Une (brève) introduction à l'infrastructure de simulation gem5

Arthur Perais (arthur.perais@univ-grenoble-alpes.fr), CNRS

ARCHI 2025 – Sète – 11 mars 2025

TIMA – CNRS – UGA/Grenoble INP

Infrastructure de simulation

- **Préambule : Quelques bouts de slides sont empruntés à « The gem5 Tutorial @ISCA 2024 » par Jason Lowe Power**

Infrastructure de simulation

- **Préambule : Quelques bouts de slides sont empruntés à « The gem5 Tutorial @ISCA 2024 » par Jason Lowe Power**

- **La complexité des puces modernes (CPU, GPUs, TPUs, NPUs, etc.) rend le coût d'un prototype matériel prohibitif**
  - Complexité => Il faut un système **complet** qui marche **correctement**
    - Même si on est intéressé par une petite partie du design
  - Temps => Fondre une puce prend plusieurs semaines
    - Problématique pour le debug, itérer sur un design
  - Coût => Fondre une puce coûte des (milliers) d'€ pour chaque prototype

Infrastructure de simulation

- **Préambule : Quelques bouts de slides sont empruntés à « The gem5 Tutorial @ISCA 2024 » par Jason Lowe Power**

- **La complexité des puces modernes (CPU, GPUs, TPUs, NPUs, etc.) rend le coût d'un prototype matériel prohibitif**
  - Complexité => Il faut un système **complet** qui marche **correctement**
    - Même si on est intéressé par une petite partie du design
  - Temps => Fondre une puce prend plusieurs semaines
    - Problématique pour le debug, itérer sur un design
  - Coût => Fondre une puce coûte des (milliers) d'€ pour chaque prototype

- **Il est donc nécessaire de prototyper virtuellement**
  - Via une infrastructure de simulation logicielle
    - Plus ou moins abstraite, plus ou moins rapide

Quelle infrastructure de simulation ?

- **Quel type de matériel ?**
  - Numérique vs. numérique/analogique
  - Bloc (CPU) vs. système entier

Quelle infrastructure de simulation ?

- **Quel type de matériel ?**
  - Numérique vs. numérique/analogique
  - Bloc (CPU) vs. système entier

- **Quel type d'étude, à quelle étape de la conception se place-t-on ?**
  - Architecture (ISA)
  - Microarchitecture

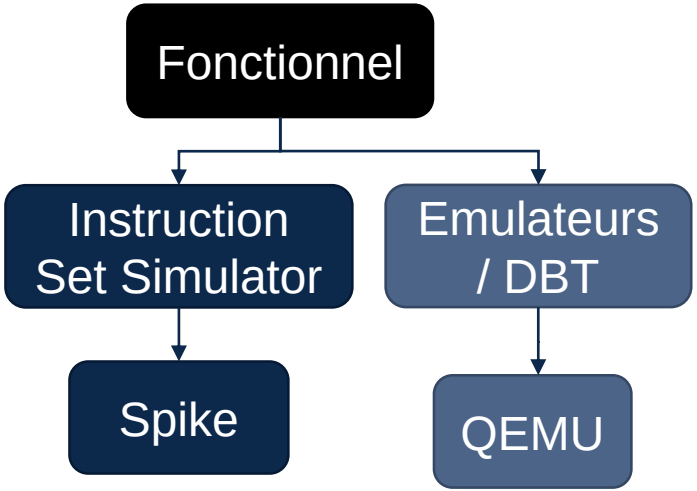
Quelle infrastructure de simulation ?

- **Quel type de matériel ?**
  - Numérique vs. numérique/analogique
  - Bloc (CPU) vs. système entier

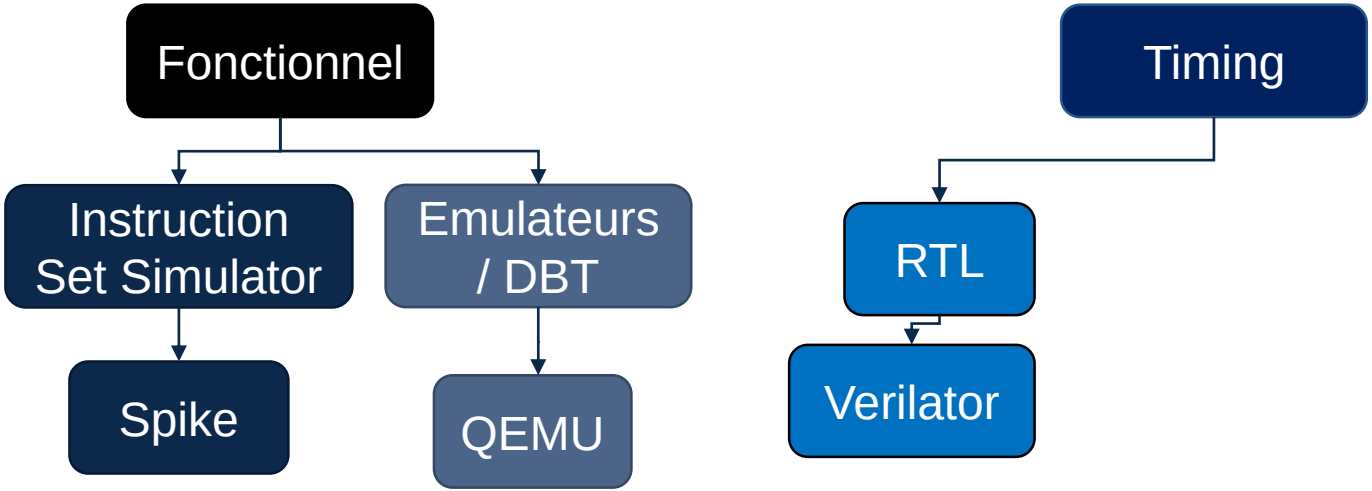
- **Quel type d'étude, à quelle étape de la conception se place-t-on ?**
  - Architecture (ISA)
  - Microarchitecture

- **Quel degré de précision / quelle rapidité ?**
  - Modèle « first order » => est-il nécessaire de simuler la machine
    - e.g., Loi d'Amdahl pour estimer la performance multicoeurs
  - En général, imprécision = rapidité de simulation
  - « cycle level » vs « cycle accurate »

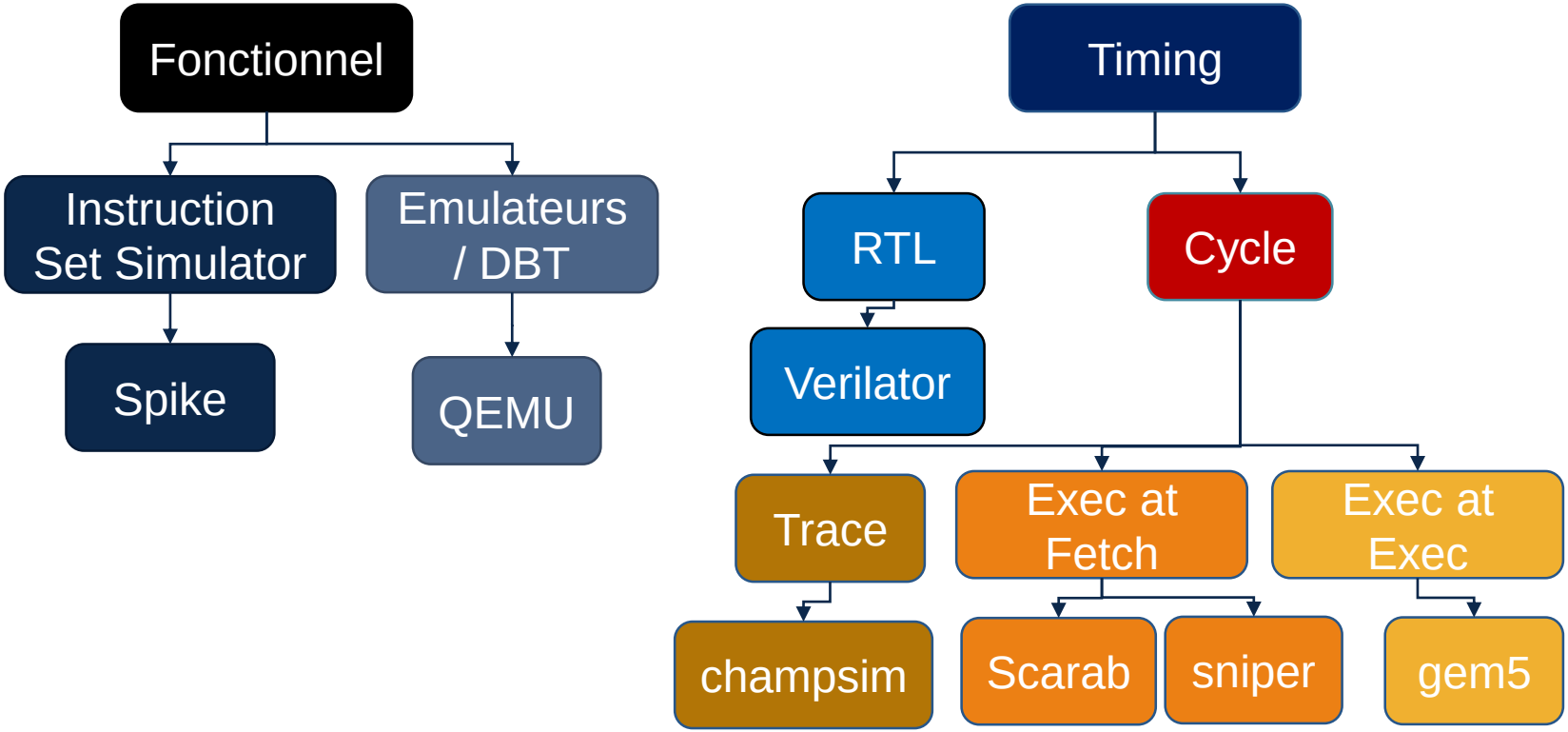
Quelle infrastructure de simulation (CPU) ?



Quelle infrastructure de simulation (CPU) ?



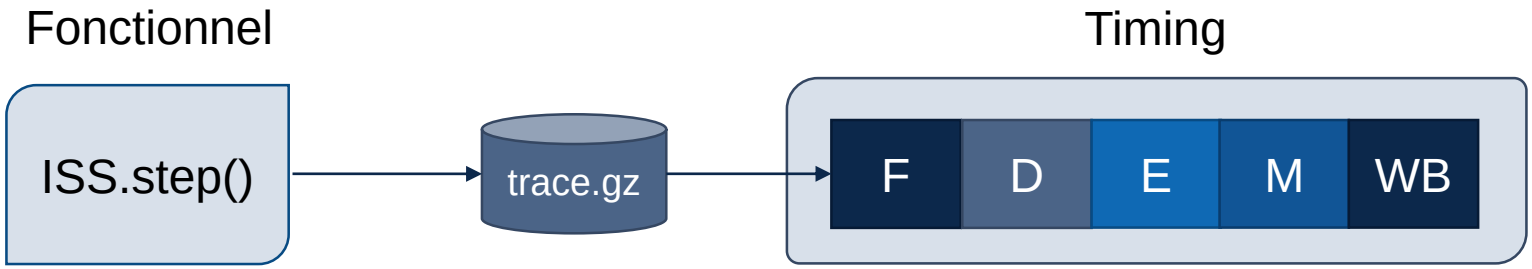
Quelle infrastructure de simulation (CPU) ?



Organisation(s) d’un simulateur niveau cycle

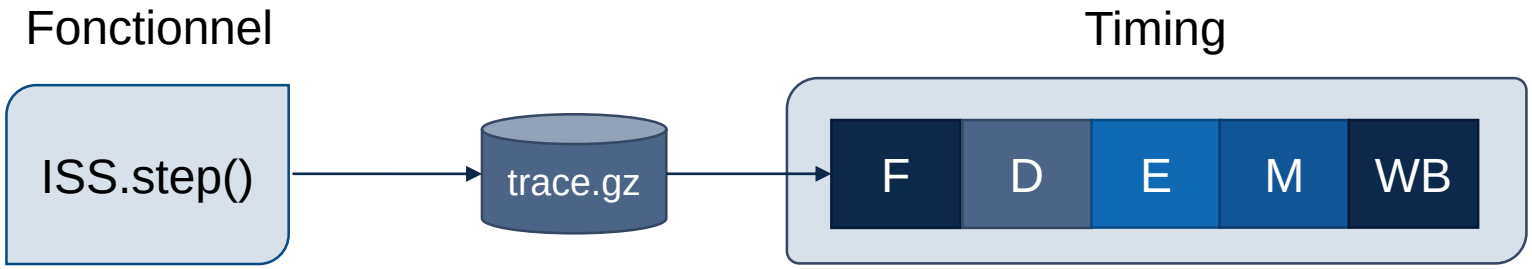
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs basés sur des traces (e.g., ChampSim)**



Trois « paradigmes » de modélisation cycle-level

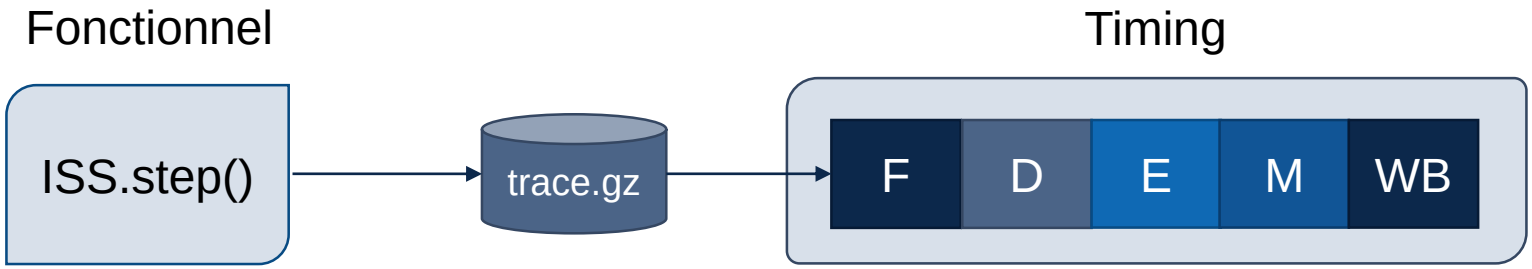
- **Simulateurs basés sur des traces (e.g., ChampSim)**



