European Union6th Framework Program

frescor

Framework for Real-time Embedded Systems based on COntRacts

FP6/2005/IST/5-034026

Deliverable: D-ND4

Reconfigurable FPGAs

Authors: Marek Peca, Pavel Píša, Michal Sojka, Jan Krákora, Zdeněk Hanzálek
Czech Technical University
Department of Control Engineering

Responsible: CTU

Version: 0.1/Draft

Purpose & Scope

This document describes FPGA as a computational resource. Case studies are presented, documenting such FPGA utilisation, and FRESCOR integration of corresponding resource management is proposed.

License and Distribution

Permission granted for receivers to re-distribute this document for use in the public domain except for the logos that belong to their respective owners.

Destinator	Function/role	For Action	For Information
FRESCOR partners		X	X
European Commission			X

Revision Histoy

file: **D-ND4-Reconfigurable-FPGAs.tex**

Release/Status	Date	Author/Reviewer	Comments
0.2/Draft	05/01/2009	Marek Peca	Integrated suggestions by Paolo Gai
0.1/Draft	16/11/2008	Marek Peca	

FRESCOR project (FP6/2005/IST/5-034026) is funded in part by the European Union's Sixth Framework Programme. This work reflects only the author's views; the EU is not liable for any use that may be made of the information contained herein.

Contents

1 Overview **2**

2 Principles of operation **2**
2.1 FPGA processing cores . . . 2
2.2 Interconnection between FPGA cores and CPU . . . 2
2.3 Reasons to employ FPGA cores . . . 3

3 Case studies **4**
3.1 Timestamp . . . 4
3.1.1 Description . . . 4
3.1.2 Motivation . . . 4
3.1.3 Implementation . . . 5
3.1.4 Advantages . . . 5
3.2 Correlator for ultrasonic localisation . . . 6
3.2.1 Description . . . 6
3.2.2 Motivation . . . 6
3.2.3 How to compute the correlation . . . 8
3.2.4 Implementation . . . 10
3.2.5 Results . . . 12

4 Integration into FRESCOR **12**
4.1 Resource reservation . . . 12
4.2 FPGA reconfiguration capabilities . . . 12
4.2.1 Dynamic reconfiguration . . . 12
4.2.2 Static reconfiguration . . . 14
4.3 FRESCOR contracts for FPGA resources . . . 15

5 Conclusion **16**

1 Overview

Field-programmable gate arrays (FPGAs) offer a possibility to constitute computing units (FPGA cores), tailored specifically for a particular task. FPGA cores are introduced, and the advantages of an implementation using FPGA cores are compared with a software implementation of the same tasks, are discussed in Sec. 2.

Next, two case studies, each using different advantages of the usage of FPGA cores, are presented. In Sec. 3.1, a utility core intended for analysis of software execution times by timestamping is described. In Sec. 3.2, an FPGA core is used to compute binary vs. integer cyclic cross-correlation, required by ultrasonic localization system.

Sec. 4 refers to FPGA reconfiguration, the possibility of carrying same taks by either hardware (FPGA core) or software (CPU), and offers a resource management scheme incorporated into FRESCOR.

2 Principles of operation

2.1 FPGA processing cores

As an alternative to traditional computing using software, specialised processing cores (units) can be created out of FPGA building blocks. Whereas the software runs on a processor (central processing unit, CPU), the FPGA cores are generic hardware structures. They work as synchronous finite automata. Regardless of timing and gate array capacity, they are theoretically equivalent to any other automaton (computer). It means that FPGA cores can perform the same operations as a CPU software, and vice versa.

The timing is usually an important factor either by means of speed, ie. computing performance, or by means of precision, ie. response time variation (jitter). The FPGA cores offer absolute timing precision up to a single clock period. On the other hand, the software is influenced by complex and hardly predictable CPU behaviour (eg. caching, bus arbitration). Under an operating system (OS), the timing of software operation is much more varied by context switching and execution of concurrent tasks. By means of timing precision, the FPGA cores are unbeatable by software.

Comparison of computing performance of the FPGA cores against the software strongly depends on a particular task. It is obvious, that machine code instructions of CPU themselves are performed by CPU much faster, than if implemented inside of the FPGA core. However, FPGA core can outperform software performance, if it is tailored specifically for the particular task: operations are wired without use of machine code (microcode), parallelism is used if applicable, and pipelines are employed to overcome propagation delays. Thus, FPGA implementation is especially suitable for simple computations performed on large amounts of data.

2.2 Interconnection between FPGA cores and CPU

The FPGA cores can constitute the entire application equipment without a need of CPU incorporated in the system. However, we will concentrate on applications, where the FPGA cores are interconnected with CPU and cooperate with software. Some tasks are processed by FPGA cores, the rest is performed in software.

There are various ways, how an FPGA core can be linked to CPU:

- **Fabric coprocessor module (FCM)** The FPGA core is connected to the CPU via a dedicated coprocessor interface. This technique is abbreviated as FCM in the case of IBM PowerPC 405 CPU cores integrated in Xilinx Virtex-4 family FPGAs. During invocation of user-defined coprocessor machine code instructions in software, data are exchanged between CPU registers and FPGA core.

  This way is especially suitable for operations, where a little amount of data is to be transferred between CPU and FPGA core. It is fast and it is not influenced by peripheral bus congestion. It is available for any CPU with coprocessor interface (in our case, it is an IBM PowerPC 405 core inside of a Xilinx Virtex-4 device [1]).

- **Memory-mapped peripheral (I/O)** The FPGA core is connected to some CPU peripheral bus. During memory access at certain address, reading/writing of data from/to the FPGA core is performed. Optionally, FPGA core can signal finished execution to the CPU by interrupt signal (IRQ).

  This way is more suitable for operations on larger amount of data, ie. such amount, that it can not fit into internal CPU registers. The use of FCM would be impractical in such a case, because the data had to be first transferred to a memory somewhere, and then processed by software. Otherwise, when the memory-mapped I/O is used, the software can operate directly on the memory-mapped data from the FPGA core. This solution requires only a bus or memory interface, which has been available for all CPUs ever.

- **Direct memory access (DMA) or core buss master access** This variant of data transfer between data storage and FPGA cores utilizes direct memory access (DMA) unit connected to main memory bus or a FPGA core driven bus master access. This solution has advantage, that there is no need to waste CPU cycles to transfer data to the peripheral unit or dual-ported memory blocks. The data are directly feed and returned back to their location into main system memory. This technique requires program driven synchronization of CPU core cache with main memory (invalidation and flushing).

