Modélisation Transactionnelle des Systèmes sur Puces en SystemC
Ensimag 3A — filière SLE
Grenoble-INP

Usage of TLM Platforms

Matthieu Moy
(transparents originaux : Jérôme Cornet)

Matthieu.Moy@imag.fr

2015-2016

Planning approximatif des séances

1. Introduction : les systèmes sur puce
2. Introduction : modélisation au niveau transactionnel (TLM)
3. Introduction au C++
4. Présentation de SystemC, éléments de base
5. Communications haut-niveau en SystemC
6. Modélisation TLM en SystemC
7. TP1 : Première plateforme SystemC/TLM
8. Utilisations des plateformes TLM
9. TP2 (1/2) : Utilisation de modules existants (affichage)
10. TP2 (2/2) : Utilisation de modules existants (affichage)
11. Notions Avancé en SystemC/TLM
12. TP3 (1/3) : Intégration du logiciel embarqué
13. TP3 (2/3) : Intégration du logiciel embarqué
14. TP3 (3/3) : Intégration du logiciel embarqué
15. 05/01: Intervenant extérieur : Jérôme Cornet
16. Perspectives et conclusion

Grenoble INP ensimag

Outline

Sommaire

Benefits of TLM (compared to RTL)

- Fast (100X to 10000X acceleration compared to RTL)
- Lightweight modeling effort
- Same functionality (bit-accurate)

But ... !

- Drawbacks of TLM:
  - “Less precise”
  - Not synthesisable ($\Rightarrow$ cannot replace RTL)
  - So, what can this be used for?!?

But ... !

- Drawbacks of TLM:
  - “Less precise”
  - Not synthesisable ($\Rightarrow$ cannot replace RTL)
  - So, what can this be used for?!?
- Usage of TLM platforms:
  - Software development, software debugging,
  - Hardware verification,
  - HW/SW partitioning, architecture exploration.

But ... !

- Drawbacks of TLM:
  - “Less precise”
  - Not synthesisable ($\Rightarrow$ cannot replace RTL)
  - So, what can this be used for?!?
- Usage of TLM platforms:
  - Software development, software debugging,
  - Hardware verification,
    $\Rightarrow$ Needs functional accuracy
  - HW/SW partitioning, architecture exploration.
    $\Rightarrow$ Needs non-functional aspects (timing, energy, ...)

Sommaire

Verification of an RTL IP

- Testing an IP
  - Run a test pattern on the RTL IP

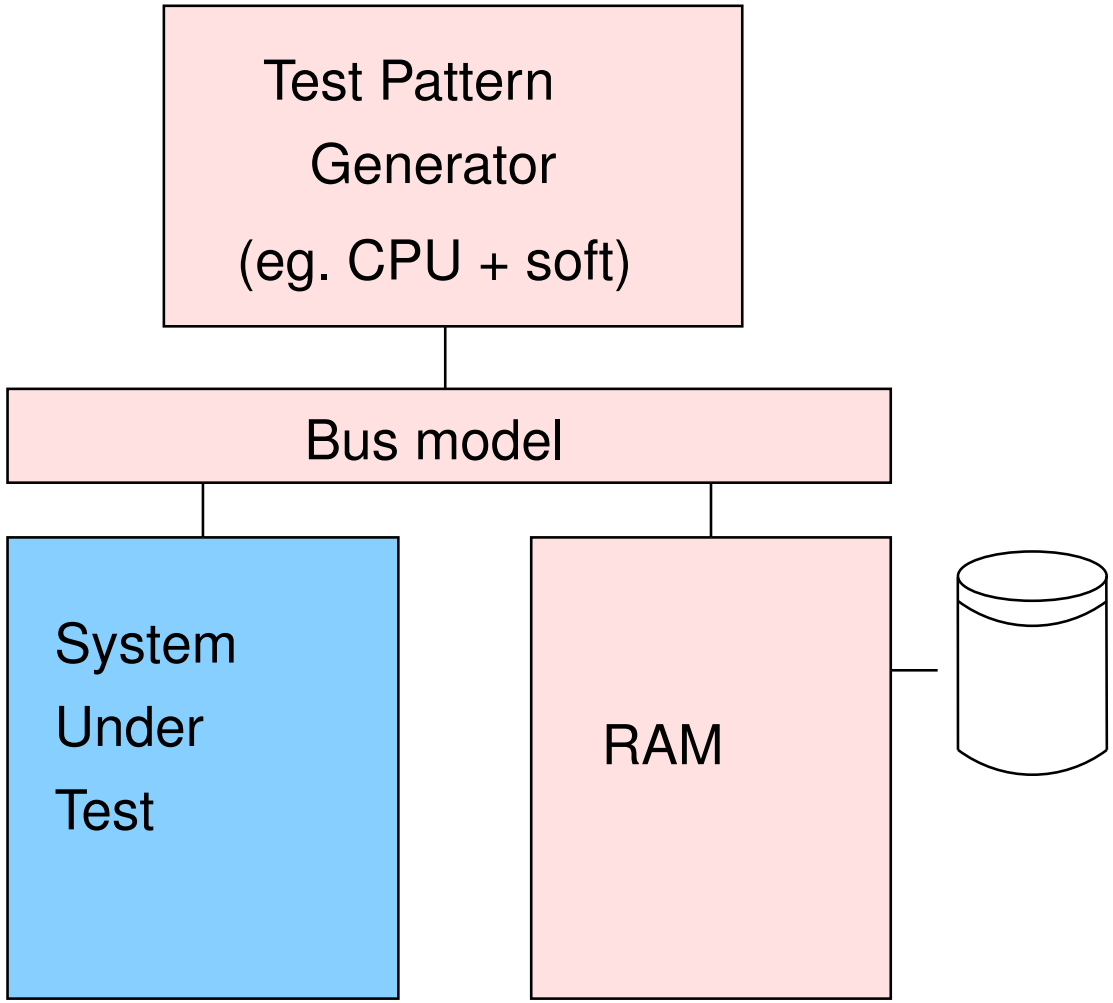
Verification of an RTL IP

- Testing an IP
  - Run a test pattern on the RTL IP
  - Run the same test pattern on the TLM IP
  - Compare

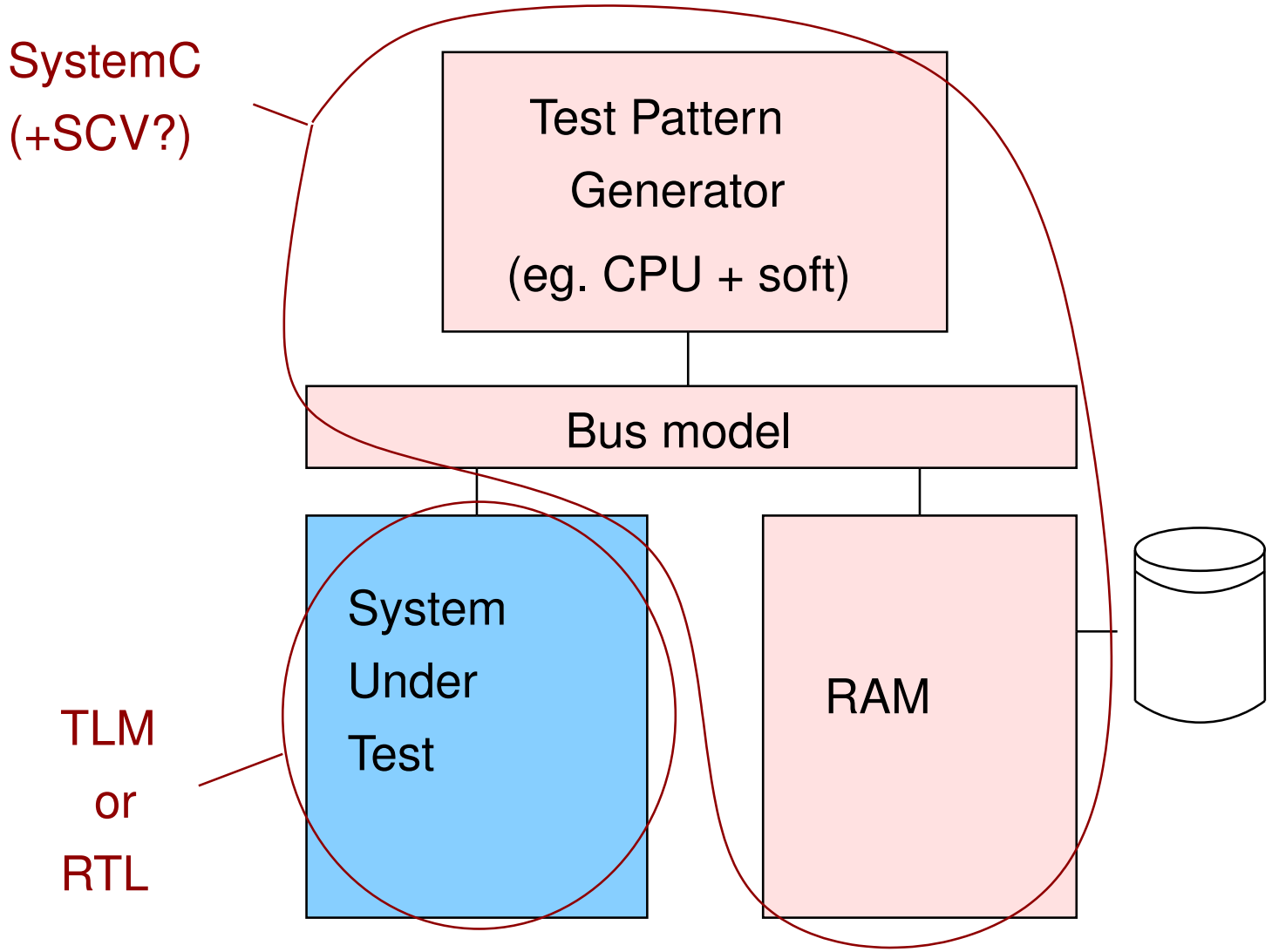
Verification of an RTL IP

- Testing an IP
  - Run a test pattern on the RTL IP
  - Run the same test pattern on the TLM IP
  - Compare
- What is a test-case
  - Often, 1 test-case = 1 piece of software (triggers read and write on the IP)

