Messages of the day

- Labs begin in earnest today – handin due 5pm next Monday
  - Submit a PDF via the link on the course website
  - This is quite a long lab; the lab for Week 3, which should be released by next Monday, is considerably shorter
- Please organize yourselves into project teams this week – please email me the membership and a proposed topic
  - Start meeting to develop a plan of attack
- Announcement on seminar topic registration should arrive by end of week – please log into Moodle over the coming week and register for the available topic that most interests you

Ch 2. FIR filters

UNSW
SYDNEY

Goal of this chapter

- Provide a basic understanding of the process of taking an algorithm and creating a good hardware design using high-level synthesis
- Provide an example of how we will study the text

Study method followed in this course:

- I assume you have read the selected materials before the lecture
- During the lecture, I will review and we will discuss the HLS concepts presented (but not generally the mathematical background)

1D Convolution

Given the impulse response of a finite impulse response (FIR) filter, we can compute the output signal for any input signal through the process of convolution.

The convolution of an N-tap FIR filter with coefficients $h[]$ by an input signal $x[]$ is described by the general difference equation:

$$y[i] = \sum_{j=0}^{N-1} h[j] \cdot x[i-j]$$

- In general, this expression requires N multiplications and N-1 additions to compute a single output value

11-tap FIR code example

```
#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;
}
```

This code is written as a streaming function – it receives one sample at a time and must therefore store the previous samples.

The `for` loop is doing two things – it performs a multiply and accumulate (MAC) operation on the input samples as well as shifting values through `shift_reg`, which acts as a FIFO.

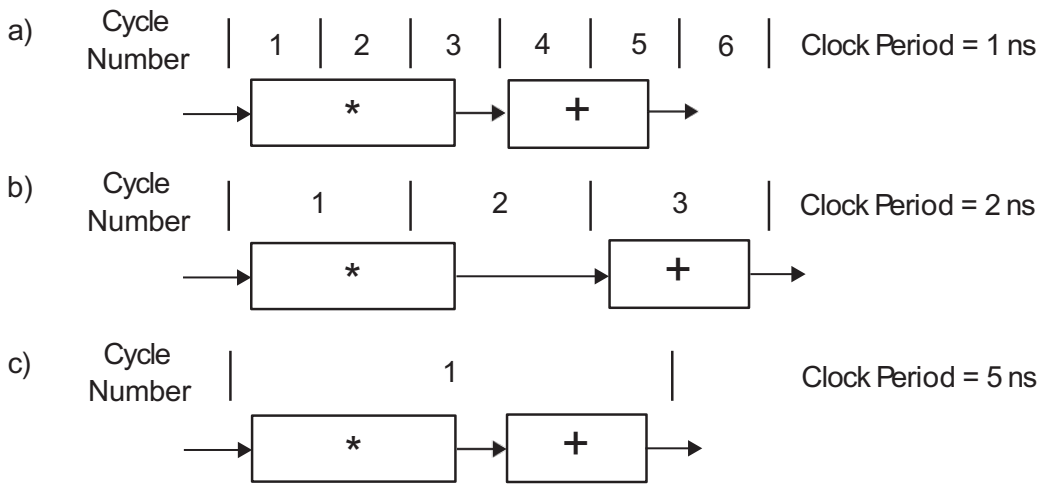
Calculating performance

When deriving the performance of a design, it is important to carefully state the metric to be used. It is also important to compare apples with apples.

HLS tools talk about designs in terms of number of cycles and the clock frequency. Together, these yield the amount of time in seconds to perform some operation.

- Vivado HLS attempts to optimize both #CC and ClkFreq
- It is possible to specify a target frequency to Vivado HLS, which primarily affects how much **operation chaining** is performed by the tool. After generating RTL, Vivado HLS provides an initial timing estimate relative to this clock target. However, some uncertainty in this timing remains, which is only resolved once the design is placed and routed
- Increasing the clock target frequency is not necessarily the best way to achieve peak performance; lower frequencies provide more leeway for multiple dependent operations to be combined in the one clock cycle

Operation chaining



Assuming a multiply op takes 3 ns and an add op takes 2 ns, the performance of a MAC operation changes depending upon the target clock period

Operation chaining is an important optimization that Vivado HLS performs in order to optimize the final design.

Unfortunately, there is no good rule to pick the optimal target clock frequency.

It is often best to explore just a small range of target clock periods and focus on optimizing other aspects of the design.

With Vivado HLS, start your exploration with a clock period of 10ns on the Zynq xc7z020 device

Code hoisting

```
  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];
    }
  }
```

The `for` loop can be restructured:

```
  acc = 0;
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;
```

In our FIR function, not only is the `if` statement inside the loop inefficient, it only serves a purpose when `i==0`.

