Automating the Design of an Asynchronous DLX Microprocessor

Manish Amde
Indian Institute of Technology
Bombay, India
manish@ee.iitb.ac.in

Ivan Blunno
Politecnico di Torino
Torino, Italy
blunno@polito.it

Christos P. Sotiriou
FORTH
Heraklion, Greece
sotiriou@ics.forth.gr

ABSTRACT

In this paper the automated design of an asynchronous DLX microprocessor is presented. The microprocessor has been designed beginning with a standard RTL-like *Verilog* specification and the *Pipefitter* design flow has been used to automatically generate both the specification for the direct implementation of the Control Unit and a synthesisable Verilog specification of the Data Path. The architecture of the DLX is locally synchronous and globally asynchronous and the delay elements for the generation of the local clock signal are automatically produced by Pipefitter as well.

The following steps of the design flows (i.e., logic synthesis, technology mapping, placement and routing) have been completed using standard tools leading to the final layout of the circuit.

The final microprocessor implements all the functionality of a standard DLX (with the exception of the floating point unit) and supports its whole set of instructions.

Some considerations on the area occupation of the microcontroller will be presented in the last section of this paper.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD)

General Terms

Design

Keywords

Asynchronous, DLX, design flow

1. INTRODUCTION

The technological evolution of microelectronics in the last decade has been the key enabler for the transition from System-On-a-Board (SOB) design to System-On-a-Chip (SOC) design. SOCs seem to be currently the technology better suited to satisfy the demands for low-power, high-performance and low-cost as imposed by the market. Therefore, components which used to be packaged and sold as a final product in order to be placed on a board, such as microcontrollers, DSPs and FPGAs, are now becoming the basic components of much more complicated systems and can be fitted on the same die. This new trend has posed the basis for the fast spread of reusable Intellectual Property (IP) cores which offer the big advantage of speeding up the design flow and therefore reducing time-to-market and decreasing costs [6].

However, integrating several cores inside the same system is in general a hard problem to solve, due to the presence of many clock domains. Each synchronous core is generally designed according to a different timing specifications, *i.e.* clock period and minimum and maximum clock skew. Thus, due to these timing assumptions communication between cores becomes a problem. In addition, the increasing delay of interconnect in current process technologies is becoming comparable to the transistors switching speeds and keeping the clock skew low over the ever-increasing increasing chip area is becoming harder and harder.

In contemporary SOCs, synchronous systems also present the following disadvantages:

- high power-consumption as the entire system, including the clock tree, consumes power even when some clocked parts are not involved in any computation;
- high Electro-Magnetic Interference (EMI), since the simultaneous switching of logic gates results in narrow current pulses and voltage glitches on both the power supply and ground pins [5]. These cause high-noise peaks at the high end of the frequency spectrum. The use of high-performance, small dimension processes for economic reasons, makes this problem relevant even for low-frequency clock circuits.

Asynchronous circuits, on the other hand, seem to be better suited for IP design and integration in SOC environments because of their well known characteristics. The replacement of the global clock signal by a local communication and synchronization protocol presents a number of advantages:

- only circuitry actually involved in computation consumes power, while other parts of the chip remain in a “stand-by” state;
- self-timed devices switch in a spread timing interval, compared to their synchronous counterparts, thus presenting small amplitude and wide current peaks which produce significanlty less high-frequency EMI components;
- asynchronous circuits easily and naturally adapt to the speed of the environment in which they operate;
- an asynchronous circuits timing constraints do impact on its performance and on the performance of a system, but do not

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2003, June 2–6, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-688-9/03/0006 ...$5.00.

propagate into global timing constraints. Thus timing constraints are much easier to satisfy and the system more scalable.

However, the advantages reported above do not seem to be sufficient to justify the considerable effort required to re-train a designer to the use of new methodologies, tools and languages often based on low levels of abstraction, unless this is really essential as in the case of a state-of-the-art microprocessor.

