QUESTION #1 (1 point) Vrai ou faux avec justifications

a) OSIntExit() est utilisé dans les interruptions matérielles alors que OSSched() est utilisé dans les interruptions logicielles. Les deux appellent OSPrioGetHighest() et OSIntCtxSw.

Faux OSIntCtxSw n'est pas appelée par OSSched()

b) Lorsqu'on utilise II=2 dans un accélérateur matériel conçu avec Vivado HLS, on doit s'attendre à diviser par 2 les ressources et la performance par rapport à II=1 atteint.

Vrai pour la performance mais pour les ressources je ne dirais pas toujours. Par exemple on a vu que pour 8 ou 16 bits il est préférable de contrôler les ressources avec ARRAY PARTITION block factor. Donc globalement Faux.

QUESTION #2 (6 points) Architecture RISC

Soit la boucle de traitement suivante sur des doubles de 64 bits[1]:

for (i = 200 ; i > 3; i--)

*AB[i] = A[4] * B[i] + A[3] * B[i-1] + A[2] * B[i-2] + A[1] * B[i-3] + A[0] * B[i-4]*

Plus précisément, un vecteur A de dimension 5 parcourt un vecteur B de dimension 200 de gauche à droite et pour chaque position on effectue une multiplication/accumulation et on dépose le résultat dans le vecteur AB. Ici on ne traite pas les effets de bord donc le dernier calcul est sur l'indice 4 (résultat dans AB[4]).

Pour le traitement DLX, on a les assignations suivantes :

- 5 registres F0, F1, F2, F3 et F4 pour emmagasiner les valeurs de A[4] à A[0] respectivement (évite de relire le vecteur à chaque itération).
- 5 registres F10, F20, F30, F40 et F50 pour chaque résultat de multiplication,
- R4 est l'adresse de B
- 1 registre résultant F80 (initialement à 0) pour réaliser un multiplicateur/accumulateur.

Plutôt que de faire tour à tour des LD, MULTD et ADDD pour chaque multiplications/accumulations (donc 5 fois), voici le code DLX avec déjà quelques optimisations :

```
1   L:  LD      F80, #0
2       LD      F10, 0(R4)
3       LD      F20, -8(R4)
4       LD      F30, -16(R4)
5       LD      F40, -24(R4)
6       LD      F50, -32(R4)
7       MULTD   F10, F10, F0
8       MULTD   F20, F20, F1
9       MULTD   F30, F30, F2
10      MULTD   F40, F40, F3
11      MULTD   F50, F50, F4
12      ADDD    F80, F80, F10
13      ADDD    F81, F80, F20
14      ADDD    F80, F80, F30
15      ADDD    F80, F80, F40
16      ADDD    F80, F80, F50
17      SD      4000(R4), F80
18      SUBI    R4, R4, #8
19      SEQI    R5, R4, #32
20      BEQZ    R5, L
```

Figure 2.1

[1] *Ce n'était pas demandé dans l'examen, mais voir page 5 pour une illustration de l'algorithme…*

a) (1 pt) La figure suivante (fig. 2.2) présente l'exécution pour les 11 premières instructions DLX. On observe qu'il n'y aucune suspension du pipeline. Expliquez l'avantage de cette optimisation de code en comparant avec l'approche conventionnelle qui aurait été de faire des LD, ADD et MULT pour 5 multiplications.

##		Instructions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	L:	LD F80, #0	*LI*	*DI*	*EX*	*ME*	*ER*														
2		LD F10, 0(R4)		*LI*	*DI*	*EX*	*ME*	*ER*													
3		LD F20, -8(R4)			*LI*	*DI*	*EX*	*ME*	*ER*												
4		LD F30, -16(R4)				*LI*	*DI*	*EX*	*ME*	*ER*											
5		LD F40, -24(R4)					*LI*	*DI*	*EX*	*ME*	*ER*										
6		LD F50, -32(R4)						*LI*	*DI*	*EX*	*ME*	*ER*									
7		MULTD F10, F10, F0							*LI*	*DI*	*E1*	*E2*	*E3*	*E4*	*E5*	*ME*	*ER*				
8		MULTD F20, F20, F1								*LI*	*DI*	*E1*	*E2*	*E3*	*E4*	*E5*	*ME*	*ER*			
9		MULTD F30, F30, F2									*LI*	*DI*	*E1*	*E2*	*E3*	*E4*	*E5*	*ME*	*ER*		
10		MULTD F40, F40, F3										*LI*	*DI*	*E1*	*E2*	*E3*	*E4*	*E5*	*ME*	*ER*	
11		MULTD F50, F50, F4											*LI*	*DI*	*E1*	*E2*	*E3*	*E4*	*E5*	*ME*	*ER*

Figure 2.2 Pipeline pour les 11 premières instructions.

Les suspensions sont par du code qui doit se faire anyway…

b) (1.75 pt) Complétez la figure 2.3 de la page suivante pour le reste des instructions (instructions 12 à 20) de la figure 2.1 et donnez le nombre d'instructions pour 1 itération de la boucle de code de cette même figure.

c) (1.75 pt) Plutôt que de faire une accumulation itérative sur 1 seul registre (dans R80), on aurait pu utiliser un arbre binaire de sommation avec 2 registres R80 et R81 comme le montre la figure 2.4. Complétez la figure 2.5 de la page suivante pour le reste des instructions (instructions 12 à 20) de la figure 2.1 et donnez le nombre d'instructions pour 1 itération de la boucle de code de cette même figure. Ne pas oublier qu'une instruction s'ajoute entre les lignes 1 et 2 qui serait LD F81, #0.

