

Modélisation Transactionnelle des Systèmes sur Puce avec SystemC Ensimag 3A — filière SLE Grenoble-INP

De l'usage des plateformes transactionnelles

Frédéric Pétrot
(transparents originaux : Jérôme Cornet, puis Matthieu Moy)

frederic.petrot@univ-grenoble-alpes.fr

2017-2018



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 Avancées 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 : Laurent Maillet-Contoz (STMicroelectronics)
- 16 Perspectives et conclusion

Sommaire

- 1 Rappel (?) : Usage des plateformes TLM
- 2 TLM pour la vérification matérielle
- 3 TLM pour le développement logiciel
- 4 TLM pour l'exploration d'architecture : évaluation de performances

Sommaire

- 1 Rappel (?) : Usage des plateformes TLM
- 2 TLM pour la vérification matérielle
- 3 TLM pour le développement logiciel
- 4 TLM pour l'exploration d'architecture : évaluation de performances

Bénéfices de TLM (comparé à RTL)

- rapide (accélération de 100 à 10000 fois vs RTL)
- effort de modélisation modéré
- à fonctionnalité équivalente (précis au bit près)

Mais ... !

- désavantages du TLM :
 - ▶ moins voire pas précis pour estimer les performances
 - ▶ pas synthétisable (\Rightarrow ne peut se substituer au RTL)
 - ▶ À quoi ça peut bien servir ? ! ?

Mais ... !

- désavantages du TLM :
 - ▶ moins voire pas précis pour estimer les performances
 - ▶ pas synthétisable (\Rightarrow ne peut se substituer au RTL)
 - ▶ À quoi ça peut bien servir?!?
- Utilisation des plateformes TLM :
 - ▶ développement logiciel, mise au point du logiciel
 - ▶ vérification du matériel,
 - ▶ partitionnement hard/soft, exploration d'architecture.

Mais ... !

- désavantages du TLM :
 - ▶ moins voire pas précis pour estimer les performances
 - ▶ pas synthétisable (\Rightarrow ne peut se substituer au RTL)
 - ▶ À quoi ça peut bien servir ? ! ?
- Utilisation des plateformes TLM :
 - ▶ développement logiciel, mise au point du logiciel
 - ▶ vérification du matériel,
 - \Rightarrow **précision fonctionnelle requise**
 - ▶ partitionnement hard/soft, exploration d'architecture.
 - \Rightarrow **besoin d'estimation de perfs en temps, conso, température, ...**

Sommaire

- 1 Rappel (?) : Usage des plateformes TLM
- 2 TLM pour la vérification matérielle
- 3 TLM pour le développement logiciel
- 4 TLM pour l'exploration d'architecture : évaluation de performances

Vérification d'une IP RTL

- test
 - ▶ appliquer des stimuli à l'IP RTL

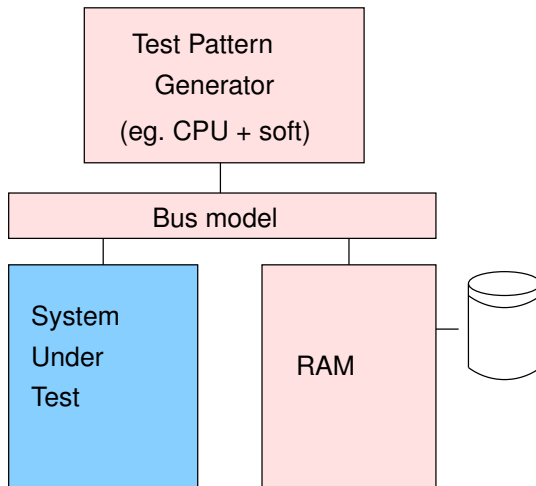
Vérification d'une IP RTL

- test
 - ▶ appliquer des stimuli à l'IP RTL
 - ▶ appliquer les mêmes stimuli à l'IP TLM
 - ▶ Comparer

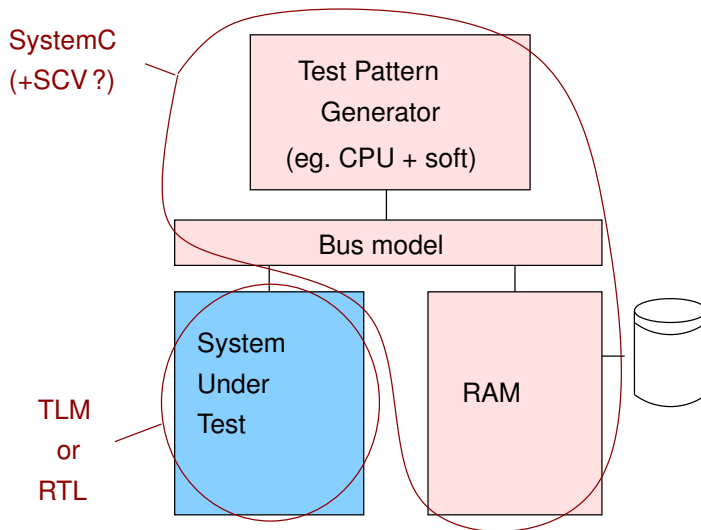
Vérification d'une IP RTL

- test
 - ▶ appliquer des stimuli à l'IP RTL
 - ▶ appliquer les mêmes stimuli à l'IP TLM
 - ▶ Comparer
- qu'est ce qu'un cas de test ?
 - ▶ généralement, 1 cas de test = 1 petit programme qui fait des accès en lecture/écriture sur l'IP