The use of an asynchronous design flow based on a standard HDL, tools and cells seems to be the only way to reduce the inertia of the industrial environment to asynchronous design. Design flows like Tangram developed by Philips Research [2, 11], Balsa developed at the University of Manchester [1] and the one used at Theseus Logic [13] have proved to be effective in designing academic and industrial asynchronous circuits [8, 16, 14]. Nevertheless, the proprietary or non-standard nature of these design flows has been a major obstacle to their widespread adoption.

The design flow we based our design on is the *Pipefitter* tool [4] developed at the Politecnico di Torino.

The design of the asynchronous DLX [9] microprocessor that we present in this paper has the twofold goal to show that:

- the design of asynchronous circuits can be much easier when supported by a standard language based design flow,
- asynchronous circuits can provide a valuable contribution in the field of reusable cores.

This paper is organized as follows. Section 2 gives a quick overview of the Verilog subset supported by Pipefitter. In sections 3 and 4 the global architecture and the specification of the asynchronous DLX will be presented and some key point of its implementation will be discussed. Section 5 describes the synthesis process. In section 6 the performance of our microprocessor will be examined and a comparison with its synchronous counterpart will be carried out. Directions and goals for the future will be also pointed out.

2. THE VERILOG SPECIFICATION

The subset of the Verilog language supported by Pipefitter is described in details in [4]. However, a brief description of the basic statements and structures supported will help to better understand the specification fragments in the following sections.

In each Verilog file only one module can be specified. Each module can be considered divided into three main sections: the *prologue*, the *initial block* and the *always block*.

In the prologue the I/O ports of the module are listed. For each port, the nature (`input` or `output`) and the size must be specified. In this section is also possible to specify internal registers (`reg`).

The `initial` block is used to specify a set of operations that will be executed only once in the beginning. Usually this section is used for specifying the reset sequence. This block is not mandatory and can be missing in a file.

The `always` block is the main loop that describes the cyclic behavior of the module and it must be present in each specification.

Blocks can be either *sequential* (included between `begin-end` statements) or *concurrent* (included between `fork-join` statements) and can recursively contain sequential and concurrent blocks. So it is perfectly legal to specify a sequence of blocks some of which are concurrent blocks containing sequential and concurrent blocks.

Inside sequential and concurrent blocks it is possible to specify assignments using the syntax `target = expression`. The target of an assignment must be either an internal register or an output.

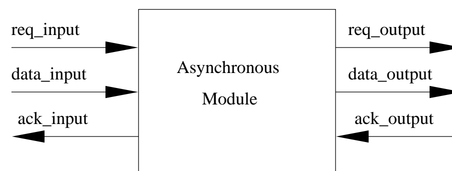

Figure 1: Handshake protocol

The expression can be an input, an internal register or an arithmetic/logic expression.

There are also some control-flow statements. The `wait` statement allows to stop the execution of the program until a specified signal doesn't switch to a given value. The `if-else` and `case` structures allow the user to specify choices.

Combining assignments and `wait` statements it is possible to specify handshakes as summarized by the following example:

```
wait(request_in);
   // Perform some operation
acknowledge_out = 1;
wait(!request_in);
acknowledge_out = 0;
```

3. INTRODUCING THE ADLX

The basic model of the microprocessor we have implemented is similar to the one presented in [9]. This microprocessor is a 5-stage pipelined processor called DLX. The DLX is a synchronous processor and each stage of the pipeline completes it's computation in one clock period. We have restyled the DLX in such a way that the basic data transfer between different stages remain the same but they take place by means of handshake signals between the stages rather than on a global clock active edge.

Our restyled version of the DLX has been called Asynchronous-DLX (ADLX). The data transfer in ADLX is synchronized by a 4-phase handshake protocol.

The clock cycle of the DLX has to to be greater than the slowest stage in the pipeline, hence the performance of the DLX are dictated by the worst-case delay of the whole architecture. As a consequence both latency and throughput can be strongly penalized. On the other hand, in the ADLX the data transfer between two stages is not dependent on a clock edge and can begin as soon as the previous 4-phase handshake for the given data has taken place. Such an architecture also allows for skipping stages when no computation has to take place allowing for better performances. Also some instructions could be executed faster than others due to different memory and ALU requirements and therefore the time for execution of different instructions will automatically adapt to the specific case. The ADLX promises to give better performances in terms of speed as it tends to give an average-case performance rather than a worst-case.

