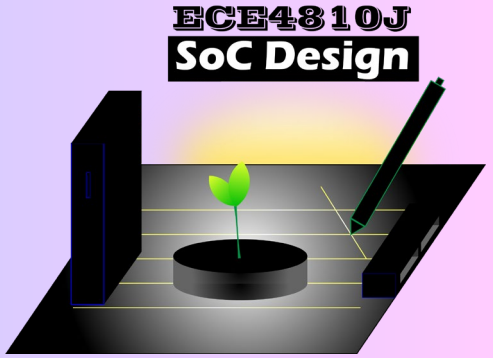
Lab #3
Applications of Vitis HLS & PYNQ

ECE4810J System-on-Chip Design



Yihua Liu
UM-SJTU Joint Institute
ayka_tsuzuki@sjtu.edu.cn
Oct. 10, 2022

Overview

High Level Synthesis

Algorithmic HLS does several things automatically that an RTL designer does manually:

- HLS analyzes and exploits the concurrency in an algorithm.
- HLS inserts registers as necessary to limit critical paths and achieve a desired clock frequency.
- HLS generates control logic that directs the data path.
- HLS implements interfaces to connect to the rest of the system.
- HLS maps data onto storage elements to balance resource usage and bandwidth.
- HLS maps computation onto logic elements performing user specified and automatic optimizations to achieve the most efficient implementation.

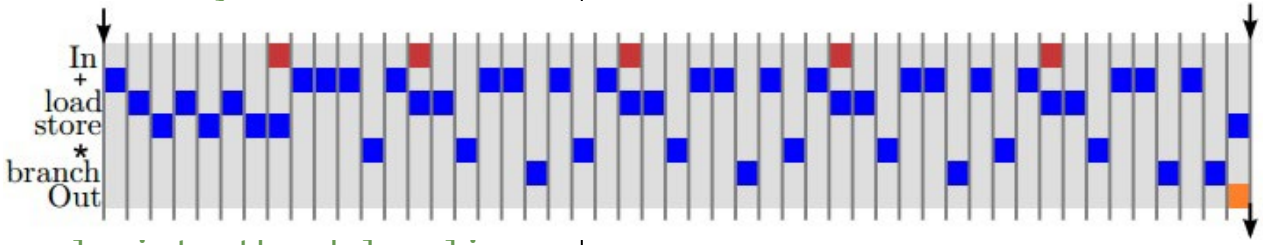
We make the following assumptions about the input function specification:

- No dynamic memory allocation (no operators like `malloc()`, `free()`, `new`, and `delete()`)
- Limited use of pointers-to-pointers (e.g., may not appear at the interface)
- System calls are not supported (e.g., `abort()`, `exit()`, `printf()`, etc. They can be used in the code, e.g., in the testbench, but they are ignored (removed) during synthesis.
- Limited use of other standard libraries (e.g., only common `math.h` functions are supported)
- Limited use of function pointers and virtual functions in C++ classes (function calls must be compile-time determined by the compiler).
- No recursive function calls.
- The interface must be precisely defined.

High Level Synthesis

```
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
```

A key question to understand is: “What circuit is generated from this code?”.

