Question 1 (5.5 points) Architecture RISC scalaire

Soit le programme DLX suivant pour un filtre en point flottant 1D FIR TAP où n = 3. En langage C on aura:

```
for (i = 100 ; i ≥ 2; i--)
   y[i] = a0 * x[i-2] + a1 * x[i-1] + a2 * x[i]
```

De plus, on a:

- 3 registres F0, F10 et F20 pour les constantes a2, a1 et a0 respectivement;
- 3 registres F30, F40 et F50 pour chaque échantillon représentant x[i], x[i1-1] et x[i-2] dans un vecteur de dimension 100,
- L'adressage varie de l'adresse 800 à 16 inclusivement via R4 (i.e., de 100 à 2 en considérant des mots de 8 octets), et
- 1 registre résultant F60 (initialement à 0) pour réaliser un multiplicateur/accumulateur (MAC).

En assembleur DLX cela correspond au programme suivant:

```
L:    LD      F60, #0
      LD      F30, 0(R4)
      LD      F40, -8(R4)
      LD      F50, -16(R4)
      MULTD   F30, F30, F20
      MULTD   F40, F40, F10
      MULTD   F50, F50, F0
      ADDD    F60, F60, F30
      ADDD    F60, F60, F40
      ADDD    F60, F60, F50
      SD      1800(R4), F60
      SUBI    R4, R4, #8
      SEQI    R5, R4, #16
      BEQZ    R5, L
```

avec au départ R4=800.

Le détail des différents types d'instructions assembleurs est donné en Annexe.

a) (1.5 pts) Donnez la trace du programme **en complétant le Tableau 1.1 de la page suivante** pour l'exécution d'une seule boucle, en tenant compte des cycles de suspension ou d'attente. Considérez un modèle de pipeline de type M4 (voir Annexe). Donnez le temps d'exécution pour une itération de boucle.

b) (2 pts) À partir de la trace obtenue en a), proposez une réorganisation (réordonnancement) du programme afin d'accélérer au maximum l'exécution du filtre FIR en déroulant la boucle 3 fois (i.e., 3 itérations à la fois). Vous devez préserver la fonctionnalité tout en minimisant les suspensions et le nombre de load. Remarque : le choix des numéros de registre (p.e. F10) a volontairement été espacé pour que vous puissiez, au besoin, en introduire de nouveaux (p.e. F11, F12, etc.).

b.1) **Complétez le Tableau 1.2 de la page 3**
b.2) Montrez le gain de l'ordonnancement résultant et comparez le gain par rapport à a).

c) (2 pt) Soit une architecture VLIW (Very Long Instruction World) avec des mots de longueur 160 bits. Considérez 2 unités de transfert (accès mémoires), 2 unités de calcul flottant et 1 unité de calcul entière (incluant les instructions de branchement). Proposez une façon d'améliorer l'exécution de ce programme sur cette architecture en ordonnançant le code obtenu en b). Vous devez donc :

c.1) **Complétez le Tableau 1.3 de la page 4** (au besoin utilisez des flèches pour illustrer le respect des dépendances inter unités).
c.2) Comparez le gain par rapport à a).

##		Instructions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	L:	LD F60, #0	*LI*	*DI*	*EX*	*ME*	*ER*																								
2		LD F30, 0(R4)		*LI*	*DI*	*EX*	*ME*	*ER*																								
3		LD F40, -8(R4)			*LI*	*DI*	*EX*	*ME*	*ER*																						
4		LD F50, -16(R4)				*LI*	*DI*	*EX*	*ME*	*ER*																					
5		MULTD F30, F30, F20					*LI*	*DI*	*E1*	*E2*	*E3*	*E4*	*E5*	*E6*	*ME*	*ER*															
6		MULTD F40, F40, F10						*LI*	*DI*	*E1*	*E2*	*E3*	*E4*	*E5*	*E6*	*ME*	*ER*														
7		MULTD F50, F50, F0							*LI*	*DI*	*E1*	*E2*	*E3*	*E4*	*E5*	*E6*	*ME*	*ER*													
8		ADDD F60, F60, F30								*LI*	*DI*	*SU*	*SU*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*												
9		ADDD F60, F60, F40									*LI*	*SU*	*SU*	*SU*	*DI*	*SU*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*									
10		ADDD F60, F60, F50													*LI*	*SU*	*SU*	*DI*	*SU*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*						
11		SD 1800(R4), F60																*LI*	*SU*	*SU*	*DI*	*SU*	*SU*	*EX*	*ME*	*ER*					
12		SUBI R4, R4, #8																			*LI*	*SU*	*SU*	*DI*	*SU*	*EX*	*ME*	*ER*		
13		SEQI R5, R4, #16																						*LI*	*SU*	*DI*	*EX*	*ME*	*ER*		
14		BEQZ R5, L																								*LI*	*DI*	*EX*			
15																											*SU*	*SU*	*LI*		

Tableau 1.1 pour question 1 a)
À compléter et remettre avec le cahier de réponse.

*25 cycles sont requis donc 26*99 = 2574*

N.B. On a mis SU au cycle 23 pour éviter que R4 change pendant la mémorisation SD mais c'est discutable…

Cycle	Pipeline 1
1 L :	*LD F60, #0*
2	*LD F61, #0*
3	*LD F62, #0*
4	*LD F30, 0(R4)*
5	*LD F40, -8(R4)*
6	*LD F50, -16(R4)*
7	*LD F51, -24(R4)*
8	*LD F52, -32(R4)*
9	*MULT F30, F30, F0*
10	*MULT F31, **F40**, F0*
11	*MULT F32, **F50**, F0*
12	*MULT F40, **F40**, F10*
13	*MULT F41, **F50**, F10*
14	*MULT F42, F51, F10*
15	*MULT F50, **F50**, F20*
16	*MULT F51, **F51**, F20*
17	*MULT F52, F52, F20*
18	*ADDF F60, F60, F30*
19	*ADDF F61, F61, F31*
20	*ADDF F62, F62, F32*
21	*ADDF F60, F60, F40*
22	*ADDF F61, F61, F41*
23	*ADDF F62, F62, F42*
24	*ADDF F60, F60, F50*
25	*ADDF F61, F61, F51*
26	*ADDF F62, F62, F52*
27	*SD F60, 1800(R4)*
28	*SUBI R4, R4, #24*
29	*SEQI R5, R4, #16*
30	*BEQZ R5, L*
31	*SD F61, 1816(R4)*
32	*SD F62, 1808(R4)*
33	

Tableau 1.2 pour question 1 b.1)

À compléter et remettre avec le cahier de réponse (au besoin, poursuivre dans le cahier de réponses).

