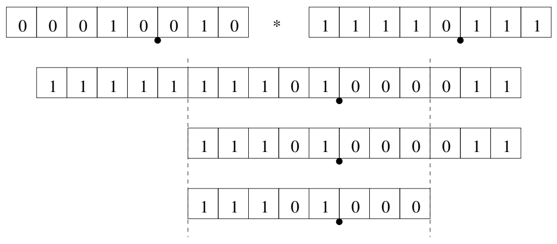
Documentation on bitwidth in MMALPHA

Edouard BECHETOILLE

September 21, 2005



Abstract

This document explains how basic hardware arithmetic is handle by MMALPHA. An ALPHA code can be written without having to care about size of bits of different variables. However, an ALPHA user considered as a Hardware Designer, may intend to control the size of every variable of his design. The different steps to set the bitwidth of an ALPHA variable are detailed here. Many examples are used to put lights on as many possibilities as possible to manipulate bitwidth. At last, the explanation of the ALPHA processing is explained for alpha developer to permit faster modifications or upgrades of this topic.

Contents

List of Figures

List of Tables

1 Introduction

One of the powerful possibility with MMALPHA remains in the parametric expression of an application. Indeed a parametric expression permits to keep a loop unrolled, which accelerate processing upon the application. Alpha generates a virtual Cell that is repeated as needed. Here comes the necessity of truncation or reduction due to the iteration of a common Cell. If the size of such a cell increases while iterating, the final design would be too big. As a matter of fact the controlling of data bitwidths' is essential and detailed hereafter. To make things clear on data types, a reminder on Fix Point Coding is presented. Ways and means to set datas bitwitdhs' are explained in Part 2 . Finally advanced MMAlpha developer will find all needed information on how bitwidth manipulation had been implemented.

2 User documentation

This part gives all clues to use and check calculations in MMALPHA.

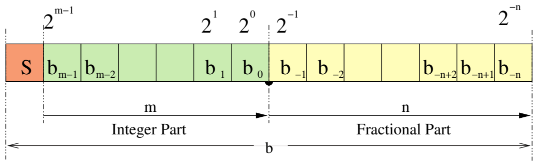
2.1 Use of Fixed Point

2.1.1 Reminder on Two's Complement

This part is inspired by the document [1]. The way to code negative number is not detailed in this document.

- **Definition :**
  A real number x is represented in Two's Complement as follows. S and b_i are equal to 0 or 1. $x = -2^m S + \sum_{i=-n}^{m-1} b_i 2^i$

  m : distance between the sign bit and the point
  n : distance between the point and the least significant bit
  b : size of bit of the container : bitwidth



- **Format :** We consider the coding format as : {**b,m,n**} with the condition that **b=m+n+1**. '1' represent the sign bit. Such a choice defines the domain of the coding D_c : $[-2^m, 2^m - 2^{-n}]$ with a quantification step q defined by : $q = 2^{-n}$. Once the value b is defined, the size of the container is set. And so wherever the Fixed Point is, it is possible to interpret the data as a signed integer. Doing so is equivalent to interpret a code {b,m,n} as a Signed Integer Form code {b,b-1,0}.

  For instance, considering the code {6,3,2}. It is possible to code numbers in the Domain : [-8,7.75] with 0.25 as quantification step. See Figure 1 which represent the following list {7.75,7.5,0.75,0.5,0,-0.25,-0.5,-0.75,-7.5,-7.75,-8} coded with {6,3,2}

If a number is out of the definition domain, it can not be coded without unexpected errors. But if a number is not divisible by the quantification step, it would be coded but approximated to the nearest underneath value.

2.1.2 Arithmetics

At the input of an arithmetic operator, the operands bitwidths' need to be equal. As shown in Figure 2, the containers are extend with signed bit if integer part is too short or with zeros if fractional part is too short. To use fixed point coding, it is essential to know the format of operands. The Format defines the definition domain of the coding D_c. But during an operation, the result could get out of this domain D_c. There is indeed two alternatives. One can increase the size of the container at the output of the operation, or one can keep the same size of container to reduce the complexity. In the second option, one must be sure that

InputForm	6bits Digits	FixedPointForm	InterpretedValue	IntegerSignedForm
7.75	011111	0111.11	7.75	31
7.5	011110	0111.10	7.5	30
0.75	000011	0000.11	0.75	3
0.5	000010	0000.10	0.5	2
0	000000	0000.00	0	0
-0.25	111111	1111.11	-0.25	-1
-0.5	111110	1111.10	-0.5	-2
-0.75	111101	1111.01	-0.75	-3
-7.5	100010	1000.10	-7.5	-30
-7.75	100001	1000.01	-7.75	-31
-8	100000	1000.00	-8.	-32

