Definition of transactional abstraction levels needed for a precise architecture evaluation in the Systems-on-Chip’s Design Flow

Florence Maraninchi, Jérôme Cornet and Laurent Maillet-Contoz

Verimag - CNRS - STMicroelectronics
Outline

1. Context
2. Timing evaluation at high level of abstraction
3. Problem
4. Research
1 Context
- Systems on Chip
- Typical design flow
- Design problems
- Transactional Level Modeling
- Research topics

2 Timing evaluation at high level of abstraction

3 Problem

4 Research
What are Systems on Chip? (SoC)

- Chips integrating all necessary electronic circuits for a "system"
- Applications: Cell phones, DVD, Set-top boxes, MP3 players, Automotive...
- Key characteristics
  - High level of integration
  - Software and hardware parts
  - Application Specific Integrated Circuits
Typical design flow

- Various abstraction levels
- Main entry point: Register Transfer Level (RTL)
- Tools to reach automatically lower levels

VHDL, Verilog → RTL → Synthesis, Optimization → Netlist → Gate level → Place & Route → Layout → Masks, Bitstream
Register Transfer Level

- **Content**
  - Components connected by wires
  - Logical values (0, 1, ...) carried by the wires
  - Synchronous circuits description

- **Time and bit accurate**

- **Available at the end of the design cycle**
Common problems

- Increasingly complex circuits: "design gap"
- Pressure on Time to market
- Skyrocketing costs
  - Very expensive to fix hardware bugs (masks cost)
  - Bugged chips already delivered?
- Two big classes of design problems:
  - Physical problems
  - System-level problems
Common problems

- Increasingly complex circuits: "design gap"
- Pressure on Time to market
- Skyrocketing costs
  - Very expensive to fix hardware bugs (masks cost)
  - Bugged chips already delivered?
- Two big classes of design problems:
  - Physical problems
  - System-level problems
System-level problems

- Slow RTL simulation
  ex: encoding/decoding one MPEG-4 frame takes 1 hour to simulate
- Late availability of RTL
- Late writing of software parts
- Late discovery of integration bugs
Transactional Level Modeling (TLM)

- New level of abstraction
  - Written before RTL
  - Fast simulation speed
    previous MPEG-4 ex: 3 seconds to simulate (× 1200)
  - Lightweight modeling effort
Transactional Level Modeling (TLM)

- Allows:
  - Early embedded software validation
  - Virtual prototyping

- Written in SystemC

- Standardized by Open SystemC Initiative (OSCI)
TLM: idea

Bus communications abstraction
TLM: description

- **Contract** between embedded software and hardware

- **Components**
  - Accurate modeling of register banks, system synchronizations

- **Communications**
  - Transactions
    - Logical operations on buses: read, write, ...
    - Master or Slave ports with assigned addresses
  - Interrupt signals

- **Characteristics**
  - Clockless
  - Bit accurate
TLM: example

Generator 1

Shared memory

Generator 2

Bus 1

Bus 2

Timer
Research topics in the team

- Formal verification of transactional-level platforms
- Automatic generation of tests
- Comparison between RTL and TLM
- Timing evaluation at transactional level
Outline

1 Context

2 Timing evaluation at high level of abstraction
   - Coarse-grain approaches
   - PV/PVT

3 Problem

4 Research
Coarse-grain approaches (1/2)

- Rough abstraction of architecture and timings
  - "Labels" on components
  - Problem: composition law between labels?

References: [RSL04], [CKT+03], [JRE04], [TCN00], [TC94]
Rough abstraction of architecture and timings

- "Labels" on components
- Problem: composition law between labels?

References: [RSL04], [CKT+03], [JRE04], [TCN00], [TC94]
Coarse-grain approaches (1/2)

- Rough abstraction of architecture and timings
  - "Labels" on components
  - Problem: composition law between labels?

