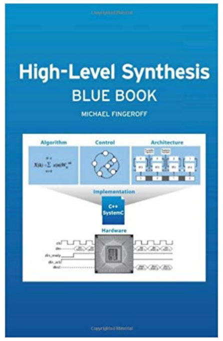
Parallel and Reconfigurable VLSI Computing (10)

HLS Optimizations

Hiroki Nakahara

Tokyo Institute of Technology

References:
[1] Micheal Fingeroff, "High-Level Synthesis Blue Book," Xlibris, 2010.
[2] Ryan Kastner, Janarbek Matai, Stephen Neuendorffer, "Parallel Programming for FPGAs," arXiv:1805.03648, 2018. https://arxiv.org/abs/1805.03648



Outline

- HLS Optimizations though FIR Design
- Code reconstruction
  - Useful cording guideline for debugging
  - Performance analysis on LLVM-IR
  - Area-time trade-off
  - Code hoisting
  - Loop fission
  - Loop unrolling
  - Array partition
  - Loop pipelining
  - Bitwidth optimization

HLS Optimizations though FIR Design

FIR Filter Background

- x_n: N sampling signals and y_n: Output signal, then

$$y_n = \sum_{k=0}^{N-1} h_k x_{n-k}$$

, where a filter coefficient h_n is given by

$$h_n = \frac{\rho_n}{2\pi} \int_0^{2\pi} d\tilde{\omega} e^{i\tilde{\omega}(n-\tilde{\tau})} H_0(\tilde{\omega}).$$

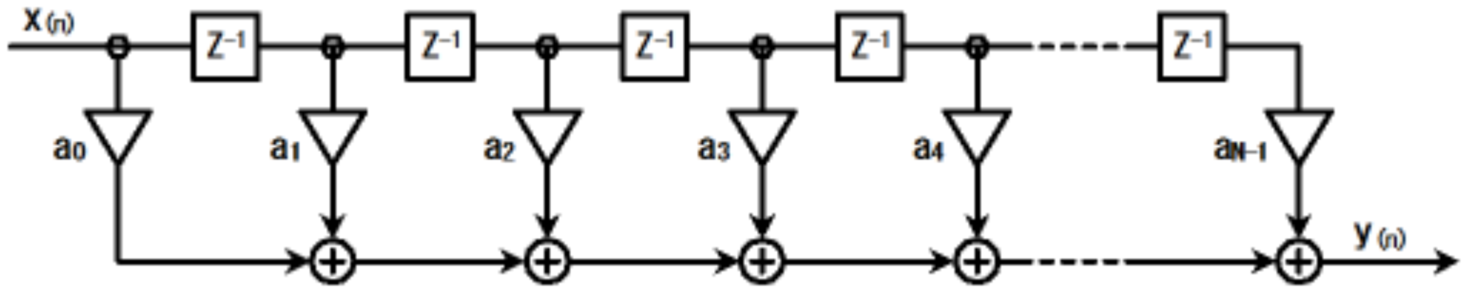
, where $\widetilde{\omega}=2\pi\omega/\omega_s$ denotes normalized freq., ω_s denotes sampling freq., $H_0(\widetilde{\omega}) \in R$ denotes frequency characteristic, p_n denotes window function, and $\tilde{\Gamma}=(N-1)/2$.

Cont'd

- Deference equation for a FIR filter:

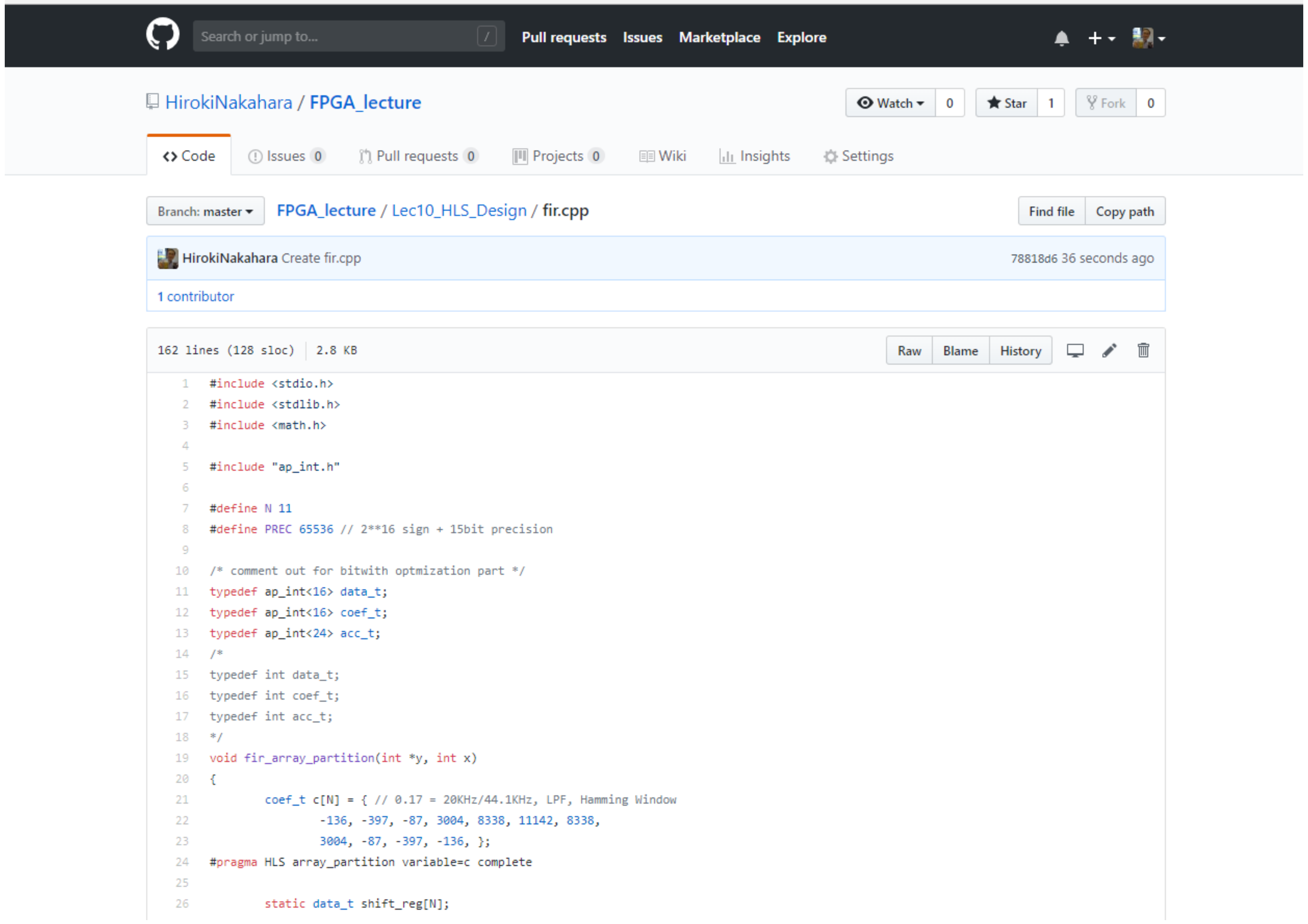
$$y[n]=a_0x[n]+a_1x[n-1]+a_2x[n-2]+a_3x[n-3]+a_4x[n-4]+\cdots+a_{N-1}x[n-(N-1)]$$

- Diagram for a FIR filter:



C++ Behavior for a FIR Filter

https://github.com/HirokiNakahara/FPGA_lecture/tree/master/Lec10_HLS_Design/fir.cpp



FPGA_lecture / Lec10_HLS_Design / fir.cpp

HirokiNakahara Create fir.cpp 78818d6 36 seconds ago

1 contributor

162 lines (128 sloc) | 2.8 KB

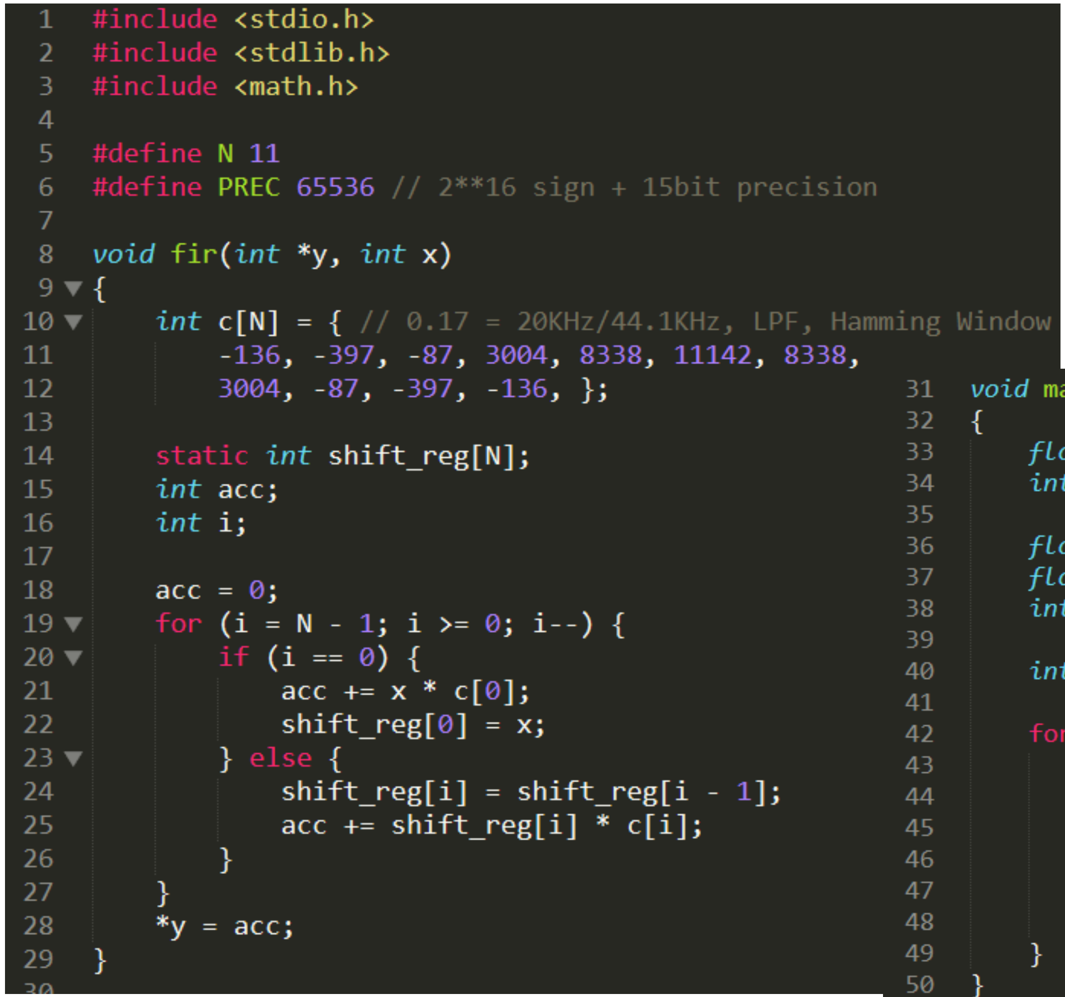
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "ap_int.h"

