written every clock cycle, or in combination with some hardware synchronization to make sure that IO is accessed in the correct c-step.

**Figure 5-4. Hardware of Unconditional IO Passed by Value**

![Diagram of hardware](image)

**Conditional IO**

An IO is considered conditional if the interface variable is mapped to a resource that has a hardware “handshake”. This handshake can consist either of a simple ready to send or receive data, or a ready/acknowledge behavior. Unlike unconditional IO, where high-level synthesis is free to move IO into and out of conditions in the C++, conditional IO cannot be moved into or out of conditions in the C++ code. The only exception to this rule is when the variable mapped to IO is pass-by-value. In this case the IO is always read once at the beginning of the design schedule and stored in registers. Using pass-by-value variables in this way can have some potentially unexpected behavior when the IO has a handshake.

**Pass by Reference**

Similar to the pass-by-reference example for unconditional IO, passing by reference using conditional IO requires that the data is setup and available before the IO is accessed. However conditional IO provides the mechanism to synchronize the transfer of data via ready/acknowledge control signals. In Example 5-3 the “threshold” variable is mapped to an IO resource that generates a ready for data strobe. It is assumed that the data is already available for reading in an off-chip FIFO so an acknowledge is not required. The “threshold” interface variable is read conditionally after the “flag” interface variable is read and evaluates to true.
Example 5-3. Conditional IO Passed by Reference

```c
void accumulate(int din[4], int &dout, int &threshold, bool &flag){
    int acc=0;
    ACCUM:for(int i=0;i<4;i++){
        acc += din[i];
        if(flag)
            if(acc > threshold)
                acc = threshold;
    }
    dout = acc;
}
```

Design constraints - ACCUM loop with II=1
threshold and dout mapped to ready to send or receive data interface
All other IO mapped to wire interfaces

Figure 5-5 shows the schedule for Example 5-3. The IO for “threshold is considered synchronous because the “ready for data” control signal is a registered signal. This is indicated in the schedule by showing the IO operation crossing the clock boundary.

Figure 5-5. Schedule of Conditional IO Passed by Reference
One of the effects of reading “threshold” inside of a condition based on “flag” is that the number of c-steps for each iteration has been increased. The “threshold” request-for-data control signal cannot be asserted until the condition based on “flag” has evaluated to true. The timing diagram of this behavior is shown in Figure 5-6. The read request for threshold is only generated when “flag” is asserted high.

**Figure 5-6. Timing of Conditional IO Passed by Reference**

Figure 5-7 shows the hardware implementation of Example 5-3. Mapping to an IO with a “request” signal causes the synthesis process to insert a hardware control signal that can be hooked up to an external FIFO to control the flow of data. Additionally, the synchronous nature of “threshold” requires that “flag” is read in the previous clock cycle than “threshold”.

**Figure 5-7. Hardware of Conditional IO Passed by Reference**
Pass by Value

The previous section illustrated how a pass-by-reference interface variable mapped to a conditional IO is only read inside of a C++ condition when the condition is true. This includes the generation of the “request” control signal. When using pass-by-value variables on the interface the behavior is different. Pass-by-value interface variables are always read at the beginning of the main loop regardless of where they are used in the C++ code. So even if the variable is read conditionally in the code, as shown in Example 5-4, a request-for-data is still generated.

Example 5-4. Conditional IO Passed by Value

```c
void accumulate(int din[4], int &dout, int threshold, bool flag){
    int acc=0;
    ACCUM:for(int i=0;i<4;i++){
       acc += din[i];
       if(flag)
           if(acc > threshold)
               acc = threshold;
    }
    dout = acc;
}
```

Design constraints - ACCUM loop with II=1
threshold and dout mapped to request for data interface
All other IO mapped to wire interfaces

The schedule for Example 5-4 is shown in Figure 5-8. “threshold” has its request-for-data control signal issued in C1_Main and “threshold” and “flag” are read once, and only once, in C2_Main since they are both pass by value. The timing diagram in Figure 5-9 shows the potentially unexpected behavior of this example. The “threshold” request-for-data control signal is asserted and “threshold” is read is regardless of the value of “flag”. This happens because pass-by-value interface variables are always read once at the beginning of the main loop no matter where they are used in the C++ code. Figure 5-10 shows the hardware synthesized for Example 5-4.
Figure 5-8. Schedule of Conditional IO Passed by Value

Figure 5-9. Timing of Conditional IO Passed by Value
Ready/acknowledge Behavior (wait)

In addition to being able to automatically map interface variables to request-for-data type resources, high-level synthesis lets users map to interface resources that have a ready/acknowledge type behavior. These type of interfaces must be used with caution since they are more restrictive in terms of how the generated hardware behaves. In particular they can produce unwanted behavior when pipelining the main loop with II=1. This more restrictive case is discussed in the next section. For now we can re-use Example 5-3 on page 91 with a different set of constraints.