Figure 1: Example of two's complement coding

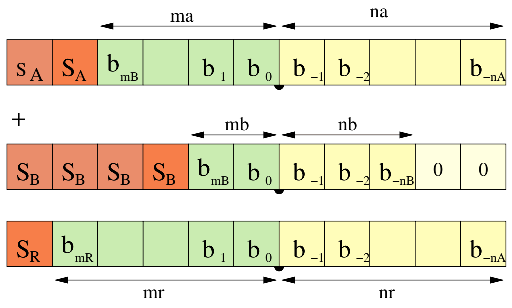


Figure 2: bitwidth representation of an addition

the datas are small enough to stay in D_c after computing. let A, B be the operands and R the Result with their respective codes $\{b_x, m_x, n_x\}$ with $b_x = m_x + n_x + 1$. In case of :

$$\begin{cases} \text{addition,} & \begin{cases} b_R = m_R + n_R + 1 \\ m_R = & \begin{cases} max(m_A, m_B) & \text{if } A + B \in D_c \\ max(m_A, m_B) + 1 & \text{if } A + B \notin D_c \end{cases} \\ n_R = max(n_A, n_B) \end{cases} \\ \text{multiplication,} & \begin{cases} b_R = m_R + n_R + 1 \\ m_R = m_A + m_B + 1 \text{ doubling of the sign bit is added to the integer part} \\ n_R = n_A + n_B \end{cases} \end{cases}$$

In MMALPHA it has been defined[1] that $b_R = max(b_A, b_B) + 1$ in case of an addition, and $b_R = b_A + b_B$ in case of a multiplication. Figure 2 illustrate bitwidth of addition.

2.2 How to set bitwith in MMAlpha

In this part, we consider that in Mathematica an alpha program had been loaded, scheduled, mapped, pipelined as wishes and transformed into Alpha0. Bitwidths had to be set after the Alpha0 process and before the Translation in AlpHard. Once bitwidths had been set, to generate VHDL, the remaining steps

[1]See 3.1.4 for developer.

are: **alpha0ToAlphard**, **fixParameter** if needed, **a2v**, **vhdlTestBenchGen** to get the Test Bench and Stimuli Files generation.

To locate the variables to modify, take a look to the system **$result** by typing : **ashow[]**. It results in something like :

```
system prodVect :{N | 3<=N}
                 (a : {i,j | 1<=i<=N; 1<=j<=N} of integer[S,16];
                  b : {i | 1<=i<=N} of integer[S,16])
       returns   (c : {i | 1<=i<=N} of integer[S,16]);
var
...
  aReg2 : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,16];
  CReg1 : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,16];
  pipeCb1 : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,16];
let
 CReg1[t,p] = CReg1Xloc[t-1,p];
 aReg2[t,p] = a[t-p+1,p];
...
  C[t,p] =
      case
        { | 0<=t<=N-1; p=0; 3<=N} : 0[];
        { | p<=t<=p+N-1; 1<=p<=N; 3<=N} : CReg1 + aReg2 * pipeCb1;
      esac;
...
```

To control the output bitwidth of the multiplication operator, a temporary variable had to be added. The command **addLocal["prod","aReg2*pipeCb1"];** performs the task. A variable "prod" is added. The modifications in the Abstract Syntax Tree (AST) viewed with **ashow[]** are showed underneath.

```
var
  prod : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,32];
...
let
  prod[t,p] = aReg2 * pipeCb1;
...
  C[t,p] =
      case
        { | 0<=t<=N-1; p=0; 3<=N} : 0[];
        { | p<=t<=p+N-1; 1<=p<=N; 3<=N} : CReg1 + prod;
      esac;
```

It is to notice that the size of prod is indeed 32 twice bigger as the size of aReg2 and pipeCb1. From this point, we might have different choices. Regarding the type of datas one has to deal with, the different kind of truncation to apply are stated further. We detail in this paper three cases : Integer, Real and Fraction$\in [0,1[$.

