Parallel and Reconfigurable VLSI Computing (11)

Practical HLS Design

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

- Discreate Fourier Transform (DFT) Design
    - Principal of DFT
    - Optimization for a matrix-vector multiplication
    - Optimizations for a DFT design

Principal of DFT

Fourier Series

- Provides an alternative way to look at a real valued, continuous, periodic signal where the signal runs over one period from -π to π
- The seminal result from Jean Baptiste Joseph Fourier states that any continuous, periodic signal over a period of 2π can be represented by a sum of cosines and sines with a period of 2π

$$\begin{aligned} f(t) &\sim \tfrac{a_0}{2} + a_1\cos(t) + a_2\cos(2t) + a_3\cos(3t) + \ldots \\ &\quad + b_1\sin(t) + b_2\sin(2t) + b_3\sin(3t) + \ldots \\ &\sim \tfrac{a_0}{2} + \sum_{n=1}^{\infty}(a_n\cos(nt) + b_n\sin(nt)) \end{aligned}$$

where the Fourier coefficients a0, a1, ... and b1, b2, ... are computed as

$$\begin{aligned} a_0 &= \tfrac{1}{\pi}\int_{-\pi}^{\pi} f(t)\,\mathrm{d}t \\ a_n &= \tfrac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos(nt)\,\mathrm{d}t \\ b_n &= \tfrac{1}{\pi}\int_{-\pi}^{\pi} f(t)\sin(nt)\,\mathrm{d}t \end{aligned}$$

Direct current (DC) term

Periodic Presentation

- Assume a function is periodic on [-L,L] rather than [-π,π], then we have

$$t \equiv \frac{\pi t'}{L}$$

and

$$\mathrm{d}t = \frac{\pi \mathrm{d}t'}{L}$$

- Solving for t' and substituting t' into original DFT equation, then

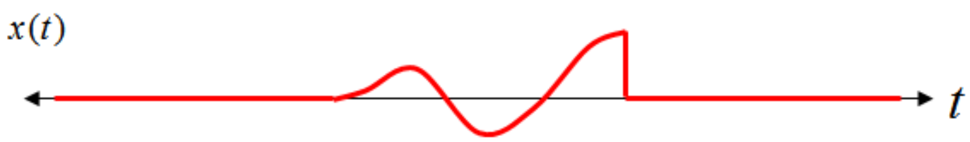
$$f(t') = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos(\frac{n\pi t'}{L}) + b_n \sin(\frac{n\pi t'}{L}))$$

$$\begin{aligned}
a_0 &= \tfrac{1}{L} \int_{-L}^{L} f(t')\,\mathrm{d}t' \\
a_n &= \tfrac{1}{L} \int_{-L}^{L} f(t') \cos(\tfrac{n\pi t'}{L})\,\mathrm{d}t' \\
b_n &= \tfrac{1}{L} \int_{-L}^{L} f(t') \sin(\tfrac{n\pi t'}{L})\,\mathrm{d}t'
\end{aligned}$$

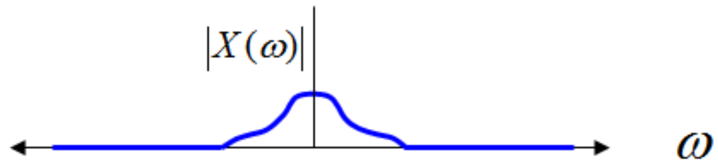
Continuous Time Fourier Transform

- Extends in time from minus infinity to plus infinity
- There is no implied repetition in time, therefore the frequency domain is a continuous function
- The frequency domain also goes from minus infinity to infinity, with no implied repetition, so the time domain is also continuous
- We can use Euler's formula $e^{jnt} = \cos(nt) + j \sin(nt)$ to give a more concise formulation

$$X(\omega) = \int_{t=-\infty}^{\infty} x(t) e^{-j\omega t} dt$$



TIME



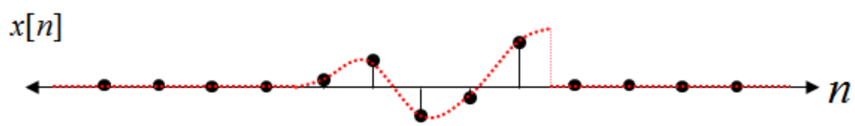
FREQ

Discrete Time Fourier Transform

- The only difference from above is we now sample in time the non-repeating time domain function
- This one change causes the frequency domain to repeat. But notice that the frequency domain is a continuous function (Because the time domain is not repeating)
- DTFT is that of a discreate time sequence

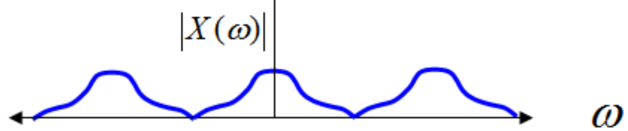
$$X(\omega) = \sum_{n=-\infty}^{\infty} x[n] e^{-j\omega n}$$

Discrete in Time



TIME

Repeating in Freq

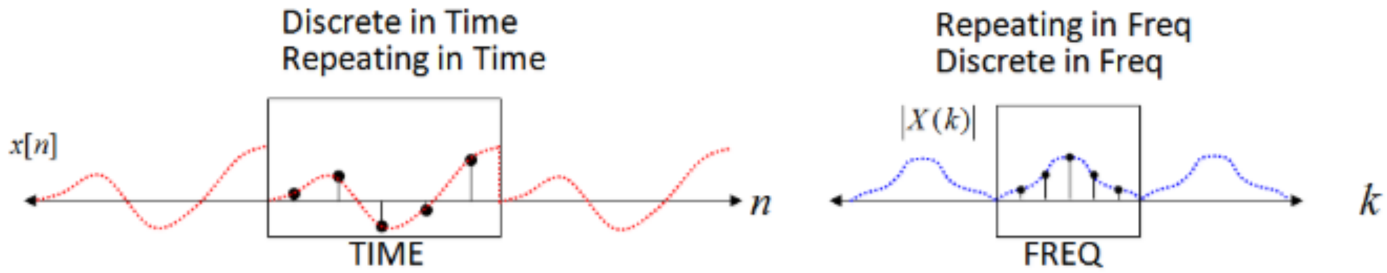


FREQ

Discrete Fourier Transform (DFT)

