Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

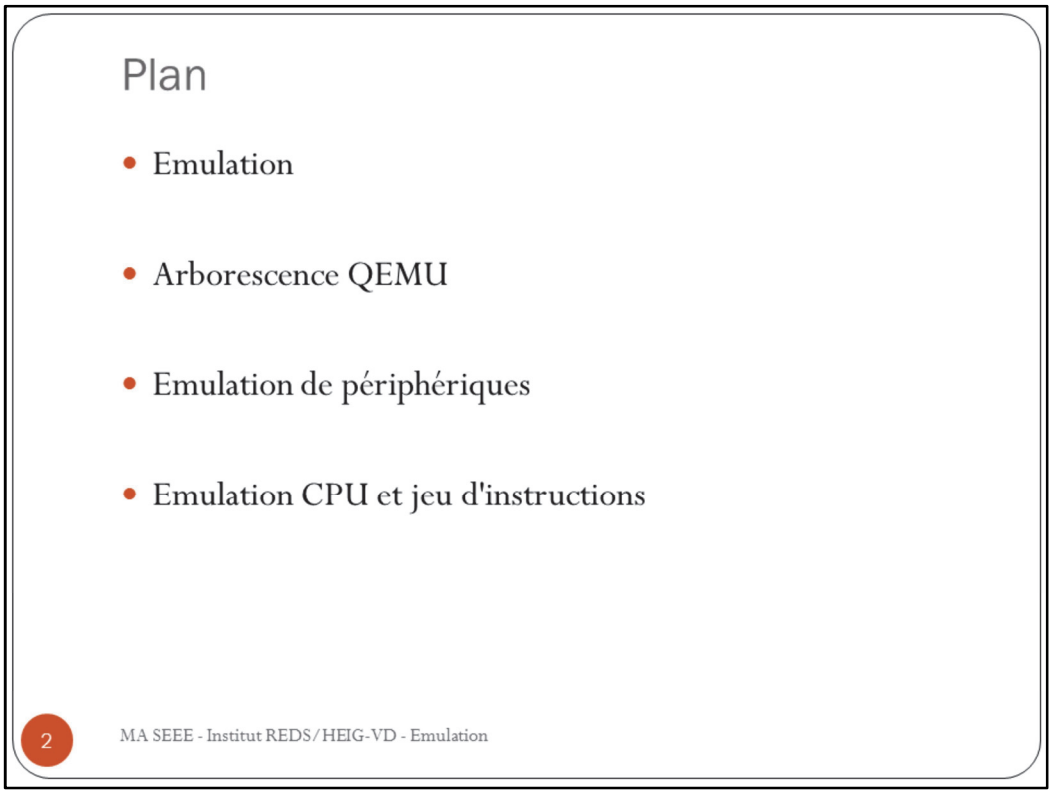
Systèmes d'exploitation et Environnements d'Exécution Embarqués (MA SEEE)
Emulation

Module d'approfondissement MSE

Prof. Daniel Rossier

Version 1.6 (2017)

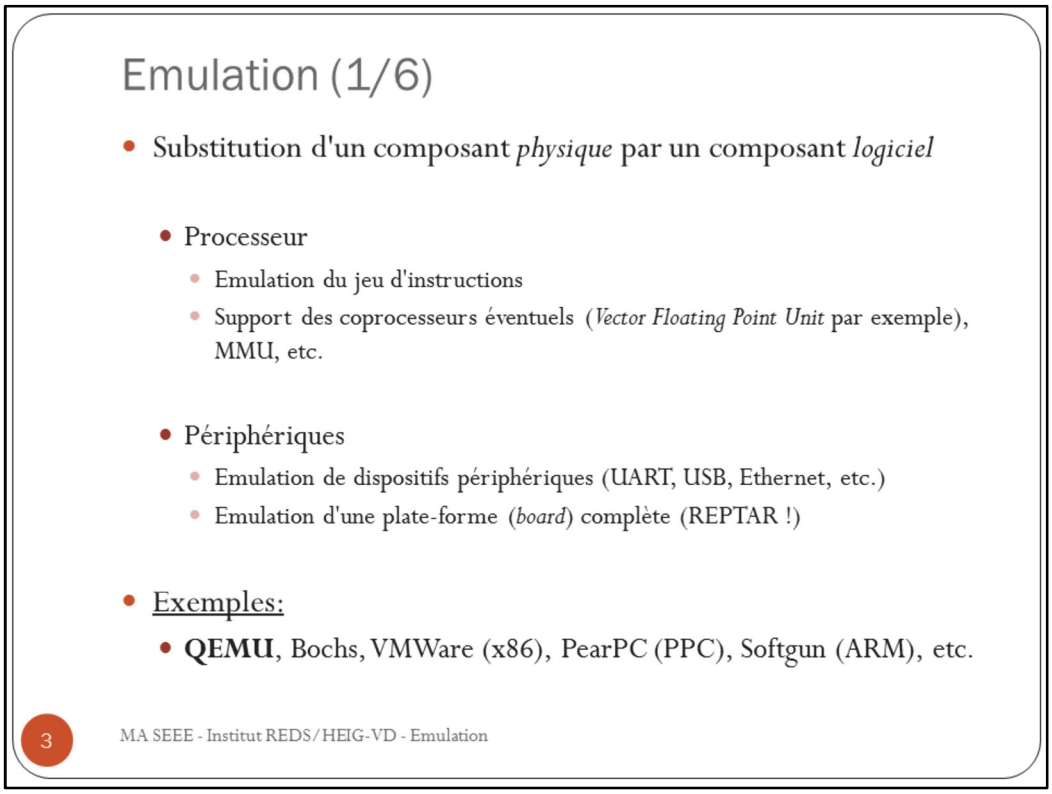
REDS
Reconfigurable & embedded Digital Systems



Les émulateurs jouent un rôle fondamentale dans le développement de logiciel bas niveau pour les systèmes embarqués. Ils permettent de *debugger* aisément les premières instructions qui s'exécutent lors du chargement d'un *bootloader* ou d'un noyau d'OS. Dès lors, ils sont très utilisés lorsqu'aucune interface série (UART) n'est disponible pour recevoir des logs, ou qu'aucun dispositif matériel n'est prévu pour la mise au point. De plus, ils évitent de devoir déployer systématiquement du code sur une cible matérielle et permettent ainsi une économie de temps non négligeable.

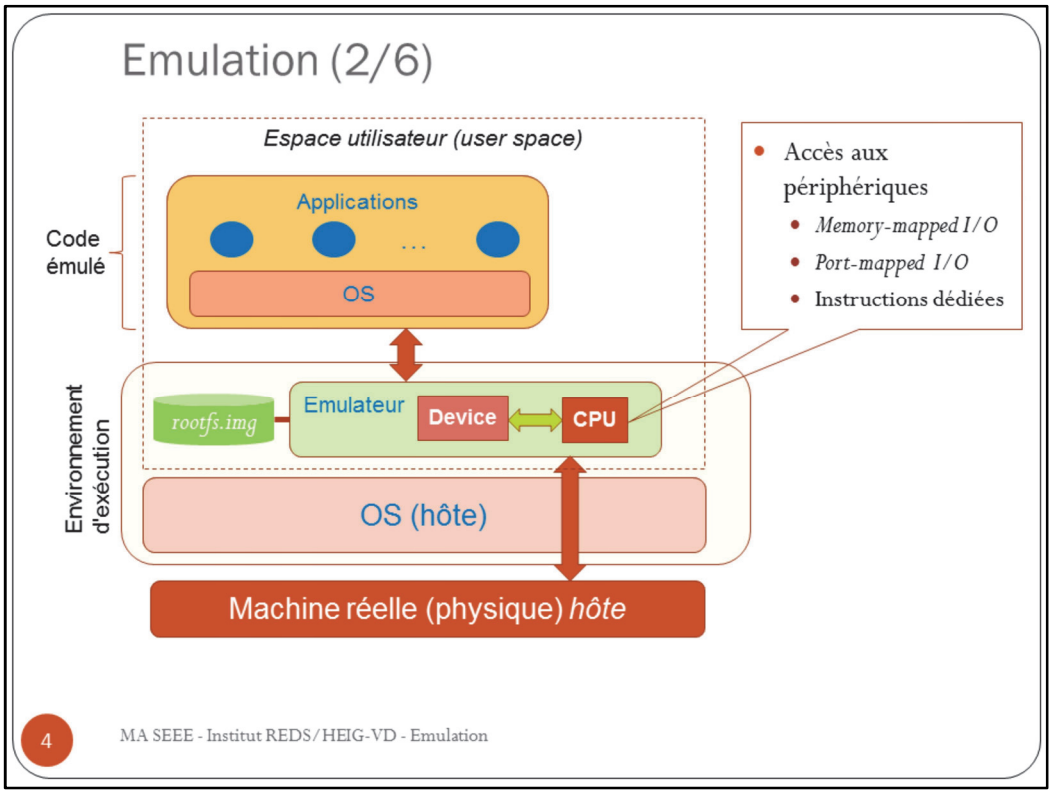
Les émulateurs ne sont pas seulement utiles pour des phases de mise au point, mais s'avèrent être une source d'information inépuisable sur le fonctionnement d'un processeur ou d'un périphérique (pour autant que l'on dispose du code source de l'émulateur). En effet, l'émulation est une représentation fidèle du matériel. On peut donc s'en inspirer pour comprendre les caractéristiques d'un matériel spécifique et le fonctionnement détaillé de celui-ci.

Finalement, l'émulation permet de rajouter des composants matériels de manière logicielle avant même que ceux-ci soient disponibles sur le marché. Par exemple, les nouvelles générations de CPUs sont émulées et des applications complètes peuvent être développées avant même que le processeur ne soit disponible.



La définition du terme émuler est « chercher à imiter ». Il faut voir dans l'émulation une imitation du comportement physique d'un matériel par un logiciel, et ne pas la confondre avec la simulation, laquelle vise à imiter un modèle abstrait. L'émulateur reproduit le comportement d'un modèle dont toutes les variables sont connues, alors que le simulateur tente de reproduire un modèle mais en devant extrapoler une partie des variables qui lui sont inconnues (exemple : la simulation du comportement d'un trou noir). Le recours à un émulateur, selon le contexte, permet de faciliter le développement ou le débogage d'un système ou de remplacer un système obsolète ou inutilisable par un autre. Dans ce cadre, il est possible de faire fonctionner le nouveau système, l'émulateur, de la même manière que le système imité.
(source: Wikipedia, janvier 2010)

Les émulateurs de processeurs et de plates-formes quasi-complètes sont nombreux et relativement stables. Ils constituent aujourd'hui des outils de choix pour la mise au point d'applications embarquées. Des OS complets peuvent facilement tourner sur un émulateur.

