FRIDGE: A Fixed-Point Design and Simulation Environment

Holger Keding, Markus Willems, Martin Coors and Heinrich Meyr
Integrated Signal Processing Systems
RWTH Aachen, University of Technology
D-52056 Aachen, Germany

Abstract

Digital systems, especially those for mobile applications are sensitive to power consumption, chip size and costs. Therefore they are realized using fixed-point architectures, either dedicated HW or programmable DSPs. On the other hand, system design starts from a floating-point description. These requirements have been the motivation for FRIDGE [1], a design environment for the specification, evaluation and implementation of fixed-point systems. FRIDGE offers a seamless design flow from a floating-point description to a fixed-point implementation. Within this paper we focus on two core capabilities of FRIDGE:

(1) the concept of an interactive, automated transformation of floating-point programs written in ANSI-C into fixed-point specifications, based on an interpolative approach. The design time reductions that can be achieved make FRIDGE a key component for an efficient HW/SW-CoDesign.

(2) a fast fixed-point simulation that performs comprehensive compile-time analyses, reducing simulation time by one order of magnitude compared to existing approaches.

1 Introduction

Digital system design is characterized by ever-increasing system complexity that has to be implemented within reduced time, resulting in minimum costs and short time-to-market. These characteristics call for a seamless design flow that allows to perform the design steps on the highest suitable level of abstraction.

For most digital systems, the design has to result in a fixed-point implementation, either in HW or SW. This is due to the fact that these systems are sensitive to power consumption, chip size and price per device. Fixed-point realizations outperform floating-point realizations by far with regard to these criteria.

A typical fixed-point design flow is depicted in Fig.1. Algorithm design starts from a floating-point description that can be analyzed by means of simulation.

[1] **F**ixed-point p**R**ogramm**I**ng **D**esi**G**n **E**nvironment

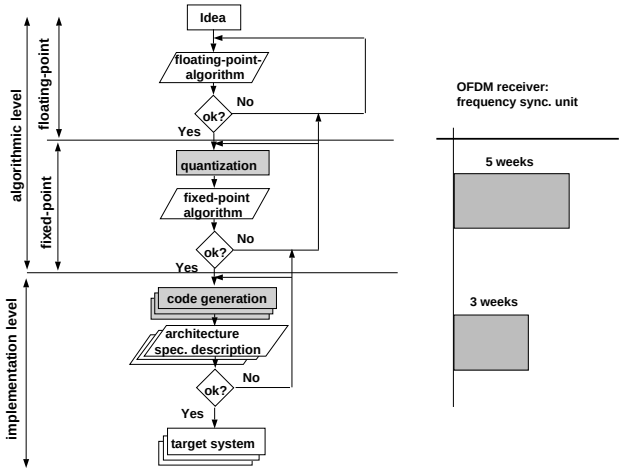

Figure 1: Fixed-point design process

This abstraction from all fixed-point effects allows an evaluation of the algorithm space before analyzing quantization effects on the algorithmic behavior. Additionally, the use of floating-point models offers a maximum degree of reusability.

The transformation to the fixed-point level is quite tedious and error-prone. It requires one to assign a fixed wordlength and a fixed exponent to every operand. For more complex designs more than 50% of the design time is spent on the algorithmic fixed-point level once the floating-point model has been specified (as illustrated by Fig.1). Two reasons can be offered:

- The transformation of the floating-point algorithm to the fixed-point algorithm has to be done manually, which is known to be time-consuming and error-prone. Even for a single transformation, modeling efficiency is very low. Especially in HW/SW-CoDesign typical designs require multiple float-to-fixed transformations. This is due to the fact that while performing a transformation on the algorithmic level, one can no longer abstract from the target system since HW and SW put different constraints on the fixed-point specification: for SW, the wordlength is already fixed and the minimization of shift operations is of interest, while for HW the wordlength is free and its minimization is a concern. Therefore, the manual float-to-fixed conversions no longer appears to be acceptable.
- The fixed-point simulation efficiency is low. This is due to the fact that the fixed-point data type has no

built-in data type on the host machine but has to be emulated. Existing concepts perform all necessary emulation steps at run-time.

