Messages of the week

- Projects
  - Time to meet and discuss project plans with your team
  - Let me know ASAP if your team has a preference for which lab session next week to present during – I will allocate date if no preference received by Monday
- Seminars
  - Meet with group, read and discuss papers suggested for your topic
- Labs
  - Discussion

Ch 4. Discrete Fourier Transform

Discrete Fourier Transform

- DFT is a fundamental signal processing technique
- Transforms a discrete (fixed number of samples) signal in the time domain to a discrete signal in the frequency domain i.e. as a sum of sinusoids
  - This allows for advanced filtering and modulation techniques as well as fast large-integer and polynomial mutiplication
- At its core, DFT performs a matrix-vector multiplication, where the matrix is a fixed set of coefficients

- I assume you have read the mathematical introduction to DFTs at the start of Ch. 4 as my review will be very brief

DFT

- The DFT converts a finite number of equally spaced samples in time into a finite number of complex sinusoids
- Given N **real-valued** samples of signal g[] in the time domain, $N/2+1$ complex signal values, G[], are computed in the frequency domain using the expression $G = S \cdot g$,

where $S = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & s & s^2 & \cdots & s^{N-1} \\ 1 & s^2 & s^4 & \cdots & s^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & s^{N-1} & s^{2(N-1)} & \cdots & s^{(N-1)(N-1)} \end{bmatrix}$ and $s = e^{\frac{-j2\pi}{N}}$

Hence, $G[k] = \sum_{n=0}^{N-1} g[n] s^{kn}$ for $k = 0, \cdots, N-1$

A visualization of DFT coefficients (8 point DFT)

$$\begin{bmatrix} G[0] \\ G[1] \\ G[2] \\ G[3] \\ G[4] \\ G[5] \\ G[6] \\ G[7] \end{bmatrix} = \begin{bmatrix} \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow \\ \rightarrow & \searrow & \downarrow & \swarrow & \leftarrow & \nwarrow & \uparrow & \nearrow \\ \rightarrow & \downarrow & \leftarrow & \uparrow & \rightarrow & \downarrow & \leftarrow & \uparrow \\ \rightarrow & \swarrow & \uparrow & \searrow & \leftarrow & \nearrow & \downarrow & \nwarrow \\ \rightarrow & \leftarrow & \rightarrow & \leftarrow & \rightarrow & \leftarrow & \rightarrow & \leftarrow \\ \rightarrow & \nwarrow & \downarrow & \nearrow & \leftarrow & \searrow & \uparrow & \swarrow \\ \rightarrow & \uparrow & \leftarrow & \downarrow & \rightarrow & \uparrow & \leftarrow & \downarrow \\ \rightarrow & \nearrow & \uparrow & \nwarrow & \leftarrow & \swarrow & \downarrow & \searrow \end{bmatrix} \begin{bmatrix} g[0] \\ g[1] \\ g[2] \\ g[3] \\ g[4] \\ g[5] \\ g[6] \\ g[7] \end{bmatrix}$$

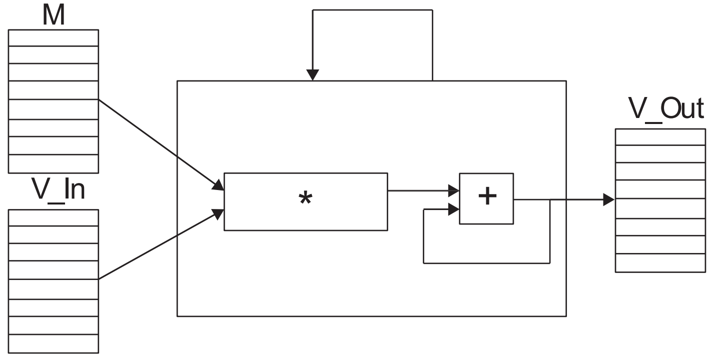
- Note the abundance of symmetry in matrix S

