Hardware/Software Design Space Exploration for a Reconfigurable Processor

Alberto La Rosa Luciano Lavagno Claudio Passerone

Dipartimento di Elettronica, Politecnico di Torino, Italy

E-mail: {larosa,lavagno,passerone}@polito.it

Abstract

This paper describes an approach to hardware/software design space exploration for reconfigurable processors. The existing compiler tool-chain, because of the user-definable instructions, needs to be extended in order to offer developers an easy way to explore design space. Such extension often is not easy to use for developer that have only a software background, thus ignoring reconfigurable architecture details or hardware logic synthesis on FPGA. Our approach differs from others because it is based on a simple extension on the standard programming model well known to software developers.

1. Introduction

Reconfigurable computing is emerging as a promising mean to tackle the ever-rising cost of design and masks for Application-Specific Integrated Circuits. Adding a reconfigurable portion to a Commercial Off The Shelf IC enables it to potentially support a much broader range of applications than a more traditional ASIC or microcontroller or DSP would. Moreover, *run-time reconfigurability* even allows one to adapt the hardware to changing needs and evolving standards, thus sharing the advantages of embedded software, with higher performance and lower power than a traditional processor, thus partially sharing the advantages of custom hardware.

A key problem with dynamically reconfigurable hardware is the inherent difficulty of programming it, since neither the traditional synthesis, placement and routing-based hardware design flow, nor the traditional compile, execute, debug software design flow directly support it. In a way, dynamic reconfiguration is a hybrid between software, where the CPU is "reconfigured" at every instruction execution and memory is abundant, and hardware, where reconfiguration occurs seldom and very partially, and memory is a scarce resource. Several approaches, some of which are summarized in the next section, have been proposed to tackle this problem, both at the architecture and at the CAD level.

In this case study, we consider an architecture ([4]) in which an FPGA has been added as one of the functional units of a RISC processor. This approach has one huge advantage, that it can exploit, as discussed in the rest of this paper, a standard software development flow. The FPGA can be managed directly by the compiler, so today the designer just needs to tag portions of the source code that must be implemented as single FPGA-based instructions. In the future, we are looking at ways of automating this extraction. The disadvantage of the approach is that the reconfigurable hardware accesses its main data memory via the normal processor pipeline (load and store instructions), thus it partially suffers from the traditional processor bottleneck. Without a direct connection to main memory, we avoid to face memory inconsistency problems ([7] [3]), thus simplifying the software programming model.

In this paper we show that, by efficiently organizing the layout of data in memory, and by exploiting the dynamic reconfiguration capabilities of the processor, one can obtain an impressive speed-up (better than 3X) for a very compute-intensive task, like turbo-codes. Power is also dramatically reduced due to the reduction in the number of operations (less accesses to instruction memory subsystem), and to the more optimized datapath layout that can be achieved within an FPGA. It is well-known, in fact, that processors are power-hungry due to the high number of high-capacitance connections that are required by their very flexible and frequent "reconfiguration". An FPGA implementing several processor instructions on a customized datapath goes in the same direction as DSPs, in order to reduce this capacitance and the number of its switchings. It also helps reducing expensive memory accesses due to register spills, thanks to the embedded flip-flops within the FPGA array.

The goal thus is to bridge the huge distance between software and hardware, currently at several orders of magnitude, in terms of cost, speed and power. The result is a piece of hardware that is reconfigured every 100 or so clock cycles (at least in the application described in this paper; other trade-offs are possible for applications with a more static control pattern), and within that interval is devoted to a single computation task. Ideally, we would like to bring a traditional, hardware-bound task, that of decoding complex robust codes for wireless communication, within the "software" computation domain.

In the case of our target processor and software/hardware development flow, the design space exploration takes the form of identifying maximally time-consuming portions of the code, bounded by a number of accesses to memory and/or registers that is within the limit of one FPGA-based instruction. In this case, Data Flow Graph fragments with at most 4 inputs and 2 outputs, for a total of 4 input/outputs (due to encoding limits of the

1530-1591/03 $17.00

DLX Instruction Set Architecture) are the objective of the search. Once they are identified, possibly with the help of (currently manual) code restructuring, they can be tagged, and their impact on the overall execution time analyzed.

The rest of the paper is organized as follows: Section 2 describes previous works in the area; Section 3 illustrates the target processor architecture and the software toolchain that we developed; Section 4 describes the methodology that we are advocating for hardware/software development on a reconfigurable architecture; Section 5 shows an example taken from an UMTS turbo-decoder, and finally Section 6 concludes the paper.