Un déroulement de 3 suffit pour enlever toutes les suspensions.

*Donc 99/3 * 32 = 1056 donc 2574/1056 = 2,43 fois plus vite.*

	UNITÉ TRANSFERT 1	UNITÉ TRANSFERT 2	UNITÉ FLOTTANTE 1	UNITÉ FLOTTANTE 2	UNITÉ ENTIÈRE
1 L:	*LD F30, 0(R4)*	*LD F40, -8(R4)*			
2	*LD F50, -16(R4)*	*LD F51, -24(R4)*			
3	*LD F52, -32(R4)*		*MULT F30, F30, F0*	*MULT F31, **F40**, F0*	
4	*LD F60, #0*		*MULT F32, **F50**, F0*	*MULT F40, **F40**, F10*	
5	*LD F61, #0*		*MULT F41, **F50**, F10*	*MULT F42, F51, F10*	
6	*LD F62, #0*		*MULT F50, **F50**, F20*	*MULT F51, **F51**, F20*	
7			*MULT F52, F52, F20*		
8					
9			*ADDF F60, F60, F30*	*ADDF F61, F61, F31*	
10			*ADDF F62, F62, F32*		
11					
12			*ADDF F60, F60, F40*	*ADDF F61, F61, F41*	
13			*ADDF F62, F62, F42*		
14					
15			*ADDF F60, F60, F50*	*ADDF F61, F61, F51*	*SUBI R4, R4, #24*
16			*ADDF F62, F62, F52*		*SEQI R5, R4, #3*
17					*BEQZ R5, L*
18	*SD F60, 1824(R4)*	*SD F61, 1816(R4)*			
19	*SD F62, 1808(R4)*				
20					
21					

Tableau 1.3 pour question 1 c.1) À compléter et remettre avec le cahier de réponse.

On observe des suspensions du à l dépendance de l'accumulation (faire un arbre de sommation avec n=3 n'en vaut pas la peine)

*Donc 99/3 * 19 = 627 donc 2574/627 = 4.10 fois plus vite.*

Question 2 (2 points) Architectures – aléas de contrôle

À l'aide de la boucle B et de la machine à 4 états à compléter de la figure 2.1, expliquez la technique de prédiction dynamique que l'on retrouve dans les architectures contemporaines tel que le Cortex A9. Considérez que R1 contient au départ la valeur 100,000 et que B fait partie d'une fonction qui est appelée 1,000 fois.

```
B:  LD      F0, 0(R1)
    ADDD    F4, F0, F2
    SD      0(R1), F4
    SUBI    R1, R1, #8
    BNEZ    R1, B
```

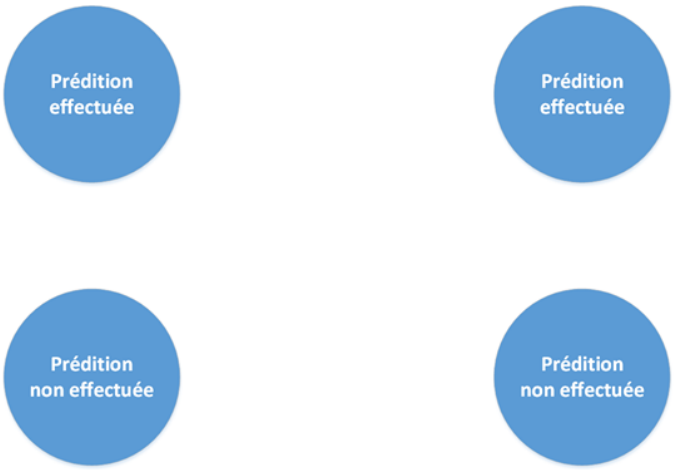
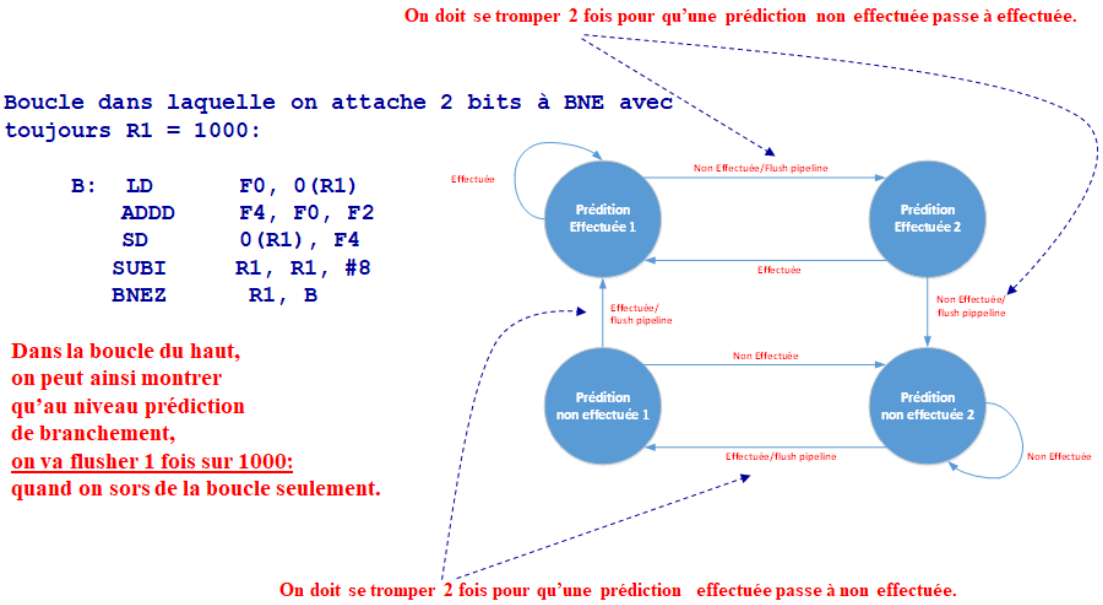


Figure 2.1 À compléter et expliquer.



