Une approche synchrone à la conception de systèmes embarqués temps réel

Dumitru Potop-Butucaru

dumitru.potop@inria.fr

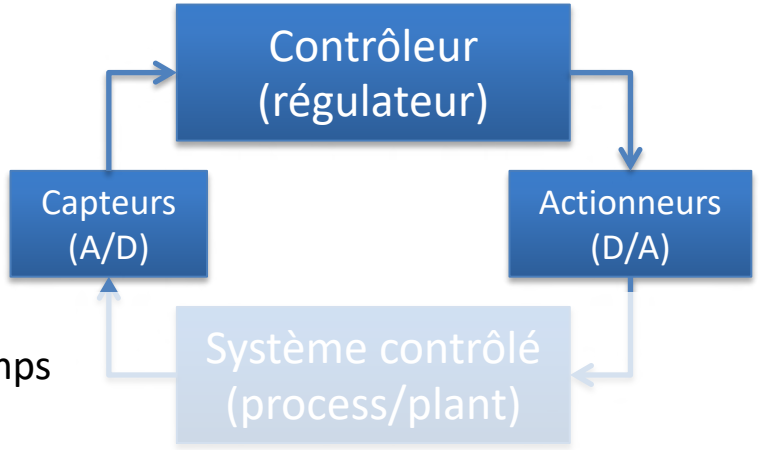
cours EPITA, 2022, 3ème séance

Contenu

- Comment construit-on une spécification fonctionnelle ?
  - Cas d'étude GNC (avionique aérospatiale)
  - Programmation de GNC
- Programmation Heptagon
  - Types de données structurés
  - Paramètres statiques
- Préparation du TP
  - Programmation de l’exemple GNC
  - La FFT (Fast Fourier Transform)
  - Discussion des rendus précédents

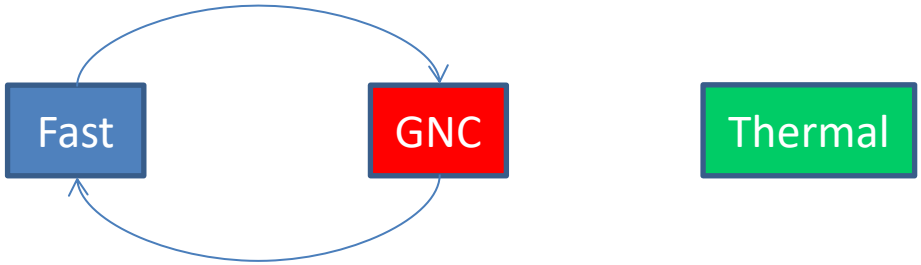
Spécification d'un contrôleur

- Que faut-il représenter ?
  - Calculs cycliques
  - État
  - Multi-tâches
    - Modularité, concurrence, temps réel
  - Exécution conditionnelle
    - Multi-périodes
      - On ne peut pas tout exécuter tout le temps
      - Actionneurs
  - Communication entre tâches
    - Mémoire partagée (multi-thread, multi-coeurs)
    - Passage de messages (distribué, dataflow, isolation spatiale)
  - Synchronisation et temps réel
    - Contraintes venant de l'automatique (e.g. freshness – non-fonctionnel converti en fonctionnel)
    - Préservation de sémantique (parallélisation)



Exemple GNC

- 3 tâches Fast, GNC, Thermal
  - Fast et GNC communiquent



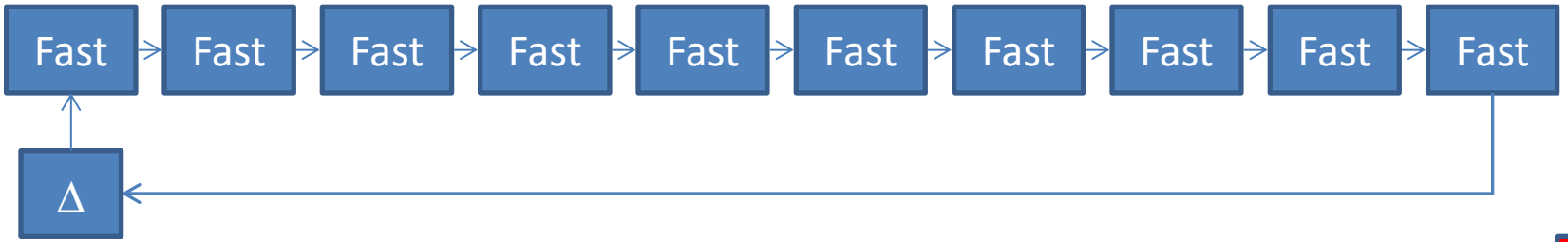
Exemple GNC

- 3 tâches Fast, GNC, Thermal
  - Fast et GNC communiquent
- Periodes: 100ms, 1s, 1s
  - Fast = I/O et contrôle en boucle rapide
  - GNC = Guidance, navigation, and control
  - Thermal = gestion thermique

https://en.wikipedia.org/wiki/Guidance,_navigation,_and_control
http://manuscript.elsevier.com/S0094576515003811/pdf/S0094576515003811.pdf
https://en.wikipedia.org/wiki/Spacecraft_thermal_control

Exemple GNC

- Periodes: 100ms, 1s, 1s
  - Chaque seconde, 10 instances de Fast et 1 instance de GNC et de Thermal
    - Hyper-période = PPCM(périodes des tâches)
  - Fast, GNC, Thermal non-synchronisés au niveau de la spécification

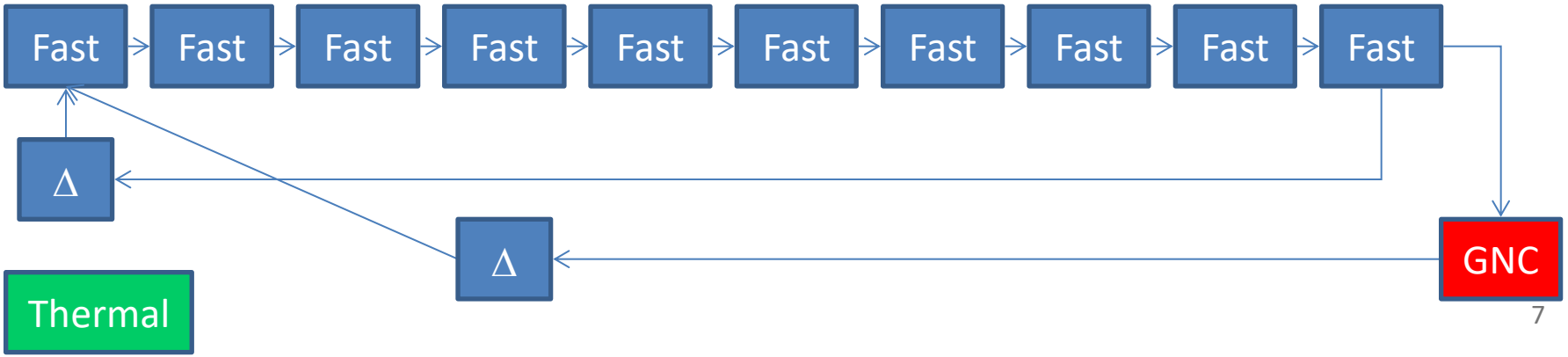


Thermal

GNC

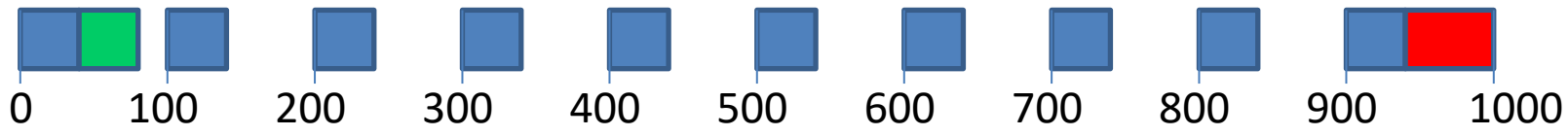
Exemple GNC

- Motif de communication déterministe désiré
  - Artéfact d'implémentation
    - Plusieurs choix possibles
  - Motif de communication sur l'hyper-période
    - Motif qui se répète dans le temps



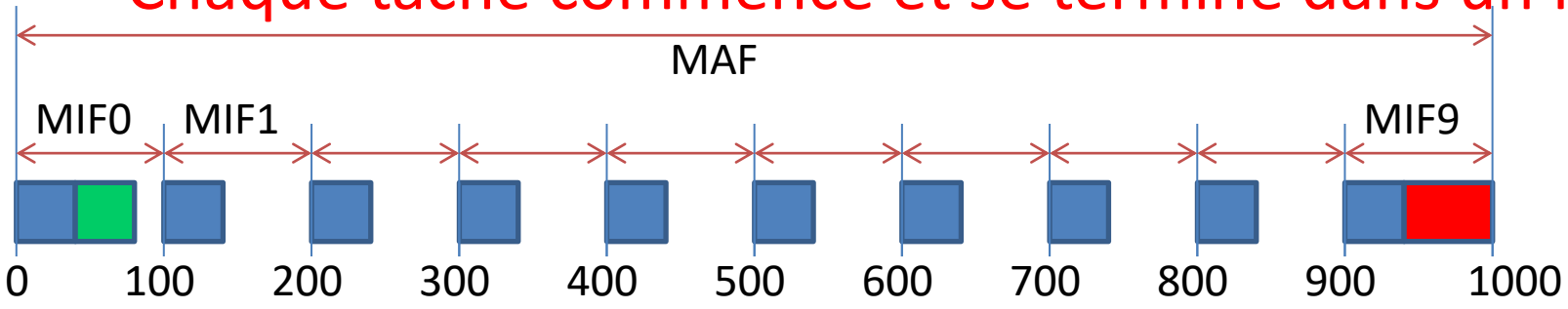
Exemple GNC

- Choix d'implantation classique :
  - Single-core
  - Ordonnancement temps réel "time-triggered"
    - Hypothèse : WCETs: 40ms, 60ms et 40ms
    - Déclenchements sur barrière de 100ms