2.2.1 Truncate integer datas

In case of Integer, the coding is a peculiar case. Indeed no fractional part means that the code is : {b,b-1,0}. As explained before, the definition domain to code integer became D_c : $[-2^{b-1}, 2^{b-1}-1]$ In our example b=16 so $D_c = [-32768, 32767]$. It is important to call back that we want this domain to be also the domain of the result. So operand values needs to be much smaller than the bounds of the domain. If inputs values are limited to the half of the bitwidth b, the result would be for sure in the domain, but it is not a necessity

InputForm	16bits Digits	FixedPointForm	InterpretedValue	IntegerSignedForm
-40	1111111111011000	1111111111011000.	-40.	-40
30	0000000000011110	0000000000011110.	30.	30
-1200	1111101101010000	1111101101010000.	-1200.	-1200
48000	1011101110000000	1011101110000000.	-17536.	-17536

Figure 3: Example of values coded with {16,15,0}

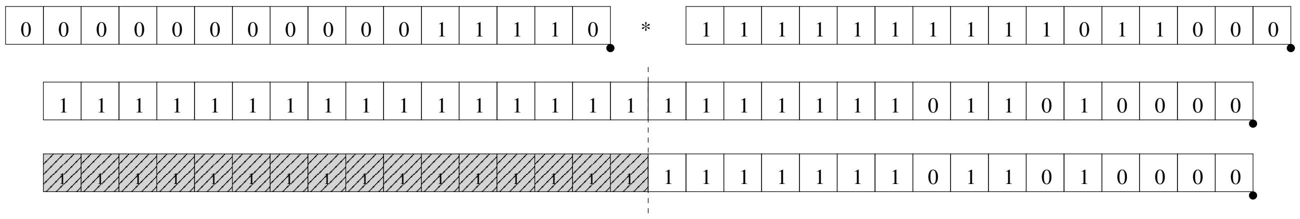


Figure 4: Truncation made by the function Truncate_MSB

Example : 30 and -1200 belongs to D_c, but 30*-1200=-4800 is not in D_c. The last value in Figure 3 : 48000 cannot be coded, and would be interpreted has -17536 because of truncating.

Now let see an example where no errors occurs. The multiplication of -40 times 30 results in -1200. The Figure 4 explains that the ′information′ of the data -1200 is on the right of the container. Thus we would be able to truncate the 16 bits on the left, with the function **Truncate_MSB** made for this.

As we could, see, because we are dealing with integer value, only one truncation is needed. So coming back to our modification of the AST, once **addLocal[″prod″,″operandA*operandB″];** had been executed, another intermediate variable has to be inserted with the command **addLocalLHS[″prodTrunc″, ″prod″];**. Type **ashow[]** to see the following modification in the AST:

```
var
  prodTrunc : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,32];
  prod : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,32];
...
let
  prodTrunc[t,p] = prod;
  prod[t,p] = aReg2 * pipeCb1;
...
  C[t,p] =
      case
        { | 0<=t<=N-1; p=0; 3<=N} : 0[];
        { | p<=t<=p+N-1; 1<=p<=N; 3<=N} : CReg1 + prodTrunc;
      esac;
```

Then we apply the function **Truncate_MSB** to this variable prodTrunc with the command : **insertFunction[″prodTrunc″, ″Truncate_MSB″];**. **ashow[]** permit again to see the modifications of the AST :

```
let
  prodTrunc[t,p] = Truncate_MSB(prod);
```

Finally, we force the bithwidth to be equal to the wanted value, in this case : 16, with the function : **setBitWidth[″prodTrunc″,16];** which modifies the AST in :

```
var
  prodTrunc : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,16];
```

Now please refer to **matvect_truncate_MSB.nb** to execute this example inorder to verify it by yourself.

InputForm	8bits Digits	FixedPointForm	InterpretedValue	IntegerSignedForm
2.875	00010111	00010.111	2.875	23
-1.125	11110111	11110.111	-1.125	-9
-3.23438	11100110	11100.110	-3.25	-26

Figure 5: bitwidth representation of a multiplication

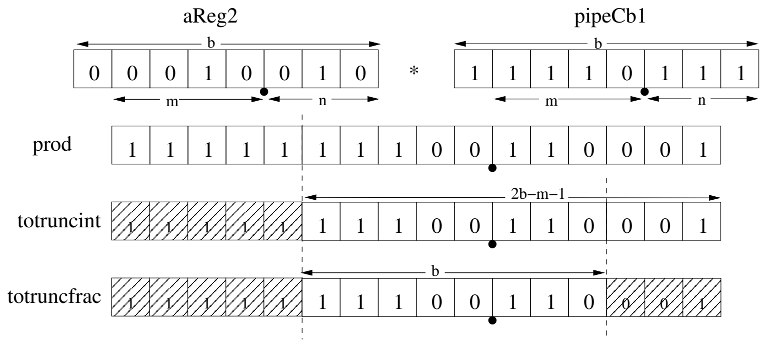


Figure 6: Truncation made by Truncate_MSB and Truncate_LSB

2.2.2 Truncate real datas

The notebook **matvect_truncate_Real.nb** shows how to truncate values with Integer Part and Fractional Part. This case is the general case. All kind of value code {b,m,n} with b=m+n+1 are covered by this part. To obtain a correct result in case of multiplication, check that input values belongs to {m/2+n+1,m/2,n}. Thus by multiplication operation, result will belongs to the domain of the coding D_c=$[-2^m, 2^m - 2^{-n}]$. However, there is less care to have for the fractional part. Indeed, if the result of a multiplication need a smaller quantification step- that is to say the fractional part 'n' need more bits- then the result will not be false, but approximated. A simple example will clear this out. If you are working with Mathematica while reading this notebook, do not hesitate to type:

```
<< TwosFunctions.m;
TwosTable[{2.875,-1.125,-1.125*2.875},{8,4,3},printDom->True, message->True]
```

