Code Generation for Compiled Bit-True Simulation of DSP Applications

L. De Coster, M. Adé, R. Lauwereins*, J. Peperstraete
Katholieke Universiteit Leuven, Department ESAT
Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium
luc.decoster@esat.kuleuven.ac.be

Abstract

Bit-true simulation verifies the finite word length choices in the VLSI implementation of a DSP application. Present-day bit-true simulation tools are time consuming. We elaborate a new approach in which the signal flow graph of the application is analyzed and then transformed utilizing the flexibility available on the simulation target. This global approach outperforms current tools by an order of magnitude in simulation time.

1. Introduction

Digital signal processing applications are specified by operations that work on signals. Finite word length effects arise when the DSP application is to be realized in VLSI. The signals are implemented by a finite number of bits and mostly fixed-point arithmetic is employed. In the event of an operator running out of bits to represent the result, a certain way of chopping is to be performed. A quantization mechanism at the least significant bit (LSB) side and an overflow mechanism at most significant bit (MSB) side are associated with each operation. In the realization of the DSP applications, the finite word lengths of the signals and the finite word length mechanisms of the operations result in effects such as quantization noise, limit cycles, etc. Those lengths and mechanisms are carefully determined in order that a trade off between the implementation cost and the performance is made. Verification of the choices should not be skipped as for example the Ariane 5 flight 501 failure demonstrates [9].

Since analytical analysis cannot validate the finite word length effects for a general application, the availability of bit-true validation by simulation is indispensable. However, the word lengths and mechanisms available on the simulation target — a processor — differ from those specified in the application. Since, in bit-true simulation every signal and operation must be realized exactly as specified, restoring operators must be included after the arithmetic operator. Restoring comprises of a sequence of logical tests and manipulations to truncate at both sides of the signal and also to correctly align the signal for the next operation.

Currently there exist a number of bit-true compilation paths, but their performance is poor. This is because the restoring code which follows practically every operator is extensive compared to the single instruction of the operator itself. A survey:
- floating-point C++ to fixed-point C++ [8]: This utility software converts a floating-point DSP program written in C or C++ to a fixed-point program with bit-true behavior as specified. The conversion is conducted by defining a new fixed-point data class and utilizing the operator overloading mechanisms of the C++ language. Simulation time has grown by one or two orders of magnitude, due to the bit-true demand. For example, the fixed-point data class consists of 6 data elements and an operator is overloaded with 20 lines of code.
- DFL to fixed-point C ([10], [11]): The signals of the DFL description are translated in short variables of C. The operations of the DFL description are realized in C by a sequence of instructions. This sequence consists of the operator corresponding to the operation, followed by some logical tests and modifications in order to guarantee the bit-trueness. Again the code size explodes because of the bit-true demand. For example, a single-precision addition takes 14 lines of C code.
- DFL to $\mathrm{ASM}_{DSP56000}$ ([10], [12]): A similar approach is taken. Signals are mapped on storage elements; operations are realized by their corresponding operators and some logical operators. The latter realize the

*L. De Coster is a Research Assistant and R. Lauwereins is a Senior Research Associate of the Belgian National Fund for Scientific Research.

overflow and quantization mechanisms and perform the necessary alignment corrections towards the following operator. For example, for a FIR filter, 38 instruction cycles are needed for each filter tap in the bit-true code, compared to only one — a single MAC instruction — in the functional-true code.
Previous literature corresponds with the implementation of the bit-true simulation tools in SPW [4] and DSP Station [6]. A close look to Matlab [5] and DSP Canvas [3] reveals that a similar approach is taken.

The increase in execution time and code size is between one and two orders of magnitude compared to the non bit-true functional simulation. This prevents the use of bit-true real-time emulation on even the simplest of DSP applications. This also means that bit-true simulation times are excessively high and as a result bit-true verification in general becomes very tedious and is often completely skipped.