- **Coût de l'exécution fonctionnelle payé en amont, une seule fois (collection de la trace via un ISS)**
  - Rapide

Trois « paradigmes » de modélisation cycle-level

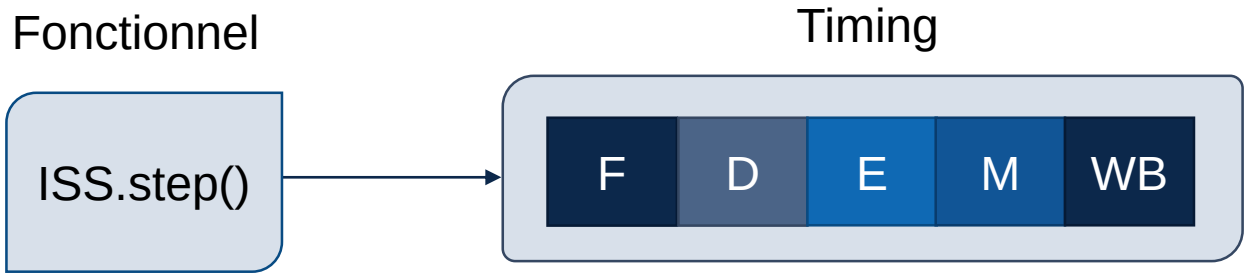
- **Simulateurs basés sur des traces (e.g., ChampSim)**



- **Coût de l'exécution fonctionnelle payé en amont, une seule fois (collection de la trace via un ISS)**
  - Rapide
- **Accès uniquement aux informations qui sont dans la trace**
  - Traces vite gigantesques si on capture les valeurs des registres et de la mémoire

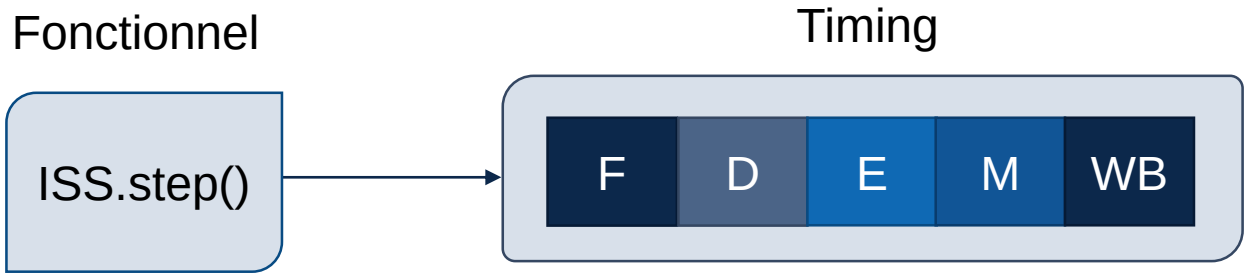
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « execute at fetch » (e.g., Scarab, Sniper)**



Trois « paradigmes » de modélisation cycle-level

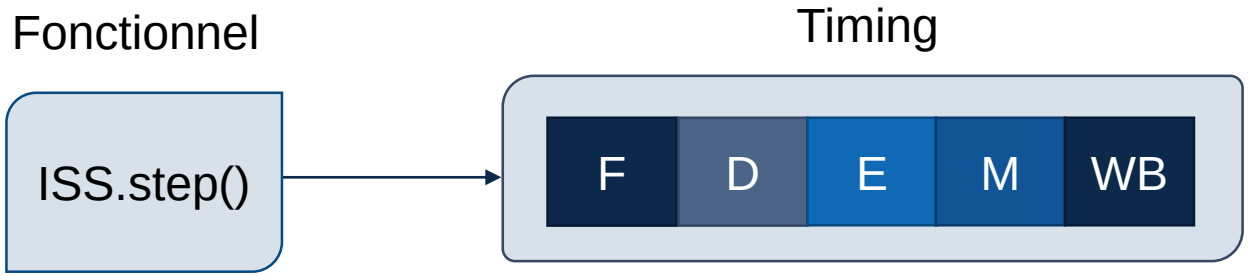
- **Simulateurs « execute at fetch » (e.g., Scarab, Sniper)**



- **Coût de l'exécution fonctionnelle payé pendant la simulation timing**
  - Plus lent

Trois « paradigmes » de modélisation cycle-level

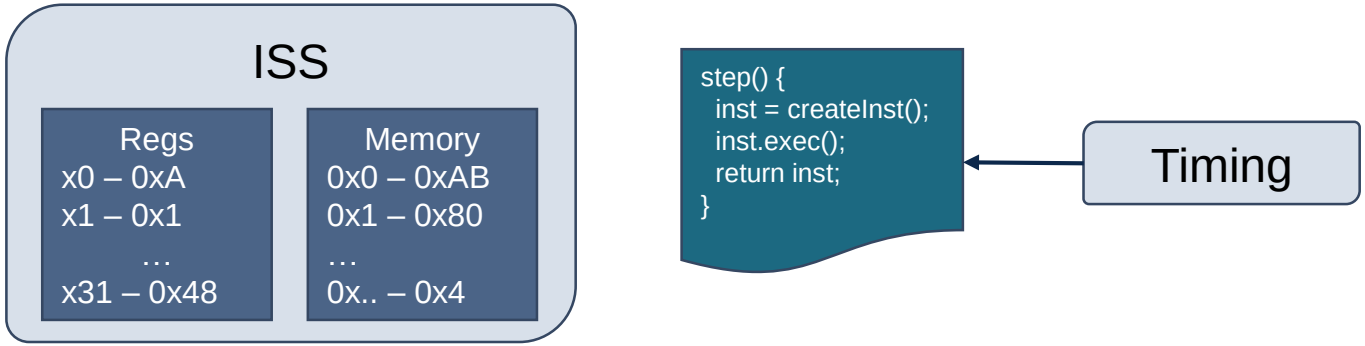
- **Simulateurs « execute at fetch » (e.g., Scarab, Sniper)**



- **Coût de l'exécution fonctionnelle payé pendant la simulation timing**
  - Plus lent
- **Accès à l'état architectural de la machine (si l'ISS le permet)**
  - Par exemple l'état de la mémoire, des registres

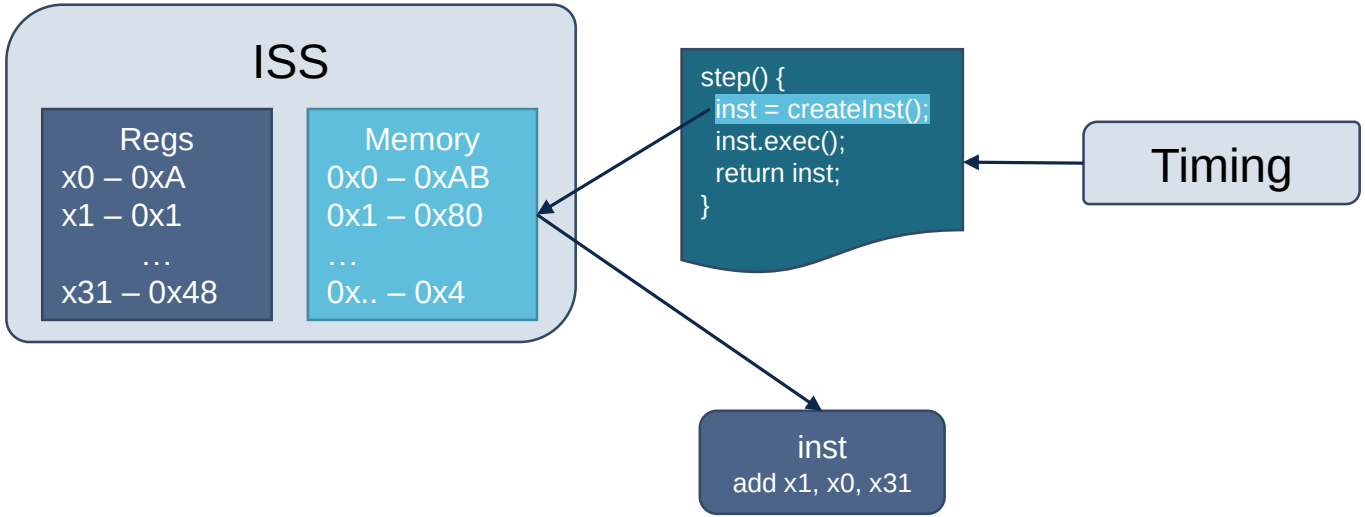
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « execute at fetch » (e.g., Scarab, Sniper)**



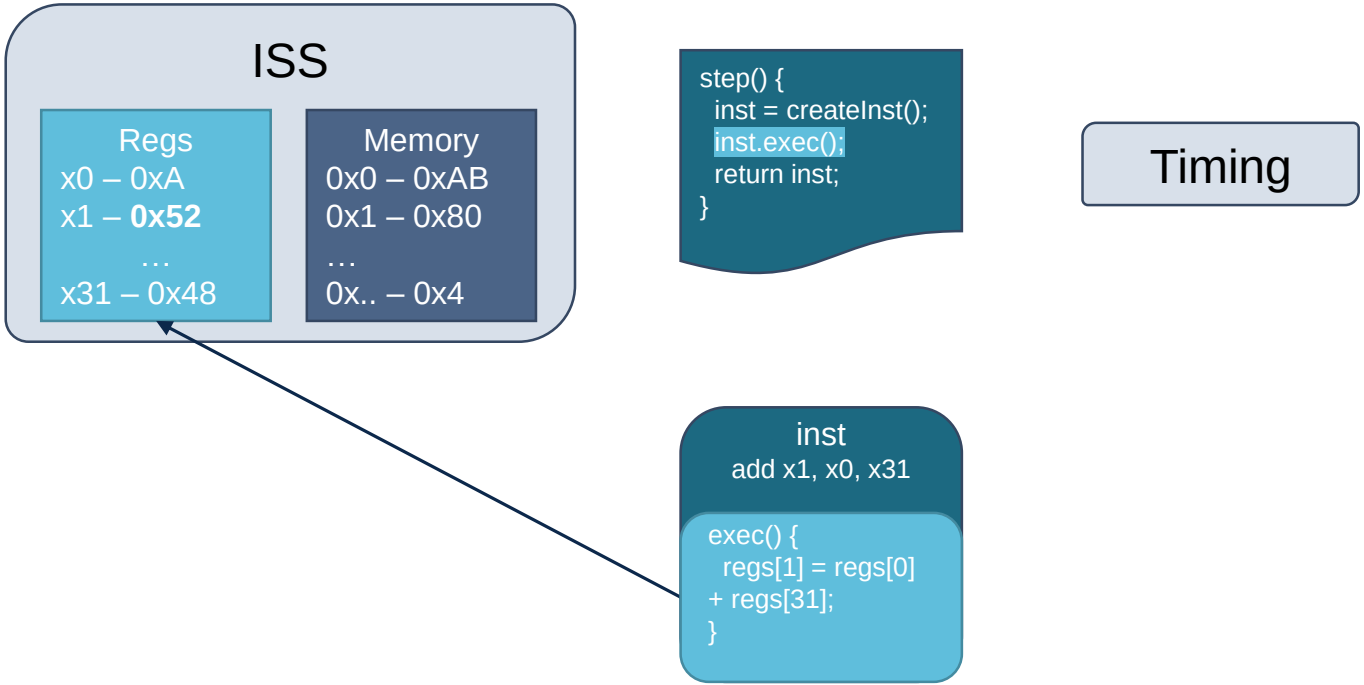
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « execute at fetch » (e.g., Scarab, Sniper)**



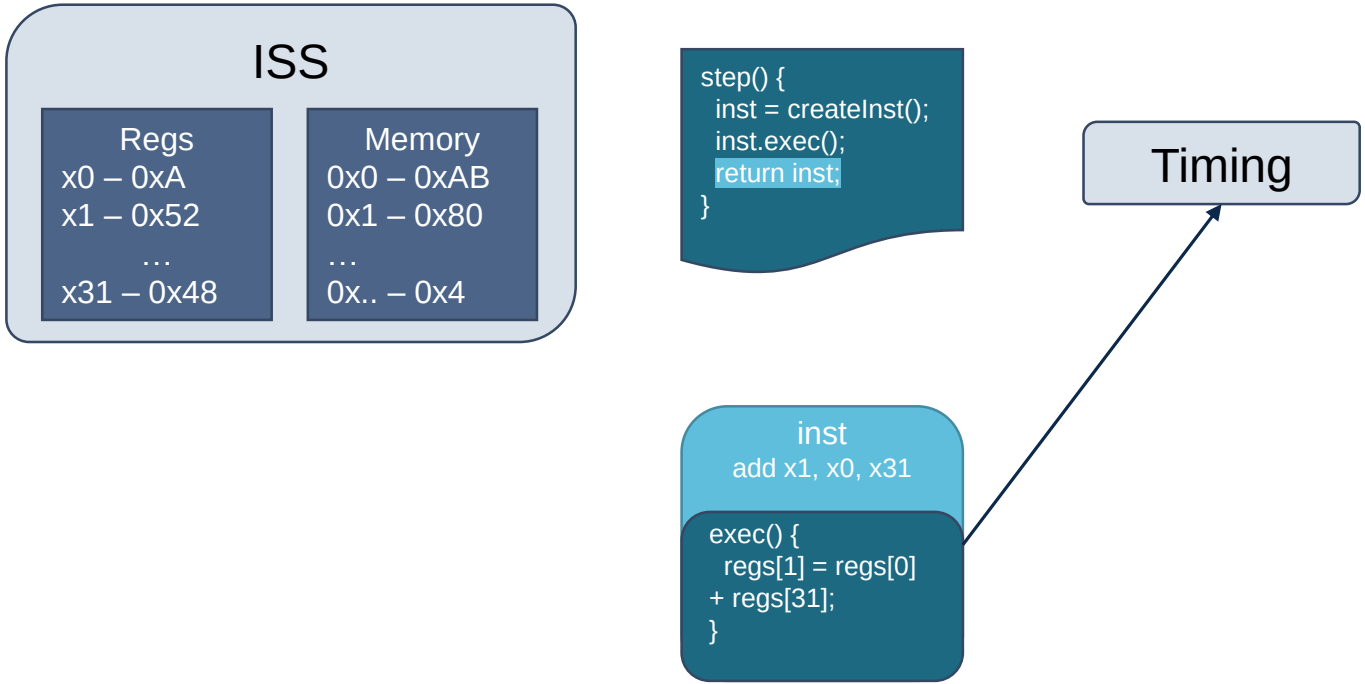
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « execute at fetch » (e.g., Scarab, Sniper)**



