QUESTION #1 (5.5 points) Architecture RISC scalaire et aléas

Soit un pipeline DLX de type M4 à cinq (5) niveaux et soit la boucle suivante (Figure 1.1) qui réalise l'opération vectorielle Y[i] = (X[i] + Y[i] / a) * Z[i] où *a* est une constante (F10) et X, Y et Z sont des vecteurs de dimension 120 de mots doubles (point flottant). Les éléments de X, Y et Z sont chargés dans F20, F30 et F40 respectivement, à partir des indices R1, R2 et R3 qui eux sont initialisés respectivement par 960, 1960 et 2960.

```
L:  LD      F30, 0(R2)
    DIVD    F30, F30, F10
    LD      F20, 0(R1)
    ADDD    F30, F30, F20
    LD      F40, 0(R3)
    MULTD   F40, F40, F30
    SD      0(R2), F40
    SUBI    R1, R1, #8
    SUBI    R2, R2, #8
    SUBI    R3, R3, #8
    BNEQZ   R1, L
```

Figure 1.1 Assembleur de l'opération vectorielle

Instructions pipelinées	**Signification**	**Nombre de cycles dans EX**	**Cycle du pipeline où l'opération termine**
LD F20, 0(R1)	Chargement d'un mot dans F20	1	ER (le résultat est dans l'accumulateur après MEM)
MULTD F40, F40, F30	Multiplie les double mots F40 et F30 et le résulta est mis dans F40	3	ER (le résultat est dans l'accumulateur après EX3)
DIVD F30, F30, F10	Divise le double mots F30 par F10 et le résultat est mis dans F30	5	ER (le résultat est dans l'accumulateur après EX5)
ADDD F30, F30, F20	Additionne les double mots F30 et F20 et le résultat est mis dans F30	2	ER (le résultat est dans l'accumulateur après EX2)
SUBI R1, R1, #32	Soustraction immédiate	1	ER (le résultat est dans l'accumulateur après EX)
SD 0(R3), F40	Rangement d'un mot à partir de F40	1	MEM
BNEQZ R2, L	Branchement si différent de 0	1	EX

Tableau 1.1 Détail des instructions du RISC. De plus, considérez qu'il s'agit d'un modèle M4 et qu'on a un seul port de mémoire aux étapes de LI et de ME du pipeline

À partir de cette boucle et à l'aide du tableau 1.1:

a) (**1.5 pt**) Complétez la trace de ce programme (Tableau 1.2) pour l'exécution d'une seule boucle pour le pipeline DLX à cinq (5) étages avec les unités points flottants pipelinés. Donnez le nombre de cycles pour 120 itérations.

*b) 120 * 21 = 2520 cycles par itération*

##		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
1	L:	LD F30, 0(R2)	*LI*	*DI*	*EX*	*ME*	*ER*																							
2		DIVD F30, F30, F10		*LI*	*DI*	*SU*	*E1*	*E2*	*E3*	*E4*	*E5*	*ME*	*ER*																	
3		LD F20, 0(R1)			*LI*	*SU*	*DI*	*EX*	*ME*	*ER*																				
4		ADDF F30, F30, F20					*LI*	*DI*	*SU*	*SU*	*SU*	*E1*	*E2*	*ME*	*ER*															
5		LD F40, 0(R3)						*LI*	*SU*	*SU*	*SU*	*DI*	*EX*	*SU*	*ME*	*ER*														
6		MULTD F40, F40, F30										*LI*	*DI*	*SU*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*										
7		SD 0(R2), F40											*LI*	*SU*	*SU*	*DI*	*EX*	*SU*	*SU*	*ME*	*ER*									
8		SUBI R1, R1, #8														*LI*	*DI*	*SU*	*SU*	*EX*	*ME*	*ER*								
9		SUB R2, R2, #8															*LI*	*SU*	*SU*	*DI*	*EX*	*ME*	*ER*							
10		SUBI R3, R3, #8																		*LI*	*DI*	*EX*	*ME*	*ER*						
11		BNEQZ R1, L																			*LI*	*DI*	*EX*							
																						SU	*SU*	*LI*						

Tableau 1.2 : À compléter pour la question a)

c) (**2 pts**) Dérouler le code du Tableau 1.2 et le réordonnancer afin d'enlever complètement les suspensions. Pour cela vous devez :

b.1) déterminer d'abord le nombre optimal de déroulements,

b.2) dérouler et ordonnancer en complétant le tableau 1.3 à la page suivante, et

b.3) donner le nombre de cycles pour 120 itérations et comparer le gain (nombre de cycles pour 120 itérations) par rapport à a).

