EXECUTING HARDWARE AS PARALLEL SOFTWARE FOR PICOBLAZE NETWORKS

Pengyuan Yu, Patrick Schaumont

ECE Department, Virginia Tech, Blacksburg, VA 24061
email: peyu1983@vt.edu, schaum@vt.edu

ABSTRACT

Multi-processor architectures have gained interest recently because of their ability to exploit programmable silicon parallelism at acceptable power-efficiency figures. Despite the potential benefit they offer over single-processor architectures, it is unresolved how one can write compact and efficient programs for multiple parallel cores. In this paper, we propose the use of a synchronous hardware description language to program a network of small PicoBlaze processors. The partitioning of a multiprocessor program over multiple cores is straightforward because the input specification is fully parallel. A systematic transformation process converts the parallel input specification into concurrent PicoBlaze programs. We demonstrate the mapping of a cryptographic design (AES) onto four picoblaze processors, showing almost linear speedup over an equivalent single-core design.

1. INTRODUCTION

While parallel cores have been hailed as the next big step in computer micro-architecture development, it has not been obvious how to write efficient programs for multiprocessors. The sequential programming model of a single core is unable to address multiple cores at once. Designers therefore have to make use of programming extensions such as threads and message passing libraries, which quickly become cumbersome to use. In addition, a systematic transformation of a sequential program in C to a parallel program that can run on multiple cores is unknown, and it remains a highly skilled design activity.

We therefore looked at the use of a hardware description language as a means to program multiple cores. The advantage is that a hardware description language is inherently parallel. The mapping problem from specification to architecture thus does not require the detection of parallelism, but it rather does require an efficient sequentialization. In this paper, we will show a feasible solution for this problem. Hardware description languages are often associated with a low level of abstraction, unsuited for programming activities. This is indeed the case for contemporary hardware description languages such as VHDL and Verilog, which use event-driven simulation semantics that model each signal change separately. Therefore, we use a cycle-based hardware description language, called GEZEL.

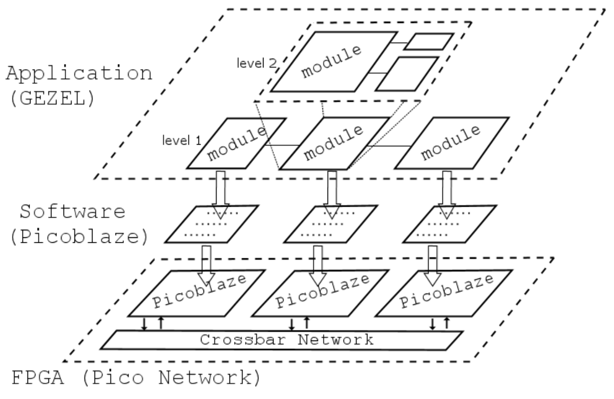


Fig. 1. GEZEL Hardware Description to Picoblaze Software

In this paper, we propose a transformation that will map synchronous hardware designs directly into optimized assembly programs that run on multiple embedded cores. The programs are fully synchronized with respect to each other by means of a common global clock, corresponding to the clock of the synchronous hardware description. Each cycle of this common global clock may take many physical clock cycles in the implementation. Compared to a direct implementation of the synchronous hardware design, we therefore expect a decrease in performance for the parallel software implementation. We will show that the reduction in performance is matched by a corresponding reduction in silicon area.

Our method shows excellent scalability over the number of cores in a design. In a fully parallel implementation, each hardware module of the input description maps to a separate core. Yet the same design can also map all hardware modules on a single global core. Our results show that this yields performance/area tradeoffs over a wider range than what is achievable by classic hardware design optimization methods, for example by logic synthesis.

```
Gezel_to_PicoNet_Mapping {
  for each FSMD_i {
    dfg = extract_Dataflow(FSMD_i);
    if is(hierarchical)
      for inner FSMD_j
        dfg_j = extract_Dataflow(FSMD_j);
        dfg = combine_Dataflow(dfg, dfg_j);
    instr = convert_to_instructions(dfg);
    instr = reorder_instructions(instr);
    regmap = register_assignment(dfg);
    assembly_i = mapping(regmap, instr);
  }
  synchronize(assembly_i, assembly_i+1,...);
}
```