Question 3 (3.5 points) Synthèse de haut niveau

Soit la boucle suivante :

int idx = 0;

#pragma HLS pipeline

for (int i=0; i<98 ; i+=3)
y[idx++] = x[i]*z[i] + x[i-1] + x[i-2]*z[i-1];

qu'on désire matérialiser. La figure 3.1 montre l'ordonnancement d'une itération de la boucle:

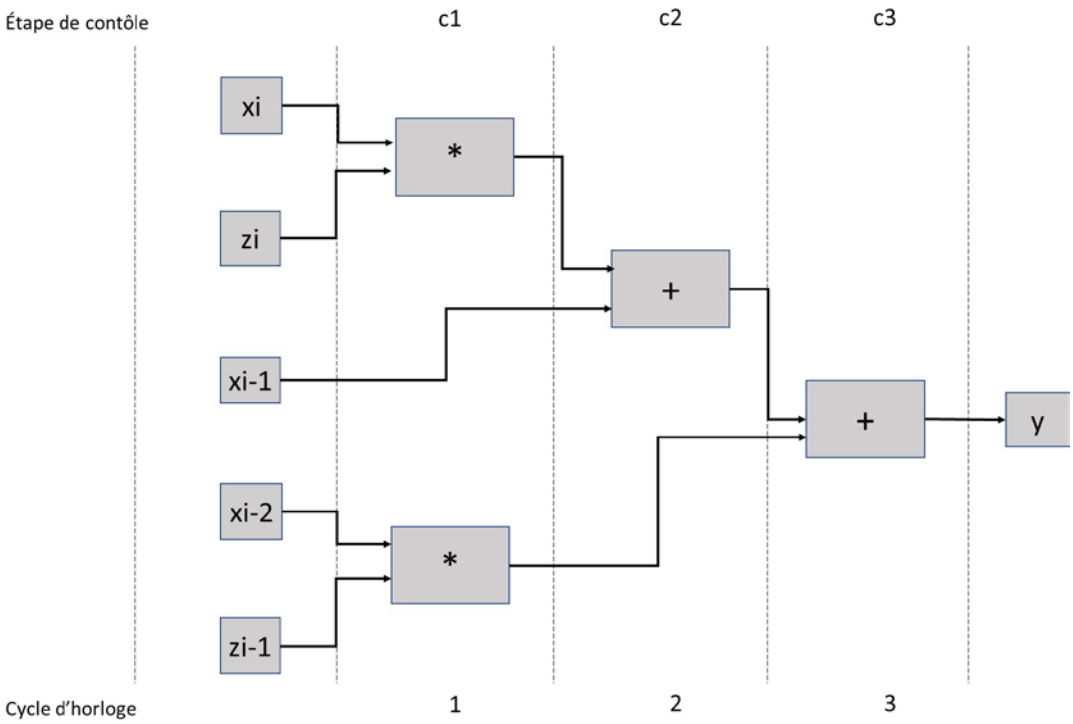


Figure 3.1 Les petites boites sont de la BRAM et les grosses boites des opérations (additionneur et multiplication)

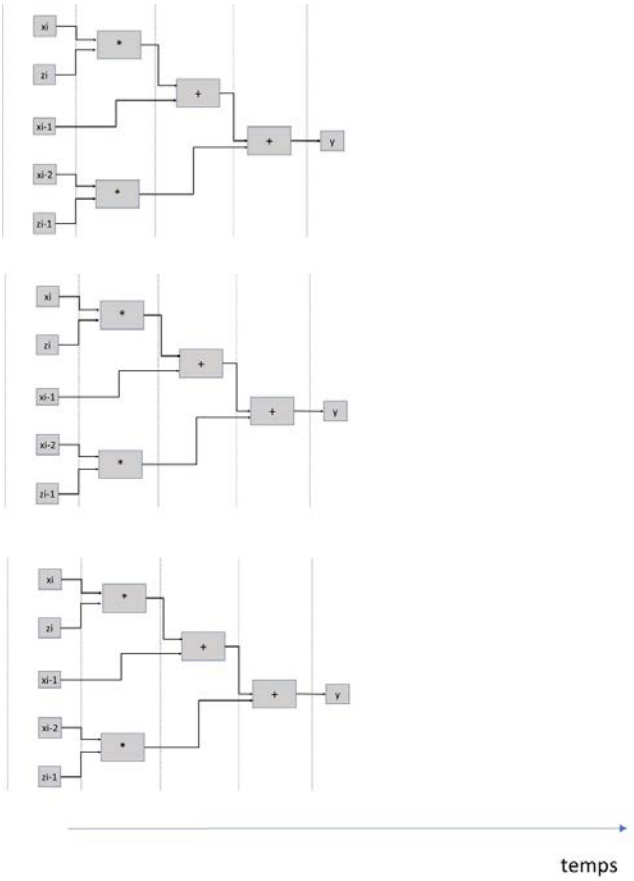
a) (1.5 pts) Pour chacune des illustrations des pages 9 à 10 (Figure 3.2a à c) représentant les 3 premières itérations de la boucle *for* avec un certain pipelinage, donnez : 1) le nombre d'additionneurs et de multiplieurs requis, 2) la valeur de II et 3) la latence requise.

Figure 3.2a) : *2 add 2 mult II = 1 et L =5*
Figure 3.2b) : *1 add 2 mult II = 2 et L =5*
Figure 3.2c) : *1 add 2 mult II = 3 et L =5*

b) (.5) Donnez le détail du *pragma array_partition* sur les variables d'entrées x et z et de sortie y de la figure 3.2a. Assumez des BRAM à 2 ports.

Pragma HLS array_partition variable = x block factor = 2
Pragma HLS array_partition variable = z block factor = 1
Pragma HLS array_partition variable = y block factor = 1

c) (1.5 pt) Dessinez dans votre cahier de réponse, ce que serait les 3 premières itérations de cette boucle si on enlevait le *#pragma HLS* pipeline en haut de la boucle et on insérait dans cette même boucle un *#pragma HLS unroll*. Indiquez également, le nombre d'additionneurs et de multiplieurs requis ainsi que le détail du *pragma array_partition* sur les variables d'entrées x et z et de sortie y.



On suppose qu'on va tenter d'exécuter 33 itérations en parallèle :
*Ici on aura besoin de 5 loads * 33 + 33 * 2 mult + 33 * 1 add + 33 * 1 add*

Est-ce que mon FPGA a 66 mult, est-ce mon FPGA a 66 add ???