However, a detailed look reveals that a lot of the restoring code is unnecessary or can be written more succinctly by taking a global scope on the application. In our global approach, the flexibility available in mapping the signals of the application on the wider storage elements of the simulation target is used for a non-normalized passing of the signals between the operations. On the one hand, the overflow and quantization mechanisms provided by the hardware are favored over software routines since they are free. On the other hand, alignment correction between operations is unfavorable, since this implies the insertion of shift instructions. An optimization problem arises. Schoofs et al. ([10], [13]) partly used the mentioned flexibility in their code generator synthesizing from DFL to bit-true ASM_{ASIP} (Application Specific Integrated Processors). However since the processors are specific for a class of applications, there is not much discrepancy between algorithm and target. This is in contrast with our domain of simulation.

The aim of the new approach is to accelerate the bit-true simulation and even to attain real-time bit-true emulation by producing execution times comparable to those for functional verification. In a later phase, an efficient bit-true simulator can also be the core of a synthesis tool for design-space exploration of the finite word length choices in the VLSI implementation of a DSP application. Work in this field ([7], [14]) is based on the existence of an efficient bit-true simulator.

The available flexibility in mapping the application on the simulation target is revealed in section 2. Next, the mapping is defined in section 3. Section 4 discusses the evolving combinatorial optimization problem. We also propose a solution strategy. In section 5, an example validates the assertions. Section 6 summarizes.

2. Flexibility

In general, the lengths of the signals of the application and the lengths of the storage elements of the simulation target do not correspond. A flexibility arises in mapping the signals on the storage elements. A reasonable mapping for fractional signal types can be as follows: an alignment of the signal to the MSB side of the storage element and an extension for the remaining bits at LSB side with zeros (Figure 1). For this mapping, the hardware provided overflow mechanism is free, since the signal is aligned to it. However, other overflow mechanisms and all quantization mechanisms are to be realized by a software routine. A sequence of logical tests and logical manipulations checks the condition and performs the correct mechanism if needed. If the quantization mechanism matches the hardware provided one and if shifting is cheap (e.g., in the presence of a barrel shifter), there is an alternative. The signal can be shifted completely to the LSB side, next the signal can make use of the hardware provided quantization mechanism, and finally the signal can be shifted back. Note that there are other mapping standards: for integer signal types an alignment to LSB side corresponds better to the target architecture.

So, depending on the mapping of the signals on the storage elements, cheap hardware provision can be used or expensive software routines are needed. In the local-based approach of the present-day bit-true compilers, a mapping standard is chosen and signals are normalized to that standard after each operation. There is no reason to do so if the compiler can keep track of non-normalized mappings (i.e. non-zero offsets of the signals in the storage elements). Unnecessary shiftings (back-and-forths) can be saved and thus more hardware provided mechanisms can become favorable to use.

3 Mapping

This section covers the issue of how the signals and operations of the application can be realized on the simulation target by the available storage elements and the available operators. After the definition of the application model and the target model, we define the mapping.

3.1. Application model

The mapping transformation starts from the signal flow graph of the application. DFL ([10], [2]) is chosen as the input language of our synthesis path. As an

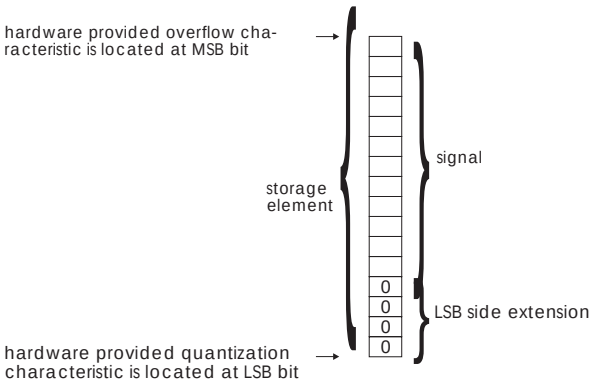

Figure 1. Sketch of a normalized mapping of a signal on a storage element.

