IPSJ Transactions on System LSI Design Methodology Vol. 3 234–243 (Aug. 2010)

Regular Paper

Semi-Automatic Control Unit Generation for Complex VLSI Designs

Benjamin Carrion Schafer[†1]
and Majid Sarrafzadeh[†2]

This paper presents a semi-automated way to generate control units for complex VLSI hardware designs based on a massive parallel micro-controller. This micro-controller can execute as many instructions in parallel as needed by the hardware design as well as having an unlimited number of input and output ports. Two versions of this control unit are presented in this paper. A generic one, which is generated from a set of parameters given by the designer and an optimized version which parses the control program that will run on the control unit in order to generate an optimized micro-controller. Results show that up to a 60% in area savings can be achieved using the optimized controller unit instead of the generic one. The presented controller was validated using a previously developed SoC design with a FSM based control unit showing that the functionality can be completely replicated at the expense of incurring in a 7.2% and 15.4% area overhead respectively.

1. Introduction

Shrinking process technology allows higher transistor densities following Moore's law allowing increasingly complex systems to be implemented on a single device. The downside of Moore's law is that hardware designers are now overwhelmed with the complexity of such systems which become extremely difficult to develop and verify. This leads to the need of new more powerful tools that help hardware designers to overcome these difficulties and provide an easier and faster way to implement their designs. A system creator tool that allows the creation of entire complex SoC designs has been therefore developed. Basic units specified in a library (e.g., memories, control units and external memory interfaces) are placed on a scratch pad and connected together. When the system has been completely specified, RTL code is automatically generated for the SoC's top level. The functionality of each module then needs to be specified in any RTL language or high level language performing high level synthesis. This paper focuses on the control unit generation as this is one of the most important unit in any hardware design. This unit is responsible for generating and synchronizing the control signals for the rest of the design units. This is the reason why a dedicated control unit generator was embedded in the system creator tool so that a flexible customized control unit for any hardware design can be generated. The control unit is micro-controller based. By setting a number of parameters in the system creator a complete RTL description of this micro-controller is automatically generated.

The contributions of this work can be summarized as follows:

- Present a semi-automatic micro-controller based control unit for complex SoCs with the potential of executing a virtually unlimited number of operations in parallel.
- Present an control unit optimizer to minimize the final controller size based on the program to be executed.
- Comprehensive experimental results to validate our proposed technique compared to a FSM approach, including a large SoC design.
- Extends previous work on synthetic benchmarks to create synthetic benchmarks for our instruction set for different source code *symmetry* conditions.

2. Related Work

Much work has been done in the past in the area of programmable processors, which has lead to multiple commercial companies offering re-targetable processors e.g., Tensilica [1], Target [2] and Silicon Hive [3] to name a few. Most of the previous research focuses on efficient retargetable compilers for flexible customizable programmable processor architectures, also called ASIPs (Application Specific Instruction-Set Processors). In ASIPs only the coarse architecture is fixed allowing designs to customize the architecture to their particular needs. Leuper's, et al. present a comprehensive overview in Ref. 4).

Marwedel, et al. presented a retargetable code generation environment for

†1 NEC Corporation, Central Research Laboratory
†2 University of California, Los Angeles

© 2010 Information Processing Society of Japan

fixed point DSP processors called CHESS in Ref. 5), based on a mixed behavioral/structural processor representation model. Praet, et al. [6] introduced a retargetable and optimizing compiler for a processor model that captures the connectivity and parallelism of Application Specific Instruction Processors (ASIP) processors.

FPGA vendors have also released families of soft processor cores, which can be tailored to any application. Examples include Altera's Nios processor [7], Xilinx's Micro Blaze [8] and Atmel's CAP micro-controller [9]. These soft processors can be customized including the CPU, peripherals, interfaces and even the arithmetic unit. Nevertheless these customizable processors are not targeted to substitute SoCs control units and have limited parallelism that makes them inadequate as generic control units. They are targeted to work as ASIPs to offload work from the dedicated hardware units.

