Bit-Alignment for Retargetable Code Generators *

Koen Schoofs Gert Goossens Hugo De Man[†]

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

When building a bit-true retargetable compiler, every signal type must be implemented exactly as specified, even when the word-length of the signal does not match the length of the available hardware. Extra operations must be introduced in the algorithmic description in order to ensure that the remaining bits do not influence the data-bits and to assure that signal types are correctly converted from one type to another. An algorithm will be presented which generates code to assure bit-trueness, optimised for the available hardware.

1 Introduction

In DSP-algorithms every signal has a certain *signal type*, indicating the number of bits in the signal, the number of bits behind the binary point, and the way of encoding (signed, unsigned, two's complement, etc ..). In most algorithms, signals with a large variety of types are present. The *word length* of these signals (the number of bits they contain) can be different from the size of the functional unit they are mapped upon. If the word length is bigger, multiple precision arithmetic must be used; if they are smaller, it must be decided to what value the remaining bits (called *non-data bits*) must be set. We have to decide how many non-data bits we allow at the most significant bit (*msb*) and least significant bit (*lsb*) side of the data word, and to what values these bits can be set in order to avoid that they corrupt the data bits during arithmetic operations (the operations must remain *bit-true*). The allowed values of these bits will depend strongly on what kind of operations will be done with these signals. (see figure 1)

The presence of different types normally implies that during computation, certain signals must be converted from one type to another. This is specified by means of *cast* operations in the algorithmic specification. A *cast* operation may imply that certain bits must be removed from the original signal, that extra copies of the sign-bit must be added at the msb-side of the data word, or that extra zeroes must be added at the lsb-side of the data word.

In this paper, we present a technique called *software alignment*, which takes care of the bit-trueness of a design and correctly implements type changes by adding operations to the design. This is needed because retargetable code generators can not simply add dedicated hardware

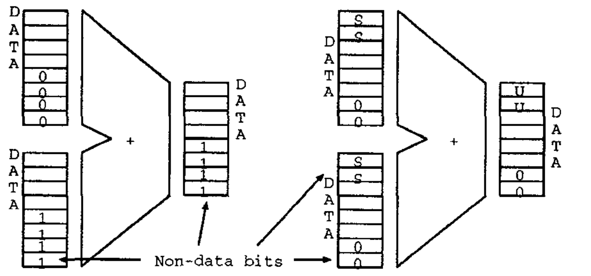

Figure 1: two possible alignments for the addition of two 4-bit data words on an 8-bit adder

to add and remove bits during type changes, since they generate code for predefined processor cores. The bit-true character is an important requirement for retargetable code generation, because for certain applications (like filters) the actual type of the signals is important to obtain the desired result. More-over programmable DSP-cores are often used to do rapid prototyping when designing custom application specific architectures (ASICs). In order to evaluate what the influence of the actual hardware bit-widths in the ASIC will be on the design, we need to have a bit-true code generator.

In this paper we assume an architecture model in which complex programmable *data paths*, consisting out of multiple functional units (*FUs*), are interconnected by busses. An example with two data paths, a multiplier/accumulator (MPY/ACC) with a downshifter bank and an ALU with an upshifter, is shown in figure 2.

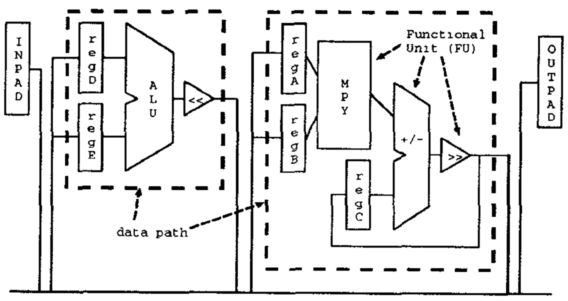

Figure 2: Example architecture of a programmable processor core.

*This research was sponsored by ESPRIT project 2260 "SPRITE" of the E.U.

