Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

Math2Mat

VHDL generation documentation

Revision	Date	Who	Comments
0.0	04.04.2011	DMO	Initial version
1.0	15.04.2011	DMO	Adds and Spelling corrections

Contents

1. Introduction

The main purpose of this report is to describe the mechanisms put in place to generate VHDL files corresponding to an octave function. To understand the details of the generation of an octave function, an initial study from the outside of the functioning of a Math2Mat block and its signals is necessary. It will then be possible to enter into the details of generations of the different elements encountered in the structure such as polynomials, conditional statements and loops. Wrappers allowing to connect the various hardware blocks representing the operations will also be discussed.

1.1 Block representation

The following figure shows a generic Math2mat block comprising “n” number of entries and “m” number of outputs:

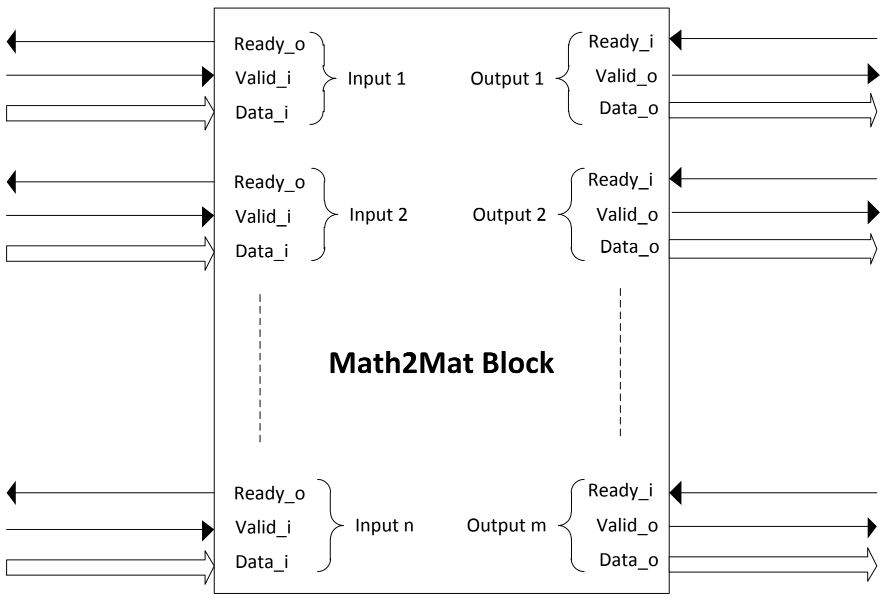


Figure 1: *Generic Math2Mat block*

Each input and output is accompanied by three signals. Input signals have the following meanings:

- Data_i: this signal is controlled by the user of the block. It indicates the actual value of the data. Its size is 32 or 64 bits depending on the chosen architecture.
- Valid_i: this signal is controlled by the user of the block. It indicates the validity of the data. It is represented by a single bit.
- Ready_o: this signal is returned by the Math2Mat block. It indicates when the input is ready to receive data. It is represented by a single bit.

Output signals have the following meanings:

- Data_o: this signal is returned by the block. It indicates the actual value of the data. Its size is 32 or 64 bits depending on the chosen architecture.
- Valid_o: this signal is returned by the block. It indicates whether the data is currently valid. It is represented by a single bit.
- Ready_i: this signal is controlled by the user of the block. It indicates when the output is ready to receive data. It is represented by a single bit.

1.2 Generic wrapper

To avoid conflict between the names of inputs/outputs of a Math2Mat block and names of inputs/outputs of different VHDL modules around the Math2Mat block, generic wrappers are generated. They used to math2math own names for each of the inputs and outputs. Inputs and outputs of wrappers are using specific names for each Math2Mat inputs and outputs. Here is an example of the generation of a Math2Mat block and using two inputs "a" and "b" and one output "c":

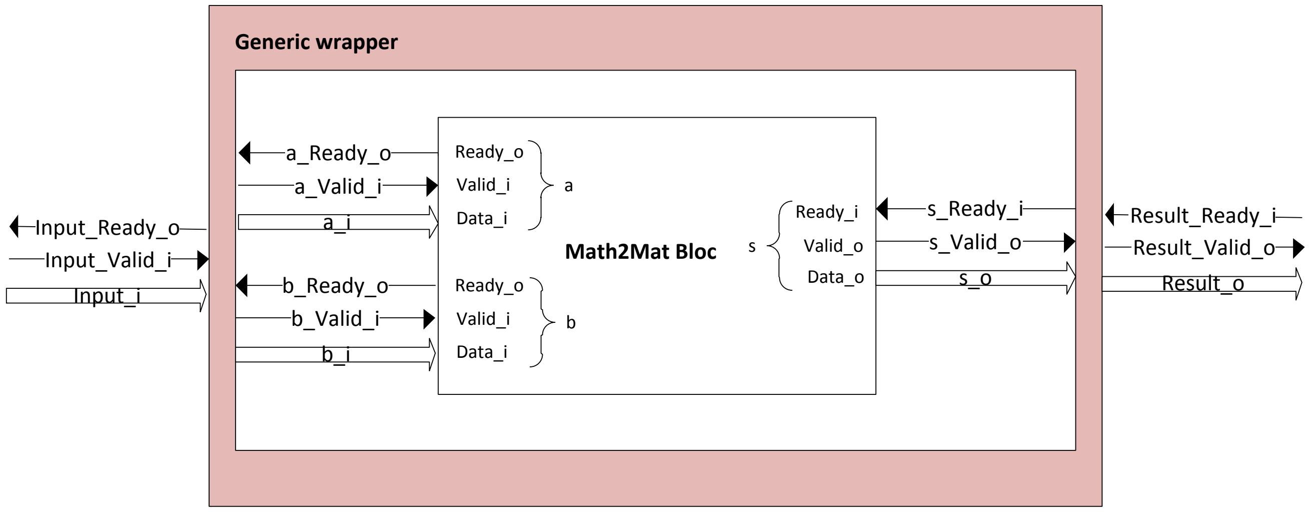


Figure 2: *Generic wrapper*

Four tables of "std_logic" can represent all input and output "ready" and "valid":

- "Input_valid_i" and "Input_ready_o" represent the "ready" and "valid" inputs.
- "Result_valid_o and "Result_ready_i" represent "valid" and "ready" outputs.

Two other tables of "std_logic_vector" "Result_o" and "Input_i" allow representing all inputs and outputs. The various indexes of all these tables must match. The figure below shows the positioning signals in these six tables:

Input_i

Input_i(0)	a_i
Input_i(1)	b_i

Input_Valid_i

Input_Valid_i(0)	a_Valid_i
Input_Valid_i(1)	b_Valid_i

Input_Ready_o

Input_Ready_o(0)	a_Ready_o
Input_Ready_o(1)	b_Ready_o

Figure 3: *input tables representation*

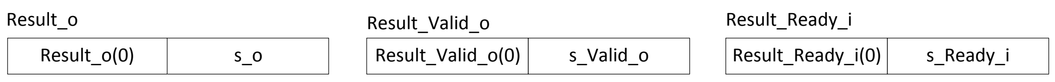


Figure 4: *output tables reprsentation*

1.3 Utilization

A Math2Mat block is seen from the outside regardless of content. The user relies on signals “ready” and “valid” to control it as he wishes. He will certainly prefer to transmit all inputs at the same time. However, some data may not be ready to receive data. We'll see later that the user has the possibility to force the block to indicate that an input is ready only if all inputs are.

