Reprogrammable Controller Design From High-Level Specification.

M. BENMOHAMMED [1],

[1] Computer Science Department, University of Cne, 25000 Constantine, ALGERIE.
Email : ibnmyahoo.fr

M. BOURAHLA [2], and S. MERNIZ [1]

[2] Computer Science Department, University of Biskra, Biskra, ALGERIE.

ABSTRACT

Existing techniques in *high-level synthesis* mostly assume a simple controller architecture model in the form of a single FSM. However, in reality more complex controller architectures are often used. On the other hand, in the case of programmable processors, the controller architecture is largely defined by the available control-flow instructions in the instruction set.

With the wider acceptance of behavioral synthesis, the application of these methods for the design of programmable controllers is of fundamental importance in *embedded system* technology. This paper describes an important extension of an existing architectural synthesis system targeting the generation of ASIP reprogrammable architectures. The designer can then generate both style of architecture, hardwired and programmable, using the same synthesis system and can quickly evaluate the trade-offs of hardware decisions.

Keys-Words: CAD-VLSI, Architectural Synthesis, High-Level Synthesis, Controller, DSP, ASIC, ASIP, FSM, VHDL.

1. INTRODUCTION

The advantage of ASIC solutions is their cost efficiency. However, for competitive markets like consumer electronics or telecommunications, ASICs often lack flexibility and programmability.

In the past, commercial DSP processors were the only choice when programmability was desired. More recently, a trend emerged in the DSP community to build an *Application-Specific Instruction-set Processors* (ASIP), which offers programmability, while maintaining low cost and power consumption as well as meeting high speed. As high-level synthesis is on the verge of being accepted in the commercial market, new techniques are necessary to support a processor style of design which is becoming as an important part of today's embedded system. In turn, this style is beginning to appear as an important part of tomorrow's system-on-a-chip [10].

This paper presents steps which have been implemented within an existing architectural synthesis system (AMICAL) [11] in order to support the design and construction of a reprogrammable controller. As shown in figure 1, the designer works with a characteristic algorithm linked to a class of applications the processor is expected to support and a set of constraints for the synthesis process. The new implementation then produces a reprogrammable architecture containing two main parts: a section for the data-calculation, and a section containing the *sequencer*, program *ROM* and *micro-instruction register* (figure 6). The resulting architecture will be optimized for performance for the given input algorithm within the defined constraints. The construction process can also be guided interactively, analogous to synthesis for hardwired architectures [11]. Working together with a firmware development system containing a retargetable *code-generator* and *instruction-set simulator* [12], the designer can quickly evaluate the trade-offs between hardware decisions.

The remainder of this paper is organized as follows: In section 2, we describe the new AMICAL design flow for reprogrammable microcoded controllers. Section 3 describes the current working architecture template and the synthesis methodologies employed. In section 4, a design example is presented. Section 5 presents some experimental results. Section 6 provides a summary and an outlook for future work.