Environnement de test typique d'IPs



Environnement de test typique d'IPs



TLM et la vérification

- SystemC/TLM fournit les outils pour :
 - ▶ construire l'environnement de test
 - ▶ construire les modèles de référence pour les IPs et les plateformes ;
 - ▶ générer les vecteurs de test (SCV/PSS ?)
- Avantages par rapport à d'autres solutions
 - ▶ gratuit, pas de dépendance vis-à-vis des vendeurs de CAD
 - ▶ en voie de standardisation, largement répandue et connue

Sommaire

- 1 Rappel (?) : Usage des plateformes TLM
- 2 TLM pour la vérification matérielle
- 3 TLM pour le développement logiciel
- 4 TLM pour l'exploration d'architecture : évaluation de performances

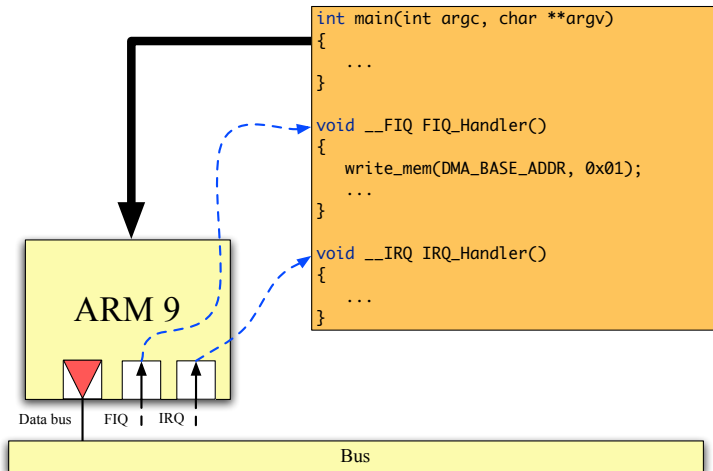
Rappel : principes du TLM

- modéliser ce dont le logiciel a besoin, et seulement ça

Rappel : principes du TLM

- modéliser ce dont le logiciel a besoin, et seulement ça
- modélisation :
 - ▶ comportement fonctionnel
 - ▶ carte des adresses (*address map*)
 - ▶ architecture matérielle vue d'avion
- abstraction :
 - ▶ micro-architecture (pipeline, ...)
 - ▶ détails des protocoles, des machines d'états précises, du matériel transparent pour le logiciel (caches, MMU HW, etc)

Interface du processeur (= API de bas niveau pour le logiciel)



Deux manières d'intégrer le logiciel

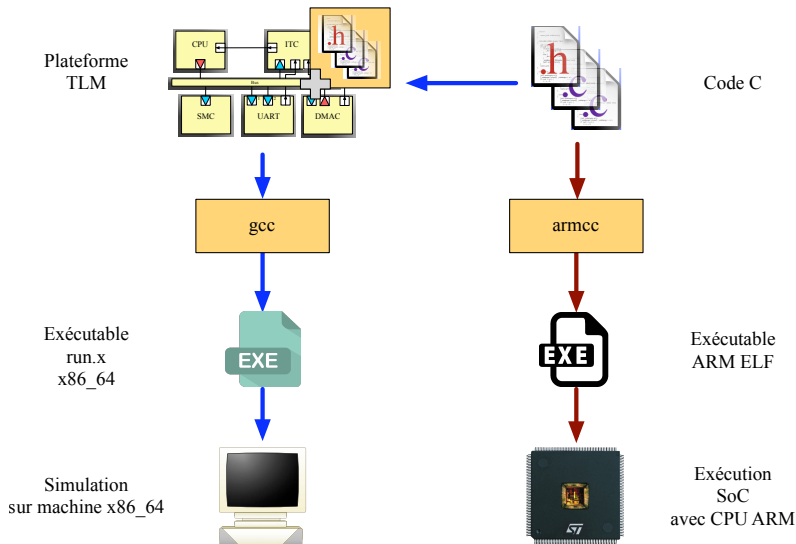
- simulateur de jeu d'instruction (*Instruction set simulator* – ISS)
 - ▶ compilation croisée du logiciel pour le processeur **cible** (**target** CPU)
 - ▶ chargement du binaire dans la mémoire simulée
 - ▶ interprétation (plus ou moins complexe) à partir du PC de reset
- encapsulation native (*native wrapper*)
 - ▶ compilation native du logiciel pour le processeur **hôte** (**host** CPU)
 - ▶ édition de lien avec la librairie SystemC
 - ▶ généralement considéré comme un processus SystemC

Sommaire de cette section

3 TLM pour le développement logiciel

- Encapsulation native
- Simulateur de jeux d'instructions (ISS)
- ISS : Comment ça marche ?
- Et les OS ?