1.3.1 Timing diagrams

To better understand how a block from the outside, consider the following timing diagram with two inputs and one output:

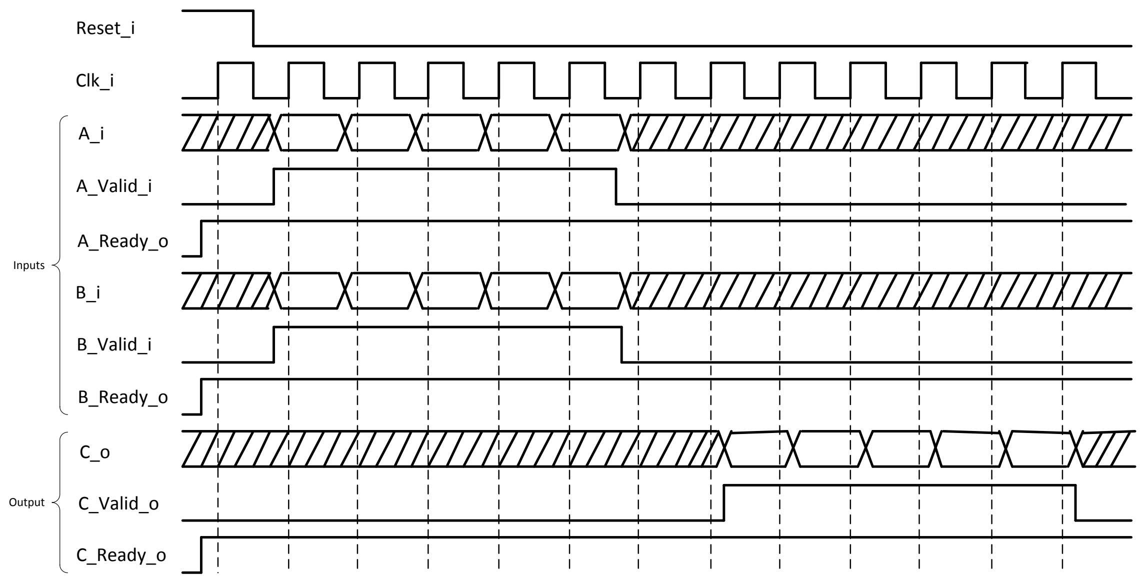


Figure 5: *timing diagram of a usual case*

In this figure, we observe the sending of five data on each input and the reception of six data on the output of the block. The processing of a given takes seven clock cycles. The block operates in its usual case. Indeed, each input is always ready to receive data, the data is sent at the same time and the external user of the block is ready to receive data also.

The following timing diagram helps to highlight the behavior of the same block in one less common case:

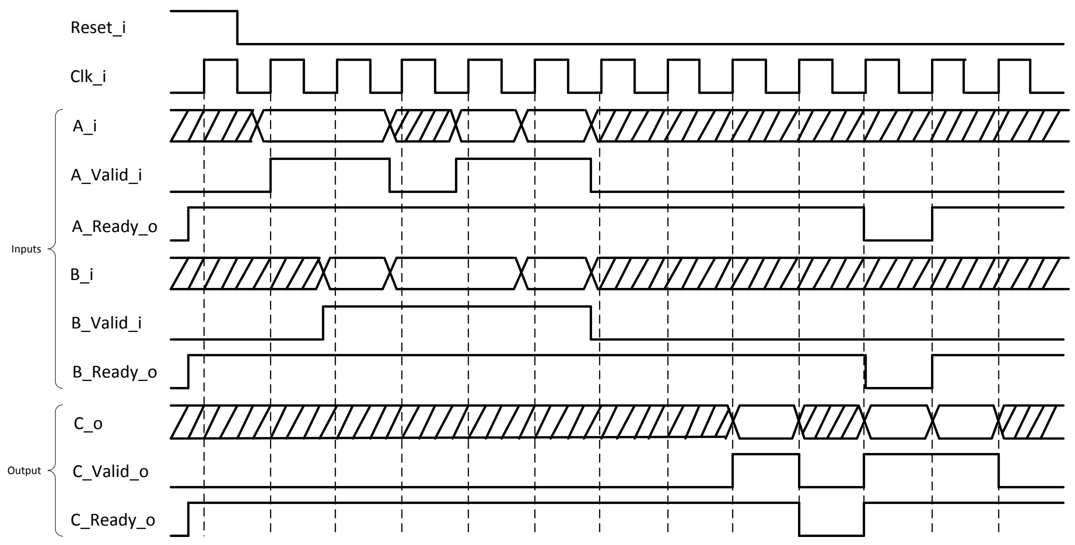


Figure 6: *timing diagram of a no-usual case*

One notice directly the influence of the control signals “valid” and “ready” on the flow of data within the block. This diagram is not absolute because the value of the “ready” input depends directly to the content of the block. Indeed, as we shall see later a number of fifos may be present within the block and cause delays on the data.

1.4 Content block

1.4.1 Usual content

The content of a block is composed of interconnections of different sub-blocks represented by operators. Each operator is used to construct a polynomial, a conditional statement or a loop. To interconnect all of these operators, wrappers including control logic are inserted. When an input is used by several operators, a combinational logic is also required. The following diagram shows the typical contents of a block:

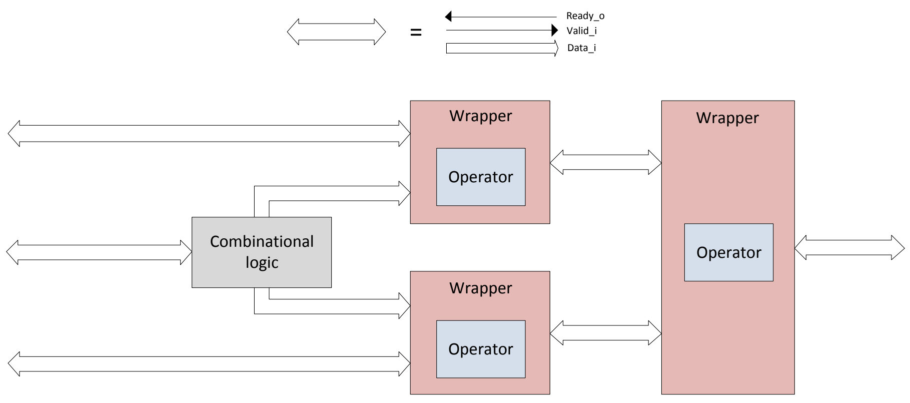


Figure 7: *block content*

The next sections of this report will present the entire framing element operators. This pattern will be partially changed on the chapter on loops where some new elements will appear.

1.4.2 Propagation of “ready”

When the user of a Math2Mat block is not ready to receive data, the “ready” signal is spread within the entire block wrapper. The following example shows this propagation:

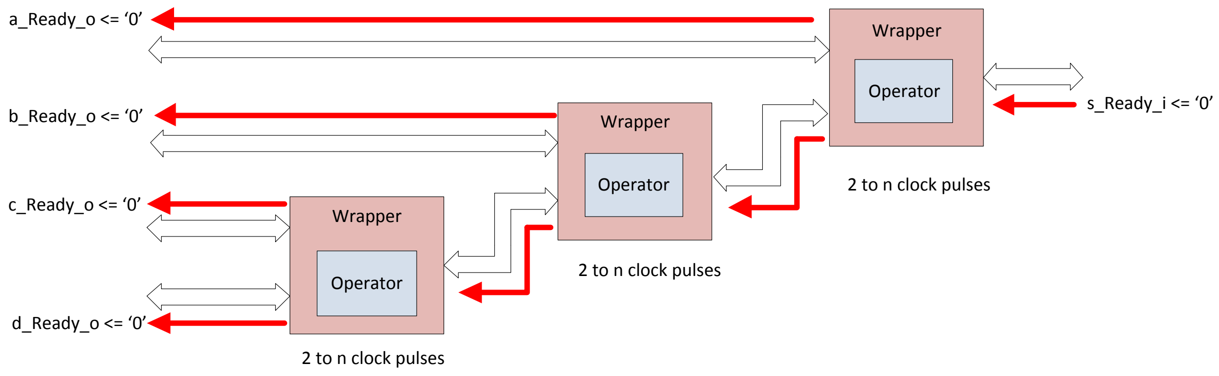


Figure 8: *propagation of "ready"*

The number “n” of clock pulses to propagate the “ready” on inputs depends on the contents of the wrapper that we will see later. The bold arrows represent the propagation of the signal on the inputs.