These inefficiencies have been the motivation for FRIDGE, an interactive design environment for the specification, simulation and implementation of fixed-point systems. Two key ideas that significantly speed up the specification and simulation of fixed-point systems are described in this paper. The structure is as follows: after an overview of existing concepts for the specification and simulation of fixed-point systems (Sec. 2), in Sec. 3 we present the interpolative approach. The realization of this concept within FRIDGE is presented in Sec. 4. The advanced fixed-point simulation capabilities of HYBRIS, the simulator that accompanies FRIDGE, are subject to Sec. 5.

2 Related Work

Several design environments allow the specification of a fixed-point algorithm, starting from a floating-point description of the system. Two concepts exist:

1. Block diagram-based algorithm specifications where blocks represent the functionality and signals the data flow among these blocks:
   The designer assigns fixed-point attributes to the signals, but the blocks' internal fixed-point behavior cannot be influenced [1, 2, 3]. The blocks' black box behavior is a severe limitation of these approaches. As a consequence, functionalities are limited to blocks with the fixed-point behavior completely specified by the interface, such as adders, multipliers, etc. More complex functionalities, e.g. filters, different fixed-point behaviors require the designer to manually exchange or even rewrite the block.

   Another design bottleneck is that a complete fixed-point specification requires to specify **all** signals. For most concepts, this calls for a **manual** assignment. Recently, Sung [4] presented a concept to reduce the effort of manual annotations based on exhaustive simulations to determine the fixed-point format of all non-specified signals. Long system response times restrict this concept to only a very few non-specified signals. Since typical designs often include 1000 signals or more [5], manual assignment is still necessary for most of the signals.
2. Textual descriptions where the system's functionality is described using a programming language:
   Examples include DFL [6] as well as C^{++}-based concepts [7]. Both concepts allow a fixed-point instantiation of variables at **declaration time**, i.e. a variable keeps a unique fixed-point representation throughout the complete program. Both make use of operator overloading, which requires minimum modifications of the code. As for the block diagram concepts, for a fixed-point algorithm specification all variables have to be annotated manually.

All these design approaches share a common simulation principle, namely performing an emulation of the fixed-point behavior at runtime. Emulation covers two aspects: emulating the fixed-point arithmetic, since no built-in general fixed-point data type exists on the host machine, and emulating the casting operations, which include overflow and quantization handling. Because of these runtime emulations, simulation time is increased by one or even two orders of magnitude compared to the corresponding floating-point simulation. To reduce the casting overhead, in [8] DeCoster presents a concept where he analyzes at compile-time whether an overflow or a quantization can appear at all. This compile-time analysis can reduce simulation time significantly. To the authors' knowledge, no concepts have been reported that determine the appropriate built-in data type for each operation at compile-time as well.

3 The Interpolative Approach

A fixed-point specification of an algorithm needs the assignment of a three-tuple $< \texttt{wl}, \texttt{iwl}, \texttt{s} >$ to every operand, with `wl` the wordlength, `iwl` the number of integer bits and `s` the sign (which might be *unsigned* or *two's complement*). See Fig.2 for the representation.

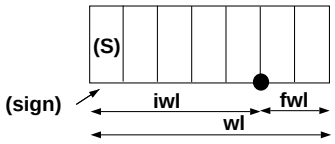

Figure 2: Fixed-point data type specification

As pointed out above, the manual annotation of all operands as required by the existing concepts is hardly acceptable for a single transformation. Since an efficient evaluation of the complex design space requires multiple transformations, it is even more of a design bottleneck.

As a consequence, we propose an alternative design flow, denoted the **interpolative approach** which is illustrated by Fig.3.

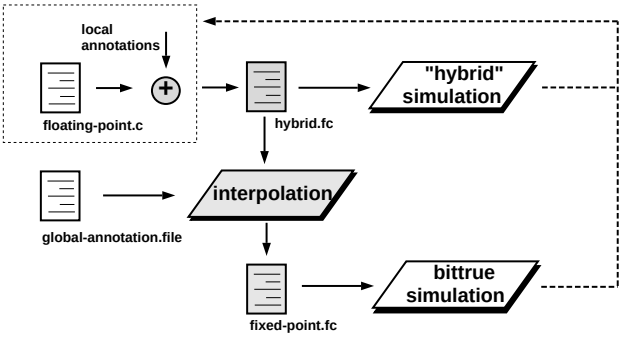

Figure 3: Design flow based on interpolation

1. Local annotations:
   Starting from the floating-point description, the designer assigns fixed-point information to **some** fixed-point operands that are critical to his design or that have a known fixed-point specification (e.g. the interface format of a system). This information might be the complete fixed-point data type, or only partial information such as the wordlength or the integer wordlength. These annotations result in a *hybrid* specification, i.e. some operands fixed-point, while

the majority (to our experience about 95%) remain floating-point.