Remarques :

1) le choix des numéros de registre (F20, F30 et F40) a volontairement été espacé pour que vous puissiez, au besoin, en introduire de nouveaux (p.e. F21, F22, etc.).

*2) **si à la fin il est impossible d'enlever toutes les suspensions, vous devrez expliquer pourquoi**.*

b.1)
On doit dérouler au moins 5 fois (latence du div). De plus 120 se divise parfaitement avec 5.
b.2)

Cycle	Pipeline 1		Pipeline 1 (suite)
1	*LD F30, 0(R2)*	35	*SD -32(R2), F44*
2	*LD F31, -8(R2)*	36	*SUBI R1, R1, #40*
3	*LD F32, -16(R2)*	37	*BNEQZ R1, L*
4	*LD F33, -24(R2)*	38	*SUBI R2, R2, #40*
5	*LD F34, -32(R2)*	39	*SUBI R2, R2, #40*
6	*LD F20, 0(R1)*	40	
7	*LD F21, -8(R1)*	41	
8	*LD F22, -16(R1)*	42	
9	*LD F23, -24(R1)*	43	
10	*LD F24, -32(R1)*	44	
11	*LD F40, 0(R3)*	45	
12	*LD F41, -8(R3)*	46	
13	*LD F42, -16(R3)*	47	
14	*LD F43, -24(R3)*	48	
15	*LD F44, -32(R3)*	49	
16	*DIVD F30, F30, F10*	50	
17	*DIVD F31, F31, F10*	51	
18	*DIVD F32, F32, F10*	52	
19	*DIVD F33, F33, F10*	53	
20	*DIVD F34, F34, F10*	54	
21	*ADDD F30, F30, F20*	55	
22	*ADDD F31, F31, F21*	56	
23	*ADDD F32, F32, F22*	57	
24	*ADDD F33, F33, F23*	58	
25	*ADDD F34, F34, F24*	59	
26	*MULTD F40, F40, F30*	60	
27	*MULTD F41, F41, F31*	61	
28	*MULTD F42, F42, F32*	62	
29	*MULTD F43, F43, F33*	63	
30	*MULTD F44, F44, F34*	64	
31	*SD 0(R2), F40*	65	
32	*SD -8(R2), F41*	66	
33	*SD -16(R2), F42*	67	
34	*SD -24(R2), F43*	68	

Tableau 1.3 À compléter pour la question 1b)

*b.3) 39*120/5 = 936 cycles au total donc une accélération de2520/936 = 2.69*
n.b. On a rempli toutes les suspensions…

d) (**2 pts**) Soit à nouveau le code de la figure 1.1 et du tableau 1.1 du no 1 et soit une machine VLIW à 5 unités de transfert : 2 pour les accès mémoire, 2 pour les instructions points flottants et 1 pour les instructions entières et de branchement. Vous devez :
c.1) ordonnancer la boucle dépliée obtenue au no 1b) en ciblant cette fois une machine VLIW et en complétant le tableau 1.4,
c.2) indiquer par des flèches le respect des dépendances de données, et
c.3) donner le nombre de cycles pour 120 itérations et comparer le gain (nombre de cycles pour 120 itérations) par rapport à a).

*Remarques : **si à la fin il est impossible d'enlever toutes les suspensions, vous devrez expliquer pourquoi**.*

	Unité transfert 1	Unité transfert 2	Unité exécution 1	Unité exécution 2	Unité entière
1	*LD F30, 0(R2)*	*LD F31, -8(R2)*			
2	*LD F32, -16(R2)*	*LD F33, -24(R2)*			
3	*LD F34, -32(R2)*	*LD F20, 0(R1)*	*DIVD F30, F30, F10*	*DIVD F31, F31, F10*	
4	*LD F21, -8(R1)*	*LD F22, -16(R1)*	*DIVD F32, F32, F10*	*DIVD F33, F33, F10*	
5	*LD F23, -24(R1)*	*LD F24, -32(R1)*	*DIVD F34, F34, F10*		
6	*LD F40, 0(R3)*	*LD F41, -8(R3)*			
7	*LD F42, -16(R3)*	*LD F43, -24(R3)*			
8			*ADDD F30, F30, F20*	*ADDD F31, F31, F21*	
9			*ADDD F32, F32, F22*	*ADDD F33, F33, F23*	
10			*ADDD F34, F34, F24*	*MULTD F40, F40, F30*	
11			*MULTD F42, F42, F32*	*MULTD F41, F41, F31*	
12			*MULTD F44, F44, F34*	*MULTD F43, F43, F33*	
13	*SD 0(R2), F40*				
14	*SD -16(R2), F42*	*SD -8(R2), F41*			*SUBI R1, R1, #40*
15	*SD -32(R2), F44*	*SD -24(R2), F43*			*BNEQZ R1, L*
16					*SUBI R2, R2, #40*
17					*SUBI R2, R2, #40*
18					
19					
20					
21					
22					

