Models for Bit-true Simulation and High-level Synthesis of DSP Applications °.

Marc Pauwels, Dirk Lanneer, Francky Catthoor*, Gert Goossens, Hugo De Man*
IMEC v.z.w.
VSDM division
Kapeldreef 75, B-3001 Heverlee, Belgium

Abstract

Real-time DSP applications require a bit-true synthesis system *to generate* correct and efficient ASICs. *This requires* concise simulation and synthesis models, *which are presented in this paper and exemplified for a* non-restoring division *operation. Such models are used in the synthesis library of our bit-true* CATHEDRAL-2ND *compiler, by which industrial size applications have been synthesised.*

1 Introduction.

Besides their *timing constraints*, real-time signal processing (RSP) applications are also characterised by an algorithmic level specification of their *behaviour*, including *finite word length characteristics* that influence the *accuracy* of the algorithm. The accuracy is determined by the overflow and quantisation characteristics of the separate operations (e.g. 2's complement addition with wrap-around overflow behaviour or saturation) and the signal types (i.e. word length and number representation) used for the signals. In many real-life industrial applications [Pau90, Lan91b], signals may have quite different signal types, when effort is spent in their optimisation; e.g. 2's complement fixed point signals of $< 24, 23 >$ (word length=24, 23 bits after binary point), $< 24, 22 >$, $< 48, 45 >$, $< 48, 44 >$ and $< 48, 43 >$, and unsigned 15-bit integers occur in [Pau90].

Currently, much research is performed to automate the high-level synthesis process, which is defined as the transformation of algorithmic level specification of the *behaviour* into a register-transfer level *structure* [Far90]. The goal is to shorten the ASIC design time and obtain correct and still very efficient realisations, compared to manual designs. For RSP applications, it is crucial that the high-level synthesis is *"bit-true"*, which means *all synthesis tasks must retain the specified bit-level behaviour.* Of course signal types have an important impact on synthesis tasks (e.g. required dimensions of allocated data paths) [Pau89]; therefore a careful analysis and optimisation of the signal types is very important before starting the bit-true synthesis, such that an efficient ASIC is obtained (Fig.1).

° This research has been sponsored by the SPRITE project of the EC.

* Professor at the Katholieke Univ. Leuven, Belgium

The first requirement to build a *bit-true synthesis system*, is that the *(bit-level) semantics* of all operations that can occur in the behavioural specification language are clearly defined. Non trivial examples which frequently occur in industrial applications are trigonometric functions, division, square root, modulo arithmetic [Pau90, Lan91b]. Secondly, it is crucial to have *concise, explicit and consistent models* to simulate and synthesise these operations in accordance with the proposed semantics. These models must be centralised in a library, which is easy to maintain, extend and keep consistent. There they can be consulted by the designer, simulator and synthesis tools, and - possibly in the near future- also (semi-)automatic formal verification tools.

Most practical synthesis systems reported so far, either do not consider different signal types [Jes88, Kra89, Pau86], or not to the full extent where also type information is used to optimise the synthesis result [DMi88, Chu89, DMa88, Har89]; e.g. if a multiplication is mapped with a Booth algorithm on an ALU, one can limit the number of iterations, depending on the types specified for the operands and product [Pau89]. Furthermore, these systems do not have concise and full explicit models, such that their bit-trueness can not be guaranteed nor verified.

In [Lan91a] we proposed a CAD framework which supports the full high-level synthesis trajectory for RSP applications, that can be used to build a practical *bit-true synthesis environment* (Fig.1). This was proven by the bit-true CATHEDRAL-2ND compiler [Lan91b] that was developed on top of the framework, and of which a prototype was already used on different industrial size RSP applications [Pau90, Lan90]. The framework itself consists of three data kernels: a *behavioural* (DSFG), a *structural* (ANL) and a *library kernel* (LIB). The latter contains the models and detailed characterisation of all implementation alternatives of behavioural operations and hardware operators that are available in the synthesis system.