[†]Professor at the Katholieke Universiteit Leuven

0-8186-5785-5/94 $03.00 © 1994 IEEE

The research described in this paper is part of the CHESS retargetable code generation project.

2 Literature survey

In literature not much attention has been paid to the subject described above. Traditional, software compilers are only concerned with type changes between integer and floating point, and between integers and doubles [4], and do not really support conversions between integers with a different number of bits. Normally the only kinds of integers supported have a length which is a multiple of the register size of the target processor. For retargetable DSP code generators the importance of signal types and the problem of bit-trueness in simulation and implementation was recognised in [5], but no clear solution was presented.

Some related work, concerning *hardware alignment*, can be found in [1]. Hardware alignment solves the same problem as software alignment but for customisable ASIC processors instead of predefined programmable processors. In [1] it is assumed that the interconnection network between the data paths is not fully defined so that the compiler can generate extra wiring to change the alignment and types of the signals. An algorithm is presented which minimises the amount of extra wiring needed for this task. The resulting architecture exactly implements the signal types as given in the original specification.

In [3] an algorithm is presented which *modifies the signal types* from the specification, without degrading the precision of the calculations, also in order to minimise the extra wiring needed to implement the different remaining type changes. This does not necessarily mean that the number of different types is reduced, but only that the number of different word lengths in the different types is reduced. The algorithm is quite useful as a preprocessing step before solving the software alignment problem discussed in this paper. It can not, however, take the place of this algorithm, because it only optimises the types, and does not generate a set of operations which actually implement the type changes.

3 Definitions

3.1 Signal type

Every signal in a DSP algorithm has a signal type which indicates how the data bits must be interpreted. If we assume that the signal is always in two's complement representation, the signal type can be denoted: $< wl, fp >$ where wl is the number of data bits of the signal, and fp indicates the position of the binary point, counting from the least significant data bit, as illustrated in figure 3. *nb* denotes the number of bits of the wire or register carrying the signal.

3.2 Signal alignment

The alignment of a signal is defined if we know the number of non-data bits at the msb and lsb side of a data word and if we know the value of these non-data bits. The alignment-attribute of a signal therefore consists of:

- *The offset*: this integer indicates the number of non-data bits either at the msb or at the lsb side of the data word (the size of the lsb or msb extension) . If one size is known, the other one can be derived since we know the length of the data word and the size of the hardware carrying the data.

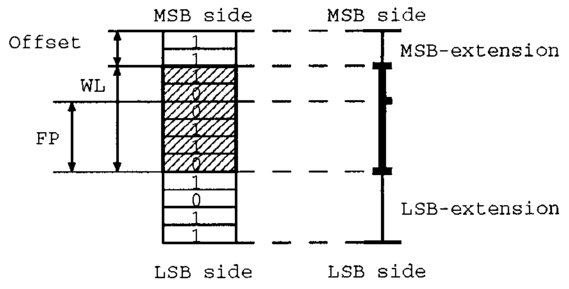

Figure 3: A signal with type <6,4> mapped on a carrier with *nb*=12. To the right is a more schematic representation in which the data bits are represented by a thick line.

- *The alignment side*: This indicates whether the offset is specified for the msb or lsb extension.

- *The msb extension*: This indicates the value of the non-data bits at the msb side of the data word.

- *The lsb extension*: This indicates the value of the non-data bits at the lsb side of the data word.

For both extensions, many different bit patterns are possible. In practice, only the following bit-patterns are useful:

- *Zero-extension*: all non-data bits are set to zero.

- *One-extension*: all non-data bits are set to one.

- *Sign-extension*: all non-data bits are set equal to the sign bit of the data word.

For algorithmic reasons we also define the following:

- *Don't care extension (x-ext)*: To be used when the number of non-data bits is zero, or when the contents of the non-data bits are *irrelevant* for the correct execution of the operation.

