Statically-scheduled application-specific processor design: a case-study on MMSE MIMO equalization

Mostafa Rizk[†‡], Amer Baghdadi[†], Michel Jézéquel[†], Yasser Mohana[‡], Youssef Atat[‡]

† Telecom Bretagne; UMR CNRS 3192 Lab-STICC; Electronics Department; Brest, France

‡ Lebanese University; Physics and Electronics Department; Hadath, Lebanon

{mostafa.rizk, amer.baghdadi, Michel.jezequel}@telecom-bretagne.eu; {yamoha, youssef.atat}@ul.edu.lb

***Abstract*— Many application-specific processor design approaches are being proposed and investigated nowadays. All of them aim to cope with the emerging flexibility requirement combined with the best performance efficiency. Application Specific Instruction-set Processor (ASIP) design approach is among the most explored, and thus in many application domains. However, this concept implies a dynamic scheduling of a set of instructions which generally lead to an overhead related to instruction decoding. To reduce this overhead, other approaches were proposed using static scheduling of datapath control signals. In this paper, we explore this last approach and illustrate its benefits through a design case-study on MMSE MIMO equalization. The proposed design has common main architectural choices as a state-of-the-art ASIP for comparison purpose. The obtained results illustrate a significant improvement in execution time while using identical computational resources and supporting same flexibility parameters.**

I. INTRODUCTION

Advanced computer architectures for application-specific processor target the accommodation of the emerging flexibility requirement as well as attaining the best performance efficiency. Such combination of flexibility and the ever increasing performance requirements demands design approach that provides better ways of controlling and managing the hardware resources. Low level design at Register Transfer Level (RTL) can lead to efficient architectures but the development time is very high for complex applications. High Level Synthesis (HLS) increases productivity by converting directly high level C language description into an RTL Hardware Description Language (HDL). The designer cannot correlate precisely the effect of application modifications to final implementation quality metrics such as area, power, clock frequency, routable layout, etc. [1]. To improve the quality, the designer can depend only on guess and try work. The result quality is noticeably low compared to manual RTL. A suitable approach to design custom processors is based on Application Specific Instruction-set Processor (ASIP) concept. It offers a compromise in terms of design productivity and implementation quality. ASIP relies on a few set of pre-defined custom instructions. An instruction decoder should be designed to decode the instructions that are then executed by the corresponding hardware at runtime. The implementation of the instruction decoder leads to a complex controller which increases power and area consumption.

Recently, the idea of a processor dedicated to an application not using an instruction set has been introduced under the name of No-Instruction-Set-Computer (NISC). The main proposal of NISC approach is that there exists no need to use an instruction set when the hardware is programmed by its designers and not by its users. NISC simplifies ASIP approach by removing the complex task of finding and designing "most profitable" custom instructions [1]. The elimination of the instruction set increases the designer productivity and shrinks the time-to-market. The hardware is simplified due to the omitting of instruction decoder what reduces the complexity and improves the performance. All major tasks of typical processor controller (instruction decoding, dependency analysis, instruction scheduling, etc.) are done by the compiler statically [1] at compilation time. The compiler, which is not restricted by die size, chip resources or timing constrains, generates the control words (CWs) that must be applied to datapath components at runtime in every clock cycle and loads them in a control memory. At run time, the controller only loads the CWs and applies them to the datapath.

In this paper, we explore and illustrate the benefits of the NISC approach in design an application-specific processor dedicated to MMSE MIMO equalization. The proposed design is in addition compared with a state-of-the-art ASIP.

The rest of the paper is organized as follows. Next section explains the used designed approach. Section 3 presents the application case-study showing the algorithm details and the target flexibility requirement. Sections 4 and 5 present the proposed architecture and the simulation results respectively. The last section concludes the paper and gives the future work.