Typical Test Environment for IPs



Typical Test Environment for IPs



TLM and Verification

- SystemC/TLM provides the tool to
  - build the test environment
  - build reference models for IPs and platforms
  - generate test-patterns (Using the SCV library)
- Benefits over specific solutions
  - Cheap
  - The same language is used by various people

Sommaire

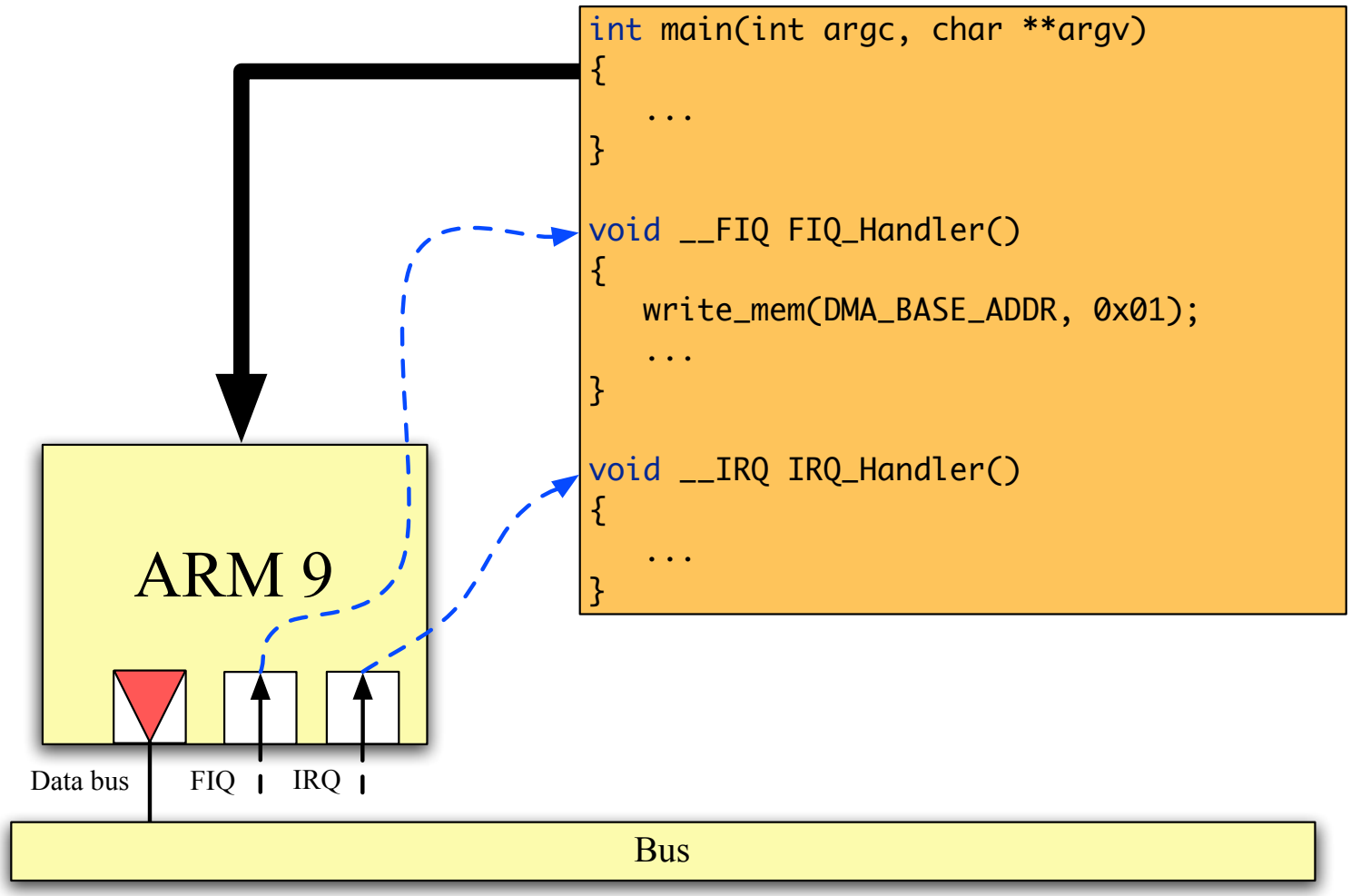
Reminder: principles of TLM

- Model what the software needs, and only that

Reminder: principles of TLM

- Model what the software needs, and only that
- Model
  - Behavior
  - Address map
  - Architecture
- Abstract away
  - Micro-architecture
  - Details of protocols

Interface of a CPU (= Low level API for Software)



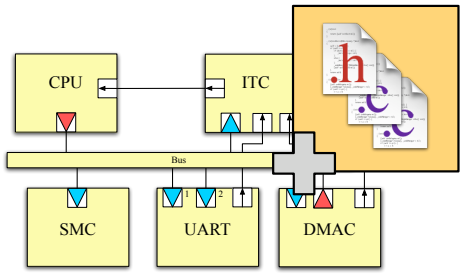
2 ways to Integrate Software

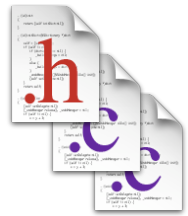
- Instruction Set Simulator (ISS)
  - Compile the software for the target CPU
  - Load and interpret the binary
- Native Wrapper
  - Compile the embedded software for the host CPU
  - Link it against the platform
  - Consider it as a SystemC process

Sommaire de cette section

Native Wrapper: Exemple

Plateforme TLM





Code C

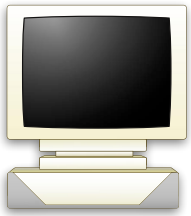
gcc

armcc

Exécutable run.x SUN

Exécutable ARM ELF

Simulation Station SUN



Exécution SoC avec CPU ARM

Grenoble INP ensimag

Problems to be Solved with Native Wrappers

- Integration in the TLM platform?
  - Several solutions: shared/static libraries, techniques from virtual-machines...

Problems to be Solved with Native Wrappers

- Integration in the TLM platform?
  - Several solutions: shared/static libraries, techniques from virtual-machines...
- Transactions generated by CPU?

Problems to be Solved with Native Wrappers

- Integration in the TLM platform?
  - Several solutions: shared/static libraries, techniques from virtual-machines...
- Transactions generated by CPU?
  - Memory access in embedded SW
  - Others... (see later)

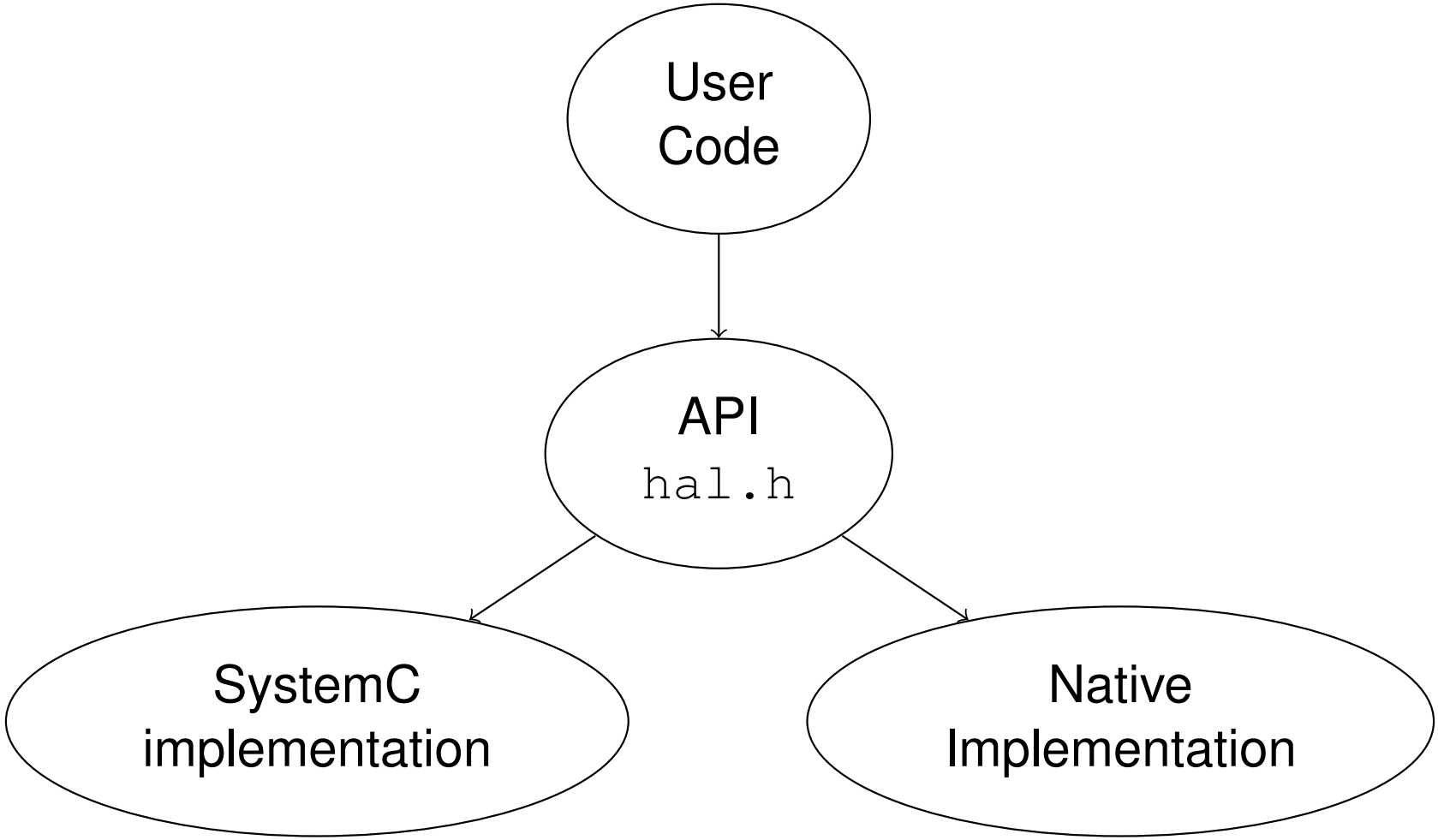
Problems to be Solved with Native Wrappers

- Integration in the TLM platform?
  - Several solutions: shared/static libraries, techniques from virtual-machines...
- Transactions generated by CPU?
  - Memory access in embedded SW
  - Others... (see later)
- Interrupt management

Problems to be Solved with Native Wrappers

- Integration in the TLM platform?
  - Several solutions: shared/static libraries, techniques from virtual-machines...
- Transactions generated by CPU?
  - Memory access in embedded SW
  - Others... (see later)
- Interrupt management
  - How to interrupt the execution of embedded software

A solution: One API, multiple implementations



A solution: One API, multiple implementations

- Hardware Abstraction Layer (HAL)
- Usage of the HAL API in the embedded software
  - Constraint imposed to programmers
  - Portable API: defined both for SystemC and real chips.
  - One primitive for each operation to perform that is different for SystemC and the real platform.

