Programmation synchrone (PSYN9)
TP3 : Manipulation de tableaux

Exercice 1 – *Buffer circulaire avec détection d'erreur*

Durant le cours, nous avons vu comment réaliser un buffer circulaire, c'est à dire un canal mono-producteur mono-consommateur implémenté via un tableau de taille n fixe et une paire d'indice évoluant dans $\mathbb{Z}/n\mathbb{Z}$. Si le rythme auquel le producteur écrit est trop différent de celui auquel le consommateur lit, deux types d'erreurs peuvent se produire :

- un *dépassement vers le haut* (en anglais *buffer overflow*) se produit lorsque toutes les cases du tableau sont occupées mais que le producteur indique une nouvelle valeur à stocker,
- un *dépassement vers le bas* (en anglais *buffer underflow* en anglais) se produit lorsque toutes les cases du tableau sont libres mais que le consommateur indique vouloir lire une nouvelle valeur.

Le but de cet exercice est de réaliser une variante évoluée du buffer circulaire vu en cours qui soit capable de détecteur ces erreurs.

1. Relisez et recopiez le buffer circulaire produit dans les notes de cours.

```
node ring_buffer<<n : int>>(e : int; w, r : bool) returns (o : int)
```

Décrivez le rôle de chaque entrée et chaque sortie.

2. Simulez `ring_buffer<<3>>` de manière à compléter les valeurs de la sortie `o` aux instants où `r` est vraie dans les chronogrammes ci-dessous.

e	12	5	8	...
w	1	0	1	...
r	1	1	1	...
o				...

e	42	1	13	...
w	1	0	1	...
r	0	1	0	...
o	×		×	...

e	8	9	10	5	3	4	7	...
w	1	1	1	1	0	0	0	...
r	0	0	0	0	1	1	1	...
o	×	×	×	×				...

Pour chaque chronogramme, indiquez le premier instant auquel se produit un éventuel dépassement, en précisant s'il s'agit d'un dépassement vers le haut ou vers le bas.

3. Proposez une variante du buffer circulaire implémentant l'interface suivante.

```
node ring_buffer_checked<<n : int>>(e : int; w, r : bool)
                            returns (o : int; of, uf : bool)
```

La seule différence entre ce nœud et `ring_buffer` doit être la présence des sorties supplémentaires `of` et `uf`. La sortie `of` doit être vraie à l'instant où un dépassement vers le haut se produit, de même pour `uf` et les dépassements vers le bas.

Indication : détecter les dépassements est plus simple si l'on représente les indices de lecture et d'écriture comme des entiers appartenant à $\mathbb{N}$ plutôt qu'à $\mathbb{Z}/n\mathbb{Z}$.

4. Le nœud précédent détecte les erreurs mais ne les ignore pas. Proposez une variante

```
node ring_buffer_safe<<n : int>>(e : int; w, r : bool)
                        returns (o : int; of, uf : bool)
```

qui, en plus de signaler les erreurs, ne prend pas en compte les écritures aux instants de dépassements vers le haut, ni les lectures aux instants de dépassement vers le bas. Vérifiez son comportement sur les chronogrammes de la question 2.

Exercice 2 – *Montre numérique*

Dans cet exercice, on souhaite bâtir une montre numérique à partir des exercices des semaines précédentes. Cette montre dispose de trois modes : affichage de l'heure courante, chronomètre, et réglage.

- Dans le premier mode, elle affiche l'heure courante sous la forme heure/minute/seconde.
- Dans le second mode, elle affiche et contrôle la sortie d'un chronomètre fonctionnant comme le nœud `chrono3` réalisé précédemment, mais qui sera maintenant affichée dans le format minute/seconde/centièmes de seconde.
- Dans le troisième mode, elle permet de régler les heure, minute, et seconde courantes.

Par rapport au chronomètre `chrono3`, la montre dispose d'une entrée supplémentaire, un flot booléen `mode`. Sa sortie sera un flot de tableaux de taille trois d'entiers.