The statements within the `if` branch can be **hoisted** out of the loop, with the loop bounds being adjusted.

The loop can then more readily be unrolled and pipelined since it's control is less complicated.

Loop fission

```
  acc = 0;
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;
```

Splitting the `for` loop in two:

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

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

We are doing two fundamentally different ops within the `for` loop – shifting data through the `shift_reg` array and multiplying and accumulating samples and coefficients.

Loop fission implements each of them in their own loop, thereby allowing us to optimize each differently.

Sometimes restructuring the code in the opposite sense, by merging loops, can result in better performance.

Loop unrolling

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

Manually unrolling the loop by a factor of 2:

```
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;
```

We can unroll the loop as above automatically by inserting the directive: `#pragma HLS unroll factor=2` into the body of the loop just below the `for` loop header.

By default, Vivado HLS synthesizes `for` loops to execute sequentially. It creates a data path that executes sequentially for each iteration of the loop, which is area efficient but limits parallelism.

Loop unrolling replicates the body of the loop by some number of times (called the factor) and reduces the number of iterations of the loop by the same factor.

Loop unrolling increases the performance provided that some (or all) of the statements in the loop can be executed in parallel. **Note that in this case we need to be able to perform two reads ops and two writes ops from the `shift_reg` array in the same cycle.**

Unrolling the MAC loop

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

Manually unrolled by a factor of 4:

```
  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];
  }
```

In the unmodified MAC loop code, the sample and coeff value loads are independent across iterations, but the additions introduce a RAW dependency across iterations.

This can be eliminated as shown.

The loop can be unrolled automatically by inserting `#pragma HLS unroll` into the code after the for loop header.

- Unroll the loop completely by not specifying a `factor` argument. For this to work, the bounds of the loop must be statically determined (known at compile time)
- An optional `skip_exit_check` argument can be added to the directive to avoid generating the final `for` loop.

⚠ Complete loop unrolling

Complete loop unrolling exposes a maximal amount of parallelism at the cost of creating an implementation that requires a significant amount of resources.

Thus, it is ok to completely unroll "smaller" `for` loops. But completely unrolling a loop with a large number of iterations (e.g., one that iterates a million times) is typically infeasible.

- Often Vivado HLS will run for a very long time and frequently fail to complete after hours of synthesis

As a novice, if your design does not synthesize in under 15 minutes, consider the effect of your optimizations - it is likely that you used some directive that significantly expanded the code, likely in an unintended way.

Loop unrolling

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

Manually unrolling the loop by a factor of 2:

```
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;
```

Exercise:

Manually unroll the TDL loop by a factor of three.

What changes are needed to the code that has been manually unrolled by a factor of 2?

Loop pipelining

By default, the Vivado HLS tool synthesizes `for` loops in a sequential manner, e.g., the `for` loop in the code to the right will perform each iteration of the loop one after another. That is, all of the statements in the second iteration are performed only when all of the statements of the first iteration are complete.

This happens even in cases where it is possible to perform statements from the iterations in parallel, or to start some statements from a later iteration before the statements in the former iteration are completed (*which does not happen unless the designer specifically states that it should).*

```
  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;
```

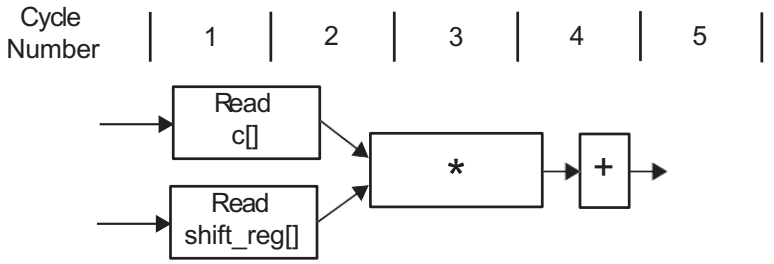
Loop pipelining allows for multiple iterations of the loop to execute concurrently.