Au niveau des load/store par itération de boucle:
2 load de x + 1 load de z + 1 store de y = 4 load/store au total
N.B. *Ce qui est important de vérifier ici c'est pour x : 2 load * 33 < 140 (limite du Xilinx SoC 7020) donc OK*

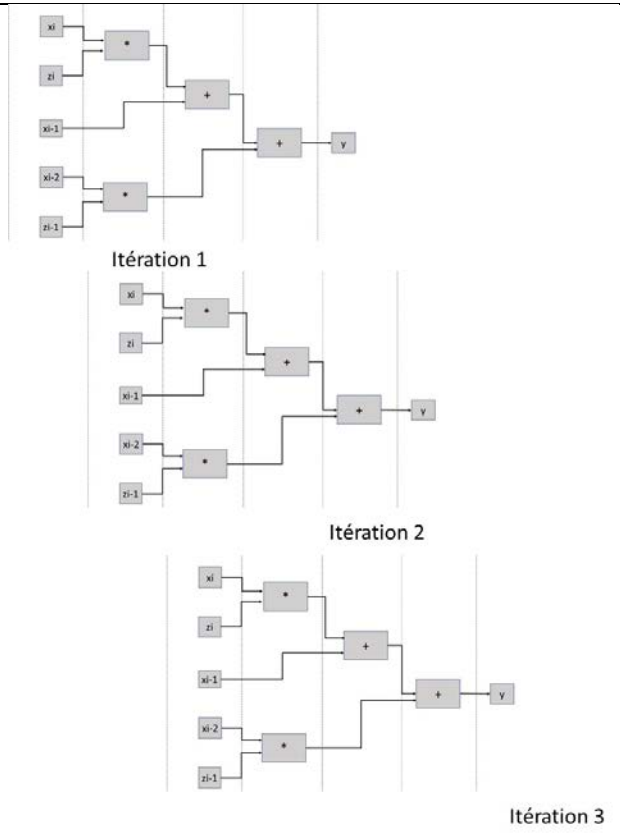


Figure 3.2a)

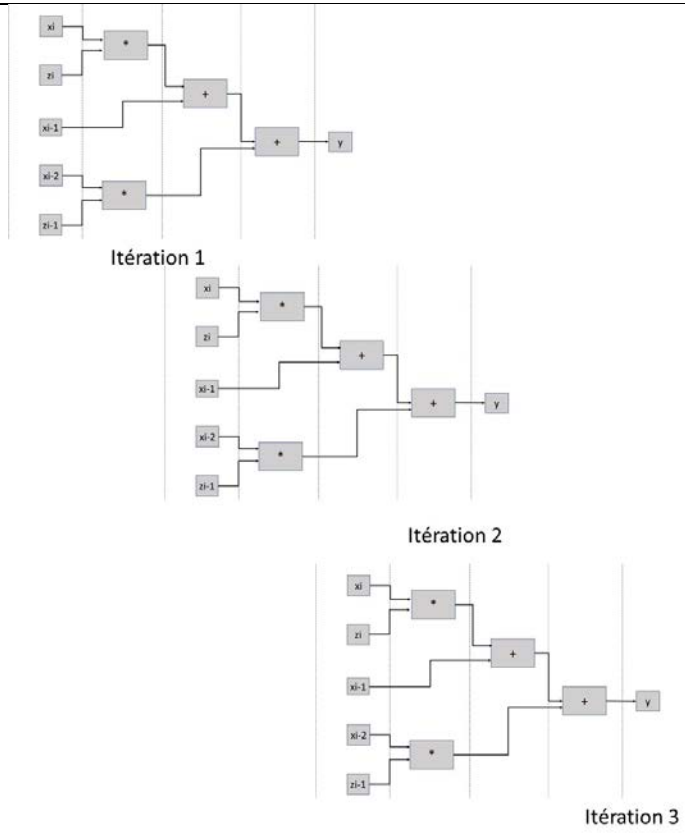


Figure 3.2b)

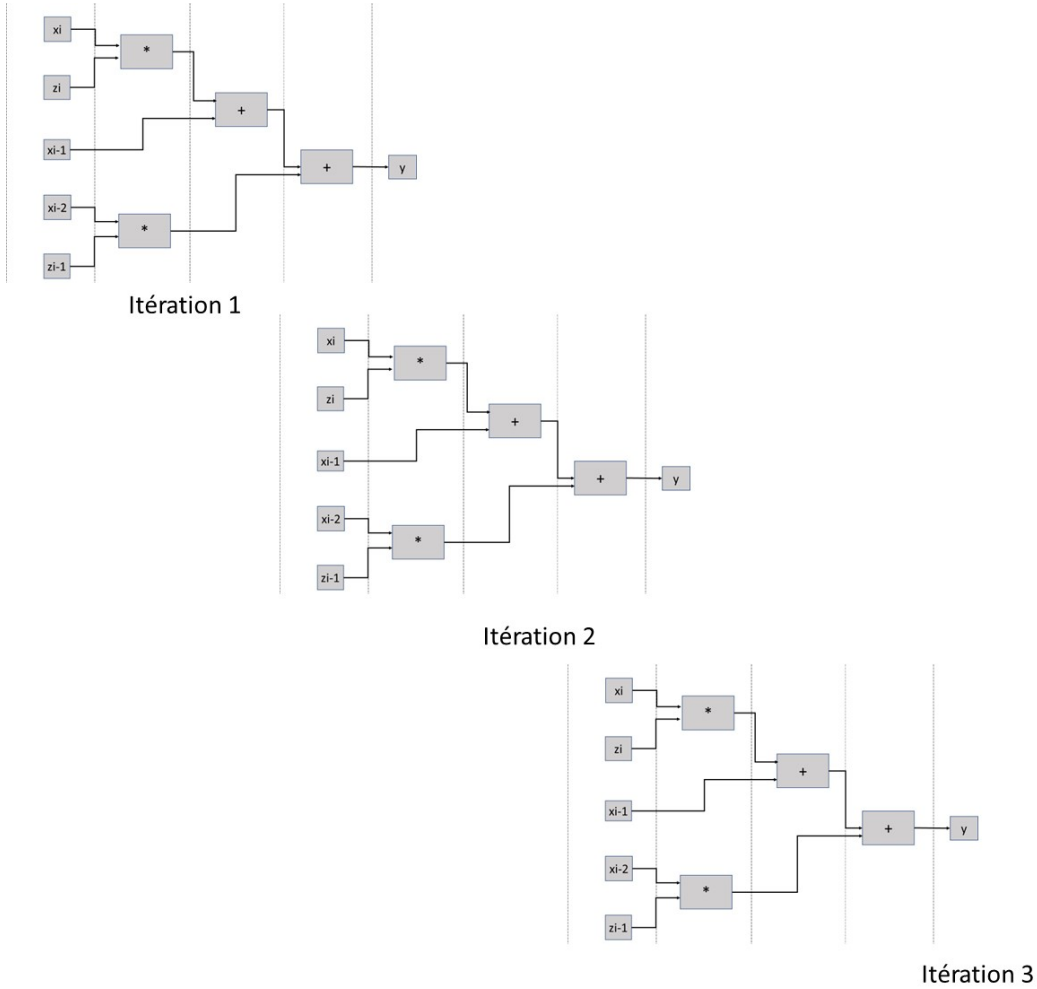


Figure 3.2c

Question 4 (4.5 points) Interconnexions AXI

Le no 4 a été vu en détail dans le power point du bloc 3 du chapitre 3 (tous les réponses s'y trouvent).

a) (1.5 pt) On a vu en classe qu'une transaction sur un bus se divise généralement en 4 étapes importantes. Soit la figure 4.1 qui représente les signaux d'entrée et de sortie de 3 acteurs de AHB.