Exemple d'encapsulation native



Problèmes liés à l'encapsulation native

- integration dans la plateforme TLM
 - ▶ plusieurs solutions : librairies statiques ou dynamiques, techniques venues des machines virtuelles (HAV), ...

Problèmes liés à l'encapsulation native

- integration dans la plateforme TLM
 - ▶ plusieurs solutions : bibliothèques statiques ou dynamiques, techniques venues des machines virtuelles (HAV), ...
- transactions produites par le CPU

Problèmes liés à l'encapsulation native

- integration dans la plateforme TLM
 - ▶ plusieurs solutions : librairies statiques ou dynamiques, techniques venues des machines virtuelles (HAV), ...
- transactions produites par le CPU
 - ▶ accès à la mémoire par le logiciel embarqué
 - ▶ et le reste (*c.f. plus tard*)

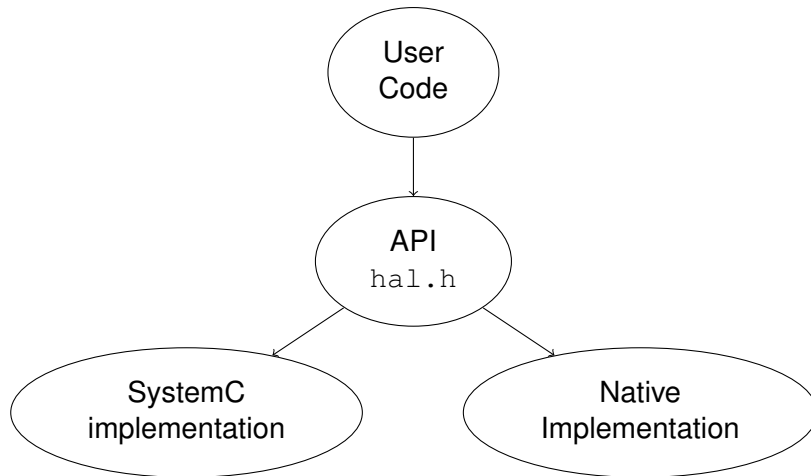
Problèmes liés à l'encapsulation native

- integration dans la plateforme TLM
 - ▶ plusieurs solutions : librairies statiques ou dynamiques, techniques venues des machines virtuelles (HAV), ...
- transactions produites par le CPU
 - ▶ accès à la mémoire par le logiciel embarqué
 - ▶ et le reste (*c.f. plus tard*)
- prise en compte des interruptions

Problèmes liés à l'encapsulation native

- integration dans la plateforme TLM
 - ▶ plusieurs solutions : librairies statiques ou dynamiques, techniques venues des machines virtuelles (HAV), ...
- transactions produites par le CPU
 - ▶ accès à la mémoire par le logiciel embarqué
 - ▶ et le reste (*c.f. plus tard*)
- prise en compte des interruptions
 - ▶ comment interrompre l'exécution du logiciel embarqué ?

Une solution : une API, plusieurs implantations



Une solution : une API, plusieurs implantations

- « para-virtualisation » à l'aide d'un Hardware Abstraction Layer (HAL)
- utilisation de cette API dans le logiciel embarqué
 - ▶ contrainte imposée au programmeur
 - ▶ API portable : définie à la fois pour le modèle SystemC et pour le circuit réel
 - ▶ implantations différentes pour SystemC et pour le circuit réel

Une solution : une API, plusieurs implantations

- « para-virtualisation » à l'aide d'un Hardware Abstraction Layer (HAL)
- utilisation de cette API dans le logiciel embarqué
 - ▶ contrainte imposée au programmeur
 - ▶ API portable : définie à la fois pour le modèle SystemC et pour le circuit réel
 - ▶ implantations différentes pour SystemC et pour le circuit réel
- premier exemple : accès à travers le bus
(aux composants cibles : mémoire, périphériques, etc)

```
void hal_write32(addr_t addr, data_t data);  
data_t hal_read32(addr_t addr);
```

- contraint le développeur à utiliser `hal_write32` et `hal_read32` au lieu de
`*addr = ...` ou `... = *addr`
- édition de lien avec l'implantation appropriée en fonction de la plateforme d'exécution

Une solution : une API, plusieurs implantations

- « para-virtualisation » à l'aide d'un Hardware Abstraction Layer (HAL)
- utilisation de cette API dans le logiciel embarqué
 - ▶ contrainte imposée au programmeur
 - ▶ API portable : dénie à la fois pour le modèle SystemC et pour le circuit réel
 - ▶ implantations différentes pour SystemC et pour le circuit réel
- premier exemple : accès à travers le bus
(aux composants cibles : mémoire, périphériques, etc)

```
void hal_write32(addr_t addr, data_t data);  
data_t hal_read32(addr_t addr);
```

Question



Voit-on tous les accès mémoire ?