#define N 11
#define PREC 65536 // 2**16 sign + 15bit precision

/* comment out for bitwith optmization part */
typedef ap_int<16> data_t;
typedef ap_int<16> coef_t;
typedef ap_int<24> acc_t;
/*
typedef int data_t;
typedef int coef_t;
typedef int acc_t;
*/
void fir_array_partition(int *y, int x)
{
        coef_t c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
                -136, -397, -87, 3004, 8338, 11142, 8338,
                3004, -87, -397, -136, };
#pragma HLS array_partition variable=c complete

        static data_t shift_reg[N];
```

Code Reconstruction

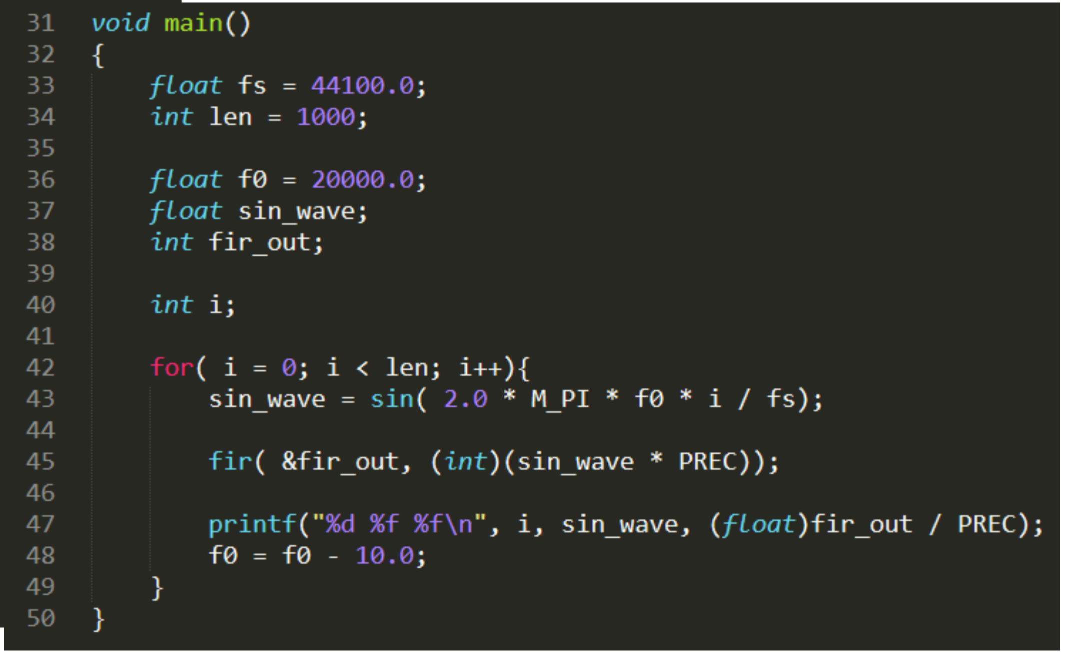
- Writing highly optimized synthesizable HLS code is often not a straightforward process.
- It involves a deep understanding of the application at hand, the ability to change the code such that the Vivado HLS tool creates optimized hardware structures and utilizes the directives in an effective manner
    - FSM-based RTL design experience will help to understand

Convert to Fixed Point Precision

See, https://github.com/HirokiNakahara/FPGA_lecture/tree/master/Lec7_Practical_RTL_design/fir_int.c



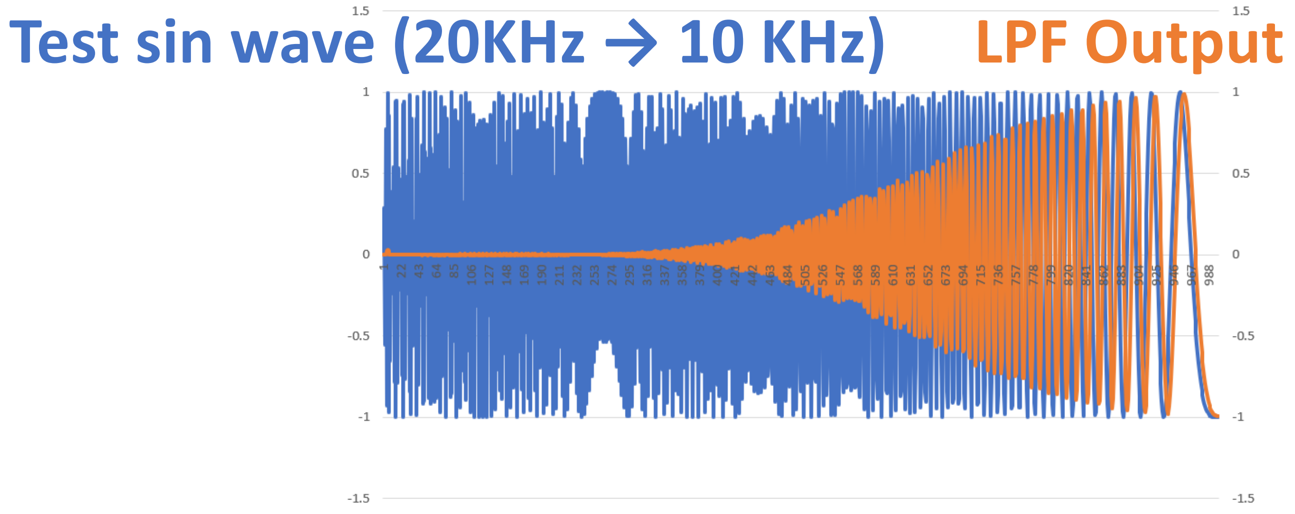
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 11
#define PREC 65536 // 2**16 sign + 15bit precision

void fir(int *y, int x)
{
    int c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };

    static int shift_reg[N];
    int acc;
    int i;

    acc = 0;
    for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```



```c
void main()
{
    float fs = 44100.0;
    int len = 1000;

    float f0 = 20000.0;
    float sin_wave;
    int fir_out;

    int i;

    for( i = 0; i < len; i++){
        sin_wave = sin( 2.0 * M_PI * f0 * i / fs);

        fir( &fir_out, (int)(sin_wave * PREC));

        printf("%d %f %f\n", i, sin_wave, (float)fir_out / PREC);
        f0 = f0 - 10.0;
    }
}
```

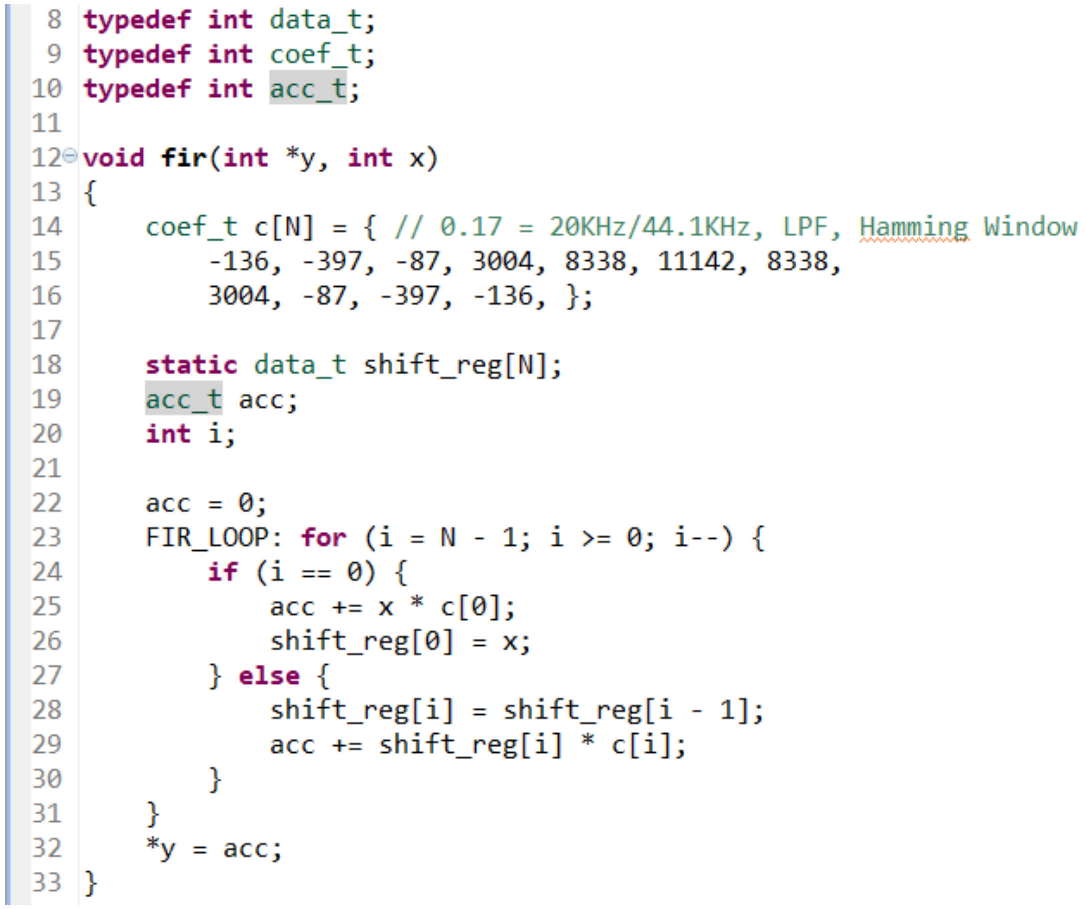
Debug for C Description

- Confirm the operation of FIR
- In/Out are reused as a testbench for HDL simulation
- Note, a parallel operation cannot be verified
- Area and speed of the circuit can not be estimated



For Useful Coding

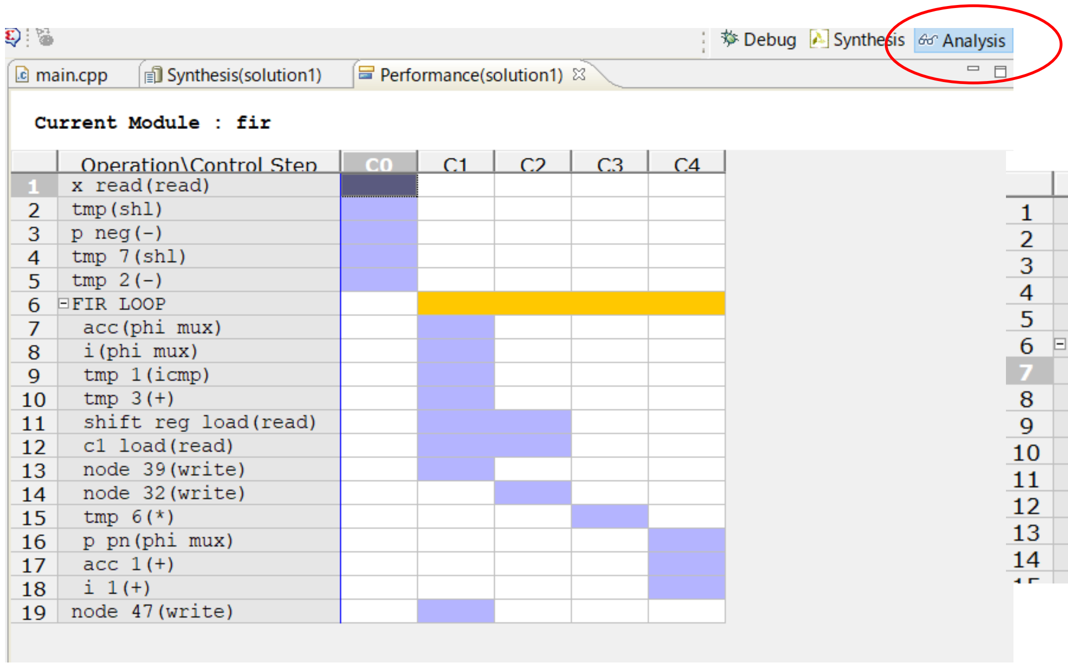
- Use typedef for different variables for changing the types of data (described later)
- Assign labels into loops for debugging



```c
typedef int data_t;
typedef int coef_t;
typedef int acc_t;

void fir(int *y, int x)
{
    coef_t c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };

    static data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    FIR_LOOP: for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

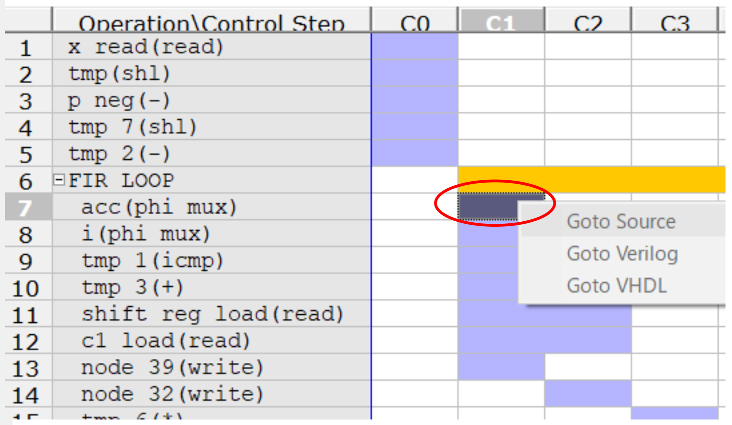
Performance Analysis on Vivado HLS

- Click "Analysis", right click on each block, then select "Goto Source"