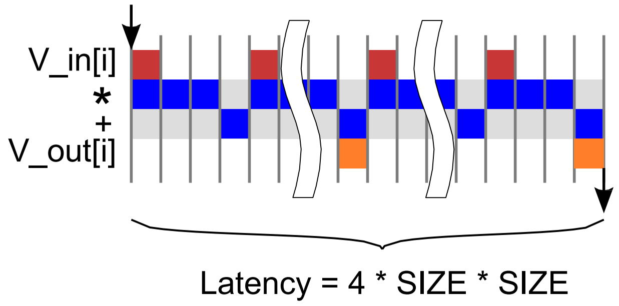
DFT as matrix-vector multiplication

- DFT can be computed as a matrix-vector multiplication
- Code to the right provides a good baseline
- The algorithm comprises a nested loop that, while simple, offers many design choices
  - Memory organization is one important factor – use wires, registers, RAM, FIFO to store variables? Each comes with its own area/performance tradeoff.
  - Amount of parallelism to exploit (when its available) is another, with obvious area/performance tradeoffs

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE],
    BaseType V_In[SIZE],
    BaseType V_Out[SIZE]) {

    int i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        BaseType sum = 0;
        dot_product_loop:
        for (j = 0; j < SIZE; j++) {
            sum += V_In[j] * M[i][j];
        }
        V_Out[i] = sum;
    }
}
```

Sequential architecture





When the MV_mult code is compiled without HLS directives, a sequential architecture with one multiplier and one adder is synthesized

- Logic to access the V_in, V_out and M arrays, stored in BRAMs, is produced
- The result is a design that does not use much area, but has a high task latency and task interval

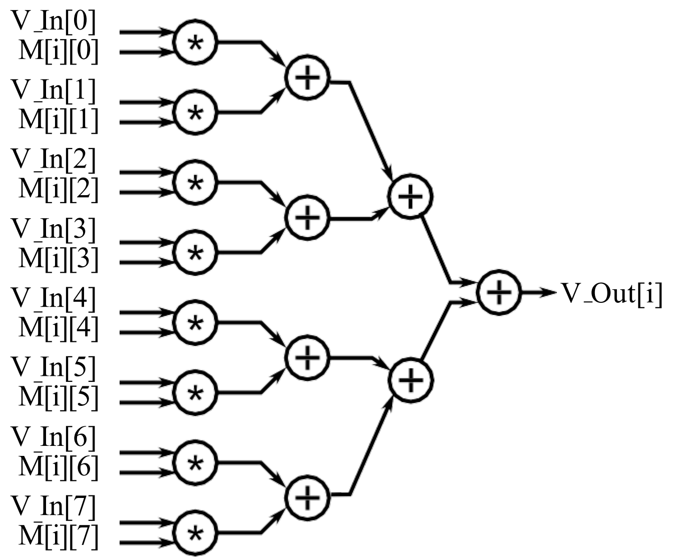
Manually unrolling the inner, `dot_product_loop`

- There is substantial opportunity to exploit parallelism – starting with the inner loop, which can be rewritten to eliminate the data dependency (in this case)
- It should now be clear that there is significant parallelism in the loop body – each multiplication could be performed in parallel and the additions could be done using an adder tree

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE],
    BaseType V_In[SIZE], BaseType V_Out[SIZE])
{

    BaseType i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        V_Out[i] = V_In[0] * M[i][0] +
                   V_In[1] * M[i][1] +
                   V_In[2] * M[i][2] +
                   V_In[3] * M[i][3] +
                   V_In[4] * M[i][4] +
                   V_In[5] * M[i][5] +
                   V_In[6] * M[i][6] +
                   V_In[7] * M[i][7];
    }
}
```

Manually unrolling the inner, `dot_product_loop`

- There is substantial opportunity to exploit parallelism – starting with the inner loop, which can be rewritten to eliminate the data dependency (in this case)
- It should now be clear that there is significant parallelism in the loop body – each multiplication could be performed in parallel and the additions could be done using an adder tree
- A dataflow graph for this computation is depicted on the right