Implantation SystemC pour la simulation TLM native

- accès aux cibles

```
void hal_write32(addr_t addr, data_t data)
{
    socket.write(addr, data);
}

data_t hal_read32(addr_t addr)
{
    return socket.read(addr, data);
}
```


Implantation pour la cible (pour la simulation à base d'ISS *et* le circuit)

- accès aux cibles

```
void hal_write32(addr_t addr, data_t data)
{
    *(volatile data_t *)addr = data;
}
```

```
data_t hal_read32(addr_t addr)
{
    return *(volatile data_t *)addr;
}
```

Deuxième problème : les interruptions

- problème :

Question



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

Deuxième problème : les interruptions

- problème :

- ▶ pas de primitive du langage pour gérer les interruptions en C ou C++
- ▶ dépend de l'architecture cible (registres, instructions assembleur, ...)
- ▶ au moins trois façons de gérer les interruptions :
 - ★ de manière asynchrone (la bonne), avec un ISR (interrupt service routine)

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

- ★ attente active, dédie le CPU à ce rôle (*polling*, c'est gâcher) :

```
while (!condition); /* nothing */
```

- ★ attente inactive, CPU en mode *sleep* (instruction assembleur `wait` (ppc), `wfi` (arm), `hlt` (x86), ...)

Deuxième problème : les interruptions

- problème :

- ▶ pas de primitive du langage pour gérer les interruptions en C ou C++
- ▶ dépend de l'architecture cible (registres, instructions assembleur, ...)
- ▶ au moins trois façons de gérer les interruptions :
 - ★ de manière asynchrone (la bonne), avec un ISR (interrupt service routine)

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

- ★ attente active, dédie le CPU à ce rôle (*polling*, c'est gâcher) :

```
while (!condition); /* nothing */
```

- ★ attente inactive, CPU en mode *sleep* (instruction assembleur `wait` (ppc), `wfi` (arm), `hlt` (x86), ...)

- ▶ Ça ne va pas marcher « tel quel » :

- ★ attente inactive simulée \equiv boucle infinie
OS hôte exécutera tout de même les autres processus, ...

Deuxième problème : les interruptions

- problème :

- ▶ pas de primitive du langage pour gérer les interruptions en C ou C++
- ▶ dépend de l'architecture cible (registres, instructions assembleur, ...)
- ▶ au moins trois façons de gérer les interruptions :
 - ★ de manière asynchrone (la bonne), avec un ISR (interrupt service routine)

```
void irq_handler(void) { ... }
```
 - ★ attente active, dédie le CPU à ce rôle (*polling*, c'est gâcher) :

```
while (!condition); /* nothing */
```
 - ★ attente inactive, CPU en mode *sleep* (instruction assembleur `wait` (ppc), `wfi` (arm), `hlt` (x86), ...)
- ▶ Ça ne va pas marcher « tel quel » :
 - ★ attente inactive simulée \equiv boucle infinie
OS hôte exécutera tout de même les autres processus, ...

Question



Mais alors, comment faire ?

Emballage natif : Les interruptions

- Solution possible : dans l'API

- ▶ Une primitive pour « rendre la main, et attendre arbitrairement »,

`hal_cpu_relax()` :

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

Emballage natif : Les interruptions

- Solution possible : dans l'API

- ▶ Une primitive pour « rendre la main, et attendre arbitrairement »,

`hal_cpu_relax()` :

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

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

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

Les interruptions (API en SystemC)

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

Les interruptions (API en SystemC)

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

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

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

Les interruptions (API en SystemC)

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

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

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

```
void NativeWrapper::hal_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 vraie puce, pour que le logiciel tourne sans modifications sur le SoC.
- `hal_cpu_relax()` (rendre la main) :

Les interruptions (l'API pour la vraie puce)

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

```
void hal_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... */ }
```

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

Les interruptions (l'API pour la vraie puce)

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

```
void hal_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... */ }
```

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

```
void hal_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 d'OS, etc
 - ▶ pas de code automodifiant : librairies dynamiques, java, javascript, etc
- pas de visibilité des transactions liée à la mémoire du programme
 - ▶ accès à la pile
 - ▶ accès au tas
 - ▶ accès instructions (*fetch*)
 - ▶ accès à la mémoire par un DMA ? Quelles adresses ?
- analyse de performance très délicate (comment ?)

Sommaire de cette section

- 3 TLM pour le développement logiciel
 - Encapsulation native
 - **Simulateur de jeux d'instructions (ISS)**
 - ISS : Comment ça marche ?
 - Et les OS ?