- We limit the time domain over a finite duration (similar to the Fourier Series Expansion), which I argue is identical (mathematically and intuitively) to repeating in time
- The DFT (for k=0, ..., N-1) is samples, evenly spaced in frequency, of the DTFT

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j(k\omega_o)n}$$



DFT (N=8) Example

- Matrix-vector multiplication

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j(k\omega_o)n}$$

$$\begin{bmatrix} X[0] \\ X[1] \\ \vdots \\ X[7] \end{bmatrix} = \begin{bmatrix} \exp(-j(0 \cdot \omega_0) \cdot 0) & \exp(-j(0 \cdot \omega_0) \cdot 1) & \cdots & \exp(-j(0 \cdot \omega_0) \cdot 7) \\ \exp(-j(1 \cdot \omega_0) \cdot 0) & \exp(-j(1 \cdot \omega_0) \cdot 1) & \cdots & \exp(-j(1 \cdot \omega_0) \cdot 7) \\ & \vdots & & \\ \exp(-j(7 \cdot \omega_0) \cdot 0) & \exp(-j(7 \cdot \omega_0) \cdot 1) & \cdots & \exp(-j(7 \cdot \omega_0) \cdot 7) \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[7] \end{bmatrix}$$

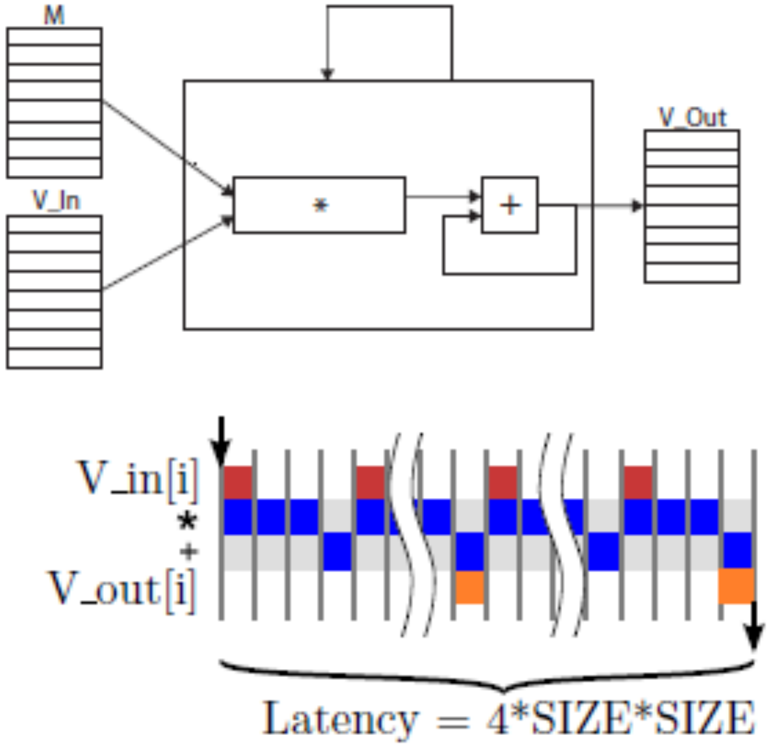
Matrix-vector Multiplication Code

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

Optimization of Matrix-Vector Multiplication

Sequential Computation of Matrix-vector Multiplication

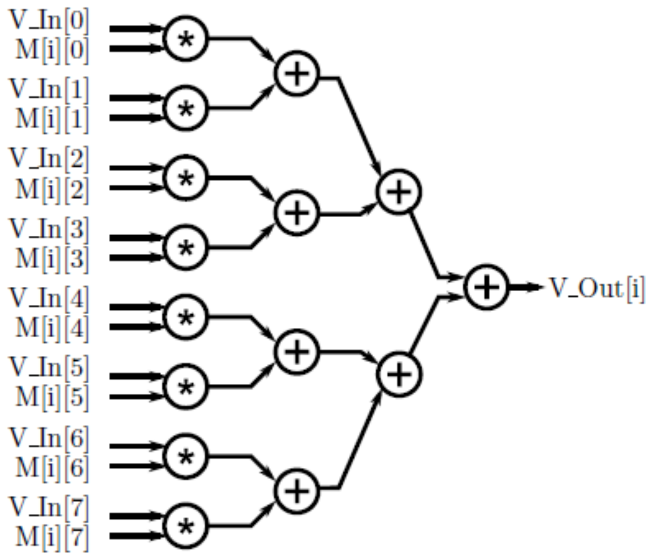
- Each element is computed and stored into the BRAM
- Vivado HLS synthesizes with no pragma (Note, a multiplication consumes 3 cycles in the lecture)