Manually unrolling the inner, `dot_product_loop`*

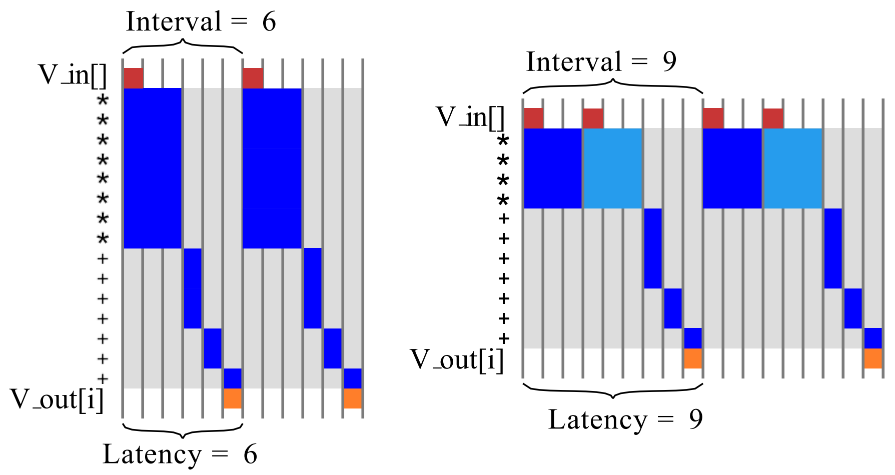
- There is substantial opportunity to exploit parallelism – starting with the inner loop, which can be rewritten to eliminate the data dependency (in this case)
- It should now be clear that there is significant parallelism in the loop body – each multiplication could be performed in parallel and the additions could be done using an adder tree

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE],
    BaseType V_In[SIZE], BaseType V_Out[SIZE])
{