- *Undefined extension (u-ext)*: To be used if the contents of the non-data bits can not be classified in any other category, (for example, because not every bit in the extension is set to the same value) or *can not be determined at compile time* (because of, for instance, carry-rippling). This does not mean that these bits are irrelevant to the operation that uses them. This extension can be generated for instance at the msb side of the result of an addition.

As an example, the following alignment attribute:

$$ali = (msb \quad 1 \quad s \quad 0)$$

means that the signal has 1 non-data bit at the msb side of the data word, of which the value is equal to the sign bit of the data word, while all non-data bits at the lsb side of the data word are set to zero.

3.3 Alignment propagation

Most operations do not allow every possible alignment for their input operands. Also for every type of operation there exists a relationship between the input and output offsets and between the input and output extensions. These relationships are parameterised expressions, which model the freedom available in the selection of the actual alignment of the signals. This information is stored once and for all in the library of the compiler, in which all supported operations are declared. In the case of the CHESS library, over 100 operations are supported in this way. *Alignment propagation* then means determining the allowed alignments for every signal in the DSP-algorithm based on the allowed alignments of the operands of the signal. More detailed information about alignment propagation is modeled can be found in [1].

3.3.1 Offset propagation

The relation between input and output offsets can be expressed in mathematical equations. In all practical examples these equations are linear. Each operation contributes a number of equations equal to or less than the number of inputs of the operation, of the form:

$$\sum_{i=1}^{\#outputs} a_i x_i + \sum_{j=1}^{\#inputs} a_j x_j = Cst$$

with a_i and a_j integer and in most practical cases equal to one or zero. Cst is also always integer.

3.3.2 Extension propagation

When we want to express the relation between the input and output extensions for a certain operation, we can not use a mathematical formulation of the same simplicity as for offsets. Instead we use lookup tables. For each operation possible on each functional unit, we require 2 tables per output (one for the msb and one for the lsb side). In each table the value of the output extension is given for each allowed combination of input extension values. As an example the lsb extension table for the the addition (table 1) is given.

PORT A PORT B	0	1	s	x
0	0	1	s	x
1	1	-	-	-
s	s	-	-	-
x	x	-	-	-

Table 1: lsb extension table for the addition

The alignment tables of several operations happening one after the other can be combined into larger alignment tables as is explained in [1].

3.4 Alignment conflicts

We have an *alignment conflict* if the alignment of one operation is unacceptable as input for the next operation. The software alignment algorithm will try to find alignments for each operation which minimise the number of alignment conflicts. However, sometimes alignment conflicts can not be avoided. In the rest of the paper techniques are presented to solve alignment conflicts. For example, extra operations can be inserted in the DSP-algorithm, called *software alignment operations*, which can resolve the alignment conflicts.

3.5 Software alignment operations

Some *FUs* can execute operations, which can modify the alignment of a signal without affecting the values of its data bits. For instance, a shift operation can be used to change the offset, while logical-OR and -AND operations can change the extensions. In a library these *software alignment operations* for the most common functional units (a superset of the hardware available for the particular processor we are generating code for) are stored.

Our library currently supports the following software alignment operations:

$$\begin{array}{rr} out = in + zero; & out = in - zero; \\ out = in + in; & out = in \wedge one; \\ out = in \vee zero; & out = in \oplus zero; \\ out = in \ll n : c_{in}; & out = in \gg n : c_{in}; \\ out = in \cdot 2^m; & \end{array}$$

where $\wedge$, $\vee$, $\oplus$, $\ll n : c_{in}$ and $\gg n : c_{in}$ represent logical AND, OR, EXOR, and up- and downshift respectively. *zero* and *one* are constants with data bits equal to 0 and 1 respectively, and with extensions that can be chosen such to set the desired extensions of the result. *n* is the shift value, which can be chosen to set the result's offset; c_{in} is the value of the bits shifted in. It can be chosen to set the result's extension.