A solution: One API, multiple implementations

- Hardware Abstraction Layer (HAL)
- Usage of the HAL API in the embedded software
  - Constraint imposed to programmers
  - Portable API: defined both for SystemC and real chips.
  - One primitive for each operation to perform that is different for SystemC and the real platform.
- First example: access to the bus
  (Memory and other target components)

```
void write_mem(addr_t addr, data_t data);
data_t read_mem(addr_t addr);
```

- Force user to use `write_mem` and `read_mem` instead of direct access.
- Link with the right implementation depending on the execution platform.

A solution: One API, multiple implementations

- Hardware Abstraction Layer (HAL)
- Usage of the HAL API in the embedded software
  - Constraint imposed to programmers
  - Portable API: defined both for SystemC and real chips.
  - One primitive for each operation to perform that is different for SystemC and the real platform.
- First example: access to the bus
  (Memory and other target components)

```
void write_mem(addr_t addr, data_t data);
data_t read_mem(addr_t addr);
```

Question

Will we see all memory accesses?

SystemC implementation (for TLM simulation)

- Bus access

```
void write_mem(addr_t addr, data_t data)
{
    socket.write(addr, data);
}

data_t read_mem(addr_t addr)
{
    data_t data = 0;
    socket.read(addr, data);
    return data;
}
```

Native implementation (for ISS simulation and the final chip)

- Bus accesses

```
void write_mem(addr_t addr, data_t data)
{
    volatile data_t *ptr = addr;
    *ptr = data;
}

data_t read_mem(addr_t addr)
{
    volatile data_t *ptr = addr;
    return *ptr;
}
```

Deuxième problème : les interruptions

- Le problème :

Question

En quoi le code lié aux interruptions est-il différent du reste?

Deuxième problème : les interruptions

- Le problème :
  - Pas de primitive pour gérer les interruptions en C ou C++
  - Dépendant de l'architecture cible (primitives assembleur, ...)
  - Au moins trois façons de gérer les interruptions :
    - ISR = Interrupt Service Routine

      ```
      void irq_handler(void) { ... }
      ```

    - polling:

      ```
      while (!condition) { /* nothing */ }
      ```

    - Attente explicite (instruction assembleur, opération bloquante, ...), via appel système si besoin

Deuxième problème : les interruptions

- Le problème :
  - Pas de primitive pour gérer les interruptions en C ou C++
  - Dépendant de l'architecture cible (primitives assembleur, ...)
  - Au moins trois façons de gérer les interruptions :
    - ISR = Interrupt Service Routine

      ```
      void irq_handler(void) { ... }
      ```

    - polling:

      ```
      while (!condition) { /* nothing */ }
      ```

    - Attente explicite (instruction assembleur, opération bloquante, ...), via appel système si besoin
  - Ça ne va pas marcher « tel quel » :
    - Le polling ferait une boucle infinie sans rendre la main
    - L'attente explicite ne compilera même pas en général

Deuxième problème : les interruptions

- Le problème :
  - Pas de primitive pour gérer les interruptions en C ou C++
  - Dépendant de l'architecture cible (primitives assembleur, ...)
  - Au moins trois façons de gérer les interruptions :
    - ISR = Interrupt Service Routine

      ```
      void irq_handler(void) { ... }
      ```

    - polling:

      ```
      while (!condition) { /* nothing */ }
      ```

    - Attente explicite (instruction assembleur, opération bloquante, ...), via appel système si besoin
  - Ça ne va pas marcher « tel quel » :
    - Le polling ferait une boucle infinie sans rendre la main
    - L'attente explicite ne compilera même pas en général

Question

Mais alors, comment faire ?

Emballage natif : Les interruptions

- Solution possible : dans l'API
  - Une primitive pour « rendre la main, et attendre arbitrairement », `cpu_relax()` :

```
while (!condition) {
  cpu_relax();
}
```

Emballage natif : Les interruptions

- Solution possible : dans l'API
  - Une primitive pour « rendre la main, et attendre arbitrairement », `cpu_relax()` :

```
while (!condition) {
  cpu_relax();
}
```

  - Une primitive pour attendre une interruption, `wait_for_irq()` :

```
programmer_timer();
wait_for_irq();
printf("Le timer a expire\n");
/* code hautement pas robuste, l'interruption
   pourrait venir de n'importe ou. */
```

Les interruptions (l'API en SystemC)

- `cpu_relax()` (rendre la main) :

Les interruptions (l'API en SystemC)

- `cpu_relax()` (rendre la main) :

```
void NativeWrapper::cpu_relax() {
  wait(1, SC_MS); /* temps arbitraire */
}
```

- `wait_for_irq()` (attendre une interruption) :

Les interruptions (l'API en SystemC)

- `cpu_relax()` (rendre la main) :

```
void NativeWrapper::cpu_relax() {
  wait(1, SC_MS); /* temps arbitraire */
}
```

- `wait_for_irq()` (attendre une interruption) :

```
void NativeWrapper::wait_for_irq() {
  if (!interrupt) wait(interrupt_event);
  interrupt = false;
}
```

- Et le signal d'interruption déclenche une `SC_METHOD` :

```
void NativeWrapper::interrupt_handler() {
  interrupt = true; interrupt_event.notify();
  int_handler(); /* surchargeable */
}
```

Les interruptions (l'API pour la vraie puce)

- Il faut aussi implémenter l'API pour la vrai puce, pour que le logiciel tourne sans modifications sur le SoC.
- `cpu_relax()` (rendre la main) :

Les interruptions (l'API pour la vraie puce)

- Il faut aussi implémenter l'API pour la vrai puce, pour que le logiciel tourne sans modifications sur le SoC.
- `cpu_relax()` (rendre la main) :

```
void cpu_relax() {
/* Rien. Sur la puce, le temps passe de toutes
   facons. Selon la puce, on peut/doit diminuer
   la priorite du processus, vider le cache pour
   s'assurer qu'on lit une valeur fraiche... */ }
```

- `wait_for_irq()` (attendre une interruption) :

Les interruptions (l'API pour la vraie puce)

- Il faut aussi implémenter l'API pour la vrai puce, pour que le logiciel tourne sans modifications sur le SoC.
- `cpu_relax()` (rendre la main) :

```
void cpu_relax() {
/* Rien. Sur la puce, le temps passe de toutes
   facons. Selon la puce, on peut/doit diminuer
   la priorite du processus, vider le cache pour
   s'assurer qu'on lit une valeur fraiche... */ }
```

- `wait_for_irq()` (attendre une interruption) :

```
void wait_for_irq() {
/* specifique a la puce cible.
   Peut-etre une instruction assembleur
   dediee, peut-etre du polling, ... */ }
```

- ... et on enregistre `int_handler()` comme traitant d'interruption.

Inconvénients de l'emballage natif

- Pas de support de l'assembleur
  - Compilation native impossible
  - Identification des communications?
- Pas de visilibité des « autres » transactions
  - Accès à la pile
  - Accès au tas
  - Accès instructions (fetch)
- Analyse de performance très difficile (comment ?)