Fig. 2. GEZEL to Picoblaze Mapping Algorithm

The starting point of our proposed transformation process are hardware models described using GEZEL[1]. As shown in Figure 1, GEZEL describes cycle-based hardware designs as a collection of parallel and hierarchical modules. Each module, including hierarchical ones, can be mapped to a sequential assembly program for an embedded core. In this way, a complete system can be mapped onto a network of parallel cores. The cores are attached to a crossbar network, which enables intermodule communication.

In our experiments, we used a Xilinx Spartan3 XC3S400 FPGA as testing platform and we chose Picoblaze microprocessors for our multi-core network. Picoblaze is an 8-bit soft-core developed by Ken Chapman, Xilinx Inc. It uses 1 BlockRAM and 96 slices of logic. This low resource usage enables typical FPGA to carry multiple Picoblaze cores. Our XC3S400 FPGA for example, can hold up to 16 cores including the crossbar network.

The remainder of this paper is organized into five sections. In section 2, related work in this research area is discussed. Section 3 presents the conversion process, and section 4 introduces a case study based on a cryptographic processor design. Section 5 presents the results obtained from these experiments, and section 6 concludes the paper.

2. RELATED WORK

The customizability and small size of Picoblaze makes it an ideal processor for many applications that are area-critical. Often, Picoblaze cores are used in small control applications. A floating-point controller[2] and neuron-network for self-learning[3] have used original or customized Picoblaze cores. Integrating Picoblaze processors into the system for each of these task-dependent applications is a cumbersome process. With our transformation process, we can map synchronous systems into Picoblaze processors as efficiently as possible independent of design tasks.

Threads are by far the most popular abstraction for parallel software programming. Lightweight implementations such as protothreads[4] and quickthreads[5] are well suited for embedded implementation. However, threads are not an easy abstraction, and they tend to make the design of real-time systems needlessly complicated[6]. Programming mechanisms such as Esterel, nesC[7], and Click[8] avoid the use of threads yet provide elegant parallel programming. Our approach falls in this latter category, however it differs from them in the way it explicitly models the progression of time as 'clock cycles' in the input description. In those other approaches, time is modeled implicitly instead.

3. MAPPING PROCESS

In this section we introduce the mapping process to convert a synchronous hardware description into parallel software. We first provide an overview of the conversion process, and illustrate our method with the step-by-step conversion of a 4-stage pipeline. Further subsections present the details of the conversion process, including register assignment, operation scheduling, operation mapping, and synchronization of multiple Picoblaze programs.