The ADLX is also free from the problem of clock distribution to the entire circuit like the DLX but has the extra area overhead of control circuitry to implement the 4-phase handshake protocols and the delay elements.

The basic 4-phase protocol for implementing the handshake is based on asynchronous channels as shown in figure 1. Each data (or group of data) must be accompanied by two wires: one for the request signal and the other for the acknowledge. The Verilog specification for the data transfer shown in figure 1 is as follows.

```
   wait(req_input) ;
```

```
data_output = data_input ;
fork
   begin
      ack_input = 1 ;
      wait(!req_input) ;
      ack_input = 0 ;
   end
   begin
      req_output = 1 ;
      wait(ack_output) ;
      req_output = 0 ;
      wait(!ack_output) ;
   end
join
```

The concurrency of the input and output handshakes and there presence in only separate parallel branches makes the input and output stages semi-decoupled [7] from each other and frees the input or output stage as soon as they have terminated their handshakes.

4. THE ADLX ARCHITECTURE

The Data Path of the ADLX is similar to that of the DLX as shown in figure 2 but is based on channel communications rather than data communications, as described in section 3.

The 5 stages of the pipeline are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), Write-Back (WB). Despite the similarity between the global architecture of synchronous and asynchronous implementations of the DLX, problems like control and data hazards had to be handled with a particular care as described in sections 4.1 and 4.2.

Only the Core of the DLX was designed meanwhile all the memory structures (i.e., instruction memory, register file and data memory) were modeled as Verilog behavioral unit for testing and debugging purposes. This opportunity showed another major advantage of using a standard language based methodology versus a special purpose language based methodology. In the former case it is indeed possible to simulate together modules which have been specified at different levels of abstraction (e.g., gate level for the Core and behavioral level for the memories modules).

The following example of fragment of Verilog behavioral specification (coming from the IF stage) and the corresponding synthesisable specification automatically generated by Pipefitter can be useful in understanding how Pipefitter works.

The fragment

```
outdata_Add = outdata_MUX_1 + 4;
```

specifies an add operation between the register `outdata_MUX_1` and the constant 4. The result must then be loaded into the register `outdata_Add`. For such a specification Pipefitter will generate a fragment of Control Unit, and some modules in the Data Path. Apart from the trivial constant element, two modules will be specified for the Data Path: an Operative Unit for the add operation and a register where the result must be stored. The synthesisable specification is listed below for both the modules.

```
module _IF_OU_0(ck, input_1_0, input_2_0, output_0);
   input ck;
   input [31:0] input_1_0;
   input [31:0] input_2_0;
   output [31:0] output_0;
   reg [31:0] output_0;

   always @(posedge ck)
      output_0 = input_1_0 + input_2_0;
endmodule

module _IF_reg_outdata_Add(ck, en, input_0,
output_0);

   input ck;
   input en;
   input [31:0] input_0;
   output [31:0] output_0;
   reg [31:0] output_0;

   always @(posedge ck) if(en) output_0 = input_0;
endmodule
```

These two modules will also have to be instantiated inside the global Data Path module as follows:

```
   ...
   _IF_OU_0 _INST_6 (_OU_0_ck_, _reg_outdata_MUX_1,
      {29'b00000000000000000000000000000,
      _constant_4}, _OU_0, _OU_0_out_);
   ...
   _IF_reg_outdata_Add _INST_8 (_reg_outdata_Add_ck_,
      _reg_outdata_Add_EN_, _OU_0, _reg_outdata_Add);
   ...
```

The signals `_reg_outdata_Add_ck_`, `_reg_outdata_Add_EN_` and `_OU_0_ck_` which come from the Control Unit are used to synchronize this fragment of Data Path with the rest of the circuit. It is also interesting to see how, according to the behavioral specification, the output of the Operative Unit feeds the input of the register.

The delay elements are also generated for both the Operative Unit and the register.

