

UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

TRAVAIL DE SESSION

PRÉSENTÉ À

MESSAOUD AHMED OUAMEUR

COMME EXIGENCE PARTIELLE

DU COURS

CONCEPTION EN VLSI

PAR

ANTHONY PINARD

PINA29049809

RAPPORT DE SYNTHÈSE A

26 OCTOBRE 2024

## Introduction

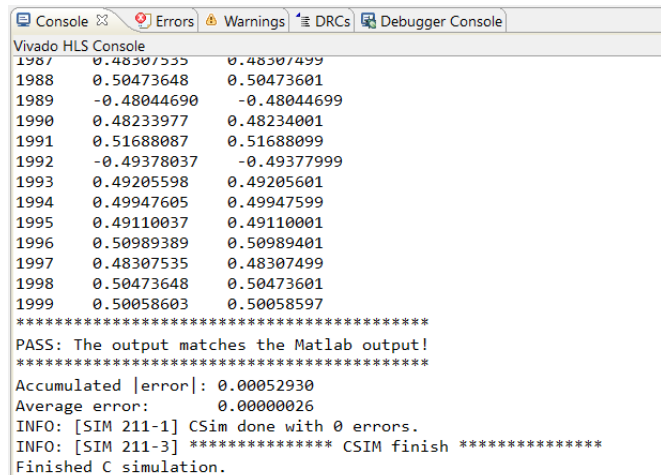
L'algorithme LMS est un filtre adaptatif souvent utilisé en traitement du signal afin d'adresser le problème d'égalisation. Ce phénomène se produit lorsqu'un canal de communication introduit de la distorsion au signal qu'il transmet. L'algorithme tente de retrouver le signal transmis en diminuant l'erreur quadratique moyenne (mean square error) entre le signal désiré et le signal reçu.

Le filtre est initialisé avec ses coefficients à 0. Pour chaque nouveau signal d'entrée, la sortie est calculée en multipliant les coefficients actuels aux signaux actuels et précédents. L'erreur est ensuite calculée entre la sortie et la sortie attendue afin d'ajuster les coefficients. Le filtre LMS est peu complexe, le rendant propice à des applications en temps réel comme en télécommunication. Également, le filtre va souvent converger vers une solution dans le cas où le pas d'adaptation est paramétré correctement. Ce pas d'adaptation doit être choisi délicatement puisque s'il est trop grand, l'algorithme peut diverger alors que s'il est trop faible, la convergence sera longue.

Dans les systèmes de communication, l'égalisation est le processus qui permet de compenser pour la distorsion et de retrouver le signal original. Il existe plusieurs types d'égaliseur et le filtre LMS est souvent utilisé pour des filtres adaptatifs linéaires. Dans ce contexte, le filtre est utilisé pour s'ajuster en fonction du canal et estimer le signal reçu.

## Validation et comparaison avec Matlab

Pour le laboratoire A1, la validation du filtre s'est faite par le test bench dans Vivado HLS. Ce dernier lit les fichiers générés par Matlab et place les variables dans différents tableaux. Ensuite, une boucle for appelle la fonction FIR\_LMS avec les valeurs précédemment lues une par une. Lorsque le nombre d'itérations atteint 19, le pas de convergence est changé de 0,52 à 0,22. Également, à la 200<sup>e</sup> itération, la variable isTraining est changée à false. Ceci permet de répliquer le comportement du filtre sur Matlab.



```

Vivado HLS Console
1987 0.48307535 0.48307499
1988 0.50473648 0.50473601
1989 -0.48044690 -0.48044699
1990 0.48233977 0.48234001
1991 0.51688087 0.51688099
1992 -0.49378037 -0.49377999
1993 0.49205598 0.49205601
1994 0.49947605 0.49947599
1995 0.49110037 0.49110001
1996 0.50989389 0.50989401
1997 0.48307535 0.48307499
1998 0.50473648 0.50473601
1999 0.50058603 0.50058597
*****
PASS: The output matches the Matlab output!
*****
Accumulated |error|: 0.00052930
Average error: 0.00000026
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] ***** CSIM finish *****
Finished C simulation.
```

Une fois que toutes les valeurs sont passées au filtre, le test bench vérifie les erreurs de calculs et affiche les résultats dans la console et dans un fichier de sortie afin de pouvoir comparer les résultats. Un message est ensuite affiché en fonction de l'erreur accumulée. La somme des erreurs absolues entre les sorties Matlab et le filtre est de 0.00052930 pour les 2000 échantillons, ce qui donne une erreur moyenne de 0,00000026.

