A MORE EFFICIENT AND FLEXIBLE DSP DESIGN FLOW FROM MATLAB-SIMULINK

P. Coussy, G. Corre, P. Bomel, E. Senn, E. Martin

LESTER LAB, UBS University

ABSTRACT

The design of complex Digital Signal Processing systems implies to minimize architectural cost and to maximize timing performances while taking into account communication and memory accesses constraints for the integration of dedicated hardware accelerator. Unfortunately, the traditional Matlab/ Simulink design flows gather not very flexible hardware blocs. In this paper, we present a methodology and a tool that permit the High-Level Synthesis of DSP applications, under both I/O timing and memory constraints. Based on formal models and a generic architecture, this tool helps the designer in finding a reasonable trade-off between the circuit's latency and its architectural complexity. The efficiency of our approach is demonstrated on the case study of a FFT algorithm.

1. INTRODUCTION

Due to the complexity of today's digital signal processing *(DSP)* applications, designers need a more direct path from the functionality down to the silicon. Layered design flow and associated CAD tools to manage DSP system complexity in a shorten time are thus needed. This led to the development of environments that can help the designer to explore the design space thoroughly and to find optimized designs. In this context, the application design begins by a specification capture of the desired functionality using Matlab/Simulink tool [4]. The system designer next selects hardware component from a library considering constraints criteria e.g. speed, area, or power etc. Follows an architecture exploration concurrently with performance analysis. The hardware and software design tasks then generate respectively an RTL description of hardware blocs and C/C++ code executed on processors.

In [1], [2], and [3] authors propose approaches that use Matlab/Simulink/Stateflow tools for the system specification and that produce a VHDL RTL architecture of the system. Based on hardware macro generators that use the "generic"/"generate" mechanisms, the synthesis process can be summarized as a block instantiation. However, though such components may be parameterizable, they rely on fixed architectural models with very restricted customization capabilities. This lack of flexibility in RTL blocks is especially true for both the communication unit which I/O data scheduling and/or timing requirements are defined, and the memory unit which data distribution is set. Additional communication interface (wrapper) has to be introduced between two components when I/O data order or I/O rates are incompatible. Unfortunately, this adaptation increases the final architecture area and also decreases system performance. In some cases, the I/O timing requirements can not be respected due to the wrapper overhead and can cause the system design to fail. High-Level Synthesis (HLS) can be used to increase flexibility of hardware components.

SystemC Compiler [5] from Synopsys, and Catapult C from Mentor Graphics, in addition to the "super state" mode, hence propose a synthesis mode called "cycle-fixed mode" that maintains a fixed I/O timing behavior that is exactly the same before and after synthesis [6]. Communication is specified using *wait* statements and is mixed with the signal processing specification what limits the flexibility of the input behavioral description. In these two tools, memory accesses are represented as multi-cycle operations in a Control and Data Flow Graph (CDFG). Memory vertices are scheduled as operative vertices by considering conflicts among data accesses. In practice, the number of nodes in their input specifications must be limited to obtain a realistic and satisfying architectural solution. This limitation is mainly due to the complexity of the algorithms that are used for the scheduling. Several other scheduling techniques also include memory issues. Among them, [7] and [8] work only with scalar and try to reduce the memory cost for a given scheduling. [9] and [10] schedule the memory-accesses but do not consider the possibility of simultaneous accesses.

In the domain of real-time and data-intensive applications, processing resources have to deal with ever growing data streams. The system/architecture design has therefore to focus on avoiding bottlenecks in the buses and I/O buffers for data-transfer, reducing the cost of data storage and satisfying strict timing constraints and high-data rates. The design of such applications thus needs methodology that relies on (1) constraint modeling for both I/O timing and internal data memory, (2) constraint analysis steps for feasibility checking and (3) high-level synthesis.

