Hardware Design and Synthesis in ForSyDe

Ingo Sander, Alfonso Acosta, and Axel Jantsch

Royal Institute of Technology
Stockholm, Sweden
{ingo,alfonsoa,axel}@kth.se

Hardware Design and Functional Languages
York, 28-29 March, 2009

Outline

1 ForSyDe Background

2 ForSyDe Modelling Framework

3 Shallow Embedded ForSyDe - Modelling Heterogeneous Systems

4 Deep Embedded ForSyDe - Hardware Synthesis, Graphical Output

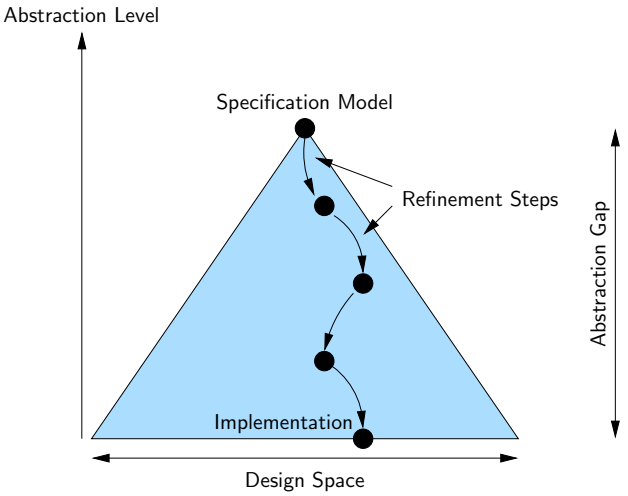
5 Conclusion and Future Work

What is ForSyDe?

ForSyDe (Formal System Design) is a design methodology for systems-on-chip that has been developed with the following objectives.

- System design must start at a high level of abstraction
  - Designer shall focus on functionality
  - Low-level implementation details shall not be an issue at this stage
- Design methodology must give a solid base for the incorporation of formal methods
  - Verification must be a first class citizen from the start
- Abstraction gap between specification and implementation must be bridged by formal refinement technique

Designing at Different Levels of Abstraction



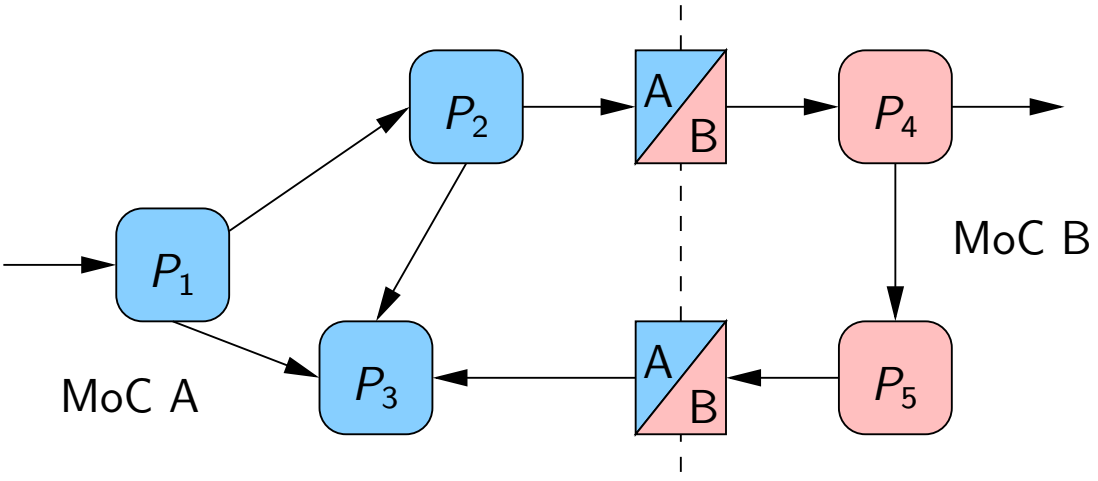
- ForSyDe allows to describe heterogeneous models at different abstraction levels.
- The idea is to 'map' a refined ForSyDe model to an implementation language such as VHDL.