Parallelism by Loop Unrolling

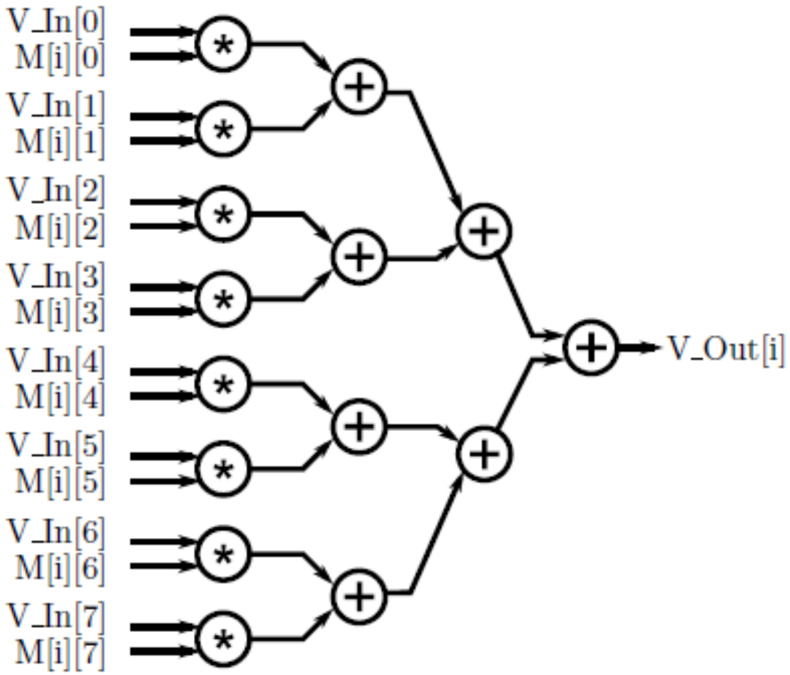
- Archived by #pragma HLS unroll

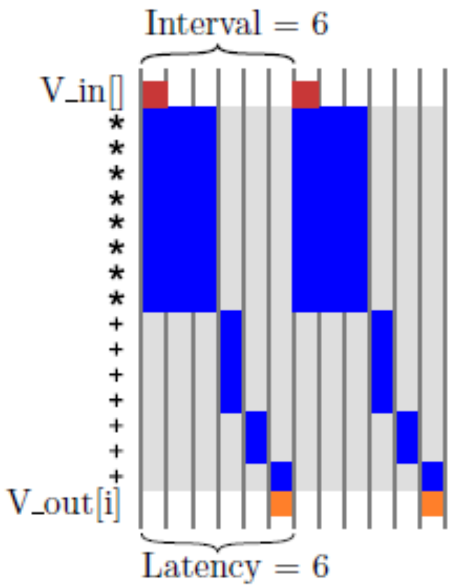
```
#define SIZE 8
typedef int BaseType;

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++) {
        BaseType sum = 0;
        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];
    }
}
```

