

ConPro: High-Level Hardware Synthesis With An Imperative Multi-Process Approach

Dr. Stefan Bosse

BSSLAB - Independent Research Laboratory, Bremen, Germany

ABSTRACT

This article gives a design overview of the new hardware synthesis tool ConPro used for development of application-specific digital logic systems providing a high-level approach with imperative programming features filling the gap between hardware and software level. It is an experimental platform for studying different hybrid scheduling strategies, too. The ConPro development tool and programming language guides the user from an algorithmic programmer software view to RTL hardware architecture. Concurrency is explicitly modelled with communicating processes and interprocess communication primitives known from traditional and well-known parallel software development using light weight processes (threads), like semaphores or mutexes. The process model provides only a single control path resulting in strict sequential instruction scheduling. Concurrency inside processes is limited to the data path. Beneath explicitly modelled concurrency, automatic exploitation of inherent concurrency is provided by the synthesis compiler using a multi-pass hybrid scheduler.

Keywords

Hardware Synthesis, High-Level-Synthesis, Digital Logic, Multiprocessing, Register-Transfer-Logik, Parallel Programming, Hardware Architecture

1. INTRODUCTION

1.1 Overview and Motivation

Mainly two different entry levels are used for developing application-specific digital logic:

Register-Transfer-Logic on Hardware Level

A hardware description language is used to model the behaviour of the hardware system, mainly VHDL and Verilog with medium abstraction level from hardware. At the top-level a Register-Transfer-Logic (RTL) specification is used to model hardware as a directed

graph. Nodes represent circuit blocks and edges correspond to interconnecting wires (signals) [6].

Although all algorithms described by an imperative programming approach can be modelled this way, a linear growing of system and algorithm complexity results generally in exponential growth of development time and resources for describing and modelling on RTL-level, because the data path consisting of functional units, registers, and multiplexers, and the control path consisting of a finite state machine, must be modelled by the developer explicitly. Arbitrary concurrency can be explored this way (hardware systems are inherently parallel), but usually programmers think in an imperative and sequential (software) way on an algorithmic level, and rather on a hardware level, and concurrency always requires synchronization [3].

Though pure functional systems can be implemented directly using combinational logic only, digital logic signal path times restrict design to only flat functional blocks concerning hierarchy.

Finally RTL-synthesis transforms the hardware behaviour and/or structural description into a netlist of basic logic gates and registers suitable for FPGA configuration or ASIC layouts.

Although a generic program-controlled approach using microprocessors can be modelled, too, usually RTL level is used to describe an optimized application-specific implementation with respect to latency and data throughput rates. The application-specific processor approach results generally in minimized resources, but with the disadvantage of increased latency and reduced data throughput rates due to a lack of parallelism.

Behavioural Description on Algorithmic Level

An imperative programming language with an inherent strict sequential order model is used, mostly a C-like language, with high abstraction level from hardware, supplying abstraction which guides the designer from a software view to hardware RTL-level. That means the above RTL-level is synthesized from this higher-level-entry description. A C-like language is preferred by most programmers because of the familiarity with this widely used language in software programming [4].

The hardware system architecture derived and synthesized from the imperative programming approach can be distinguished into:

To be published! Personal author version.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission from the author(s): Dr. Stefan Bosse BSSLAB - Independent Research Laboratory, Bremen, Germany
Copyright 1.2.2009 .

1. pure application-specific RTL derived directly from the imperative description, mapping program instructions to states of FSMs (control path), and assignments to register access together with functional units (data path), and
2. a program-controlled approach with application-specific microprocessor cores, retrieved by the well-known hardware-software-co-design using hardware synthesis and software program compilation.

The advantages and disadvantages of these two system architectures were already discussed above.

Using a C-like language seems promising from the software point of view, but seems to be too restricted from the hardware point of view. Remember that synthesis of hardware circuits has several aims: 1. implementation of the algorithm itself, and 2. fulfilling time and resource constraints. The pure software approach can only cover the first point.

Additionally, C belongs to the class of direct memory-mapped languages using pointers. This restricts RTL synthesis to memory based architectures with limited sequential access.

Furthermore, an imperative language implies a strict sequential execution model. Usually, concurrency on software level is provided with (lightweight) processes executed in parallel, known as threads and multi-threading, most familiar implementations are POSIX threads, and using pipelining on instruction level and hardware level, actually with a trend to true hardware-supported multi-processing, assigning different threads or processes to different processors in parallel.

1.2 Concurrency and Synchronization

Two kinds of concurrency models can be distinguished:

Explicit Parallelism

The programming model explicitly describes parallelism, that means the programmer is responsible for modelling concurrency using for example processes or threads and synchronization primitives.

Implicit Parallelism

The compiler tries to explore and derive parallelism from an initially sequential code specification, described with an imperative language, or using functional languages with (hidden) inherent concurrency [6]. Mostly concurrency is derived from loops using unroll techniques with allocation of resources in parallel, but concurrency can be explored in blocks of data-independent expressions, too. For example, both expressions $x \leftarrow x + 1$ and $y \leftarrow y + 1$ can be scheduled (using RTL only) in the same time step requiring two adders.

Nearly always **synchronization** is required in the presence of parallel execution models. Usually, an algorithm is partitioned into functional blocks executed in parallel. For example, the calculation of a sum of a vector of size N can be easily partitioned into M fragments of size N/M (handled by loop constructs). For each fragment, the sum can be calculated independently. Synchronization is only necessary at the beginning of the calculation, indicating the availability

of new data (**data distribution phase**), using for example, an event signal triggered by one process, and at the end of each fragment calculation, for example using a counting semaphore (**data collection and merging phase**).

In the case of input and output data dependencies and dependencies of partial computation results, the synchronization becomes more complex. Other kinds of synchronization is required for access of shared resources, for example global registers, provided by mutual exclusion with different kinds of access scheduling. But parallelization and synchronization is well-known from software programming on parallel computers, with both shared and distributed memory architectures.

1.3 Closing the Gap

Explicit modelling of parallelism, for example using a multi-process model with interprocess-synchronization, has some advantages over an exclusive implicit modelling approach, exploring inherent parallelism of algorithms by the compiler:

1. A simple programming model enables better understanding of synthesized hardware architecture and provides faster error tracking, which has great impact on modelling and understanding hardware design,
2. the synthesis process is simplified, synchronization is visible to the programmer,
3. multi-process models are well known and accepted from software programming, with explicit inter-process communication,
4. user has more control over hardware resource allocation and the degree of concurrency.

The algorithmic high-level model used for the ConPro synthesis framework on the boundary between hardware- and software models described in this article tries to close the gap between the strict sequential software- and parallel hardware-model without using the traditional hardware-software-co-design. First of all it seems to be important that there are different levels of concurrency:

1. to increase the acceptance of hardware-synthesis for software-programmers and to simplify porting software algorithms to hardware, explicit described **coarse-grained concurrency** both in the **data- and control path** using a **multi-process model** with initial strict sequential instruction scheduling inside the process, similar to familiar processor-program-models, is provided, because many algorithms can only be coarse-grained partitioned for parallelization, and
2. **fine-grained concurrency** on **data-path** level for data-instructions in **bounded basic blocks** [5] within processes either from implicit contained parallelism derived from the synthesis-tool or explicitly specified by the programmer, is required for exploration of a higher degree of parallelism, not available on process level.

Fig. 1 shows these different levels of concurrency provided by ConPro.

The ConPro synthesis compiler translates the imperative program level into hardware behaviour descriptions on RTL level and creates VHDL output.

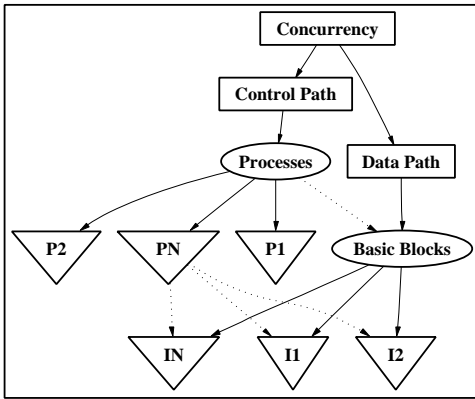


Figure 1: Different levels of concurrency are shown in the control and data path.