- **Direct extension of CPU instruction set** This method is suitable only for open soft-core CPU modules and allows to adapt instruction set end extend it by application specific instructions which invoke data transfers and processing by units based on prepared FPGA cores. This method is very difficult not only for requirement of deep CPU architecture knowledge but the CPU core modification and extension can cause significant problem to meet timing criteria.

A single FPGA core can take advantage of one or more of these techniques to implement its task. Data flow use to be bidirectional, but unidirectional flow makes a sense as well. Eg. a random number generator core is output only.

2.3 Reasons to employ FPGA cores

The reasons, why to carry out some tasks by FPGA cores instead of software, are following:

- **Computing performance** Certain tasks can be computed by FPGA core in substantially shorter time, than by software on comparable embedded CPU.
- **Parallelism** The tasks are performed by FPGA cores in a perfectly parallel manner. The execution is not interrupted nor blocked by processing inside of CPU or neighbour FPGA cores.
- **Real-time** FPGA cores offer absolute timing precision and well predictable response time. Usually, the response time can be very short and constant.

Although whole system can be built of FPGA cores solely, without use of a software, it is usually advantageous or even necessary to use the software for common tasks. Beside FPGA capacity limitation, the software is easier to develop and integrate, especially for common tasks involving user interface, OS and software libraries. In our work, we consider the FPGA cores in connection with CPU and software only.

3 Case studies

Following FPGA cores demonstrate the principles presented. We have implemented the cores on Xilinx Virtex-4 FPGA hardware containing PowerPC 405 CPU. Both cores have worked together with an operating system-less software, as well as with a software under Linux OS (and virtually any other OS). For the hardware development, Xilinx ISE 9.2i and Xilinx EDK 9.2i software tools have been used. The PowerPC software has been compiled by GNU gcc 4.3.2, and run either operating system-less or under Linux 2.6.27-rc4. To enable use of FCM user-defined instructions, Linux kernel and GNU assembler have been patched.

3.1 Timestamp

3.1.1 Description

A timestamp FPGA core allows to record a time, at which a certain position in instruction sequence has been executed by CPU. Acting as a FCM, the core provides a user-defined machine code instruction. Whenever this instruction is executed, the time is instantaneously recorded in block memory. Then, several records can be read from the memory by software. The time is an integer value, incremented by a free-running counter, driven by the same clock as a CPU. The instruction can be called with an arbitrary operand (tag), recorded with the time together, to distinguish between different occurrences of the instruction in a code.

3.1.2 Motivation

The timestamp is a useful tool for measurement of software execution times. The times and its variation (jitter) can be measured between crossing of the same position in a code, or between several positions. It can be used for any kind of code – OS process, OS kernel, or operating system-less program.

3.1.3 Implementation

The timestamp FPGA core uses both FCM interface and memory mapped I/O. Recording act should be performed quickly, with low jitter and directly synchronised with instruction execution. The most desired is to bypass CPU and system instruction pipeline feed advance and write buffers latencies. Thus, the FCM interface has been employed for this task. On the other hand, reading of recorded times (and tags) from a memory is not a real-time urgent process, so the memory is connected to an I/O bus.

The block diagram of the timestamp core is on Fig. 1. Time is a 32-bit number, continuously generated by a free-running counter. When a FCM instruction is decoded, a FCM controller performs a write to the memory, and increments an address counter to point to next memory location. Lower 32 bits of memory word are filled by the time value, upper 32 bits are filled by an optional tag (instruction argument). The memory is composed of two 16 Kb block RAMs, available as a hard core[1] in an FPGA chip used. The block RAM is dual port. One port is used for the memory write, and the second port is connected to 64-bit wide bridge between block RAM and PowerPC processor local bus (PLB). The bridge is a standard library core, allowing easy connection of the memory to PLB I/O bus.

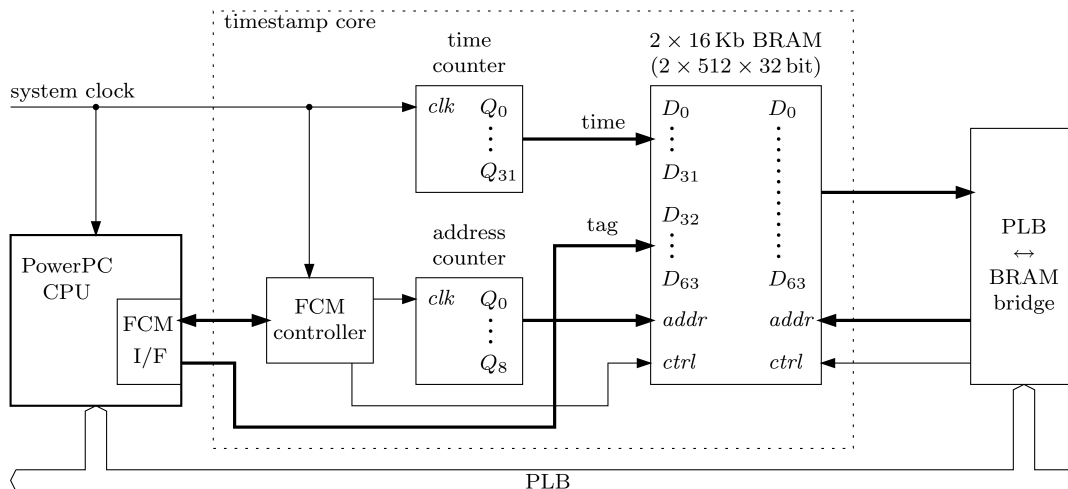

Figure 1: Block diagram of timestamp core

A very simple usage example of the timestamp is shown in code snippet on Fig. 2. An instruction (`udi0fcm`, macro `UDI_TSTAMP`) is called with two different tags (`1`, and then `0xabeceda`). Then, timestamp log is dumped off the memory in a loop. This example can be executed either operating system-less, as well as under OS. Under Linux, the kernel had to be patched to allow execution of user-defined FCM instructions in user-space.

3.1.4 Advantages

The timestamp implemented as an FPGA core has no jitter and low computational overhead. Each record involves execution of one FCM instruction. If user does not care about tagging of different timestamp instructions, the instruction can be executed with any of CPU registers containing any value

[1]Hardware structure, ie. a core, which is not composed of elementary macro-cells.

```
#define UDI_TSTAMP(id) __asm__ __volatile__("udi0fcm 0,%0,%0" : : "r"(id))

/* ... */

  for (;;) {
    c = getchar();
    UDI_TSTAMP(1);
    UDI_TSTAMP(0xabeceda);

    for (i = 0; i < 0x400; i += 2)
      if (bram[i] != 0)
        printf("%02x: (0x%08x) 0x%08x\n", i/2, bram[i], bram[i+1]);

    printf("--\n");
  }
```

Figure 2: Example usage of timestamp core

as an operand. Then, the execution takes 2 CPU cycles. If a tag has to be used, it must be assigned to the register prior to execution of the timestamp instruction.