II. DESIGN APPROACH

The NISC design approach offers an open source toolset [2] that can be used either as a free C-to-RTL (i.e. C to Verilog) synthesis tool or to design embedded custom-processors. The designer needs only to specify the datapath and the custom-functional units and then uses the toolset to compile the application C code on the devised architecture. Results can be refined and improved by modifying the application C code or the datapath and reusing the toolset to generate new results. Structural details of the architecture are captured by Generic Netlist Representation (GNR), which is a formal architecture description language (ADL) [3]. The key feature of GNR is that the structural information is enhanced with *types* and tool-specific information called *aspects*. *Types* and *aspects* are used in validation, datapath connection, optimization, compilation and implementation. GNR uses the eXtensible Markup Language (XML) to describe models. GNR syntax is defined in XML Schema to enforce syntax and semantics checking on the given input model [4]. The datapath is captured in GNR [5] that describes the *components*, *ports*, *connections* and *aspects*. The component type can be a basic RTL component (register, multiplexer, functional unit, etc.) or a module, which is a hierarchical component that can have an internal netlist. Each port is parameterized by its *bitwidth* and type (Clock "*clock*", control port "*ctrlPort*", input "*inPort*", output "*outPort*", and control word port "*cwPort*"). *Connections* link in between

978-3-9815370-0-0/DATE13/©2013 EDAA

source ports (*outPort*) and destination ports (*inPort*) of different components. *Aspects* describe the behavior of the component for different tools in the toolset [5]. According to the component type the *compilation aspect* defines one or more machine actions (MA), which are very low-level functionalities of the component that determines both the timing and the control values of each control port at each operation. *Synthesis* and *simulation aspects* contain HDL information of the component. The functionality of the component can be described by its *compilation aspect*, and by netlist specifications or HDL description. For an example, Fig.1 shows the GNR description of a multiplexer that can implement two operations as shown in its *compilation aspect*; hence its control port *"sel"* is 1-bit wide. The multiplexer is of type "*Mux*" and has two parameters: *BIT_WIDTH* and *DELAY*. The HDL description of the multiplexer is input through the file "*Mux2.v*".

In some applications, the hardware must be controlled directly through specific instructions. The toolset provides pre-bound functions and variables that have common C syntax. The compiler maps them to specific hardware resources [6]. For a specific module, the designer should declare pre-bound functions in the *compiler aspect* of the module. The compiler generates proper control bits to access their corresponding hardware resources. The declaration should define the control values for enabling the function and the ports that are used as inputs and outputs. Figure 2 shows an example of a pre-bound function *write* that controls loading data to a register. Both input port "*i*" and control value of the control port "*load*" are specified. The pre-bound functions have no implementation and are treated similarly to other operations. Therefore, they can be scheduled in parallel with other operations [6].

High performance designs are achieved by having direct control of hardware resources. Direct compiling of C code descripting complex application using NISC gave inefficient hardware results. To achieve high performance, we described manually all the control signals for each clock cycle. Pre-bound functions were used to have direct control of hardware resources using the toolset. To increase design productivity, we exploit the automatic completion of the GNR to reduce the datapath description. And we made use of syntax checking and rule validation provided by the toolset to quickly detect and fix errors. Also we used the toolset compiler to schedule statically and to arrange automatically the control signals in memory.

III. MMSE MIMO EQUALIZATION APPLICATION

Multi-Input Multi-Output (MIMO) techniques with multiple antennas at transmission and reception sides proliferates in wireless communication systems. Turbo-equalization concept [7] can be used at the MIMO receiver to cancel the effects of MIMO co-antenna interference. When used in an iterative scheme, the Minimum Mean Square Error Interference Cancellation (MMSE-IC) algorithm implements MIMO equalization with an acceptable tradeoff between complexity and performance. The variety of transmission standards and environments imposes the requirement of architecture flexibility in emerging wireless communication applications to accommodate different algorithmic variants and to follow the market pressure. All the components of a MIMO receiver should support different system configurations concerning the time selectivity of the channel (block, quasi-static and fast fading) and different techniques of transmission diversity (2×2, 3×3 and 4×4 space-time coding).