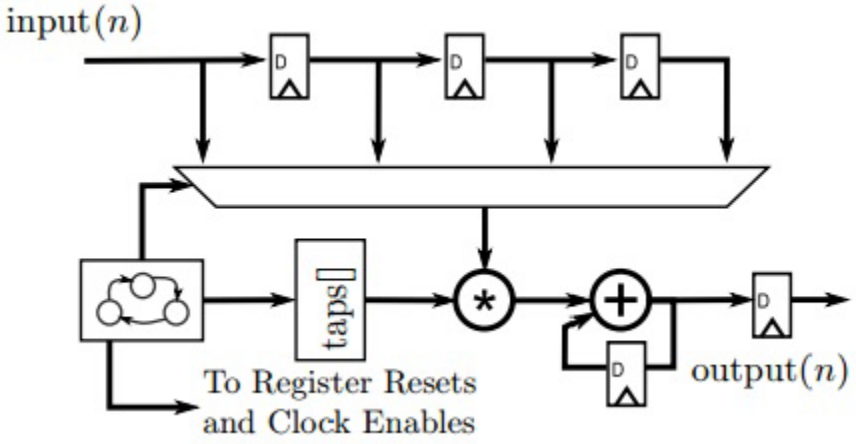


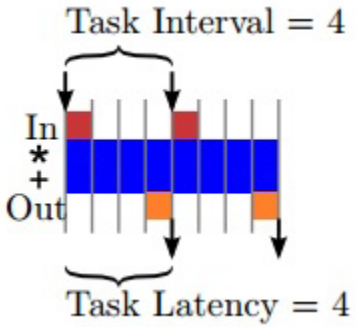
One possible circuit would execute the code sequentially, as would a simple RISC microprocessor.

microblazeel-xilinx-linux-gnu-gcc -01 -mno -xl -soft -mul S fir.c

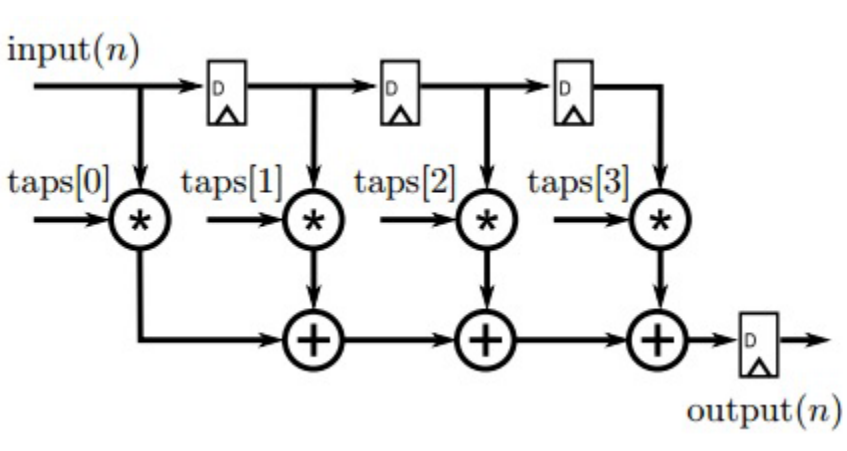
High Level Synthesis

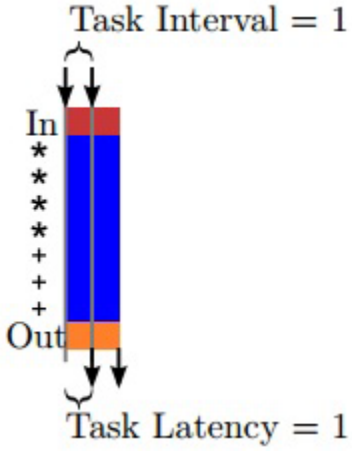
However, the Vitis HLS tool can also generate higher performance pipelined and parallel architectures. One important class of architectures is called a *function pipeline*. A function pipeline architecture is derived by considering the code within the function to be entirely part of a computational data path, with little control logic. Loops and branches in the code are converted into unconditional constructs. The Vitis HLS tool can be directed to generate a function pipeline by placing the `#pragma HLS pipeline` directive in the body of a function.





A "one tap per clock" architecture for an FIR filter.





A "one sample per clock" architecture for an FIR filter.

High Level Synthesis

JOINT INSTITUTE
交大密西根学院

Arbitrary Precision Data Types Library

Language	Integer Data Type	Required Header
C++	ap_[u]int<W> (1024 bits) Can be extended to 4K bits	#include <ap_int.h>
C++	ap_[u]fixed<W,I,Q,O,N>	#include <ap_fixed.h>

W	Word length in bits	
I	The number of bits used to represent the integer value	
Q	Quantization mode	
	AP_RND	Rounds to plus infinity
	AP_RND_ZERO	Rounds to zero
	AP_RND_MIN_INF	Rounds to minus infinity
	AP_RND_INF	Rounds to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity (default)
	AP_TRN_ZERO	Truncation to zero
O	Overflow mode	
	AP_SAT	Saturation
	AP_SAT_ZERO	Saturation to zero
	AP_SAT_SYM	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	This defines the number of saturation bits in overflow wrap mode	