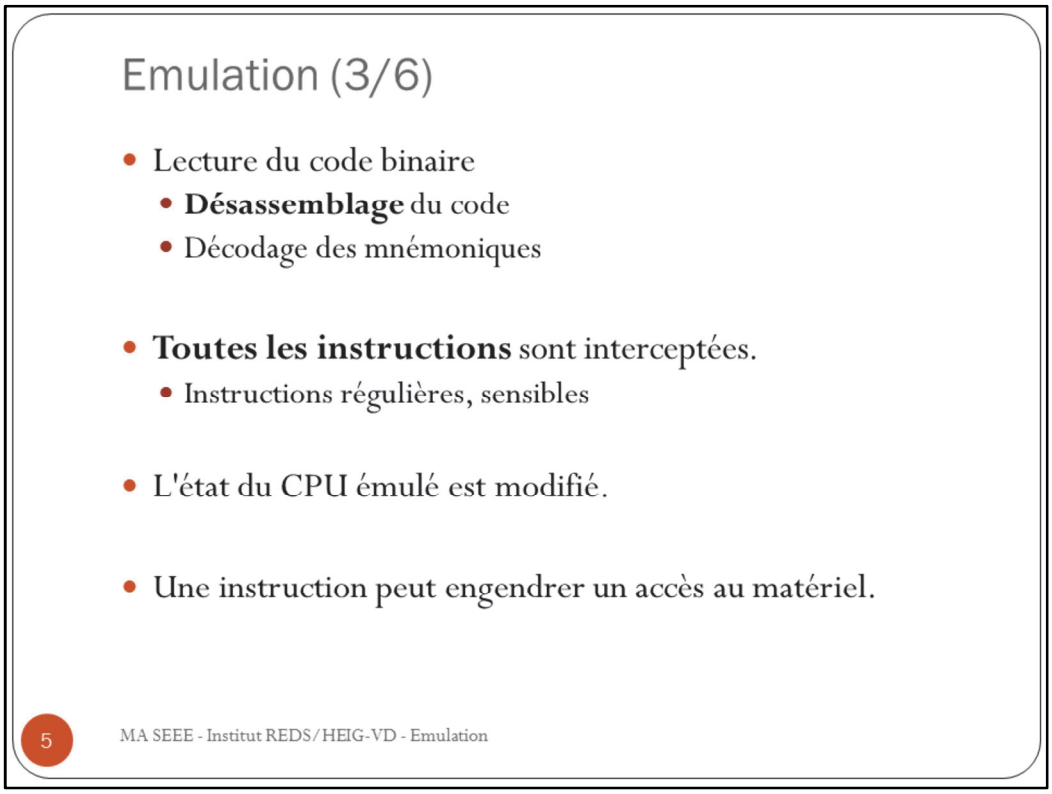


Le principe de l'émulateur repose sur un mécanisme de traitement des instructions destinées à être exécutées sur une **machine *cible*** *(*machine *émulée*) alors que la **machine *hôte*** sur laquelle tourne l'émulateur ne possède pas forcément le même jeu d'instructions.

Le système d'exploitation (OS) natif et l'émulateur fournissent un environnement d'exécution complet pour du code compilé destiné à être exécuté sur une **architecture cible différente de la machine hôte**. Par exemple, une application *cross-compilée* pour un processeur ARM peut tourner sur la machine hôte, équipée d'un processeur x86. De même, un OS entier (avec ses applications) peut tourner sur ce même environnement.

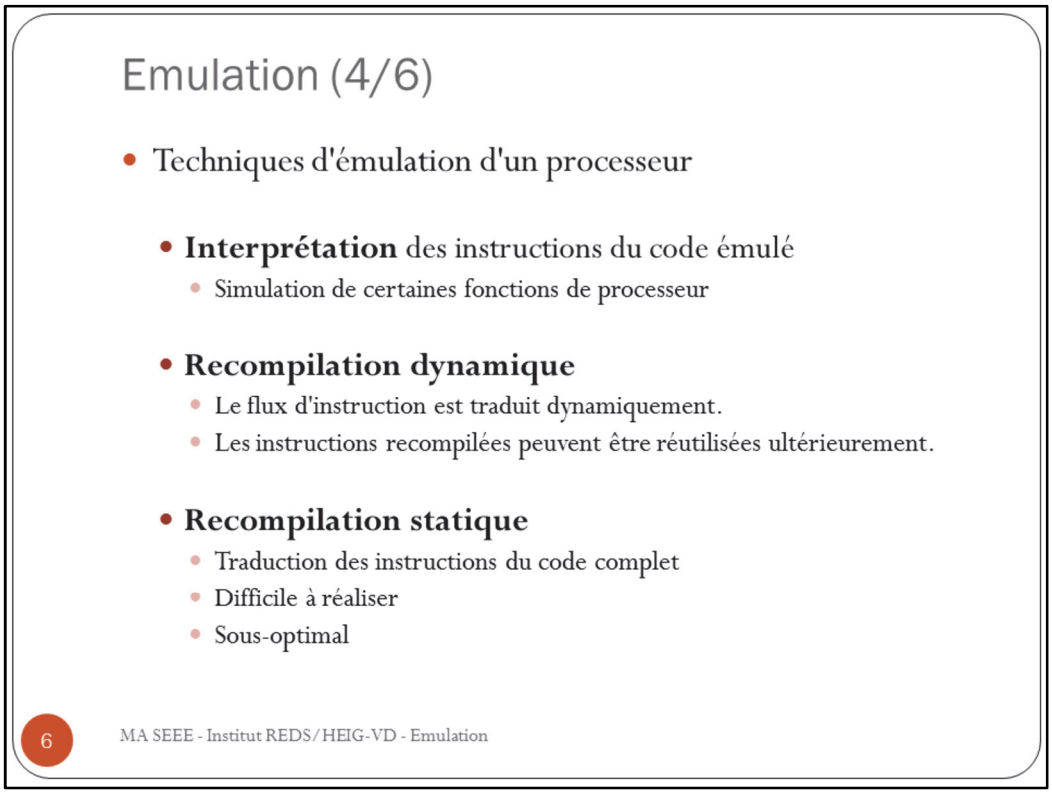
Comme le montre la figure ci-dessus, l'émulateur peut être considéré comme une application à part entière, capable de lire un fichier binaire (par exemple une image contenant un noyau d'OS) et d'interpréter les instructions représentant le code du programme.

De plus, l'émulateur peut émuler des périphériques présents sur la machine cible en respectant les méthodes d'accès (I/Os mappés en mémoire par exemple). Dans ce cas, l'émulateur *intercepte* les accès mémoire, et testent les adresses utilisées. Si ces adresses correspondent à des zones réservées aux périphériques, l'émulateur maintient à jour l'état du périphérique correspondant.



Lors de la lecture d'un code binaire, l'émulateur peut faire la différence entre une instruction *régulière* et une instruction *sensible*. La traduction ne consiste pas seulement à "trouver" une équivalence entre instructions, mais à maintenir à jour une (ou plusieurs) machines d'états. Typiquement, une machine d'état lié au processeur émulé permettra de déterminer l'état courant des *flags*, registres, état des coprocesseurs éventuels, etc.

De plus, certaines instructions peut conduire à l'interaction avec des périphériques physiques connectés à la machine hôte; lors d'une émulation de disque dur ou de mémoire flash, l'émulateur utilise un fichier stocké localement contenant une image du disque ou de la flash. Il en va de même avec l'émulation d'une carte réseau. L'émulateur gère la carte émulée et peut maintenir un lien avec une *vraie* carte réseau sur le PC. Ainsi l'environnement émulé peut disposer d'une vraie connexion réseau TCP/IP.



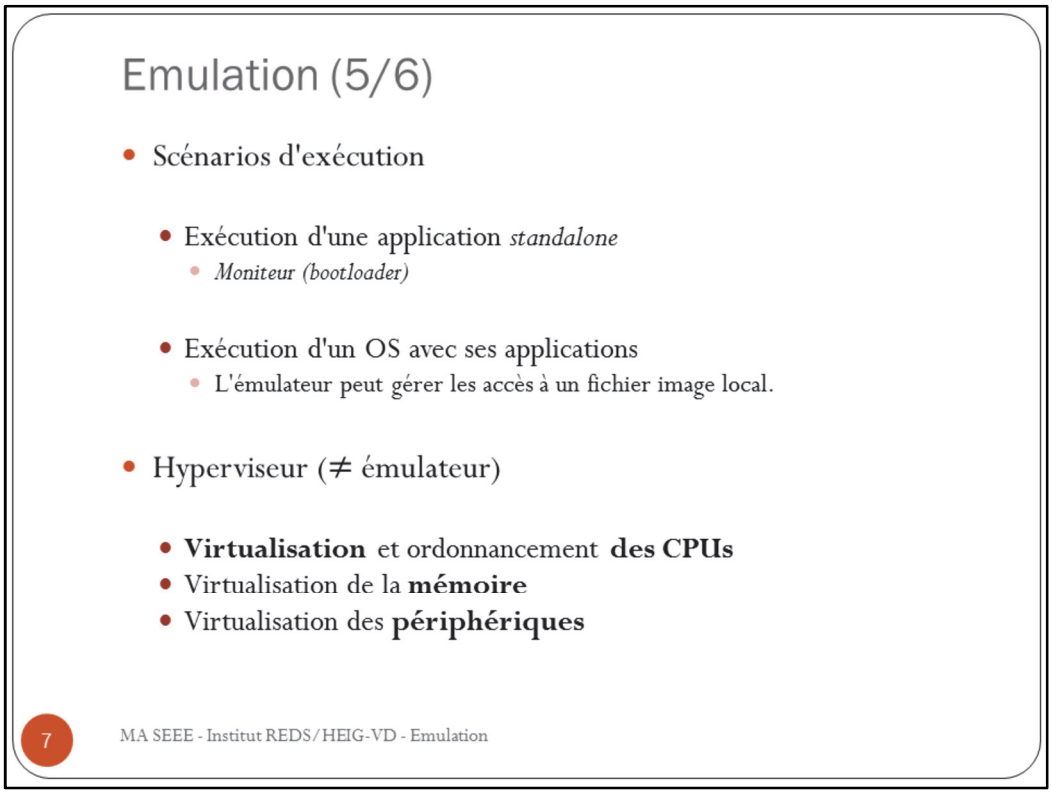
Il existe principalement trois techniques d'émulation d'un processeur:

- **L'interprétation** des instructions du code émulé consiste à traiter chaque instruction individuellement. Les instructions sont d'abord *lues, décodées* et l'émulateur applique l'effet des instructions sur une représentation interne du processeur. Il n'y a pas directement d'interactions avec la machine hôte.