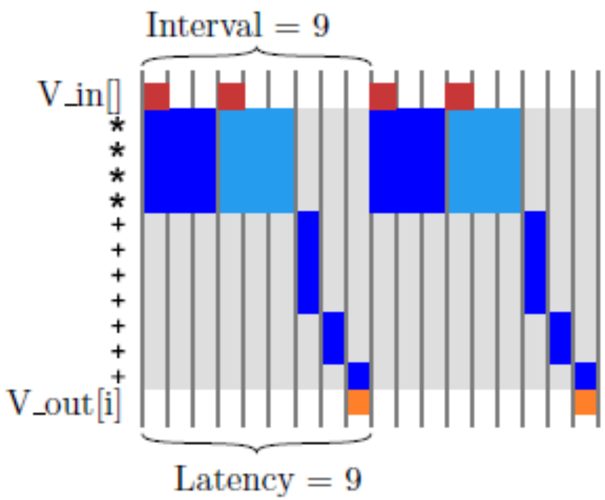


Sequential Execution from the Unrolled Inner Product

- Latency of 6 cycles for each iteration and requires 8 multipliers and 7 adders

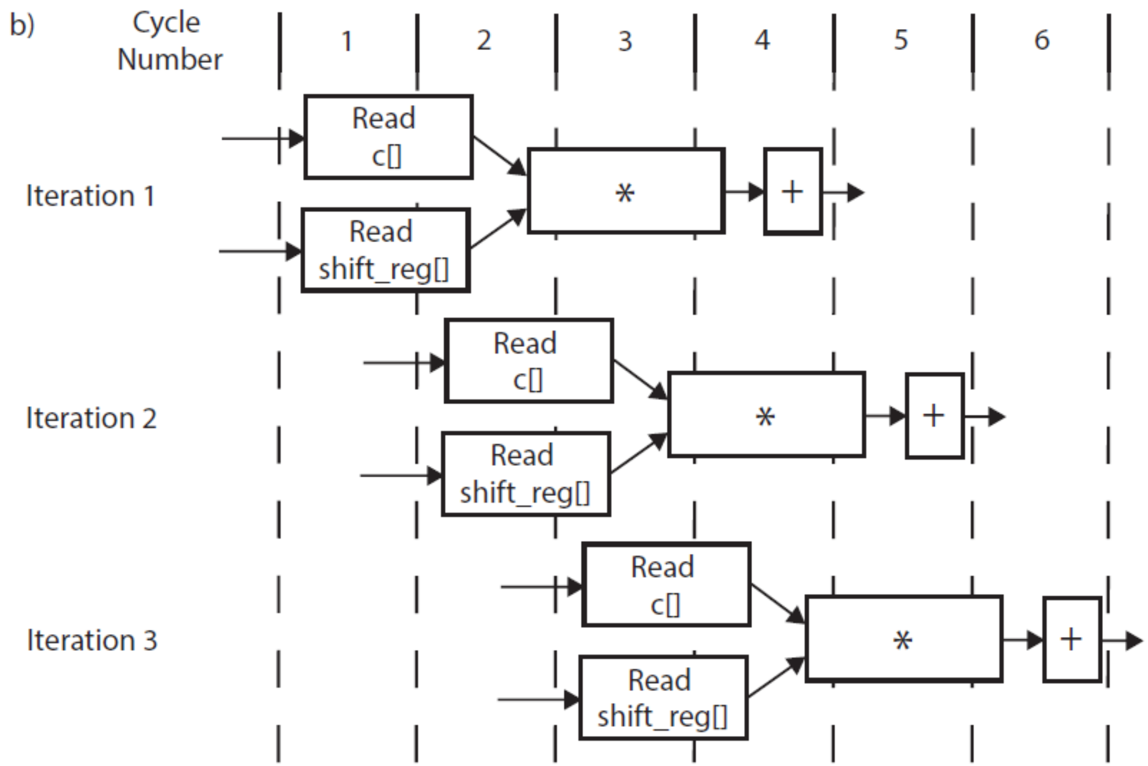






Loop Pipelining

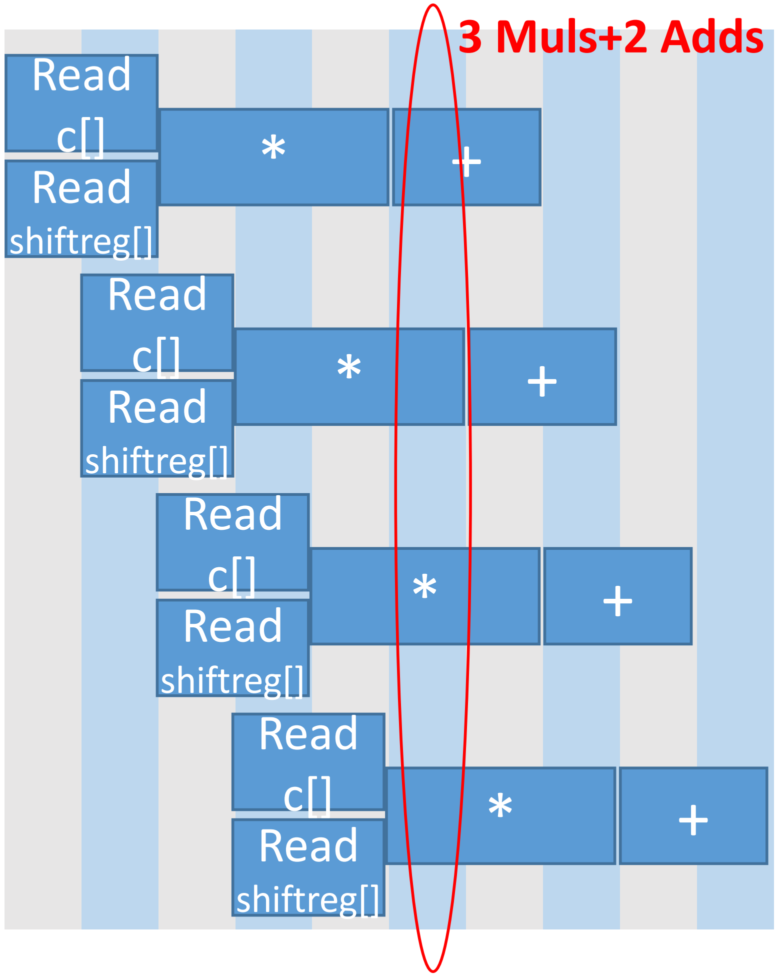
- All of the statements in the second iteration happen only when all of the statements from the first iteration are complete
- Schedule for three iterations of a pipelined version of the MAC for loop



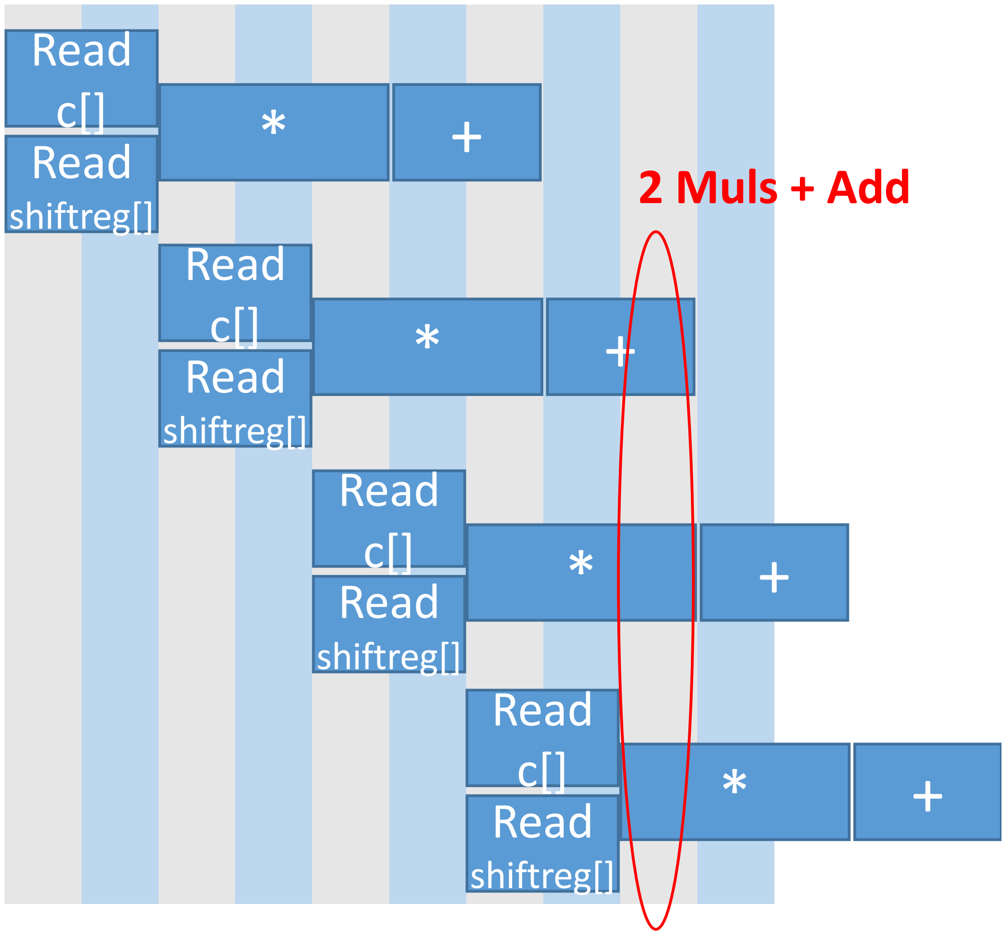
Loop Initiation Interval (II)