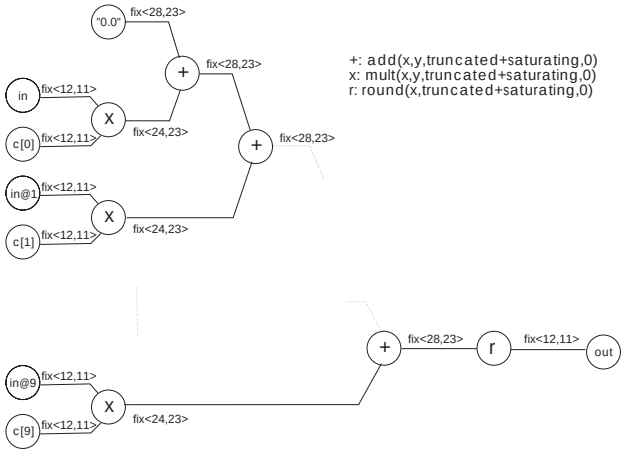

Figure 2. Graphical representation of the 10-tap FIR filter.

example, the textual description of a 10-tap FIR filter and its graphical representation are given (Figure 2).

```
#define sw fix<12,11> /* single prec. */
#define dw fix<24,23> /* double prec. */
#define aw fix<28,23> /* accum. prec. */
oper*(x,y)=mult(x,y,truncated+saturating,0);
oper+(x,y)=add(x,y,truncated+saturating,0);
c[]=sw({ /* e.g. LP filt. coef. */ });
func FIR (in:sw)out:sw =
begin
(i:1..9)::
   in@@i=0.0;
tmp[0]=aw(0.0);
(i:0..9)::
   tmp[i+1]=aw(tmp[i]+dw(c[i]*in@i));
out=sw(round(tmp[10],truncated+saturating,0));
end;
```

The example reveals that a DSP application is in fact a graph of signals interconnecting operations. Note that an operation is characterized by three behaviors: the function itself and two finite word length mechanisms.

Definition 1 *operation = (function, overf. mech., quant. mech., signal$_{in1}$, signal$_{in2}$, signal$_{out}$)*[1][2] *where:*
- *function* $\in$ {*add, mult, round, etc.*}
- *overf. mech.* $\in$ {*saturating, wrapped, etc.*}
- *quant. mech.* $\in$ {*truncated, rounded, etc.*}
- *signal* = <*wl,fwl*>[3]
  - *wl* $\in N_0$, *and fwl* $\in Z$

[1] Monadic operations do not contain a signal$_{in2}$ argument.

[2] Note a syntactical difference between the DFL specification of operations and definition 1. Essentially the same semantics are specified.

[3] Word length and fractional word length.

3.2. Target model

First, only signals with a word length corresponding to an available storage element width are allowed. There are a discrete number of storage element widths available on the target. For example, the DSP56000 processor allows single, double and accumulator precision widths (24, 48 and 56 bits). After the mapping transformation, only those signal lengths will remain. Second, the finite word length mechanisms are extended in behavior. While the overflow and quantization are usually tested and executed on the boundaries of the signal, we also allow such mechanisms in the middle of the signal. For example, an overflow can also be tested and fired if needed on the penultimate bit at MSB side instead of the standard version which works on the ultimate bit. This is because in general the real borders of the signal will differ from the storage element borders. Only when the overflow or quantization mechanism is provided by the hardware and when it is to be executed on the border location, hardware provision can be used. In other cases, a sequence of operators will realize the behavior. Third, an operation begins with alignment corrections for each input. This is to align the results of the previous operations to the operation.

In order to steer the mapping, costs are associated with the operators. We conclude that the target model is a set of operators.

Definition 2 *operator = (function, ext. overf. mech., ext. quant. mech., alignment correction$_{in1}$, alignment correction$_{in2}$, storage element$_{in1}$, storage element$_{in2}$, storage element$_{out}$) where:*

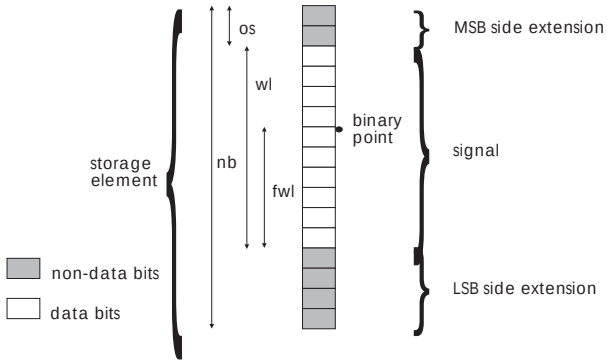

Figure 3. Mapping a signal on a storage element.