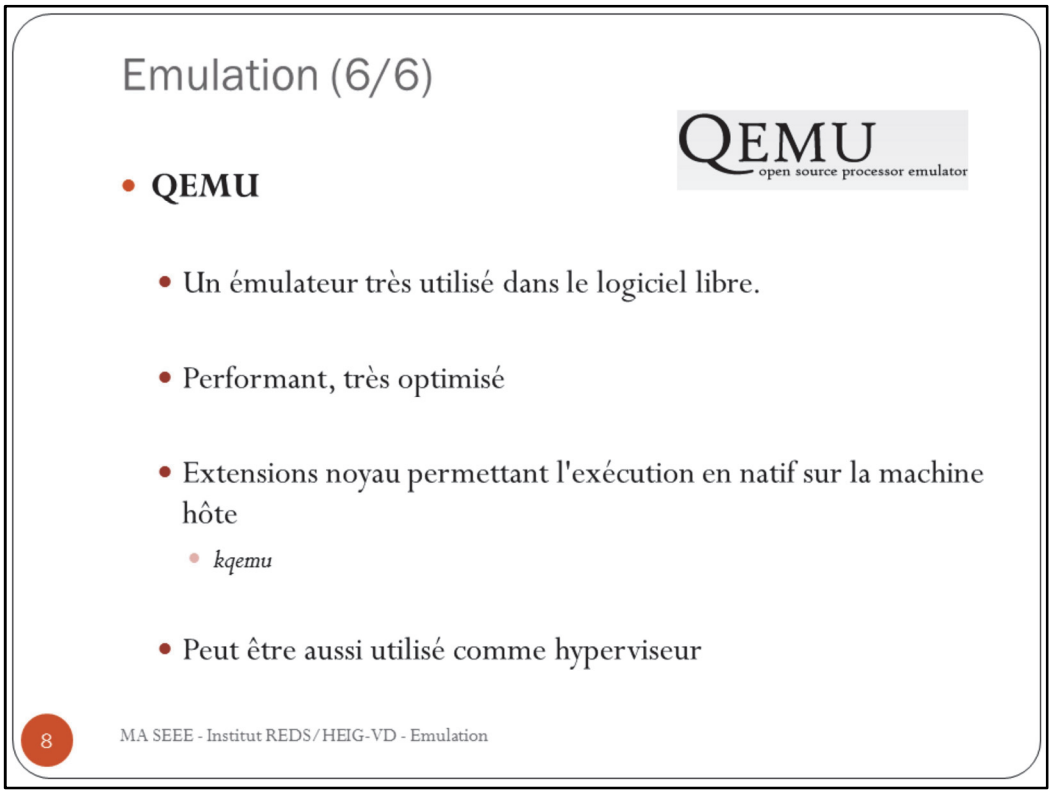
- La **recompilation dynamique** utilise une approche radicalement différente. Le code est **recompilé** à la volée et le processeur hôte exécute les instructions de manière natives. Cette dernière technique repose sur le fait que les processeurs sont dotés de fonctionnalités quasi-similaires. Cette approche est très intéressante si le code émulé utilise des périphériques qui peuvent être en lien avec les périphériques de la machine hôte (par exemple une carte réseau) ou une mémoire RAM.

- La **recompilation statique** est similaire à la recompilation dynamique, sauf que la traduction des instructions émulées sont faites avant l'exécution de celles-ci. Si cette technique peut être intéressante pour du code déterministe, la traduction d'un code binaire quelconque peut s'avérer très complexe et sous-optimale.

Alors que le fonctionnement sous forme d'interpréteur peut s'avérer très lent (car il faut systématiquement lire, décoder et exécuter des fonctions C équivalentes aux instructions, la recompilation dynamique est très performante mais dépend fortement du processeur hôte. Nous verrons par la suite comment réduire cette dépendance au matériel.



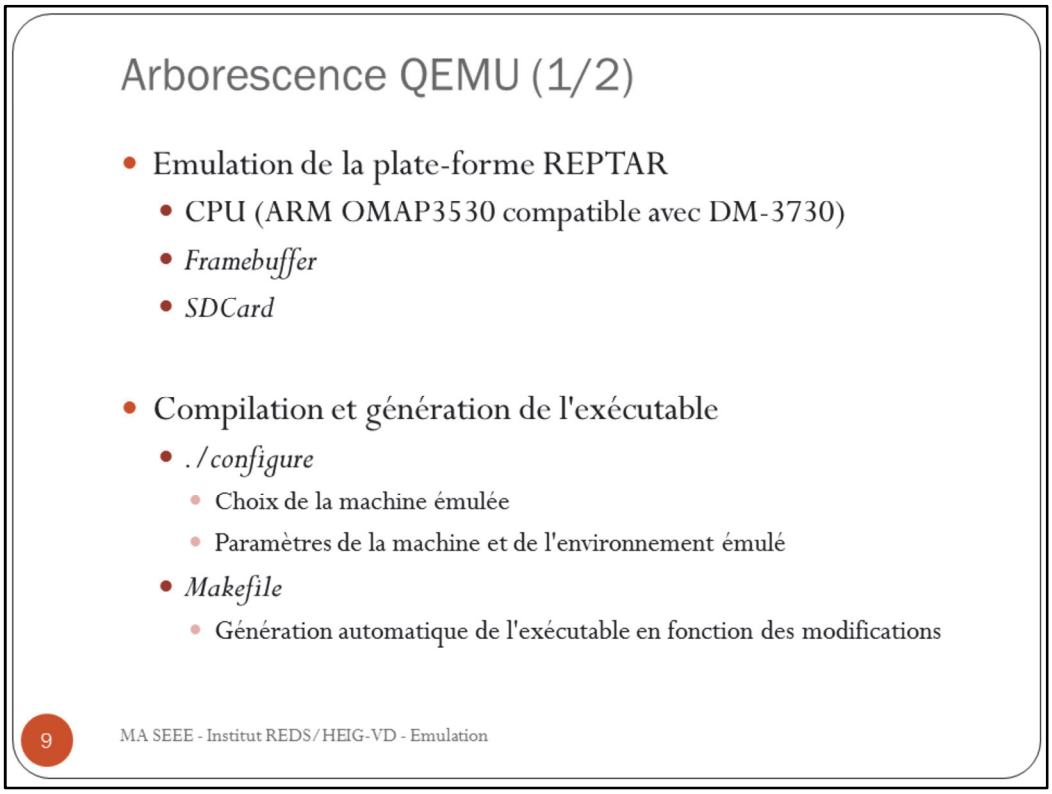
En principe, un émulateur n'a pas la fonction de gérer plusieurs instances d'un même matériel dans le cas où plusieurs OSes tourneraient sur une machine émulée. Ce serait typiquement le cas de la virtualisation qui nécessite des fonctions d'ordonnancement de plusieurs CPU virtuels, des fonctions de multiplexage et démultiplexage de requêtes vers un périphérique physique (celui-ci étant *virtuellement répliqué*), etc. Dans ce cas, on ne parle plus d'un émulateur, mais d'un moniteur de machine virtuel, ou *virtualiseur,* ou encore ***hyperviseur***.



QEMU supporte les plates-formes suivantes:

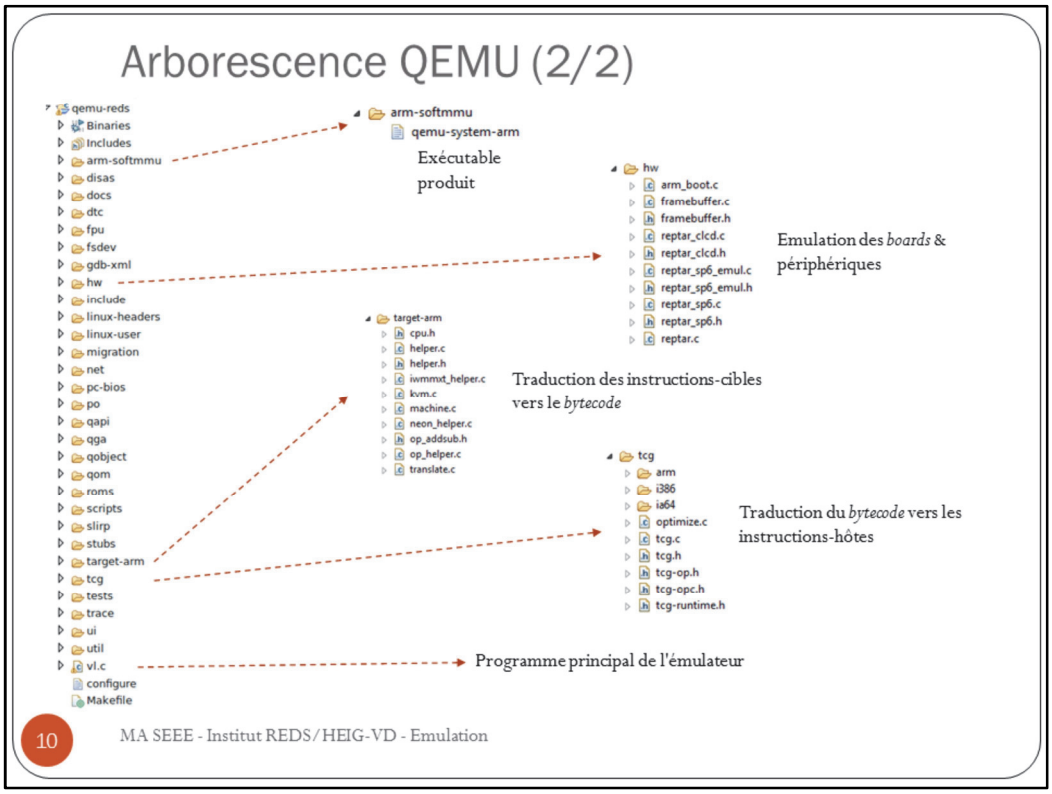
- PC (x86 or x86_64 processor)
- ISA PC (old style PC without PCI bus)
- PREP (PowerPC processor)
- G3 Beige PowerMac (PowerPC processor)
- Mac99 PowerMac (PowerPC processor, in progress)
- Sun4m/Sun4c/Sun4d (32-bit Sparc processor)
- Sun4u/Sun4v (64-bit Sparc processor, in progress)
- Malta board (32-bit and 64-bit MIPS processors)
- MIPS Magnum (64-bit MIPS processor)
- ARM Integrator/CP (ARM)
- ARM Versatile baseboard (ARM) (iMX21)
- ARM RealView Emulation baseboard (ARM)
- Spitz, Akita, Borzoi, Terrier and Tosa PDAs (PXA270 processor)
- Luminary Micro LM3S811EVB (ARM Cortex-M3)
- Luminary Micro LM3S6965EVB (ARM Cortex-M3)
- Freescale MCF5208EVB (ColdFire V2).
- Arnewsh MCF5206 evaluation board (ColdFire V2).
- Palm Tungsten|E PDA (OMAP310 processor)
- N800 and N810 tablets (OMAP2420 processor)
- MusicPal (MV88W8618 ARM processor)
- Gumstix "Connex" and "Verdex" motherboards (PXA255/270).
- Siemens SX1 smartphone (OMAP310 processor)

- **REPTAR board (based on OMAP3530)**
- **MACH-SEEE board**



L'arborescence des fichiers et répertoires de *Qemu* n'est pas très compliquée. Le script de configuration *./configure* permet de générer le fichier *Makefile* nécessaire à la compilation de l'application. Le *Makefile* permet de générer un exécutable unique correspondant à l'émulation d'une (seule) machine spécifique, ce qui accélère la compilation du code.