Much work has been done in the area of re-targetable architectures and code generation, but to the best of our knowledge so far, it has not been attempted to create a dedicate control unit for complex SoCs' optimizing the hardware architecture based on the program being executed and which can be especially tuned for FPGAs making use of their internal block RAM for program memory.

3. Control Unit

Every hardware design needs a control unit. This unit can be considered the brain of the system. It generates the control signals for the different units and synchronizes the inputs and outputs. Some examples include the address lines, the read and write signals for the different memory units and the control signals for the multiplexers. **Figure 1** shows a block diagram of a typical SoC containing memory, which can be divided into different units to increase bandwidth and to allow the execution of tasks in parallel, some dedicated hardware accelerator units, interconnect (bus or crossbar switch) and a control unit.

Two main approaches are normally taken when designing control units. 1) Finite State Machine (FSM) based or 2) Micro-controller based. Both of these approaches have pros and cons. State machine based are normally extremely efficient as these are designed specifically for a given design, but do not allow any future changes. In the FPGA case the FPGA could be re-configured to accommodate any changes in the control unit making it unnecessary to build a flexible control unit. Although this might be true in some cases, in many applications once the initial design is fixed the smallest and therefore cheapest FPGA will be used in order to save costs. This might lead to not being able to modify the control unit in the future due to lack of logic resource or the design might become un-routable. On the other hand there are micro-controller based control units, which are more flexible as they can be reprogrammed and are normally easier to test, but are in general less efficient than their state machine based counterparts in terms of speed and area. Despite the FSM efficiency the only way to automate the hardware generation of generic control unit is the second approach where a customized micro-controller based control unit is generated for a given design. In this approach the controller can be easily parameterized, where the only parameters that need to be specified are the number and size of input and output ports and the number of instructions that need to be executed in parallel. The next section describes in detail how the dedicated control unit is generated using our system creator tool and the controller's internal structure.

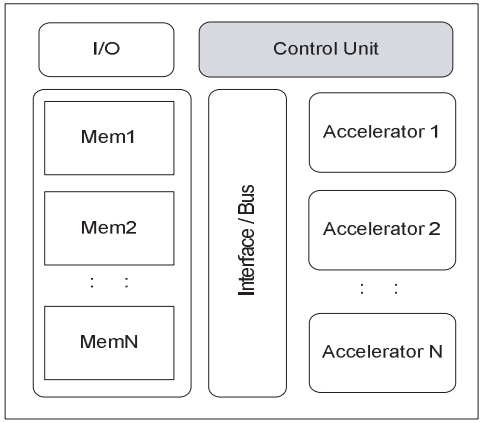

Fig. 1 Typical SoC design block diagram.

3.1 Control Unit Generation

Figure 2 shows a flow graph with the basic steps needed to generate a customized control unit for a specific design using our proposed method. The first

© 2010 Information Processing Society of Japan

step consists of specifying the architectural parameters of the control unit. This step involves defining the number and type of ports needed (direction and size), the number of instruction execution units (IEU), which indicate the number of programs that can be executed in parallel and the number of decoder units, which indicate the maximum number of instructions that can be executed in parallel per IEU. One more architectural parameter that needs to be specified is the size of the memory unit that will hold each separate program data. Once these parameters are defined our method generates a configuration file describing the control unit's architecture, which the compiler reads in order to verify that the assembly code written by the user can be run on the control unit. The designer can the proceed writing assembly code. Two options are available once the assembly code for the control unit has been written. 1) Keep the generic control unit, especially desirable if the program will be updated in the future or 2) optimize the control unit for the the given program. If it is known beforehand that the control unit will always execute the same code, and optimizer parses the program and eliminates the redundant logic not used and program memory size

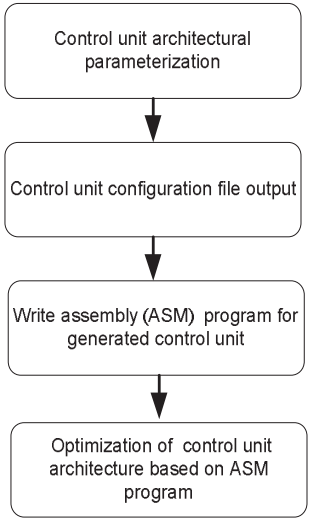

Fig. 2 Control Unit generation flow graph.

reducing therefore the area of the control unit.