    BaseType i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        V_Out[i] = V_In[0] * M[i][0] +
                   V_In[1] * M[i][1] +
                   V_In[2] * M[i][2] +
                   V_In[3] * M[i][3] +
                   V_In[4] * M[i][4] +
                   V_In[5] * M[i][5] +
                   V_In[6] * M[i][6] +
                   V_In[7] * M[i][7];
    }
}
```

* How would you unroll the loop automatically?

Sequential implementation of the unrolled inner loop

- Assuming a multiplication operation has a latency of 3 cycles, and an addition op has a latency of 1 cycle then the mults are completed after 3 cycles, and the summation takes another $\log_2 8 = 3$ cycles
- Hence the body of the `data_loop` now has a latency of 6 cycles and requires 8 multipliers and 7 adders
- We could reuse the adders, but these are not typically shared, since they cost no more in LUTs than the MUXes that are then needed to share them
- One could also get away with using less multipliers for slightly higher latency

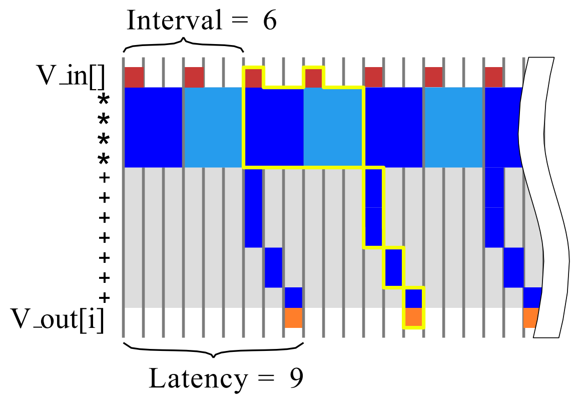


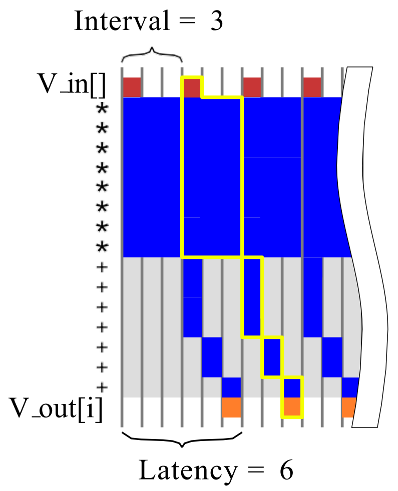
- Here the inner loop is unrolled but not pipelined, and therefore is executed sequentially – but note the periods of underutilization that lead to inefficiency

Pipelined execution of the inner loop

Observing that *each iteration of `data_loop` is independent*, why not execute them concurrently? I.e., start executing the next iteration of the loop while the previous execution is still processing…

- We could unroll `data_loop` (just as we unrolled `dot_product_loop`), but this would consume a lot of resources
- Alternatively, start each iteration as soon as possible using loop pipelining
- Loop pipelined behaviour of the design is shown on the right

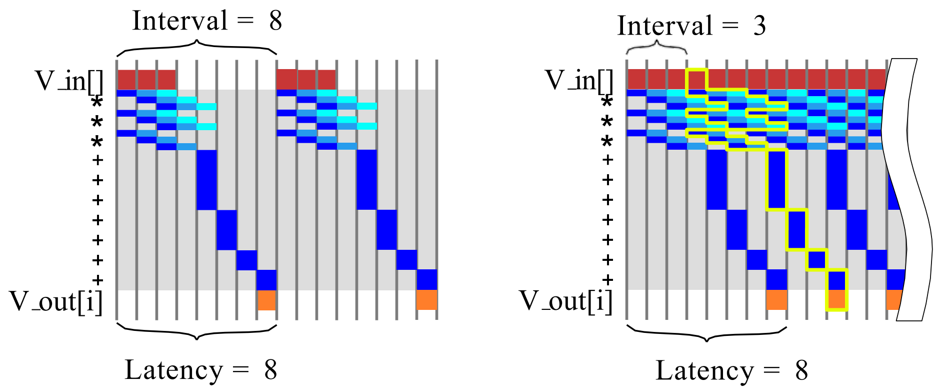




Pipelined operators e.g. multipliers

- Most functional units on modern FPGAs are fully pipelined with an interval of 1
- While a multiplier may have a latency of 3 cycles, it can have 3 different operations in flight simultaneously, with a new multiply op starting every cycle!
- We could thus reduce the latency of the inner loop while using fewer multipliers
- The examples on the right use 3 multipliers to perform the 8 multiplications contained in the inner loop

Note that pipelining is thus possible at the operator, loop and function level. Furthermore, pipelining at different levels is largely independent.



- Sequential iteration execution is shown here on the left; pipelined on the right
- Concurrent processing of multiplications is shown followed by the addition operations that are dependent upon the multiplication results

