**Error: Macro Include(TocTme) failed**

```
pre-0.11 Wiki macro Include by provider <class 'includemacro.macros.IncludeMacro'> no longer sup
```

# TME 7 : Simulation logico-temporelle



# Objectif

On souhaite réaliser dans ce TME un petit simulateur logico-temporel, permettant de simuler un réseau Booléen temporisé, où les expressions Booléennes sont représentées par des arbres ABL (voir TME5).

Les simulateurs à événements discrets permettent de simuler des systèmes matériels constitués d'un ensemble de composants matériels interconnectés par des signaux. Les signaux véhiculent fondamentatement deux tensions VSS et VDD représentant respectivement les valeurs Booléennes 0 et 1, mais le simulateur doit traiter plus de valeurs pour gérer les cas spéciaux comme les signaux en haute impédance, les conflits électriques, ou les valeurs indéfinies. A titre indicatif, le VHDL standard utilise 9 valeurs pour les signaux. Pour simplifier, nous nous limiterons dans ce TME aux trois valeurs logiques 0, 1, et U (indéfini).

Dans le cas général, Chaque composant est modélisé par un processus, et tous les processus s'exécutent en parallèle. Chaque processus utilise les valeurs de ses signaux d'entrées pour calculer les valeurs de ses signaux de sortie. Un signal possède un seul émetteur, mais peut avoir plusieurs destinataires. On définit, pour chaque processus, un sous-ensemble des signaux d'entrée, appelé liste de sensibilité du processus : un changement de valeur sur un signal appartenant à la liste de sensibilité peut entraîner un changement de valeur sur un signal de sortie du processus. Un processus doit donc être évalué à chaque fois que l'un des signaux de la liste de sensibilité change de valeur. Par exemple, la liste de sensibilité d'un processus représentant un automate de Moore ne comporte qu'un seul signal, qui est le signal d'horloge.

On s'intéresse dans ce TME au cas particulier des réseaux Booléens: un processus correspond à une expression Booléenne multi-niveaux, représentée par un arbre ABL. Dans ce cas particulier, un processus possède donc un seul signal de sortie, et la liste de sensibilité contient tous les signaux d'entrée. Le réseau Booléen peut être représenté par un graphe biparti comportant deux types de noeuds : des processus et des signaux. Les noeuds à la périphérie du réseau sont toujours des signaux.

En d'autres termes, un processus a toujours au moins un signal entrant et un signal sortant. Les signaux qui n'ont pas d'arrête entrante sont les entrées primaires du réseau, les signaux qui n'ont pas d'arrête sortante sont les sorties primaires du réseau. Les autres signaux sont appelés signaux internes.

On appelle événement le changement de valeur d'un signal à un certain instant. Un événement est donc défini par

un triplet (signal, date, valeur).

Simuler le fonctionnement d'un circuit consiste donc à calculer pour chaque signal la succession des événements, appelée forme d'onde. L'ensemble des formes d'ondes de tous les signaux constitue un chronogramme.

La simulation suppose que l'on possède une fonction d'évaluation qui calcule la valeur du signal de sortie du processus en fonction de la valeur des signaux d'entrée. Dans notre cas, il faudra disposer d?une fonction eval_abl() capable de traiter les trois valeurs (0,1,U).

Créez un répertoire de travail TME7, et copiez dans ce répertoire tous les fichiers et répertoires utiles pour ce TME.

```
cp -rp /users/enseig/encadr/cao/tme7 .
```

# A) Structures de données

On utilise deux structures de données pour représenter :

- le réseau Booléen, c'est à dire le graphe biparti des processus et des signaux,
- l'échéancier, c'est à dire l'ensemble ordonné des événements.

**A1) réseau Booléen**

Le réseau Booléen est constitué d'un ensemble de signaux et d'un ensemble de processus. Les signaux sont regroupés dans trois sous-ensembles disjoints suivant leur type (IN, OUT, INTERNAL).

```
typedef struct boolnet_t {
char            * NAME;         // nom du circuit modélisé
signal_t        * IN ;          // ensemble des signaux IN
signal_t        * OUT ;         // ensemble des signaux OUT
signal_t        * INTERNAL;     // ensemble des signaux INTERNAL
process_t       * PROCESS;      // ensemble des proceessus
} boolnet_t;
```

L?ordre dans les listes des signaux et des processus n'est pas significatif. Ils sont chaînés dans l'ordre de leur création.