2. Related work

Coupling a general purpose processor with an FPGA generally requires to embed them on the same chip, otherwise communication bandwidth limitations may severely impact the performance of the entire system. There are several ways of designing the interaction between them, but they can be grouped into two main categories:

1. the reconfigurable array is another functional unit of the processor pipeline;
2. the reconfigurable array is a co-processor communicating with the main processor.

In both cases, the FPGA can be reprogrammed on the fly during execution to accommodate new kernels, if the available space does not permit to have all of them at the same time. Often, a cache of configurations is maintained close to the array to speed up this process.

In the first category we find designs such as PRISC [10], Chimaera [6, 13] and ConCISe [8].

The second category includes the GARP processor [2, 3]. Since the FPGA array is external, there is an overhead due to the explicit communication using dedicated instructions that are needed to move data to and from the array. But because of the explicit communication, the control hardware is less complicated with respect to the other case, since almost no interlock is needed to avoid hazards. If the number of clock cycles per execution of the array is relatively high, then the overhead of communication may be considered negligible.

In all the cited examples a modified C compiler is used to program the processor and to extract candidate kernels to be downloaded on the FPGA. In particular, the GARP compiler [3] and the related Nimble compiler [12] identify loops and find a pipelined implementation, using profiling-based techniques borrowed from VLIW compilers and other software optimizations.

Tensilica offers a configurable processor, called Xtensa, where new instructions can be easily added at design time within the pipeline. Selection of the new instructions is performed manually by using a simulator and a profiler. When the Xtensa processor is synthesized, a dedicated development tool-set is also generated that supports the newly added instruction as function intrinsics.

A Tensilica processor is specialized for a given algorithm at fabrication time. On the other hand, a reconfigurable processor such as the XiRisc can be customized directly by the software designer. Of course, this added flexibility has a cost: an FPGA-based implementation has area, delay and power cost that is about 10 times larger than an ASIC implementation, such as Tensilica.

The Lisa tool-set [9] was designed specifically to facilitate the development of Application-Specific Instruction Processors, because it generates a complete tool chain from a single specification of the ASIP's instruction set. However, it is not clear from the published literature whether generation of a new suite supporting a new instruction is easy and fast enough to be used in a tight loop during simultaneous design space exploration of the application and of the underlying reconfigurable processor's instruction set.

Our approach is similar to the Tensilica one, in that we rely on manual identification of the extracted computational kernels. However, we do not require re-generation of the complete tool chain whenever a new instruction is added. We also provided support for new DSP-like instructions and for a VLIW-like datapath.

3. The software development tool chain

3.1. Processor Architecture

We are targeting a reconfigurable processor that is being developed by the University of Bologna and is described in [4]. Here we will briefly outline the main characteristics, that are useful to understand the rest of the paper.

The processor is based on a 32-bit RISC (DLX) reference architecture. However, both the instructions set and the architecture have been extended to support:

- Dedicated hardware logic to perform multiply-accumulate calculation and end-of-loop condition verification, by using special DSP-like instructions added to the standard DLX ISA.
- A double data-path, with concurrent VLIW execution of two instructions. The added data-path is limited to only arithmetic and logic instructions, and cannot directly access the external memory subsystem. However, both data-path can simultaneously access the register file, which supports four read and two write operations. The two ALUs are fully bypassed to avoid data dependencies.
- An FPGA to implement in hardware special kernels and instructions. The FPGA can be dynamically reconfigured at run-time; each cell may store up to 4 configurations that can be instantly switched when needed. Depending on the FPGA configuration, it is possible to realize complex user-defined pipelined data-paths that can have variable latency and throughput.

3.2. The modified *gcc* toolchain

We have implemented a design flow to support the reconfigurable processor. It starts from a *fully C* initial specification, where sections that must be moved to the FPGA

are manullay annotated, and automatically generates the assembler code, the simulation model, and a hardware model useful for instruction latency and datapath cost estimation. A key characteristic is that it supports compilation and simulation of software including user-definable instructions, without the need to recompile the tool chain every time a new instruction is added.

The design flow is based on the *gcc* toolchain, because it is freely available for a large number of embedded processor architectures, and features state-of-the-art optimization capabilities. This section illustrates the main changes that were needed to support the reconfigurable processor.

3.2.1. The Compiler

We retargeted the compiler by changing the machine description files found in the `gcc` distribution, to describe the extensions to the DLX architecture and ISA.