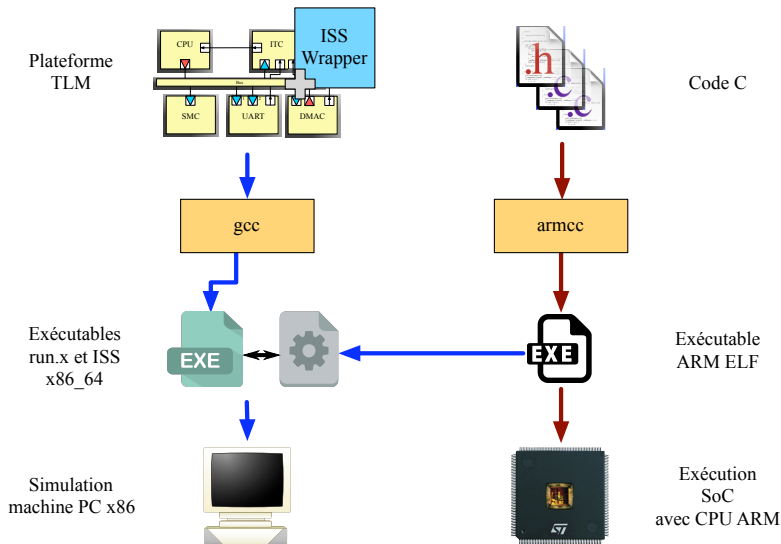
Présentation

- simulateur de jeu d'instructions ou Instruction Set Simulator (ISS)
 - ▶ simule le comportement des instructions d'un processeur donné
 - ▶ simule éventuellement sa microarchitecture (pipeline, caches, etc.)
- plusieurs niveaux de « fidélité temporelle » 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 : interprétation des instructions \Rightarrow très lent

Utilisation d'un ISS : exemple



Sommaire de cette section

3 TLM pour le développement logiciel

- Encapsulation native
- Simulateur de jeux d'instructions (ISS)
- **ISS : Comment ça marche ?**
- Et les OS ?

ISS de base : interprétation instruction par instruction

```
pc = 0;
while (true) {
    ir = read(pc);
    pc += sizeof(ir);
    switch(OPCODE(ir)) {
        case ADD:  regs[OP2(ir)] = regs[OP1(ir)] + regs[OP2(ir)];
                   delay = 1;
                   break;
        case MULT: regs[OP2(ir)] = regs[OP1(ir)] * regs[OP2(ir)];
                   delay = 4;
                   break;
        case JMP:  pc = ...;
                   break;
        ...
        default:   raise(illegal_instruction);
    }
    wait(delay);
}
```

Ne pas oublier la TLB si l'on veut faire tourner un OS : plus compliqué qu'il n'y paraît !

Exemple : ISS du MicroBlaze (TP3)

- MicroBlaze = Xilinx “Softcore”, sur FPGA seulement
- ISS en pur C++ disponible, open-source, \approx 1200 lines of code
 - ▶ fournit une fonction `step()` qui exécute un tour de la boucle
 - ▶ et communique avec l'extérieur grâce à : `getDataRequest()`, `setDataResponse()`, `setIrq()`, ...
- emballé dans un `SC_MODULE` connecté à un `ensitlm` socket & bus.

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

Exemple : ISS du MicroBlaze (TP3)

- MicroBlaze = Xilinx “Softcore”, sur FPGA seulement
 - ISS en pur C++ disponible, open-source, \approx 1200 lines of code
 - ▶ fournit une fonction `step()` qui exécute un tour de la boucle
 - ▶ et communique avec l'extérieur grâce à : `getDataRequest()`, `setDataResponse()`, `setIrq()`, ...
- ⇒ vous **n'avez pas** à le coder
- emballé dans un `SC_MODULE` connecté à un `ensitlm` socket & bus.

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

⇒ vous **avez** à le coder

ISS à base de traduction binaire dynamique

- traducteur cible→bytecode→hôte
- traduit les blocs de base à la volée (\Rightarrow 1 traduction, n exécutions)
- Exemples :
 - ▶ QEMU, open source
connexion à SystemC complexe \Rightarrow support commercial : Antfield/GreenSoCs
 - ▶ SimSoC, projet de recherche, open source, parti d'une feuille blanche
 - ▶ OVP : Open Virtual Platform
solution propriétaire commerciale
 - ▶ et d'autres "ISS rapides" (ARM "fast models")
- plus lent que simulation native, mais parfois plus rapide que le « vrai » circuit

Sommaire de cette section

3 TLM pour le développement logiciel

- Encapsulation native
- Simulateur de jeux d'instructions (ISS)
- ISS : Comment ça marche ?
- Et les OS ?

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
- ▶ ...quoi qu'avec QEMU...