1.4 Requirements and Implementation

Process Model

Parallelism is modelled on highest level explicitly with processes executing concurrently. Processes and their instructions are mapped to states of a FSM using register-transfer-logic (RTL). Alternatively an application-specific microprocessor approach (s-RISC), fully adapted and scaled to each process, can be synthesized too (under development). In both cases, the source code is identical. In the default case, each process instruction requires at least one time step without further user constraints. The process model allows access of both shared global and local data objects.

Instructions and expressions can be bounded to blocks featuring different **synthesis parameters** and models:

- mainly different resource allocation and scheduling strategies,
- enabling of unrolling of loops
- and explicit parallelization,
- selecting different expression models like resource sharing or register inference and many more features.

Programming Language

A traditional **imperative programming language** is used with side effects specifying the program and data flow. By default, statements execute sequentially - one instruction each time step and in the order they appear in the program code. The language is strong typed. Expressions can contain operands from all basic data objects, including aggregation objects (arrays and structures), and all typical unary and binary functional operators (arithmetic, logic and bool type classes).

In addition to real data objects, there are **abstract objects**. Data objects are manipulated within expressions and assignments, whereas abstract objects are handled with their respective methods. Therefore,

some limited kind of object-orientated programming is supported, too.

Data objects are allocated statically only. **Functions** are supported, but restricted to non-recursive calls and scalar basic type function parameters. Only value semantic is supported, i.e. no dynamic references (pointers) to data objects are provided.

External hardware blocks modelled with a hardware description language (VHDL) can be easily integrated, for example optimized arithmetic functional blocks, bus and communication interface wrappers, and many more fields of applications requiring the integration of existing VHDL modules. This is an important feature of ConPro which distinguished this system from others like SystemC or HardwareC.

Data Types and Data Objects

Provided storage data objects are single **registers** with CREW- and **variables** with EREW-model stored in RAM blocks (though some limited CREW access behaviour can be implemented using multi-port RAM models). An arbitrary number of (named) **RAM blocks** are allowed. Additionally, the **signal** object exists for interfacing to hardware components, externally provided by VHDL models. The basic data type set consists of boolean, character, (unsigned) logic and (signed) integer types of arbitrary data width. Data objects can be shared by different processes. Access to **shared objects** is always automatically and implicitly guarded using a mutex. Aggregation is provided with arrays and structures. **Arrays** consist of elements of same data- and object type, **structures** consist of elements with different data types, but same object type, for example registers.

Abstract Objects

Beneath real data objects there are many built-in abstract objects (AO), for example processes itself, timers, serial communication links, and below described synchronization objects. In contrast to real data objects appearing in expressions, and AOs are handled with a defined set of methods.

Synchronization and Communication

The generic multi-process model requires explicit interprocess synchronization. Different synchronization objects are available, known from the multi-process software approach: **mutex**, **semaphore**, **signal event**, **channel**, **queue** and many more. Communication and data exchange between different processes including different kinds of synchronization can be done implicitly using (invisibly guarded) **shared data objects** like global registers or variables stored in RAM blocks, or explicitly using **channels or queues** providing some kind of a message passing model.

Architecture Model

Each process is modelled with a separate data- and control path using an RTL-architecture. The control path is implemented with a Moore-FSM. In the default case, each process instruction or bounded instruction block is typically mapped to one state of the FSM. Alternatively, a programm-controlled

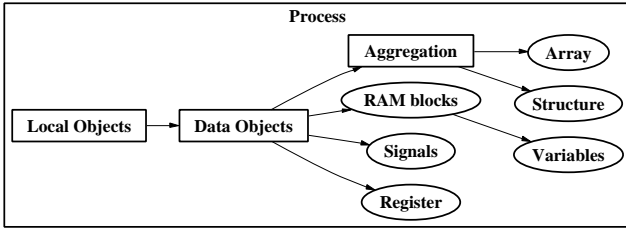


Figure 2: The process model consists of local data objects.

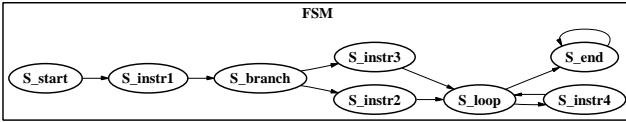


Figure 3: The process model consists of the control path with FSM. A state is mapped to a basic ConPro instruction or an instruction block.

implementation with a simple-RISC architecture can be used (actually under development), too. In this case, process instructions are compiled into processor instructions stored in a program RAM/ROM block.

There are different expression models supported:

1. **flat expression model**, results in a 1-to-1 mapping of (arithmetic, logical, boolean) operators to functional hardware blocks contained in expressions, and the expression is completely scheduled in one time step (or $N \geq 1$ time steps in the case of access of guarded objects),
2. **binary expression model**, same as 1., but the expression is splitted into binary operations with temporary register transfer, scheduling each operation in one time step, and finally
3. **the ALU model**, in which all operators are bound to one (or more) process-shared ALU block(s).

2. ARCHITECTURE MODEL AND SYNTHESIS

The control- and data flow is comparable to those of traditional multi-VLIW-processors. There is a global module and a local process level.

2.1 System Hierarchy

A ConPro module consists of global object implementations and processes. A process consists of local objects and a finite state machine with associated data paths. Figures 2 and 3 show this architecture model for one process, and figure 4 the architecture model of a modul. Processes are bound to a module. A module implements shared data objects like registers and abstract objects required for example for process synchronization, whereby processes can only implement local data objects.

2.2 Processes

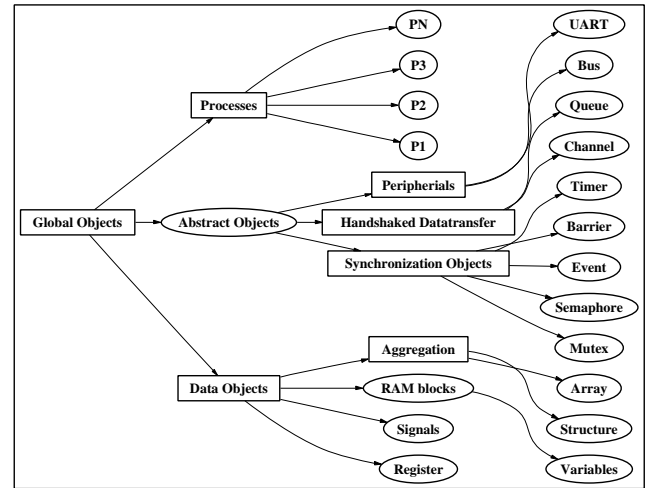


Figure 4: The module model contains several processes, data and abstract objects.

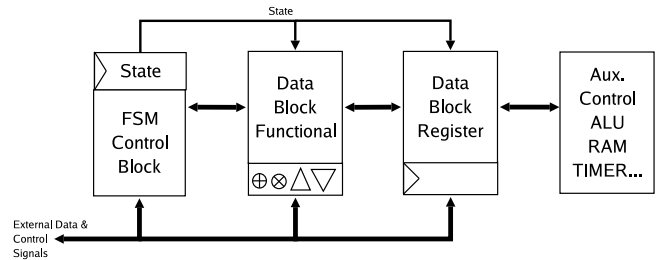


Figure 5: The process system architecture consists of control and data blocks.

All processes execute concurrently and independently. A process can be controlled by other processes. A process is handled by the programming language as an abstract object), too. The following set of methods is provided: $\{start, stop, call\}$. The latter one suspends the caller process until the called process reaches it endstate, required for shared function (block) implementation. There is a main (root) process, mainly used to control others processes. A process, except the main one, must be started explicitly.

Each process consists of a control and data path, modelled with different hardware blocks, each implemented with concurrent process environments in VHDL (see figure 5):

Control Block

Finite-State-Machine (Moore-type), clock synchronous. In the case of direct RTL synthesis of process instructions, typically each instruction or a bounded instruction block is mapped to one state, and in the case of RISC-synthesis, the FSM implements the command processing unit. The FSM is connected to external and internal data and control objects (signals).

