Math2mat User Manual

HES-SO // Isys project

User manual of Math2mat GUI
Version: 1.0
Date of issue: May, 2011
Report originator: Yann Thoma
Contributors: Yann Thoma[1], Etienne Messerli[1], Michel Starkier[1], Sébastien Masle[1], Daniel Molla[1], Cédric Bardet[1], Grégory Trolliet[2], Claude Magliocco[3], Samuel Tâche[3], Philippe Crausaz[3], Christophe Bianchi[4], Oliver Gubler[4], Denis Prêtre[5]

[1]HEIG-VD
[2]hepia
[3]EIA-FR
[4]HES-SO // VS
[5]HE-Arc

Revision history

Version	Date	Description
1.0	18.05.2011	First draft

Contents

1 User manual **1**

1.1 Introduction . . . 1
1.1.1 Conventions . . . 1
1.2 Requirements . . . 2
1.2.1 Simulation . . . 2
1.2.2 Ubidule tests . . . 3
1.3 User's guide . . . 3
1.3.1 Project management . . . 3
1.3.1.1 Empty project creation . . . 3
1.3.1.2 Project creation from .m . . . 4
1.3.1.3 Folder validation . . . 4
1.3.2 Folders structure . . . 5
1.3.3 Mathematical grammar . . . 5
1.3.3.1 EBNF . . . 6
1.3.4 Views . . . 7
1.3.4.1 Editor . . . 7
1.3.4.2 Dynamic view . . . 8
1.3.4.3 Flat/tree view . . . 9
1.3.4.4 Messages view . . . 9
1.3.5 Code generation . . . 10
1.3.6 Verification . . . 10
1.3.6.1 Reference generation . . . 11
1.3.6.2 Simulation . . . 12
1.3.7 Ubidule tests . . . 12
1.3.8 External tools . . . 14
1.4 Tutorial . . . 15
1.5 Commands . . . 16
1.5.1 Menu File . . . 16
1.5.2 Menu Edit . . . 17
1.5.3 View . . . 18

1.5.4 Tools . . . 19
1.5.5 Configure... . . . 20
1.5.6 Help . . . 21

2 Blocks **22**
2.1 Introduction . . . 22
2.1.1 Purpose of the project . . . 22
2.1.2 IEEE 754 format . . . 22
2.1.2.1 Single precision format . . . 22
2.1.2.2 Rounding . . . 22
2.1.2.3 Exceptions . . . 23
2.1.3 Cell interface . . . 23
2.1.4 Floating point algorithms . . . 25
2.1.4.1 Addition / Subtraction . . . 25
2.1.4.2 Multiplication . . . 25
2.1.4.3 Division . . . 25
2.1.4.4 Square root . . . 26
2.2 VHDL description of the base cells . . . 26
2.2.1 Top level . . . 26
2.2.2 Pipe . . . 27
2.2.3 AddFP, MultFP, DivFP, SqrtFP . . . 27
2.2.3.1 Input exceptions . . . 28
2.2.3.2 Output exceptions . . . 29
2.2.3.3 Operation . . . 29
2.2.4 Compare block . . . 30
2.3 VHDL code organization . . . 30
2.3.1 Source files . . . 30
2.3.2 Version of the operators . . . 30
2.3.2.1 Add2 . . . 31
2.3.2.2 Mult2 . . . 31
2.3.2.3 Div2 . . . 31
2.3.2.4 Sqrt . . . 32
2.4 Synthesis results . . . 32
2.5 Square root algorithm . . . 35
2.5.1 Principle . . . 35
2.5.2 Non-restoring square root . . . 35
2.5.2.1 Example . . . 36
2.6 Blocks schematics . . . 38
2.6.1 Addition . . . 38
2.6.2 Multiplication . . . 42
2.6.3 Division . . . 47
2.6.4 Square root . . . 51
2.6.5 Comparison . . . 55

3 Generation **59**
3.1 Introduction . . . 59
3.1.1 Block representation . . . 59
3.1.2 Generic wrapper . . . 60
3.1.3 Utilization . . . 61
3.1.3.1 Timing diagrams . . . 61
3.1.4 Content block . . . 61
3.1.4.1 Usual content . . . 61
3.1.4.2 Propagation of ready . . . 62
3.2 Wrapper . . . 62
3.2.1 Introduction . . . 62
3.2.2 Implementation . . . 63
3.2.2.1 Internal fifo . . . 64
3.2.2.2 Combinational logic . . . 64
3.2.2.3 Synchronisation . . . 65
3.3 Mathematical function . . . 65
3.3.1 Introduction . . . 65
3.3.2 Multi-used signal . . . 66
3.3.2.1 Structure modifications . . . 66
3.3.2.2 Combinatorial logic . . . 67
3.3.2.3 Fifos compensation . . . 68
3.3.3 Latency time . . . 69
3.3.4 Internal fifo justification . . . 70
3.4 Conditional statement . . . 70
3.4.1 Introduction . . . 70
3.4.2 Condition signal . . . 70
3.4.2.1 Principle . . . 70
3.4.2.2 Latency . . . 71
3.5 For Loop . . . 72
3.5.1 Introduction . . . 72
3.5.2 Principle . . . 72
3.5.2.1 Desired rate . . . 72
3.5.2.2 First decomposition . . . 73
3.5.2.3 Second decomposition . . . 74
3.5.2.4 Third decomposition . . . 74
3.5.3 Body description . . . 75
3.5.3.1 Connections . . . 75
3.5.3.2 Iteration latency . . . 77
3.5.3.3 Iterator and loop condition . . . 77
3.5.4 Blocks description . . . 78
3.5.4.1 Init_Input . . . 78
3.5.4.2 Counter_Wr . . . 78
3.5.4.3 Counter_Rd . . . 79

3.5.4.4 Memory 79

4 Verification **81**
4.1 General Structure of the Testbench 81
4.2 Inputs/outputs of the DUT 81
4.3 Data File Structure 82
4.4 Channels Configuration 84
4.5 Stimuli Generator 84
4.6 Driver 84
4.7 Monitors 84
4.8 Data Checker 84
4.9 Timing Monitor 85
4.10 Testbench coverage 85
4.11 Reporting 86
4.12 Implementation 86
4.13 Format 87
4.14 Examples 87
4.15 Summary 87

List of Tables

2.1 Exceptions of IEEE 754 format 23
2.2 Synthesis results with the synthesizer XST (Xilinx) and the FPGA VirtexII xc2v1000-6bg575 33
2.3 Synthesis results with the synthesizer Precision and the FPGA Virtex5 5VLX110FF676. 34

4.1 Testbench parameters 89
4.2 Timing monitor statistics 90
4.3 Simulation message format 90

List of Figures

1.1 Math2mat graphical interface 4
1.2 Dynamic view showing the hardware structure 8
1.3 Internal structure corresponding to the operation $s = a + b$; 9
1.4 Simulation settings tab 11
1.5 Picture of a ubidule board (top and bottom) 12
1.6 Schematics of a ubidule 13
1.7 External tools properties dialog 14

2.1 Basic cell interface 24
2.2 Delay component 25
2.3 Mult2 top level interface 26
2.4 MultFP top level interface 27
2.5 Pipe top level interface 27
2.6 Pipe structure 56
2.7 Structure of an operation 57
2.8 Compare block interface 58

3.1 Generic Math2Mat block 59
3.2 Generic wrapper 60
3.3 Input tables representation 61
3.4 Output tables representation 61
3.5 Timing diagram of a usual case 62
3.6 Timing diagram of a non-usual case 63
3.7 Block content 64
3.8 Propagation of ready Timing diagram of a usual case 64
3.9 Block diagram of a wrapper 65
3.10 Wrapper for a basic operator 66
3.11 Structure state before generation 66
3.12 Structure state after generation 67
3.13 Combinational logic of multi-used signal 67
3.14 Principle of multi-used variables logic 68

3.15 Example of case requiring fifos compensation 68
3.16 Example of case with fifos compensation 69
3.17 Example of latency time calculation 69
3.18 Problematic case without internal fifos 70
3.19 Graphical representation of a conditional instruction if 71
3.20 Latency time of a conditional instruction if 72
3.21 Example of desired rate 73
3.22 First decomposition of a for loop 73
3.23 Second decomposition of a for loop 74
3.24 Third decomposition of a for loop 75
3.25 Connections between inputs, Init_Input blocks and loop inputs 76
3.26 Iteration latency of a loop 77
3.27 Principle of the loop condition the iterator 78
3.28 Init_Input block 78
3.29 Counter_Wr block 79
3.30 Counter_Rd block 80
3.31 Memory block 80

4.1 Structure of the testbench 82
4.2 Testbench class diagram 83

Chapter 1

User manual

1.1 Introduction

The aim of the Math2mat software is to automatically generate a VHDL description of a mathematical function using floating point operators. It also generates a full SystemVerilog testbench that allows to validate the hardware description by comparing it to the initial mathematical description. The initial description corresponds to the Matlab/Octave syntax, and allow operations such as addition, subtraction, multiplication, division, and structures such as if/then/else and for loops.

The main features of Math2mat are:

- Editor for typing Octave code
- Dynamic code analysis, during typing
- Generation of synthesizable VHDL code
- Floating point operations
- SystemVerilog testbench generation
- Fully automated test suite for the generated code
- Graphical interface with internal structure viewing

As the software is still under development, don't hesitate to submit a wish on the Wiki page of Math2mat:
http://www.tobecompleted.org

1.1.1 Conventions

The following conventions apply in this manual:

- Octave code is presented in this `format`
- VHDL code is presented int this `format`

- Menu selection is shown **This→Way**
- Files and folders are shown `/like/this`
- File content is shown in a box that also highlights the filename:

filename.ext

```
Various source code;
And more;
```

1.2 Requirements

Math2mat has been developped in Java, with the Eclipse framework. The release should be self-contained, and should run on Windows and Linux. The following systems have been tested:

- Windows XP
- Windows 7
- Linux Ubuntu 10.10

VHDL code can be generated without additional software.

The software allows to view the internal representation of the structure that will lead to the VHDL generation. This view, far from being user-friendly, can be useful for developers, and is generated using dot. This software can be downloaded from

http://www.graphviz.org/

1.2.1 Simulation

For validating the generated files, Octave and QuestaSim are required. Octave is an open-source software closely related to Matlab, as the accepted syntax is the almost the same. Octave can be downloaded from

http://www.gnu.org/software/octave/

QuestaSim is an HDL simulator, provided by MentorGraphics. It allows simulation of SystemVerilog designs and testbenches. It can be purchased from

http://www.mentor.com/

Math2mat has been validated with QuestaSim versions

- 6.5
- 6.6

Tests were run with these versions, but the system should work with further versions of QuestaSim. Previous versions of QuestaSim are not supported, as they do not fully accept SystemVerilog constructs. One of OVM or UVM methodology is also

required for correct simulation of the testbench. The files of the methodology are supplied by Math2mat, but the simulator needs to support these features.

Please not that for Linux users, dot and Octave can usually installed with the help of the software manager.

1.2.2 Ubidule tests

Math2mat allows to automatically generate a bitstream for a Xilinx Spartan3 embedded on a board called ubidule. The automatic bitstream creation requires Xilinx ISE to be installed. Tests have been conducted with version 12.2, but further versions should work. ISE can be retrieved by visiting the Xilinx website:

http://www.xilinx.com

1.3 User's guide

Math2mat provides a graphical interface as show in figure 1.1. Command line operations using a subset of the Java classes are also possible, and are documented in the developer guide.

1.3.1 Project management

A Math2mat project consists in a .m file that contains the mathematical source code, and in a certain number of parameters.

A project file has the extension "m2m", and contains a XML description of the entire project. This only file describing the entire project, the .m file is automatically regenerated by the tool if it is missing.

Standard Open, Save and Save as commands allow to open and save the project. Creation of a project can be executed in two ways: Through an empty project, or by importing a .m file.

1.3.1.1 Empty project creation

The creation of an empty project can be triggered through menu **File→Create empty project**. In that case, a template .m file is generated, and contains the following code:

<FileName>.m

```
% Here is a simple function that adds two numbers
% Feel free to modify it

function s=<FileName>(a,b)
    s=a+b;
endfunction
```

Where <FileName> is the name of the project.

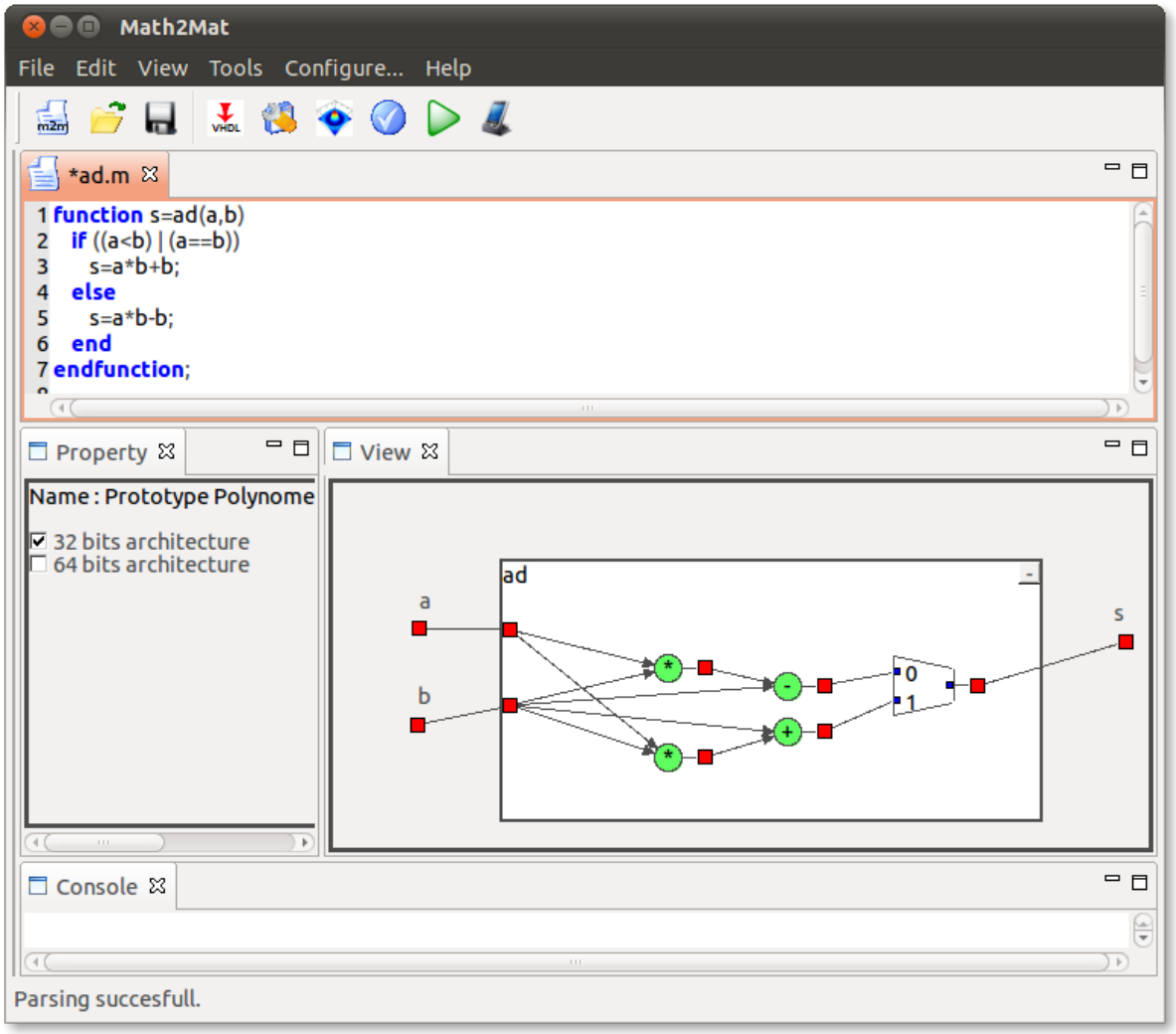


Figure 1.1: Math2mat graphical interface

1.3.1.2 Project creation from .m

The creation of a project from an existing .m file allows the user to select the name of the project, and then the .m file to import. The .m file is copied into the project directory, and so the original one won't be modified by Math2mat.

1.3.1.3 Folder validation

There is a possibility to launch the validation on a folder by selecting **Tools→Validate projects**. It recursively open every project found in the hierarchy, and launch the full verification process on the projects. At the end, a report in the console view indicates if errors were found.

1.3.2 Folders structure

During the creation of a new Math2mat project, a folder with the name `m2m_<funcName>` is created in the same directory as the project file (<funcName> is the function name). Inside this folder, a couple of other folders are created:

```
/
└─ m2m_<funcName>
   ├─ report ..... Contains the reports from simulation.
   ├─ src_VHDL ..... Contains all the VHDL files generated by the software
   │  ├─ wrappers ..... Contains generic wrappers for the generated system
   │  └─ M2M_Test ..... Contains VHDL files for the ubidule implementation
   ├─ src_SV ..... Contains all the SystemVerilog files generated for the
   │                simulation
   │  ├─ ovm ..... Contains modified OVM source files
   │  ├─ ovm_original Contains original OVM source files
   │  └─ uvm ..... Contains original UVM source files
   ├─ src_m2m ..... Contains the Octave source file
   ├─ src_OCT ..... Contains all the other Octave files needed for generat-
   │                ing the reference outputs
   ├─ sim ..... Contains the script that will be executed by QuestaSim
   │  │             for the simulation
   │  ├─ comp ..... Will contain the compiled simulation files
   │  └─ iofiles ..... Will contain the input and expected output data files
   │                   generated by Octave
   └─ ubidule ..... Contains the files needed for testing the system on a
                    real ubidule hardware.
```

1.3.3 Mathematical grammar

The source file of a Math2mat project consists in a file that must conform to the Octave language. To date, the parser accept a subset of the language :

- Calculus operator (+,-,*,/,$\sqrt{x}$) ;
- Logical operator (and, or, $=$, $\neq$, $<$, $>$, $\leq$, $\geq$) ;
- Loop "for" ;
- Conditional structure (if/then/else/elseif) .

A function accepts any number of inputs and outputs.
The following code illustrates a simple function with 2 inputs and 1 output:

example01.m

```
function s=example01(a,b)
    s = a*2;
    s = s+b;
endfunction
```

The following code illustrates the if/then/else structure:

example02.m

```
function s=example02(a,b)
    if (a<b)
        s = a*2+b*3;
    else
        s = a*2-b*3;
    end
endfunction
```

The following code illustrates a function with 2 inputs and 2 outputs, as well as a for loop:

example03.m

```
function [s1,s2]=example03(a,b)
    s1 = 0;
    for i=1:1:10
        s1=s1+a;
    end
    s2 = s1+b*4;
endfunction
```

1.3.3.1 EBNF

Here is the complete EBNF syntax supported by the parser.

```
entry          ::= { pramga } , function ;
affect         ::= ident , [ tab_ind ], '=' , expr_lvl1 , [ ';' ] ;
body           ::= { instruction } ;
cmp_op         ::= "==" | "!=" | '<' | '>' | "<=" | ">=" ;
expr_lvl1      ::= expr_lvl2, { ( '+' | ".+" ), expr_lvl2 } ;
expr_lvl2      ::= expr_lvl3, { ( '-' | ".-" ), expr_lvl3 } ;
expr_lvl3      ::= expr_lvl4, { ( '*' | ".*" ), expr_lvl4 } ;
expr_lvl4      ::= expr_lvl5, { ( '/' | "./" ), expr_lvl5 } ;
expr_lvl5      ::= func_var, { ( "**" | '^' ), func_var } | '(' expr_lvl1 ')' ;
func_param     ::= func_var, { ',' , func_var } ;
func_var       ::= ( [ '-' ], ( ident, [ '(', func_param, ')' ] | number )
                   | '[', [ '-' ], number, { ',', [ '-' ], number } ] ) ;
function       ::= function_begin , body , function_end ;
function_begin ::= "function" , ( '[', ident, { ',', ident } ']' | ident ),
                   "=", ident, '(' , param , ')' ;
function_end   ::= ( "end_function" , ident , ';' | "endfunction" [ ';' ] );
ifthenelse     ::= "if" , '(', logexpr, ')' , [ "then" ] , body ,
                   { "elseif", logexpr, body }, [ "else" , body ] ,
                   [ "end" | "endif" ] ;
ident          ::= char , { digit | char | '_' } ;
instruction    ::= affect | loop | ifthenelse | op_switch | not_instruction_cmd ;
logexpr        ::= logterm, [ log_op, logexpr ] ;
logterm        ::= logfactor, [ cmp_op, logterm ] ;
logfactor      ::= [ '!' ], ( func_var | '(', logexpr, ')' ) ;
log_op         ::= "or" | "and" ;
loop           ::= loop_begin , body , [ "end" | "endfor" ] ;
loop_begin     ::= 'for' , ident , '=' , func_var , ":" ,
                   func_var , [ ':' , func_var ] ;
```

```
not_instruction_cmd
                ::= ( "printf" | "error" ), '(', any_string, ')', [ ';' ] ;
number          ::= digit, { digit }, [ '.', digit, { digit } ] ;
op_switch       ::= "switch", func_var, { "case",
                    ( func_var | '[', func_var, { ',', func_var } ),
                    [ ',' ], body }, [ "otherwise", [ ',' ], body ], "end" ;
param           ::= ident , { ',' , ident } ;
pragma          ::= "%m2m" , ident , ':' , ident ;
tab_ind         ::= '(', number, ')' ;

char            ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
                    'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
                    'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
                    'v' | 'w' | 'x' | 'y' | 'z' |
                    'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
                    'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
                    'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' |
                    'V' | 'W' | 'X' | 'Y' | 'Z' ;
digit           ::= '0' | '1' | '2' | '3' | '4' |
                    '5' | '6' | '7' | '8' | '9' ;
```

1.3.4 Views

Different views are proposed to the user, the main one being the editor in which the mathematical code has to be written. From this code, a flat/tree view allows to visualize the internal representation of the function, and a dynamic view presents the hardware structure that can be generated.

1.3.4.1 Editor

A text editor allows to edit the mathematical code. It highlights the text following the Octave syntax.
The editor executes a live parsing every time the user modifies the text. The error messages are displayed at the bottom of the main view, and allow to rapidly verify the code. Moreover, if the dynamic view is shown, then the content of this view is dynamically modified if the code corresponds to a correct syntax.
Using the **Edit** menu, the following actions can be processed on the text:

- Execute Undo/Redo actions
- Cut the selected text and put it into the clipboard
- Copy the selected text and put it into the clipboard
- Paste the content of the clipboard
- Delete the selected text
- Select the entire text
- Toggle the comments on the selected text
- Comment and uncomment the selected text

1.3.4.2 Dynamic view

The dynamic view corresponding to the source code can be shown by selecting the menu **View→Dynamic view**. It then contains a graphical description of the hardware that will be generated. This view is dynamically regenerated every time the source code changes, i.e. every time the code is valid.
Through this view, moving the pointer on the graphical objects displays their properties in the Properties view.
The following code will create the view presented in figure 1.2.

test.m

```
function s=test(a,b,c)
   if (a>b)
      s=a-b+c;
   else
      s=b-a+c;
   end
endfunction
```

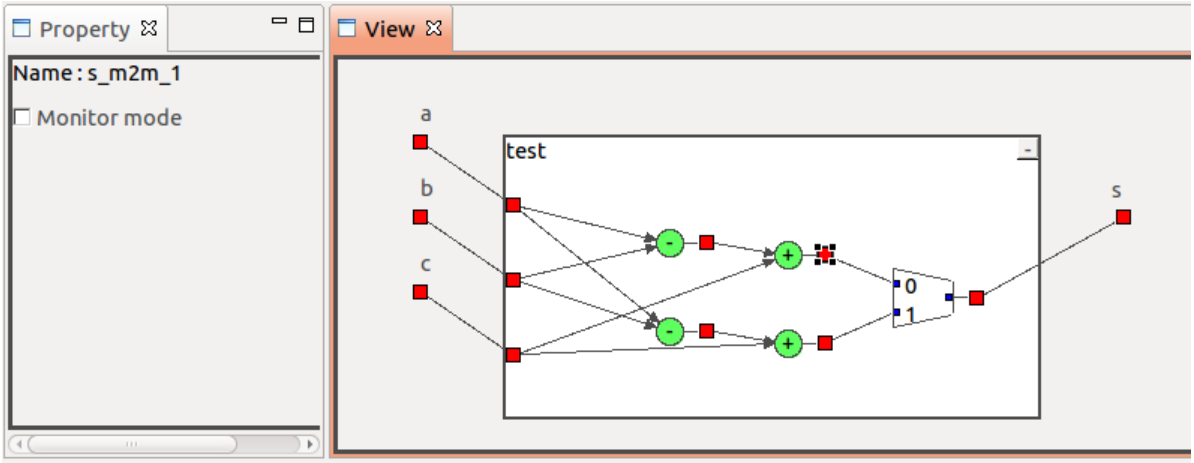


Figure 1.2: Dynamic view showing the hardware structure

Different properties are displayed and can potentially be changed, depending on the kind of object:

View outside function The type of floating point operations can be chosen: 32 or 64 bits.