3.2 Correlator for ultrasonic localisation

3.2.1 Description

A correlator core computes discrete cyclic cross-correlation function (a "correlation" in the following). The correlation $R_{xy}[n]$ between two sequences $x[k], y[k]$ where $k = 0 \dots N-1$, is defined as $R_{xy}[n] = \Sigma_{m=0}^{N-1} x[m] y[(m+n) \mod N]$. The correlation should be evaluated for all possible displacements $n = 0 \dots N-1$. For the specific application of ultrasonic localisation, one sequence (received signal) is quantised to 8-bit integer numbers, and the second is a pseudorandom binary sequence (PRBS), composed of binary values, ± 1.

3.2.2 Motivation

Calculation of the correlation is one of the most computationally intensive parts of the particular variant of ultrasonic localisation problem. An ultrasonic localisation system has been developed for two-dimensional (2D) wheeled mobile robot navigation inside a restricted rectangular playground. The motivation for this task is the EUROBOT competition. The playground is rectangular, $3 \times 2.1\,\mathrm{m}$ in size, and there can be placed three arbitrary beacons around it.

The ultrasonic localisation method we have used is based on time-of-flight of the sound measurement, see [2, 3]. The system transmits continuously the PRBSes, and uses the correlation to determine time difference, proportional to a distance. There are two variants of the system:

- mobile transmitter on the robot and static receivers in place of beacons;

- static transmitting beacons and mobile receiver on the robot.

The correlator presented is applicable for both approaches. In our particular system, the second option of three beacon transmitters and a receiver on the robot has been chosen. With such a setup, the robot inherently gets the required information, without a need of separate data link between the robot and beacons.

Signal path is shown on Fig. 3. Three different PRBSes, chosen to be highly uncorrelated (Gold codes), are transmitted synchronically by the three beacons around the field. The sequences are $2^7 - 1 = 127$ bits long and run at 3000 chips[2] per second. They are binary phase shift keying (BPSK) modulated into ultrasonic band, 39...69 kHz. The three signals are divided both by code and frequency multiplex[3].

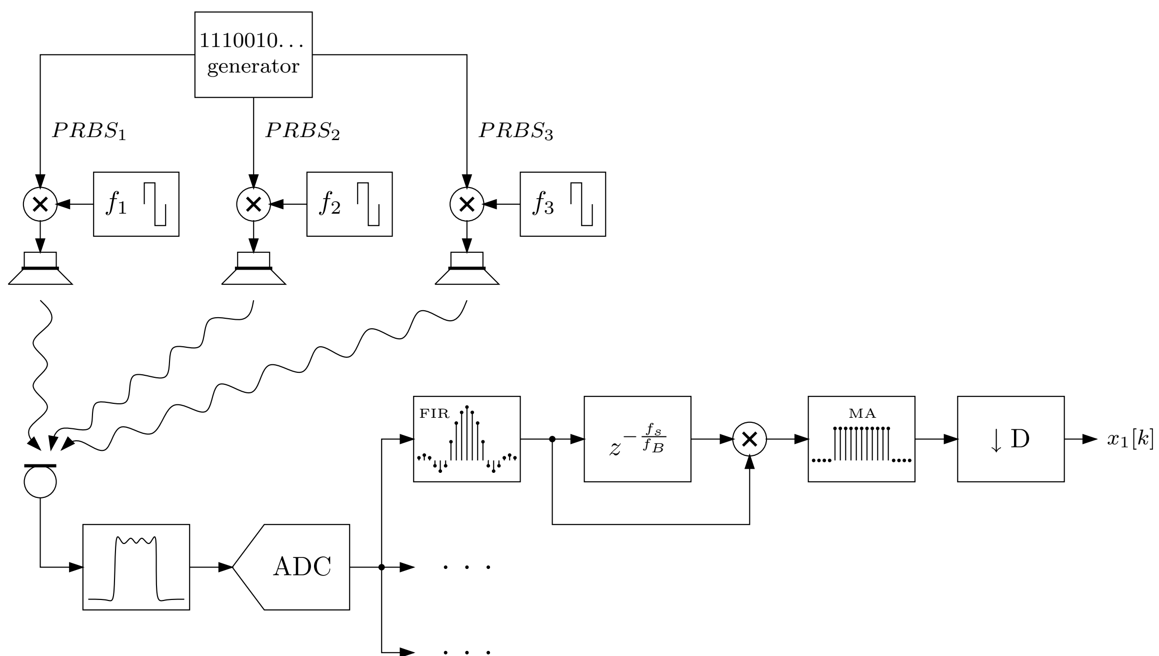

Figure 3: Ultrasonic signal path, filtration and demodulation

Resulting signals travel through the air, and delayed by the path length, they are mixed and received by an ultrasonic transducer at the robot. Signal passes through analogue anti-aliasing band-pass filter, sampled directly out of the baseband at 72 kHz sampling frequency, and analogue-to-digital converted. Then, the three signals are separated by band-pass FIR filters, digitally BPSK demodulated, filtered by moving average (MA), and then subsampled to 12 kHz, ie. 4 samples per chip.

At this point, the signals enter the correlation process. After collecting one PRBS period, ie. 4×127 samples, a peak in correlation between received and known transmitted sequence should reveal the time difference, see Fig. 4. After correlation, a low-pass interpolation can take place to get finer time resolution.

[2] Chip is one bit of the sequence.
[3] The reason is to give sufficient dynamic reserve between weak useful signal, and strong crosstalks.

The three peak times are sufficient to regularly resolve two position coordinates and a time difference between transmitter and receiver clocks. To get feasible results, direct analytic calculation should not be used (albeit it is possible), because of measurement uncertainty. Instead, the position should be estimated in a stochastic manner. In our case, we have used a Monte Carlo estimator, also known as a particle filter or condensation algorithm, to estimate robot position, mutual transmitter/receiver clock offset and drift, and to perform data fusion with wheel odometry.

Estimated trajectory of a 28.8 s long robot motion is shown on Fig. 5. Each trajectory point corresponds to one PRBS period. The "cloud" represents estimated robot's position probability at the end point, approximated by the Monte Carlo estimator particles. Photograph of our robot, carrying ultrasonic receiver atop of itself, is on Fig. 6.