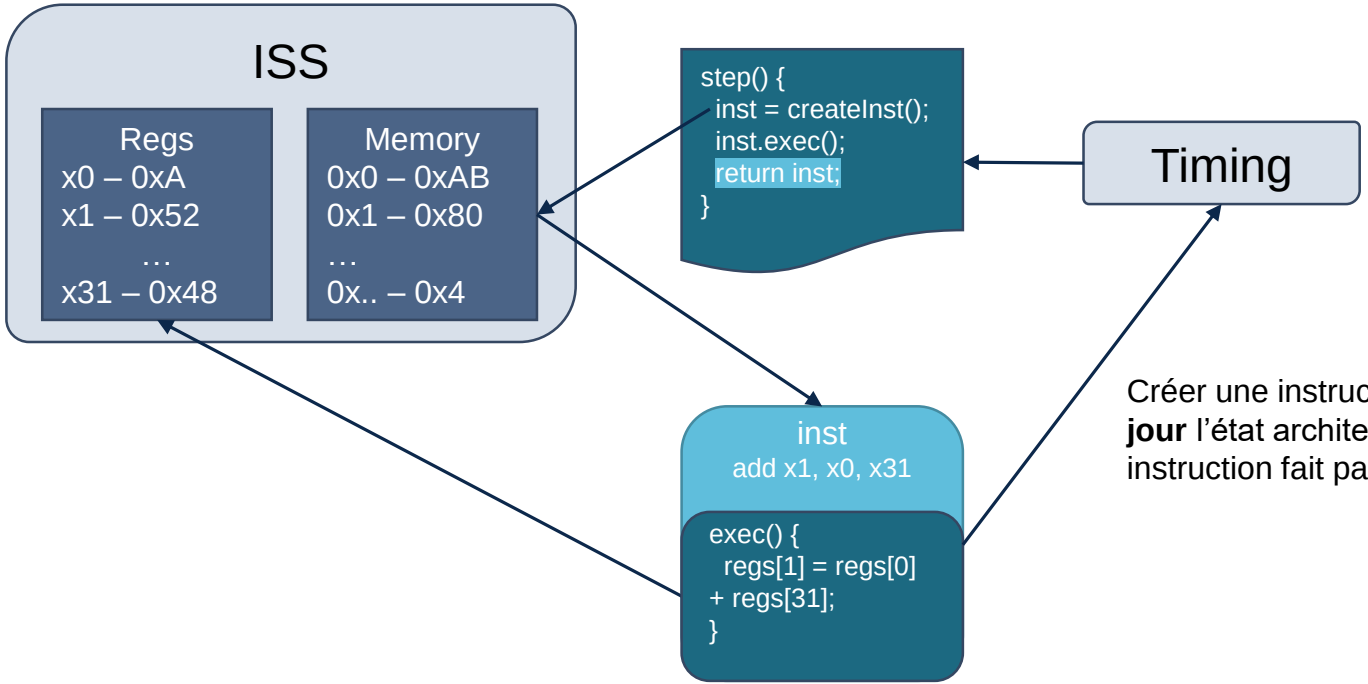
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « execute at fetch » (e.g., Scarab, Sniper)**



Trois « paradigmes » de modélisation cycle-level

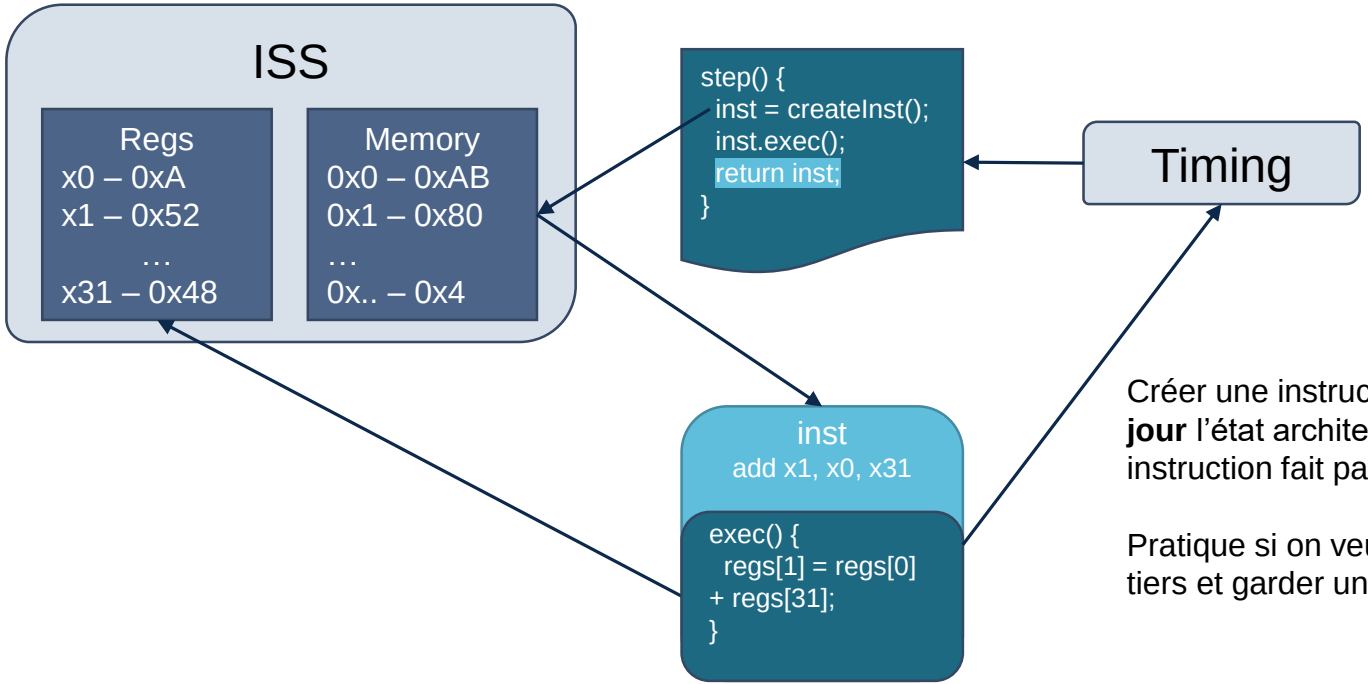
- **Simulateurs « execute at fetch » (e.g., Scarab, Sniper)**



Créer une instruction à simuler **met à jour** l'état architectural => cette instruction fait partie du programme

Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « execute at fetch » (e.g., Scarab, Sniper)**

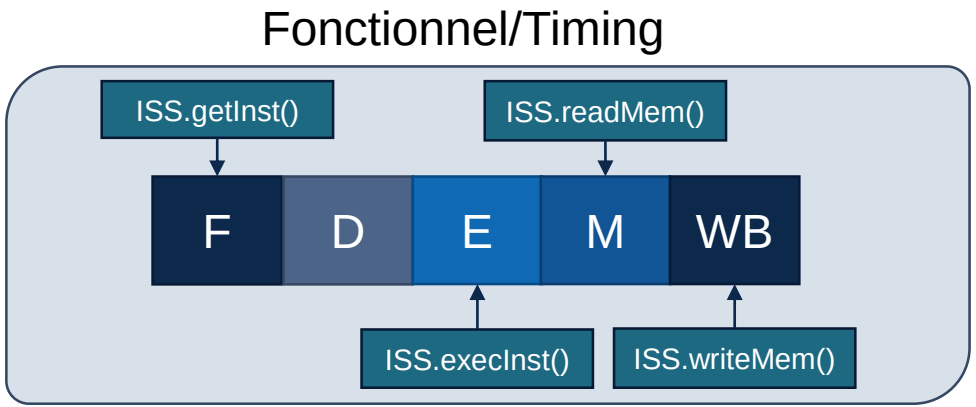


Créer une instruction à simuler **met à jour** l'état architectural => cette instruction fait partie du programme

Pratique si on veut utiliser un ISS tiers et garder une interface resserée

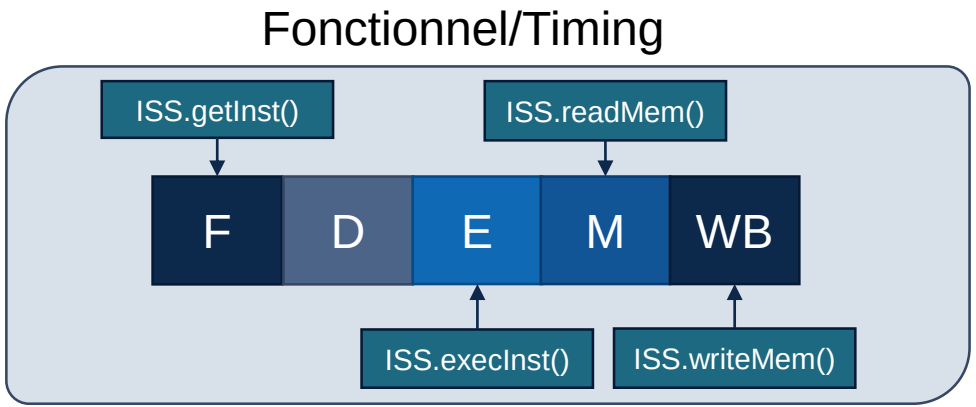
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « execute at execute » (e.g., gem5)**



Trois « paradigmes » de modélisation cycle-level

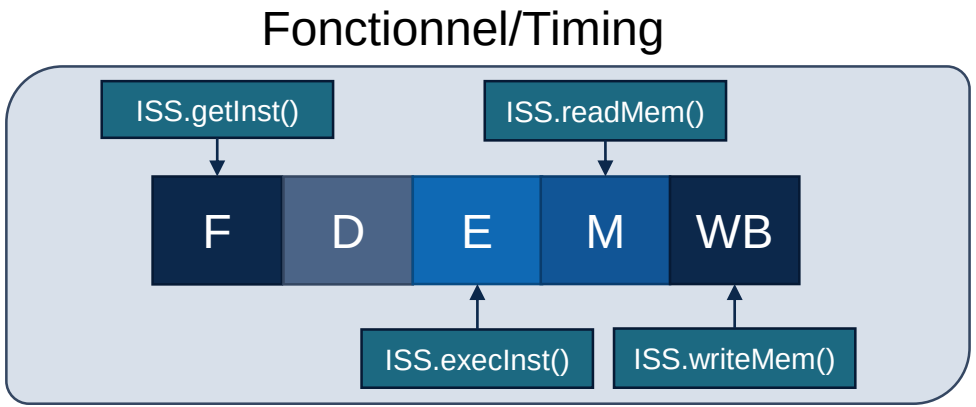
- **Simulateurs « execute at execute » (e.g., gem5)**



- **Exécution fonctionnelle spéculative**
  - Encore plus lent

Trois « paradigmes » de modélisation cycle-level

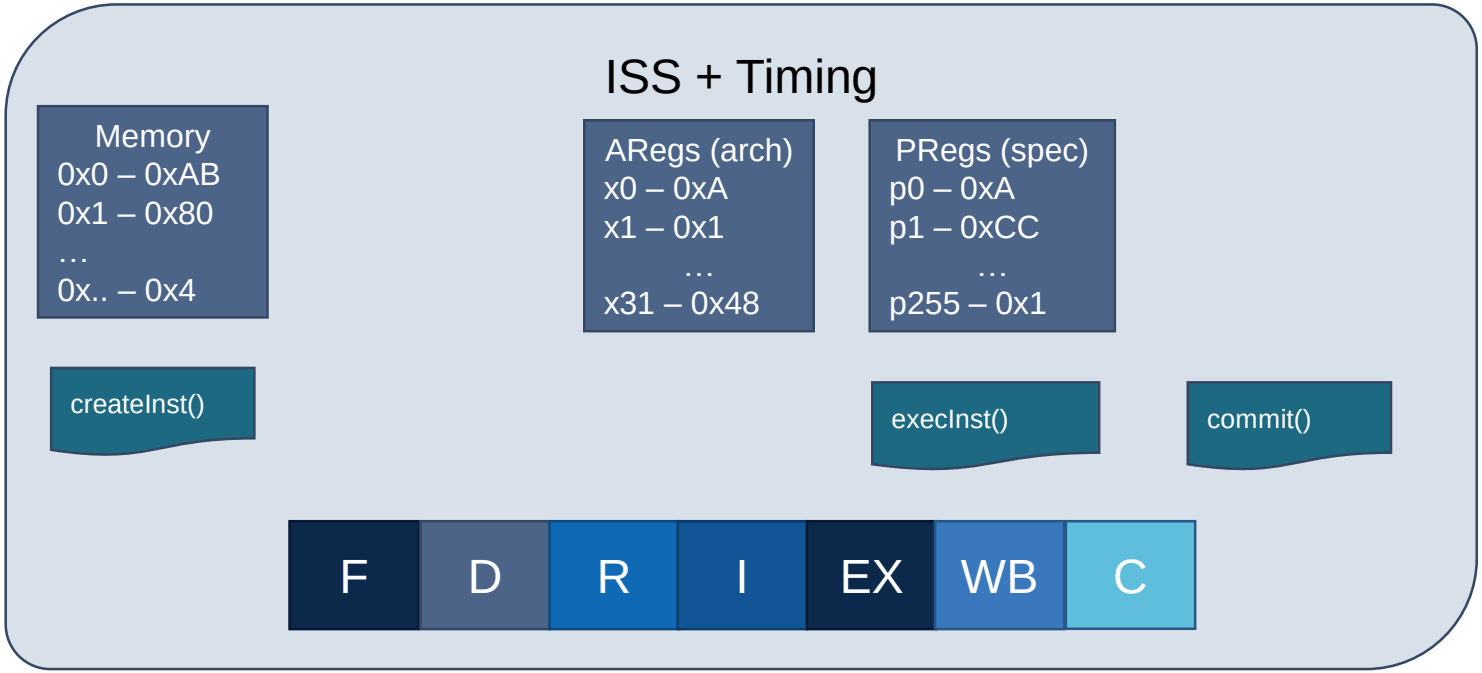
- **Simulateurs « execute at execute » (e.g., gem5)**