Au sein de l'institut REDS, nous avons étendu *Qemu* avec le support de la plate-forme REPTAR. Pour l'instant, seul le processeur et quelques périphériques de base (carte réseau, *framebuffer*, *sdcard*) sont supportés. Un des exercices de laboratoire de ce cours consistera à étendre la plate-forme émulée REPTAR avec un FPGA et quelques composants périphériques.



Afin de comprendre le fonction de l'émulateur *Qemu*, nous allons examiner un peu le contenu de son arborescence.

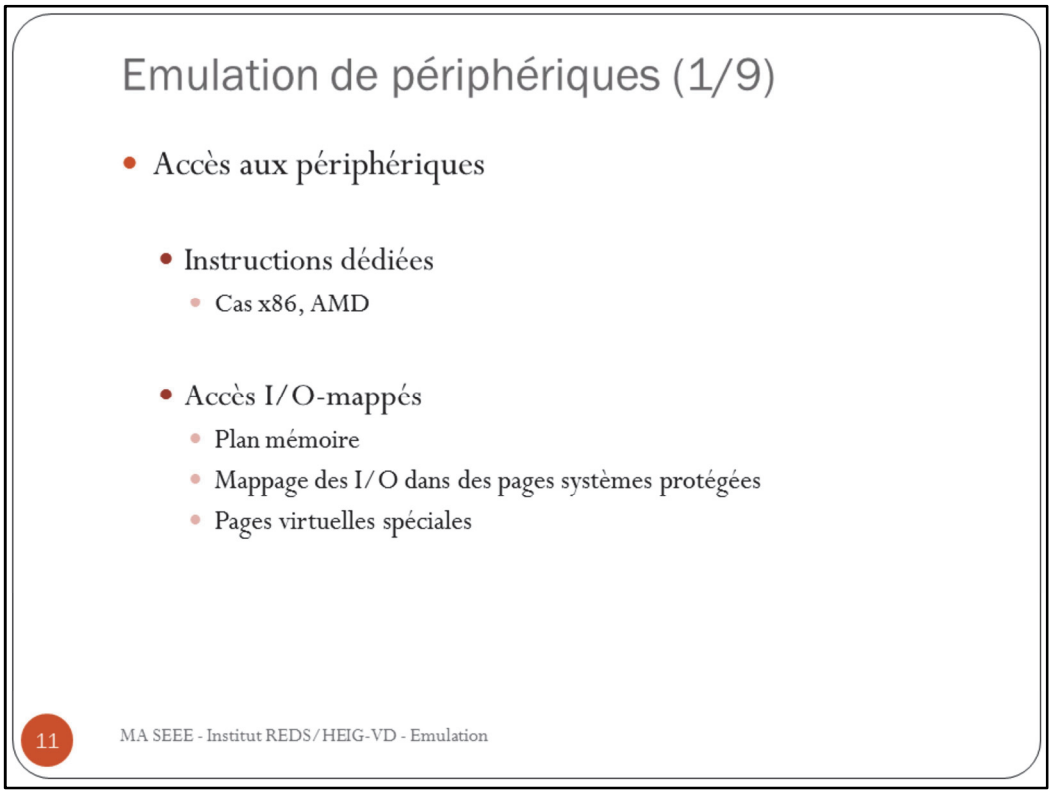
Le fichier *vl.c* à la racine de l'arborescence constitue le programme principal: c'est le cœur de l'émulateur. Toutes les structures et *threads* de l'application sont initialisés dans ce fichier. Puis, l'appel à la fonction de boucle principale *(main_loop())* démarre l'exécution du code à exécuté sur la machine émulée.

Le répertoire *hw* contient le code de l'émulation des machines et des périphériques. Rajouter un périphérique à émuler consistera à rajouter un fichier ou (un sous-répertoire) dans ce répertoire.

Le répertoire *target-arm* contient le code nécessaire au désassemblage des instructions ARM, des instructions liées aux coprocesseurs arithmétiques (*neon, VFP, DSP,* etc.). Les instructions désassemblées sont ensuite traduites dans un langage intermédiaire (*bytecode*) spécifique à *Qemu* (cf plus loin).

Le répertoire *tcg* contient le code permettant la traduction du *bytecode* vers le code natif (typiquement *x86* ou *ia64*).

Le répertoire *arm-softmmu* contient le résultat de la compilation (l'ensemble des codes objets ainsi que l'exécutable se trouvent dans ce répertoire). Ce répertoire est utile pour examiner quels codes sources ont été compilés.

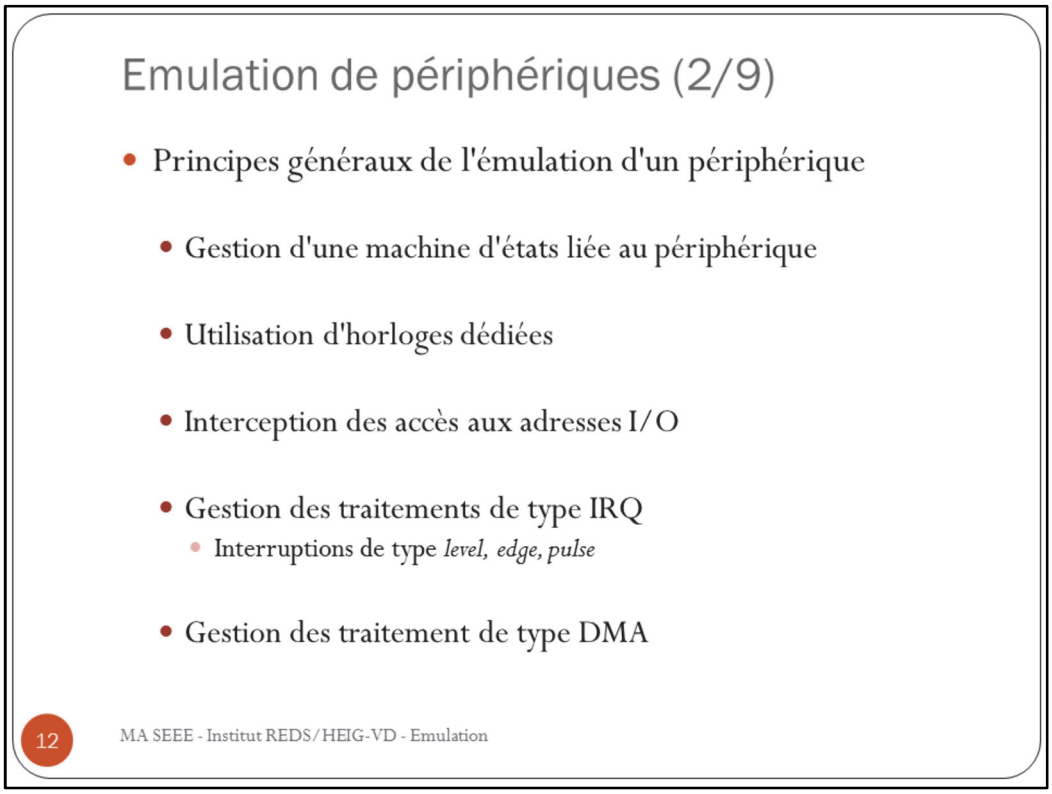


L'émulation logicielle de périphériques embarqués sur une plate-forme permet de mettre au point les *drivers* et de tester différentes variantes matérielles sans devoir utiliser le *vrai* matériel (celui-ci peut ne pas être disponible au moment du développement logiciel). Bien entendu, cela n'a un sens que si l'émulation du périphérique soit le plus proche possible du matériel réel.

Le périphérique est généralement associé à un contrôleur programmable contenant une série de registres et des mémoires. Le processeur possède des instructions d'entrée-sortie particulières (*in, out*) utilisant des numéro de port, ou, dans le cas de la majorité des microcontrôleurs, le processeur accède aux contrôleurs via des adresses physiques I/O.

C'est donc naturellement via le plan d'adressage physique du microcontrôleur que l'on pourra déterminer les plages d'adresses associées aux périphériques.

Lorsque le code émulé effectue des instructions de transfert mémoire via des adresses I/O, l'émulateur devra ***intercepter*** les instructions effectuant ces accès et effectuer le traitement adéquat. Lorsque la MMU est activée, les adresses virtuelles sont associées à une *PTE (Page Table Entry*) contenant les bits d'attributs qui indiqueront les droits d'accès (*read, write, execute, etc.*). D'une manière générale, les adresses I/O sont toujours mappées dans une zone protégée de l'espace virtuel, c-à-d nécessitant l'exécution des instructions en **mode privilégié**.