It shows the value of two operands in the Domain defined by {8,4,3} and the result of their multiplication. As you could see in Figure 5 the result -3.234375 is truncated and will be approximated to -3.25.

Now let see how to specify such a truncation in MMAlpha. As explained before, the alpha system had already been translated in Alpha0. And in this system exists an element which defines a multiplication of which we would like to control the bitwidth, by coding the datas with the Encode:{b,m,n}. The operation : ″operandA*operandB″ had been set to ″prod″ with **addLocal[″prod″,″operandA*operandB″];**. In order to truncate the variable **prod** which has the encode {2b,2m+1,2n} to a variable that has the encode {b,m,n}, two steps are needed. First, we will truncate the exceeding bits on the left with **truncate_MSB**. The size of the outputted variable container had to be specified (totruncint for instance) with its biwidth b′=m+2n+1=2b-m-1. Secondly, on this resulted variable, the exceeding bits on the right will be removed with **truncate_LSB**. The container of the outputted variable had to be specified (totruncfrac for instance) with its bitwidth b″=b. See Figure 6 to fix the ideas.

To perform this task in MMAlpha, use the following steps. Specify the encoding you need by giving Integer values to {b,m,n}. Add a local variable **totrunctint**. Insert the function **truncate_MSB**. And set the bitwith to **2b-m-1**.

```
Coding={16,6,9};
addLocalLHS["totruncint","prod"];
```

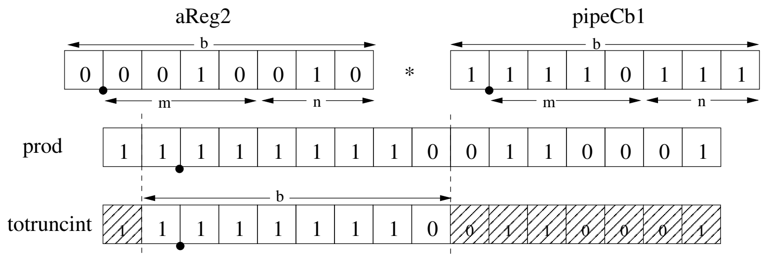


Figure 7: Effect of procedure Truncate_LSSB

```
insertFunction["totruncint","Truncate_MSB"];
setBitWidth["totruncint",Coding[[2]]+2*Coding[[3]]+1];
```

Repeat the last three steps with **totruncfrac**, exept that addLocal is not applied to prod but to totruncint, and that the bithwidth had to be set to b.

```
addLocalLHS["totruncfrac","totruncint"];
insertFunction["totruncfrac","Truncate_LSB"];
setBitWidth["totruncfrac",Coding[[1]]];
```

Those steps gives modifications to the alpha system example shown underneath.

```
...
var
  totruncfrac : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,16];
  totruncint : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,25];
  prod : {t,p | p<=t<=p+N-1; 1<=p<=N; 3<=N} of integer[S,32];
...
let
  totruncfrac[t,p] = Truncate_LSB(totruncint);
  totruncint[t,p] = Truncate_MSB(prod);
  prod[t,p] = aReg2 * pipeCb1;
...
  C[t,p] =
      case
        { | 0<=t<=N-1; p=0; 3<=N} : 0[];
        { | p<=t<=p+N-1; 1<=p<=N; 3<=N} : CReg1 + totruncfrac;
      esac;
```

It is important to repeat that this case is a general case, so all kind of truncation could be done this way. However one might want to use less steps such as described in Section 2.2.1 and 2.2.3.