Current Module : fir

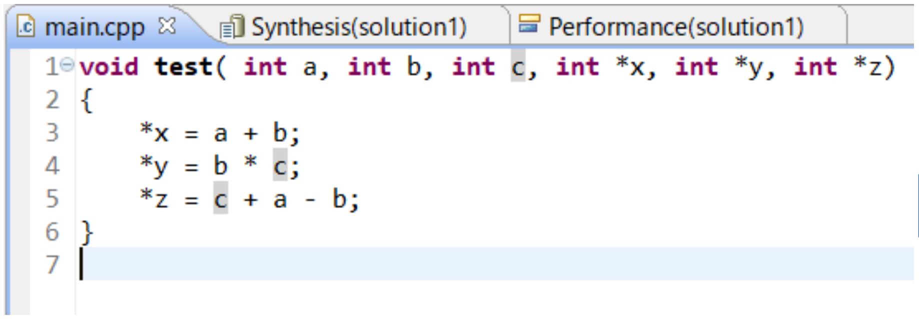
	Operation\Control Step	C0	C1	C2	C3	C4
1	x read(read)	■				
2	tmp(shl)	▨				
3	p neg(-)	▨				
4	tmp 7(shl)	▨				
5	tmp 2(-)	▨				
6	⊟FIR LOOP		🟨	🟨	🟨	🟨
7	acc(phi mux)		▨			
8	i(phi mux)		▨			
9	tmp 1(icmp)		▨			
10	tmp 3(+)		▨			
11	shift reg load(read)		▨	▨		
12	c1 load(read)		▨	▨		
13	node 39(write)		▨			
14	node 32(write)			▨		
15	tmp 6(*)				▨	
16	p pn(phi mux)					▨
17	acc 1(+)					▨
18	i 1(+)					▨
19	node 47(write)		▨			



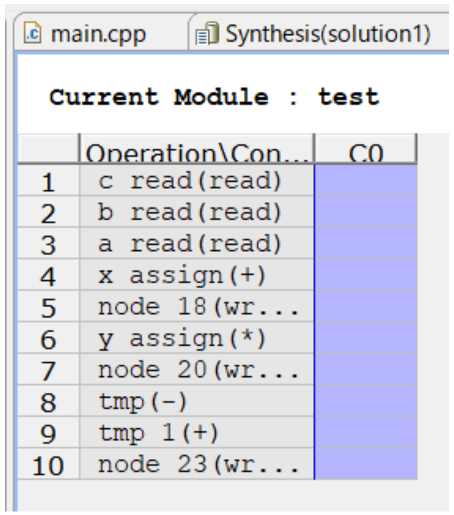
	Operation\Control Step	C0	C1	C2	C3
1	x read(read)	▨			
2	tmp(shl)	▨			
3	p neg(-)	▨			
4	tmp 7(shl)	▨			
5	tmp 2(-)	▨			
6	⊟FIR LOOP		🟨	🟨	🟨
7	acc(phi mux)		■		
8	i(phi mux)		▨		
9	tmp 1(icmp)		▨		
10	tmp 3(+)		▨		
11	shift reg load(read)		▨	▨	
12	c1 load(read)		▨	▨	
13	node 39(write)		▨		
14	node 32(write)			▨	

Parallel Computation Manner

- As same as the RTL design, independent operations are executed in parallel

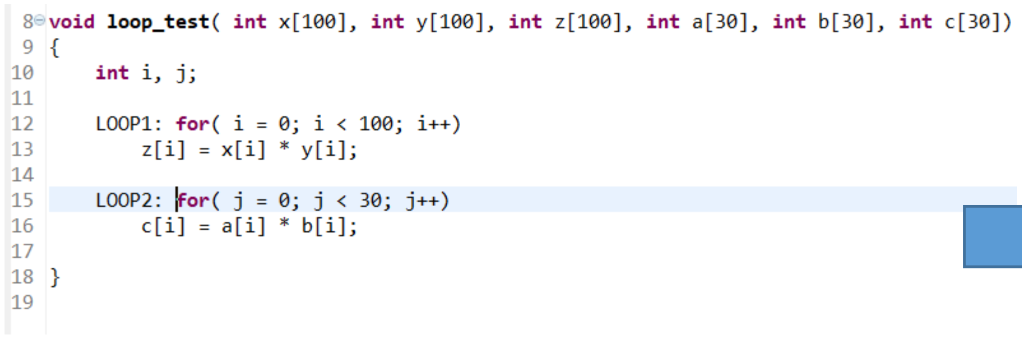


```
void test( int a, int b, int c, int *x, int *y, int *z)
{
    *x = a + b;
    *y = b * c;
    *z = c + a - b;
}
```

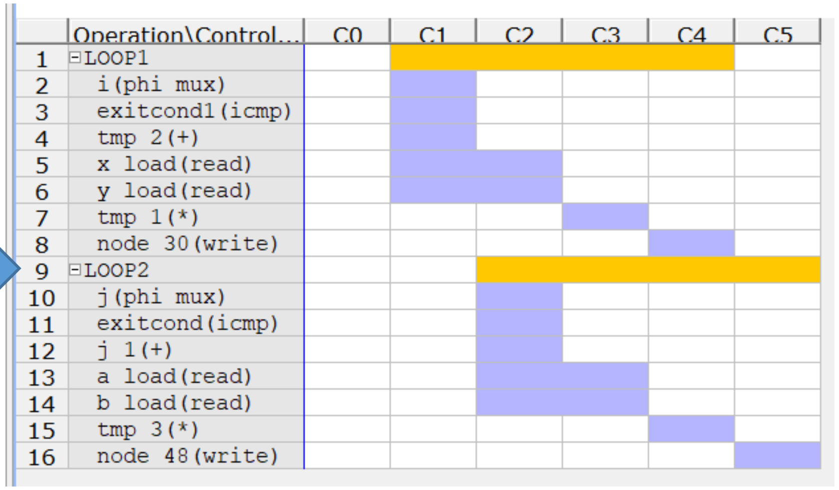


Current Module : test

	Operation\Con...	C0
1	c read(read)	🟦
2	b read(read)	🟦
3	a read(read)	🟦
4	x assign(+)	🟦
5	node 18(wr...	🟦
6	y assign(*)	🟦
7	node 20(wr...	🟦
8	tmp(-)	🟦
9	tmp 1(+)	🟦
10	node 23(wr...	🟦



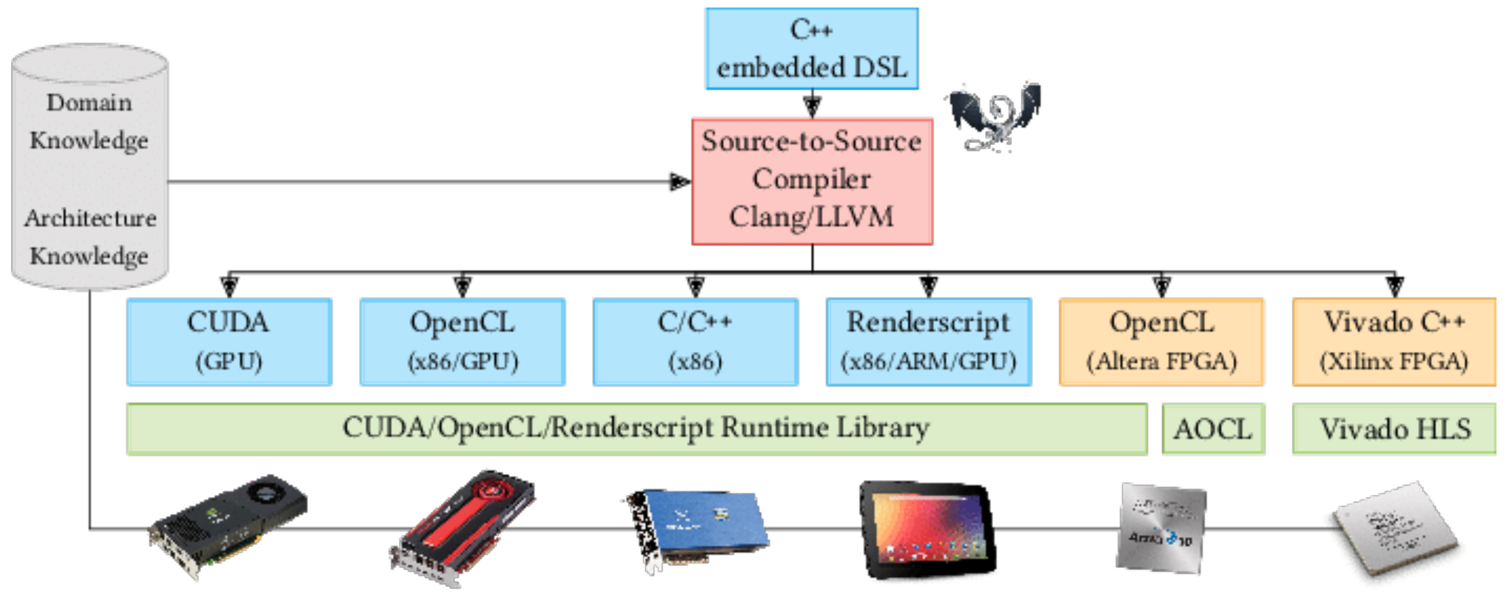
```
void loop_test( int x[100], int y[100], int z[100], int a[30], int b[30], int c[30])
{
    int i, j;

    LOOP1: for( i = 0; i < 100; i++)
        z[i] = x[i] * y[i];

    LOOP2: for( j = 0; j < 30; j++)
        c[i] = a[i] * b[i];

}
```



	Operation\Control...	C0	C1	C2	C3	C4	C5
1	⊟LOOP1		🟨	🟨	🟨	🟨	
2	i(phi mux)		🟦				
3	exitcond1(icmp)		🟦				
4	tmp 2(+)		🟦				
5	x load(read)		🟦	🟦			
6	y load(read)		🟦	🟦			
7	tmp 1(*)				🟦		
8	node 30(write)					🟦	
9	⊟LOOP2			🟨	🟨	🟨	🟨
10	j(phi mux)			🟦			
11	exitcond(icmp)			🟦			
12	j 1(+)			🟦			
13	a load(read)			🟦	🟦		
14	b load(read)			🟦	🟦		
15	tmp 3(*)					🟦	
16	node 48(write)						🟦

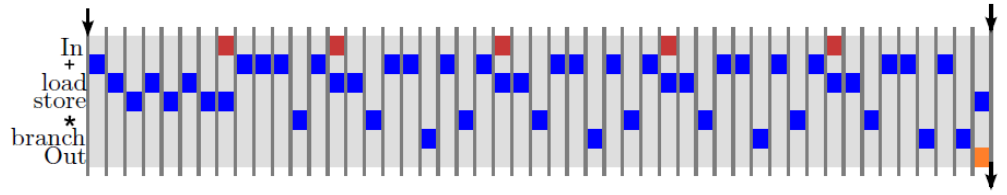
Low Level Virtual Machine (LLVM)

- Modularized, reusable compiler and toolchain technology
- Front end of C, C ++, Objective-C etc.
- Convert to LLVM-IR (Internal Representation)
- Then, optimized for Hardware (FPGA)
- Rust, Clang, LDC, Vivado HLS, Intel OpenCL



LLVM-IR Example: FIR Filter

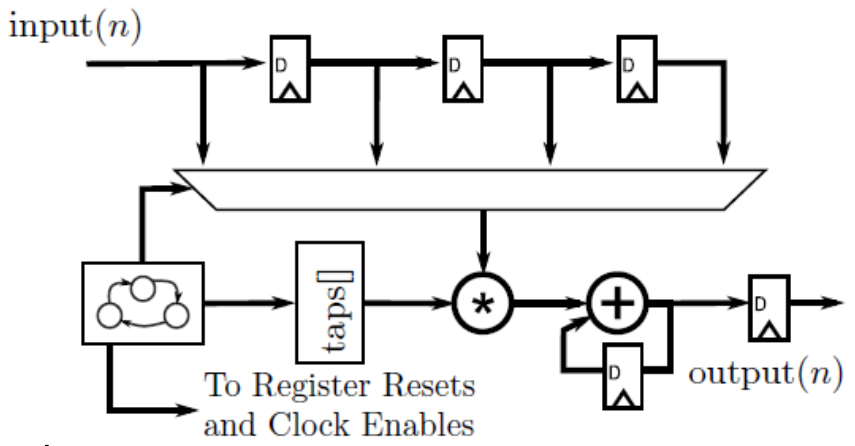
```
fir:
.frame r1,0,r15 # vars= 0, regs= 0, args= 0
.mask 0x00000000
addik r3,r0,delay_line.1450
lwi r4,r3,8 # Unrolled loop to shift the delay line
swi r4,r3,12
lwi r4,r3,4
swi r4,r3,8
lwi r4,r3,0
swi r4,r3,4
swi r5,r3,0 # Store the new input sample into the delay line
addik r5,r0,4  # Initialize the loop counter
addk r8,r0,r0 # Initialize accumulator to zero
addk r4,r8,r0 # Initialize index expression to zero
$L2:
muli r3,r4,4 # Compute a byte offset into the delay_line array
addik r9,r3,delay_line.1450
lw r3,r3,r7 # Load filter tap
lwi r9,r9,0 # Load value from delay line
mul r3,r3,r9 # Filter Multiply
addk r8,r8,r3 # Filter Accumulate
addik r5,r5,–1 # update the loop counter
bneid r5,$L2
addik r4,r4,1 # branch delay slot, update index expression

rtsd r15, 8
swi r8,r6,0  # branch delay slot, store the output
.end fir
```