Problème

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

Solution ISS

- ▶ Fonctionne...
- ▶ ...mais très lent
- ▶ ...quoi qu'avec QEMU...

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
 - ≡ notion de paravirtualisation des hyperviseurs et autres micro-kernels

OS Emulation (2/2)

- exemple : Linux

- ▶ portage « SystemC/C++ TLM » des fonctions¹ du noyau
- ▶ compilation du logiciel embarqué pour la machine de simulation
- ▶ résultats :

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

- ▶ effort de portage conséquent, intrusif sur les sources
- ▶ à refaire/maintenir pour chaque version du noyau
- ▶ nécessite les sources : souvent difficile dans un contexte industriel
- ▶ souvent limité à des petits OS maison

1. Lesquelles ? Ou couper ?

Sommaire

- 1 Rappel (?) : Usage des plateformes TLM
- 2 TLM pour la vérification matérielle
- 3 TLM pour le développement logiciel
- 4 TLM pour l'exploration d'architecture : évaluation de performances

Sommaire de cette section

- 4 TLM pour l'exploration d'architecture : évaluation de performances
 - problèmes à résoudre
 - Approches naïves
 - Approche PV+T de STMicro
 - Modèles AT utilisés en pratique

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 fonctionnelle** en temps-réel

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 fonctionnelle** en temps-réel
 - ▶ analyse d'architecture ?

LT/AT/CA : présentation

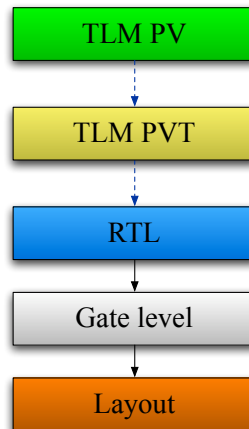
- besoin conflictuels en TLM
- timed/untimed, granularité...

TLM Loosely Timed (Programmer's View (PV) en TLM-1)

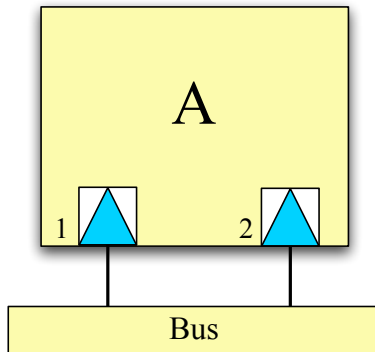
- temps sans signification
- communications gros grain
- utilisation : développement du logiciel embarqué, intégration système

TLM Approximately Timed (AT) (Programmer's View with Time (PVT) en TLM-1)

- temps **précis** induits par la microarchitecture
- communications à la taille du bus
- utilisation : évaluation d'architecture

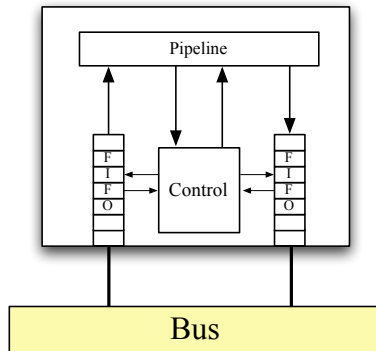


LT/AT : modèle de microarchitecture



- ① granularité
- ② fonctionnalité de microarchitecture (fifos, pipeline...)
- ③ durées des traitements

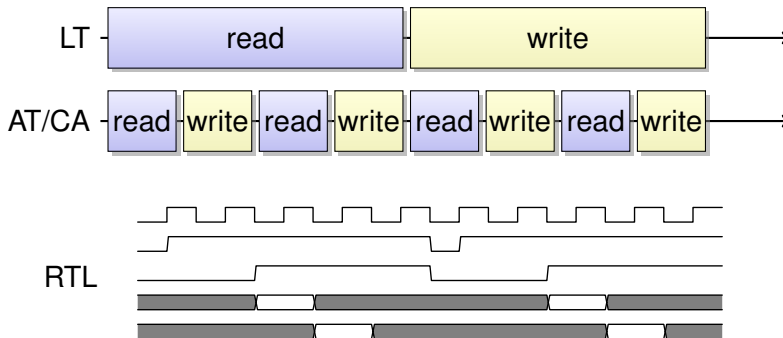
LT/AT : modèle de microarchitecture



- 1 granularité
- 2 fonctionnalité de microarchitecture (fifos, pipeline...)
- 3 durées des traitements

LT/AT/CA : exemple de traces

- exemple transfert mémoire :

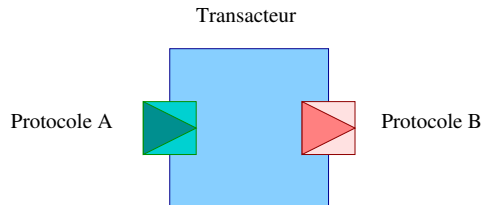


Sommaire de cette section

- 4 TLM pour l'exploration d'architecture : évaluation de performances
 - problèmes à résoudre
 - **Approches naïves**
 - Approche PV+T de STMicro
 - Modèles AT utilisés en pratique