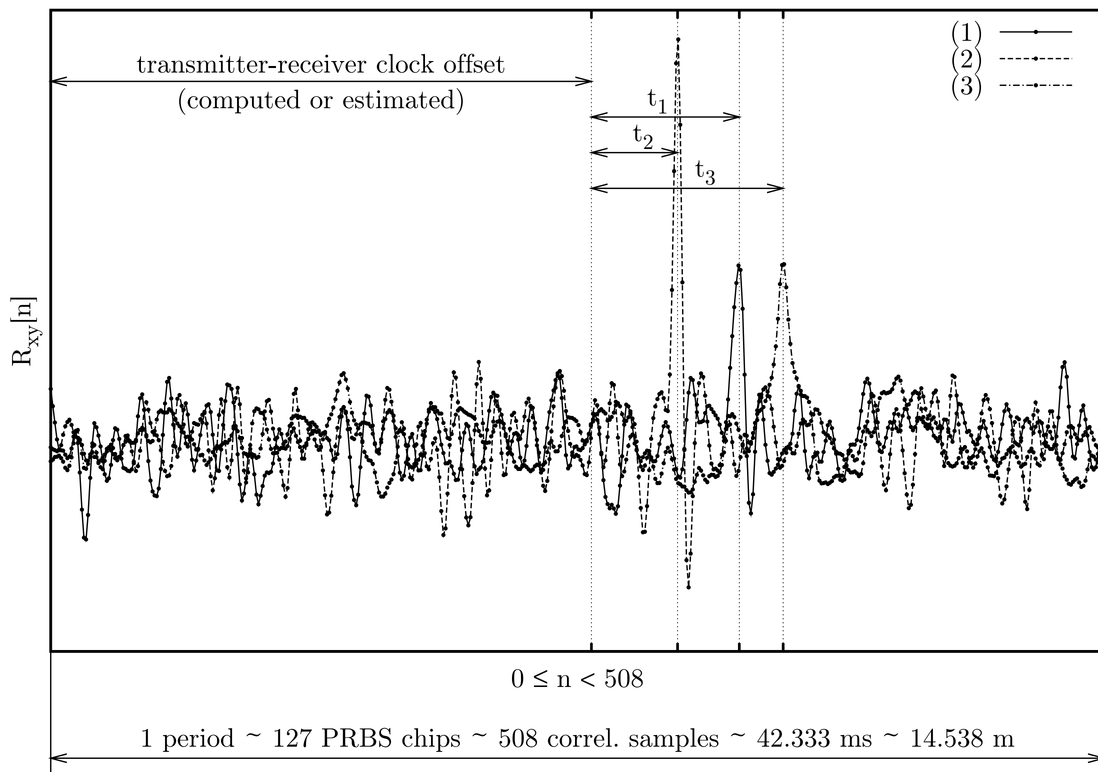

Figure 4: Correlation of all three received signals against corresponding known PRBSes; t_i is time-of-flight from i-th beacon, clock zero offset is marked

3.2.3 How to compute the correlation

Computation of correlation for all $n = 0 \ldots N-1$ following the definition involves N^2 multiplications of x and y and $\sim N^2$ additions. Since one of the numbers is ± 1, the multiplication reduces to conditional sign change of the second (integer) number. The correlator implemented in FPGA core performs N^2 conditional sign changes and additions. In our application, $N = 4 \times (2^7 - 1) = 508$, so the correlator performs 258064 such operations in one run.

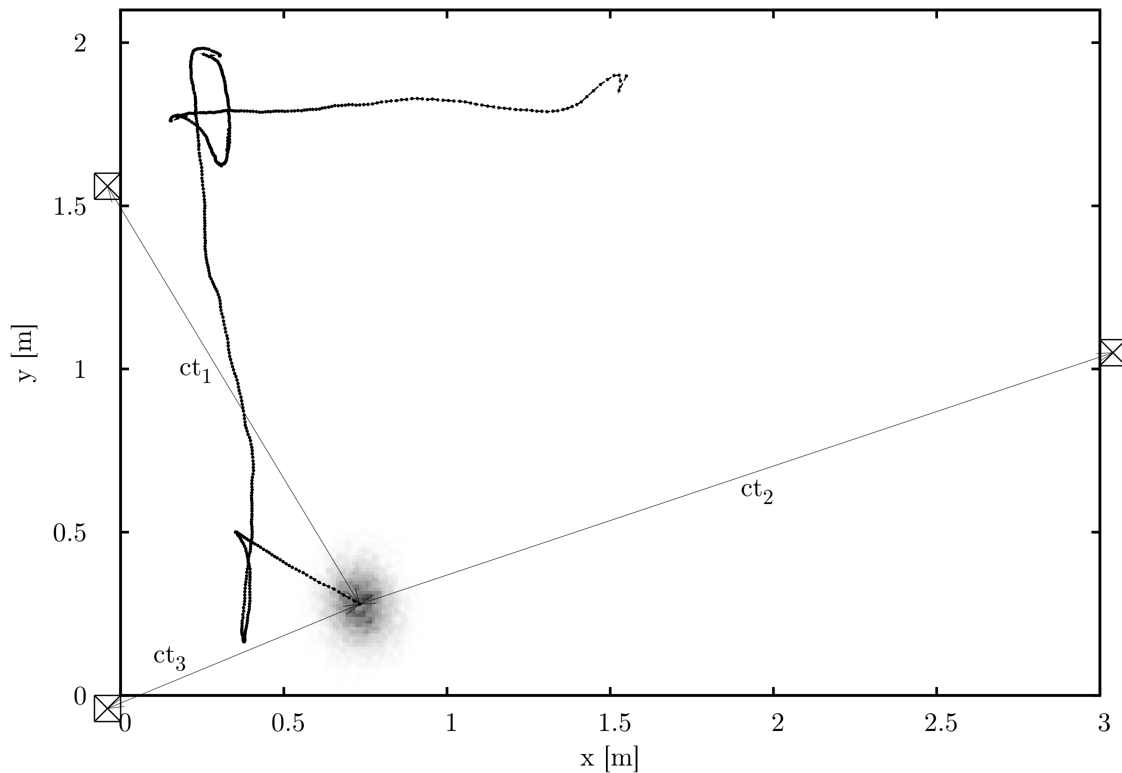

Figure 5: Estimated trajectory composed of correlation distance measurements; one point $\sim$ one PRBS period; c is sound velocity, t_i time-of-flight

Of course, there exist much more efficient methods to compute the correlation, but they are defined only for sequences with too restrictive properties.

If both $x, y \in \{\pm 1\}$, the multiplication reduces to boolean exclusive-or (XOR) function, and an implementation of correlation in ordinary CPU as well as in FPGA would be very fast and simple. The computation remains $\mathcal{O}(N^2)$, but whole vectors are element-wise multiplied by bitwise XOR. However, in our particular localisation problem, there can occur strong crosstalk from beacon in neighbour band, stronger than useful signal. Thus the 1-bit instead of 8-bit quantisation can lead to unusable results.