Data Block - combinational logic

The data block (connected to external and internal signals) contains resources required for instruction mapping to hardware blocks:

- Data multiplexer,
- Functional units (arithmetical, logical and boolean kind),
- Control multiplexer for object control signals, like write-enable signals of a register or RAM addresses.

Data Block - register logic

Transitional register logic is separated from combinational data path logic:

- Local and temporary process registers, clock synchronous.

Auxiliary Object Blocks

Some more process-toplevel hardware-blocks:

- RAM blocks
- ALU blocks and functional modules, like multipliers
- Abstract objects, like stacks, timer, serial communication links.
- Process control logic

2.2.1 Process Block Interface and System Interconnect

Each ConPro process block is implemented with a separate VHDL entity component. Each process is connected to the system clock and reset signals {conpro_system_clk, conpro_system_reset}, common to all processes. Additionally, there are two individual signals controlling the process: {PRO_xx_ENABLE, PRO_xx_END}. Global objects (on ConPro-module toplevel) are connected to the process over the VHDL-port, too, shown in figure 6. The example 1 shows a VHDL port declaration for a ConPro process.

```
entity ex1_p1 is port(
  signal TIMER_t_GD: in std_logic;
  signal REG_x_RD: in std_logic_vector(7 downto 0);
  signal REG_x_WR: out std_logic_vector(7 downto 0);
  signal REG_x_WE: out std_logic;
  signal PRO_p1_ENABLE: in std_logic;
  signal PRO_p1_END: out std_logic;
  signal conpro_system_clk: in std_logic;
  signal conpro_system_reset: in std_logic );
end ex1_p1;
```

Example 1 Process port.

2.3 Process synchronization

There are different process synchronization features completely synthesized into hardware (see figure 7). All synchronization objects are handled like abstract objects. Access of abstract objects is done only by applying methods to the AOs. Method calls are mapped to hardware control signals (see fig. 6).

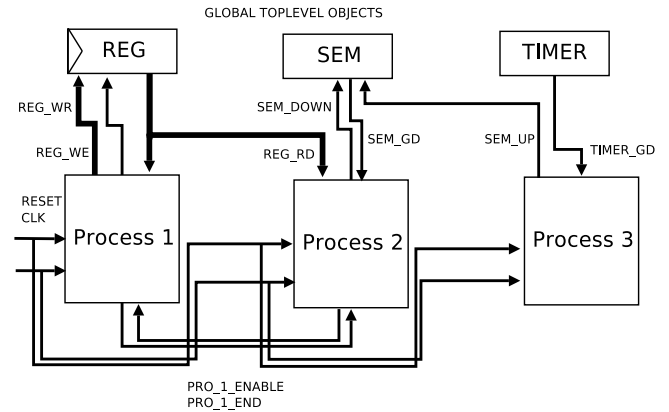


Figure 6: The process block interface and system interconnect requires different signals for the control and data path. Shared objects can be connected to different processes, requiring control signals for atomic access (called guards). All processes and objects are sourced by one system clock and reset signal, thus all functional blocks operate synchronously.

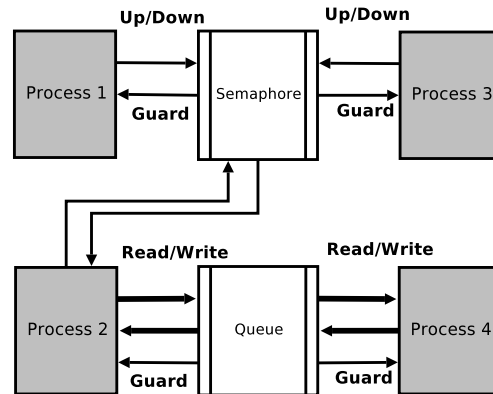


Figure 7: Multi-Process architecture and interprocess communication, implemented with semaphores and queues. Shared objects require guards for atomic access.

Mutex

A mutual exclusion lock is used when different processes access a shared non-atomic data resource, like structures. There are explicit and implicit locks: explicit using the AO mutex, implicit in all shared data objects like registers and variables, and for the access of this AO, too! Two different access schedulers are provided: 1. **static priority scheduling**, where each process requiring access gets a unique static priority at synthesis time, resulting in lowest hardware resource requirement, and 2. **dynamic FIFO-based priority scheduling**, resulting in much higher resource requirements. The simple but hardware cost efficient model can lead to race conditions, so that only some processes always get the lock, whereby other lower priority processes loose the race, the second method avoids racing. The set of operations (methods) is: `{lock,unlock,init}`.

Semaphore

A semaphore implements a counting lock with a value always equal or greater zero. The set of operations (methods) is: `{down,up,init}`. The *down* operation decrements the semaphore value until semaphore value zero is reached. The semaphore value can become never a negative value. Thus a process calling the *down* operation is blocked (suspended) until the down operation results in a semaphore value greater equal zero. The *up* operation increments the semaphore value, and it is a non-blocking operation. Semaphores can be used to implement producer-consumer tasks. Here again, both static and FIFO scheduling is provided.

Barrier

Similar to semaphores, a barrier enables the synchronization of a group of processes. A barrier has a value set reaching from zero to an upper bound level *U* (*U* = number of group members). All processes reaching the barrier (by calling the *await* method) are suspended until all *U* processes contributed to the barrier.

Event

Commonly called monitors: one or more processes wait for an (abstract) event, and one (or more) process(es) signal(s) the event. All processes waiting for the event are blocked until the event occurs. The set of operations is: (methods): `{await,wakeup}`.

Timer

The timer object is similar to the event synchronization. A timer signals periodically (or one-shot) an event. Processes can wait for this event, while they are blocked until the timer event occurs. Timers can be started, stopped, and the cycle time can be selected (also during runtime). The set of operations is (methods): `{set,init,start,stop,await}`. The *set* operation is used to set the timer interval in specified time units.

Queue [Core]

A data queue supports two operations: *read* and *write* of a data word. A process performing the read operation is blocked until the queue contains at

least one data word. A process performing the write operation is blocked until at least one data word space is available in the queue. A queue is modelled with a FirstIn-FirstOut (FIFO) behaviour. The queue object belongs to the class of core data objects, therefore queues can be used in expressions like any other data object. Queues can be of any real data type, including structures of different types, implemented with different queues sharing one access scheduler.

Channel [Core]

A channel is used to synchronously transfer data between processes, similar to a queue. There are two different channel models: buffered and unbuffered. The buffered version is just a queue with one cell, but the unbuffered version consists only of combinational logic providing a handshaken communication link. A channel is also a basic core data object, and can be used in expressions, too.

The next example 2 shows three synchronization object definitions and their usage inside processes.

```
1: object sem: semaphore;
2: object ev: event;
3: object mu: mutex;
4: channel ch: logic[8] with model=buffered;
5: -- two concurrent processes
6: process p1:           process p2:
7: begin                 begin
8:   reg x:logic[8];      ch <- 'A';
9:   x <- ch;             ...
10:  sem.down ();         mu.lock ();
11:  ev.await ();         ev.wakeup();
12:  ...                 mu.unlock ();
13: end;                  end;
```

Example 2 Examples for interprocess communication.

2.4 Functions

User-defined functions can be implemented in two different ways: 1. inlined not-shared function macros and 2. shared function blocks. In the first case, all function instructions are replaced by the function call, function parameters are replaced by their respective arguments. In the second case, a function is modelled using the above described process with an additional function control block containing a function call lock (access scheduler) and registers required for passing function parameters and results. Only call by value arguments of atomic objects can be implemented. The remaining functionality is provided by the underlying process model using the *call* method.