The next section describes the internal structure of the controller as well as its instruction set, compiler environment and the assembly code structure.

4. Control Unit Structure

The control unit design is based on a simple micro-controller architecture. The most important architectural aspect of a control unit is the number of instructions that can be executed in parallel. Different hardware designs need different amount of control signal and it is very important that these control signals can be processed concurrently as demanded by the system. In order to avoid performance penalties, the architecture is pipelined so that each IEU can execute 1 instruction per cycle (if there are not simultaneous accesses to the same register). The only case when there might be performance degradation compared to a custom FSM control unit is when a feedback signal is set. Input signals can be treated as interrupt signals and the IEU needs jump to the specific subroutine and flush the instructions pipeline being executed.

Figure 3 shows a block diagram of the controller's main building blocks. It consists of four main units. The Format Program Data Unit formats the 32 bits program stream downloaded to the SoC into the necessary format and stores the program into the required program memory unit. The Program Memory Unit (s)

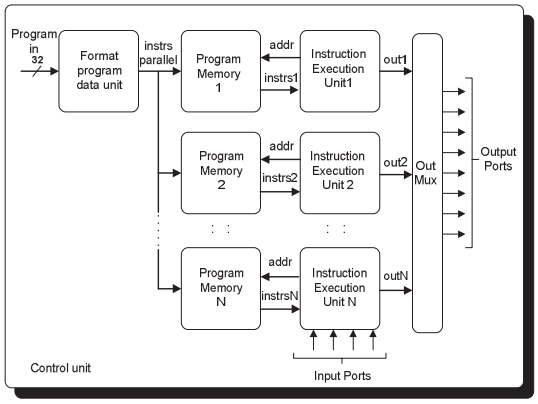

Fig. 3 Control Unit block diagram.

© 2010 Information Processing Society of Japan

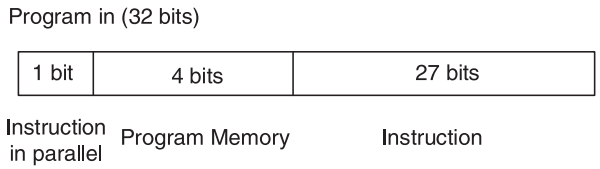

Fig. 4 Program data structure.

stores the program data. In the case of the FPGA implementation presented in the experimental section embedded block RAM is used as program memory. The Instruction Execution Unit (s) decodes and executes the instructions. The last unit is the Output Multiplexer Unit which is responsible for routing the data from the Instruction Execution Units (IEUs) to the specific output ports. As the controller also needs to read feedback signals generated by the system the input signals are connected directly to the instruction execution unit as needed.

4.1 Format Program Data Unit

The Format Program Data Unit formats the 32 bits input stream generated by the compiler into the desired format so that instructions that need to be executed in parallel are stored in the same program memory. In order to format the instructions correctly it has to know the predefined data format generated by the compiler. **Figure 4** shows the data format of the 32 bits generated for every instruction by the compiler. The first bit indicates if this instruction has to be executed in parallel with the previous instruction. The next four bits indicate the program memory unit where this instruction needs to be stored limiting the number of programs execute in parallel to 2^4 and the last 27 bits contain the instruction to be executed. The program is downloaded once to the program memories before the SoC starts operating, but can be updated anytime.

4.2 Program Memory Unit

The Program Memory Units hold the different programs that will be executed in parallel. The control unit will have as many program memory units as Instruction Execution Units. The Format Program Data Unit stores the instructions in the correct memory unit parallelizing the instructions that must be performed in parallel (**Fig. 5** shows how data is stored in the program memory units). As indicated before, in the case of the FPGA implementation presented in the experimental section block RAM is used for the Program Memory Unit as it allows the customization of the program memory bitwidth so that as many instructions as needed are stored at the same memory location and are therefore able to be read, decoded and executed at the same time. Every instruction consists of a 27 bit opcode meaning that Nx27 bits is the bitwidth required to execute N instruction in parallel. The wordlength of the program memory depends on the size of the program. Note that the bitwidth and wordlength of each program memory can differ.