4.1 Prevention of Data Hazards

Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the value read from a register can be different from the one we expected to read in an unpipelined machine. This happens when an instruction depends on the results of a previous instruction still in the pipeline. The data hazards which are possible in the ADLX are of RAW(read after write) type. To prevent the reading of wrong value from the General Purpose Registers a mechanism of *Register Locking* has been implemented.

There are 32 32-bit General Purpose Registers (GPR) in the ID stage of the ADLX. We also have a 32-bit lock register where each bit corresponds to a GPR. The lock bit is set to one if the corresponding GPR is going to be written back in the the final WB stage. After the write back to the GPR takes place, the corresponding bit is again assigned 0.

If any instruction tries to read a GPR while its corresponding lock bit is 1, then the instruction is stalled till the write-back to that GPR takes place and lock bit value changes to 0.

When GPR_i is going to be written back we have

```
lock[i] = 1;
```

On Write Back, the following takes place

```
GPR[i] = write-back_data ;
lock[i] = 0 ;
```

It is also possible to use *instruction scheduling* to reduce the number of data hazards in the pipeline. With this technique, the compiler can be used to reschedule the code by rearranging the code sequence in order to avoid hazards.

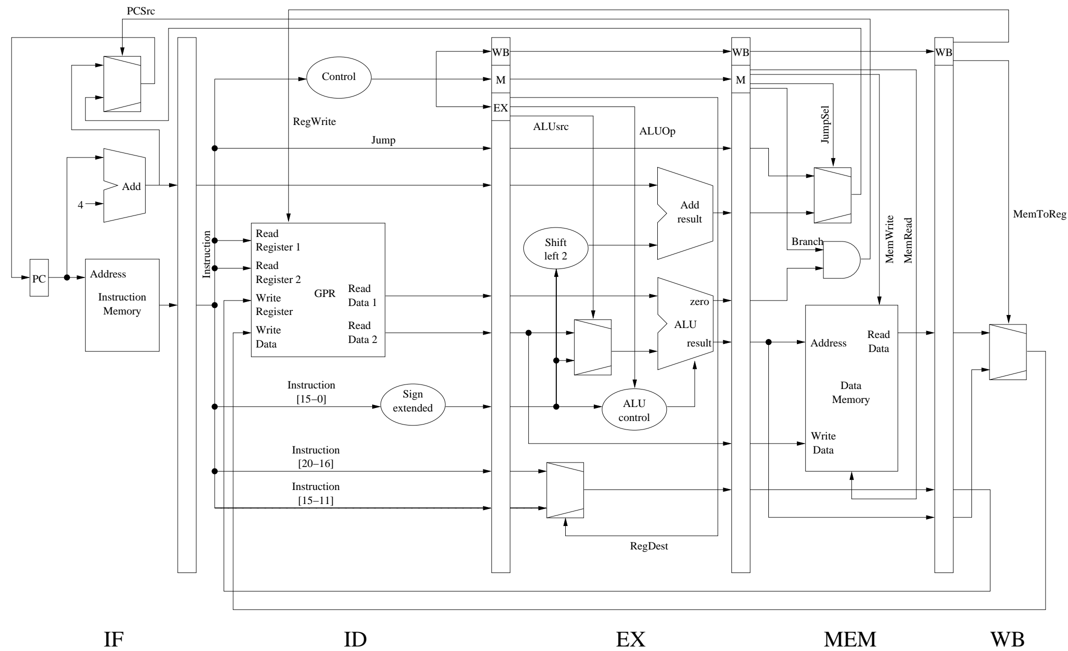

Figure 2: DLX Data Path

4.2 Prevention of Control Hazard

When a branch is executed, it may or may not change the PC (program counter) to something other than its current value plus 4. If a branch changes the PC to its target address, it is a taken branch; if it falls through, it is not taken.

If the instruction is a taken branch, then the PC is normally not changed until the end of MEM stage, after the completion of the address calculation and comparison(see Fig. 2). Therefore the next 2 instructions may or may not be the right ones to be executed.