In [10] and [12], we proposed a SoC design methodology based on algorithmic IP core re-using. Based on high-level synthesis techniques under I/O timing constraints, our approach aims to optimally synthesize the IP by taking into account the system integration constraints: the data rate, technology, bus format, and I/O timing properties specified by timing frames of transfers. In [13], we introduced a new approach to take into account the memory architecture and the memory mapping in the behavioral synthesis of real-time VLSI circuits. We formalized the memory mapping as a set of constraints for the synthesis, and defined a Memory Constraint Graph and an accessibility criterion to be used in the scheduling step. We used a memory-mapping file to include those memory constraints in our HLS tool GAUT [14]. ***In this paper, we propose a design flow based on formal models that allow high-level synthesis under both I/O timing and memory constraints for digital signal processing algorithms. DSP systems designers hence specify the I/O timing, the computation latency, the memory distribution and the application's data rate requirements that are the constraints used for the synthesis of the hardware components. This approach can be integrated in the Matlab/Simulink's design flow in order to increase the flexibility of hardware blocs.***

This paper is organized as follows: First in section 2 we formulate the problem of synthesis under I/O timing and memory constraints. Section 3 presents the main steps of our approach, and its underlying formal models. In section 4, we demonstrate the efficiency of our approach with the didactic example of the Fast Fourier Transform (FFT).

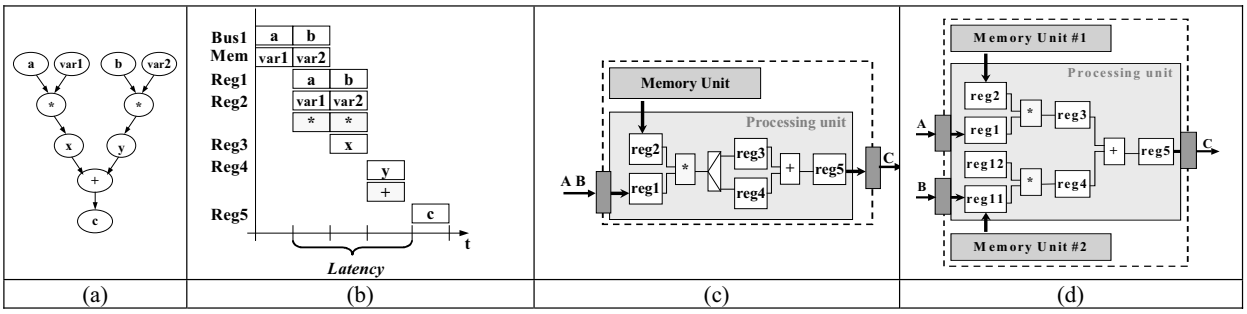


Fig. 1: (a) Signal Flow Graph SFG, (b) Timing behaviour, (c) Sequential architecture, (d) Parallel architecture

2. PROBLEM FORMULATION

Let us consider a hardware component based on a generic architecture composed of two main functional units: one memory unit *MU* and one processing unit *PU*. Suppose the computation processed to be *c = (a*var1)+(b*var2)* where *var1* and *var2* are variables values stored in memory. Fig. 1(a) shows the *Signal Flow Graph (SFG)* of this algorithm. This component receives input data *a* and *b* from the environment through *bus1* and sends its result *c* on *bus2*. All the data used and produced by the processing unit are respectively read and writen in a fixed order sequence *S =(a,b,c)*: i.e. $t_a < t_b < t_c$. The read sequence of the internal variable is completely deterministic i.e.: $t_{var1} < t_{var2}$. with $t_{var1} = t_a$ and $t_{var2} = t_b$. In this context, a single memory bank is therefore sufficient to satisfy the I/O timing requirement depicted in Fig. 1(b) where latency is equal to 2 cycles. Fig. 1 (c) presents a possible corresponding architecture of the component that includes 1 multiplier, 1 adder and 5 registers.