The input vector of the MIMO turbo receiver shown in Fig.3 is given by the following expression:

```
<Mux type="Mux2">
 <Params>
  <Param n="BIT_WIDTH"/>
  <Param n="DELAY" val="0"/>
 </Params>
 <Ports>
  <InPort n="i0" bitWidth="{@BIT_WIDTH}"/>
  <InPort n="i1" bitWidth="{@BIT_WIDTH}"/>
  <CtrlPort n="sel" bitWidth="1" default="x"/>
  <OutPort n="o" bitWidth="{@BIT_WIDTH}"/>
 </Ports>
 <Annot_verilog>
  <Synthesis topModuleName="Mux2">
   <VerilogParams>
    <Param n="BIT_WIDTH" val="{@BIT_WIDTH}"/>
   </VerilogParams>
   <VerilogCode>
    <File n="Mux2.v"/>
   </VerilogCode>
```

```
  <Simulation topModuleName="Mux2">
   <VerilogParams>
    <Param n="BIT_WIDTH" val="{@BIT_WIDTH}"/>
   </VerilogParams>
   <VerilogCode>
    <File n="Mux2.v"/>
   </VerilogCode>
  </Simulation>
 </Annot_verilog>
 <Annot_compiler>
  <Transfers>
   <DataTransfer inPort="i0"outPort=o transferDelay="{@DELAY}>
    <Ctrl val="0" port="sel"/>
   </DataTransfer>
   <DataTransfer inPort="i1" outPort=o transferDelay="{@DELAY}">
    <Ctrl val="1" port="sel"/>
   </DataTransfer>
  </Transfers>
 </Annot_compiler>
</Mux>
```

Fig. 1. GNR description of a multiplexor

```
<Function n="write"stateDependency="none"stages="1"delay="0"setupTime=0 holdTime="0">
 <Input port="i"/>
 <Ctrl port="load" val="1"/>
</Function>
```

Fig. 2. Pre-bound function used for loading data to a register

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{w} \tag{1}$$

where $\mathbf{y}$ is vector of size of number of receiver antennas (N_r), $\mathbf{x}$ is a vector of size of number of transmitter antennas (N_t), $\mathbf{H}$ is the MIMO channel matrix of size $N_r \times N_t$ and $\mathbf{w}$ is a vector of Additive White Gaussian Noise (AWGN) of size N_r.

The MMSE-IC Linear Equalizer (LE) removes the co-antenna interference and provides the estimated symbol vector $\tilde{\mathbf{x}}$ of size N_t and the corresponding bais vector $\mathbf{g}$. Along the feedback path, the decoder provides *a posteriori* information to a soft mapper that provides the *a priori* information to the equalizer as decoded symbol vector $\hat{\mathbf{x}}$ of size N_t. The equalizer considers that a symbol of the vector $\mathbf{x}$ is distorted by the $N_t - 1$ other symbols of the vector and the noise channel and tries to combat both. Equation 1 can be written in the following form:

$$\mathbf{y} = \mathbf{h_j}.\mathbf{x_j} + \sum_{\mathbf{i}\neq\mathbf{j}} \mathbf{h_i}.\mathbf{x_i} + \mathbf{w}\,, \mathbf{j} \in \{\mathbf{0}, \mathbf{Nt} - \mathbf{1}\} \tag{2}$$

where h_i and $\mathbf{h}_j$ are the i^{th} and j^{th} column of $\mathbf{H}$ matrix.

Using the Wiener filter $\mathbf{a}_j^H = \lambda_j.\mathbf{P}_j^H$, the estimation of $\mathbf{x}$ is given by:

$$\tilde{\mathbf{x}}_j = \lambda_j.\mathbf{P}_j^H\,(\mathbf{y} - \mathbf{H}\hat{\mathbf{x}} + \mathbf{h_j}\hat{\mathbf{x}}_j) \tag{3}$$

where $j \in \{0, N_t - 1\}$, $\hat{\mathbf{x}}_j$ is the j^{th} element of **vector** $\hat{\mathbf{x}}$, $\mathbf{h}_j$ is the j^{th} column of $\mathbf{H}$ matrix and $(.)^H$ is the Hermitian operator. $\mathbf{P}_j$ and λ_j are defined as:

$$\mathbf{P_j} = \mathbf{E^{-1}h_j} \text{ where } \mathbf{E} = \left(\sigma_x^2 - \sigma_{\hat{x}}^2\right)\mathbf{HH^H} + \sigma_w^2\mathbf{I} \tag{4}$$

and σ_x^2, $\sigma_{\hat{x}}^2$ and σ_w^2 are variances of transmitted symbols, decoded symbols and noise. $\mathbf{I}$ is identity matrix of size $N_r \times N_r$.