2. Simulation:
   The hybrid specification is simulated to check whether the locally annotated specification still meets the design criteria. If not, modifications to the local annotations or even to the structure of the algorithm become necessary.

3. Interpolation:
   Once the annotated program matches the design criteria, the remaining floating-point operands are transferred to fixed-point operands by interpolation. 'Interpolation' means the determination of the fixed-point parameters of the non-annotated operands from the information that is inherent to the annotated operands.

   The interpolation concept is based on three key ideas:

   i) Format propagation:
   The fixed-point parameters *iwl* and *sign* of an operand can be determined once the range $[min, max]$ is known that this operand can take (two's complement representation assumed):

$$sign = \begin{cases} u & if \quad min \geq 0 \\ s & else \end{cases}$$

$$iwl = \lceil \max\{ld|min| + 1, \frac{ld|max|}{ld(2^{-1} - 2^{-wl})}\rceil$$

   It is important to notice, that without the additional knowledge of the fixed-point parameter *wl* the computation has to be more conservative, i.e. to guarantee, that the maximum (minimum) can be represented one additional bit has to be spent:

$$iwl = \max\{\lceil ld|min| + 1\rceil, \lfloor ld|max|\rfloor + 2\}$$

   The parameter *wl* can then be determined using the information about the fractional wordlength *fwl = wl - iwl*. It is only necessary to represent those fractional bits which actually carry any information.

   Given the information of the operator and the fixed-point format of the inputs to the operation, both the range and the relevant fractional wordlength of the result can be determined.

   Format propagation requires an analysis of the data flow and control flow of the program at compile-time, i.e. prior to executing the program. This concept guarantees that no information can be lost ,except for the division.

   ii) Global annotations:
   While local annotations express fixed-point information for single operands, global annotations describe restrictions that have to be matched throughout the complete design.

   For different targets, different global restrictions apply. For SW, the functional units to perform specific operations are already defined. Consider a 16x16 bit multiplier, writing to a 32 bit register. A global annotation can inform the interpolator that the wordlength of a multiplication operand is not allowed to exceed 16 bit, while the result may have a wordlength of 32 bits. For an ASIC implementation, no fixed wordlength constraints exist for specific arithmetic units. So global annotations might inform about a maximum wordlength *max* that shall not be exceeded. Once the propagation would result in a wordlength exceeding *max_wl*, it is reduced to the *default_wl* wordlength.

   iii) Designer support:
   If an interpolation is not possible for the complete design since the annotated information is not sufficient, the interpolator can inform the user about the location where it is impossible to continue and can ask for additional information.

   The interpolation supplies a fully annotated program, where a unique fixed-point data type is assigned to each operand. Therefore, the effects of local and global annotations become completely visible to the designer.

4. Simulation:
   Since the global annotations might have changed the algorithmic performance of the specification, the fixed-point program has to be simulated again. If one finds that the system does not fulfill the design criteria, the initial description might be modified by adding or changing annotations.

The interpolative design flow comes with several advantages compared to existing approaches:

- design time reduction: the designer can concentrate on the specifications which are important to the design while the time-consuming task of annotating the remaining parts is done in an optimum way by the interpolator. This interpolation only takes about as long as compiling the program.
- designer's control: the designer fully controls the transformation process since he can assign all information that is critical for the design. The interpolation makes visible the effects on those parts of the design that have not been specified explicitly by local annotations. This simplifies iterative modifications when the designer wants to assign additional annotations.
- Design space evaluation: the evaluation of different fixed-point specifications becomes very easy since only some annotations have to be exchanged while the remaining specifications are automatically derived from this information.

4 The FRIDGE Framework

FRIDGE is a complex and most advanced design environment for the specification and code generation for fixed-point architectures, as illustrated in Fig. 4.