Status of ForSyDe

- ForSyDe allows to model heterogeneous systems and is implemented as domain specific language in Haskell
- The original ForSyDe implementation was developed for the modelling purpose (*shallow-embedded implementation*)
  - Several libraries for different models of computation exist and can be simulated as integrated model
- Recently, a new ForSyDe implementation was developed giving access to the internal structure of the system model (*deep-embedded implementation*)
  - New back-ends have been developed for hardware design, synthesis (VHDL) and graphical representation (GraphML)
- Shallow-embedded and deep-embedded models can be co-simulated giving access to powerful test benches

ForSyDe System Model

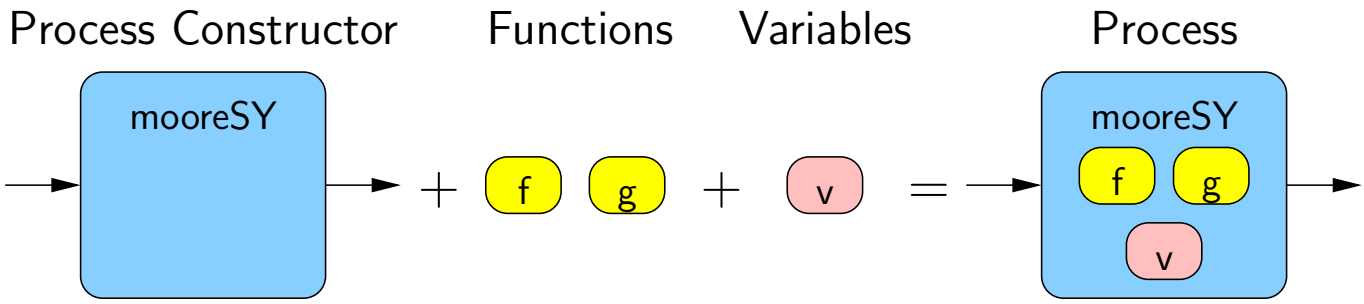
- A system is modelled as hierarchical concurrent process model
- Processes of different models of computation communicate via domain interfaces
  - Supported MoCs: Synchronous, Untimed (Synchronous Data Flow), Continuous Time



ForSyDe Process

A process takes *m* input signals as argument and produces *n* output signals. ForSyDe processes are deterministic.

- A process is always designed by means of a *process constructor*
- The process constructor defines the communication interface of the process
- The process constructor takes side-effect free *functions* and *variables* as arguments and returns a process



Process Constructor Types

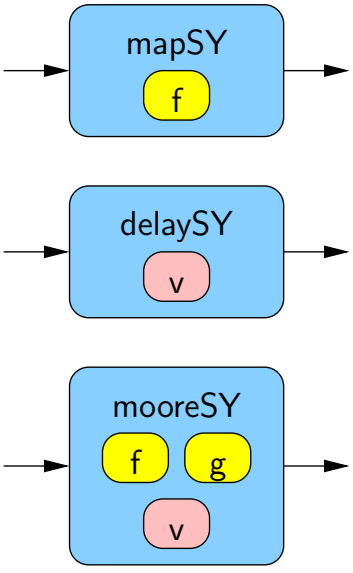
There are three main categories of process constructor