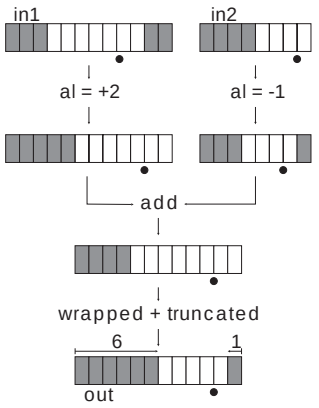

Figure 4. Example of mapping an operation on an operator.

- *function* $\in$ {*add, mult, round, etc.*}
- *ext. overf. mech.* = <*overf. mech., position*>[4]
- *ext. quant. mech.* = <*quant. mech., position*>
  - *overf. mech.* $\in$ {*saturating, wrapped, etc.*}
  - *quant. mech.* $\in$ {*truncated, rounded, etc.*}
  - *position* $\in N$
- *alignment correction* = <*al*>
  - *al* $\in Z$
- *storage element* = <*nb*>[5]
  - *nb* $\in$ {*single, double, accum., etc.*}
    - *single, double, accum.* $\in N_0$

3.3. Mapping

We restrict the flexibility for mapping signals on storage elements by the following two assumptions: (1) The data bits are mapped contiguous on the storage element. (2) The LSB side of the signal is extended with zeros and the MSB side of the signal with sign bits (Figure 3). Then, an offset value indicates the data mapping.

Definition 3 *data mapping = (signal, storage element, offset) where: offset* = <*os*> *and os* $\in N$

Next, an operation with its corresponding signals is mapped on an operator with its corresponding storage elements. The operator has to realize the operation's behavior for the signals mapped in the storage elements.

Definition 4 *mapping = (operation, operator, data mapping*$_{in1}$*, data mapping*$_{in2}$*, data mapping*$_{out}$*)*

[4]The position parameter points to the location where the mechanism is to be performed.

[5]Number of bits.

The following example clarifies the mapping definition (Figure 4):
- operation = (add, wrapped, truncated, <7,2>, <4,1>, <5,1>)
- operator = (add, <wrapped,6>, <truncated,1>, +2, -1, accum., double, accum.)
  - accum. = 12
  - double = 8
- data mapping$_{in1}$ = (<7,2>, accum., 3)
- data mapping$_{in2}$ = (<4,1>, double, 4)
- data mapping$_{out}$ = (<5,1>, accum., 6)

4. Optimization problem

The way the signals are mapped in the storage elements determines the costs for the operations. This cost can be decomposed in four parts:
- function cost: The target supported functions take only a small fixed amount of cycles. For example, an addition on the DSP56000 takes one cycle. We conclude that the function cost is constant or invariant under provided flexibility and will not play any role in the optimization problem.
- overflow mechanism cost and quantization mechanism cost: The cycle cost has a very irregular behavior as a function of the signal mapping. When the signal is aligned to hardware provision to perform the specified mechanism, the mechanism is free. No extra code is needed. In other cases, a software routine is to be included. Two techniques then are used. If the alignment is not far from the hardware provision or in the case a barrel shifter is present, shifting towards hardware provision is possible. Otherwise logical tests and manipulation should be included. We conclude that the overflow and quantization mechanisms cost is non-

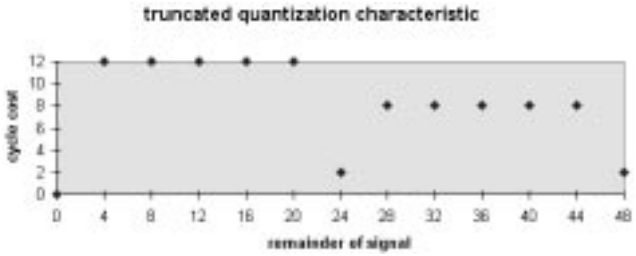

Figure 5. The truncated quantization mechanism cycle cost on the DSP56000.

linearly dependent on the offset values of the signals neighboring the operator.
- alignment correction cost: The alignment correction consist of shifting the result of the function to the specified output offset value. In fact, it is setting out the signal for the next operation. The cost for it is strongly dependent on the shift cycle cost. It is cheap and constant if a barrel shifter is present on the target. It is proportional to the difference in offset values if only one bit shifting can be done per instruction.