- 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

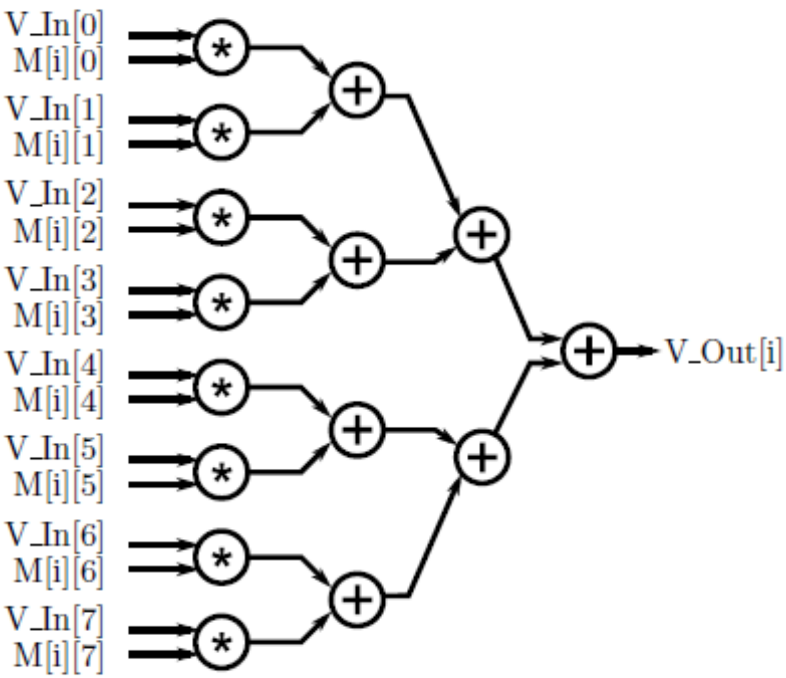


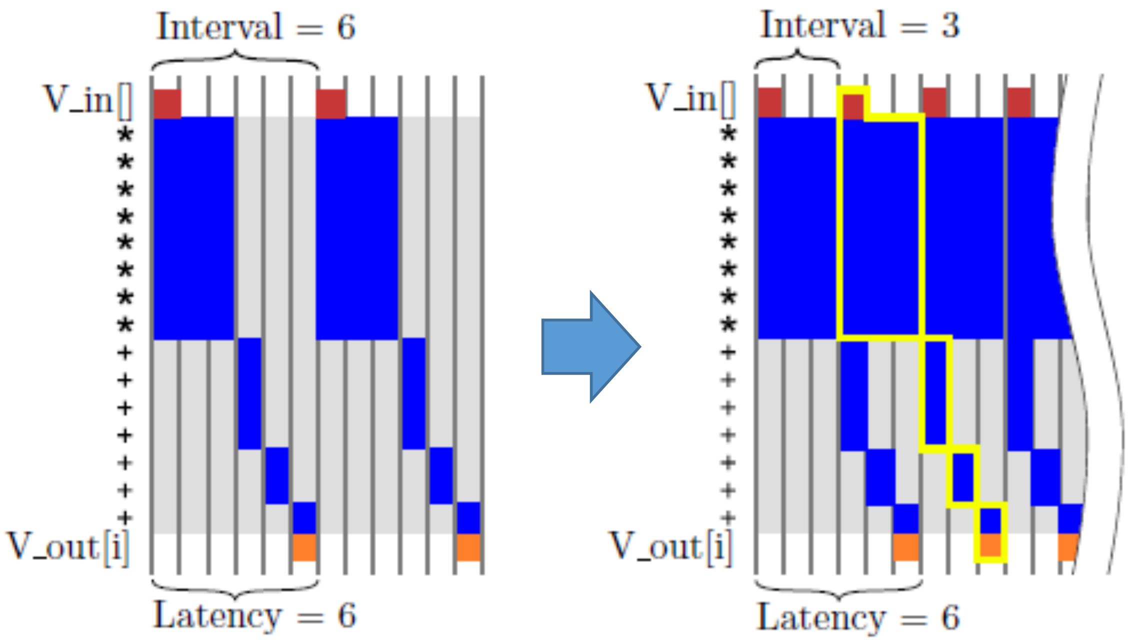
#pragma HLS pipeline II=2



Pipelined Implementation from the Unrolled Inner Loop

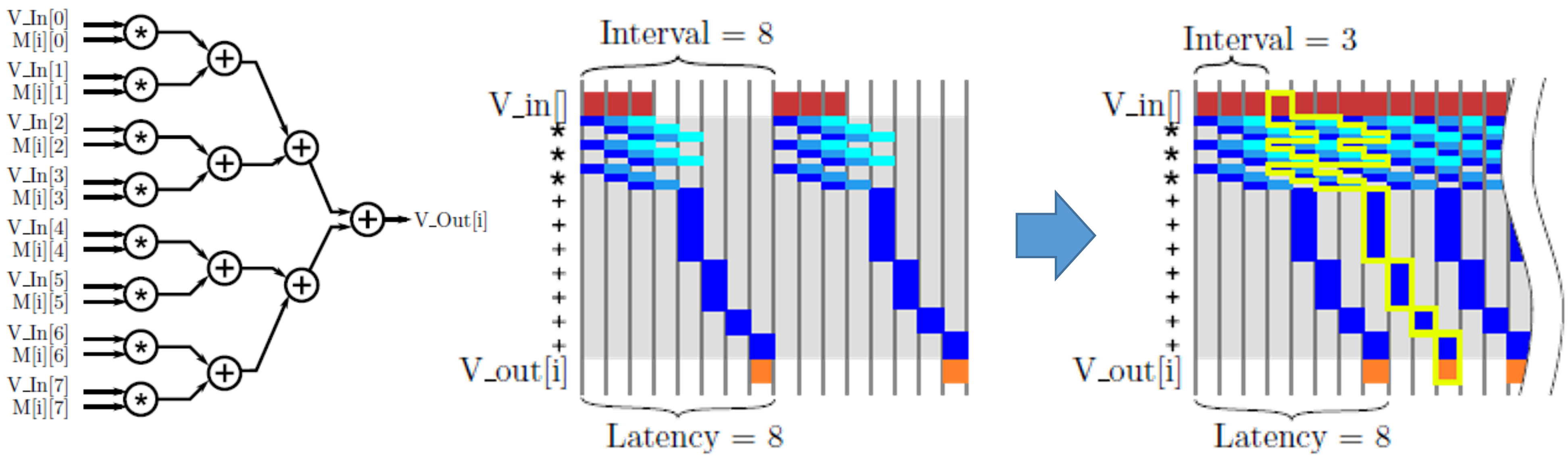
- Use #pragma HLS pipeline II=3 to the unrolled loop
- It reduces the interval of a loop to be reduced, however does not affect the latency