Storage tradeoffs

- So far we have assumed all the data we need is available each clock cycle, so let’s now take a look at the memory constraints on acceleration
- In contrast to processors, that have fixed memory architectures, FPGAs and HLS allow us to explore and leverage different memory structures
- While processors leverage caches to bring in and reuse memory from off-chip and network attached storage, FPGAs provide flip-flops distributed throughout the device that allow read-modify-write operations in a single cycle, and BRAMs that can store up to 4KB amounts of data, but with only one or two accesses per clock cycle
  - Note that the relatively small Zynq 7020 device that you are targeting in labs and projects has only 106,400 (≈13KB) FFs and 140 x 36kbit (≈560KB) BRAMs
- Limited FF storage makes it infeasible to store large matrices on-chip; on the other hand, using BRAMs instead severely limits the number of concurrent accesses possible

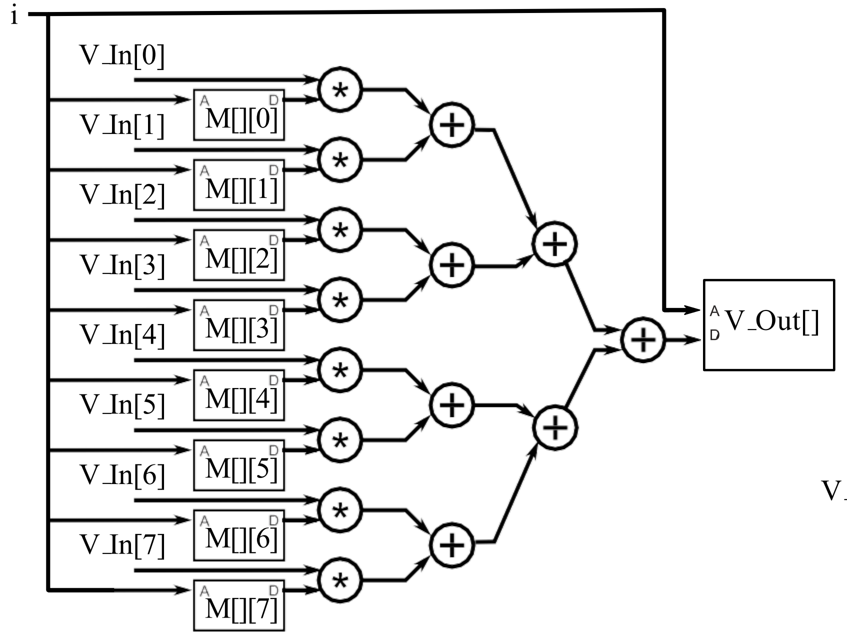
Array partitioning

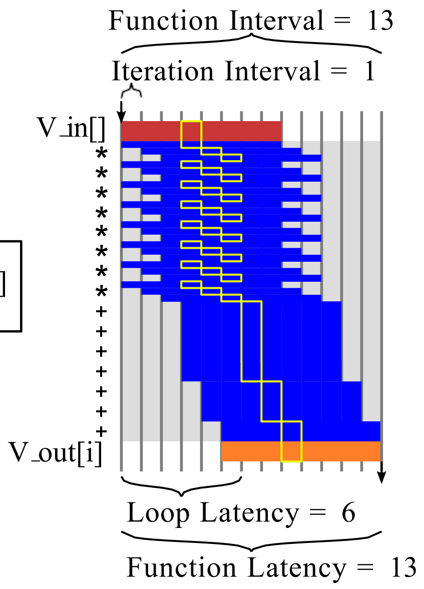
- In practice larger arrays need to be strategically divided into smaller BRAM memories, a process called *array partitioning*
- Smaller arrays can be partitioned completely into individual scalar variables and mapped into FFs
- Matching pipeline choices and array partitioning to maximize the efficiency of operator usage and memory usage is an important aspect of design space exploration in HLS
- Vivado HLS will perform some array partitioning automatically, but it is often necessary to guide the tool using the `#pragma HLS array_partition variable=XXX` directive with options to completely partition the variable into FFs or to split it in some way across BRAMs
  - For example, the array X containing [1 2 3 4 5 6 7 8 9] is split into sub arrays containing [1 2 3 4 5] and [6 7 8 9] across two BRAMs using the directive `#pragma HLS array_partition variable=X factor=2 block`, whereas using **`cyclic`** rather than **`block`** partitioning results in [1 3 5 7 9] and [2 4 6 8], which doubles data access when the array is to be processed sequentially

MV multiplication with array partitioning

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE],
    BaseType V_In[SIZE],
    BaseType V_Out[SIZE]) {

    #pragma HLS array_partition
        variable=M dim=2 complete
    #pragma HLS array_partition
        variable=V_In complete
    int i, j;

    data_loop:
    for (i = 0; i < SIZE; i++) {
        #pragma HLS pipeline II=1

        BaseType sum = 0;
        dot_product_loop:
        for (j = 0; j < SIZE; j++) {
            sum += V_In[j] * M[i][j];
        }
        V_Out[i] = sum;
    }
}
```