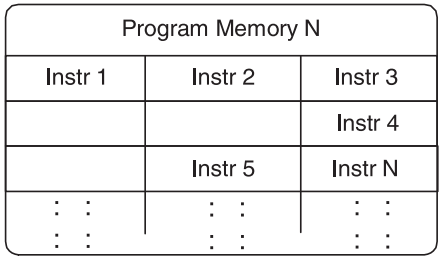

Fig. 5 Program data memory layout.

4.3 Instruction Execution Unit

The Instruction Execution Unit (IEU) consists of two main units. The decoder unit which is responsible for decoding the different instructions and the program counter which generates the addresses to read the next instruction from the program memory units. In case that a *jump* is found in the code the program counter will jump to the specified address. Two general purpose registers and accumulators are also given in each IEU. The internal architecture is pipelined in order to be able to execute instruction block read from the program data in parallel per cycle.

Figure 6 shows a block diagram of this unit. As shown in the diagram as many decoder units as requested are instantiated in the design so that as many instructions as needed can be executed in parallel. The input signals generated by other units in the SoC are connected directly to the IEU that needs to read these signals. The output of the IEU are the control signals that will be routed to the requested output port, which in turn is connected to different units in the SoC, connected manually beforehand in the system creator tool.

© 2010 Information Processing Society of Japan

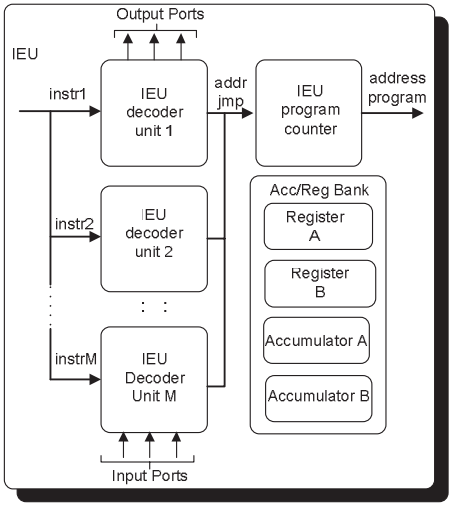

Fig. 6 Instruction execution unit block diagram.

Table 1 Control Unit micro-controller instruction set.

Operation	Instruction
Jumps	JMP, JMPNA, JMPNB, JMPZA, JMPZB
Data transf.	WRITE, WREGA, WREGB, LDA, LDB, MOVA, MOVB
Logic opr.	SHLA, SHLB, SHRA, SHRB, ROLA, ROLB, RORA, RORB, ANDA, ANDB, ORA, ORB
Arith. oper.	ADDA, ADDB, SUBA, SUBB
Miscellaneous	WAIT, NOP

4.4 Controller Instruction Set

In order to enable the controller to execute as many simple instructions in parallel as needed by the system a reduced instruction set with the main basic operations was developed. **Table 1** summarizes these instructions. There are three types of jumps. A direct jump, a jump if accumulator A or B is zero and a jump if it the accumulator is negative. Four types of data transfer, six types

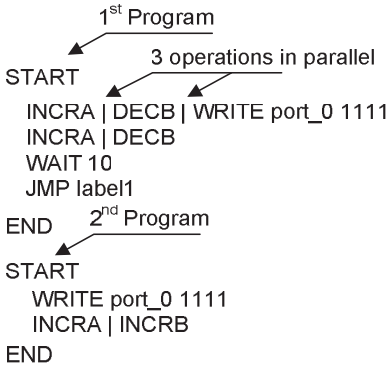

Fig. 7 Control Unit assembly code example.

of logical operations, two types of arithmetic operations and two types of no operations (wait operations is transformed by the compiler to NOPs).