$$\lambda_j = \frac{\sigma_x^2}{1+\sigma_{\hat{x}}^2\beta_j} \text{ where } \beta_j = \mathbf{P}_j^H\mathbf{h}_j \tag{5}$$

Equation 3 can be written as:

$$\tilde{\mathbf{x}}_j = \lambda_j.\mathbf{P}_j^H(\mathbf{y} - \mathbf{H}\hat{\mathbf{x}}) + \mathbf{g_j}\hat{\mathbf{x}}_j \text{ where } \mathbf{g}_j = \lambda_j.\beta_j \tag{6}$$

The computation of $\mathbf{E}, \mathbf{P}, \beta, \lambda$ and $\mathbf{g}$ is performed one time for a whole block of symbols for which channel is considered as constant whereas $\tilde{\mathbf{x}}$ is computed repeatedly for each equalization iteration.

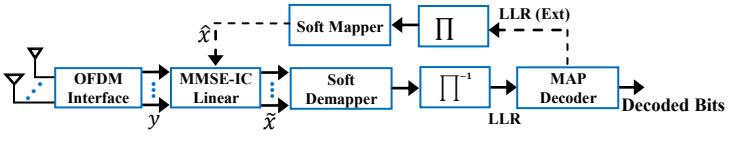

Fig. 3. MIMO turbo receiver scheme

IV. MMSE-IC Linear Equalizer NISC Architecture

One can note that the computation of coefficients and symbol estimation have common arithmetic operations, however they execute at different time. Allocating separate resources for each task will result in an inefficient architecture in case of quasi-static and block-fading channel. Sharing hardware resources between the two tasks ensures efficiency and flexibility related to time selectivity of the channel. Regarding the flexibility requirement of transmission diversity, the allocation of hardware resources according to the most complex configuration will result in inefficient architecture for low complex ones. To meet the requested requirements, complex matrix operations are decomposed into basic real arithmetic operations. In the following we introduce the proposed hardware resources capable of performing complex operations using basic real arithmetic operators.

A. Complex number operations

Addition and subtraction/negation of two complex numbers require two adders and subtractors respectively whereas the conjugation is completed using only one subtractor. Using the Eq. 7 complex number multiplication needs three adders, two subtractors and three real multipliers.

$$(a+bj)(c+dj) = a(c+d)-d(a+b)+j\{a(c+d)+c(b-a)\} \quad (7)$$

NISC_CCASM shown in Fig.4 is a NISC module that has similar architecture as the combined complex adder, subtractor and multiplier (CCASM) [8]. It can perform all complex addition, subtraction, negation and conjugation. Equation 8 shows the inversion of a complex number. It can be obtained by using resources in *NISC_CCASM* in addition to a pre-computed lookup table (LUT) to retrieve the inverse of a^2+b^2.

$$\frac{1}{a+bj} = \frac{a}{a^2+b^2} - j\frac{b}{a^2+b^2} \quad (8)$$

B. Complex Matrix operations

Using four instances of *NISC_CCASM,* the addition, subtraction, negation, and hermitian of complex matrix can be performed with efficient resource utilization. Multiplication of two 2×2 matrices requires four additional adders to sum up the multiplication result. For 3×3 and 4×4 matrices, the multiplication of two matrices requires six additional adders. Matrix inversion is performed using the analytic method. The 2×2 matrix inversion is given by:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-cb}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \quad (9)$$

4×4 matrices are divided into four 2×2 matrices and inverted

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} W & X \\ Y & Z \end{bmatrix} \quad (10)$$

where

$W = A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1}$ $\quad X = -A^{-1}B(D - CA^{-1}B)^{-1}$

$Y = -(D - CA^{-1}B)^{-1}CA^{-1}$ $\quad Z = (D - CA^{-1}B)^{-1}$

C. Fixed point representation

All numbers used in the computation have a fixed point representation. Using 16-bit signed representation with different bits for integer and fractional part in different computation steps ensures low performance loss for all supported configurations and enables the reuse of hardware resources. The needed precision for each arithmetic, load or store operation was achieved via long simulations [8].All used adders and subtractors are capable to detect overflow/underflow occurrence and to fix the output at its maximum/minimum values. Fixed point converter module was established to perform the conversion at the output of each multiplier. Each converter module selects the required bits for the integer and fractional parts, and detects the occurrence of overflow/underflow.