Sommaire de cette section

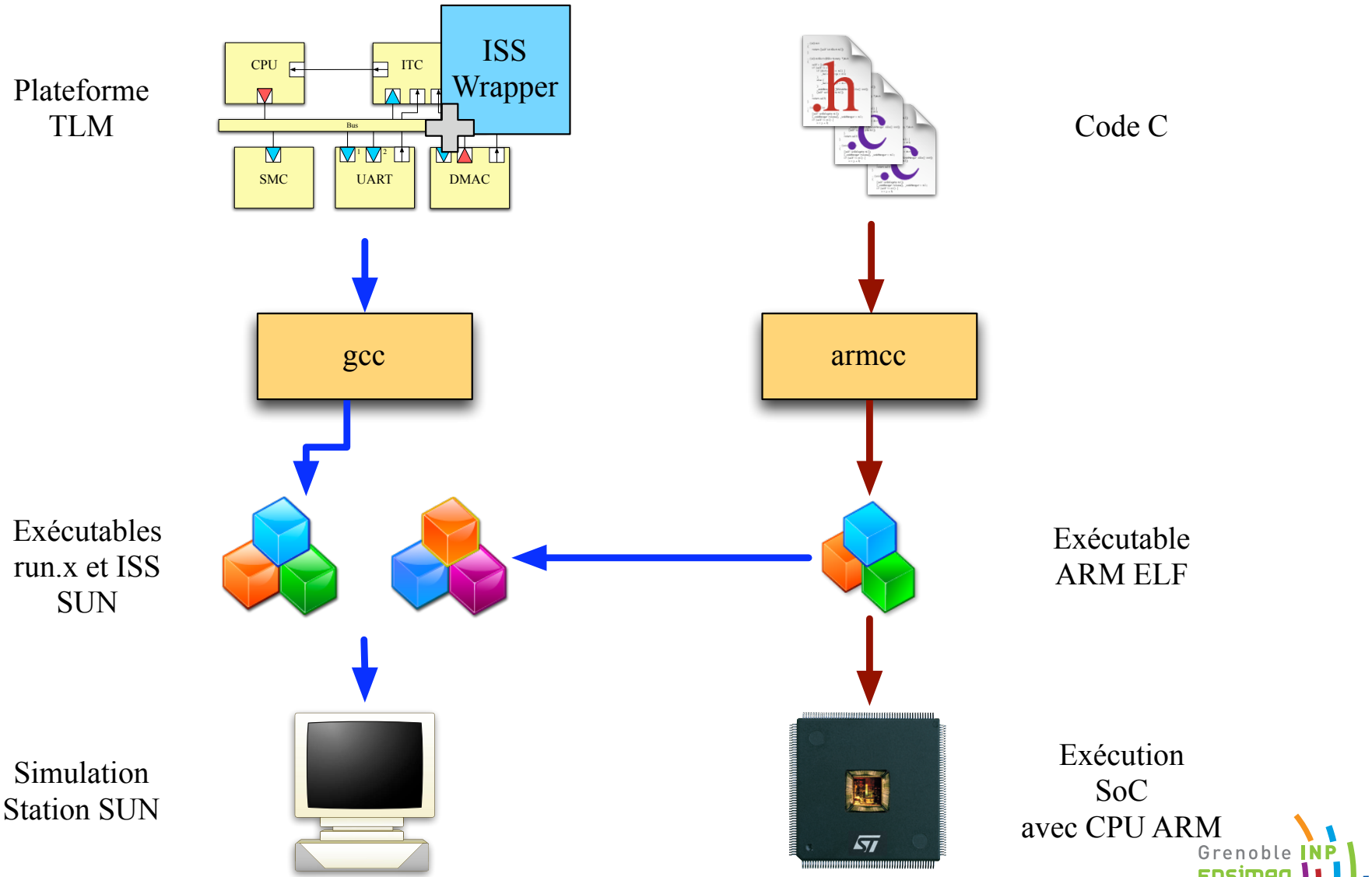
Présentation

- Simulateur de jeu d'instructions ou Instruction Set Simulator (ISS)
  - Émule les instructions d'un processeur donné
  - Simule éventuellement la microarchitecture (pipeline, caches, etc.)
- Plusieurs niveaux de fidélité à l'exécution
  - Instruction accurate
  - Cycle accurate
  - Cycle callable...

Utilisation

- En pratique : exécutable indépendant intégré par un emballage SystemC
  - Exécution sous forme d'un processus indépendant
  - Transformations des accès mémoires en transactions
  - Retransmission des interruptions à l'ISS
- Inconvénient : émulation $\Rightarrow$ très lent

Utilisation d'un ISS : exemple



Sommaire de cette section

Simplest ISS: Binary Interpreter

```
pc = 0;
while(true) {
    ir = read(pc);
    pc += sizeof(ir);
    switch(OPCODE(ir)) {
        case ADD:
            regs[OP2(ir)] = regs[OP1(ir)]
                          + regs[OP2(ir)];
            break;
        case SUB: // ...
        case JMP:
            pc = ...; break;
        default: abort();
    }
    wait(period);
}
```

Example : MicroBlaze ISS (TP3)

- MicroBlaze = Xilinx “Softcore” (FPGA only)
- Pure C++ ISS available, open-source, ≈ 1200 lines of code
  - Provides a `step()` method to execute one run of the loop,
  - Communicates with the outside world with `getDataRequest()`, `setDataResponse()`, `setIrq()`, ...

- Wrapped in an `SC_MODULE` connected to an `ensitlm` socket & bus.

```
while(true) {
    // do read/write on the bus as needed
    m_iss.step();
    wait(period);
}
```

Example : MicroBlaze ISS (TP3)

- MicroBlaze = Xilinx “Softcore” (FPGA only)
- Pure C++ ISS available, open-source, ≈ 1200 lines of code
  - Provides a `step()` method to execute one run of the loop,
  - Communicates with the outside world with `getDataRequest()`, `setDataResponse()`, `setIrq()`, ...

  ⇒ you don’t have to code it
- Wrapped in an `SC_MODULE` connected to an `ensitlm` socket & bus.

```
while(true) {
    // do read/write on the bus as needed
    m_iss.step();
    wait(period);
}
```

⇒ you will have to code it

More advanced ISS: Dynamic Translation Techniques

- JIT (Just-In-Time) compiler Target→Bytecode→Host
- Compile basic blocks as it reaches them (⇒ compile once, execute many times)
- Examples:
  - QEmu (Open Source, work needed to connect to SystemC)
  - SimSoC (research project, open-source)
  - OVP : Open Virtual Platform (Proprietary, commercially supported)
  - Many other modern “fast ISS”
- Still slower than native simulation (but can be faster than real chip)

Sommaire de cette section

Problème

- Intégration de logiciel tournant sur un système d'exploitation?

Problème

- Intégration de logiciel tournant sur un système d'exploitation?

Solution ISS

- Fonctionne...
- ...mais très lent

Problème

- Intégration de logiciel tournant sur un système d'exploitation?

Solution ISS

- Fonctionne...
- ...mais très lent

Solution emballage natif

- Nécessité de compiler l'OS pour la machine de simulation
- Portions de l'OS bas niveaux en assembleur...
- Correspondance appels bas niveaux/transactions?

OS Emulation (1/2)

- Émulation du système d'exploitation (OS Emulation)
  - Rien à voir avec l'émulation en général...
- Objectifs
  - Simulation rapide
  - Intégration transparente du logiciel embarqué
  - Production des transactions dans la plateforme...
- Généralisation de la couche d'abstraction du hardware des OS

OS Emulation (2/2)

- Exemple : Linux
  - Portage « SystemC/C++ TLM » des fonctions du noyau
  - Compilation du logiciel embarqué pour la machine de simulation
  - Résultats :

Technique	Time for boot
ISS	3 min
Native	less than 3 s

Sommaire

Sommaire de cette section

Questions

- Rôle du temps en SystemC?
- Rôle du temps en TLM?

Questions

- Rôle du temps en SystemC?
  - `wait(temps)` change l'ordre des actions en SystemC
  - Mélange entre mesure du temps et fonctionnalité
- Rôle du temps en TLM?

Questions

- Rôle du temps en SystemC?
  - `wait(temps)` change l'ordre des actions en SystemC
  - Mélange entre mesure du temps et fonctionnalité
- Rôle du temps en TLM?
  - Exécution correcte du logiciel embarqué non dépendante du temps... (robustesse)
  - Fonctionnement correct d'une plateforme non dépendant du temps?
    - ★ Synchro par le temps : mauvaise pratique!

Questions

- Rôle du temps en SystemC?
  - `wait(temps)` change l'ordre des actions en SystemC
  - Mélange entre mesure du temps et fonctionnalité
- Rôle du temps en TLM?
  - Exécution correcte du logiciel embarqué non dépendante du temps... (robustesse)
  - Fonctionnement correct d'une plateforme non dépendant du temps?
    - ★ Synchro par le temps : mauvaise pratique!
    - ★ Mais... notion de temps fonctionnel

Questions

- Rôle du temps en SystemC?
  - `wait(temps)` change l'ordre des actions en SystemC
  - Mélange entre mesure du temps et fonctionnalité
- Rôle du temps en TLM?
  - Exécution correcte du logiciel embarqué non dépendante du temps... (robustesse)
  - Fonctionnement correct d'une plateforme non dépendant du temps?
    - ★ Synchro par le temps : mauvaise pratique!
    - ★ Mais... notion de temps fonctionnel
  - Analyse d'architecture?

LT/AT/CA : présentation

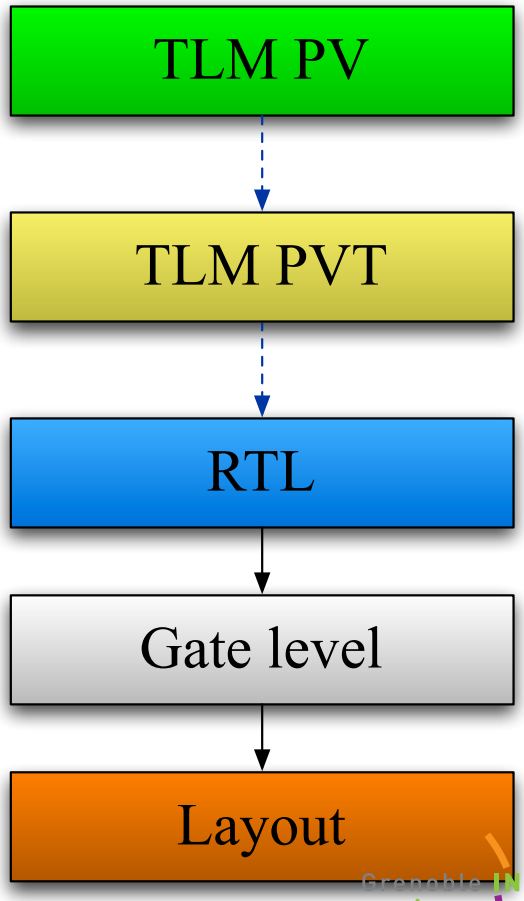
- Besoin conflictuels en TLM
- Timed/Untimed, Granularité...