2.2.3 Truncate fractional datas between 0 and 1

If $Datas \in [0, 1[$, no Integer part is needed. The encoding is equivalent to {b,0,b-1}. The multiplication of two variables **aReg2** and **pipeCb1** with an encoding {**b,0,b-1**} will result in an encoding {**2b,1,2b-2**}. Warning, watch out that a kind of bit appears in the integer part. This phenomena is often called the doubling of the sign bit. See Figure 7 to see such a representation. In order to take this case into account, the VHDL procedure **Truncate_LSSB** had been added to the library **Definition.vhd**. Hence, for an encoding {16,0,15}, the doubled sign bit and the exceeding bits on the right will be removed by executing the following steps:

```
addLocalLHS["prodTrunc","prod"];
insertFunction["prodTrunc","Truncate_LSSB"];
setBitWidth["prodTrunc",16];
```

Encoding : {b,m,n} with b=m+n+1 Hypothesis : it exists in the alpha system C=D+A*B Truncation		
Integer	Real	Fraction
addLocal["prod","A*B"] addLocalLHS["Res", "prod"] insertFunction["Res","Truncate_MSB"] setBitWidth["Res",b]	addLocal["prod","A*B"] addLocalLHS["Res", "prod"] insertFunction["Res","Truncate_MSB"] setBitWidth["Res",m+2n+1] addLocalLHS["Result", "Res"] insertFunction["Result","Truncate_LSB"] setBitWidth["Result",b]	addLocal["prod","A*B"] addLocalLHS["Res", "prod"] insertFunction["Res","Truncate_LSSB"] setBitWidth["Res",b]

Table 1: Summary table for truncation

See notebook : **matvect_truncate_LSB.nb** for full example.

2.2.4 Reduction

The VHDL reduction procedure check that no overflow or underflow occurs while truncating a value. No overflow or underflow occurs when only redundant sign bits are removed. Overflow represents a truncating error of a positive data. In such a case, the outputted value is set to the highest value. Underflow represents a truncating error of a negative data. In such a case, the outputted value is set to the lowest value. This procedure is useful in case of Integer value to saturate the output instead of adding unexpected errors.

To see the effect of the **Reduce_MSB** procedure, we need input values to reach the bounds of the encoding. Warn that the limits would also be reached for the '+' operator if used in the application. In the example of matrix vector multiplication, there is multiplications and additions. Thus we will force the addition to use a '+' operator that take into account the case explained Page 2.1.2 : $m_R = max(m_A + m_B) + 1 if A + B \notin D_c$.

```
addLocal["prod","aReg2*pipeCb1"];
addLocalLHS["prodTrunc","prod"];
insertFunction["prodTrunc","ProcedureReduce_MSB"];
setBitWidth["prodTrunc",16];
addLocal["plusT","CReg1+prodTrunc"];
addLocalLHS["plusTrunc","plusT"];
insertFunction["plusTrunc", "ProcedureReduce_MSB"];
setBitWidth["plusT",17];
```

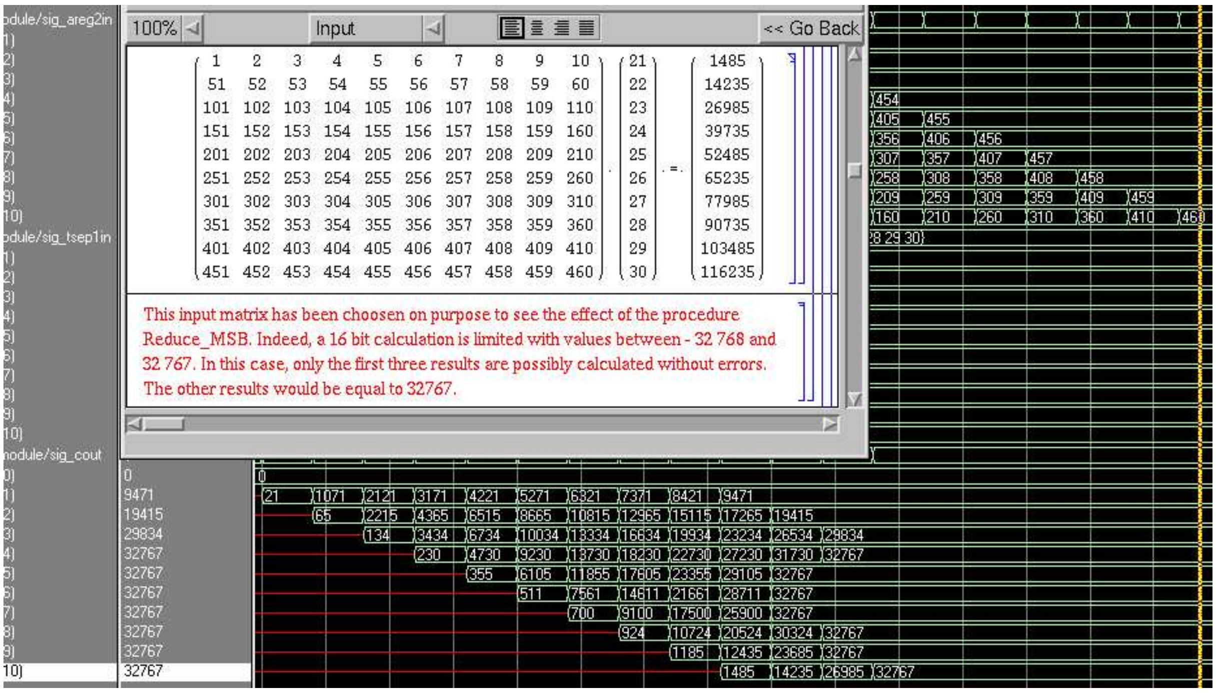
Variable plusT is indeed set to $17bits = max(16, 16) + 1$ in this example. This added bit requires an addition operator which take into account the last carry. Such operator had been added to the file Definition.vhd as a function. Hence came the necessity to manually change the generated VHDL files where the operator '+' is used. Instead of "plusT ¡= CReg1 + prodTrunc;", write "plusT ¡= plusOne(CReg1, prodTrunc);". If you evaluate the notebook **matvect_reduce_MSB.nb** and if the above modification is made to cellprodVectModule1.vhd and cellprodVectModule2.vhd the Figure **??** might be observed in ModelSim.