In this paper we want to focus on the *library kernel* and more specially on the bit-true aspects. But first, in section 2, will be explained how the simulator and synthesis tools were combined into a *"bit-true simulation and synthesis environment"*. Then, the *general concepts of the* LIB *kernel* will be introduced (section 3) and exemplified by means of a *non-restoring*

0-8186-2610-0/92 $3.00 © 1992 IEEE

division operation (section 4). This is an interesting example, because it is not obvious what its bit-level behaviour should be; so, this will be clarified before discussing the *bit-true simulation and synthesis models*. Finally, in the last section, the general *results and conclusions* will be summarised.

2 The bit-true simulation and synthesis environment.

Fig.1 shows the Bit-true Simulation and Synthesis Environment that was build around our CATHEDRAL-2ND compiler. This compiler is targeted towards highly complex RSP applications with sample rates from a few kHz up to 1MHz, containing scalar, vector, matrix and decision making operations. It therefore makes use of highly multiplexed micro-coded multiprocessor architectures [Lan91b]. However, the presented ideas are not restricted to CATHEDRAL-2ND but can be applied in any compiler.

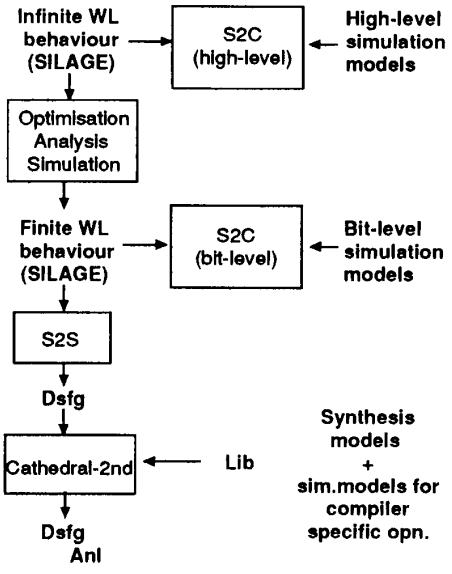

Figure 1: *The bit-true simulation and synthesis environment.*

The SILAGE language is used as behavioural input specification language [Hil90]. The behaviour of an RSP algorithm is described by means of `operations`, acting on `signals`, with `finite signal types`. The `operations` are either *primitive* SILAGE operations (e.g. `+`, `*`, `cast`, `div`) or (user-defined) SILAGE *functions*, defined in terms of *primitives* (e.g. `myBiquad`). The *primitive* operations must have a clearly defined bit-level behaviour, explicitly known by the System designer, who provides the input specification, and the Simulator and Compiler designers, such that all rely on the same semantics. This also implies that a default *type-propagation* is defined for the operations. For instance (see also Fig.2):

- the + operation: adds two fixed point 2's complement (FIX TC) signals with wrap-around overflow characteristic; the types of the operands and sum must be identical (FIX TC $< N, M >$).
- the * operation: multiplies two FIX TC signals with types $< N_1, M_1 >$ and $< N_2, M_2 >$; the product type is FIX TC $< N_1 + N_2, M_1 + M_2 >$.
- the `cast` operation: allows to modify the word length and binary point position of signal types. This is basically selecting bits out of a bitstring, possibly combined with adding sign bits at the most significant side and zero bits at the least significant side.

Notice that until now no clear definition of the `div` operation exists; this is a shortcoming of the current SILAGE definition. We made the assumption that `div` corresponds to an integer, sign-magnitude division. No default type exists for the quotient and therefore `div` must always be succeeded by a `cast`.

The SILAGE description can be simulated by the SILAGE-TO-C simulator S2C [Nac91] either neglecting the signal types and assuming quasi-infinite word lengths ("high-level") or considering the finite signal types ("bit-level"). Fig.2 shows simulation results for a simple example.

Once the signal types are optimised, the bit-true synthesis can start. S2S (SILAGE to DSFG, the *Decorated signal flow graph*) expands the SILAGE functions and translates the *primitive* SILAGE operations in DSFG operations, to be stored in the internal DSFG kernel of the Synthesis Framework. The DSFG kernel initially contains only the behaviour of the algorithm; in later stages of the synthesis process, synthesis related data is added to this kernel (e.g control flow, hardware sources for operations, ...). The LIB contains all possible implementations of the DSFG operations that are generated by S2S; these implementations must maintain the semantics of the corresponding *primitive* SILAGE operations.