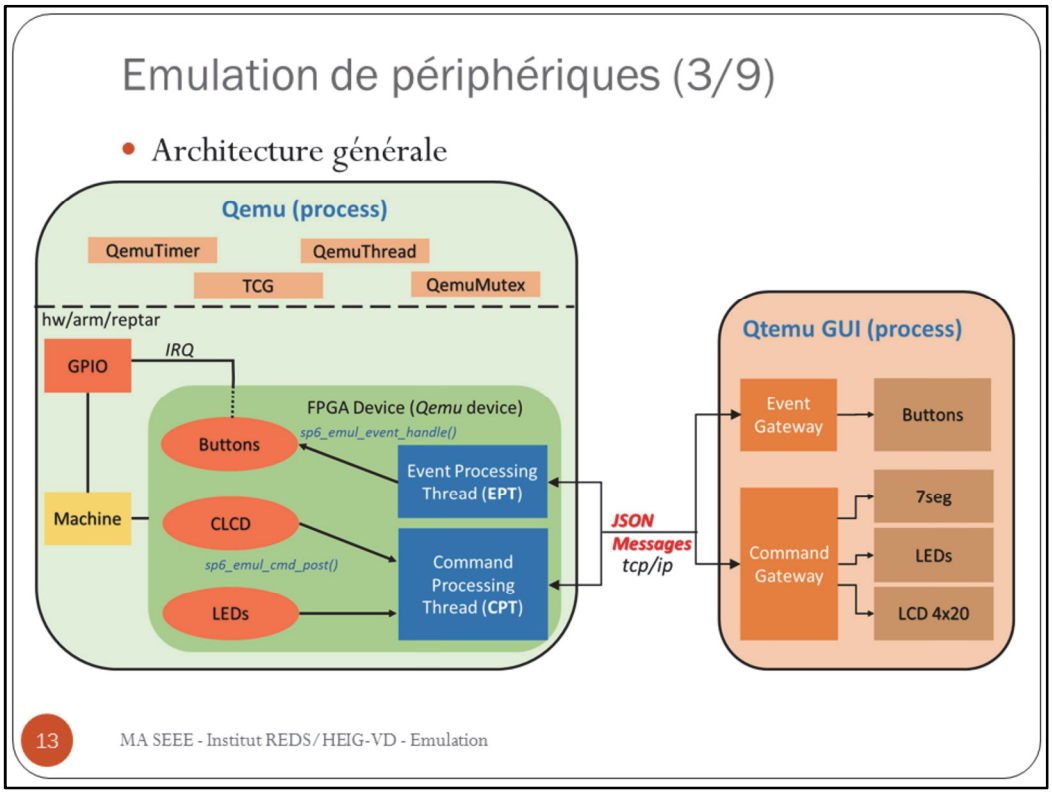
Le principe d'émulation ne varie guère entre un périphérique et un autre. Le périphérique peut nécessiter une ou plusieurs **horloges** (*timers*) dont la source est matérielle (*timer dédié*) afin que celui-ci puisse exécuter son microcode. Il est à noter que le contrôleur d'un périphérique évolue de manière indépendante (donc parallèle) au processeur.

Afin de maintenir l'état du périphérique, le contrôleur nécessite une **machine d'état** (*Finite State Machine*) qui sera initialisé lors de la mise sous tension et évoluera au gré des accès du processeur et du périphérique. L'émulation de cette partie est relativement simple à réaliser, la machine d'état pouvant être implémentée à l'aide de blocs "*switch/case*" en langage C.

Les accès du processeur (en provenance du *driver*) vers le contrôleur s'effectue **via les adresses** comme nous l'avons vu précédemment.

La communication du périphérique vers le processeur s'effectuera à l'aide **d'interruptions matérielles** (*Interrupt ReQuests)*. L'émulation du périphérique génèrera l'interruption de manière adéquate, qu'elle soit de nature ***edge*** (niveau), ***level*** (flanc montant/descendant), ***pulse*** (impulsion).

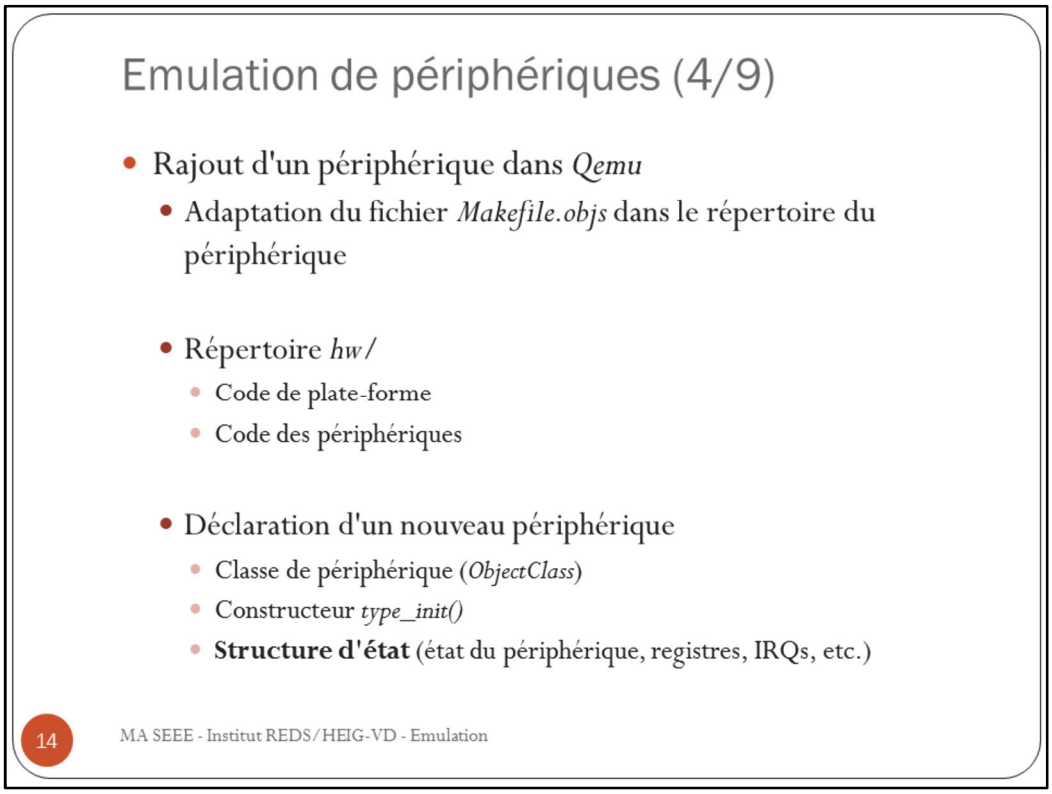
Finalement, le périphérique peut initier un transfert de données en bloc à l'aide d'un **accès DMA** (*Direct Memory Access*). Le transfert s'effectue directement de la mémoire du périphérique vers la RAM, sans l'intervention du processeur. Une interruption signalera à ce dernier que les données sont disponibles.



Le schéma ci-dessus montre une extension de périphérique possible (cas d'une *FPGA* sur la plate-forme *REPTAR*). La *FPGA* constitue le nouveau périphérique (*device*) que l'on souhaite émuler. Il est doté d'un plan mémoire permettant l'accès aux divers contrôleurs de périphériques internes. Le *device FPGA* est donc enregistré comme *device Qemu* et réagit sur l'accès aux adresses I/O prédéfinies. De plus, il a la possibilité d'émettre des interruptions matérielles (*IRQs*) pour une communication asynchrone vers le processeur.

L'émulation de la plate-forme est également associée à une application graphique externe à *Qemu* (GUI) permettant à l'utilisateur d'interagir aisément avec celle-ci. La communication entre *Qemu* et l'application graphique s'effectue à l'aide d'une passerelle (*gateway*) traitant les commandes en provenance de la *FPGA* et permettant la transmission d'événements (par exemple générés lorsque l'on appuie sur un boutton). Ces événements sont ensuite "convertis" entre interruption matérielle vers le processeur.

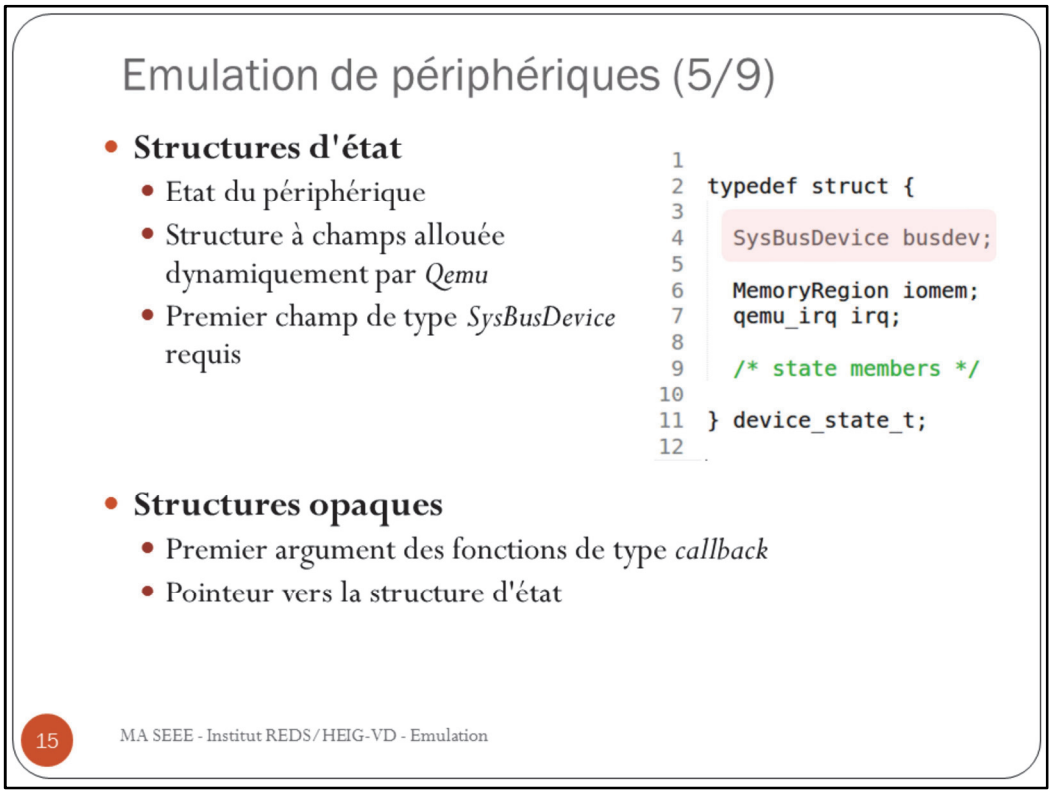
La communication entre *Qemu* et l'application GUI est prise en charge par deux *threads* de contrôle (*Event Processing Thread* et *Cmd Processing Thread*). Les *devices* internes peuvent envoyer des commandes au *CPT* à l'aide de la fonction *sp6_emul_cmd_post()* et implémenter le *callback sp6_emul_handle_event()* invoqué par l'EPT.



La compilation d'un nouveau périphérique émulé s'effectue simplement en rajoutant une cible dans le fichier *Makefile.target* (ensemble des fichiers de code objet rajoutés).
Les composants doivent être placés dans le répertoire *hw/* .