In order to describe the availability of the second datapath, we doubled the multiplicity of all existing functional units that implement ALU operations. The presence of the reconfigurable unit was modeled as a new function unit. To support different user-defined instruction on the FPGA unit, we classified the FPGA instructions according to their latency. Thus the FPGA function unit was defined as a pipelined resource with a set of possible latencies.

3.2.2. The Assembler

The `gcc` assembler is responsible for the expansion of macro instructions into sequences of machine instructions, the scheduling of machine instructions to improve latency in the presence of shared resources and multi-cycle instructions, and the generation of binary object code. We had to modify both the scheduling and the generation of the object code to support the processor architecture.

We modified the scheduler to handle the second datapath. Since only the ALU has been duplicated, only instructions such as arithmetic, logical, and relational operations, can be performed in parallel. All those instructions that require non duplicated units, such as the DSP-like instructions, loads, stores, multiplications, jumps and branches must be issued one at a time, by inserting a `nop` or a non-data-dependent ALU operation in the second issue slot. Moreover, some static scheduling to avoid the parallel issue of two instructions that read and write the same register is required. This means that the instruction scheduler in the assembler inserts enough `nop`s to avoid such hazards at compile time.

We also modified both the assembler instruction mnemonics and their binary encodings to add two classes of instructions: one for the DSP instructions (multiply/accumulate and decrement/branch), that are treated just as normal instructions and assigned some of the unused opcodes, and another for the FPGA instructions.

3.2.3. The Simulator and Debugger

The standard `gdb` distribution already contains a flexible interpreted instruction set simulator, that however has only limited performance analysis capabilities, and does not support the second data-path, the DSP-like instructions, nor the on-board FPGA. The simulation models for DSP instructions were permanently added to the simulator, since they are a fixed part of the architecture. On the other hand, FPGA instructions depend on the application, and thus cannot be statically linked.

Simulating FPGA Instructions To simulate an FPGA instruction we use a C model of its behavior, which is called at simulation time when the instruction is encountered. The latency of the instruction is also specified by the model, and it is used when computing the performance. Obviously, when the object code for download on the target processor is generated, the simulation model is removed.

An FPGA instruction is identified in the source code using a pragma directive. The format used inlcudes the mnemonic name of the instruction, its opcode, the latency in clock cycles, the number of outputs and inputs, and list of output and input variables, in this order. A simple shift and add operation can be modeled in the following way:

```
#pragma fpga shift_add 0x12 5 1 2 c a b
  c = (a << 2) + b;
#pragma end
```

The annotated source code is then processed to generate a new C program where, for each FPGA instruction, we define two assembler macros, one to be used to generate code for the target architecture, the other to be used to generate code for the simulator, and a function, which is called only during simulation.

The macro defined for the FPGA simply specifies the correspondence between variables and operands, plus the opcode as an immediate operand, and the latency. The one for the simulation model requires some special handling: we use a fixed set of processor registers to pass the inputs to the corresponding simulation function, to get back outputs, and to return to the simulator the actual (possibly data-dependent) cycle count for the instruction. A shadow register file saves the processor registers upon entering the instruction simulation routine, and restores them at the end, so none of the argument passing registers, nor those used by the routine, need to be saved by the caller of the FPGA instruction. Two special instructions, called `tofpga` and `fmfpga`, are used to handle the shadow register file.

The function generated to implement the specified behavior is simply the code extracted from the user-annotated section, preceded and followed by a sequence of assembler instructions that transfers the input and output data from and to the argument passing registers. The last assembler instruction also loads a pre-defined register with the total cycle count, to be used during profiling.

Simulating VLIW Instructions The simulator provided with `gcc` is able to simulate a processor architecture with a single pipeline chain. Our objective was to simulate VLIW instructions that run in parallel on the double data

path of the processor, and to simulate the extended instructions using the shared functional units that were added to the ISA.

Since the target architecture is almost fully bypassed, and since the assembler already prevents possible structural hazards to occur by inserting `nop` instructions for the added pipeline chain (see Section 3.2.2), it is sufficient to simulate instructions sequentially even though they are executed in parallel on the processor. This is a very important assumption because it dramatically simplifies the changes required to the simulator.

3.2.4. Performance Simulation

We assume that the processor never stalls, since most hazards can be already resolved at compilation time. Therefore, it is sufficient to increment the cycle count at each instruction that is simulated, with two notable exceptions:

1. FPGA instructions should increment the clock cycle count by the amount that their simulation code returns in a pre-defined register.

2. Two parallel VLIW instructions are executed in the same clock cycle. Therefore, only one of them should increment the clock cycle count.