2.2.5 summary of truncation to apply

See Table 1.

2.3 VHDL procedures in Definition.vhd

To perform the MMAlpha truncation tool, we choose VHDL procedure instead of function. The advantage of a procedure is that the Attribute Length of the output signal : outSignal'length may be used in the body



$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 \\ 101 & 102 & 103 & 104 & 105 & 106 & 107 & 108 & 109 & 110 \\ 151 & 152 & 153 & 154 & 155 & 156 & 157 & 158 & 159 & 160 \\ 201 & 202 & 203 & 204 & 205 & 206 & 207 & 208 & 209 & 210 \\ 251 & 252 & 253 & 254 & 255 & 256 & 257 & 258 & 259 & 260 \\ 301 & 302 & 303 & 304 & 305 & 306 & 307 & 308 & 309 & 310 \\ 351 & 352 & 353 & 354 & 355 & 356 & 357 & 358 & 359 & 360 \\ 401 & 402 & 403 & 404 & 405 & 406 & 407 & 408 & 409 & 410 \\ 451 & 452 & 453 & 454 & 455 & 456 & 457 & 458 & 459 & 460 \end{pmatrix} . \begin{pmatrix} 21 \\ 22 \\ 23 \\ 24 \\ 25 \\ 26 \\ 27 \\ 28 \\ 29 \\ 30 \end{pmatrix} = \begin{pmatrix} 1485 \\ 14235 \\ 26985 \\ 39735 \\ 52485 \\ 65235 \\ 77985 \\ 90735 \\ 103485 \\ 116235 \end{pmatrix}$$

This input matrix has been choosen on purpose to see the effect of the procedure Reduce_MSB. Indeed, a 16 bit calculation is limited with values between - 32 768 and 32 767. In this case, only the first three results are possibly calculated without errors. The other results would be equal to 32767.

Figure 8: Saturation to 32767 due to the procedure Reduce_MSB

of the procedure, even for an Unconstrained output signal. Therefore, no additional input size parameter were include. Thanks to this possibility, truncation procedure are really simple.

procedure Truncate_MSB(SIGNAL totrunc : IN SIGNED; SIGNAL truncated : OUT SIGNED) is
Begin truncated <= totrunc(totrunc'low + truncated'length-1 downto totrunc'low); end Truncate_MSB;
procedure Truncate_LSB(SIGNAL totrunc : IN SIGNED; SIGNAL truncated : OUT SIGNED) is
Begin truncated <= totrunc(totrunc'high downto totrunc'high -truncated'length+1); end Truncate_LSB;
procedure Truncate_LSSB(SIGNAL totrunc : IN SIGNED; SIGNAL truncated : OUT SIGNED) is
Begin truncated <= totrunc(totrunc'high-1 downto totrunc'high -truncated'length); end Truncate_LSSB;

Figure 9 represents how the input **totrunc** and the output **truncated** are aligned.