Red square A red square represents an internal signal. It can be monitored or not. If checked, then the state of this signal is verified during simulation.

Operator An operator is a basic block that performs computation. Different implementations are proposed and can be chosen by the user. When selecting a block,

it is also possible to apply this change to all operators executing the same computation.

Multiplexer A multiplexer is used when if/then/else statements are present in the source code. Putting the pointer on a multiplexer highlights in blue the signals necessary for the condition calculation. It is also possible to monitor the condition of the multiplexer.

For increment In a "for loop" block, the increment is represented by a rectangle.

1.3.4.3 Flat/tree view

Math2mat offers the possibility to view the state of the internal structure. This structure is created during the code parsing, and is then manipulated in order to prepare the VHDL code generation. A standard user should note really need to open this view, letting this to developers. The tree view corresponds to the state of the structure after parsing, while the flat one represents its state before code generation.
These view requires "dot" software to be installed (cf. section 1.2, page 2). Math2mat generates a dot file called `fname_flat.dot` (or `fname_tree.dot`) that is then processed by dot to create a png file called `fname_flat.png` (or `fname_tree.png`) . This last file is then open by Math2mat. The *.dot and *.png files can be found in the `/src_m2m` folder. Figure 1.3 illustrates the schematics corresponding to the flat view of the following code:

adder.m

```
function s=adder(a,b)
   s=a+b;
endfunction
```

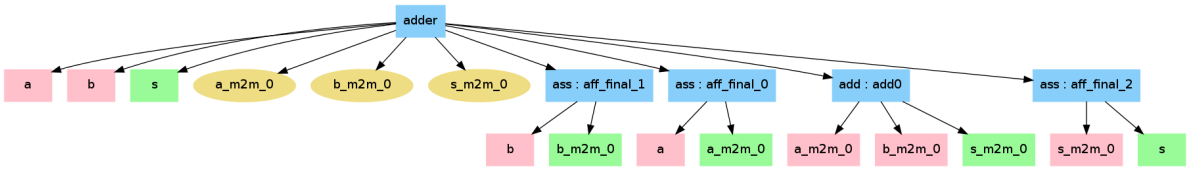


Figure 1.3: Internal structure corresponding to the operation $s = a + b$;

As the schematics generated can be very big, it is often easier to open them using a traditional image viewer instead of using Math2mat.

1.3.4.4 Messages view

At the bottom of the graphical interface, a message view corresponding to a console displays useful information in order to let the user know what happens.
This information depends on the type of processing the software is doing:

- **Editing**. Compilation errors or warnings
- **Octave code generation**. Messages about what files are created
- **Simulation**. Messages retrieved from QuestaSim
- **Ubidule testing**. Messages showing what happens during the synthesis, placement/routing processes, and during the test on the real hardware
- **Always**. Potential internal errors

1.3.5 Code generation

Code generation can be executed by selecting the menu **Tools→Generate VHDL**. The VHDL files are put in `/src_VHDL`. The description of the files structure and implementation can be found in another document.

1.3.6 Verification

Math2mat being able to generate VHDL code, checking that this code really does what it should allows to better trust the tool. For this purpose, simulation can be automatically launched in order to verify the generated system functionality. The testbench is generated in the folder `src_SV`, and is composed of SystemVerilog files. QuestaSim is required for the simulation of these SystemVerilog files, and is launched by Math2mat. The **Configure...→External Tools** menu allows to specifiy where the vsim executable is located, in case it is not accessible through the common path search. As Octave is also required to generate reference date files, it should also been reached by the common path or located through the same menu.
The verification is done in two phases:

1. Firstly reference data are generated by Octave;
2. Secondly SystemVerilog takes advantage of the references to simulate and verify the system behavior.

Simulation settings can be chosen through the menu **Configure→Simulation**. It opens a tab in which the settings can be entered. Figure 1.4 illustrates this view.
The following settings can be chosen:

- The calculation precision defines how error calculations are detected. It corresponds to the number of bits of the mantissa that can differ between the expected and the real results;
- The number of samples to test. One sample corresponds to a single call to the function;
- The system frequency is the frequency at which the system should work;
- The input frequency corresponds to the rate at which input data should be applied to the system;

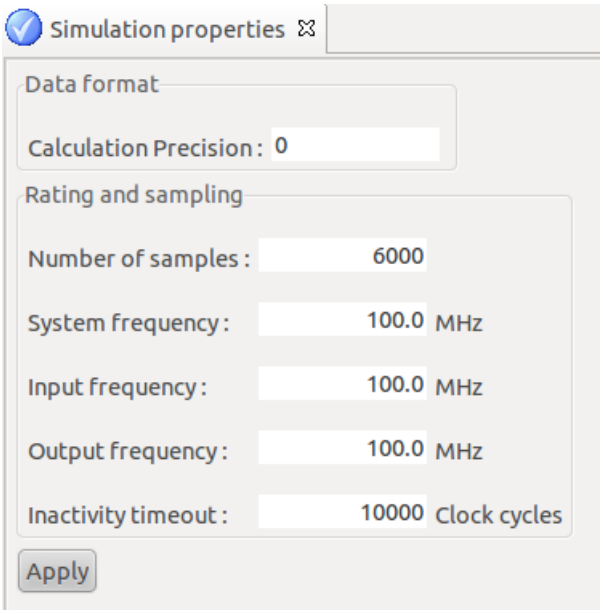


Figure 1.4: Simulation settings tab

- The output frequency corresponds to the rate at which output data should be retrieved;
- The period of inactivity allows to automatically end the simulation if no activity is detected on the output for a certain number of clock cycles.

The three frequencies are used in order to more or less stress the system. For each input of the system (each input variable), a valid input is applied at a random rate with probability $\frac{inputfrequency}{systemfrequency}$. For each output of the system (each output variable), a ready signal is applied at a random rate with probability $\frac{outputfrequency}{systemfrequency}$.
The new settings are set when pressing on the Apply button, and will also be stored in the project file.

1.3.6.1 Reference generation

Octave is used to generated files describing the inputs and files in which the reference output values are stored. All these files are stored in `/sim/iofiles`, and have names being `file_input1.dat`, `file_input2.dat`, ... for the input, and `file_output1.dat`, `file_output2.dat` for the ouptut. The number corresponds to the place of the input/output in the Octave source code. The calculation is performed using the source file edited in the Math2mat editor. However, if internal variables have to be monitored, then the octave file is regenerated in order to create data files for the internal variables. In that case, the menu **Tools→Test regenerated Octave code** allows to validate the regenerated code, ensuring that its functionality is the same as the original file.

1.3.6.2 Simulation

Simulation is handled by QuestaSim. QuestaSim is launched in batch mode and executes the script that was previously generated: `/sim/simulation.do`. The output of the simulation is displayed in the console view, the end of the message allowing to observe if the simulation went well or not. The last lines look like this:

Successful		Unsuccessful	
M2M_WARNING :	0	M2M_WARNING :	0
M2M_ERROR :	0	M2M_ERROR :	1000
M2M_FATAL :	0	M2M_FATAL :	0

If by any unlucky day the simulation does not go well, then QuestaSim can be used with its graphical interface. The user only needs to go to the `/sim` directory and to launch the script `simulation.do`.

1.3.7 Ubidule tests

Simulation allows to trust the generated files in terms of functionality. There are guaranties that the VHDL description is synthesizable, and tests with real hardware can be done. For this purpose, the software can validate the description using a ubidule (cf. figure 1.5).
This feature requires a ubidule to be set up with a server running. It is therefore not available to all users at any time.

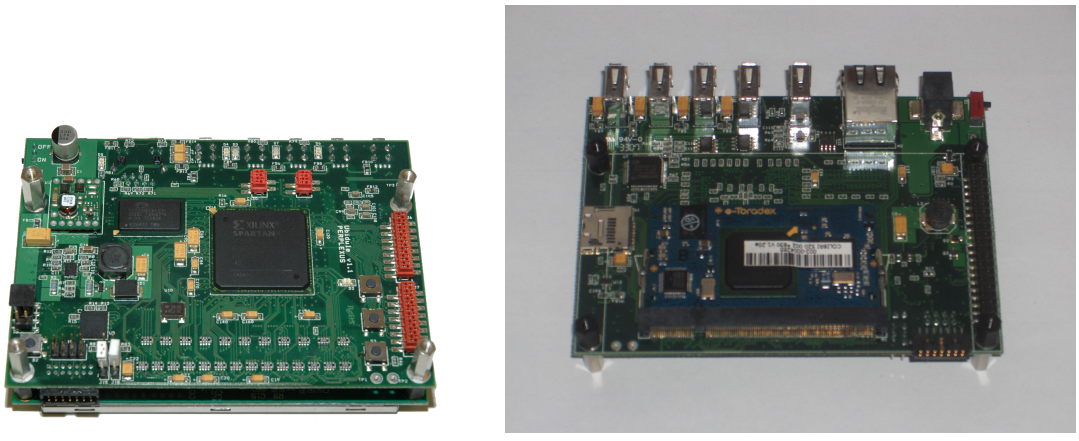


Figure 1.5: Picture of a ubidule board (top and bottom)

A ubidule embeds an ARM processor running embedded Linux, and a Xilinx Spartan3, as shown on figure 1.6.
In this context, Math2mat can automatically perform validation on the real hardware by performing the following actions:

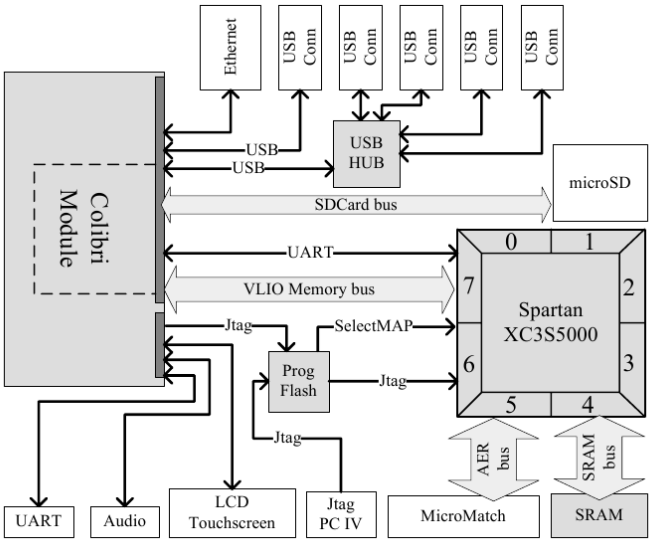


Figure 1.6: Schematics of a ubidule

1. Generate VHDL files that instantiate the mathematical function and adds everything needed to work on the board;
2. Generate scripts for launching the Xilinx tools
3. Launch Xilinx ISE for synthesis, placement and routing
4. Launch Xilinx Impact for generating the bitstream
5. Connect to a remote server that runs on the ubidule
6. Send the bitstream to the server
7. Send input data
8. Retrieve output data
9. Compare the calculation output to the expected ones

Through the GUI menus, the following actions can be launched:

- The menu **Tools→Ubidule test→Launch entire ubidule process** launches this entire process.
- The menu **Tools→Ubidule test→Launch ubidule files generation** launches steps 1 to 4.
- The menu **Tools→Ubidule test→Launch ubidule verification** launches steps 5 to 9.

The VHDL files for the ubidule implementation are generated in `/src_VHDL/M2M_Test/`, while the scripts are generated in `/ubidule/`.

1.3.8 External tools

As many different tools are required for the different actions the software can perform, a dialog allows to specify some information about these tools, as shown in figure 1.7.

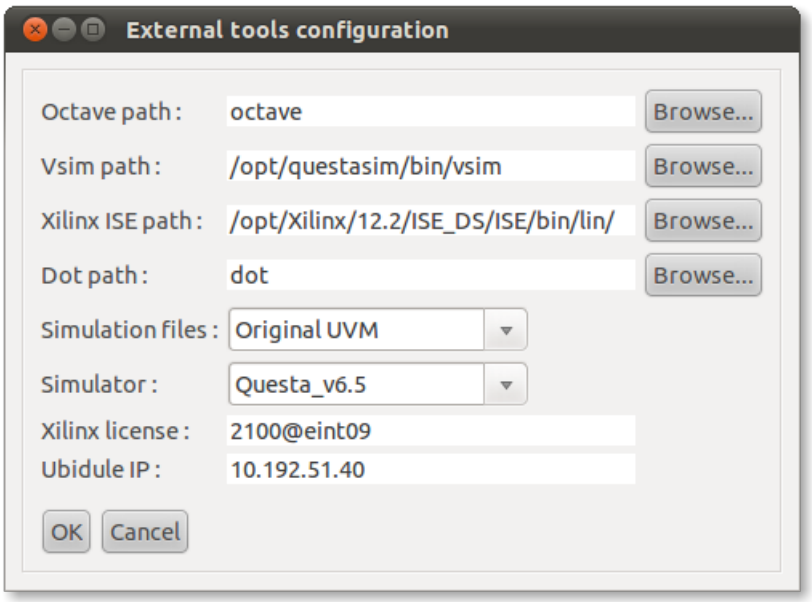


Figure 1.7: External tools properties dialog

As already mentioned, no external tool is required to simply generate a VHDL description. Octave and QuestaSim needs to be accessible for verification, Xilinx ISE only for ubidule tests, and dot for visualizing the internal structure.
For these four tools, by default their name is proposed by Math2mat. If they are not accessible through the path, then the user can specifiy the location of the executable, as shown in figure 1.7 for Vsim (the executable for QuestaSim).

⚠ For ISE, only the directory where ISE is installed should be set, not the executable itself.

Two combo boxes allow to specify general settings for the simulation. The simulation files lets the user choose to use OVM, UVM, or modified OVM files. The only changes during simulation are the way messages are displayed in the console. The prefix will be:

Simulation files	Prefix
OVM	`M2M_ERROR : 0`
Original OVM	`OVM_ERROR : 0`
Original UVM	`UVM_ERROR : 0`

If the user needs to perform simulation using the QuestaSim Gui by executing the script `/sim/simulation.do`, then it is better to use the original options, as the errors will be identified by QuestaSim.
The simulator allows to specify what version of QuestaSim is installed on the computer. Depending on the version (up to 6.5 or from 6.6), different SystemVerilog code is generated in order to attain internal signals in the VHDL hierarchy.
The last two fields are required for the ubidule tests. The license for the Xilinx tools has to be specified, as well as the IP address of the ubidule on which the testing server is running.

1.4 Tutorial

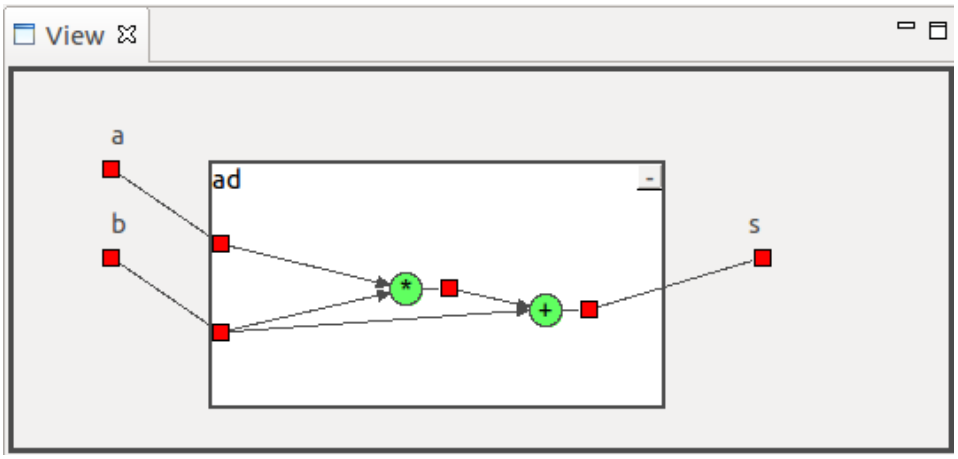
This tutorial shows you the steps required to create a new project, generate the VHDL code, and simulate it. If something goes wrong with the steps, it is probably because Math2mat does not find the external tools. In that case messages should appear, and you'll be able to specify their location with the menu **Configure...**→**External tools**.
So, here are the steps:

1. Select the menu **File**→**New empty project**;
2. In the dialog, choose the name of the project. You should create it in an empty directory, as it will be populated with the directories of Math2mat;
3. The new file contains a function that performs an addition;
4. Open the dynamic view (**View**→**Show dynamic view**) to observe the kind of digital system that will be generated;
5. Feel free to modify the function, but at least you can start with the simple adder;
6. To generate the files and simulate them, select the menu **Tools**→**Run all**, or click on the  icon;
7. And there you are... Too simple, no?
8. Add a little bit of complexity to the function, making it look like this:

<FileName>.m

```
function s=<FileName>(a,b)
   s=a*b+b;
endfunction
```

The dynamic view should now look like this:



9. Move the pointer onto different places of the dynamic view. You can observe the properties of the objects;
10. Click on one of the red squares, and select Monitor mode in the property view. It will then monitor this signal during the next simulation;
11. Click on the  icon to generate and simulate the new design.

1.5 Commands

This section lists all the commands accessible from the different menus. A reference allows to rapidly go to the section in which more details about the command can be found.

1.5.1 Menu File

File→New empty project Section 1.3.1, page 3

Create a new empty project. The user is asked to choose a directory and a name for the project. The .m file is automatically created and corresponds to the name of the project. A function template is also written into the file, in order the names to match. This function is

<FileName>.m

```
% Here is a simple function that adds two numbers
% Feel free to modify it

function s=<FileName>(a,b)
   s=a+b;
endfunction
```

Where <FileName> is the name of the project.

Create new project from .m file Section 1.3.1, page 3

Create a project with an initial source code file copied from an existing one.

Open Section 1.3.1, page 3

Open a Math2mat project file.

Save Section 1.3.1, page 3

Save the current project, as well as the .m file

Save as... Section 1.3.1, page 3

Save, with a new name, the current project.

Exit

Quit the application.

1.5.2 Menu Edit

Undo Section 1.3.4.1, page 7

Undo the last action.

Redo Section 1.3.4.1, page 7

Redo the actions that have been undone.

Cut Section 1.3.4.1, page 7

Cut the selected text and put it into the clipboard.

Copy Section 1.3.4.1, page 7

Copy the selected text into the clipboard.

Paste Section 1.3.4.1, page 7

Paste the content of the clipboard into the text editor.

Delete Section 1.3.4.1, page 7

Delete the selected text.

Select all Section 1.3.4.1, page 7

Select the whole text of the current editor.

Toggle comment Section 1.3.4.1, page 7

Toogle the comments for the selected lines.

Comment Section 1.3.4.1, page 7

Comment the selected lines.

Uncomment Section 1.3.4.1, page 7

Uncomment the selected lines.

1.5.3 View

Show the flat view Section 1.3.4.3, page 9

Show the flat view of the internal structure in the form of a .png file loaded in the GUI.

Show the tree view Section 1.3.4.3, page 9

Show the tree view of the internal structure in the form of a .png file loaded in the GUI.

Show the dynamic view Section 1.3.4.2, page 8

Show the dynmic view corresponding to the mathematical code.

Clear the console

Simply clear the console view.

1.5.4 Tools

Generate VHDL Section 1.3.5, page 10

Generate the VHDL files corresponding to the Octave code.

Launch test generation (with Octave) Section 1.3.6.1, page 11

Generate the input/output data files by launching Octave.

Launch Verification Section 1.3.6, page 10

Start the verification using QuestaSim. It requires the VHDL and the input/output files to be up to date.

Run all Section 1.3.6, page 10

Run the entire flow: VHDL generation, input/output files generation, and simulation.

Test regenerated Octave code Section 1.3.6.1, page 11

This command allows to verify that the regenerated Octave code executes the same functionality as the original one.

Validate Projects Section 1.3.1.3, page 4

Allow to recursively validate all the projects found in a directory.

Launch ubidule files generation Section 1.3.7, page 12

Launch the ubidule files generation:

1. Generate VHDL files that instantiate the mathematical function and adds everything needed to work on the board;
2. Generate scripts for launching the Xilinx tools
3. Launch Xilinx ISE for synthesis, placement and routing
4. Launch Xilinx Impact for generating the bitstream

Launch ubidule verification Section 1.3.7, page 12

Launch the ubidule verification, taking into account that the files have been previously generated:

1. Connect to a remote server that runs on the ubidule
2. Send the bitstream to the server
3. Send input data
4. Retrieve output data
5. Compare the calculation output to the expected ones

Launch entire ubidule process Section 1.3.7, page 12

Launch the entire ubidule test process:

1. Generate VHDL files that instantiate the mathematical function and adds everything needed to work on the board;
2. Generate scripts for launching the Xilinx tools
3. Launch Xilinx ISE for synthesis, placement and routing
4. Launch Xilinx Impact for generating the bitstream
5. Connect to a remote server that runs on the ubidule
6. Send the bitstream to the server
7. Send input data
8. Retrieve output data
9. Compare the calculation output to the expected ones

1.5.5 Configure...

External tools Section 1.3.8, page 14

Configure the external tools properties.

Simulation Section 1.3.6, page 10

Modify the settings for the simulation.

Optimization Section 1.3.5, page 10

Modify the settings for the optimization.

1.5.6 Help

About Math2mat

A standard About dialog, that displays the version number, as well as the way to to contact the developing team and some information about Math2mat.

Key assist...

Allow the user to view the shortcut keys.

Chapter 2

Blocks

2.1 Introduction

2.1.1 Purpose of the project

The base cells of Math2mat project are to perform basic operations such as +, -, *, /, sqrt. The project's goal is to optimize the speed of operations, that's why a pipelined version of these operators should be considered. Cells must manipulate floating point numbers. IEEE standard 754 has been imposed (single and double precision).

2.1.2 IEEE 754 format

IEEE 754 is a standard for the description of floating point numbers. Single precision format includes 32 bits, 1 for the sign, 8 for the exponent and 23 bits for the mantissa. Double precision format includes 64 bits, 1 for the sign, 11 for the exponent and 52 bits for the mantissa.

2.1.2.1 Single precision format

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
s	exponent	mantissa

With the standard IEEE 754, the equation 2.1 allows to compute the real number.

$$number = (-1)^s * 1.mantissa * 2^{(exponent-127)} \tag{2.1}$$

2.1.2.2 Rounding

The IEEE 754 defines four rounding modes :

- Round toward $+\infty$
- Round toward $-\infty$

- Round toward 0
- Round to nearest

For the math2mat project, we only implemented one rounding mode, round to nearest. This rounding is done by checking the value of the 24th bit generated during a computation of the mantissa in single precision. If this bit is '1', the mantissa is incremented by '1', otherwise she's unchanged.
Example :

$$1.1100...101 \xrightarrow{round} 1.1100...11$$
$$1.1100...100 \xrightarrow{round} 1.1100...10$$

2.1.2.3 Exceptions

Type	Exponent	Mantissa
Zeros	0	0
Denormalized numbers	0	different to 0
Normalized numbers	1 to $2^e - 2$	any
Infinity	$2^e - 1$	0
NaNs	$2^e - 1$	different to 0

Table 2.1: Exceptions of IEEE 754 format

Math2mat project doesn't handle the denormalized numbers described by the IEEE 754 standard.

2.1.3 Cell interface

The interface of base cells (+, -, *, / and sqrt) was defined as follows (for square root, only one input of data) :

Signal	: Description
D1_i	: Data 1 (IEEE 754 format).
D2_i	: Data 2(IEEE 754 format).
valid_i	: When high, the inputs are valid.
clk_i	: System clock.
reset_i	: System reset.
stall_i	: When high, stop the operation (!enable).
m_g	: Generic constant, defines the computation in single (m_g=32) or double (m_g=64) precision.
pipe_g	: Generic constant, defines the combinatorial (= 0) or pipelined (!= 0) version.
latency_g	: Generic constant, defines the duration of the operation for the combinatorial version.
type_g	: Generic constant, defines the algorithm used for the operation.
wValid_g	: Generic constant, defines the width of the signal valid_i
m_o	: Result of the operation in floating point (IEEE 754 format).
valid_o	: When high, the result is valid.
ready_o	: When high, a new computation can be done.

Figure 2.1 illustrates the ports of a basic cell.