Whereas SILAGE and S2C are *compiler independent*, S2S is not, because it translates only the *subset* of SILAGE primitives that are supported by the compiler (Fig.3). On the other hand, the compiler can provide the designer with some *"compiler-specific" operations* (e.g. add with saturation, non-restoring division,..). These are defined as operations with a very efficient, compiler specific implementation (given by the *synthesis model*), that can not be described straightforwardly as a function of SILAGE primitives; e.g. the expand function of the non restoring division (section 4.3) can not be translated in a one-to-one way in SILAGE primitives. But in order to make use of these functions, the compiler designer should also provide a *simulation model* (Fig.1); this can be a C-LIKE or sometimes even a SILAGE function, the latter being used only for simulation and not for synthesis purposes.

3 The synthesis library.

Basically, our synthesis library is *implementation-oriented* in the sense that each entry is a 3-tuple containing: the *operation*, the required *hardware operator*,

```
#define  WORD  fix<4,3>
/* 4 bit fixed point, two's complement word,
   3 bits after binary point */
FUNC main(a,b:WORD) sum,prd:WORD =
  BEGIN
  sum  = a + b;
  // default type of this a+b is WORD
  prd  = WORD( a * b );
  // default type of this a*b is <8,6>,
  //   which is cast into the type WORD
  END;

With a = 0.5   = "0.100"
     b = 0.625 = "0.101"
High-Level simulation results:
     sum = 1.125  = "01.001"
     prd = 0.3125 = "00.010100"
Bit-Level simulation results:
     sum = "1.001" = -0.875
     prd = "0.010" =  0.25
```

Figure 2: *Example of SILAGE function and simulation results.*

```
Silage to Dsfg translation for Cathedral-2nd:
  ---Silage Primitives---
  +      add
  *      mult
  cast   cast
  div    NOT-SUPPORTED
  ---Compiler Specific Functions---
  ganrd  ganrd  // a non-restoring div

Dsfg description of function "main":
  // dsfg format: output operation ops=(operands)
  a   inp             typ=(fix tc 4 3) scp=(main);
  b   inp             typ=(fix tc 4 3) scp=(main);
  sum add  ops=(a b) typ=(fix tc 4 3) scp=(main);
  tmp mult ops=(a b) typ=(fix tc 8 7) scp=(main);
  prd cast ops=(tmp) typ=(fix tc 4 3) scp=(main);
```

Figure 3: *Translation table for S2S (top) and DSFG description of the SILAGE function of Fig.2(bottom).*

and a *list of properties* to characterise the implementation of the operation on that operator. The minimal dimension of the operator is an example of such a property.

Our LIB [Lan91a] is quite different from other systems like [Pau86, Har89, Wol86, Sto88, Mar90], because it is *implementation-oriented*, describes all aspects related to the *bit-trueness of the implementations* with special-purpose *properties*, and contains "primitive" (e.g. *"add"* on adder) as well as "high-level" (expandable) operations (e.g. *"div"*, expanded as *"add","subtract","shift"...*) [Lan91a]. Moreover, combined operations like "add-shift", which can be executed in one clock cycle on chained data paths, don't have to be described separately, because they can be composed automatically by a tool out of the primitive operations "add" and "shift" [Lan91b]. In this way the size of the LIB kernel is reduced.

Operations and operators are classified in an *inheritance tree* (Fig.4) such that *common properties* can be shared. In this way redundancy is avoided and the LIB also becomes maintainable and easily extendable. Three layers can be distinguished in the operation tree (and similarly in the operator tree). In the first or top level, operations are grouped together in *abstract classes* (e.g. mathematical operations). The second level contains abstract operations with a specific behaviour; in fact these are the DSFG operations in which S2S will translate a SILAGE description. Finally, the third or bottom level enumerates all physical implementation alternatives for the abstract operations.