High Level Synthesis

JOINT INSTITUTE 交大密西根学院

```
#include "cpp_ap_int_arith.h"
void cpp_ap_int_arith (din_A inA, din_B inB, din_C
inC, din_D inD, dout_1 *out1, dout_2 *out2, dout_3
*out3, dout_4 *out4) {
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;
}
```

```
#ifndef _CPP_AP_INT_ARITH_H_
#define _CPP_AP_INT_ARITH_H_
#include <stdio.h>
#include "ap_int.h"
typedef ap_int<6> dinA_t;
typedef ap_int<12> dinB_t;
typedef ap_int<22> dinC_t;
typedef ap_int<33> dinD_t;
typedef ap_int<18> dout1_t;
typedef ap_uint<13> dout2_t;
typedef ap_int<22> dout3_t;
typedef ap_int<6> dout4_t;
void cpp_ap_int_arith (dinA_t inA, dinB_t inB,
dinC_t inC, dinD_t inD, dout1_t *out1, dout2_t *out2,
dout3_t *out3, dout4_t *out4);
#endif
```

The default maximum width allowed is 1024 bits. You can override this default by defining the macro `AP_INT_MAX_W` with a positive integer value less than or equal to 4096 before inclusion of the `ap_int.h` header file.

```
#define AP_INT_MAX_W 4096 // Must be defined before next line
#include "ap_int.h"
ap_int<4096> very_wide_var;
```

One disadvantage of AP data types is that arrays are not automatically initialized with a value of 0. You must manually initialize the array if desired.

If C++ Arbitrary Precision Integer Types are synthesized, it results in a design that is functionally identical to Standard Types. To allow assignment of values wider than 64-bits, the `ap_[u]int<>` classes provide constructors that allow initialization from a string of arbitrary length. To avoid unexpected behavior during co-simulation, do not initialize `ap_uint<N> a ={0}`.

```
ap_uint<96> wide_var("76543210fedcba9876543210", 16);
wide_var = ap_int<96>("0123456789abcdef01234567", 16);
```

High Level Synthesis

Base FIR Architecture

```
#define N 11
#include "ap_int.h"

typedef int coef_t;
typedef int data_t;
typedef int acc_t;
void fir(data_t *y, data_t x) {
    coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static data_t shift_reg[N];
    acc_t acc;
    int i;
    acc = 0;
Shift_Accum_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;
}
```