2.5 μ -Code Intermediate Representation

Source code is parsed and analyzed in the first compilation step into symbol lists and program syntax graphs. A node of the program graph is a (complex) instruction, for example a loop, or a block environment. There is one graph for each process. In the second compilation step, the process instruction graph structure is flattened to a linear list of microcode program instructions (μ -Code). The main advantage of this

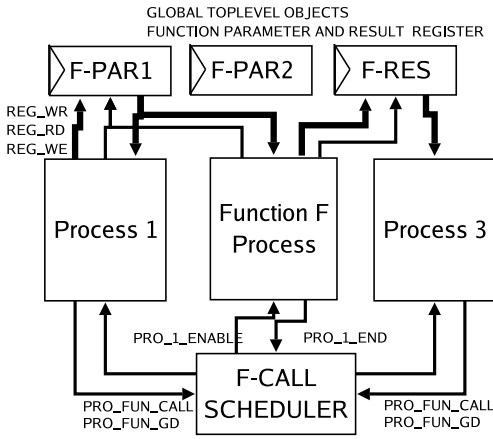


Figure 8: Shared function blocks are implemented with a process block and a function call scheduler.

approach is the simplified handling and optimization of a linear list of simple instructions from a small set of instructions rather than a complex graph structure with several different complex programming environments and control statements, for example loops and branches. Furthermore, this μ -Code approach provides both alternative inference and instantiation of microprocessor cores (s-RISC) and RTL, too. Additionally, μ -Code can be both exported and imported using assembler code. Therefore two different programming entry levels are available: imperative high- and low level language models. Though control level concurrency can be only implemented using processes, data path concurrency can be modelled both using multiple processes and bounded instruction blocks inside a process. This data path concurrency is available in the μ -Code representation, too, providing the VLIW execution model.

2.5.1 μ -Code instructions

Table 1 gives an overview of the instructions required for process data- and control-flow implementation. Table 2 explains operands used in expressions.

The following examples 3 and 4 show the mapping of Con-Pro process instructions to μ -Code. The `bind` instruction is used a.) for binding of data and control instructions (`expr & falsejump`) and b.) for binding of concurrent data path instructions, either bound by the programmer (lines 12-15) or explored during synthesis by the compiler.

```

1: reg x: logic[8];
2: object t: timer;
3: process p1:
4: begin
5:   reg a,b: int[8];
6:   for i = 1 to 10 do
7:     begin
8:       x <- x + 1;
9:       if x = 10 then x <- 11 else x <- 0;
10:      t.await ();
11:     end;
12:     begin
13:       a <- a + 1;
14:       b <- b + 1;
15:     end with bind;
16: end;

```

Example 3 ConPro process instructions, source code.

This process code is compiled to MicroCode, shown in example 4.

```

i1_for_loop:   move (LOOP_i_0,1)
i1_for_loop_cond: bind (2)
                expr ($immed.[1],10,>=,LOOP_i_0)
                falsejump ($immed.[1],%END)
i2_assign:    expr (x,x+,1)
                nop
i2_assign_end: nop
i3_branch:   bind (2)
                expr ($immed.[1],x,=,10)
                falsejump ($immed.[1],i5_assign)
i4_assign:   move (x,11)
i4_assign_end: jump (i5_fun)
i5_assign:   move (x,0)
i5_assign_end: nop
i3_branch_end: nop
i6_fun:     fun(t.await,())
i6_fun_end:
i1_for_loop_incr: bind (3)
                expr (LOOP_i_0,LOOP_i_0+,1)
                nop
                jump (i1_for_loop_cond)
i1_for_loop_end: nop
i7_bind_to_8: bind (4)
                expr (a,a+,1)
                nop
                expr (b,b+,1)

```

Example 4 MicroCode of the compiled process.

Mnemonics	Descriptions
<code>move(dst,src)</code>	Data transfer
<code>expr(dst,op1,op,op2)</code>	Data transfer with binary expression evaluation
<code>jump(label)</code>	Unconditional branch
<code>falsejump(cond,label)</code>	Conditional branch
<code>bind(n)</code>	Bind n following instructions to a parallel execution block
<code>fun(obj.meth,args)</code>	Abstract Object Method Call
<code>nop</code>	No operation place holder, mostly a result of optimization

Table 1: μ -Code instructions.

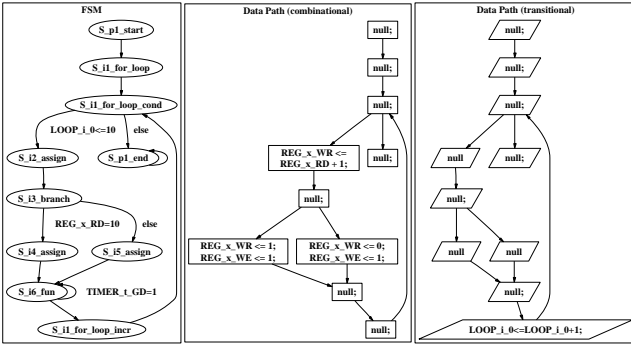


Figure 9: FSM states and associated data paths (combinational and transitional) derived from example (2) are shown.

Mnemonics	Descriptions
$v = [0..9]^+$	Constant value
$id = [a..b A..Z _] [a..b A..Z _ 0..9]^+$	Identifier for data objects (registers, variables, signals)
$\$immed.[v]$	Virtual data object only used immediatly in an bounded expression
$\$tmp.[v]$	Numbered temporary register
$\$tmp.[id]$	Temporary register associated with a data object.

Table 2: μ -Code operands described with pseudo-regular expressions.

2.6 RTL architecture

The μ -Code of a process is translated into a state graph and finally into VHDL code. The state graph is mapped to and synthesized in a FSM. There are different blocks building the RTL architecture of a ConPro process:

1. Finite-State-Machine with state transitions mapped to the μ -Code instructions,
2. Data path (I), implements expressions with global objects, containing arithmetic, logical and relational functional units, and data selectors (multiplexers), provides additionally auxilliary signals for global object access, pure combinational logic,
3. Data path (II), implements expressions with local and temporary registers, containing arithmetic, logical, and relational operators and data multiplexers, combinational and transitional logic.

Figure 9 shows a part of the synthesized state diagram of the FSM from example (2) and the corresponding data paths.

2.7 Synthesis Overview

Figure 10 shows details about the overall synthesis process. The analysis pass transforms the source syntax graph, which contains both module definitions and process instructions, into (complex) process instruction graphs, one for each process. All global and process local objects are stored in symbol tables.

The process instruction graphs are processed with the reference stack scheduler, performing the first scheduling pass, explained in section 2.9, followed by several optimizations, like constant folding in expressions, moving of loop- and branch-invariant instructions outside the loop or branch environment, and removing of dead objects and instructions. After the optimization pass, the reference stack pass is applied again, until no further optimization progress can be performed.

After first pass of scheduling and optimization on process instruction level, these instructions are compiled into μ Code assembler instruction lists, following the second scheduling and optimization pass on the next lower level. The expression scheduler is explained in section 2.10 and provides time constraint scheduling. After this scheduler step, the μ Code is partitioned into basic blocks, see section 2.11. These basic blocks are bounded data instruction blocks (containing only assignments) within a control path boundary, i.e., a block without any side traces (jumps inside). Using a data dependency graph derived from each basicblock, the basicblock-scheduler tries to schedule all N instructions of this block initially scheduled in at least N time units in fewer time units $M \leq N$. The basicblock scheduler is explained in section 2.11. No further resources are allocated in this pass, but the execution time can be reduced. The μ Code level was chosen for this third scheduling and optimization pass because of presence of auxilliary expressions inferred for example from loops (index register expressions).

Now process ALUs and temporary registers can be allocated and mapped.

To get information about the scheduling result, a frame time calculation is performed, providing time unit values for each basic block and block environments linked with source code locations. These timing information can be used for incremental synthesis to optimize latency and/or fulfill timing constraints.

The optimized μ CODE instruction list for each process is synthesized into FSM states. Finally, all toplevel objects like registers and synchronization objects and the control FSM itself are synthesized into RTL using VHDL output.

During the synthesis process **SYNTAX** \rightarrow **μ CODE** \rightarrow **FSM-STATES** \rightarrow **RTL-VHDL**, initial block structures from high level control environments (for example branches) are kept, together with source code informations.

2.8 Resource Allocation and Scheduling

Resource allocation is mainly done statically and is determined by one of the user-specified expression models: **1.** flat, **2.** binary, **3.** ALU.

