Reconfigurable Computing based on Universal Configurable Blocks – A New Approach for Supporting Performance- and Realtime-dominated Applications

Christian Siemers and Sybille Siemers
University of Applied Sciences Heide, Rungholtstr. 9, D-25746 Heide, Germany
christian.siemers@computer.org

Abstract
A novel architecture for reconfigurable computing based on a coarse grain FPGA-like architecture is introduced. The basic blocks contain all arithmetical and logical capacities as well as some registers and will be programmable by sequential instruction streams produced by software compiler. Reconfiguration is related to hyperblocks of instructions. For the composed reconfigurable processors a classification is introduced for describing realtime, multithreading and performance capabilities.

1. Motivation of the Approach

Since the introduction of the von-Neuman-model of processors, scientists all over the world were looking for methods of parallelizing the execution of sequential programs resulting in many architectures and algorithms. With upcoming field-programmable hardware (FPL), first available in 1977 and since 1985 capable of integrating complete systems on one chip, a new programming paradigm was introduced called structural programming. The difference between both programming paradigms results directly from basic architecture models: Control-flow based (von-Neuman) versus data-flow based (structural hardware).

Until now, no common execution model for both paradigms has been presented. Even the connection between microprocessor and field-programmable hardware seems to be much more improvable [1]. The most important advantage of one unified (hardware) model for sequential as well as structural programming could be the runtime variability of executing program units. The software designer or even the operating system could decide how to ‘execute’ any part in a more software (sequential) or hardware (structural) manner.

In the authors’ opinion the missing common model and connectivity are the most important reasons for all difficulties of introducing Hardware/Software Co-Design methods on a broad base. Despite that applications with great advances in speed-up through co-design are innumerable, there exists any general model – with exceptions. A good overview for co-design methodologies and related topics is presented in [2] [3].

The most important exception are the F-CCMs, which means the class of Customized Computing Machines based on Field Programmable Gate Arrays. For subclasses of the F-CCMs, e.g. the PDD-CCMs (Procedurally Data Driven, [2]), models are presented with the result of successful introduction of automatic methods.

The success of these PDD-CCMs, e.g. the Xputer [4], makes it worth to research for a more global model for ‘executing programs’ (in whatever context this is interpreted). This is the goal of the paper: to present a (partial) unified execution model. The model is based on the architecture of the Universal Configurable Blocks (UCB) in connection with the Procedural Driven Structural Programming algorithm (PDSP), which translates a flow of (non-control-)instructions into the UCB structure.

The remainder of this paper will introduce and briefly discuss the UCB architecture (section 2) and the PDSP algorithm (section 3). Universal Configurable Machines (UCM) will be introduced in section 4. This is a new class of programmable machines using UCB for block based computations, and as these processor-like machines represent a non-classical approach, a classification system for UCMs is introduced. Section 5 scratches the software side of the reconfigurable architecture by a short view on the design methodologies and the impacts on the operating system. The use of blocks for the execution of programs and the availability of a multi-block machine enables the designer to manually decide between program-sequential and exclusive execution. This kind of hardware/software co-design will be also discussed. The paper finishes with a conclusion, an outlook to future work and some references.

2. Universal Configurable Blocks (UCB)

A Universal Configurable Block (UCB) (see Figure 1) consists of a register file, a configurable asynchronous network with some arithmetical and logical capacity and optionally some auxiliary circuits. Optionally a load/store pipeline as well as private memory (random access or stack organised) may be added to the UCB. The width of data busses and registers inside is related to the actual
implementation and may vary between different instantiations.

The sub-units of a UCB are designed to create a configurable block for computations using a partial or complete network, and the following types are included inside:

- **Arithmetical Unit** (AU, type A): The computations inside the UCB are mostly performed by the AUs. Differing from the well-known Arithmetical-Logical-Unit, the AU includes just one or few (configurable) computational combinations, for example, adding two integer values. Connecting the inputs to the sources, the result is available at the output of this subunit for use inside the whole network.

- **Compare Unit** (CoU, type B): This sub-unit type generates conditional bits inside the network used for the conditional execution of computations or moves. Characteristics for the CoU are two input busses, one conditional bit as an output and the possibility for configurable comparisons.