Within this section we focus on the transformation of the floating-point algorithm to the fixed-point algorithm, which is based on the interpolative approach as described in Sec. 3. The hybrid simulator HYBRIS is described in more detail in Sec. 5.

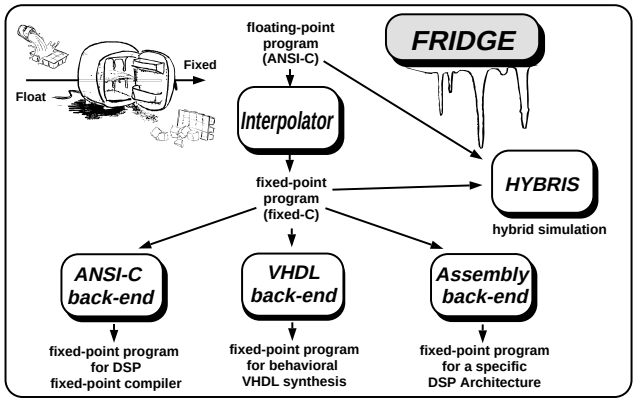

Figure 4: The FRIDGE Framework

4.1 fixed-C

The input to FRIDGE is a floating-point program written in ANSI-C. Most block diagram based design environments come with a C-code generator which allows one to transfer these specifications into ANSI-C as well [1, 3, 6, 9].

ANSI-C offers no efficient support for fixed-point data types [10], but since the interpolative approach calls for an option for hybrid specifications of the algorithm, the ANSI-C syntax has been extended to the language *fixed-C* by introducing two parameterizable fixed-point data types, named `fixed` and `Fixed`.

4.1.1 The Data Type `fixed`

```
fixed     a,*b,c[8];
```

A variable is declared to be of data type `fixed`, but no instantiation is performed at declaration time. *fixed-C* permits pointers and arrays as known from ANSI-C.

```
a=fixed(wl,iwl,sign,cast,*b);
```

`a` receives data type `<wl,iwl,sign>`, the value of `*b` is casted according to `cast`. The casting mode specifies how to handle overflow as well as quantization. The different casting modes are shown in Tab. 4.1.1.

casting modes	overflow handling *saturation*	overflow handling *wrap around*	no overflow handling
quantization by *rounding*	sr	wr	nr
quantization by *truncation*	st	wt	nt
no quantization handling	sn	wn	nn

Table 1: Different modes for cast to `fixed`

Every assignment to a variable overwrites all prior instantiations. This concept of **assignment-time instantiation**(ATI) allows the assignment of different, context-specific fixed-point formats to the same variable within the same program. This is motivated by the specific design flow: transformation starts from a floating-point program, where the designer does not care about fixed-point requirements when he specifies variables for an assignment. Instead he uses the floating-point description to abstract from these problems. Fig.5 shows the differences between ATI compared to the concept of declaration-time instantiation (DTI), as supported e.g. by [6, 7].

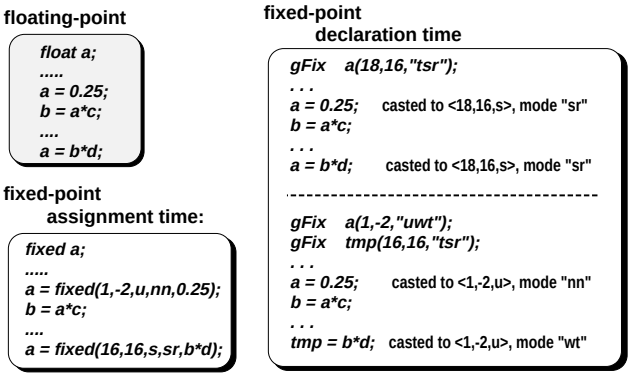

Figure 5: Assignment-time vs. declaration-time instantiation

For the first assignment to `a`, a fixed-point format $< 1, -2, \mathtt{u} >$ is sufficient to represent the constant 0.25 without losing information, while for the second assignment a format $< 16, 16, \mathtt{s} >$ may be necessary. For DTI, two options exist:

- the declaration of variable `a` merges the requirements for all assignments to the variable (2 fractional bits from the first assignment, 16 integer bits from the second). This information being e.g. the input to an implementation tool will force the usage of an 18-bit register whenever `a` is used.
- variable `a` might be renamed, with a context specific instantiation at declaration time. This will force the designer to restructure his code. All usages of `a` in this context have to be replaced as well.