Le code du nouveau composant contiendra les structures de données permettant de conserver son état et toutes les variables nécessaires à son fonctionnement, ainsi qu'un ensemble de fonctions de type *callback* qui seront invoquées par le noyau de *Qemu* lorsque des accès en lecture/écriture seront effectués par le code émulé, ou lorsque des alarmes seront déclenchées. Par conséquent, *Qemu* doit être **informé** de la présence de ce nouveau périphérique et connaître un minimum d'information, **mais sans la nécessité** de devoir "tout" connaître de lui; cette indépendance au niveau des périphériques émulés est garant de portabilité et d'extensibilité de *Qemu*.

Afin de faciliter la gestion des périphériques, *Qemu* introduit la notion de ***classe*** de périphérique permettant de gérer l'ensemble des périphériques d'une manière aussi modulaire et générique que possible. Dans ce but, les mécanismes internes au compilateur et au préprocesseur *gcc* sont largement utilisés.

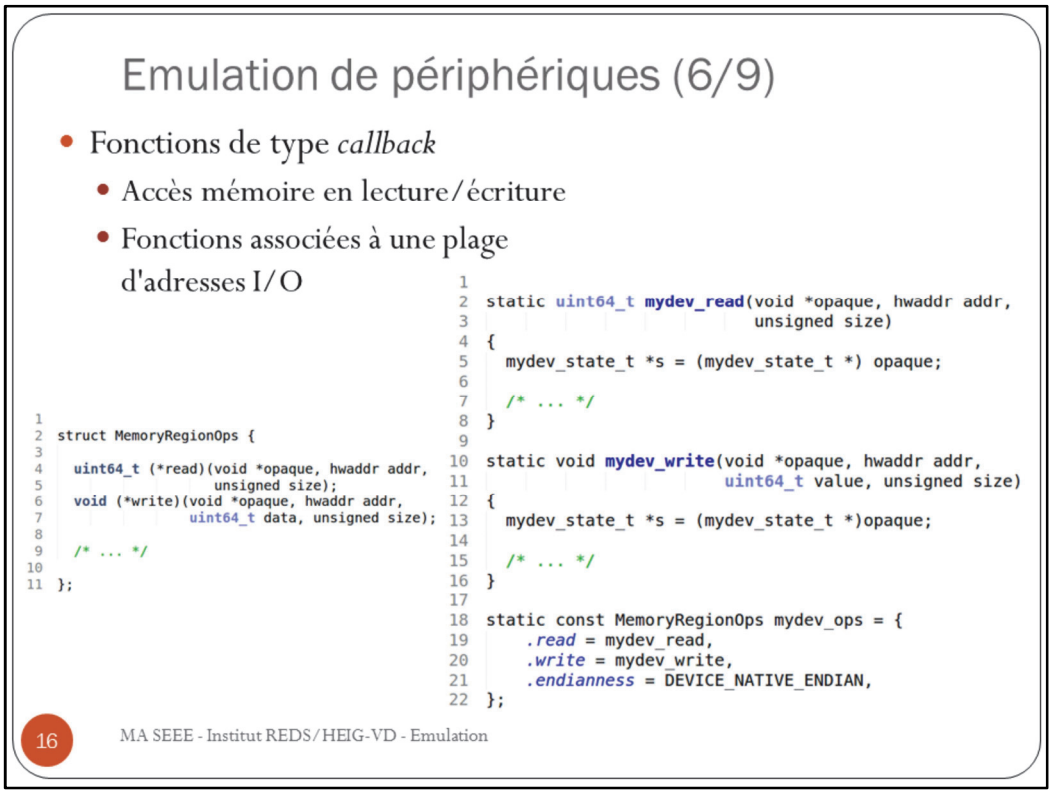


Le champ *SysBusDevice* doit être impérativement le premier champ de la structure, car il permet à *Qemu* de faire le lien avec ses propres structures et n'importe quelle structure d'état.

Ainsi, il est possible de récupérer une référence vers la structure d'état à partir du pointeur vers ce premier champ (de type *SysBusDevice*).

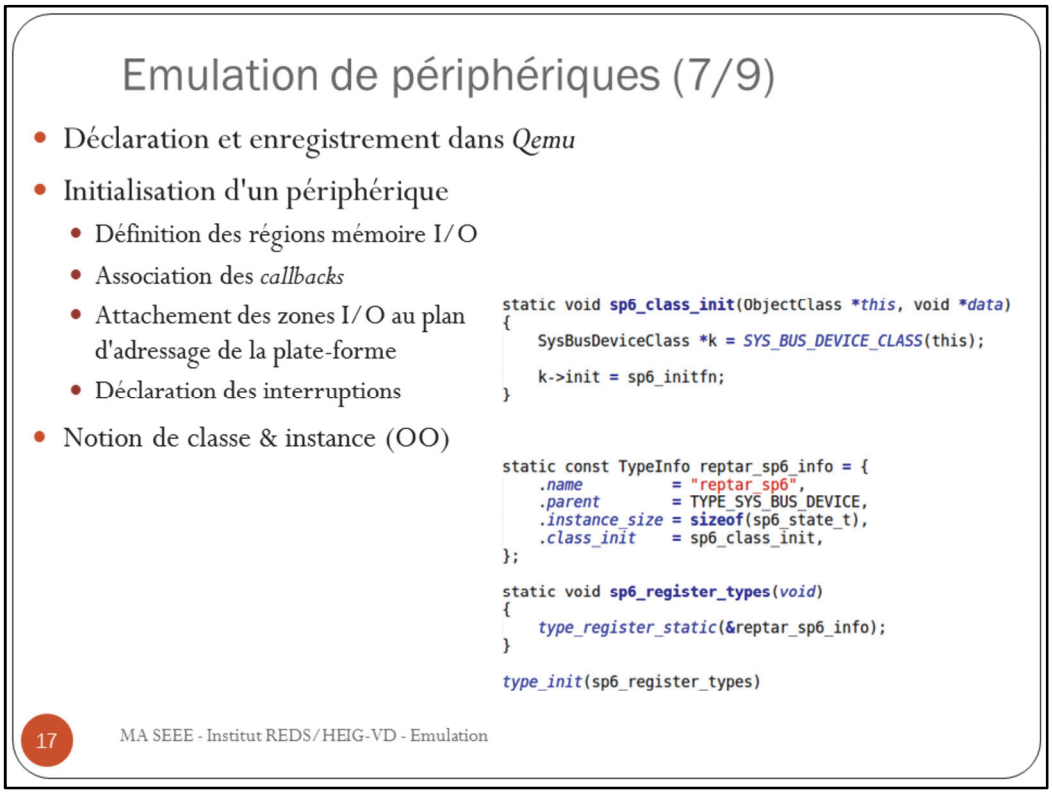
Comme nous le verrons plus loin, la récupération de la structure d'état dans la première fonction d'initialisation du périphérique, à partir d'un *SysBusDevice,* s'effectuera à l'aide des macros *DEVICE()* et *OBJECT_CHECK()*.

Ce mécanisme peut paraître compliqué, mais il est prévu pour gérer dynamiquement l'apparition et la disparition de périphériques.



Le développement d'un périphérique sous *Qemu* consiste à implémenter les fonctions de type *callback* qui seront appelés par le cœur de *Qemu*. Typiquement, ce dernier doit savoir quelles fonctions appelées lors des accès en lecture et en écriture, c-à-d lorsque le processeur exécute une instruction de transfert mémoire avec une adresse I/O associée à ce périphérique.

L'exemple ci-dessus montre bien le passage d'un argument de type *void ** qui permettra la récupération de la structure d'état associé au périphérique concerné par cette fonction.

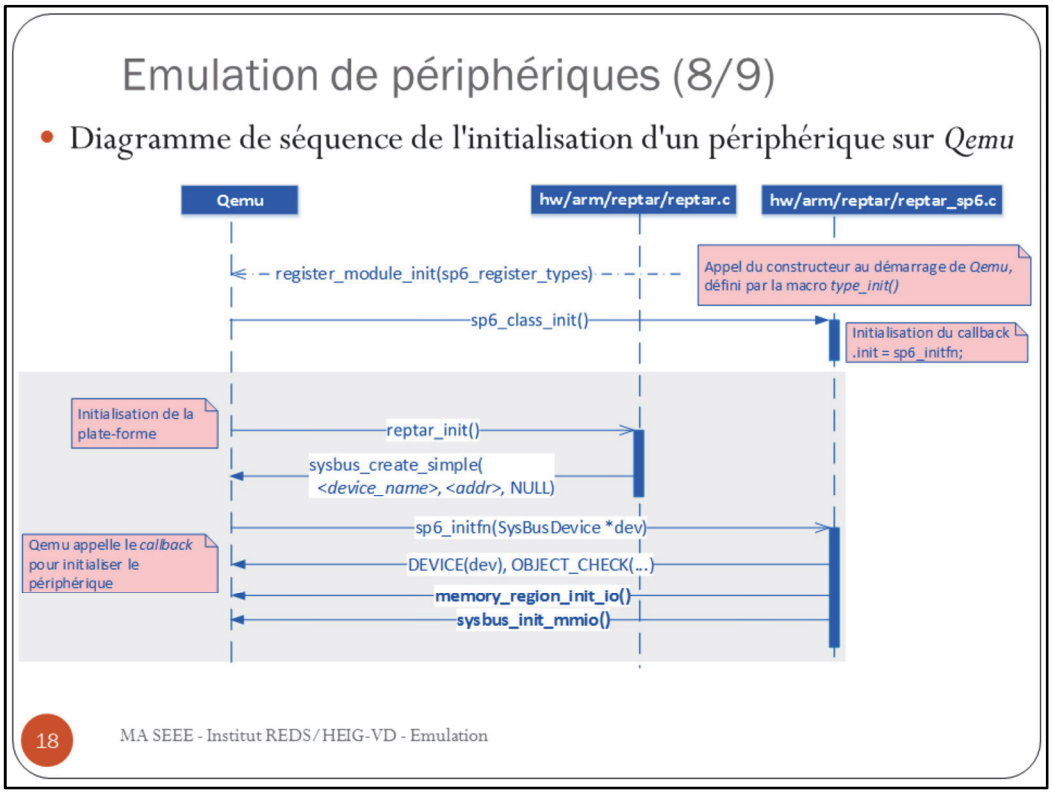


L'initialisation d'un périphérique nécessite l'enregistrement de celui-ci dans le modèle interne de *Qemu*. Un exemple typique de code requis pour un tel enregistrement est montré ci-dessus.

L'association entre les plages d'adresses I/O utilisées par le périphérique et les fonctions concernées s'effectuera ensuite avec des fonctions d'enregistrements dédiés.