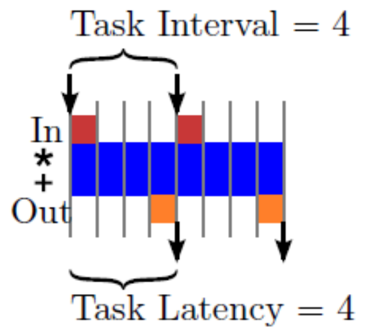


This code is generated using microblazeel-xilinx-linux-gnu-gcc -O1 -mno-xl-soft-mul -S fir.c

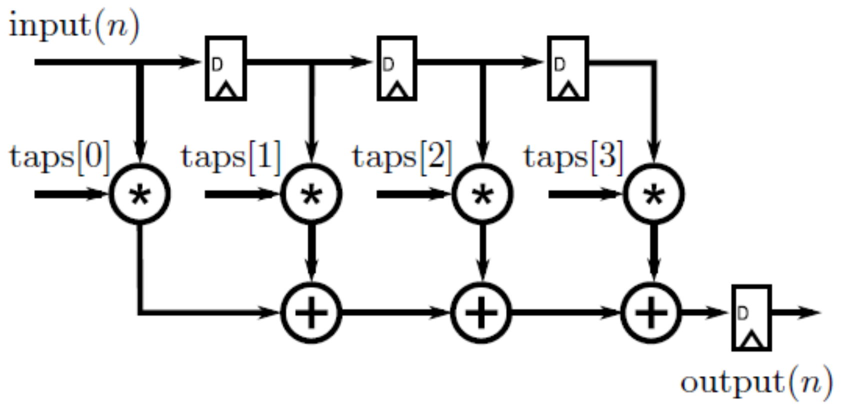
Different Architectures

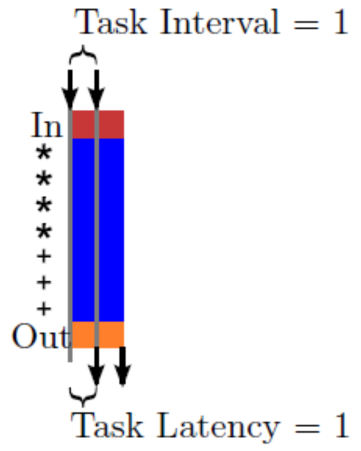
- Sequential manner





- Pipeline manner

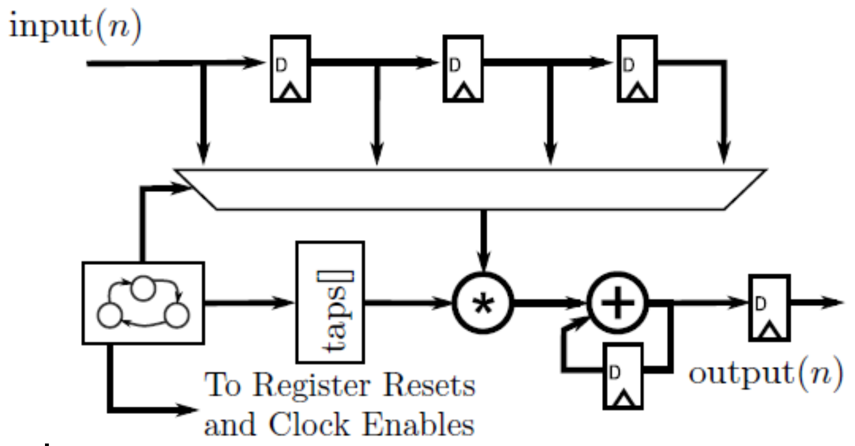


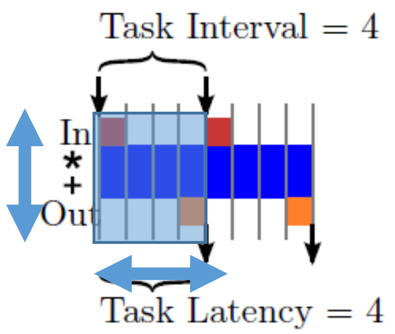


Area-Time Trade-off

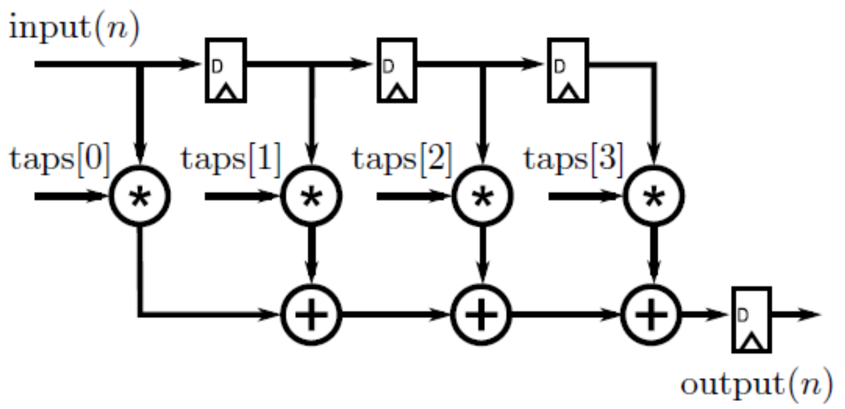
Vertical (Area)
Horizontal (Time)

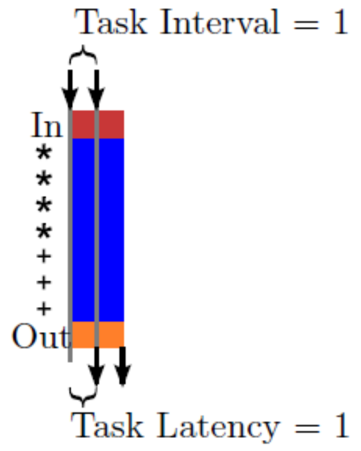
- Sequential manner





- Pipeline manner

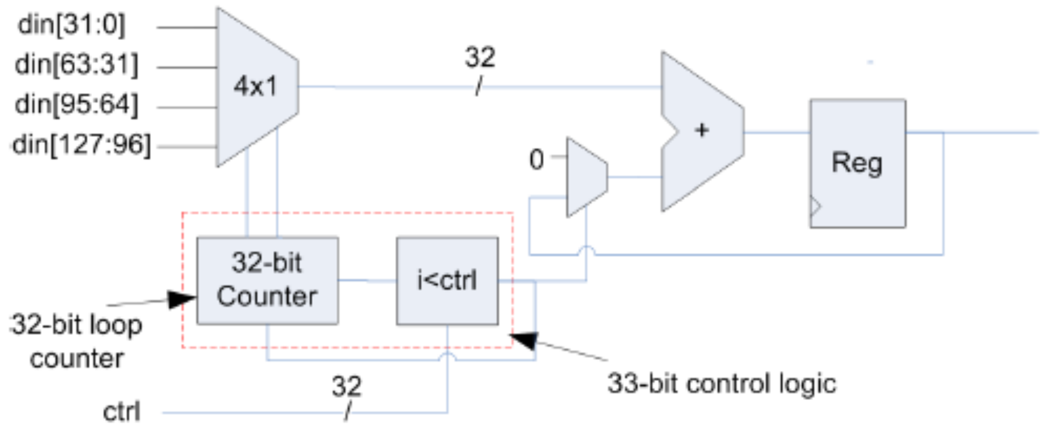




Loop with Conditional Bounds

- Having a variable as the loop upper or lower bound often results in the loop counter hardware being larger than needed
    - Having an unconstrained bit width on the loop exit condition results in control logic larger than needed

```
#include "accum.h"
#include <ac_int.h>
void accumulate(int din[4], int &dout, unsigned  int ctrl){
  int acc=0;
 ACCUM:for(int i=0;i<ctrl;i++){
    acc += din[i];
  }
  dout = acc;
}

```

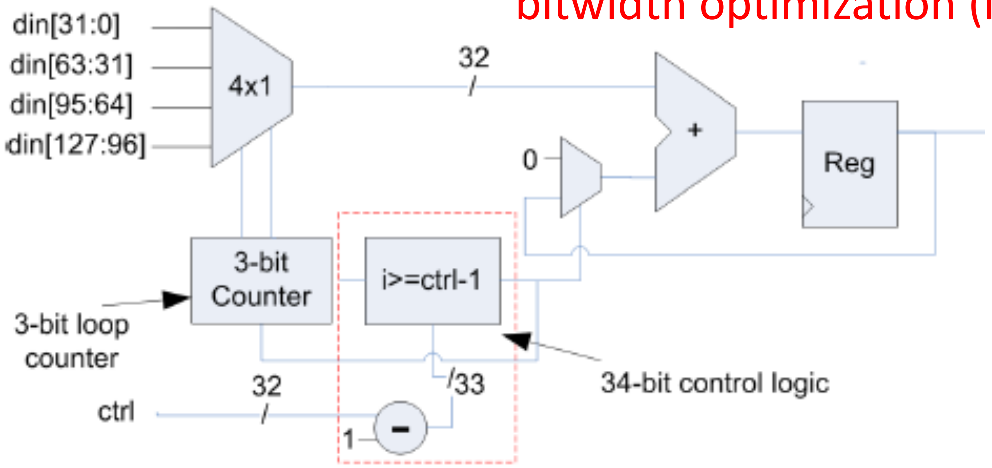


Optimizing the Loop Counter

- In order for HLS to reduce the bit width of the loop counter the loop upper bound should be set to a constant
- However, since the execution of each loop iteration is determined by the variable, "ctrl"
- It is done by using a conditional break in the loop body

```
#include "accum.h"
#include <ac_int.h>
void accumulate(int din[4], int &dout, int ctrl){
  int acc=0;
 ACCUM:for(int i=0;i<4;i++){
     acc += din[i];
     if(i>=ctrl-1)
       break;
  }
  dout = acc;
}
```

It will be reduced by bitwidth optimization (later)

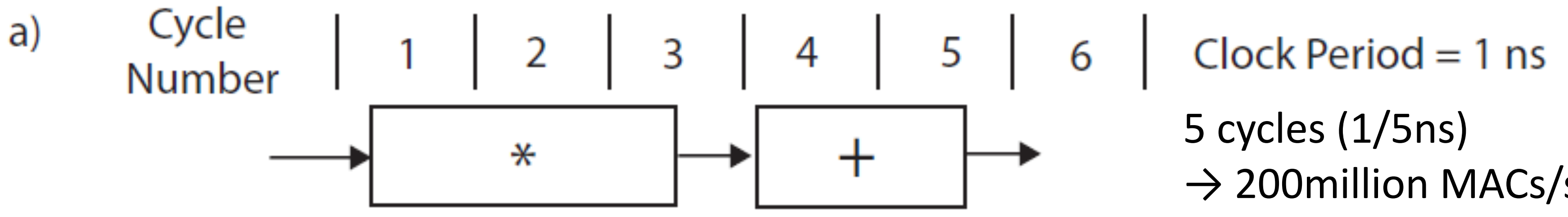


Calculating Performance

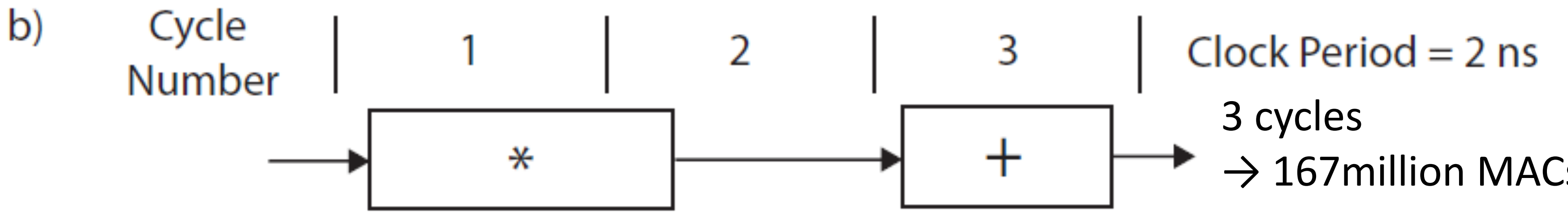
- Necessary to dene precise metrics
- What is "fast" design?
    - Efficiency?
        - operations/sec
        - MACs/sec
        - bits/sec
    - Latency? Throughput? Computation time?
- High-level synthesis tools talk about the designs in terms of number of cycles, and the frequency of the clock
- Select adequate measurement of a target application
- Compare them using the same metric