The concept of local annotations relies on ATI simply because it is the most convenient way to assign operand specific information without any possible side-effects on different contexts in the code.

4.1.2 The Data Type `Fixed`

```
Fixed<wl,iwl,sign>     d,*e,g[8];
```

Different from data type `fixed`, FRIDGE performs a data type check for every assignment to a `Fixed` variable. Example:

```
Fixed<6,3,s>  d;
d = fixed(7,4,s,sr,*e); /* type mismatch */
```

The right value of the assignment has format $< 7, 4, \mathtt{s} >$. This does not match the format required for the left value `d` as specified by the declaration ($< 6, 3, \mathtt{s} >$). FRIDGE behaves differently depending on the selected mode:

- warning mode: FRIDGE informs the designer about the mismatch but continues to assign the right hand side to variable `d` (so `d` receives format $< 7, 4, \mathtt{s} >$). Except for the warning, there is no difference between `d` declared as `Fixed` or `fixed`.
- forced mode: FRIDGE interrupts execution and calls for annotations that guarantee the correct assignment to `d`.

`Fixed` is the data type of choice for variables that serve as an interface to other functionalities. Therefore, `Fixed` is the enabling feature for concurrent engineering.

The *fixed-C* syntax is fully consistent to C^{++}. This makes it very intuitive for the user and allows one to compile a simulation on every system that comes with a C^{++} compiler.

4.2 Interpolation

The interpolator integrated in FRIDGE is based on the interpolation principles as described in Sec.3. A more detailed description can be found in [11]. Some capabilities shall be highlighted by the following sections.

4.2.1 Sequential Code

FRIDGE input	FRIDGE output
`global_cast(sr);`	
`float b=2.75;`	`fixed b=fixed(4,2,u,sr,2.75);`
`float c=5.0;`	`fixed c=fixed(3,3,u,sr,5.0);`
`float a;`	`fixed a;`
`a = c - b;`	`a=fixed(4,2,u,sr,c-b);`

For constants, the interpolator determines the minimum requirements on *iwl, wl* and *sign* that are necessary to represent the data without losing any information. By interpolation, it determines the range (and therefore *iwl* and *sign*) and the sufficient fractional wordlength *fwl* of `a`.

4.2.2 Conditional Structures

FRIDGE input	FRIDGE output
`global_cast(sr);`	
`float b=2.75;`	`fixed b=fixed(4,2,u,sr,2.75);`
`float c=5.0;`	`fixed c=fixed(3,3,u,sr,5.0);`
`float a;`	`fixed a;`
`if (condition)`	`if (condition)`
`a = c - b;`	`  a=fixed(4,2,u,sr,c-b);`
`else`	`else`
`a = 1.875;`	`  a=fixed(4,1,u,sr,1.875);`
`d = a;`	`d=fixed(5,2,u,sr,a);`

Depending on the executed branch, `a` is instantiated differently. Before using `a` as an operand following these branches, the interpolator merges the information inherent to all possible assignments so that no information can get lost (here: $iwl = 2$ from the `if-`branch, $fwl = 3$ from the `else-`branch). Notice the effects of ATI: for each assignment to `a`, the fixed-point requirements become visible separately and can be handled in an optimum way for implementation purposes.

4.2.3 Loop Constructs

FRIDGE input	FRIDGE output
`global_cast(sr);`	
`float b[2]=2.75,-3.5;`	`b[2]={fixed(4,2,u,sr,2.75),` `  fixed(4,3,s,sr,-3.5)};`
`float a;`	`fixed a;`
	`int wl_a[]=5,3;`
	`int *pwl_a=wl_a;`
	`int iwl_a[]=3,1;`
	`int *piwl_a=iwl_a;`
`a = 0;`	`a = 0;`
`for (i=0;i<2;i++)`	`for (i=0;i<2;i++)`
`  a = a + b[i];`	`  a=fixed(*pwl_a++,*piwl_a++,` `  s,sr,a + b[i]);`

FRIDGE analyzes the number of iterations. For each iteration it determines the necessary fixed-point parameters separately. If FRIDGE identifies that the parameters are not equal for all iterations, it automatically generates an array containing the iteration specific fixed-point format. These arrays are accessed via pointers. Notice the advantage of iteration-specific instantiations: if, for the final implementation, one decides to unroll or parallelize the loop, he has full access to the minimum fixed-point requirements.