Pour le laboratoire A2, le banc de test est très similaire à celui du laboratoire A1 à l'exception de l'utilisation des nouveaux types de variables et de la lecture des fichiers .txt en

convertissant implicitement les valeurs en point fixe. Également, au lieu de comparer l'erreur accumulé le banc de test compare individuellement chaque sortie du filtre. On peut remarquer que l'erreur accumulée est bien plus grande dans ce cas puisqu'on utilise des variables à virgule fixe. Cependant, comme l'erreur entre chaque sortie et sortie attendue est moins de 0,01, aucune erreur n'est reportée. Ceci confirme le bon fonctionnement du filtre en virgule fixe.

```

Vivado HLS Console
1989 -0.48925781 -0.48730469
1990 0.50097656 0.50390625
1991 0.48144531 0.48535156
1992 0.50292969 0.50585938
1993 0.50292969 0.50683594
1994 -0.51464844 -0.51171875
1995 0.50195313 0.50585938
1996 0.50195313 0.50683594
1997 -0.49707031 -0.49414063
1998 0.50195313 0.50390625
1999 0.48144531 0.48632813
*****
PASS: The output matches the Matlab output!
Number of errors: 0
*****
Accumulated |error|: 6.81542969
Average error: 0.00340771
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] ***** CSIM finish *****
Finished C simulation.

```

Dans le cadre du laboratoire A3, Les 10 premières valeurs des sorties Matlab ont été écrites dans les tableaux du test bench afin de procéder à la vérification RTL. Comme l'erreur entre les valeurs attendues et les sorties du filtre étaient plus petite que l'épsilon, on conclut que le filtre optimisé et synthétisé est bel et bien fonctionnel dans ce cas aussi.

```

Comparaison des solutions HLS VS Matlab
0.000000 0.000000
0.011697 0.011697
-0.015214 -0.015214
0.016090 0.016090
0.058589 0.058589
-0.098622 -0.098622
0.071088 0.071088
0.127212 0.127212
-0.039312 -0.039312
0.079047 0.079047
*****
Reussite: Les resultats sont acceptables !!!
*****

```

## Analyse des performances et des ressources en virgule flottante vs fixe

Les différences en termes de ressources et de latence entre la synthèse en virgule fixe et en virgule flottante est bien présente. En effet, la latence a été réduite à moins de la moitié, passant de 54 – 96 coups d'horloge à 21 – 31 cycles d'horloge comme le démontre les figures suivantes (virgule flottante à gauche, virgule fixe à droite).

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.42	1.25

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
54	96	54	96	none

Detail

Instance

Loop

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.39	1.25

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
21	31	21	31	none

Detail

Instance

Loop

Également, l'utilisation des ressources a grandement diminuée. On peut voir le nombre de DSP48 passer de 5 pour la virgule flottante à 3 pour la virgule fixe et les Flips-Flops et les Look Up Tables passent respectivement de 688 et 699 pour le filtre à virgule flottante à 123 et 208 pour le filtre à virgule fixe. En parallélisant les boucles dans le laboratoire suivant, on pourrait s'attendre à voir le nombre de ressources utilisées augmenter puisque cette technique d'optimisation a pour conséquence d'augmenter l'utilisation des ressources matérielles.

#### Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	63
FIFO	-	-	-	-
Instance	-	5	355	349
Memory	0	-	128	6
Multiplexer	-	-	-	281
Register	-	-	205	-
<b>Total</b>	<b>0</b>	<b>5</b>	<b>688</b>	<b>699</b>
Available	650	600	202800	101400
Utilization (%)	0	~0	~0	~0

#### Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	3	-	-
Expression	-	-	0	81
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	50	3
Multiplexer	-	-	-	124
Register	-	-	73	-
<b>Total</b>	<b>0</b>	<b>3</b>	<b>123</b>	<b>208</b>
Available	650	600	202800	101400
Utilization (%)	0	~0	~0	~0

Ces gains de performance et la diminution de l'utilisation des ressources est possible puisque les opérations en point fixes sont plus efficaces. En effet, on n'a pas besoin de la logique complexe permettant de traiter les opérations en virgule flottante ce qui rend le design plus compacte et rapide. Les opérations de bases tel que les additions, les multiplications et les soustractions nécessitent donc un matériel plus simple puisqu'il n'y a pas d'exposant à décaler ni de mantisse à aligner.