We also developed a simple profiler which takes a trace as input and generates the total number of clock cycles of the simulation run, the total number of clock cycles spent for each different instruction (opcode), and the total number of clock cycles spent in each line of the source code.

4. Our approach to design exploration

Comparing reconfigurable processors with traditional ones, software developers have to adopt an extended programming model that includes reconfigurable units besides their functional specification.

Such kind of extensions often require developers to provide architecture specific items and to call user-defined instructions directly through assembler code, thus lowering the abstraction level provided by generic programming languages like C or C++. As consequence, the main drawback of such approaches is that software developers need to master reconfigurable processor details and hardware description languages before being productive in writing software for such kind of computer architectures.

The main goal of compiler tools is that to hide architecture details, by providing a standardized programming model (variables, arithmetic and logical operators, control constructs and functions and procedure calls) that permits code reuse among different computer architectures.

Our approach to reconfigurable computing tries to extend the standard programming model to include easy and transparent use of the reconfigurable instruction set.

The design flow is divided in two main phases:

- *Design space exploration*: here software developers analyze existing source code (often written without specific knowledge of the target processor), and identify groups of statements that may be implemented as user-defined instructions on the FPGA. Each new instruction is characterized by cost figures (latency, power, number of required CLB) and a software functional model, later used in ISS simulation for performance estimation and functional verification. In this way several hardware/software partitions can be explored and compared.

- *Software and hardware synthesis*: this phase represents the final link to the target architecture, when the binary code and the bit-stream for FPGA is generated through software and hardware compiler tools. Starting from the functional model of each user-defined instruction, a netlist of gates is synthesized and mapped on FPGA CLBs. In this step real cost figures are computed. Then the software compilation chain can be used again, this time with real cost figures to optimize instruction scheduling and register allocation.

The design exploration methodology that we propose is primarily aimed at optimizing performance in term of speed, making use of a limited number of FPGA cells. The reduction in number of executed instructions and memory accesses also has a very beneficial effect on energy consumption.

The first step consists of identifying which statements within the application, if implemented in hardware, may improve performance. We use common profiling tools (like `gprof`), analyzing the timing and the number of executions of each function. After selecting the most time consuming functions, a further profiling step annotates each line of source code with the number of cycles spent in it during execution. Thus we identify which statements need further investigation and are possible candidates to become user-defined instructions.

In most cases, the chosen group of statements is a computation kernel inside a loop. Such kernels are surrounded by code fragments that access data memory to read input values and to store results. Considering memory accesses only, we distinguished those that read input values and write output data, from those which read and write intermediate values during kernel computation. The first set of accesses depends on the application and on its memory data structure, while the second set is dependent of the computation kernel and its implementation.

At this point two different kinds of design exploration paths are possible:

- implement in hardware the entire computation kernel (or part of it) using the FPGA.

- optimize memory data structures in order to minimize memory accesses for input and output or intermediate values [5].

After extracting the control-data flow graph (CDFG) of each identified kernel, we decompose it into non overlapping subgraphs that can be implemented in hardware as a set of single FPGA instruction. In our target architecture the FPGA unit can receive up to four input operands from the register file and output computed data up to two

registers, with a maximum of four register accesses per instruction due to instruction encoding constraints. Thus we looked for subgraphs having up to four input edges and up to two output edges.

Each extracted instruction includes some statements (taken from the original source code of the computation kernel). Those statements are used both as source code for the hardware implementation and as simulation model for the reconfigurable instruction set simulator.

By exploring various splitting of the CDFG, different possible implementations of the computation kernel on our reconfigurable processor can be tried. Then a trade-off analysis between cost and performance will help in selecting the optimal implementation.

In order to evaluate cost and performance, in the case of hardware implementation, we characterized each subgraph according to the estimated latency and number of required CLBs. These numbers can be derived manually if the developer has enough knownledge digital hardware design, or through an automated estimator available in our software development chain.

After partitioning we recompile the source code with our modified `gcc` (see Section 3.2). The compiler schedules instructions so as to best exploit both the standard data-paths and the FPGA, by using the latency assigned to each FPGA instruction.

Then we use an instruction set simulator to analyze system performance. The DLX Instruction Set Simulator that is distributed with `gcc` was adapted to count execution time of each native and user-defined instruction, according to the estimated latency, while executing the code for software implementation.

Once the optimal implementation is selected, a finer analysis of performance and functionality is possible through RTL simulation of the entire system (core and FPGA). We first synthesize hardware starting from an HDL translation of each CFDG subgraphs and then we place and route the FPGA. Information extracted from reports will help in refining user-defined instruction latencies and costs, that were previously only estimated.