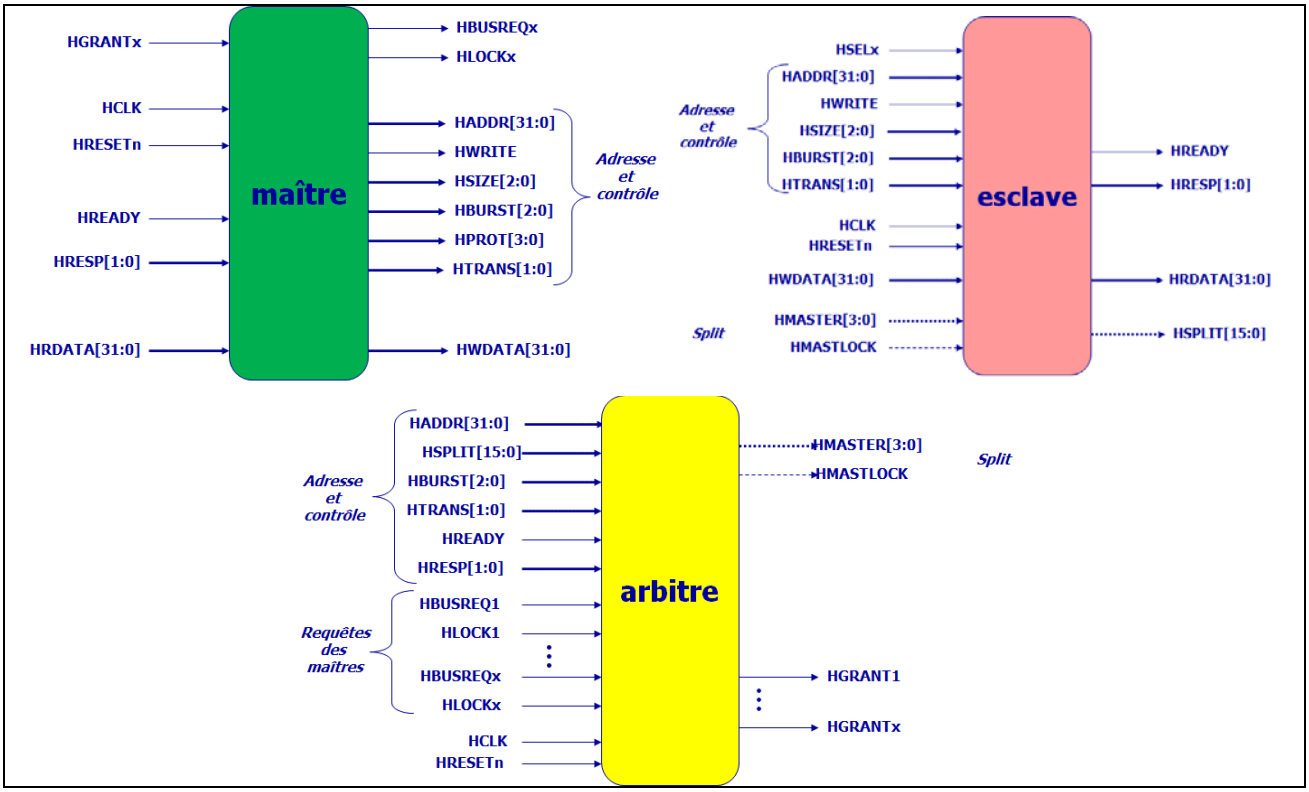


Figure 4.1

Décrivez d'abord et expliquez le rôle de chaque étape, puis pour chacune de ces étapes indiquez les signaux de la figure 4.1 qui sont impliqués. Au besoin regrouper les signaux.

b) (.5 pt) Il manque au moins 2 acteurs aux 3 acteurs du haut. Qui sont-ils et quels rôles jouent-ils brièvement?

c) (1.5 pt) Que représente le diagramme de la Figure 4.2 (page suivante). Décrivez les différents cycles d'horloge.

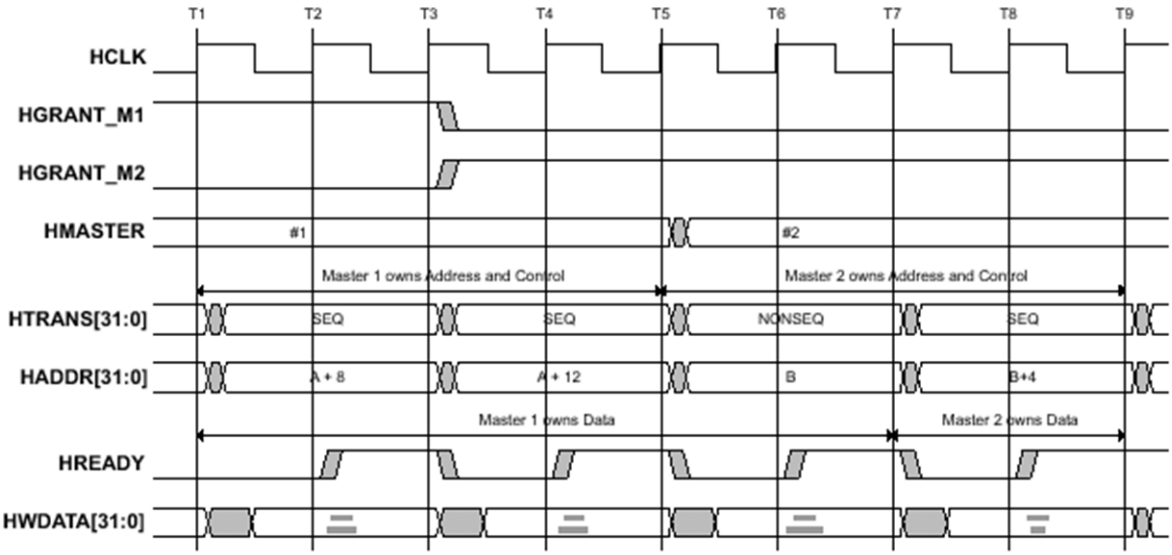


Figure 4.2

d) (1 pt) Même questions qu'en c) mais pour le diagramme de la figure 4.3 entre un maître (signaux orangés) et de la DDR (signaux bleu) et expliquez les grandes différences avec le diagramme de la figure 4.2.