Scheduling of process instructions is performed in different steps on different layers, though the first step has impact on resource allocation, too:

Reference Stack Scheduler

Propagation of storage objects (assignments to objects Θ_i , registers or variables, further only variables

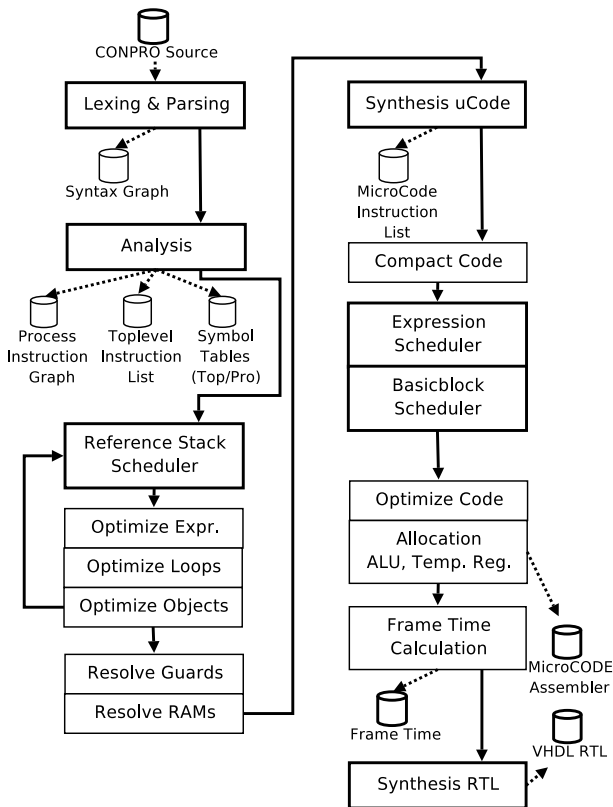


Figure 10: This figure gives an overview about the ConPro synthesis process, from source code to VHDL RTL.

named) and references of objects in expressions is evaluated into an expression sequence stack. The reference stack algorithm efficiently resolves constant and variable propagation, dead objects and code, conditional assignment to variables, and multiple assignments to variables. Many of the control steps (instructions) needed to implement assignments to variables can be eliminated. An expression is delayed as last as possible. Forward substitution and constant folding methods can create reduced and optimized meta expressions.

The reference stack scheduler is applied on program syntax graph level. Initially, these meta expressions are mapped to one time unit and scheduled with ALAP scheduling behaviour conserving data dependencies. Consider this example:

```
x <- 1;
y <- x + 1;
x <- y;
```

This source code fragment consists of three instructions (assignments). Without any further automatic scheduling and constraints, these instructions require three time steps, each for one assignment. This is the default scheduling model. Using the reference stack algorithm together with constant folding results in:

```
y <- x + 1 => 1 + 1 => 2;
x <- y => x + 1 => 1 + 1 => 2;
```

In case that the variable x is no further referenced, the first assignment can be removed (dead code elimination). The next example shows the transformation of an expression sequence of objects x and y into meta expressions, only one for each object, using forward substitution:

```
x <- a;
x <- x + b;
x <- x + 1;
y <- x - 1 + c;
x <- y;
```

Here we get a compacted meta expression (again with additional constant folding):

```
y <- a + b + 1 - 1 + c => a+b+c;
x <- a+b+c;
```

μ Code-Synthesis Scheduling

Compilation of complex process instructions into μ -Code instructions determines basic scheduling and resource allocation. This resource allocation and scheduling can be parameterized by the programmer on syntax block level.

Time-Constrained Expression Scheduler

Initially, an expression, either the original one provided by the programmer or the optimized meta expression, is scheduled into one time unit (on condition that flat expression model is used), meaning an instruction is mapped to one state of the control FSM. But a complex expression can result in a chain

of functional units with different execution times. For operational and stability reasons of the synthesized digital logic circuit, the longest path time required for a chain of functional units may not exceed one clock cycle time in a synchronous system (worst case heuristic). Depending on the execution time for each single functional unit (for example an adder), subexpressions must be split and scheduled into different time units using temporary registers.

The expression scheduler evaluates all (meta) expressions under given time constraints with ASAP scheduling behaviour. The timing decision is derived from user-supplied time weight functions for each operation, for example addition or logic operations. The weight value of an operation is in the range between 0.0 and 1.0. Expressions are bound to one time step until the sum of all weights exceeds 1.0 (one half clock cycle).

Time-constrained scheduling of expressions is done on μ -Code level rather than (source) program graph level because expressions are already split into binary operations (with two operands). Time and area constraints will assign time steps to a block of binary expressions. Consider the μ -Code from the last example:

```
bind (2)
expr($immed.[1],a,+,b)
expr(y,$immed.[1],+,c)
```

Before scheduling, the expression is scheduled into one time step, specified by the bind instruction. The `$immed` operand is only a symbolic place holder. Suppose the system clock frequency is 10 MHz, with a clock cycle time of 100 ns, and a 32 bit addition requires 70 ns (all above registers should have 32 bit data width, too), the scheduler must split the expression into two time steps with a temporary register inferred (therefore this scheduling step also has some impact on resource allocation):

```
expr($tmp.[1],a,+,b)
expr(y,$tmp.[1],+,c)
```

Basic-Block Scheduler

The previous scheduling step results in a reduction of concurrency (here of subexpressions) and an increase of execution time. But in general there are sequences of different and data-independent expressions, which can be scheduled in the same time unit without an increase of resources. Moreover, a smaller FSM results, which decreases resource area!

Data path concurrency can be finally explored again on μ -Code-level in basic blocks initially consisting of a sequence of expressions (assignments), discussed later.

2.9 Reference Stack

The reference stack algorithm (derived from original algorithm in [3] and [8]) is the first central step used in automatic scheduling and resource allocation with the aim of optimization of expressions and assignments concerning both time and area (logic gates). It is a symbolic computation method.

For each data object Θ ever appearing on the left-hand-side of an assignment, there is a one reference stack $T(\Theta)$.

Each time an assignment of a new value (expression) occurs, this value is added to the top of the stack. The top of the object stack gives the actual value. Stack elements are indexed from 0 to $N-1$, whereby $N-1$ is the top of the stack. If an object is referenced on the right-hand-side of an expression, the object is replaced by the actual top of stack reference. For example:

```
1. x <- 1;
   T(x)=[1]
2. y <- x+1; => T(x,0)+1 => 1+1 2;
   T(y)=[2]
3. x <- 2;
   T(x)=[2;1]
4. z <- u-x*y; => T(u,0)-T(x,1)*T(y,0) => T(u,0)-4
   T(z)=[4], T(u)=[*]
5. u <- x*u+1; => T(x,1)*T(u,0)+1 => 2*T(u,0)+1
   T(u)=[T(x,1)*T(u,0)+1;*]
```

In the last expression there is an object appearing both on LHS and RHS. The first stack element $T(u,1)$ is undefined (*), because the RHS occurrence of the object is the first found without any further knowledge about the value.

Using ALAP scheduling behaviour with forward substitution and symbolic simplifications (constant folding, identity reduction...), the lines 1..5 are reduced to only four instead five assignments emitted as last as possible, for example at the end of an instruction block, each for one modified data object, just assigning the top (the most actual value) of the stack (flush of reference stack top element):

```
z <- u-4;
y <- 2;
x <- 2;
u <- 2*u+1;
```

Some care must be taken in the presence of data dependencies between expressions and assignments, before and within loop and branch contexts. For example the assignment for object u must be emitted after the assignment of z .

Loops and branches must be handled in a different way. If an object was modified inside a loop body, a flush of the object assignment is required with top of reference stack before the loop is entered.