Operation Chaining

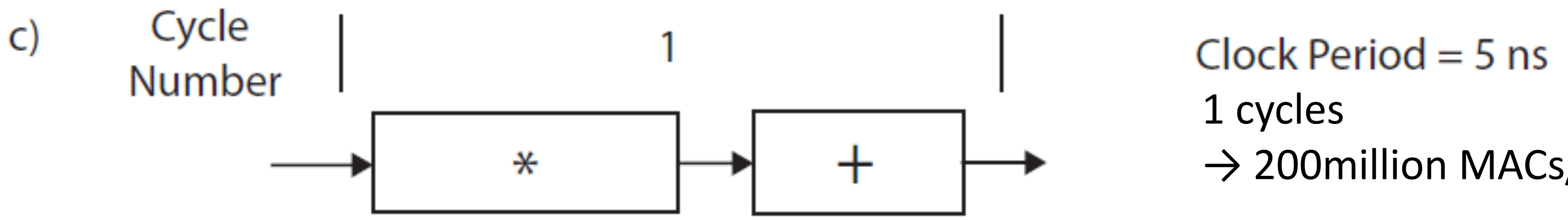
- Consider the multiply accumulate operation that is done in a FIR filter tap
- Assume that the add operation takes 2 ns to complete, and a multiply operation takes 3 ns



5 cycles (1/5ns)
→ 200million MACs/sec



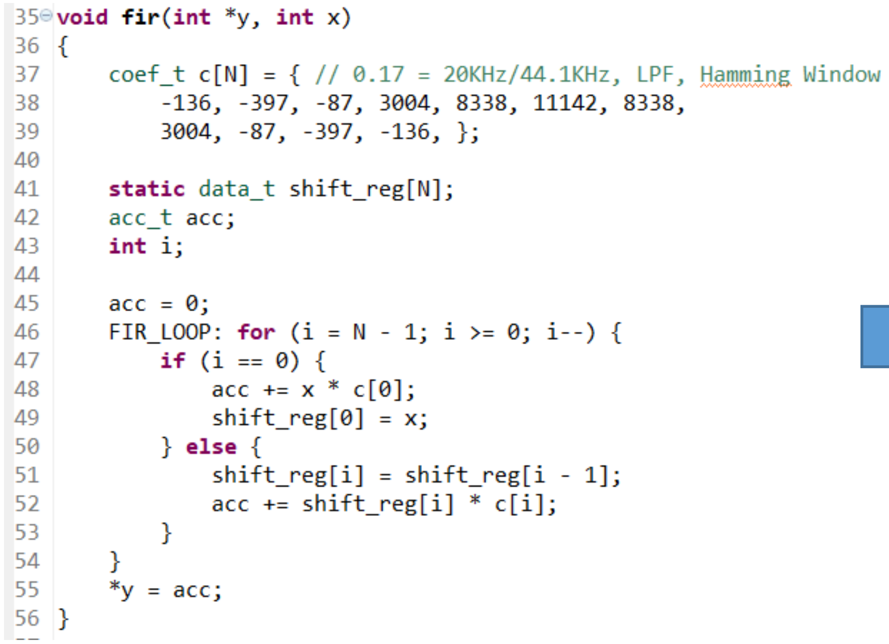
3 cycles
→ 167million MACs/sec



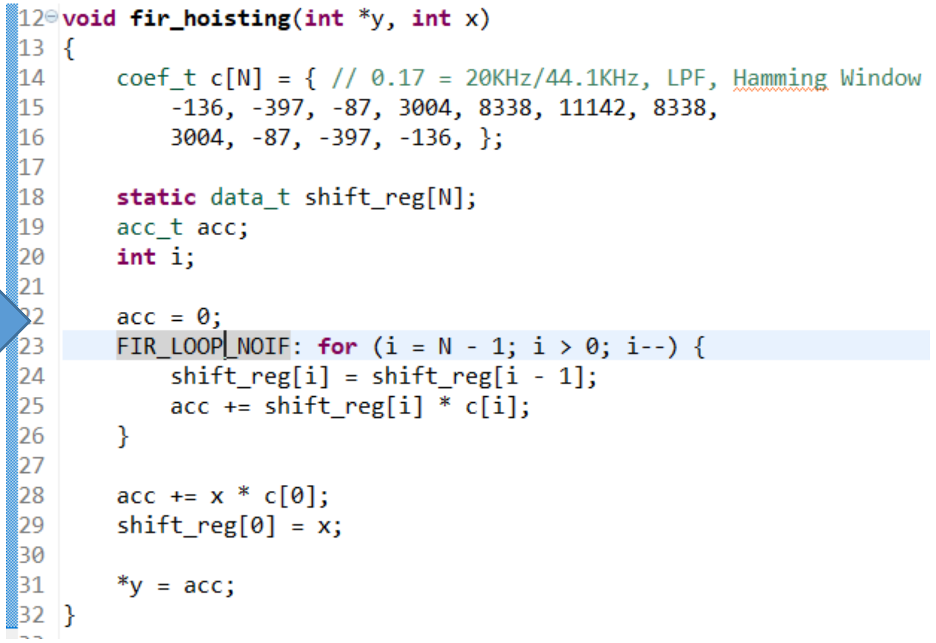
1 cycles
→ 200million MACs/sec

Code Hoisting

- The if/else statement inside of the for loop is inefficient.
- For every control structure in the code, the Vivado HLS tool creates logical hardware that checks if the condition is met
- Therefore, the statements within the if branch can be "hoisted" out of the loop



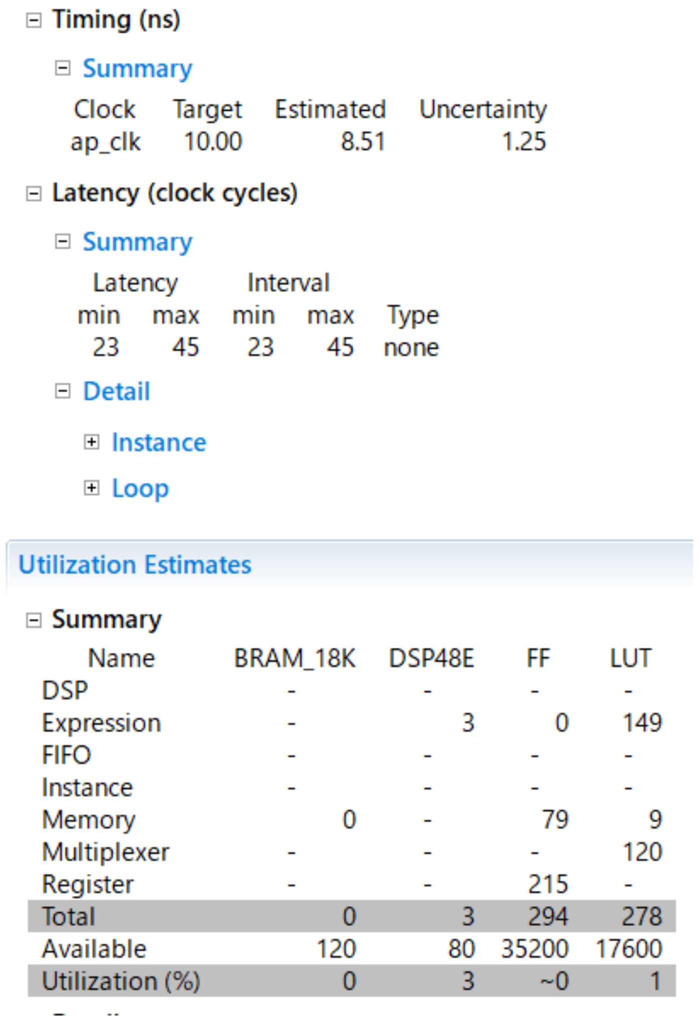
```
void fir(int *y, int x)
{
    coef_t c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };

    static data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    FIR_LOOP: for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```



```
void fir_hoisting(int *y, int x)
{
    coef_t c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };

    static data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    FIR_LOOP_NOIF: for (i = N - 1; i > 0; i--) {
        shift_reg[i] = shift_reg[i - 1];
        acc += shift_reg[i] * c[i];
    }

    acc += x * c[0];
    shift_reg[0] = x;

    *y = acc;
}
```

Comparison

- Original



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.51	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
23	45	23	45	none

Detail

Instance

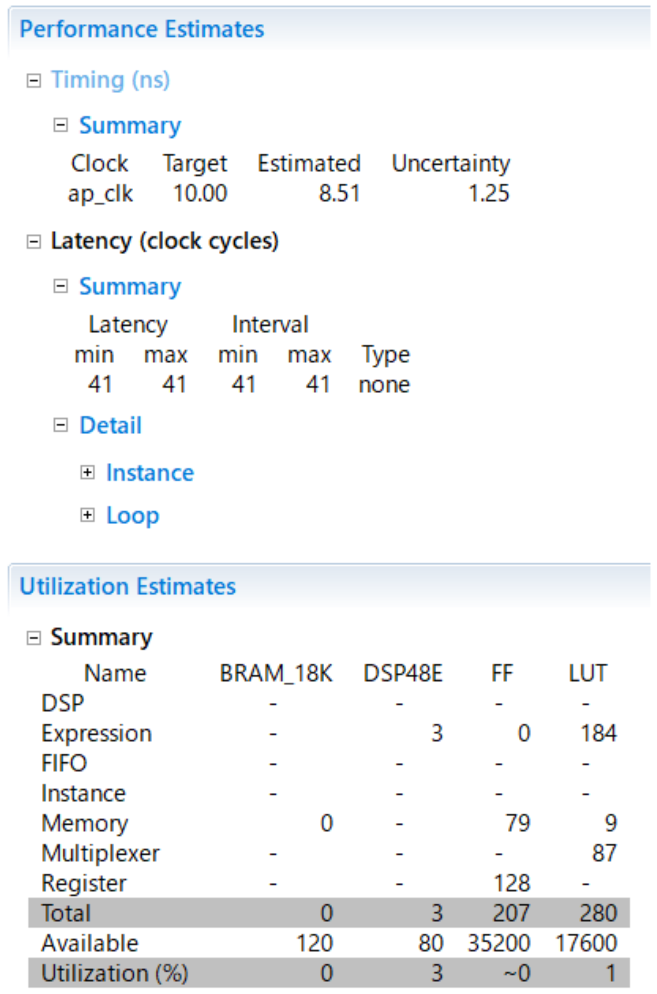
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	149
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	79	9
Multiplexer	-	-	-	120
Register	-	-	215	-
Total	0	3	294	278
Available	120	80	35200	17600
Utilization (%)	0	3	~0	1

with Hoisting of "if"



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.51	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
41	41	41	41	none

Detail

Instance

Loop

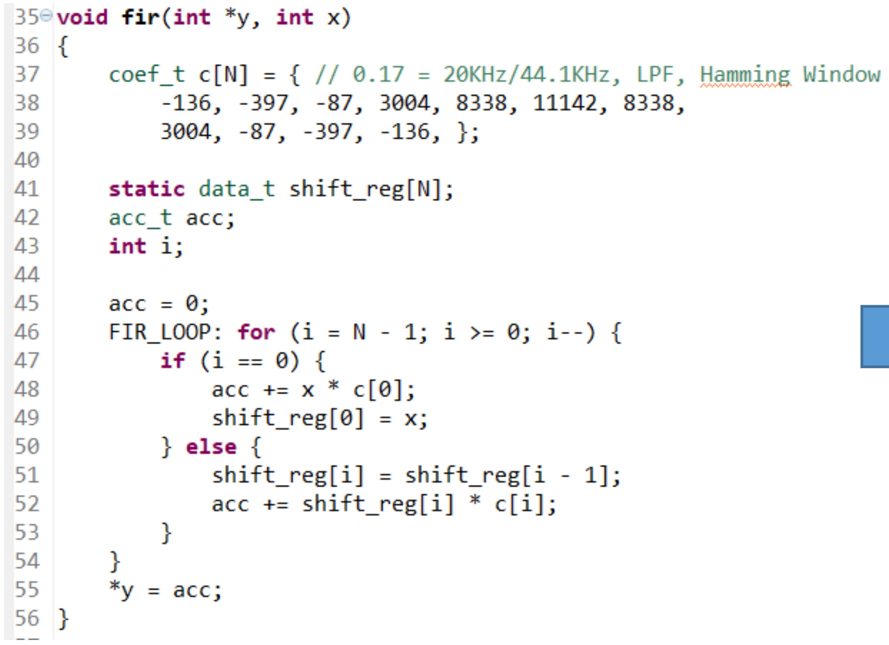
Utilization Estimates

Summary

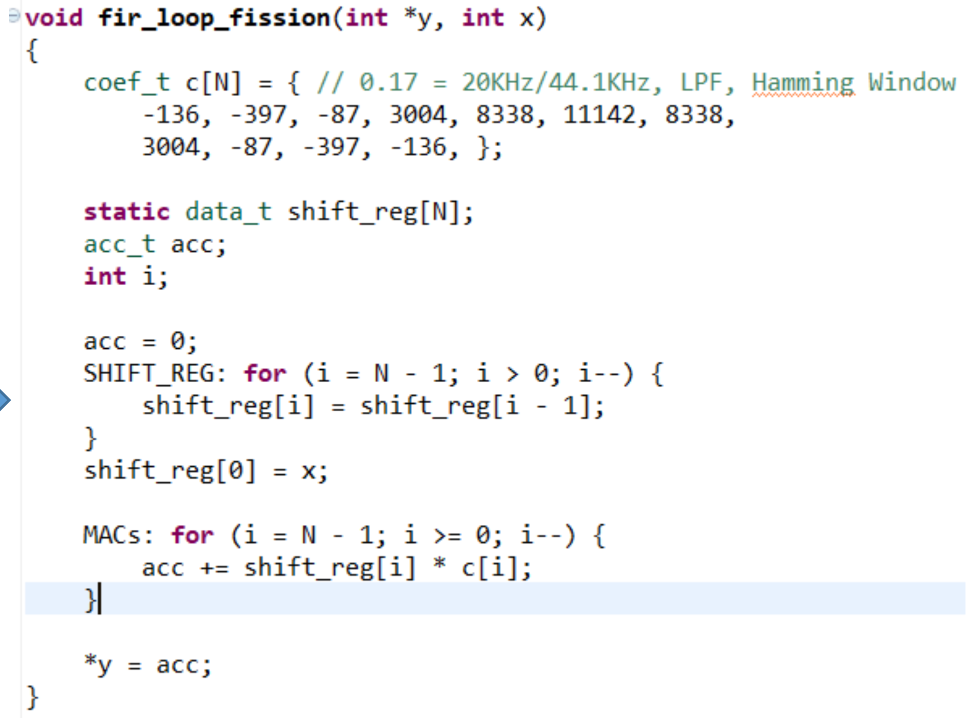
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	184
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	79	9
Multiplexer	-	-	-	87
Register	-	-	128	-
Total	0	3	207	280
Available	120	80	35200	17600
Utilization (%)	0	3	~0	1

Loop Fission

- The FIR has two fundamental operations: Shifts the data through the shift_reg array, and the MAC operations
- Loop fission takes these two operations and implements each of them in their own loop
    - Each one is optimized independently, so it is a decomposition of an FSM