Pipelined Implementation from the Pipelined Multipliers

- Pipelining is possible at different levels of hierarchy, including the operator level, loop level, and function level
- Example: #pragma HLS pipeline II=3 is applied to the pipelined multipliers

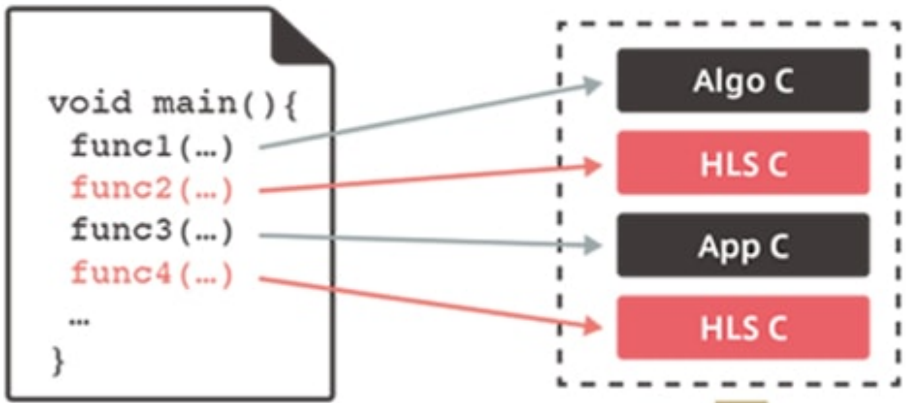


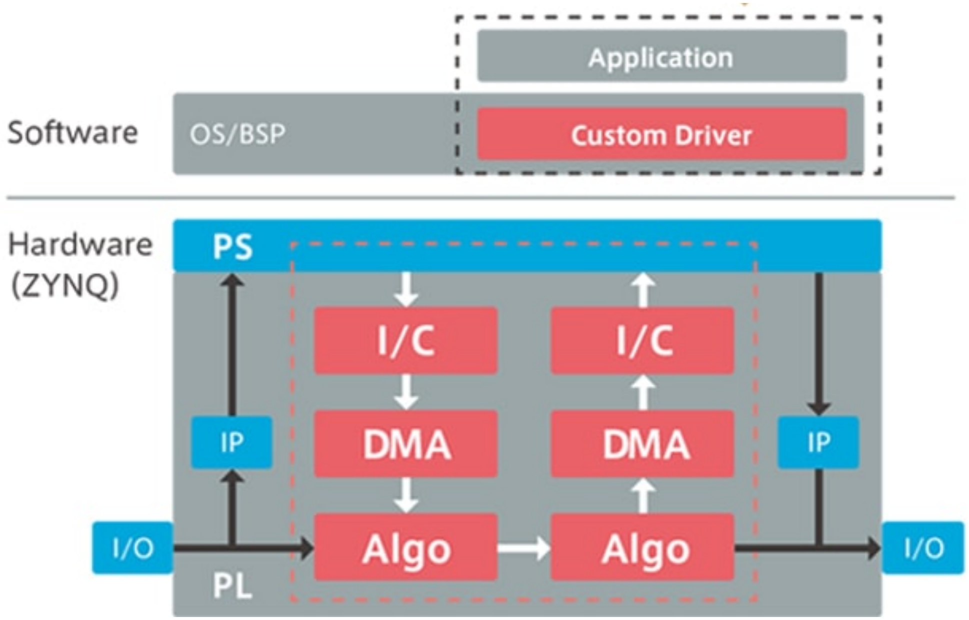
Storage Trade-offs

- In ideal case, arras (data and coefficient) are accessible at anytime
- In practice, the placement of the data plays a crucial role in the performance and resource usage
- In most processor systems, since the memory architecture is fixed, we can only adapt the program to attempt to best make use of the available memory hierarchy
  - Taking care to minimize register spills and cache misses
- In an FPGA design, we can also explore and leverage different memory structures and try to find the memory structure
  - Off-chip memory (DRAM), BRAM, and register

Storages on/off FPGA-based System

- Data access times for DRAMs are typically too long
- Primary choices for on-chip storage (BRAM or FFs)
  - BRAMs offer higher capacity (Mbits), and limited to two different ports
  - FFs allow for multiple reads at different addresses in a single clock, but typically limited to around 100 KB