2.9.1 Rules and Algorithm

1. Each data object Θ is traced with its own **reference stack** T :

$$T(\Theta)=[\text{expr}_{N-1};\text{expr}_{N-2};\dots;\text{expr}_0;]$$

The newest/last one expr_{N-1} is the top of the stack. Tracked data objects are: *registers* and *variables*. Stack elements can be of different type:

```
type stackelement =
  RS_self( $\Theta$ ) | RS_expr(expr) |
  RS_branch(B0:[stackelement;..];B1:[.];..) |
  RS_loop([stackelement;..]) |
  RS_ref (path list)
```

2. Scheduling of an **assignment**

$$\text{LHS}(\Theta) \leftarrow \text{RHS}(\Theta_1;\Theta_2;\dots)$$

is delayed inside a scheduling block B as late/last as possible (ALAP).

3. Create a **new reference stack** of object Θ on first occurrence of Θ on LHS in an assignment:

$$T(\Theta)=[RS_expr(RHS)]$$

4. Create a **new reference stack** of object Θ on first occurrence of Θ on RHS (assignment, boolean expression): $T(\Theta)=[RS_self(\Theta)]$
5. Flush all delayed assignments at the end of the scheduling block B and if there is no control statement like branches and loops before. After a flush, the reference stack top is again **RS_self**.

Resolve dependencies of objects and stack top expressions, sort assignments before flushing in correct dependency order.

Relocate all references in expressions in the flushed assignments.

6. Branches:

[A branch consists of a branch selection expression E_b and several conditional blocks B_i depending on the evaluation of the selection expression]

- (a) Evaluate all conditional blocks $B_{cond,j}$ of the branch, total b different blocks, each for one branch condition, $j=0\dots b-1$.

All objects modified in the conditional block get a **RS_branch**($[RS_expr;T_0]$) stack element on the top of stack. All further modifications are stored in this **RS_branch** stack element.

The initial top element T_0 of the branch is either a reference **RS_ref** to a previous expression or **RS_self**.

Objects appearing first time on the RHS get a **RS_branch**($[T_0]$) top stack element.

Each conditional block is handled with its own substack T_j with j =Number of conditional blocks:

$$T(\Theta)=[RS_branch[T_{j-1};T_{j-2};\dots;T_0];\dots]$$

- (b) Objects only appearing on the RHS in each branch, that means they were not modified, $T_{j-1}=[RS_self|RS_ref] \mid RS_expr \notin T_{j-1}$ for all $j=0\dots b-1$, are transformed into references to previous stack element T_{n-1} in the case $T_{n-1}=RS_expr$ (top of stack before branch).

$$\begin{aligned} T(X) &= [RS_branch([RS_self]; \\ &\quad [RS_self]); \\ T_{n-1}; \dots] \\ &\Rightarrow \\ T(X) &= [RS_branch([RS_ref(T_{n-1})]; \\ &\quad [RS_ref(T_{n-1})]); \\ T_{n-1}; \dots] \end{aligned}$$

- (c) After all conditional blocks have been evaluated, all objects with **RS_branch**($[T_{j-1};\dots]$) and $T_{j-1} \neq [RS_self] \mid RS_expr \in T_{j-1}$ (modified inside block) must be flushed before the branch instructions (exception: all possible branches modify Θ)

if there is an expression element **RS_expr** before the actual branch stack, and at the end of the conditional block they were modified.

- (d) If the branch is complete (all values covered), and an object Θ holds the same expression **RS_expr**(E) on the top of all branch stacks, and a.) iff Θ is not referenced in any branch and the branch selection expression (except reference of Θ in E), E can be moved one time before the branch, and b.) iff Θ is only referenced in the branch selection expression, E can be moved after the branch.
- (e) Modification of the stack is required after branch: If the object Θ was not modified, but only referenced, and the previous top element (before the branch) was an expression **RS_expr**, a reference **RS_ref**(**RS_expr**) is made to this last expression (new top). If the object was modified, **RS_self** is the new top of the stack.

7. Loops:

[A loop consists of a loop expression E_l and one loop block B_l depending on the evaluation of the loop expression]

- (a) Evaluate all objects appearing in expression of counting loop before loop block is evaluated, flush in case of conditional loops.
- (b) Evaluate body block B_{loop} of the loop: Objects appearing first time in loop body on RHS get a **RS_loop**($[T_0]$) top stack element, with $T_0=RS_self$. All objects modified the first time inside the loop block get a **RS_loop**(T_{loop}) expression on the top of stack. All further modifications are stored in this **RS_loop**(T_{loop}) expression.
- (c) Objects only appearing on the RHS inside the loop body, that means $T_{loop}=[RS_self] \mid RS_expr \notin T_{loop}$, are transformed into references **RS_ref**(T_{n-1}) to previous stack element in the case $T_{n-1}=RS_expr$.
- (d) Modification of the stack is required after loop evaluation: If the object Θ was not modified, but only referenced, and the previous top element (before the branch) was an expression **RS_expr**, a reference **RS_ref**(**RS_expr**) is made to this last expression (new top). If the object was modified, **RS_self** is the new top of the stack.

2.10 Expression Scheduler

Time-constraint driven scheduling is performed on μ -Code instruction level with ASAP time behaviour. The bounded μ -Code instruction block of an expression resulting either from the last symbolic computation step or from program-bounded blocks using the reference stack method (initially bound to one time unit) is partitioned into subexpressions which are scheduled in $N \geq 1$ time units according to user constraint specifications. Additional temporary registers are inferred for each additional subexpression.

For each functional arithmetic or logical operator, or a group of these operators, a timing function can be specified. This function $\Gamma(DW,CT)$ is used for the scheduling

of each individual (binary) subexpression, whereby DW is the actual data width used in the subexpression and CT the system clock cycle time. The function must return a timing cost parameter in the range [0.0,1.0] specifying the amount of a clock cycle required for this operation, that means an estimation of the time of the longest signal path of the functional block, either dependent on the data width in cascaded digital logic architectures, or independent in the case of flat architectures. These functions are specified on module toplevel using the `system` object:

```
open System;
object sys: system;
sys.clock (30 megahz);
sys.res_time("+,-","logic","DW * 2 nanosec / CT");
sys.res_time("lor","logic","5 nanosec / CT");
```

The next example 5 is applied to the previous settings.

```
====>> After Reference Stack Scheduler
var v: logic[8] in ram; ...
a <- v+v+v+v+v+v+32; ...
====>> After Synthesis
i4_assign: move($tmp.[v],v)
i4_assign_S0: bind (6)
    expr ($immed.[6],$tmp.[v],+,32)
    expr ($immed.[5],$immed.[6],+,$tmp.[v])
    expr ($immed.[4],$immed.[5],+,$tmp.[v])
    expr ($immed.[3],$immed.[4],+,$tmp.[v])
    expr ($immed.[2],$immed.[3],+,$tmp.[v])
    expr (a,$immed.[2],+,$tmp.[v])
====>> After Expression Scheduler
i4_assign: move($tmp.[v],v)
i4_assign_S0: bind (2)
    expr ($immed.[6],$tmp.[v],+,32)
    expr ($tmp.[%I5],$immed.[6],+,$tmp.[v])
i4_assign_S1: bind(2)
    expr ($immed.[4],$tmp.[%I5],+,$tmp.[v])
    expr ($tmp.[%I3],$immed.[4],+,$tmp.[v])
i4_assign_S2: bind(2)
    expr ($immed.[2],$tmp.[%I3],+,$tmp.[v])
    expr (a,$immed.[2],+,$tmp.[v])
```

Example 5 Results from the Expression Scheduler. An expression was scheduled in three time steps.

2.11 Basicblock Scheduler

Finally, data path analysis of basic blocks (blocks without any control side effects) is performed on μ Code-level to bind as many as possible independent object assignments and expressions to one schedule time step.

For this purpose, the control path of each process is partitioned into major blocks. A major block can be of kind DATA, CONTROL, or just FIXED, and has one control flow entry point at the beginning. Major blocks of kind DATA are further partitioned into minor blocks. A minor block encapsulates one atomic data assignment. A set of data dependency graphs (DDG) is built from those minor blocks. Each single graph describes data object dependencies and flow inside the major block. Each DDG of the set of DDGs can be treated independently.