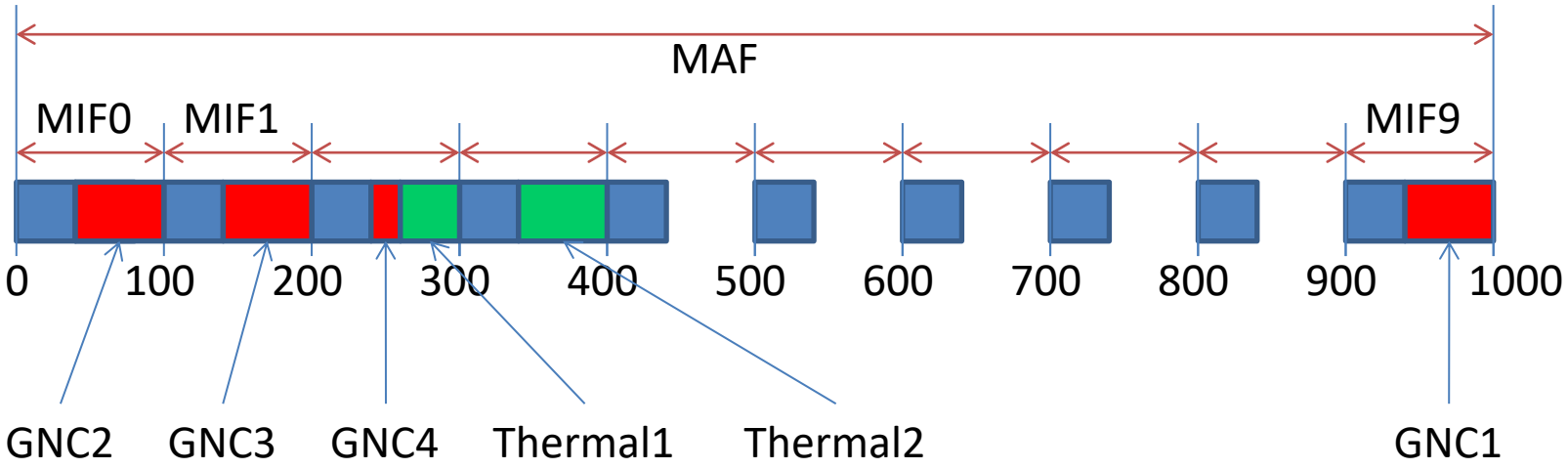
Exemple GNC

- Sous-cas : MIF/MAF
  - Major Frame (MAF) = hyper-période
  - Minor Frame (MIF) = période la plus courte
    - Déclenchement périodique en début de MIF
  - Chaque tâche commence et se termine dans un MIF



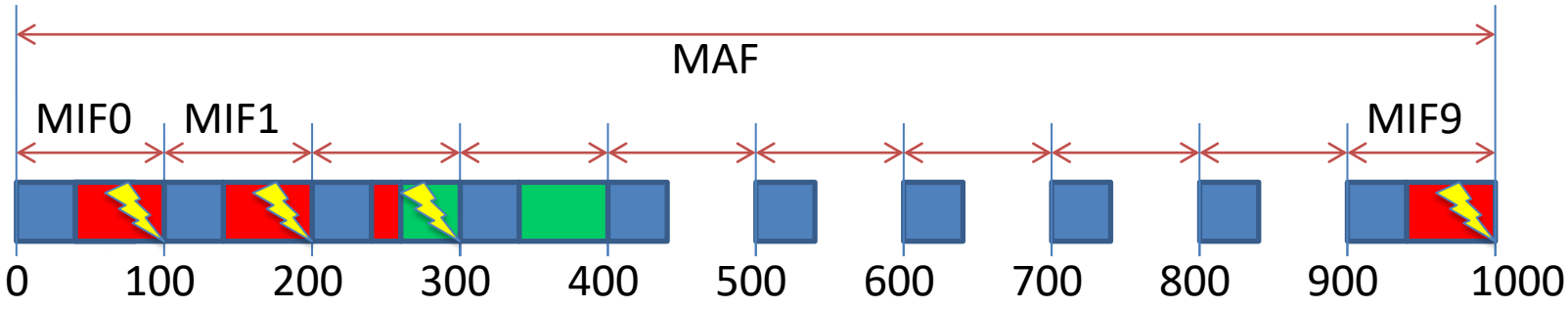
Exemple GNC

- Et si on augmente le WCET de GNC à 200ms et de WCET de Thermal à 100ms (plus réaliste) ?
  - Solution 1 : découper manuellement :
    - GNC -> GNC1, GNC2, GNC3, GNC4
    - Thermal -> Thermal1, Thermal2



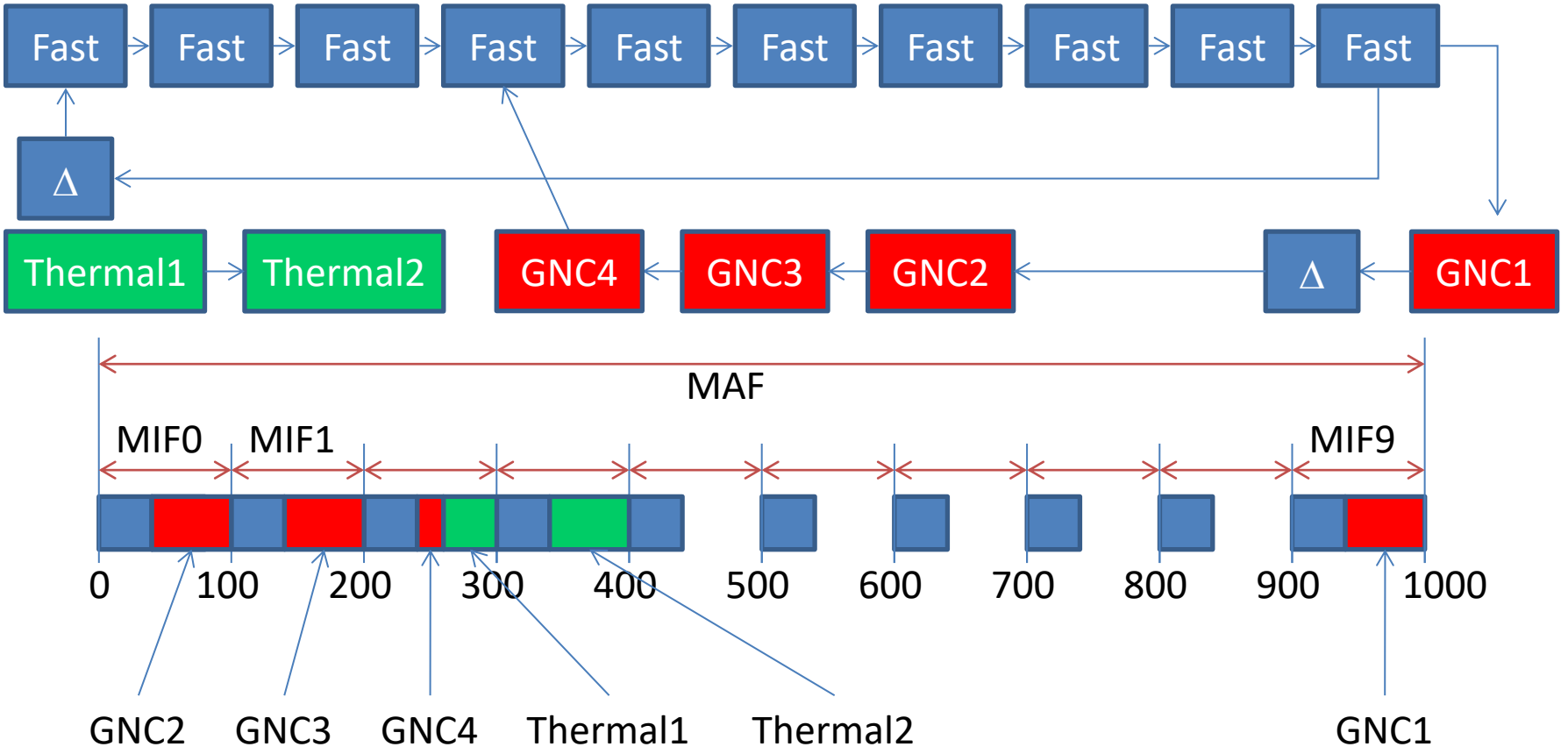
Exemple GNC

- Et si on augmente le WCET de GNC à 200ms et de WCET de Thermal à 100ms (plus réaliste) ?
  - Solution 2 : utiliser un OS préemptif (e.g. ARINC 653)



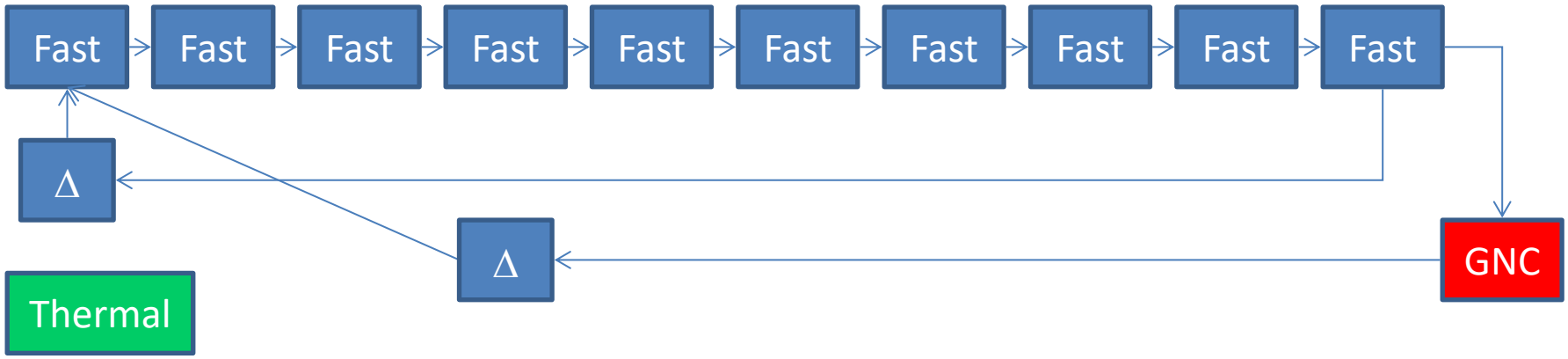
Exemple GNC

- Attention : la fonction change !

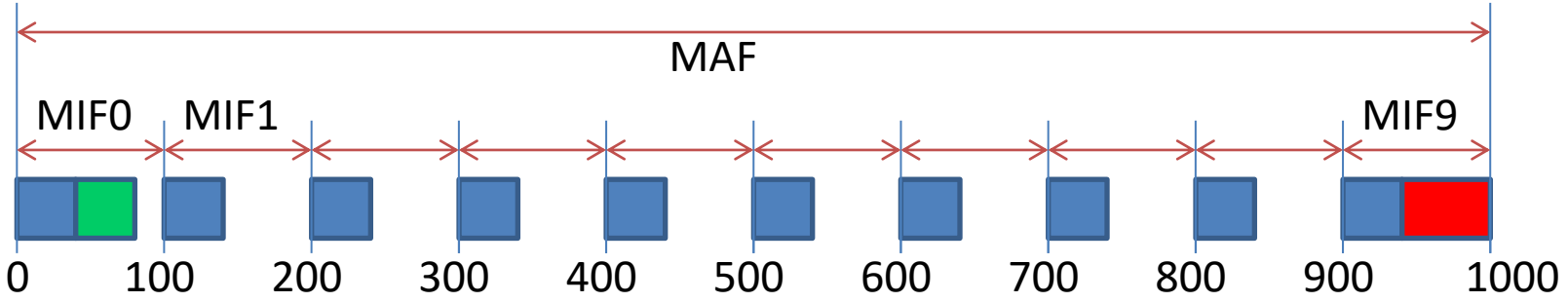


Programmation de l'exemple GNC

- Fonctionnalité

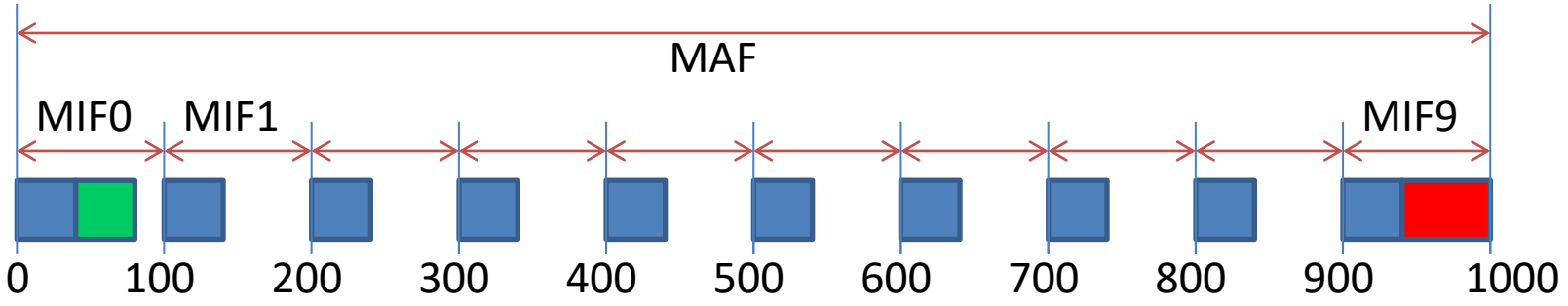


- Implémentation désirée



Programmation de l'exemple GNC

- Programme Heptagon
  - Un cycle du programme synchrone = un MIF
    - Code résultant de la compilation à appeler à chaque MIF
      - Solution complète pour le cas séquentiel
      - Pour du code parallèle, le problème de parallélisation reste non-résolu
    - L'affectation des tâches aux MIFs est faite par le programmeur