Un signal possède un type, un pointeur sur une variable Booleenne (var_t), qui définit son nom et sa valeur (0,1 ou U). Il possède également un pointeur sur le processus dont il est la sortie, et un pointeur sur la liste chaînée des processus dont il est une entrée.

```
typedef struct signal_t {
boolnet_t       * BOOLNET;      // pointeur sur le réseau Booléen
sigtype_t         TYPE;         // type du signal
var_t           * VAR;          // variable Booléenne associée
bip_t           * TO_PROCESS;   // liste des process destinataires
process_t       * FROM_PROCESS; // process source
signal_t        * NEXT;         // signal suivant de même type
} signal_t;
```

Le type pourra prendre une des valeurs du type enuméré suivant :

```
enum sigtype_t {
IN              = 0,
OUT             = 1,
INTERNAL        = 2};
```

Un processus est une expression Booléenne qui définit la valeur du signal de sortie, en fonction des valeurs des signaux d?entrée. Il est caractérisé par un temps de propagation qui est le retard entre un événement sur une entrée et l'événement sur la sortie qui en est la conséquence. Dans le cas général, les temps de propagation dépendent de l'entrée qui commute, mais pour simplifier le simulateur, on suppose que le temps de propagation ne dépend pas de l'entrée qui commute.

```
typedef struct process_t {
boolnet_t       * BOOLNET;      // pointeur sur le réseau Booléen
signal_t        * SIGNAL;       // pointeur sur le signal de sortie
bip_t           * ABL;          // pointeur sur l?arbre ABL associé
long            DELAY;          // retard entrée -> sortie (en ps)
bip_t           * SUPPORT;      // liste des signaux d?entrée
process_t       * NEXT;         // processus suivant
} process_t
```

**A2) échéancier**

L'échéancier permet d'enregistrer et d'ordonner les événements dans le temps. Ceux-ci sont donc rangés dans une liste doublement chaînée, par dates croissantes. Un événement représente une affection de valeur à un signal à une certaine date.

```
typedef struct event_t {
  signal_t  * SIGNAL;   // signal modifié
  unsigned    VALUE;    // nouvelle valeur
  long        DATE;     // date de la modification
  event_t   * PREV ;    // événement précédent dans l?echéancier
  event_t   * NEXT;     // événement suivant dans l?echéancier
} event_t;
```

L'échéancier peut contenir plusieurs événements à la même DATE, mais portant sur des signaux différents. Ces événements synchrones sont rangés consécutivement dans la liste chaînée, mais l'ordre entre ces événements synchrones n'est pas significatif. L'échéancier représente la liste ordonnée des événements passés, présents, et futurs. Il contient donc en particulier la variable TC qui représente la date courante.

```
typedef struct scheduler_t {
  long        TC;       // Temps Courant
  event_t   * CURRENT;  // pointeur sur le premier événement à a date TC
  event_t   * FIRST;    // pointeur sur le premier événement de l'échéancier
} scheduler_t;
```

# B) Fonctions d'accès aux structures de données

On présente ici les différentes fonctions permettant d?accéder aux structures de données définies ci-dessus.

Dans un premier temps le code binaire exécutable de ces fonction vous est fourni, et vous pourrez utiliser ces fonctions pour construire en mémoire le réseau Booléen, construire et initialiser l?échéancier, et écrire la boucle principale de simulation qui a été présentée en cours.

Dans un deuxième temps, il vous sera demandé d?écrire vous-même le code de ces fonctions.

**B1) construction réseau Booléen**

```
boolnet_t * cons_boolnet (char * name)
```

Cette fonction construit un réseau Booléen "vide" de nom *name*. Les pointeurs sur les listes de signaux et de processus sont initialisés à NULL.

```
signal_t * cons_signal (boolnet_t * bn, sigtype_t type, var_t * var)
```

Cette fonction prend pour argument une variable Booléenne préalablement créée. Elle crée un signal et l?insère dans la liste des signaux du réseau correspondant à son type. Elle initialise à NULL les pointeurs FROM_PROCESS et TO_PROCESS. La valeur de la variable Booléenne est forcée à U.

```
process_t * cons_process (boolnet_t * bn, signal_t * sig, char * abl, long delay)
```

Cette fonction prend pour argument un pointeur sur le signal de sortie, un pointeur sur un arbre ABL préalablement créé (en utilisant par exemple la fonction parse_abl du TME5), et la valeur du temps de propagation. Elle crée une structure process_t, calcule le support de l'abl (sous forme d?une liste de signaux d?entrée), et met à jour les pointeurs contenus dans les structures représentant les signaux impliqués.