```c
void fir(int *y, int x)
{
    coef_t c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };

    static data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    FIR_LOOP: for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```



```c
void fir_loop_fission(int *y, int x)
{
    coef_t c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };

    static data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    SHIFT_REG: for (i = N - 1; i > 0; i--) {
        shift_reg[i] = shift_reg[i - 1];
    }
    shift_reg[0] = x;

    MACs: for (i = N - 1; i >= 0; i--) {
        acc += shift_reg[i] * c[i];
    }

    *y = acc;
}
```

Loop Unrolling

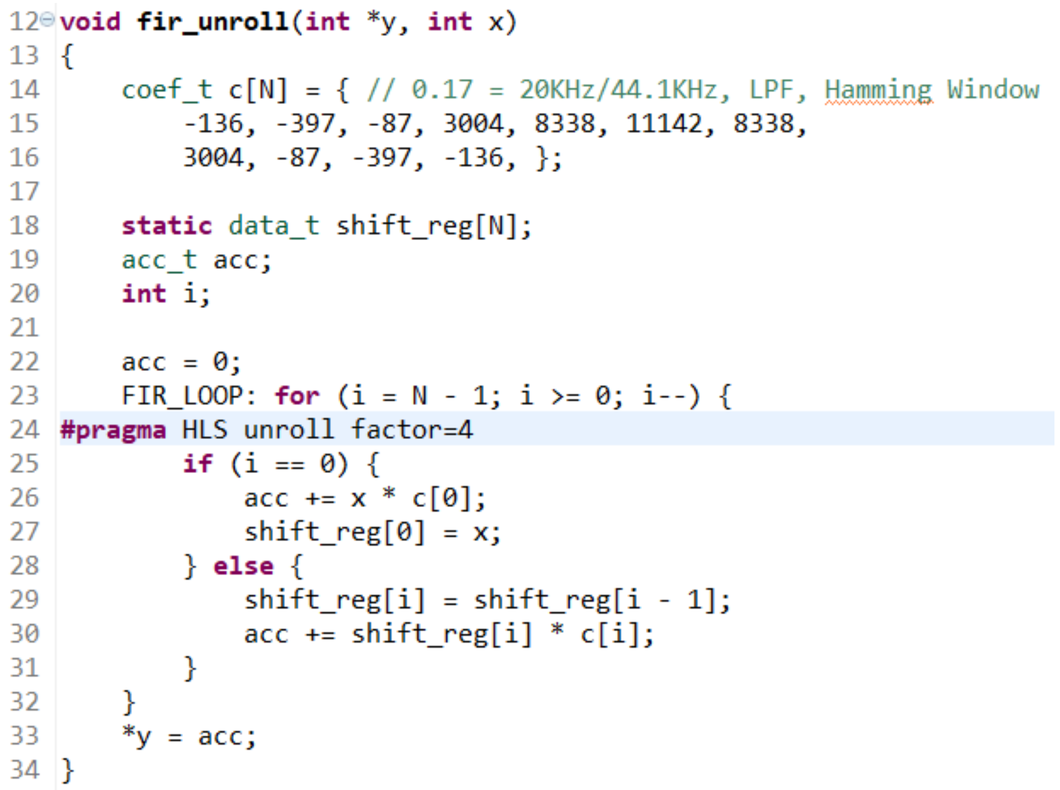
- By default, the Vivado HLS tool synthesizes for loops in a sequential manner
- The data path executes sequentially for each iteration of the loop
- Manually unrolling the SHIFT_REG loop

```
for (i = N − 1; i > 1; i = i − 2) {
    shift_reg[i] = shift_reg[i − 1];
    shift_reg[i − 1] = shift_reg[i − 2];
}
if (i == 1) {
    shift_reg[1] = shift_reg[0];
}
shift_reg[0] = x;
```

Unroll Pragma

#pragma HLS unroll factor=n

(if factor is none, the HLS tries to unroll all operations!!)