On certain functional units (like an adder-subtractor) a number of different software alignment operations are possible. We can indicate this by combining the extension tables, of the possible software alignment operations into a multiple output table. This is an alignment table with for each combination of input values at most n output-values, each corresponding to a different mode of the functional unit. When the most interesting output extension is finally decided upon, the functional unit is set in the corresponding mode. In the rest of the paper we will ignore this option, in order to reduce the complexity of the examples, without any loss of generality.

4 Software alignment algorithm

Software alignment has to assure the bit-true character of a design. It does this by determining the correct alignment for every signal in the DSP-algorithm, by means of alignment propagation. The propagation minimises the number of alignment conflicts, but in most practical cases, still some conflicts will remain. Remaining conflicts can be solved in two ways by the software alignment algorithm. It can try solve them by replacing existing pass operations by software alignment operations (*software alignment without introduction of extra cycles*). If this is not sufficient it can introduce extra operations (*software alignment with introduction of extra cycles*). Finally software alignment must also implement type changes, where a signal type changes from one type to another.

These three aspects of software alignment will be explained separately, and will be illustrated by means of a small example, mapped upon the architecture presented in figure 2.

4.1 Software alignment without the introduction of extra cycles

If we want to implement the algorithm $z = (a * b) + c$ on the hardware presented in figure 2 , we can first execute $tmp = a * b$ on the multiplier and store tmp in regc. Next, we can compute $z = tmp + c$. In order for a signal to get to its destination, it must travel through a number of FUs which are in pass mode. If we explicitly write the pass modes (this is done already by the instruction selection tool of the retargetable code generator [6]) the algorithm becomes :

$$z = ((a * b) \underbrace{+0}_{\text{pass}}) \underbrace{\gg 0}_{\text{pass}} + c \underbrace{*1}_{\text{pass}} \qquad (1)$$

If we now replace the pass modes by software alignment operations, we can at the same time implement alignment changes required to solve possible alignment conflicts and transport the signal from one operation to the next.

$$z = (a * b \underbrace{+ zero)}_{\text{align}} \underbrace{\gg n : c_{in}}_{\text{align}} + c \underbrace{*2^m}_{\text{align}} \qquad (2)$$

This equation must now be solved in the offset and extensions of the different signals and constants. If it can not be solved, extra variables can be introduced, by adding extra software alignment operations, which causes an overhead of extra cycles to be executed by the DSP-algorithm. This can happen when the data path does not have any FUs in pass-mode, or if these FUs in pass-modes are unusable to solve the alignment conflict. How often this happens, depends upon the hardware the DSP-algorithm is mapped upon.

4.2 Software alignment with the introduction of extra cycles

If even with the techniques described in the previous section some alignment conflicts remain, extra operations must be added in between to convert the alignment to something that is acceptable. If the conflict occurs between two operations that execute on different data paths, the signal is simply rerouted through a number of other data paths, during which the alignment of the signal is modified. To determine an efficient solution, the compiler has to look up the possible software alignment operations on these additional data paths. For each possible path between an data path input and output, these operations are represented in a separate term, called *data path term (DP term)*. DP terms are generated by the compiler in a preprocessing step. For example, for the MPY-ACC data path in figure 2, a possible DP term would be:

$$out = (in * (2^n) + zero) \gg m : c$$

to indicate that an incoming signal can be multiplied with a power of 2 (which modifies the offset), then added with zero and then downshifted over m positions, with the shift-in bits set to c. The contents of these DP terms can be pruned to take into account encoding restrictions imposed by the processor's controller, if two different functional units can not be in a certain mode at the same time.

From the list of terms, a term is chosen which can transform the conflicting alignment into an acceptable form. If no such term exists a concatenation of these terms must be used, indicating that the signal must pass through several data paths in order to get the correct alignment. If it can be proven that no combination of DP terms will yield a valid solution, we have an error condition. This is the case when the addition of any DP term to all chains of DP terms already computed, does not yield a new term which makes new combinations of input and output alignments possible.