As explained in [Lan91b], a compiler can be considered as a script in which a number of tools are invoked in a specific order; each tool performs a generic synthesis task and modifies the data in the DSFG and ANL kernels (Fig.1). E.g. it is the *expansion tool* that gradually transforms (by a combined refinement-expansion process) each abstract operation of the DSFG description in the most appropriate implementation. In order to find the best implementation, it must make use of some *properties* stored in the LIB (e.g. minimal dimension of an operator and the corresponding area estimate). Because the LIB contains all design knowledge that must be used by the tools, *it is crucial that these properties are very concise* in order to obtain correct and efficient synthesis results.

As will be exemplified for the *division operation* in the next section, the following properties are important for bit-true modelling and should therefore be coded with each implementation tuple in the LIB kernel:

- requirements on the signal types of operands and result (e.g. a *two's complement* multiplier will not give correct result if the operands are *unsigned*).
- the minimally required dimensions of the hardware operators, depending on the types of the operands and result. It should also be proven that the operation behaves identically on over-dimensioned operators; this is mainly useful for highly multiplexed architectures, where the worst case dimension can be imposed by another operation that is also assigned to this operator.
- requirement on how the signals must be aligned on over-dimensioned hardware (when signal word lengths are smaller than the hardware dimensions).
- the number of clock cycles needed to compute the result of the operation, when it is a high-level operation that must be expanded in primitive operations.
- the contents of the expand function, since it defines the behaviour of an expandable operation in terms of primitive operations.

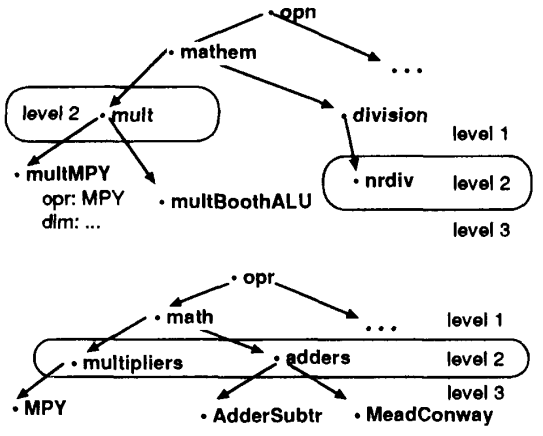

Figure 4: *Operation and operator trees of the LIB.*

4 Modelling a division.

In this section, as an example the bit-true modelling of a *non-restoring division* operation will be discussed. The division is an interesting example, because it is not obvious what its bit-level behaviour should be. Many different algorithms exist to implement a division in hardware, each with a slightly different bit-level behaviour (e.g. the remainder may be always positive, negative or have the sign of the dividend) [Was82]. Let us first consider the division of *2 integers* X (*dividend*) and Y (*divisor*), yielding an *integer quotient* Q and a *remainder* R, such that $X/Y = Q + R/Y$; later we will extend this to arbitrary *fixed point numbers.*

4.1 Integer division.

The *non-restoring division* (NRD) algorithm is a well-known algorithm to implement a division [Was82]. It calculates the quotient bits in an iterative way and consists of the following steps:

• **Step 1:** Calculate an *encoded signed digit representation* of $Q = q_{n-1}q_{n-2}\cdots q_1q_0$ in which $q_i = 0$ corresponds to the value -2^i and $q_i = 1$ to the value $+2^i$. This is done as follows:

$q_{n-1} = sign(Y) \oplus \overline{sign(X)}$; $R_n = X$;
/* $sign(V) = 0$ for $V \geq 0$ and 1 for $V < 0$ */

For i=(n-1)...1::

$$R_i = (q_i = 0)?R_{i+1} + 2^i \times Y : R_{i+1} - 2^i \times Y;$$
$$q_{i-1} = sign(Y) \oplus \overline{sign(R_i)};$$

$$R_0 = (q_0 = 0)?R_1 + Y : R_1 - Y;$$

• **Step 2:** Convert the *encoded signed digit representation* into a *two's complement representation* being: $Q = \overline{q_{n-1}}q_{n-2}\cdots q_1q_0 1$.

• **Step 3:** Correct Q and R_0 by a *post-processing* such that the final remainder has always the same sign as the dividend, and is strict smaller than Y. The correction on Q consists of a *conditional increment.* This NRD algorithm behaves like an *Integer Truncated Sign-Magnitude division* as exemplified in Table 1.