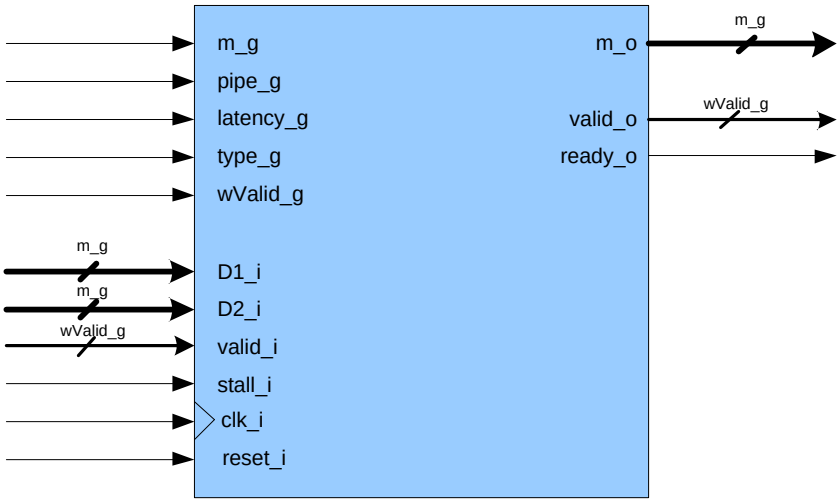


Figure 2.1: Basic cell interface

Even if the component allows to set generically the width of data bus, some algorithms have so far been implemented in single precision, ie for a fixed bus width of 32 bits.

For some operation, it's useful to have a cell, which one must delay a computation that must be synchronized with another, so the next block was created. It's a shift register (delay_g = nb of registers) where each output m_o(i) has its validation

signal `valid_o(i)`, as shown in figure 2.2.

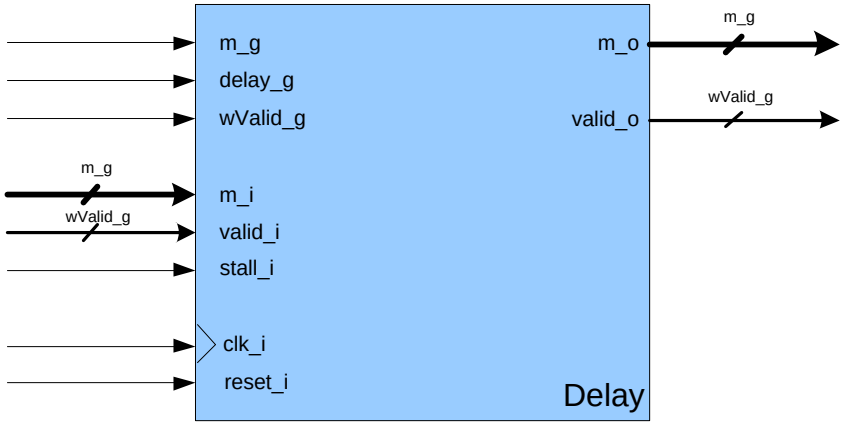


Figure 2.2: Delay component

2.1.4 Floating point algorithms

2.1.4.1 Addition / Subtraction

1. Take the largest exponent (= exponent of the result).
2. Shift the mantissa of the smallest number, the gab is the value of the difference of the exponents.
3. Addition / Subtraction of the mantissas.
4. Normalization of the result : mantissa between $[1 : 2[$, modification of the exponent if necessary
5. Round to nearest.

2.1.4.2 Multiplication

1. Addition of the exponents.
2. Multiplication of the mantissas.
3. Round to nearest.
4. Normalization of the result : mantissa between $[1;2[$, modification of the exponent if necessary.

2.1.4.3 Division

1. Subtraction of the exponents.
2. Division of the mantissas.

3. Round to nearest.
4. Normalization of the result: mantissa between$[1;2[$, modification of the exponent if necessary.

2.1.4.4 Square root

1. Division by 2 of the exponent.
2. Square root of the mantissa.
3. Round to nearest.
4. Normalization of the result : mantissa between $[1;2[$, modification of the exponent if necessary.

2.2 VHDL description of the base cells

2.2.1 Top level

The top level of a base cell (Add2, Mult2 (cf. figure 2.3), Div2 ou Sqrt) is a structural description composed of 2 or more components :

1. **AddFP**, **MultFP** (cf. figure 2.4), **DivFP**, **SqrtFP** : floating point computation (combinatorial or pipelined).
2. **Pipe**(cf. figure 2.5) : shift register used in various cases.
3. *(AddFP_1, MultFP_1,...) : another algorithm. If so, the first algorithm becomes AddFP_0, MultFP_0.*

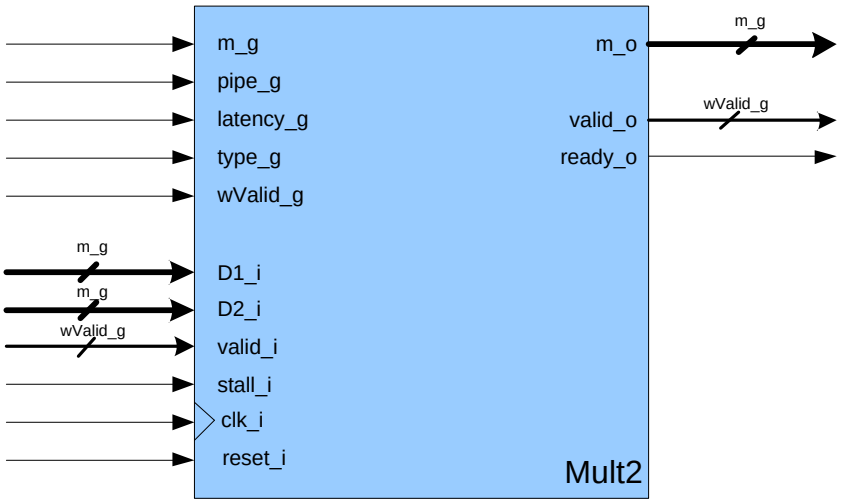


Figure 2.3: Mult2 top level interface

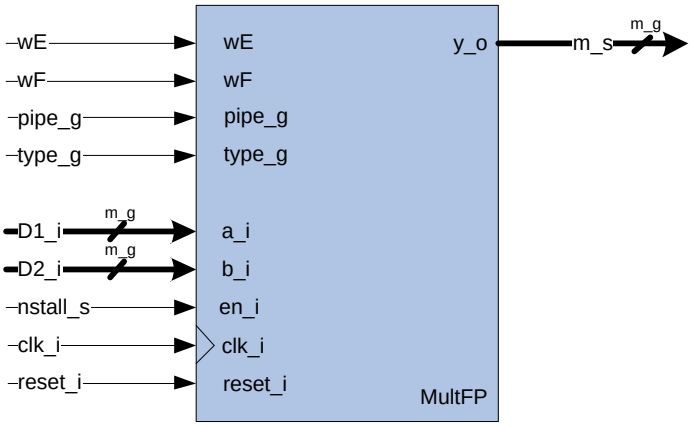


Figure 2.4: MultFP top level interface

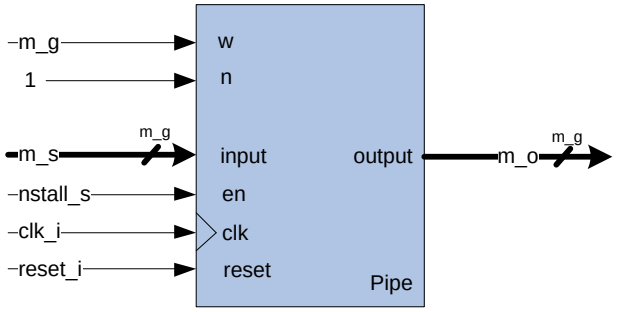


Figure 2.5: Pipe top level interface

In figures 2.3, 2.4 and 2.5, wE and wF generic constants define the width of exponent(wE) and mantissa(wF, F for Fraction).

2.2.2 Pipe

A component Pipe was created to generate a variable shift register with a variable width bus. Its structure is shown in figure 2.6.

2.2.3 AddFP, MultFP, DivFP, SqrtFP

The operation in floating point was implemented as follow (cf. figure 2.7 :

- The input signals are separated in sign, exponent and mantissa.
- The operation is executed in combinatorial or pipelined version. The depth of the pipeline depends on the operation.
- In parallel, a test of inputs is executed to check the exceptions values . The exception is propagated to output to force the result.

- Another block determines if an exception occurred during the computation and force the result in output if necessary.

2.2.3.1 Input exceptions

Addition/Subtraction

	Nb	0	$+\infty$	$-\infty$	NaN
Nb	Nb	Nb	$+\infty$	$-\infty$	NaN
0	Nb	0	$+\infty$	$-\infty$	NaN
$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN	NaN
$-\infty$	$-\infty$	$-\infty$	NaN	$-\infty$	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Multiplication

	Nb	0	$+\infty$	$-\infty$	NaN
Nb	Nb	0	$+\infty$	$-\infty$	NaN
0	0	0	NaN	NaN	NaN
$+\infty$	$+\infty$	NaN	$+\infty$	$-\infty$	NaN
$-\infty$	$-\infty$	NaN	$-\infty$	$+\infty$	NaN
NaN	Nb	Nb	NaN	NaN	NaN

Division

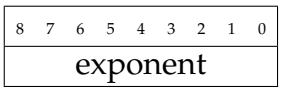
	Nb	0	$+\infty$	$-\infty$	NaN
Nb	Nb	NaN	0	0	NaN
0	0	NaN	0	0	NaN
$+\infty$	$+\infty$	NaN	NaN	NaN	NaN
$-\infty$	$-\infty$	NaN	NaN	NaN	NaN
NaN	Nb	Nb	NaN	NaN	NaN

Square root

A	$\sqrt{A}$
Nb	Nb
0	NaN
$+\infty$	$+\infty$
$-\infty$	0
NaN	NaN

2.2.3.2 Output exceptions

During an operation, the result may be outside the limits of the standard IEEE 754 single or double precision. In this case, the result must be limited to values 0 or ∞. For the single precision, the computation of exponents is realized on 9 bits (1 bit for carry). The treatment of exceptions is done as following :



1. exponent = "011111111" $\Rightarrow$ result = ∞
2. exponent= "000000000" $\Rightarrow$ result = 0
3. bit 8 and 7 = "10" $\Rightarrow$ result = ∞
4. bit 8 and 7 = "11" $\Rightarrow$ result = 0

Explications point 3 and 4 :

- The largest possible result is obtained by multiplying two numbers with the exponent 011111110". During this operation, the exponent becomes :

$011111110 + 011111110 - 1111111 = 101111101$
This example shows that when the result is too large and must apply the infinite, the two most significant bits of the exponent are "10".

- When the division of a small number to a large one, the exponent result is :

$000000011 - 011000000 + 1111111 = 111000010$
This example shows that when the result is too large and must apply the infinite, the two most significant bits of the exponent are "11".

2.2.3.3 Operation

A detailed description of the various operators are found at the end of this chapter, in the form of schematics:

- Addition: section 2.6.1, page 38
- Multiplication: section 2.6.2, page 42
- Division: section 2.6.3, page 47
- Square root: section 2.6.4, page 51
- Comparison: section 2.6.5, page 55

2.2.4 Compare block

For the if...else implementation, we need to compare two values and know which number is bigger, smaller or if the two signals are equal. This block, completely combinatorial, performs this operation. It also includes the component pipe to propagate the signal `valid_i` to indicate when the result is valid (depending on the latency). Figure 2.8 shows the interface of this block.

2.3 VHDL code organization

2.3.1 Source files

Files name :

- A file with `pkg_` is a package.
- A file with `_tb` is a testbench.
- A file without this two terms is a source code.

Description :

`misc.vhd`	: this file contains the component pipe. If another useful component must be created, insert it in this file.
`delay.vhd`	: cell delay to delay an output and its validation signal.
`add2.vhd`	: cell addition.
`mult2.vhd`	: cell multiplication.
`div2.vhd`	: cell division.
`sqrt.vhd`	: cell square root.
`pkg_definition.vhd`	: Constants and functions declaration.
`pkg_cellule.vhd`	: components add2, mult2, div2, sqrt and delay declaration.

Compilation script:
A script compile.do was realized to execute the compilation of source files in the correct order. This file is in the folder `/do`.

2.3.2 Version of the operators

The version of the operator is determined using the generic constants of the component. The generic constant `type_g` is used to determine the algorithm. Only the addition is implemented with one algorithm and therefore doesn't have this constant. The generic constant `pipe_g` determines the combinatorial or pipelined version. `latency_g` constant determines the latency time (number of clock period)

before the output is stable for the combinatorial version. And finally the generic constant `m_g` choose the single or double precision (be careful, not all algorithms are implemented in double precision).
The algorithms SRT4 for the division and SRT2 for the square root were taken from the library FPLibrary directed at ENS Lyon.
`http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/`

2.3.2.1 Add2

3 generic constants : `m_g`, `pipe_g` et `latency_g`

- `m_g` = 32 for single precision and `m_g` = 64 for double precision
- `latency_g` = 0 and `pipe_g` != 0 , pipelined version
- `pipe_g` = 0, combinatorial version, `latency_g` determines the time (number of clock period) before the output is stable.

2.3.2.2 Mult2

4 generic constants : `m_g`, `pipe_g`, `latency_g` and `type_g`

- `m_g` = 32 for single precision and `m_g` = 64 for double precision (double only for wired multiplier).
- `type_g` = 0 : multiplier carry save adder
  - `pipe_g` != 0 and `latency_g` = 0, pipelined version.
  - `pipe_g` = 0, combinatorial version, `latency_g` determines the time (number of clock period) before the output is stable.
- `type_g` = 1 : wired multiplier
  - `pipe_g` != 0 and `latency_g` = 0, pipelined version.
  - `pipe_g` = 0, combinatorial version, `latency_g` determines the time (number of clock period) before the output is stable.

2.3.2.3 Div2

4 generic constants : `m_g`, `pipe_g`, `latency_g` and `type_g`

- `m_g` = 32 for single precision and `m_g` = 64 for double precision (double only for SRT4 algorithm).
- `type_g` = 0 : SRT4 algorithm
  - `pipe_g` != 0 and `latency_g` = 0, pipelined version.

  - `pipe_g` = 0, combinatorial version, `latency_g` determines the time (number of clock period) before the output is stable.

- `type_g` = 1 : algorithm "Array of soustracteurs"
  - `pipe_g` != 0, pipelined version.
  - `pipe_g` = 0, combinatorial version

- `type_g` = 2 : algorithm "Successive approximations".
  - `pipe_g` = 0, combinatorial version.

2.3.2.4 Sqrt

4 generic constants : `m_g`, `pipe_g`, `latency_g` and `type_g`

- `m_g` = 32 for single precision and `m_g` = 64 for double precision
- `type_g` = 0 : SRT2 algorithm
  - `pipe_g` != 0 and `latency_g` = 0, pipelined version.
  - `pipe_g` = 0, combinatorial version, `latency_g` determines the time (number of clock period) before the output is stable.

- `type_g` = 1 : non-restoring algorithm
  - `pipe_g` != 0 and `latency_g` = 0, pipelined version.
  - `pipe_g` = 0, combinatorial version, `latency_g` determines the time (number of clock period) before the output is stable.

2.4 Synthesis results

A synthesis process was conducted on all blocks.
Table 2.2 shows all the synthesis results made with the synthesizer XST (Xilinx) and the FPGA VirtexII xc2v1000-6bg575.
Synthesis with Virtex xc2v8000-5ff1152 :

- 1x Add2(pipelined) : 393 Slices, 1%
- 5x Add2(pipelined) : 1971 Slices, 4%
- 1x Mult2(pipelined) : 979 Slices, 2%
- 5x Mult2(pipelined) : 4900 Slices, 10%

Table 2.3 shows all the synthesis results achieved with the synthesizer Precision and the FPGA Virtex5 5VLX110FF676. (17280 Slices)

	Single precision				Double precision			
TYPE	**Slices**	**%**	F_{max}	**Pipes**	**Slices**	**%**	F_{max}	**Pipes**
Add2								
Combinatorial	664	12%			1801	35%		
Pipelined	408	7%	170 MHz	7	2714	53%	21 MHz	7
Mult2								
Wired multiplier								
Combinatorial	105	2%			345	6%		
Pipelined	155	3%	112 MHz	5	442	8%	76 MHz	5
Carry save adder								
Combinatorial	885	17%						
Pipelined	956	18%	203 MHz	7				
Div2								
SRT4								
Combinatorial	598	11%			2490	48%		
Pipelined	1049	20%	139 MHz	16	4166	81%	100 MHz	30
Array of substractor								
Combinatorial	2056	40%	6.8 MHz					
Pipelined	1904	37%	139 MHz	30				
Successive approximations								
Pipelined	3195	62%	6.3 MHz					
Sqrt								
SRT2								
Combinatorial	244	4%			901	17%		
Pipelined	400	7%	132 MHz	16	1438	28%	96 MHz	32
Non-restoring								
Combinatorial	426	8%			1698	33%		
Pipelined	701	13%	163 MHz	28	2778	54%	122 MHz	57

Table 2.2: Synthesis results with the synthesizer XST (Xilinx) and the FPGA VirtexII xc2v1000-6bg575

	Single precision				Double precision			
TYPE	**Slices**	**%**	F_{max}	**Pipes**	**Slices**	**%**	F_{max}	**Pipes**
Add2								
Combinatorial	106	0.61%	80 MHz	1	278	1.61%	60 MHz	1
Pipelined	122	0.71%	267 MHz	7	276	1.6%	170 MHz	7
Mult2								
Wired multiplier								
Combinatorial	27	0.16%	97 MHz	1	98	0.57%	60 MHz	1
Pipelined	47	0.30%	206 MHz	5	102	0.59%	76 MHz	5
Carry save adder								
Combinatorial	416	2.41%	87 MHz	1				
Pipelined	318	1.84%	338 MHz	7				
Div2								
SRT4								
Combinatorial	211	1.22%	23 MHz	1	863	5.00%	9 MHz	1
Pipelined	321	1.86%	242 MHz	16	1295	7.49%	201 MHz	30
Array of subtractors								
Combinatorial	263	1.52%	7 MHz	1				
Pipelined	547	3.17%	156 MHz	30				
Successive approximations								
Combinatorial	500	2.89%	7 MHz	1				
Sqrt								
SRT2								
Combinatorial	112	0.65%	24 MHz	1	434	2.51%	9 MHz	1
Pipelined	147	0.85%	230 MHz	16	530	3.07%	163 MHz	32
Non-restoring								
Combinatorial	183	1.06%	18 MHz	1	781	4.52%	6.7 MHz	1
Pipelined	261	1.51%	265 MHz	28	990	5.73%	216 MHz	57
Compare								
Combinatorial	17	0.10%	333 MHz					
Mult2 with a constant value (wired mulplier)								
Pipelined	40	0.23%	206 MHz					

Table 2.3: Synthesis results with the synthesizer Precision and the FPGA Virtex5 5VLX110FF676.

2.5 Square root algorithm

2.5.1 Principle

Number representation in floating point is calculated as following:

$$F = M * 2^E$$

The square root of a number in floating point :
If E is even : $\sqrt{F} = \sqrt{M} * 2^{E/2}$
If E is odd : $\sqrt{F} = \sqrt{M/2} * 2^{E/2+1}$
The square root of the mantissa was implemented with two different algorithms. The first was taken from an existing library (SRT2 algorithm) and the other was realized by ourselves (non-restoring algorithm).

2.5.2 Non-restoring square root

The square root of the mantissa is calculated through the following recurrence equation:

$$\begin{cases} X_0 = 0, r_0 = M \\ q_0 = 1, \\ r_{i+1} = 2 * r_i - 2 * X_i * q_{i+1} - 2^{-(i+1)} \end{cases} \qquad q_{i+1} = \begin{cases} +1 & \text{if } r_i \geq 0 \\ -1 & \text{else} \end{cases}$$

Explication :

- r_i is the ith partial remainder.
- X_i is the ith bit of square root result.

The computation is divided into four stages :

1. the value of the partial remainder is shifted a bit left to get $2 * r_i$.
2. The value of q_{i+1} is inferred by the sign of the partial remainder. If it's positive, $q_{i+1} = 1$ and if it's negative, $q_{i+1} = -1$.
3. the value of the square root is shifted a bit left to get $2 * X_i$.
4. The computation of the partial remainder r_{i+1} can be done using the results of the first three stages.

Computation of X at the ith step, example :

$$X_1 = 0.1_2$$
$$if\ q_2 = -1, X_2 = X_1 - 0.01_2 = 0.01_2$$
$$if\ q_2 = 1, X_2 = X_1 + 0.01_2 = 0.11_2$$

2.5.2.1 Example

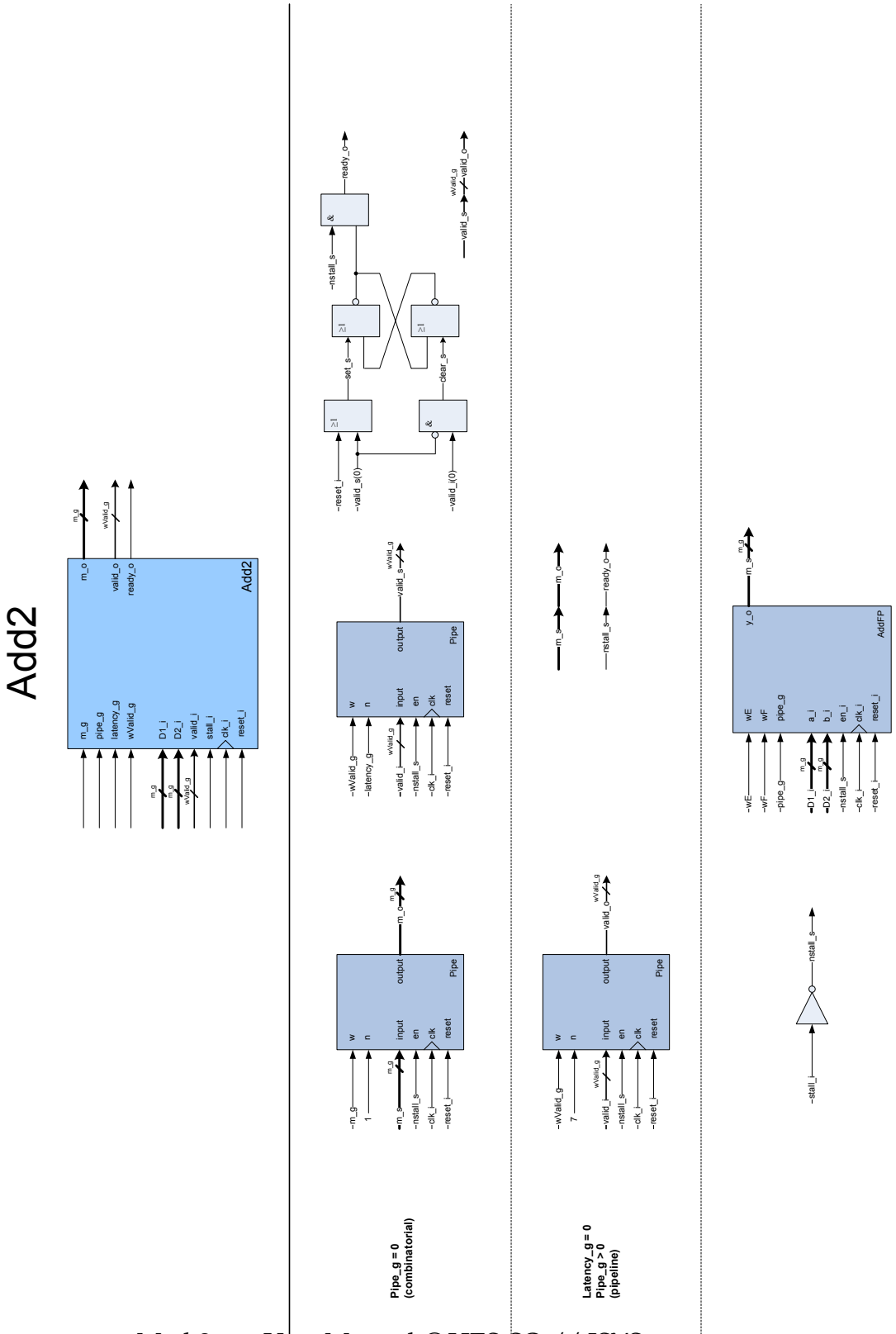
Square root of 0.5 on 6 bits : theoretical value$= \sqrt{0.5} = 0.707107$