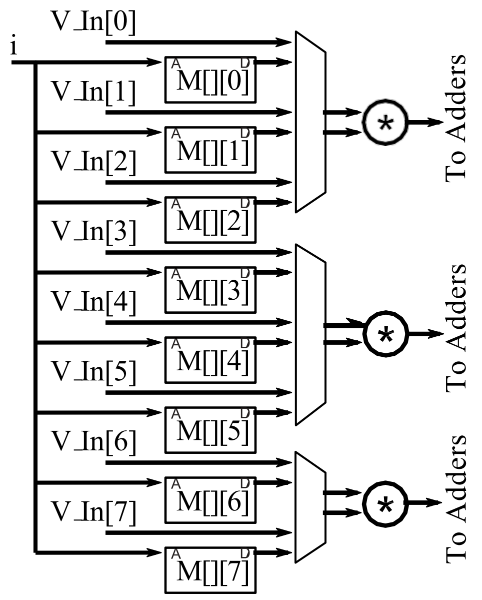
- A highly parallel implementation of the code can be achieved by adding just a few directives
- The inner `dot_product_loop` is automatically unrolled and every use of `j` is replaced by constants



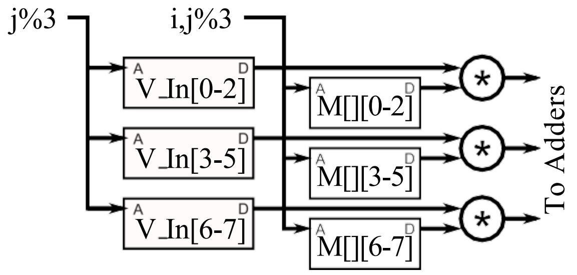


Architectures for II=3

- With complete array partitioning and II=3, muxes are needed to select the desired array entries each cycle



- Partitioning the arrays with `factor=3` avoids the need for muxes but involves the `j` loop variable in the array addressing



Partially unrolling & pipelining the inner loop

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE],
    BaseType V_In[SIZE], BaseType V_Out[SIZE]) {

    #pragma HLS array_partition
        variable=M dim=2 cyclic factor=2
    #pragma HLS array_partition
        variable=V_In cyclic factor=2

    int i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        BaseType sum = 0;
        dot_product_loop:
        for (j = 0; j < SIZE; j+=2) {
            #pragma HLS pipeline II=1

