(int din[2][4], int dout[2], bool flag){
    int acc[2];
    int idx;
    COL:for(int j=0;j<4;j++){
        ROW:for(int i=0;i<2;i++){
            idx = flag ? i: 1-i;
            if(j==0)
                acc[idx] = 0;
            acc[idx] += din[i][j];
            if(j==3)
                dout[i] = acc[i];
        }
    }
}
```

The following code changes were made in Example 4-21.

- Lines 6 and 7- reversed the order of the ROW and COL loops
- Line 8- Moved the index computation into the inner loop body
- Lines 9 and 10 - Moved the clearing of the accumulators into the inner loop body and made it conditional so that they are only cleared once at the beginning
- Lines 12 and 13 - Moved the writing of the output into the inner loop body and made the writes conditional so that the output is only written on the final iteration of COL

Sequential Loops

It is not uncommon to have multiple consecutive loops in a C++ design. Although these loops execute sequentially in the simulation of the C++, HLS can be directed to automatically merge these loops and execute them in parallel in hardware. However there are many cases where the C++ code can be written in such a way as to make automatic loop merging impossible. In these cases either the C++ code must be re-written to manually merge the loops if better performance is required, or explicit hierarchy should be used (See “Hierarchical Design” on page 191).

It is important for designers to understand the behavior of the hardware when loop merging does and does not take place so there are no unexpected results.

Simple Independent Sequential Loops

Example 4-22 shows the case where there are two sequential loops that are used to separately accumulate two four-element arrays.
Example 4-22. Independent Sequential Loops

#include “accum.h”
void accumulate(int din0[4], int din1[4],int &dout0, int &dout1){
    int acc0=0;
    int acc1=0;
    ACCUM0:for(int i=0;i<4;i++){
        acc0 += din0[i];
    }
    ACCUM1:for(int i=0;i<4;i++){
        acc1 += din1[i];
    }
    dout0 = acc0;
    dout1 = acc1;
}

High-level synthesis can automatically merge these loops because there are no dependencies between the loops and the indexing of the arrays is base solely on the loop iterators. With loops left rolled and automatically merged, and the main loop pipelined with II=1, the resulting schedule looks like that shown in Figure 4-35

Figure 4-35. Schedule of Merged Sequential Loops

The schedule shown above indicates that the loop iterations in each of the ACCUM loops can be run at the same time, resulting in a design that has two accumulators and runs in four clock cycles (Figure 4-36). If this kind of performance and increase in area is not required, automatic loop merging can be disabled during synthesis, allowing the loops to execute sequentially. This is discussed in the next section.
Effects of Unmerged Sequential Loops

In some instances sequential loops are not automatically merged. This can occur either intentionally because the design does not require the extra performance, usually at the cost of higher area, or because there are dependencies between the loops that break loop merging optimizations. Other operations such as conditional index assignment for reading or writing an array can also prevent loop merging optimizations. In either of these cases it results in designs that have both longer latency and throughput.

Consider the following design shown in Example 4-23. In this example the accumulated result from the ACCUM0 loop is used as the starting value for the ACCUM1 loop. These loops are not automatically merged since the ACCUM0 loop must finish before the ACCUM1 loop can start.

**Example 4-23. Unmerged Sequential Loops**

```c
#include "accum.h"
void accumulate(int din0[4], int din1[4], int &dout0, int &dout1){
    int acc0=0;
    int acc1=0;