If the conflict does not happen at data path borders, the conflicting signal must first be exported out of the data path (using pass-modes, which can be replaced by software alignment operations). Then the terms described above must be used, and finally the signal must be imported again to the position where the original conflict occurred, again using pass-modes which can be replaced by software alignment operations.

Assume, in the example of equation 1 that a conflict occurs after the multiplication "$a * b$", and that the conflict can be solved with the upshift software alignment operation:

$$out = in \ll n : c_{in}$$

We use the following DP term on the ALU-Shifter data path:

$$out = (in + zero) \ll n : c_{in}$$

We would then get the following equation:

$$\begin{aligned} z \;=\; & ((((a * b) \underbrace{+ zero) \gg n_1 : c_{in_1}}_{\text{export and align}} \\ & \underbrace{+zero) \ll n_3 : c_{in_3}}_{\text{align via DP term}} \\ & \underbrace{*2^{m_1} + zero) \gg n_2 : c_{in_2}}_{\text{import and align}} + c * 2^{m_2}. \end{aligned}$$

The more software alignment operations we add, the more degrees of freedom we have to set our extensions and offset correctly. If more than one solution is possible, clearly the solution which requires the least number of extra cycles must be chosen.

4.3 Cast operations implemented with software alignment

Figure 4 shows the cast of a signal of type $< 4, 3 >$ with alignment =(msb 4 x x) to a type $< 5, 2 >$ with alignment =(msb 1 0 1)

We can split up each cast operation in a number of elementary transformations using the following procedure (see figure 4):

- Step A: First we identify which of the bits in the original type will still be present in the final type.

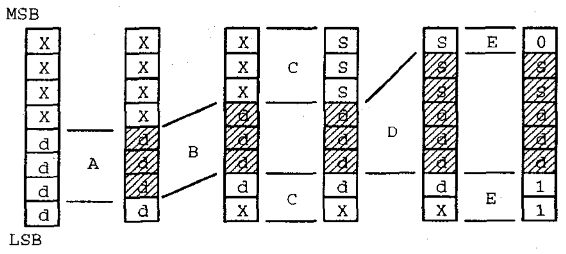

Figure 4: A cast-operation, split up in a number of elementary transformation; Steps B, C and E are to be implemented with software alignment. Shown here is the cast of a $< 4,3 >$ ali=(msb 4 x x) to a $< 5,2 >$ ali=(msb 1 0 1) on a 8 bit architecture. "d" indicates a databit.

- Step B: These bits are then shifted to the position in which they must appear in the final result.
- Step C: Then the sign-extension at the msb side and zero-extension at the lsb-side are generated, if necessary.
- Step D: The bits needed in the signal of the new type are now ready. We still have to identify which bits are now present in the new type.
- Step E: Finally the correct alignment extensions are generated if needed

Steps A and D do not require any special operations, they are just internal bookkeeping. Step B is an offset change, and steps C and E are equivalent to the setting of alignment extensions. Steps B, C, and E can be implemented using the software alignment techniques presented in section 4.2.

4.4 Outline of the software alignment procedure

- Based on information about the available processor hardware, an exhaustive list of all possible DP terms is compiled.
- Where possible, the pass operations in a design are replaced by software alignment operations.
- Alignment attributes are checked throughout the design for conflicts. After this step the alignment of every signal in the design is known. The method for doing this is identical to the alignment propagation technique presented in [1].
- All remaining alignment conflicts are solved using the techniques described in section 4.2.
- The cast operations are implemented using the techniques described in section 4.3.