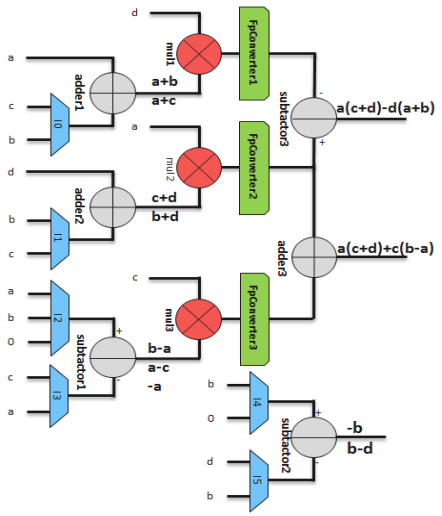

Fig. 4. NISC_CCASM block diagram

D. Architecture resources

The proposed architecture is presented in Fig. 5. It includes three memory blocks, a control unit, and the MMSE-IC LE module. *Cmem* memory block stores the control words generated by NISC compiler. *Chmem* memory block saves the constant data of the channel. $\frac{1}{x}$ *LUT* memory block contains pre-computed inverse values used in complex number inversion. The control unit is simple and only derives control signals at runtime. The equalizer module called *EquaNISC* shown in Fig. 6 is composed of three main units:

1) Storage unit that contains three groups of 16-bit registers. Each pair of registers is proposed to store a complex number, one storing the real part and the other the imaginary part. Each group can store one 4×4 complex matrix. The three groups of registers save data loaded from memory blocks or results of intermediate computations. In addition to the register groups, four registers are instantiated to store σ_x^2, $\sigma_{\hat{x}}^2$, σ_w^2 and $\sigma_x^2 - \sigma_{\hat{x}}^2$.

2) Computational unit that contains all resources that perform the computation operations. Four *NISC_CCASM* and three *Complex Adders* (CA) form the core of the computational unit. Each *NISC_CCASM* includes six multiplexers, three real adders, three real subtractors, three signed multipliers and three fixed point converters. Each complex adder is simply composed of two real adders.

3) Multiplexing unit that arranges all data transfers in between storage unit, computational unit and memory blocks. It is composed of several multiplexers that construct a chain between different components of the proposed architecture.

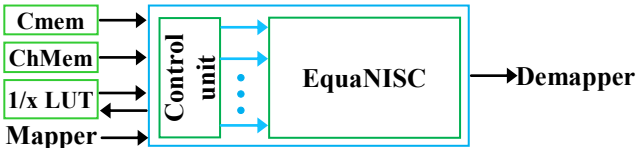

Fig. 5. Block diagram of proposed architecture

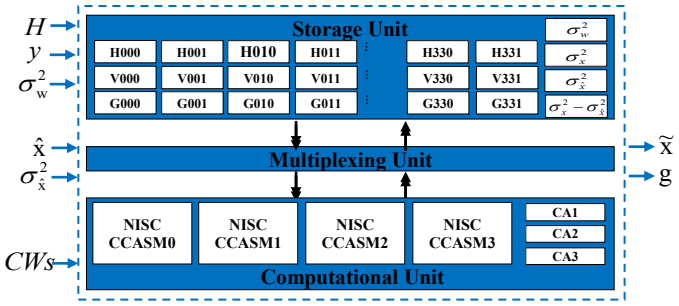

Fig. 6. *EquaNISC* block diagram

E. Architecture design

In our architecture, all basic components such as multiplexers, adders, subtractors, registers, shift registers, multipliers, fixed point converters, and memory blocks were described by their HDL description (i.e. Verilog). Hierarchical modules such as *NISC_CCASM* and *EquaNISC* units described using netlist description using GNR.

Pre-bound functions were used to achieve efficient utilization of resources and accurate execution timing. Different computational operations that can be executed in the same clock cycle were merged to the same pre-bound function to maximize the exploitation of hardware resources and time. Each pre-bound function describes all required control values of all resources used to perform the operation(s) in one clock cycle. Figure 7 shows a sample pre-bound function that loads data from $\frac{1}{x}$ *LUT* to register H000 in the storage unit. The control values of the load port of the register and the selection port of the multiplexer, connected to the input of the register, are specified explicitly.