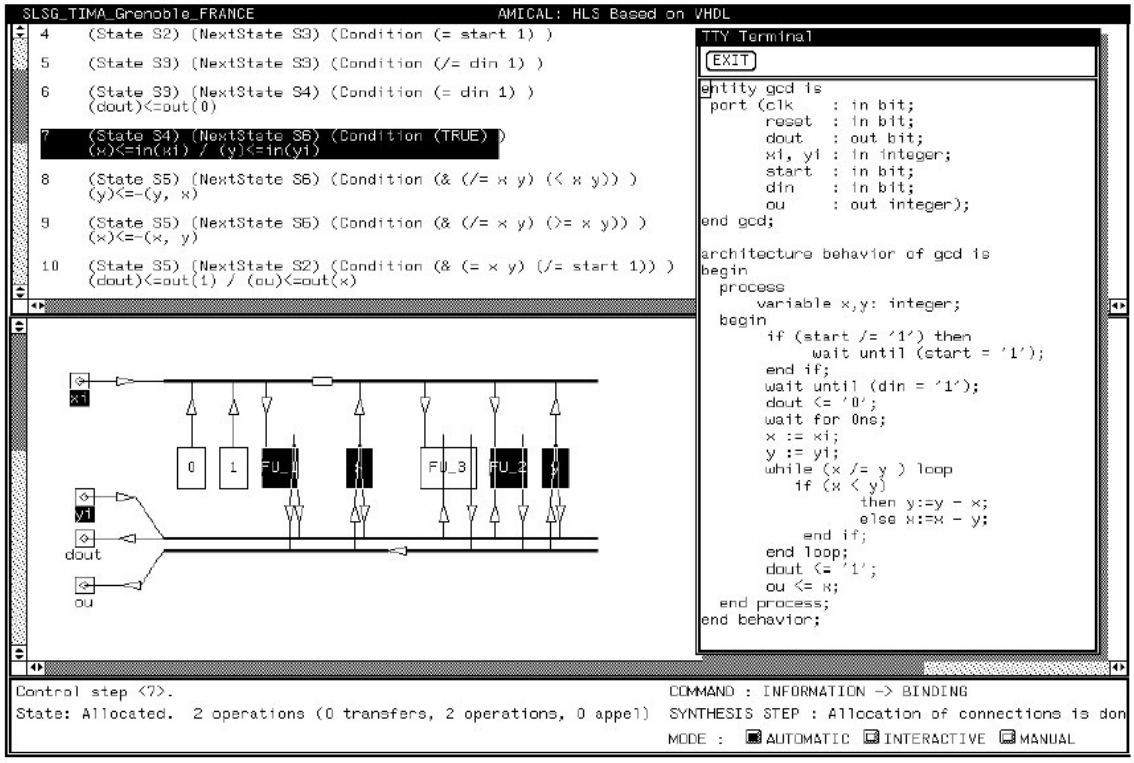


Figure 1 : AMICAL Windows.

2 NEW GENERAL DESIGN FLOW

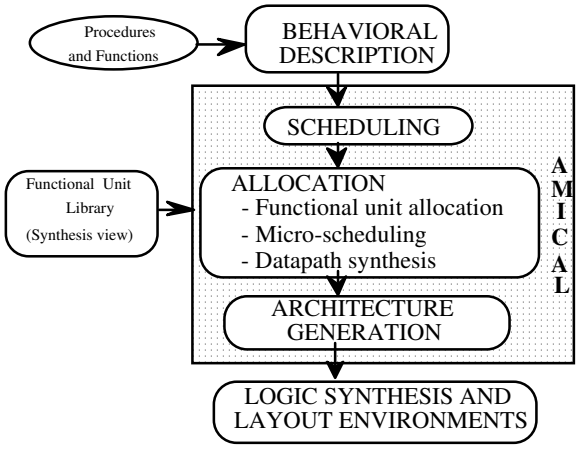


Figure 2 : AMICAL design-flow

AMICAL is an interactive high-level synthesis system targetted towards control-flow dominated circuits [11][16]. It starts with two kinds of information: a behavioral specification given in VHDL and an external library of functional units. This corresponds to the second step of the methodology introduced above. The first step, system-level analysis and partitioning is performed manually. The AMICAL design-flow is illustrated by figure 2. The behavioral description may make use of complex sub-systems through call to procedures and functions. However for each procedure or function used, the library must include at least one functional unit able to execute the corresponding operation. During the different steps of the high-level synthesis, the functional units are used as black boxes. The only pieces of information required about each functional unit are included in the synthesis view. However to complete the description at the register transfer level, the details of the functional units (implementation view) are required.