2. Wrapper

2.1 Introduction

A wrapper is used to interface input and output signals “data”, “valid” and “ready” with a particular operator. Each operator can have one, two or three inputs and one output. It also provides the following entries:

- Valid_in: this signal indicates whether the operator entries are valid.
- Valid_out: this signal indicates if the output of the operator is valid.
- Ready_out: this signal indicates whether the operator is ready to receive data.
- nStall: his active low signal is used to stop the chain of calculation of the operator.

All these signals are represented by a single bit. The outer part of the block diagram below illustrates the interface to develop for a wrapper who includes an operator with two inputs and one output:

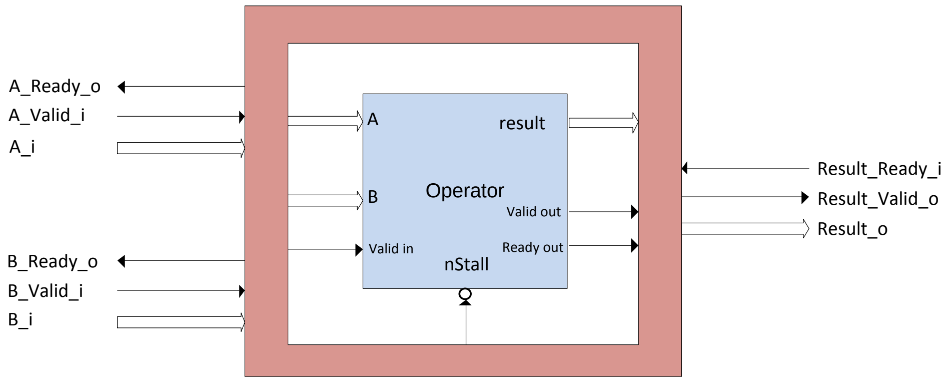


Figure 9: *bloc diagram of a wrapper*

The number of generated wrappers must match the number of different operators used in the octave function. Automatic generation of wrapper must be flexible on the number of inputs and outputs of the operator.

2.2 Implementation

A wrapper is composed of three separate parts:

- The first part allows the synchronization of “ready” and signals “valid” inputs of the wrapper.
- The second part includes the implementation of the combinational logic for the management of “valid” and “ready” signals to the operator and the wrapper.
- The third part maintains a valid output when disabling the block by the signal “ready_in”.

The following diagram represents a wrapper for an operator having two inputs and one output:

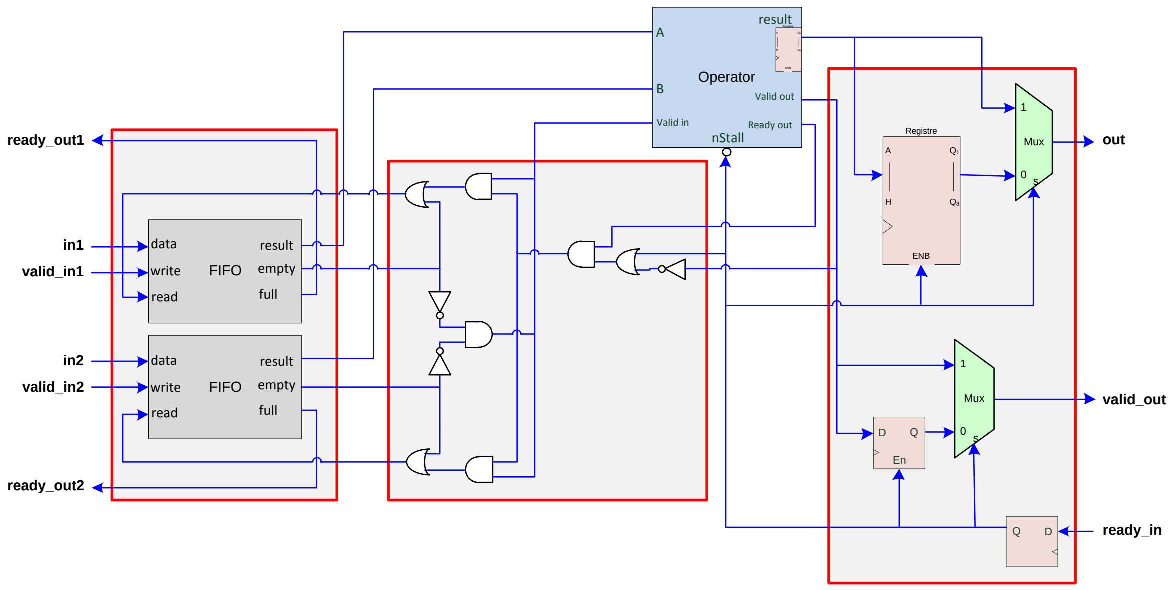


Figure 10: *wrapper for a basic operator*

2.2.1 Internal fifo

The internal fifos will eliminate some combinational loops between different wrappers. These loops can occur when two wrappers are bound together by their inputs and outputs. This phenomenon will be discussed in the section on polynomials.

These fifos can overcome the combinatorial link between “valid” and “ready” input for a wrapper. Their size is set by default to two to remove the combinatorial link and allow data sends in bursts. Indeed, if their size was one, the HDL description fifos could not send bursts. The fifo would be full, then empty, then full, etc.

Thereafter, we'll see that these fifos will also serve as a buffer for a multi-used signal.

2.2.2 Combinational logic

The combinational logic between the fifos and operator is used to:

- Manage the influence of the inputs of the operator with each other.
- Indicate the validity of inputs present on the operator content in the wrapper.
- Spread the state of the “ready” signal of the wrapper on the wrapper operator inputs.
- Propagate the state of the “ready” signal of the operator contained in the wrapper to the inputs of the operator.

2.2.3 Synchronisation

This block allows the management of three different signals:

- The output of the operator is connected to a multiplexer and a register also connected to this multiplexer. This register keeps the output state of the operator when the input “ready” of the wrapper is inactive. This “ready” is then used to select the direct output of the operator if it is active or memorize output of the register if it is not. Thus, the output of the operator is not lost and can be obtained during the reactivation “ready”.

- The mechanism used for the output of the operator is also done for “valid” signal of the operator.
- The “ready” input is connected to a registry to limit the combinatorial path of this signal if the block Math2Mat includes a large number of operators. This synchronization makes it possible to save the output data of the operator depending on the state of the “ready” as we seen previously.

3. Mathematical function

3.1 Introduction

The first generation produced is that of polynomials. It mainly covers the elements previously seen by adding a combinational logic multi-used signals manager. To move from an octave code to a VHDL description, a number of transformations are needed on the structure. Indeed, the structure contains a high level description of the function and includes not all control signals seen previously. This section also presents mechanisms to obtain the latency of a function and adding compensation fifos for some operators to ensure this latency.