Tableau 1.4 : À compléter

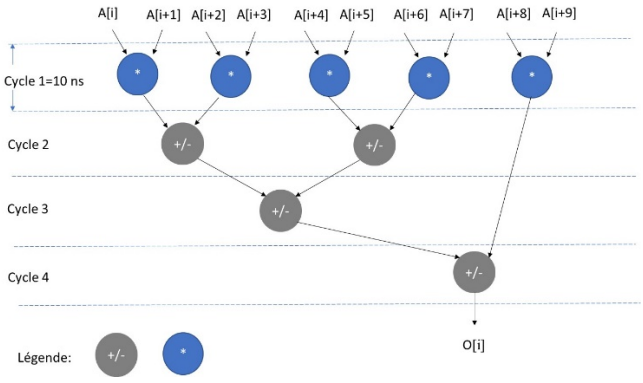
*18*120/5 = 528 cycles au total donc une accélération de 2520/432 = 5.83*

QUESTION #2 (4.5 points) HLS et optimisation

Soit l'expression en langage C :

O[i] := ((A[i] * A[i+1]) + (A[i+2] * A[i+3])) – ((A[i+4] * A[i+5]) + (A[i+6] * A[i+7])) + (A[i+8] * A[i+9])

qu'on retrouve dans une boucle pour laquelle *i* varie de 1 à 100 à travers un tableau A de dimension 109, alors que O est un tableau de dimension 100. Un ordonnancement pipeliné et ses caractéristiques sont donnés à la figure 2.1.



a)

Caractéristiques

Ressources : 5 multiplieurs et 2 additionneurs/soustracteurs

Latence : 4 * 10ns = 40 ns

Débit : II=4 cycles i.e. 1 résultat à tous les 40 ns

b)

Figure 2.1 Ordonnancement pipeliné avec II=4

a) (**1.5 pt**) À l'aide de l'outil Vivado HLS et de certains pragma, expliquez comment on peut optimiser davantage la caractéristique de débit de la figure 2.1b pour obtenir II=1. Indiquez également si des ressources supplémentaires sont requises.
Pragma requis : #pragma HLS pipeline II=1 et aussi #pragma HLS array_partition variable=A block factor=5
Ressources : 5 multiplieurs + 4 additionneurs/soustracteur

b) (**2 pts**) Montrez ce que serait le résultat d'ordonnancement avec l'optimisation appliquée en a) pour les 5 premières itérations de la boucle. Pour cela, complétez la figure 2.2 de la page suivante. Spécifiez bien les entrées de chaque itération.

c) (**1 pt**) Toujours avec l'optimisation en a) et à l'aide de la figure obtenue en b), indiquez le temps total d'exécution pour 100 itérations.
*Temps total : 40 ns + 99 * 10ns = 1030 ns*

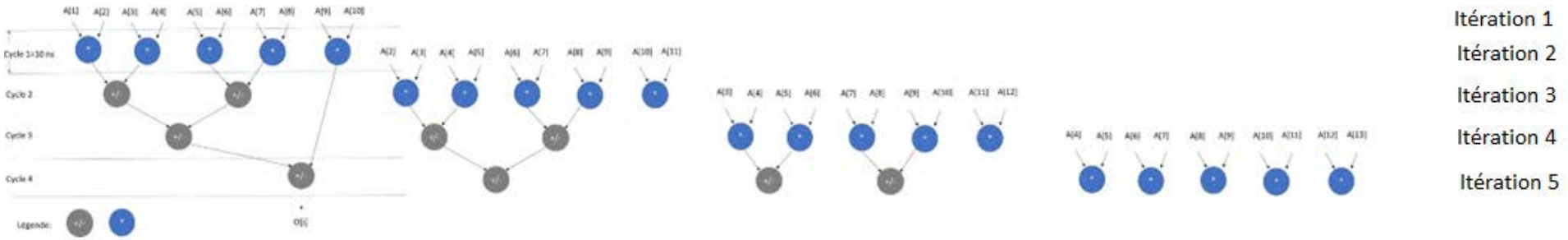


Tableau 2.1 À compléter

QUESTION #3 (3 points) AHB et interconnexions AXI

a) **(1 pt)** J'ai montré en classe qu'une transaction se divise en 4 étapes de base et que la 4e étape correspond à indiquer au maître le statut dans lequel s'est terminé la transaction. Dans le cas du AHB l'esclave (la cible) peut fournir 4 réponses différentes à travers HRESP : OKAY, ERROR, RETRY et SLIT. Expliquez chacune des réponses et ce qui les différencie.