3 Developer documentation

3.1 Reminder on VHDL Translator

3.1.1 From Alpha to VHDL

In a notebook, at the loading of a program in Mathematica, an Abstract Syntax Tree (AST) is created. From that point, every modification made on this AST is made by analysing its content and modifying it if wishes. For instance, **analyze[]** checks that the program is written correctly and that all the definition domains of all variables are correct. One might want to **schedule[]** the program, and map the recognized dependences to the system with the function **appSched[]**. There is also possibilities to pipeline variable if necessary. Once the alpha program had been manipulated by aim of application, it is time to transform it in the subset Alpha0 language. The operation to the AST is made during the step **toAlpha0v2[]**. This step add also control signals to what will be the hardware architecture. It is the right moment to apply truncation to the wanted elements in respect to the chosen type of coding. Then the finals steps are : **alpha0ToAlphard[]**, **fix-**

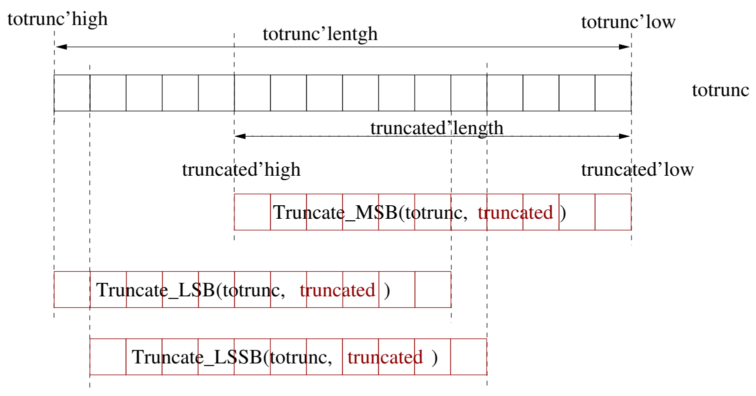

totrunc'high
totrunc'lentgh
totrunc'low
totrunc
truncated'length
truncated'high
truncated'low
Truncate_MSB(totrunc, truncated)
Truncate_LSB(totrunc, truncated)
Truncate_LSSB(totrunc, truncated)

Figure 9: Truncation alignement

Parameter["Param", value], **$library = Drop[$library,-1];** to remove the variable used to create the system elements, **a2v[$library]**, **getSystem["NameOfTheSystemModule"]**, **vhdlTestBenchGen[]**. All those steps permit VHDL files and their test bench to be generated.

3.1.2 VhdlCell

Let us focus on the transformation AlpHard to VHDL made by **a2v[]**. The AST which is set to the variable **$result** is written in a particular manner defined by the semantic of Alpha. For instance, the modifications apply in Table 1 for Real datas has include in the variable **$result** :
equation["totruncfrac", call["Truncate_LSB", var["totruncint"]]], equation["totruncint", call["Truncate_MSB", var["AB"]]]
To interpret the AST and transform it into VHDL, the file **vhdlCell.meta** is used to parse the AST and apply the function corresponding to the pattern encountered. Such function is written in the file **vhdlCell.sem**. To continue our example, let see interesting part of **vhdlCell.meta**:

```
  EQUATION ::=
    { equation[_,_call] -> CALLSTATEMENT,
      _equation -> ASSIGNMENT,
      _use -> USESTATEMENT
    }
...
  CALLSTATEMENT ::=
    equation[ leftHandSide: _String,
              funcCall: CALLEXP
            ]
:> semFuncCell[ "equation", leftHandSide, funcCall ,trfuncCall,  opts]
...
  CALLEXP ::=
    call[
         funcName: _String,
 funcOps: {SUBEXPRESSION}
         ]
         :> { "expression" }
```

The .sem and .meta files seems fastidious, but they permit faster modifications. A little explanation is unavoidable. Once the pattern **equation[_,_call]** is encountered, **CALLSTATEMENT** apply the function **semFuncCell** with a text string as first parameter : **"equation"**, the argument that came across are **leftHandSide** and **funcCall**. **trfuncCall** represent the transmitted result of argument **funcCall** recognize as a

CALLEXP. **trfuncCall** is in fact a list of one element text string: {**"expression"**}. Consequently, if a pattern like : *equation["aName", call["aProcedureName", var["anotherName"]]]* appear in variable **$result**, the function :
semFuncCell["equation", "aName" , call["aProcedureName", {var["anotherName"]}] , {"expression"}, opts] will be applied.