3.1. Input Specification and Mapping Algorithm

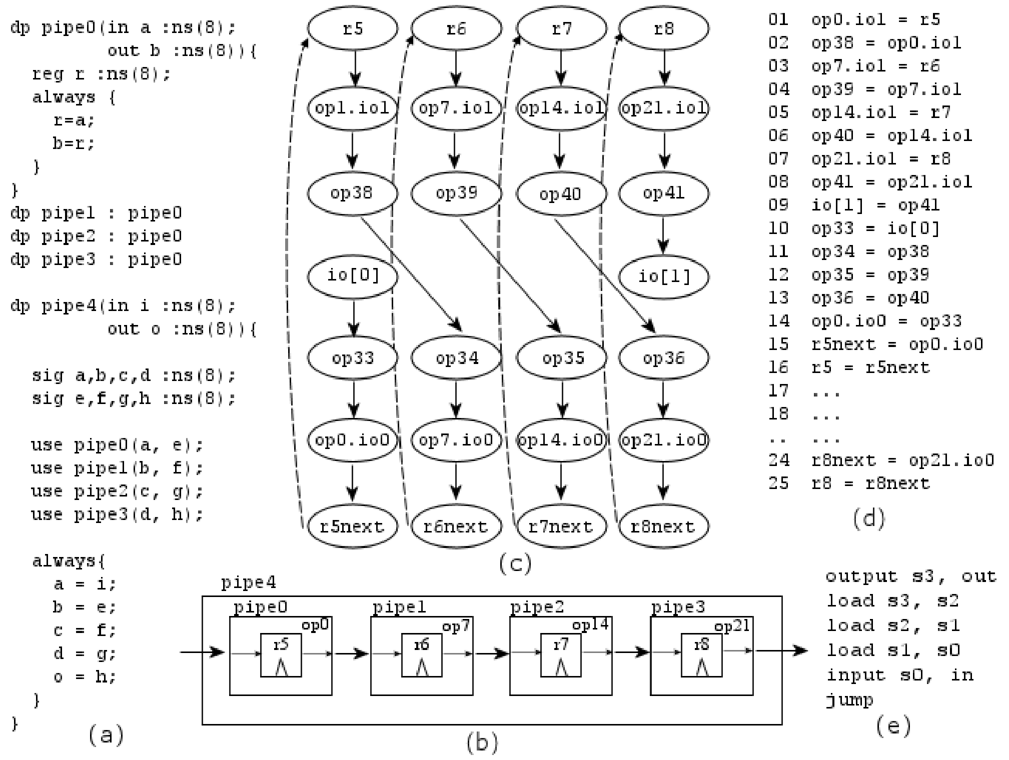
Hardware modules in GEZEL are described using the Finite State Machine with Datapath(FSMD) model-of-computation. Our method is to translate each GEZEL FSMD, or each hierarchy of FSMDs, into an assembly program for a Picoblaze processor. The pseudocode of the mapping algorithm is shown in Figure 2.

In our current approach, we assume that all picoblazes are connected to a crossbar network. System-level communications that cross the boundaries of an FSMD will therefore translate into communications over the crossbar network. We now discuss the details of each step.

3.2. Dataflow Graph and Converted Instructions

The first step of the conversion is to express each GEZEL FSMD as a dataflow graph (DFG). These are directed graphs in which nodes capture operations and edges represent data dependencies. The input and output nodes of the graph correspond to input/output operations of an FSMD. Hierarchical FSMDs are captured in a single DFG by flattening them first. In the DFG, this has the effect of connecting input/ouput operations of lower-level FSMD with additional data precedences.

Figure 3(a) lists the GEZEL description of a 4-stage pipeline with hierarchical structure, containing 4 single-stage pipeline modules. The block diagram is shown in figure 3(b). All variables in GEZEL are assigned a single node with an unique number. GEZEL state variables are represented with 2 nodes:



```
dp pipe0(in a :ns(8);
         out b :ns(8)){
  reg r :ns(8);
  always {
    r=a;
    b=r;
  }
}
dp pipe1 : pipe0
dp pipe2 : pipe0
dp pipe3 : pipe0

dp pipe4(in i :ns(8);
         out o :ns(8)){

  sig a,b,c,d :ns(8);
  sig e,f,g,h :ns(8);

  use pipe0(a, e);
  use pipe1(b, f);
  use pipe2(c, g);
  use pipe3(d, h);

  always{
    a = i;
    b = e;
    c = f;
    d = g;
    o = h;
  }
}
```

```
01  op0.io1 = r5
02  op38 = op0.io1
03  op7.io1 = r6
04  op39 = op7.io1
05  op14.io1 = r7
06  op40 = op14.io1
07  op21.io1 = r8
08  op41 = op21.io1
09  io[1] = op41
10  op33 = io[0]
11  op34 = op38
12  op35 = op39
13  op36 = op40
14  op0.io0 = op33
15  r5next = op0.io0
16  r5 = r5next
17  ...
18  ...
..  ...
24  r8next = op21.io0
25  r8 = r8next
```

```
output s3, out
load s3, s2
load s2, s1
load s1, s0
input s0, in
jump
```

Fig. 3. (a) GEZEL description of a 4-stage pipeline using hierarchy. (b) Block Diagram of GEZEL description (c) Flattened Dataflow Graph extracted. (d) Collected sequential operations. (e) Picoblaze assembly code

one for the current state and one for next state with same number appended with word "next". In figure 3(c), the flattened data-flow graph of the pipeline is shown. After the dataflow graph is constructed, it is converted to a sequence of instructions that simulate the original GEZEL system. A sequence consisting of 25 instructions is created from the dataflow graph, as shown in figure 3(d). The scheduling of the operations in the data-flow graph to a sequential time axis can be done in different ways. We are using an ASAP (as soon as possible) scheduling strategy with some additional heuristics as will be discussed later. First, we explain how the register assignment works.