References: [RSL04], [CKT+03], [JRE04], [TCN00], [TC94]
Coarse-grain approaches (2/2)

- Actually **dataflow** analysis oriented
- Suited for using theory of performance evaluation
  - Queuing Networks
  - Layered Queuing Networks
  - Stochastic Automata Networks
  - ...
- Component structure not always appropriate for performance analysis
Two abstraction levels inside TLM
- Programmer’s View
- Programmer’s View with Time
PV/PVT: description

**PV level**
- Designed for:
  - Embedded software validation
  - Platform integration
- Time has no meaning at this level

**PVT level**
- Designed for:
  - Architecture evaluation
  - Timing-sensitive parts of embedded software validation
- Contains: PV + microarchitecture model
## PV/PVT: description

### PV level
- Designed for:
  - Embedded software validation
  - Platform integration
- Time has no meaning at this level

### PVT level
- Designed for:
  - Architecture evaluation
  - Timing-sensitive parts of embedded software validation
- Contains: PV + microarchitecture model
Timing evaluation at high level of abstraction

PV/PVT: microarchitecture model

Model’s ingredients

1. Granularity
2. Microarchitecture features (fifos, pipeline...)
3. Timings
Timing evaluation at high level of abstraction

PV/PVT: microarchitecture model

Model’s ingredients

1. Granularity
2. Microarchitecture features (fifos, pipeline...)
3. Timings
PV/PVT: microarchitecture model

Model's ingredients

1. Granularity
2. Microarchitecture features (fifos, pipeline...)
3. Timings
PV/PVT : constraints (1/2)
Timing evaluation at high level of abstraction

PV/PVT: constraints (1/2)

Same functional behavior

TLM PV

TLM PVT

RTL

Jérôme Cornet

Synchron 2005
PV/PVT : constraints (1/2)

TLM PV

Same functional behavior

TLM PVT

Same functional behavior

RTL
PV/PVT : constraints (1/2)

- TLM PV
- TLM PVT
- RTL

Same functional behavior
Same timed behavior
Same functional behavior

Timing evaluation at high level of abstraction PV/PVT

Jérôme Cornet Synchro 2005
Traces example:

PV

Read
Write

PVT

Read
Write
Read
Write
Read
Write
Read
Write

RTL Transactions

Read
Write
Read
Write
Read
Write
Read
Write

PV reuse in PVT
Timing evaluation at high level of abstraction

PV/PVT: constraints (2/2)

- **Traces example:**

  ![Traces Example Diagram]

  - **PV**:
    - Read
    - Write

  - **PVT**:
    - Read
    - Write
    - Read
    - Write
    - Read
    - Write
    - Read
    - Write

  - **RTL Transactions**:
    - Read
    - Write
    - Read
    - Write
    - Read
    - Write
    - Read
    - Write

  - **RTL**:
    - 1
    - 0
    - 1
    - 0
    - 1
    - 0
    - 1
    - 0
    - 1
    - 0

- **PV reuse in PVT**

Jérôme Cornet  Synchron 2005
Outline

1. Context

2. Timing evaluation at high level of abstraction

3. Problem
   - Description
   - Example

4. Research
Problem description (1/2)

Problem

- Given:
  - PV reference
  - Microarchitecture, temporal laws

- Build: PVT
  - With same functionality as PV
  - With same temporal behavior as RTL microarchitecture

- No automatic build of PVT for now
Approach: for each component,

- Build $PVT = PV \oplus T$
- $PV$: unmodified PV model of the component
- $T$: standalone microarchitecture model
- $\oplus$: "glue", synchronization between $PV$ and $T$, some kind of weaving?

Hopes:

- Maintain $PV$’s functionality by construction
- Consistency of behavior between $PV$ and $T$
- Guarantee correct behavior of composed components
Example (PV)

Generator 1
PV

Shared memory
PV

Generator 2
PV

Timer
PV

Bus 1

Bus 2
Example (PVT)
Generator 1

PVT
PVT module’s structure

Generator 1
PV
PVT module’s structure

Generator 1
PV

Generator 1
T
PVT module’s structure

Generator 1
PV

Generator 1
T

Jérôme Cornet
Synchron 2005
Outline

1. Context
2. Timing evaluation at high level of abstraction
3. Problem
4. Research
   - Summary
   - Model
Formalizing the approach

- Building an abstract model that captures $PV$, $\oplus$, $T$, $PVT$
- Proving properties about:
  - Functional consistency between $PV$ and $PVT$
  - Logical consistency between $PV$ and $T$
- Separation of elements added to $PV$
  - $\oplus$ and $T$
  - Basic building blocks within $T$
- Automation?
Model
Conclusion

PV/PVT: a good example of
- Separation of concerns: functional/timed
- Fine-grain architecture modeling

Research
- Guarantee properties about the process "by construction"
- Rules about writing PV and T
- Provide some kind of automation in the long term

To do (also)
- Complex PV/PVT behaviors
- How to model components with functional time? (timer, uart, etc.)
Generalities

- Component architecture description
- Behavior’s description by automata inside components
- Two levels of parallelism:
  - Between components
  - Inside components ("threads")
Generalities