Array Partition

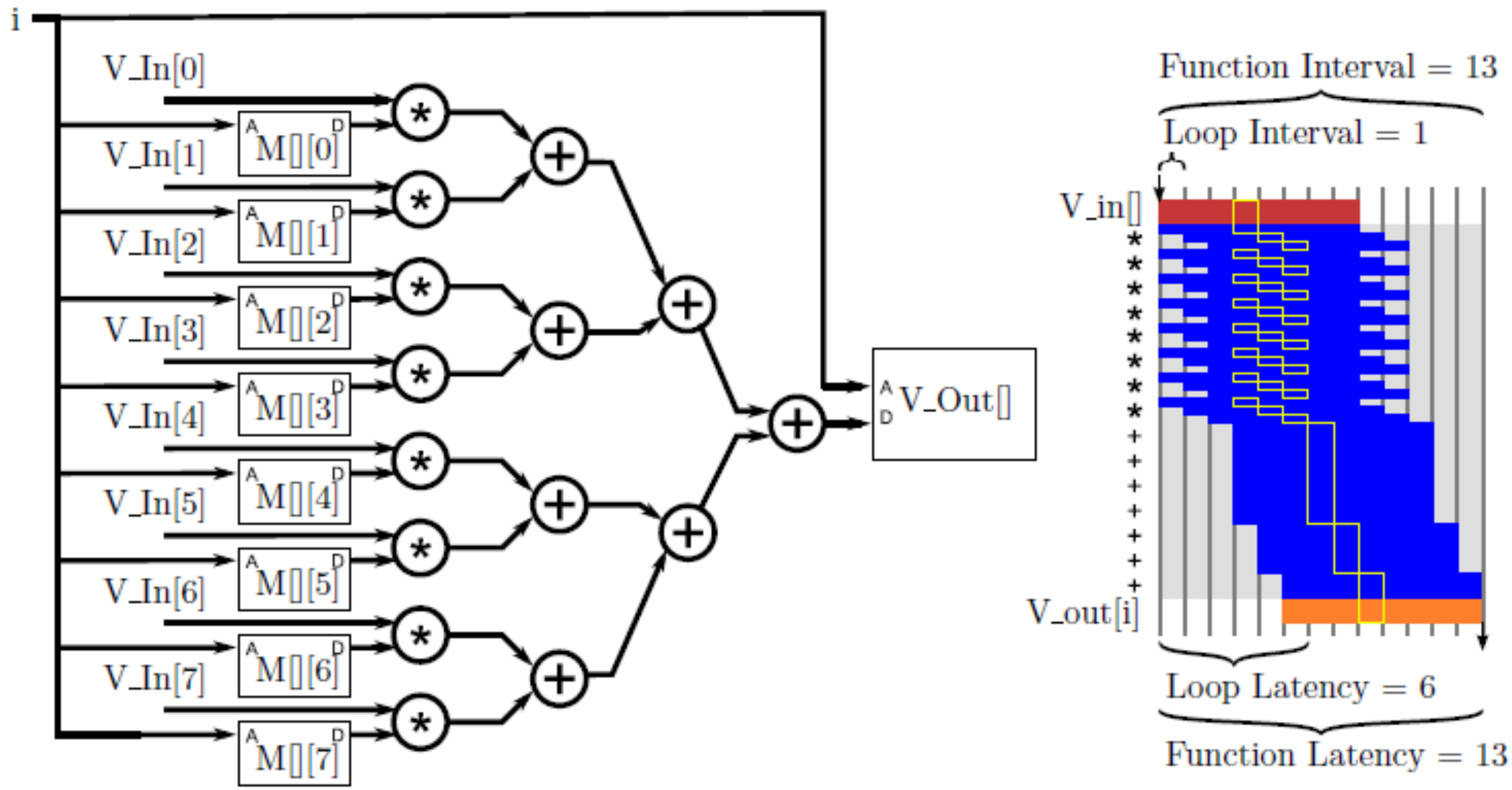
- If throughput is the number one concern, all of the data would be stored in FFs
  - #pragma HLS array_partition variable=XX complete
  - However, as the size of arrays grows large, it is not feasible
- Using a single composite (large) BRAM means that we can only access two ports at a time
  - Prevents higher performance HW
- For instance, most designs require large arrays to be strategically divided into smaller BRAMs
  - #pragma HLS arrary_partition variable=XX factor=X cyclic/block

Matrix-vector Multiplication with Array Partitioning (complete)

```
#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
    BaseType 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;
    }
}
```

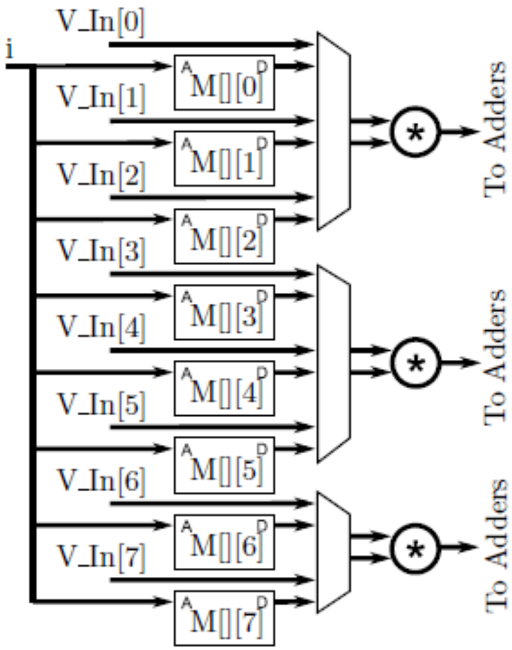
Matrix-vector Multiplication Architecture with Array Partitioning (Complete)

- The pipelining registers have been elided

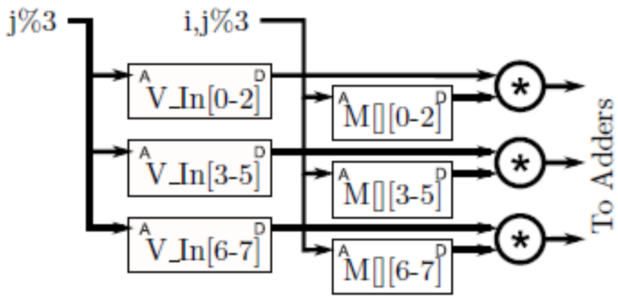


Matrix-vector Multiplication Architecture at II=3 with Array Partitioning

- On the left, the arrays have been partitioned more than necessary, resulting in multiplexers
- On the right, the arrays are partitioned with factor=3
  - In this case, multiplexing has been reduced, but the j loop index becomes a part of the address computations