Combinational Process has no internal state

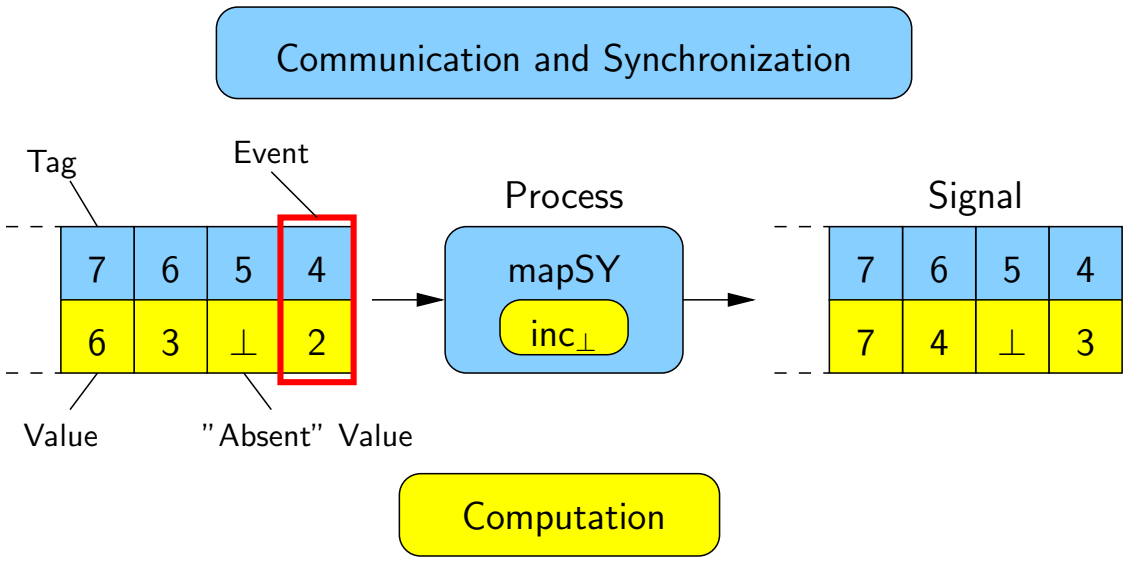
Delay Process delays input

Sequential Processes have an internal state and contain a delay process

These categories exist in all models of computation



Synchronous Process



Synchronous Assumption

The outputs of a system (process) are synchronized with the inputs. The reaction of the system is instantaneous and takes no observable time!

Process Constructor - Benefits

The concept of process constructor

- separates communication from computation
  - process constructor: communication and interface
  - function: computation
- forces the designer to develop a structured formal model that allows for formal analysis
  - transformational refinement
  - implementation mapping
  - formal verification

Modelling Heterogeneous Systems

The Shallow Embedded Implementation of ForSyDe

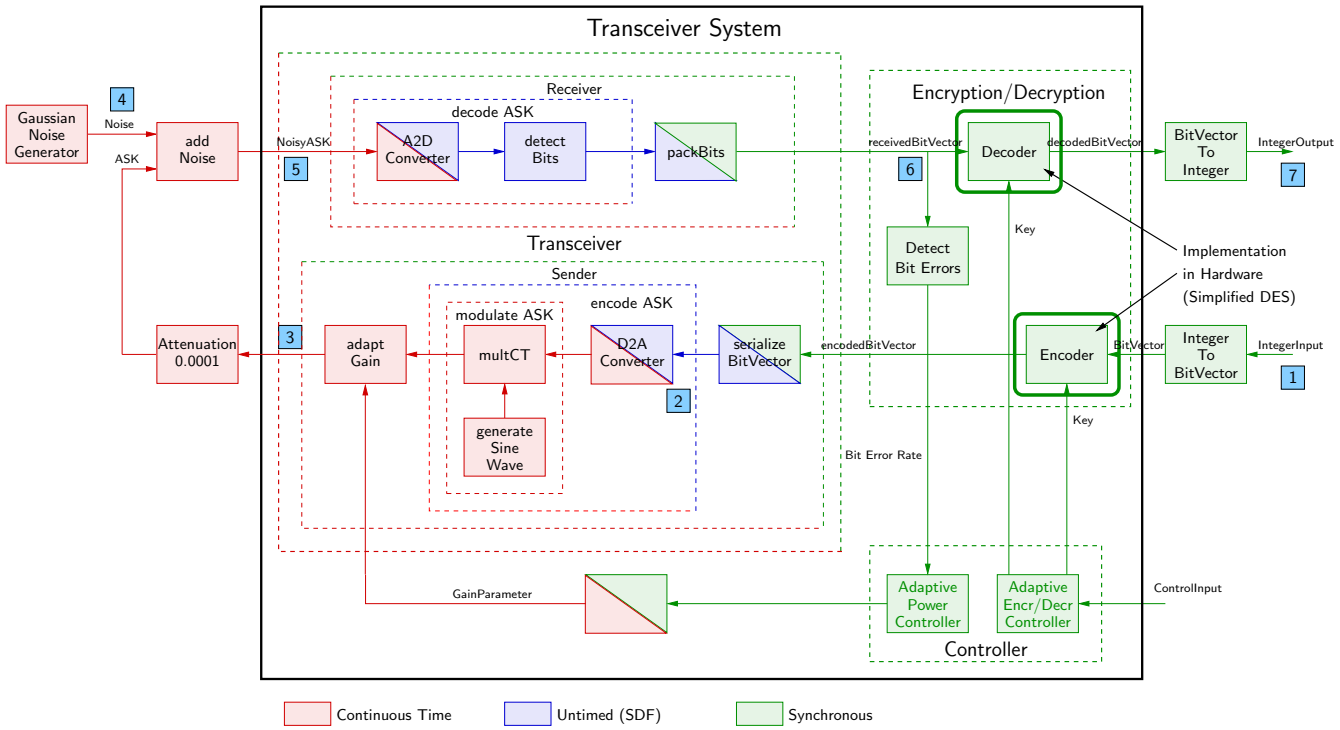
ForSyDe has initially been mainly developed for the modelling purpose.