OKAY : Transaction terminé correctement

ERROR : Erreur, on doit tout reprendre depuis le début

RETRY : Erreur, on doit reprendre à partir de la dernière transaction immédiatement

SPLIT : Erreur, on doit reprendre à partir de la dernière transaction plus tard (quand l'arbitre avisera).

b) **(1 pt)** La figure 3.1 illustre le concept de *wait state*. Que signifie le *wait state* et de quel acteur origine-t-il ? Justifiez.

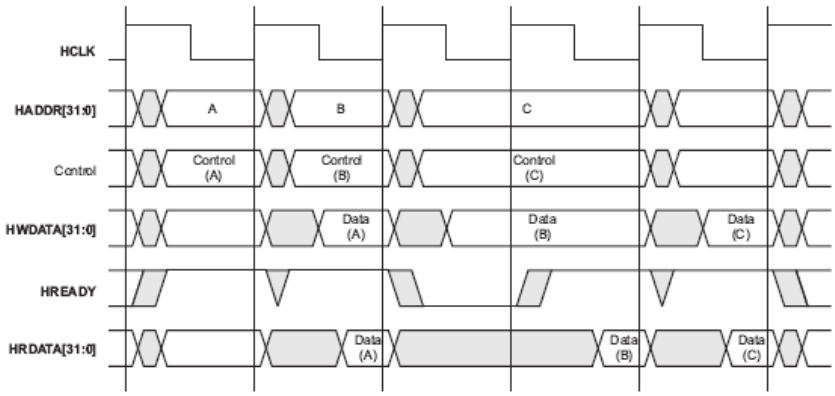


Figure 3.1

Causé par la lenteur d'un périphérique (ex. UART qui demande plus que 1 cycle à répondre.

c) **(1 pt)** Expliquez le schéma de la figure 3.2, le protocole de communication employé derrière cette architecture et ses principales caractéristiques. Donnez finalement ses avantages/inconvénients par rapport au protocole de bus AHB.

Il faut expliquer que le AXI permet de faire des connexions entre chaque maitre et ses n esclaves (n^2) plutôt qu'un nombre constant de connexions comme le AHB. La contention est donc nettement inférieure avec le AXI, par rapport au AHB.

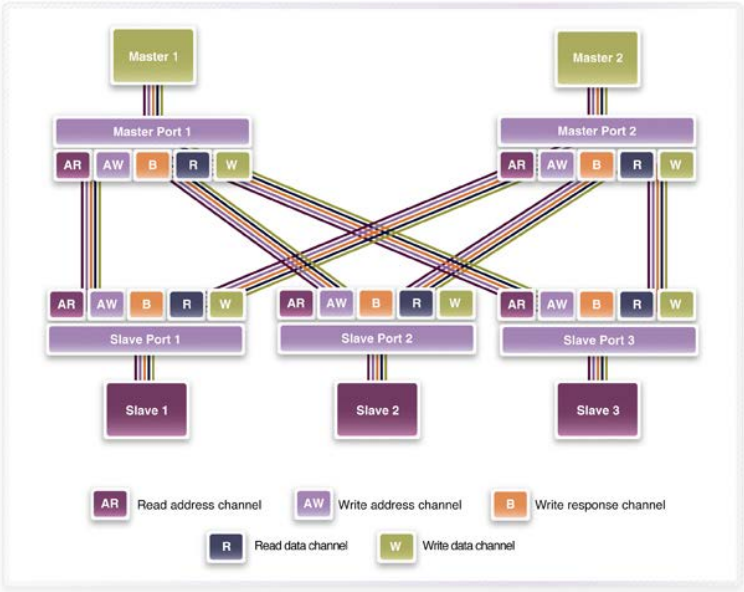

Master 1
Master 2
Master Port 1
Master Port 2
AR
AW
B
R
W
Slave Port 1
Slave Port 2
Slave Port 3
Slave 1
Slave 2
Slave 3
Read address channel
Write address channel
Write response channel
Read data channel
Write data channel

Figure 3.2

QUESTION #4 (4 points) Laboratoires no 2

a) (**2 pts**) Vrai ou faux avec justification pour chacune des 2 affirmations suivantes :

1) Soit la solution 3 de l'exploration architecturale de *hls_design2* avec la demi-précision (*FP16*) pour laquelle vous avez au laboratoire déterminé DIM=54 comme dimension de matrice. En changeant l'option *complete* du pragma HLS *ARRAY_PARTITION* par *block factor = 14* on aurait alors II=3.