**Attention :** La fonction support_abl du TME5 retourne une liste de variables, mais le champs SUPPORT de l'objet process_t doit pointer sur une liste de signaux!

Cette fonction doit vérifier le typage des signaux, c'est-à-dire qu'un signal de type IN ne peut avoir de processus générateur et qu'un signal de type OUT ne peut apparaître dans le support d'une fonction.

**B2) construction échéancier**

```
scheduler_t * cons_scheduler()
```

Cette fonction fabrique un échéancier « vide ». La date courante TC est initialisée à une valeur négative, et les pointeurs FIRST et CURRENT sont initialisés à NULL.

```
event_t * cons_event(scheduler_t * sch, signal_t * sig, long date, unsigned val)
```

Cette fonction construit un événement sur le signal sig, à un certaine date, avec la valeur val, et introduit cet événement dans l?échéancier. Cette fonction met à jour les deux pointeurs FIRST et CURRENT dans l?échéancier, si cela est nécessaire. **Attention** : Cette fonction ne doit être utilisée que dans la phase d?initialisation, pour introduire dans l?échéancier les événements portant sur les signaux d?entrée du circuit (stimuli).

**B3) simulation**

```
void add_event (scheduler_t * sch, signal_t * sig, long delta, unsigned val)
```

Cette fonction construit un événement portant sur le signal sig, à la date TC + delta, avec la valeur val, et le range dans l'échéancier. A la différence de la précédente, cette fonction utilise un temps relatif (delta), et ne modifie pas le les pointeurs FIRST et CURRENT. C?est cette fonction qui doit être utilisée dans la boucle de simulation. Dans le cas général, cette fonction doit en principe vérifier qu?il n?existe pas un autre événement postérieur sur le même signal, et retirer cet événement parasite s?il y a lieu. Dans notre cas, cela n?est pas nécessaire. Cette importante simplification est dûe à l'hypothèse qu'un processus est caractérisé par un unique temps de propagation, qui ne dépend pas de l'entrée qui commute. Avec cette hypothèse, tous les événements prévus se produisent effectivement.

```
bip_t * get_events (scheduler_t * sch)
```

Cette fonction recherche dans l'échéancier les événements prévus au temps TC. Il peut y avoir plusieurs événements à la même date. Elle renvoie ces événements sous forme d'une liste chaînée de bi-pointeurs (qu'il faut penser à libérer quand ils ne sont plus utiles).

```
unsigned update_tc(scheduler_t * sch)
```

Cette fonction modifie la valeur de la variable TC en lui donnant la valeur de la date du premier événement strictement postérieur au temps courant. Elle met à jour le pointeur CURRENT de l?échéancier. Cette fonction

renvoie la valeur 0 quand elle ne trouve pas d?événement postérieur au temps courant, ce qui correspond à la fin de la simulation. Cette valeur doit donc être testée à chaque itération de la boucle de simulation.

```
bip_t * update_signals ( bip_t * events)
```

Cette fonction prend pour argument un pointeur sur une liste d'événements (liste de bipointeurs). Elle modifie la valeur des variables Booléennes associées à chacun des signaux pour lesquels il existe un événement. Elle rend la liste des signaux qui ont été modifiés (une autre liste de bipointeurs, qu?il faut également libérer quand ils ne sont plus utilisés).

```
bip_t * get_processes (boolnet_t * bn, bip_t * signals)
```

Cette fonction prend comme argument un pointeur sur le réseau Booléen, et une liste de signaux, et elle renvoie un pointeur sur une autre liste contenant l?ensemble de tous les processus ayant au moins un signal d?entrée appartenant à l?ensemble *signals*. (ici encore, il faut penser à libérer la mémoire occupée par les bi-pointeurs quand ils ne sont plus utiles).

```
void eval_processes( scheduler_t * sch, bip_t * p)
```

Cette fonction prend pour argument une liste de processus, ainsi qu?un pointeur sur le scheduler, et évalue la valeur du signal de sortie pour chacun des processus de la liste, en fonction des valeurs des signaux d?entrée. Elle utilise pour cela une version de la fonction eval_abl() modifiée pour traiter une logique à trois valeurs (0,1,U). Elle compare la valeur future du signal de sortie à sa valeur présente, et insère un événement dans l?échéancier si cette valeur change (en utilisant la fonction add_event).