3.3. Register Assignment

One problem with generating low-level assembly code is to assign GEZEL wires and state variables to Picoblaze registers. Picoblaze has 16 byte-wide general purpose registers and 64-byte of scratchpad RAM. Operations involving scratchpad RAM will require additional write/read operations (so-called 'spilling'), resulting in costly access. We need to optimize the use of registers and resort to scratchpad only when absolutely necessary. For register allocation, we choose to used the left-edge algorithm[9] with additional heuristics for optimized register assignment.

3.3.1. Heuristic 1: Keep GEZEL Data State variables in a single Picoblaze register

As seen in the dataflow graph in figure 3(c), the dashed edges making the graph cyclic represent negative precedences, which occurs between two nodes that represent a data state. We use one Picoblaze register to represent each data state. Each iteration of the Picoblaze program uses the current state values from these registers and stores the next state values into the same registers. Shown in figure 4(a), the first 8 variables representing 4 GEZEL data states are all forced to the same Picoblaze register before left-edge algorithm starts. This heuristic removes 1 instruction for each GEZEL data state update and potentially reduces the number of Picoblaze registers in use at any time of execution.

3.3.2. Heuristic 2: Same register allocation for consumed operand and created result

All Picoblaze instructions operate on two operands except for shifting instructions. The result of an operation is stored into the register containing the first operand. During register assignment, we need to assign the same register to both result and first operand. If the first operand is still needed after this instruction, it is possible to switch the two operands if second operand is consumed by this operation. For instruc-

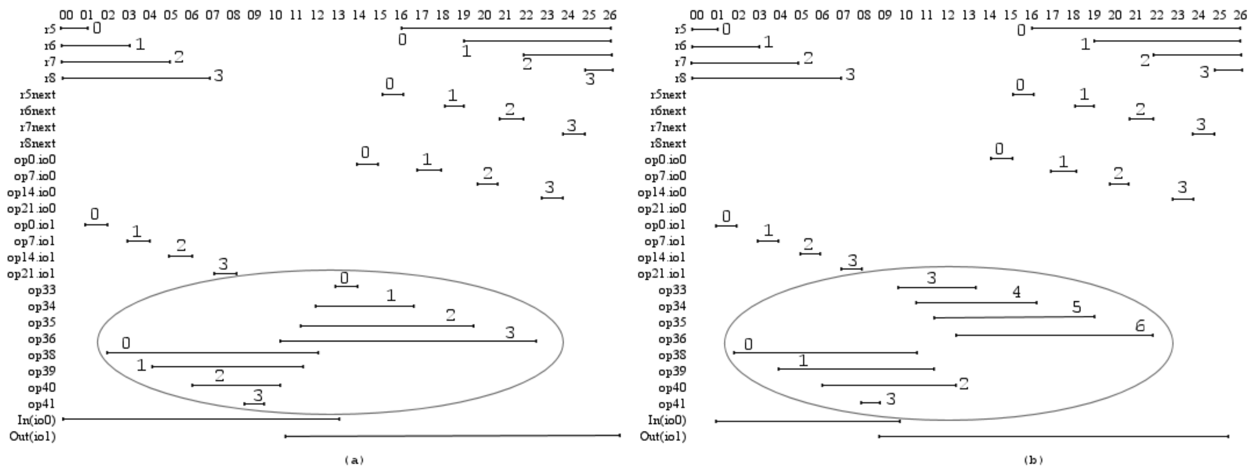


Fig. 4. (a)optimized register assignment with reorder of instructions (b) non-optimized register assignment based on original instruction sequence

tions that are not commutative, like the subtraction operation, the content of the first register needs to be preloaded into a temporary register, and subtraction can proceed with the temporary register as the first operand. This prohibits assigning same registers to result and first operand variables.