During the latter two steps, a combination of software alignment operations has to be found which allow us to go from one predefined alignment to another. This is done with a branch and bound method. For each alignment conflict which requires the introduction of extra cycles and for the steps B, C and E of each cast operation, a tree is built starting from the consumption alignment (*ali-cons*) and building towards the production alignment (*ali-prod*). At each level of the tree all available DP terms are applied to the alignments resulting from the previous level of the tree. *Branches which do not contain any new combinations of offset and extension are pruned.* The tree has a finite depth because there are only a limited number of combinations of extensions and offsets. Indeed, the number of extensions is limited, and the offset is an integer number which has as an upper bound the size of the hardware. If finally none of the branches of the tree match the initial alignment, we have an alignment conflict which can not be solved on the available hardware.

It is not required to calculate the entire tree. The tree is built level by level. *As soon as one of the branches of the tree at a certain level contains the initial-alignment, no further levels need to be computed.* This is because the cost function which is used to evaluate the quality of different solutions is simply the number of cycles they need when implemented in the DSP-algorithm. This number of cycles corresponds to the depth of the tree at which the solution was found. If more than one branch contains the targeted initial-alignment, that branch is selected which contains the operations that generates the least number of resource conflicts during scheduling. All this is illustrated in figure 5.

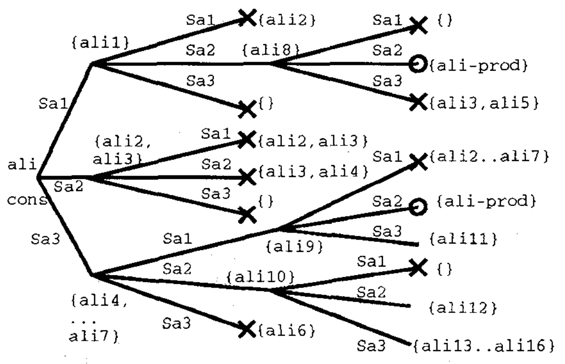

Figure 5: A software alignment tree from ali-cons to ali-prod, for the case where three software alignment operation Sa_1, Sa_2 and Sa_3 are available. A cross indicates a dead-end branch, a circle indicates we have reached the destination alignment.

5 Experiments

In this section a small experiment is presented to show how a software alignment tree is generated. From the cast example in section 4.3 we will generate the operations necessary for step C. We assume only the following DP terms are available:

$$\begin{aligned} out &= in + zero \\ out &= in \gg 1 : sign\_bit \\ out &= in \ll 1 : 0 \end{aligned}$$

and each DP term can be used after itself and after each other DP term. We also do not concern ourselves with importing and exporting the signal to and from data paths. We start with a signal with alignment ali=(msb 3 x u), and we want to generate a signal with ali=(msb 3 s u).