- **Multiplexer** (Mul_C, type C): Type C Multiplexer are used to route sources to the inputs of AUs and CoUs with two input busses (of full bus width). These multiplexers connect the different sources with the inputs of the following.

- **Multiplexer** (Mul_D, type D): The routing of a number of possible sources to a certain register can be done by a multiplexer with only one output bus. Type D multiplexer connect the destination registers to the outputs of the AUs.

- **Demultiplexer** (Demul, type E): The results from the compare units (CoUs) have to be routed to the corresponding AUs, therefore a pair of CoU and AU establishes a conditional execution of an instruction.

- **Compare Unit 2** (CoU2, type F): This optional unit generates the Ready-signal for other control parts of more integrated devices of the reconfigurable architecture. The purpose of this unit is to integrate loops on hardware.

- The additional hardware is integrated to synchronize clocks with external and internal control signals.

The optional private RAM shown in Figure 1 was originally not included in the preceding work [6] [7] [8]. The analysis of potential algorithms e.g. from network applications or signal processing has shown that they often demand for data tables exceeding the size of the register file. The integration of a private memory enables the explicit coding of such algorithms inside one UCB without extensive data accesses.

The focus of UCBs is set on fitting execution to a complete hyperblock of instructions [5]. These are assembled by optimising compiler supporting execution inside superscalar processors. As shown in [6] [7] [8] for the Sputer and in [9] for an alternative co-design approach, the hyperblock is the ideal part inside any program to be translated into a structure. For the scope of this paper a hyperblock of instructions will be called a hyperinstruction.

From configuration viewpoint, a UCB is configured by the multiplexer system to the active datapath of a hyperinstruction (with some small extensions, e.g. when a specific arithmetic operation is selected inside a configurable AU). This results in a comparatively low number of configuration bits but a coarse grain FPL architecture. Table 1 summarises the main features of all FPL classes.
Table 1. Comparison UCB versus common FPL classes

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>CPLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design for High Speed Applications</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Design for Usability/Flexibility</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Design for High Speed Reconfiguration</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Design for Use with µP-compatible Programs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. The PDSP Algorithm

The UCB itself was introduced as a programmable structure being capable of performing state machine or even asynchronous logic. For practical use it has to be discussed how to obtain the actual configuration.

For the integration into a microprocessor, the automatic link between the UCB and the sequential instruction flow is mandatory whenever software languages like C, C++ or Java shall be used. The idea behind this is based on the approach to provide a configurable structure to the system (software) designer, who may use this like structurable hardware or for processing sequential instruction flows.

This kind of reconfigurable computing requests for an automatic transformation of the instruction flow into the configuration flow, which is performed within the PDSP (Procedural Driven Structural Programming) algorithm. It generates an equivalence between the hyperblock of instructions and the structurable hardware inside the UCB.

The basic idea for the algorithm to convert the flow of instructions into the structure information is to map each instruction into the UCB structure. For that purpose, all instructions inside a hyperblock of instructions are in-order mapped to a corresponding Arithmetic or Compare Unit. Under the assumption of a complete network inside a UCB, the algorithm may use several look-up tables (LUT) for the generation of the configuration inside.

The basic look-up table is organised as follows. For every sub-units inside one UCB there exists a unique logical description. As it is possible that a sub-unit contains more than one operation (and the actual used is selectable by configuration bits), several logical units may fit into one physical by these configuration bits. With these assumptions PDSP consists of the following steps:

1. Transformation of all translatable instructions [6] [7] [8] into an internal format, finishing if a control flow instruction is detected. This internal format may possibly be omitted but enables much easier processing during steps 2 and 3.
2. Looking for a free logical unit inside the basic look-up table. If found, all logical units belonging to the same physical unit will be marked as occupied. This steps also integrates register renaming if requested. Constant values are performed within a special algorithm, as these values were stored inside the program code. The constant is stored in an additional register after looking through a table containing all formerly stored values. Instead of the constant value the number of the extra register is inserted in the configuration bitfield.