# C) Travail à réaliser

On se propose de simuler le schéma suivant, qui réalise un additionneur 2 bits. A chaque porte est associé une expression Booléenne représentée par un arbre ABL.

## C1) simulation du circuit *etou*

Le fichier *etou.c* contient un tout petit réseau Booléen ne contenant que deux noeuds, et 4 signaux. Compilez ce fihier, et exécutez la simulation.

```
> make
```

Lancez l'exécution duprogramme (construction du réseau Booléen et simulation).

```
> ./etou
```

Vous pouvez visualiser le réseau Booléen avec la commande:

```
> xv etou.gif
```

Vous pouvez visualiser le chronogramme résulat de la simulation avec la commande:

```
> xpat -l etou
```

## C2) Construction réseau Booléen de l'additionneur

En vous inspirant du fichier *etou.c*, écrivez le fichier *adder.c* qui construit en mémoire le réseau Booléen correspondant au circuit additionneur 2 bits décrit ci-dessus. On utilisera pour cela les fonctions cons_boolnet(), cons_process() et cons_signal(). On prendra une valeur de 1 ns pour le temps de propagation de la porte NAND2, et de 2 ns pour la porte XOR2. Pour vérifier la structure du réseau Booléen, on utilisera la fonction *drive_boolnet()*. Cette fonction construit une représentation graphique du réseau Booléen, et la sauvegarde dans un fichier au format .gif ou .ps.

Modifiez le Makefile permettant de compiler ce programme *adder.c*, et exécutez-le.

## C3) Construction et initialisation de l?échéancier

Complêter le fichier *adder.c* main() pour créer l?échéancier et initialiser les événements sur les signaux d? entrée a0, b0, c0, a1 et b1 de façon à respecter le chronogramme ci-dessous. On utilisera les fonctions cons-scheduler() et add_event(). Attention : le passage de la valeur U (indéfinie) à une valeur 0 ou 1 constitue un événement : Dans ce chronogramme, il y a donc un événement sur tous les signaux d?entrée au temps T = 0.

Pour vérifier que l?échéancier est correctement initialisé, on pourra utiliser la fonction *drive_scheduler ()*. Cette fonction peut être utilisée avant même l'exécution de la fonction de simulation, pour générer un fichier au format .pat qui décrit le chronogramme des signaux d'entrée. Vous pouvez visualiser ce chronogramme avec l?outil XPAT.

## C4) Simulation effective du circuit additionneur

Introduisez dans le fichier *adder.c* la fonction *simulate()* qui effectue la simulation du réseau Booléen, jusqu'à ce qu'il n'y ait plus aucun événement à traiter dans l'échéancier. Compilez ce programme, et analysez le chronogramme résultant

## C5) Ecriture de la boucle de simulation

Ecrire en langage C la fonction *simulate()*. Cette fonction contient une boucle *while* qui enchaîne les deux phases ?update? et ?exécute? de l'algorithme de simulation *event-driven* présenté en cours.

```
void simulate (boolnet_t * bn, scheduler_t * sch)
```

On utilisera pour cela les fonctions get_events(), update_tc(), update_signals(), get_process() et eval_process(). On sauvera sur disque le contenu de l?échéancier en sortie de la boucle de simulation en utilisant la fonction drive_scheduler(), de façon à visualiser le chronogramme avec l?outil XPAT.

## C6) Ecriture des fonctions d?accès

Bien que le texte de cette question soit très court, cette question est évidemment la plus importante du TME : Ecrivez vous-même le code des 11 fonctions d?accès aux structures de données du simulateur décrites dans la section B, et introduire progressivement votre code à la place des fichiers .o qui vous ont été fournis.

# Compte-Rendu

Il ne vous est pas demandé de compte-rendu écrit pour ce TME, mais vous devrez faire une démonstration de votre code au début du prochain TME.