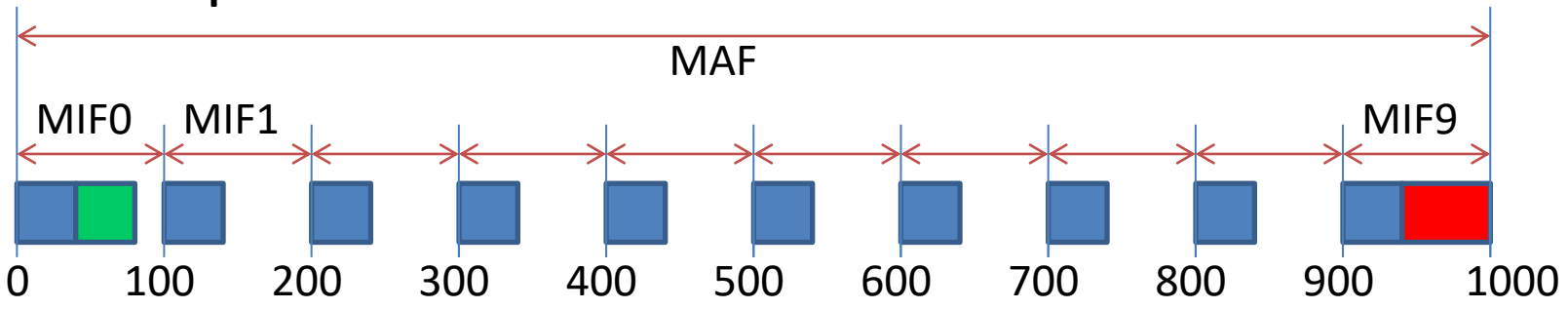


Programmation de l'exemple GNC

- Conditions d'activation
  - MIF modulo counter

```
(* Clock computation *)
mif_cnt = 0 fby ((mif_cnt+1)%10) ;
clk_f = true ;
clk_thermal = (mif_cnt = 0) ;
clk_gnc = (mif_cnt = 9) ;
```

  - f activé à chaque MIF
  - thermal activé au premier MIF de chaque MAF
  - gnc activé au dernier MIF (d'index 9) de chaque MAF, après Fast

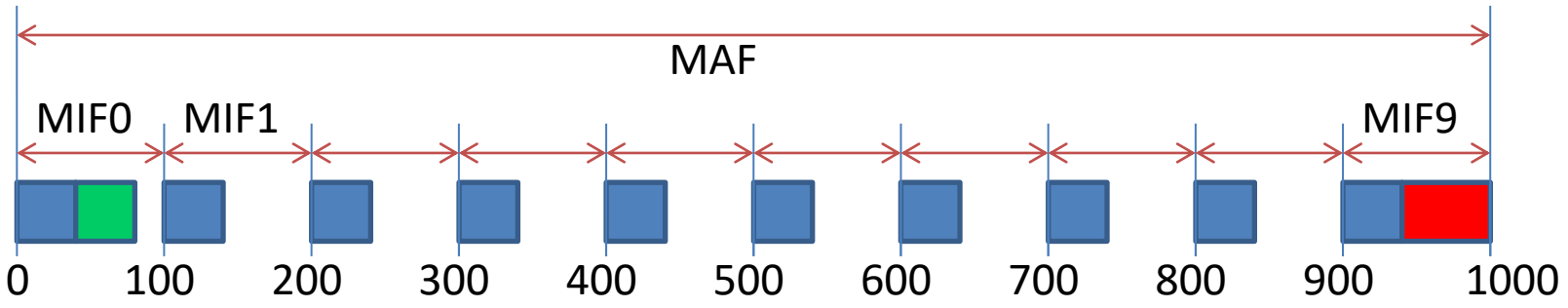


Programmation de l'exemple GNC

- Activation conditionnelle et maintien des valeurs de sortie – construction CONDACT

```
node condact_gnc<<x_init:int>>(c:bool; y:int) returns (x:int)
let
  x = merge c
        (true  -> gnc (y when c))
        (false -> (x_init fby x) whenot c) ;
tel
```

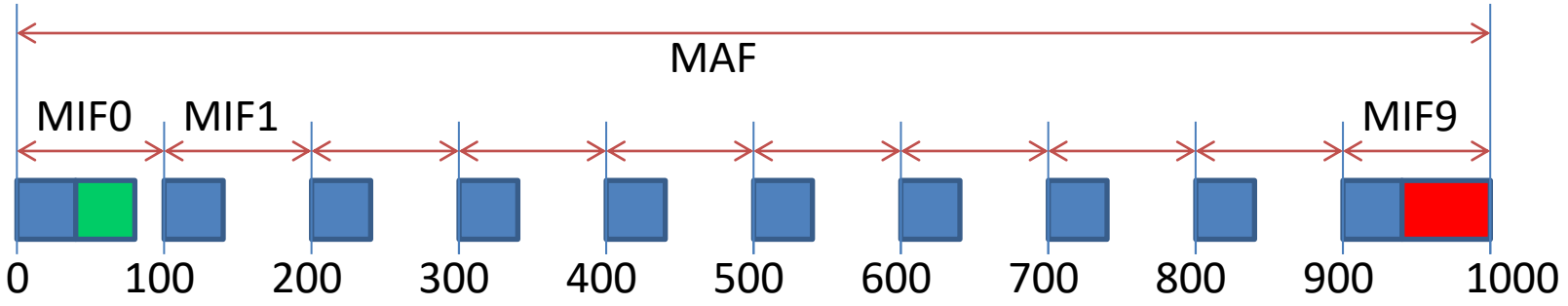
argument statique



Programmation de l'exemple GNC

- Activation conditionnelle et maintien des valeurs de sortie – construction CONDACT

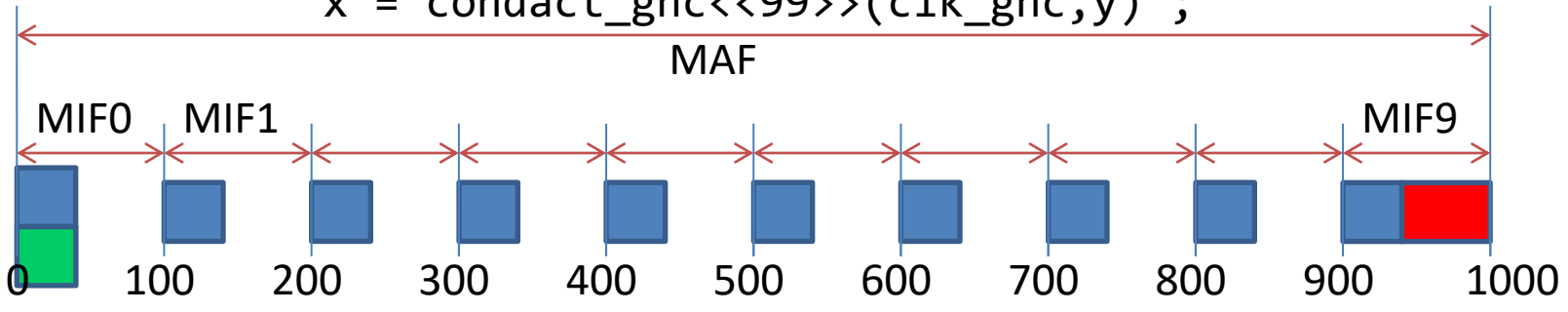
cycle	0	1	2	3	4	5	…
y	10	7	21	3	4	1	…
c	f	t	f	t	f	t	…
gnc(y)		gnc(7)		gnc(3)		gnc(1)	…
condact_gnc<<99>>(c,y)	99	gnc(7)	gnc(7)	gnc(3)	gnc(3)	gnc(1)	…



Programmation de l'exemple GNC

- Flot de données global – attention au MIF 0

```
(* Clock computation *)
mif_cnt = 0 fby ((mif_cnt+1)%10) ;
clk_f = true ;
clk_thermal = (mif_cnt = 0) ;
clk_gnc = (mif_cnt = 9) ;
(* Flot de données *)
y = condact_f<<31>>(clk_f,0 fby x) ;
()= condact_thermal(true when clk_thermal) ;
x = condact_gnc<<99>>(clk_gnc,y) ;
```



Programmation de l'exemple GNC

- Flot de données global

```
node main() returns ()
var mif_cnt,x,y:int; clk_f,clk_gnc,clk_thermal:bool; let
  (* Clock computation *)
  mif_cnt = 0 fby ((mif_cnt+1)%10) ;
  clk_f = true ;
  clk_thermal = (mif_cnt = 0) ;
  clk_gnc = (mif_cnt = 9) ;
  (* Flot de données *)
  y = condact_f<<31>>(clk_f,0 fby x) ;
  ()= condact_thermal(true when clk_thermal) ;
  x = condact_gnc<<99>>(clk_gnc,y) ;
tel
```

Contenu

- Comment construit-on une spécification fonctionnelle ?
  - Cas d'étude GNC (avionique aérospatiale)
  - Programmation de GNC
- Programmation Heptagon
  - Types de données structurés
  - Paramètres statiques
- Préparation du TP
  - Programmation de l’exemple GNC
  - La FFT (Fast Fourier Transform)

Programmation synchrone en Heptagon

- Ce que l'on a défini déjà :
  - Appel de fonctions
  - Hiérarchie
  - État
  - Contrôle simple
    - Opérateur "if"
  - Exécution conditionnelle
    - Horloges logiques (clocks)
    - Opérateurs "when" et "merge"
      - C'est le flot de données qui conditionne un calcul!
    - Sucre syntaxique de style impératif
      - Conversion vers flot de données