Design constraints – ACCUM loop with II=1
threshold mapped to request-grant interface
dout mapped to request for data interface
All other IO mapped to wire interfaces

Mapping the “threshold” interface variable to a ready/acknowledge type resource yields essentially the same schedule as that shown in Figure 5-5 on page 91. However the timing and hardware implementation is slightly different. The timing diagram for this example is shown in Figure 5-11. The “threshold” acknowledge control signal is used to determine when data is available. If data is being requested, “threshold” request driven high, and the acknowledge signal is low, the current loop iteration stalls the entire design until acknowledge goes high. This requires that the current data that is driven from “off-chip” must be held stable until the iteration completes. The timing diagram shows that this type of ready/acknowledge interface behavior is well suited for connecting to an off-chip FIFO, where ready is connected to the FIFO read, and acknowledge is connected to the FIFO empty flag, Figure 5-12.
Figure 5-11. Timing of IO with Wait

Figure 5-12. Hardware of IO with Wait

Stalling the Pipeline

Using IO resources with ready/acknowledge behavior showed that it is possible to stall the execution of a loop until data is available without any unwanted behavior. In the previous examples the main loop was left unconstrained while the ACCUM loop was pipelined with II=1. This allows the ACCUM loop to ramp-up and ramp-down, which in turn allows any data in the pipeline to “flush” after the last input is read. In designs with pipelines consisting of more than one stage, this “flushing” does not occur if the main loop is pipelined with II=1, and the whole pipeline can stall before the last output is written. Example 5-5 shown below is the four element accumulator that was used in previous sections.
Example 5-5. Stalling the Pipeline with Conditional IO

```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
- ACCUM loop fully unrolled
- din mapped to ready-acknowledge resource
- dout mapped to ready resource

Fully unrolling the ACCUM loop creates a two-stage balanced adder tree. In this example it is assumed that the clock frequency is sufficiently fast so that the adder tree stages are scheduled in separate c-steps. Figure 5-13 shows the schedule for Example 5-5. This shows that the read for iteration 1 of the main loops happens before the write of iteration 0. If the read data is not available the pipeline stalls.

**Figure 5-13. Schedule of Pipelined Main Loop with Conditional Wait IO**

Figures 5-14 and 5-15 show the timing and hardware diagrams for Example 5-5. They illustrate how the previous read is completing in pipeline stage 2 while the current read is needed for pipeline stage 1. If the current read data is unavailable the previous read data gets stuck in the
pipeline. In other words, the pipeline does not flush if there is no data available for reading at the input.

**Figure 5-14. Timing of Pipeline Stall**

![Timing of Pipeline Stall Diagram](image_url)

**Figure 5-15. Hardware of Pipelined Main Loop with Conditional Wait IO**

![Hardware of Pipelined Main Loop Diagram](image_url)

Having the pipeline stall is sometimes unacceptable for certain types of designs, especially designs that do not have continuously running data. Video is a good example of this, where horizontal and vertical blanking create gaps in the pixel data. One way to prevent the pipeline from stalling is to not pipeline the main loop but pipeline the inner loops. Pipelining the inner loops allows the pipeline to ramp down, flushing all data. The downside of not pipelining the main loops is that there is a multi-clock cycle penalty for the time it takes to ramp-up and ramp-down the pipeline. If the main loop must be pipelined for performance reasons the other solution is to manually code the “ack” into the C++ code to allow the pipeline to flush.

---

**Caution**

Pipelining the main loop when using handshaking IO can prevent the pipeline from flushing.
Manually Flushing the Pipeline

It’s possible to manually flush the pipeline in a design with a pipelined main loop by explicitly coding the acknowledge into the C++ interface. Example 5-6 has taken the code from Example 5-5 and modified it so that acknowledge is now part of the top-level C++ interface.

Example 5-6. Manually Flushing the Pipeline

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

**Design constraints**
Main loop pipelined with II=1
ACCUM loop fully unrolled
din mapped to ready for data resource
dout mapped to ready to send resource

The C++ is written so that an output is always produced every time “ack” is true. When “ack” is false the loop is skipped and the function exits. This behavior allows the pipeline to flush since the design doesn’t have to wait if data is not available. One side effect of this is that the read-for-data signal for “din” is not asserted until “ack” goes high. In the previous example that mapped “din” to a ready/acknowledge interface the ready-for-data signal was issued immediately regardless of the value of the acknowledge. Figure 5-16 shows the timing of Example 5-6.

**Figure 5-16. Timing of Manually Flushing the Pipeline**
Writing IO for Throughput

All of the IO examples covered previously have been able to schedule with the main loop pipelined with II=1. This was because either there was only a single IO access per loop iteration or because the IO accesses were automatically merged when inside of an unrolled loop. There are instances when multiple IO accesses or conditional IO inside of unrolled loops are not merged, which in turn prevents pipelining a design and limits the throughput. When this happens it usually requires modifying the C++. Example 5-7 shows the four element accumulator with a slight modification. The “flag” array is tested inside the loop to determine which element of “din” is read and accumulated. If all elements of “flag” are false then “din” is never read. This design can not be pipelined when the ACCUM loop is unrolled because multiple IO access are created and cannot be merged.

**Example 5-7. IO Throughput Limiting Design**

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

**Design constraints**
ACCUM loop fully unrolled
One ADD takes most of
din maped to ready for data resource
dout mapped to ready to send data resource

Figure 5-17 shows the schedule for Example 5-7 where the main loop is not pipelined. Reading the IO “din” conditionally, where each condition is different “flag[i], in this case causes four separate conditional reads that are not merged, even though each read is only accessing a slice of “din”. This can be better understood by looking at Example 5-8 which shows
Example 5-7 with the ACCUM loop manually unrolled. What this illustrates is that each condition must be evaluated before “din” can be read and accumulated. Thus there is a dependency between each loop iteration and this prevents the IO accesses from being merged into a single access, causing four separate reads of “din”.

**Example 5-8. Manual Unrolling of IO Throughput Limiting Design**

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

The reason why this design cannot be pipelined is evident from Figure 5-18 which shows that overlapping iterations of the main loop would require simultaneous multiple reads from the IO port “din”, which is physically impossible.
Figure 5-18. Design that Can’t be Pipelined Due to Unmerged IO

Making IO Mergable

The code for Example 5-7 should be rewritten to either make the read of “din” unconditional when possible, or to simplify the condition so that the reads can be merged. Example 5-7 illustrates how you often get what you asked for, but not what you want, when writing synthesizable C++ code. Let’s assume that after analyzing the undesirable scheduling results from HLS, it is determined that “din” can, and should, be read every iteration of the main loop, since in hardware it is expected that all four values of “din” come in parallel from an external FIFO. With this assumption the C++ code can be rewritten as shown in Example 5-9. All elements of “din” are read in the beginning of the design regardless of the value of “flag”, and then stored in the internal variable “din_int”. The internal variable is then used in the ACCUM loop.
Example 5-9. Making IO Read Unconditional