            sum += V_In[j] * M[i][j];
            sum += V_In[j+1] * M[i][j+1];
        }
        V_Out[i] = sum;
    }
}
```

- Here, the `array_partition` directives are not needed if the memory is dual ported
- But what happens if we unroll the inner loop more than twice?

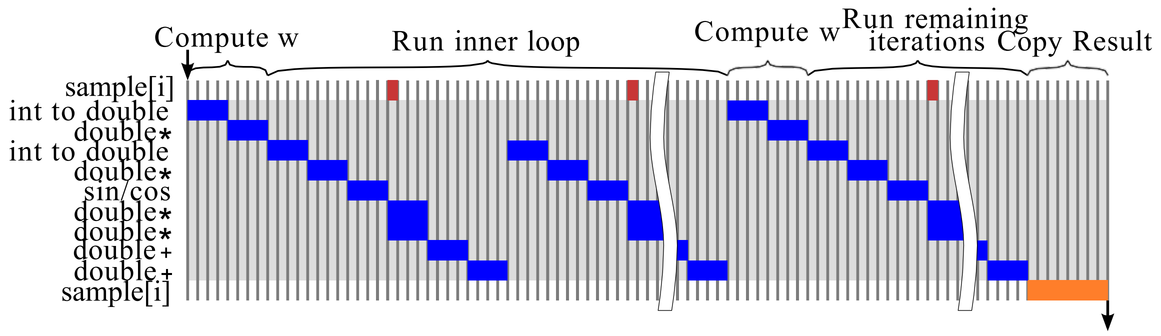
Baseline DFT code

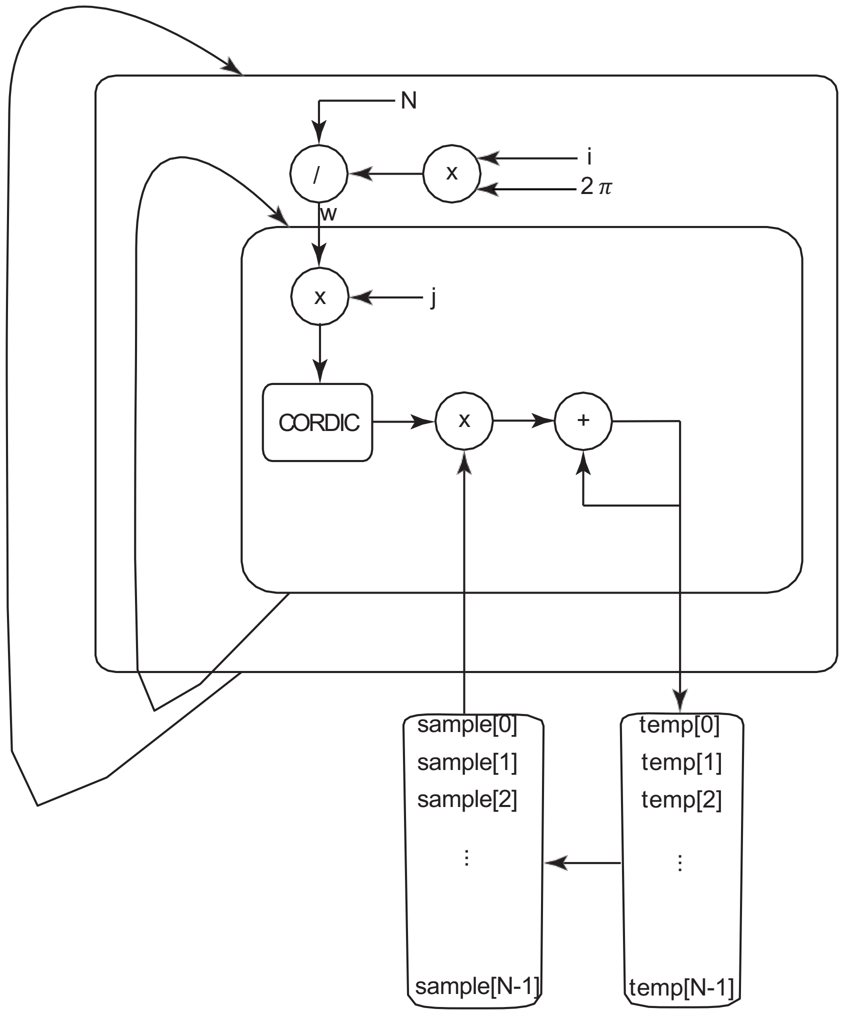
The code has to deal with additional complications:

1. Complex samples & complex mults/adds
2. The latency inherent in floating point operations
3. The scalability of storing a large S matrix

```
#include <math.h>          // for cos and sin fns
typedef double IN_TYPE;    // input signal data type
typedef double TEMP_TYPE;  // temp variable data type
#define N 256              // DFT Size

void dft(IN_TYPE sample_real[N],
         IN_TYPE sample_imag[N]) {

    int i, j;
    TEMP_TYPE w;
    TEMP_TYPE c, s;

    // Temp arrays to hold intermediate
    // frequency domain results
    TEMP_TYPE temp_real[N];
    TEMP_TYPE temp_imag[N];

    // Calculate each freq domain sample iteratively
    for (i = 0; i < N; i += 1) {
        temp_real[i] = 0;
        temp_imag[i] = 0;

        // (2 * pi * i)/N
        w = (2.0 * 3.141592653589 / N) * (TEMP_TYPE) i;

        // Calculate the jth freq sample sequentially
        for (j = 0; j < N; j += 1) {

            // Utilize HLS tool to calculate sine and
            // cosine values for angle –j*w radians
            c = cos(j * w);
            s = -sin(j * w); // clockwise rotation