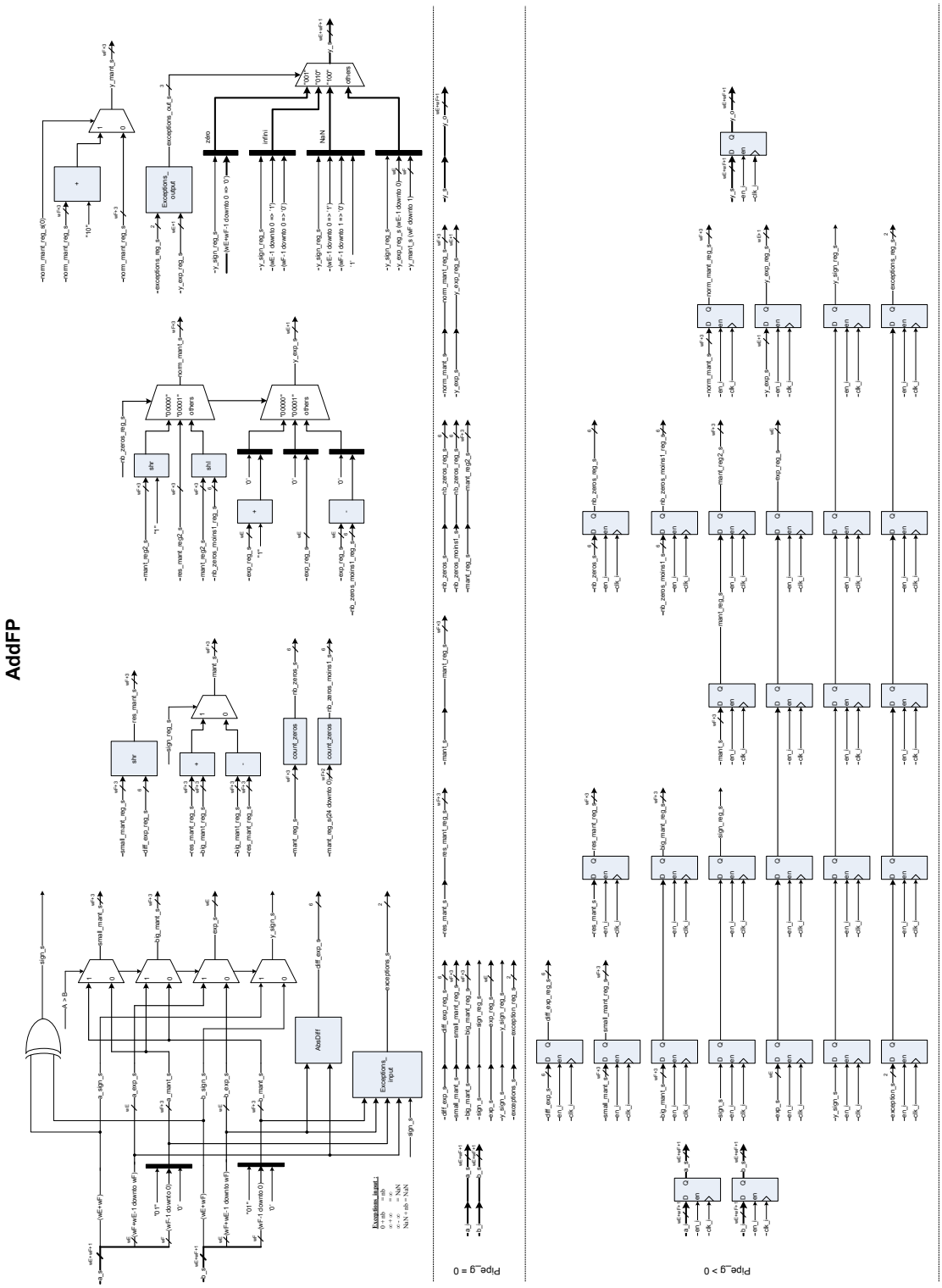
$R_0 = 0.5$		0	0	.	1	0	0	0	0	0	$X_0 = 0$
$2 * r_0$		0	1	.	0	0	0	0	0	0	$q_1 = 1$ $X_1 = 0.1$
$-(2 * X_0 + 2^{-1})$	-	0	0	.	1	0	0	0	0	0	
r_1		0	0	.	1	0	0	0	0	0	
$2 * r_1$		0	1	.	0	0	0	0	0	0	$q_2 = 1$ $X_2 = 0.11$
$-(2 * X_1 + 2^{-2})$	-	0	1	.	0	1	0	0	0	0	
r_2		1	1	.	1	1	0	0	0	0	
$2 * r_2$		1	1	.	1	0	0	0	0	0	$q_3 = -1$ $X_3 = 0.101$
$+(2 * X_2 - 2^{-3})$	+	0	1	.	0	1	1	0	0	0	
r_3		0	0	.	1	1	1	0	0	0	
$2 * r_3$		0	1	.	1	1	0	0	0	0	$q_4 = 1$ $X_4 = 0.1011$
$-(2 * X_3 + 2^{-4})$	-	0	1	.	0	1	0	1	0	0	
r_4		0	0	.	0	1	1	1	0	0	
$2 * r_4$		0	0	.	1	1	1	0	0	0	$q_5 = 1$ $X_5 = 0.10111$
$-(2 * X_4 + 2^{-5})$	-	0	1	.	0	1	1	0	1	0	
r_4		1	1	.	0	1	1	1	1	0	
$2 * r_4$		1	0	.	1	1	1	1	0	0	$q_6 = -1$ $\underline{X6 = 0.101101_2}$
											$\underline{X6 = 0.703125}$

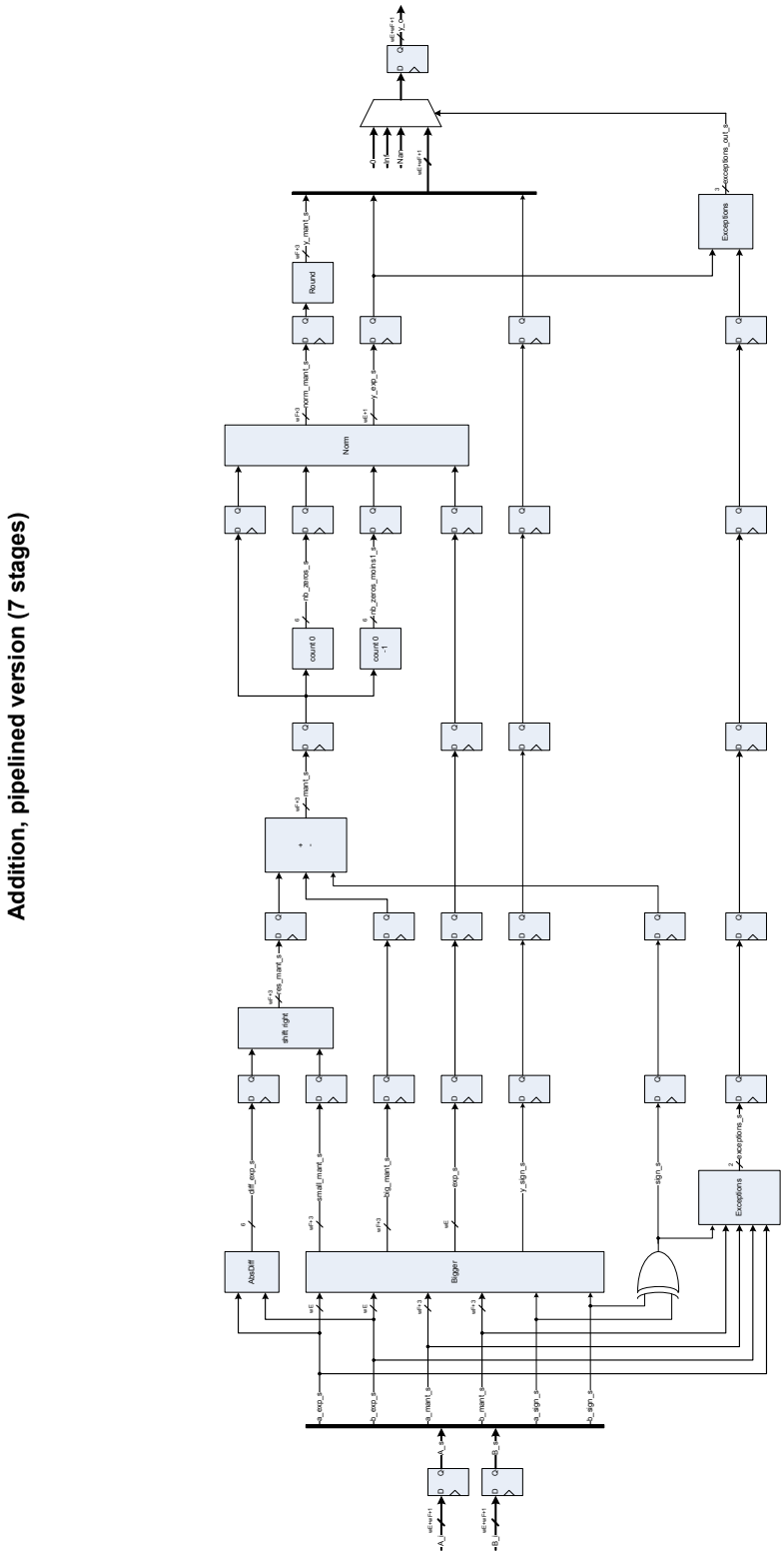

M2M

2.6 Blocks schematics

2.6.1 Addition

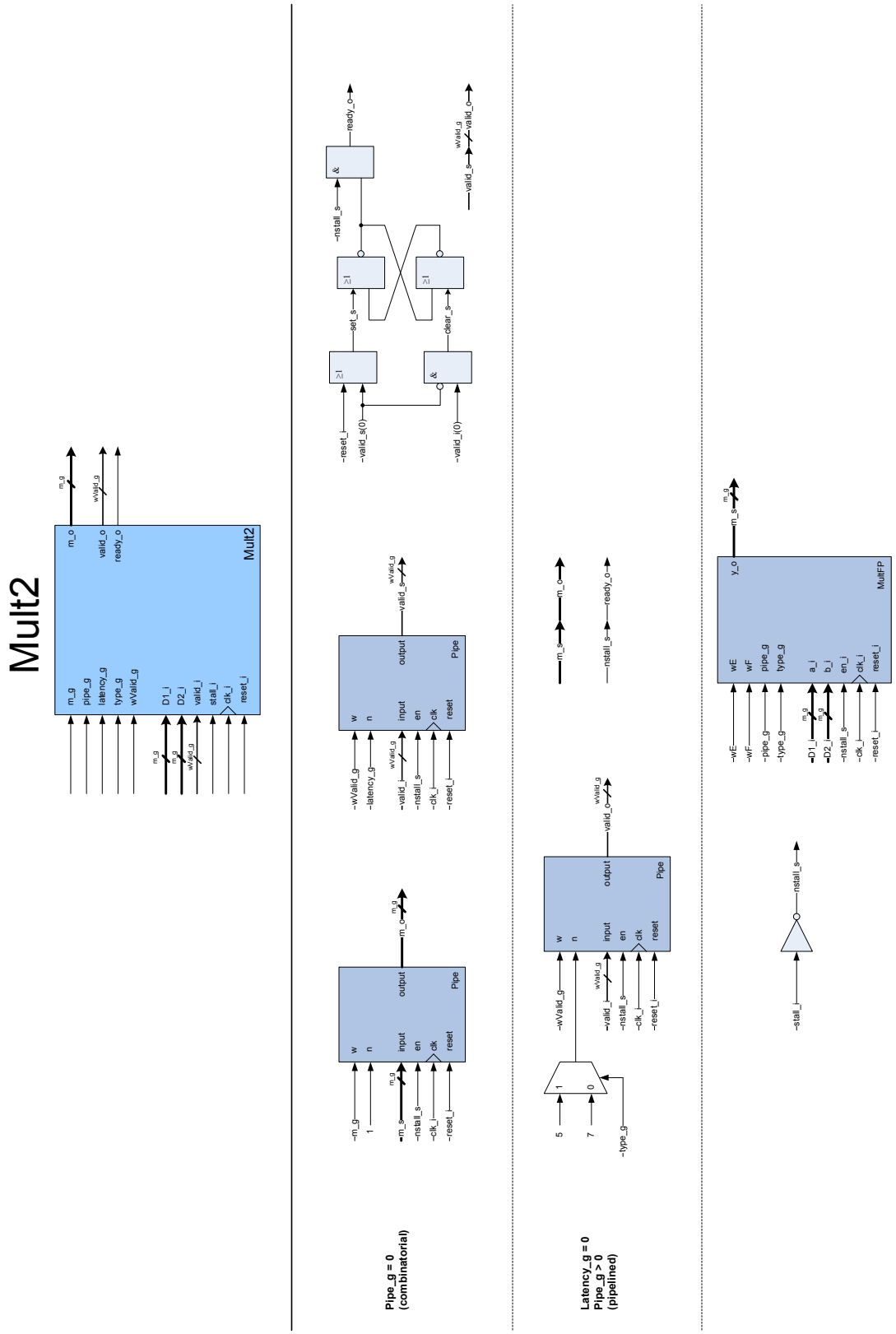


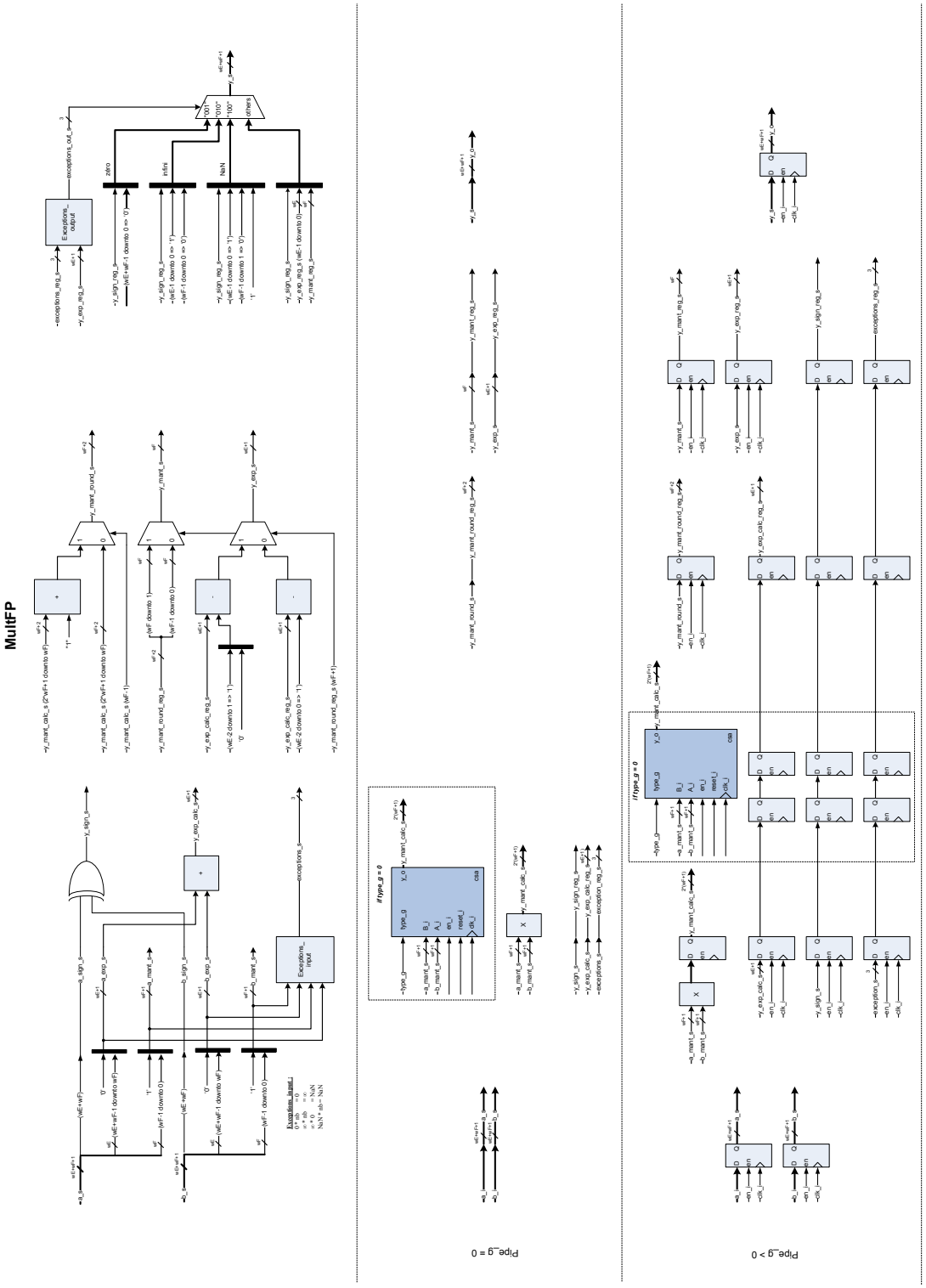

AddFP
Pipe_g = 0
Pipe_g > 0

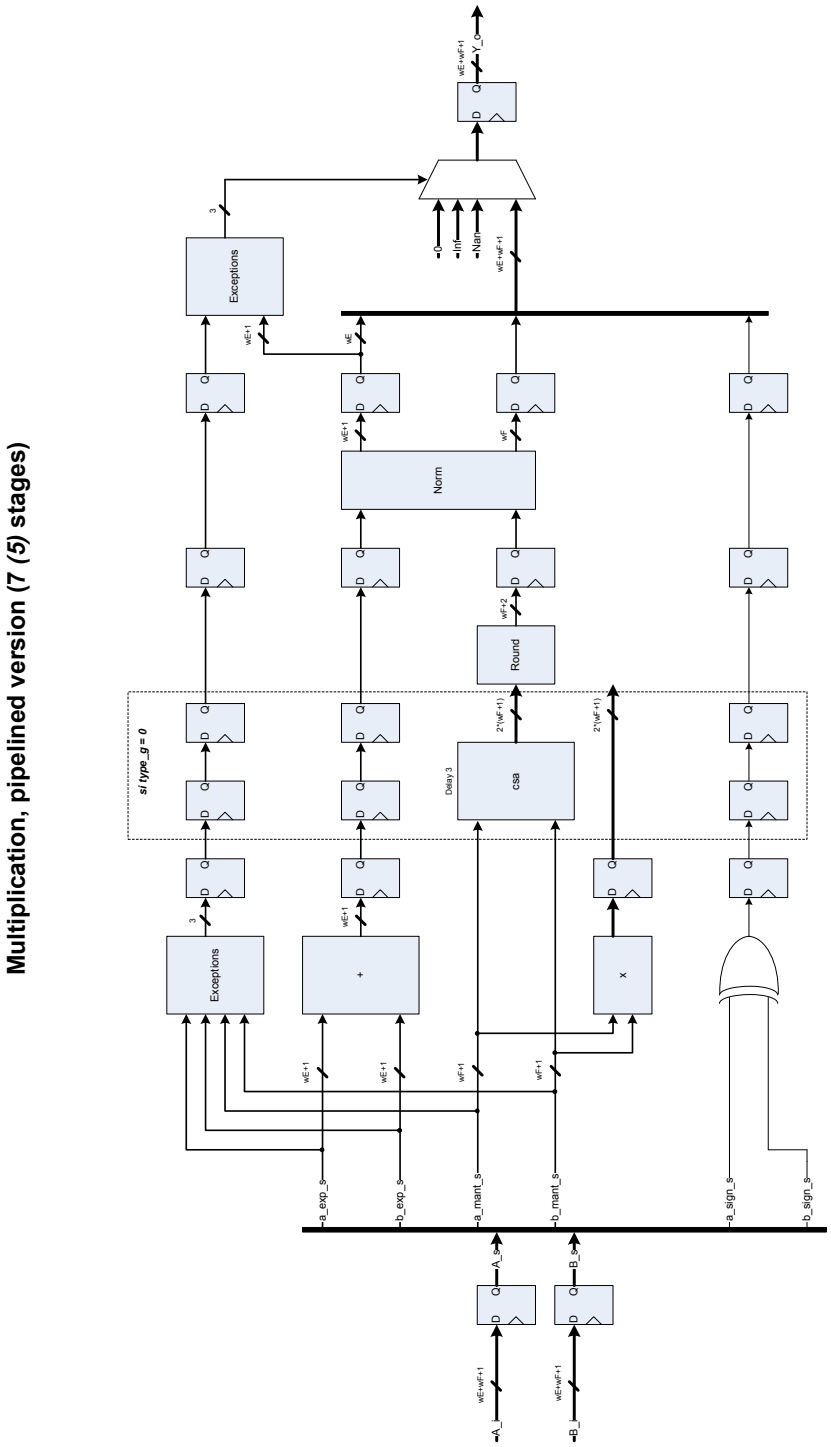

Addition, pipelined version (7 stages)
Round
Norm
Exceptions
AbsDiff
Bigger
shift right
count 0
count 0
Exceptions


M2M

2.6.2 Multiplication

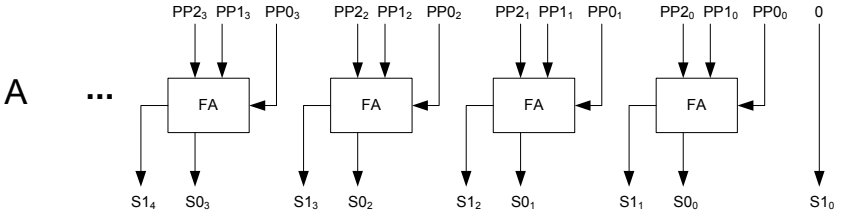


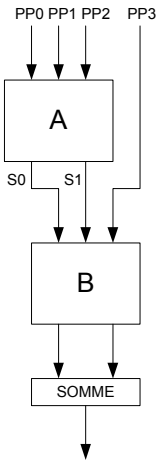

MultFP
Exceptions_output
zero
infini
NaN
others
Exceptions_input
if type_g = 0
type_g
clk_i
rst_i
en_i
A_i
B_i
reset_i
csa
Pipe_g = 0
Pipe_g > 0

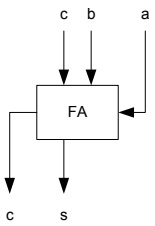

Multiplication, pipelined version (7 (5) stages)
si type_g = 0
Exceptions
+
Norm
Round
Delay 3
csa
x
wE+1
wF+1
wF+2
2*(wF+1)
wE+wF+1
a_exp_s
b_exp_s
a_mant_s
b_mant_s
a_sign_s
b_sign_s
A_s
B_s
A_i
B_i
Y_o
Zero
Inf
Nan

Multiplication (carry save adder)

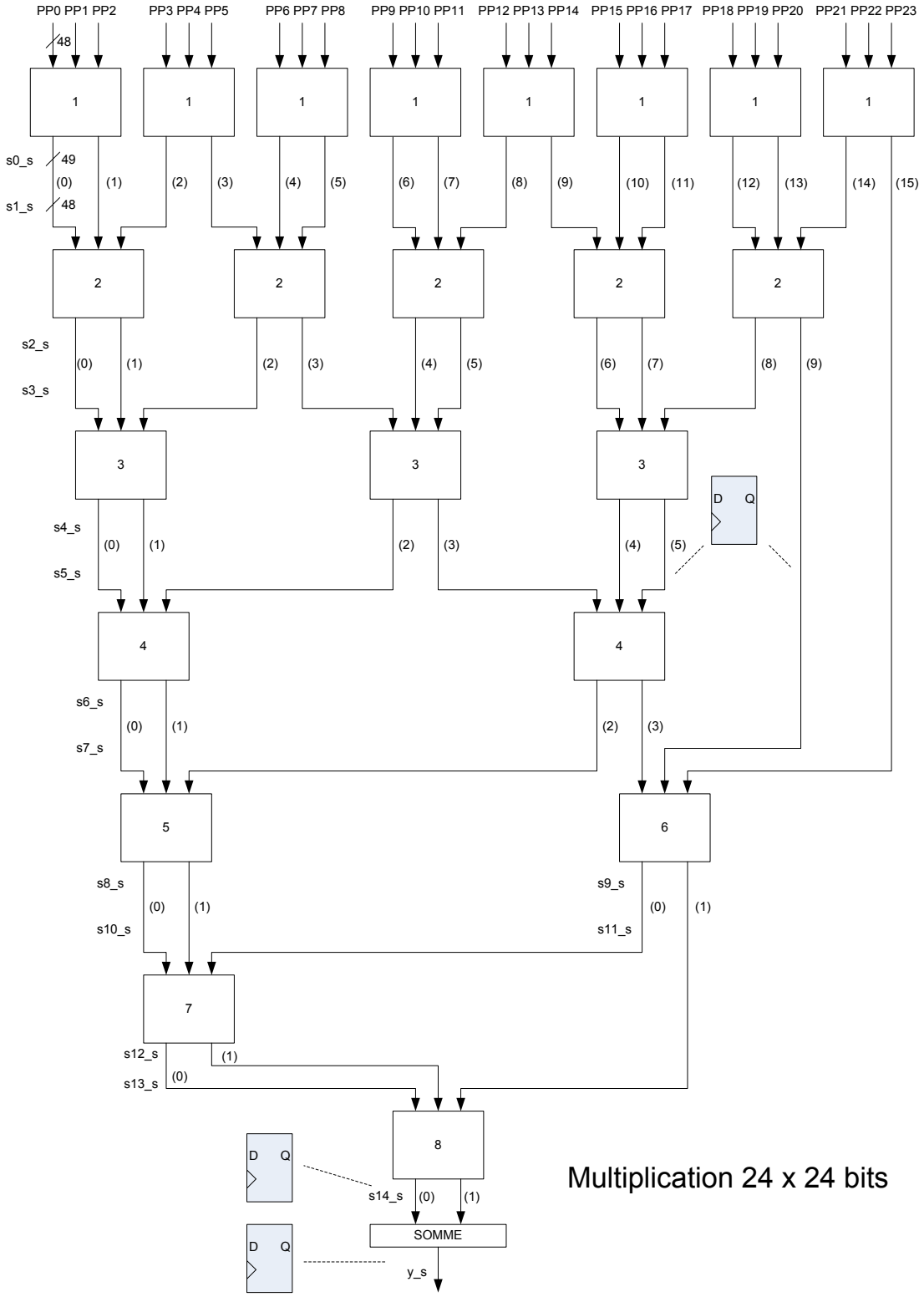
```
        1001
       x1101
A      0001001   PP0
       0000000   PP1     => PP0 + PP1 + PP2 = S0 + S1
       0100100   PP2
       1001000   PP3
       1110101
```