```c
void accumulate(int din[4], int &dout, bool flag[4]){
    int acc=0;
    int din_int[4];
    bool flag_int;

    DIN:for(int i=0;i<4;i++)
        din_int[i] = din[i];
    ACCUM:for(int i=0;i<4;i++){
        if(flag[i])
            acc += din_int[i];
    }
    dout = acc;
}
```

Design constraints
Main loop pipelined with II=1
DIN and ACCUM loops fully unrolled
din mapped to ready for data resource
dout mapped to ready to send resource

Figure 5-19 shows the scheduled design for Example 5-9 where making the read of all elements of “din” unconditional allows them to be merged into a single read, which in turn allows the design to be pipelined with II=1. Although the reads have been merged, the three adders in the design have a dependency because of the conditional accumulate based on “flag[i]”. This dependency prevents adder tree balancing and can result in sub-optimal hardware, especially as the clock frequency is increased to the point where the adders must be scheduled in separate clock cycles.

Figure 5-19. Schedule of Unconditional IO Read

One potential problem with the re-write of Example 5-9 is that the read of “din” always occurs regardless of whether any of the “flag[i]” elements are set, which is different from the behavior of Example 5-7. If the desired behavior is to only read “din” if at least one element of “flag[i]” is set the code can be rewritten to give this type of behavior, shown in Example 5-10.
Example 5-10. Simplifying Conditional IO to Help Merging

```c
void accumulate(int din[4], int &dout, bool flag[4])
{
    int acc=0;
    int din_int[4];
    bool flag_int;
    FLAG:for(int i=0;i<4;i++)
        flag_int |= flag[i];
    DIN:for(int i=0;i<4;i++)
        if(flag_int)
            din_int[i] = din[i];
    ACCUM:for(int i=0;i<4;i++)
        if(flag[i])
            acc += din_int[i];
        else
            acc += 0;
    dout = acc;
}
```

**Design constraints**
- Main loop pipelined with II=1
- FLAG, DIN and ACCUM loops fully unrolled
- din mapped to ready for data resource
- dout mapped to ready to send resource

Example 5-10 has made the read of “din” conditional by creating a boolean variable that is equal to the “OR” of all of the “flag[i]” elements, which is done in the FLAG loop. If “flag_int” is true then “din” is read. Using the simple condition inside of the DIN loop allows the IO reads to be merged.

---

**Caution**

Conditionally reading arrays mapped to IO inside of unrolled loops has the potential to prevent pipelining. Make the IO reads unconditional when possible by reading the entire array into an internal array.

---

**Memories**

HLS not only allows users to map arrays to IO resources, where the array elements are available in parallel with or without a handshake, but also allows them to map arrays to memory type resources. Both internal arrays and arrays on the top-level function interface can be mapped to memory resources. If the array is on the top-level interface HLS creates the address, data, and control signals required to interface to an off-chip memory. If the array is internal to the design HLS not only creates the necessary address, data, and control signals to access the memory, but it also instantiates the memory model. In the case of targeting ASIC designs this instantiation of the memory is only used for simulation, and is black boxed for synthesis since ASIC synthesis does not infer memories. The “black-box” memory can then be replaced with the physical memory produced by the users memory compiler. In the case of FPGA design, the instantiated memory is used not only for simulation, but is inferred as a memory by the FPGA RTL synthesis tool.