5. Case study: turbo decoding

We applied the design flow described in the previous section to a decoder for turbo-codes [11]. The goal of the exploration was to speed up the execution of one of the most data intensive algorithms of future cellular telecommunications, by exploiting the programmable instructions of the reconfigurable processor. We used as a starting point for our optimization a publicly available version of the decoder from [1]. That code was originally implementing a Viterbi decoder, and we modified it so that it used iterations in order to implement a real turbo-decoder.

A very important aspect of the optimization was a reorganization of the memory layout, and a conversion of floating point values to fixed point (int and short int) for the target processor, which does not have a floating point unit. By appropriately pairing shorts into a single memory word, the memory access bandwidth, one of the bottleneck of the DLX-based architecture that we are using, can be doubled. We also re-arranged the code so that accesses to arrays in main memory was minimized. Since these are standard transformations, that have little to do with the reconfigurable nature of the processor, we refer the interested reader to [5] for more details. *All comparisons between reconfigurable and non-reconfigurable processor below are given based on the memory-optimized version of the source code, for both processors.*

Figure 1 shows the profile of the beginning of the backward recursion step. The integer before each line represents the number of clock cycles required to execute it. The code already includes the FPGA instruction pragmas, but the profiling has been done on the bare DLX architecture. The same code sequence is repeated 4 times with different inputs and outputs.

Figure 2 shows the profile of the same code exploiting the FPGA[1]. It also shows clock cycle counts for each FPGA-based instruction. We estimated that the execution of the instruction on the FPGA would require about 8 rows, for a total of about 256 CLBs, and a delay of 2 clock cycles (mostly due to the 3 16-bit additions; the fast carry chain of the FPGA can do 1 32-bit addition in a clock cycle).

The total execution time for decoding 1 bit on the basic DLX is 56261 clock cycles. The total execution time for the version using the FPGA is 16715, representing a speed-up of approximately 3.3X for the whole algorithm.

6. Conclusions

In this paper we showed how a reconfigurable processor can be used to dramatically speed up the execution of a highly data and control intensive task, the decoding of turbo-codes. We used a design flow that enables a software designer, who is mostly unaware of hardware design subtleties, to quickly assess the costs and gains due to executing selected pieces of C code as single instructions on an FPGA-based reconfigurable DLX functional unit.

The result is a factor of 3.3 speed-up on the whole decoding algorithm (not just on its inner kernel, where the advantages are much bigger). It was achieved, simultaneously with the memory layout optimizations (packing multiple data in a single memory access word, replacing matrices with vectors), with only about 2 weeks of work by software designers without any previous exposure to hardware design, synthesis, or reconfigurable computing.

In the future we are planning to further automate the design flow, especially in the direction of automated generation of the FPGA programming files from the extracted C code. This would dramatically improve the accuracy of the cost and performance estimation for a given FPGA instruction, with respect to the current technique, based on quick synthesis and judicious estimation of how many fast carry chains are being used.

We will also look at ways of automatically performing an optimized extraction, based on profiling results, cost and performance estimations, and limitations on the maximum number of register accesses.

[1] VLIW execution and DSP-like instructions were not used for this particular example. Therefore, the speed-up is due only to the FPGA.

```
  0  void Recursion_backward(unsigned int beta_metric[40][8/2],
           unsigned int branch_metric[40][16/2]) {
  0    unsigned int betaout, betain, beta[4], temp[4];
  0    unsigned int i, j, br0, br1, sw0, sw1, minval;
 12    beta[0] = 0x00008000; beta[1] = 0x80008000;
  4    beta[2] = 0x80008000; beta[3] = 0x80008000;
410    for (j = (40-1); j >= 0; j--) {
  0  #pragma fpga maxtot 0x21 2 1 3 temp[0] beta[0] \
           branch_metric[j][0] branch_metric[j][1] {
  0      short int a, b, br00, br01, br10, br11;
  0      short int betain0, betain1, betaout0, betaout1;
 80      br00 = (0x0000ffff & branch_metric[j][0] >> 16);
 80      br01 = (0x0000ffff & branch_metric[j][0]);
 80      br10 = (0x0000ffff & branch_metric[j][1] >> 16);
 80      br11 = (0x0000ffff & branch_metric[j][1]);
 80      betain0 = (0x0000ffff & beta[0] >> 16);
240      betain1 = (0x0000ffff & beta[0]);
240      a = br00 + betain0;
240      b = br01 + betain1;
429      betaout0 = (a > b) ? a : b;
160      a = br11 + betain0;
240      b = br10 + betain1;
430      betaout1 = (a > b) ? a : b;
635      minmxstar = (minmxstar < betaout0)?minmxstar:betaout0;
475      minmxstar = (minmxstar < betaout1)?minmxstar:betaout1;
240      temp[0] = (betaout0 << 16) — (0x0000ffff & betaout1);
  0    }
  0  #pragma end
  0  #pragma fpga maxtot 0x21 2 1 3 temp[1] beta[1] \
           branch_metric[j][2] branch_metric[j][3] {
  0      short int a ,b, br00, br01, br10, br11;
  0      short int betain0, betain1, betaout0, betaout1;
 80      br00 = (0x0000ffff & branch_metric[j][2] >> 16);
 80      br01 = (0x0000ffff & branch_metric[j][2]);
 80      br10 = (0x0000ffff & branch_metric[j][3] >> 16);
 80      br11 = (0x0000ffff & branch_metric[j][3]);
         ...
```