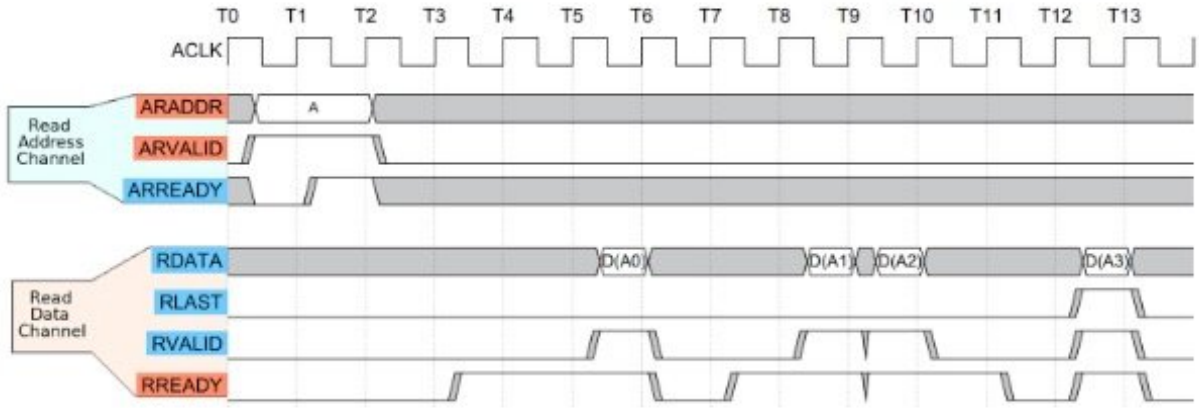


Figure 4.3

Il s'agit de l'exemple dans le bloc 3 du chap. 3. Il s'agit d'une opération de lecture avec AXI pour laquelle l'initiateur (orange) fourni une adresse et la cible fournie 4 données (consécutive) à partir de cette adresse.

Question 5 (3 points) En vrac, laboratoire no 1 et no 2

a) (1 pt) Dans le laboratoire no 1, Partie 3, pourquoi avoir décidé de passer une variable *timedly* pour que *TaskComputing* fasse aussi un délai *OSTimeDlyHMSM(0, 0, delai_pour_vider_les_fifos_sec, delai_pour_vider_les_fifos_msec, OS_OPT_TIME_HMSM_STRICT, &err)*? Justifiez.

Il faut expliquer que TaskComputing n'a pas le temps alors de vider ses fifos et donc rien ne sort du routeur et les fifos finissent par déborder.

b) (1 pt) En principe à la fin du laboratoire no 2, vous auriez dû observer une petite différence d'accélération entre la solution no 5 du design 1 (42 x 42 sans diviser pour régner) et la solution no 1 du design 1 (42 x 42 avec diviser pour régner), c'est-à-dire approximativement 45 vs 42 respectivement. Expliquez en quoi cette petite différence est attribuable?

Diviser pour régner appel 4 fois donc 4 fois plus de communication à travers le bus HP et le DMA.

c) (1 pt) Les figures 5.1 et 5.2 illustrent deux résultats d'implémentations de synthèse de haut niveau classiques. Expliquez ce que représentent ces 2 implémentations, ainsi que les avantages d'un par rapport à l'autre.

La figure 5.1 est un accumulateur sériel i.e. O(N) en ressources où N est le nombre d'étages (p.e. N=50 dans le lab 2) alors que la figure 5.2 est un arborescence donc O(log N).

d)

