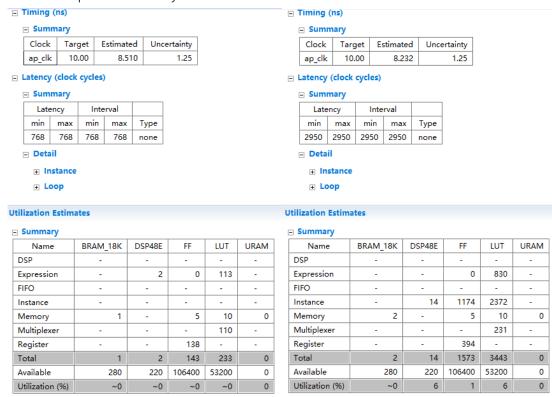
CSE237C Project1: FIR filter design

Yiming Ren

y5ren@ucsd.edu

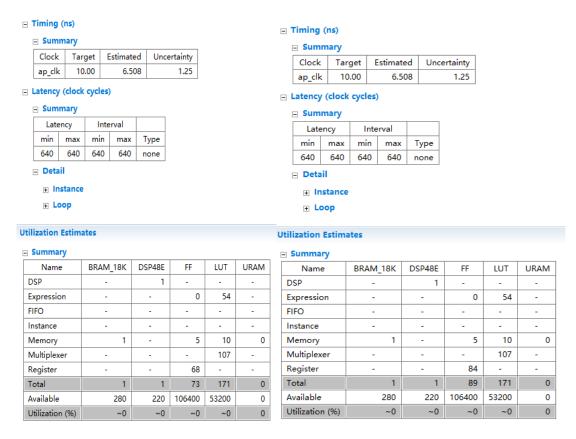
Question 1:

In the design for Q1 I only modify the variable type of "data" from data_t to various types, from char up to float. The synthesis results are shown below.



Int (4 bytes) Throughput: 0.15MHz double (8 bytes)
Throughput: 0.04MHz

From int to double, both bitwidth of variable and the latency are increasing exponentially, but the latency obviously grows more rapidly, and it seems like using the hardware recklessly. The first possible reason is the increasing propagation time caused by increasing number of used BRAM. If the BRAM memories are cascaded, it spends more time to access the stored memory and transmit the data. Also, since type double is 64 bits wide, BRAM has to combine its two 36 bits I/O ports to support 64 bits data transmission. This could double the propagation time.



Char (1 byte)
Throughput: 0.24MHz

short (2 bytes)
Throughput: 0.24MHz

Compared to the difference between int and byte, it changes mildly from char to short. Only # of FFs changes with others remain the same. The throughput reaches the vertex with still the highest accuracy (No mismatch with Golden results). For both performance and efficiency, the design with char type could be adopted for optimization.

Question 2:

In design for Q2 I add 2 instructions '#pragma HLS PIPELINE II=x' in the 2 loops, and increment 'x' by 1 each time to increase initiation interval. The synthesis results of 'II= 1-3' and baseline are below.

■ Latency (clock cycles)

Summary

Late	Latency		Interval	
min	max	min max		Type
768	768	768	768	none

Baseline Throughput = 0.15MHz

□ Latency (clock cycles)

Summary

La	Latency		Interval		
min		max	min max		Туре
261		261	261	261	none

II=1
Throughput = 0.45MHz

□ Latency (clock cycles)

Summary

Late	Latency		Interval	
min	max	min max		Туре
515	515	515	515	none

II = 2
Throughput = 0.23MHz

■ Latency (clock cycles)

Summary

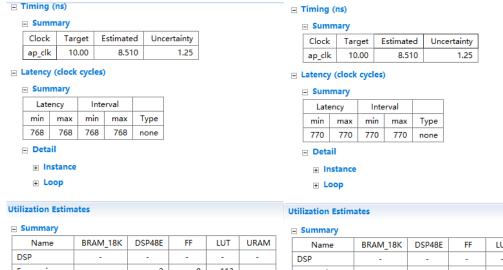
Late	ency	Interval		
min	max	min max		Туре
642	642	642	642	none