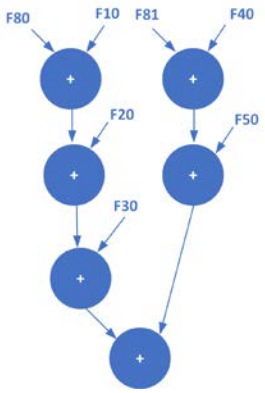


Figure 2.4 Arbre binaire de sommation

##	Instructions	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
12	ADDD F80, F80, F10	*LI*	*DI*	*E1*	*E2*	*E3*	*ME*	*ER*																		
13	ADDD F80, F80, F20		*LI*	*DI*	*SU*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*															
14	ADDD F80, F80, F30			*LI*	*SU*	*SU*	*DI*	*SU*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*												
15	ADDD F80, F80, F40						*LI*	*SU*	*SU*	*DI*	*SU*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*									
16	ADDD F80, F80, F50									*LI*	*SU*	*SU*	*DI*	*SU*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*						
17	SD 4000(R4), F80												*LI*	*SU*	*SU*	*DI*	*EX*	*SU*	*SU*	*ME*						
18	SUBI R4, R4, #8															*LI*	*DI*	*SU*	*SU*	*EX*	*ME*	*ER*				
19	SEQ R5, R4, #32																*LI*	*SU*	*SU*	*DI*	*EX*	*ME*	*ER*			
20	BEQZ R5, L																			*LI*	*SU*	*SU*	*LI*			
21																										

Figure 2.3 Accumulation itérative sur 1 registre et branchement. À compléter.

##	Instructions	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
12	ADDD F80, F80, F10	*LI*	*DI*	*E1*	*E2*	*E3*	*ME*	*ER*																		
13	ADDD F81, F81, F40		*LI*	*DI*	*E1*	*E2*	*E3*	*ME*	*ER*																		
14	ADDD F80, F80, F20			*LI*	*DI*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*															
15	ADDD F81, F81, F50				*LI*	*SU*	*DI*	*E1*	*E2*	*E3*	*ME*	*ER*	*E1*													
16	ADDD F80, F80, F30						*LI*	*DI*	*SU*	*E1*	*E2*	*E3*	*ME*	*ER*												
17	ADDD F80, F80, F81							*LI*	*SU*	*DI*	*E1*	*E2*	*E3*	*ME*	*ER*											
18	SD 4000(R4), F80									*LI*	*DI*	*EX*	*SU*	*SU*	*ME*											
19	SUBI R4, R4, #8										*LI*	*DI*	*SU*	*SU*	*EX*	*ME*	*ER*									
20	SEQ R5, R4, #32											*LI*	*SU*	*SU*	*DI*	*EX*	*ME*	*ER*								
21	BEQZ R5, L														*LI*	*SU*	*SU*	*LI*								
22																										

Figure 2.5 Accumulation avec arbre binaire de sommation. À compléter

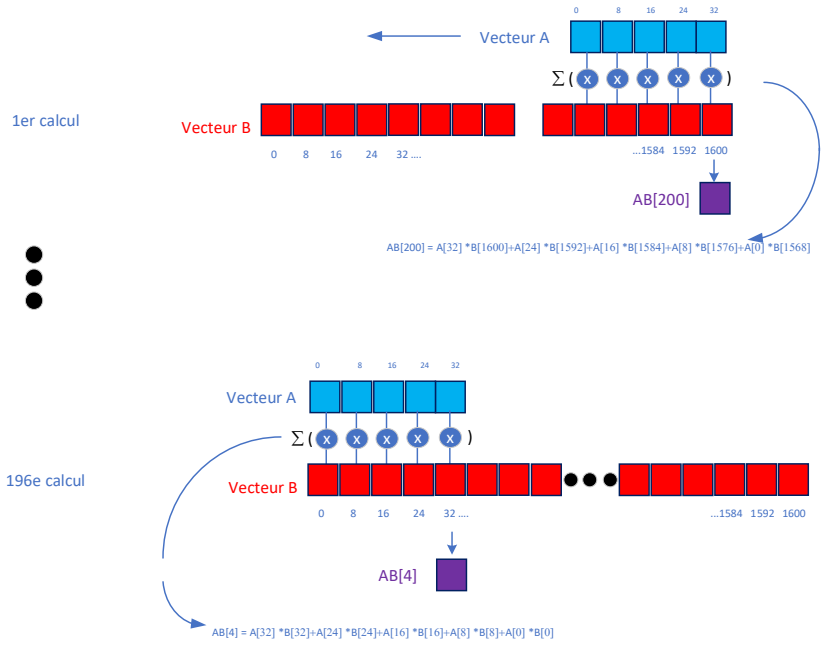
On voit qu'en b) on a besoin de 32 instructions alors qu'en c) on a besoin de 27 instructions

d) (1.5 pts) On suppose maintenant un processeur similaire au Cortex A9 superscalaire (M5) tel que vous avez sur le SoC de votre carte PYNQ Z2 (Processing Subsystem) avec les caractéristiques suivantes :

- 2 chargements et décodage d'instructions à la fois
- 2 unités entières du type ALU (pour instructions comme SUBI, SEQI, BNEQ),
- 2 unités de load/store (LD et SD) et
- 2 unités point flottantes (FP/Neon) capables de faire 2 opérations similaires (ADDD, MULTD, DIVD, etc.) sur des mots de 64 bits.

Complétez la figure 2.6, en assignant chaque opération de la figure 2.1 à un cycle, pour ainsi accélérer encore davantage par rapport à b) et c). Si cela est plus rapide, utilisez l'arbre binaire de sommation. Utilisez des flèches pour montrer les dépendances. Si parfois des suspensions sont requises tentez de les remplacer par du code utile, sinon indiquez-les avec SU. Finalement, donnez le nombre de cycles que prendra l'exécution complète et comparez l'accélération par rapport à b).

Illustration du calcul de la fig 2.1 :



Notez que cette fonctionnalité est celle d'un filtre FIR sauf qu'ici au lieu d'avoir 5 coefficients dans des registres, on les a dans le vecteur A.