*Faux ça serait II=2 car BRAM on 2 ports (14*2) = 28 et on fera 2 lectures donc une latence de 2.*

2) Soit la solution 3 de l'exploration architecturale de *hls_design2* avec la demi-précision (*FP16*). Si on désirait avoir un calcul matriciel sur 2 matrices ayant DIM=20 et II=1, la seule condition manquante au niveau ressource pour que cette multiplication soit synthétisable sur la puce Zynq 7020 est d'avoir 1600 DSP48E (sachant que chaque multiplication/accumulation demande 4 DSP48E).

Faux ! Bien que 20^2 x 4 = 1600, si on fait le array partition de la matrice B sur des BRAM on a besoin de $20^2/2$ blocks. Or Zynq limite à 140 blocks.

b) (**1 pt**) Basé sur vos expérimentations du laboratoire no 2, expliquez la différence entre les figures 4.1 et 4.2 au niveau pour une multiplication de matrice carrée 4 x 4 : profondeur de pipeline, complexité du temps d'exécution et des ressources et besoin au niveau des accès en BRAM.

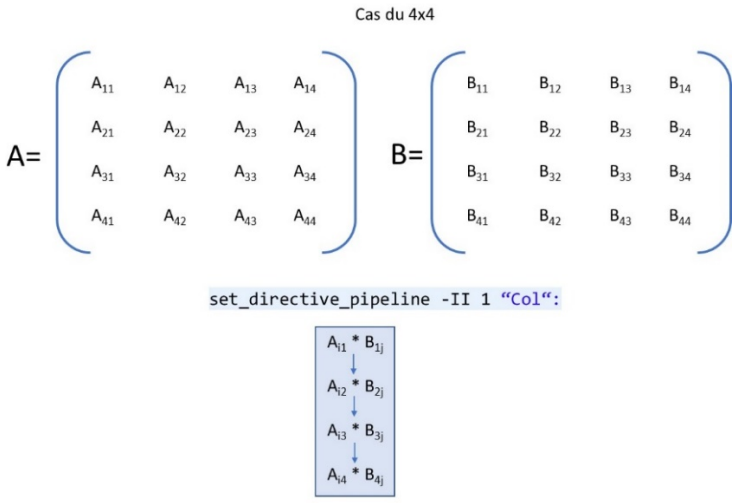


Figure 4.1

$O(N^2)$ en temps d'exécution et $O(N)$ en ressources

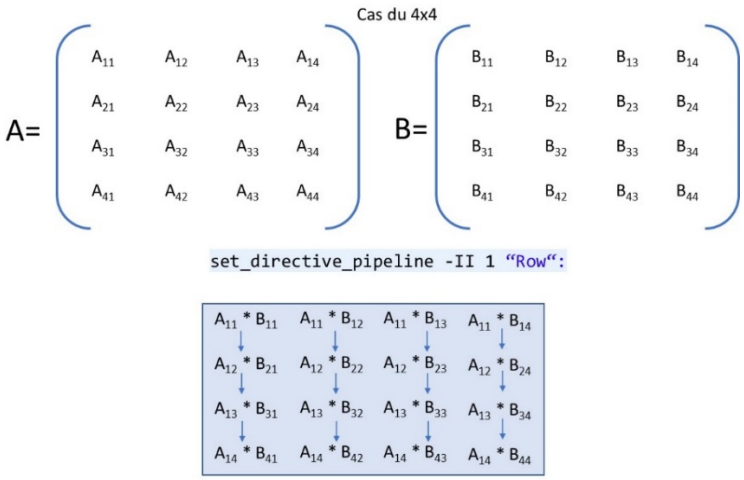


Figure 4.2

O(N) en temps d'exécution et O(N^2) en ressources

c) (**1 pt**) Soit la figure 4.3 qui représente un ordonnancement pour une multiplication de matrice carrée 4 x 4. À quelle figure correspond cet ordonnancement, celui de la fig. 4.1 ou 4.2 ou aucune de ces figures ? Justifiez.