Let us now consider the following data transfer sequence $S_{busses} = (a \mid b, c)$: i.e. $t_a = t_b < t_c$. If the latency required to produce the result is long enough (≥ 3 cycles) to allow a reordering (serialization) of input data *a* and *b*, then the previously designed architecture including one memory bank can be reused. However, this solution need to design an input wrapper composed of 1 register, 1 multiplexer and 1controller. If the required latency is not long enough (i.e. = 2 cycles), the designer must design a new component including 2 multipliers, 1 adder, 7 registers and 2 memory banks (see Fig. 1(d)). has shown in this section, designers can use pre-designed component or macro/generator but it relies on a fixed architectural model with very restricted customization capabilities that can cause the system design to fail.

Hence, a new design flow based on synthesis under constraints is needed to get flexibility and ease the DSP component design. This includes (1) modeling styles to represent I/O timing and memory constraints, (2) analysis steps to check the feasibility of the constraints (3) methods and techniques for optimal synthesis.

3. DESIGN APPROACH OVERVIEW

The input of our HLS tool [14] is an algorithmic description that specifies the functionality disregarding implementation details. Fig. 2 presents an example of a FIR16 algorithmic description. This DSP specification is first compiled in order to obtain an intermediate representation: the Signal Flow Graph *(SFG* Fig. 3).

```
1. tmp := xn * H(N-1);
2. for i in 1 to N-1 loop
3.     tmp := tmp + x(i) * H(N-i-1);
4. end loop;
5. yn <= tmp;
6. for i in N-1 downto 2 loop
7.     x(i) := x(i-1);
8. end loop;
9. x(1) := xn;
```

Fig. 2: Fir16 algorithm example

3.1. Timing Constraint Graph

In a second step, we generate an Algorithmic Constraint Graph *ACG* from the operator latencies and the data dependencies expressed in the *SFG*. The latencies of the operators are assigned to operation vertices of the *ACG* during the operator's selection step of the behavioral synthesis flow.

Starting from the system description and its architecture model, the integrator, for each bus or port that connects the component to design to others system components, specifies I/O rates, data sequence orders and transfer timing information. We defined a formal model named *IOCG (IO Constraint Graph)* that supports the expression of integration constraints for each bus (port) that connects the component to the others in the system.

Finally we generate a Global Constraint Graph *(GCG)* by merging the *ACG* with the *IOCG* graph. Merging is done by mapping the vertices and associated constraints of *IOCG* onto the input and output vertices set of *ACG*. A minimum timing constraint on output vertices (earliest date for data transfer) of the *IOCG* are transformed into the *GCG* in maximum timing constraints (latest date for data computation/production).

After having described the behavior of the component and the design constraints in a formal model, we analyze the feasibility between the application rate and the data dependencies of the algorithm, in function of the technological constraints. We analyze the I/O timing specifications according to the algorithmic ones: we check if the required constraints on output data are always verified with the behavior specified for input data. The entry point of the IP core design task is the global constraint graph *GCG*.

3.2. Memory Constraint Graph

As outlined in the previous subsection, a Signal Flow Graph (*SFG*) is first generated from the algorithmic specification. In our approach, this *SFG* is parsed and a memory table is created. All data vertices are extracted from the *SFG* to construct the memory table. The designer can choose the data to be placed in memory and defines a memory mapping. For every memory in the memory table, we construct a weighted Memory Constraint Graph (*MCG*). It represents conflicts and scheduling possibilities between all nodes placed in this memory. The *MCG* is constructed from the *SFG* and the memory mapping file. It will be used during the scheduling step of the synthesis.

Fig. 5(b) shows a MCG for the presented example with one simple port memory bank. The variable data *var2* and *var1* are placed consecutively in one bank. Dotted edges represent sequential accesses (two adjacent memory addresses) and plain edges for random accesses (non adjacent addresses).

Further information about the formal models and the memory design can be found in [10], [12] and [13].