The scheme which we use to get around this problem is called *delayed branch*. In this scheme, the next 2 sequential successors of the branch instruction are always executed. It is the job of the compiler to make the 2 successor instructions valid and useful or, if not possible, to schedule a `NOP` instruction instead. This scheme is used to reduce complexity of the ID stage of the pipeline and allows normal flow of instructions in the pipeline rather than stalling for a "stage cycle" in the code.

After this the third sequential instruction or the instruction at the branch address is executed depending on whether the branch is taken or not. The behavior of a delayed branch is the same whether or not the branch instruction is taken.

5. THE SYNTHESIS PROCESS

In this section the entire synthesis process will be described in details. This process has been carried out in three main steps: *high level synthesis*, *logic synthesis* and *physical design*.

5.1 High Level Synthesis

The input specification as described in section 4 has been fed as input to Pipefitter in the form of 5 separate input files (one for each processor pipeline stage). In order to complete the first step of the synthesis process, Pipefitter also needs some additional informations. A file containing a list of the available resources and their characteristics (i.e., I/O size and list of operations performed) must be provided for Pipefitter to map each arithmetic/logic operation onto a physical device. This mapping process is performed through a genetic algorithm [3] that provides a solution optimized based on directives imposed by the designer (e.g., minimum area occupation, minimum latency, etc.).

Pipefitter provides as output a set of files:

- An RTL-synthesisable Verilog specification for each device in the Data Path (e.g., registers, Operative Units, constants, etc.).
- A Verilog netlist implementing the Data Path by the instantiation and the proper interconnection of all the modules described in the previous item.
- A complete specification for the Control Unit to be fed as input to *AFSMGEN* [15]. This tool, integrated in the automated design flow, provides a Relative Timed or Speed Independent (SI) [12] implementation of the Control Unit based on David Cells (using the approach described in [10]).
- A Verilog netlist implementing the matched delay elements required for the generation of the local clock signal.
- A Verilog netlist implementing any additional combinational logic required to map AFSM states into Data Path signals.
- A file reporting all the timing constraints that will have to be satisfied during the Physical Design phase.
- A top-level netlist comprising the entire circuit, which instantiates and properly interconnects the Data Path, Control Unit and the matched delay elements.

5.2 Logic Synthesis and Physical Design

Logic Synthesis consists of technology mapping all the files generated by Pipefitter in a bottom-up fashion. We performed logic synthesis using Synopsys DC, however any synthesis tool may be used. The bottom-up hierarchical technology mapping is necessary, in order to satisfy the asynchronous local constraints and to

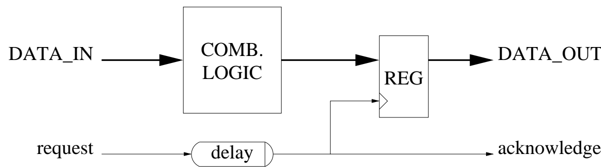

Figure 3: Bundled Data Architecture

hide feedback loops from the synthesis tool. Most of the files generated by Pipefitter are readable by Synopsys DC, except for the control unit.

The Data Path is synthesized first. Every Data Path subcircuit is generated by Pipefitter with a register at its output and local clock signal. Hierarchical synthesis on the asynchronous Data Path works by technology mapping and optimizing each local Data Path subcircuit by using the local clock signal as a "virtual" clock, set to the minimum clock frequency. Ultimately, every local clock signal is replaced by an asynchronous driver, based on the AFSM and the matched delay elements. After each Data Path block has been synthesised, any remaining non-timing critical Data Path glue logic may be mapped.

Next, the asynchronous control is synthesised. The AFSMGEN tool maps the AFSM specification of the control circuit into a GTECH (Synopsys Generic TECHnology library) netlist which can be read an mapped by Synopsys DC. If the circuit generated is not SI, then appropriate timing constraints should be applied during synthesis [15]. Even if the AFSM is SI, minimum-delay hierarchical timing constraints are applied during synthesis to optimize the logic of every state signal to ensure fast switching between states and good performance.