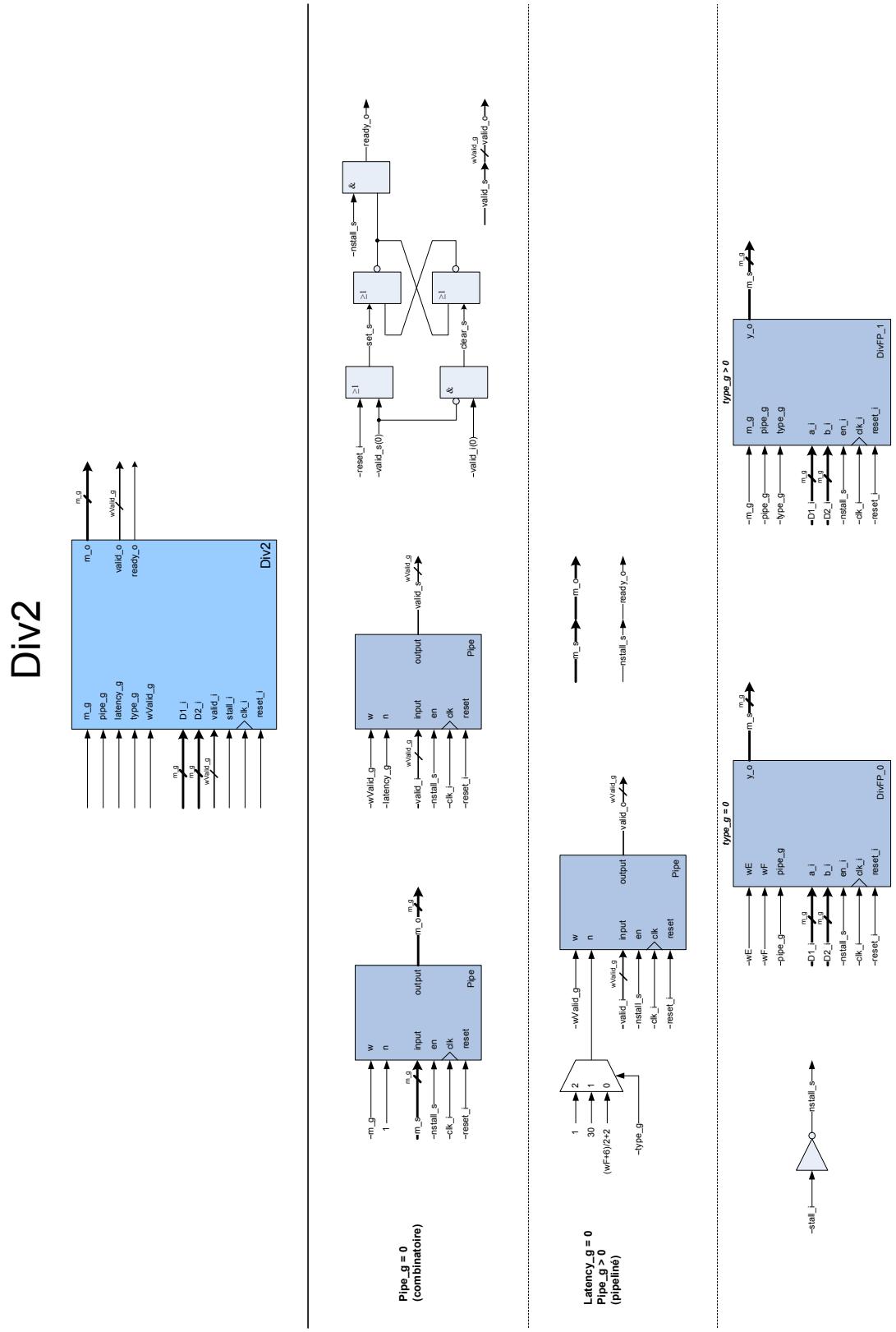


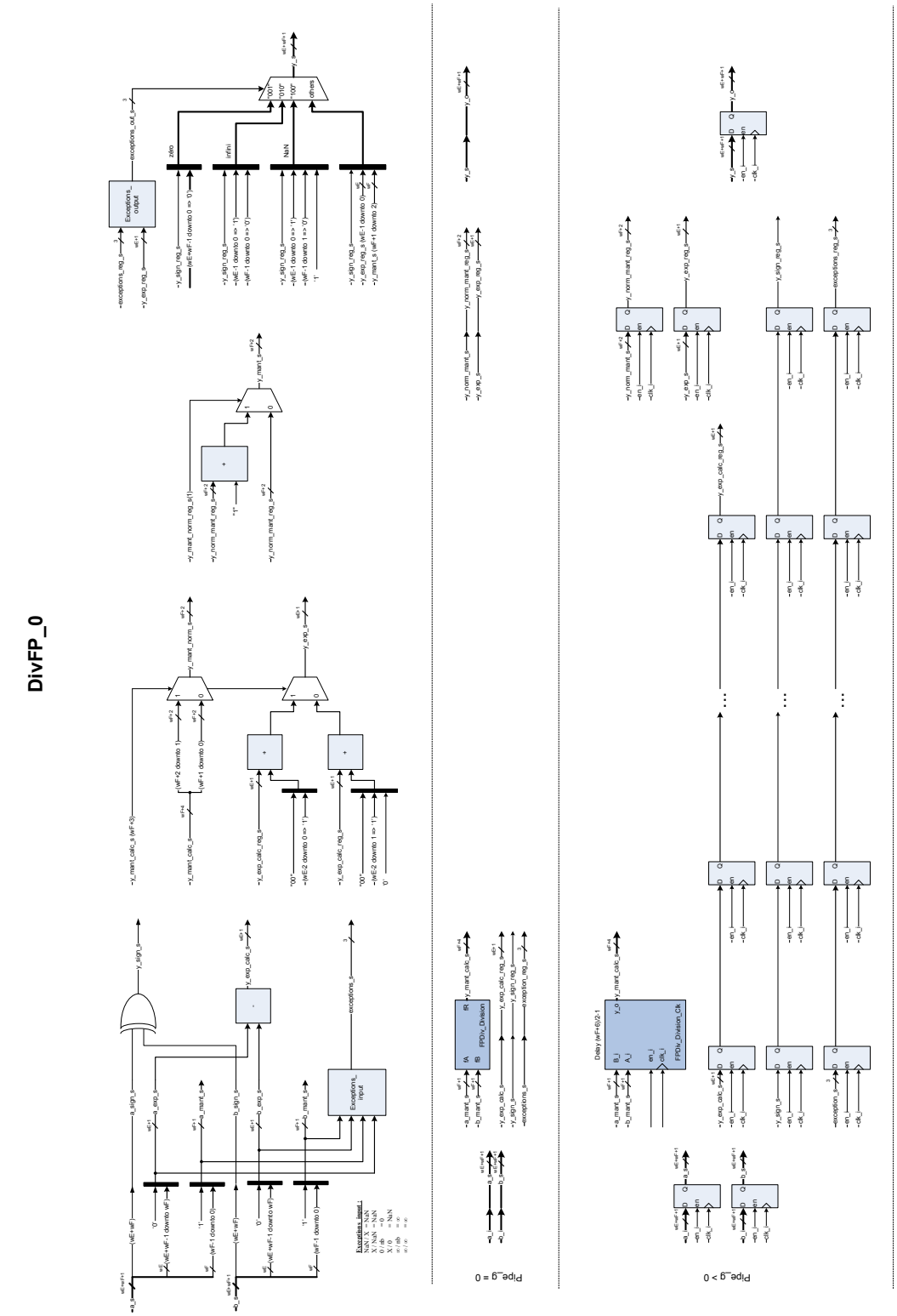


s = a xor b xor c
c = (a and b) or (a and c) or (b and c)

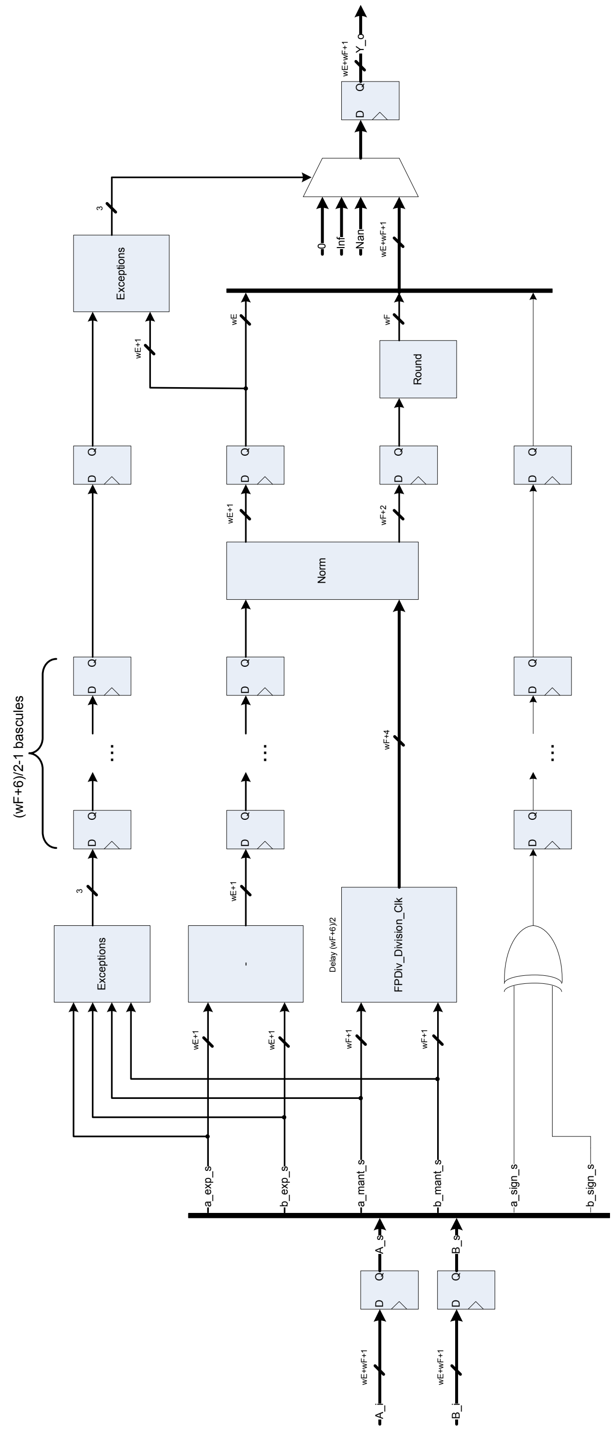

PP0 PP1 PP2
PP3 PP4 PP5
PP6 PP7 PP8
PP9 PP10 PP11
PP12 PP13 PP14
PP15 PP16 PP17
PP18 PP19 PP20
PP21 PP22 PP23
48
1
s0_s
49
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)
(11)
(12)
(13)
(14)
(15)
s1_s
48
2
s2_s
s3_s
3
D Q
s4_s
s5_s
4
s6_s
s7_s
5
6
s8_s
s9_s
s10_s
s11_s
7
s12_s
s13_s
8
s14_s
SOMME
y_s
Multiplication 24 x 24 bits

2.6.3 Division

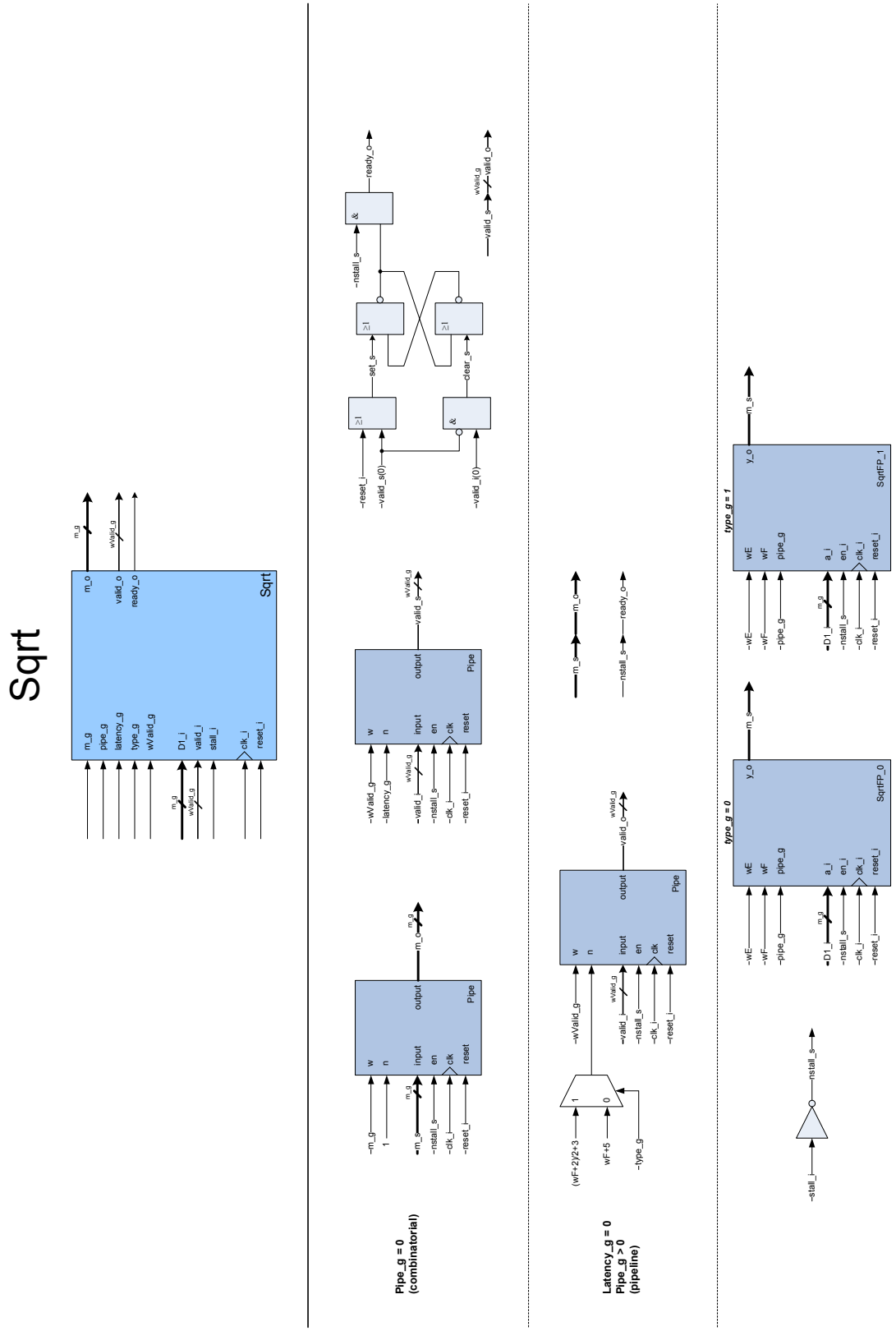


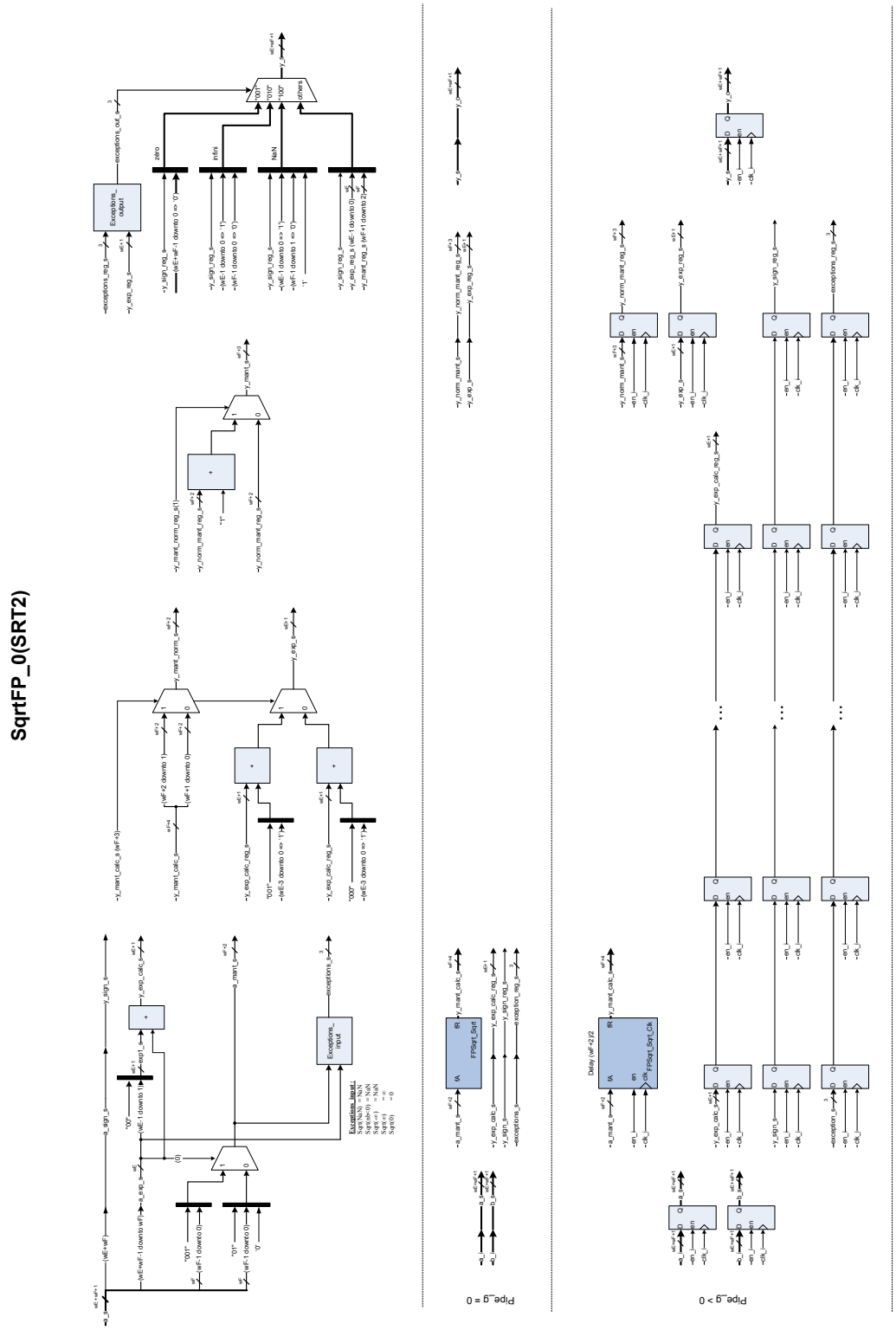

DivFP_0
Pipe_g = 0
Pipe_g > 0
FPDiv_Division
Exceptions_Logic
Exceptions_Input
Delay (wF+6)/2-1

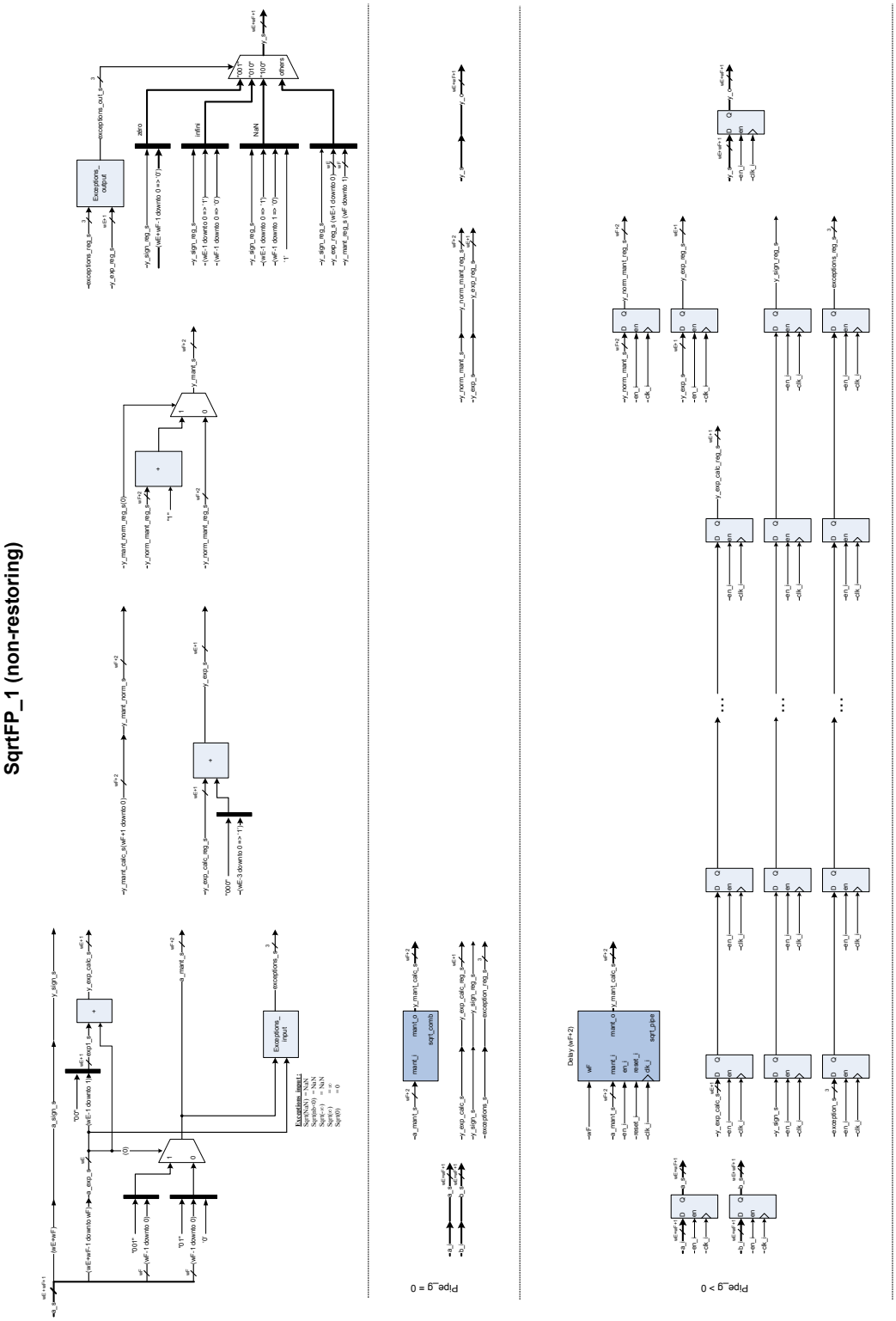
Division, pipelined version ((*(wF*+6)/2+2) stages)