    ACCUM0:for(int i=0; i<4; i++){
        acc0 += din0[i];
    }
    acc1 = acc0;
    ACCUM1:for(int i=0; i<4; i++){
        acc1 += din1[i];
    }
    dout0 = acc0;
    dout1 = acc1;
}
```

Figure 4-37 shows the schedule when the main loop of Example 4-23 is pipelined with an II=1. It also illustrates the effect of pipelining the main loop when there are unmerged sequential
loops in the design. Pipelining the main loop causes all loops in the design to be flattened, which in turn causes the last iteration of the ACCUM0 loop to be overlapped with the first iteration of the ACCUM1 loop. Although this improves the design performance slightly it has the impact of requiring two adders to implement the hardware. If performance is not an issue it is better to pipeline the ACCUM0 and ACCUM1 loops individually. This should then allow the operations scheduled in each loop to be shared, reducing the area. However pipelining the loops individually can impact the performance since each loop must then ramp-up and ramp-down separately.

**Figure 4-37. Schedule of Unmerged Sequential Loops with Main II=1**

Figure 4-38 shows the schedule when the ACCUM0 and ACCUM1 loops of Example 4-23 are pipelined with II=1 instead of pipelining the main loop. In this case there is no overlap between the loops and a single adder can be used to implement the hardware. However there is a two cycle performance penalty incurred due to the un-pipelined main loop (C1 Main and C2 Main).

**Figure 4-38. Schedule of Unmerged Sequential Loops with ACCUM(s) II=1**

**Manual merging of sequential loops**

It is up to the designer to manually merge sequential loops in situations where HLS does not do it automatically, and merged loops is the desired behavior. This usually means rewriting the
C++ code. Example 4-24 shows the manual rewrite of the code in Example 4-23 in order to achieve the best possible performance.

**Example 4-24. Manually Merged Sequential Loops with Main II=1**

```c++
#include “accum.h”
void accumulate(int din0[4], int din1[4],int &dout0, int &dout1){
    int acc0=0;
    int acc1=0;
    int tmp;
    ACCUM0_1:for(int i=0;i<4;i++){
        tmp = din0[i];
        acc0 += tmp;
        acc1 += din1[i]+tmp;
    }
    dout0 = acc0;
    dout1 = acc1;
}
```

The example shown above manually merged the sequential loops so that the design runs in four clock cycles when pipelining the main loop with II=1. However this design is larger than the previous implementations because it requires three adders, shown in the schedule in Figure 4-39.

**Figure 4-39. Schedule of Manual Merged Sequential Loops with Main II=1**

---

**Pipeline Feedback**

The initiation interval can be set anywhere from a synthesis tool dependent maximum down to an II=1 on any feed-forward design. However, a design with feedback limits the initiation interval to be no less than the delay of the feedback path. There are three types of feedback, data dependent, control dependent, and inter-block feedback. Inter-block feedback is discussed in later chapters covering system level design.

**Data Feedback**

Data feedback occurs when the input to a data path operation is dependent on a variable computed in the previous loop iteration. If the only loop in the design is the main loop the
variable must have been declared as static for there to be feedback. Consider the following design:

**Example 4-25. Data Feedback Design**

```c
1  void accumulate(int a, int b, int &dout){
2     static int acc=0;
3     int tmp = acc*a;
4     acc = tmp+b;
5     dout = acc;
6 }
```

**Design Constraints**
- Clock frequency slow
- Main loop pipelined with II=1

If the clock frequency for this design is assumed to be very slow the schedule and hardware would look approximately like Figures 4-40 and Figure 4-41. The design schedule shows that pipelining with II=1 is possible since each iteration of the main loop finishes computing “acc” before the next iteration starts. This is also obvious by looking at the hardware diagram.
Now consider the same design from Example 4-25 re-synthesized with the following constraints:

**Design Constraints**
- Clock frequency very fast
- Main loop pipelined with II=1
- Multiplier constrained to a two-cycle pipelined multiplier

This design cannot be pipelined with II=1 with the given set of constraints listed above. The failed schedule shown in Figure 4-42 illustrates why. To pipeline with II=1 would mean that “acc” is available to be read in the second clock cycle. However, the first pipeline stage is not finished computing “acc” until the edge of the third clock cycle. Another way to look at this is to examine the hardware that is synthesized, shown in Figure 4-43. It takes two clock cycles to compute “tmp” in the feed-forward path. “tmp” is then added to the current value of “b” and fed back to the multiplier. Lines 3 and 4 of Example 4-25 show that each time a new value of “acc” is computed it is available in the next iteration to compute “tmp”. Thus the hardware pipeline cannot be made to run every clock cycle since it must allow the multiplier to flush for each computation of “tmp”. The best possible performance would be pipelining with II=2.
The solution to getting the design discussed above to pipeline with II=1 is to modify the design to balance the delays between the feed-forward and feedback paths. This means introducing delay elements in the C++ along the feedback path. The functionality is different from the original design, but there is no other way to pipeline with II=1 and have the RTL match the C++ exactly. Example 4-26 shows Example 4-25 rewritten to balance the delay along the feedback path to match the two cycle feed forward delay. This is done by creating a two element shift register to delay “acc”. The hardware synthesized for Example 4-26 is shown in Figure 4-44.

In general the number of shift register elements needed in the feedback path can be computed as:

\[
\text{Num Shift Elements} = \frac{\text{feed-forward latency}}{\text{Initiation Interval (II)}}
\]

### Example 4-26. Balancing Feedback Path Delays