The Left-Edge algorithm works on the birth-death table of variables. Obviously, the operation schedule will affect the birth and death time of variables. At least one schedule will result in a register assignment that minimizes the number of instructions needed for the mapping process. Taking this observation into account, we discuss some heuristics for the rearrangement of execution sequence that will optimize the register assignment.

3.4. Reordered Execution Sequence

Consider again the design in figure 3. If we don't change the execution order as defined in 3(d), we obtain a sub-optimal solution, using 10 instructions with 7 registers. By rearranging instruction #10 to #13 and executing them backward, we can obtain an optimal solution that will simulate above pipeline in only 6 Picoblaze instructions with 4 Picoblaze registers as shown in Figure 3(e). The register assignment comparison of before and after reordering of the execution sequence is shown in Figure 4.

To simplify the register assignment process, several heuristics have been used in the execution sequence reordering:

- ASAP scheduling starts with nodes representing current state values only. This will free up Picoblaze registers early for use in other instructions. When parallelism is encountered, priority is given to instructions closer to producing system outputs.
- All next state instructions are moved toward the end of the program, to prevent negative precedence rule from locking up registers.
- Inputs need to be stored into registers before arithmetic operations such as ADD, SUB. Once loaded, they need to be consumed as fast as possible. To achieve this, group all instructions that depend on inputs together. Push this group of instructions as far back as possible, right before next-state assignment instructions. This will give more freedom in register assignment for operations that do not depend on inputs.

After the register assignment is optimized with reordering of instructions, the operation mapping starts.

3.5. Operation mapping

Mapping starts by taking in the optimized register assignment and the set of reordered instructions. Using a translation table, most operations in GEZEL can be translated to Picoblaze assembly using 1 or 2 instructions. Certain operators supported in GEZEL cannot be translated efficiently. Such operations are for example multi-bit extraction. These require shifting and bit masking and are costly to implement in PicoBlaze. Other costly operations are multiplication and modulo-operators.

3.6. Synchronization

To ensure the correct operation of the mapped Picoblaze network, we need to synchronize the Picoblaze programs. Every Picoblaze instruction executes in 2 cycles. This predictable behavior makes static synchronization possible. The two phases of synchronization are shown in Figure 5.

Phase One is internal synchronization. This phase is required when branching occurs during execution. Data-

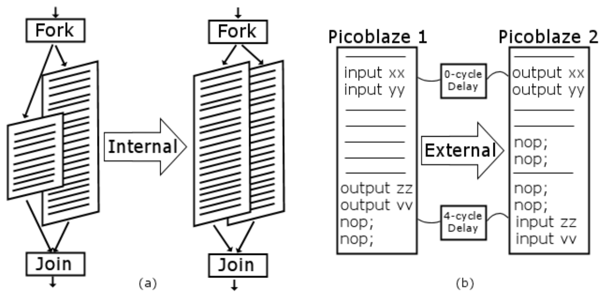


Fig. 5. Two types of synchronization: Internal and External

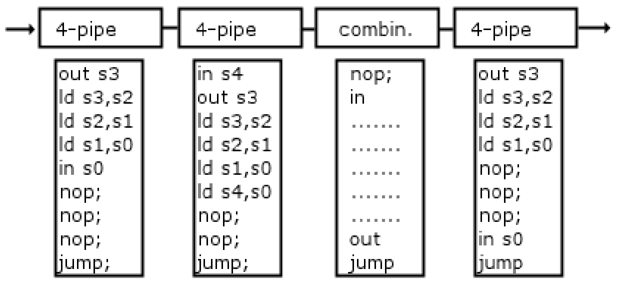


Fig. 6. Synchronization With Respect to Combinational Circuits

dependent branching instructions vary the program execution length. We need to balance the two branches by inserting fake instructions (NOP). As shown in Figure 5(a), internal synchronization ensures that the program always takes the same amount of clock cycles to execute from the fork point to the join point.