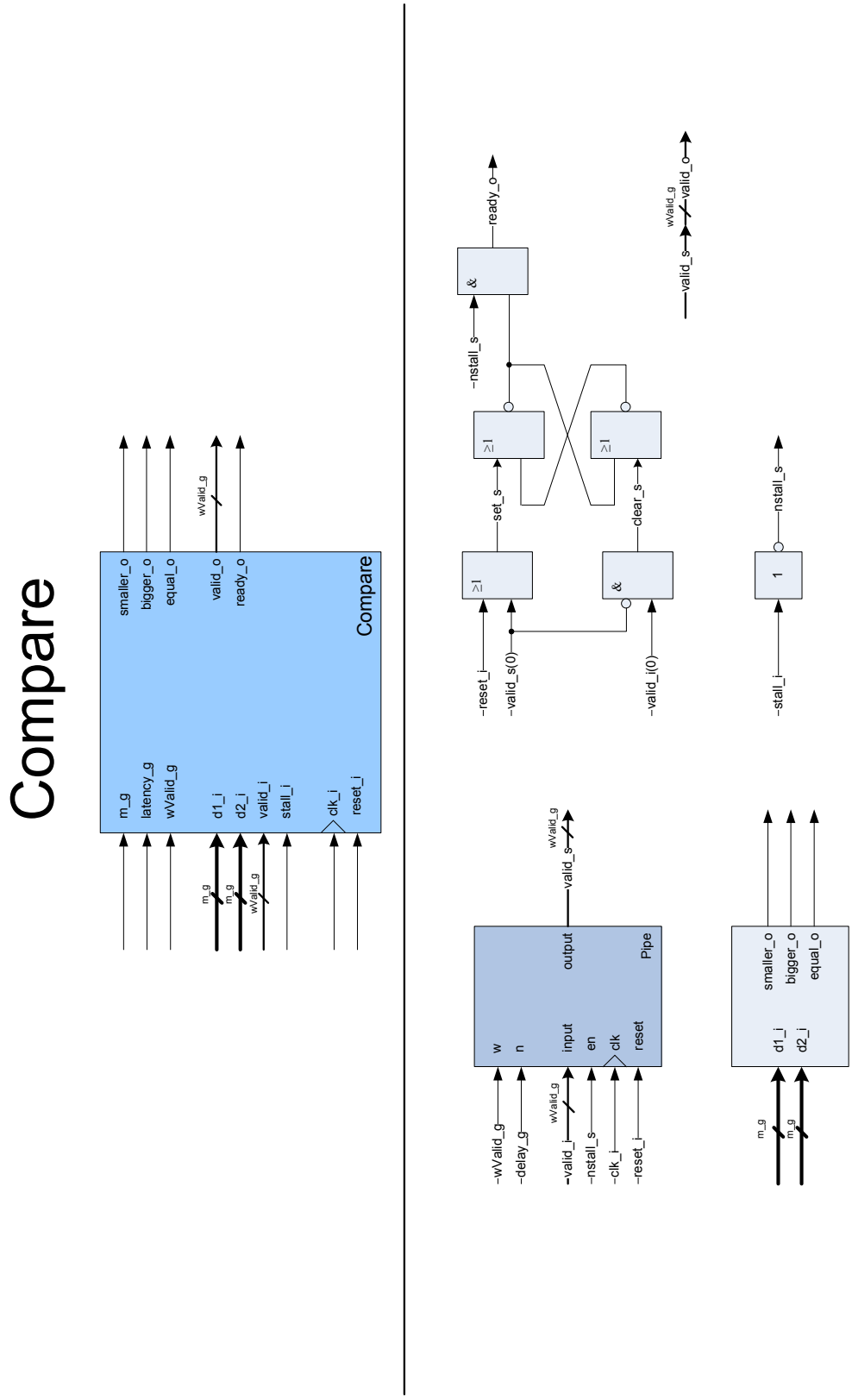
2.6.4 Square root

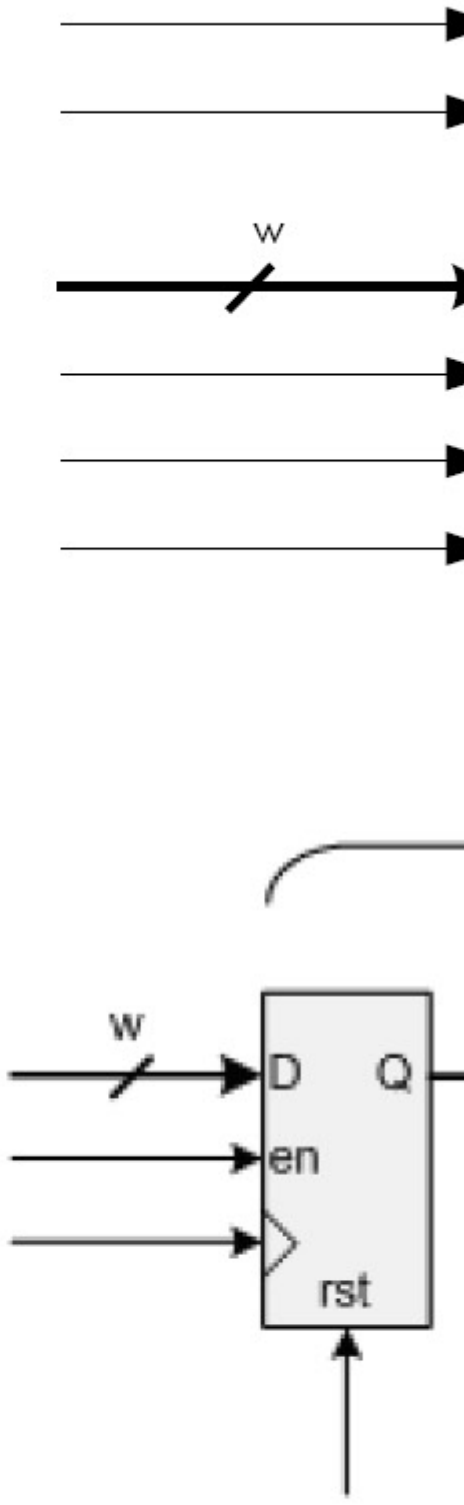



SqrtFP_0(SRT2)
Pipe_g = 0
Pipe_g > 0


SqrtFP_1 (non-restoring)
Pipe_g = 0
Pipe_g > 0

2.6.5 Comparison




w
w
D
Q
en
rst

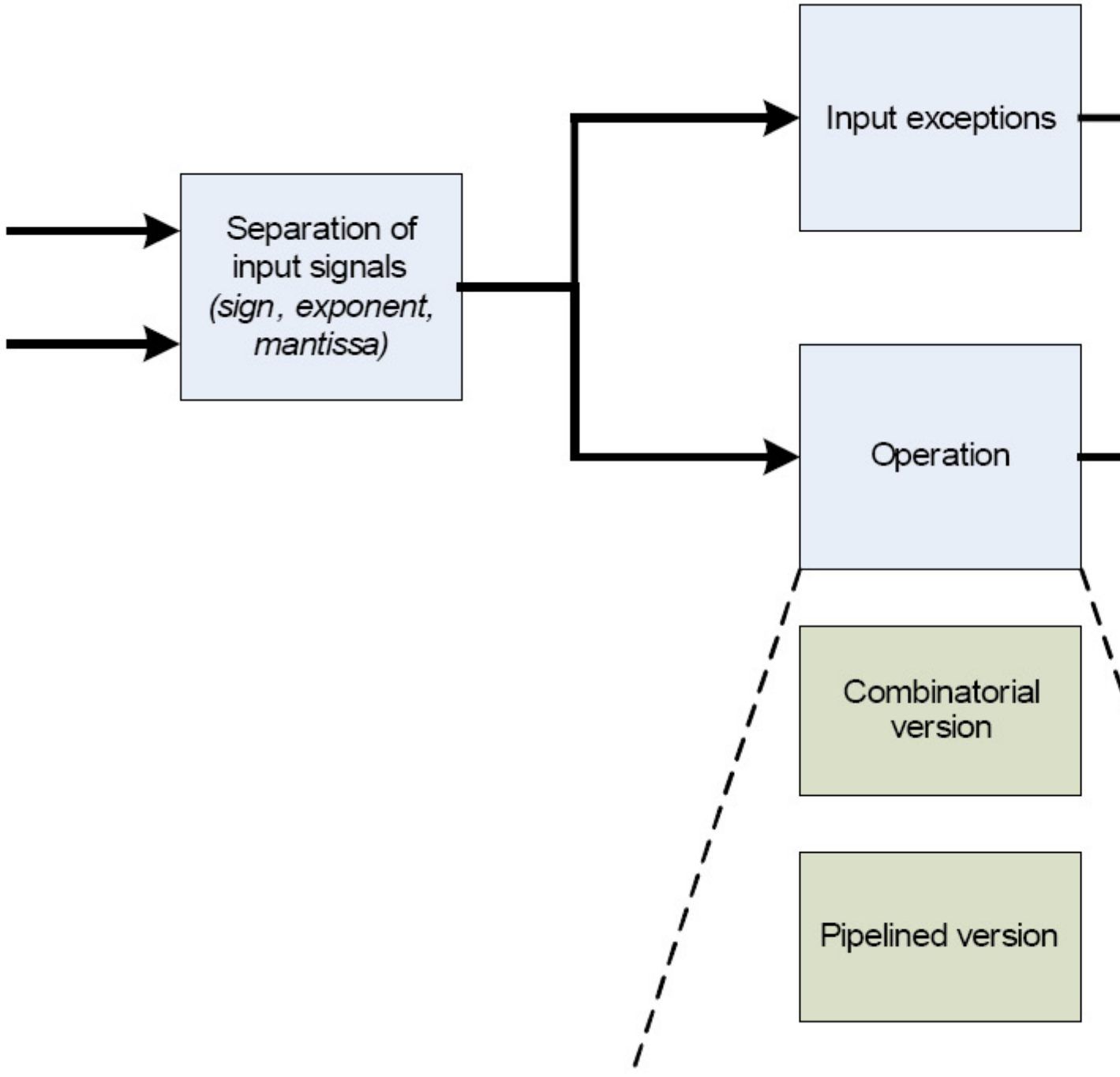


Figure 2.7: Structure of an operation

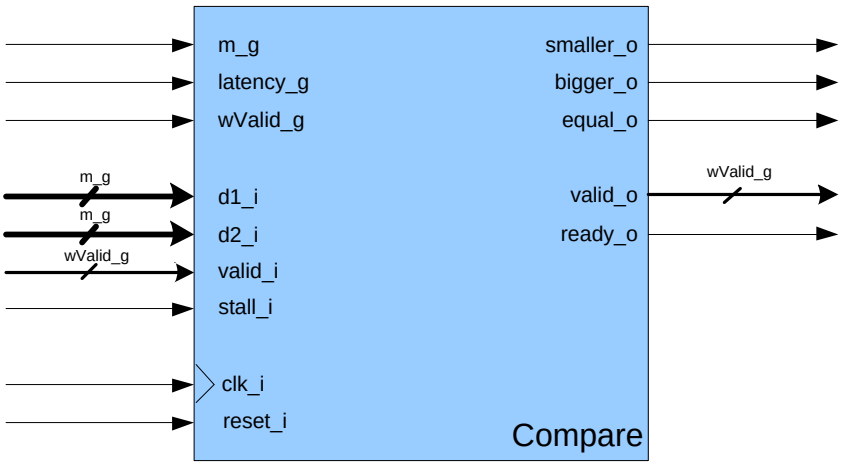


Figure 2.8: Compare block interface

Chapter 3

Generation

3.1 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.

3.1.1 Block representation

The figure 3.1 shows a generic Math2mat block comprising n number of entries and m number of outputs.

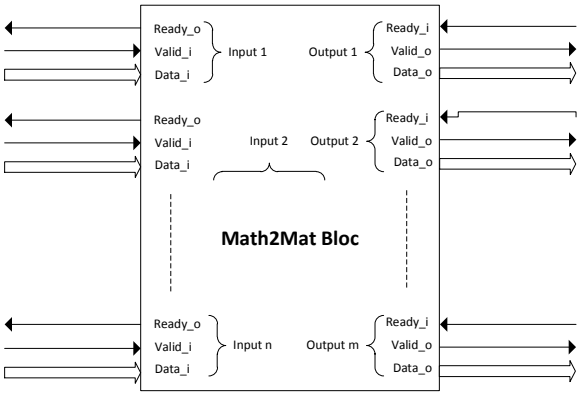


Figure 3.1: Generic Math2Mat block

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.

3.1.2 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. Figure 3.2 illustrates an example of the generation of a Math2Mat block and using two inputs `a` and `b` and one output `c`.

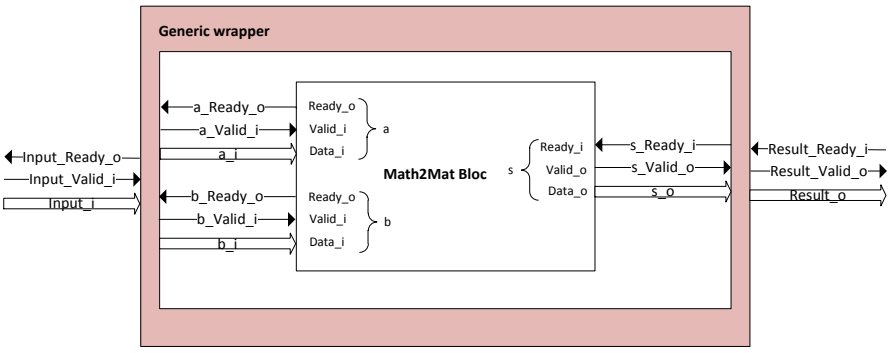
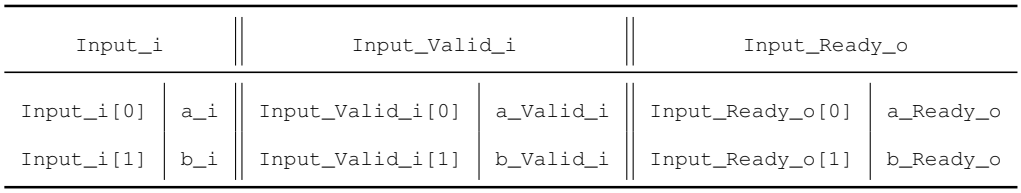


Figure 3.2: Generic wrapper

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.
- coucou

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. Tables 3.3 and 3.4 shows the positioning signals in these six tables.



Input_i		Input_Valid_i		Input_Ready_o	
Input_i[0]	a_i	Input_Valid_i[0]	a_Valid_i	Input_Ready_o[0]	a_Ready_o
Input_i[1]	b_i	Input_Valid_i[1]	b_Valid_i	Input_Ready_o[1]	b_Ready_o

Figure 3.3: Input tables representation

Result_o		Result_Valid_o		Result_Ready_i	
Result_o[0]	s_o	Result_Valid_o[0]	s_Valid_o	Result_Ready_i[0]	s_Ready_i

Figure 3.4: Output tables representation

3.1.3 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.

3.1.3.1 Timing diagrams

To better understand how a block from the outside, consider the timing diagram of figure 3.5 with two inputs and one output.
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 timing diagram of figure 3.6 helps to highlight the behavior of the same block in one less common 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.

3.1.4 Content block

3.1.4.1 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

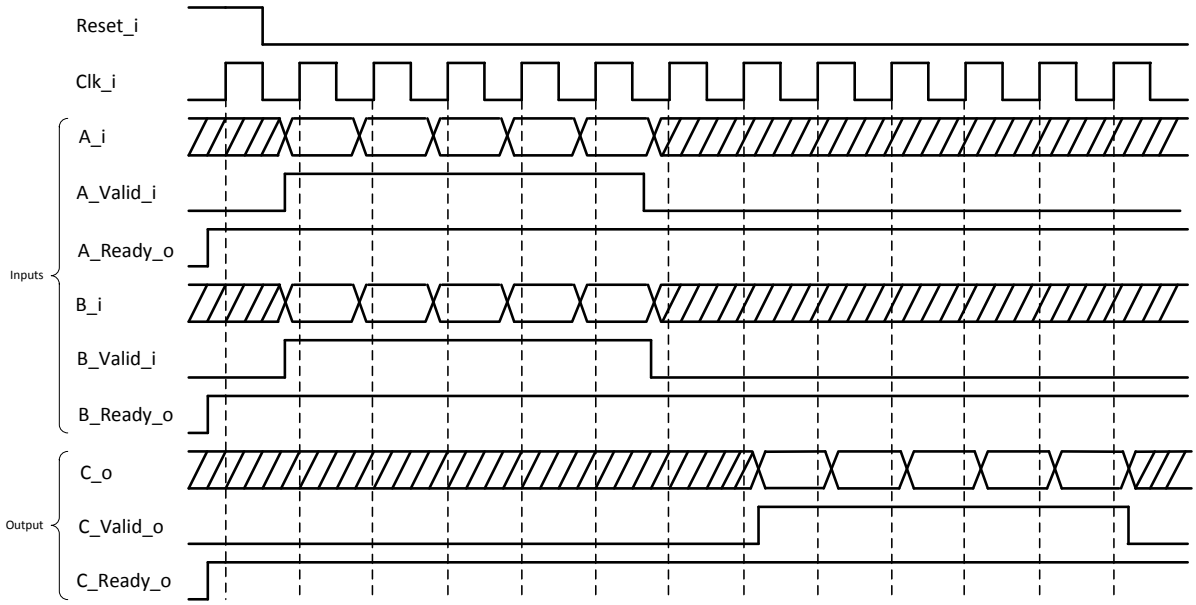


Figure 3.5: Timing diagram of a usual case

logic is also required. The diagram of figure 3.7 shows the typical contents of a block. 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.

3.1.4.2 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. Figure 3.8 shows this propagation.
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.

3.2 Wrapper

3.2.1 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.

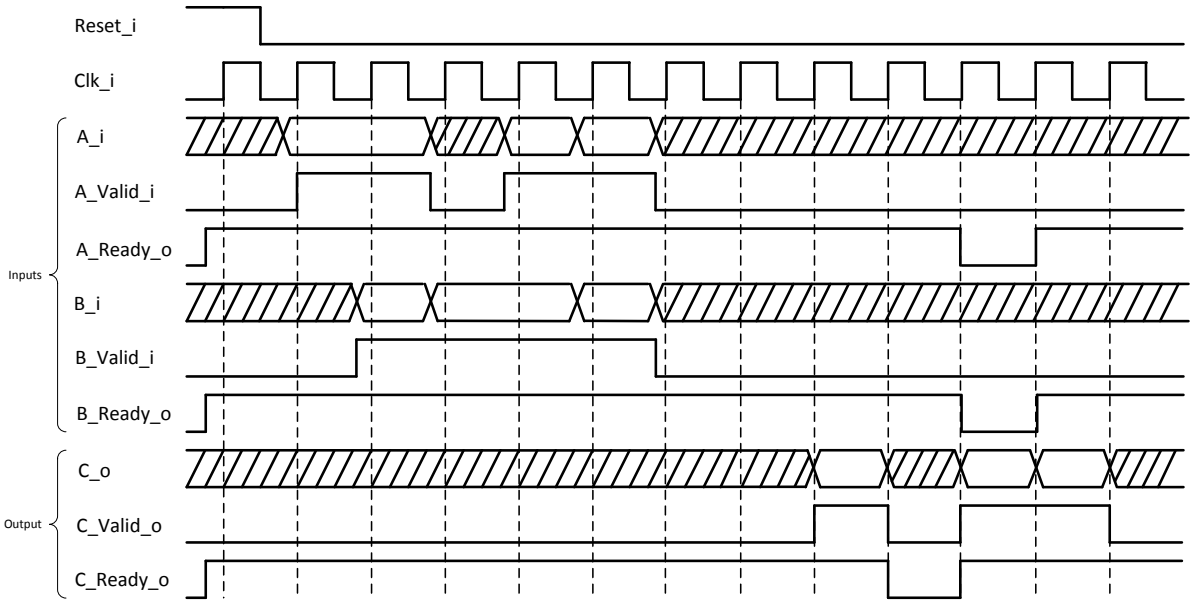


Figure 3.6: Timing diagram of a non-usual case

- 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 3.9 illustrates the interface to develop for a wrapper who includes an operator with two inputs and one output.
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.

3.2.2 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`.

Figure 3.10 represents a wrapper for an operator having two inputs and one output.

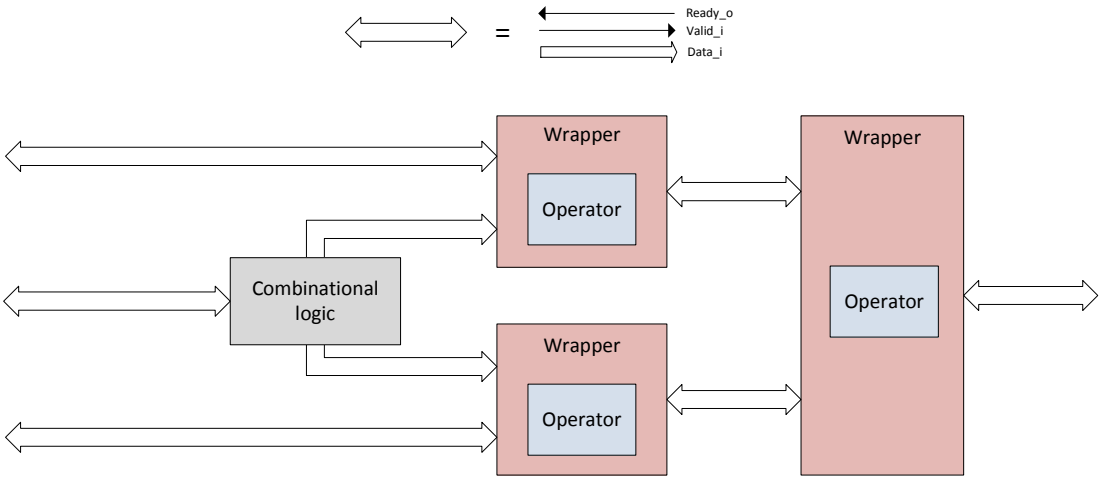


Figure 3.7: Block content

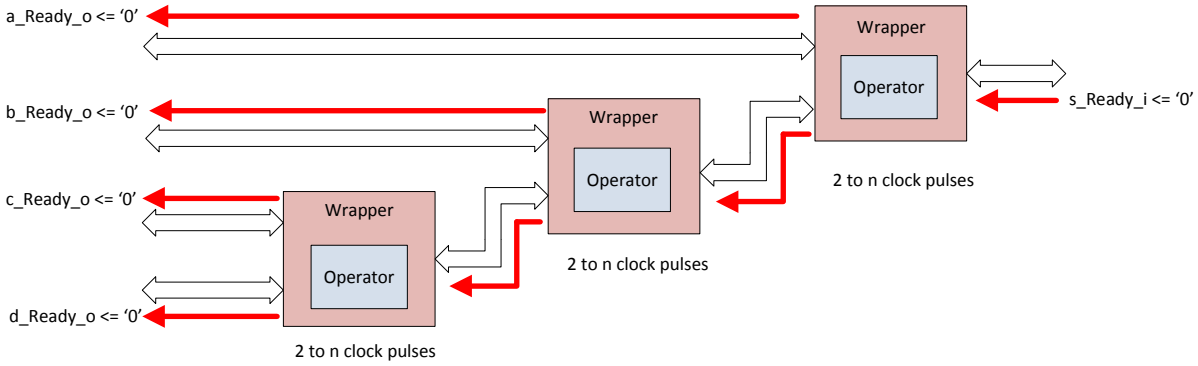


Figure 3.8: Propagation of ready Timing diagram of a usual case

3.2.2.1 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.

3.2.2.2 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.

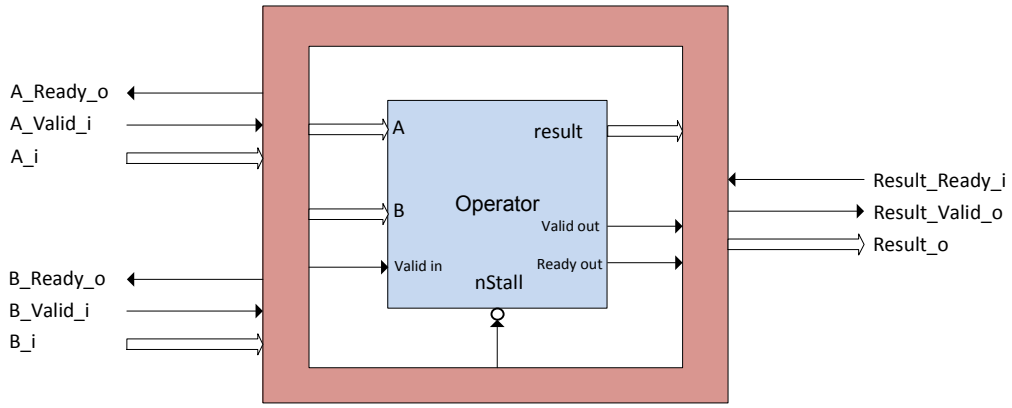


Figure 3.9: Block diagram of a wrapper

- 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.