```c
1  void accumulate(int a, int b, int &dout){
2     static int acc=0;
3     static int acc_old0;
4     static int acc_old1;
5     int tmp0 = acc_old1*a;
6     acc = tmp0+b;
7     acc_old1 = acc_old0;
8     acc_old0 = acc;
9     dout = acc;
10  }
```

The details of Example 4-26 are:

- Lines 3 and 4 define two static variables used to implement the feedback delays.
- Line 6 uses the delayed feedback “acc_old1” as the input to the multiplier.
- Lines 8 and 9 implement the shift register to delay “acc” by two clock cycles.
Control Feedback

Pipelining failures due to feedback are also possible due to the loop control in a design. The deeper the nesting of loops in a design, the more complicated the control becomes, which in turn limits the clock frequency and ability to pipeline a design. Adhering to the recommended coding practices eliminates many of these potential issues. The following design, Example 4-27, illustrates how “bad” coding style can lead to problems when trying to pipeline. This design does not only have larger area than needed, but also fails pipelining for high clock frequencies due to control feedback. The cause of this is due to the 32-bit interface variables being used for the loop upper bounds. The impact of writing the C++ this way was covered in detail in “Optimizing the Loop Counter” on page 54 and “Optimizing the Loop Control” on page 55. Essentially there is a long combinational path created to evaluate the loop exit conditions. The outer loop “X” has to know when the inner loops are finished so it can exit immediately. Figure 4-45 shows the approximate hardware structure for Example 4-27. Although this is a very rough approximation it clearly shows that there is a combinational path through both 32-bit loop bounds comparisons, which severely impacts performance as the clock frequency is increased. A secondary problem is that the unbounded loops generate 32-bit logic for the loop counters. This can also prevent pipelining due to the feedback on the loop accumulator.
Example 4-27. Control Feedback

```c
void control(int din[8][8],
              int dout[8],
              int x_size,
              int y_size){
  int acc;
  X:for(int x=0; x<x_size; x++){
    acc = 0;
    Y:for(int y=0; y<y_size; y++){
      acc += din[x][y];
      dout[x] = acc;
    }
  }
}
```

Figure 4-45. Control Feedback

To minimize the possibility of feedback failures, Example 4-27 should be rewritten using the recommended style discussed previously. This is shown below in Example 4-28. The loops have been bounded, and the control logic for the loop exit reduced by using the appropriate bit widths on “x_size” and “y_size”
Example 4-28. Minimizing Control Feedback

```c
#include <ac_int.h>
void control(int din[8][8],
int dout[8],
ac_int<4,false> x_size,
ac_int<4,false> y_size){
    int acc;
    X:for(int x=0;x<8;x++){
        acc = 0;
        Y:for(int y=0;y<8;y++){
            acc += din[x][y];
            dout[x] = acc;
            if(y==y_size-1)
                break;
        }
        if(x==x_size-1)
            break;
    }
}
```

Conditions

Sharing

HLS can automatically share resources when it can prove mutual exclusivity. This means that HLS can theoretically share any similar operators that are in mutually exclusive branches of a condition, no matter how deeply nested the condition. The reality is that there are a number of ways that the C++ can be written and/or constrained so that the proof of mutual exclusivity is not possible. Usually this is due to a combination of either bad coding style or overly complex or deeply nested conditions. Good coding practices should always allow the maximum amount of sharing.

Conditional expressions are specified using the switch-case and if-else statements.

if-else statement

The if-else statement has the following form:

