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).

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 [DMa88, Chua89, 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 [Pau90]. Furthermore, these systems do not have concise and full explicit models, such that their bit-true nature 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
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 multi-processor architectures [Lan91b]. However, the presented ideas are not restricted to Cathedral-2nd but can be applied in any compiler.

The SILAGE language is used as behavioural input specification language [Hi190]. 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 < N1,M1 >);
- the * operation: multiplies two FIX TC signals with types < N1,M1 > and < N2,M2 >; the product type is FIX TC < N1 + N2, M1 + M2 >;
- 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 [Na91] 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 is 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 simulator 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,
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 [Pan86, 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.
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 \( Q = q_{n-1}q_{n-2}\ldots q_{0} \) 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:
  
  \[ q_{n-1} = \text{sign}(Y) \oplus \text{sign}(X); R_n = X; \]
  
  /* sign(V) = 0 for V \geq 0 and 1 for V < 0 */
  
  For \( i=(n-1)\ldots1: \)
  
  \[ R_i = (q_i = 0) ? R_{i+1} + 2^i \times Y : R_{i+1} - 2^i \times Y; \]
  
  \[ q_{i-1} = \text{sign}(Y) \oplus \text{sign}(R_i); \]
  
  \[ R_0 = (q_0 = 0) ? R_1 + Y : R_1 - Y; \]

- **Step 2:** Convert the encoded signed digit representation into a two's complement representation being: \( Q=q_{n-1}q_{n-2}\ldots q_{0}1 \).

- **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 strictly smaller than Y. The correction on Q consists of a conditional increment. This NRD algorithm consists of a conditional increment.

This NRD algorithm consists of a conditional increment. The correction on Q consists of a conditional increment. This NRD algorithm behaves like an Integer Truncated Signed-Magnitude division as exemplified in Table 1.

![Figure 4: Operation and operator trees of the LIB.](image)

<table>
<thead>
<tr>
<th>NRD</th>
<th>X &gt; 0 ( (X \text{ nrd } Y) )</th>
<th>X &lt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y &gt; 0</td>
<td>( (X \text{ nrd } Y)-1 )</td>
<td>( X \text{ non multiple of } Y: )</td>
</tr>
<tr>
<td>Y &lt; 0</td>
<td>( (X \text{ nrd } Y)-1 )</td>
<td>( X \text{ non multiple of } Y: )</td>
</tr>
</tbody>
</table>

Table 1: Behaviour of NRD and ANRD division algorithms.

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 Table 2, and calculates a quotient \( Q_{\text{anrd}} = q_nq_{n-1}\ldots q_{0} \) as follows:

- \( q_n = \text{sign}(Y) \oplus \text{sign}(X); q_n = \overline{q_n}; R_n = X; \)

- For \( i=(n-1)\ldots0: \)
  
  \[ R_i = (q_{i+1} = 0) ? R_{i+1} + 2^i \times Y : R_{i+1} - 2^i \times Y; \]
  
  \[ q_i = \text{sign}(Y) \oplus \text{sign}(R_i); \]

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:

\[ Q_{\text{ganrd}} = X_{\text{ganrd}} \times Y, \]
with

\[ <Nz, Dz>, <Ny, Dy> \text{ and } <Nq, Dq> \]

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

\[ IQ = TanrdN; \]

- The most significant part of the remainder is stored in a \( Ny \)-bit wide register ('Z-reg'), while the LSB part is stored in an \( Nz \)-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 \( Nv = 1 \) or \( Nv = 2 \) 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 \( Nv = 2 \), as can be deduced from the initialisation operations in the DSG expansion of the GANRD operation:

```
--- symbol table ---
$1$ first operand \( X \)
$2$ second operand \( Y \)
$T_i$ msb of \( X \) in Z-reg
$T_i$ lab of \( X \) in PS-reg
$T_3$ built up in SP-reg
$T_4$ sgnl nexor sgnR

--- pseudo dsfg code---
sl sign ops=($1$)
ps1 writePS ops=($1$)
s2 sign ops=($2$) // s2=0:$2=pos
ps2 shiftUpPS ops=($1$)// over 1 bit
$T_4$ nexor ops=($s1$)
xor xor ops=($s1$)
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 $s1$ (0 or -1)
$T_1$ shiftUp1 ops=(tmp bit)/over 1 bit
$T_2$ shiftUpP1 ops=(ps2)/ over 1 bit
// shift LSB "bit" out
$T_2$ shiftUpP1 ops=(ps2)/ over 1 bit
```

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 \( Nz + Ny - Nov + 1 \) 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 \( Nz = 8, Ny = 4, Nov = 2 \):

```
ffefedf (remainder)
YYYY (divisor)
ssssssss (sum)
```

However, the dimension can immediately be reduced to \( Ny + 1 \) bits because the \( Nz - Nov \) 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].
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:

$$< IT + (N_{ov} - 1), Dz - D_y + IT - N_z + (N_{ov} - 1) >$$

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

$$IT = \max(0, D_q - D_z + D_y + N_z - (N_{ov} - 1))$$

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:**

\[
\begin{align*}
\text{z} & \leftarrow \text{PS; SP} \\
\text{ops} & \leftarrow \text{Y} \\
\text{TMP} & \leftarrow \text{PS; SP} \leftarrow \text{qbit, newop; newZ, newPS; newSP} \\
\text{Values after Initialisation:} & \\
0000 & 00100000; 111111 + \\
\text{Iteration 1:} & \\
0000 & 00100000; 111111 + 1110 \\
1110 & 00100000; 111111 < 1 + \\
\text{Iteration 2:} & \\
1100 & 01000000; 111111 - 1110 \\
1110 & 01000000; 111111 < 1 - \\
\text{Iteration 3:} & \\
1100 & 10000000; 111111 - 1110 \\
1110 & 10000000; 111111 < 1 - \\
\text{Iteration 4:} & \\
1101 & 00000000; 111111 - 1110 \\
1111 & 00000000; 111111 < 1 - \\
\text{Iteration 5:} & \\
1110 & 00000000; 111111 - 1110 \\
0000 & 00000000; 111111 < 0 + \\
\text{Iteration 6:} & \\
0000 & 00000000; 111110 + 1110 \\
1110 & 00000000; 111110 < 1 - \\
\text{Iteration 7:} & \\
1100 & 00000000; 111101 - 1110 \\
1110 & 00000000; 111101 < 1 - \\
\text{Iteration 8:} & \\
1100 & 00000000; 111101
\end{align*}
\]

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 as 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.

**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 as 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-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-dimensional 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 \( D_{n, r} \) division in terms of \( D_{s, r} \) 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 DSP operations (arithmetic, memory, boolean, comparative ...) for which several bit-true implementations 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 DSP applications [Pau90, Lan90].

References