Rules to build a DDG forest:

1. Each minor block contains two lists: children C & parents P. The DDGs are built from these linked lists.

2. Each time a new minor block M is inserted into a DDG, the dependency of LHS and RHS objects of M is checked to all LHS and RHS objects (can be bounded blocks with several assignments!) of already inserted minor blocks M'.

If there is a dependency, then both linked lists are updated:

```
IF FOREACH M': LHS(M)=RHS(M') OR
                LHS(M)=LHS(M') OR
                RHS(M)=LHS(M') THEN
    IF M ∉ C' THEN C' ← C' + [M]
    IF M' ∉ P THEN P ← P + [M']
END IF
```

3. After all minor blocks have been inserted into at least one DDG, the DDG forest must be build from the minor block list.

A root of one DDG is a minor block with P=[] (and LHS ≠ []).

An ASAP scheduler now extracts from each (independent) DDG at least one node (minor block) and binds all minor blocks (assignments) to one time step until there are no more available minor blocks inside the DDG forest. There may be only one minor block with a guarded (global shared) data object scheduled in one time step.

Though different minor blocks from a DDG are dependent, some minor blocks of one DDG can be scheduled in the same time step, too, without violation of sequence order of the respective instructions. For example $x \leftarrow T+1$ and $T \leftarrow y-1$ can be executed in the same step, presuppose the behaviour of RTL and that the RHS is always evaluated first, than the assignment is done (intrinsic hardware model of registers).

The next example 6 illustrates the effect of reference and basicblock scheduling.

```
process pX:
begin
reg a: int[32];
a <- 23;
x <- 0;
for i = 1 to 10 do
begin
x <- x + a;
end;
end with schedule=custom(X);
```

Example 6 ConPro source code used for demonstration of different scheduler settings

First default scheduling `X=default` is used. This result in less optimized code, shown in example 7.

```
i1_assign: move (a,23)
i1_assign_end: nop
i2_assign: move (x,0)
i2_assign_end: nop
i3_for_loop: move (LOOP_i_0,1)
i3_for_loop_cond: bind(2)
    expr ($immed.[1],10,>=,LOOP_i_0)
```

```

falsejump ($immed.[1],%END)
i4_assign:  expr (x,x+,a)
           nop
i4_assign_end: nop
           nop
i3_for_loop_incr: bind(3)
                expr (LOOP_i_0,LOOP_i_0+,1)
                nop
                jump (i3_for_loop_cond)
i3_for_loop_end:

```

Example 7 Synthesis result from example 6 with default scheduler settings.

Now the reference scheduler is invoked, $X=refstack$.

```

i1_assign:  move (x,0)
i1_assign_end: nop
i2_for_loop: move (LOOP_i_1,1)
i2_for_loop_cond: bind (2)
                  expr ($immed.[1],10,>=,LOOP_i_1)
                  falsejump ($immed.[1],i4_assign)
i3_assign:  expr (x,x+,23)
           nop
i3_assign_end: nop
i2_for_loop_incr: bind (3)
                  expr (LOOP_i_1,LOOP_i_1+,1)
                  nop
                  jump (i2_for_loop_cond)
i2_for_loop_end: nop
i4_assign:  move (a,23)
i4_assign_end:

```

Example 8 Synthesis result from example 6 with $X=refstack$.

Finally the reference and basicblock scheduler are invoked, $X=refstack,basicblock$.

```

BLOCKBOUND5_1: bind (3)
                i1_assign: move (x,0)
                i1_assign_end: nop
                i2_for_loop: move (LOOP_i_2,1)
i2_for_loop_cond: bind (2)
                  expr ($immed.[1],10,>=,LOOP_i_2)
                  falsejump ($immed.[1],BLOCKBOUND1_1)
BLOCKBOUND3_1: bind (6)
                i3_assign: expr (x,x+,23)
                nop
                i3_assign_end: nop
i2_for_loop_incr: expr (LOOP_i_2,LOOP_i_2+,1)
                nop
                jump (i2_for_loop_cond)
i2_for_loop_end: nop
BLOCKBOUND1_1: bind (1)
                i4_assign: move (a,23)
                i4_assign_end:

```

Example 9 Synthesis result from example 6 with $X=refstack,basicblock$.

The last code result is best optimized regarding required area resource and time. Table 3 summarizes the synthesis results.

Scheduling	Time Units
default	33 TU
refstack	33 TU
refstack & basicblock	22 TU

Table 3: Synthesis results from above example code.

3. DESIGN EXAMPLES AND SYNTHESIS RESULTS

3.1 Parity Calculator

The first real-world example demonstrates both the default- and user-guided synthesis process combined with automatic optimization and scheduling using the reference stack, resulting in different behaviours, time and gate resources. The initial algorithm implementation uses default ConPro synthesis behaviour: flat expressions (expression boundary is kept untouched), no automatic scheduling, the instructions are scheduled in the order they appear, temporary registers are reused (shared model), for-loops are synthesized into sequential loops, not unrolled. The algorithm, implemented with a shared function block, calculates the parity of logic vector with a width of WIDTH bits. Below the important parts of the ConPro source code are shown in example 10.

```

1: const WIDTH: value := 64;
2: reg d: logic[WIDTH];
3: function parity (x: logic[WIDTH])
   return (p: logic):
4: begin
5:   p <- 0;
6:   for i = 0 to WIDTH-1 do
7:     begin
8:       p <- p lxor x[i];
9:     end <with BLOCK_PARAMETER>;
10: end;
11: process main:
12: begin
13:   reg p: logic;
14:   d <- 0x12345670;
15:   p <- parity(d);
16: end;

```

Example 10 ConPro source code for the parity calculator.

Table 4 shows synthesis results with different block parameters<with BLOCK_PARAMETER> of the for-loop. The number of time units (TU) required for the function call is calculated by the ConPro compiler, the equivalent gate count and the longest path time is derived from standard-cell ASIC synthesis (using the SXLIB library from Lip6, [7]). The used resource timing functions were (assuming simple models of ripple-carry adders and logical operators):

```

("+,-","logic","(DW * 2 nanosec) / CT")
("lxor","logic","1 naosec / CT")

```

Block Parameter	Time, Gates, Register, Path
default	130 TU, 310, 11, 5.9 ns
unroll	65 TU, 424, 7, 5.05 ns
unroll & schedule=auto*	2 TU, 219, 2, 3.42 ns

Table 4: Synthesis results of parity calculator with different for-loop block parameters. There are additional 5 TU for function argument and result transfers (*: refs-tack,expr,basicblock).

The default 1:1 scheduling model requires the largest amount of time units, but not the largest amount of digital logic gates (area). The automatically scheduled model results in lowest execution time, area and lowest longest path time.

3.2 Simple Protocol Stack

The next example implements a simple protocol stack for serial communication, using a built-in RS232 UART module and remote procedure call semantic with messages. There is a register file (RAM block) with 256 cells of 12 bit data width. Remote procedure calls can interact with this register file. There are two messages: the read message starting with an 'R' character followed by an ASCII coded two-digit hexadecimal register file address, returning the content of the addressed cell (three hexadecimal digits, ASCII coded), and the write message, starting with a 'W' character, followed by the cell address and the new data to be written into the cell.

```

-- UART
object u1: uart;
  u1.baud (9600);
  u1.rxd (top.rxd);
  u1.txd (top.txd);
const RU_SIZE: value := 256;
-- Register dual port RAM
block ru_ram with model=dualport;
array ru: var[RU_SIZE] of logic[12] in ru_ram;
1: function char_to_num(x:logic[8]) return (y:logic[8]):
2: begin
3:   if x >= '0' and x <= '9' then
4:     y <- x - '0'
5:   else if x >= 'A' and x <= 'F' then
6:     y <- x - 'A' + 10
7:   else
8:     y <- 0;
9: end;
10: function char_of_num(x:logic[4]) return (y:logic[8]):
11: begin
12:   if x <= 9 then
13:     y <- x + '0'
14:   else
15:     y <- x + 'A' - 10;
16: end;
17: process inter_p1:
18: begin
19:   reg data_c,addr,data_x,data_x2: logic[8];
20:   reg data_12: logic[12];
21:   u1.baud (38400);
22:   u1.start ();
23:   always do
24:   begin