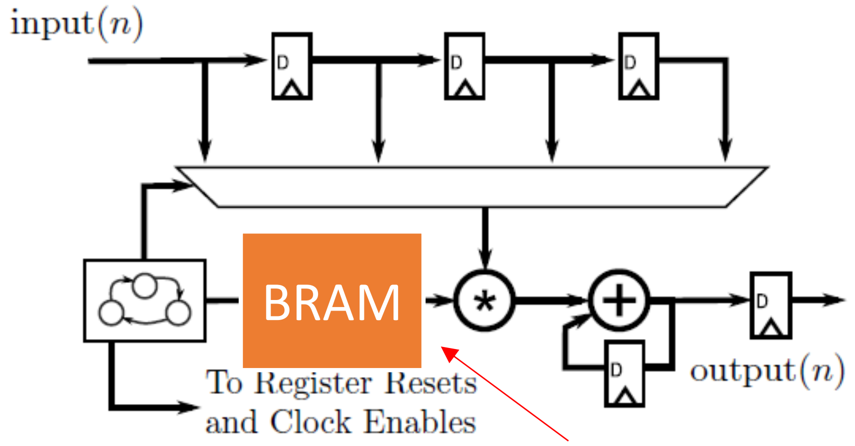


```
void fir_unroll(int *y, int x)
{
    coef_t c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };

    static data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    FIR_LOOP: for (i = N - 1; i >= 0; i--) {
#pragma HLS unroll factor=4
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

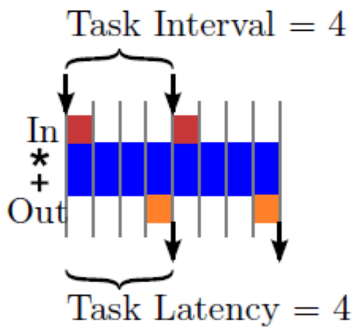
If you design does not synthesize in under 15 minutes, you should carefully consider the effect of your optimizations. It is certainly possible that large designs can take a significant amount for the Vivado HLS to synthesize them.

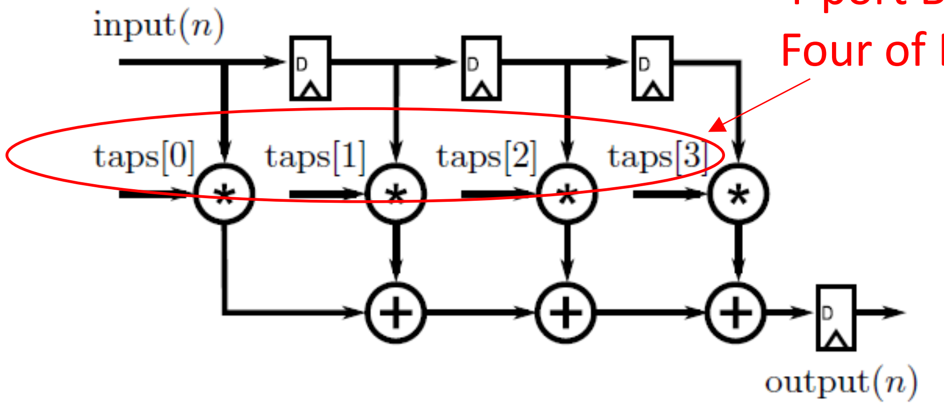
Partition BRAM into Smaller One?

- Use "#pragma HLS array_partition"

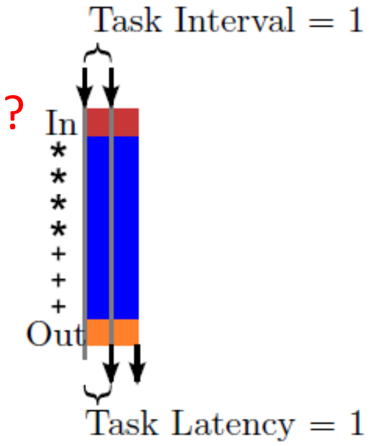


Single port BRAM



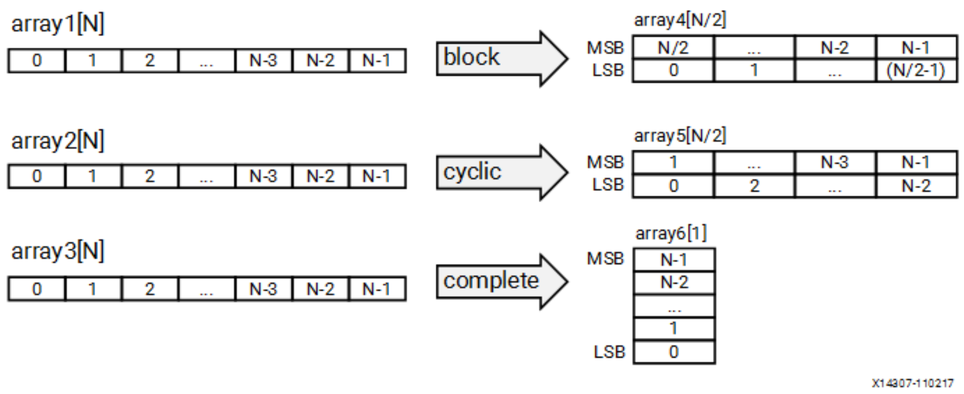


4-port BRAM?
Four of BRAMs?



Array_Partition

#pragma HLS ARRAY_PARTITION variable=(variable name) (access pattern) factor=(# of partitions) dim=(array dimension)



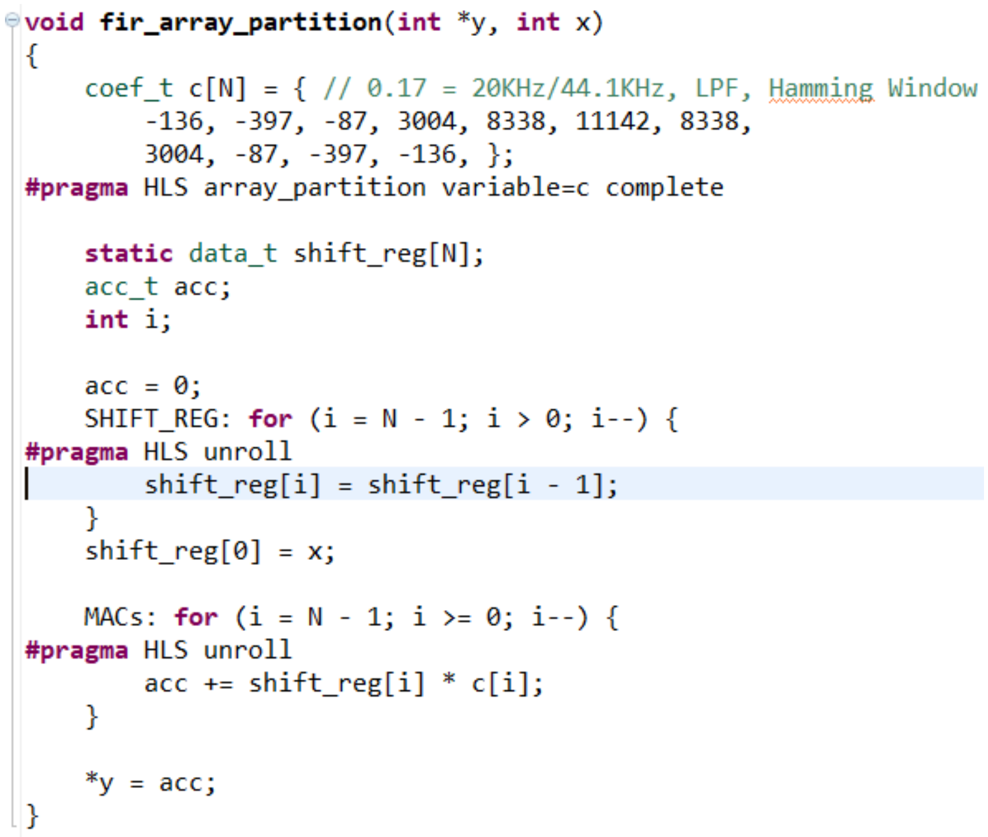
my_array[10][6][4] ➔ partition dimension 1 ➔ my_array_0[6][4]
my_array_1[6][4]
my_array_2[6][4]
my_array_3[6][4]
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]

access pattern

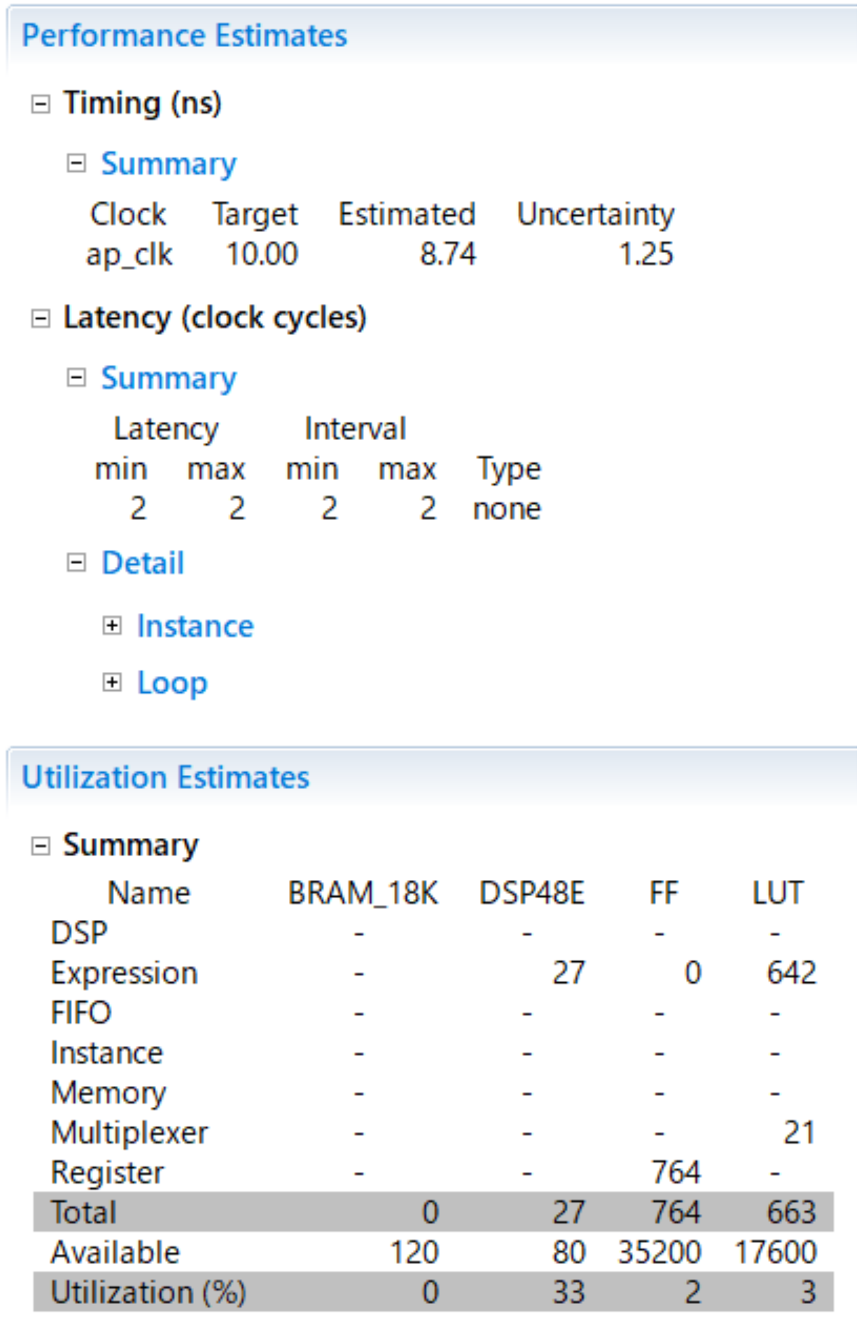
dimension
(dim=0 denotes all dimension are partitioned)

Example

- Good performance! But...



```
void fir_array_partition(int *y, int x)
{
    coef_t c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };
#pragma HLS array_partition variable=c complete

    static data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    SHIFT_REG: for (i = N - 1; i > 0; i--) {
#pragma HLS unroll
        shift_reg[i] = shift_reg[i - 1];
    }
    shift_reg[0] = x;

    MACs: for (i = N - 1; i >= 0; i--) {
#pragma HLS unroll
        acc += shift_reg[i] * c[i];
    }