The correlation can be also very efficiently computed using fast Fourier transform (FFT) algorithm, working (roughly speaking) in $\mathcal{O}(N \log N)$, which already offers significant performance boost for sequences of hundred or more elements. However, the FFT is not applicable for prime-length sequences and almost useless for sequences of length, factorized into few primes[4]. Not surprisingly, the lengths of Gold PRBS codes of desired correlation properties are likely to be factorized in a very few primes. The lengths are in general $2^B - 1$. In our case, we use PRBS of $2^7 - 1$ chips, $4\times$ oversampled, so the sequence length is prime-factorized as $508 = 2 \times 2 \times 127$. This three-step FFT would not save anything. Of course, we could change the sequence length to eg. $4 \times 2^7 = 512$, but the correlation properties of the PRBS would degrade, so we decided to maintain the quality and employ an FPGA

[4] Algorithms for efficient computation of discrete Fourier transform for prime-length sequences have been developed, however, they are much slower than our simple computation, exploiting special element values ± 1.

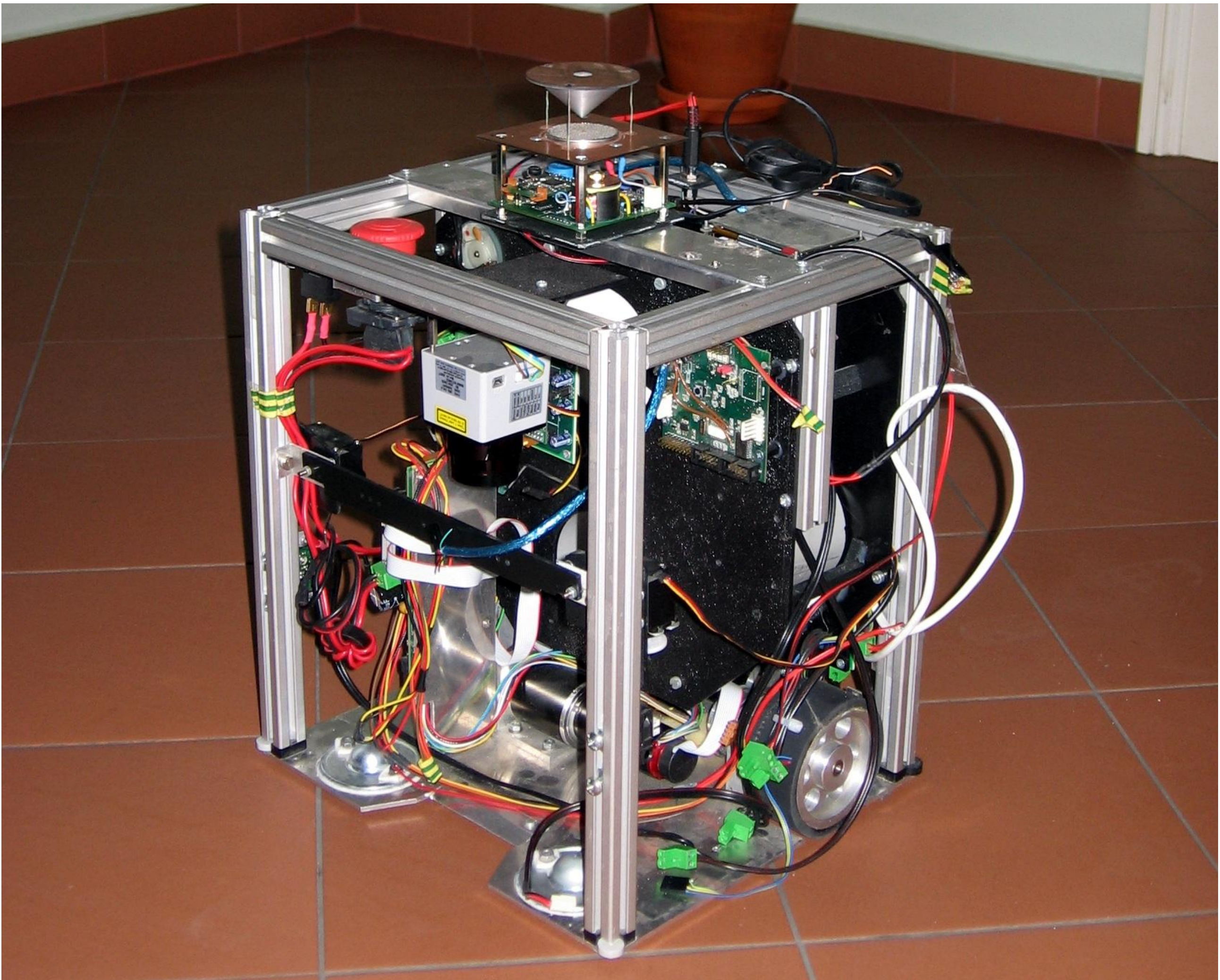

Figure 6: Robot equipped with ultrasonic receiver (electrostatic ultrasonic transducer adapted by cone for omnidirectional reception)

core for the computation, instead.

3.2.4 Implementation

The correlator uses only memory mapped I/O. It operates in a sequential manner upon two blocks of memory. One block contains input data, received 8-bit signed sequence $x[k]$. The second block is being filled with computed correlation, 32-bit signed $R_{xy}[n]$. Constant 508 bit code sequence $y[k]$ is loaded as a hardware initial state, ie. it is not accessible by CPU. After the input memory is loaded by $x[k]$, processing can be started by activation of `start` signal. This can be done eg. by general purpose I/O signal (GPIO). After the processing is finished and the output memory contains $R_{xy}[n]$, output signal `done` is raised. It can activate a hardware interrupt (IRQ), it can be checked by GPIO, or it can be safely ignored, if the minimal execution time of instruction sequence processed by CPU after FPGA

core activation ensures that elapsed time is greater than FPGA core processing time.