3.1.3 Procedure or function parsing

All the functions to apply are described in vhdlCell.meta. See below the functions concerned by our matter.

```
semFuncCell[
  "equation", lhs:_, rhs:_call, {"expression"}, opts:___Rule ]:=
  "\n    "<>semFuncCell[ "call", lhs, rhs, opts ];
...
semFuncCell[ "call", lhs_, call[funcName:_String,  opListAst:_], opts:___Rule ]:=
Module[{listArg,stringArg,res},
listArg=Map[semFuncCell["expression", lhs, # , opts] &,opListAst];
stringArg=Map[StringJoin[#,","] &,listArg];
stringArg=Apply[StringJoin,stringArg];
stringArg=StringDrop[stringArg,-1]; (*remove last comma *)
Which[
StringMatchQ[funcName,"Truncate*",IgnoreCase -> True],
 res="\n    "<>funcName<>"("<>stringArg<>", "<>lhs<>")",
StringMatchQ[funcName,"Procedure*",IgnoreCase -> True],
 res="\n    "<>StringReplace[funcName,"Procedure"->"",IgnoreCase -> True]
     <>"("<>stringArg<>", "<>lhs<>")",
True,
 res="\n    "<>lhs<>" <= "<>funcName<>"("<>stringArg<>")"
] ;
  res
];
...
semFuncCell[
 "expression", lhs:_, var[ id:_ ]|affine[var[id:_],_], opts:___Rule ]:= id;
```

Three semFuncCell functions a stated above. The First is only removing the argument {**"expression"**} to apply semFuncCell with **"call"** as first argument instead of **"equation"**. The second semFuncCell function apply the third one to all the element of the second argument of call[#1,#2]. In our example *equation["aName", call["aProcedureName", var["anotherName"]]]*. There is only one element : "anotherName", so it would be returned and be equal to **listArg**. Then, still in this second semFuncCell function described above, pattern matching is done. If the first element of the call is like **"Truncate..."**, it would be interpreted as a Procedure in VHDL. Moreover, to permit evolution in the package **Definition.vhd**, a pattern matching with the keyword "Procedure" at the beginning of a funcName is done. Every procedure called "ProcedureSomething" would have its prefix Procedure removed for instance: g=ProcedureReduce_MSB(f) will be transformed in Reduce_MSB(f, g). If funcName is neither "Truncate..." nor "Procedure...", the call will be instantiate as a VHDL function. Please keep in mind those points:

- Truncate_XSB might also be instantiate with **insertFunction["","ProcedureTruncate_XSB"]**
- A VHDL function must not be name "Procedure..." or it would be interpreted as a procedure.
- out=vhdlFunction(in1,in2,in3...) will be transformed in : $out <= vhdlFunction(in1, in2, in3...)$
- out=ProcedureSomething(in1,in2,in3) will be transformed in : $ProcedureSomething(in1, in2, in3, out)$

- **In Mathematica when a modification is made to either vhdlCell .sem or .meta, execute the lines below to update modifications**

  ```
  <<Alpha/vhdlCell.sem;
  OLDPWD=Directory[]
  setMMADir[{"lib","Packages","Alpha"}];
  meta["vhdlCell"];
  SetDirectory[OLDPWD]
  <<Alpha/vhdlCell.m;
  ```

  The meta function transform vhdlCell.sem and vhdlCell.meta into vhdlCell.m which needs to be reloaded.

3.1.4 Vhdl2

Has refered in section 2.1.2. The size of outputted bitwidth is defined by the lines :
bitWidthOfExpr[sys_,binop[add , arg1_,arg2_],options___Rule]:= Max[bitWidthOfExpr[sys,arg1], bitWidthOfExpr[sys,arg2]] + 1;
bitWidthOfExpr[sys_,binop[mul , arg1_,arg2_],options___Rule]:= bitWidthOfExpr[sys,arg1] + bitWidthOfExpr[sys,arg2];

3.2 Add new VHDL functions or procedure into the Definition Package

The file **Definition.vhd** is written from Vhdl2.m Vhdl2.m calls function**ToVhdlPackage[]** present in **VhdlLibGen.m**. The function function**ToVhdlPackage[]** analyse the variable VHDLpk and writes the correspondent instantiation and body instantiation. The variable **types** works together with the keyword **SameType** and permit to iterate the definition of a function for multiple types : SIGNED, STD_LOGIC_VECTOR ...

To add a new procedure or function to the package, modify the Variable VHDLpk in $MMALPHA/ lib/ Packages/ Alpha/ VhdlLibGen.m. Type **?ToVhdlPackage** in Mathematica to see the different manner to add an element.

4 Conclusion

This documentation shows how it is possible to use easily Integer, Real or fractional datas coded in two's Complement. The demos Notebooks , matvect_truncate_MSB.nb, matvect_truncate_LSB.nb, matvect_truncate_Real.nb, matvect_reduce_MSB.nb demonstrate it. Results are correct if the known limits of Fixed Point encoding are respected.

This document also shows to developer an example on how the meta function works. Indeed, parts of MMAlpha is develop in a Multi-Modelling manner. This is the case for the generation of VHDL cells and so for variable bitwidths' manipulation.

References

[1] Olivier SENTIEYS, *Implantation d'algorithmes de traitement du signal sur les architectures virgule fixe*, ARCHI05.