Figure 1. A fragment of the original source code of the backward reduction step

An interesting additional result of this research will be a better understanding of possible architectural enhancements for the target processor, e.g. in the direction of improving the memory access bandwidth of the FPGA-based functional unit. Of course we will not forget the key advantage of the architecture, its very user-friendly programmer's model.

References

[1] E. Boutillon and J. Sanchez-Turon. `http://lester.univ-ubs.fr:8080/~boutillon/sanchezt/main.htm`.

[2] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, Apr. 2000.

[3] T. Callahan and J. Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2000.

[4] F. Campi, R. Canegallo, and R. Guerrieri. IP-reusable 32-bit VLIW Risc core. In *Proceedings of the European Solid-State Circuits Conference*, Sept. 2001.


```
  0  void Recursion_backward(unsigned int beta_metric[40][8/2],
           unsigned int branch_metric[40][16/2]) {
  0    unsigned int betaout, betain, beta[4], temp[4];
  0    unsigned int i, j, br0, br1, sw0, sw1, minval;
 12    beta[0] = 0x00008000; beta[1] = 0x80008000;
  4    beta[2] = 0x80008000; beta[3] = 0x80008000;
410    for (j = (40-1); j >= 0; j--) {
480      asm("_maxtot %0,%1,%2,%3" :"=r" (temp[0]) : "r" (beta[0]),
             "r" (branch_metric[j][0]), "r" (branch_metric[j][1]));
480      asm("_maxtot %0,%1,%2,%3" :"=r" (temp[1]) : "r" (beta[1]),
             "r" (branch_metric[j][2]), "r" (branch_metric[j][3]));
480      asm("_maxtot %0,%1,%2,%3" :"=r" (temp[2]) : "r" (beta[2]),
             "r" (branch_metric[j][4]), "r" (branch_metric[j][5]));
480      asm("_maxtot %0,%1,%2,%3" :"=r" (temp[3]) : "r" (beta[3]),
             "r" (branch_metric[j][6]), "r" (branch_metric[j][7]));
  ...
  maxstar 1680
  butterfly1 1200
  maxtot 640
  butterfly0 400
  sum3swap 320
  sum3 320
  swap1 160
  reorder 160
  compute_gamma 160
```

Figure 2. A fragment of the source code of the backward reduction step using the new FPGA-based instruction called "maxtot"

[5] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecapelle, editors. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[6] S. Hauck, T. Fry, M. Hosler, and J. Kao. The Chimaera reconfigurable functional unit. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1997.

[7] J. Jacob and P. Chow. Memory Interfacing and Instruction Specification for Reconfigurable Processors. In *Proc. ACM Intl. Symp. on FPGAs*, 1999.

[8] B. Kastrup, A. Bink, and J. Hoogerbrugge. ConCISe: A compiler-driven CPLD-based instruction set accelerator. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1999.

[9] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA-machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the Design Automation Conference*, June 1999.

[10] R. Razdan and M. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, Nov. 1994.

[11] W. E. Ryan. A Turbo Code Tutorial. Technical report, New Mexico State University. http://telsat.nmsu.edu/ wryan/turbo2c.ps.

[12] L. Yanbing, T. Callahan, R. Harr, and U. Kurkure. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the Design Automation Conference*, June 2000.

[13] Z. Ye, N. Shenoy, and P. Banerjee. A C compiler for a processor with a reconfigurable functional unit. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2000.