The interpolator is based on a powerful data flow and control flow analysis. It covers pointers, arrays and static variables, as described in detail in [11].

The interpolator can extract information about the fixed-point parameters not only from direct information about wordlength, integer wordlength and sign, but also from indirect or user supplied information. An example is given here:

```
a=fixed(minim(-2.35),maxim(4.74), *b );
```

As described above, this information is sufficient for the determination *iwl* and *sign*.

5 HYBRIS: HYBRId Simulation Classes

HYBRIS are *C++* Classes for a bit-true simulation of *fixed-C*. It permits to simulate algorithms containing both floating-point and fixed-point data types (hybrid code).

HYBRIS makes use of the advanced compile-time optimization of FRIDGE (described in Sec. 5.1) and thereby outperforms existing fixed-point simulations by far.

5.1 Fast Fixed-Point Simulation

Fixed-point simulation increases runtime by one or even two orders of magnitude compared to the corresponding floating-point simulation. This is due to the fact that the fixed-point specification has to be emulated on the host machine which, for the most part, have different data formats than the target machine. Moreover floating-point simulations for complex designs run for hours since often $10^9 - 10^{12}$ data samples have to be processed to receive sufficient statistics. These simulation time increases are hardly acceptable. Therefore, advanced concepts are necessary to speed up fixed-point simulations.

The key idea for a fast fixed-point simulation is to take advantage of all compile-time information to minimize the processing effort that has to be spent at simulation time. Two areas for compile-time optimization have been integrated into FRIDGE/HYBRIS: casting optimization and data type emulation.

5.1.1 Casting Mode Optimization

Whenever an operation result is forced to a specific fixed-point format, a casting mode information is necessary to describe how overflow and quantization effects have to be handled. For completely specified fixed-point programs, this casting mode has to be defined for each assignment and for each intermediate result.

Rebuilding the bit-true behavior of the casting process is a time-consuming task since there is no built-in function at the host machine. A typical cast requires 10 operations written in C to handle the result of a simple operation such as an addition. Obviously, it is highly desirable to

dismiss the casting operation wherever possible. This becomes possible if at compile time it is known that no overflow or change in the quantization can occur at all. Existing concepts do not perform this compile-time analysis but check the casting conditions for every operation.

In contrast, FRIDGE/HYBRIS offers casting mode optimizations that have also been independently proposed by DeCoster [8]. Whether an overflow check is necessary or not depends on the range that an operation result can take. If the possible range can be represented by the available number of integer bits, no overflow check is necessary at all. The same holds for quantization if it is known that no information can get lost due to the casting mode. This range analysis and fractional wordlength determination is exactly what is performed by the interpolator, as described in Sec. 4.

The tight connection of the casting mode analysis to the interpolator's functionality does not come as a surprise since the key idea of interpolation has been to determine the fixed-point specification of the non-annotated operands so that no information can get lost. If no information can get lost, absolutely no casting is necessary. Therefore, if a simulation program results from an interpolation using FRIDGE, a casting has to be performed only for those fixed-point specifications, that have been directly effected by local or global annotations.

5.1.2 Data Type Selection

A general fixed-point data type, e.g. defined by *fixed-C*, has no built-in data type on the host machine. Therefore, its arithmetic behavior must be emulated. Different concepts for this emulation exist.

In DFL [6], this is done using *short* variables in C. Variables (or signals in DFL syntax) that cannot be represented by a data type *short* are represented using arrays of *short*. In order to receive the bit-true specification, logical tests and modifications using shift operations to adjust the different binary points have to be included. This is done at runtime.

Kim [7] makes use of the operator overloading capabilities of C^{++}. Every variable is described by 6 class members. The bit string is represented by a *long* variable, the integer wordlength by a *short* variable. For every operation a complex analysis of the mantissa and the integer wordlength is performed at runtime. E.g., for a simple addition, the simulator performs about 20 operations. In addition to this runtime overhead, the representation of the mantissa by a *long* variable restricts the wordlength to a specific, host machine-dependent length (typically 32). The same holds for the integer wordlength representation using a *short* variable (typically 16).