Il en va de même pour les interruptions susceptibles d'être générées par le périphérique. Ce dernier peut être relié au contrôleur d'interruption ou transmettre l'interruption via une ligne *GPIO*. Dans les deux cas, des fonctions de configuration sont disponibles et sont invoquées à l'initialisation du périphérique.

La déclaration des zones d'adresses réservées à un périphérique ainsi que la déclaration les interruptions générées par celui-ci doivent faire l'objet d'une initialisation du **côté de la plate-forme** (*board*) et du **côté du périphérique**.

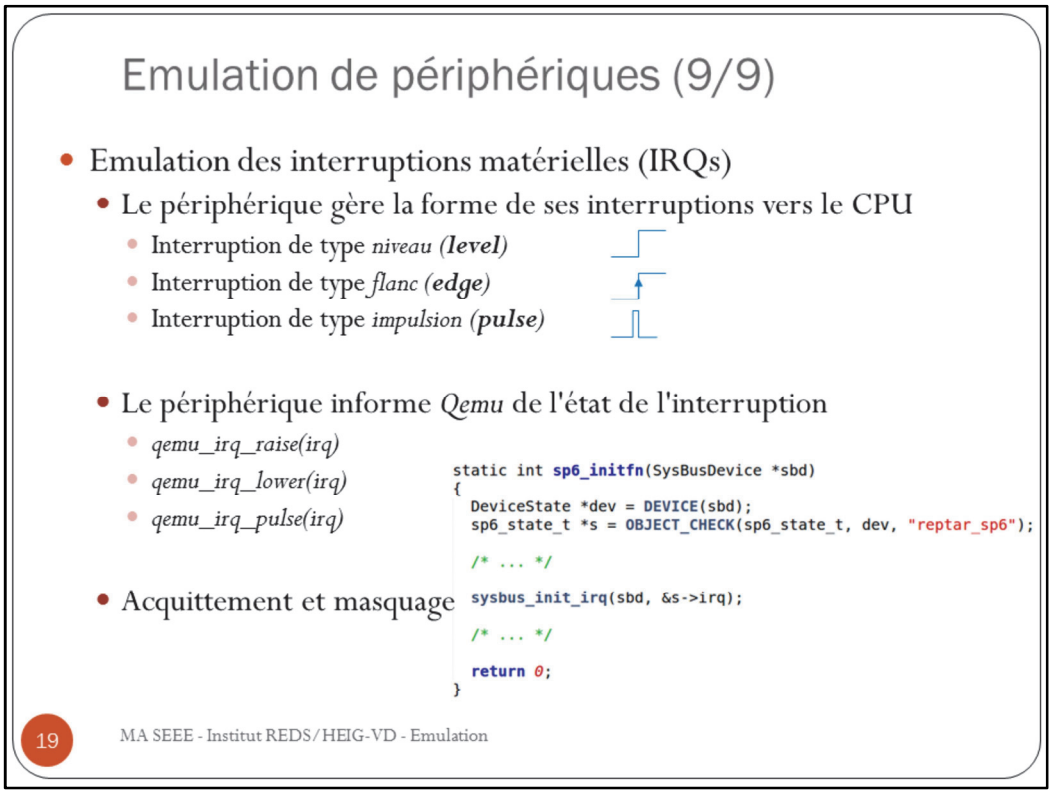


L'initialisation du périphérique est effectuée grâce à un mécanisme particulier du compilateur *gcc*. La macro *type_init()* substitue le code par une fonction préfixée par la directive suivante : *__attribute__ ((constructor)).* Toutes les fonctions préfixées ainsi seront automatiquement appelées juste après le chargement du code en mémoire (pas d'appel explicite par le programme).

L'initialisation d'un périphérique dans *Qemu* démarre donc par son constructeur qui invoque la fonction ***sp6_class_init()*** qui devra définir la fonction à appeler pour initialiser le périphérique dans la structure de type *SysBusDeviceClass*.

La fonction d'initialisation doit ensuite appeler la fonction ***memory_region_init_io()*** permettant de définir les plages d'adresses I/O du périphérique et d'y associer les fonctions *callbacks* correspondantes. La fonction ***sysbus_init_mmio()*** permet d'initialiser les structures requises pour la gestion mémoire.

En contrepartie, le code d'initialisation de la plate-forme devra également instancier les régions mémoire I/O précédemment définies afin que celles-ci apparaissent dans le plan mémoire de la plate-forme; l'instanciation s'effectuera à l'aide de la fonction ***sysbus_create_simple()***.



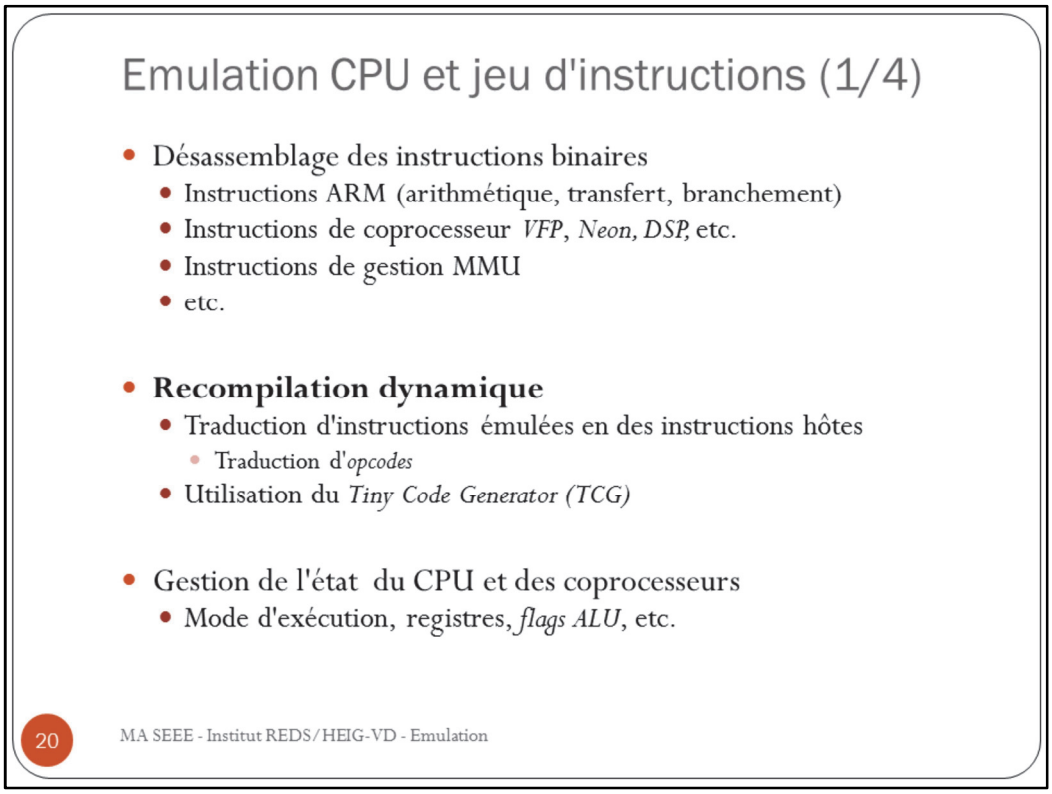
Au niveau du périphérique émulé, l'utilisation d'une interruption consiste à invoquer en premier lieu la fonction ***sysbus_init_irq()*** qui permet de récupérer un identifiant unique lié à l'interruption. Puis, au niveau de la plate-forme, il s'agit de "relier" la ligne d'interruption au périphérique concerné (contrôleur *GPIO* par exemple), en utilisation la fonction ***sysbus_connect_irq().*** Voici un exemple :

```
static void reptar_init(MachineState *machine)
{
  /* ... */

  sysbus_connect_irq(SYS_BUS_DEVICE(s->aDevice), 0,
                     qdev_get_gpio_in(s->cpu->gpio, 29));

  /* ... */

}
```

L'activation d'une interruption repose sur les fonctions suivantes :

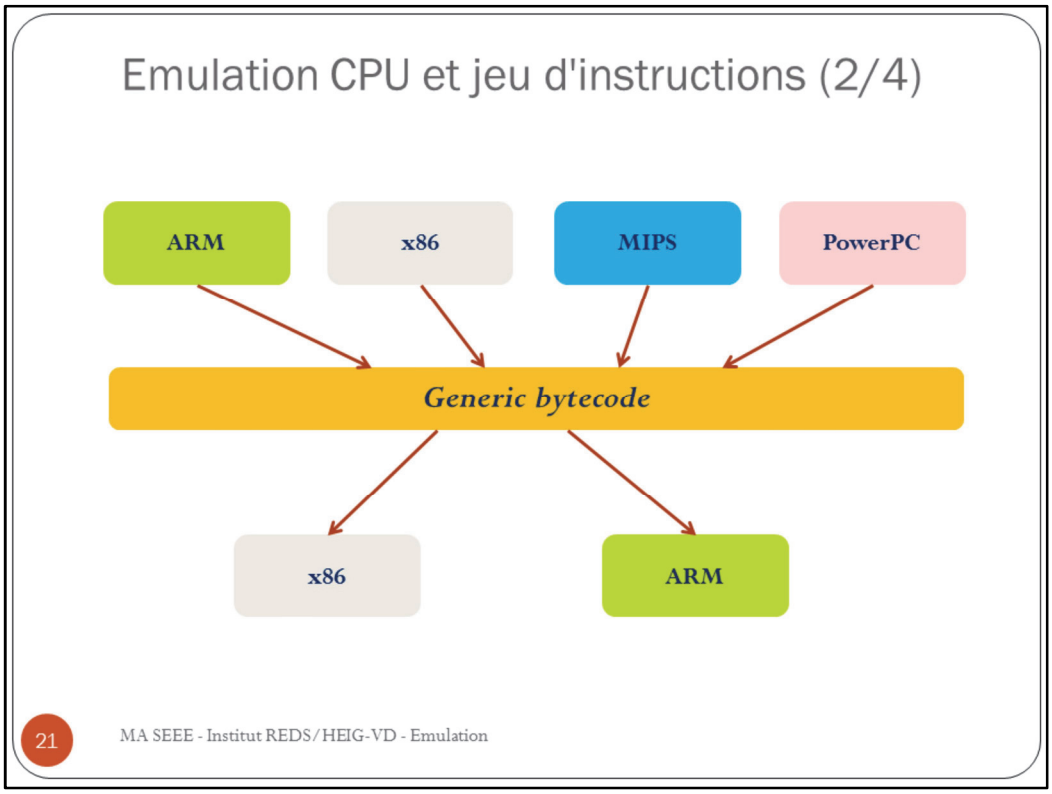
qemu_irq_raise(), qemu_irq_lower(), qemu_irq_pulse()

La fonction *qemu_irq_raise()* permet d'activer la ligne d'interruption (côté périphérique), la fonction *qemu_irq_lower()* de la désactiver. Quant à la fonction *qemu_irq_pulse()*, elle effectue l'appel aux deux fonctions précédentes, dans cet ordre. Il est entendu que le périphérique émulé doit maintenir à jour l'état des interruptions (actif / inactif), ainsi que la politique de masquage s'il en existe une, ainsi que tous les autres paramètres liés aux interruptions.