Cycle	LD/SD1	LD/SD2	Entier1	Entier2	FP1/Neon1	FP2/Neon2
1 L1	LD F80, #0					
2	LD F10, 0(R4)	LD F40, -24(R4)				
3	LD F20, -8(R4)	LD F50, -32(R4)				
4	LD F30, -16(R4)				MULTD F10, F10, F0	MULTD F40, F40, F3
5					MULTD F20, F20, F1	MULTD F50, F50, F4
6					MULTD F30, F30, F2	
7					SU	SU
8					SU	SU
9					ADDD F80, F80, F10	ADDD F81, F81, F40
10					SU	SU
11					SU	SU
12					ADDD F80, F80, F20	ADDD F81, F81, F50
13					SU	
14					SU	
15					ADDD F80, F80, F30	
16					SU	
17					SU	
18					ADDD F80, F80, F81	
19						
20						
21	SD 4000(R4), F80					
22			SUBI R4, R4, #8			
23			SEQ R5, R4, #32			
24			BEQZ R5, L			

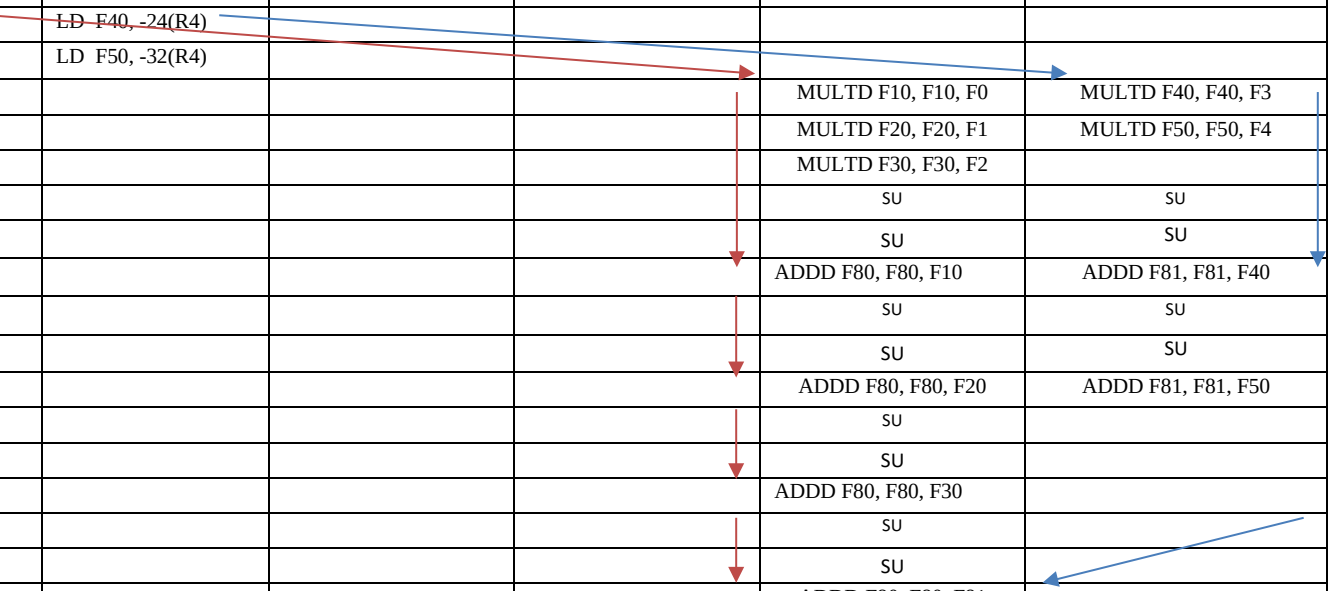


Figure 2.6 À compléter

On a 24 cycles par itération, pourquoi un gain aussi modeste par rapport à c) ? Les SU font mal pour l'arbre d'addition. On aurait eu un plus grand gain avec un arbre binaire de sommation de p.e. 16 ou 32 plutôt que 5. Les additions en parallèle auraient pu remplir efficacement les SU.

QUESTION #3 (4 points) HLS pragma

Soit l'application vectorielle sur le float 64 bits de la figure 3.1 similaire à celle de la question 2 et que l'on souhaite cette fois synthétiser dans l'outil HLS Vivado:

```
 8 typedef float mat_type;
 9
10 int const DIM1 = 5;
11 int const DIM2 = 200;
12
13 void mmult_hw (mat_type A[DIM1], mat_type B[DIM2], mat_type AB[DIM2])
14 {
15     mat_type temp;
16     L1:for (int ia = DIM2; ia > DIM1-1; --ia) {
17         mat_type sum = 0;
18
19         L2:for (int ib = 0; ib < DIM1; ++ib)
20             sum = sum + A[ib] * B[ia-ib];
21
22         AB[ia] = sum;
23         };
24
25     return;
26 };
```

Figure 3.1 *Voir aussi le schéma page 5*

a) (1 pt) Si on exécute ce code sans aucun pragma, estimez le nombre de DSP requis et le nombre de cycles requis sachant qu'une multiplication/accumulation demande 5 DSP en ressources et demande 11 cycles en temps d'exécution. Expliquez le calcul de vos estimations. Avons-nous bien une complexité de O(N*M) où N = DIM1 et M=DIM2 ?

b) (2 pts) Expliquez comment avec l'aide de pragma on peut faire passer la complexité à O(M). Décrivez ce(s) pragma et leur position dans le code (entre quelles lignes). Estimez le nombre de DSP et de BRAM requis et le nombre de cycles.

c) (1 pt) Proposez une approche qui offrirait la même complexité que b) en termes du nombre de cycles d'exécution sans l'utilisation de DSP48E, sachant qu'il existe en librairie des additionneurs/soustracteurs (FAddSup_nodsp) et multiplieurs (FMul_nodsp) conçus à partir de FF et LUT.

a)

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	62
FIFO	-	-	-	-
Instance	-	5	348	711
Memory	-	-	-	-
Multiplexer	-	-	-	86
Register	-	-	123	-
Total	0	5	471	859
Available	280	220	106400	53200
Utilization (%)	0	2	~0	1

Summary

Latency		Interval		
min	max	min	max	Type
11173	11173	11173	11173	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1	11172	11172	57	-	-	196	no
+ L2	55	55	11	-	-	5	no

Notez que 55 x 196 = 10780 était la réponse attendue (ici HLS Vivado estime un petit overhead i.e. 57 au lieu de 55 de traitement de boucle…). On a donc 5 x DIM1 x DIM2 = 5 x 11 x 196 et on a bien une complexité de O(M x N).