All used registers are marked with READ, WRITE or READ_WRITE depending on their actual state of use. This results in a second look-up table for the next step. The algorithm finishes at this point if no logical unit is available for the requested operation, other resource limitations like the number of constant registers occur or if the destination register is marked with WRITE or READ_WRITE and register renaming is not performed.

3. The third step performs data forwarding for all source registers marked with READ_WRITE or WRITE in former steps. Using the information provided by the second look-up table, the source for the register is inserted as source for the operation in progress.

This algorithm translates the hyperblock-related sequential instruction flow into a structure based on the reconfigurable UCB architecture. If no restriction concerning the internal network of the UCB exist, the algorithm may be implemented in hardware really using look-up table (in practice, look-up tables provide only poor performance and will be substituted by direct logic).

Estimations and simulations show that assuming a k parallel fetch for n instructions with a total sum of d data dependancies detected for the instructions inside one fetch block, the number of cycles cyc for a pipelined version of the PDSP algorithm is given by

\[ cyc = \frac{n}{k} + d + 1. \]

4. Classification and Implementations of UCMs

As shown, the Universal Configurable Blocks may be compared with a coarse grain FPL architecture including arithmetical capacities, and the PDSP algorithm makes this structure available to sequential streams of instructions. Despite this positive effect determined hardware has to be added to receive a complete processor. Due to the fact that a UCB/PDSP will fit only to a hyperblock of instructions these additional units have to load the block or to handle the sequences of blocks.
UCB implementations in connection with loading/sequencing hardware will be called Universal Configurable Machines (UCM). To receive a full flexibility of the complete concept, additional, partial configurable hardware beside the loader and sequencer will be included. This may lead to some confusion, but while any UCB may fit to a hyperinstruction on a block-by-block base, the overall configuration of the UCM will be at least quasi-statically.

The additional hardware will contain configurable connections between several UCBs (for communication and/or connecting UCBs to perform larger hyperinstructions), load/store pipelines for memory data access and configurable input/output system for external connections similar to well-known controller technology. The UCM will differ from known microcontroller implementations by the internal microarchitecture.

Before several implementations of the UCM are listed, a classification scheme is presented. This classification consists of a triple notation of the unsigned integers \( b \) and the boolean variables \( h \) and \( s \). The variable \( s \) describes the ability of the UCM to sequence hyperinstructions during runtime \((s = 1)\) or not \((s = 0)\). A non-sequencing UCM seems to be no computing machine but makes sense for small applications as shown in one implementation.

The boolean \( h \) determines the ability of the UCM to execute more than one hyperinstruction in parallel \((h = 1)\) or not \((h = 0)\). For a multi-hyperinstruction UCM the presence of an at least rudimentary scheduler and more than one UCB inside is essential. Due to the fact that small threads which will be called microthreads will be able to fit into one UCB, the terms microthreads and hyperinstruction describe the same object but from different point of views.

Last not least the unsigned integer variable \( b \) counts the number of UCBs inside the UCM. This number is deterministic, and there will be \( 1 \)- and \( b \)-UCM \((b > 1)\) implementations for different application classes. In summary, every UCM will be classified by a bhs-triple \((b, h, s)\).

### 4.1 Small-sized fieldbus-computers implemented as (1,0,0)-UCM

The smallest possible implementation of a UCM is rather similar to the UCB and consists of one block executing just one hyperinstruction (or in terms of the application viewpoint: one microthread). This could be a well-designed architecture for small applications as members of a fieldbus system or even independent of others. The difference between a simple UCB/PDSP and the (1,0,0)-UCM is located inside the additional control unit, which contains no sequencer but a hyperblock loader to switch between several operational conditions or to run a sequence of configurations. Figure 2 shows the implementation of the (1,0,0)-UCM.

The control unit serves several input lines from the outside world, e.g. any process, or from inside computation. The task of this unit is to identify the operational condition from the process and to switch to the next hyperinstruction. The sequence of hyperinstructions is therefore equivalent to a sequential finite state machine (SFSM). Please note that any fieldbus interface is not included in Figure 2 but could be easily integrated through input/output or process state lines.

### 4.2 The >S<puter as a (1,0,1)-UCM

The >S<puter model was introduced in [6] [7] [8] as an execution model for superscalar and VLIW processors using reconfigurable hardware inside. The main issue of this architecture is to introduce another dispatching algorithm with advantages in data forwarding and in the usability of the internal resources.