### Impacts de l'optimisation sur la vitesse et les ressources

Le code en C a été optimisé à l'aide des directives HLS de trois façons : optimisation des boucles seulement, optimisation des fonctions seulement et optimisation complète. Voici l'estimation de performance et l'utilisation des ressources du filtre sans optimisation :

#### Performance Estimates

##### Timing (ns)

###### Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.35	1.25

##### Latency (clock cycles)

###### Summary

Latency		Interval		Type
min	max	min	max	
11	32	11	32	none

#### Utilization Estimates

##### Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	19	0	487
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	256	12
Multiplexer	-	-	-	158
Register	-	-	379	-
<b>Total</b>	<b>0</b>	<b>19</b>	<b>635</b>	<b>657</b>
Available	650	600	202800	101400
Utilization (%)	0	3	~0	~0

### Optimisation des boucles

La solution afin d'obtenir les meilleurs résultats en termes de latence et d'intervalle a été d'ajouter les directives Pipeline et Unroll au niveau des boucles Shift\_Accum\_Loop et Update\_Weights\_Loop. Ces directives ont permis d'obtenir une latence et un intervalle minimal de 2 cycles et maximale de 4 cycles.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.19	1.25

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
2	4	2	4	

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	68	0	1238
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	25
Register	-	-	1336	-
Total	0	68	1336	1263
Available	650	600	202800	101400
Utilization (%)	0	11	~0	1

### Optimisation des fonctions

Il a été également possible d'obtenir des résultats similaires en optimisant les fonctions au lieu des boucles. Le résultat de la synthèse comporte toute fois des différences au niveau des ressources utilisées. Les directives de Pipeline ont été utilisées sur les fonctions FIR et LMS et les tableaux utilisés dans ces mêmes fonctions ont été partitionné. Cette optimisation a permis d'obtenir les mêmes performances en termes de latence et d'intervalle mais, à un cout plus élevé en matière de ressources. Le nombre de Flip-Flops et de Look up Tables utilisés est plus important dans ce scénario d'optimisation. Il est important de souligner que la directive Dataflow a aussi été utilisée mais, celle-ci offrait de moins bonne performance en termes d'intervalle et de latence que la directive Pipeline.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.19	1.25

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
2	4	2	4	
none				

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	57
FIFO	-	-	-	-
Instance	-	68	1335	1213
Memory	-	-	-	-
Multiplexer	-	-	-	25
Register	-	-	391	-
Total	0	68	1726	1295
Available	650	600	202800	101400
Utilization (%)	0	11	~0	1

### Optimisation complète

Lors de l'optimisation complète, les résultats de la synthèse sont identiques aux résultats précédents de l'optimisation des fonctions seulement. Encore dans ce cas-ci, l'optimisation a été faite de sorte à obtenir les meilleurs résultats en matière de performance. Les instructions utilisées dans la fonction principale FIR\_LMS sont simplement que deux directives d'array partition pour les tableaux input et updated\_c.

Ensuite, les sous fonctions FIR et LMS ont été pipelinées, leurs tableaux c et shift\_reg ont été partitionnés et leur boucle interne ont été également pipelinées et dépliées.

Performance Estimates					Utilization Estimates				
⊟ Timing (ns)					⊟ Summary				
⊟ Summary					Name	BRAM_18K	DSP48E	FF	LUT
Clock	Target	Estimated	Uncertainty		DSP	-	-	-	-
ap_clk	10.00	8.19	1.25		Expression	-	-	0	57
⊟ Latency (clock cycles)					FIFO	-	-	-	-
⊟ Summary					Instance	-	68	1335	1213
					Memory	-	-	-	-
					Multiplexer	-	-	-	25
					Register	-	-	391	-
Latency		Interval			Total	0	68	1726	1295
min	max	min	max	Type	Available	650	600	202800	101400
2	4	2	4	none	Utilization (%)	0	11	~0	1

### Meilleure optimisation

Parmi les trois choix d'optimisation présenté, la meilleure solution à choisir serait celle de l'optimisation des boucles seulement. Ce choix est basé sur les performances du filtre ainsi que son utilisation des ressources. Puisque la latence des trois optimisations est équivalente, l'utilisation moindre des ressources de ce profil d'optimisation le rend le plus avantageux.