Le principe de l'émulation consiste à lire les instructions binaires compatibles avec la machine cible et de les transformer (traduire) en des instructions compatibles avec la machine hôte. Différentes instructions peuvent alors être traduites: les instructions de base, les instructions du coprocesseur arithmétique (s'il y en a un), les instructions de la MMU, etc. Chacune de ces instructions ont leur propre algorithme de traduction et génère une ou plusieurs instructions qui seront exécutées sur la machine hôte. Ce mécanisme correspond à celui d'un compilateur et de ce fait, dans le contexte de l'émulation, cette traduction revient à recompiler dynamiquement du code (désassemblé). Dans l'univers QEMU, la génération de code hôte est contrôlée par le *Tiny Code Generator (TCG)*.

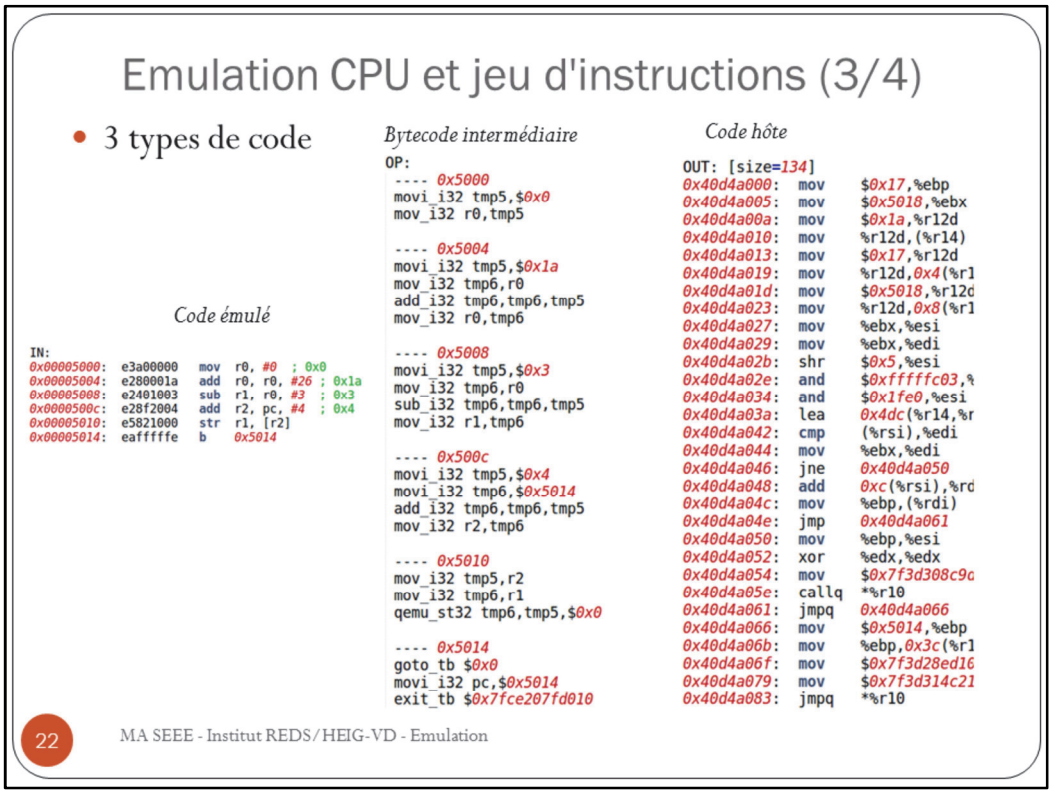
L'émulateur doit également préserver l'état courant du processeur (et de ses coprocesseurs). Typiquement, il s'agit de préserver le registre d'état pouvant contenir les *flags* nécessaires aux opérations arithmétiques (*flags* de l'ALU) ainsi que le **mode d'exécution** du processeur. Bien entendu, les registres du processeur émulé doivent être également mis à jour au fur et à mesure que les instructions cibles sont désassemblées.



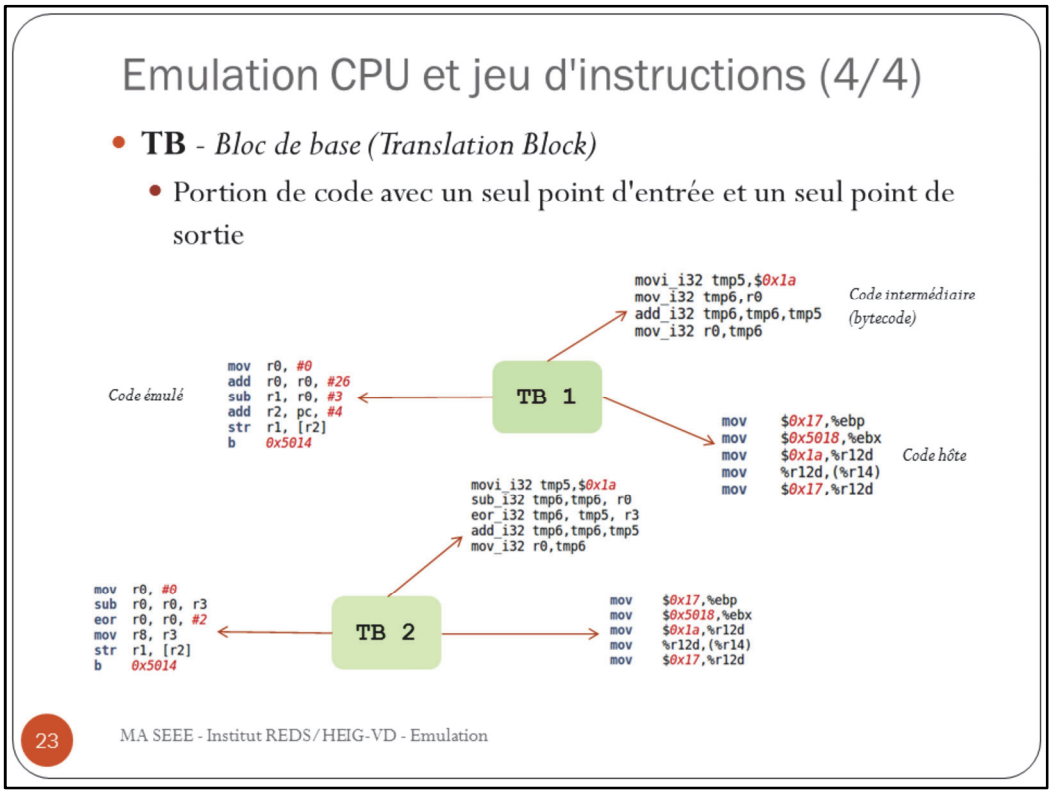
Le code émulé est traduit dans un code intermédiaire appelé *bytecode* par le TCG (*Tiny Code Generator*). Ce *bytecode* est traduit ensuite dans le code de la machine-hôte. Cette traduction intermédiaire permet de simplifier grandement l'émulateur. En effet, si l'on dispose de M architectures émulées et de N architectures hôtes, il faudrait théoriquement M x N traducteurs. L'utilisation d'un code commun permet de disposer de M traducteurs vers ce code, puis de N traducteurs du code commun vers le code hôte. Dès lors, M + N traducteurs sont nécessaires.

L'exécution du code émulé dans *QEMU* est découpé en quatre phases distinctes:

1. Lecture d'un bloc de base du code émulé
2. Désassemblage de ce bloc de base et traduction en un bloc de *bytecode* TCG
3. Traduction du bloc de *bytecode* TCG en un bloc d'*opcodes* natifs du processeur hôte
4. Exécution de ce bloc natif sur le processeur hôte



Le *bytecode* généré permet non seulement de réduire le nombre de traducteurs *hôte-cible* qu'il faudrait mettre en place pour assurer l'émulation de l'ensemble des architectures sur toutes les machines hôtes possibles, mais permet également de moduler la traduction d'une instruction émulée en fonction des capacités du processeur hôte. Par exemple, certaines instructions arithmétiques (rotation) ne sont pas toutes supportées par un processeur hôte. Il est donc possible de donner un *schéma* de traduction correspondant à l'instruction émulée en plusieurs instructions de *bytecode* qui seront facilement traduisibles par des instructions hôtes. Il s'agit en quelque sorte d'une passerelle entre une instruction émulée et une instruction hôte.



Un bloc de base contient une portion de code émulé qui s'exécute de manière déterministe. La dernière instruction peut lancer l'exécution d'un autre bloc de base. Par conséquent, une instruction de saut conditionnel, une exception, un mode de *debug* pas-à-pas, un changement de page mémoire, sont des **conditions de terminaison** d'un bloc de base.

Le *Translation Block (TB)* est lié à un bloc de base: il contient une référence vers la portion de code émulé, une référence vers le code intermédiaire (*bytecode*) ainsi qu'une référence vers le code hôte. Lorsqu'une instruction émulée est désassemblée, le TCG vérifie si l'adresse de l'instruction est associée avec un TB existant (un *cache* de TBs est alors disponible) et si c'est le cas, le code hôte résultant est immédiatement disponible et peut être exécuté rapidement. Cette approche permet une optimisation efficace du code émulé.

De plus, il peut arriver qu'une instruction de saut (non conditionnel) amène à l'exécution d'instructions ne nécessitant pas un changement de page mémoire: dans ce cas, le bloc de base suivant (correspondant aux instructions à l'adresse de saut) est *fusionné* avec le bloc de base précédent. En réalité, *Qemu* n'utilise qu'un TB contenant les deux blocs de bases qui se retrouvent *chaînés* dans celui-ci. C'est une différence subtile entre bloc de base et *Translation Block*.

Références

- Fabrice Bellard, « QEMU, a Fast and Portable Dynamic Translator », http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard_html/index.html
- QEMU Emulator User Documentation, http://qemu.weilnetz.de/qemu-doc.html#sec_005finvocation

- GCC, Declaring Attributes of Functions, http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html
- Linux Assembly HOWTO, Calling conventions, http://asm.sourceforge.net/howto/conventions.html