3.2 Multi-used signal

3.2.1 Structure modifications

To show the necessary changes in the structure, consider the following octave code and its corresponding structure:

```
tmp = a+b;
S = tmp+a+tmp;
```

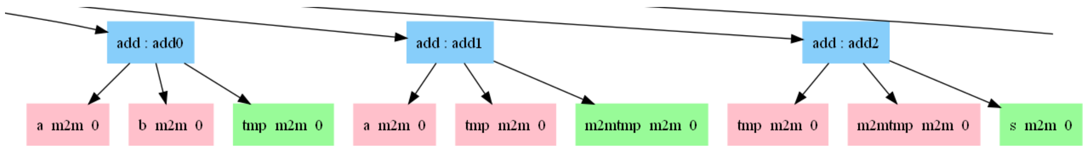


Figure 11: *structure state before generation*

We note that signals “a” and “tmp” are used twice. Materially, it is impossible to represent these signals as one signal. That's why these signals should be decomposed into two in order to avoid possible conflicts. We obtain the following structure:

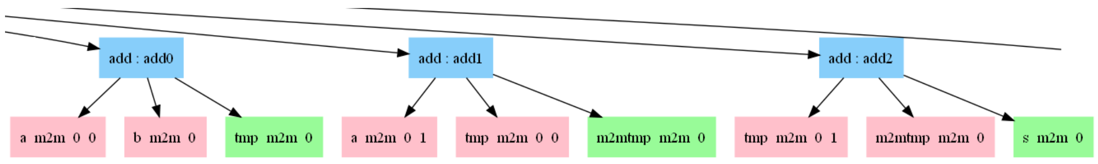


Figure 12: *structure state after generation*

Note that both signals have indeed been separated into two. It is necessary to define a logic which interacts with both signals.

That transformation of the structure must be present for all signals contained in octave functions. These transformations naturally generalize when the signals are used more than twice.

3.2.2 Combinatorial logic

The combinatorial logic must respect the following requirements:

- The output "ready" signal must combine all "ready" signals.
- The input "valid" signal must be shared by all inputs.
- If one input is not ready to receive data, the other inputs related to this input should not be ready either.

The first two points are relatively easy to achieve. Indeed, just ready to consolidate and decompose the signal valid. Here is an example for a signal "a_m2m_0" used twice:

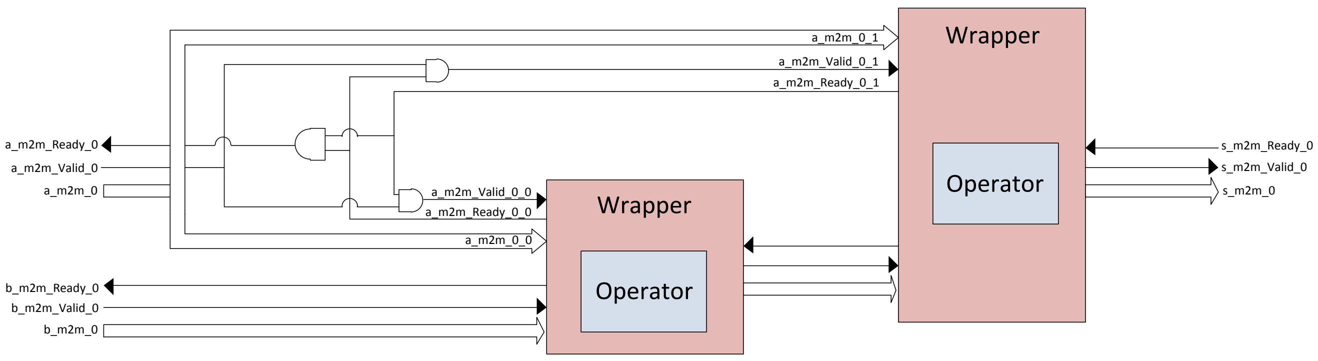


Figure 13: *combinational logic of multi-used signal*

The data signal is directly shared by the two decomposed signal. The "ready" signal is simply a "and" between the two "ready" decomposed signals. The "valid" signals are obtained in a slightly more complex manner. Indeed, in order to prevent the sending of new data on common inputs to decomposed inputs where the value is valid, it is necessary to interact with the "valid" inputs. The following scenario is used to represent this case:

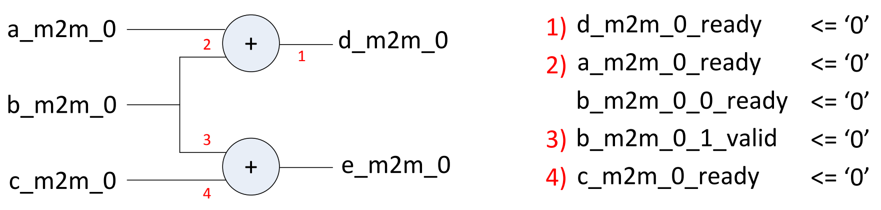


Figure 14: *principle of multi-used variables logic*

The main point of this operation is to disable the signal through the wrapper "c_m2m_0_ready" signal "b_m2m_0_1_valid". Again, this mechanism can be generalized to a large number of inputs and dependencies.

3.2.3 Fifos compensation

To ensure a maximum rate, some compensation fifos are needed. The figure below shows a case requiring such compensation:

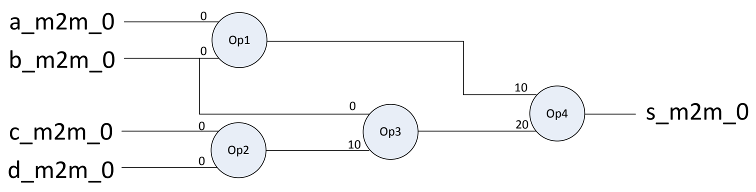


Figure 15: *Example of case requiring fifos compensation*

Assuming that all these additions have the same latency of 10, we can fix as above the latency of each entry. If one observes the two inputs of operator “Op3” and “Op4”, we notice that their latencies differ by a value of 10. These are two different cases:

- The case relative to “Op3” is the classic case where a multi-used signal is used by operators whose inputs have different latencies. It must then compensate for these latencies relative to the signal connected to the input of the lowest latency. In the case above the input who has the lowest latency is “a_m2m_0”. Input “b_m2m_0” connected to “Op3” must matched 10.
- The case relative to “Op4” is not related to multi-used inputs. It is present only when both inputs of an operator have different latencies. It must then check whether the entries sharing common signal and offset the difference in this case. On the figure above the inputs of “Op4” share the input “b_m2m_0”.

The figure below shows the same case with compensation fifos of depth 10:

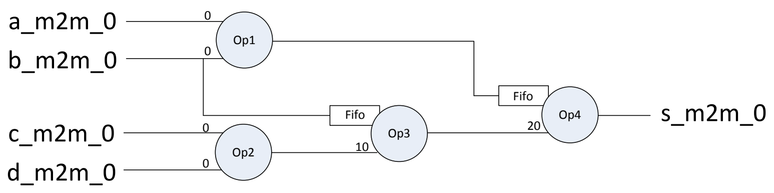


Figure 16: *Example of case with fifos compensation*

Regarding the calculation of latency inputs within the structure, it takes into account the following values:

- Latency signal whose input comes.
- Latency of the operator.
- Internal fifos located in the wrappers.

This principle can also be generalized to a higher number of multi-used variables. Note that if you want to send data at the same time, compensation fifos must be added as soon as the latencies of the two inputs are different. This choice can be made before the generation with a particular option of the tool. We will see later that for a loop, the inputs must always be served simultaneously. Then all these fifos are essential. Adding fifos is changing the size of internal fifos already present inside wrappers.

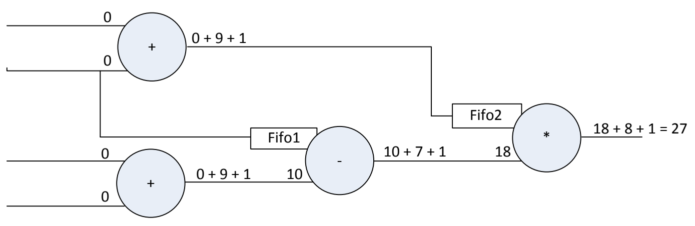
3.3 Latency time

The latency time of a polynomial can be directly linked to the previous point. In fact, it corresponds to the latency of the maximum output. Assuming that data is sent to each clock pulse, the time needed to calculate n data will be as follows:

$$latency\ time = number\ of\ data\ to\ send + maximum\ output\ latency\ time - 1$$

Consider the following scenario:

Number of data to send:	5000		
Latency time for «-» operator:	7		
Latency time for «*» operator:	8	Fifo1 size:	10
Latency time for «+» operator:	9	Fifo2 size:	8



Latency time = 5000 + 27 - 1 = 5026

Figure 17: *Example of latency time calculation*

The value of 1 is added to compensate the clock pulse required to write/read access in internal fifos inside wrappers. For the final calculation, the value of 1 is subtracted given that the clock pulse n°0 is the first. This result corresponds to the minimum latency that can be obtained.