## Réponses à toutes les questions des Laboratoires

### Laboratoire A1

#### Quelle est la différence entre la synthèse et la simulation ?

La simulation permet de vérifier que le composant se comporte tel qu'on s'attend sans synthétiser le code, elle sert à valider la logique du code. La synthèse, elle, permet de transformer le code en C vers un code de type description matériel.

#### Quelle peut être l'utilité d'avoir plusieurs solutions dans un même projet ?

Avoir plusieurs solutions dans un même projet permet de garder une trace des optimisations faites. On peut copier les directives d'une solution afin de continuer le processus d'optimisation tout en gardant les itérations précédentes afin de les comparer entre eux. Ceci permet d'explorer plusieurs pistes d'optimisation tout en sauvegardant tous les états intermédiaires de la solution en cours.

#### Expliquez et discutez les résultats de la synthèse.

L'horloge estimée du système dans le pire scénario est de 8,42 ns, ce qui respecte le délais requis de 10 ns moins l'incertitude de 1,25 ns. La

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- shift_loop	8	8	2	-	-	4	no
- compute_loop	40	40	8	-	-	5	no
- weights_loop	40	40	8	-	-	5	no

latence du composant est minimalement de 54 coups d'horloge et maximalement de 96 coups d'horloge. Cet écart d'environ 40 coups d'horloge est la différence entre quand le filtre est en entrainement et quand il est en opération normale.

Dans le résumé des ressources estimées, on peut voir que le design utilise 5 DSP48E. Cette figure peut sembler importante pour un filtre FIR-LMS. Le design a besoin d'autant de DSP48E parce que les données sont des float qui utilisent 32 bits. Dans le prochain laboratoire, en utilisant des nombres à point fixes, il sera possible de diminuer le nombre de DSP48E.

#### Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	63
FIFO	-	-	-	-
Instance	-	5	355	349
Memory	0	-	128	6
Multiplexer	-	-	-	281
Register	-	-	205	-
<b>Total</b>	<b>0</b>	<b>5</b>	<b>688</b>	<b>699</b>
Available	650	600	202800	101400
Utilization (%)	0	~0	~0	~0

Une fois la synthèse effectuée, quelle est la différence entre «Latency» et «Interval» ?

La Latency est le nombre de coups d'horloge que le composant prend avant de sortir son résultat. L'Interval est le nombre de coups d'horloge avant de pouvoir prendre de nouvelles données en entrée.

## Laboratoire A2

### Différences en termes de ressources et de latence

Les différences en termes de ressources et de latence entre la synthèse en virgule fixe et en virgule flottante est bien présente. En effet, la latence a été réduite à moins de la moitié, passant de 54 – 96 coups d'horloge à 21 – 31 coups d'horloge comme le démontre les figures suivantes (virgule flottante à gauche, virgule fixe à droite).

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.42	1.25

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
54	96	54	96	none

Detail

+ Instance

+ Loop

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.39	1.25

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
21	31	21	31	none

Detail

+ Instance

+ Loop

Également, l'utilisation des ressources a grandement diminuée. On peut voir le nombre de DSP48 passer de 5 pour la virgule flottante à 3 pour la virgule fixe et les FF et les LUT passent respectivement de 688 et 699 pour le filtre a virgule flottante à 123 et 208 pour le filtre a virgule fixe. En parallélisant les boucles dans le prochain laboratoire, on pourrait s'attendre a voir le nombre de ressources utilisées augmenté puisque cette technique d'optimisation a pour conséquence d'augmenter l'utilisation des ressources matérielles.

#### Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	63
FIFO	-	-	-	-
Instance	-	5	355	349
Memory	0	-	128	6
Multiplexer	-	-	-	281
Register	-	-	205	-
<b>Total</b>	<b>0</b>	<b>5</b>	<b>688</b>	<b>699</b>
Available	650	600	202800	101400
Utilization (%)	0	~0	~0	~0

#### Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	3	-	-
Expression	-	-	0	81
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	50	3
Multiplexer	-	-	-	124
Register	-	-	73	-
<b>Total</b>	<b>0</b>	<b>3</b>	<b>123</b>	<b>208</b>
Available	650	600	202800	101400
Utilization (%)	0	~0	~0	~0

### Différence entre #pragma HLS DATAFLOW et #pragma HLS PIPELINE

La directive #pragma HLS PIPELINE a pour but d'optimiser les boucles en parallélisant différentes étapes de la même boucle. Cette directive sert à réduire le nombre de coups d'horloge entre le début et la fin de chaque itération ce qui augmente le throughput. La directive #pragma HLS DATAFLOW a pour but de paralléliser les fonctions ou les blocs de code de sorte à ne pas attendre la fin d'une fonction pour appeler la suivante. Cette directive agit un peu comme un pipeline mais au niveau supérieur entre les fonctions et les boucles.

### Différence en termes de ressources utilisées et en termes de latences entre une directive qui pipeline la boucle interne seulement et une directive qui pipeline la boucle externe seulement ?

Lorsqu'on pipeline la boucle externe, la latence du système va grandement diminuer parce qu'on n'attend pas la fin de la dernière boucle externe avant de lancer la suivante. Cela dit, l'utilisation des ressources va grandement augmenter puisqu'on parallélise à la fois la boucle externe et la boucle interne qui y est contenue.

Lorsqu'on pipeline la boucle interne seulement, la latence du système va également diminuer mais, l'impact est moins important que dans le cas précédent puisque la boucle externe reste non optimisée. Cependant, comme seulement la boucle interne est parallélisée, l'utilisation des ressources est moins grande.

Le choix de quelle boucle on veut optimiser sera fait en fonction de la latence désirée du système et des ressources disponibles.



## Laboratoire A3

Justifiez vos choix de directives pour chacune de vos optimisations

Filtre non optimisé :

Voici le Schedule Viewer du filtre lorsqu'il n'est pas optimisé :

Current Module : fir_LMS									
	Operation\Control Step	C0	C1	C2	C3	C4	C5	C6	C7
1	nbTrain_read(read)								
2	ref_V_read(read)								
3	mhu_V_read(read)								
4	data_V_1(read)								
5-15	Shift_Accum_Loop								
16	node_51(write)								
17	updated_c_V_load_1...								
18	node_50(write)								
19	p_val2_3(*)								
20	p_val2_4(+)								
21	train_count_load(r...								
22	tmp_1(icmp)								
23	e_V(-)								
24	r_V(*)								
25...	Update_Weights_Loop								
35	tmp_2(+)								
36	node_94(write)								
37	node_97(write)								

Optimisation des boucles :

On peut remarquer que la boucle Update\_Weights\_Loop prend plusieurs cycles d'horloge à parcourir. Afin d'optimiser les boucles, une directive de pipeline et de dépliage permettraient de réduire la latence et ainsi améliorer les performances du filtre. C'est pourquoi les directives Unroll et Pipeline ont été choisies. Elles parallélisent la boucle et permettent le démarrage des itérations plus rapidement. Voici le Schedule Viewer une fois les boucles du filtre optimisées :

Current Module : fir LMS						
	Operation\Control Step	C0	C1	C2	C3	C4
1	updated_c_V_1_load(read)					
2	updated_c_V_2_load...					
3	updated_c_V_3_load...					
4	updated_c_V_4_load...					
5	input_4_V(read)					
6	p_Val2_1_i(*)					
7	input_3_V(read)					
8	node_41(write)					
9	p_Val2_1_1_i(*)					
10	p_Val2_2_1_i(+)					
11	input_2_V(read)					
12	node_49(write)					
13	p_Val2_1_2_i(*)					
14	input_1_V(read)					
15	node_57(write)					
16	p_Val2_1_3_i(*)					
17	nbTrain_read(read)					
18	ref_V_read(read)					
19	mhu_V_read(read)					
20	input_0_V(read)					
21	updated_c_V_0_load...					
22	p_Val2_2_2_i(+)					
23	p_Val2_2_3_i(+)					
24	node_64(write)					
25	p_Val2_s(*)					
26	p_Val2_1(+)					
27	train_count_load(r...					
28	tmp_1(icmp)					
29	tmp_2(+)					
30	node_127(write)					
31	e_V(-)					
32	r_V(*)					
33	p_Val2_9_i(*)					
34	p_Val2_9_1_i(*)					
35	p_Val2_9_2_i(*)					
36	p_Val2_9_3_i(*)					
37	p_Val2_9_4_i(*)					
38	c_V_0_load(read)					
39	p_Val2_10_i(+)					
40	node_88(write)					
Performance		Resource				