The different steps involved in the synthesis process are : scheduling, allocation and architecture generation (figure 2). During the first step, the scheduler reads in the VHDL description and produces a finite state machine presented as a transition table. Each transition corresponds to the execution of a control step under a given condition. All the operations of a given transition may be executed in parallel. An operation may correspond to a standard operation of VHDL (e.g. +, -, ...) or to a procedure call. After scheduling, architectural synthesis starts with two kinds of information, namely the scheduled description and an external functional unit library. The functional unit allocation step associates a functional unit with each operation in the state table. The micro-scheduling is then generated according to the execution

scheme for each operation. Each operation is decomposed into a set of transfers, which are scheduled into micro-cycles. Each micro-cycle contains a set of parallel transfers that take one basic clock cycle to execute. The datapath synthesis includes the component (functional unit and register) placement and the connection allocation (buses and switches). The response time of AMICAL is very short and the combination of automatic and manual synthesis allows a quick and broad exploration of the design space in real time. Furthermore, AMICAL provides many facilities for analyzing the generated architecture (statistics, evaluation,...).

AMICAL produce a system composed of a complex datapath and a controller (figure 3). The datapath may include complex functional units. These are described as a co-processor, executing complex procedures and functions. This model allows for design re-use and a recursive design methodology, i.e. a design produced may be as a component in a more complex design.

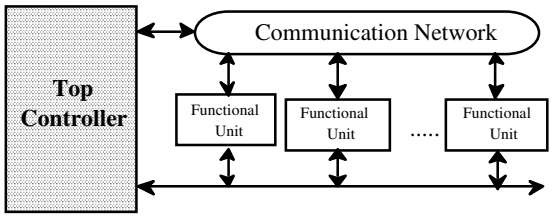


Figure 3. Target architecture

In order to allow late changes when designing the chip architecture, the controller needs to be programmable. In the present version of AMICAL (figure 1) [11], a non-programmable system-level controller can be described as a flat FSM. The goal of this work is to develop an extension of AMICAL, which will allow the generation of a reprogrammable controller.

Figure 4 shows the new design flow for the generation of the two styles of architectures: hardwired and reprogrammable micro-coded controllers. For programmable architectures, a set of modifications on the initial AMICAL tool were necessary keeping the high-level design methodologies. Figure 4 shows the new design flow, where solid lines represent the initial flow and dashed lines indicate the modifications introduced.

The main scheduling algorithm in AMICAL is called *Dynamic Loop Scheduling* (DLS) [13]. This algorithm has been adopted mainly in order to suit control flow dominated designs. The scheduler produces a behavioral FSM where in all conditions are executed in the control part of the resulting architecture. The result is a hardwired FSM controller. Two major modifications had to be done in order to generate a reprogrammable architecture:

- Definition of a new architecture template as explained in section 4.
- Transformation of the scheduling result: All condition evaluations have to be executed in the datapath. Therefore, a resource constrained transformation engine has been adopted to extract conditions from a set of mutually exclusive data flow graphs and to produce their corresponding FSM.

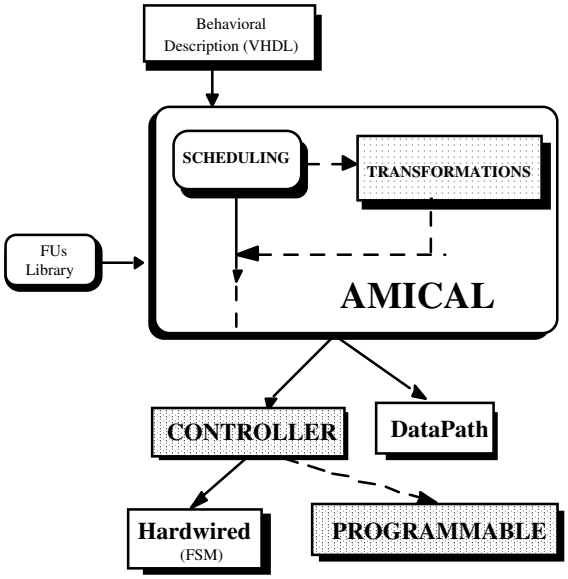


Figure 4: An Extended AMICAL Design-Flow

The generated controller is in a microcode form (figure 5). A first application is used to generate a prototype of a reprogrammable processor architecture. For a new application, the prototype architecture will be re-used: we have just to generate the new microcode, which will be loaded in the ROM of the programmable processor: AMICAL will take into account the overall constraints (resource and time) to generate an other microcode without altering the method of generating of a datapath.