TLM Programmer's View (PV) / Loosely Timed (LT)

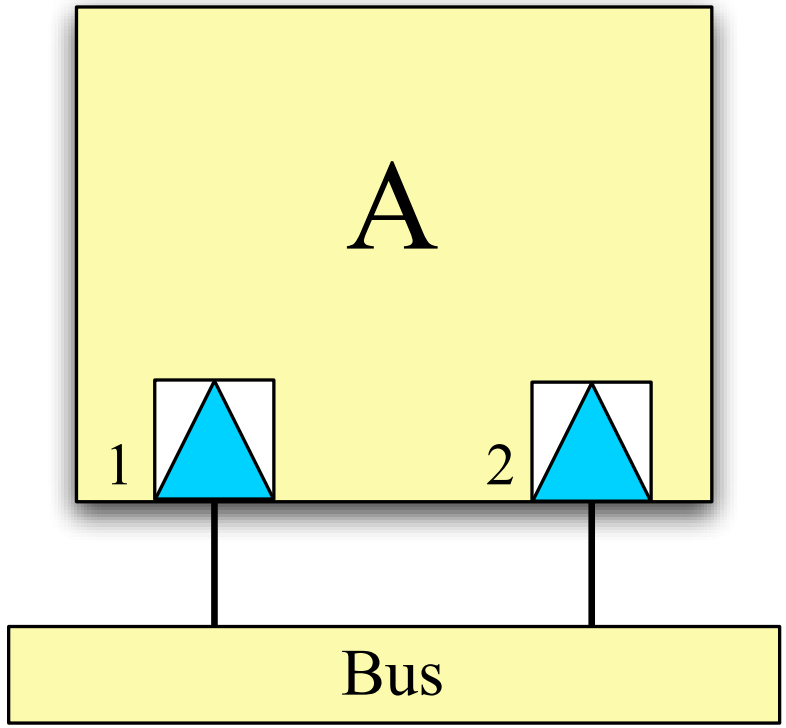
- Le temps n'a pas de signification
- Communications gros grain
- Utilisation : Développement du logiciel embarqué, Intégration Système

TLM Programmer's View with Time (PVT) / Approximately Timed (AT)

- Temps précis induits par la microarchitecture
- Communications à la taille du bus
- Utilisation : Évaluation d'Architecture

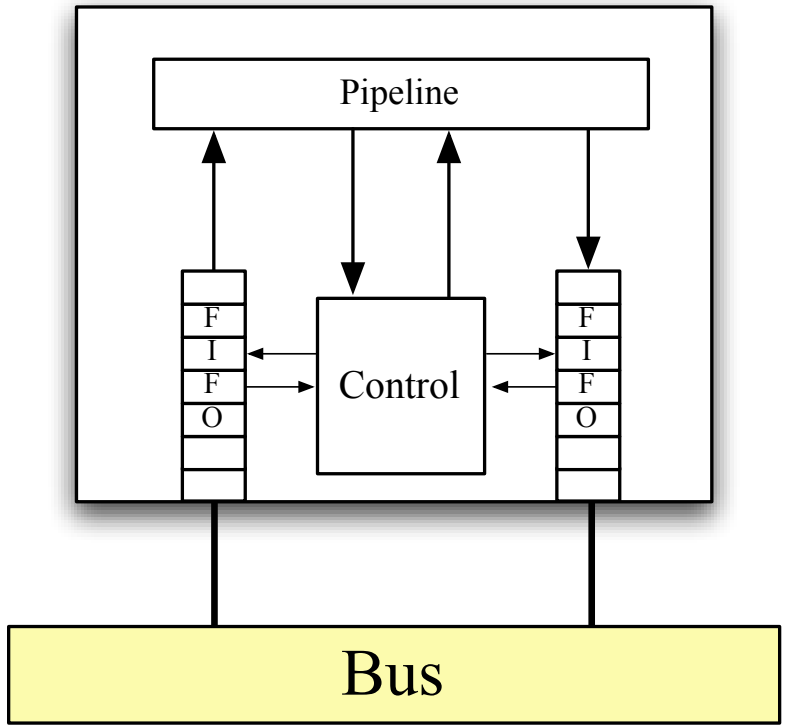


LT/AT : modèle de microarchitecture



1. Granularité
2. Fonctionnalité de microarchitecture (fifos, pipeline...)
3. Durées de traitement

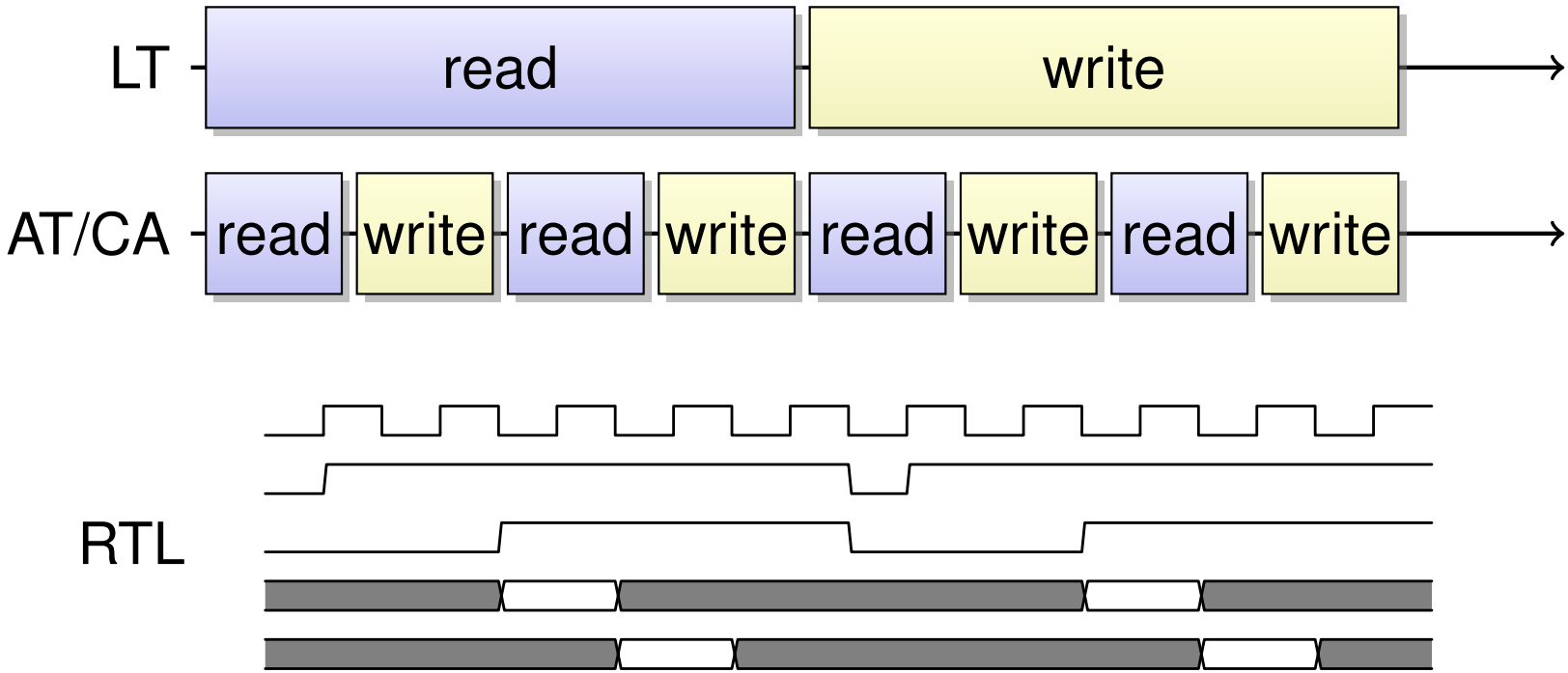
LT/AT : modèle de microarchitecture



1. Granularité
2. Fonctionnalité de microarchitecture (fifos, pipeline...)
3. Durées de traitement

LT/AT/CA : exemple de traces

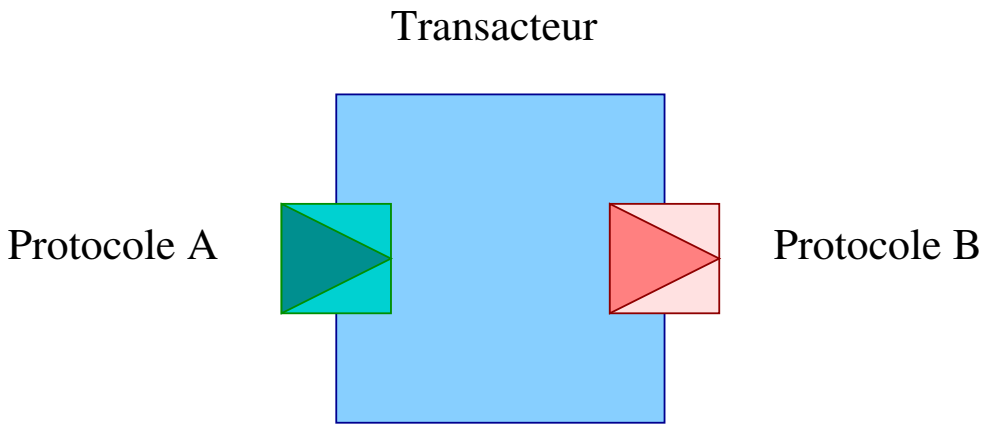
- Exemple Transfert Mémoire :