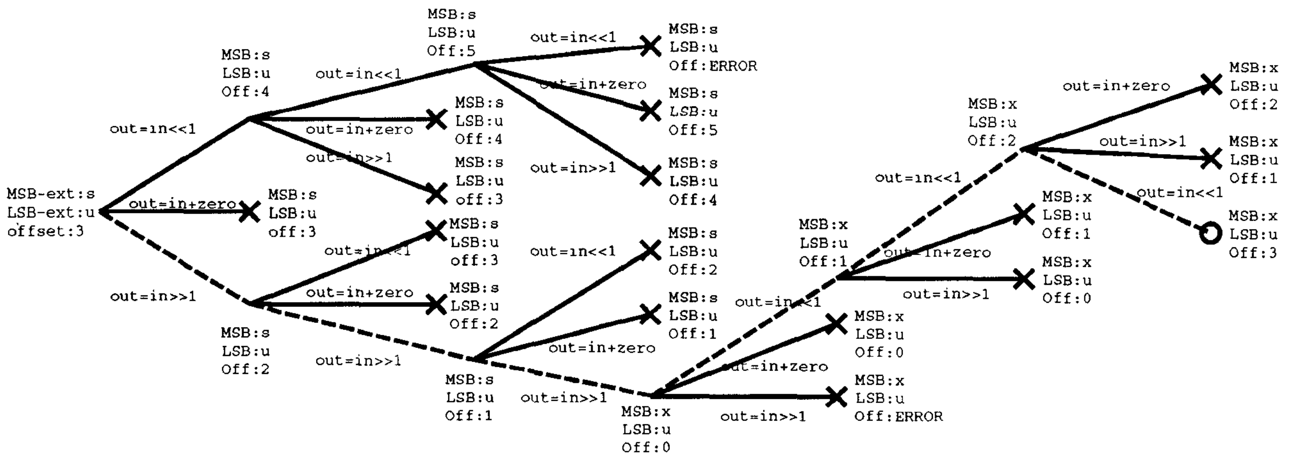

Figure 6: Example of a software alignment tree needed to go from ali-in =(msb 3 x u) to ali-out=(msb 3 s u), if only an addition, shift-up and shift-down are available. The dotted line indicates the best solution. The solution consists of 3 upshifts followed by 3 downshifts.

We build our tree starting from the consumption alignment and we work backwards toward the production alignment. The resulting tree is shown in figure 6. Since the tree is built from consumption to production, the alignment consumed by a DP term is written on the right side of the branch and the alignment produced at the left side. This way for a downshift operation (which increments the offset), the lowest offset is located at the right side of the branch.

Note that in the tree we assume that every branch, and therefore every DP term has only one output extension. In reality most DP terms will have more than one possible resulting alignment. This however does not affect the number of branches in the tree. At each level in the tree we have 3 branches corresponding to each of the possible DP terms. At the first level we see that the resulting alignment in one branch is the same as the alignment at the beginning of the branch. This branch can be pruned, because any solution found in this branch will be more costly than the final solution, because it contains a useless first step. Each new level in the tree represents a new DP term in the equation, and therefore also an extra cycle needed in the DSP-algorithm to execute the alignment change. The production alignment is reached when the tree is 6 levels deep. At this point we had to examine 24 branches. Note that we can stop when the first solution is found. Any other solution would be more costly to implement, because it would be found at deeper levels in the tree and thus would require more cycles to execute in the DSP-algorithm.

By following the path in the tree from production alignment to consumption alignment (i.e from right to left) , we can see that in order to implement the alignment change we need 3 upshifts followed by 3 downshifts, in that order. In the tree we can also find the intermediate alignments of the signal during the transformation. The operations can now be added to the DSP-algorithm.

6 Conclusions

The problem of bit-alignment is important in retargetable code generation for DSP, since DSP algorithms contain signals of many different types. The problem has been largely ignored in literature up till now. The purpose of the software alignment algorithm is to find a bit-true mapping of the design while minimising the required number of extra operations. The algorithm is heuristic in nature. Future work will include the combination of the software alignment algorithm with the type-optimisation technique presented in [3].

References

[1] K. Schoofs, G. Goossens, H. De Man, "Bit-Alignment in Hardware Allocation for Multiplexed DSP Architectures", Proc EDAC 1993, p 289-293.

[2] D. Lanneer, et al, "Architectural Synthesis for Medium and High Throughput Signal Processing with the new CATHEDRAL Environment", published in *"High-Level VLSI Synthesis"*, edited by R.Camposano and W.Wolf, Kluwer, 1991.

[3] K. Schoofs, G. Goossens, H. De Man, "Signal Type Optimisation in the design of time-multiplexed DSP-architectures.", Proc. EDAC1994.

[4] A. V. Aho, R. Sethi, J. D. Ullman, "Compilers, Techniques and Tools", Addison-Wesley Publishing Company, p344-247, 359-364.

[5] D. Genin, J. De Moortel, D. Desmet, E. Van de Velde, "System Design, Optimization and Intelligent Code Generation for Standard Digital Signal Processors.", *Proc. ISCAS 1989*, p 565-569.

[6] J. Van Praet, G. Goossens, D. Lanneer, H. De Man, "Instruction Set Definition and Instruction Selection for ASIPs", *Proc of High Level Synthesis Workshop*, Ontario, 1994.