An example of a control unit program is given in **Fig. 7**. The key words START and END define the limits of a program run on a single instruction execution unit. This program is stored in a single program memory unit. The delimiter "|" specifies instructions that have to be executed in parallel. An error message is generated if these instructions try to access the same resource (register, accumulator or port) or if there are more instructions trying to be executed in parallel than decoder units exist in the IEU. The compiler knows how many programs can be executed in parallel, the maximum number of instructions that can also be performed in parallel per instruction execution unit, the name of the ports and the maximum size of each program, as this information is specified in the control unit generator configuration file which is read by the compiler.

The user has two options once the program for the control unit has be written. If a generic control unit wants to be kept in order to allow future code modifications, then the original controller created is kept. The second option is to executed the controller optimizer that reads the program written and optimizes the controller architecture as much as possible to execute this particular program.This first approach makes more sense in ASIC design to allow certain degree of flexibility in the future. In the case of FPGAs an optimized control unit makes more sense as the control unit can always be reconfigured. Section 5

© 2010 Information Processing Society of Japan

describes in detail how the optimizer works.

4.5 Output Multiplexer

The output multiplexer is a crossbar switch that connects the data to be written to the output ports to the specific port. The compiler detects any case in which two different IEUs try to access the same port generating an error message. This implementation is obviously not scalable for a very large number of control signals. In this case a hierarchy of independent switches could be implemented.

5. Control Unit Optimization

Hardware designers always need to make their designs as small as possible because silicon is extremely expensive and designers will therefore always want to generated the smallest possible designs. In the case of FPGAs lower end FPGAs are about two orders of magnitude cheaper then their larger counterparts (e.g., Xilinx's Spartan v Virtex [8] or Altera's Cyclone vs. Stratix [7] FPGAs).

To reduce the size of the controller an optimizer was created in order to minimize the resources used. The input to this optimizer is the compiled binary code of the assembly program written for the initial controller and the architectural configuration description of the controller. The optimizer will then analyze this code and generate a customized control unit, removing any unused logic and customizing the memory needed to hold each program. The next sub sections explain in detail how the optimizations affect the individual units described in the previous section.

5.1 Optimized Format Program Data Unit

This unit does remain unmodified after the optimization stage as data is still formatted in the same way.

5.2 Optimized Program Memory Unit

The number of program memory units will remain the same as in the original case, as the number of programs to be executed in parallel still remains the same. The only parameter that changes is the aspect ratio of every Program Memory Unit (bitwidth and wordlength). Every program has a different size. This determines the wordlength of the memory unit. Moreover different programs will not always executed the same number of instructions in parallel. This influences the bitwidth of the memory units. Therefore each program memory's size is customized to the program it needs to hold.

5.3 Optimized Instruction Execution Units

Two level of optimizations are performed in the IEU optimization.The first is a coarse grain optimization that optimizes the number of decoder units, the accumulators and register bank. Different programs will have to decode different number of instructions in parallel. The second stage is a fine grain optimization that optimizes the instructions decoded by each decoder. Every decoder only needs to to decode a subset of the instruction set, thus saving further area. The program code is analyzed and only the decoding logic needed for those instructions actually being decoded is instantiated. Last the number of ports that the IEU writes to and the number of input is also optimized. The program counter on the other hand remains unmodified.

5.4 Optimized Output Multiplexer

The output switch is also optimized so that the IEU outputs are routed only to the ports that are used in the assembly code, not being able to write to each single output port as in the generic version.

6. Experimental Results

First, we describe how the micro-controller based control unit was validated using a complex SoC previously developed. Then, we show a set of comprehensive results to measure the effectiveness of the optimizer together with explanations on the implication and analysis of the data.

6.1 Controller Validation

In order to validate the control unit method presented in this work an SoC generated beforehand with a FSM based control unit is used. The SoC developed in Ref. 10) is a dedicated hardware architecture to accelerate the execution of the Discrete Element Method (DEM).