- **Exécution fonctionnelle spéculative**
  - Encore plus lent
- **Accès à l'état architectural et microarchitectural (!) de la machine**
  - Par exemple le contenu des registres physiques si on modélise le renommage de registres

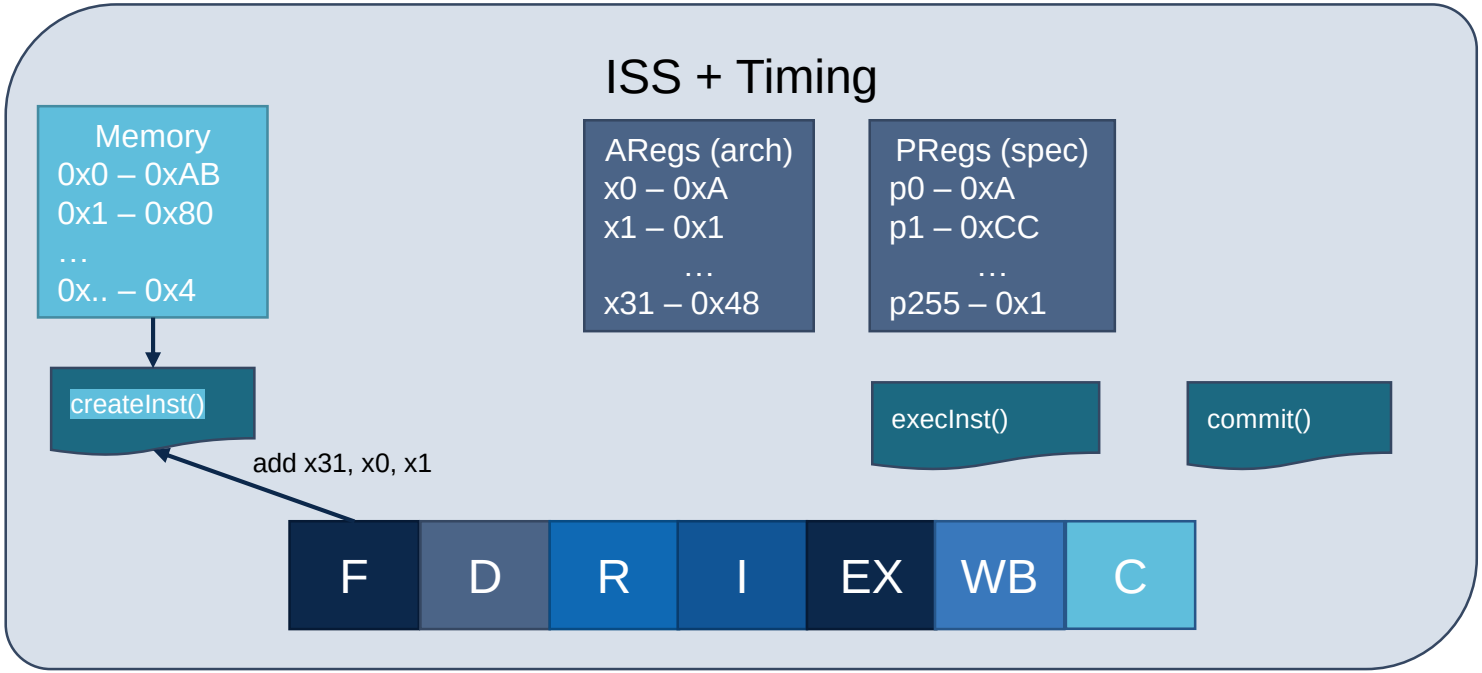
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « Le modèle timing est l'ISS » (e.g., gem5)**



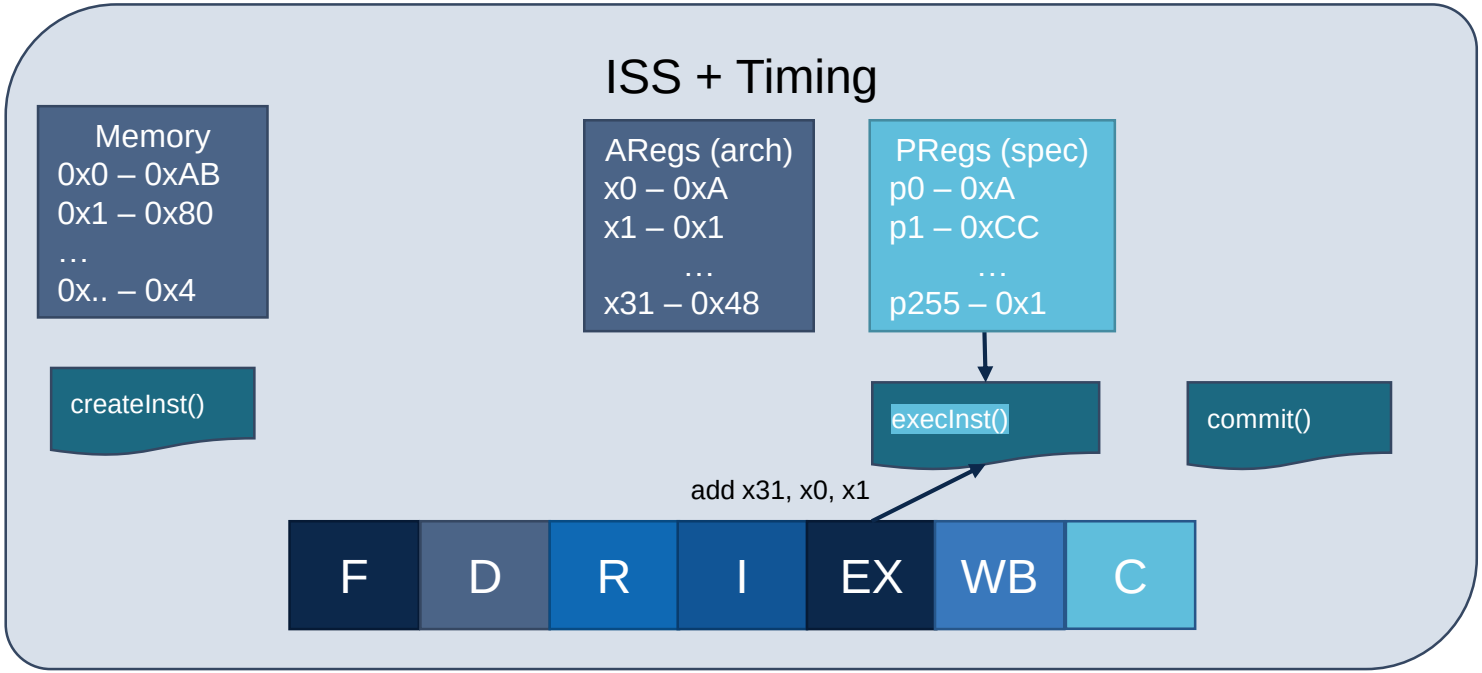
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « Le modèle timing est l'ISS » (e.g., gem5)**



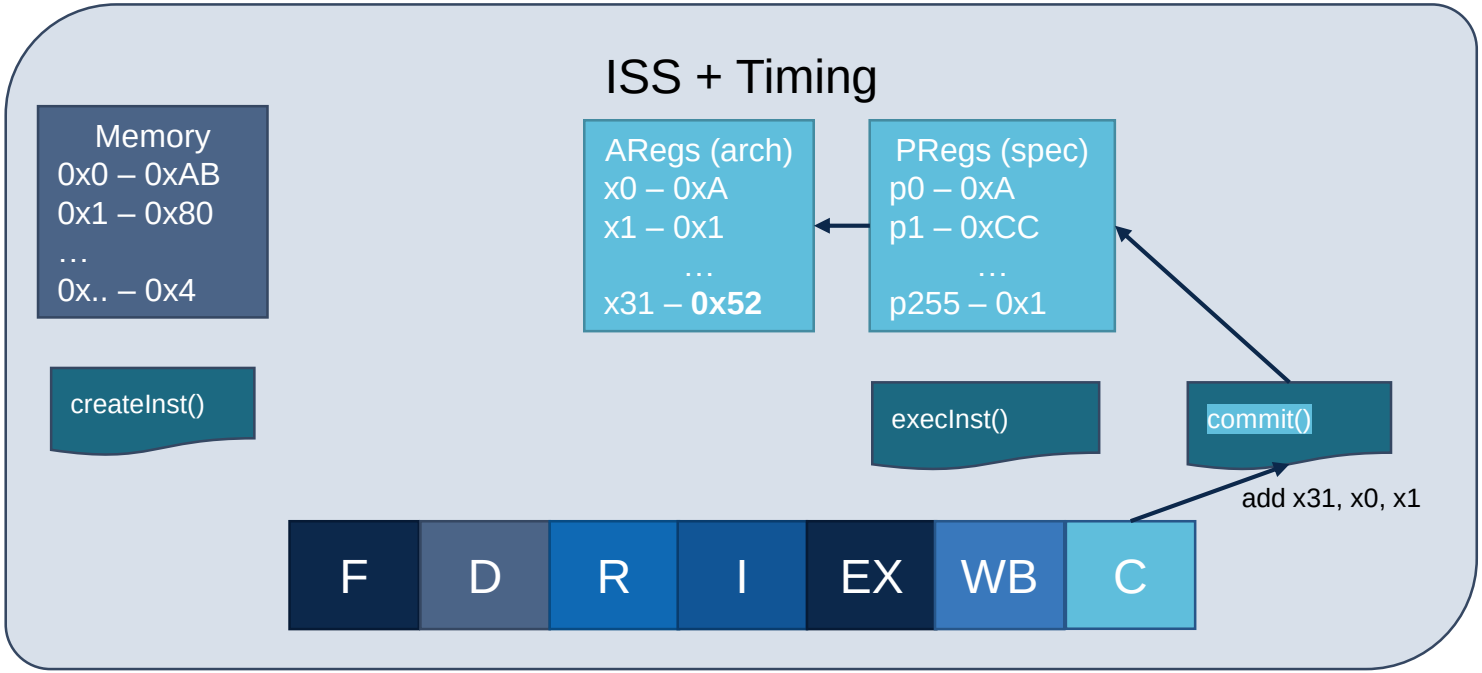
Trois « paradigmes » de modélisation cycle-level

- **Simulateurs « Le modèle timing est l'ISS » (e.g., gem5)**



Trois « paradigmes » de modélisation cycle-level

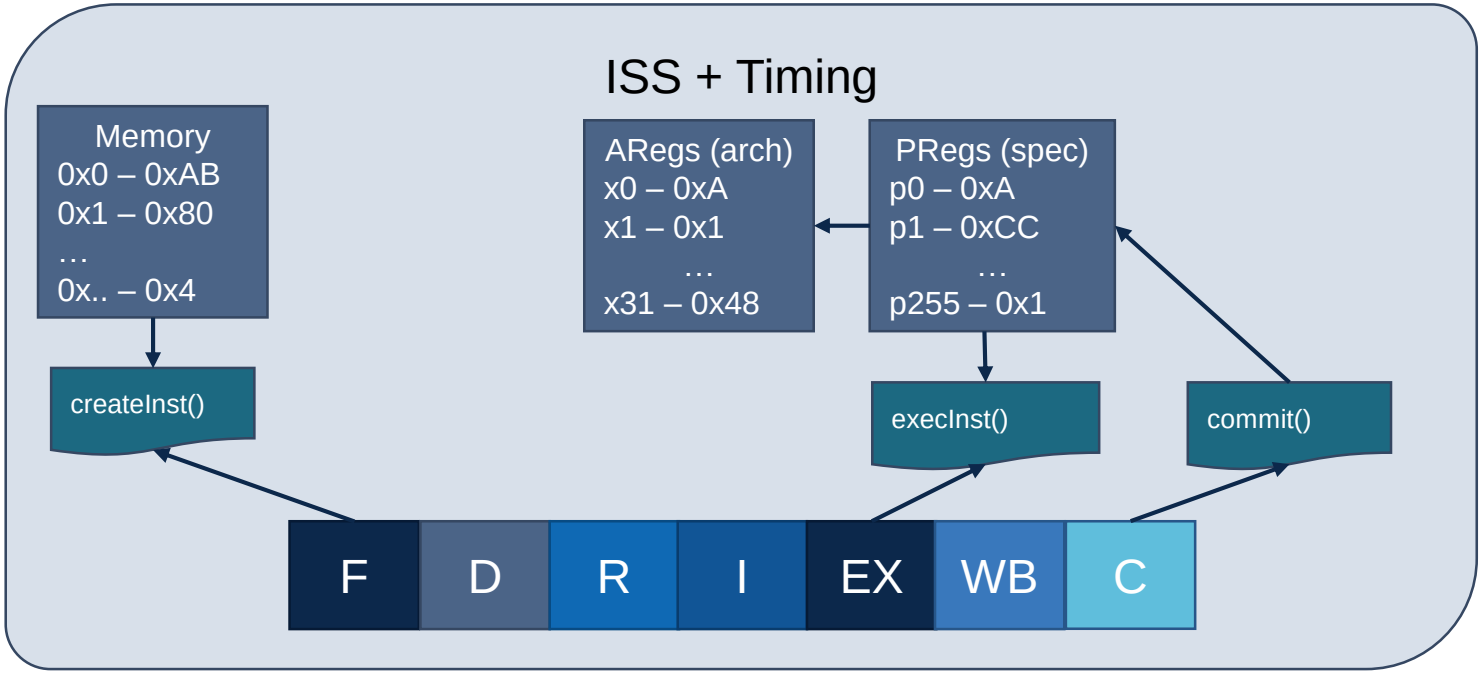
- **Simulateurs « Le modèle timing est l'ISS » (e.g., gem5)**



Trois « paradigmes » de modélisation cycle-level

Créer une instruction à simuler **ne met pas** à jour l'état architectural => cette instruction ne fait peut-être pas partie du programme

- **Simulateurs « Le modèle timing est l'ISS » (e.g., gem5)**



Trois « paradigmes » de modélisation cycle-level

- **Trace ou « execute at fetch » implique que la prochaine instruction est toujours « la bonne »**
  - Plus difficile de modéliser l'effet de la spéculation (prédiction de branchement) dans le processeur (surtout avec des traces)