Types de données structurées

- Enregistrements

```
type complex =
  {
    re : float ;
    im : float (* no ";" here *)
  }
```

  - Traduction en C par "struct"
  - Accès aux éléments d'un enregistrement :

```
(* Accès à un champ *)
z = x.re ;
(* Définition d'une valeur à partir de var et const *)
(* Attention, c'est 17.0, et non pas 17 ou 17. *)
x = { re = r ; im = 17.0 }
(* Valeur par modification d'une autre *)
u = { x with .im = 36.7 }
```

Types de données structurées

- Vecteurs
  - La declaration est possible, mais pas nécessaire

```
type monvecteur = int^10 (* type alias *)
type monvecteur2d = int^10^15 (* 15 fois 10 entiers *)
...
var
  x : monvecteur ;
  y : int^10 ;...
let
  y = x ;...
```

Types de données structurées

- Vecteurs
  - La declaration est possible, mais pas nécessaire

```
type monvecteur = int^10 (* type alias *)
type monvecteur2d = int^10^15 (* 15 fois 10 entiers *)
...
var
  x : monvecteur ;
  y : int^10 ;...
let
  y = x ;...
```

  - Traduction en vecteurs et pointeurs du langage C

```
// myfile_types.h
typedef int Myfile__monvecteur[10] ;
// myfile.c
for(i=0;i<10;++i) { y[i] = x[i]; } /* copie de vecteur*/
```

Types de données structurées

- Vecteurs
  - Peuvent être arguments de noeuds
    - Transmis par référence dans le code généré
      - mais un vecteur C est déjà une référence

```
// Heptagon definitions
type monvecteur = int^10 (* type alias *)
node mynode(i1:int;i2:monvecteur) returns (o:monvecteur)

// C code – generated or hand-written
void Myfile__mynode_step(int i1;
                         Myfile__monvecteur i2;
                         Myfile__monvecteur o;) {...}
```

  - **Attention à la définitions de fonctions externes !**

Types de données structurées

- Vecteurs
  - Accès aux champs d'un vecteur `x:int^10`
    - Sûreté par construction
      - Pas d'accès en dehors des limites du tableau
    - Accès par index constant

```
y = x[4] ; (* Accès par indice constant => OK *)
z = x[10] ; (* Indice erronné, rejeté par heptc *)
```

    - Accès par index variable (deux variantes) – code coûteux

```
i = f(j) ;
t = x[>i<] ; (* Accès par indice variable. Vérification
              * dynamique. Si <0, x[0], si >=4, x[3] *)
u = x.[i] default 25.0 ; (* Indice variable. Si erronné,
                            25.0 *)
```

Types de données structurées

- Vecteurs
  - Construction de vecteurs :

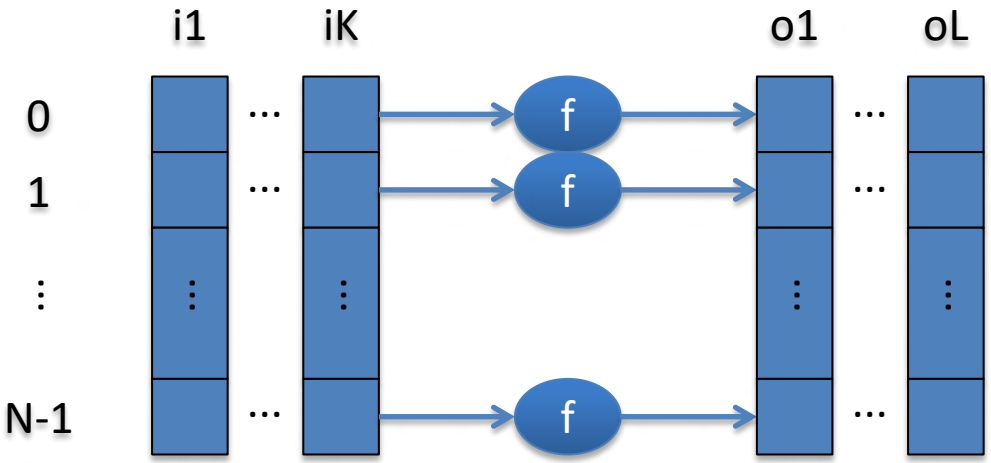
```
const x:float^4 = [ 0.5, 1.5, (-.2.0), 3.4 ]
const c:complex^2 = [ { re = 0.7 ; im = 45.6 },
                      { re = 6.3 ; im =  0.9 } ]


(* attention aux opérations sur flottants *)
y = [ (c[1]).re +. 23.0,
      a -. b,
      345.66 ] ;

(* concaténation de vecteurs *)
u = c@z ;
```

Types de données structurées

- Vecteurs
  - Manipulation de vecteurs : itérateur **map**




```
(* Function signature *)
fun f(i1:t1;...;iK:tK) returns (o1:tt1;...;oL:ttL)

(* Use of iterator *)
(o1,...,oL) = map<<N>> f(i1,...,iK) ;
```

Types de données structurées

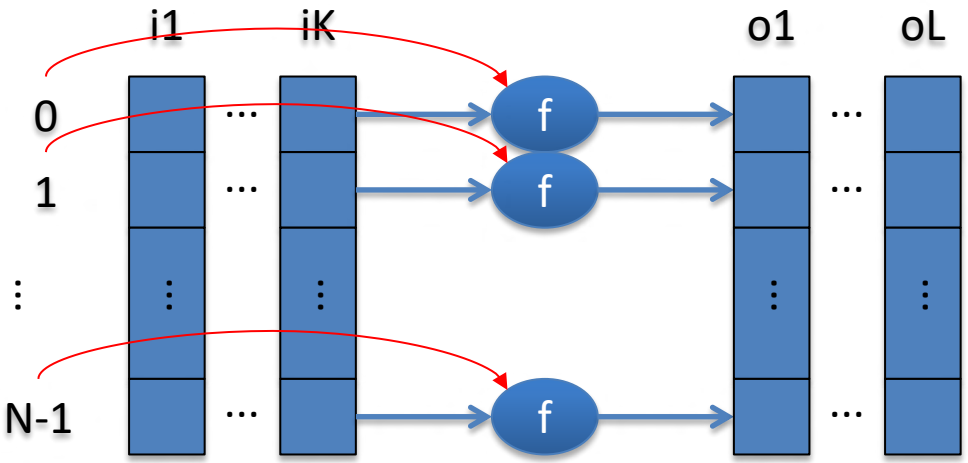
- Vecteurs
  - Manipulation de vecteurs : itérateur **map**

```
i1 = [20, 44, 10 ] ;
i2 = [13, 16, 77 ] ;
...
(* Calcule y[i] = f(i1[i],i2[i],i2[i]), i=0..2 *)
y = map<<3>> f (i1,i2,i2) ;
...
(* Calcule x[i] = i1[i]+i2[i] , i=0..2 *)
x = map<<3>> (+) (i1,i2) ;
...
(* Calcule (m[i],n[i]) = g(i1[i],y[i],x[i]) , i=0..2 *)
(m,n) = map<<3>> g (i1,y,x) ;
```

  - Taille du map et des vecteurs doivent être identiques
    - Approche "ceinture et bretelles"

Types de données structurées

- Vecteurs
  - Manipulation de vecteurs : itérateur **mapi**




```
(* Function signature *)
fun f(i:int;i1:t1;...;iK:tK) returns (o1:tt1;...;oL:ttL)

(* Use of iterator *)
(o1,...,oL) = mapi<<N>> f(i1,...,iK) ;
```

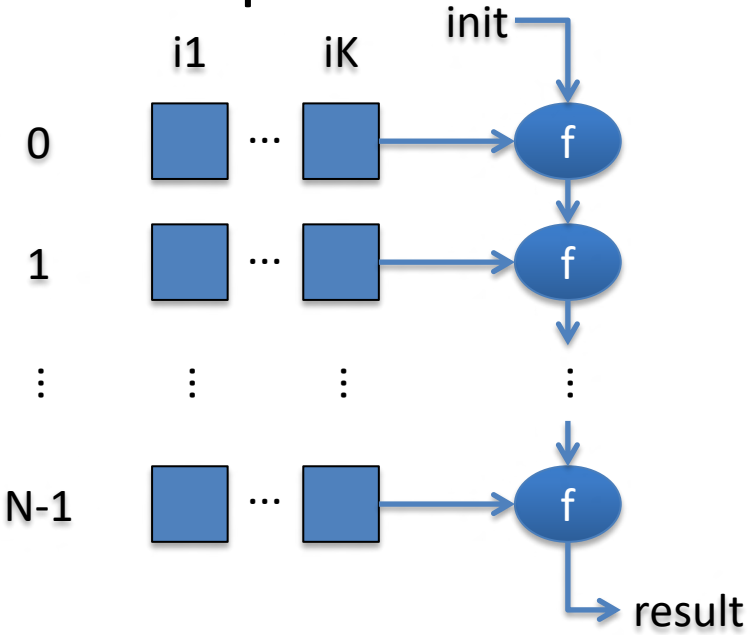
Types de données structurées

- Vecteurs
  - Manipulation de vecteurs : itérateur **mapi**

```
i1 = [20, 44, 10 ] ;
i2 = [13, 16, 77 ] ;
...
(* Calcule y[i] = ff(i,i1[i],i2[i],i2[i]) *)
y = mapi<<3>> ff (i1,i2,i2) ;
...
(* Calcule x[i] = i+i2[i] *)
x = mapi<<3>> (+) (i2) ;
...
(* Calcule (m[i],n[i]) = gg(i,i1[i],y[i],x[i]) *)
(m,n) = mapi<<3>> gg (i1,y,x) ;
...
fun h(x:int) returns (y:int) let y = x tel
...
z = mapi<<3>> h () ; (* combien vaudra z ? *)
```

Types de données structurées

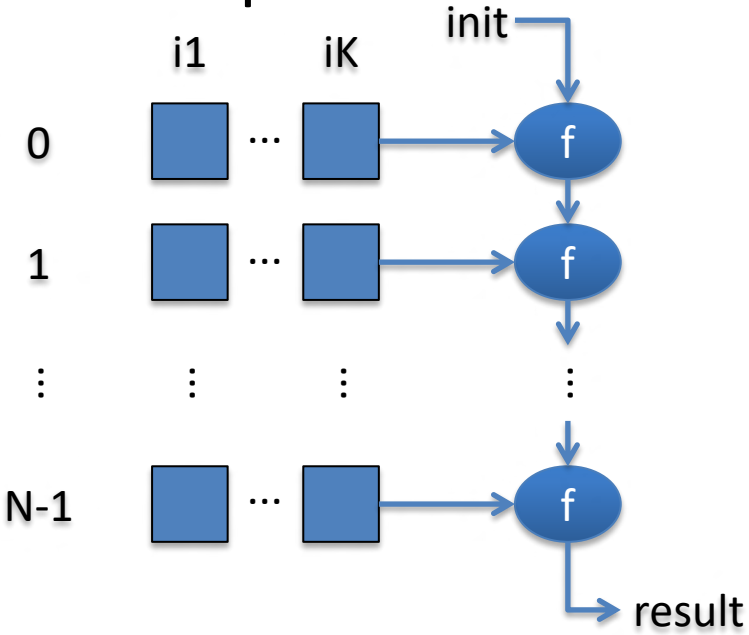
- Vecteurs
  - Manipulation de vecteurs : itérateur **fold**




```
(* prototype de f *)
fun f(i1:t1;...;iK:tK;acc:t) returns (res:t)
(* Application *)
r = fold<<N>> f (i1,...,iK,init) ;
```