3. ARCHITECTURE TEMPLATE

The target ROM-based architecture is shown in figure 5. It is composed of a top controller and a datapath. The controller is composed of a ROM to store micro-instructions and a sequencer to compute the next micro-instruction address depending on the conditions returned by the datapath. The sequencer is fixed in advance. The controller architecture can be adapted to any set of control flow instructions. The branching that can be executed can be a *2-way* or an *m-way* conditional one. This is beyond the commonly-found micro-sequencer capabilities.

No modifications are needed in the datapath synthesis process. The architecture consists of a set of functional units communicating through a network. The FUs may run in parallel. The FUs produce a set of flags known as the *Code Condition Register* (CCR) which can be used for modeling an unconditional as well as a conditional branch. The *Micro-Instruction Register* (MIR) is composed of three parts:

- An *action* section, which corresponds to the set of commands required to activate the appropriate resources in the datapath.

- A *Next Addr.* (Next Address) section, which represents the next micro-instruction address.
- A *Mode* section, modeling the branch type (Mode = 0 represents an unconditional branch and Mode = 1 represents a conditional branch).

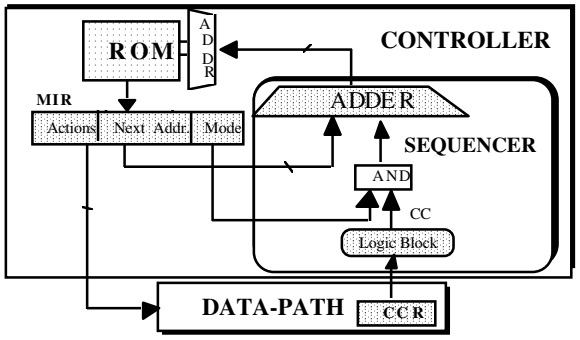


Figure 5: Architecture Template

4. 1. Sequencer Model

The sequencer contains an *adder* and a *logic block* to compute the displacement in the ROM depending upon the content of the *CCR* and the *Mode* type. The two modes are:

1- Conditional branch (figure 6(a)): in this case Mode = 1, the logic block, depending on the CCR, determines the displacement which corresponds to the selected branch. The Next Address in the ROM is computed by adding the address presented in the MIR and the displacement.

2- Unconditional branch (figure 7(b)): in this case Mode = 0 and the sequencer delivers the address present in the MIR.

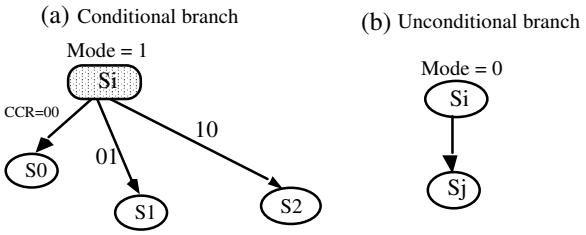


Figure 6: Conditional and Unconditional Branche

4. 2. Synchronization Scheme

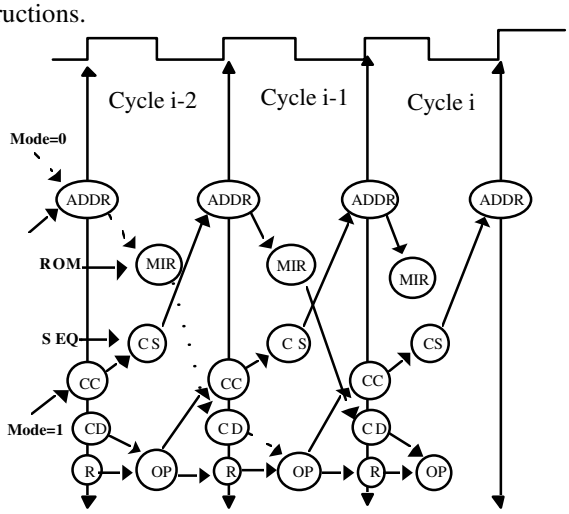
The programmable architecture follows a three stage *pipelined model* as shown in figure 7. Each instruction takes 3 cycles to be executed:

- **Cycle 1**: *Fetch* : next ROM address computation.
- **Cycle 2**: *Instruction decoding.*
- **Cycle 3**: *Instruction execution.*

Once the micro-instruction is decoded and stored in the MIR, two cases are possible:

1. An unconditional branch execution (Mode = 0): the sequencer and the datapath continue to work simultaneously.
2. A conditional branch execution (Mode = 1): the sequencer has to wait for the datapath to update the flag register CCR to be able to compute the displacement. The sequencer waits for the computed flag.

We will see in the next section, how the scheduler ensures that the three parts work together to execute the micro-instructions.



ADDR: Current ROM Address, CD: Commands,
CS: Control: Sequencer, CC: Condition Code,
OP: Operations, R: Register value

Figure 7: Synchronization Scheme

4. DESIGN EXAMPLE

This section contains an example: the synthesis of the *Greatest Common Divider* (GCD) of two integers. In particular, we will show the different transformations of the resulting FSM in order to capture the programmable architecture style. This includes the execution of the conditions in the datapath and the synchronization between the different parts of the architecture. Figure 8 shows the VHDL description of the GCD example.

```
entity gcd is
port (start: in bit;
      xi, yi: in integer;
      ou: out integer);
end gcd;
architecture behavior of gcd is
begin  process  variable x, y: integer;
       begin
          wait until (start = '  1'  );
          x:= xi; y:= yi;                    (1)
          while (x /= y ) loop
          if (x < y )  then y := y-x;        (2)
                    else  x := x-y; (3)
          end if;
          end loop;
          ou <= x;                           (4)
       end process;
end behavior;
```

Figure 8: Description of the GCD example

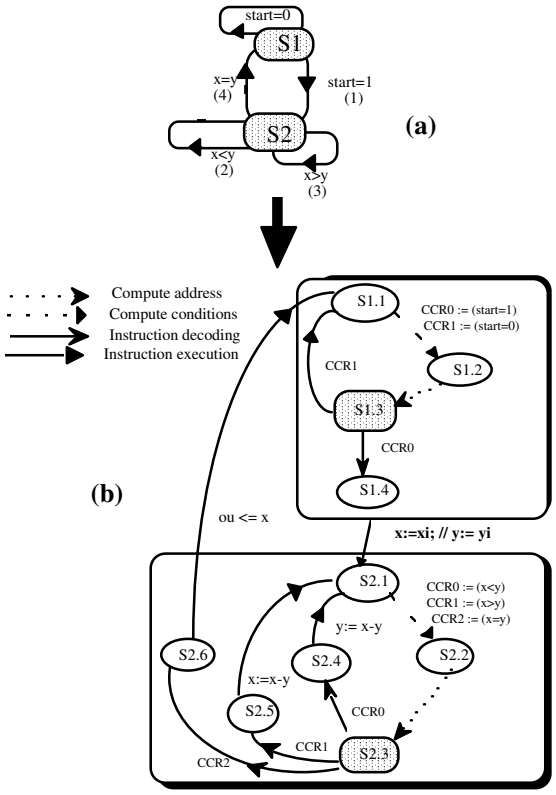


Figure 9. (a) Scheduled FSM of the GCD example, (b) Transformed FSM

The first step is performed by the DLS [13] algorithm and result in a *Mealy* FSM as shown in figure 9(a). Each transition consists of a set of actions to be executed and a set of conditions to be evaluated. Since, the conditions have to be calculated in the datapath, another scheduling step is needed. Each condition is represented by a data flow graph. All the data flow graphs corresponding to a multiple branch (the associated transitions are issued from the same state) are scheduled independently and merged onto the same FSM. In the GCD example, the conditions are simple and can be scheduled in the same state as shown in figure 9(b). More complex conditions can be handled The result of each condition is stored in the appropriate flag of the CCR. The third step is to synchronize the three parts (ROM, Datapath, sequencer) in a pipelined fashion. The sequencer stalls to wait for the computed flags. This necessitates the insertion of idle cycles (states S1.3, S2.3). Depending on the presence of actions to be executed, the datapath may (states S1.4, S2.4, S2.5, S2.6) or may not (state S1.4) have to wait for the instruction decoding.