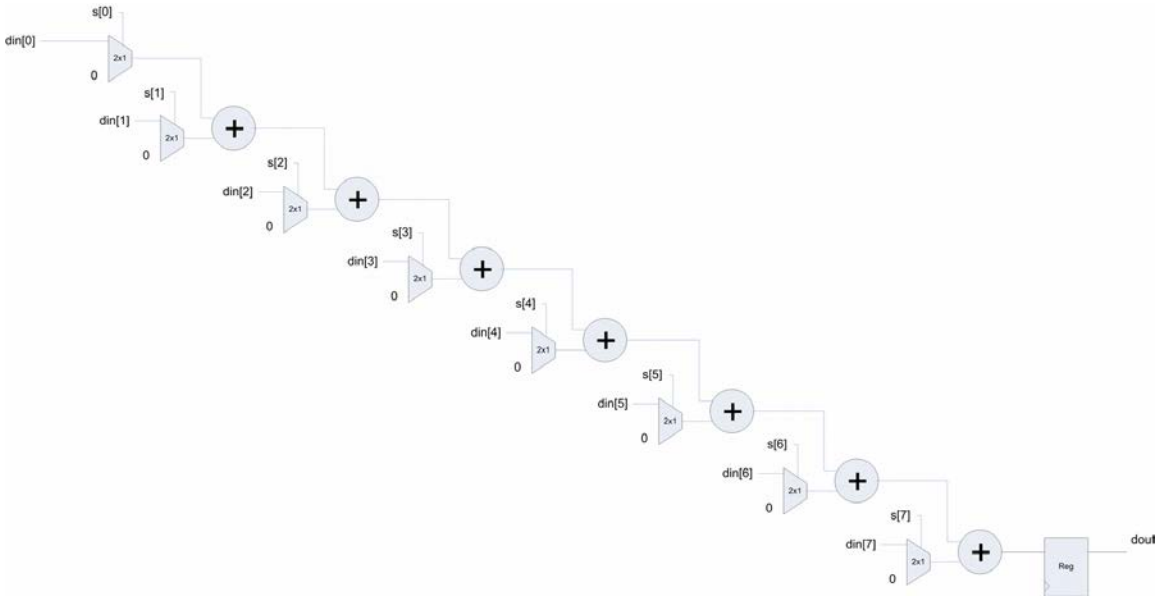


Figure 5.1

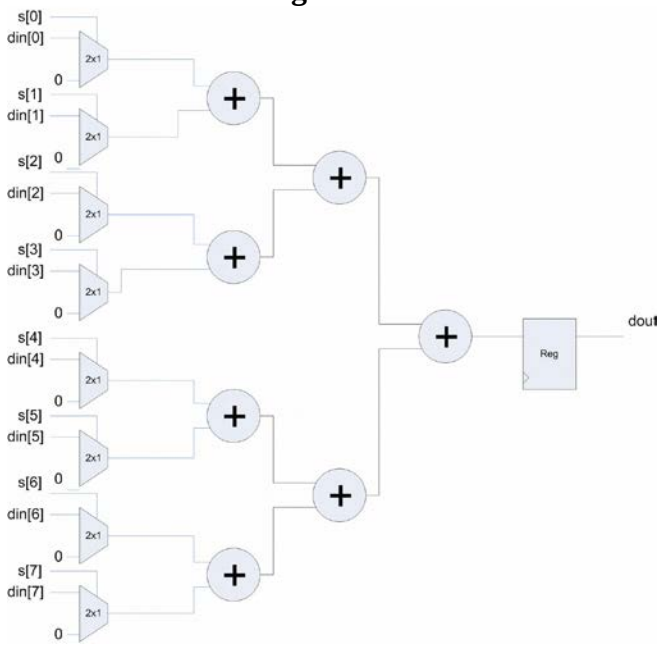


Figure 5.2

Question 6 (1.5 points) Blocage

a) (.75 pt) Expliquez la différence entre préemption et blocage dans le contexte de l'ordonnancement des tâches de priorités différentes et utilisant des mutex.

Préemption quand on a une tâche (de faible priorité) n'avance plus à cause de tâches plus prioritaires et blocage quand une tâche de haute priorité n'avance plus à cause de l'attente d'un mutex bloqué par une tâche moins prioritaire.

b) (.75 pt) Soit 4 tâches T1, T2, T3 et T4 partageant 5 ressources critiques A, B, C, D et E chacune gérée par 5 mutex. Le tableau 6.1 indique pour chaque ressource, le temps d'exécution maximal demandé par chaque tâche (un 0 indique que la ressource n'est pas utilisée par la tâche).

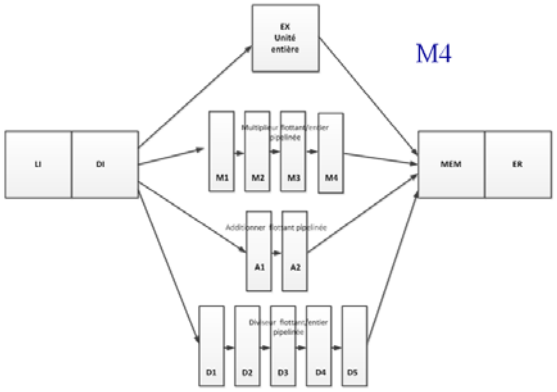
	Priorité	A	B	C	D	E
T1	1	12	5	9	8	0
T2	2	10	0	7	0	6
T3	3	8	3	5	7	13
T4	4	10	0	8	0	5

Tableau 6.1

En considérant le protocole héritage de priorité et une assignation des priorités selon uC/OS-III, estimez une borne supérieure sur le temps de blocage maximum de la tâche T2. Expliquez clairement vos calculs.

Pour l'héritage de priorité, T2 peut être bloquée par le maximum d'exécution de chaque ressource critique qu'il partage avec les autres tâches : max(A) + max(C) + max(D) = max(8, 10) + max(5, 8) + max(13, 5) = 10 + 8 + 13 = 31 cycles.

•Modèle pipeline étendu aux opérations flottantes.



- **Un seul accès d'instruction en mémoire à la fois**
- **Un seul accès de données en mémoire à la fois**

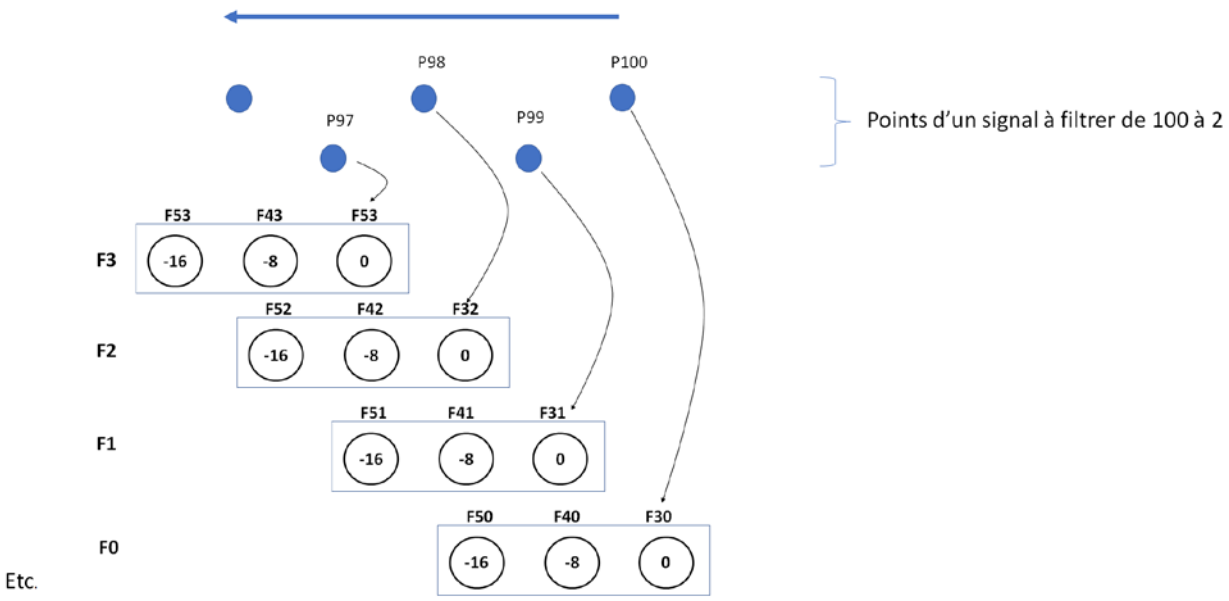
Détails des instructions pouvant être pipelinées	**Nom de l'instruction**	**Nombre de cycles dans EX**	**Cycle du pipeline où l'opération termine**
LD F10, 0(R1)	**Charg. mot dans F10**	**1**	**ER (le résultat est dans l'accumulateur après MEM)**
LD F10, #0	**Charge directement à 0**	**1**	**ER**
ADDD F20, F0, F10	**Additionne double mots F0 et F10**	**3**	**ER (le résultat est mis dans l'accumulateur après EX3)**
MULTD F20, F0, F10	**Multiplie double mots F0 et F10**	**6**	**ER (le résultat est mis dans l'accumulateur après EX6)**
ADDI R1, R1, #8	**Addition immédiate**	**1**	**ER (le résultat est mis dans l'accumulateur après EX)**
SUBI R1, R1, #8	**Soustration immédiate**	**1**	**ER (le résultat est mis dans l'accumulateur après EX)**
SEQI R5, R4, #val	**si (R4 = #val) alors R5 <= 1 sinon R5 <= 0**	**1**	**ER (le résultat est mis dans l'accumulateur après EX)**
SD 0(R2), F6	**Rang. d'un mot à partir de F6**	**1**	**MEM**
BEQZ R3, etiq	**Branch. si zéro**	**1**	**EX**