```c
if( condition0 ) {
    statement-list0;
}
else if( condition1 ) {
    statement-list1;
}
... else {
    statement-listN;
}
```
The conditions evaluate to a boolean expression and can range from simple boolean conditions to complex function calls. The statement list can be any number of C++ assignments, conditional expressions, or function calls.

**switch statement**

The switch statement has the following form:

```cpp
switch( expression ) {
    case 0: statement list0;
    break;
    case 1: statement list1;
    break;
    ...
    case N: statement listN;
    break;
    default: statement list;
    break;
}
```

The "expression" is typically an integer that selects one of the possible cases. The statement list can be any number of C++ assignments, conditional expressions, or function calls. The statement list for a selected case executes and is followed by a break.

**Note**

Although it is possible to have a “case” without a “break” this is not generally good for synthesizable C++. The behavior in C++ is to drop through to the next “case”. However in C++ synthesis this can sometimes cause replication of logic.

**Keep it Simple**

Think about what you want the hardware to do and code your design using good design practices. While it is easy to write complex deeply nested conditions and rely on the HLS tool to share everything, it is just as likely to get less sharing than expected. Consider the following design example (Example 4-29) that conditionally accumulates one of four different arrays based on several IO variables. Each condition branch calls the “acc” function with one of four arrays as the input.
Example 4-29. Automatic Sharing and Nested Conditions

```c
1  int acc(int data[4]){
2      int tmp = 0;
3      ACC:for(int i=0;i<4;i++)
4          tmp += data[i];
5      return tmp;
6  }
7  void test(int a[4], int b[4], int c[4], int d[4],
8             bool sel0, bool sel1, bool sel2, int &dout){
9      int tmp;
10     if(sel0){
11         if(sel1)
12             tmp = acc(a);
13         else if(sel2)
14             tmp = acc(b);
15         else
16             tmp = acc(c);
17     }else
18         tmp = acc(d);
19     dout = tmp;
20  }
```

Design Constraints
Clock frequency slow
Main loop pipelined with II=1
All loops unrolled

There are several potential problems with the design in Example 4-29.

1. The four calls to the “acc” function are by default all inlined during synthesis. This means that there are four copies of the “ACC” loop that are inlined and optimized. Although it is possible that HLS can still share everything this will in general lead to longer synthesis runtimes since all four copies must be merged back together and shared. One possible solution to improve sharing and runtime would be to make “acc” into a component using a HLS component flow.

2. Even if everything is shared it is likely that HLS will perform fine-grained sharing, which leads to more MUX logic since each individual operator is shared separately. One possible solution to minimize MUX logic would be to make “acc” into a component.

3. The conditions in this example are simple and the clock frequency is slow enough so that everything is scheduled in the same clock cycle. As the conditions become more complex, the nesting becomes deeper, and/or the clock frequency increases, it is likely that operators will be scheduled in different clock cycles. This can limit sharing. Making “acc” into a component will not help in these types of situations. The best solution is to rewrite the code so that “acc” is called once.

Example 4-30 shows Example 4-29 rewritten to facilitate sharing. The key is to use the conditions to compute the MUXing of data and control and call the function only once.
Example 4-30. Explicit Sharing and Nested Conditions