            // Multiply the current phasor with the
            // appropriate input sample and keep
            // running sum
            temp_real[i] += (sample_real[j] * c –
                             sample_imag[j] * s);
            temp_imag[i] += (sample_real[j] * s +
                             sample_imag[j] * c);
        }
    }

    // Perform an inplace DFT, i.e., copy result into
    // the input arrays
    for (i = 0; i < N; i += 1) {
        sample_real[i] = temp_real[i];
        sample_imag[i] = temp_imag[i];
    }
}
```

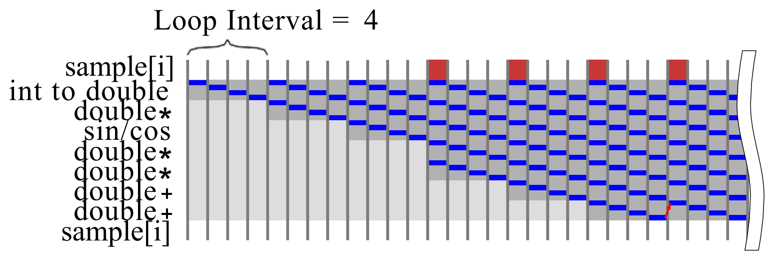
Schedule & architecture for baseline code





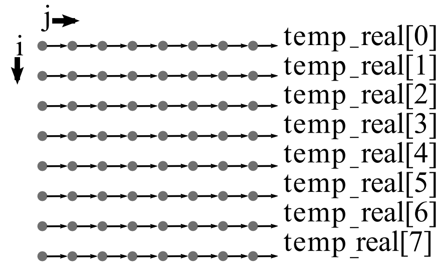
DFT optimization

- Implementing floating point operations is typically very expensive and requires many pipeline stages, particularly for double precision
- This significantly affects the performance of the loop when executed sequentially
- With pipelining, the effect of these high-latency operations is less critical, since multiple executions of the loop can execute concurrently
- However, the recurrence in accumulating `temp_real` and `temp_imag` limits the achievable II when pipelining the inner loop



DFT optimization – achieving II=1

- Use of 64-bit double data types hampers performance, so one opportunity to improve the performance is to use 32-bit float or 16-bit half data types instead – if greater loss of precision is acceptable, replacing the floating point data types with fixed point data can help reduce area and boost performance significantly – perhaps allowing the loop to be pipelined at II=1
- A more general approach to achieving II=1 is to avoid the recurrences by swapping the inner and outer loops, an optimization that is called *loop interchange*
  - It may not be obvious that we can swap the inner and outer loops here because of the extra code inside the outer `i` loop, but the diagonal symmetry inherent in the S matrix allows `i` and `j` to be interchanged in the computation of `w`



DFT optimization – sin & cos lookups

- The trigonometric functions can also be eliminated by storing the sin and cos values for the second row of the S matrix on-chip, since all S matrix entries are represented in this row alone
  - Storing just the one row of S for a cost of $O(N)$ storage and appropriately indexing into it saves a factor of $O(N)$ storage overall – this is significant as N scales
- Accessing multiple entries per cycle of this one table is fraught though, because both odd and even entries are needed to construct the entries for any other row in order to compute the entries of the frequency domain vector
  - Since these S values are only ever read though, they can be treated as a ROM, which Vivado HLS will readily replicate

DFT optimization - interface

- Instead of using the input `sample_real` and `sample_imag` values to return the DFT results in place, separate output ports for the `temp_real` and `temp_imag` results would avoid the need for temporary storage and not constrain the computation of the results to suit the storage arrangement chosen for the inputs

Concluding remarks

- The DFT is a fundamental signal processing operation
- At its core, it relies on matrix-vector multiplication, which we looked at in detail to optimize its performance
- We considered several opportunities for optimizing functionally correct DFT code, including
    - Pipelining
    - Data type conversion
    - Loop interchange
    - Trig function lookups
- In the lab this week, you'll take a careful look at optimizing matrix-vector multiplication and applying these techniques to DFT software
- Next week, we'll look at optimizing a fast approach to computing DFTs