3.4 Internal fifo justification

The internal fifos is fully justified when the inputs and outputs of two wrappers are linked. The following case from the dynamic view present this phenomenon:

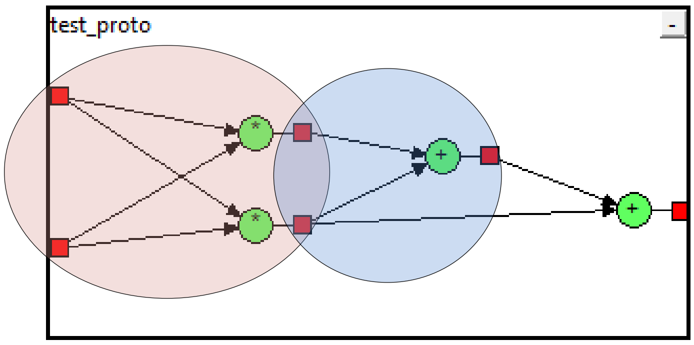


Figure 18: *Problematic case without internal fifos*

The right circle represents the dependence between the outputs of both wrappers multiplications. The left circle represents the strong dependence between the two inputs of the two wrappers. When a particular output is not ready to receive data, the absence of internal fifos create a combinational loop.

4. Conditional statement

4.1 Introduction

Conditional statements “if” echo many of the concepts covered during the generation of polynomials. Indeed, the bodies of “if” and all that surrounds them are constitutive of polynomial. Only the status of “if” has not been discussed previously. This section presents the details of managing the condition.

4.2 Condition signal

4.2.1 Principle

Consider the following octave code example to express a simple instruction conditional “if”:

```
if (a < c) then
  s = a-b;
else
  s = c-d;
endif
```

The following diagram represents the graphical representation of the code above. Surrounded operators show additions to a simple polynomial:

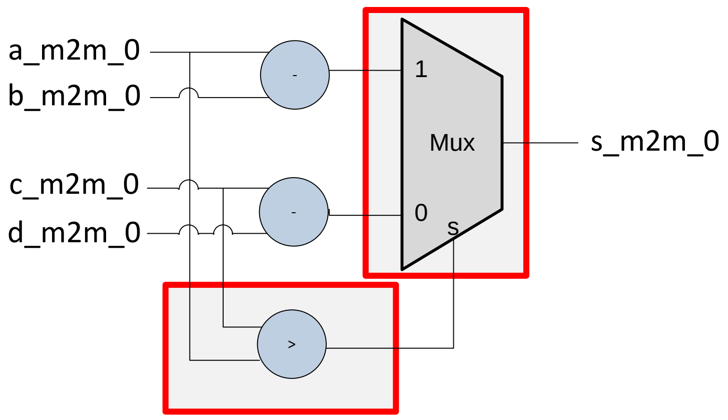


Figure 19: *Graphical representation of a conditional instruction “if”*

The first surrounded operator is a single operator “>”. Every operator is surrounded by a wrapper and has its own latency. This operation is identical to any block presented in the previous section. Note that the output of this operator has the particularity of being a single bit. The second surrounded operator is a multiplexer operator. It was the only operator consisting of three inputs. It is surrounded by a wrapper generalized to three inputs.

4.2.2 Latency

In the previous example and after setting all the latencies we obtain the following result:

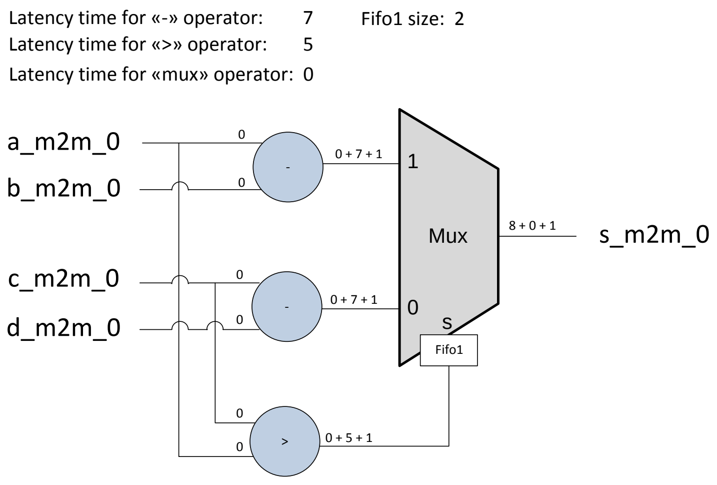

Latency time for «-» operator: 7
Latency time for «>» operator: 5
Latency time for «mux» operator: 0
Fifo1 size: 2
a_m2m_0
b_m2m_0
c_m2m_0
d_m2m_0
0
-
0 + 7 + 1
>
0 + 5 + 1
1
Mux
0
s
Fifo1
8 + 0 + 1
s_m2m_0

Figure 20: *Latency time of a conditional instruction "if"*

It is noted that only one fifo is necessary. It is determined simply by generalizing the principles previously seen but for a three inputs operator. In this example, we note that the input condition depends on “a_m2m_0” and “c_m2m_0”. The input “1” of multiplexer also depends on “a_m2m_0” and therefore implies a fifo. If no dependency was present between the different inputs, this fifo can be automatically set for allowing the sending of all input data at the same time.

5. For Loop

5.1 Introduction

The “for” loop are the most complex part of the generation. Indeed, its complexity is due to differences in the loop “octave” and its representation in the structure and the loop material to build. The purpose of this section is to show how the missing information in the structure was filled. A certain number of units developed for this purpose will therefore be presented.

5.2 Principle

To simplify the description of generation, we will break it down into several steps. At each stage a higher level of specification will be presented.

5.2.1 Desired rate

The idea is to circulate a data per clock cycle inside the loop and store each output data in a memory to the resort in order. In fact, the condition is unique to each data implying that came up after another in the loop can exit before. It is therefore necessary to buffer these data in a memory to guarantee an order consistent among the output data. A data can enter the loop only if a memory index is available. The diagram below illustrates this principle:

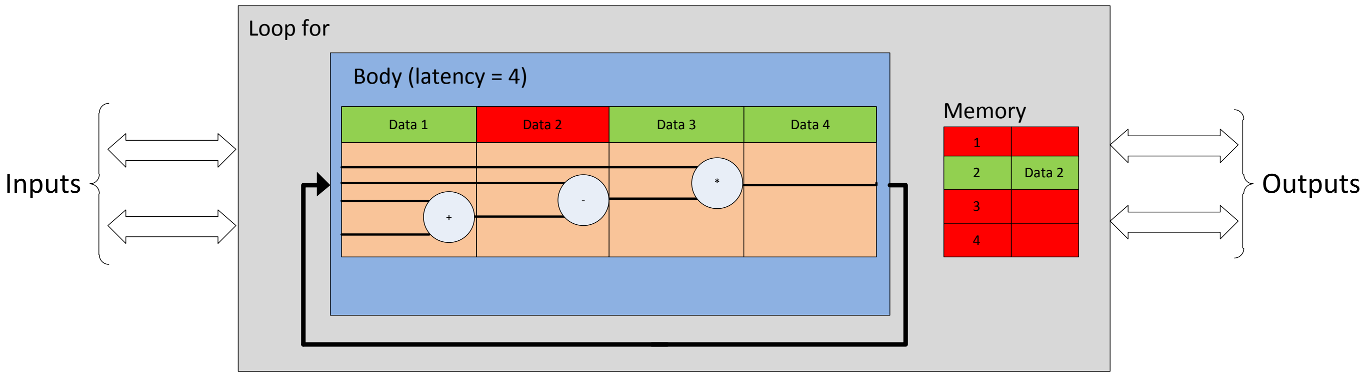


Figure 21: *example of desired rate*

The diagram above represents a loop with a latency of 4. Four data may buckle in both in the loop. Data is located on each floor of the body of the loop. The end condition of the second data has already occurred and the data was written in memory. It will exit the loop when the condition of the first data will be realized. Meanwhile the data continues to turn in the loop but its result is no more use.