Figure 3 shows the >S<puter architecture using one UCB and the requested additional unit for sequencing the hyperblocks of instructions. This unit is responsible for the flow of hyperinstructions \((s = 1)\). Due to [6] [7] [8] the main tasks inside are to detect the class of the incoming instruction and to transfer it to the corresponding unit.
inside the processor architecture.

Non-control instructions are translated through the PDSP unit into the hyperinstruction structure for the UCB, while including the load/store instructions is still optional. Control flow instructions are handled within the sequencer unit, and all efforts for this task will determine the complexity of the sequencer.

During earlier work speed-up of the >S<puter compared with other superscalar architectures was determined within the interval >1 .. 2, expressed in cycle times.

4.3 PDD-CCMs as \((b,0,0)\)-UCM

As mentioned in the beginning of this section the UCM will be configurable not only within the UCBs (for the hyperinstructions) but also inside the additional hardware for more flexibility. A \((b,0,0)\)-UCM is at the first glance usable as a multi-microthreaded machine like the \((1,0,0)\)-UCM, and the connection of two or more UCB to a larger unit is another example for this, but there are extended possibilities of the UCM to adopt alternative architectures.

To prove this, a well-understood alternative architecture known as Procedurally Data Driven Customized Computing Machine (PDD-CCM) is described by a \((b,0,0)\)-UCM. One of the interesting implementations is the Xputer [4], and application-specific speed-ups of 2000 are reported compared to state-of-the-art processors.

To prove this, a well-understood alternative architecture known as Procedurally Data Driven Customized Computing Machine (PDD-CCM) is described by a \((b,0,0)\)-UCM. One of the interesting implementations is the Xputer [4], and application-specific speed-ups of 2000 are reported compared to state-of-the-art processors.

To prove this, a well-understood alternative architecture known as Procedurally Data Driven Customized Computing Machine (PDD-CCM) is described by a \((b,0,0)\)-UCM. One of the interesting implementations is the Xputer [4], and application-specific speed-ups of 2000 are reported compared to state-of-the-art processors.

4.4 A generalized \((b,h,s)\)-UCM

In general, a UCM may contain more than one block serving more than one thread. This makes an extended control unit necessary, which now integrates not only the sequencing but also the scheduling subtask inside the UCM. Figure 5 shows this fact: The sequencing/scheduling unit sequences for every UCB inside the flow of hyperinstructions on thread base, and the \((b,1,1)\)-UCM will in general act as a multithreaded computing machine.

Multithreading on the other side does not necessarily impact an extended interface to program memory. In comparison with the \((1,0,0)\)-UCM, a thread could consist only of one hyperinstruction, which is uninterrupted executed inside one dedicated UCB. Threads of this kind were introduced as microthreads and will produce high reactivity inside the UCM-application.

The microthread concept will work in connection with embedded system applications. Small control threads like boundary checks of digital values or data transmissions to/from memory will fit into one UCB, and for more flexibility UCBs will be connectable to integrate a greater hyperinstruction (or microthread).

Independent (non-micro-)threads on the other side will have no communication between UCBs and will execute independent from each other. The memory interfaces have to be designed for the extended access, this is subject to further simulation.
5. Impacts on Development System, Compiler and Operating System

It has been one of the most important objectives of this approach to introduce a reconfigurable architecture with minimal impact on the development methodology, but there will be some – and there should be also some. For explaining details Figure 6 presents the different views to an application using a Y-diagram.

The Y-diagram shows some layer with links between the several views. Complete tasks for example will be coded within one algorithm using classes etc. for homogeneous programmable systems. Inside the UCM concept a complete algorithm could be translated to an \( >\)S<puter ((1,0,1)-UCM) device or to a Xputer ((b,0,0)-UCM), but the algorithm and the coding in high level language would remain (nearly) the same. These more self-contained applications may use \((b,1,1)\)-UCMs as microprocessor and/or as hardware accelerator, and it is subject of the compiler technology to translate the sourcecode to well-performing executable code using the configurability of the UCM.