The Discrete Element Method is a numerical method to model the behavior of particle assemblies. They can be bonded together to represent rock or remained unbonded to represent soil. Bonded together they can represent entire structures, such as dams or bridges.As the process is explicit, the time step must be limited to a very small value, thus making the DEM extremely computationally expensive. Its wide-spread use is therefore hampered, though the amount of parallelism

© 2010 Information Processing Society of Japan

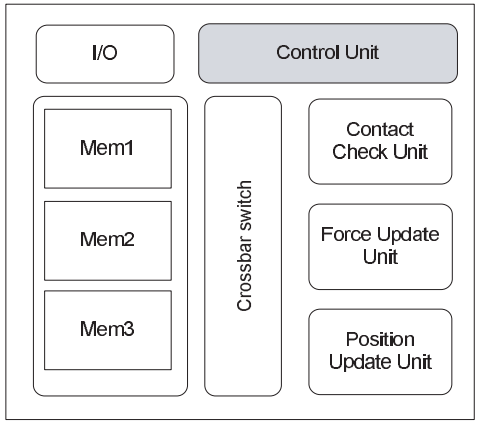

Fig. 8 Discrete Element Method (DEM) SoC design block diagram.

involved in it is also extraordinarily high. A dedicated hardware architecture implemented on a Xilinx XVC2000E FPGA was presented in Ref. 10) running at 40 MHz and using $\tilde{8}0\%$ of the CLBs. A block diagram of the architecture is shown in **Fig. 8**. The main operations to be performed in each time step are:

(1) Check which particles are in contact with one another
(2) Compute the inter-particle forces
(3) Update the co-ordinate and velocity of each particle

The control unit needs to generate the control signals for all 3 accelerator units in addition to transferring data in an out of the FPGA's internal memory overlapping communication and computation for further speed-up. Data needs to be read from the memory containing the particles concurrently for each of the units and written back. All four programs are executed at the same time in order to allow all four tasks to be executed in parallel. The control unit for this hardware implementation was initially designed as a FSM. It was re-designed using the generic and optimized micro-controller based control unit presented in this work. An RTL simulation was performed in order to validate our method an compared against the original FSM based control unit. **Table 2** shows the difference in size and speed between the different implementations.

As expected the state machine based control unit is the one that uses less hardware resources. Then comes the optimized micro-controller based control unit and lastly the generic micro-controller based control unit. The functionality of the original FSM could be completely be replicated comparing an RTL simulation of the original hard coded FSM vs. our proposed micro-controller based at the expense of a 7.2% and 15.4% area overhead for the optimized and generic controller version. The state machine based control unit is slower than the micro-controller based units because of some nested if clauses used to jump from one state to another. A re-design of these if clauses would probably improve the maximum frequency, but was not done for this design because the control unit was not the bottleneck of the design, which operated at 40 MHz.

Table 2 Comparison between different DEM control unit implementations.

	Area (CLBs)	Max. freq [MHZ]
FSM control unit	1,516	47
Generic micro-controller	1,792	63
Optimized micro-controller	1,635	67

Table 3 Control unit programs.

Unit Controlled	# lines assembly code
I/O	88
Contact Check	77
Force Update	82
Position Update	94

Table 3 shows the number of assembly lines of code used for each of the control programs. Up to three instructions where executed in parallel in some IEUs. No performance degradation was observed compared to the FSM approach because no input signals had to be serviced in this case, due to the predictability of the SoC behavior.

6.2 Control Unit Optimizer Analysis

In order verify the efficiency of the optimizer we decided to use synthetic benchmarks for our experimental results. These present multiple advantages over real benchmarks. First of all as many benchmarks as needed can be generated auto-

© 2010 Information Processing Society of Japan

matically. Secondly, they provide full control over the benchmark's most important characteristic parameters, such as circuit size, interconnection structure and functionality. The main advantage is the controllability of a single characteristic parameter at a time. The major drawback of synthetic benchmarks is that it is hard to prove that they are equivalent to certain real benchmarks.