Figure 5 gives an example of the very capricious dependencies of the cost functions on the available flexibility. The truncated quantization is cheap for remainder values 0, 24, and 48. Values larger than 24 imply some logical work. Values smaller than 24 are more complecated. Since that segment of the accumulator does not have logic capabilities, also some transfers are needed. More figures and numbers of the cost function for the DSP56000 can be found in [1].

As a consequence, the mapping is a combinatorial optimization problem: the offsets correspond to the variables, the cycle counts to the costs, a legal mapping to a solution, and the best mapping to the optimal solution of the combinatorial optimization problem. However, the size of the problem is huge. For the example (10-tap FIR filter on a DSP56000), there are 42 offset variables while the range of these offset variables is at least 10 (e.g. for a <24,23> signal in a <48> storage element, the range equals 24). This result in a size for the combinatorial optimization problem of at least 10^{40}. The simple enumeration leads to unacceptable computation times for the compilation process. Since cost functions are very non-linear, integer linear programming does not help either.

In the mapping tool we have developed, the enumeration technique still is optioned but it is extended with three additional techniques. First bounds are put in. They preliminary stop the exploration of partial mappings wich have an interim cost that is already higher than the up to that point optimal full solution cost. Second the application graph is not arbitrary but it can be characterized as a connected aggregate of chains. Careful ordering of the variable list of the problem enables the divide & conquer technique. The fixing of junction variables and thereafter the fixing of the variables halfway the chains break up the problem. Heuristics are used to find the mentioned type of variables within the graph. Third the memoization technique caches the solutions of recurring subproblems. All techniques together makes the tool able to solve realistic problems. Remark that still the complete search space is explored.

5. Example

We discuss the mapping of a 10-tap FIR filter on the DSP56000. The local-based bit-true compiler, described in [12], produces slow code for this example (Figure 6). Each tap, after the multiplication of a delayed input with a coefficient, a right shift over four bits is performed. This is to correctly align for the accumulation. The shifting is expensive because only a one bit shift is possible per cycle. Finally after the 10 filter taps, the accumulation is rounded to 12 bits again.

Figure 7 represents the method applied when having a global scope in mind. The four shift instructions are moved up, over the multiplication, to the coefficient signal. Since shifting of constants can be done at compile-time, this alignment correction is free. An interesting advantage of the optimization techniques is the intelligent mapping of the constant signals of the application graph in order to reduce the number of runtime alignment corrections.

Next, an analysis of the magnitude of the signals in the entire signal flow graph leads to further optimization. See [1] for details. In this example, we can state that an accumulation of 10 signals at most needs 4 extra bits. Since those bits are available in the application, we predict an overflow is not going to happen. The implied overflow mechanism becomes irrelevant. The bit-true demand in the loop is automatically fulfilled and the cycle cost per tap is reduced to one as in the functional, non bit-true case. However, the rounding of the accumulated sum to the result signal is still realized by a software routine, since the exact finite word length behavior is not hardware provided. The cycle cost for that part is identical for local- and global-based cases. Table 1 summarizes the three cases for this FIR application and also for a LMS adaptive filter application.

	cycle cost
10-tap FIR filter:	
- non bit-true case	10 x 1 + 7 = 17
- local-based bit-true case	10 x 29 + 19 = 309
- global-based bit-true case	10 x 1 + 19 = 29
10-tap LMS adaptive filter:	
- non bit-true case	10 x 2 + 9 = 29
- local-based bit-true case	10 x 41 + 64 = 474
- global-based bit-true case	10 x 7 + 71 = 141

Table 1. The cycle cost of two basic applications on the DSP56000 for different compilation paths.