NRD	7	-7	8	-8
2	3	-3	4	-4
-2	-3	3	-4	4
ANRD	7	-7	8	-8
2	3	-4	4	-4
-2	-4	3	-5	3

Table 1: *Behaviour of NRD and ANRD division algorithms.*

ANRD	$X \geq 0$	$X < 0$
$Y > 0$	(X *nrd* Y)	X non multiple of Y: (X *nrd* Y)-1
		X multiple of Y: (X *nrd* Y)
$Y < 0$	(X *nrd* Y)-1	X non multiple of Y: (X *nrd* Y)
		X multiple of Y: (X *nrd* Y)-1

Table 2: *Model of the behaviour of the integer ANRD algorithm.*

The adapted algorithm (ANRD) that we implemented differs from the normal NRD in the sense that the conversion *from signed digit in two's complement representation* is incorporated in the algorithm and that *no post-processing step* is performed. This reduces the implementation cost (area and number of iterations). We could afford to leave out the last step because -for most applications- we are not interested in the value of the remainder and an error on the quotient of at most 1 least significant bit compared to the NRD, is acceptable. The ANRD algorithm behaves like shown in Table1 and 2, and calculates a quotient $Q_{anrd} = q'_n q_{n-1}\cdots q_1q_0$ as follows:

• $q_n = sign(Y) \oplus \overline{sign(X)}$; $q'_n = \overline{q_n}$; $R_n = X$;

• For i=(n-1)...0::

$$R_i = (q_{i+1} = 0)?R_{i+1} + 2^i \cdot Y : R_{i+1} - 2^i \cdot Y;$$
$$q_i = sign(Y) \oplus \overline{sign(R_i)};$$

The fact that the ANRD algorithm behaves like a division was formally proven in [Ver92].

4.2 Non-Integer division.

The integer ANRD model can now be used to construct a model for a *generalized, fixed point, adapted NRD*:

$$Q_{ganrd} = XganrdY,$$

with

$$< Nx, Dx >, < Ny, Dy > and < Nq, Dq >$$

the fixed point, two's complement types of the operands and required quotient. The new model is as follows:

- Transform the problem to the integer division

$$IQ = TanrdN;$$

$$T = SX \times 2^{max(0, Dq-(Dx-Dy))}; SX = X \times 2^{Dx};$$

$$N = SY \times 2^{max(0, -Dq+(Dx-Dy))}; SY = Y \times 2^{Dy};$$

- Apply the integer ANRD model.
- Retransform the result to the required non-integer quotient without loosing any bits:

$$RQ = real(IQ) \times 2^{-Dq};$$

$$Q =< Nq, Dq > (RQ)$$

Consider the following example:

```
X= <8,4>(0.5)    = "0000.1000";
Y= <4,3>(-0.25) = "1.110";
Q= <6,2>(?);

SX=  0.5*16 =  8; T = SX*2 = 16;
SY= -0.25*8 = -2; N = SY*1 = -2;

IQ= (16 anrd -2) = (16 nrd -2)-1 = -9;
RQ= real(-9)/4   = -2.25;
Q = <6,2>(-2.25) = "1101.11";
```

4.3 Implementation of the non-integer division.

The kernel of the ANRD consists of a *conditional add-subtract operation* of the remainder and the divisor, shifted over i bits. This can be mapped on an `adder-subtractor` and a `1-bit up-shifter`, which performs the shifting in an iterative way. In principle the `adder-subtractor` should be $Nx + Ny - Nov + 1$ bits wide, with Nov being the number of overlapping bits of the shifted remainder and divisor; the extra bit is needed to avoid overflow. For instance, for $Nx = 8, Ny = 4, Nov = 2$:

```
   rrrrrrrr   (remainder)
 yyyy         (divisor)
sssssrrrrrr   (sum)
```

However, the dimension can immediately be reduced to $Ny + 1$ bits because the $Nx - Nov$ least significant bits (LSB) of the sum must not be computed. Moreover, it can be proven that *no overflow* occurs during this addition, such that a Ny-bit `adder-subtractor` is sufficient; to prove this, we did an extensive analysis and verification (manual and semi-automatic) of the implementation [Ver92].