Tableau 1.1

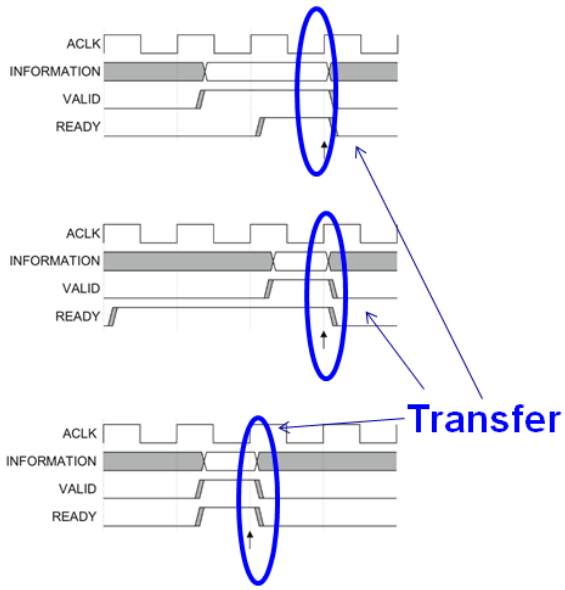
Note aussi que :

- Une seule lecture d'instruction par cycle est permise et une seule lecture/écriture de données par cycle est permise
- On peut avoir plusieurs opérations en parallèle dans EX mais pas dans les autres étages.

Figure A1. Détail des instructions du DLX pour no 1



- Le transfert a lieu seulement quand:
  - La source est Valid, et
  - la destination est Ready
- Sur chaque canal, le maître et l'esclave peuvent limités le flôt.
- Très flexible



- **Avec l'héritage de priorité on a:**

$$B_i = \sum_{k=1}^{K} usage(k,i)CS(k)$$

Où:

- ***usage*** **est une fonction 0/1: *usage*(*k*,*i*) = 1 si la ressource *k* est utilisé par au moins une tâche de priorité inférieure à *i*, et au moins une tâche de priorité supérieure ou égale à *i*, sinon 0.**
- **CS(*k*) le temps requis pour passer au travers la section critique *k*.**

pragma HLS unroll

Description

Unroll loops to create multiple independent operations rather than a single collection of operations. The UNROLL pragma transforms loops by creating multiples copies of the loop body in the register transfer level (RTL) design, which allows some or all loop iterations to occur in parallel.

Syntax

Place the pragma in the C/C++ source within the body of the loop to unroll.

```
#pragma HLS unroll factor=<N> region skip_exit_check
```

Where:

- `factor=<N>`: Specifies a non-zero integer indicating that partial unrolling is requested. The loop body is repeated the specified number of times, and the iteration information is adjusted accordingly. If `factor=` is not specified, the loop is fully unrolled.
- `region`: An optional keyword that unrolls all loops within the body (region) of the specified loop, without unrolling the enclosing loop itself.
- `skip_exit_check`: An optional keyword that applies only if partial unrolling is specified with `factor=`. The elimination of the exit check is dependent on whether the loop iteration count is known or unknown:
  - Fixed (known) bounds: No exit condition check is performed if the iteration count is a multiple of the factor. If the iteration count is not an integer multiple of the factor, the tool:
    1. Prevents unrolling.
    2. Issues a warning that the exit check must be performed to proceed.
  - Variable (unknown) bounds: The exit condition check is removed as requested. You must ensure that:
    1. The variable bounds is an integer multiple of the specified unroll factor.
    2. No exit check is in fact required.

pragma HLS pipeline

Description

The PIPELINE pragma reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations.

Syntax

Place the pragma in the C source within the body of the function or loop.

```
#pragma HLS pipeline II=<int> enable_flush rewind
```

Where:

- `II=` <int>: Specifies the desired initiation interval for the pipeline. The HLS tool tries to meet this request. Based on data dependencies, the actual result might have a larger initiation interval. The default II is 1.
- `enable_flush`: Optional keyword that implements a pipeline that will flush and empty if the data valid at the input of the pipeline goes inactive.

  TIP: This feature is only supported for pipelined functions: it is not supported for pipelined loops.

- `rewind`: Optional keyword that enables rewinding, or continuous loop pipelining with no pause between one loop iteration ending and the next iteration starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:
  - Is considered as initialization.
  - Is executed only once in the pipeline.
  - Cannot contain any conditional operations (if-else).

  TIP: This feature is only supported for pipelined loops; it is not supported for pipelined functions.

pragma HLS array_partition

Description

Partitions an array into smaller arrays or individual elements and provides the following:

- Results in RTL with multiple small memories or multiple registers instead of one large memory.
- Effectively increases the amount of read and write ports for the storage.
- Potentially improves the throughput of the design.
- Requires more memory instances or registers.

Syntax

Place the pragma in the C source within the boundaries of the function where the array variable is defined.

```
#pragma HLS array_partition variable=<name> \
<type>  factor=<int>  dim=<int>
```

where

- `variable=<name>`: A required argument that specifies the array variable to be partitioned.
- <type>: Optionally specifies the partition type. The default type is `complete`. The following types are supported:
  - `cyclic`: Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. For example, if `factor=3` is used:
    - Element 0 is assigned to the first new array
    - Element 1 is assigned to the second new array.
    - Element 2 is assigned to the third new array.
    - Element 3 is assigned to the first new array again.
  - `block`: Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the `factor=` argument.
  - `complete`: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. This is the default <type>.
- `factor=<int>`: Specifies the number of smaller arrays that are to be created.

  IMPORTANT: For complete type partitioning, the factor is not specified. For block and cyclic partitioning the `factor=` is required.

- `dim=<int>`: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to <N>, for an array with <N> dimensions:
  - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
  - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.