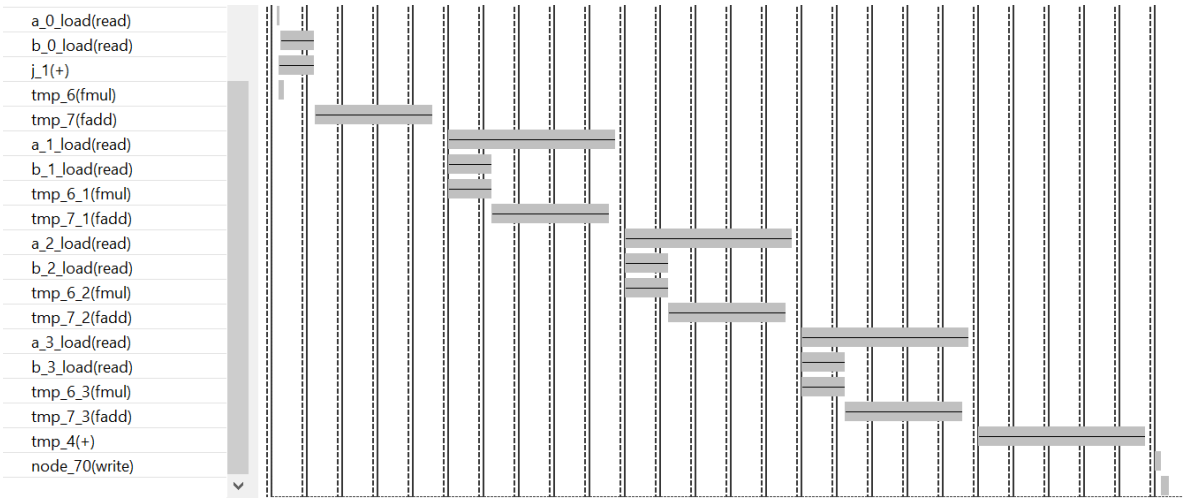
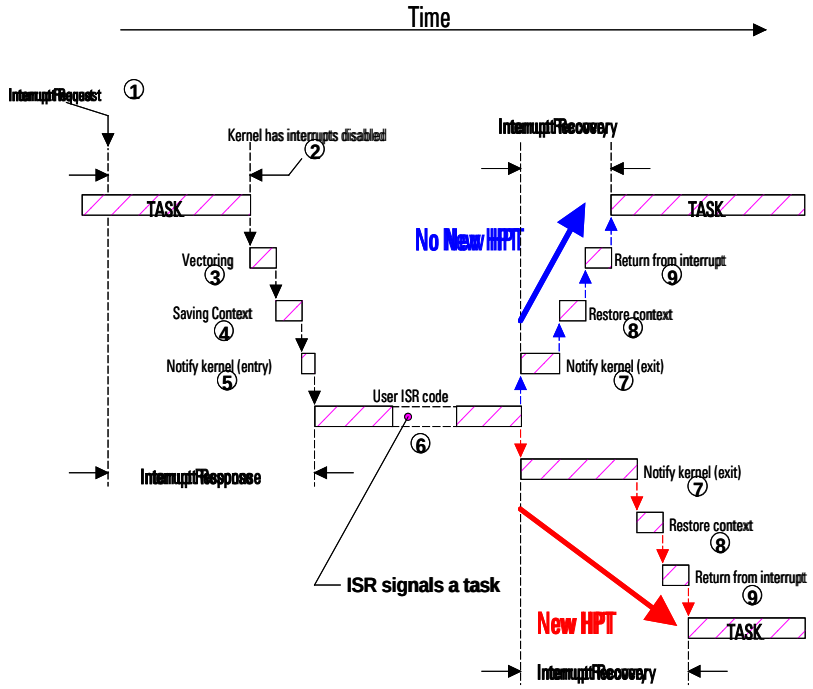


Figure 4.3

Correspond à la Figure 4.1 on a 4 multiplieurs…

QUESTION #5 (3 points) Retour sur l'intra et le laboratoire no 1

a) (**2 pts**) Soit la figure 5.1a) qu'on a vue en classe. On a ajouté à cette figure une liste de fonctions du pilote *Push Button* de la carte PYNQ-Z2 (ou Zedboard) permettant d'exploiter les boutons pressoirs (Fig. 5.1b). Assigner chacune de ces fonctions à un numéro de la figure 5.1a). Justifiez brièvement chaque assignation.



a)

8 fonctions du pilote *Push Button* :

OSIntEnter(),
OSIntExit()
OSPrioGetHighest()
OS_CPU_ARM_EXCEPT_IRQ()
XIntc_DeviceInterruptHandler()
uCOS_IntHandler()
OSIntCtxSw()
gpio_isr()

b)

Figure 5.1

Il s'agit du déroulement en mode préemptif d'une interruption matérielle.