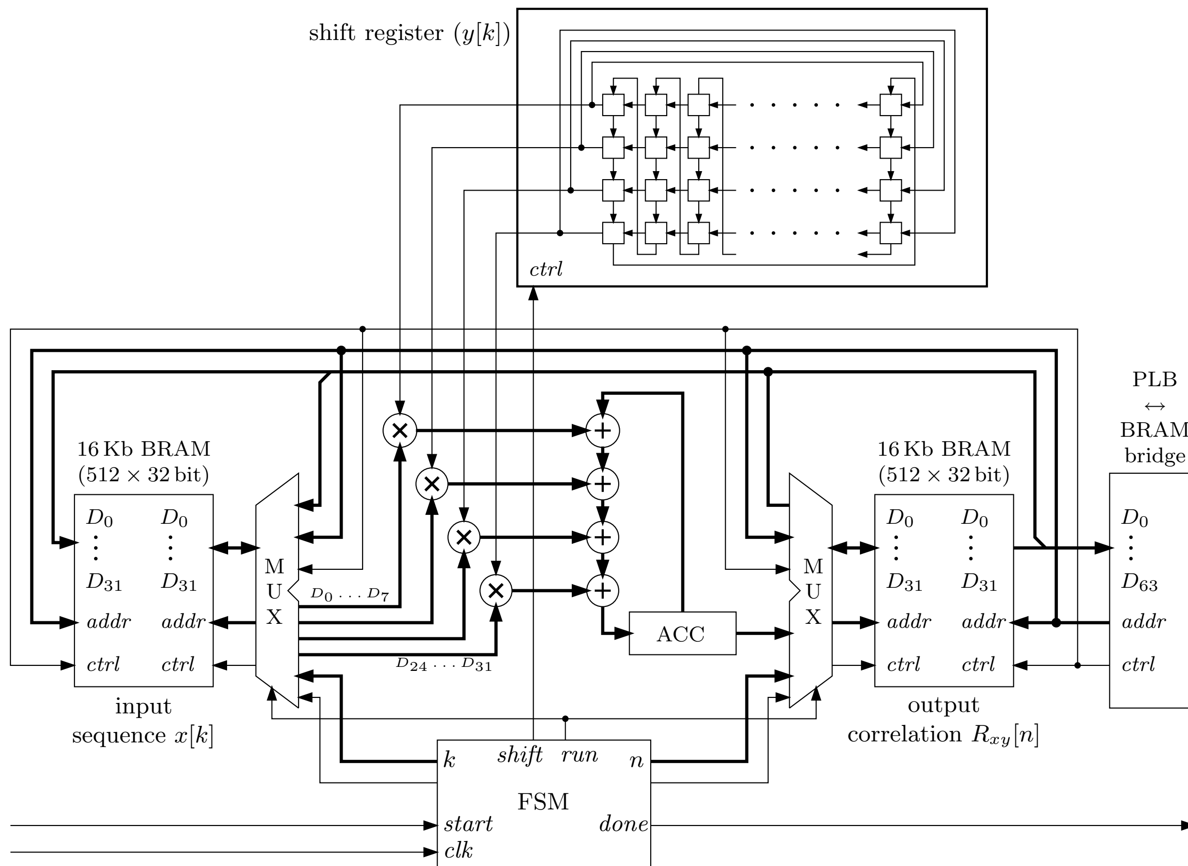

Figure 7: Block diagram of correlator core (summand pipeline register not shown)

Block diagram of the core is on Fig. 7. Input sequence $x[k]$ is being read from a block RAM. Block RAM interface can be up to 32 bits wide, so four 8-bit values are read at once. These values are "multiplied" by conditional sign change, conditioned by $y[k]$ code sequence bits, coming from shift register. Then, values are extended to 32 bits and summed into the accumulator. After cycling over all input numbers, the summed result $R_{xy}[n]$ is written to address n into the output memory. Then, n is incremented and the process repeated for all 508 displacements n. The sequential process contains two nested loops, inner over k (input memory address), and outer over n (output memory address).

Inside of both loops, the code sequence in shift register is shifted in a different manner. In the inner loop, the shift is effectively by 4 bits, because 4 numbers are multiplied at once. In the outer loop, the shift is only by 1 bit. At the end of the loop, the sequence must be shifted back to the initial state.

The operation is controlled by a state-machine (FSM), which cycles over both address loops, controls register shifting, and directs an access to the memories. During the computation, both block RAMs are connected by a multiplexer (MUX) to the computing kernel. However, when the processing is done, MUX is switched and both ports of dual port block RAMs are available for a bus memory access. Usage of both ports in parallel allows a 64-bit wide bus connection, maintaining contiguous storage of input and output data. The same 64-bit PLB bridge has been be used, as in Sec. 3.1.3.

The presented core interface uses memory mapped I/O on both input and output, acting as an external computing unit, connected only to CPU bus, and independent on any other hardware. However, in an application, the input data may come from an external hardware directly, possibly from another FPGA core. In such a case, the input block RAM and a `start` signal would be connected directly to the input source hardware. Then, the software would only wait for IRQ and read the computed correlation values $R_{xy}[n]$ from the output block RAM.

3.2.5 Results

The correlator has been implemented into Xilinx Virtex-4 XC4VFX12 FPGA. Running at 300 MHz clock, the correlation for $N = 508$ took 0.215 ms. On a 400 MHz MCU Freescale MPC5200B with PowerPC 603e core, the equivalent correlation implemented in software took approx. 20 ms. Running on a Virtex-4 bulit-in PowerPC core at 400 MHz, it would take the same or a slightly longer time. Provided that a PRBS period is 42.3 ms long, the computational load of PowerPC CPU required by the correlation in real-time is 47%.

Compared to recent embedded CPU, the processing is done by the correlator FPGA core 93× faster. Moreover, the correlation is performed in parallel, so it does not spend any CPU time.

Example input to the correlator, measured in real conditions, is on Fig. 8b. The correlator output is on Fig. 8c. The algorithm uses integer arithmetic and does not contain any round-off errors (as eg. integer FFT does).

4 Integration into FRESCOR

4.1 Resource reservation

From the FRESCOR framework point of view, FPGA is an additional computing resource. The FPGA is able to constitute one or more FPGA cores. Each core can substitute a part of software for a particular task, lowering overall CPU load. Within the frame of resource reservation, manager can decide, whether to execute the task in software, or whether to harvest the FPGA and use the FPGA core instead.

4.2 FPGA reconfiguration capabilities

One or more FPGA cores can occupy the FPGA at once. As application needs change in time, it may be desirable to reconfigure the FPGA, ie. to interchange currently used FPGA core set by a different one. There are two possible reconfiguration paradigms: dynamic and static.

4.2.1 Dynamic reconfiguration

With dynamic (often called partial) reconfiguration, content of the FPGA is changed only partially during the reconfiguration. Individual FPGA cores can be loaded into free FPGA areas, preserving other cores, already present in the FPGA.