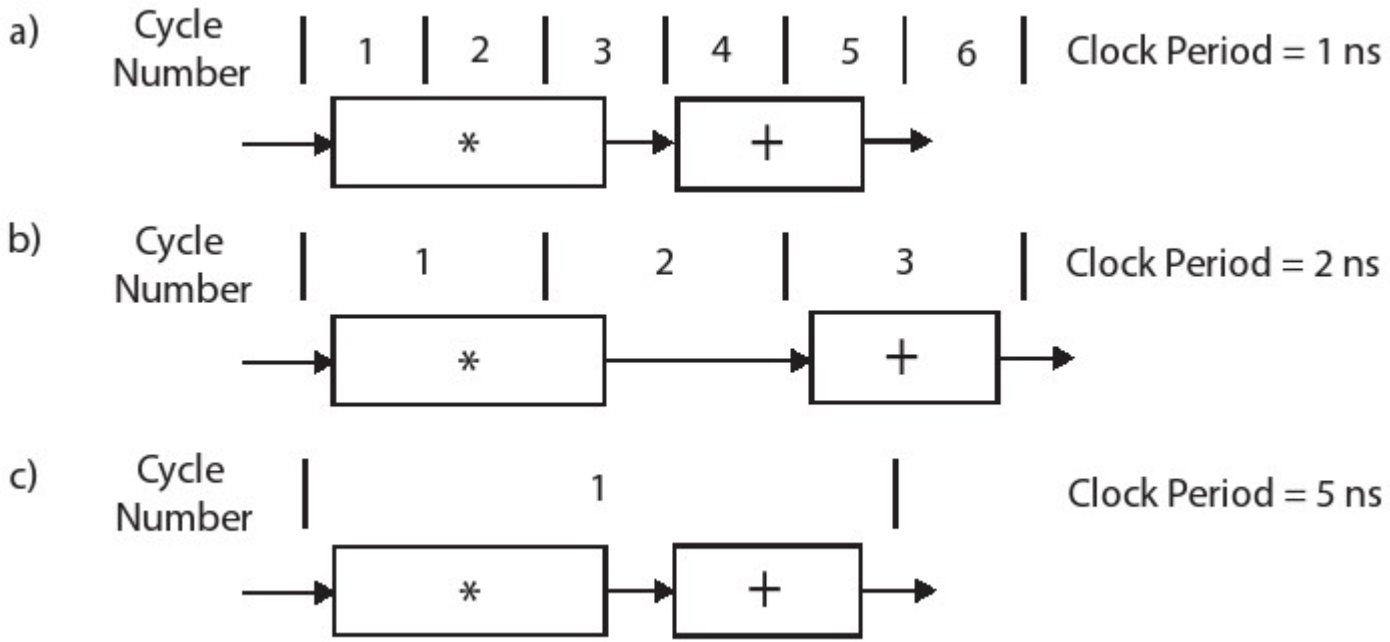
The label `Shift_Accum_Loop`: is not necessary. However it can be useful for debugging. The Vitis HLS tool adds these labels into the views of the code.

Calculating Performance

It is possible to specify a target frequency to the Vitis HLS tool using the `create_clock` tcl command. For example, the command `create_clock -period 5` directs the tool to target a clock period of 5 ns and equivalently a clock frequency of 200 MHz. Vitis HLS deals with clock frequency estimates including some margin to account for the fact that there is some error in the estimate. The goal of this margin is to ensure enough timing slack in the design that the generated RTL can be successfully placed and routed. This margin can be directly controlled using the `set_clock_uncertainty` TCL command.

High Level Synthesis

Operation Chaining



Code Hoisting

```
Shift_Accum_Loop:
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;
```

Removing the conditional statement from the `for` loop creates a more efficient hardware implementation.

Loop Fission

```
TDL:                                    Tapped Delay Line
for (i = N - 1; i > 0; i--) {
    shift_reg[i] = shift_reg[i - 1];
}
shift_reg[0] = x;

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

Loop fission alone often does not provide a more efficient hardware implementation. However, it allows each of the loops to be optimized independently, which could lead to better results than optimizing the single, original `for` loop.

The reverse is also true; merging two (or more) `for` loops into one `for` loop may yield the best results. This is highly dependent upon the application, which is true for most optimizations.

High Level Synthesis

Loop Unrolling