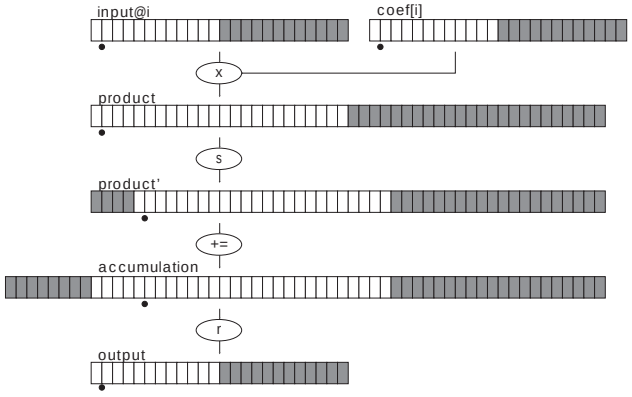

Figure 6. Local approach for the 10-tap FIR filter on the DSP56000.

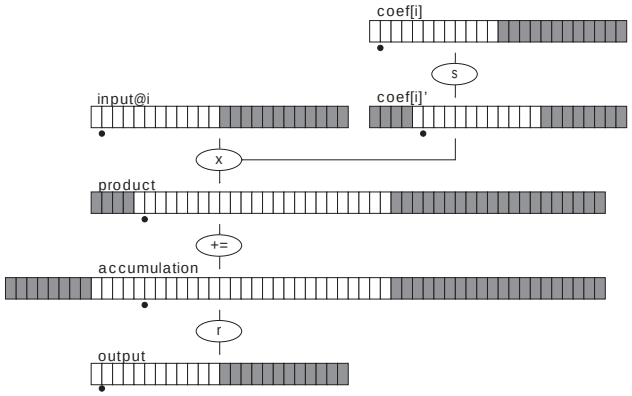

Figure 7. Global approach for the 10-tap FIR filter on the DSP56000.

6. Conclusion

Simulation and emulation are important steps in the design of DSP applications. Bit-trueness of the compiler is essential, in order to check and evaluate the accuracy level of the outputs of the application. Because the finite word length mechanisms of the signals and operators of the application differ from those on the simulation target, extra code is inserted around each operation. An entire dataflow analysis combined with a total exploration of the available mapping flexibility results in a better merging of the specification of the application and the features of the target. Based on examples, we proved that the new technique leads to code sizes and execution times one order of magnitude smaller than those of present-day compilers.

References

[1] L. De Coster, M. Engels, R. Lauwereins, and J. Peperstraete. Global versus local approach for compiled bit-true simulation of DSP applications. Technical report, K.U.Leuven, ESAT-ACCA, July 1995.

[2] P. Hilfinger. A high-level language and silicon compiler for digital signal processing. In *Proc. of the IEEE Custom Integrated Circuits Conf.*, pages 213–6, May 1985.

[3] http://www.angeles.com/.

[4] http://www.cadence.com/.

[5] http://www.mathworks.com/.

[6] http://www.mentorg.com/.

[7] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A fixed-point design and simulation environment design. In *Proc. of the Automation and Test Conf. in Europe*, Feb. 1998.

[8] S. Kim and W. Sung. Fixed-point simulation utility for C and C++ based digital signal processing programs. In *Proc. of the 28th Asilomar Conf.*, volume 1, pages 162–6, Oct. 1994.

[9] J.-L. Lions. Ariane 5 flight 501 failure. Technical report, ESA, http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, July 1996.

[10] Mentor Graphics. *DSP Architect DFL user's and reference manual*, 1996.

[11] Mentor Graphics. *DSP Station C code generation user's manual*, 1996.

[12] Mentor Graphics. *DSP Station DSP56000 code generation user's manual*, 1996.

[13] K. Schoofs, G. Goosses, and H. De Man. Bit-alignment for retargetable code generator. In *Proc. of the 7th Int. Symp. on high-level synthesis*, pages 76–81, May 1994.

[14] W. Sung and K.-I. Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *IEEE Trans. Signal Processing*, Dec. 1995.