Types de données structurées

- Vecteurs
  - Manipulation de vecteurs : itérateur **fold**

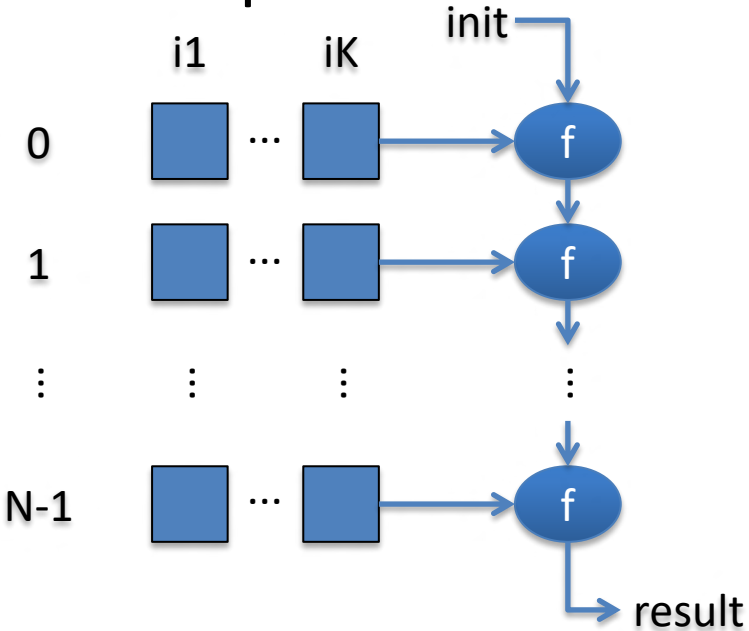



```
y = [0, 1, 2] ;
(* Calcule z = mf(y[2],
                  mf(y[1],
                     mf(y[0],123)
                  )
               ) *)
z = fold<<3>> mf (y,123) ;
```

```
(* prototype de f *)
fun f(i1:t1;...;iK:tK;acc:t) returns (res:t)
(* Application *)
r = fold<<N>> f (i1,...,iK,init) ;
```

Types de données structurées

- Vecteurs
  - Manipulation de vecteurs : itérateur **fold**



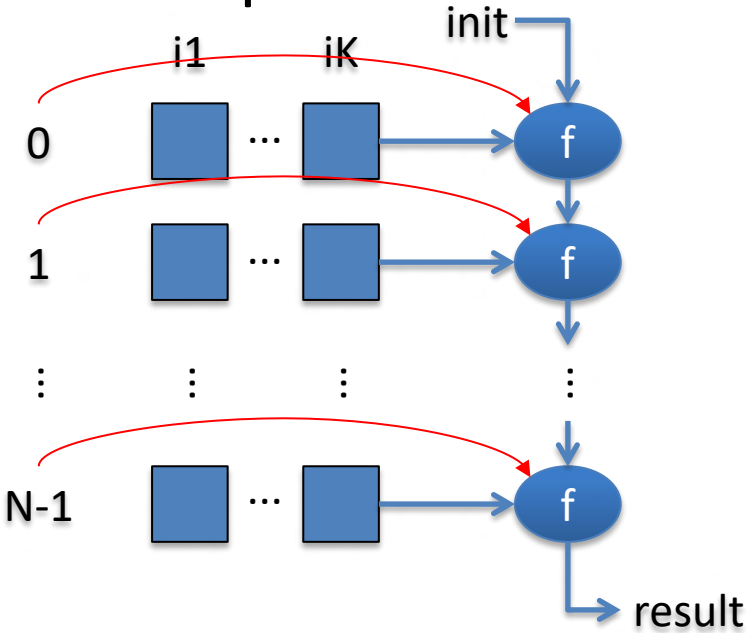

```
y = [0, 1, 2] ;
(* Calcule z = mf(y[2],
                  mf(y[1],
                     mf(y[0],123)
                  )
               ) *)
z = fold<<3>> mf (y,123) ;

(* Exercice : Que calcule la
   ligne suivante ? *)
x = fold<<3>> (+) (y,0) ;
```

```
(* prototype de f *)
fun f(i1:t1;...;iK:tK;acc:t) returns (res:t)
(* Application *)
r = fold<<N>> f (i1,...,iK,init) ;
```

Types de données structurées

- Vecteurs
  - Manipulation de vecteurs : itérateur **foldi**

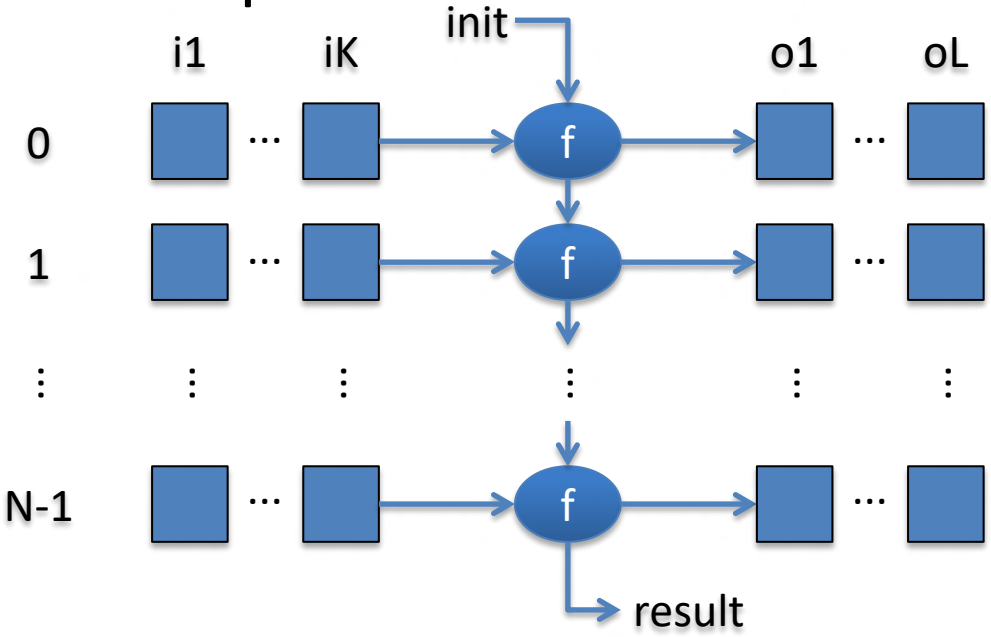



```
y = [0, 1, 2] ;
(* Calcule z = mf(2,y[2],
                  mf(1,y[1],
                     mf(0,y[0],123)
                  )
               ) *)
z = foldi<<3>> mf (y,123) ;
```

```
(* prototype de f *)
fun f(i:int;i1:t1;...;iK:tK;acc:t) returns (res:t)
(* Application *)
r = foldi<<N>> f (i1,...,iK,init) ;
```

Types de données structurées

- Vecteurs
  - Manipulation de vecteurs : itérateur **mapfold**




```
(* Function signature *)
fun f(i1:t1;...;iK:tK;acc:t) returns (o1:tt1;...;oL:ttL;res:t)
(* Use of iterator *)
(o1,...,oL,res) = mapfold<<N>> f(i1,...,iK,init) ;
```

Paramètres statiques

- Une fonction ou un noeud peut avoir des paramètres statiques
  - Ne changent pas entre les appels successifs d'un noeud
  - Spécifient : tailles de tableaux, constantes

```
node fp<<m:int>>(a:int^m;b:int^m) returns (o:int^m)
let
  o =map<<m>> (+)(a, b);
tel
```

  - Un paramètre statique ne peut pas dépendre d'un autre

```
node bad<<m:int;b:int^m>>(a:int^m) returns (o:int^m)
let
  o =map<<m>> (+)(a, b);
tel
```

Paramètres statiques

- Un vecteur peut avoir la taille définie avec :
  - Une constante nommée (globale)
  - Un paramètre statique du noeud

- Note: une constante vecteur non-initialisée d'un fichier .ept doit être initialisée en C
  - Comme si elle était définie dans un fichier .epi

```
const n:int = 10
const tab : int ^ n
```

Contenu

- Comment construit-on une spécification fonctionnelle ?
  - Cas d'étude GNC (avionique aérospatiale)
  - Programmation de GNC
- Programmation Heptagon
  - Types de données structurés
  - Paramètres statiques
- Préparation du TP
  - Programmation de l’exemple GNC
  - Nombres complexes
  - Itérateurs
  - La FFT (Fast Fourier Transform)

Objectif 1 – programmation de GNC

- Programmer GNC (transparent 19)
- Les tâches Fast, GNC, Thermal
  - Chaque tâche est un noeud programmé en Heptagon dont l'état est formé d'un seul compteur d'instances idx, incrémenté (en Heptagon) à chaque exécution
  - La fonction calculée par GNC est x = y-idx, la fonction calculée par Fast est y = 2*x + idx
  - Chaque instance d'un noeud doit imprimer : le nom du noeud, la valeur d'idx, et (pour GNC et Fast) les valeurs d'entrée et de sortie
    - Fonctions externes d' impression à écrire en C et interfacer .epi
- Utiliser usleep pour assurer que chaque cycle d'exécution a une durée d'au moins 1 seconde (au lieu de 100ms)
  - Comme pour l'exemple "counter" du TP1

Objectif 1 – programmation de GNC

- Attention – la fonction thermal n'a pas d'entrées, ni de sorties, donc la programmation de « condact_thermal » ne suit pas le schéma normal
  - Un nœud supplémentaire avec entrée inutile est nécessaire

Objectif 2 : nombres complexes

```
type complex =
  {
    re : float ;
    im : float
  }
```

- complex.ept
  - Définition du type complexe
  - Programmation en Heptagon des fonctions mathématiques de base

```
fun complex_add(i1:complex;i2:complex) returns (o:complex)
fun complex_sub(i1:complex;i2:complex) returns (o:complex)
fun complex_mul(i1:complex;i2:complex) returns (o:complex)
```

- complex_io.epi, complex_io.[ch], complex_io_types.h
  - Programmation en C et interfaçage Heptagon des fonctions externes de lecture (depuis le clavier) et d'affichage de complexes

```
fun read_complex() returns (o:complex)
fun print_complex(i:complex) returns ()
```

  - Un fichier .epi peut faire "open" d'un fichier .ept qui définit des types nécessaires

Objectif 2 : nombres complexes

- complexes.ept
  - Écriture d'une fonction Heptagon sans entrées et sans sorties nommée « complexes » qui lit deux complexes depuis le clavier (à l'aide de "read_complex") et affiche leur somme, différence, produit (avec "print_complex")
- main.c
  - Écriture de la fonction C « main » qui exécute cycliquement la fonction Heptagon « complexes »
- Compilation et exécution