3.2.2.3 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.

3.3 Mathematical function

3.3.1 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

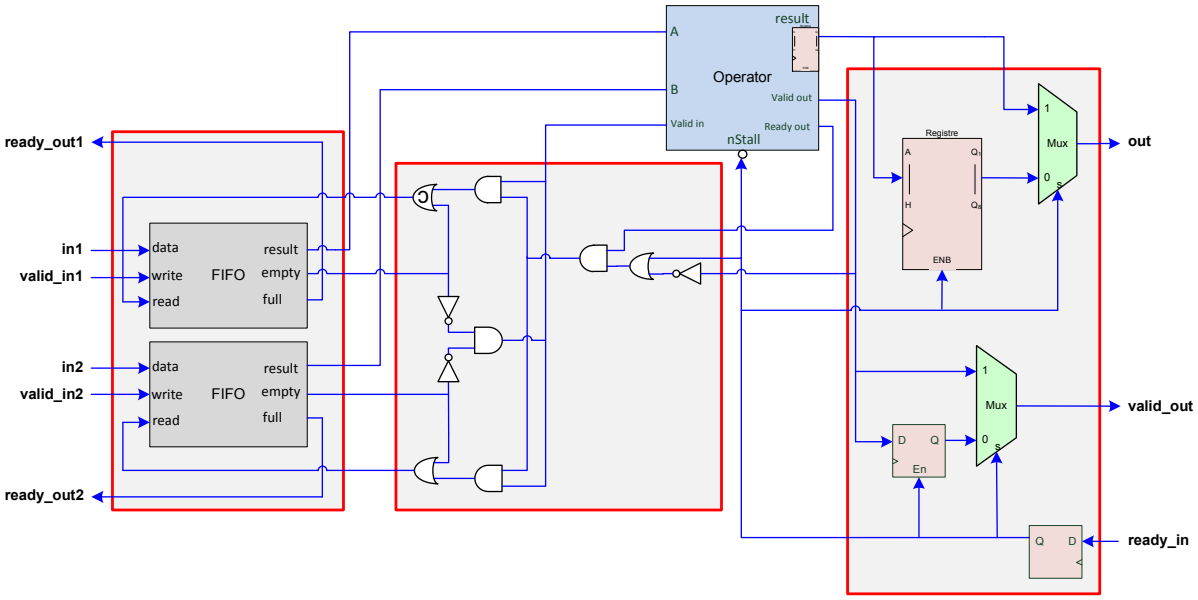


Figure 3.10: Wrapper for a basic operator

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.

3.3.2 Multi-used signal

3.3.2.1 Structure modifications

To show the necessary changes in the structure, consider the following octave code and its corresponding structure shown in figure 3.11.

Example

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

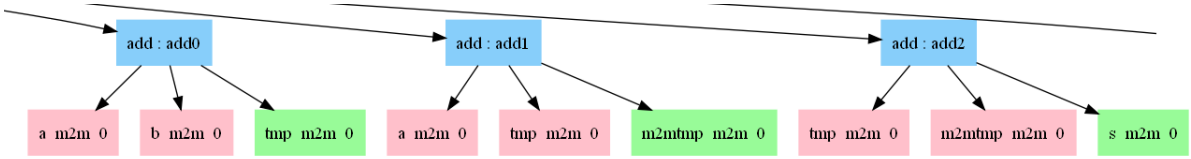


Figure 3.11: Structure state before generation

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 structure shown in figure 3.12.

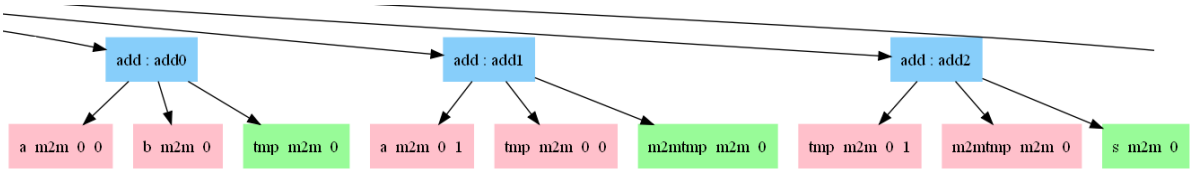


Figure 3.12: Structure state after generation

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.

3.3.2.2 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. Figure 3.13 shows an example for a signal `a_m2m_0` used twice.

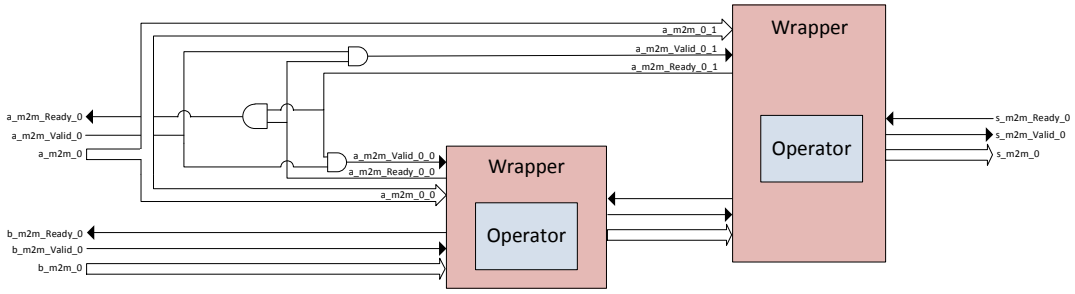


Figure 3.13: Combinational logic of multi-used signal

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 scenario shown in figure 3.14 is used to represent this case.

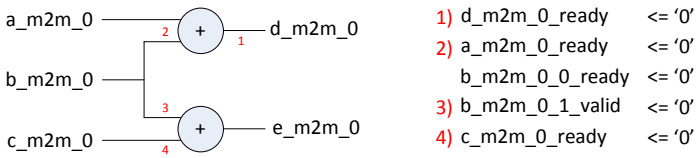


Figure 3.14: Principle of multi-used variables logic

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.

3.3.2.3 Fifos compensation

To ensure a maximum rate, some compensation fifos are needed. Figure 3.15 shows a case requiring such compensation.

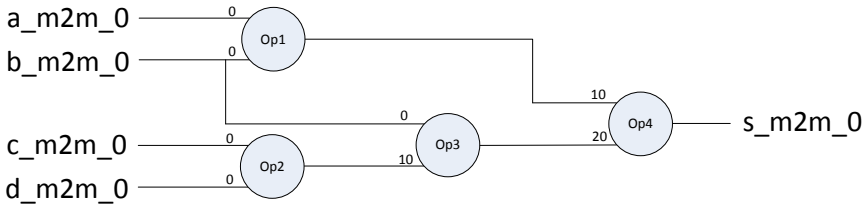


Figure 3.15: Example of case requiring fifos compensation

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`.

Figure 3.16 shows the same case with compensation fifos of depth 10.
Regarding the calculation of latency inputs within the structure, it takes into account the following values:

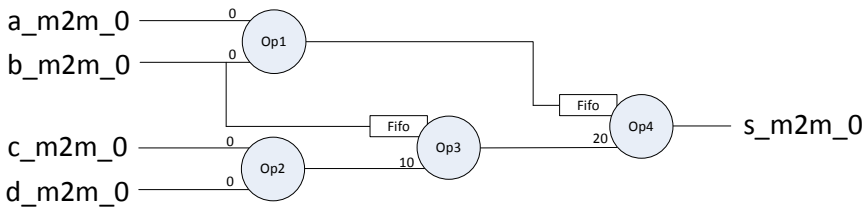


Figure 3.16: Example of case with fifos compensation

- 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.

3.3.3 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:
Consider the scenario shown in figure 3.17.

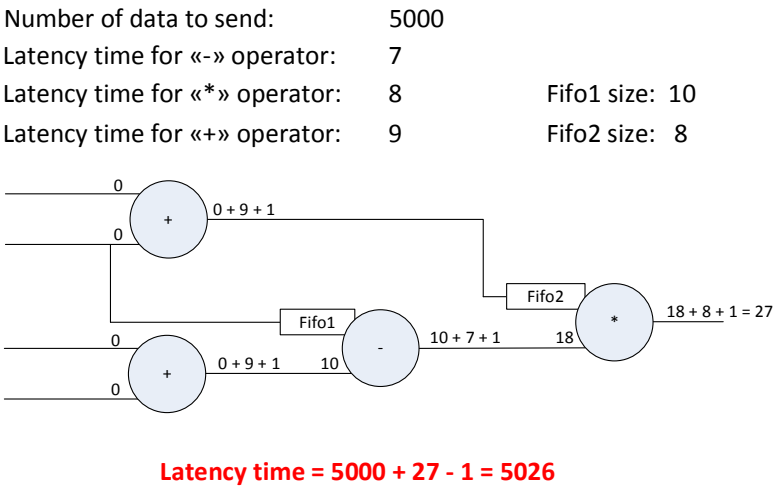


Figure 3.17: Example of latency time calculation

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 nř0 is the first. This result corresponds to the minimum latency that can be obtained.

3.3.4 Internal fifo justification

The internal fifos is fully justified when the inputs and outputs of two wrappers are linked. Figure 3.18 illustrates a case from the dynamic view presenting this phenomenon.

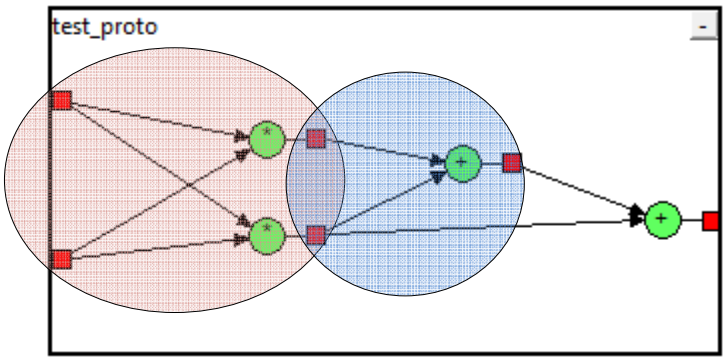


Figure 3.18: Problematic case without internal fifos

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.

3.4 Conditional statement

3.4.1 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.

3.4.2 Condition signal

3.4.2.1 Principle

Consider the following octave code example to express a simple instruction conditional **`if`**:

Example

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

Figure 3.19 shows the graphical representation of the code above. Surrounded operators show additions to a simple polynomial.

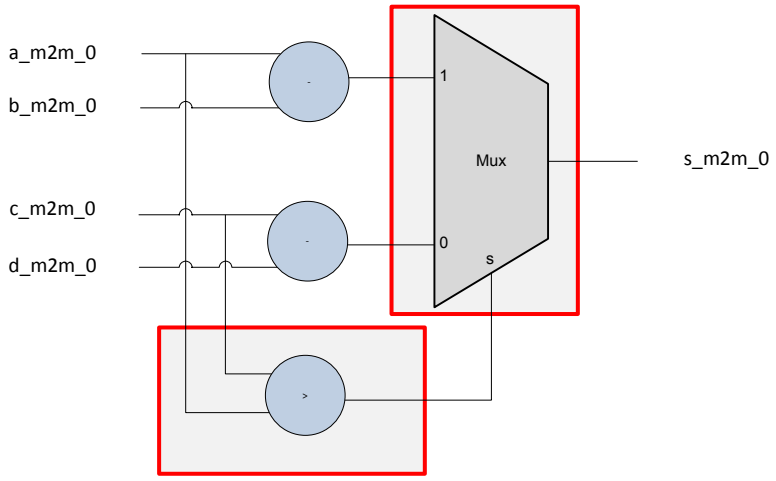


Figure 3.19: Graphical representation of a conditional instruction if

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.

3.4.2.2 Latency

In the previous example and after setting all the latencies we obtain the result shown in figure 3.20.
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.

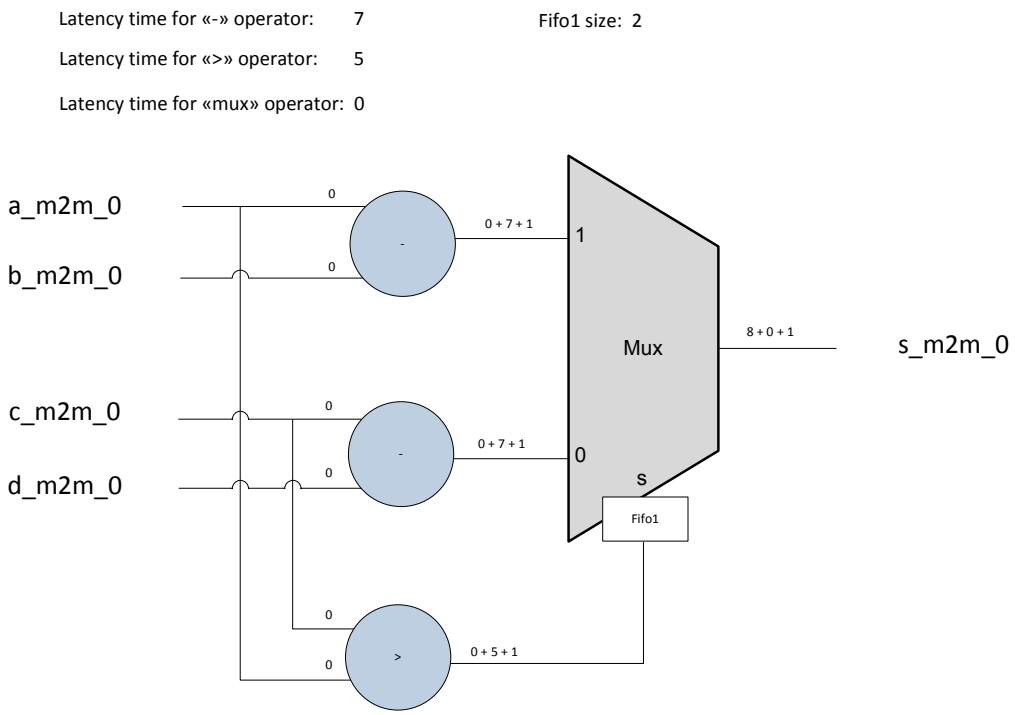


Figure 3.20: Latency time of a conditional instruction if

3.5 For Loop

3.5.1 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.

3.5.2 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.

3.5.2.1 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 of figure 3.21 illustrates this principle.

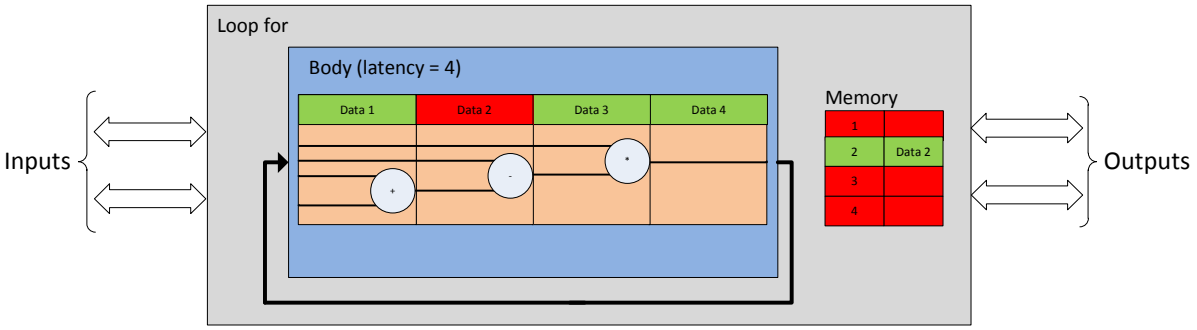


Figure 3.21: Example of desired rate

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.

3.5.2.2 First decomposition

A for loop consists of three distinct parts as presented in figure 3.22.

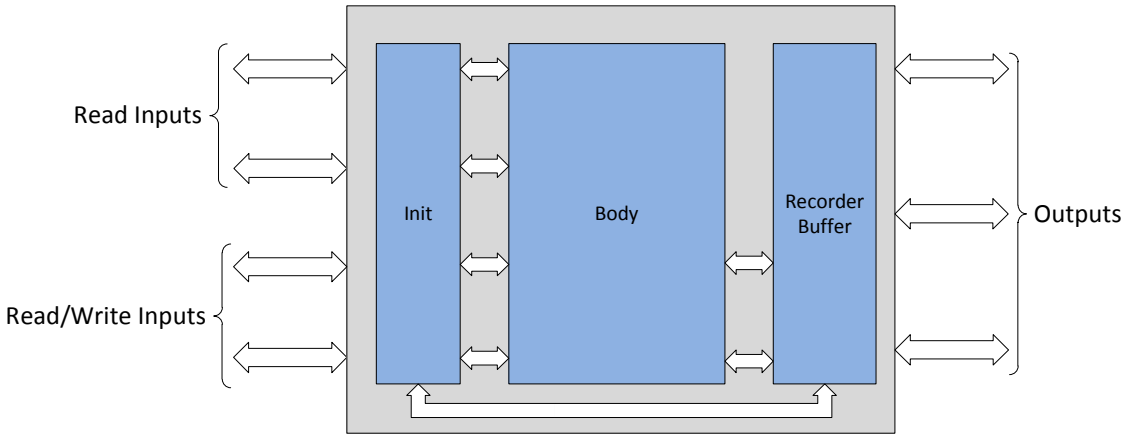


Figure 3.22: First decomposition of a for loop

- 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.

3.5.2.3 Second decomposition

Let's look specifically at blocks `init` and `memory` using figure 3.23.

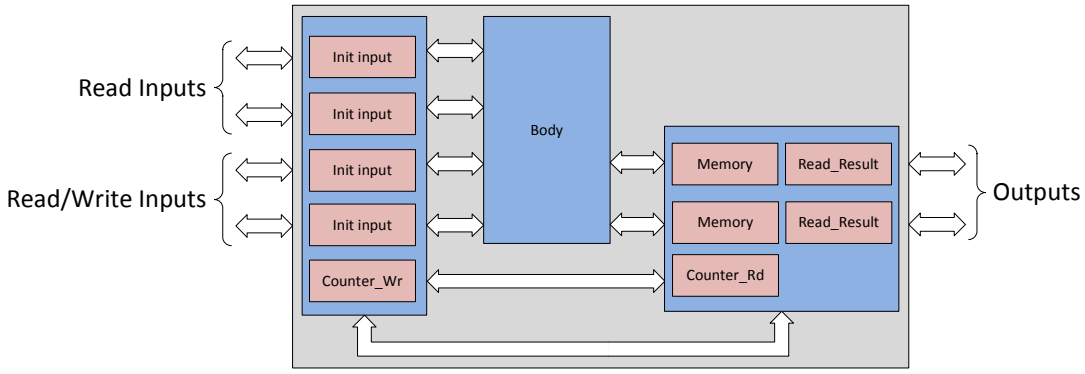


Figure 3.23: Second decomposition of a for loop

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.

3.5.2.4 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. Figure 3.24 shows the paths described above.

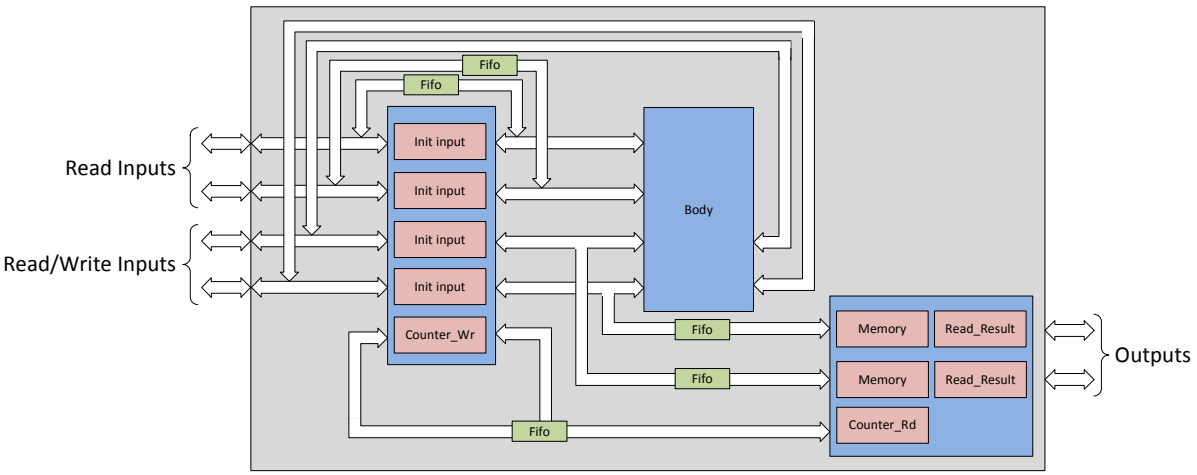


Figure 3.24: Third decomposition of a for loop

3.5.3 Body description

3.5.3.1 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 scheme shown in figure 3.25.

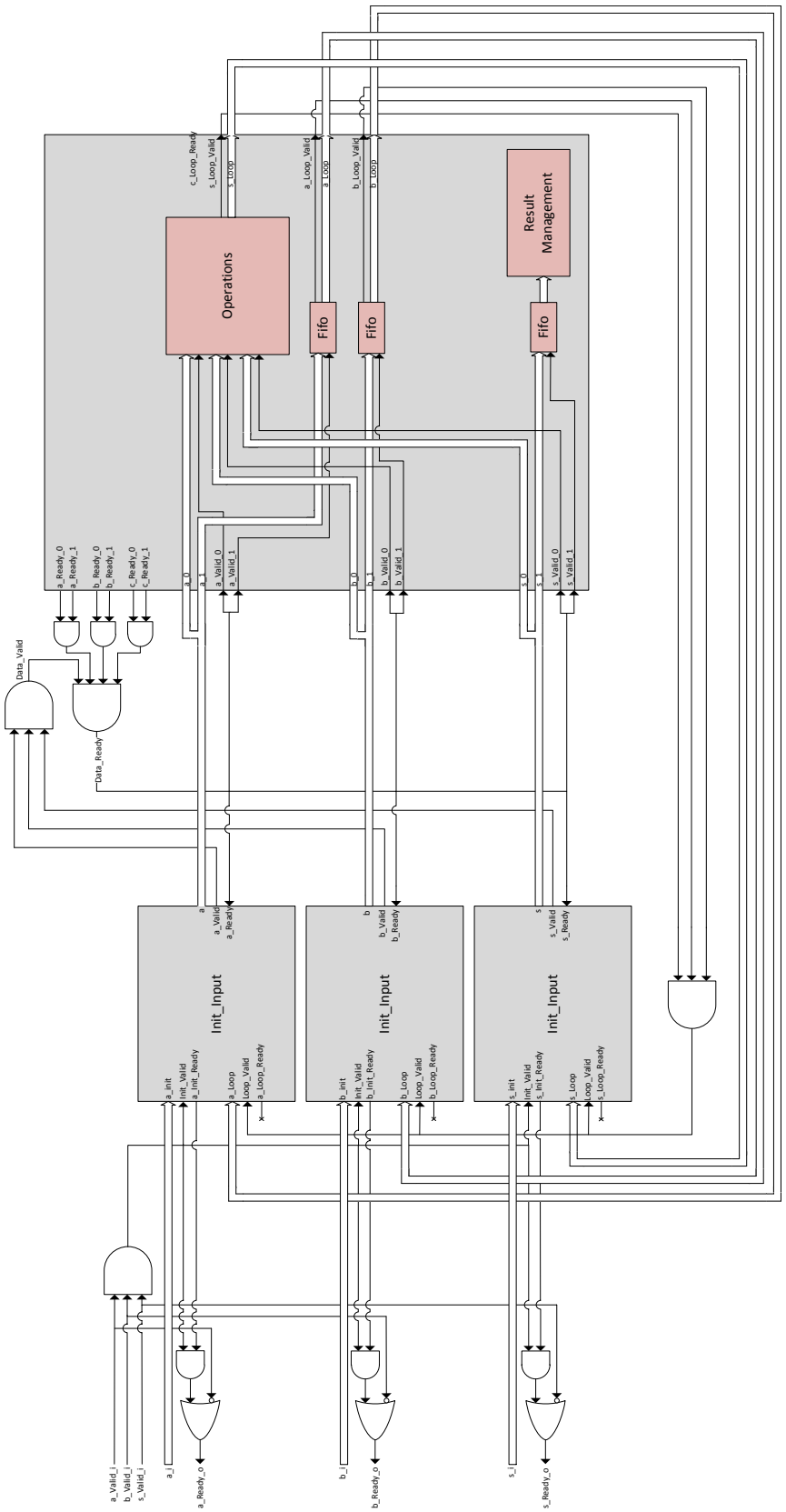


Figure 3.25: Connections between inputs, Init_Input blocks and loop inputs

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.