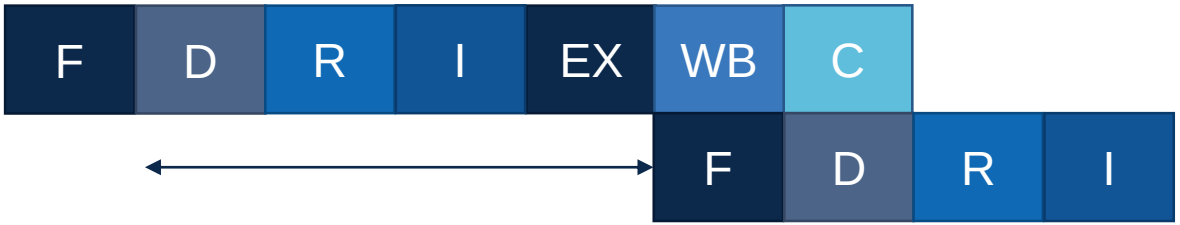
Trace

Modèle CPU prédit « pris », donc devrait faire rentrer l'instruction @#label dans le pipeline, mais l'instruction n'est pas dans la trace (car branchement non pris lors de l'exécution)

Trois « paradigmes » de modélisation cycle-level

- **Trace ou « execute at fetch » implique que la prochaine instruction est toujours « la bonne »**
  - Plus difficile de modéliser l'effet de la spéculation (prédiction de branchement) dans le processeur (surtout avec des traces)

Trace

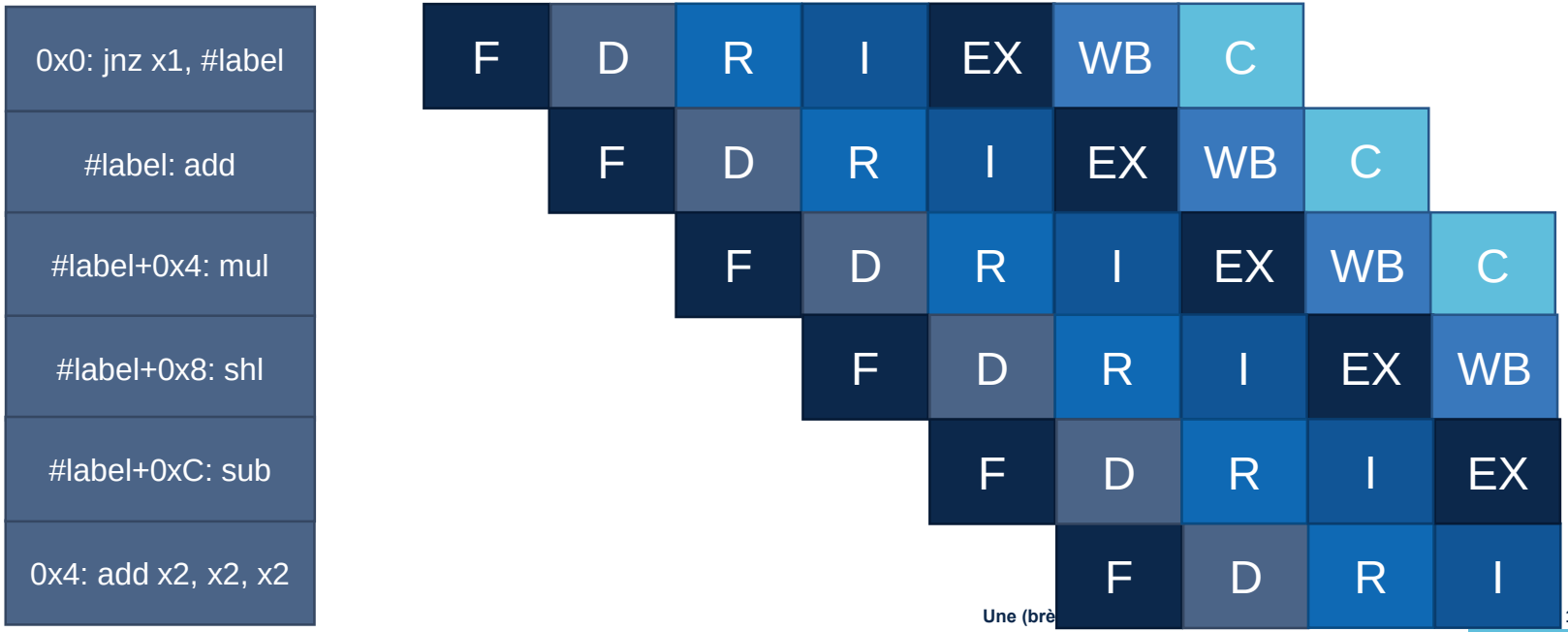


Instruction sur le mauvais chemin non simulée, peut avoir des effets sur les caches, TLBs…

Trois « paradigmes » de modélisation cycle-level

- **Trace ou « execute at fetch » implique que la prochaine instruction est toujours « la bonne »**
  - Plus difficile de modéliser l'effet de la spéculation (prédiction de branchement) dans le processeur (surtout avec des traces)

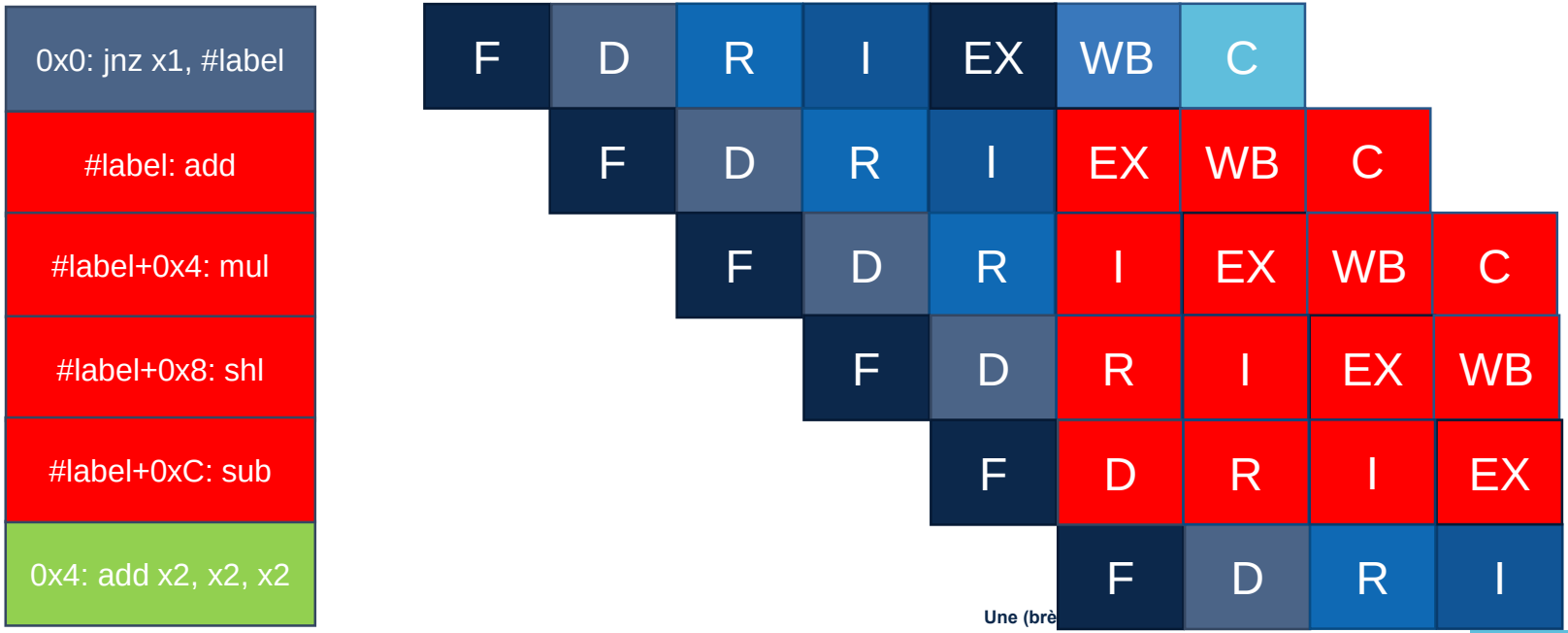
exec-at-exec



Trois « paradigmes » de modélisation cycle-level

- **Trace ou « execute at fetch » implique que la prochaine instruction est toujours « la bonne »**
  - Plus difficile de modéliser l'effet de la spéculation (prédiction de branchement) dans le processeur (surtout avec des traces)

exec-at-exec



Trois « paradigmes » de modélisation cycle-level

- **Trace ou « execute at fetch » implique que la prochaine instruction est toujours « la bonne »**
  - Plus difficile de modéliser l'effet de la spéculation (prédiction de branchement) dans le processeur (surtout avec des traces)
    - Pas fait : ChampSim (trace), Sniper (EAF)
    - Fait : Scarab (EAF, mais à ses limites)

Trois « paradigmes » de modélisation cycle-level

- **Trace ou « execute at fetch » implique que la prochaine instruction est toujours « la bonne »**
  - Plus difficile de modéliser l'effet de la spéculation (prédiction de branchement) dans le processeur (surtout avec des traces)
    - Pas fait : ChampSim (trace), Sniper (EAF)
    - Fait : Scarab (EAF, mais à ses limites)
  - Plus difficile de modéliser le contenu des registres et de la mémoire avec une trace (traces énorme si on veut la valeur des registres)
    - Etude sur compression de cache, de registres, prédiction de valeur plus difficile

Trois « paradigmes » de modélisation cycle-level

- **Trace ou « execute at fetch » implique que la prochaine instruction est toujours « la bonne »**
  - Plus difficile de modéliser l'effet de la spéculation (prédiction de branchement) dans le processeur (surtout avec des traces)
    - Pas fait : ChampSim (trace), Sniper (EAF)
    - Fait : Scarab (EAF, mais à ses limites)
  - Plus difficile de modéliser le contenu des registres et de la mémoire avec une trace (traces énorme si on veut la valeur des registres)
    - Etude sur compression de cache, de registres, prédiction de valeur plus difficile

- **Mais simulation plus rapide que « execute at execute »**
  - Trace plus rapide que « execute at fetch »

L’infrastructure de simulation gem5

gem5 : Rapide histoire

- **m5 : « A tool for simulating systems »**
  - Crée à l'université du Michigan par Steve Reinhardt et Nathan Binkert
- **GEMS : « Detailed memory system »**
  - Crée à l'université du Wisconsin à Madison « par des étudiants de Mark Hill et David Wood »
- **gem5 = Michigan m5 + Wisconsin GEMS**
  - Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. **2011.** The gem5 simulator. SIGARCH Comput. Archit. News 39, 2 (August 2011), 1-7

gem5 et moi

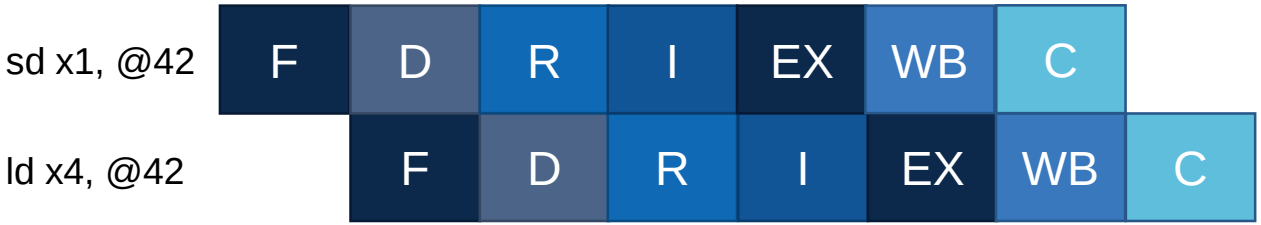
- **Thèse en microarchitecture en 2015**
  - Simulation de CPUs généralistes sur gem5 (~4/5 ans)
- **Interlude industrie : Pas de gem5**
- **TIMA depuis 2020**
  - Retour à gem5 (~4/5 ans)
- **Presque dix ans passé sur gem5**
  - Tous les jours une surprise (pas toujours bonne…)

Pourquoi gem5 ?

- **Libre + support de la communauté => Dynamique**
  - Ajout régulier de fonctionnalités
    - Support RISC-V, ARMv8 (par ARM)
    - Support modèle GPU (par Amd)
  - Mais beaucoup de code qui fonctionne mais dont l'expertise a disparue et n'est plus maintenu
    - Support x86 (SSE2 max)
    - Modèle CPU out-of-order (certaines parties du modèle sont obsolètes)

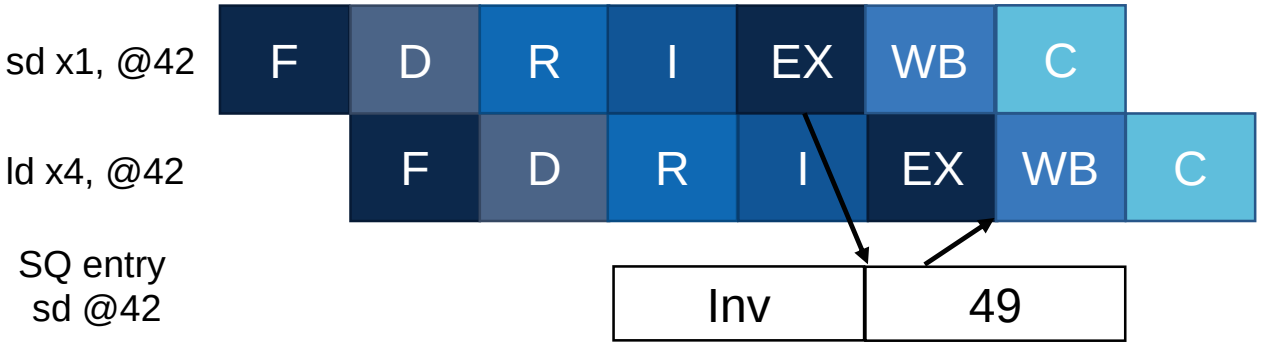
Pourquoi gem5 ?