```
TDL:
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;
```

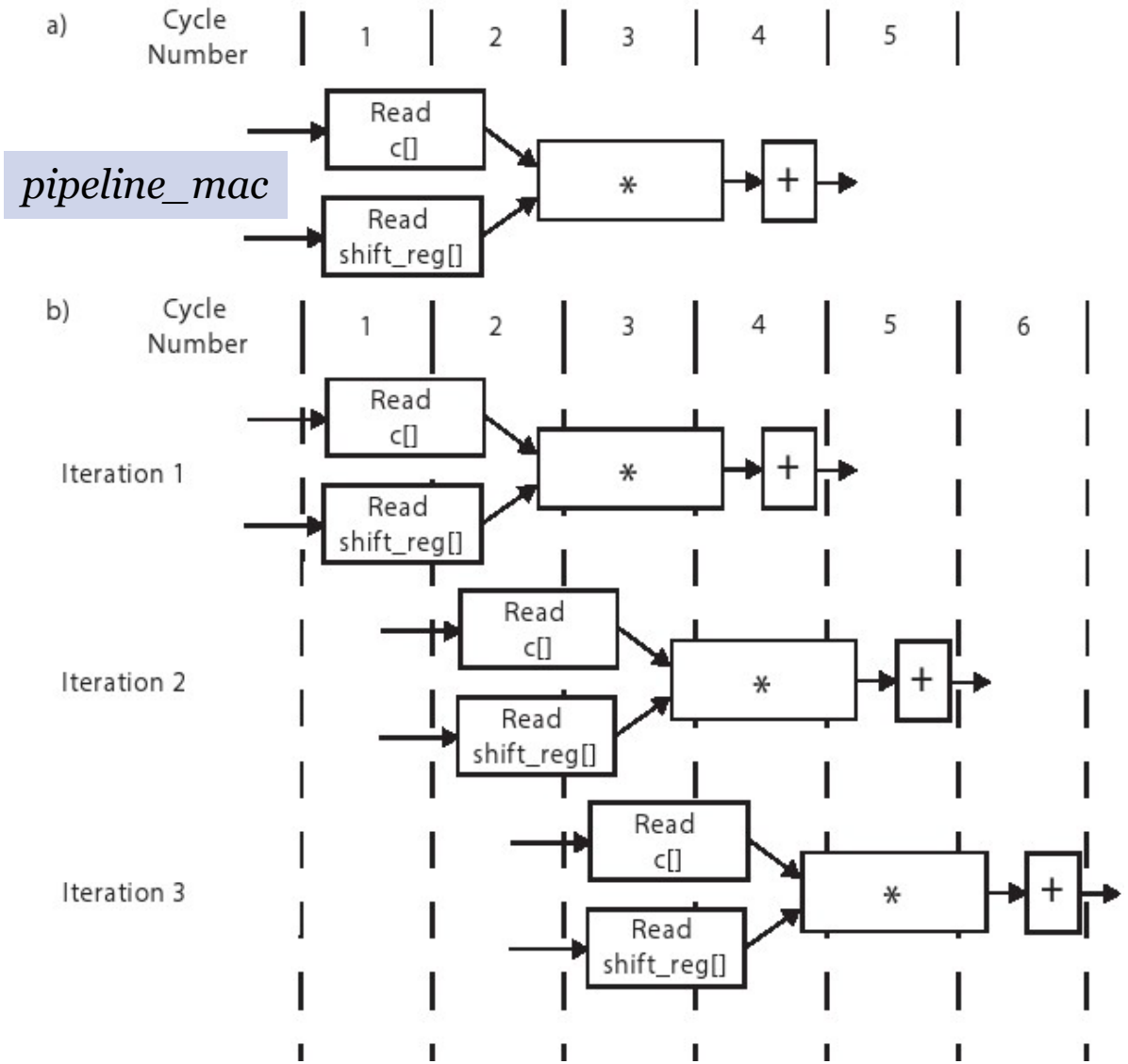
Loop unrolling replicates the body of the loop by some number of times (called the factor). Assume that we store the `shift_reg` array in one BRAM that has 2 read ports and 2 write port, we can perform two read operations and two write operations from the `shift_reg` array in the same cycle. You can tell the Vitis HLS tool to put all the values in the `shift_reg` array into registers using the directive `#pragma HLS array_partition variable=shift_reg complete`.

A user can tell the Vitis HLS tool to automatically unroll the loop using the unroll directive. To automatically perform the unrolling, we should put the directive `#pragma HLS unroll factor=2` into the body of the code, right after the `for`-loop header. By specifying the optional argument `skip_exit_check` in that directive, the Vitis HLS tool will not add the final `for` loop to check for partial iterations.

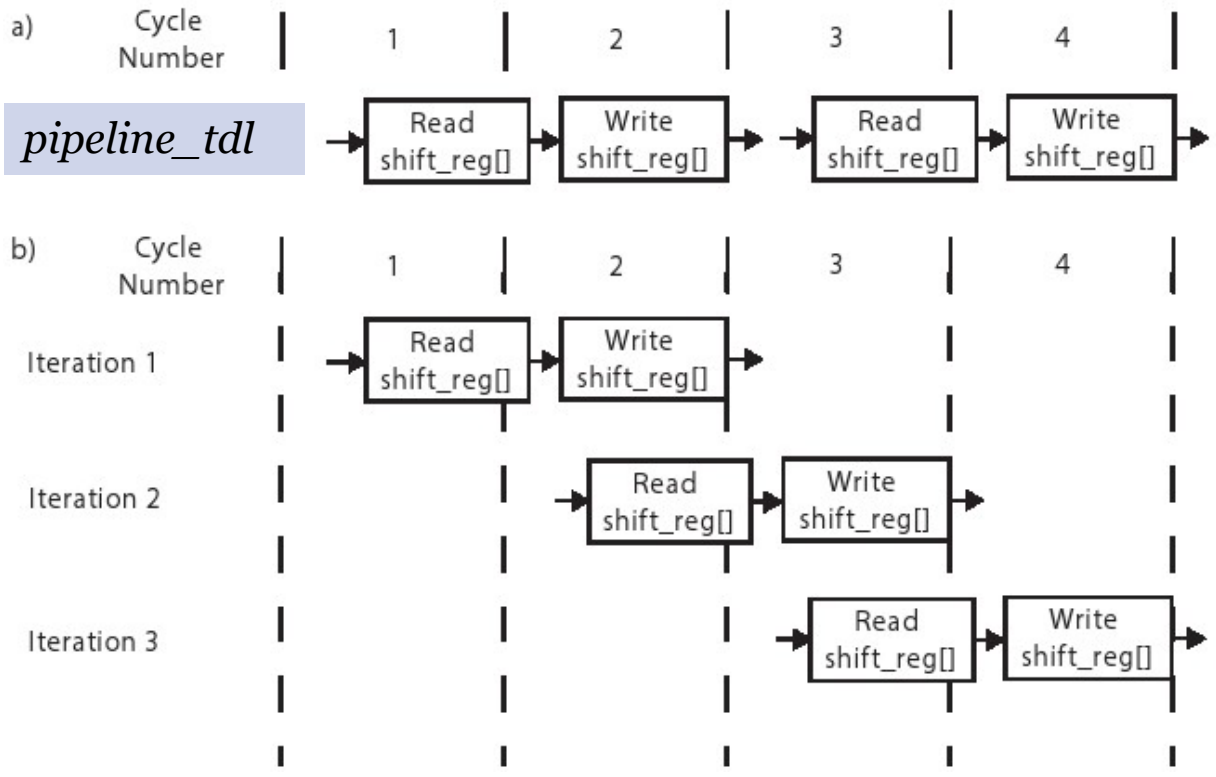
```
acc = 0;
MAC:
for (i = N - 1; i >= 3; i -= 4) {
    acc += shift_reg[i] * c[i] +
    shift_reg[i - 1] * c[i - 1] +
    shift_reg[i - 2] * c[i - 2] +
    shift_reg[i - 3] * c[i - 3];
}
for (; i >= 0; i--) {
    acc += shift_reg[i] * c[i];
}
```

High Level Synthesis

Loop Pipelining