b) Page suivante

- mat_type
- DIM1
- DIM2
- mmult_hw
  - A
  - % HLS ARRAY_PARTITION variable=A complete dim=1
  - B
  - % HLS ARRAY_PARTITION variable=B complete dim=1
  - AB
  - L1
    - % HLS PIPELINE II=1
    - L2

P.S. dim = 0 fonctionne aussi car on a ici un vecteur… Dans les 2 cas (dim = 1 ou 0) on passe de vecteurs à des registres

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	29256
FIFO	-	-	-	-
Instance	-	25	1740	3555
Memory	-	-	-	-
Multiplexer	-	-	-	910
Register	0	-	2386	416
Total	0	25	4126	34137
Available	280	220	106400	53200
Utilization (%)	0	11	3	64

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1	232	232	38	1	1	196	yes

Notez qu'on ne voit pas la BRAM, probablement parce qu'elle utilise des LUT (plus difficile à voir). Mais si on fait le calcul comme vu en classe on aura pour A plafond de 5/2 = 3 BRAM et pour B on aura plafond de 200/2 = 100.

Evidemment ce résultat sur loop ne pouvait être obtenu à l'examen mais c'est juste pour montrer qu'en utilisant un pipeline et en permettant des accès en parallèle sur les vecteurs A et B (avec ARRAY PARTITION ci-haut) on a 38 (latence pour remplir le pipeline) + 196 itération de boucles donc 232 cycles de temps total d'exécution. On est bel et bien dans une complexité de O(M) si on compare avec a)….

Chose bizarre que j'ai noté en faisant ce numéro sur Vivado HLS : quand j'utilise les pragma suivant :

- mat_type
- DIM1
- DIM2
- mmult_hw
  - A
  - % HLS ARRAY_PARTITION variable=A block factor=3 dim=1
  - B
  - % HLS ARRAY_PARTITION variable=B block factor=100 dim=1
  - AB
  - L1
    - % HLS PIPELINE II=1
    - L2

Je note l'erreur suivant :

Name	Details
All Categories	
SCHEDULE	
[SCHED 204-61]	Option 'relax_ii_for_timing' is enabled, will increase II to preserve clock frequency constraints.
[SCHED 204-69]	Unable to schedule 'load' operation ('B_98_load_2', mmult_accel.cpp:20) on array 'B_98' due to limited memory ports. Please consider using a memory core with more
[SCHED 204-69]	Unable to schedule 'load' operation ('B_97_load_4', mmult_accel.cpp:20) on array 'B_97' due to limited memory ports. Please consider using a memory core with more

Ça dit qu'il n'y a pas assez de BRAM et il me donne un II=3 :

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1	616	616	32	3	1	196	yes

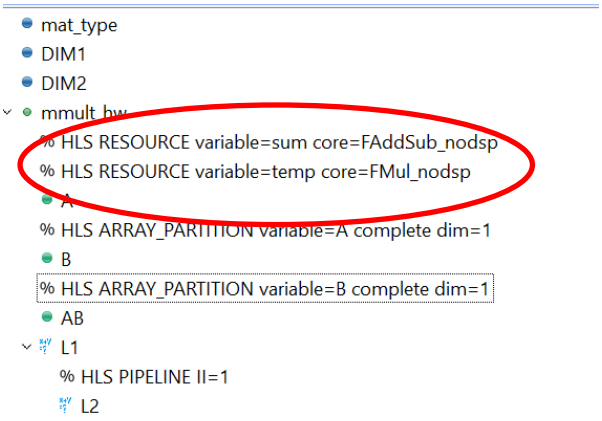
Comment peut-il alors avoir une solution avec II=1. Encore une fois ça reste des estimations… Peut être qu'à la synthèse final on aurait II=3… à moins de baisser la dimension du tableau B…

c)

On pourrait séparer le multiplication accumulation :

```
mat_type temp;
L1:for (int ia = DIM2; ia > DIM1-1; --ia) {
    mat_type sum = 0;
    L2:for (int ib = 0; ib < DIM1; ++ib) {
            temp = A[ib] * B[ia-ib];
               sum = sum + temp;
            }
            AB[ia] = sum;
        };
        return;
```

Et ajouter les pragma suivants



Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	29256
FIFO	-	-	-	-
Instance	-	0	2345	7290
Memory	-	-	-	-
Multiplexer	-	-	-	910
Register	0	-	2467	416
Total	0	0	4812	37872
Available	280	220	106400	53200
Utilization (%)	0	0	4	71

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1	233	233	39	1	1	196	yes

Notez que la même interrogation persiste pour l'utilisation d'un block factor = 100…

QUESTION #4 (2 points) Résultats en ressources du laboratoire no 2

Soit la solution 3.2 du laboratoire pour une matrice 160x160 sur des éléments 8 bits avec DSP48E. La figure 4.1 présente les résultats de ressources à 3 étapes différentes de la synthèse :

1) Vivado HLS en exploration architecturale (fig. 4.1a),
2) Vivado HLS lors de la synthèse de l'IP (fig. 4.1b) et
3) Vivado lors de l'assemblage final pour une exécution dans SDK (fig. 4.1c).

Expliquez les différences entre ces 3 rapports : représentent-ils des estimations et/ou des résultats finaux? Qu'est-ce qui les différencie(nt)?