- Signals have been modelled as streams of data.

```
data Signal a = NullS | Signal a
```

- Pros
  - Rapid modelling of heterogeneous systems
  - No need for advanced or non-standard features of Haskell
  - Easy to include new models of computation
- Cons
  - Restriction to simulation only, since there is no access to the abstract syntax tree

Tutorial Example

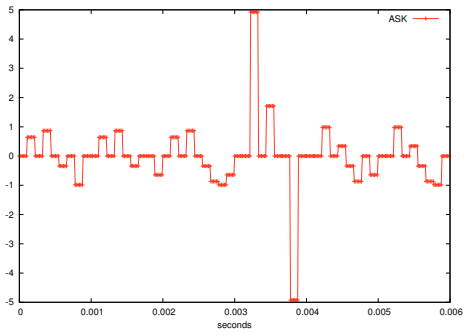


Simulating a Heterogeneous System

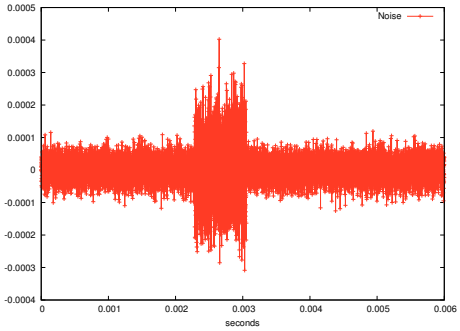
1 Synchronous Input Signal

`in = {0,1,2,3,4,5}`

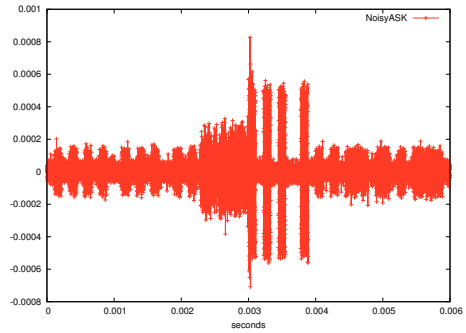
3 Transceiver Output



4 Gaussian Noise



5 Noisy Transceiver Input



7 Synchronous Output Signal

`out = {0,1,10,3,4,5}`

Simplified DES Algorithm

- Both the encryption and decryption units are implemented with the simplified DES algorithm
- The simplified DES algorithm has similar properties and structure as DES, but is developed for educational purposes.

Simplified DES Algorithm

$$ciphertext = (IP^{-1} \circ f_{K_2} \circ SW \circ f_{K_1} \circ IP)\ plaintext$$
$$plaintext = (IP^{-1} \circ f_{K_1} \circ SW \circ f_{K_2} \circ IP)\ ciphertext$$

IP initial permutation

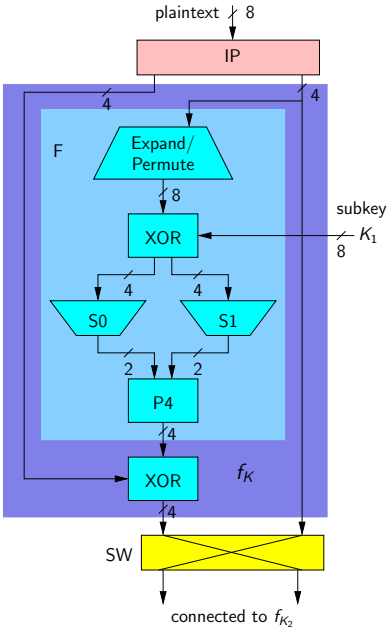
IP^{-1} inverse of initial permutation

SW switches halves of data

f_{K_i} permutation and substitution of data using the subkey i. Two 8-bit subkeys are generated from an initial 10-bit key.

Simplified DES - The f_K Functionality

Process networks are modelled as set of equations.




```
...
import Data.Param.FSVec
import qualified Data.Param.FSVec as FS

f :: FSVec D8 Bit -> FSVec D4 Bit -> FSVec D4 Bit
f subkey nibble
    = p4 (out_S0 FS.++ out_S1)
      where out_S0  = s0matrix out_xor
            out_S1  = s1matrix out_xor
            out_xor = zipxor subkey out_ep
            out_ep  = exp_perm nibble

f_k :: FSVec D8 Bit -> FSVec D8 Bit
    -> FSVec D8 Bit
f_k subkey input
   = outLeft FS.++ outRight
     where
       outLeft = FS.zipWith xor inpLeft fOut
       outRight = inpRight
       fOut = f subkey inpRight
       inpLeft = FS.take d4 input
       inpRight = FS.drop d4 input
```


Fixed-Sized Vectors

For hardware design vectors of fixed size are of crucial importance for both modelling and an efficient design implementation. ForSyDe offers the data type FSVec[1] that allows to specify vectors of constant size, which can also be synthesized to hardware.

```
...
import Data.Param.FSVec
import qualified Data.Param.FSVec as FS

splitBlock :: FSVec D8 Bit -> (FSVec D4 Bit, FSVec D4 Bit)
splitBlock block = (FS.take d4 block, FS.drop d4 block)
```

Fixed Size Vectors are heavily used during the tutorial example.