Loop unrolling replicates the body of the loop by some number of times (called the factor). Assume that we store the `shift_reg` array in one BRAM that has 2 read ports and 2 write port, we can perform two read operations and two write operations from the `shift_reg` array in the same cycle. You can tell the Vitis HLS tool to put all the values in the `shift_reg`

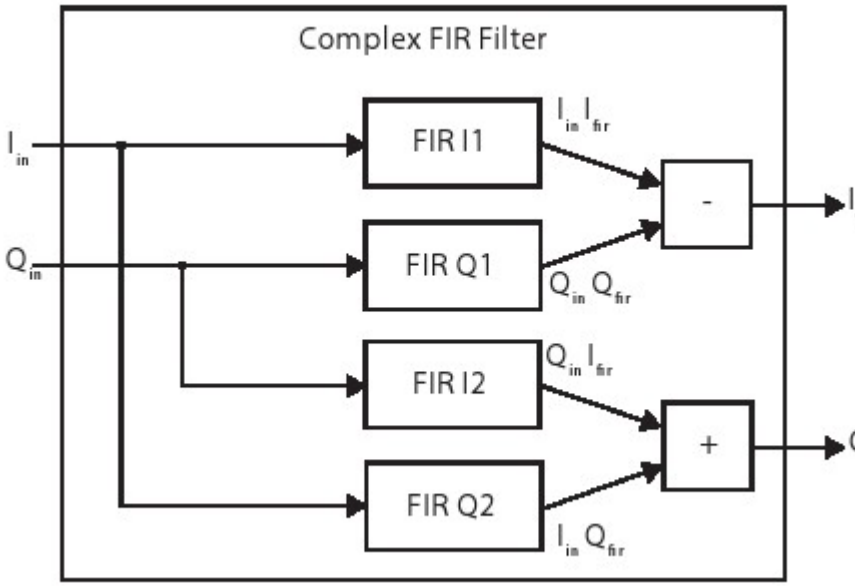


High Level Synthesis

Bitwidth Optimization

`coef t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};`
The maximum absolute value for any of these 11 entries is 500, which requires $\lceil \log 2\ 500 \rceil = 9$ bits.
Since we need negative numbers, we add an additional bit. Thus, `coef_t` can be declared as `ap_int<10>`.
Consider the operation `a = b + c` where `ap_uint<10> b` and `ap_uint<10> c`. The data type for `a` is `ap_uint<z>` where $z = \max(x, y) + 1$.
The value the bitwidth `z` given the bitwidths `x` and `y` (i.e., `ap_int<z> a`, `ap_int<x> b`, `ap_int<y> c`) for the operation `a = b * c` is $z = x + y$.

$$(I_{in} + jQ_{in})(I_{fir} + jQ_{fir}) = (I_{in}I_{fir} - Q_{in}Q_{fir}) + j(Q_{in}I_{fir} + I_{in}Q_{fir})$$



Complex FIR Filter

`firI1`, `firQ1`, `firI2`, and `firQ2` have the exact same code; can we call the same function 4 times?

?

`static` keyword used within the `fir` function for the `shift_reg`