- Component architecture description
- Behavior’s description by automata inside components
- Two levels of parallelism:
  - Between components
  - Inside components ("threads")
Communications between components

- **Master (initiator) actions**
  - `!write(port, @, data)`
  - `!read(port, @)` returns a data
  - `!sync(data)` (for communications by interruptions)
  - Blocked until execution of `return` action

- **Slave (target) actions**
  - `?write(port, @, data)`
  - `?read(port, @)`
  - `?sync(data)` (for communications by interruptions)
  - `return`
Component with master port: Generator PV

d := !read(port1, a)
[a < 0x2000]
write(port1, a, 42)
a++

Component with master port: Generator PV

d := !read(port1, a)
[a < 0x2000]
write(port1, a, 42)
a++
Component with slave port: Memory PV

\[
\text{?read(port, @)}
\]
\[\text{return}\]

\[
\text{?write(port, @, data)}
\]
\[\text{return}\]

port
Model elements for PVT

- Previous elements

- Distinguishing different transactions size with different actions for convenience
  - `littlewrite(port, @, data)` (both ? and !)
  - `littleread(port, @)` (both ? and !)

- Action for time elapse: `wait(time)`
Example

- Generator PVT

Generator
PVT
Example

- Generator PVT

![Diagram of Generator PVT with subcomponents PV and T]
Generator T

Example (Generator T)
Example (Generator T)

- Generator T

```
?read(port1, @)
!read(port2, @)
i := 0
[i < 10]
!littlewrite(port3, @, data)
i++
!write(port2, @, data)
i := 0
[i ≥ 10]
return

?write(port1, @, data)
!write(port2, @, data)
i := 0
[i < 10]
!littlewrite(port3, @, data)
i++
[i ≥ 10]
return
```
Example (Generator T)

```
?read(port1, @)
!read(port2, @)
i := 0
[i < 10]
    !littlewrite(port3, @, data)
i++
!write(port2, @, data)
i := 0
[i < 10]
    !littlewrite(port3, @, data)
i++
[i ≥ 10]
    return
!write(port2, @, data)
i := 0
[i < 10]
    !littlewrite(port3, @, data)
i++
[i ≥ 10]
    return
```

Jérôme Cornet
Synchon 2005
Example (Generator T)

\[
\begin{align*}
\text{read}(\text{port}1, @) & \quad \text{if } [i \geq 10] \quad \text{return} \\
\text{read}(\text{port}2, @) & \quad \text{if } [i < 10] \\
\text{write}(\text{port}1, @, \text{data}) & \quad \text{if } [i \geq 10] \\
\text{write}(\text{port}2, @, \text{data}) & \quad \text{if } [i < 10] \\
\end{align*}
\]

\[i := 0\]

\[
\begin{align*}
\text{read}(\text{port}1, @) & \quad \text{if } [i \geq 10] \quad \text{return} \\
\text{read}(\text{port}2, @) & \quad \text{if } [i < 10] \\
\text{write}(\text{port}1, @, \text{data}) & \quad \text{if } [i \geq 10] \\
\text{write}(\text{port}2, @, \text{data}) & \quad \text{if } [i < 10] \\
\end{align*}
\]

\[i := 0\]
Memory PVT

Example

[Diagram showing Memory PV and Memory T]
Example (Memory T)
Communications inside components

- Shared variables
- Event actions
  - \( \uparrow e \) (immediate notification on \( e \))
  - \( \downarrow e \) (wait for \( e \))

\[\begin{align*}
\text{write}(\text{port1}, @, \text{data}) & \quad [\@ = 0x04 \land \text{data} = 1] \\
\text{read}(\text{port1}) & \quad \text{return} \\
\text{write}(\text{port2}, a, d) & \quad [a \geq 0x2000] \\
\text{read}(\text{port2}, a) & \quad \text{return} \\
\end{align*}\]
Samarjit Chakraborty, Simon Künzli, Lothar Thiele, Andreas Herkersdorf, and Patricia Sagmeister.
Performance evaluation of network processor architectures: combining simulation with analytical estimation.

Marek Jersak, Kai Richter, and Rolf Ernst.
Performance analysis for complex embedded applications.

Gopal Raghavan, Ari Salomaki, and Raimondas Lencevicius.
Model based estimation and verification of mobile device performance.
K. Tindell and J. Clark.
Holistic schedulability analysis for distributed hard real-time systems.

Lothar Thiele, Smarjit Chakraborty, and Martin Naedele.