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	96	-	-
Expression	-	0	0	3386
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	75
Register	0	-	2689	96
Total	0	96	2689	3557
Available	280	220	106400	53200
Utilization (%)	0	43	2	6

a)

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	96	-	-
Expression	-	0	0	4275
FIFO	-	-	-	-
Instance	0	-	36	40
Memory	176	-	0	0
Multiplexer	-	-	-	2804
Register	0	-	2954	128
Total	176	96	2990	7247
Available	280	220	106400	53200
Utilization (%)	62	43	2	13

b)

Name	Slice LUTs (53200)	Block RAM Tile (140)	DSPs (220)	Bonded IOPADs (130)	BUFGCTRL (32)	Slice Registers (106400)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)
design_1_wrapper	6428	93	96	130	1	4448	2457	6187	241
design_1_i (design_1)	6428	93	96	0	1		2457	6187	241
axi_dma_1 (design_...	1287	5	0	0	0		633	1177	110
axi_interconnect_0 (...	1045	0	0	0	0		420	984	61
axi_interconnect_1 (...	537	0	0	0	0		258	476	61
axi_timer_1 (design...	293	0	0	0	0		114	293	0
HLS_accel_1 (desig...	3248	88	96	0	0		1154	3240	8
proc_sys_reset (desi...	18	0	0	0	0		12	17	1
processing_system7...	0	0	0	0	1		0	0	0
xlconcat_1 (design_...	0	0	0	0	0		0	0	0

c)

Figure 4.1

Dans Fig. 4.1a) il s'agit du résultat de synthèse de du calcul matriciel, on dit souvent le calcul proprement dit. Dans Fig. 4.1b) on a ajouté l'interface de communication i.e. le DMA d'où l'apparition des BRAM. Fig. 4.1c) présente le résultat de tous le système i.e. IP + interface de communication + toute les composant pour son exécution et/ou sa validation (processeur(s) A9, timer, interruption, etc.). Fig. 4.1a) et b) sont des estimations et Fig. 4.1c) est le résultat final après bitstream generation (niveau placement et routage).

QUESTION #5 (4 points) Interconnexions AXI et laboratoire no 2

a) (1 pt) Quel est le rôle du DMA dans le laboratoire no 2? Expliquez son fonctionnement dans le contexte de la figure 5.1 (page suivante).

Il réalise l'interface de communication avec le AXI stream à partir du axi_interconnect_0 (MM2S) et jusqu'à son retour (S2MM). Voir cercle en rouge sur la fig. 5.2.

b) (2 pts) À l'aide du schéma de la figure 5.1, indiquez quelle partie de votre design utilise:
c.1) le AXI lite *(initialisation du DMA et des autres périphériques à travers axi_inteconnect_1)*
c.2) le AXI *(entre la DDR et axi_interconnect_0)*
c.3) le AXI stream *(voir réponse a)*
Vous pouvez indiquer directement sur la figure 5.1. Aussi justifiez le choix de l'interconnexion HP.

c) (1 pt) Soit une puce Z-7045 qui contient 4 fois plus de ressources que la puce de votre PYNQ Z-2 (c'est-à-dire la Z-7020). En vous inspirant de la figure 5.1, aurait-on pu l'étendre pour accélérer 160 fois la multiplication matricielle de 42X42 (solution 1.2 du lab2) par rapport à une implémentation logicielle sur ARM A9? Justifiez et dans l'affirmative, indiquez le nombre d'accélérateurs (hls_accel), de DMA et d'interconnexions HP requis avec un schéma général.

*On a sur un 7020 220 DSP et 140 BRAM en double ports. On prend pour un float 42 DSP (en ce laissant au moins 5% de jeu). On en prendrait donc 168 sur la 7045 et on a **900** DSPs. Donc OK!.*

*Pour la BRAM, on a **545**/140 fois = 3.89 plus de BRAM sur la 7045. Toutefois, il faut se rappeler qu'un ARRAY PARTION block factor = 21 suffisait pour la solution 1.2. En fait avec la solution 1.2 on utilise 15% de la BRAM (21/140) par matrice donc environ 30% de la BRAM au total avec 2 matrices i.e. 42 BRAM. Si on multiplie par 4 on a 168 BRAM. Donc OK.*

Notez pour une dimension de 42x42, 5 BRAM sont utilisés pour le transfert avec DMA. Donc on ajouterai 20 BRAM donc total d'environ 188 BRAM, ce qui encore bien en deçà de 545.

Pour le reste de la logique ça devrait amplement aller puisque la majorité des ressources pour faire le calcul utilisent les DSP (et non des LUT).

Finalement, on a 4 canaux HP pour alimenter les 4 coprocesseurs hls_accel. Si on regarde la dernière page de l'Annexe, on aurait 600 Mo/sec plutôt que 800 Mo/sec avec 1 port HP. (donc 25% de dégradation potentiel à cause des communications) Cette perte de performance est due au contrôleur de la DDR qui est alors très sollicité. C'est probablement la que se trouve la question à savoir si on peut faire un 160 fois d'accélération avec 4 calculs

matricielles en parallèle. Au pire s'il y a dégradation des performances à cause des communication, on aura toutefois quand même 160.75% = 120 fois…*