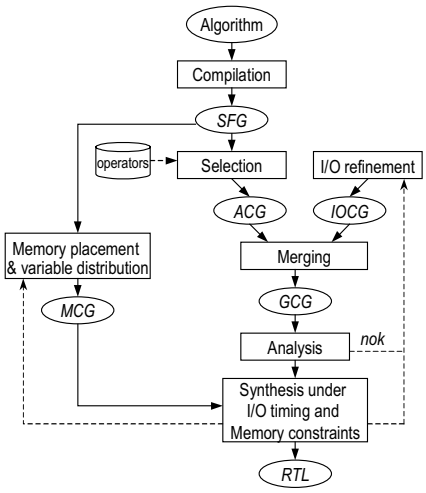


Fig. 3: Proposed Synthesis Flow

3.3. Scheduling under I/O and Memory Constraints

The classical "list scheduling" algorithm relies on heuristics in which ready operations (operations to be scheduled) are listed by priority order. In our tool, an early scheduling is performed on the *GCG*. In this scheduling, the priority function depends on the mobility criterion. The operation mobility is the difference between its ASAP and its ALAP. For operations that have the same mobility, the priority is defined using the operation margin. Operation margin is defined as the difference, in number of cycles, between the current cycle and the operation deadline. Operations are next scheduled and bind to operators (see Fig. 4).

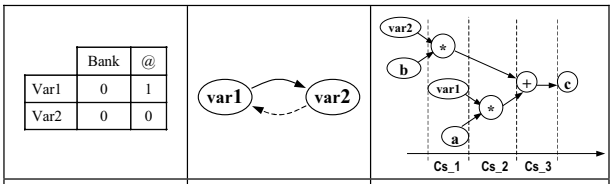
```
Scheduling_Function
1)    Operation_Mobility_computing(GCG)
2)    For (time = 0; time < End; time = time + t_cycle)
3)       List = Operation_Priority_listing(GCG)
4)       Ready_Ops = Find_schedulable_operation(List, time)
5)       Binding(Ready_Ops, operators_set, MCG, time)
6)    End for

Binding Function
1)    While (Ready_Ops!= NULL)
2)      Ops_low_mobility = Get_first(Ready_Ops)
3)    if(Op_low_mobility->margin > 0)
4)      If(Find_mem_conflic(MCG, Ops_low_mobility) = FALSE)
5)        If(operators_set != NULL)
6)          Ops_Binding(sh_list, operator)
7)        else //no opr or mem conflict
8)          Posponed(Ops_low_mobility)
9)    else // margin = 0
10)     If(Find_mem_conflic(MCG, Ops_low_mobility) = FALSE)
11)       Operator_cretation()
12)       Ops_Binding(sh_list, operator)
13)   else
14)       Exit(cycle, operator, operation, memory bank, …)
15)   end if
16)   End while
```

Fig. 4: Pseudo code of the scheduling algorithm

An operation can be scheduled if the current cycle is greater than the ASAP time. Whenever two ready operations need to access the same resource (this is a so-called resource conflict), the operation with the lower mobility has the highest priority and is scheduled. The other is postponed. When the mobility is equal to zero, one new operator is allocated to this operation. To perform a scheduling under memory constraint, we introduce memory access operators and add an accessibility criterion based on the *MCG*. A memory has as much access operators as access ports. The list of ready operations is still organised according to the mobility criterion, but all the operations that do not match the accessibility condition are removed from this list. Hence, when the mobility is equal to zero, the synthesis process exits and the designer have to target an alternative solution for the component architecture by reviewing the memory mapping and/or modifying some communication features.

Our scheduling technique is illustrated in Fig. 5 using the previously presented example where the timing constraints are now the following: $S = (a|b,c)$ i.e. $t_a = t_b < t_c$. The memory table Fig. 5(a) is extracted from the *SFG*. The designer has defined one memory mapping in memory table 1. Internal data *var1* and *var2* are respectively placed at address @1 and @0 in the bank0. Our tool constructs one Memory Constraint Graph *MCG* (Fig. 5(b)). In addition to the mapping constraint the designer also specifies two latency *Lat1=3 cycles* and *Lat2=2cycles*.