Loop pipelining

Consider the MAC loop body from before:

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

One iteration of the loop could be scheduled as shown below:



Notes:

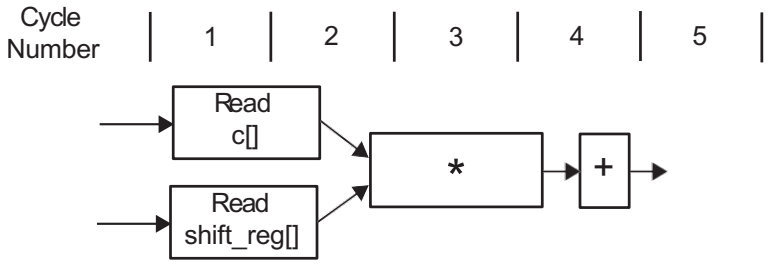
- The array reads take 2 cycles (one cycle to present the address to memory, and another to retrieve the data) and can be performed in parallel as they have no dependencies
- The multiply is assumed to take three cycles, starting in cycle 2
- The addition can be chained with the multiply op to complete in the 4th cycle
- `acc` is updated at the start of the 5th cycle

Loop performance metrics

Consider the MAC loop body from before:

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

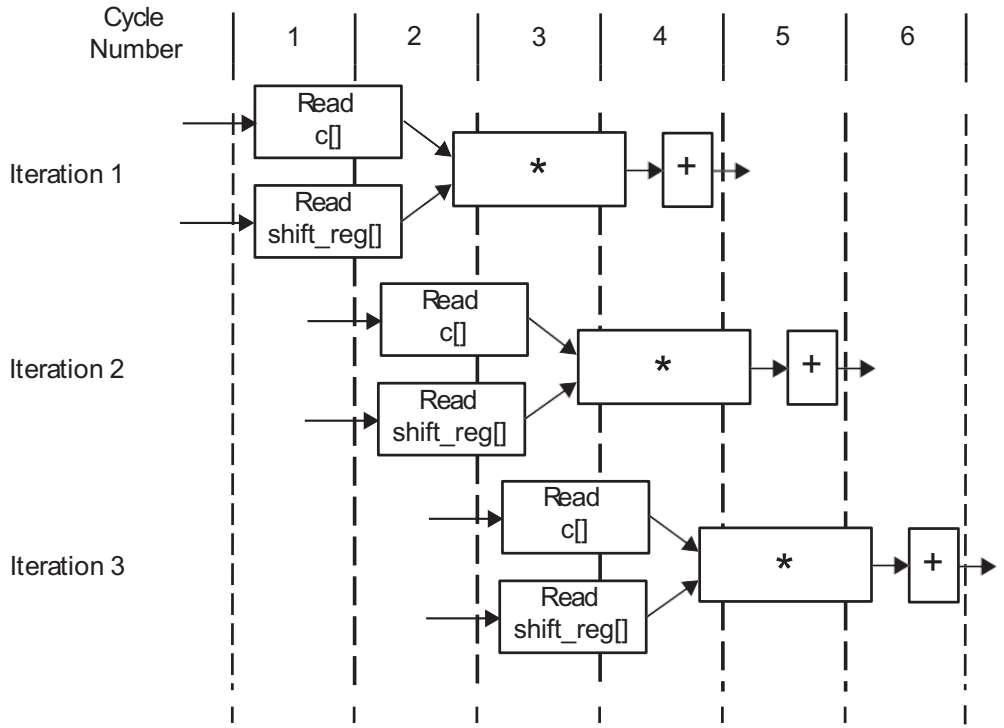
One iteration of the loop could be scheduled as shown below:



- The ***iteration latency*** is the number of cycles that it takes to perform one iteration of the loop body (in this case, 4 cycles)
- The ***loop latency*** is the number of cycles required to complete the entire execution of the loop (in this case, 11 x 4 = 44 cycles, **plus one** to determine the loop is finished, or equivalently, to write the result of the last iteration).
- Vivado HLS defines the loop latency as the cycle in which the last output data are ready, and therefore does not include the last write-back/store cycle.

Loop pipelining

Loop pipelining is an optimization that overlaps multiple iterations of a for loop
In this case, three concurrent iterations of the MAC loop body are shown



- In our FIR code, the final iteration will start in cycle 11 and complete in cycle 14. Accordingly, the loop latency is 14
- The loop ***initiation interval (II)*** is another important performance metric and is defined as the number of cycles until the next iteration of the loop can start (here the II=1)
- The desired II can be specified using the pipeline directive, e.g.