Phase Two is external synchronization. Static synchronization of multiple Picoblaze programs means that communication will not use explicit handshaking. During the output instruction of transmitter, the receiver Picoblaze must execute an input instruction. If delay exists in the communication network, it has to be in multiples of 2 cycles. For $2n$ delay, the input instruction in the receiver can be delayed by n instructions. In figure 5(b), the top part of external synchronization shows no delay, where each output instruction is matched with an input instruction. In the bottom part, a 4 cycle delay permits input instructions to be offset by 2 instructions.

A combinational circuit requires current input before any processing can be done. Synchronization with combinational circuits is troublesome because the positions of input and output instructions are fixed by the input-output data dependency. For a sequential circuit such as the 4-stage pipeline, sacrificing performance by adding one more instruction provide a better overall performance. In figure 6, a combinational circuit that requires 8 instructions is inserted into a 12-stage pipeline build using 3 4-stage pipelines.

For the second 4-stage pipeline, instead of executing the same as the first pipeline, we can input data to a temporary register and load it to the correct register later. This allows us to move the new input instruction anywhere in the program space before the last load instruction. From Figure 6, we see that for the same 4-stage pipeline, by using 1 extra instruction like the second pipeline, we can efficiently synchronize sequential circuits with each other. With combinational circuit in the picture, we have to synchronize the sequential circuits with respect to the combinational circuit. Unfortunately, if 2 combinational circuits are connected directly, then no optimized synchronization can be done between these 2 blocks.

4. CASE STUDY WITH AES

Advanced Encryption Standard (AES)[10] is a block cipher algorithm that can be implemented quite efficiently on a byte-wide processor. Its modular structure also makes it ideal for running on a Picoblaze network.

4.1. Picoblaze Network

In order to run AES on multiple Picoblaze cores, we need to efficiently allocate the plaintext and userkeys into each core. In our example, we used 4 Picoblaze processors. We will use column allocation so MixColumn operations can be done directly while ShiftRow operations will be routed on the network. Shown in figure 7 is the Picoblaze network structure and AES data and key states distribution.

4.2. AES Mapping

Sbox requires 256 bytes of storage space. This forces us to use synchronous BlockRAM on our XC3S400 FPGA to implement ROM storage. Each ROM storage can be connected to either 1 or 2 Picoblaze cores using single-port or dual-port configuration. Each Picoblaze has a 8:1 multiplexor selecting different inputs. Four inputs are connected to the 4 outputs of ROM. Three inputs are connected to the Picoblaze outputs other than itself. The last input is for new data.

ShiftRow operation can be combined with Subbyte operation because the ROM outputs can be routed to any Picoblaze processors. On the fly key-expansion takes almost 50% of runtime. This is because key expansion is not parallel since each column of key depends on previous columns. This creates a combinational circuit behavior that does not map to Picoblaze as efficiently as sequential circuits.

5. RESULTS

The optimized mapping of AES in Picoblaze assembly produces competitive results against other 8-bit processors. The resulting cycle count is much less than other 8-bit processors

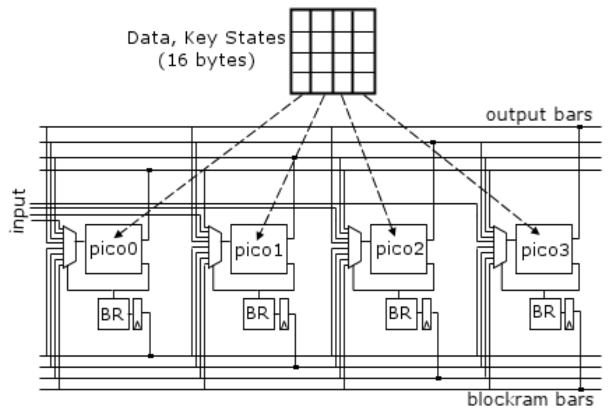


Fig. 7. Picoblaze architecture in cross-bar topology. Each Picoblaze can communicate with any other Picoblaze.