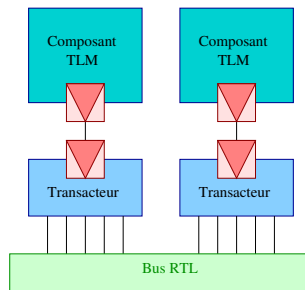
Transacteurs

- transacteur = composant « pont » entre deux protocoles

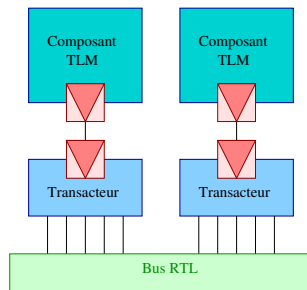


- toutes combinaisons de a et b (tlm, rtl, différents protocoles)

Plateforme « timée » avec des transacteurs



Plateforme « timée » avec des transacteurs



Question



Où est la limitation ?

Plateforme « timée » 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
```

Plateforme « timée » 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 ?

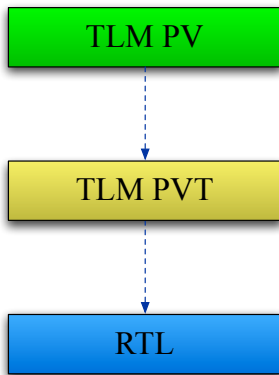
Conclusion intermédiaire

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

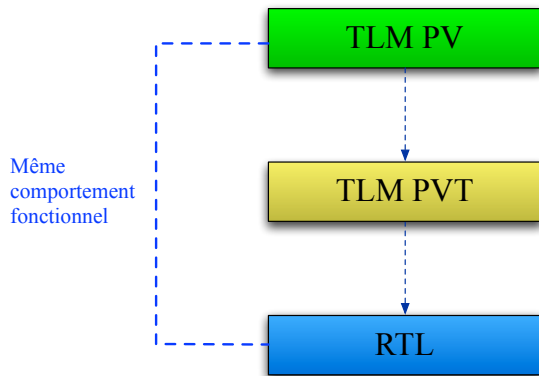
Sommaire de cette section

- 4 TLM pour l'exploration d'architecture : évaluation de performances
 - problèmes à résoudre
 - Approches naïves
 - **Approche PV+T de STMicro**
 - Modèles AT utilisés en pratique

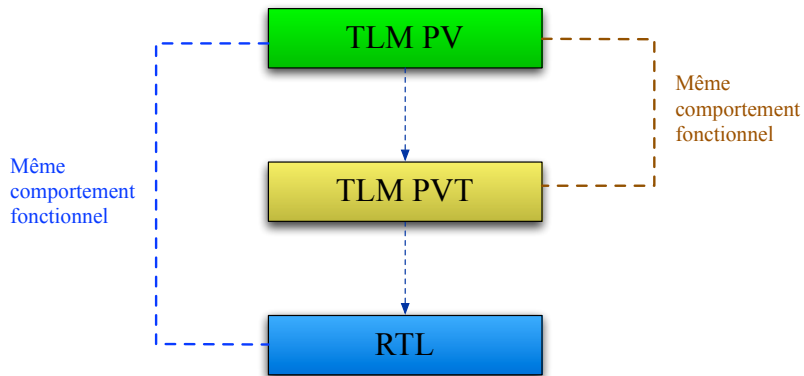
PV+T : contraintes



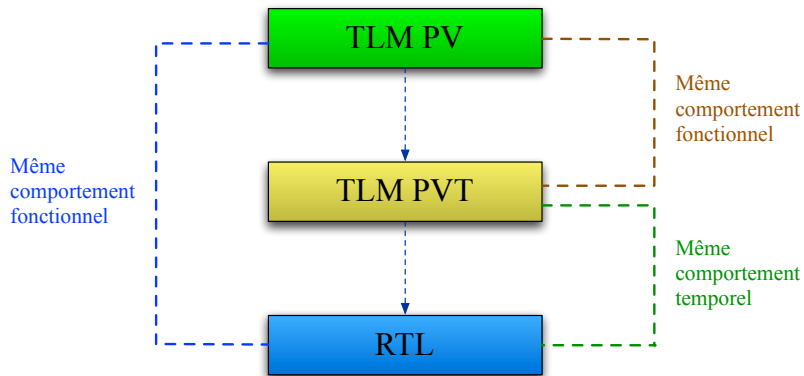
PV+T : contraintes



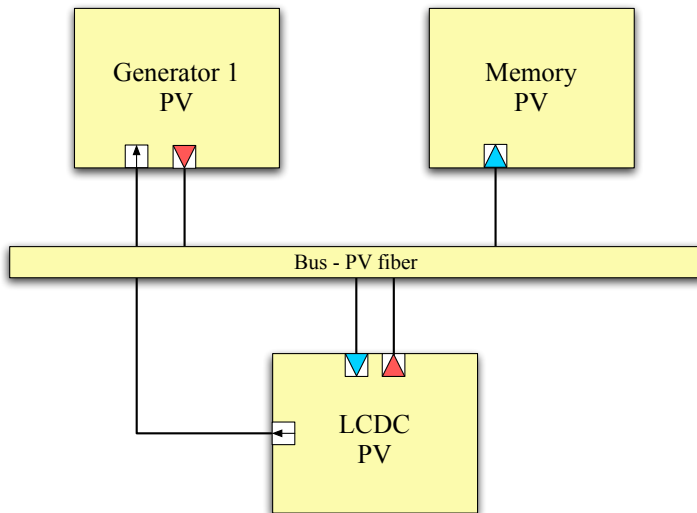
PV+T : contraintes



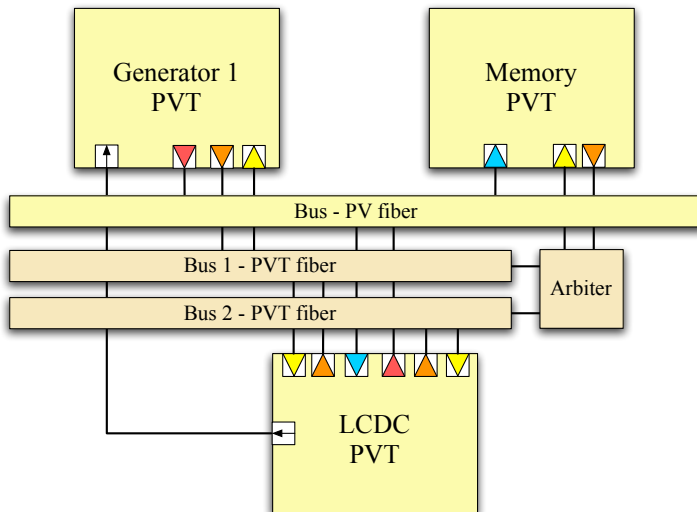