```

```

25:   data_c <- u1;
26:   match data_c with
27:   begin
28:     when 'R':
29:     begin
30:       -- get address bytes
31:       addr <- 0;
32:       for i = 1 to 2 do
33:       begin
34:         data_c <- u1;
35:         data_x <- char_to_num(data_c);
36:         addr <- addr lsl 4;
37:         addr <- addr + data_x;
38:       end;
39:       data_12 <- ru.[addr];
40:       -- send data bytes
41:       for i = 1 to 3 do
42:       begin
43:         data_c <- char_of_num(data_12[8 to 11]);
44:         u1 <- data_c;
45:         data_12 <- data_12 lsl 4;
46:       end;
47:     end;
48:     when 'W':
49:     begin
50:       -- get address bytes
51:       addr <- 0;
52:       data_x2 <- 0;
53:       for i = 1 to 2 do
54:       begin
55:         data_c <- u1;
56:         data_x <- char_to_num(data_c);
57:         addr <- addr lsl 4;
58:         addr <- addr + data_x;
59:       end;
60:       -- get data bytes
61:       data_12 <- 0;
62:       for i = 1 to 3 do
63:       begin
64:         data_c <- u1;
65:         data_x <- char_to_num(data_c);
66:         data_12 <- data_12 lsl 4;
67:         data_12 <- data_12 + data_x;
68:       end;
69:       ru.[addr] <- data_12;
70:       u1 <- '0'; u1 <- 'K';
71:     end;
72:   end;
73: end;

```

Example 11 ConPro source code of the serial communication protocol stack implementation.

Table 5 shows synthesis results with different block parameters<with BLOCK_PARAMETER> of the for-loop. The number of time units (TU) required for the function call is calculated by the ConPro compiler, the equivalent gate count and the longest path time are derived from standard-cell ASIC synthesis (using the SXLIB library from Lip6, [7]). The used resource timing functions were (assuming simple models of ripple-carry adders and logical operators):

```

("+,-","logic","(DW * 2 nanosec) / CT")
("lxor","logic","1 naosec / CT")

```

Block Parameter	Time, States, Gates, Register, Path
default	$\geq 4.. \geq 73$ TU, 61, 1463, 111, 6.9 ns
schedule=auto*	$\geq 4.. \geq 71$ TU, 53, 1380, 111, 5.0 ns

Table 5: Synthesis results of the protocol stack with different scheduling parameters. (*: refstack,expr,basicblock).

Here, the difference between the default and full automatic scheduling is small concerning digital logic gate count and concerning the required boundary limits of the execution time. Only the number of FSM states differs about 15% and the longest path about 38%. This example code cannot be optimized anymore.

3.3 Robot Joint Control

The MOTCON6 motor control board [1][2] was fully implemented using FPGA technologies. The system architecture contains an enhanced version of the protocol communication stack shown in section 3.2, a PID position controller and several peripherals like PWM generators supplying signals for motor drivers. The original MOTCON6 card was equipped with a Spartan-II 100k Gates FPGA. Most system components are implemented with the ConPro language (about 900 lines of code, implementing 9 concurrent processes), only a few with native VHDL components (about 650 lines), embedded into the ConPro framework, synthesized into VHDL-RTL (about 6100 lines). Table 6 summarizes the synthesis results.

Parameter	Value
number of source code lines (ConPro)	900
number of embedded VHDL lines	650
max. estimated clock frequency	45 MHz
longest path time	22 ns
number of 4-input LUTs	2260
number of adders	27
number of FSMs	11
number of flip-flops	775

Table 6: Summarization of VHDL synthesis results of robot joint controller using Xilinx ISE 9.2 and a Xilinx Spartan II FPGA

3.4 SVF interpreter

The Serial Vector Format (SVF) language is used to provide access to the JTAG in-system programming port of digital logic systems, like FPGAs. A fully integrated SVF interpreter with JTAG Test Access Point machine (TAP) was implemented entirely in digital logic. SVF commands can be sent over a serial link. The main SVF interpreter is im-

plemented with a protocol stack process like the one shown in section 3.2. Different scheduling parameter settings applied to this protocol stack process (about 800 lines ConPro code) shown in table 7 demonstrate the ability to reduce both digital logic and execution time/latency. Most reduction of control path states is achieved with the basicblock scheduler (about 37%), combined with the reference stack scheduler the highest reduction results (about 60%).

Block Parameter	Control Path States
default	407
schedule=custom (basicblock)	296
schedule=custom (refstack)	349
schedule=custom (refstack, basicblock)	253

Table 7: Synthesis results of the SVF interpreter process with different scheduling parameters.

4. CONCLUSIONS

The ConPro high level synthesis compiler is able to synthesize RTL from an imperative multi-process programming approach, using state machines and combinational logic only. Though this multi-process approach was derived from software programming models familiar to most people coming from the software world, the synthesized RTL-VHDL code can be efficiently fitted to modern digital logic design using FPGAs. The hybrid scheduler and optimizer approach containing the reference stack scheduler on syntax level and the basicblock scheduler on microcode/RTL level enables fast and powerful reduction of digital logic resources and state machine latency. The ConPro compiler is able to synthesize 10000 lines of source code with all optimization and scheduling paths enabled in less than 30 seconds on a modern workstation or PC. *Some remarks about ConPro: the ConPro compiler was entirely programmed in the functional programming language OCaml from the INRIA institute, France. It consists only of about 70000 lines of source code, and is compiled into a bytecode program executed by a virtual machine, providing portability and operating system and machine independency.*

5. ABBREVIATIONS

ALAP: As-Last-As-Possible
ALU: Arithmetic-Logical-Unit
AO: Abstract Object
ASAP: As-Soon-As-Possible
CREW: Concurrent-Read-Exclusive-Write
DDG: Data-Dependency-Graph
EREW: Exclusive-Read-Exclusive-Write
FIFO: First-In-First-Out
FSM: Finite-State-Machine
LHS: Left-Hand-Side of an expression
RAM: Random-Access-Memory
 μ RISC: MicroCode Reduced-Instruction-Set-Computer
RHS: Right-Hand-Side of an expression
RTL: Register-Transfer-Logic
SVF: Serial Vector Format

TU: Time Unit

VLIW: Very-Large-Instruction-Word

6. REFERENCES

- [1] Markus Eich, Stefan Bosse, Felix Grimminger, Dirk Spenneberg, Frank Kirchner
ASGUARD: A Hybrid Legged Wheel Security and SAR-Robot Using Bio-Inspired Locomotion for Rough Terrain
IARP/EURON Workshop on Robotics for Risky Interventions and Environmental Surveillance (IARP/EURON-08), January 7-8, Benicassim, Spain
Online-Proceedings, Benicàssim (Spain), 1 2008
- [2] Dirk Spenneberg, Stefan Bosse, Jens Hilljegerdes, Andreas Strack, Heiko Zschenker
Control of an Bio-Inspired Four-Legged Robot for Exploration of Uneven Terrain.
In Proc. of ASTRA 2006 Workshop, ESA-ESTEC. Noordwijk, NL, 2006.
- [3] David Ku, Giovanni De Micheli
High Level Synthesis of ASICs Under Timing and Synchronization Constraints
Kluwer, 1992
- [4] David Ku, Giovanni De Micheli
HardwareC – A Language for Hardware Design
Technical Report: CSL-TR-90-419 Year of Publication: 1990, Stanford University
- [5] Steven S. Muchnick
Advanced Compiler Design Implementation
Academic Press, 1997
- [6] Richard Sharp
Higher-Level Hardware Synthesis
Springer, 1998
- [7] LIP6, Université Pierre et Marie Curie
Alliance - a free VLSI CAD system
<http://www-asim.lip6.fr/recherche/alliance>
- [8] Thomas Fahringer, Bernhard Scholz
Advanced Symbolic Analysis for Compilers
Springer, LNCS 2628, 1998
- [9] Wen-Mei W. HWU et al.
The superblock: An Effective Technique for VLIW and Superscalar Compilation
Journal of Supercomputing, **7**, 229-248, , 1992