[1] The implementation was done by Alfonso Acosta

Deep-Embedded ForSyDe

In order to allow to give structural information to a ForSyDe model, recently a deep-embedded version of ForSyDe has been developed [2]. The main objectives have been

- to allow for efficient hardware synthesis of synchronous ForSyDe models
- to be able to simulate synchronous ForSyDe models
- to be able to integrate further back-ends for analysis and transformation
- to be able to combine and co-simulate shallow-embedded and deep-embedded ForSyDe models

[2] The implementation was done by Alfonso Acosta

Why an Embedded Compiler?

- **Feasibility.** Does require less time than development of traditional compiler
- **Saves unnecessary effort.** The goal is to translate the system structure, not any arbitrary Haskell program
- **Previous success.** Successful embedded compiler has been developed for Lava
- **Maintainable.** The compiler is packed with ForSyDe's Library
- **Independent of third-party tools.** No risk of getting outdated due to external design changes

Process Function

- ForSyDe uses Template Haskell to be able to extract the Abstract Syntax Tree.
- Each function that is argument to a process constructor needs to be declared using Template Haskell.

```
p4Fun :: ProcFun (FSVec D2 Bit -> FSVec D2 Bit -> FSVec D4 Bit)
p4Fun = $(newProcFun
            [d| p4 :: FSVec D2 Bit -> FSVec D2 Bit -> FSVec D4 Bit
                p4 outS0 outS1 = outS0!d1 +> outS1!d1 +>
                                 outS1!d0 +> outS0!d0 +> empty |])
```

Use of Template Haskell

- The declaration `[d| ... |]` is used to give a definition of a function so that its abstract syntax tree is accessible at compile-time.
- `$(..)` means "evaluate at compile time". Here a `newProcFun` using the declaration in `[d| ... |]` is evaluated at compile-time and gives access to the abstract syntax tree.

System Function and Definition

Processes are created using process constructors and process functions as arguments. An additional argument is the process identifier that can be used by the compiler back-end.

```
p4Proc :: Signal (FSVec D2 Bit) -> Signal (FSVec D2 Bit)
       -> Signal (FSVec D4 Bit)
p4Proc = zipWithSY "p4Proc" p4Fun
```

A system function describes a system. Additional parameters are system identifier and input and output ports.

```
p4Sys :: SysDef (Signal (FSVec D2 Bit) -> Signal (FSVec D2 Bit)
      -> Signal (FSVec D4 Bit))
p4Sys = newSysDef p4Proc "p4" ["S0", "S1"] ["out"]
```

Composing a Larger System

Systems can be used as components to create larger systems.

```
fProc :: Signal (FSVec D4 Bit) -> Signal (FSVec D8 Bit)
      -> Signal (FSVec D4 Bit)
fProc nibble subkey = out
    where
       out = (instantiate "p4Sys" p4Sys) s0 s1
       s0 = (instantiate "outputS0Sys" outputS0Sys) rS0 cS0
       s1 = (instantiate "outputS1Sys" outputS1Sys) rS1 cS1
       rS0 = (instantiate "rowS0" rowS0Sys) xorsubkey
       rS1 = (instantiate "rowS1" rowS1Sys) xorsubkey
       cS0 = (instantiate "colS0" colS0Sys) xorsubkey
       cS1 = (instantiate "colS1" colS1Sys) xorsubkey
       xorsubkey = (instantiate "xorsubkeySys" xorSubkeySys) subkey expperm
       expperm = (instantiate "expPermSys" expPermSys) nibble

fSys :: SysDef (Signal (FSVec D4 Bit) -> Signal (FSVec D8 Bit)
                -> Signal (FSVec D4 Bit))
fSys = newSysDef fProc "fSys" ["nibble", "subkey"] ["out"]
```


Simulation

Deep-embedded models can be simulated using the command `simulate`. The system process `encryptSys` encrypts a plain text and `decryptSys` is the corresponding decryption.

```
key1 = [H +> L +> H +> L +> L +> L +> L +> L +> H +> L +> empty]
subkey_1 = fst $ simulate subkeysSys key1
subkey_2 = snd $ simulate subkeysSys key1
enc = simulate encryptSys subkey_1 subkey_2 plain
dec = simulate decryptSys subkey_1 subkey_2 enc

> plain
[<L,L,H,L,L,H,L,L>]
> enc
[<H,H,H,L,L,L,H,H>]
> dec
[<L,L,H,L,L,H,L,L>]
```

Deep-embedded and shallow-embedded models can be co-simulated in the 'shallow-embedded world'.