5.2.2 First decomposition

A for loop consists of three distinct parts as presented in the following figure:

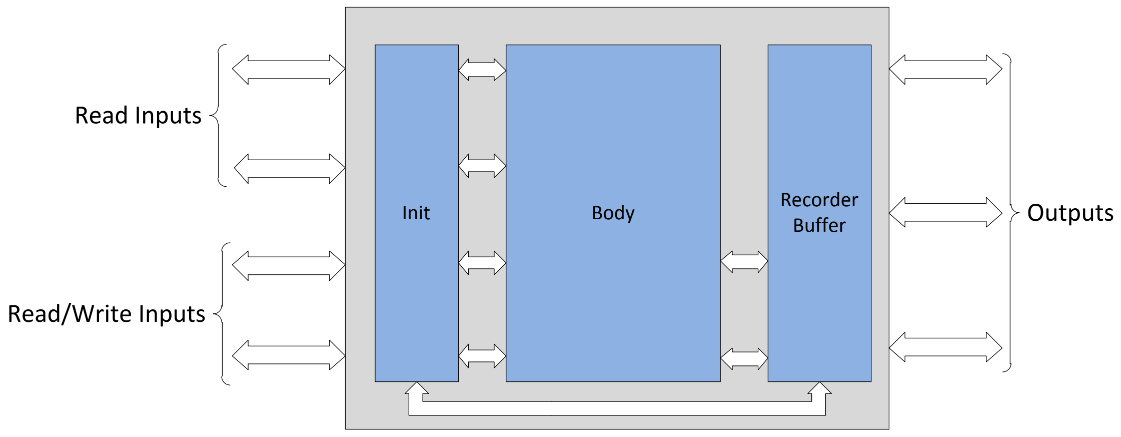


Figure 22: *First decomposition of a "for" loop*

- The first part allows the initialization and management of the loop inputs. There are two types of inputs. The inputs used only in reading in the loop and those used in reading and writing. The inputs used also in writing can be corresponding to an output of the loop.
- The second part represents the content of the loop and its end condition test. Unlike polynomial or conditional instruction "if", a number of additional synchronizations are required.
- The third and final part manages the results of the loop. We will see later that these are managed by a recording mechanism.

The parts are intimately linked. The remainder of this section presents their content and their interactions.

5.2.3 Second decomposition

Let's look specifically at blocks "init" and "memory" using the following figure:

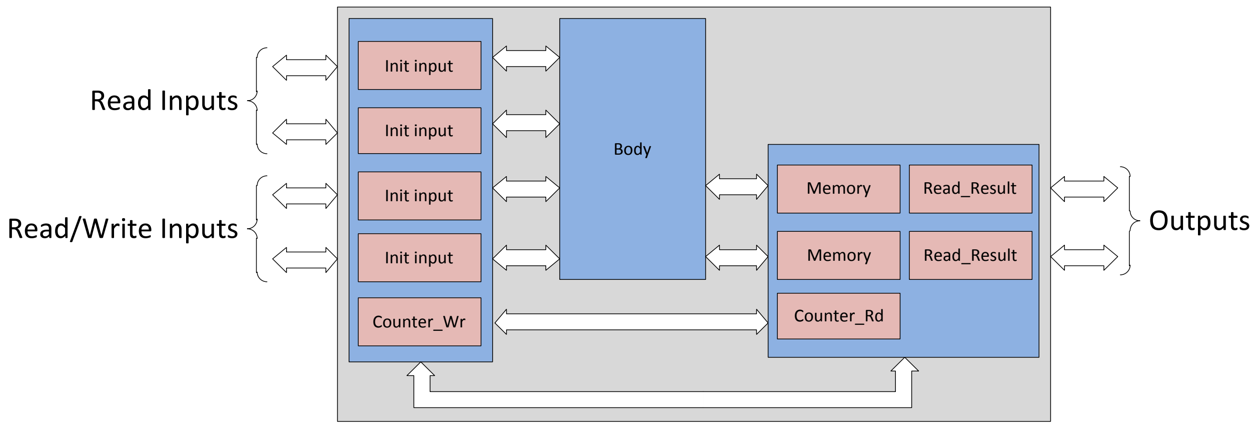


Figure 23: *Second decomposition of a "for" loop*

The block "Init" includes two different blocks:

- The “Init_Input” block allows to select the signals associated with its specific input or else those who are through the “for” loop. The loop condition associated with the loop data is used to perform this selection.
- The “Counter_Wr” block allows associating a number to each data in the loop. This number is the index for writing the data into the memory. When new data is ready to enter into the loop, a new number is assigned only if the maximum number of data in the loop and the memory is not reached.

The block “Memory” includes three different blocs:

- The “Memory” blocks memory stores the output data of the loop. Given that the data does not emerge automatically in the same order in which they came, this memory allows reordering the data.
- The "Read_Result" block manages the loop communication with the output signals. It allows reading a data in the memory and keeping the data on the output of the memory when an output is not ready to receive data.
- The “Counter_Rd” block provides the index of the next data to be read according to the size of the memory.

5.2.4 Third decomposition

This last decomposition highlights the paths of loop data, result data and read/write indexes. Depending on the type of data, different paths are used. There are two different paths for loop data:

- The first one is used by read input data. Since these data are not changed by the body of the loop, it is necessary to buckle them before the body. These data should also be delayed with fifos to compensate the latency of the body.
- The second one is used by read/write data. Since these data are modified by the loop body, it is necessary to buckle them after the body. Provided that all the outputs of the loop body have the same latency, it is not necessary to insert fifos. If their latencies are different they may be delayed into the body by internal fifos.

The path used by the output data may seem surprising. In fact, when the output occurs before the loop body, the condition is not yet calculated. Thus, we must delay the data and write into the memory only after the calculation of the condition. An additional complete iteration of the loop is therefore necessary to obtain the output data.

Each write index is associated with a read/write data and must buckle at the same time as its data. These indexes should also be delayed with fifos to compensate the latency of the body. A read index is only used when reading in memory. It is unique between two readings. He did not need to buckle.