### Optimisation des fonctions :

Comme les boucles se retrouvent dans les fonctions, en optimisant celles-ci on peut réduire le temps d'exécution des boucles. C'est pourquoi la directive `array partition` a été utilisée afin d'optimiser l'accès aux variables des fonctions et qu'ensuite les fonctions ont eux-mêmes été pipelinées. Avec ces directives les fonctions peuvent parcourir les boucles plus rapidement et efficacement. Voici le Schedule Viewer une fois les fonctions du filtre optimisées :

Current Module : fir LMS						
	Operation\Control Step	C0	C1	C2	C3	C4
1	x_V_read(read)					
2	updated_c_V_0_load...					
3	updated_c_V_1_load...					
4	updated_c_V_2_load...					
5	updated_c_V_3_load...					
6	updated_c_V_4_load...					
7	fir(function)					
8	nbTrain_read(read)					
9	ref_V_read(read)					
10	mhu_V_read(read)					
11	train_count_load(r...					
12	tmp_1(icmp)					
13	tmp_2(+)					
14	node_59(write)					
15	LMS(function)					
16	node_49(write)					
17	node_51(write)					
18	node_53(write)					
19	node_55(write)					
20	node_57(write)					
21	node_62(write)					

Optimisation complète :

L'optimisation complète possède déjà toutes les directives des deux optimisations précédentes en plus d'ajouter la directive d'array partition aux tableaux input et updated\_c. Comme ces tableaux sont passés aux fonctions FIR et LMS, et que dans ces fonctions les tableaux sont déjà optimisés, aucun effet n'est observable. Une directive de Dataflow a été ajoutée au niveau de la fonction principale mais, a été retirée puisque la latence était augmentée. Le Schedule Viewer après l'optimisation complète est identique au précédent, le voici:

Current Module : fir LMS						
	Operation\Control Step	C0	C1	C2	C3	C4
1	x_V_read(read)					
2	updated_c_V_0_load...					
3	updated_c_V_1_load...					
4	updated_c_V_2_load...					
5	updated_c_V_3_load...					
6	updated_c_V_4_load...					
7	fir(function)					
8	nbTrain_read(read)					
9	ref_V_read(read)					
10	mhu_V_read(read)					
11	train_count_load(r...					
12	tmp_1(icmp)					
13	tmp_2(+)					
14	node_59(write)					
15	LMS(function)					
16	node_49(write)					
17	node_51(write)					
18	node_53(write)					
19	node_55(write)					
20	node_57(write)					
21	node_62(write)					

Analysez et comparez la différence en termes de latences, de ressources utilisées et de période minimale pour chacun des types d'optimisations demandés

Les trois optimisations sont équivalentes en termes de latence et de période minimale d'horloge avec 2 à 4 cycles de latence et 8,19ns de période estimée. Les synthèses n'utilisent pas de BRAM et l'utilisation des DSP est pareil dans les trois optimisations. Là où l'optimisation des boucles est plus avantageuse est en termes de ressources. En effet, celle-ci utilise moins de Flip-Flops et de Look up Tables que l'optimisation des fonctions ainsi que l'optimisation complète. L'optimisation des boucles serait alors le meilleur choix des trois.