Objectif 2 : nombres complexes

- Opérations en nombres complexes – rappel
  - x = x.re + i*x.im, où i*i = -1
  - Opérations élémentaires
    - Produit : (a+i*b)*(c+i*d) = (a*c-b*d)+i*(a*d+b*c)
    - Somme : (a+i*b)+(c+i*d) = (a+c)+i*(b+d)
    - Soustraction : (a+i*b)-(c+i*d) = (a-c)+i*(b-d)

Objectif 3 : itérateurs

- complex_vec_io.epi, complex_vec_io.[ch], complex_vec_io_types.h
  - Définir des fonctions externes pour la lecture et l'impression de vecteurs de complexes de taille 3

```
fun read_complex_vector() returns (o:complex^3)
fun print_complex_vector(i:complex^3) returns ()
```

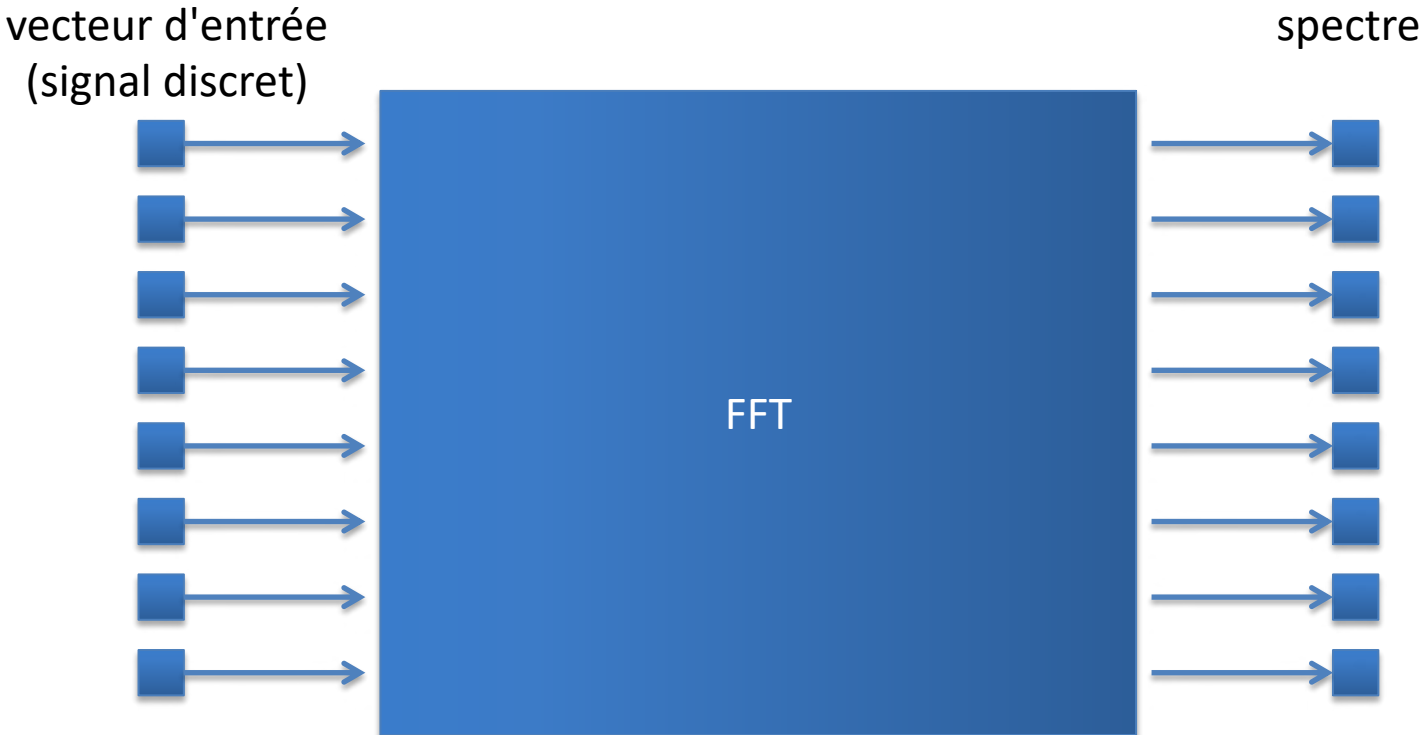
- complex_vec.ept
  - Programmer en heptagon des fonctions à paramètres statiques permettant de :
    - Additionner deux vecteurs complexes de taille N
      - Résultat = un vecteur complexe de taille N
    - Calculer la somme des élements d'un vecteur de complexes
      - Résultat = un seul complexe
  - Utilisation obligatoire d'itérateurs pour les calculs
- complex_vectors.ept
  - Écriture d'une fonction Heptagon « vectors » sans entrées et sans sorties qui lit deux vecteurs complexes depuis le clavier et affiche leur somme (un vecteur) et la somme des éléments du vecteur-somme (un complexe)
- main.c
- Compilation et exécution

La FFT en Heptagon

- FFT = Fast Fourier Transform
  - Implémentation efficace de la Discrete Fourier Transform
    - Appliquée à un vecteur de données, la DFT/FFT calcule son spectre discret
    - Au lieu de voir un signal comme une suite d'échantillons, le voir comme une somme pondérée de fréquences de base (fonctions périodiques).
  - Algorithme fondamental en traitement de signal
    - Compression, en gardant les composantes principales
    - Identification d'erreurs (analyse vibratoire)
    - Radar et échographie (identification d'effet Doppler)
    - Filtrage (en gardant les fréquences d'intérêt)
      - Traitement de la parole en Machine Learning
    - Musique : **Changement de timbre**, etc.
      - Temps réel !
  - Exemple de base dans toute évaluation de performances pour architectures/compilateurs/etc.