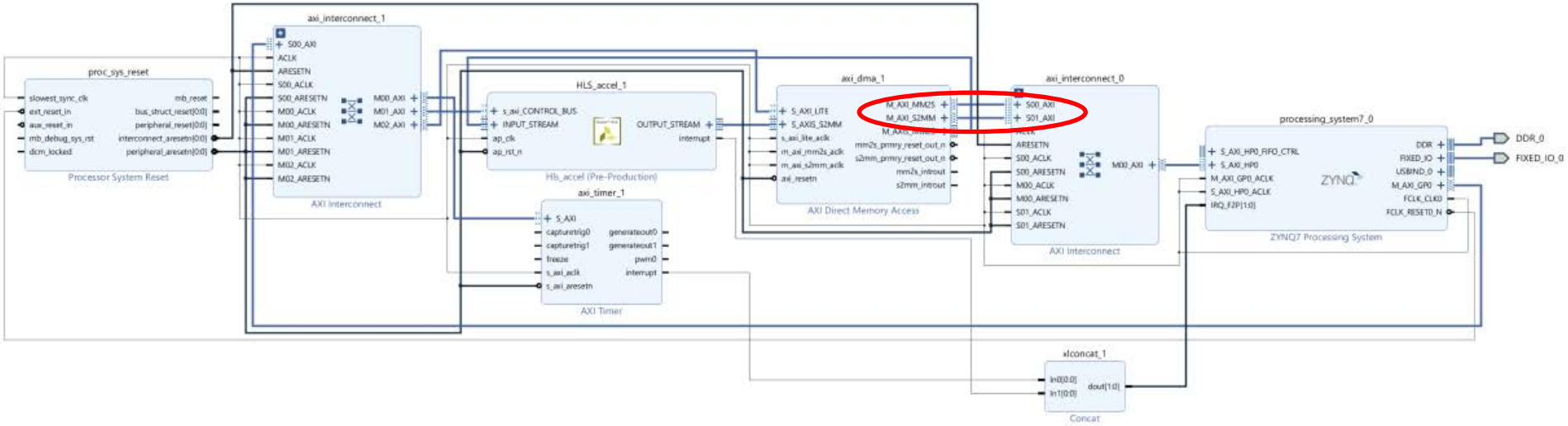

axi_interconnect_1
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M01_ACLK
M01_ARESETN
M02_ACLK
M02_ARESETN
M00_AXI
M01_AXI
M02_AXI
AXI Interconnect
proc_sys_reset
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
HLS_accel_1
s_axi_CONTROL_BUS
INPUT_STREAM
ap_clk
ap_rst_n
OUTPUT_STREAM
interrupt
Hls_accel (Pre-Production)
axi_timer_1
S_AXI
capturetrig0
capturetrig1
freeze
s_axi_aclk
s_axi_aresetn
generateout0
generateout1
pwm0
interrupt
AXI Timer
axi_dma_1
S_AXI_LITE
S_AXIS_S2MM
s_axi_lite_aclk
m_axi_mm2s_aclk
m_axi_s2mm_aclk
axi_resetn
M_AXI_MM2S
M_AXI_S2MM
mm2s_prmry_reset_out_n
s2mm_prmry_reset_out_n
mm2s_introut
s2mm_introut
AXI Direct Memory Access
axi_interconnect_0
S00_AXI
S01_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
S01_ACLK
S01_ARESETN
M00_AXI
AXI Interconnect
processing_system7_0
S_AXI_HP0_FIFO_CTRL
S_AXI_HP0
M_AXI_GP0_ACLK
S_AXI_HP0_ACLK
IRQ_F2P[1:0]
DDR
FIXED_IO
USBIND_0
M_AXI_GP0
FCLK_CLK0
FCLK_RESET0_N
ZYNQ7 Processing System
DDR_0
FIXED_IO_0
xlconcat_1
In0[0:0]
In1[0:0]
dout[1:0]
Concat

Figure 5.1

QUESTION #6 (3 points) Minuterie watchdog

Soit le code sous uC/OS-III de la figure 6.1 pour lequel la minuterie du watchdog est 100 fois plus lente que celle de la minuterie principale et pour lequel StartupTask a une priorité de 5.

a) Décrivez les caractériques des tâches TaskA et TaskB (priorité P_i, période T_i et temps d'exécution maximum C_i).
b) Décrivez ce que réalise exactement l'exécution de ce code.
c) Décrivez les paramètres et le rôle en détail du watchdog. Expliquez également pourquoi dans un cas (FlagGroup1) on fait des Pend sans consume et dans l'autre cas (FlagGroup2) on fait des Pend avec consume ?

```
#define          TaskA_PRIO    15
#define          TaskB_PRIO    14
#define          All_Tasks    0x11
#define          TaskA_flag    0x01
#define          TaskB_flag    0x10

CPU_STK                 StartupTaskStk[UCOS_START_TASK_STACK_SIZE];
OS_TCB                  StartupTaskTCB;

CPU_STK                 TaskAStk[TASK_STK_SIZE];
OS_TCB                  TaskATCB;
CPU_STK                 TaskBStk[TASK_STK_SIZE];
OS_TCB                  TaskBTCB;

OS_TMR                  Scheduler;

OS_FLAG_GRP             FlagGroup1;
OS_FLAG_GRP             FlagGroup2;

int modulo2 = 0;
void                    TaskA(void *data);
void                    TaskB(void *data);
void                    SchedulerFct(OS_TMR  *p_tmr, void *p_arg);
void                    StartupTask (void *p_arg);

int main (void)
{
…
OSTmrCreate(&Scheduler,              /* p_tmr          */
            "Scheduler",             /* p_name         */
             0,                     /* dly            */
             10,                     /* period         */
             OS_OPT_TMR_PERIODIC,    /* opt            */
             SchedulerFct,           /* p_callback     */
             0,                      /* p_callback_arg */
             &err);                  /* p_err          */

// On suppose ici la création des 3 tâches TaskA, TaskB et StartUpTask

OSFlagCreate(&FlagGroup1, "Flag Group", (OS_FLAGS)0, &err);
OSFlagCreate(&FlagGroup2, "Flag Group", (OS_FLAGS)0, &err);

OSStart(&err);
return 0;                                          // Start multitasking
}
```

Figure 6.1 partie 1

```
void  TaskA (void *data)
{
    OS_ERR  err;
    CPU_TS ts;
    OS_TICK actualticks = 0;
    int WAITFORTICKS;

    while(1)
       {
       OSFlagPend(&FlagGroup1, TaskA_flag, 0, OS_OPT_PEND_FLAG_SET_ALL +
OS_OPT_PEND_BLOCKING, &ts, &err);

       do { WAITFORTICKS = (rand() %1000); } while (WAITFORTICKS == 0);
       actualticks = OSTimeGet(&err);
       while(WAITFORTICKS + actualticks > OSTimeGet(&err));

       OSFlagPend(&FlagGroup2, TaskA_flag, 0, OS_OPT_PEND_FLAG_SET_ALL +
OS_OPT_PEND_BLOCKING + OS_OPT_PEND_FLAG_CONSUME, &ts, &err);
       }
}

void  TaskB (void *data)
{
    OS_ERR  err;
    CPU_TS ts;
    OS_TICK actualticks = 0;
    int WAITFORTICKS;

    while(1)
       {
       OSFlagPend(&FlagGroup1, TaskB_flag, 0, OS_OPT_PEND_FLAG_SET_ALL +
OS_OPT_PEND_BLOCKING, &ts, &err);

       do { WAITFORTICKS = (rand() %500); } while (WAITFORTICKS == 0);
       actualticks = OSTimeGet(&err);
       while(WAITFORTICKS + actualticks > OSTimeGet(&err));

       OSFlagPend(&FlagGroup2, TaskB_flag, 0, OS_OPT_PEND_FLAG_SET_ALL +
OS_OPT_PEND_BLOCKING + OS_OPT_PEND_FLAG_CONSUME, &ts, &err);
       }
}

void  SchedulerFct (OS_TMR  *p_tmr, void    *p_arg)
{
  OS_ERR  err;
  modulo2 += 1;

  if (modulo2 % 2 == 1) OSFlagPost(&FlagGroup2, TaskB_flag, OS_OPT_POST_FLAG_SET,
&err);
  else
         OSFlagPost(&FlagGroup2, All_Tasks, OS_OPT_POST_FLAG_SET, &err);
}

void StartupTask (void *p_arg)
{
    // On a mis uniquement les 3 derniàres lignes de StartupTask

    OSTmrStart(&Scheduler, &err);        // Démarrage ici si on est en mode périodique
    OSFlagPost(&FlagGroup1, All_Tasks, OS_OPT_POST_FLAG_SET, &err);
    OSTaskSuspend((OS_TCB *)0,&err);
```