Sommaire de cette section

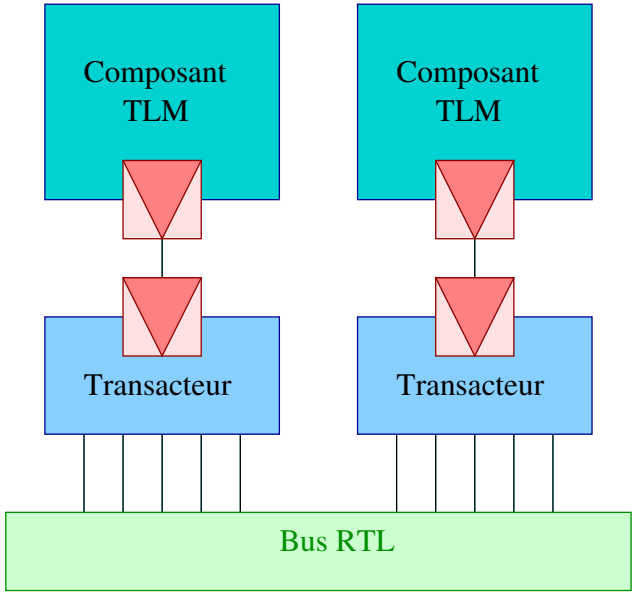
Transacteurs

- Transacteur = composant « pont » entre deux protocoles

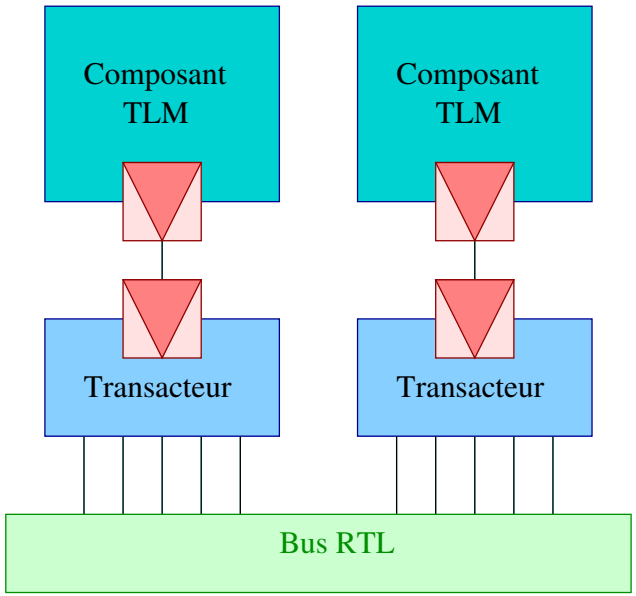


- Toutes combinaisons de A et B (TLM, RTL, différents protocoles)

PVT avec des transacteurs



PVT avec des transacteurs



Question

Où est la limitation?

PVT avec de l'instrumentation de code

Avant

```
Image i = read_image(addr1);
Image i2 = encode_image(i);
write_image(i2, addr2);
```

Après

```
Image i = read_image(addr1);
wait(42, SC_MS); // time to read
Image i2 = encode_image(i);
wait(234, SC_MS); // time to encode
write_image(i2, addr2);
wait(54, SC_MS); // time to write
```

PVT avec de l'instrumentation de code

Avant

```
Image i = read_image(addr1);
Image i2 = encode_image(i);
write_image(i2, addr2);
```

Après

```
Image i = read_image(addr1);
wait(42, SC_MS); // time to read
Image i2 = encode_image(i);
wait(234, SC_MS); // time to encode
write_image(i2, addr2);
wait(54, SC_MS); // time to write
```

Question

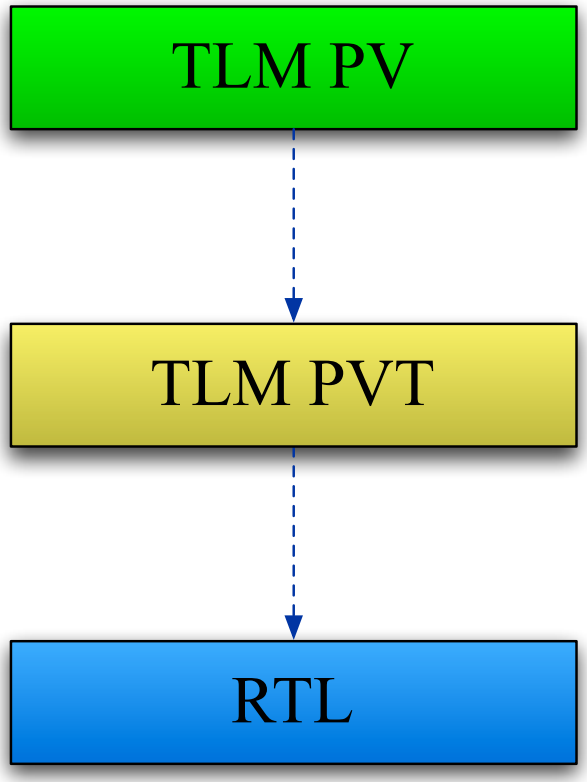
Où est la limitation?

Conclusion intermédiaire

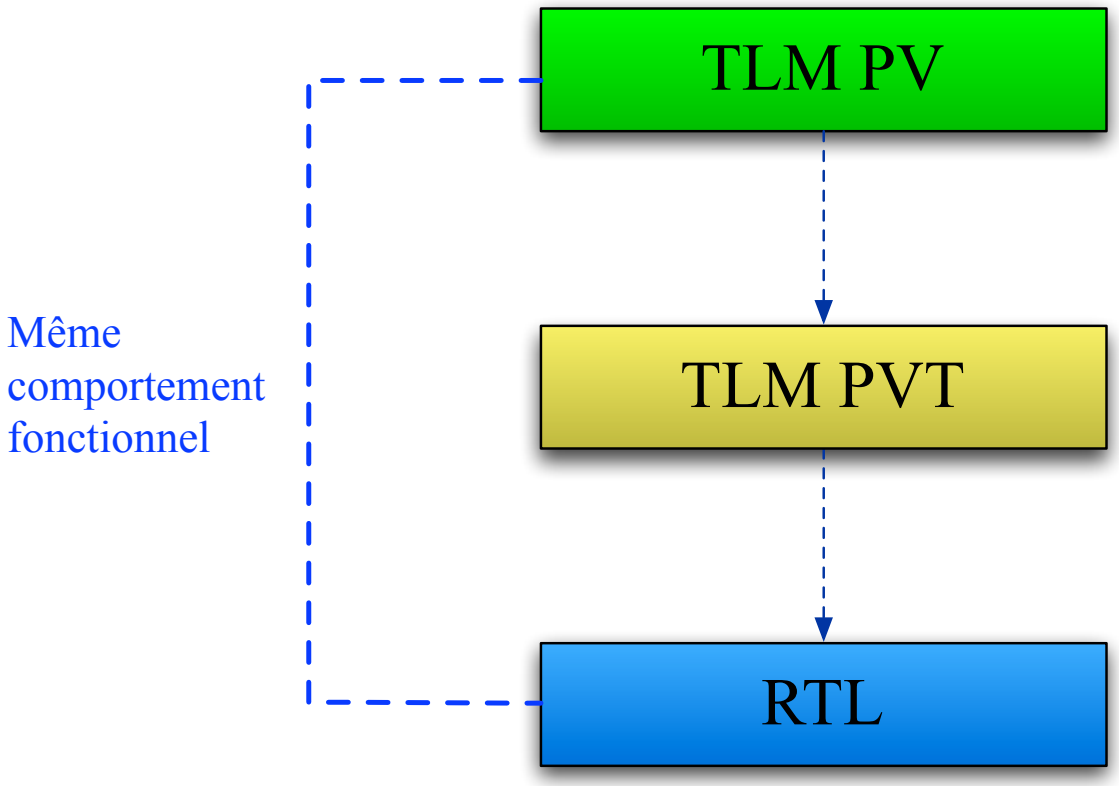
- Transacteurs, instrumentation : solutions très imparfaite, mais vraiment utilisées par des vrais gens.
- Faire du AT correctement est un problème difficile.