The most significant part of the remainder is stored in a Ny-bit wide register ("Z-reg"), while the LSB part is stored in a Nx-bit parallel-serial register ("PS-reg"). Each iteration the MSB and LSB parts of the remainder are shifted 1 bit up, and one quotient bit is shifted as LSB in a Nq-bit serial-parallel register ("SP-reg") (Fig.5).

In order to get the correct quotient and guarantee the convergence of the iterative process, the initial value of the remainder ($R_n = X$) should be well-aligned in the Z and PS registers. Again it can be proven [Ver92], that only with $Nov = 1$ or $Nov = 2$ no divergence occurs; under these circumstances $R_{n-1} < Y$, and once a R_i was smaller than Y, all succeeding values will be smaller than Y. In our implementation, we choose for the 2-bit overlap ($Nov = 2$), as can be deducted from the initialisation operations in the DSFG expansion of the GANRD operation:

```
--- symbol table ---
$1   first operand X
$2   second operand Y
$T1  msb of X (R) in Z-reg
$T2  lsb of X (R) in PS-reg
$T3  Q built up in SP-reg
$T4  sgnY nexor sgnR
$S   result (quotient)

--- pseudo dsfg code---
s1   sign       ops=($1)
ps1  writePS    ops=($1)

s2   sign       ops=($2) // s2=0:$2=pos
ps2  shiftUpPS  ops=(ps1)// over 1 bit
$T4  nexor      ops=(s1 s2)
xor  exor       ops=(s1 s2)

sp0  initSP     ops=(xor)
// SP=0 (-1) if signs=equal (not equal)
tmp  pass       ops=(zero)  (if s1=0)
tmp  dec        ops=(zero)  (if s1=1)
// tmp == signextension of $1 (0 or -1)
$T1  shiftUp1   ops=(tmp bit)//over 1 bit
bit  shUpPS_bit ops=(ps2)// over 1 bit
// shift MSB "bit" out
$T2  shiftUpPS  ops=(ps2)// over 1 bit

for( i=0; i<IT; i++){
  tmpi add        ops=($T1 $2) (if $T4=0)
  tmpi sub        ops=($T1 $2) (if $T4=1)
  $T1  shiftUp1   ops=(tmpi biti)
  biti shUpPS_bit ops=($T2)
  $T2  shiftUpPS  ops=($T2)
  stmp sign       ops=(tmpi)
  qbit nexor      ops=(stmp s2)
  $T3  shiftUpSP  ops=($T3 qbit)
  // over 1 bit, shift qbit in as LSB
  $T4  nexor      ops=(stmp s2)
```

```
}
q    readSP      ops=($T3)
$S   cast        ops=(q) //to output type
```

Now still the number of iterations must be calculated such that the exact quotient bitstring is obtained. An extensive analysis lead to the following formula for the type of the quotient that is obtained after IT iterations:

$$< IT + (Nov - 1), Dx - Dy + IT - Nx + (Nov - 1) >$$

The more iterations are performed, the higher the accuracy of the quotient will be. If one wants to calculate a quotient of type $< Nq, Dq >$, then

$$IT = max(0, Dq - Dx + Dy + Nx - (Nov - 1))$$

iterations must be performed. This can be checked on the example we used in section 4.2, which requires 8 iterations:

```
Symbolic representation of iteration:
      Z        PS; SP
  opn Y
    TMP <      PS; SP < qbit newopn
   newZ     newPS; newSP

Values after Initialisation:
   0000   001000oo;  111111       +
Iteration 1:
   0000   001000oo;  111111
 + 1110
   1110 < 001000oo;  111111 < 1   -
Iteration 2:
   1100   01000ooo;  111111
 - 1110
   1110 < 01000ooo;  111111 < 1   -
Iteration 3:
   1100   1000oooo;  111111
 - 1110
   1110 < 1000oooo;  111111 < 1   -
Iteration 4:
   1101   000ooooo;  111111
 - 1110
   1111 < 000ooooo;  111111 < 1   -
Iteration 5:
   1110   00oooooo;  111111
 - 1110
   0000 < 00oooooo;  111111 < 0   +
Iteration 6:
   0000   0ooooooo;  111110
 + 1110
   1110 < 0ooooooo;  111110 < 1   -
Iteration 7:
   1100   oooooooo;  111101
 - 1110
   1110 < oooooooo;  111101 < 1   -
Iteration 8:
   110o   oooooooo;  111011
 - 1110
   1110 < oooooooo;  111011 < 1   -
ReadSP:          Q = 110111  Correct!
```