```
typedef int data_t;
void firI1(data_t *y, data_t x);
void firQ1(data_t *y, data_t x);
void firI2(data_t *y, data_t x);
void firQ2(data_t *y, data_t x);
void complexFIR(data_t Iin, data_t Qin,
data_t *Iout, data_t *Qout) {
    data_t IinIfir,QinQfir,QinIfir,IinQfir;
    firI1(&IinIfir,Iin);
    firQ1(&QinQfir,Qin);
    firI2(&QinIfir,Qin);
    firQ2(&IinQfir,Iin);
    *Iout = IinIfir + QinQfir;
    *Qout = QinIfir - IinQfir;
}
```

High Level Synthesis

Bitwidth Optimization

The Vitis HLS tool does not optimize across function boundaries, i.e., each `fir` function synthesized independently, and treated as a black box in the `complexFIR` function. You can use the `inline` directive if you want the Vitis HLS tool to co-optimize a particular function within its parent function.

1. can increase the potential for benefits in performance and area
2. creates a large amount of code that the tool must synthesize, which may take a long time, even fail to synthesize, or may result in a non-optimal design
3. eliminates any overhead associated with performing the function call.

Therefore, use the inline directive carefully. Also note that the Vitis HLS tool may choose to inline functions on its own. These are typically functions with a small amount of code. You can force the tool to keep the function hierarchy by `inline off`. The `inline` directive also has a `recursive` argument that recursively adds the code into the parent functions from every child function.

```
float mul(int x, int y) {
    return x * y;
}
float top_function(float a, float b, float c, float d) {
    return mul(a, b) + mul(c, d) + mul(b, c) + mul(a, d);
}
float inlined_top_function(float a, float b, float c, float d) {
    return a * b + c * d + b * c + a * d;
}
```

Reference

① Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. “Parallel Programming for FPGAs.” Sept. 6, 2022.