For the memory mapping and latency Lat1, the sequential access sequence is *var2* → *var1*: it includes one dotted edge (with weight Wseq) *var2* → *var1*. To deal with the memory bank access conflicts, we define a table of accesses for each port of a memory bank. In our example, the table has only one line for the single memory bank0. The table of memory access has Data_rate / Sequential_access_time elements. The value of each element of the table indicates if a fictive memory access operator is idle or not at the current time (control step c_step). We use the *MCG* to produce a scheduling that permits to access the memory in burst mode. If two operations have the same priority (margin = Lat1-T(+)-T(*) = 1 cycles) and request the same memory bank, the operation that is scheduled is the operation that involves an access at an address that follows the preceding access. For example, multiplication operation *(a*var1)* and *(b*var2)* have the same mobility. At c_step *cs_1*, they are both executable and the both operands *var1* and *var2* are stored in bank0. *MCG_1* indicates that the sequence *var2* → *var1* is shorter than *var1* → *var2*. We then schedule *(b*var2)* at c_step *cs_1* and *(a*var1)* at c_step *cs_2* to favour the sequential access (see Fig. 5(c)).



(a)Memory Table (b) MCG (c) Scheduling

Fig. 5: Scheduling under I/O timing and latency constraint

For the memory mapping and latency Lat2, multiplication operation *(a*var1)* and *(b*var2)* have the same mobility that is null. Both operations must then be scheduled in c_step *cs_1*. Because of the memory access conflict, there is no solution to the scheduling problem: the designer has hence to review its design constraints. He can target an alternative solution by adding one memory bank or by increasing the latency.

4. EXPERIMENTAL RESULTS

We described in the two previous sections our synthesis design flow and the scheduling under I/O timing and memory constraints. We present now the results of synthesis under constraints obtained using the HLS tool *GAUT* [14]. The algorithm used for this experience is a Fast Fourier Transform (FFT). This FFT reads 128 real input Xr(k) and produces the

output Y(k) composed of two parts: one real Yr(k) and one imaginary Yi(k). The *SFG* includes 16897 edges and 8451 vertices. Several syntheses have been realized using a 200MHz clock frequency and a technological library in which the multiplier latency is 2 cycles and the latency of the adder and the subtractor is 1 cycle.

4.1 Synthesis under I/O timing constraints

In this first experiment we synthesized the FFT component under I/O timing constraints and analyzed the requirements on memory banks. In order to generate a global constraint graph *GCG*, minimum and maximum timing constraints have been introduced between I/O vertices of the *ACG* graph using *IOCG* model. The FFT latency is defined by a maximum timing constraint between the first input and the first output vertices. The specified latency (that is the shortest one according to the data dependencies and the operator latencies) corresponds to a delay of 261 cycles. The FFT component is constrained to read one *Xr* sample and to produce one Y sample every cycle.
The resulting FFT component contains 20 multipliers, 8 adders and 10 subtractors (Table 1). 8 memory banks are required for those I/O timing constraints. *However, the internal coefficients are mapped in a non-linear scheme in memory. A large amount of memory bank is needed to get enough parallel access guaranteeing hence the specified latency. Moreover coefficient can be present in multiple banks what requires the design of a complex memory unit.*

Memory bank.	Input busses	Output busses	Sub.	Add.	Mult.	Latency *(in cycle)*
8	1	2	10	8	20	261

Table 1: Synthesis under I/O timing constraints

4.1. Synthesis under memory constraints

In this second experiment we synthesized a FFT component only under memory constraints. Nevertheless, only the maximal number of concurrent access to memory banks limits the minimal latency value. Hence, with a large amount of operators, a latency equal to the critical path delay of the SFG could be obtained. For the reason we synthesized the FFT component with the amount of operators presented in the first experiment. We then analyzed the requirement on I/O ports and computation latency. The memory constraints are: 2 memory banks respecting a simple mapping constraint: the 128 real coefficient Wr in bank0 and the 128 imaginary coefficient Wi in bank1.