- **gem5 est execute-at-execute (EAE)**
  - Couple la modélisation d'une fonctionnalité avec le temps
    - Par ex., load lit une donnée au cycle où l'instruction le ferait dans le pipeline



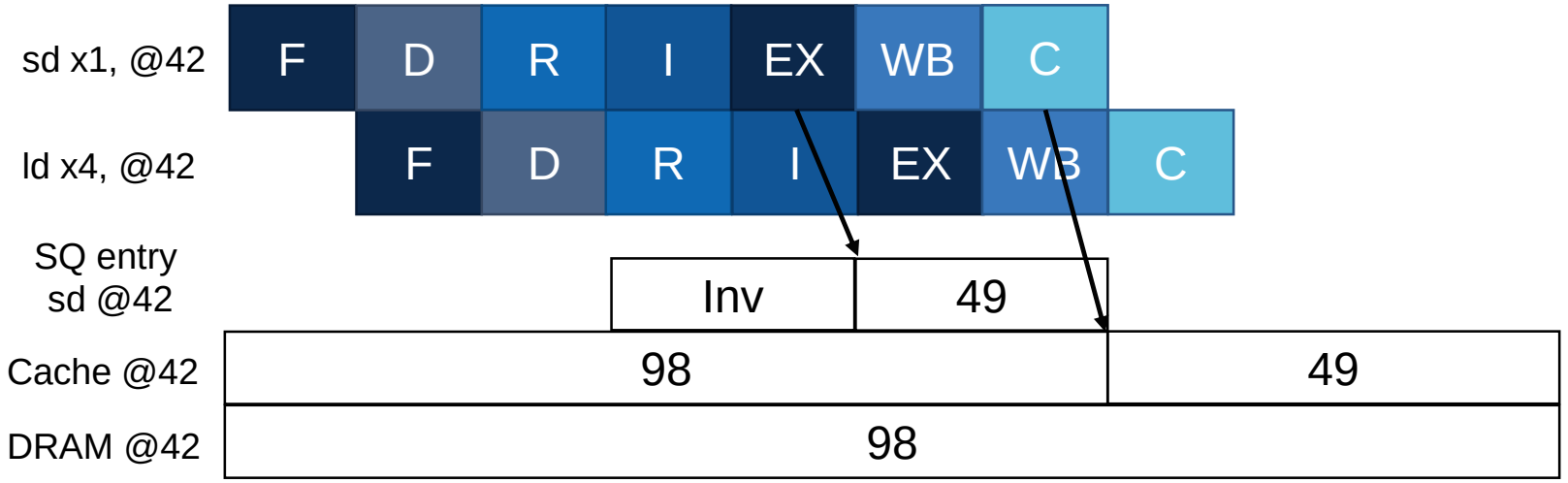
Pourquoi gem5 ?

- **gem5 est execute-at-execute (EAE)**
  - Couple la modélisation d'une fonctionnalité avec le temps
    - Par ex., load lit une donnée au cycle où l'instruction le ferait dans le pipeline



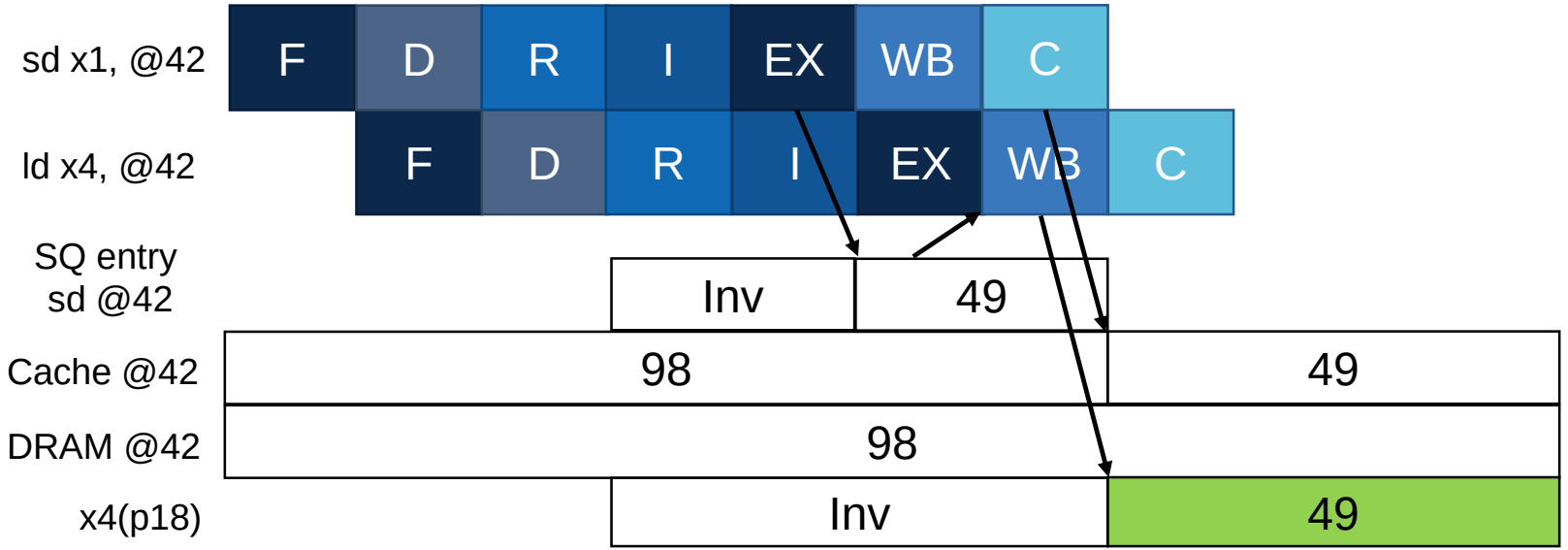
Pourquoi gem5 ?

- **gem5 est execute-at-execute (EAE)**
  - Couple la modélisation d'une fonctionnalité avec le temps
    - Par ex., load lit une donnée au cycle où l'instruction le ferait dans le pipeline



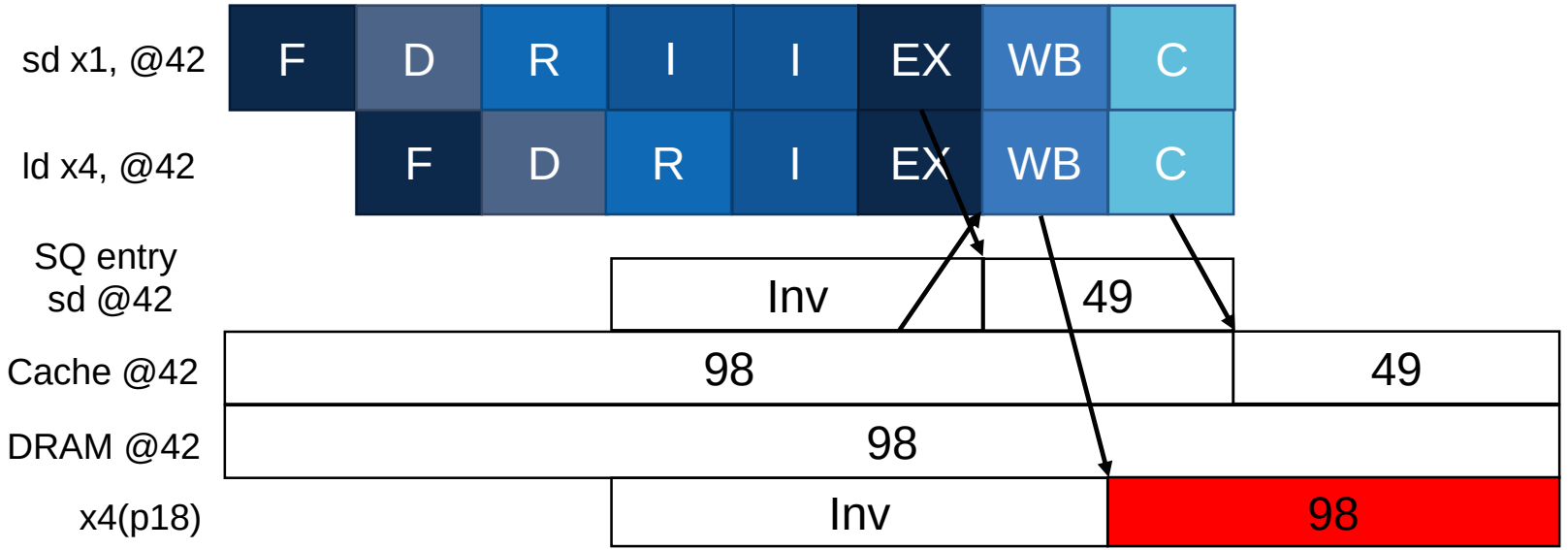
Pourquoi gem5 ?

- **gem5 est execute-at-execute (EAE)**
  - Couple la modélisation d'une fonctionnalité avec le temps
    - Par ex., load lit une donnée au cycle où l'instruction le ferait dans le pipeline



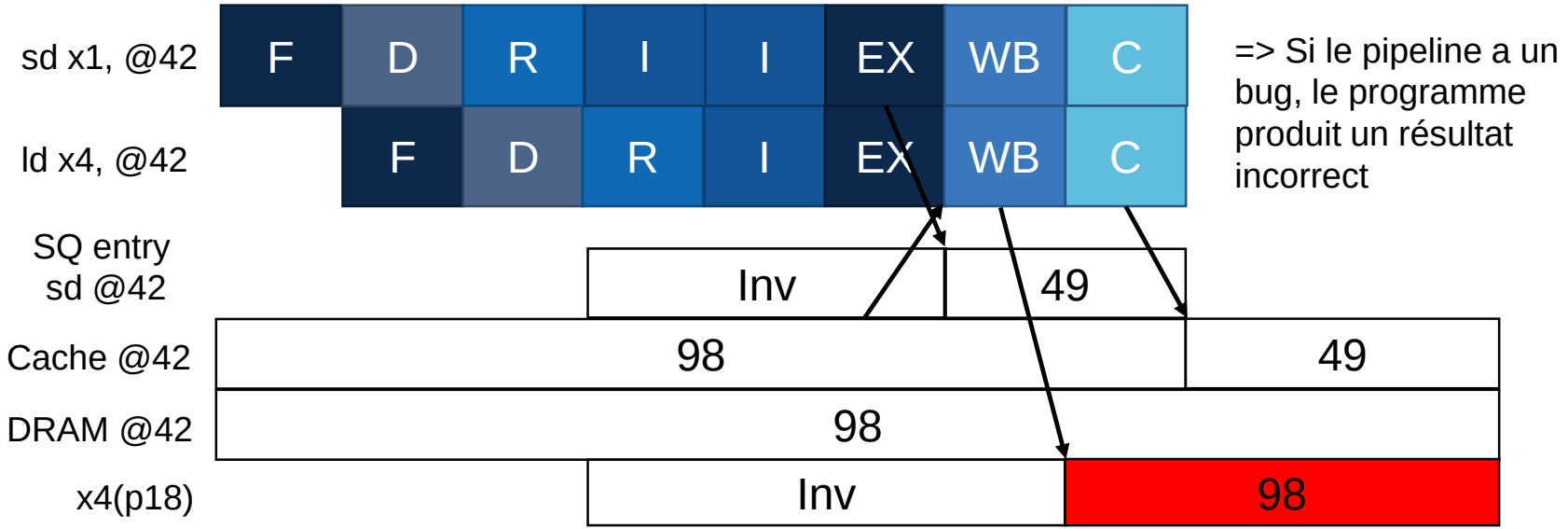
Pourquoi gem5 ?

- **gem5 est execute-at-execute (EAE)**
  - Couple la modélisation d'une fonctionnalité avec le temps
    - Par ex., load lit une donnée au cycle ou l'instruction le ferait dans le pipeline



Pourquoi gem5 ?

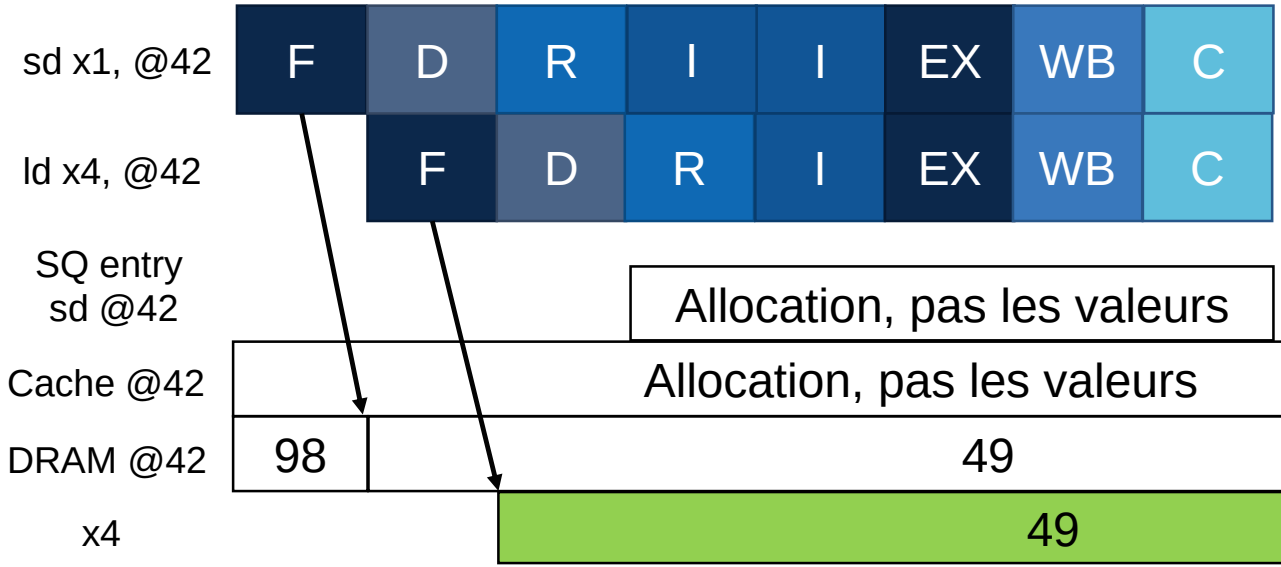
- **gem5 est execute-at-execute (EAE)**
  - Couple la modélisation d'une fonctionnalité avec le temps
    - Par ex., load lit une donnée au cycle ou l'instruction le ferait dans le pipeline



=> Si le pipeline a un bug, le programme produit un résultat incorrect

Pourquoi gem5 ?