PV+T : contraintes



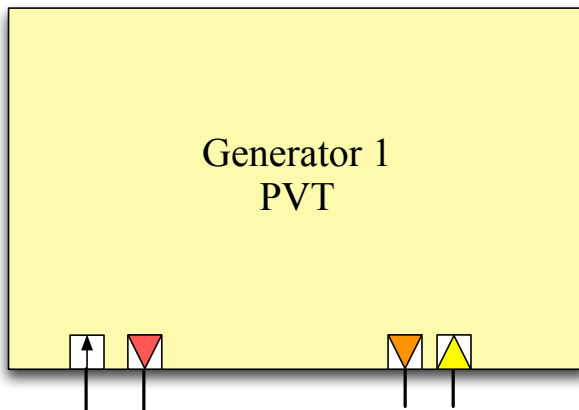
Exemple (PV \approx LT)



Exemple (PVT \approx AT)

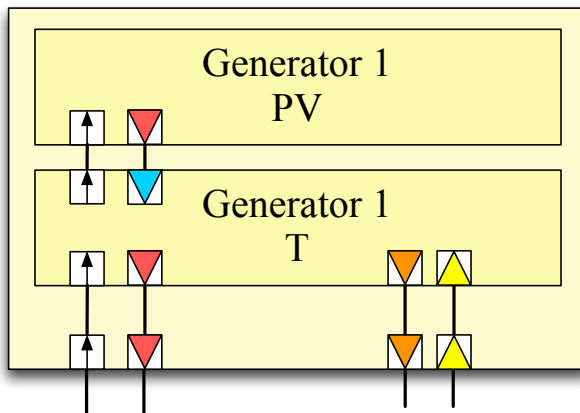


Module PVT



- Generator 1 PVT

Module PVT



- Generator 1 PVT

Sommaire de cette section

- 4 TLM pour l'exploration d'architecture : évaluation de performances
 - problèmes à résoudre
 - Approches naïves
 - Approche PV+T de STMicro
 - Modèles AT utilisés en pratique

Modèles AT utilisés en pratique

- rien n'est parfait en ce bas monde
 - estimation précise du temps impose
 - ▶ effort de modélisation important
 - ▶ temps de simulation élevé
- ⇒ rapport $\frac{\text{coût}}{\text{bénéfice}}$ discutable
- approches RTL souvent préférées (+ co-simulation/co-emulation/...)

Différents niveaux de précision temporelle

- Cycle accurate, Bit Accurate (CABA)
- Approximately-timed (AT \approx PVT) :
tentative de s'approcher d'une évaluation correcte du temps
- Loosely-timed (LT \approx PV) :
vise la capacité de modéliser le temps, pas qu'il soit précis
permet l'utilisation de timer par exemple
- Purely untimed :
execution totalement asynchrone, pas de notion de temps