In contrast to these concepts, HYBRIS makes use of the compile-time information about the fixed-point specification of each operand.

The key concept is based on a simple idea: perform integer arithmetic using ANSI-C integer data types whenever this is sufficient to rebuild the fixed-point specification. Only if the fixed-point specification cannot be rebuilt by ANSI-C does one switch to a fixed-point emulation using C^{++} overloading capabilities.

ANSI-C Integer Code Generation The concept of ANSI-C Integer Code Generation ('Integer'-C generation) from a *fixed-C* specification is described in detail in [12]. Therefore, here we restrict ourselves to presenting only the basic concepts.

Fig.6 shows the principle: the relevant bit-string with $wl = 5$ is embedded into the integer bit-string that is available on the host machine, denoted by *mwl* (machine wordlength). *mwl* corresponds to the wordlength of the built-in data types `short, int` or `long`.

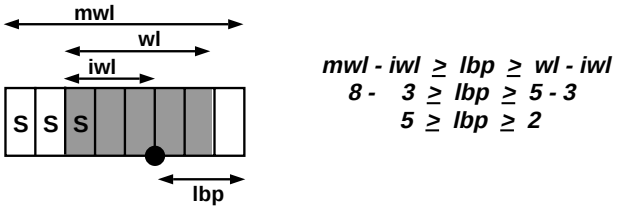

Figure 6: Embedding a relevant bit-string with *wl=5* into an integer bit-string with *mwl=8*

lbp denotes the location of the binary point once the relevant bit-string has been embedded. If $mwl > wl$, there is no unique transformation, but some degree of freedom exists, expressed by the inequality for *lbp*.

Additional restrictions on each operand's individual *lbp* come with the arithmetic operations, as it is illustrated in Fig.7 for the addition.

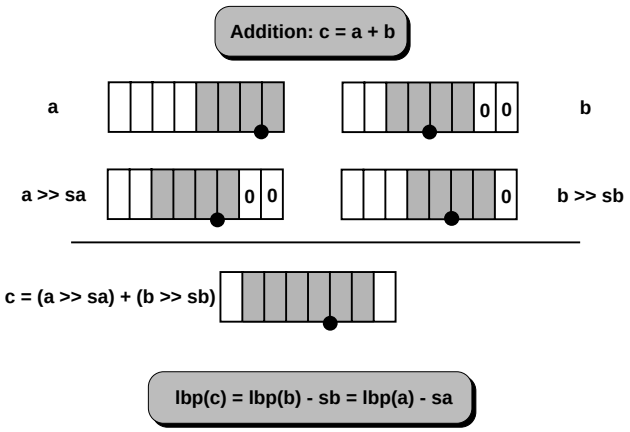

Figure 7: Requirements on *lbp's* imposed by the addition

Using this concept it is possible to describe the transformation space through a set of inequalities. This holds for rebuilding the casting operations as well. HYBRIS identifies the appropriate shift operations to rebuild the functionality and generates an Integer-C code whenever possible [12].

Emulating the fixed-point behavior It is impossible to emulate the fixed-point behavior using Integer-C if the operand does not fit into the machine word, e.g. `c = fixed(37,12,s,nn,a+b)` given a machine wordlength *mwl=32*. Since the complete fixed-point information is available at compile time, these non-fitting

operands can be identified at compile-time. If so, the fixed-point behavior must be rebuilt using an emulation data type.

The emulation data type `fixed_emu` represents the relevant bit-string by an array of `long`, and carries the information how to interpret this bit-string (*iwl*, *sign*). All arithmetic operators are defined for this data type, therefore operator overloading becomes possible.

In addition, interfacing routines exist for converting integer operands into `fixed_emu` operands and vice versa:

```
fixed_emu a;
int       b;

/* cast from emu to int */
b=a.return_int(wl,iwl,sign,cast,lbp)
/* cast from int to emu */
a=fixed_emu(wl,iwl,sign,lbp,b)
```

Therefore, mixed specifications become possible and can be compiled by any C^{++} compiler:

```
fixed_emu  *coeff, *state;
int        sum,i;
....
sum=0;
for (i=0; i < 5; i++)
 sum=(sum >> 2) +
 (*coeff++**state++).return_int(16,13,s,sr,5);
```

The multiplication is performed using the emulation class, since the multiplication result shall be represented with maximum accuracy before its wordlength is reduced. The addition can be performed using integer arithmetic. The second operand that is casted from the `fixed_emu` type is requested to be a long variable that contains the information of *wl=16* relevant bits, *iwl=13* of them integer bits. This is a signed representation, with the result of the multiplication casted to this format according to mode *sr*, while the relevant 16 bits are embedded into the `int` string with an *lbp=5* (while *fwl=3*, so the two LSB-bits of the returned integer string are set to 0).