The matched delay elements are synthesized based on the minimum "cycle" time, *i.e.* critical path delay of their corresponding Data Path subcircuits. Thus, their appropriate delays are only known after each Data Path subcircuit has been synthesized. The delay elements are synthesised based on the same gate-level Verilog netlist, but each element is synthesized with a different synthesis script, based on the amount of delay required. In this way, Synopsys DC inserts the appropriate amount of buffers/inverters to meet the timing between input and output signal of a delay element. The delay elements apply delay in a unidirectional manner, *i.e.* only on the rising edge of the input (the request/acknowledge signal). This is because data is matched only to one of the two edges but not both, so using the same delay for both edges would slow down the circuit unnecessarily. The implementation of a delay element is shown in Figure 4.

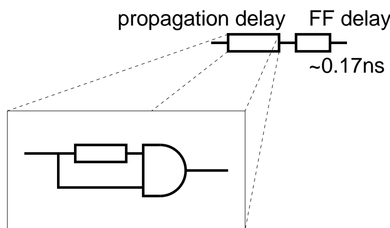

Figure 4: Unidirectional Delay Element

When the Data Path, control, any extra combinational logic and the matched delay elements are all generated, they are combined into the entire circuit. In the case of the DLX, the five Data Path stages were implemented separately by Pipefitter and then combined to form the entire CPU. At this stage, a design is able to enter the physical part of the flow.

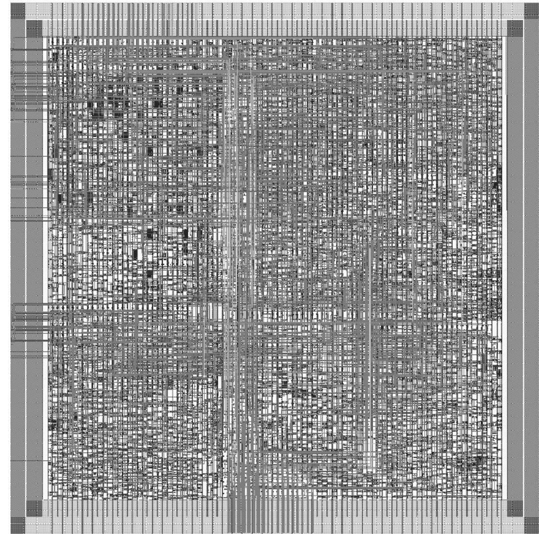

Figure 5: DLX layout. $700\mu m \times 700\mu m$

During the physical part of the flow, the hierarchy is to be preserved, in order to enable control of the placement and routing (P&R) process of every sublock of the design. This is due to fact that timing constraints are local and not global, as in a synchronous design. Control sub-circuits (AFSMs) are SI, thus do not possess any timing assumptions, and are always guaranteed to operate correctly. Thus, the placement and routing process of control units does not possess timing assumptions, however a good placement does improve performance.

The timing assumptions that must be preserved during the (P&R) process reside into the Data Path subcircuit. These include the magnitude of the matched delays, which should not become less than the critical path of the corresponding circuit, and the bundling of data signals, *i.e.* that signals of Data Path blocks arrive as a bundle. Thus, these assumptions must be guaranteed by the P&R process.

The two timing assumptions of Data Path blocks may be satisfied by physical design tools. In the same way that we use synchronous synthesis to synthesize the Data Path blocks and then replace the clock with an asynchronous control signal, in the physical part of the flow timing-driven placement, clock-tree generation and timing-driven routing of each Data Path element may be performed as if the module was clocked by the asynchronous signal. Next, when modules are interconnected, the clock signal is replaced by an asynchronous driver signal. In-placement optimization and resizing can also be used to re-calibrate the matched delay elements according to the P&R timings.

The only disadvantage of the hierarchical flow we are using is that due to the fact that it is bottom-up it consumes more designer time. The P&R process of the DLX has not been completed as yet. However, we have performed trial P&R to estimate the area of the design and to extract parasitics. Figure 5 shows a trial standard-cell layout of the DLX processor.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented the design of a 32 bit pipelined asynchronous DLX microprocessor based on a *fully automated design flow* for asynchronous circuits.