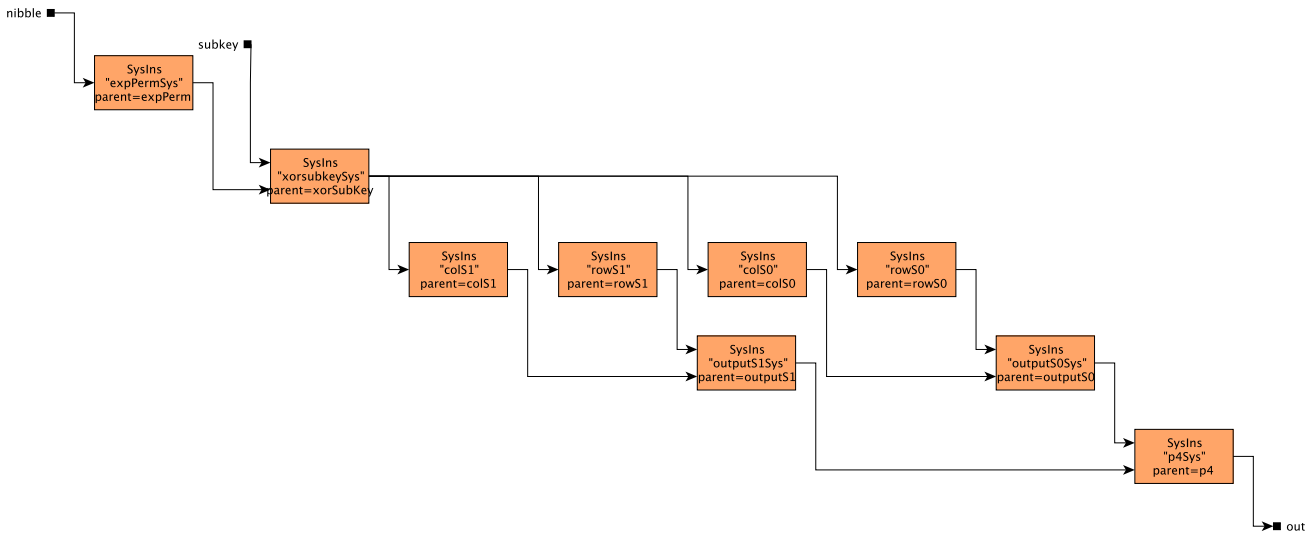
```
combSim shallowSig = (shallowProcess . signal . simulate DeepSys
                      . fromSignal) shallowSig
```

Graphical Output

It is possible to obtain a graphical representation using ForSyDe's GraphML back-end.

```
> writeGraphMLOps defaultGraphMLOps{yFilesMarkup=True} fSys
```

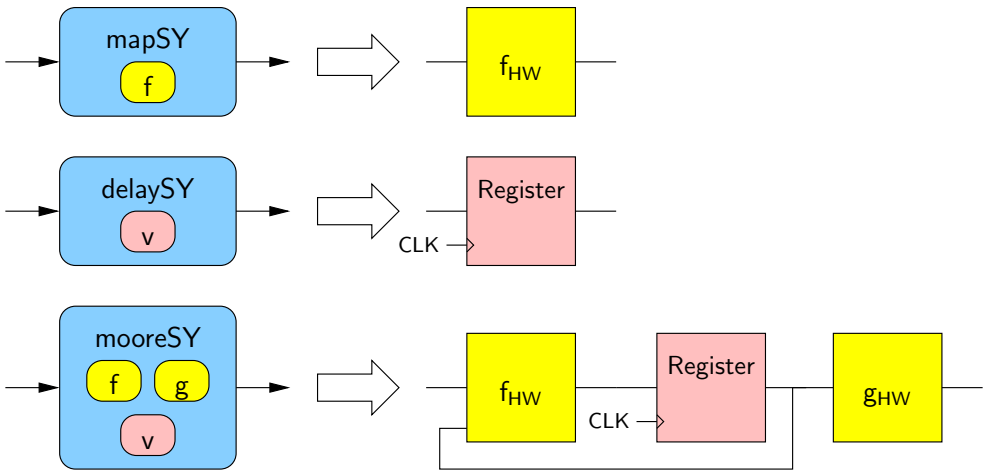
The diagram can then be viewed using the editor *yEd* from *yWorks*:



Hardware Synthesis - Processes

The main ideas for hardware synthesis are summarized in this and the following slide:

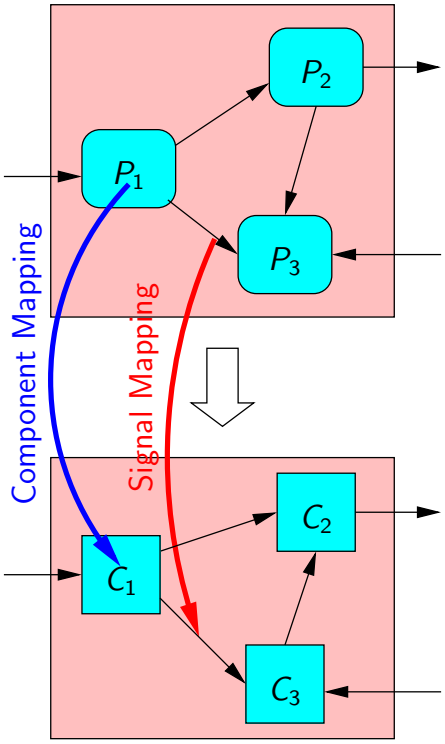
- Each process constructor has a corresponding VHDL-template
- ForSyDe functions and data types are translated to VHDL



Hardware Synthesis - Process Networks

ForSyDe process networks are mapped to networks of components

- ForSyDe signals are mapped to VHDL signals
- ForSyDe processes are mapped to VHDL components

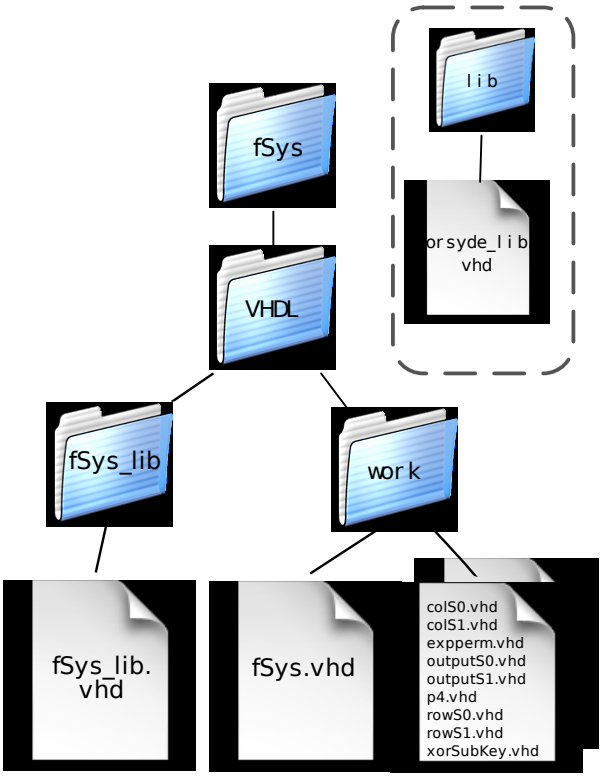


VHDL Synthesis

ForSyDe's embedded compiler is able to translate system definitions to VHDL. This is done through the `writeVHDL` function.

```
> writeVHDL fSys
```

The function generates a tree structure containing the VHDL code of all included subsystems.



FPGA Synthesis using Altera Quartus

ForSyDe allows to directly create a downloadable file for a specific Altera FPGA circuit, given that the Altera tools are installed.