5. EXPERIMENTAL RESULTS

The GCD example has been extended to a three input operation (GCD3) and synthesized at the behavioral level by AMICAL in the same manner as the design example in section 5. "Bubble" represents the synthesis of an example to bubble sort for integers between 0 and 255. "Answer" is the controller section of an automatic answering machine. "M. Estimator" (Motion Estimator) is a part of a videophone CODEC chip [26]. The CODEC chip codes and decodes sequences of pictures through a pipeline of 12 operators. Table 1 shows the results obtained for the two architecture AMS Technology) : the hardwired and the programmable. It can be observed that, compared to the hardwired solution, the datapath corresponding to the ROM-based architecture has an average increase area factor of 168 % on the examples listed in table 1. This is due to the migration of the condition evaluation from the controller towards the datapath (Fig. 12 and 13). Also, an average increase factor of the clock period required for the programmable architecture is 161 %. In the hardwired architecture of the "GCD" and "GCD3" examples, the slowest operation is the subtraction operation, which determines the clock period. However, in the ROM based architecture, the slowest operation is the read operation of the ROM being used (Fig. 12 and 13). This is not likely to be the limiting factor in larger examples; therefore, the impact on the required clock cycle is expected to be smaller [5]. The two solutions (Hirdwired and Programable (ROM)) was simulated with the same tool (SYNOPSYS) with GCD3 example. We say (Fig. 10 and 11) that both solutions give the same behaviour.

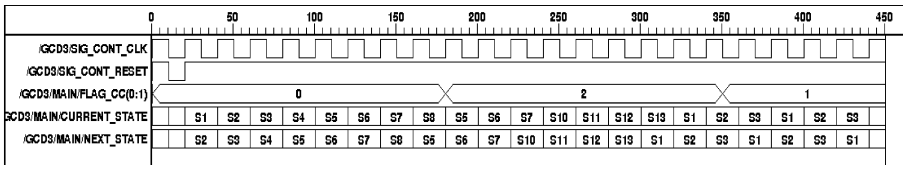


Figure 10. Hirdwired Controller Simulation (GCD3)

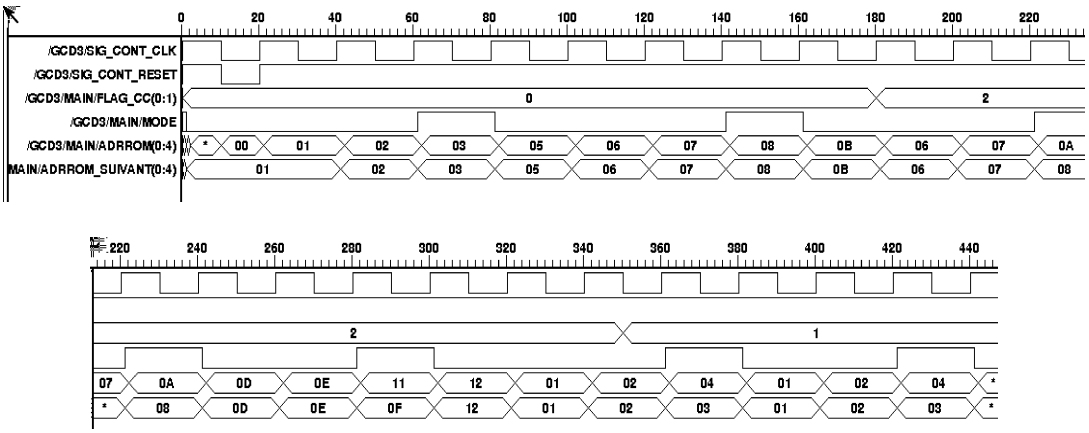


Figure 11. Programmable (ROM) Controller Simulation (GCD3)