- *OS_CPU_ARM_EXCEPT_IRQ() s'exécute après 3 (la table des vecteurs fait un branchement à l'adresse de cette fonction.*
- *OSIntEnter()*
- *uCOS_IntHandler() est appelé dans le GIC donc dans 5*
- *XIntc_DeviceInterruptHandler()* et *gpio_isr()* sont du code de driver sur FPGA donc appelé dans 6

- *OSIntExit()* permet de voir qu'on a pu d'interruption à traiter et d'appeler *OSPrioGetHighest() pour savoir si on va sur la branche du haut ou celle du bas. Donc appelé de 7.*
- *OSIntCtxSw() Une fois OSIntExit() complété et qu'on a bel et bien pu d'interruption, on peut procéder au changement de contexte. Donc appelé de 8.*

b) (**1 pt**) Soit les **8 fonctions** de uc/OS-III suivantes : *OSIntEnter(), OSIntExit()* et *OSSched(), OSCtxSw(), OSIntCtxSw(), OSPrioGetHighest() et OSStartHighRdy().*

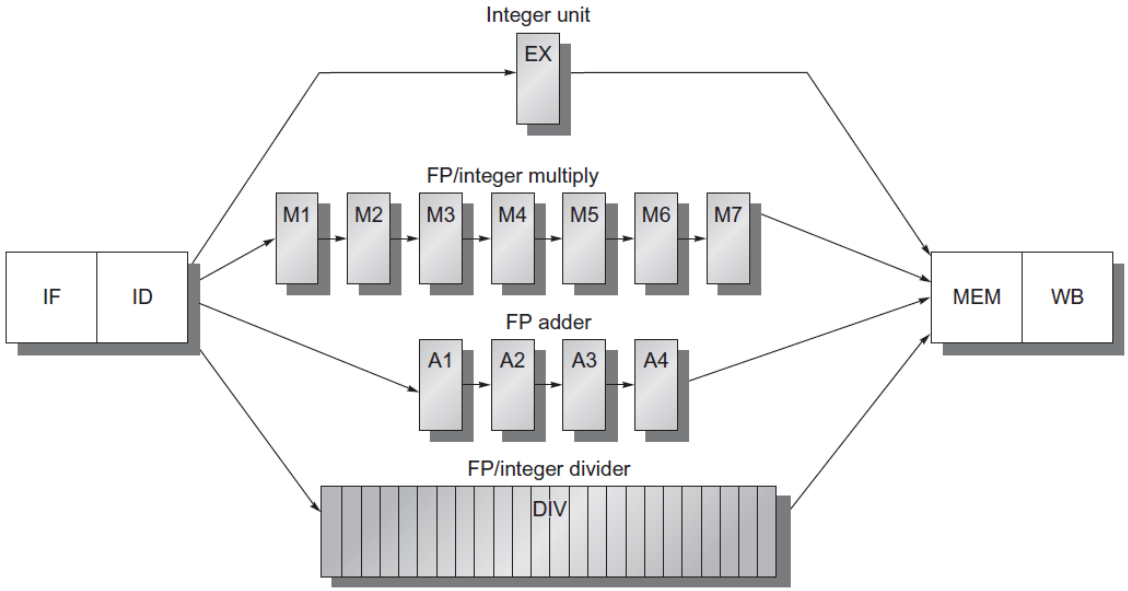
b.1) Lesquelles de ces 8 fonctions sont-elles appelées lors de l'exécution de *OSTimeTick().* Indiquez également dans quel ordre se font les appels.

La fonction OSTimeTick() est équivalent à gpio_isr() de a) donc appelé de l'étiquette 6 d'une interruption. Elle appelle donc OSntEnter() et OSIntExit(). Ce dernier appelle OSPrioGetHighest() puis OSIntCtxSw().

b.2) En supposant un mutex la valeur d'un mutex M à 0, lesquelles de ces 8 fonctions sont-elles appelées lors de l'exécution de *OSMutexPend(&M, …).* Indiquez également dans quel ordre se font les appels.