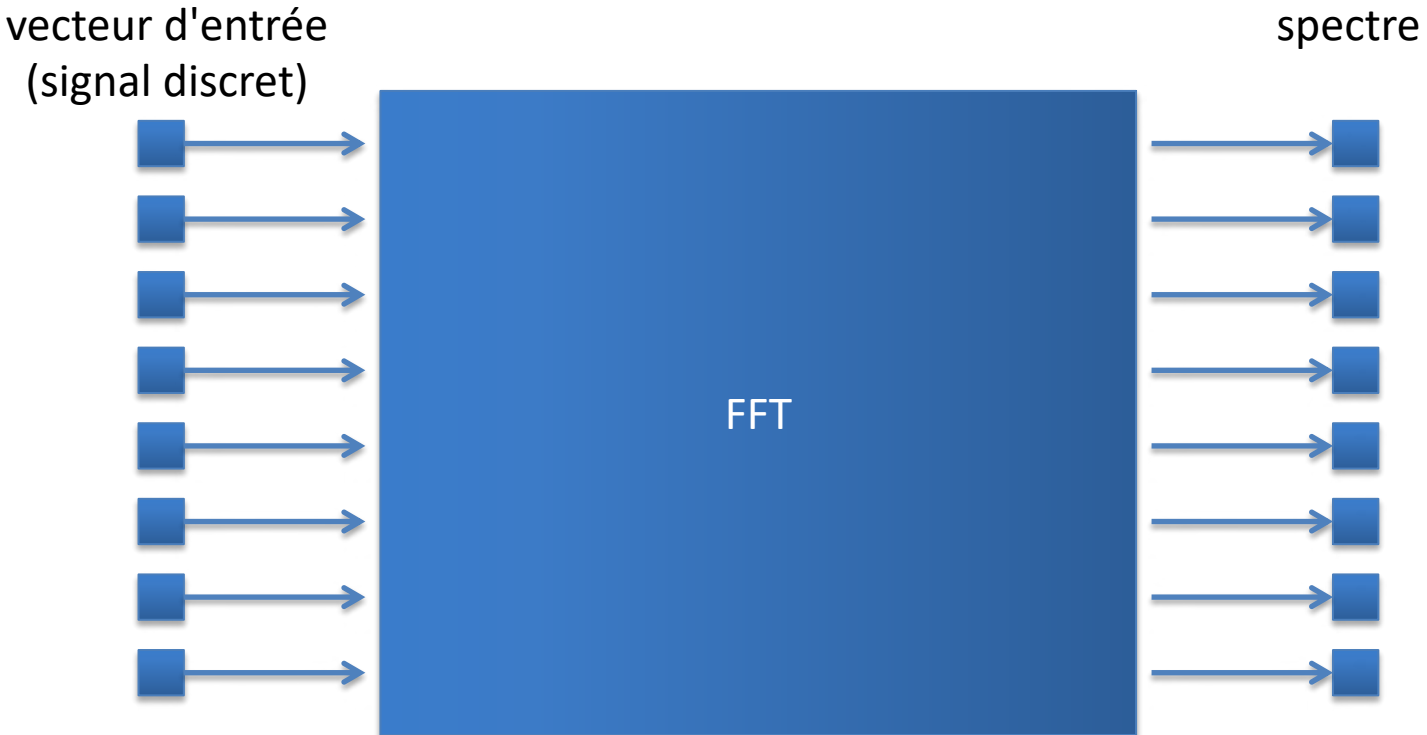
La FFT en Heptagon

- FFT = Fast Fourier Transform



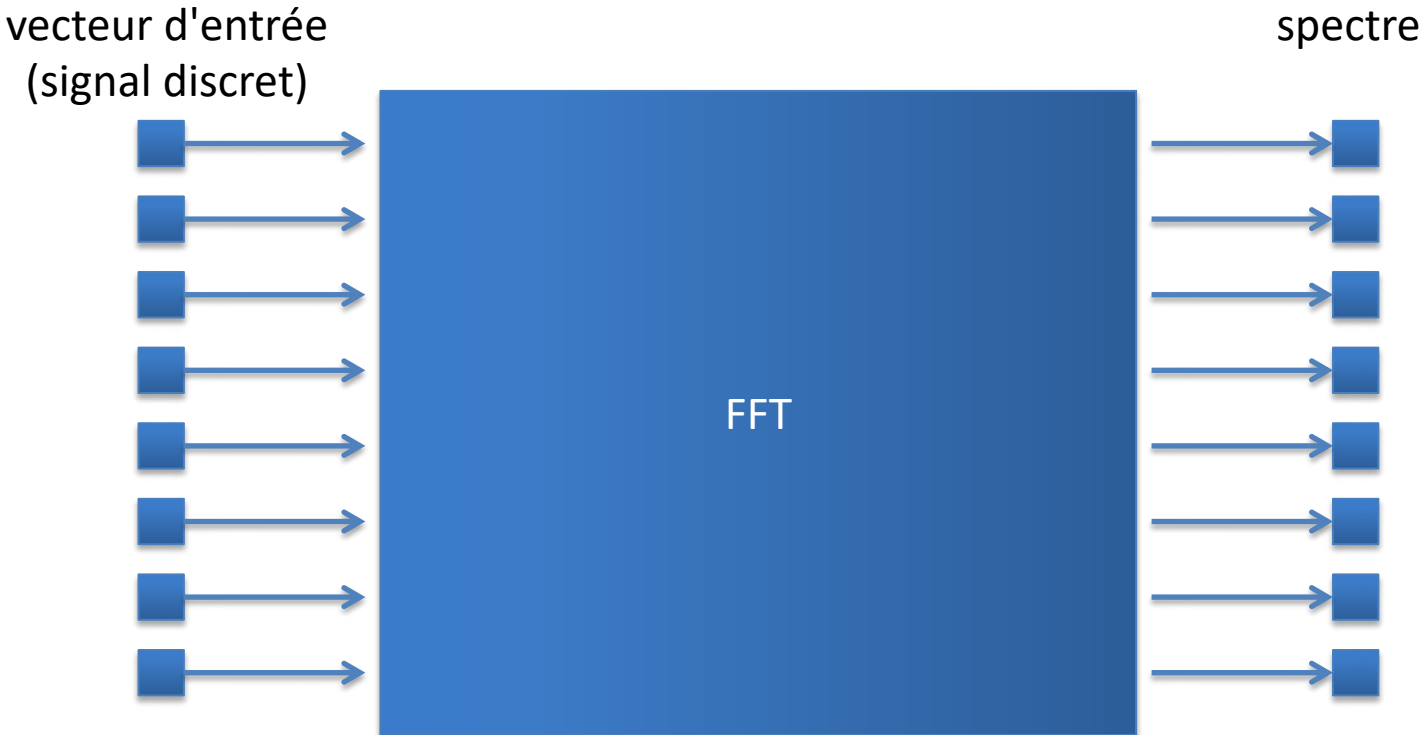
La FFT en Heptagon

- FFT = Fast Fourier Transform
  - Notre choix d'algorithme : radix 2 Cooley-Tukey
  - Taille du vecteur N=2^logN



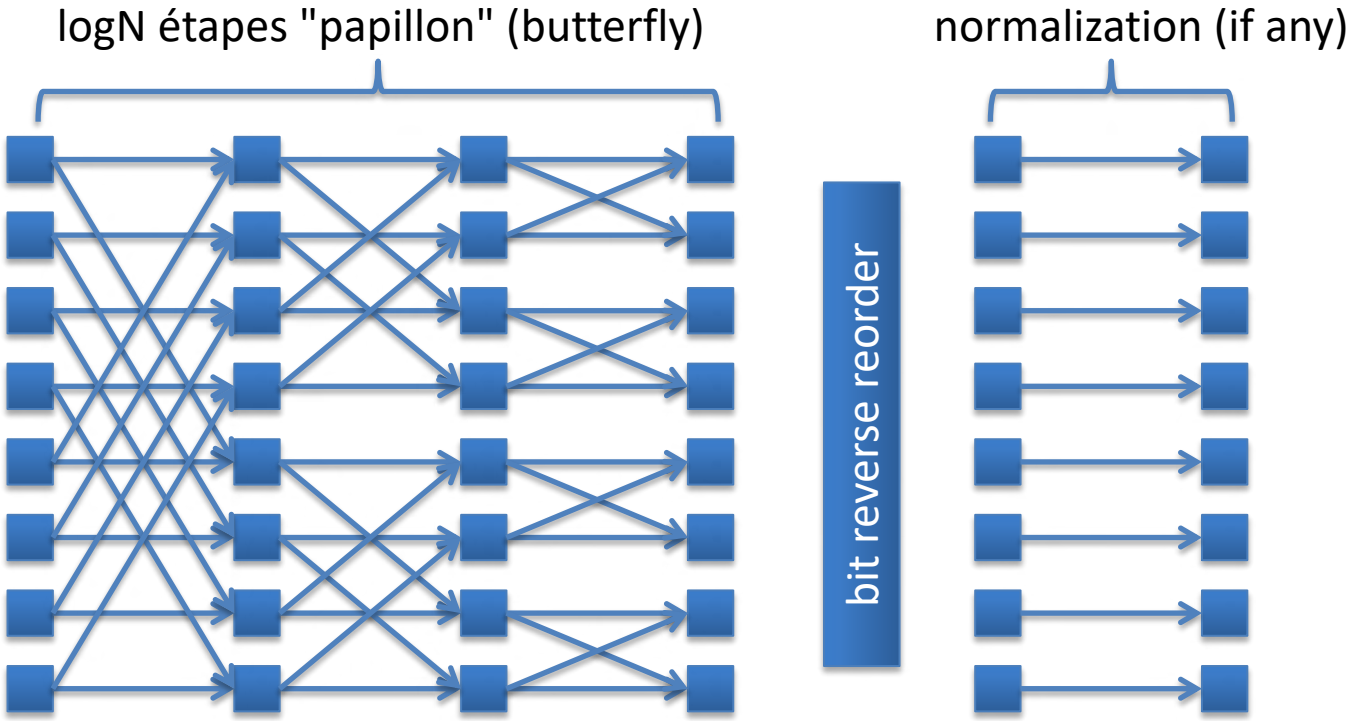
La FFT en Heptagon

- FFT = Fast Fourier Transform
  - Notre choix d'algorithme : radix 2 Cooley-Tukey
  - Taille du vecteur N=2^logN => algo. très régulier



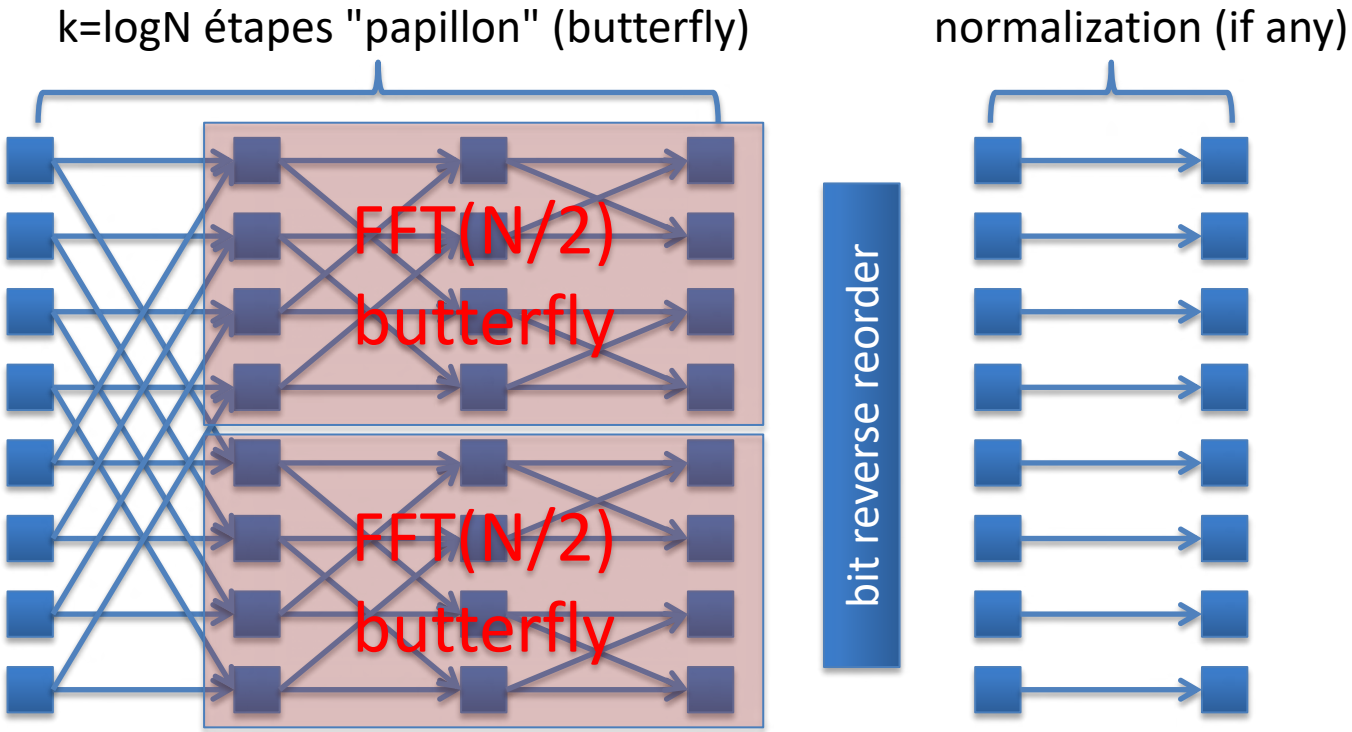
La FFT en Heptagon

- FFT = Fast Fourier Transform
  - Notre choix d'algorithme : radix 2 Cooley-Tukey
  - Taille du vecteur N=2^logN => algo. très régulier