V. Simulation Results

The used toolset generates the final HDL description of the architecture in addition to the memory block. The simulation results show a good performance in computing the equalizer coefficients. Table 1 shows the number of clock cycles required for the computation of the coefficients using our proposed design. The results are in addition compared with that of *EquASIP*, which is an ASIP with a custom instruction set dedicated for MMSE-IC equalization [8]. Both processors use identical computational resources and support same flexibility parameters. The comparison between the two designs shows significant improvement in terms of execution time: 34% for MIMO 4×4 and 33% for MIMO 2×2. In *EquASIP,* the devised instruction set and pipeline structure can lead for data dependency issue. For cases where the current instruction needs the results computed by a previous instruction which still

```
<Function n="LoadLUTtoH000 stateDependency=all stages=1 delay=0"setupTime=0"holdTime=0>
 <Ctrl port="Eq_Mux_iH000_sel" val="0111"/>
 <Ctrl port="Eq_H000_load" val="1"/>
</Function>
```

Fig. 7. Pre-bound function that loads data from $\frac{1}{x}$ LUT to the register H000

TABLE I. MMSE-IC LE Execution Time Comparison

Expression	MIMO 2×2 (cycles)		MIMO 4×4 (cycles)	
	Proposed design	EquASIP [8]	Proposed design	*EquASIP* [8]
E (Eq.4)	14	18	39	50
E^{-1} (Eq.4)	10	14	48	68
P_j (Eq.4)	2	12	16	39
β_j (Eq.5)	4	7	17	27
λ_j (Eq.5)	12	23	14	23
$\lambda_j . P_j^H, g_j$ (Eq.5 , Eq.6)	4	7	7	14

under execution in the pipeline, one or more No Operation (NOP) instructions are added. This scheme induces additional execution time overhead. On the other hand, the proposed *EquaNISC* design implements directly the control signals on hardware resources and executes each operation in one clock cycle. Also the merging of different independent operations into one pre-bound function, which executes in one clock cycle, reduces the execution time.

VI. Conclusion

In this paper, we have presented a flexible application-specific processor dedicated to MIMO MMSE-IC LE (*EquaNISC*). The use of NISC design flow ensures promising design quality and productivity. The direct controlling of resources in *EquaNISC* ensures less execution time compared to *EquASIP*. Sharing and reusing resources provide efficient hardware utilization through all required computations for different system configurations.

In addition to these results, the use of simple controller architecture (without instruction decoder) should lead to a significant gain in area and power consumption. Synthesis results and related measures and evaluations are the objectives of our current research work on this topic.

References

[1] M. Reshadi, "No-Instruction-Set-Computer Technology Modeling and Compilation," Thesis desiration 2007.

[2] NISC toolset website: http://www.ics.uci.edu/~nisc/.

[3] B. Gorjiara, M. Reshadi, D. Gajski, "GNR: A Formal Language for Specification, Compilation, and Synthesis of Custom Embedded Processors," in *Processor Description Languages: Applications and Methodologies*., 2008, ch. 13.

[4] B. Gorjiara, M. Reshadi, P. Chandraiah, D. Gajski, "Generic Netlist Representation for System and PE Level Design Exploration," in *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS).*, 2006.

[5] B. Gorjiara, M. Reshadi, D. Gajski, "Generic Architecture Description for Retargetable Compilation and Synthesis of Application-Specific Pipelined IPs," in *International Conference on Computer Design (ICCD)*, 2006.

[6] M. Reshadi, D. Gajski, "Interrupt and Low-level Programming Support for Expanding the Application Domain of Statically-scheduled Horizontally-microcoded Architectures in Embedded Systems," in *Design Automation and Test in Europe (DATE)*, 2007.

[7] C. Douillard, M. Jézéquel, C. Berrou, A. Picart, P. Didier, and A. Glavieux, "Iterative correction of inter symbol interference: Turbo equalization," *European Trans. on Telecom*, vol. 6, no. 5, pp. 507–511, Sept-Oct 1995.

[8] A.R. Jafri, "Multi-ASIP Architectures for Flexible Turbo Receiver," Electronics, Telecom Bretagne, Brest, PhD dissertation 2011.