Son fonctionnement est le suivant :

- au début, elle affiche l'heure, initialisée à 00 : 00 : 00 ;
- une pression sur le bouton `mode` la fait passer d'un mode au suivant, dans l'ordre : heure – chronomètre – réglage – heure – ...
- dans le mode affichage de l'heure, une pression sur les boutons `rst`, `start_stop` ou `pause` n'a pas d'effet ;
- dans le mode chronomètre, elle se comporte comme `chrono3` ;
- dans le mode de réglage de l'heure, on règle successivement les heures, minutes et secondes courantes :
  - une pression sur `start_stop` incrémente l'entier correspondant (modulo 24 pour les heures, modulo 60 pour les minutes et les secondes),
  - une pression sur `pause` fait passer des heures aux minutes puis des minutes aux secondes, puis des secondes aux heures et
  - une pression sur `rst` réinitialise l'heure à 00 : 00 : 00.
- Dans le mode de réglage de l'heure, l'heure de la montre continue de tourner.

Programmez un nœud `montre` obéissant à la spécification ci-dessus.

Pour la simulation, on définira un nœud `affiche_montre` pour faire un affichage confortable de la simulation de la montre numérique et appellera `montre` avec pour l'entrée `hs` le flot de booléens `true`.

Exercice 3 – *Gaz carbonique d'ordre supérieur*

Le but de cet exercice est de programmer un capteur qui vérifie que la concentration en CO2 des `m` salles d'un bâtiment reste inférieure à 0,05. On dispose d'une mesure de CO2 par salle. Pour chacun des nœuds `noeud` suivants, on programmera également une instance `noeud_5` pour faire une simulation dans le cas de 5 salles.

1. Programmez un nœud d'interface

```
node alertes_co2<<m:int>>(co2 : float^m) returns (alertes : bool^m)
```

tel que, à chaque instant, `alerte[i]` est vrai ssi `co2[i]` est strictement plus grand que 0,05.

2. Programmez un nœud d'interface

```
node nb_alertes_co2<<m: int>>(co2 : float^m) returns (nb_alertes : int)
```

tel que, à chaque instant, `nb_alertes` est le nombre d'alertes détectées.

3. En plus du nombre d'alertes courantes, on souhaite à des fins statistiques calculer le nombre moyen d'alertes par instant depuis le début du système. Programmez un nœud d'interface

```
node stats_co2<<m: int>>(co2 : float^m) returns (nb_alertes, moy_alertes :
    int)
```

tel que `nb_alertes` est le nombre d'alertes détectées et `moy_alertes` est le nombre d'alertes moyen par instant depuis le début de l'exécution du système.

Exercice 4 – *Éclairage et ordre supérieur*

On s'intéresse maintenant à tester la luminosité des `m` salles d'un bâtiment, représentées comme dans l'exercice prévédent. Les ticks d'horloge ont lieu chaque seconde, et à chaque tick on mesure `lux[i]`, la luminosité de la salle `i`. La variable `lux : float^m` sera l'entrée de vos opérateurs Heptagon. Pour chacun des nœuds `noeud` suivants, on programmera également une instance `noeud_5` pour faire une simulation dans le cas de 5 salles.

1. Calculez à chaque tick la luminosité moyenne `lm : float` de toutes les salles.

```
node luminosite_moyenne<<m: int>>(lux : float^m) returns (lm : float)
```

2. Soit `sombres[i]` vraie ssi `lux[i] < 20`. Calculez à chaque tick le tableau `sombres`.

```
node sombreur<<m: int>>(lux : float^m) returns (sombres : bool^m)
```

3. Dans cette question, on suppose que `m`> 3. Calculez la sortie `obscur : int` qui compte le nombre de ticks depuis le démarrage du système pendant lesquels la salle numéro 3 était sombre.

```
node obscurite<<m:int>>(lux : float^m) returns (obscur : int)
```

Que se passe-t-il si on instancie ce nœud dans le cas de 3 salles ou moins ?