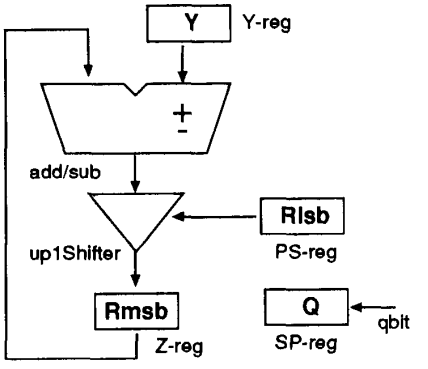

Figure 5: *Operators required for the GANRD division.*

An important aspect that still needs verification is whether the implementation behaves identical when the operators are larger than the minimal dimensions. This frequently occurs in algorithms that are mapped on highly multiplexed architectures, which is the target domain of our CATHEDRAL-2ND compiler. Restrictions on the parameters of module generators might be another reason: e.g. an ALU which contains all operators required for the GANRD typically has only 1 bit width parameter. For our example, this bit width would be at least 8 (the width of the PS-register). Due to the fact that the remainder is stored as 2 sub-words (a LSB and MSB part) in different registers, and that it must be up-shifted over 1 bit each iteration, the LSB part (stored in the PS-reg) should be MSB aligned and extended with zero-bits at the LSB side; similarly, the MSB part (stored in Z-reg) must be LSB aligned and sign-extended at the MSB side. Consequently, also the sum signals (*tmp* and *tmpi* in the expansion code) will be LSB aligned on the data path. Fortunately, because no overflow can occur during the addition, the sum is also sign-extended at its MSB side such that a correct sign bit (i.e. the MSB) will be used to calculate the quotient bits. Because the quotient bits are shifted in the SP-register as LSB, the quotient will be LSB aligned in an over-dimensioned SP-register.

4.4 Summary.

In summary, one can state that the following *bit-true related properties* need to be modelled to guarantee a correct synthesis:

- requirements on input and output signal types: must be *fixed point, two's complement.*
- minimal set of required operators and their minimal dimension: an adder/subtractor(Ny), parallel-serial register (Nx), serial-parallel register(Nq), shifter(Ny) and registers(Ny).

• the required signal alignments to get an identical behaviour when the algorithm is performed on over-dimensioned hardware.
• formula for the number of iterations that must be performed to obtain the quotient with the specified accuracy. This is a parameter in the expand function.
• the expand function, describing the exact behaviour and implementation of the GANRD division in terms of DSFG primitives.

5 Conclusions.

In this paper we presented the models that are required to build a *bit-true simulation and synthesis environment.* The bit-true aspects of the models were exemplified for a *non-restoring division operation.* Similar models have been constructed for the most common RSP operations (arithmetic, memory, boolean, comparative ...) for which several bit-true implementation alternatives on different hardware have been coded in the our synthesis library (including both single and multiple-precision implementations). Currently the library consists of about 400 operations and 5,500 lines of code, coded in a special-purpose LIB language. This library has already been successfully used by the CATHEDRAL-2ND compiler for the bit-true synthesis of industrial size RSP applications [Pau90, Lan90].

References

[Far90] McFarland, Parker, Camposano, "The High-Level Synthesis of Digital Systems", Proc. of the IEEE, Vol.78, no.2, pp301-318, Feb.1990.

[Jes88] J.Jess, R.v.d.Born, L.Stok, "Synthesis of concurrent hardware structures", Proc. IEEE Int. Symp. on Circuits and Systems, Helsinki, Finland, pp.2757-2760, June 1988.

[Kra89] H.Kramer, W.Rosenstiel, "Synthesis of multiprocessor architectures from behavioral descriptions" 4th Int. Workshop on High-level Synthesis, Kennebunkport ME, Oct. 1989.