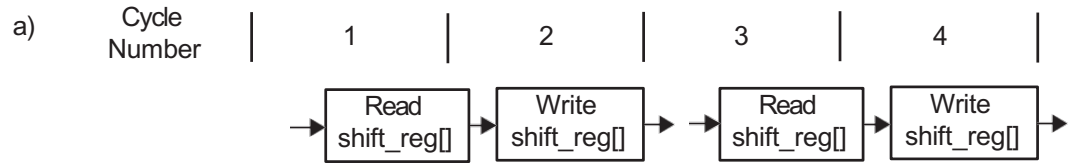
```
#pragma HLS pipeline II=2
```

- The tool will attempt to achieve the specified II, or to minimize the II when it is not specified.
- Attempts to create a design with a specified II may not succeed due to resource constraints or dependencies within the code

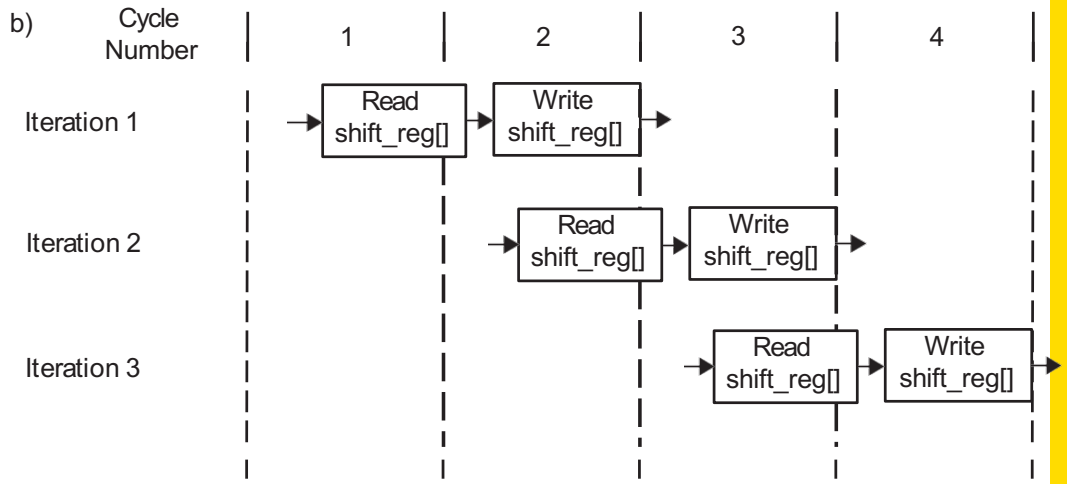
Pipelining the TDL loop

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

The loop body takes two cycles to read an element and the write can be chained in the 2nd cycle. Thus two sequential iterations might be scheduled as



In order to pipeline the loop with an II=1, we need to use memory that has two ports: one to perform a read, and the other to complete the write in the same cycle. With single port memory, HLS would be forced to use II=2 to pipeline the loop:



UNSW SYDNEY

The `resource` directive

- The `resource` directive allows the user to force the Vivado HLS tool to map an operation to a particular type of hardware core
- For example, to map the `shift_reg` array to a single port BRAM core, the directive `#pragma HLS resource variable=shift_reg core=RAM_1P` could be inserted into the code near the declaration of `shift_reg`
- Variables can also be mapped to particular types of hardware, e.g. `#pragma HLS resource variable=a core=AddSub.DSP` maps the add operation in the expression `a = b + c;` to a DSP block, rather than using LUTs to perform the addition

- In general, it is best to let Vivado HLS decide which resources to use, and if these aren't what you want, then use directives to override these choices

Bitwidth optimization

- C offers many different data types e.g. int, long, char, float, double
- In software, reducing the data type size can save considerable space
- This is also true for hardware implementations, where, additionally
  - smaller operations may require fewer clock cycles to execute, or
  - allow more instructions to be executed in parallel
- HLS generates a custom data path matched to the specific data type The op `a = b * c` will have different latency and resource usage depending upon the data type
  - If all veriables are 32 bits wide, more primitive Boolean operations (and consequently more resources) need to be performed than if they were 8 bits wide
  - Additionally, more complex logic typically requires more pipelining to achieve the same frequency

Bitwidth optimization

- While C data types (and their software implementations) vary in width by powers of 2, minimal hardware should be customized to the minimum width necessary e.g. 10, 12 or 14 bits, say
- Vivado HLS provides arbitrary precision data types through a couple of classes:
  - Unsigned: `ap_uint<width>`, and
  - Signed: `ap_int<width>`, where `width` is in [1,1024].
- To use these data types, you must use C++ (filename with `.cpp` extension) and `#include "ap_int.h"`
- To estimate appropriate data widths, remember to:
  - account for the largest magnitude and sign of the data you need to store,
  - sum the widths of variables that are multiplied together, and
  - increment the width of the widest variable when two variables are added together.
- Note that storing a wider data type value to a narrower variable results in the *most significant bits* of the wider value being dropped