The shortest latency imposed by the memory mapping and the amount of operators corresponds to a delay of 215 cycles (Table 2) what is shorter that those obtained in the previous experiment. This architecture requires 36 input busses (ports) and 14 output. *However, a large amount of busses which data orders are not trivial (non-linear data index progression) is needed. If the environment required the component to exchange data over few I/O busses, this requires the design of a communication unit. This communication unit can add extra latency to serialize data.*

Memory bank.	Input busses	Output busses	Sub.	Add.	Mult.	Latency *(in cycle)*
2	36	14	10	8	20	215

Table 2: Synthesis under memory constraints

4.3 Synthesis under I/O timing and memory constraints

In this last experiment we synthesized the FFT component under both I/O timing and memory constraints. We kept the memory mapping used for the second experiment and founded the shortest latency that allows to respect the I/O rates defined in the first experiment. The resulting architecture contains 17 multipliers, 8 adders and 10 subtractors (Table 3). It produces its first result after 343 cycles.

Memory bank.	Input busses	Output busses	Sub.	Add.	Mult.	Latency *(in cycle)*
2	1	1	10	8	17	343

Table 3: Synthesis under I/O timing and memory constraints

Because of both the memory mapping and the I/O constraints, the latency is greater than in experiment 1 and 2. However, the architecture complexity is equivalent to the previous ones in term of amount of operators. Hence, it appears that synthesis under both I/O timing and memory constraints allows to manage both the system's communication and memory, while keeping a reasonable architecture complexity.

5. CONCLUSION

In this paper, a design methodology for DSP component under I/O timing and memory constraints is presented. This approach, that relies on constraints modeling, constraints analysis, and synthesis, help the designer to efficiently implement complex applications. Experimental results in the DSP domain show the interest of the methodology and modeling that allow trade-offs between the latency, I/O rate and memory mapping.
We are currently working on heuristic rules that could help the designer in exploring more easily different architectural solutions, while considering memory mapping and I/O timing requirements.

Acknowledgements

These works have been realized within the French RNRT Project ALIPTA.

6. REFERENCES

[1] J. Ruiz-Amaya, and Al., " MATLAB/SIMULINK-Based High-Level Synthesis of Discrete-Time and Continuous-Time ΣΔ Modulators", *In Proc. of DATE 2004*.

[2] L. Reyneri, and al.. « A hardware/software co-design flow and IP library based on Simulink. », *In Proc. of DAC, 2001.*

[3] Codesimulink, http://polimage.polito.it/groups/codesimulink.html

[4] Mathworks, http://www.mathworks.com/

[5] H. Ly, D. Knappand al., « Scheduling using behavioral templates," in *Proc. Design Automation Conference DAC'95*, June 1995

[6] D. Knapp, and *al.*, « Behavioral synthesis methodology for HDL-based specification and validation," in *Proc. of DAC*, 1995.

[7] C. Gebotys, « Low energy memory and register allocation using network flow », in *Proc. of DAC*, 1997

[8] R. Saied and Al. « Scheduling for minimizing the number of memory accesses in low power applications », in *Proc. VLSI Signal Processing*,. 1996,

[9] N. Passos, and al, « Multi-dimensional interleaving for time-and-memory design optimization », in *Proc. of ICCD, 1995*

[10] A. Nicolau and S. Novack, « Trailblazing a hierarchical approach to percolation scheduling," in *Proc. ICPP'93*, 1993,

[11] P. Coussy, A. Baganne, E. Martin, " Communication and Timing Constraints Analysis for IP Design and Integration*", In Proc. of IFIP WG 10.5 VLSI-SOC Conference, 2003*.

[12] P. Coussy , D. Gnaedig, and al., " A Methodology for IP Integration into DSP SoC: A Case Study of a MAP Algorithm for Turbo Decoder", *In Proc. of ICASSP, 2004*

[13] G. Corre, E. Senn, and al., "Memory accesses Management During High Level Synthesis", *In Proc. of CODES-ISSS, 2004.*

[14] GAUT - HLS Tool for DSP, *http://lester.univ-ubs.fr:8080/*