[Pau86] Paulin, Knight, Girczyc, "HAL: a multi-paradigm approach to automatic data path synthesis", Proc. 23rd ACM/IEEE Design Automation Conf., Las Vegas, Nevada, pp.263-270, June 1986.

[DMi88] G. De Micheli, D.C. Ku, "HERCULES - A System for High-Level Synthesis", Proc. 25th ACM/IEEE Design Automation Conf., San Francisco CA, pp.483-488, June 1988.

[Chu89] C-M.Chu, M.Potkonjak, M.Thaler, J.Rabaey, "HYPER: an interactive synthesis environment for high performance real-time applications", Proc. IEEE Int. Conf. Computer Design, Rochester NY, pp.432-435, Oct. 1989.

[DMa88] H.De Man et al., "Cathedral-II - a computer-aided synthesis system fir digital signal processing VLSI systems", IEE Computer-Aided Eng. Journal, pp.55-66, April 1988.

[Har89] B.S.Haroun, M.I.Elmasry, "SPAID: An Architectural Synthesis Tool for DSP Custom Applications", IEEE Journal of Solid-State Circ., Vol.24, No.2, pp.426-435, April 1989.

[Wol86] W.Wolf, "An object-oriented, procedural database for VLSI chip planning", 23th ACM/IEEE Design Automation Conf., pp744-751, 1986.

[Sto88] L.Stok, R van den Born, "EASY: Multiprocessor Architecture Optimisation", Proc.Int. Workshop on Logic and Architectural Synthesis, Grenoble, May 1988.

[Mar90] Marwedel, "Matching System and Component Behaviour in MIMOLA Synthesis Tools", Proc. EDAC90, pp.146-156, Glasgow, Scotland, March 1990.

[Pau89] M.Pauwels, F.Catthoor, D.Lanneer, H.De Man, "Type-handling in bit-true silicon compilation for DSP", Proc. Europ. Conf. on Circ. Theory and Design, ECCTD, Brighton, U.K., pp. 166-170, Sep. 1989.

[Pau90] M.Pauwels, F.Catthoor, K.Schoofs, M.Masschelein, H.De Man, "A Dynamic Range Compressor Architecture for Audio, used as a test-vehicle for type-handling in the CATHEDRAL-2nd Synthesis Environment.", Proc. 5th Eur. Signal Proc. Conf., EUSIPCO-90, Barcelona, Spain, September 1990.

[Lan90] D.Lanneer, F.Catthoor, G.Goossens, M.Pauwels, J.Van Meerbergen, H.De Man, "Open-ended system for high-level synthesis of flexible signal processors" Proc. Eur. Design Autom. Conf., Glasgow, Scotland, pp.272-276, March 1990.

[Lan91a] D. Lanneer, G. Goossens, F. Catthoor, M. Pauwels, J. Van Meerbergen, H. De Man, "An object-oriented framework supporting the full high-level synthesis trajectory", 10th Int. Symp. Comp. Hardw. Descr. Lang. (CHDL-91), Marseille, France, April 1991.

[Lan91b] Lanneer et al., "Architectural synthesis for medium and high throughput signal processing with the new Cathedral environment", published in "Trends in High-Level Synthesis", edited by R.Camposano, W.Wolf, Kluwer, 1991.

[Hil90] P.N.Hilfinger, J.Rabaey, D.Genin, C.Scheers, H.De Man, "DSP specification using the Silage language", Proc. Int. Conf. on Acoustics, Speech and Signal Processing, Albuquerque, NM, April 1990.

[Nac91] Lode Nachtergaele, Ivo Bolsens, Hugo De Man, "A Specification and simulation front-end for hardware synthesis of digital signal processing applications", Int.Journal of Computer Simulation, Special Issue on "Multi processor simulation", 1991.

[Was82] Shlomo Waser, Michael J. Flynn, "Introduction to arithmetic for digital system designers", *CBS College Publishing*, ISBN 0-03-060571-7, 1982

[Ver92] D.Verkest, L.Claesen, H.De Man, "A Proof of the Non Restoring Division Algorithm and its Implementation on the Cathedral-II ALU", Proc. of "Workshop on designing correct circuits", Lyngby, 6-8 January 92.