II = 3
Throughput = 0.18MHz

The resources usage is not presented since the difference is ignorable except that 1 more BRAM is used. It makes sense since pipeline requests more utilization of processor. I found out that the performance is decaying as II increases. And I can assume that when II reaches 4, the performance will be less than baseline. Since the loop contains 3 steps, which are the shifting, multiplying and accumulating, pipeline with 4 time intervals would be nonsense to optimization.

Question 3:

My baseline code doesn't include any conditional statements, so I rewrite a design with conditional statements. I did this by inserting the statement of assigning input 'x' to 'data[0]' into the shifting loop. In the loop, if 'I' decreases to 0, assign 'x' to 'data[0]'.



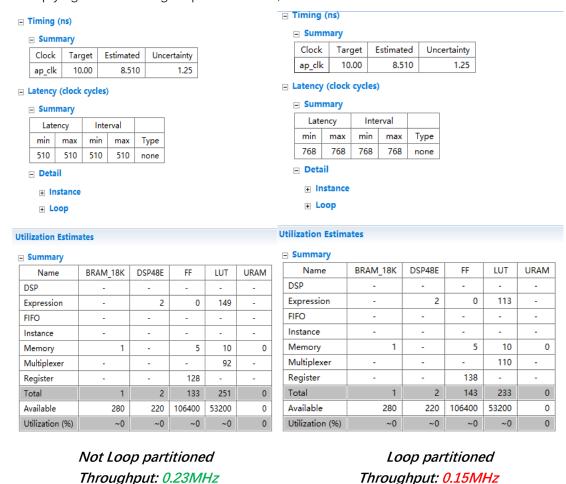
Summary					
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	0	113	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	1	-	5	10	0
Multiplexer	-	-	-	110	-
Register	-	-	138	-	-
Total	1	2	143	233	0
Available	280	220	106400	53200	0
Utilization (%)	~0	~0	~0	~0	0

LUT URAM Expression 113 FIFO Instance 10 Memory 0 Multiplexer 125 Register Total 170 248 0 Available 280 220 106400 53200 0 ~0 0 Utilization (%)

No conditional branch Throughput: 0.1530MHz with conditional branch Throughput: 0.1526MHz From the synthesis result we can see that conditional statements do have a little bit impact on the performance, but the influence is ignorable. However, conditional statements influence a lot on resources usage. My guess is that since conditional statements use more registers in order to compare the result and need 1 more branch instruction than basic code, more flipflops are used as registers and synchronous logic. And, consequently, more LUTs are used to access the memory and registers.

Question 4:

My baseline design is loop partitioned originally, so I combine the shifting and multiplying+accumulating loops. Meanwhile, I tried to avoid conditional statements.



I get a different result with the speculation in the project instruction. I think this should be correct since I'm now dealing with a 128 FIR filter. For this amount of tap the compiler prefer not to unroll the loop. So in unpartitioned mode, the processor will run 2 loops while it only contains 1 loop with loop partition. And 2 loops means double loop branches. It also needs additional registers for the branch. However, since the running time is still linear, the performance does not change significantly.

Question 5:

I insert a statement '#pragma HLS array_partition variable=data xxx factor=2' to separate the 'data' array to 2 parts, and store them in 2 'xxx' type partitioned BRAM. The 'xxx' varies between 'block' and 'cyclic', used to compare with 'complete' type.