Figure 6.1 partie 2

TaskA est une tâche périodique qui prend au maximum .5 msec

TaskB est une tâche périodique qui prend au maximum 1 msec

On utilise un watchdog périodique pour ordonnancer ces 2 tâches.

Au départ TaskA et TaskB sont bloqués et attendent le signal de StartupTask . Une fois le signal lancé par OSFlagPost(&FlagGroup1, All_Tasks, OS_OPT_POST_FLAG_SET, &err);. TaskA et TaskB débloque (et ne bloqueront plus à cet endroit lors des prochaines itérations du while car pas de consume)..

TaskB passe en premier car plus prioritaire. Quand les 2 ont été exécutées chaque tâche bloque sur OSFlagPend(&FlagGroup2 et on utilise ici un consume qui permettra de rebloquer à la prochaine itération de la boucle.

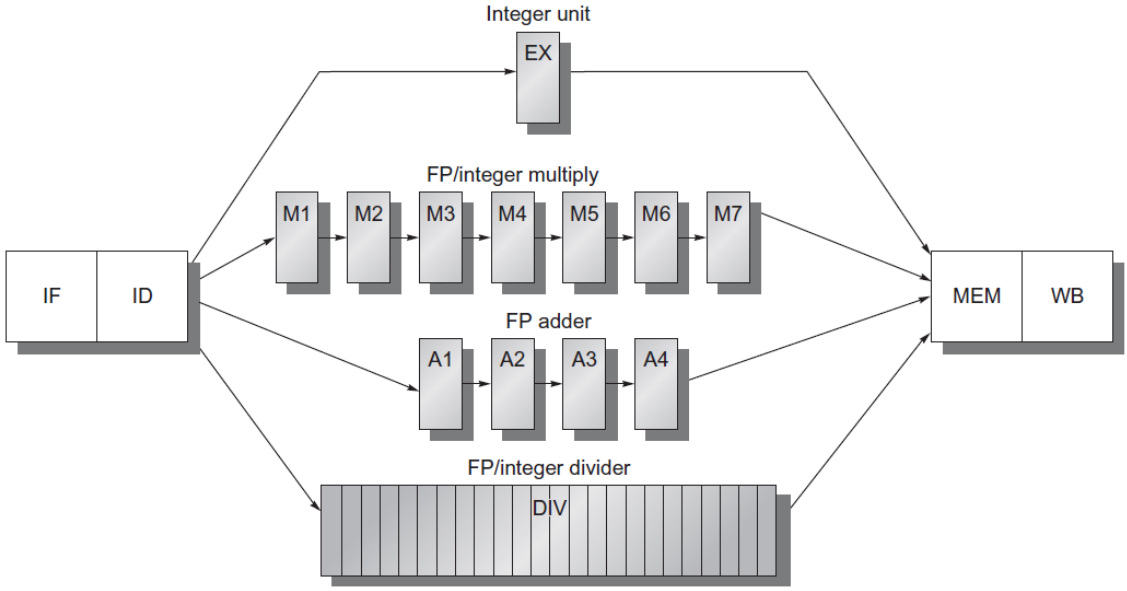
*Le timeout du watchdog est à tous les 10 * (100) = 1000 ticks donc 1 sec et c'est la fonction de callback du watchdog qui va assurer le re-démarrage de TaskA et TaskB, sauf que 1 fois 2 (modulo2) on repart seulement TaskB et 1 fois 2 TaskA et TaskB.*

Notez le temps d'exécution de TaskA et TaskB est inférieur à la période de 1 sec du watchdog donc OK.

Annexe

Détails des instructions pouvant être pipelinées	Nom de l'instruction	Nombre de cycles dans EX	Cycle du pipeline où l'opération termine
LD F1, 0(R1)	**À partir de l'adresse contenue dans R1 auquel on additionne 0, chargement du double mot dans F1**	**1**	**ER (le résultat est dans l'accumulateur après MEM)**
ADDD F1, F1, F3	**Addition de deux doubles mots : F1 –> F1 + F3**	**3**	**ER (le résultat est mis dans l'accumulateur après E3)**
MULTD F1, F1, F5	**Multiplication de deux doubles mots : F1 –> F1 * F5**	**5**	**ER (le résultat est mis dans l'accumulateur après E4**
SUBI R1, R1, #8	**Soustraction entière**	**1**	**ER (le résultat est mis dans l'accumulateur après EX)**
SD 0(R2), F6	**Rangement d'un mot à partir de F6**	**1**	**MEM**
SEQI R5, R4, #val	**si (R4 = #val) alors R5 <= 1 sinon R5 <= 0**	**1**	**ER (le résultat est mis dans l'accumulateur après EX)**
BNEQ R3, etiq	**Branch si non nul**	**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 port de mémoire à l'étape de LI et un port à l'étape de ME du pipeline



Exemple de modèle M4

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.

```
#pragma HLS resource variable=<variable> core=<core>\
latency=<int>
```

Where:

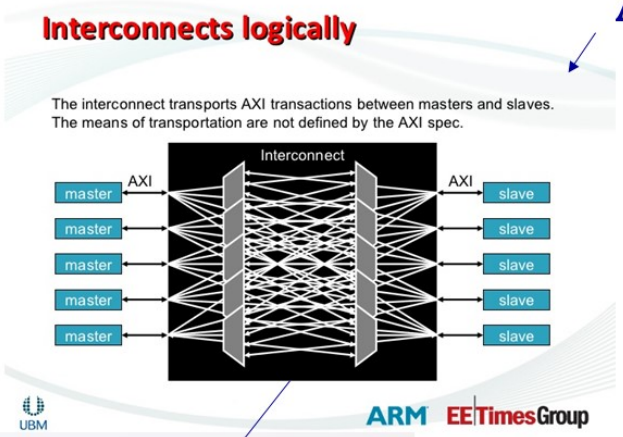
- `variable=<variable>`: A required argument that specifies the array, arithmetic operation, or function argument to assign the `RESOURCE` pragma to.
- `core=<core>`: A required argument that specifies the core, as defined in the technology library.
- `latency=<int>`: Specifies the latency of the core.

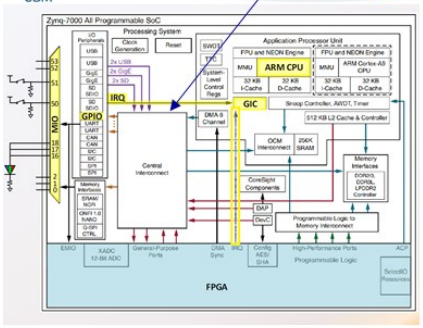
void OSTmrCreate (OS_TMR *p_tmr,
CPU_CHAR *p_name,
OS_TICK dly,
OS_TICK period,
OS_OPT opt,
OS_TMR_CALLBACK_PTR p_callback,
void *p_callback_arg,
OS_ERR *p_err)

Arguments

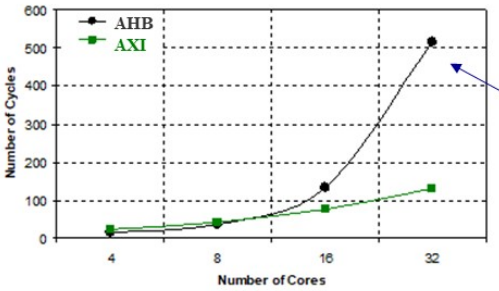
dly
specifies the initial delay (specified in timer tick units) used by the timer (see drawing above). If the timer is configured for ONE-SHOT mode, this is the timeout used. If the timer is configured for PERIODIC mode, this is the timeout to wait before the timer enters periodic mode. The units of this time depends on how often the user will call OSTmrSignal() (see OSTimeTick()). If OSTmrSignal() is called every 1/10 of a second (i.e., OS_CFG_TMR_TASK_RATE_HZ set to 10), dly specifies the number of 1/10 of a second before the delay expires.
period
specifies the period repeated by the timer if configured for PERIODIC mode. You would set the “period” to 0 when using ONE-SHOT mode. The units of time depend on how often OSTmrSignal() is called. If OSTmrSignal() is called every 1/10 of a second (i.e., OS_CFG_TMR_TASK_RATE_HZ set to 10), the period specifies the number of 1/10 of a second before the timer repeats.
opt
is used to specify whether the timer is to be ONE-SHOT or PERIODIC:
OS_OPT_TMR_ONE_SHOT specifies ONE-SHOT mode
OS_OPT_TMR_PERIODIC specifies PERIODIC mode

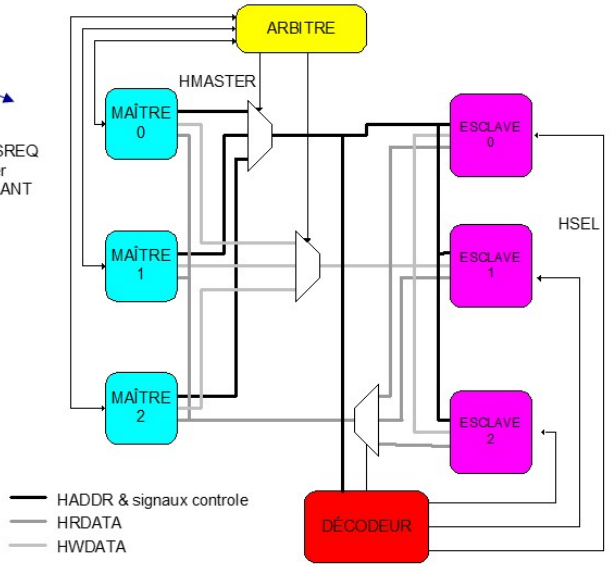
AXI vs AHB





Réf cours no 5





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

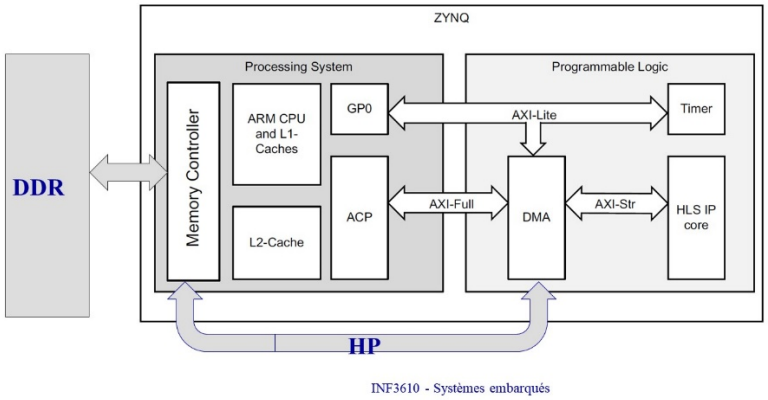


Table 1: **Zynq-7000 and Zynq-7000S SoCs *(Cont'd)***

	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									

Notes:
1. Restrictions apply for CLG225 package. Refer to the UG585, *Zynq-7000 SoC Technical Reference Manual* (TRM) for details.
2. Security is shared by the Processing System and the Programmable Logic.
3. Refer to PG054, *7 Series FPGAs Integrated Block for PCI Express* for PCI Express support in specific devices.