Sommaire de cette section

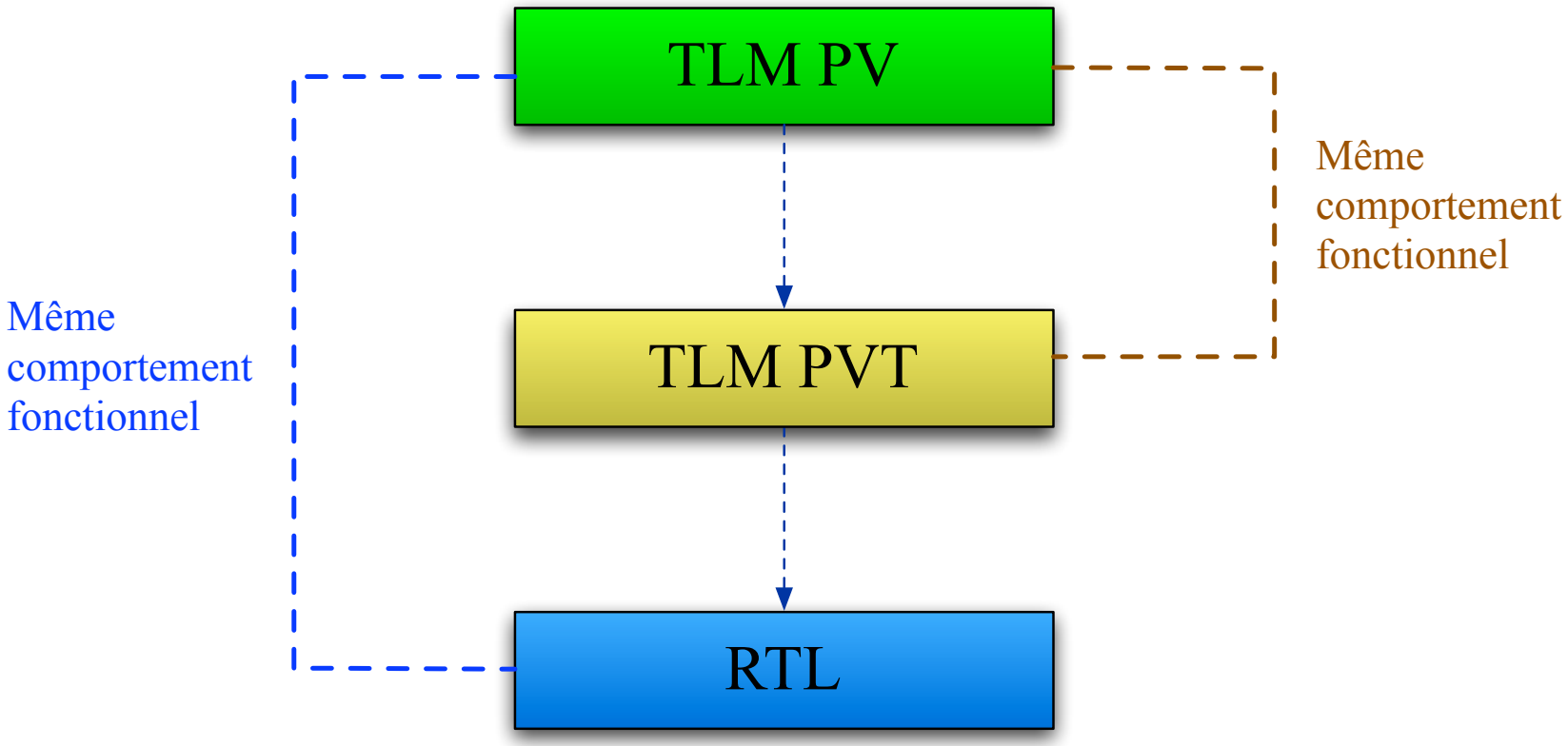
PV+T : contraintes



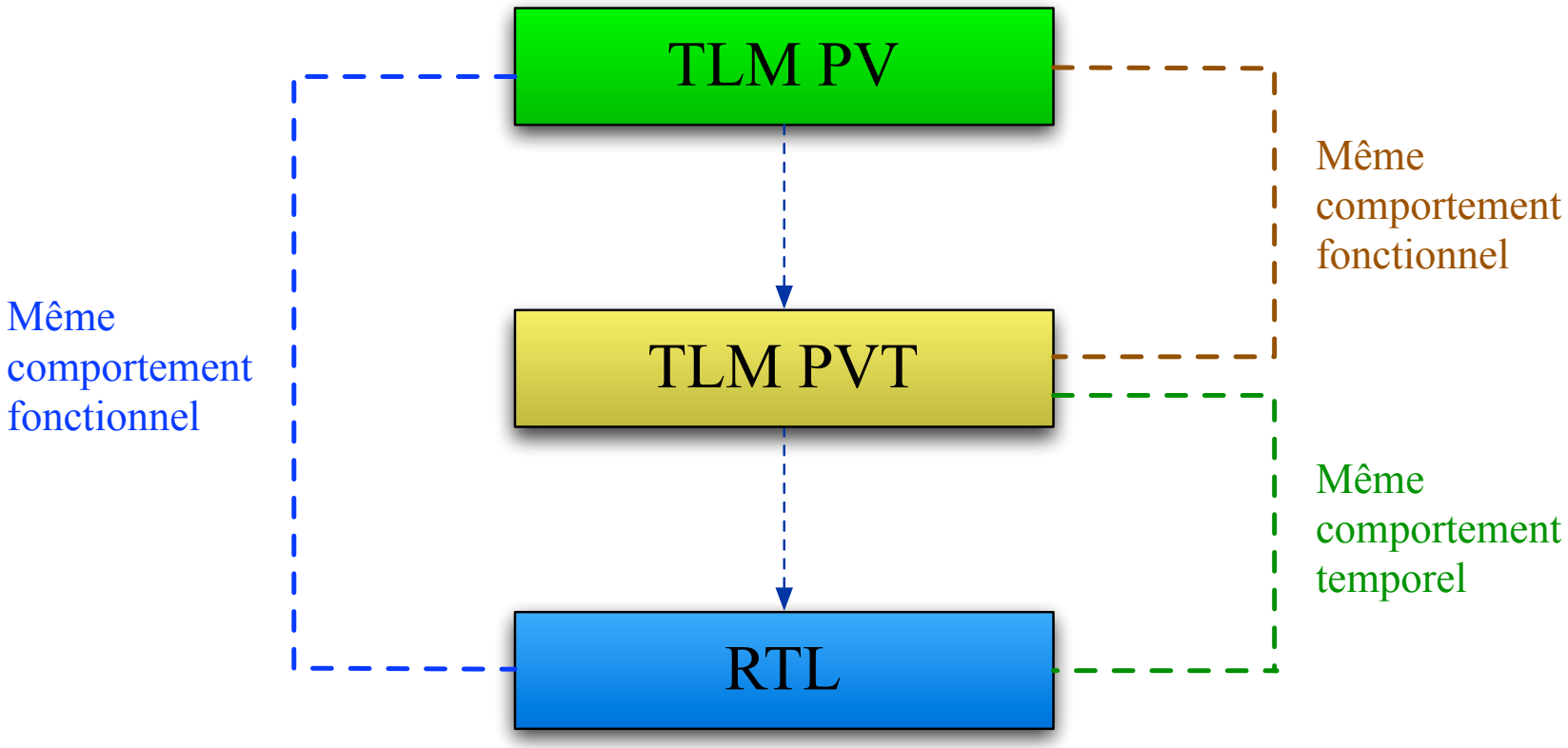
PV+T : contraintes



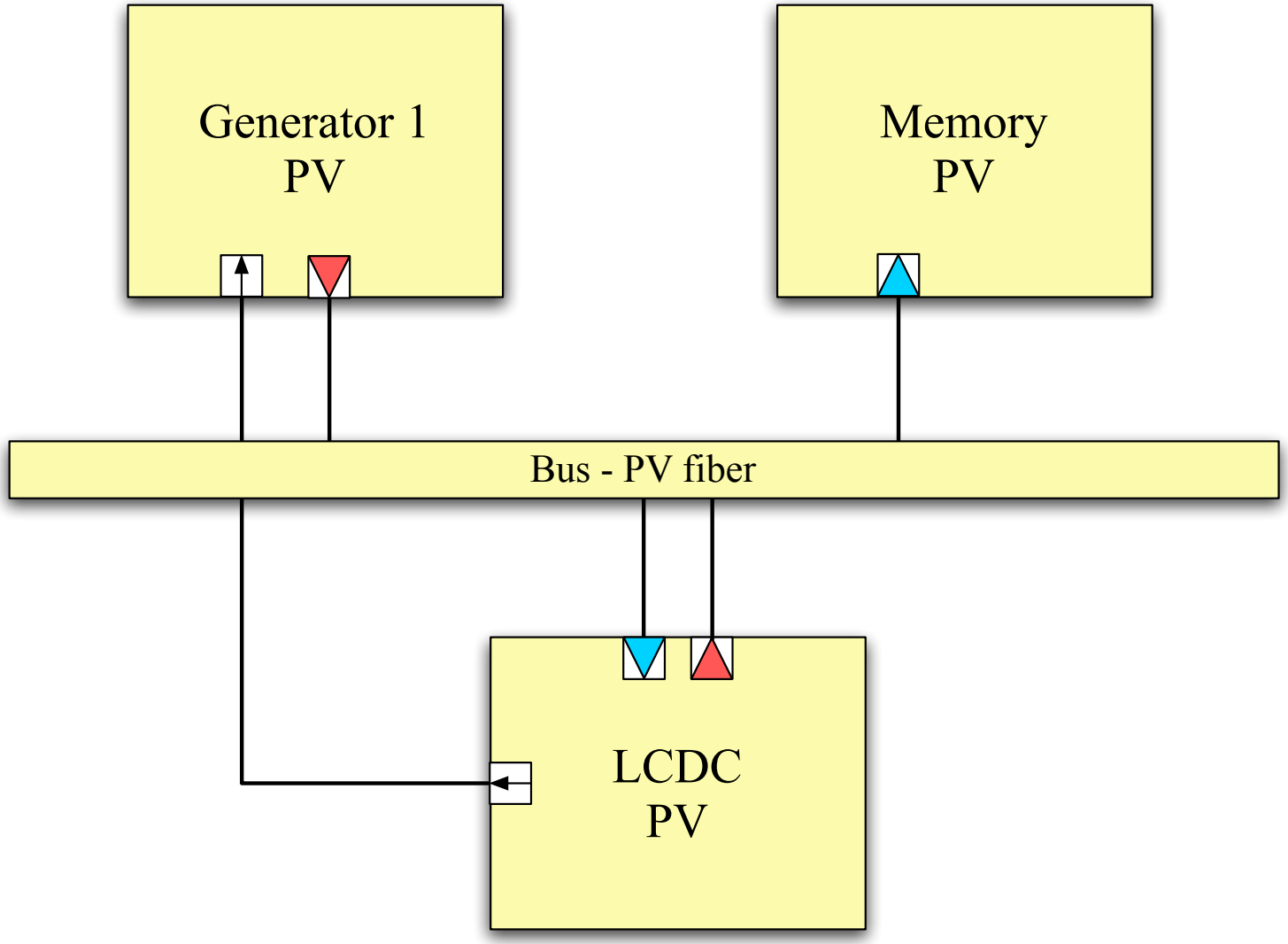
PV+T : contraintes



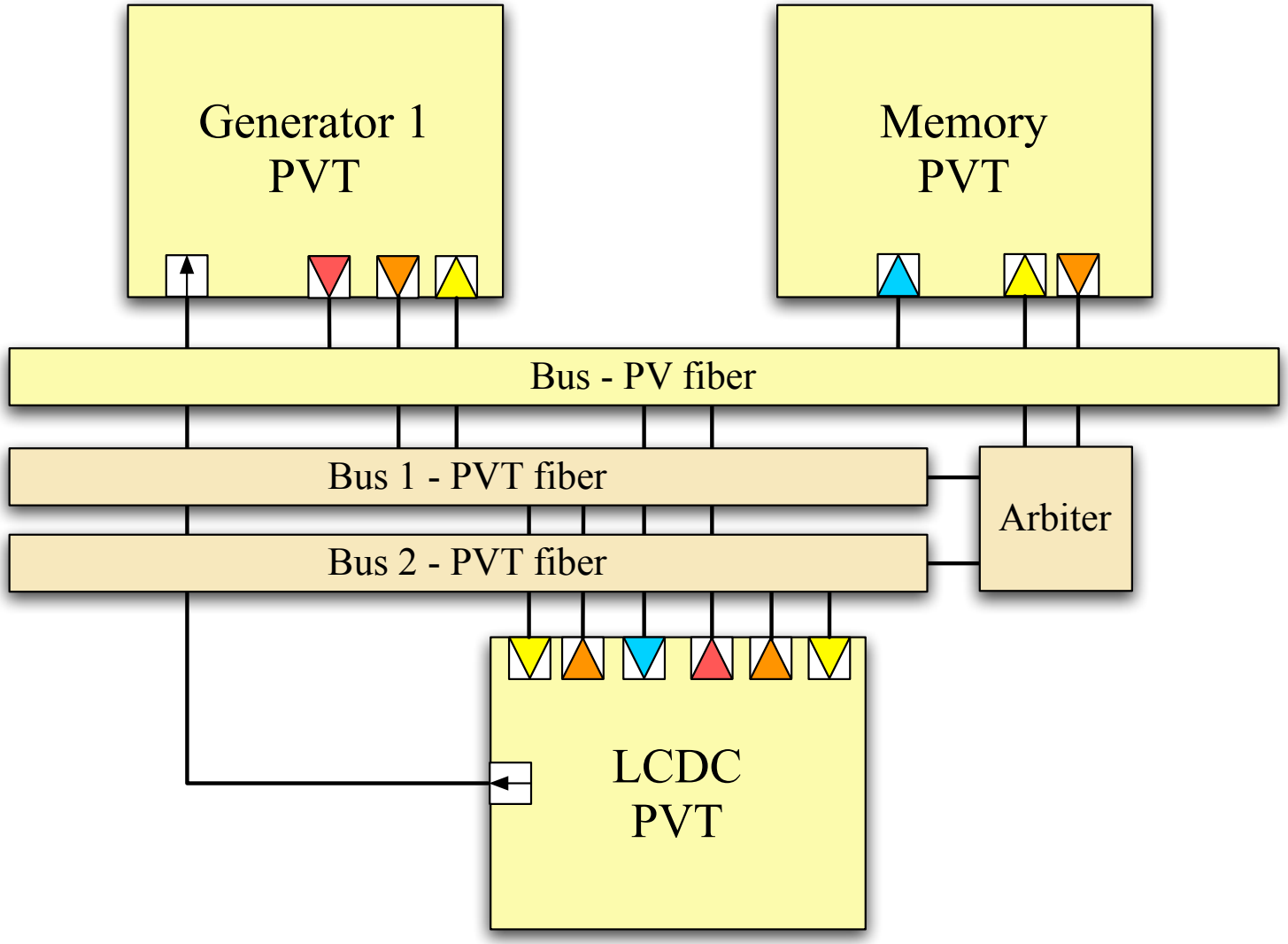
PV+T : contraintes



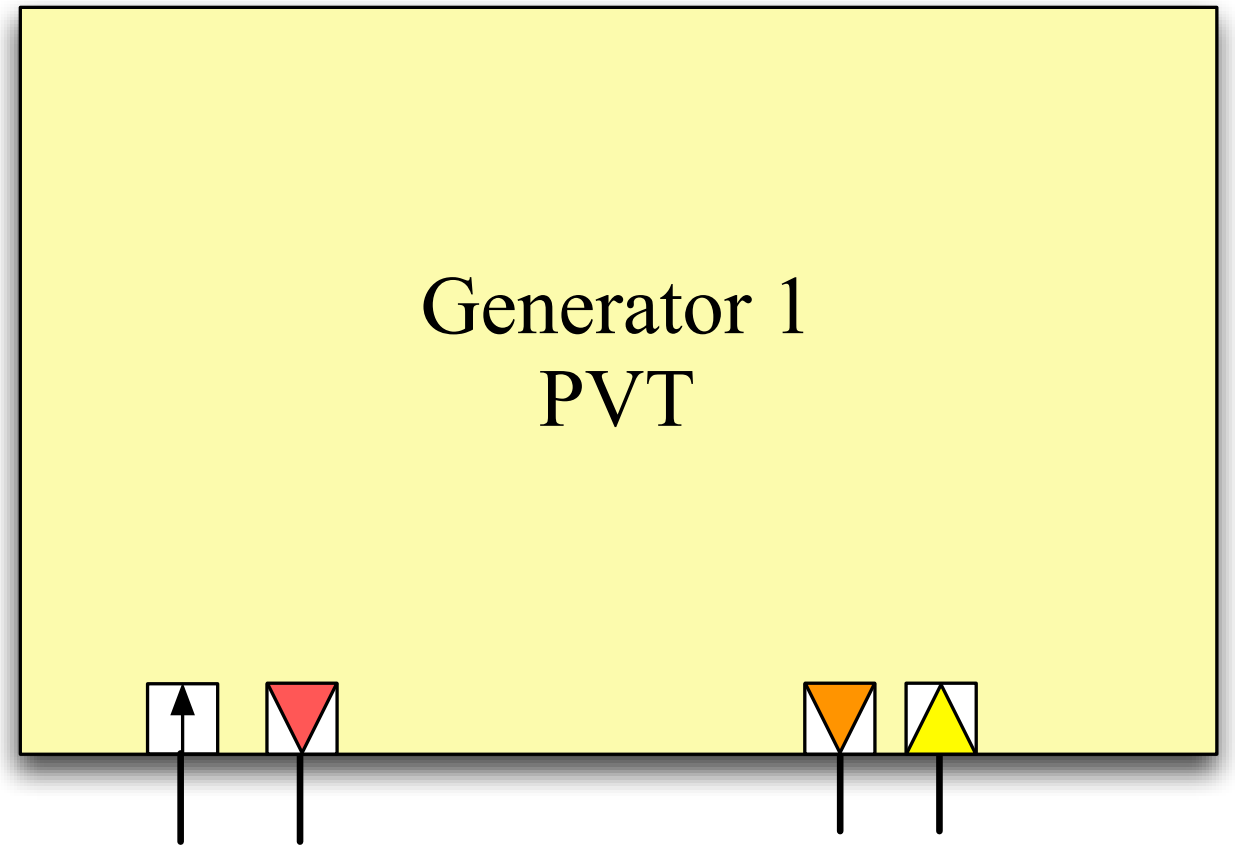
Exemple (PV ≈ LT)



Exemple (PVT ≈ AT)

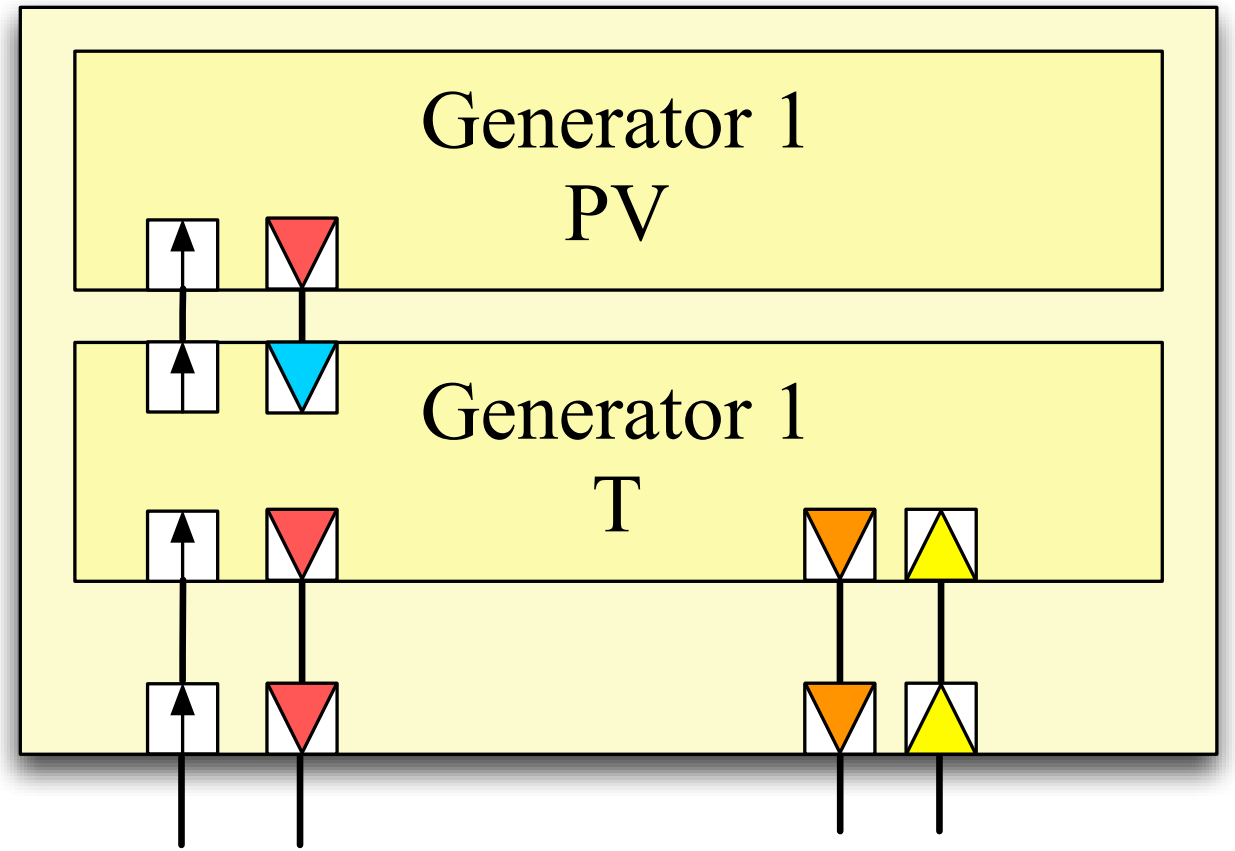


Module PVT



- Generator 1 PVT

Module PVT



- Generator 1 PVT

Sommaire de cette section

AT Models In Practice Today

- No perfect solution exist
- Precise timing requires
  - Important modeling effort
  - Slower simulation compared to LT or untimed

  $\Rightarrow$ Cost/benefit not as good as LT
- Some people prefer RTL (+ Co-simulation/co-emulation/...)

Different Levels of Timing Precision

- Cycle accurate, Bit Accurate (CABA)
- Approximately-timed (AT $\approx$ PVT) :
  Tries to be timing-accurate.
- Loosely-timed (LT $\approx$ PV) :
  Doesn't target timing accuracy, but can use time to work (e.g. timers)
- Purely untimed:
  Purely asynchronous execution, nothing relies on timing.