Notice that with HYBRIS there is no general restriction on the wordlength, as it is inherent to other concepts. Since most designs can be simulated using the integer emulation, the increase in simulation time compared to the floating-point model is mostly due to casting operations. Applying the HYBRIS approach as presented above, this overhead is reduced to a minimum.

5.2 Integration into existing environments

HYBRIS can easily be integrated into all C-based simulation environments. Test vector generation, post processing and performance analysis can be done by these environments, while the HYBRIS classes provide the handling of the fixed-point data types.

6 Summary

Existing approaches make it necessary to annotate fixed-point specifications to **all** operands manually, an error-prone and time-consuming task. This is hardly acceptable for a single transformation but becomes an unacceptable situation for an efficient evaluation of the complex design space. The interpolative approach, which is a key feature of FRIDGE, a fixed-point specification starting from local annotations for **some** specific operands of the floating-point program. This became possible by introducing *fixed-C*, ANSI-C extended by two fixed-point data types.

The verification of each transformation has to be performed by means of simulation where existing fixed-point simulation concepts increase simulation time by one or even two orders of magnitude compared to the corresponding floating-point simulation. The efficient fixed-point simulation using HYBRIS uses advanced compile-time analysis concepts analyzing necessary casting operations and selecting the appropriate built-in data type on the host machine.

These features, in combination with target-specific implementation strategies, make FRIDGE a powerful design environment for the specification, evaluation and implementation of fixed-point algorithms. This environment can easily be integrated into existing C-based design environments.

References

[1] Cadence Design Systems, 919 E. Hillsdale Blvd., Foster City, CA 94404, USA, *SPW User's Manual*.

[2] Angeles Systems, *VANDA-Design Environment for DSP Systems*, 1994.

[3] Mathworks Inc., *Simulink Reference Manual*, Mar. 1996.

[4] W. Sung and K. Kum, "Simulation-Based Word-Length Optimization Method for Fixed-Point Digital Signal Processing Systems," *IEEE Transactions on Signal Processing*, vol. 43, pp. 3087 – 3090, Dec. 1995.

[5] P. Zepter, T. Grötker, and H. Meyr, "Digital Receiver Design using VHDL Generation from Data Flow Graphs," in *Proc. 32nd Design Automation Conf.*, June 1995.

[6] Mentor Graphics, 1001 Ridder Park Drive, San Jose, CA 95131, USA, *DSP Station User's Manual*.

[7] S. Kim, K. Kum, and W. Sung, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs," in *Workshop on VLSI and Signal Processing '95*, (Osaka), pp. 197–206, Nov. 1995.

[8] L. DeCoster, M. Engels, R. Lauwereins, and J. Peperstraete, "Global Approach for Compiled Bit-True Simulation of DSP-Applications," in *Proceedings of Euro-Par'96*, vol. 2, (Lyon), pp. 236–239, Aug. 1996.

[9] Synopsys, Inc., 700 E. Middlefield Rd., Mountain View, CA 94043, USA, *COSSAP User's Manual*.

[10] W. Sung and K. Kum, "Word-Length Determination and Scaling Software for a Signal Flow Block Diagram," in *Proceedings of ICASSP '94*, pp. II 457– 460, Apr. 1994.

[11] M. Willems, V. Bürsgens, H. Keding, T. Grötker, and H. Meyr, "System Level Fixed-Point Design Based on an Interpolative Approach," in *Proceedings of the Design Automation Conference (DAC)*, (Anaheim), pp. 293–298, Jun. 1997.

[12] M. Willems, V. Bürsgens, and H. Meyr, "FRIDGE Enables Floating-Point Programming of Fixed-Point DSPs," in *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, (San Diego), pp. 1000–1005, Sep. 1997.