For application in the context of embedded systems, the design flow may take another path. Identifying and defining microthreads or threads in the problem domain will make it possible to describe the runtime behaviour of the system with more exactness. This is important for embedded and responsive systems with hard real-time capabilities, and the reconfigurable UCM architecture enables the developer to determine the system behaviour corresponding to the system description. This is performed by dedicated UCBs inside a \((b,1,1)\)-UCM and microthreads.

---

**Figure 5. Architecture of the generalized \((b,1,1)\)-UCM**

**Figure 6. Y-diagram for reconfigurable computing using UCB/PDSP**
Concluding the impacts on software, the following statements describe the situation with this approach for reconfigurable computing:

- The compiler technology for superscalar processors including the creation of hyperblocks is sufficient for the UCBs using the PDSP algorithm. The PDSP itself may be integrated in hardware or software (post-compiler run). Minor adaptations for generating UCB-optimized code may be integrated in the compiler. Beside this specialized compilers e.g. for PDD-CCMs as well as fault tolerant applications could be developed generating the corresponding UCM configuration.

- The operating system has to support threading and microthreading, compiletime as well as runtime priorities and scheduling strategies to enable the complete system for deterministic behaviour under hard constraints. The interface between operating system, probably description language and UCM seems to be the most relevant part on the software side.

- The developer may define threads or microthreads, and he should do this especially for embedded systems to obtain hard real-time capabilities.

6. Classes of Applications

One of the remaining questions will be the discussion of those application classes, where this methodology for hardware/software co-design should be preferred. As the designer receives the task to explicitly define threads for using the advantages of the architecture, there should be some benefits, while automatic tools for co-design are promising at least more comfort during design phase.

The first class will be consist of small applications using the aspect of reconfiguration in the sense of re-use the hardware. A good example for this class are all kinds of build-in self tests (BIST) for controller and memory.

A memory build-in self test (MBIST) consists of some very small test algorithms trying to identify the usability and correctness of RAM cells. It could be shown during a small project that realising the MBIST in a (1,0,0)-UCM using several reconfigurations would offer 20% space and 33% time reduction, compared with a dedicated hardware solution. This UCM was designed as a sequential finite state machine.

The unexpected time reduction results from more flexibility for the algorithm parts using reconfigurable hardware, not from increasing clock (the operating frequency of the studied MBIST is about 5 MHz). The additional benefit is of course re-usability of the UCB/UCM.

The second class of applications is inside the area of embedded systems with hard real-time constraints and a strongly event-driven environment. In this case the developer has to be sure to fulfil all demands, because the product has to work under all circumstances. This will lead to explicit multithreading with the use of (micro-) threads for interrupt services and dedicated UCBs for high priority interrupts.

In this case the use of one description language (e.g. C) for describing both sequential execution and dedicated hardware and the well-defined threading behaviour of the solution to guarantee real-time demands will count as the main benefits.

High performance applications inside digital signal processing using specialised approaches like the Xputer form the third class for using UCMs. For this case it was shown in [4] that no explicit thread programming is needed to obtain useful results. As the Xputer was identified to be a special configuration for the (b,1,1)-UCM, the Xputer results (they consist of configurations) may be transferred to the UCM.

7. Conclusion and Outlook

It was the goal of this paper to introduce a new approach for reconfigurable computing. The introduced components are concluded in Figure 7 and show basic blocks and compound devices. Special configurations will be available to adapt the (b,1,1)-UCM to different applications.

---

**Figure 7. UCB/UCM relations**
This approach promises to adapt very well to certain application constraints, especially in the area of real-time. Future work on the hardware side will continue with simulation of several UCB architectures to achieve the 'ideal' UCB and configurable UCM architectures to ensure the adaptivity.

The UCB/UCM architecture contains a more-dimensional scalability inside. The UCB block size is scalable, an optional connectivity between blocks makes them more flexible, and last not least the additional hardware units to form a UCM range from very small to large including operating system capabilities. This makes the UCM approach to be interesting for semiconductor industry.

On the software side the focus is set to the interface between operating system and UCM as well as operating system and the description by the developer. The last interface gains its importance by the approach to introduce an architecture, which allows the maximum of determinism even on the level of the operating system.

References