The figure below shows the paths described above:

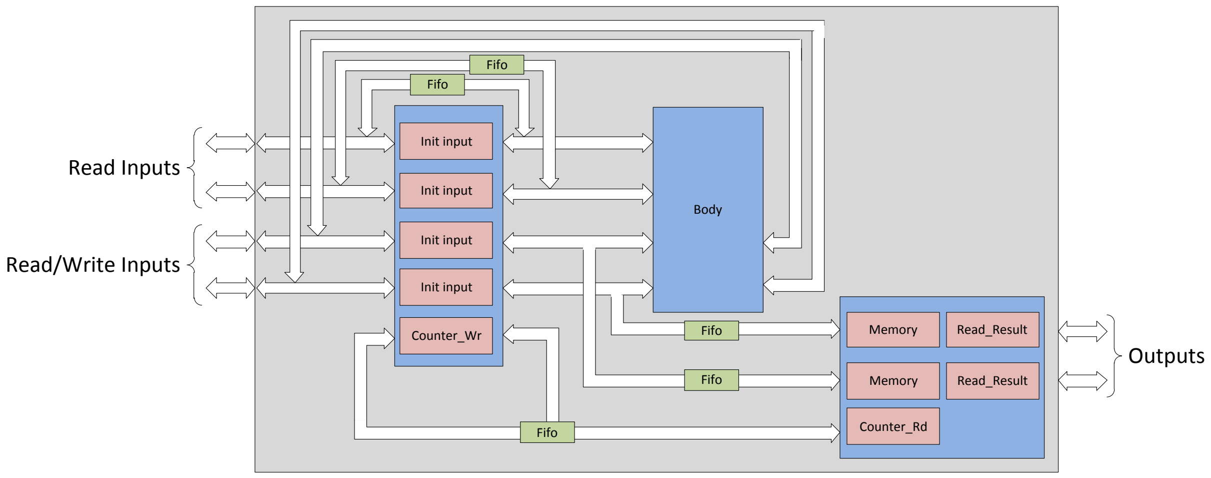


Figure 24: *Third decomposition of a "for" loop*

5.3 Body description

5.3.1 Connections

A number of connections are needed on the input signals from the body of the loop. The loop works correctly only if the input data is perfectly synchronized. With the help of an example with three inputs "a", "b" and "c", we will present the various interconnections made. To do this, let us analyze the following scheme:

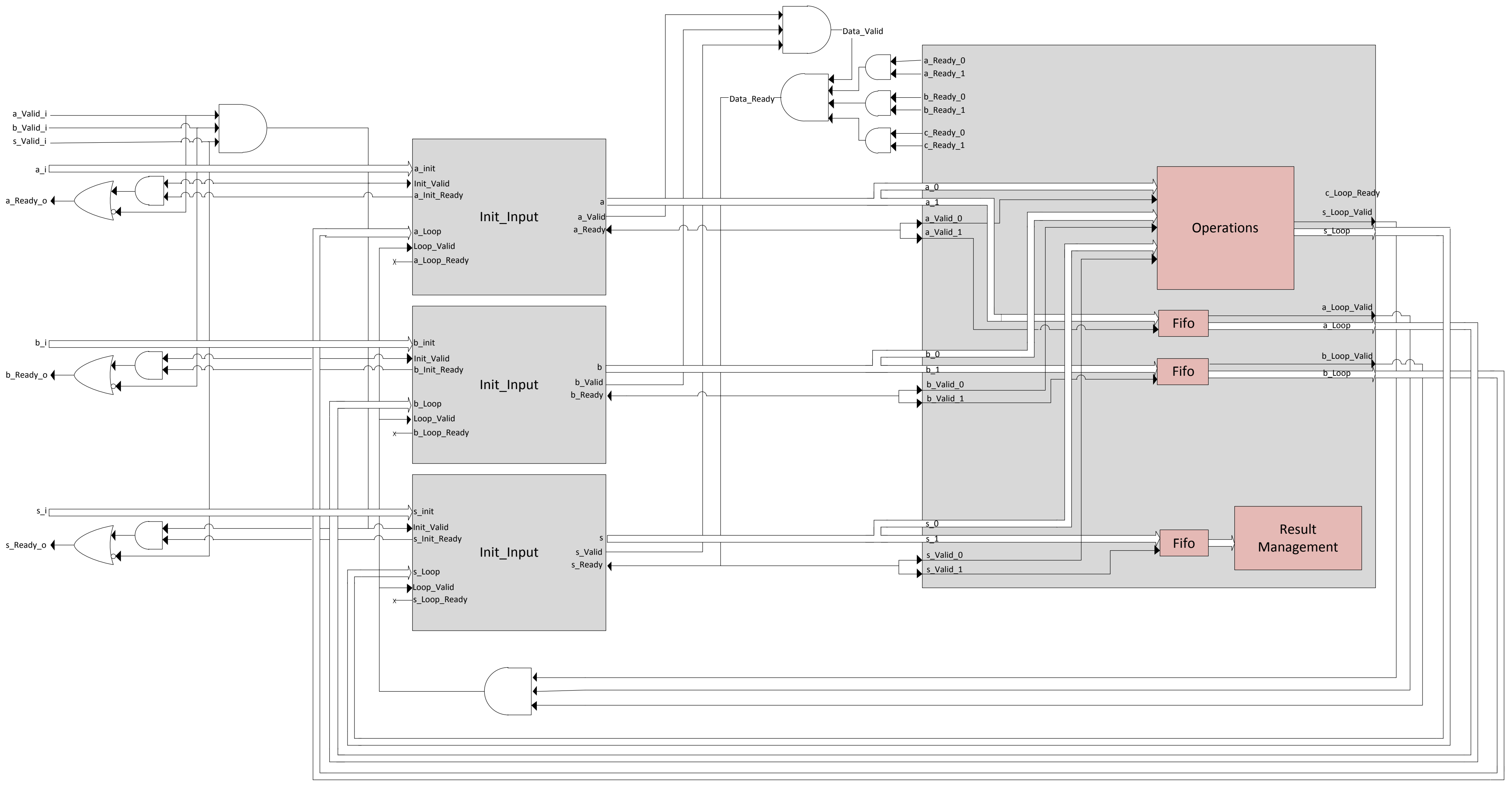


Figure 25: *Connections between inputs, "Init_Input" blocks and loop inputs*

All input “ready” and “valid” interconnections for a loop are presented above. Two outputs are used for each entry. The first is used by the content of the loop. The second is used as the loop variable. For a read/write input, the result of the loop body is used as a loop variable. Its current value is timed and recorded in memory if the end condition is reached. These mechanisms can be generalized to any number of inputs. All these connections can enter new data and turn data into the loop on all inputs simultaneously. Without these connections, one could accept an input data besfore the other involving a total desynchronization of the loop.

5.3.2 Iteration latency

Still not out of sync the data in the loop, all inputs of the operators contained in the loop body must have the same latency. Thus each latency difference between two inputs is compensated even if no dependencies between the inputs are present. In fact, all the inputs are dependent on each other in a loop. The following diagram shows a case of compensation for different latencies within a loop:

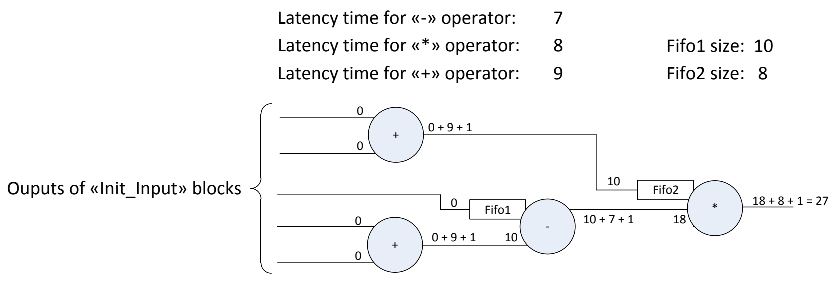

Latency time for «-» operator: 7
Latency time for «*» operator: 8
Latency time for «+» operator: 9
Fifo1 size: 10
Fifo2 size: 8
Ouputs of «Init_Input» blocks
0
0
+
0 + 9 + 1
10
Fifo2
*
18 + 8 + 1 = 27
0
Fifo1
-
10 + 7 + 1
18
0
0
+
0 + 9 + 1
10

Figure 26: *Iteration latency of a loop*

We obtain easily the maximum latency of the loop. This latency represents the maximum number of data in the loop. It therefore set the minimum size of memories, fifos and the number of read /write index.