Synthesis of `encryptSys`

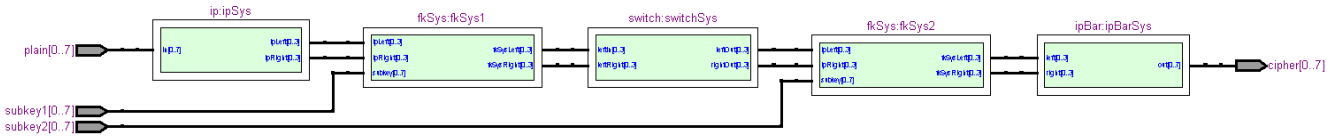
```
compileQuartus_encryptSys :: IO ()
compileQuartus_encryptSys = writeVHDLOps vhdlOps encryptSys
 where vhdlOps = defaultVHDLOps{execQuartus=Just quartusOps}
       quartusOps = QuartusOps{action=FullCompilation,
                               fMax=Just 50, -- in MHz
                               fpgaFamiliyDevice=Just ("CycloneII",
                                                       Just "EP2C35F672C6"),
                               -- Possibility for Pin Assignments
                               pinAssigs=[]
                              }
```

Quartus generates then reports, which inform about size and speed of the design (`encryptSys` needs 36 logic elements, 32 pins, and the worst case delay is 18.836 ns)

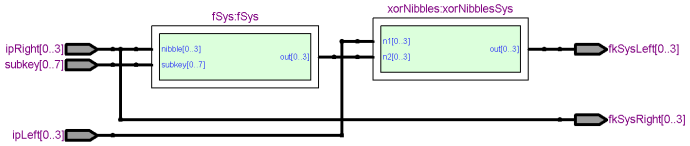
FPGA Synthesis Using Altera Quartus

Output from RTL Viewer

- Implementation of `encryptSys`



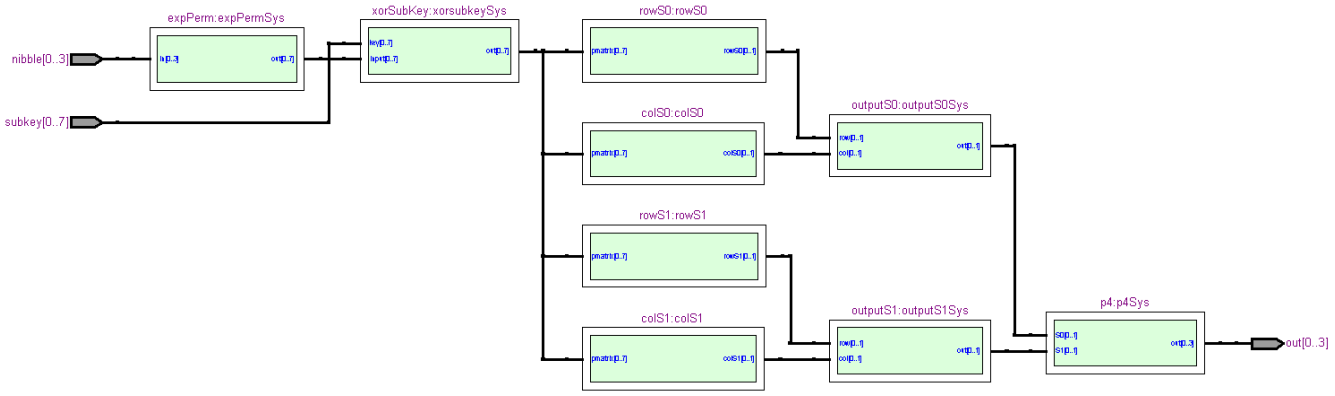
- Implementation of `fkSys`



FPGA Synthesis Using Altera Quartus

Output from RTL Viewer

- Implementation of `fSys`



Conclusion

- Complex heterogeneous systems can be modelled and simulated in ForSyDe
- Deep-embedded ForSyDe implementation allows to extract structural information of a ForSyDe model
  - used for hardware synthesis
    - Connection to Altera Quartus (FPGA-Synthesis)
    - Connection to Modelsim (VHDL-Simulator)
  - used for graphical output (GraphML)
- So far deep-embedded presentation is only possible for synchronous models
- Not all desired constructs can yet be synthesized
- Template Haskell allows to extract the structural information, but implies a less intuitive syntax

Future Work

- Extension of VHDL-backend
  - More synthesizable constructs
    - At present only `Int8`, `Int16`, and `Int32` are supported
    - Higher-order functions on fixed-sized vectors cannot be synthesized, such as `FS.zipWith.xor s1 s2`.
  - Better support for pattern matching
  - Introduce optimization capabilities into VHDL-backend
    - So far compiler does only map ForSyDe-model to VHDL
- Extension of deep-embedded implementation to additional models of computation
  - So far only synchronous model is supported
- Adaptation to new problem areas
  - Software synthesis
  - Formal verification
  - Performance analysis
  - Design transformation

Thanks for your attention!

More information on ForSyDe

`http://www.ict.kth.se/forsyde/`