Discutez l'effet de la directive *inline* sur les ressources utilisées

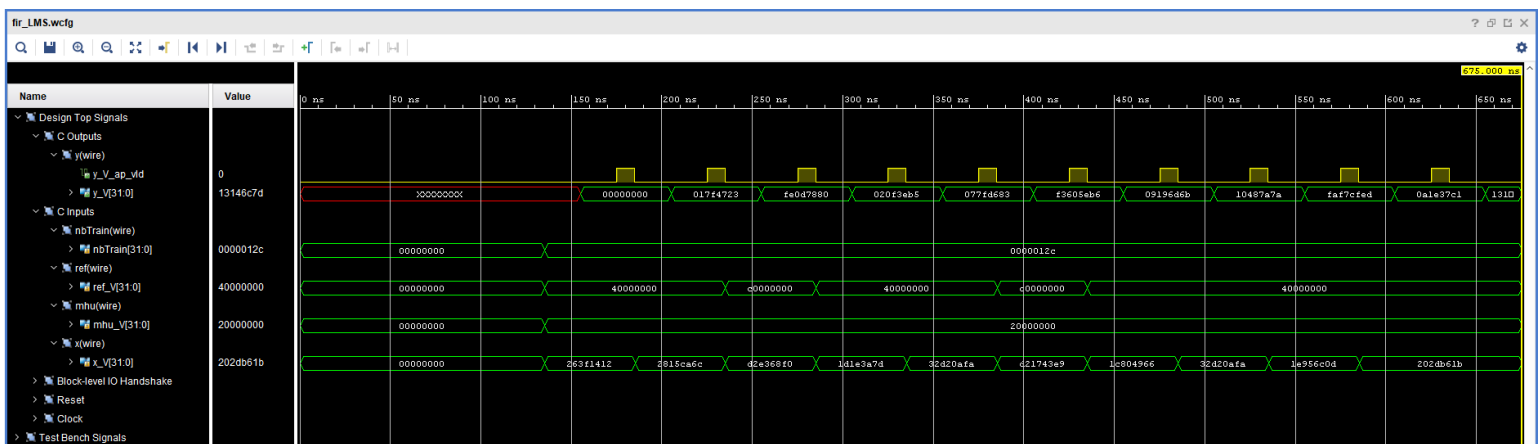
La directive Inline permet de diminuer la latence au cout de quelques ressources supplémentaires. La latence passe de 12 – 34 sans la directive à 11 – 32 avec la directive. Si la fonction est appelée à plusieurs endroits dans le code, la directive Inline permettrait de réduire la latence du système en répliquant le code de la fonction à l'endroit où elle est appelée.

Dans quelles circonstances une directive de type *inline off* pourrait être utile

La directive Inline off pourrait être utile dans le contexte où les ressources sont précieuses et qu'une fonction est appelée à plusieurs reprise. La directive permettrait de limiter l'utilisation des ressources en s'assurant que l'outil de synthèse ne l'utilise pas automatiquement. La logique de la fonction serait alors présente dans le matériel sous forme d'une seule entité physique.

Montrez un exemple concret du bon fonctionnement de votre filtre

On peut observer le signal nbTrain prendre une valeur de 0x12C, ce qui représente 300. En retournant dans le code du banc de test on peut valider cette valeur et confirmer que le filtre prend bel et bien des valeurs en entrées. Également, on peut observer la dixième sortie y avec la valeur 0x0A1E37C1 qui se traduit par 0.0001010000111100011011111000001 sur 32 bits avec un bit d'entier. En décimal la valeur est de 0.0790471 ce qui correspond à notre dixième sortie de référence. On peut alors en déduire le bon fonctionnement du filtre puisqu'à ce point une erreur dans les calculs serait remarquable.



## Conclusion

Dans l'optique de synthétiser le code écrit dans les laboratoires A1 et A2, il aurait été important de prendre en considération l'étape d'optimisation. En effet, le design actuel des laboratoires A1 et A2 ne permettait pas de bien synthétiser le filtre. L'estimation de la période de l'horloge une fois le filtre optimisé est trop longue même avec des variables plus courtes, soit de 10,49ns. Ceci peut être dû à un chemin critique trop long, ralentissant les opérations. Également, la latence du filtre est de 3 à 4 cycles, soit un cycle minimal de plus que le code fourni optimisé. Ces écarts peuvent être expliqués par le manque de variables tampons dans les boucles qui permettent de briser la dépendance entre les itérations. Également, on peut optimiser la boucle de décalage en y effectuant les opérations en même temps au lieu de les faire dans une seconde boucle.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	10.49	1.25

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
3	4	3	4	none