For logic designs two parameters are extremely important to obtain realistic benchmarks: (a) The Rent's exponent and (b) the net degree distribution as explained in detail in Ref. 11). We extended the previous work on synthetic benchmarks to adapt it to the generation of control programs for our customizable micro-controller control unit. Specifically the user specifies the maximum number of instructions that can be executed in parallel, the size of the program (in terms of number o assembly code lines), and the *symmetry* of the program. Symmetry is defined as the difference between the parallelism of programs executed on different IEUs. At the initial architectural parametrization there is no way to specify fine grain parallelism. Therefore the original control unit is dimensioned to accommodate the maximum parallelism needed, but not all of the programs need the same amount of parallelism. High symmetry therefore means that each program executed on different IEUs exploits the maximum possible architectural parallelism, while small symmetry means that there is an asymmetry between the amount of parallelism in each program. Intuitively the more asymmetry the more architectural optimizations can be performed reducing the total controller area (logic and memory).

In order to test the efficiency of the control unit optimizer five benchmarks of 1,000 lines of assembly code each were generated with different degrees of asymmetry. The first benchmark has virtually no asymmetry and uses most of the instructions. The rest of the programs have increasingly a higher of asymmetry and they do not use the complete instruction set. **Table 4** shows the characteristics of each benchmark in term of the % of the instruction set used and the amount of asymmetry between the five programs executed concurrently in all of them.

Table 5 compares the area and block RAM used by the un-optimized and optimized control unit for the five different benchmarks described above. The un-optimized control unit is the same for all five benchmarks as it has the same

Table 4 Synthetic benchmarks description.

Name	#progr. concurrent	Instr. Set used [%]	Asymmetry [%]
Bench 1	5	100	0
Bench 2	5	80	10
Bench 3	5	60	20
Bench 4	5	40	30
Bench 5	5	20	40

Table 5 Optimized control units results.

	Un-optimized Control. Unit		Optimized Control Unit	
	Slices	BlockRAM [bits]	Slices	BlockRAM [bits]
Bench 1	562	5,184	504	5,184
Bench 2	562	5,184	398	4,625
Bench 3	562	5,184	374	4,320
Bench 4	562	5,184	288	3,862
Bench 5	562	5,184	225	3,456

number of instruction execution units and decoder units. **Figure 9** displays the results of Table 5 graphically. As predicted the amount of asymmetry in the different control unit programs impacts considerably on the amount of resources used by the control unit (CLBs and Block RAM). Other factors also increase the area savings of the optimized control unit like:

- The number of different instructions executed by every IEU as the decoder unit will not have to decode all the instructions, saving therefore more area.
- The output ports accessed by the different IEUs (output multiplexer will be simplified)

The Block RAM usage depends as pointed out in previous sections on the length of the programs and on the maximum number of instructions that need to be executed in parallel. The more asymmetry in the code the more Block RAM is saved by the optimizer as each program memory is optimized separately.

6.3 Conclusion

This paper presents a semi-automated way to generate generic control units

© 2010 Information Processing Society of Japan

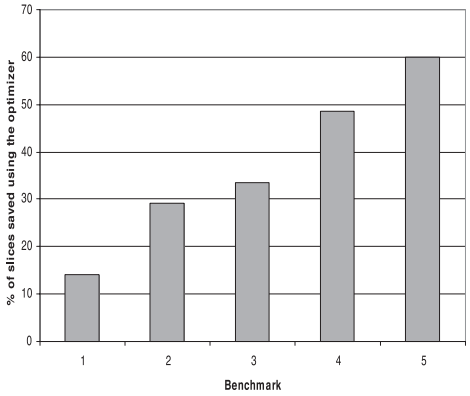

Fig. 9 Area savings due to control unit optimizer.

for SoC designs. The control unit generator is embedded in a system creator tool that allows the complete specification of top level designs. An unlimited number of input and output ports can be specified, as well as a virtually unlimited number of instructions can be executed in parallel. The user only needs to write an assembly program for the control unit. A simple instruction set and compiler was written to support the program generation for the control unit. As many programs as instantiated Instruction Execution Units can be executed in parallel. In order to validate the design and FPGA implementation compared to a previously developed FSM based control unit for an SoC is presented. The FPGA Block RAM is used as the program memory allowing to store multiple instructions in parallel to be decoded and executed in parellel. An optimizer was introduced in order to reduce the resources used by the controller. Results show that the more asymmetry there is in the different control unit programs executed concurrently the more resources can be saved compared to a generic control unit. It can be concluded that this parameterizable control unit is an easy and fast way to generate control units for SoC designs.The main limitation of this method is that a single centralized control unit might be inefficient for larger SoCs, causing timing degradation due to long wires and/or leading severe area and power penalties due to the exponential area increase of the crossbar switch for the control signals. To deal with these limitations an automatic partitioning step could be introduced, which would build a set of independent control units given the initial control unit description.