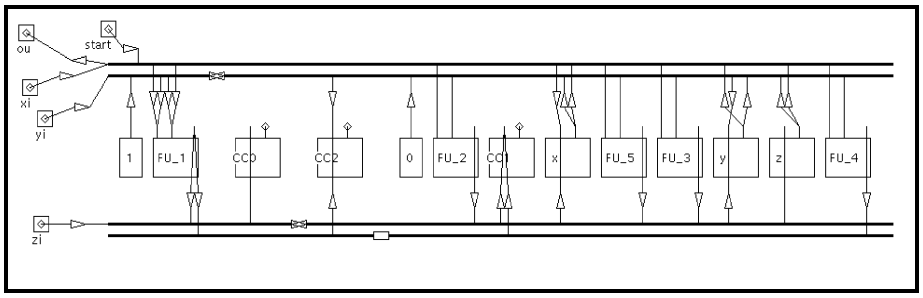


Figure 12. Programable Controller Data-Path (GCD3)

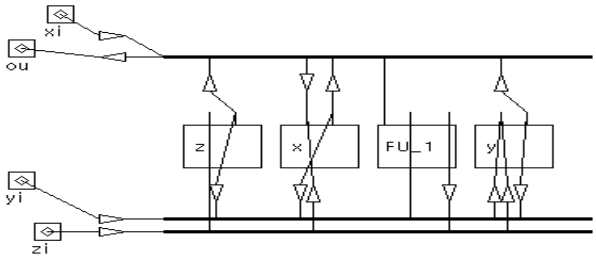


Figure 13. Hirdwired Controller Data-Path (GCD3)

Examples	# VHDL ines	# Transistions			Area (# Transistors)			Cycle Time (ns)		
		Hard.	Prog.	%	Hard.	Prog.	%	Hard	Prog.	%
GCD	36	12	13	+9%	11243	30356	+170%	15	35	+133%
GCD3	38	12	18	+50%	12675	30644	+166%	25	70	+180%
BUBBLE	85	38	55	+42%	21332	40299	+88%	28	75	+167%
ANSWER	280	74	137	+85%	36857	62619	+68%	21	56	+166%
M.ESTIMATOR	441	182	408	+124%	76635	344857	+350%	23	60	+160%
AVERAGE							+168 %			+161%

Hard: Hardwired, Prog: Programmable, %:Overhead.
Table 1: Area and Time Estimation

6. CONCLUSION AND FUTURE WORK

This paper presented a method implemented in an existing architectural synthesis system to extend it for the generation of reprogrammable controllers. This system was initially intended to generate hardwired architectures. Automatic generation of processors provides several benefits of the behavioral architectural synthesis including:

- Library-based functional units allocation for the datapath.
- Optimization to meet Timing Constraints.
- Automatic and Interactive Scheduling.
- Rapid prototyping.

The main contribution of this work has been the proposal and implementation of an architecture template style which enables to obtain a reprogrammable controller from an architectural synthesis system.

The directions for future work consists of developing new styles for the controller and the sequencer. We also aim an additional support for the resource and time constraint' s based scheduling and allocation to reprogramming the controller for new behaviors.

ACKNOWLEDGMENTS

We gratefully acknowledge the help of the following researchers at TIMA/CNRS-INPG (Grenoble, France): Polen Kission, Hong Ding, P.Vijay Raghavan, Clifford Liem, Carlos Alberto Valderama, Mohamed Romdhani.

REFERENCES

[1] M. Benmohammed, A. Rahmoune and P. Kission, "Generating Reprogrammable Microcoded Controllers within a High-Level Synthesis Environment", AJSE, **Arabian journal for Science and Engineering**, Saudi Arabia, vol. 24 no 1B, pp. 57-67, April 2000.