- **gem5 est execute-at-execute (EAE)**
  - Execute-at-fetch/trace découple la fonctionnalité du temps
    - Load lit la mémoire au fetch de façon atomique, timing modélisé en amont



=> Si le pipeline a un bug, le programme continue **correctement** sans rien dire

Pourquoi gem5 ?

- **gem5 est execute-at-execute (EAE)**
  - EAE couple la modélisation d'une fonctionnalité avec le temps
    - Etat architectural programme généré par le pipeline simulé **=> Pipeline incorrect = programme incorrect**
  - Execute-at-fetch/trace découple la fonctionnalité du temps
    - Etat architectural programme généré par un ISS/trace devant le pipeline simulé => **Pipeline incorrect = programme correct**

Pourquoi gem5 ?

- **Polyvalence côté OS**
  - Peut booter un Linux complet. Modélisation :
    - Appels systèmes
    - Mémoire virtuelle
    - IO
  - Peut aussi faire tourner un programme et émuler les appels systèmes
    - Simulation bien plus rapide si l'activité OS ne nous intéresse pas
    - Mais ne supporte pas tous les appels systèmes

- **Configurable et inter-opérable**
  - Simulation d'un système complet (cœurs, caches cohérents, mémoire, accélérateurs e.g. modèle GPU supporté par Amd)
  - Tutoriels pour intégration avec simulateurs/outils dédiés de manière plus ou moins officielle (Ramulator, McPAT)

Pourquoi (pas) gem5 ?

- **Bugs majeurs connus « officieusement » qui mettent du temps à être résolus**
  - Comportement de la prédiction de branchement incorrect depuis ~2018 (bug de performance)
  - Quelques autres bugs de performance moins critiques

*Workshop on Duplicating, Deconstructing and Debunking (WDDD) @ ISCA'14

Pourquoi (pas) gem5 ?

- **Bugs majeurs connus « officieusement » qui mettent du temps à être résolus**
  - Comportement de la prédiction de branchement incorrect depuis ~2018 (bug de performance)
  - Quelques autres bugs de performance moins critiques

- **« On se protège quand on utilise une tronçonneuse donc protégez vous quand vous utilisez gem5 » (David Wood, 2014)***
  - Faire un tour sur le repo git (Issues et Discussions) + Mailing List
  - Utiliser le système de logging pour vérifier que tout se passe au moment où on l'attend
  - Comprendre quelles parties du design sont très importantes vs. peu importantes et vérifier le comportement des parties importantes
  - Ne pas faire confiance aveuglément aux variables servant à configurer les composants

*Workshop on Duplicating, Deconstructing and Debunking (WDDD) @ ISCA'14

Pourquoi (pas) gem5 ? Résumé

- **Paradigme execute-at-execute puissant**
  - Spéculation, résultat incorrect si mauvaise modélisation
- **Support communautaire**
  - Nombreuses fonctionnalités et modèles
  - Modèles modernes vs. obsolètes
- **Support baremetal, émulation OS et OS complet**
- **Lent (max quelques MIPS pour 1 CPU OoO + système mémoire)**
  - QEMU émule à 1 GIPS (x100-x1000)

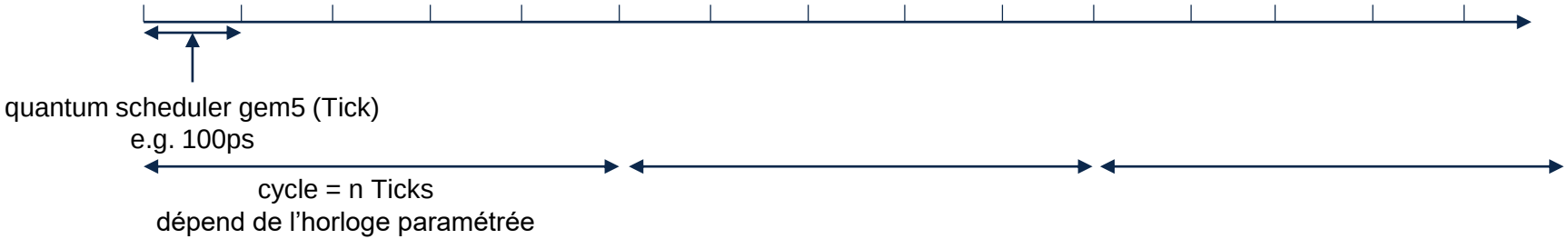
gem5 : Concepts

- **Simulateur évènementiel**

quantum scheduler gem5 (Tick)
e.g. 100ps

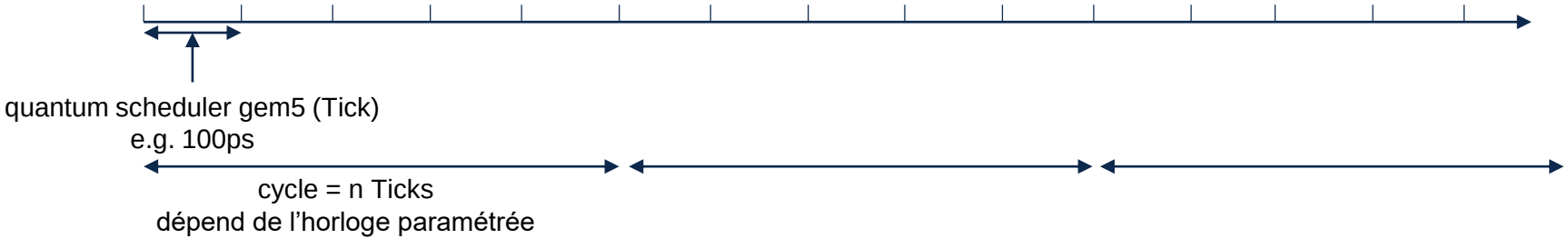
gem5 : Concepts

- **Simulateur évènementiel**



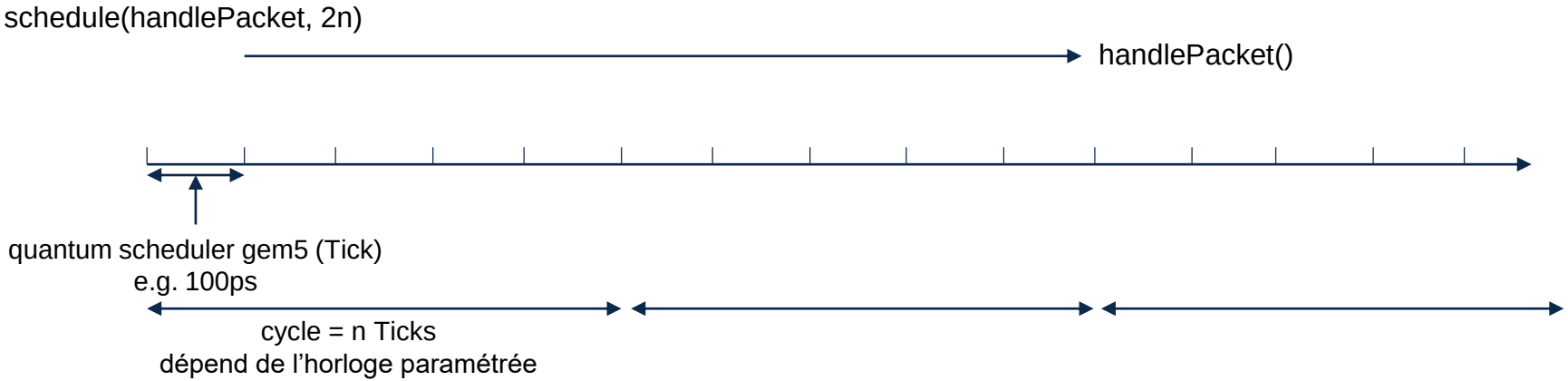
gem5 : Concepts

- **Simulateur évènementiel**
  - On ordonnance des évènements (fonctions) qui arriveront dans le **futur**



gem5 : Concepts

- **Simulateur évènementiel**
  - On ordonnance des évènements (fonctions) qui arriveront dans le **futur**
    - Par exemple, un paquet arrivera au contrôleur mémoire dans 2 cycles, donc appel de handlePacket() dans deux cycles



gem5 : Concepts

- **Simulateur évènementiel**
  - On ordonnance des évènements (fonctions) qui arriveront dans le **futur**
    - Par exemple, un paquet arrivera au contrôleur mémoire dans 2 cycles, donc appel de handlePacket() dans deux cycles

- **Par opposition à une boucle qui simule chaque cycle**
  - Plus efficace pour les modèles qui n'ont pas quelque chose à faire à chaque cycle (système mémoire)
  - Moins efficace pour les modèles qui ont (presque) tout le temps quelque chose à faire (CPU)

gem5 : Concepts

- **Coût de l'ordonnancement**

```
while(!eventQ.empty()) {
  auto& event = eventQ.front();
  if(curTick != event.tick)
    curTick = event.tick;
  event();
  eventQ.pop_front();
}
```

```
while(true) {
  cpu.cycle();
  curCycle++;
}
```

Boucle simulation

```
void cycle() {
  fetch();
  decode();
  rename();
  iew();
  commit();
  schedule(tick, 1 cycle);
}
```

```
void cycle() {
  commit();
  iew();
  rename();
  decode();
  fetch();
}
```

Modèle CPU

gem5 : Concepts

- **Simulateur évènementiel**
  - On ordonnance des évènements (fonctions) qui arriveront dans le **futur**
    - Par exemple, un paquet arrivera au contrôleur mémoire dans 2 cycles, donc appel de handlePacket() dans deux cycles

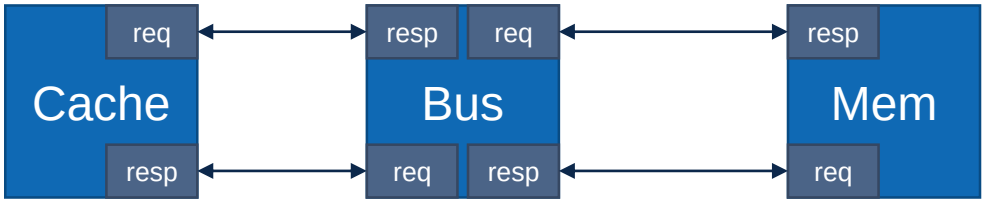
- **Par opposition à une boucle qui simule chaque cycle**
  - Plus efficace pour les modèles qui n'ont pas quelque chose à faire à chaque cycle (système mémoire)
  - Moins efficace pour les modèles qui ont (presque) tout le temps quelque chose à faire (CPU)

- **Debug parfois complexe car on ne peut pas juste suivre le code source pour connaître l'ordre d'appel des fonctions des différents composants simulés**

gem5 : Concepts

- **SimObject**
  - Un composant du système (e.g., CPU, mémoire, caches)
  - Ecrit en C++ (ou Python si aime aller lentement)
    - Manipulable depuis Python via bindings (pybind)
  - ➢ Système construit via fichier de configuration Python

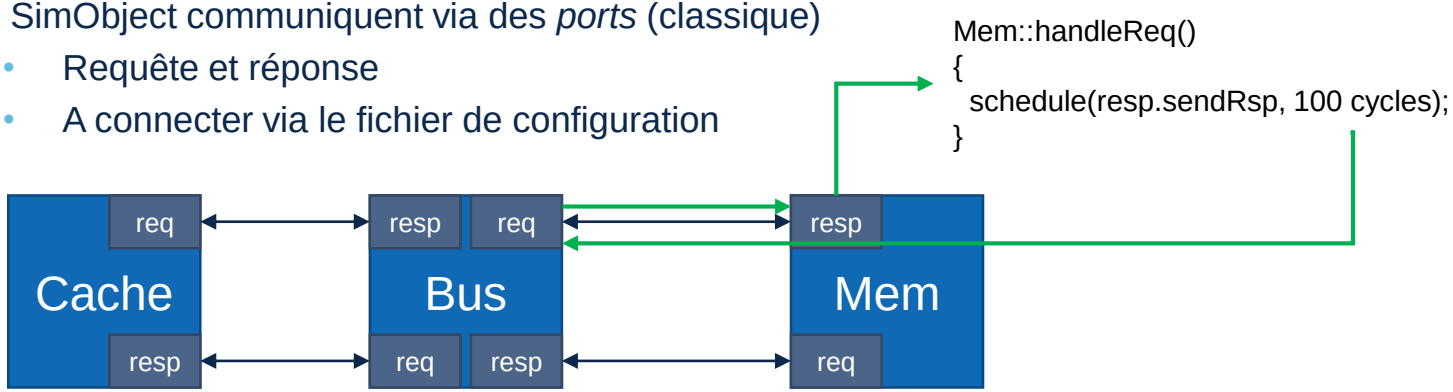
- **Ports**
  - Les SimObject communiquent via des *ports* (classique)
    - Requête et réponse
    - A connecter via le fichier de configuration



gem5 : Concepts

- **SimObject**
  - Un composant du système (e.g., CPU, mémoire, caches)
  - Ecrit en C++ (ou Python si aime aller lentement)
    - Manipulable depuis Python via bindings (pybind)
  - ➢ Système construit via fichier de configuration Python

- **Ports**
  - Les SimObject communiquent via des *ports* (classique)
    - Requête et réponse
    - A connecter via le fichier de configuration



gem5 : Concepts

- **Etendre un modèle**
  - Pour modéliser plus de comportements, ou un comportement différent
  - Héritage côté modélisation (C++)

```
class O3CPU : public BaseCPU // Modèle de CPU out-of-order
```