Table 1. AES on 8-bit processors

	Instructions	Cycles/Instr.	Cycles
8051	3168[12]	12	38016
68HC08	8390[12]	1	8390
AVR	4009[13]	1	4009
1 Pico*	3270	2	6540
2 Pico*	1304	2	2608
4 Pico	791	2	1582

*estimated numbers

such as Intel 8051, Motorola 68HC08 and Atmel AVR. The results are shown in Table 1.

Compared with above 8-bit processors, running AES on 4 Picoblaze cores is considerably faster, using less than $1/5$ of the cycle count for 68HC08, $2/5$ of the cycle count for AVR, and less than $1/20$ of the cycle count for 8051 processor. Cycle counts are estimated for 2-Picoblaze and 1-Picoblaze network. The cycle counts are not linear because we have to use scratchpad memory in both cases and key expansion does not scale well with the number of processors.

The FPGA implementation of an AES algorithm will take between 3000 to 10000 slices depending on the degree of unrolling and pipelining[11]. The performance is between $200Mbits$ and $2Gbits$. In comparison, simulation in 4-Picoblaze network is considerably slower. Using 4 Picoblaze in crossbar network for AES, the area is estimated to be 500 slices. This is a lot smaller than the solutions analyzed in [11].

6. CONCLUSION AND FUTURE WORKS

This paper shows a method to convert hardware descriptions into software for parallel processors. The result shows excellent scalability over the number of cores in terms of area and performance. Hardware designers, who are used to design parallel behavior, can make use of a tool to convert hardware into parallel software. In case of AES, where data flow is a more important task than control, parallel Picoblaze produces reasonable and better result than other similar processors.

This mapping process is not complete yet. We are working on the automation of entire process. Currently, automated conversion for longer word length and multiple input/output are not optimized. In the future, additional features such as combining smaller FSMDs into bigger FSMD to match the result with other big FSMD will be explored.

7. REFERENCES

[1] The GEZEL Design Environment, Virginia Tech. [Online]. Available: http://rijndael.ece.vt.edu/gezel2

[2] J. Kadlec and R. Gook, “Floating point controller as a picoblaze network on a single spartan 3 fpga,” in *MAPLD05*, 2005.

[3] J. A. Starzyk, Y. Guo, and Z. Zhu, “Dynamically reconfigurable neuron architecture for the implementation of self-organizing learning array,” in *Proceedings of the 18th IPDPS*, 2004.

[4] A. Dunkels, “The protothreads library.” [Online]. Available: http://www.sics.se/ adam/pt/documentation.html

[5] D. Keppel, “Tools and techniques for building fast portable threads packages,” in *UWCSE 93-05-06*. U. Washington, 1993.

[6] E. A. Lee, “Absolutely positively on time: what would it take?” *IEEE Trans. Comput.*, vol. 38, no. 7, pp. 85–87, 2005.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “nesc language: A holistic approach to networked embedded systems,” in *In Proceedings of Programming Language Design and Implementation (PLDI)*, 2003.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” in *ACM Transactions on Computer Systems 18(3)*, August 2000, pp. 263–297.

[9] T.-Y. Wu and Y.-L. Lin, “Register minimization beyond sharing among variables,” in *Proceedings of 32nd Design Automation Conference*. ACM, 1995, pp. 164–169.

[10] AES Rijndael Algorithm Specification. [Online]. Available: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf

[11] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar, “An fpga-based performance evaluation of the aes block cipher candidate algorithm finalists,” *IEEE Trans. VLSI Syst.*, vol. 9, no. 4, pp. 545–557, Aug. 2001.

[12] J. Daemen and V. Rijmen, “The block cipher rijndael,” in *Smart Card Research and Applications. Third International Conference, CARDIS’98*, ser. Lecture Notes in Computer Science, vol. 1820. Springer, 2000, pp. 277–284.

[13] AES Software Modules for Atmel AVR Microcontrollers. [Online]. Available: http://jce.iaik.tugraz.at/sic/products /c_products/crypto_software_for_microcontrollers