La fonction OSMutexPend(&M, …) provoque une interruption logicielle. Elle appelle OSSched() qui appelle OSPrioGetHighest() et OSCtxSw().

Annexe



Exemple de modèle M4

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.

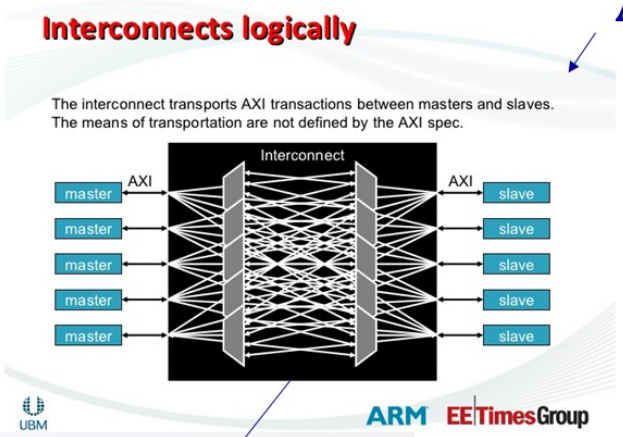
A propos de notre cible SoC

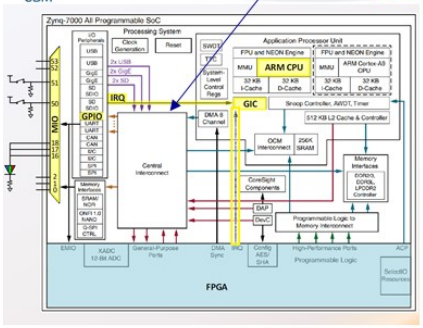
Table 1: **Zynq-7000 and Zynq-7000S SoCs *(Cont'd)***

	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix®-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex®-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA
	Programmable Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Block RAM (# 36 Kb Blocks)	1.8 Mb (50)	2.5 Mb (72)	3.8 Mb (107)	2.1 Mb (60)	3.3 Mb (95)	4.9 Mb (140)	9.3 Mb (265)	17.6 Mb (500)	19.2 Mb (545)	26.5 Mb (755)
	DSP Slices (18x25 MACCs)	66	120	170	80	160	220	400	900	900	2,020
	Peak DSP Performance (Symmetric FIR)	73 GMACs	131 GMACs	187 GMACs	100 GMACs	200 GMACs	276 GMACs	593 GMACs	1,334 GMACs	1,334 GMACs	2,622 GMACs
	PCI Express (Root Complex or Endpoint)(3)		Gen2 x4			Gen2 x4		Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security(2)	AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication									

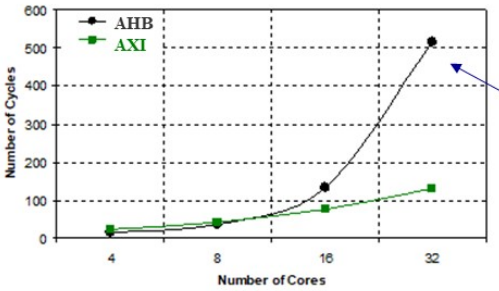
À ajouter

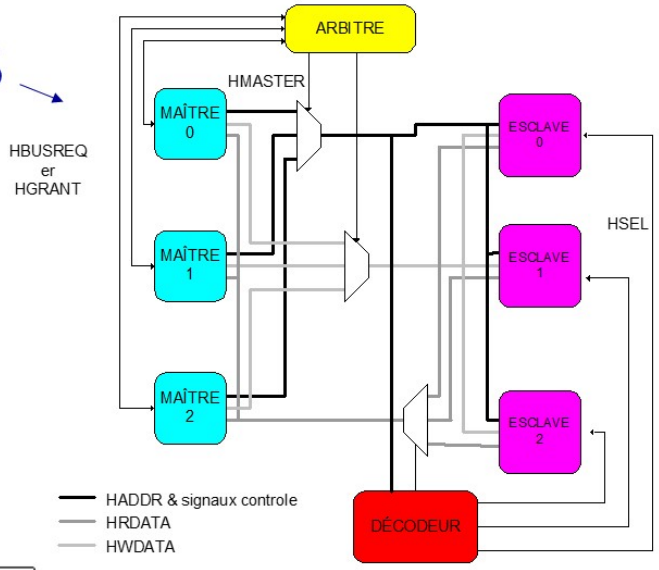
AXI vs AHB





Réf cours no 5





On appelle ça un phénomène de contention de bus (bus contention)

INF3610 - Systèmes embarqués 67