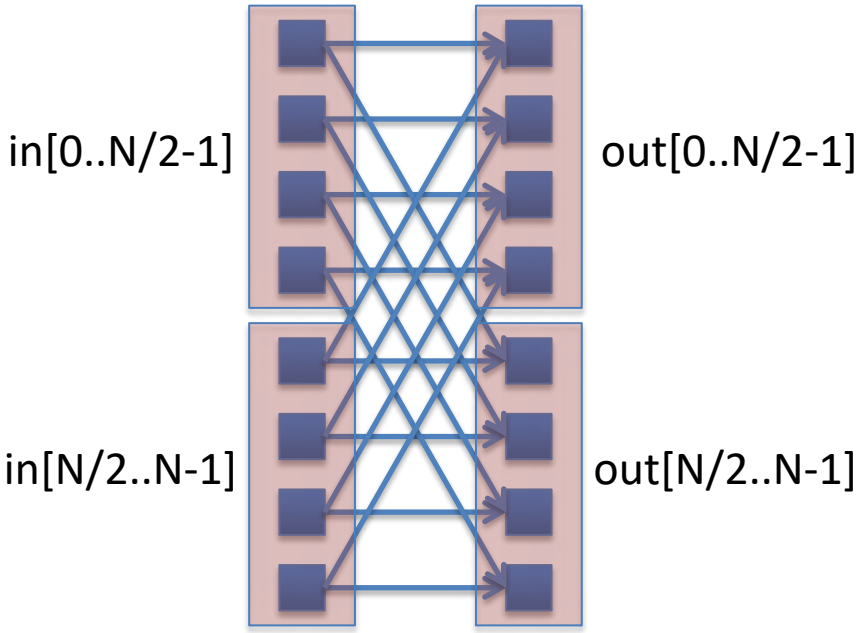
La FFT en Heptagon

- FFT = Fast Fourier Transform
  - Récursion statique, facile à paralléliser



La FFT en Heptagon

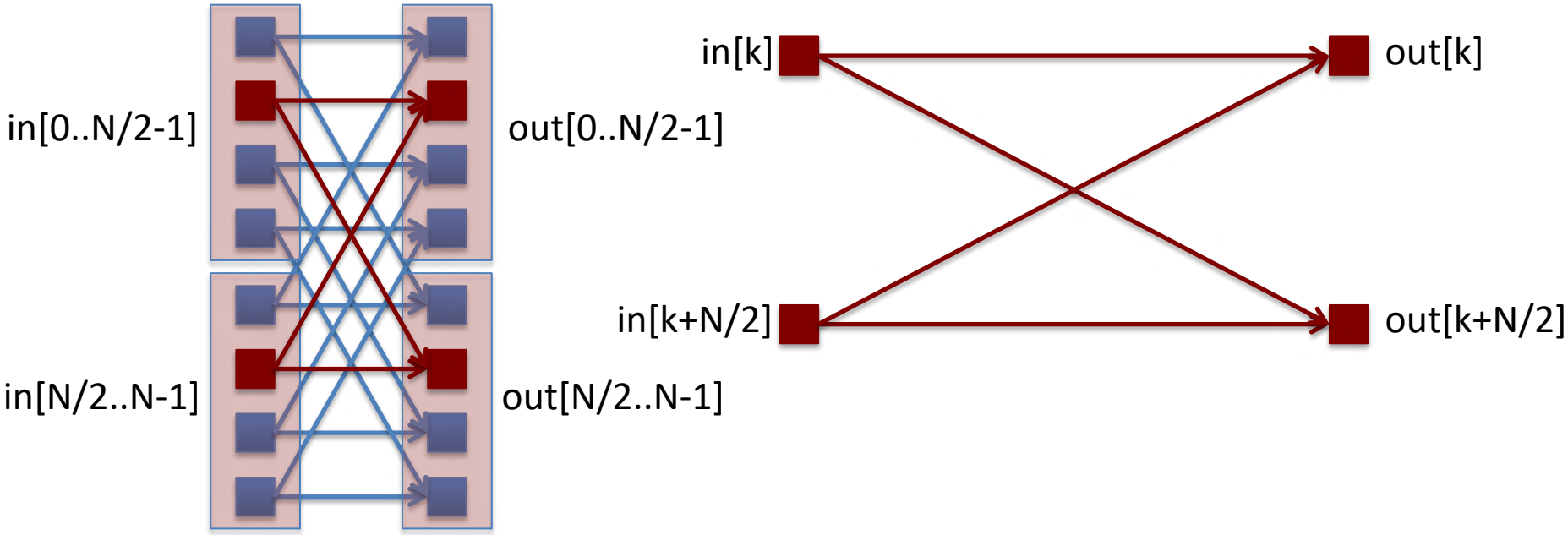
- FFT = Fast Fourier Transform
  - Focus sur une étape "papillon" générique de taille N=2^logN




```
fun bf<<n:int>>(i1:complex^n;i2:complex^n)
        returns (o1:complex^n;o2:complex^n), n=N/2
```

La FFT en Heptagon

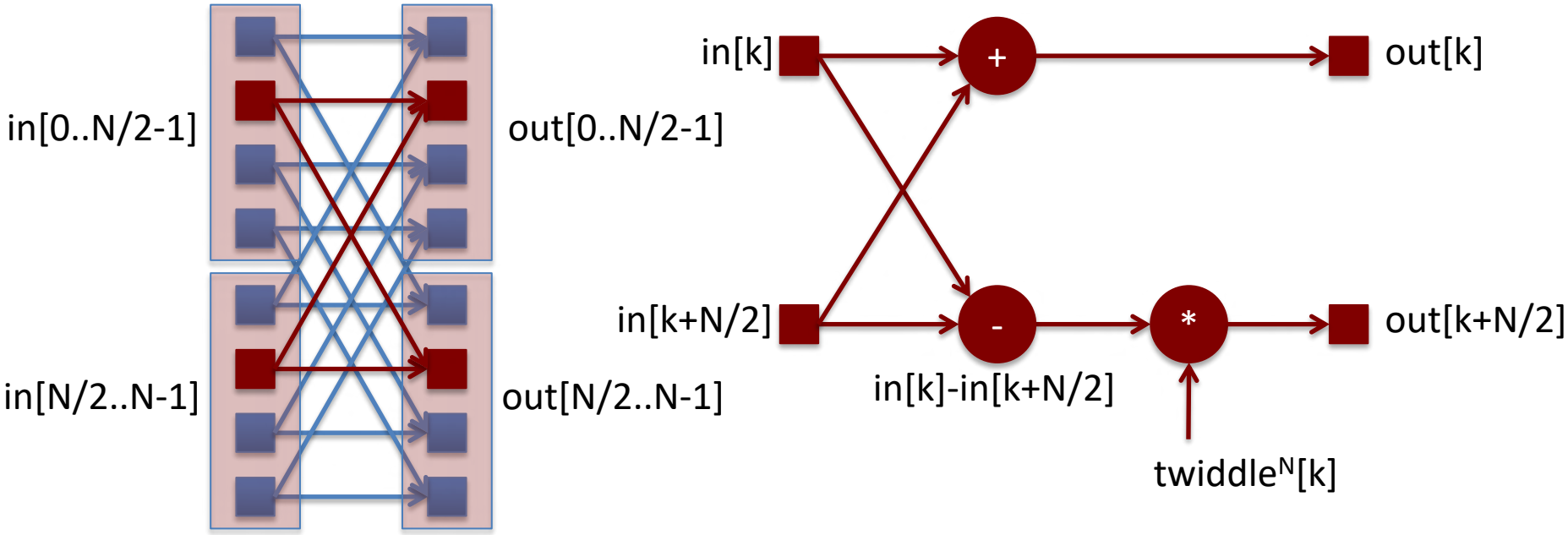
- FFT = Fast Fourier Transform
  - Echantillons traités par paire (k=0..N/2-1) :



  - Opération de type **map**

La FFT en Heptagon

- FFT = Fast Fourier Transform
  - Echantillons traités par paire (k=0..N/2-1) :

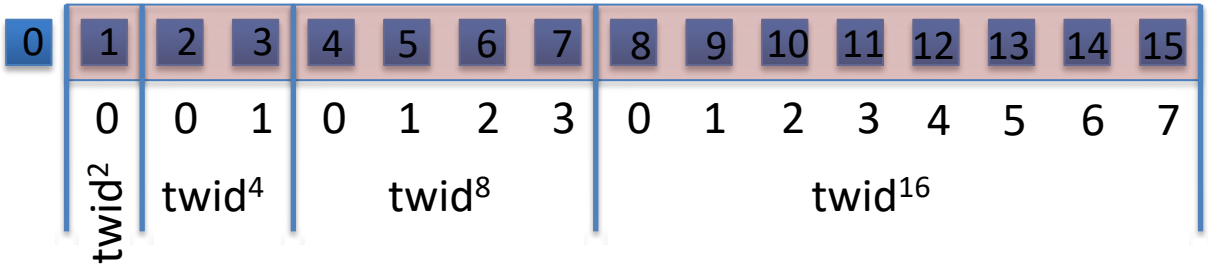


La FFT en Heptagon

- Facteurs "twiddle" = racines de l'unité
  - $\text{twiddle}^N[k] = \exp(i*r_k) = \cos(r_k)+i*\sin(r_k)$
    - where $r_k = -2*pi*k/N$
    - where pi = 3.14159265358979

Préparation du TP – objectif 4 - FFT

- twiddle.[ch]
  - Définition (en C) d'une constante tableau "twiddle"
    - stocker les constantes pour N<=1024
    - Type complexe !
  - La fonction permettant de remplir ce tableau est fournie
    - twiddle_init.[ch]
  - Structure du tableau pour N<=16 :



Préparation du TP – objectif 4 - FFT

- twiddle.epi, twiddle.[ch]
  - Définition d'une fonction externe qui accède aux éléments du tableau

    ```
    fun twiddle(k:int;n:int) returns (o:complex)
    ```

  - Attention:
    - La fonction "twiddle" pourra être appelée depuis le code Heptagon
    - La fonction "init_twiddle1024" devra être appelée depuis main.c une seule fois, avant les autres initialisations
    - twiddle_init.[ch] utilisent le type Complex__complex défini dans le code généré par compilation de complex.ept

Préparation du TP – objectif 4 - FFT

- fft.ept
  - Étape 1: écrire la fonction réalisant un étage d'opération "butterfly", ayant le prototype :

```
fun bf<<n:int>>(i:complex^n) returns (o:complex^n)
```

    - Attention:
      - On peut utiliser des expressions comme paramètres statiques

```
o = somefun<< (n/2) >>(i,j)
```

        » Utiliser des parenthèses et laisser des espaces (parseur fragile)

Préparation du TP – objectif 4 - FFT

- fft.ept
  - Étape 2: Écrire les fonctions fft_aux2, fft_aux4, fft_aux8, fft_aux16 réalisant l'ensemble des opérations "papillon" pour N=2,4,8,16

```
fun fft_aux2 (in:complex^2 ) returns (out:complex^2 )
...
fun fft_aux16(in:complex^16) returns (out:complex^16)
```

    - Attention : le langage ne permet pas la récursion. Alors, il est impossible de paramétrer comme pour bf
      - Il faut créer une fonction pour chaque puissance de 2
      - `fft_aux16` instancie deux fois `fft_aux8`…

Préparation du TP – objectif 4 - FFT

- bitrev.epi, bitrev.[ch], bitrev_types.h
  - La fonction bitreverse est fournie en C dans le fichier bitrev.c. Il faut lui créer des interfaces pour permettre l'appel depuis heptagon pour N=8,16

```
fun bitrev8 (in:complex^8 ) returns (out:complex^8 )
fun bitrev16(in:complex^16) returns (out:complex^16)
```

    - Fonctions à écrire en C, à interfacer en Heptagon
    - Attention : la fonction C fournie travaille en place
      - Inacceptable en dataflow, il faut l'encapsuler
      - Réaliser une copie de la valeur d'entrée avant d'appeler bitrev

Préparation du TP – objectif 4 - FFT

- fft.ept
  - Étape 3: La FFT pour tailles de vecteurs 8 et 16

```
fun fft8 (in:complex^8 ) returns (out:complex^8 )
fun fft16(in:complex^16) returns (out:complex^16)
```

  - Normalisation = diviser chaque valeur par N (8 ou 16)
- complex_vec_io.epi, complex_vec_io.[ch]
  - Fonctions d'impression de vecteurs complexes de taille 8

Préparation du TP – objectif 4 - FFT

- fft_test.ept
  - Définition d'une fonction sans entrées et sans sorties qui appelle fft8 sur les vecteurs de test suivants et imprime le résultat

```
fft8([1,0,0,0,0,0,0,0]) = [1/8,1/8,1/8,1/8,1/8,1/8,1/8,1/8]
fft8([0,1,0,0,0,0,0,0]) =
  [ 0.125, 0.088-i*0.088,-i*0.125,-0.088-i*0.088,
   -0.125,-0.088+i*0.088, i*0.125, 0.088+i*0.088]
```

- main.c
  - Appel de fft_test (une seule fois suffit)
  - Attention – il faut initialiser les twiddles.

Préparation du TP – objectif 5 - IFFT

- La FFT(N) inverse (IFFT(N)) se calcule de la manière suivante :
  - Inverser partie réelle et imaginaire dans chaque élément du vecteur d'entrée
    - fonction à définir, et application avec **map**
  - Calculer la FFT(N) sur le nouveau vecteur
  - Ré-inverser parties réelle et imaginaire
  - Normaliser en multipliant chaque élément par N
- Programmer la IFFT(8) et vérifier que IFFT(FFT(v)) = v pour les deux vecteurs de test