The total area occupation for the ADLX was about $0.49mm^2$ and

the relative occupation of Control Unit, Data Path and Delay Elements is shown in table 1.

MODULE	CU	DP	DE
IF	15%	79%	6%
ID	27%	67%	6%
EX	27%	67%	6%
MEM	26%	63%	11%
WB	32%	56%	12%
ADLX	**25%**	**68%**	**7%**

Table 1: Relative area occupation for the ADLX

This first version of the ADLX was intended to prove the effectiveness and practical usability of Pipefitter. Particular care was not taken to implement a high-performance microprocessor. In order to evaluate the final implementation of the ADLX, it was compared to an equivalent synchronous implementation. However it must be pointed out that the two projects (synchronous and asynchronous) were carried out separately and by different groups and even though they were based on similar architectures, some differences in the implementations can be responsible for a significant difference in the comparison. Nevertheless, some interesting conclusions can be drawn. The conditions under which the comparison was carried out, are the following:

- Area comparison between the two implementations was made at gate-level. Since no clock tree was generated for the synchronous version, in order to keep the comparison fair, the delay elements of the asynchronous implementation were not taken into consideration.
- Each module of the pipeline was individually compared for the two implementations as well as the complete DLX.
- Memory elements (instruction and data memories and register file) were simulated at a behavioral level and therefore did not have any impact on the overall performances.

The most significant results can be summarized as follows:

1. The introduction of synchronization logic blocks in the asynchronous implementation, is responsible for an area overhead. This overhead is more significant in strongly control-driven stages where the data path is very small or not present at all (e.g., MEM and WB), while it is almost negligible in data-driven stages (e.g., EX).
2. Synchronization logic blocks are also responsible for a timing overhead and have the same impact on different stages as described in the previous item.
3. Both area and timing overhead have a small impact on the final ADLX. In fact, most of the area occupation in the DLX is due to adders, ALU and registers whose area is the same in the two implementations thanks to the bundled data approach. Similarly, the throughput is imposed by the slowest stage in the pipeline which is, in our case, the EX. As stated before, this stage suffered a very small timing overhead (less than 4%) which reflects on the overall performance.

Some improvements will be carried out in future implementations that will enable the ADLX to have better performance i.e. skipping handshakes which are not needed by certain instructions.

7. REFERENCES

[1] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, Apr. 1997.

[2] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. pages 384–389, 1991.

[3] I. Blunno and M. Lazarescu. Asynchronous scheduling/binding using a genetic approach. In *MIPRO 2002*, May 2002.

[4] Ivan Blunno and Luciano Lavagno. Automated synthesis of micro-pipelines from behavioral verilog hdl. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 84–92. IEEE Computer Society Press, Apr. 2000.

[5] M. Coenen. On-chip measures to achieve emc. In *IEEE International Symposium on EMC*, pages 31–36, Feb. 1997.

[6] J. Dalton et al. Opencores home page, 2002. See http://www.opencores.org.

[7] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(2):247–253, June 1996.

[8] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, et al. AMULET1: a micropipelined ARM. In *Proceedings of the IEEE COMPCON*, pages 476–485, 1994.

[9] J.L̃. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publisher Inc., 1990.

[10] L. A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, Dec. 1982.

[11] Joep Kessels and Ad Peeters. The Tangram framework: Asynchronous circuits for low power. In *Proc. Asia and South Pacific Design Automation Conf.*, pages 255–260, Feb. 2001.

[12] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits*. Kluwer Academic Publishers, 1993.

[13] Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, and Alex Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 114–125. IEEE Computer Society Press, Apr. 2000.

[14] R. Smith, K. Fant, D. Parker, R. Stephani, and C. Y. Wang. An asynchronous 2-D discrete cosine transform chip. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 224–233, 1998.

[15] C.P̃. Sotiriou. Implementing asynchronous circuits using a conventional eda tool-flow. In *Proc. Design Automation Conf.*, June 2002.

[16] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, et al. A fully asynchronous low-power error corrector for the DCC player. *Journal of Solid State Circuits*, 29(12):1429–1439, Dec. 1994.