    *y = acc;
}
```



Performance Estimates

⊟ **Timing (ns)**

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.74	1.25

⊟ **Latency (clock cycles)**

⊟ Summary

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

⊟ Detail

⊞ Instance

⊞ Loop

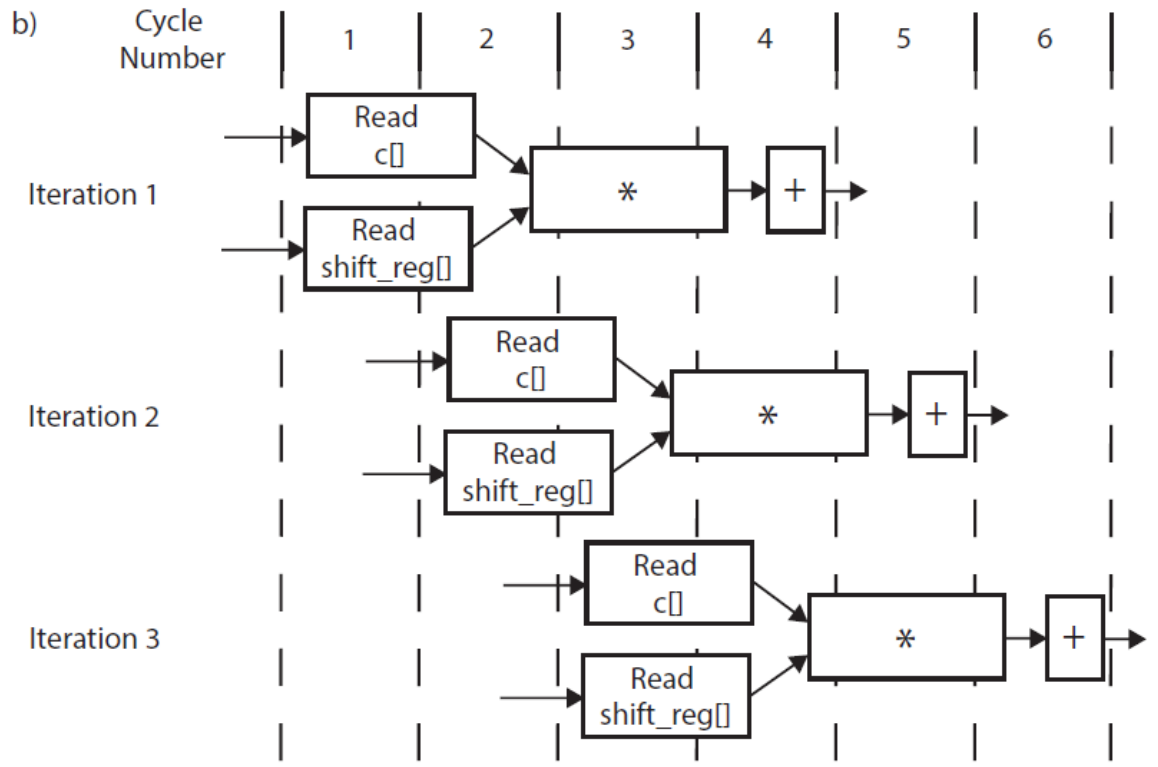
Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	27	0	642
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	21
Register	-	-	764	-
Total	0	27	764	663
Available	120	80	35200	17600
Utilization (%)	0	33	2	3

Loop Pipelining

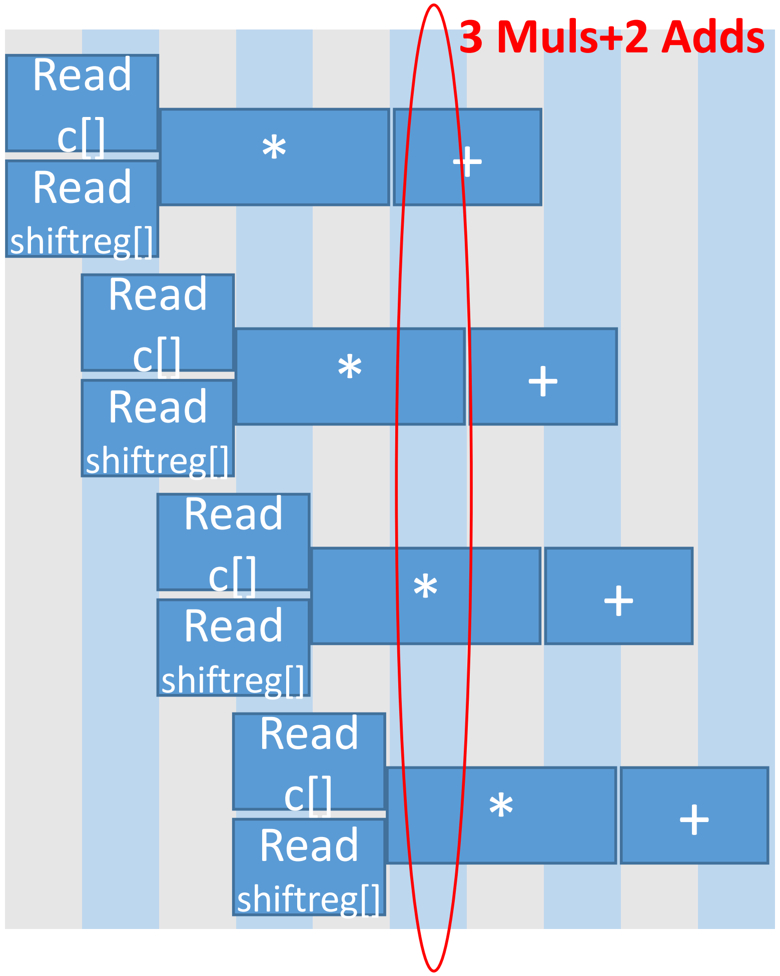
- All of the statements in the second iteration happen only when all of the statements from the first iteration are complete
- Schedule for three iterations of a pipelined version of the MAC for loop



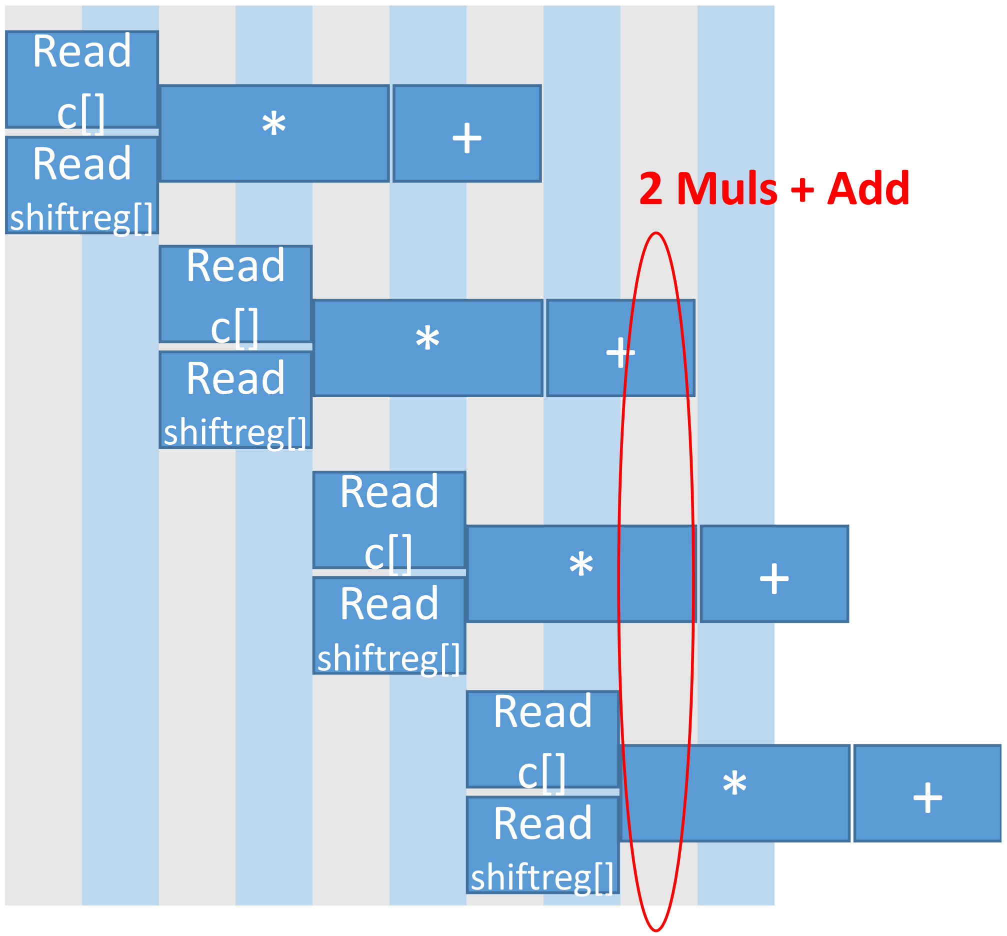
Loop Initiation Interval (II)

- The number of clock cycles until the next iteration of the loop can start
- Note that, this may <u>not always be possible</u> due to resource/timing constraints and/or dependencies in the code

#pragma HLS pipeline II=1



#pragma HLS pipeline II=2



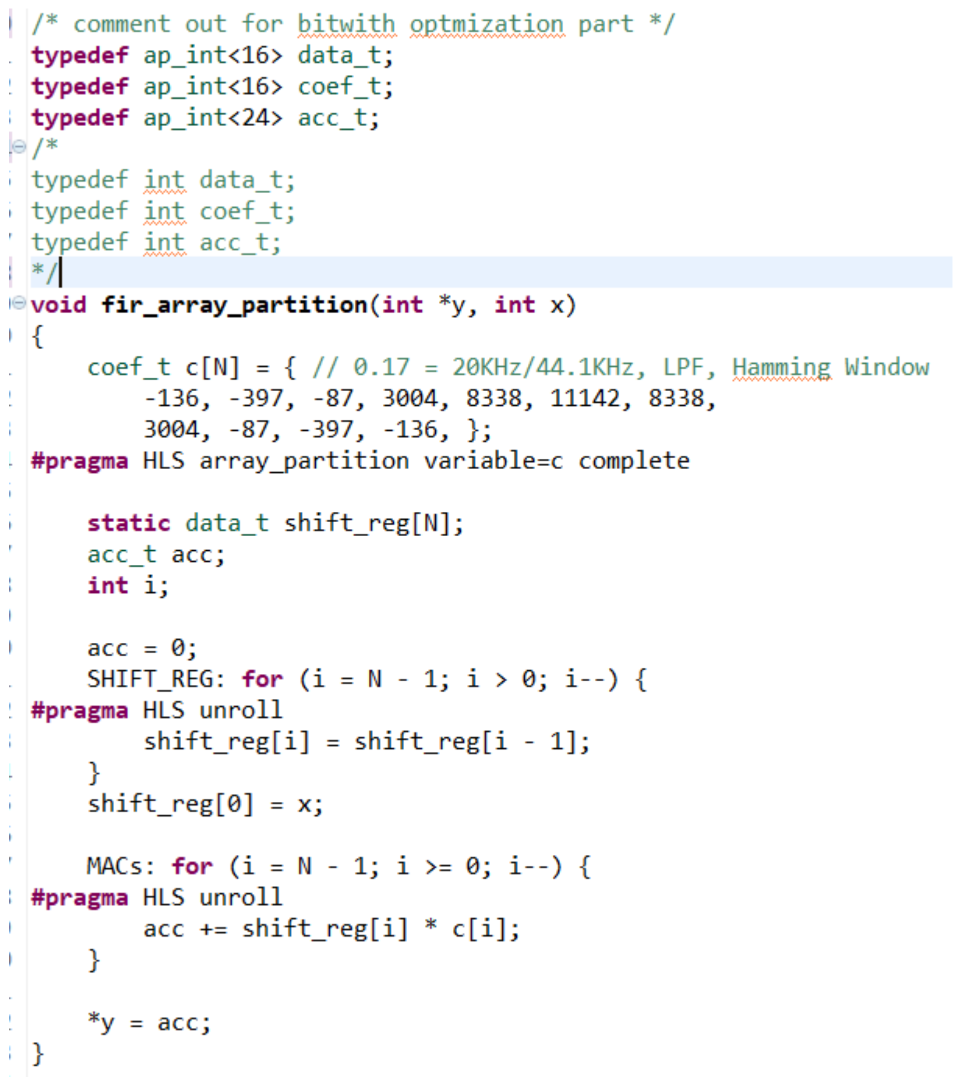
Data Type in C-language

- C language provides many different data types to describe different kinds of behavior
- The primary benefits of using these different data types in software revolve around the amount of storage that the data type require
- All of these data type have a size which is a power of 2
    - (unsigned/singed) int
    - float
    - double
    - (unsigned/singed) char
    - short long
    - long
    - long long

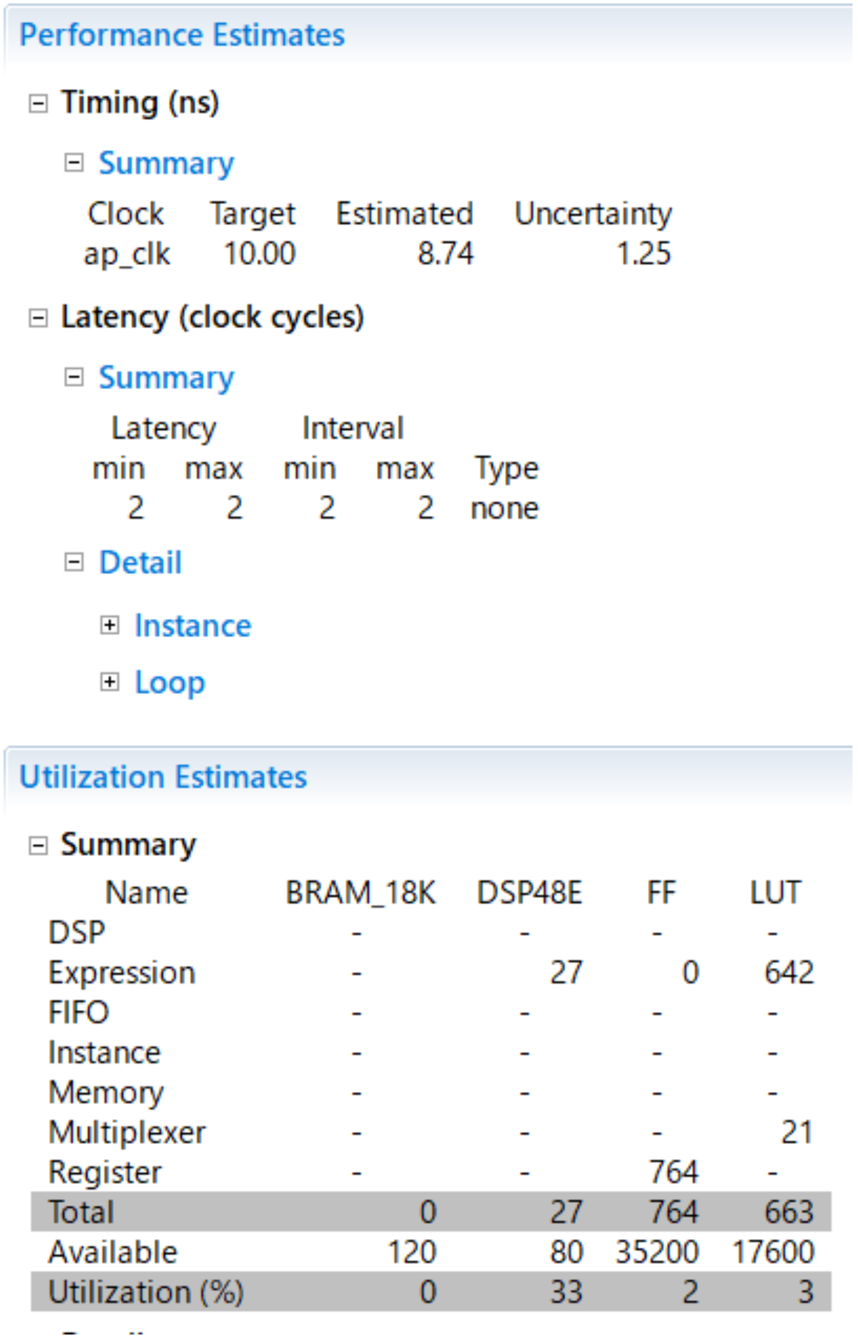
Bitwidth Optimization

- The same benefits are seen in an FPGA implementation, but they are even more pronounced
  - Since the Vivado HLS supports a custom (arbitrary precision) data types
- #include "ap_int.h", then you can use
  - unsigned: ap_uint<width>, where width takes 1 to 1024
  - signed: ap_int<width>

More Reduced!



```
/* comment out for bitwith optmization part */
typedef ap_int<16> data_t;
typedef ap_int<16> coef_t;
typedef ap_int<24> acc_t;
/*
typedef int data_t;
typedef int coef_t;
typedef int acc_t;
*/
void fir_array_partition(int *y, int x)
{
    coef_t c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };
#pragma HLS array_partition variable=c complete

    static data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    SHIFT_REG: for (i = N - 1; i > 0; i--) {
#pragma HLS unroll
        shift_reg[i] = shift_reg[i - 1];
    }
    shift_reg[0] = x;

    MACs: for (i = N - 1; i >= 0; i--) {
#pragma HLS unroll
        acc += shift_reg[i] * c[i];
    }

    *y = acc;
}
```



Performance Estimates

⊟ **Timing (ns)**

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.74	1.25

⊟ **Latency (clock cycles)**

⊟ Summary

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

⊟ Detail

⊞ Instance

⊞ Loop

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	27	0	642
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	21
Register	-	-	764	-
Total	0	27	764	663
Available	120	80	35200	17600
Utilization (%)	0	33	2	3

Exercise

1. (Mandatory) Compare an unrolling FIR design with a pipelined one with respect to HW resource and performance
2. (Optional) Execute an unrolling FIR design on your ZYBO board

If you meet any troubles, don't hesitate to contact me.

nakahara@ict.e.titech.ac.jp

Deadline is *7th, Aug.*, 2020 (At the beginning of the next lecture)