Name	BRAM_18K	DSP48E	FF	LUT	URAM	
DSP	-	-	-	-	-	D
Expression	-	2	0	177	-	E
FIFO	-	-	-	-	-	FI
Instance	-	-	-	-	-	In
Memory	2	-	5	10	0	N
Multiplexer	-	-	-	131	-	N
Register	-	-	139	-	-	R
Total	2	2	144	318	0	T
Available	280	220	106400	53200	0	A
Utilization (%)	~0	~0	~0	~0	0	U

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	0	200	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	2	-	5	10	0
Multiplexer	-	-	-	131	-
Register	-	-	147	-	-
Total	2	2	152	341	0
Available	280	220	106400	53200	0
Utilization (%)	~0	~0	~0	~0	0

Cyclic

•	Block
•	Dioci

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	0	115	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	0	-	5	10	-
Multiplexer	-	-	-	1158	-
Register	-	-	8363	-	-
Total	0	2	8368	1283	0
Available	280	220	106400	53200	0
Utilization (%)	0	~0	7	2	0

Late	Latency		Interval	
min	max	min max		Туре
895	895	895	895	none

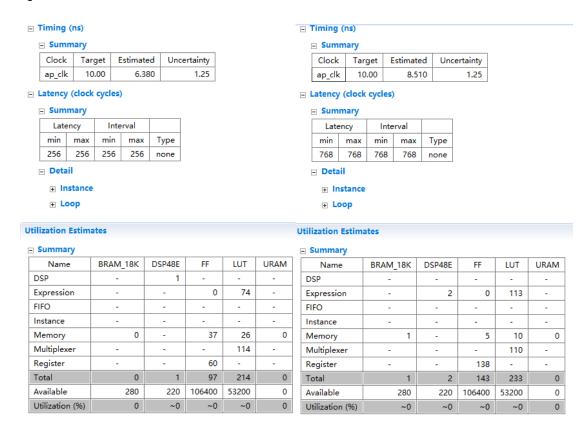
Complete resource usage and latency
Throughput: 0.13MHz

The performances of those 'Cyclic' and 'block' types of memory partitioning are not shown since they do not influence performance at all. However, 'complete' memory partitioning is an awful choice for optimization. It not only increases the latency but also brings overwhelming works for the processor. This is because the 'complete' array partition divides an array to the least units, and store these units in registers (FFs). So it has nothing to do with the memory, but it largely limit the performance of processor. Thus, the latency is obviously growing. But even though more than 8k FFs are used, there are still 106k FFs available, so the

processor could still stay efficient under such a pressure.

Though the result of 'Cyclic' and 'Block' partition does not show the significantly change of performance, but I assure you 'Cyclic' is better for FIR128. When the processor executes the statements in parallel, it will continuously read and write to a memory. 'Cyclic' could avoid a lot of read and write confliction. For example, if array 'ABCD' is partitioned into 2 parts, 'cyclic' would store 'AC' in RAM1 and 'BD' in RAM2, while 'block' would store 'AB' in RAM1 and 'CD' in RAM2. When the processor accesses the data in sequence, it could access the 2 RAMS alternatively when using 'cyclic', and thus, it doesn't have to wait the occupied port to be freed. So it could do read and write consecutively in pipeline. By contrast, if the processor is doing operations on B, it needs to wait the operations on A to be finished, since those operations are occupying RAM1 ports.

Question 6:



Optimized Basic
Throughput: 0.61MHz Throughput: 0.15MHz

In all, the optimization could be accomplished by combining all those better methods. I use 'char' for variables type to reduce latency and resources usage, use 'II=1' pipeline to further reduce latency, avoid conditional statements for less registers usage, combine loops to reduce execution time, and apply 'cyclic' array partition to help releasing BRAM's stress. Though this combined architecture uses more FFs and LUTs than simply changing variable type to 'char', it reaches a performance that the latter is far from. This architecture is more balanced compare to other optimizations.

Theoretically, a design that could outputs every cycle could be achieved. Only if the machine is huge enough to use millions of registers parallelly. In this way, it doesn't need to access the memory and should be quick enough to get result in 1 cycle. In assembly code view, that is unrolling the loop and using enough registers at the same time for all the operations.