3.5.3.2 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. Figure 3.26 shows a case of compensation for different latencies within a loop.

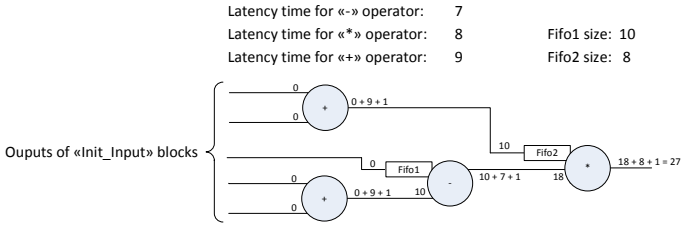


Figure 3.26: Iteration latency of a loop

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.

3.5.3.3 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.

Diagram 3.27 shows the management of the loop condition and the iterator.

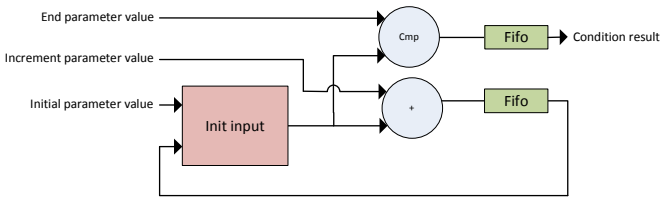


Figure 3.27: Principle of the loop condition the iterator

3.5.4 Blocks description

3.5.4.1 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`. Figure 3.28 shows the inputs / outputs of a block `Init_Input`.

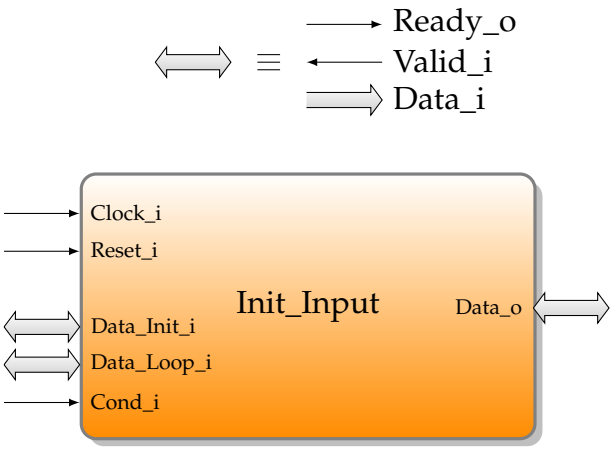


Figure 3.28: Init_Input block

3.5.4.2 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. Figure 3.29 shows the `Counter_Rd` block.

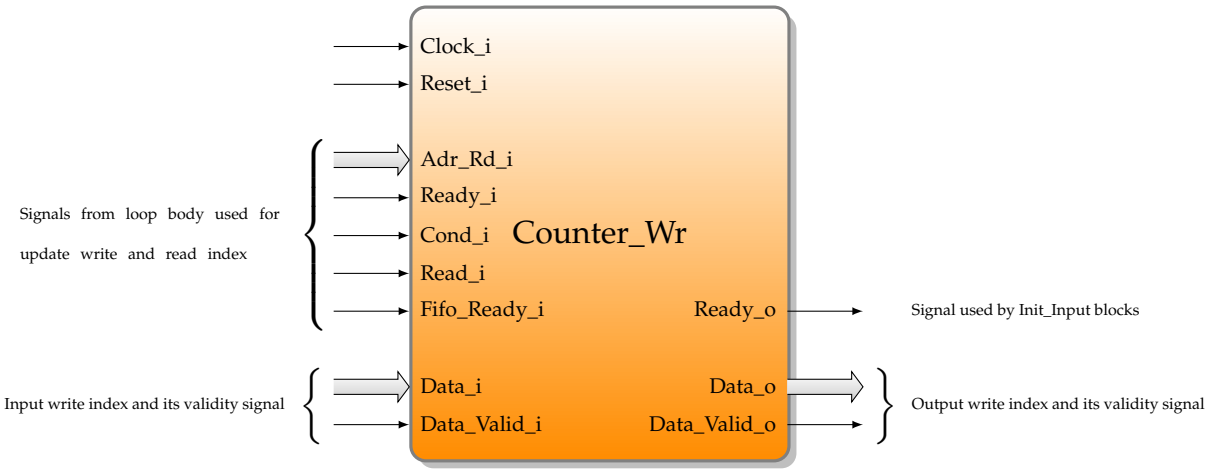


Figure 3.29: Counter_Wr block

3.5.4.3 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. Figure 3.30 describes the input/output block.

3.5.4.4 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

Figure 3.30: Counter_Rd block

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 3.31 describes the memory block.

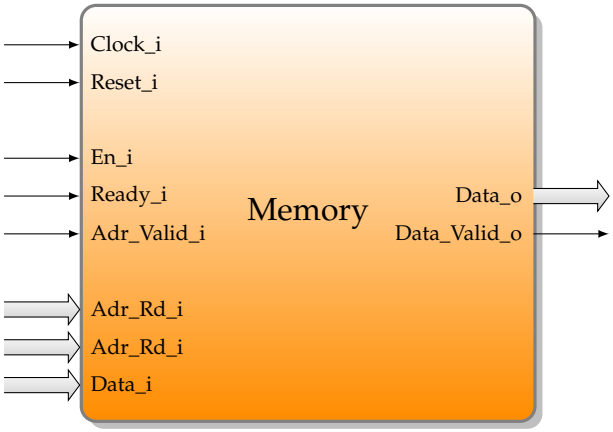


Figure 3.31: Memory block

Chapter 4

Verification

4.1 General Structure of the Testbench

Figure 4.1 shows the structure of the generated testbench.
The testbench is able to adjust easily to all types of designs with the few parameters shown in table 4.1. These parameters are configured in the `common.sv` file.
An Octave function can have different implementations in VHDL. For example, if a function takes a vector of 3 scalars as input, this can be translated into 3 different parallel inputs in the DUT, or into a unique input which takes the data values in serial. The `common.sv` file puts all these necessary parameters together to realise the configuration.
Figure 4.2 shows a graphical UML-like representation of the testbench software structure. All classes whose name begins with M2M_ and the class Reporting have been implemented for this project. All other classes belong to libraries which have been included into the project to make some tasks easier to implement.

4.2 Inputs/outputs of the DUT

Input and output names are directly inspired by the names given in the Octave function. Data input port names match exactly to the input names of the Octave function. Each data input is associated with 2 complementary signals, one that indicates that the input data is valid and takes the name of the input port with the suffix "`_Valid`", and another one that indicates that the DUT is ready to accept a new data on the input port, and takes the name of the input port with the suffix "`_Ready`".
Result output port names match exactly to the output names of the Octave function. Each result output is associated with 2 complementary signals, one that indicates that the result on the output is valid and takes the name of the output port with the suffix "`_Valid`", and another one that indicates to the DUT that the system is ready to receive a new result and takes the name of the output port with the suffix "`_Ready`". Monitored internal signals are handled as result output ports.

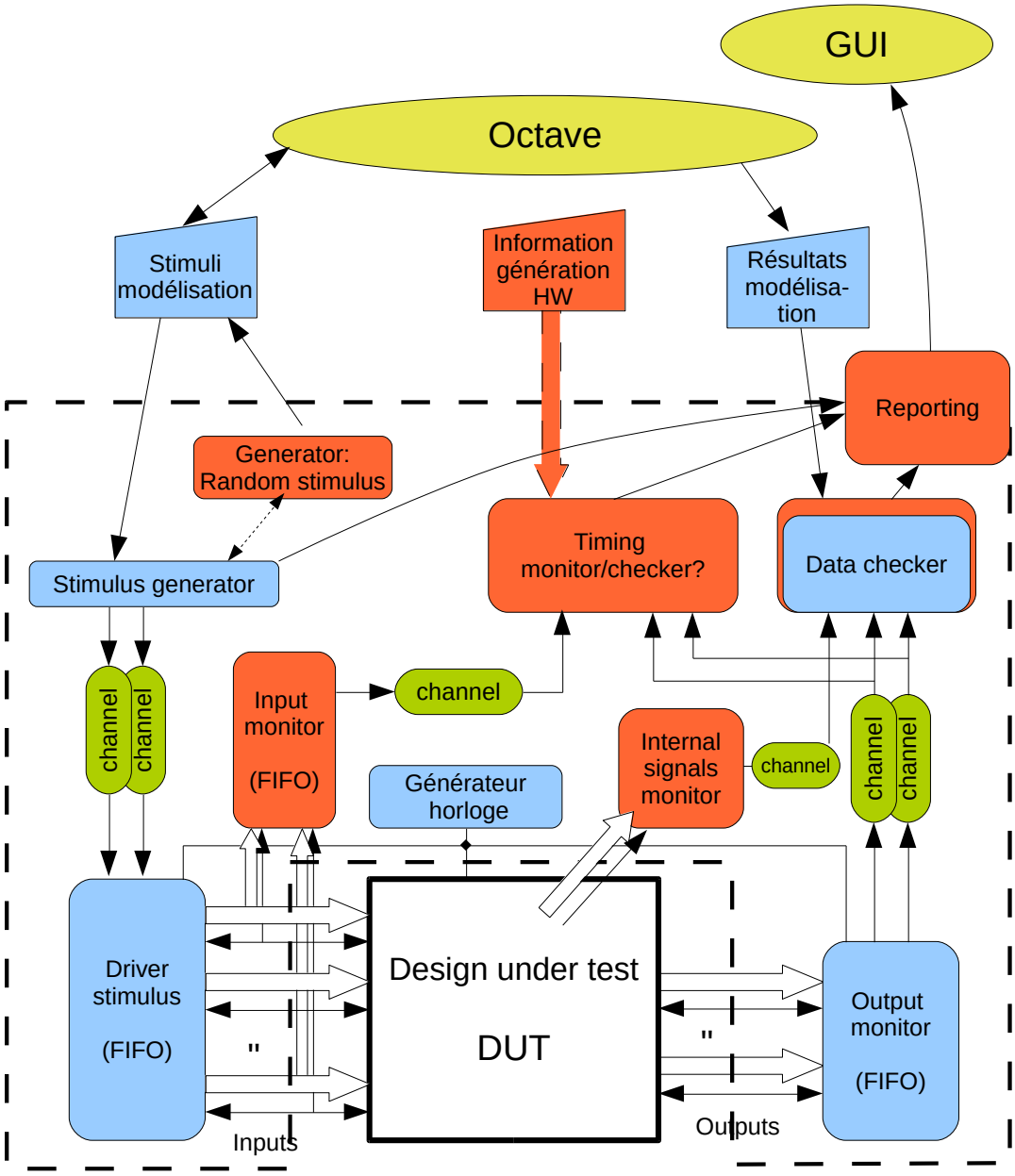


Figure 4.1: Structure of the testbench

4.3 Data File Structure

All files with data values have the same internal structure. There is one file input, internal or output variable of the Octave function. Each file begins with the number of samples. This is followed by the length of the port vector, which is 1 if it is a scalar or more than 1 if it is a vector. After that the sample values are listed.

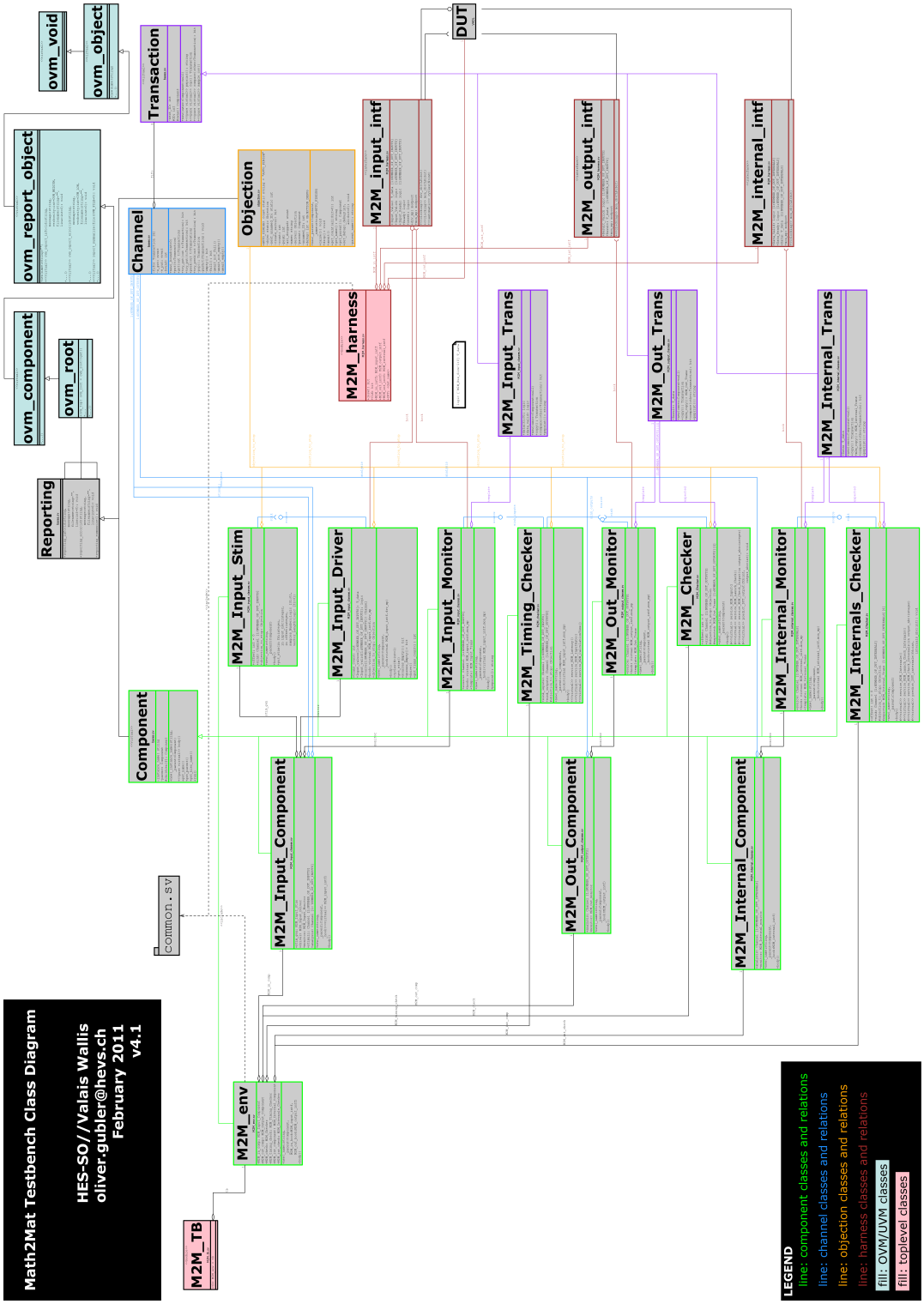


Figure 4.2: Testbench class diagram

4.4 Channels Configuration

The testbench has an array of channels (see also Figure 4.1) and there are as many channels as inputs, internals and outputs in the DUT. The size of this array will be configured to make the testbench adaptable to different DUTs, using the constants `NUMBER_OF_DUT_INPUTS`, `NUMBER_OF_DUT_INTERNALS` and `NUMBER_OF_DUT_OUTPUTS`.

4.5 Stimuli Generator

The stimuli generator reads the input data files. With the knowledge of `NUMBER_OF_INPUTS`, `NUMBER_OF_DUT_INPUTS` and `INPUT_PARALLEL` it derives the number of data samples to be read, the number of channels and which inputs are parallel or serial. For each input of the Octave function, the stimuli generator checks if this input is parallel or serial in the DUT: if the input is parallel, it will decompose the data in the file to send the 1st data on the 1st channel, the 2nd data on the 2nd channel, and so on until each channel has received data. The stimuli generator does the same job again beginning at the 1st channel. If the input is serial, the stimuli generator sends data one after another to a single channel. The data rate is controlled by the control signals coming from the DUT. The DUT indicates to the testbench if it is ready or not to accept new data. The testbench will act in consequence and send new data only when the DUT is ready to receive one. In a further step the stimuli generator will have to take care about the way to send data. This is not implemented yet.

4.6 Driver

4.7 Monitors

The three monitors for input signals, internal signals and output signals detect valid data passed on their interfaces and pass them as transaction to the corresponding channel. There is one channel per `NUMBER_OF_DUT_INPUT`, `NUMBER_OF_DUT_INTERNALS` or `NUMBER_OF_DUT_OUTPUT`.

4.8 Data Checker

The data checker works the same way as the stimuli generator does. It is based on the constants `NUMBER_OF_OUTPUTS`, `NUMBER_OF_DUT_OUTPUTS`, and `OUTPUT_PARALLEL` to determine which data read from the Octave file corresponds to the channel received from the corresponding monitor. The data checker has a data verification task (`service_M2M_Launch_Check`). It runs this task as many times as there are output channels, i.e. as many as there are DUT outputs. Each task will examine one and only one channel. The function `predict_DUT_output` has the responsibility

to parse output data files. It reads a file, extract data and spreads theses data in one or more queues depending if the result is expected serial or parallel. When the task receives a transaction from the output monitor, it compares this transaction with the data contained in the queue that corresponds to the channel. Monitored internal signals are handled the same way.

4.9 Timing Monitor

The timing monitor receives data samples from all other monitors. As the channel transactions are executed in zero simulation time, these data samples can be used for timing monitoring. The statistics shown in Table 4.2 are monitored. As this monitor works in concurrence with the data checker, the function peek is used to get the transactions from the channel without destroying them.

4.10 Testbench coverage

The testbench stops when all raised objections have been dropped. Tasks that raise objections are the following:

- `M2M_Input_Driver:body()` ⇒ the testbench cannot stop while all data in input files have not been sent to the DUT (through the tasks `get_stim()` and `drive()`).
- `M2M_Input_Stim:body()` ⇒ opens the stimuli files and launches the `put_stim()` task as many times as there is inputs in the Octave function.
- `M2M_Input_Stim:put_stim()` ⇒ the testbench cannot stop while all stimuli have not been read in the files and placed into channels.
- `M2M_checker:service_M2M_Check_Output` ⇒ the testbench cannot stop while it receives data from the output monitor.
- `M2M_internals_checker:service_M2M_Check_Internals` ⇒ the testbench cannot stop while it receives data from the output monitor.
- `M2M_timing_checker:service_M2M_Latency` ⇒ the testbench cannot stop while it is measuring the latency
- `M2M_timing_checker:service_M2M_CalculationTime` ⇒ the testbench cannot stop while it is measuring the total calculation time
- `M2M_timing_checker:service_M2M_InputRate` ⇒ the testbench cannot stop while it is measuring the input rate

In this configuration, the testbench uses all the samples that are in input files, but does not do any verification on output files (i.e. there is no error if the testbench stops before all values in output files have been treated).

4.11 Reporting

Reporting is unified in the Reporting class (see 4.2). It provides reporting functions to the testbench to report information messages (reporting_info) or error messages (reporting_error) during the simulation run as well as a summary of occurred messages at the end of the simulation run (reporting_summarize). All messages are sent to the GUI and stored to a local file (`transcript.txt`). The GUI filters the messages according to its needs.

4.12 Implementation

The OVM library1 is used to implement these functions. This choice has been done, as the OVM library is a free and open source library based on SystemVerilog and is supported by most of the EDA tools vendors. Furthermore this approach allowed us to implement a reporting function in a short time, to have more time to concentrate on the real verification parts of the TB. If needed, the implementation of these functions can be replaced in a way to be independent of any library.
On a further action to be independent of precompiled libraries provided by the simulator vendors, the used source files are delivered with the testbench and compiled on each testbench run, as seen in the simulation script excerpt below.

Simulation script

```
## compile ovm
vlib m2mOvm
vmap m2mOvm $Path/comp/m2mOvm
vlog -work m2mOvm +incdir+$srcSVPath/ovm/src
   $srcSVPath/ovm/src/ovm_pkg.sv
```

Furthermore the simulation command had to be adapted to use the compiled OVM library instead the precompiled one as shown below.

Simulation script

```
## compile project
#vlog -work $workPath $srcSVPath/M2M_harness.sv
   $srcSVPath/M2M_tb.sv
vlog -work $workPath -L m2mOvm
   $srcSVPath/M2M_harness.sv $srcSVPath/M2M_tb.sv
#vsim -vopt work.M2M_TB work.M2M_harness
vsim -vopt -L m2mOvm work.M2M_TB work.M2M_harness
```

On a nice side-effect, the furnished OVM source code has been altered such that messages wear the prefix M2M_ instead of OVM_ (see also 4.15).

4.13 Format

The different fields of a message as shown in Table 4.3 are separated by space and additional markers (please refer to 4.15). SystemVerilog provides the same severity levels natively.

4.14 Examples

Below some examples for messages generated with the Reporting's functions.

Examples

```
# M2M_INFO ../src_SV/bases.sv(181) @ 0: tb [tb]
   Constructed tb

# M2M_INFO ../src_SV/objection.sv(313) @ 0:
   AUTO_FINISH [objection.AUTO_FINISH] Objections to
   AUTO_FINISH from tb.M2M_in_comp.stim_gen: new peak
   = 2

# M2M_INFO ../src_SV/M2M_input_classes.sv(435) @
   50345: driver [tb.M2M_in_comp.driver] task
   tb.M2M_in_comp.driver.get_stim finished

# M2M_ERROR ../src_SV/M2M_checker.sv(259) @ 775: check
   [tb.check] Bad DUT output no. 1.4956: M2M_Out_Trans
   : Result = 0xbe3247ea = -0.174102 expected:
   M2M_Out_Trans : Result = 0xbe3247e6 = -0.174102
```

4.15 Summary

The following summary is generated at the end of the simulation run. It shows the number of messages grouped by severity level and by ID.

Summary

```
# --- OVM Report Summary ---
#
# ** Report counts by severity
# M2M_INFO : 4643
# M2M_WARNING :    0
# M2M_ERROR :  457
# M2M_FATAL :    0
# ** Report counts by id
# [objection.AUTO_FINISH]    13
# [tb]     3
# [tb.M2M_in_comp]     3
# [tb.M2M_in_comp.driver]    59
# [tb.M2M_in_comp.monitor]     3
# [tb.M2M_in_comp.stim_gen]     4
# [tb.M2M_out_comp]     3
# [tb.M2M_out_comp.monitor]     2
# [tb.check]  5003
# [tb.timing_check]     7
Questa verification finished
```

Name	Type	Description
M2M_Bus_Size	integer	
NUMBER_OF_INPUTS	integer	number of inputs of the Octave function
NUMBER_OF_DUT_INPUTS	integer	number of inputs of the DUT
READ_INPUT_WAY	bit[1:`NUMBER_OF_INPUTS]	the way to read data when the input is a vector (not used now)
INPUT_PARALLEL	bit[1:`NUMBER_OF_INPUTS]	array with a size equal to NUMBER_OF_INPUTS, which informs if an input is built in parallel (1) or in serial(0) in the DUT
NUMBER_OF_OUTPUTS	integer	number of outputs of the Octave function
NUMBER_OF_DUT_OUTPUTS	integer	number of outputs of the DUT
READ_OUTPUT_WAY	integer	the way to read data when the output is a vector (not used now)
OUTPUT_PARALLEL	bit[1:`NUMBER_OF_OUTPUTS]	array with a size equal to NUMBER_OF_OUTPUTS, which informs if an output is built in parallel (1) or in serial (0) in the DUT
CLOCK_TICKS	integer	number of simulation time ticks during one clock cycle used in harness to generate the system clock used in checker to calculate clock timings
NUMBER_OF_INTERNALS	integer	number of internals of the Octave function
NUMBER_OF_DUT_INTERNALS	integer	number of internals of the DUT
INTERNAL_PARALLEL	bit[1:`NUMBER_OF_INTERNALS]	array with a size equal to NUMBER_OF_INTERNALS, which informs if an internal is built in parallel (1) or in serial (0) in the DUT

Table 4.1: Testbench parameters

Indicator	Measure	Calculation	Remarks
Total Input Time	First input to last input	-	-
Input Rate	-	TotalInputTime/ (NumberOfInputs-1)	Average time between two data inputs
Latency	First input to first output	-	For a future version, the average latency of all input/output pairs could be measured and calculated
Total Calculation Time	First input to last output	-	Total time the DUT is processing data

Table 4.2: Timing monitor statistics

Field	Description
Severity	Can be Error or Info/Warning
Tag	Depending on the severity: M2M_ERROR or M2M_INFO
File	name of the file the message originates from
Line	line in the file the message originates from
Time	simulation time in simulator time units
ID	name of the object the message originates from
Message	refer to examples in 4.14

Table 4.3: Simulation message format