5.3.3 Iterator and loop condition

The iterator and the loop condition are handled slightly differently:

- The iterator will always consist of an addition between the current value of the iterator and the increment parameter value. The current value of the iterator is its initial parameter value for the first iteration and its value for next iteration loop. Once the value for the next iteration calculated, it is necessary to delay the result by using a fifo. The iterator buckle with the data which it is associated. An “Init_Input” block is inserted to the iterator as if was a standard input.
- The condition is calculated from the current value of the iterator and the end parameter condition. It is also necessary to delay the result of the comparison by using a fifo. The result is different at each iteration and does not need to buckle with a specific data.

The following diagram shows the management of the loop condition and the iterator:

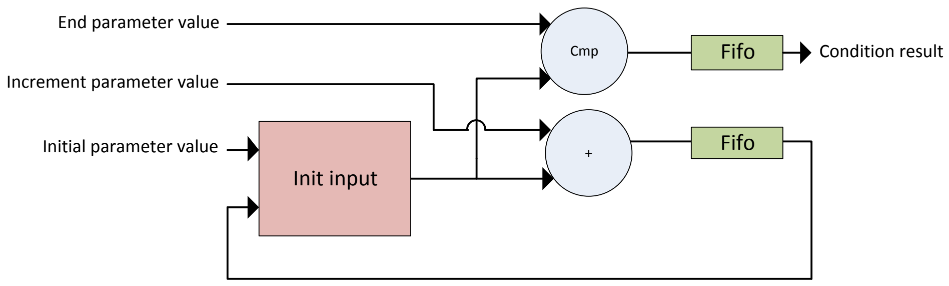


Figure 27: *principle of the loop condition the iterator*

5.4 Blocks description

5.4.1 Init_Input

This block takes as data two different data:

- An input from outside the loop representing the initial data. It is used once only during the first iteration.
- A data from inside the loop representing the loop data. It is recovered at the output of the loop body if it is an entry for reading and writing and entered the body if it is an input only used for reading.

The input "Cond_i" selects one of the two inputs. If the condition is equal to '1' then enter "Loop" is selected. Otherwise,the entry "Init" is selected. A similar logic is performed to the management of "ready" and "valid".

The diagram below shows the inputs / outputs of a block "Init_Input":

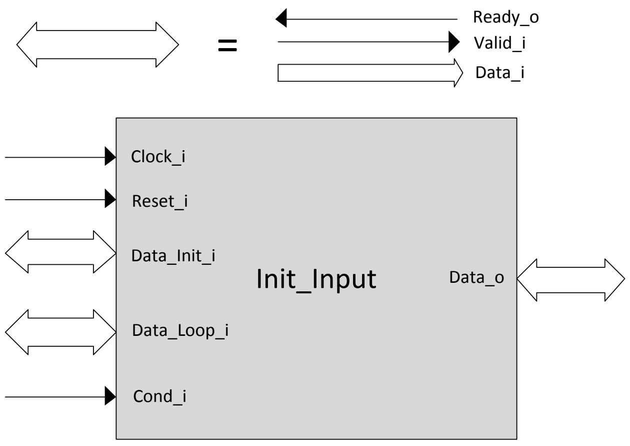


Figure 28: *"Init_Input" block*

5.4.2 Counter_Wr

As we have seen previously, the minimum size of the output memory can be sized using the latency of the loop body. From this value we can know the number of indexes needed to write in the loop. The challenge of this pack is to provide the next write index of the memory to new data. There may however be no index available. Indeed, if a maximum number of data is already in the loop or if a number of data have already been written but not read yet, no index should be reallocated.

Two vectors are used to indicate the index of reading and writing in the memory. The write index is updated at each writing event. The read index is received from the "Counter_Rd" block. Each write index is associated with a valid signal. Writing will actually occur when the validity signal of the write index is set to '1'. From these data and those provided by the loop body, this block can provide a valid write index and a signal "ready" indicating to the "Init_Input" block when it can send a new data. The write index and its validity signal buckle at the same time that the data on which they are associated. Input signals representing the write index and its validity signal indicate directly if the index is available or not.

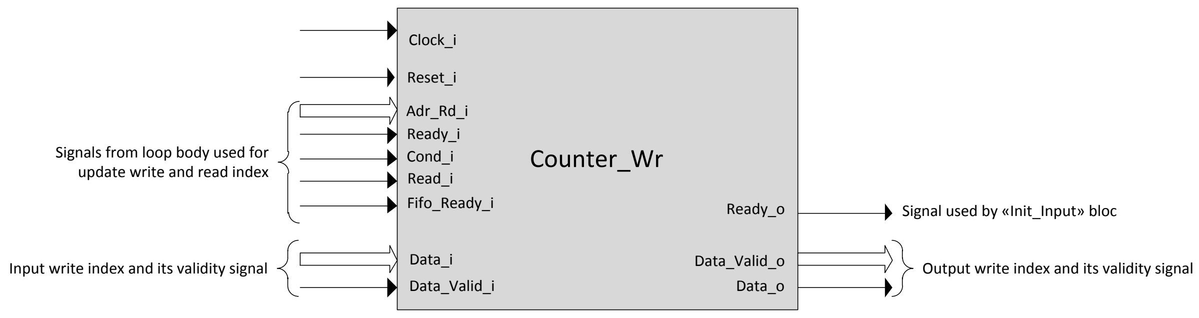

Clock_i
Reset_i
Adr_Rd_i
Ready_i
Cond_i
Read_i
Fifo_Ready_i
Counter_Wr
Ready_o
Signals from loop body used for update write and read index
Signal used by «Init_Input» bloc
Data_i
Data_Valid_i
Data_Valid_o
Data_o
Input write index and its validity signal
Output write index and its validity signal

Figure 29: *"Counter_Wr" block*

5.4.3 Counter_Rd

This counter is much simpler than "Counter_Wr" block. The "ready" signal comes from the outputs of the loop. It is active if all outputs are ready to be served. The "enable" signal indicates whether the data present at output of the memory are valid. Based on these two signals, the read index is incremented or maintained. The figure below describes the input/output block:

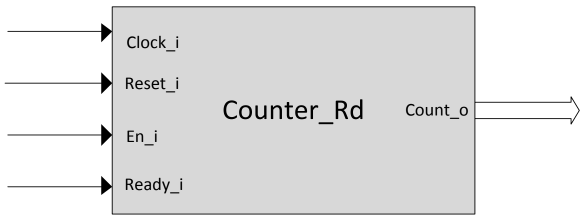


Figure 30 : *"Counter_Rd" block*

5.4.4 Memory

The minimum size of memory is set according to the maximum latency of the loop body. A memory is associated with each output data. The write address signal and its validity from the "Counter_Wr" bloc. A writing occurs when the write address is valid and the signal "Wr_i" is active. A reading is possible when all the outputs of the loop are ready to receive data. The output data from memory at a writing is accompanied by a valid signal.

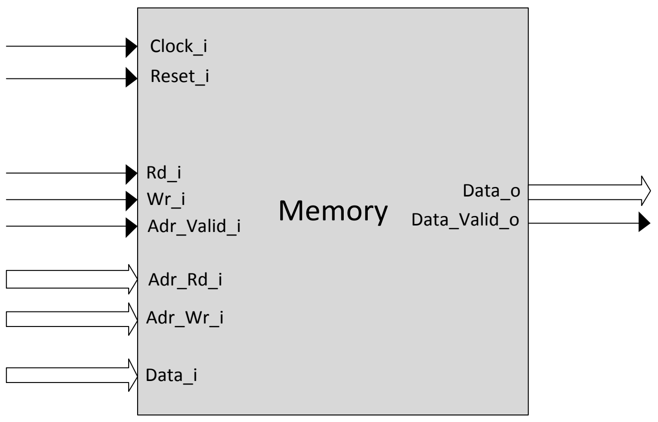


Figure 31: *“Memory” block*