```cpp
1  int acc(int data[4]){
2      int tmp = 0;
3      ACC:for(int i=0;i<4;i++)
4          tmp += data[i];
5      return tmp;
6  }
7  void test(int a[4], int b[4], int c[4], int d[4],
8              bool sel0, bool sel1, bool sel2, int &dout){
9      int tmp, data[4];
10     for(int i=0;i<4;i++)
11        if(sel0){
12           if(sel1)
13              data[i] = a[i];
14           else if(sel2)
15              data[i] = b[i];
16           else
17              data[i] = c[i];
18        }else
19            data[i] = d[i];
20      tmp = acc(data);
21     dout = tmp;
22   }
```

**Design Constraints**
Clock frequency slow
Main loop pipelined with II=1
All loops unrolled

Example 4-30 will in general give better results than Example 4-29. This is because the nested conditional expression is only used to control the selection of the input array. Once the input array is selected the “acc” function is called once. Doing this allows HLS to easily optimize the adder tree for the “acc” function and the MUX logic is only needed to select the input data. In essence this is coarse grained sharing. This style should be used when using component flows does not give the desired sharing. This example will also have better runtime in general since the “ACC” loop is only inlined once.

**Functions and Multiple Conditional Returns**

Although multiple returns in function calls are allowed by both C++ and HLS, they are in general a bad idea. This is true both from a code debugging perspective as well as a synthesis quality of results issue. HLS balances the pipeline stages of all conditional branches. Having a return in the branch complicates this and makes it more difficult to pipeline a design. It is best to use a single return at the end of the function. It’s especially bad to use multiple returns to try and make things mutually exclusive.

Consider the following example:
Example 4-31. Multiple Conditional Returns

```c
1 int acc(int data[4]){
2   int tmp = 0;
3   ACC:for(int i=0;i<4;i++)
4     tmp += data[i];
5   return tmp;
6 }
7 int test(int a[4], int b[4], bool sel0, bool sel1){
8   if(sel0){
9     return acc(a);
10 }   
11   if(sel1){
12     return acc(b);
13 }  
14 }
```

Example 4-31 has several problems that will prevent good QofR.

1. Although sel0 and sel1 are mutually exclusive in that they are never evaluated together, HLS will typically not be able to prove this and will not share “acc”.

2. The function returns on both lines 9 and 12. If HLS cannot prove mutual exclusivity it will not be able to pipeline with II=1 if the function return is mapped to an IO.

3. The function only returns if “sel0” or “sel1” is true. This means that the return value can be undefined. This undefined behavior may cause logic to be optimized away or simulation behavior of the RTL may not match the C++.

Example 4-31 is rewritten to have only one return, shown below.

Example 4-32. Single Function Return

```c
1 int acc(int data[4]){
2   int tmp = 0;
3   ACC:for(int i=0;i<4;i++)
4     tmp += data[i];
5   return tmp;
6 }
7 int test(int a[4], int b[4], bool sel0, bool sel1){
8   int tmp = 0;
9   if(sel0){
10     tmp = acc(a);
11 }   
12   else if(sel1){
13     tmp = acc(b);
14 }  
15   return tmp;
16 }
```

Example 4-32 has made the conditions mutually exclusive. A temporary variable “tmp” is used to store the result of each condition and is then returned on line 15. The temporary variable is initialized to zero as well so the return value will never be undefined.
Replacing Conditional Returns with Flags

It is also possible to bypass entire sections of code by using a conditional return. This should also be avoided. It is always possible to replace the conditional return with a flag variable that can bypass the code. Consider the following code fragment:

**Example 4-33. Conditional Return to Bypass Code**

```c
... 
tmp = 0;
tmp += a;
if(sel0)
    return tmp;
tmp += b;
return tmp;
```

Example 4-33 only adds “b” to “tmp” if “sel0” is false. It should be rewritten as:

**Example 4-34. Using Flags to Bypass Code**

```c
... 
bool flagl = false;
tmp = 0;
tmp += a;
if(sel0)
    flag = true;
if(flag)
    tmp += b;
return tmp;
```

Example 4-34 replaces the conditional return with a flag that is set conditionally. The flag is then used to conditionally bypass the same sections of code that were bypassed by the conditional return. A single return is used at the end of the function.

**Note**

A function should have one and only one return.

**References**