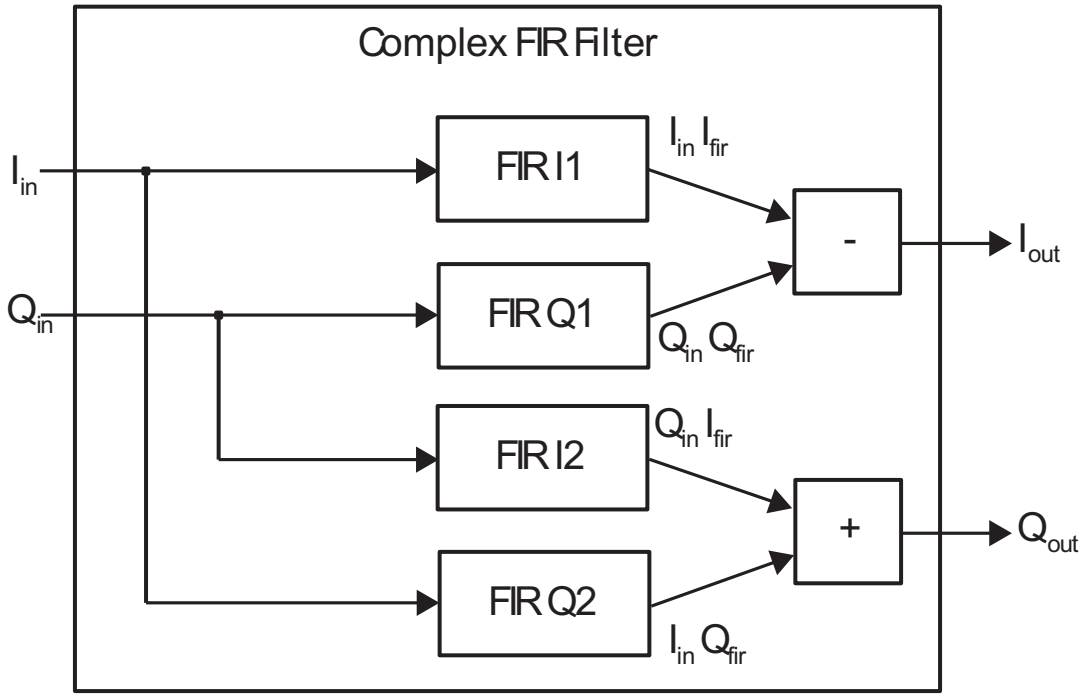
Bitwidth optimization

```
#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;
}
```

Exercises

1. What is an appropriate data type for the coeff array, `c`?
2. What is an appropriate data type for the loop control variable `i`?
3. What about for variables of type `data_t`?
4. And how about for `acc`?

Complex FIR filter




```
typedef int data_t;
void firI1(data_t *y, data_t x); // Re(x)*Re(coeff)
void firQ1(data_t *y, data_t x); // Im(x)*Im(coeff)
void firI2(data_t *y, data_t x); // Im(x)*Re(coeff)
void firQ2(data_t *y, data_t x); // Re(x)*Im(coeff)

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;
}
```

This comes about by virtue of the fact that for complex numbers x = a + *j*b & y = c + *j*d,
the product x*y = ac – bd + *j*(ad +bc),
whereby FIRI1 computes ac, FIRQ1→bd, FIRI2→bc and FIRQ2→ad

Complex FIR filter

- The fns contain the same code but need to be replicated due to the different `static` data and coeffs in each
- The fn calls act as interfaces.
- Vivado HLS does not optimize across fn boundaries
  - Use the inline directive if you want the Vivado HLS tool to co-optimize a particular fn within its parent fn
  - While this can increase the potential for benefits in performance and area, it can also create a large amount of code that the tool must synthesize, which may therefore fail to synthesize or results in non-optimal code

```
typedef int data_t;
void firI1(data_t *y, data_t x); // Re(x)*Re(coeff)
void firQ1(data_t *y, data_t x); // Im(x)*Im(coeff)
void firI2(data_t *y, data_t x); // Im(x)*Re(coeff)
void firQ2(data_t *y, data_t x); // Re(x)*Im(coeff)

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;
}
```

Concluding remarks

- The first step in the HLS transformation process is to understand the algorithm that is to be implemented, otherwise your ability to write (synthesizable and) efficient HLS code is likely to be limited
- Creating an optimal architecture requires a basic understanding of how the HLS tool works
  - It isn't necessary to understand the exact algorithms used to schedule, bind and allocate resources, but having a general idea helps you write code that maps well to hardware
- We discussed several fundamental HLS optimizations including their limitations – more examples of these and other techniques will be examined over the next 3 weeks

- The labs are intended to complement the lecture material. This week's lab asks you to carry out some of the exercises from this chapter of the book and report your observations by next Monday afternoon.