The dynamic reconfiguration imposes the following difficulties:

- **Design constraints** At least certain common structures, typically buses, must occupy fixed locations inside the FPGA. All of the cores must be compiled with respect to preserve these common

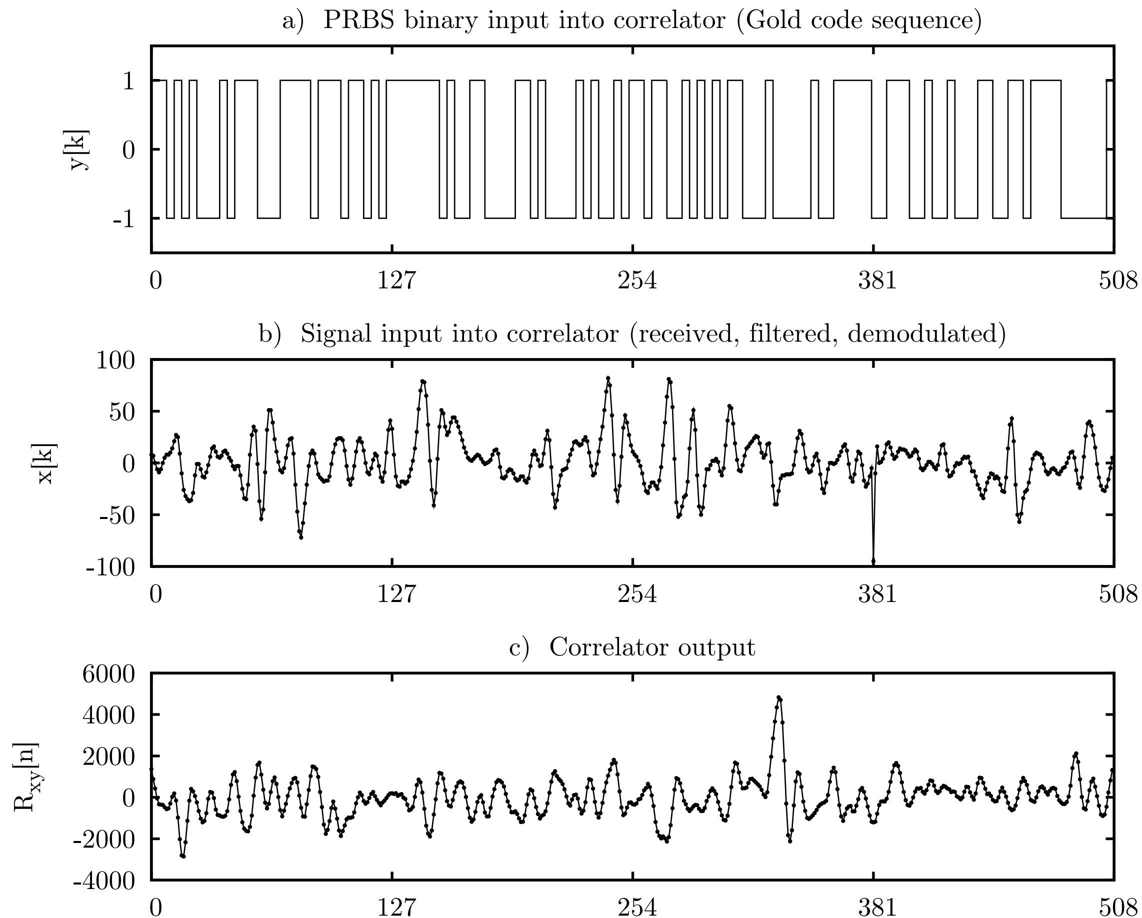

Figure 8: Example correlator input (a, b) and output (c), using signal (b) measured in real conditions; the output reveals correlation peak at position $n = 331$

structures, what constrains the design (compilation). Such a constraint may also lower the performance, because resulting signal paths and delays may be longer than needed.

Moreover, it is desirable to compile individual FPGA cores in such a way, to make them location independent. Location independent core can be loaded dynamically into one of several prepared areas of the FPGA, without collision with other cores, loaded into another areas. Architecture of such areas and their connection to the buses is another substantial design constraint.

- **Run-time partial loading** Commonly, the bitstream is loaded into FPGA at once. For the dynamic reconfiguration, there is a need to load only a part of the FPGA, leaving neighbour cores untouched. Moreover, execution of the other cores should continue during the loading (run-time reconfiguration).

- **Real-time loading** Loading of a bitstream to the FPGA can take a long time. The delay should be taken into account when designing a real-time system.

- **HW/SW state transition** State transition between FPGA and software implementation of the

same task can be optionally implemented. The state transition allows to transplant running task from software to FPGA core, and vice versa. Without possibility of the state transition, FPGA cores must be loaded before execution of their tasks begin, and they can be replaced (erased) only after the execution is finished. Implementation of the state transition is very complicated and task specific, see [4].

The main advantage of the dynamic reconfiguration is a flexibility in that FPGA cores can be loaded independently up to available capacity. The run-time reconfiguration can proceed during uninterrupted operation of running FPGA cores. The main disadvantage is a difficult design. Also, a resulting FPGA implementation of cores is slightly suboptimal due to design constraints, in comparison with the static reconfiguration.

The dynamic reconfiguration is a promising paradigm, however, it still is not a mature technology in industrial practice. Although FPGA manufacturers offer tools and application notes for implementation of dynamic reconfiguration [5], and a research has been done [6, 7], it is still a very difficult way. During our development, we have not employed the dynamic reconfiguration.

4.2.2 Static reconfiguration

For every desirable core set, an FPGA bitstream[5] is created (compiled) offline. Then, it is possible to replace whole core set by another one.

There are no reconfiguration specific design constraints imposed. Every desirable core set is compiled as a whole, as one large hardware, containing all the selected FPGA cores. If there are N FPGA cores present in the system, there are up to 2^N core sets. However, substiantially smaller number has to be actually compiled. Some of combinations can be impractical or useless in an application. Also, there is no need to compile a set, if its superset already fits into the FPGA[6]. Again, if a set can not fit into the FPGA at a whole, it will not be compiled, as well as all of its supersets. The limit case of this paradigm is that only each one of the FPGA cores itself is compiled, and only one of the cores at a time can be loaded into the FPGA.

Following difficulties are encountered even in simple case of static reconfiguration:

- **HW/HW state transition** In contrary to dynamic reconfiguration, whole content of the FPGA is replaced during each reonfiguration. Thus, if a task, running on an FPGA core, has to continue after the reconfiguration, it should be interrupted and its state must be transferred to new incarnation of the same FPGA core. However, the same core in a new bitstream may be placed in a different location, moreover, it may use slightly different logic building blocks[7]. Solution of such a general transition is a very difficult task. If the state transition is not implemented, the reconfiguration can proceed only when all FPGA cores are inactive.

- **HW/SW state transition** The issue of HW/SW transition is the very same as in case of the dynamic reconfiguration, see Sec. 4.2.1.

[5] Bitstream is a serialized representation (block of data), desribing an FPGA content on the lowest possible level. The bitstream is a breath of life, blown into the FPGA.

[6] Unless we concern about a power consumption.