- **Spécialiser un modèle**
  - Pour avoir un modèle avec des paramètres spécifiques
  - Héritage côté configuration (Python)

```
class AppleM1(O3CPU):
  decodeWidth = 8
  renameWidth = 6
  …
```

gem5 : ISA

- **Support d'ISA multiples**
  - x86-64, Armv7, Armv8, RISC-V
- **Abstraite (à delta près) dans les modèles**
  - CPU : StaticInst & DynInst représentent les instructions statiques et dynamiques de n'importe quelle ISA
    - API à implémenter : execute(), initiateAcc(), completeAcc()
  - Mémoire : Request & Packet représentent les communications mémoire
    - ~Indépendantes de l'ISA : doivent être orchestrées en respectant le modèle mémoire
- **ISA définie dans des scripts Python puis fichiers C++ générés à la compilation**
  - Difficile de rentrer dedans, pas toujours le même style pour chaque ISA…

gem5 : Modèles CPU

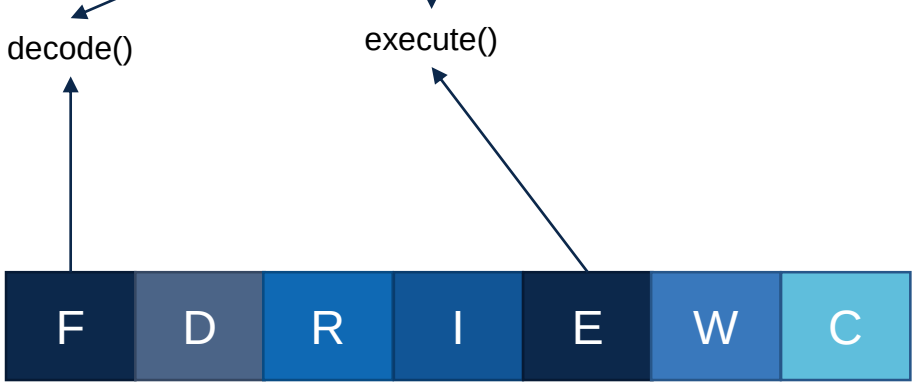
- **atomic**
  - Un ISS, exécute une instruction par cycle, ne modélise pas le temps d'accès à la mémoire. Principalement utile pour le fast-forward.
- **timing**
  - Une instruction par cycle mais modélise le temps d'accès à la mémoire (y compris les caches si il y en a). Utile si on s'intéresse uniquement au système mémoire.
- **Minor**
  - Modélise un processeur à exécution dans l'ordre. Utile si on s'intéresse à la performance du processeur (ou des caches/prefetchers).
- **o3**
  - Modélise un processeur a exécution dans le désordre à. Utile si on s'intéresse à la performance du processeur (ou des caches/prefetchers).
- **kvm**
  - Utilise le processeur hôte pour simuler, donc extrêmement rapide mais ne modélise rien. Utile pour le fast-forward, mais uniquement si ISA hôte == ISA simulée

gem5 : Modèles CPU & ISA

- **Tous les modèles supportent toutes les ISA (sauf kvm ☺)**

atomic

ALU Inst



O3CPU

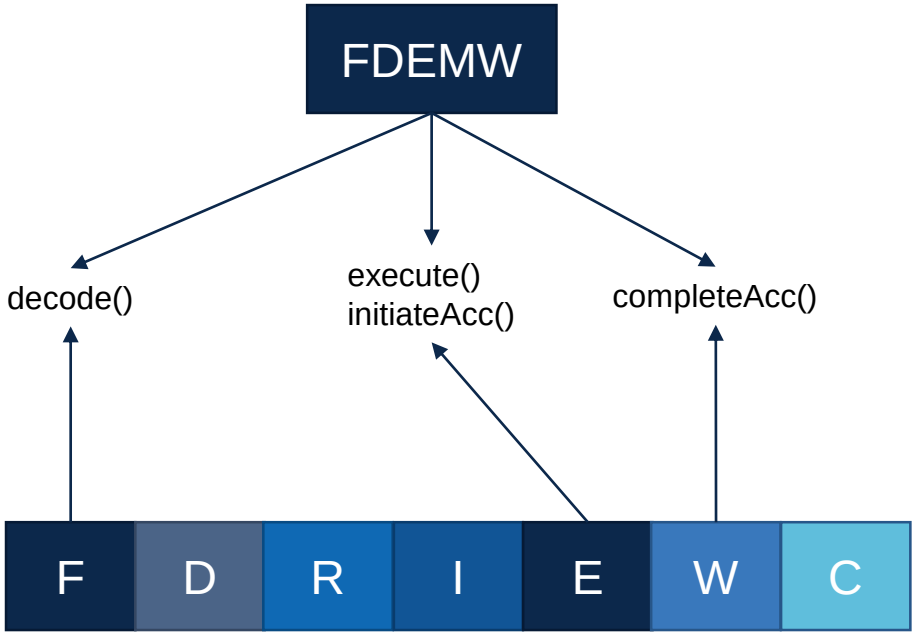
gem5 : Modèles CPU & ISA

- Tous les modèles supportent toutes les ISA (sauf kvm ☺)

atomic

Load Inst

O3CPU



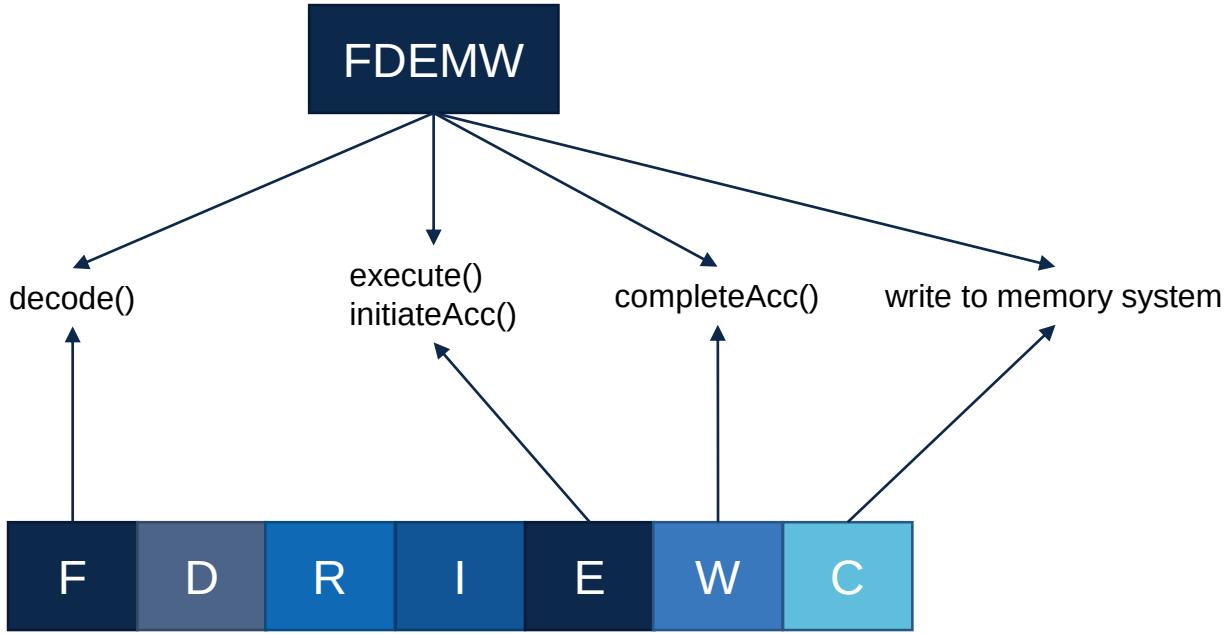
gem5 : Modèles CPU & ISA

- Tous les modèles supportent toutes les ISA (sauf kvm ☺)

atomic

Store Inst

O3CPU



gem5 : Système mémoire

- **Deux systèmes mémoires (caches + cohérence) possibles (pas vraiment compatibles)**
  - « classic » supporté notamment par Arm
    - Plus rigide, basé sur un protocole spécifique
  - ruby
    - Plus générique, permet notamment de définir son propre protocole de cohérence et sa topologie

- **Collection de modèles pré-existants**
  - Caches et politiques de remplacement (Random, LRU, etc.)
  - Prefetchers attaché aux caches
  - DRAM et politiques d'accès (FCFC, FR-FCFC, etc.)
  - Interconnect (cohérent ou non)

gem5 : Exemple minimaliste (TP, atomic.py)

```
system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '3GHz'
system.clk_domain.voltage_domain = VoltageDomain()

system.mem_mode = 'atomic'
system.mem_ranges = [AddrRange('4GiB')]
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]

system.cpu = X86AtomicSimpleCPU()
…

system.membus = SystemXBar()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
…
system.system_port = system.membus.cpu_side_ports
system.mem_ctrl.port = system.membus.mem_side_ports

system.workload = SEWorkload.init_compatible(“path/to/hello/world”)
process = Process()
process.cmd = [“path/to/hello/world”]
system.cpu.workload = process
system.cpu.createThreads()
root = Root(full_system = False, system = system)
m5.instantiate()
exit_event = m5.simulate()
print('Exiting @ tick {} because {}'.format(m5.curTick(), exit_event.getCause()))
```

gem5 : Exemple minimaliste (TP, atomic.py)

```
system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '3GHz'
system.clk_domain.voltage_domain = VoltageDomain()

system.mem_mode = 'atomic'
system.mem_ranges = [AddrRange('4GiB')]
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]

system.cpu = X86AtomicSimpleCPU()
…

system.membus = SystemXBar()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
…
system.system_port = system.membus.cpu_side_ports
system.mem_ctrl.port = system.membus.mem_side_ports

system.workload = SEWorkload.init_compatible(“path/to/hello/world”)
process = Process()
process.cmd = [“path/to/hello/world”]
system.cpu.workload = process
system.cpu.createThreads()
root = Root(full_system = False, system = system)
m5.instantiate()
exit_event = m5.simulate()
print('Exiting @ tick {} because {}'.format(m5.curTick(), exit_event.getCause()))
```

Le système a (au moins) une horloge et une tension (modélisation DVFS)

gem5 : Exemple minimaliste (TP, atomic.py)

```
system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '3GHz'
system.clk_domain.voltage_domain = VoltageDomain()

system.mem_mode = 'atomic'
system.mem_ranges = [AddrRange('4GiB')]
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]

system.cpu = X86AtomicSimpleCPU()
…

system.membus = SystemXBar()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
…
system.system_port = system.membus.cpu_side_ports
system.mem_ctrl.port = system.membus.mem_side_ports

system.workload = SEWorkload.init_compatible(“path/to/hello/world”)
process = Process()
process.cmd = [“path/to/hello/world”]
system.cpu.workload = process
system.cpu.createThreads()
root = Root(full_system = False, system = system)
m5.instantiate()
exit_event = m5.simulate()
print('Exiting @ tick {} because {}'.format(m5.curTick(), exit_event.getCause()))
```

Le système a une mémoire (ici, 4GB, fonctionnelle, modélisée comme de la DDR3 1600)

gem5 : Exemple minimaliste (TP, atomic.py)

```
system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '3GHz'
system.clk_domain.voltage_domain = VoltageDomain()

system.mem_mode = 'atomic'
system.mem_ranges = [AddrRange('4GiB')]
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]

system.cpu = X86AtomicSimpleCPU()
…

system.membus = SystemXBar()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
…
system.system_port = system.membus.cpu_side_ports
system.mem_ctrl.port = system.membus.mem_side_ports

system.workload = SEWorkload.init_compatible(“path/to/hello/world”)
process = Process()
process.cmd = [“path/to/hello/world”]
system.cpu.workload = process
system.cpu.createThreads()
root = Root(full_system = False, system = system)
m5.instantiate()
exit_event = m5.simulate()
print('Exiting @ tick {} because {}'.format(m5.curTick(), exit_event.getCause()))
```

Le système a un processeur (ici, un modèle fonctionnel, x86). Le contrôleur d’interruption est omis

gem5 : Exemple minimaliste (TP, atomic.py)

```
system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '3GHz'
system.clk_domain.voltage_domain = VoltageDomain()

system.mem_mode = 'atomic'
system.mem_ranges = [AddrRange('4GiB')]
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]

system.cpu = X86AtomicSimpleCPU()
…

system.membus = SystemXBar()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
…
system.system_port = system.membus.cpu_side_ports
system.mem_ctrl.port = system.membus.mem_side_ports

system.workload = SEWorkload.init_compatible(“path/to/hello/world”)
process = Process()
process.cmd = [“path/to/hello/world”]
system.cpu.workload = process
system.cpu.createThreads()
root = Root(full_system = False, system = system)
m5.instantiate()
exit_event = m5.simulate()
print('Exiting @ tick {} because {}'.format(m5.curTick(), exit_event.getCause()))
```

Interconnect et branchement des ports CPU et contrôleur mémoire

gem5 : Exemple minimaliste (TP, atomic.py)

```
system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '3GHz'
system.clk_domain.voltage_domain = VoltageDomain()

system.mem_mode = 'atomic'
system.mem_ranges = [AddrRange('4GiB')]
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]

system.cpu = X86AtomicSimpleCPU()
…

system.membus = SystemXBar()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
…
system.system_port = system.membus.cpu_side_ports
system.mem_ctrl.port = system.membus.mem_side_ports

system.workload = SEWorkload.init_compatible(“path/to/hello/world”)
process = Process()
process.cmd = [“path/to/hello/world”]
system.cpu.workload = process
system.cpu.createThreads()
root = Root(full_system = False, system = system)
m5.instantiate()
exit_event = m5.simulate()
print('Exiting @ tick {} because {}'.format(m5.curTick(), exit_event.getCause()))
```

Mise en place workload, instantiation et lancement simulation

gem5 : stdlib

- **Depuis gem5 v22, effort pour créer une « librairie standard » gem5**
  - Systèmes ou sous systèmes pré-existants
    - Boards, hiérarchie mémoire, etc
  - Ressources
    - Images, kernel, benchmarks

- **Pas couvert dans cette présentation (car pas expert), mais de la documentation est disponible**
  - https://www.gem5.org/documentation/gem5-stdlib/overview

- **Et de manière générale, de nombreuses ressources**
  - Tutoriels gem5 en conférence sur youtube / slides
  - La documentation s'améliore

gem5 : Conclusion

- **Une boîte à outil puissante pour simuler des systèmes simples et complexes au *niveau cycle***
  - CPU + Caches + Mémoire + Interconnect + Accélérateurs
- **Avec des avantages**
  - Relativement dynamique
  - ISA multiples, Full System, execute-at-execute
- **Avec des risques**
  - Pas de garantie que les modèles représentent du « vrai » matériel
  - Pas exempt de bug (à vous de pull request)
  - Lennnnt (O3 et surtout multicoeur)
- **Est-ce le bon outil pour vous ?**
  - A vous de voir ☺