References

1) Tensilica. http://www.tensilica.com/
2) Target. http://www.retarget.com/
3) Silicon Hive. http://www.siliconhive.com/
4) Leupers, R. and Marwedel, P.: Retargetable Code Generation bsed on Structural Processor Descriptions, *Design Automatin for Embedded Systems*, Vol.3, No.1, pp.1–38 (1998).
5) Marwedel, P., Lanneer, D., Van Praet, J., Kifli, A., Schoofs, K., Geurts, W., Thoen, F. and Goossens, G.: CHESS: Retargetable Code Generation for Embedded DSP Processors, Code Generation for Embedded Processors, Kluwer Academics, pp.85–102 (1995).
6) Van Praet, J., Lanneer, D. and Goossens, G.: Processor Modeling and Code Selection for Retargetable Compilation, *ACM Trans. Design Automation Electronic Systems* (*TODAES*), Vol.6, No.3, pp.277–307 (2001).
7) Altera. http://www.altera.com/
8) Xilinx. http://www.xilinx.com/
9) Atmel. http://www.atmel.com/
10) Carrion Schafer, B., Quigley, S.F. and Chan, A.H.C.: Analysis and Implementation of the Discrete Element Method using a dedicated highly parallel Architecture in Reconfigurable Computing, *IEEE Symposium on Field-Programmable Custom Computing Machines* (*FCCM*), Napa Valley, California. IEEE Computer Society (2002).
11) Stroobandt, D., Depreitere, J. and Van Campenhout, J.: Generating new benchmark designs using a multi-terminal net model, *VLSI Journal*, Vol.27, pp.113–129 (1999).

(Received November 30, 2009)
(Revised February 9, 2010)
(Accepted April 14, 2010)
(Released August 16, 2010)

(Recommended by Associate Editor: *Akihisa Yamada*)

© 2010 Information Processing Society of Japan

Benjamin Carrion Schafer received the B.Eng. degree in electrical engineering from the Polytechnic University of Madrid, Madrid, Spain, the M.Sc. degree in microelectronics from Birmingham City University, Birmingham, U.K., and FH-Darmstadt, Darmstadt, Germany. After completing his Ph.D. at the University of Birmingham, he was a Postdoctoral Researcher with the Computer Science Department, University of California Los Angeles (UCLA), from 2003 to 2004. He was a Visiting Researcher at Seoul National University, Seoul, Korea, from 2005 to 2007 at the School of Electrical Engineering and Computer Science. Currently, he works as an Assistant Manager at NEC Corporation's R&D Central Laboratories, EDA Center, Kawasaki, Japan. He served on the TPC of CASES 2006, as a committee member at the RECONFIG conference and as a TPC at DAC's user track.

Majid Sarrafzadeh received his B.S., M.S. and Ph.D. degrees in 1982, 1984, and 1987 respectively from University of Illinois at Urbana-Champaign in Electrical and Computer Engineering. He joined Northwestern University as an Assistant Professor in 1987. In 2000, he joined the Computer Science Department at University of California at Los Angeles (UCLA). His recent research interests lie in the area of Embedded and Reconfigurable Computing, VLSI CAD, and design and analysis of algorithms. Dr. Sarrafzadeh is a Fellow of IEEE for his contribution to "Theory and Practice of VLSI Design". He has served on the technical program committee of numerous conferences in the area of VLSI Design and CAD, including ICCAD, DAC, EDAC, ISPD, FPGA, and DesignCon. He has served as Committee Chairs of a number of these conferences. He is on the executive committee/steering committee of several conferences such as ICCAD, ISPD, and ISQED. He was the Program Committee and the General Chair of ICCAD in 2004 and 2005 the premiere conference in CAD.

© 2010 Information Processing Society of Japan