[7] due optimization during compilation

- **Real-time loading** The FPGA is loaded whole at once, without preservation of any running cores or content. Thus, it is possible to use common software tools. If a real-time operation is required, a loading time should be taken into account. Also, an interface which assure deterministic timing should be used on a CPU side. On the FPGA side, eg. a JTAG[8] boundary-scan interface may be used.

The static reconfiguration is simple to use, a design is not specifically constrained. However, the HW/HW state transition is very difficult. The other big disadvantage is that every desirable combination of FPGA cores must be precompiled and stored somewhere in a memory. With growing number of the FPGA cores, there is a combinatoric explosion of possible core sets. Selecting only few of them results in a waste of possibly utilizable FPGA capacity.

4.3 FRESCOR contracts for FPGA resources

To support FPGAs in the contract framework a way for applications to specify their requirements in contracts has to be defined. In the context of FRESCOR, FPGA cannot work as stand-alone computing entity; it is used as a coprocessor which means it is always accompanied by CPU. Therefore, the contract for FPGA resource has always to be accompanied by a CPU contract forming a transaction.

Transaction was defined in [8] as *"A part of an application consisting of multiple threads executing code in multiple processing nodes, and exchanging messages with information and events. It is also called a global activity."* As one can see this definition is tailored to CPUs and distributed systems consisting of multiple processing nodes. The properties of the "transaction", as used in this deliverable, are almost the same as in the previous definition, but in order not to confuse the reader familiar with the previous definition, we define the transaction more generally as: *"A part of an application consisting of activities on multiple resources and synchronising these activities by some means."*

The tasks of the contract framework with respect to the FPGAs are the following:

1. Decide which cores should be loaded to the FPGA depending on application requirements.
2. For applications that can run their tasks either entirely in software or accelerated by an FPGA core, decide which application will run which variant.

For the framework to provide this functionality, applications must specify which cores they need in FPGA and CPU requirements for accelerated and software only (if available) variants.

To specify the needed cores a new contract block [9] named `FRES_BLOCK_FPGA` was defined. Its content is defined in `frsh_forb/resources/fpga/res_fpga_idl.idl`. There is only one field in the block which is a 64 bits wide bitmap with each bit specifying whether a particular core is needed by the application or not. For FPGAs, timing requirements are not specified in the contract as it is supposed that the a single core is not shared by multiple applications. In FRESCOR we assume that the FPGA as a whole can be shared by multiple applications (i.e. there are multiple cores loaded in the FPGA) but each application uses its own core.

An example transaction of application using the correlator core described in Sec. 3.2 is depicted in Fig. 9. The transaction involves two resources CPU and FPGA. Contracts for these resources are

[8]Joint Test Action Group, common name for IEEE 1149.1: "Standard Test Access Port and Boundary-Scan Architecture"

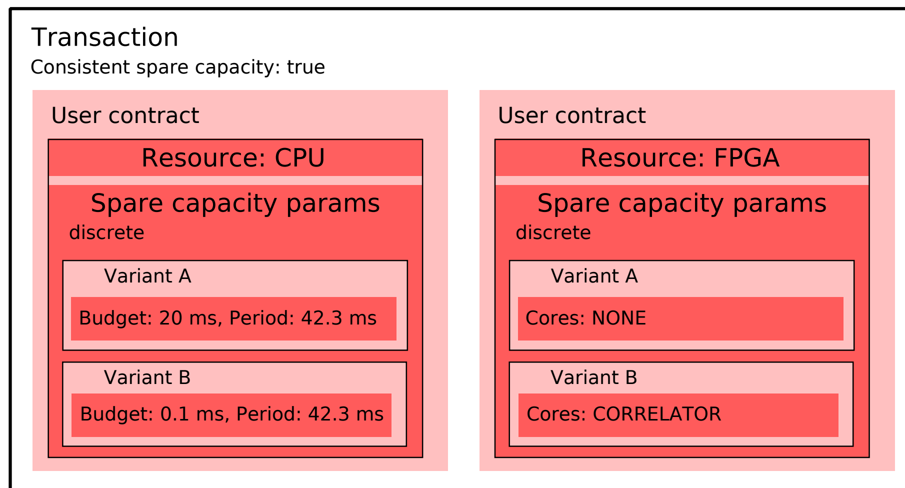

Figure 9: Example of data structures describing the transaction involving CPU and FPGA in the contract framework. There are two variants of resource allocation: A – software only and B – accelerated.

depicted in the figure. Both contracts use the spare capacity block to describe different reservations needed for software only variant and for FPGA accelerated variant. For proper execution of the transaction it must be ensured that the spare capacity is allocated to the contracts consistently within the whole transaction (i.e. it has no sense if CPU is allocated for variant B and FPGA for variant A). For this to be feasible, a new way of spare capacity distribution had to be designed in [9].

5 Conclusion

Benefits of FPGA cores as additional helpers cooperating with software has been presented and confirmed by case studies. Within the frame of FRESCOR, resource management for statically reconfigurable FPGAs has been proposed.

References

[1] PowerPC 405 processor block reference guide. User guide, Xilinx Inc., May 2008.

[2] A. R. Jiménez, F. Seco, R. Ceres, and L. Calderón. Absolute localization using active beacons: A survey and IAI-CSIC contributions. White paper, Instituto de Automática Industrial – CSIC, Madrid, Spain, 2004.

[3] Holger Linde. *On Aspects of Indoor Localization*. PhD thesis, Fakultät für Elektro- und Informationstechnik, Universität Dortmund, August 2006.

[4] Yvan Eustache and Jean-Philippe Diguet. Reconfiguration management in the context of RTOS-based HW/SW embedded systems. *EURASIP J. Embedded Syst.*, 2008(2):1–10, 2008.

[5] Two flows for partial reconfiguration: Module based or difference based. Application note, Xilinx Inc., September 2004.

[6] Lukáš Kohout. Partial dynamic reconfiguration in Xilinx FPGA circuits. In *CAK Embedded Systems Colloquium*, February 2007. http://rtime.felk.cvut.cz/kolokvium/2007/presentations/kohout_lukas.pdf (last visited: Nov 16 2008).

[7] A. Donato, F. Ferrandi, M. D. Santambrogio, and D. Sciuto. Operating system support for dynamically reconfigurable soc architectures. In *IEEE International SOC Conference, 2005. Proceedings.*, September 2005.

[8] Michael González Harbour, Daniel Sangorrín, and Michal Sojka. Distributed transaction manager - proof of concepts. FRESCOR Deliverable D-ND5v1, Universidad de Cantabria, 2007.

[9] Michal Sojka, Martin Molnár, Jiří Trdlička, Petr Jurčík, Petr Smolík, and Zdeněk Hanzálek. Wireless networks – documented protocols, demonstration. FRESCOR Deliverable D-ND3v2, Czech Technical University in Prague, 2008.