[2] P. Paulin, et. Al., "Trends in Embedded Systems Technology: An Industrial Perspective", NATO **Advanced Study Institute on Hardware/Software Co-Design**, Tremezzo, Italy, June 1995.

[3] C. Liem, P. Paulin, M. Cornero and A. A. Jerraya, "Industrial Experiments Using Rule-driven Retargetable Code Generation for Multimedia Application", Proc. of **the Int. Sym. Sys. Synthesis**, September 1995, pp. 60-65.

[4] M. McFarland, A. Parker and R. Composano, "The High-Level Synthesis of Digital Systems", Proc. of the **IEEE**, Vol. 78, No. 2, February 1990, pp. 301-318.

[5] D. Gajski, W. Wolf, *High Level Synthesis*, Kluwer Academic Publisher, 1992.

[6] R. Composano, R. A. Bergamashi, "Synthesis Using Path-Based Scheduling: Algorithms and Exercices", **27th DAC**, pp 450-455, Orlando, June 1999.

[7] D. C. Ku, DeMicheli, "Relative Scheduling under timing constraints", **IEEE Trans. on CAD/ICAS**, May 2000.

[8] J. Rabaey, et. al., *CATHEDRAL-II: a synthesis system for multiprocessor DSP systems*, in D. Gajski Ed., Silicon Compilation, Adison-Wesley, pp. 311-360, 1993.

[9] D. Knapp, T. Ly, D. MacMillen, R. Miller, "Behavioral Synthesis Methodology for HDL-Based Specification and Validation", Proc. of the **32nd ACM/IEEE DAC**, June, 1994 pp. 286-291.

[10] P. Paulin, J. Frehel, E. Berrebi, C. Liem, J. C. Herluison and M. Harrand, "High-Level Synthesis and Codesign Methods: An Application to Videophone Codec", Invited paper **EuroDAC/VHDL**, Brighton, September 1995.

[11] P. Kission, H. Ding, A. A. Jerraya, "Structured Design Methodology for High-Level Design", **Proc. of the 31st ACM/IEEE DAC**, June 1994.

[12] P. Paulin, C. Liem, T. May, S. Suturwala, "FlexWare: an Flexible Firmware Development Environment for Embedded System", in ***Code Generation for Embedded Processors***, Ed. P. Marweded, G. Goosens, Kluwer Academic Publisher, 1995.

[13] M. Rahmouni, K. 0' Brien and A.A. Jerraya, "A loop based Scheduling Algorithm For Hardware Description Languages", **Parallel Processing Letters**, Vol. 4 No 3, pp. 351-364, 1994.

[14] A. Kifli, et. Al., "Flag/Condition Handling an Branch Assignment for Large Microcoded Controllers", G. Saucier, editor, ***Control Dominated Synthesis From a RT Description***, Elsevier Science Publisher, 1999.

[15] J. Zegers, P. Six, J. Rabaey, H. De Man, "CGE: Automatic Generation of Controllers in the CATHEDRAL-II Silicon Compiler", **Proc. European Design Automation Conference**, pp. 617-621, 2000.

[16] M. Benmohammed and A. Rahmoune, "Automatic Generation of Reprogrammable Microcoded Controllers within a High-Level Synthesis Environment", **IEE Journal : Computers and Digital Techniques**, UK, Vol. 145, no : 3, Mai 1998, pp. 155-160.

[17] F. Poirot and G. Saucier, "Controller Synthesis in the ASYL System", In North Holland, editor, **International Workshop on *Logic and Architecture Synthesis for Silicon Compilers***, 1999.

[18] A. Kiffli, G. Goosen and H. D. Man, "A Unified Scheduling Model for High-Level Synthesis and code Generation", **ED & TC**, pp. 234-238, Paris, March 1995.

[18] M. Benmohammed and K. Polen, "The Application of HLS Techniques for the Generation of Pipelined Microcontrollers", Proceeding of the **7th IEEE International Conference on Electronic, Circuits and Systems**, ICECS2K, Dec. 17-20, 2000, Beyrouth, Liban, pp. 992-997.