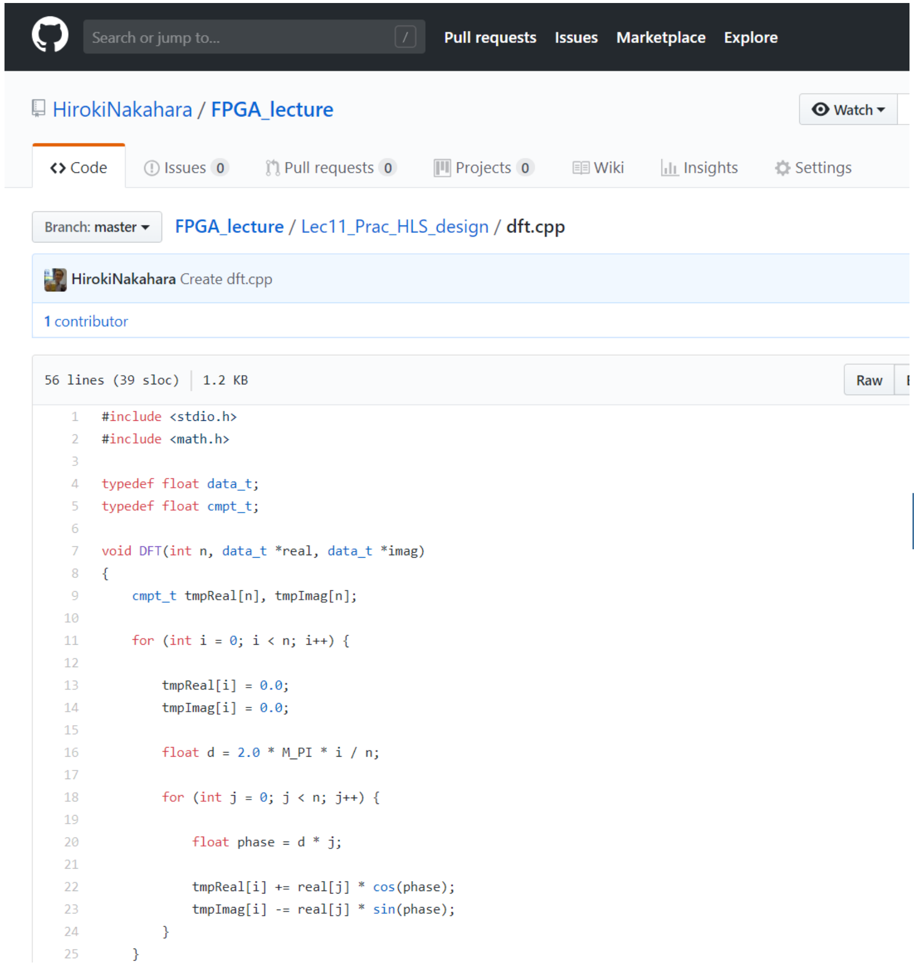
Question: (block?cyclic?)

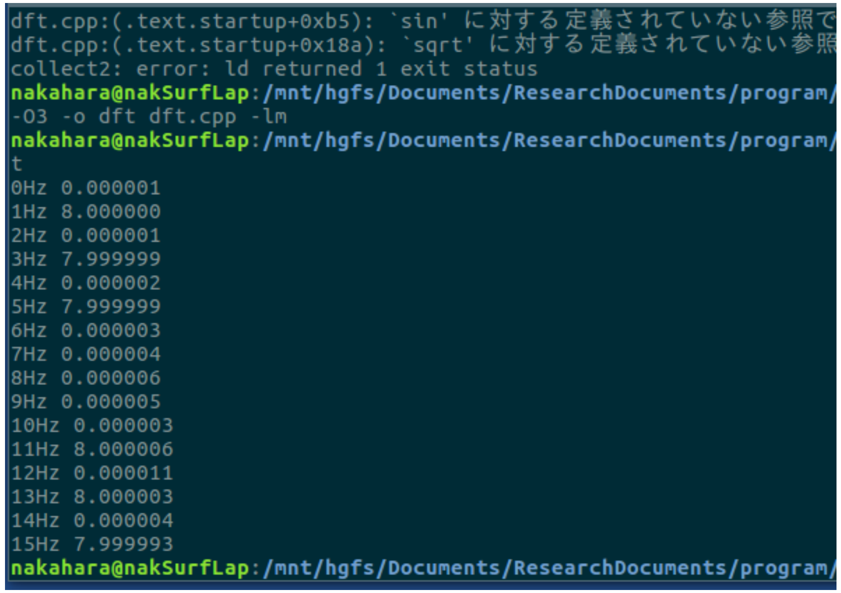


Optimizations of DFT Design

Baseline C++ Code

- See, Github: https://github.com/HirokiNakahara/FPGA_lecture/blob/master/Lec11_Prac_HLS_design/dft.cpp





Straightforward HLS Realization

- See, an HLS function DFT() in https://github.com/HirokiNakahara/FPGA_lecture/blob/master/Lec11_Prac_HLS_design/dft_hls.cpp
  - Re-write constant memory size
  - Bounded loop repetition
  - Assigned loop labels (DFT_LOOP, DFT_MAC, WB)

```
DFT_LOOP: for (int i = 0; i < 16; i++) {

    tmpReal[i] = 0.0;
    tmpImag[i] = 0.0;

    float d = 2.0 * M_PI * i / 16;

    DFT_MAC: for (int j = 0; j < 16; j++) {

        float phase = d * j;

        tmpReal[i] += real[j] * cos(phase);
        tmpImag[i] -= real[j] * sin(phase);
    }
}
```

These sentences become bottleneck

Table Loop-Up for Trigonometric Functions

- See, an HLS function DFT_trigo_tbl() in https://github.com/HirokiNakahara/FPGA_lecture/blob/master/Lec11_Prac_HLS_design/dft_hls.cpp
- Computation bottlenecks are removed

```
cmpt_t sin_tbl[16][16]={
{0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0
{0.000000,0.382683,0.707107,0.923880,1.000000,0.923880,0
{0.000000,0.707107,1.000000,0.707107,-0.000000,-0.707107
{0.000000,0.923880,0.707107,-0.382683,-1.000000,-0.38268
{0.000000,1.000000,-0.000000,-1.000000,0.000000,1.000000
{0.000000,0.923880,-0.707107,-0.382683,1.000000,-0.38268
{0.000000,0.707107,-1.000000,0.707107,-0.000000,-0.70710
{0.000000,0.382683,-0.707107,0.923880,-1.000000,0.923880
{0.000000,-0.000000,0.000000,-0.000000,0.000000,-0.00000
{0.000000,-0.382683,0.707107,-0.923879,1.000000,-0.92387
{0.000000,-0.707107,1.000000,-0.707107,0.000000,0.707107
{0.000000,-0.923880,0.707107,0.382683,-1.000000,0.382683
{0.000000,-1.000000,-0.000000,1.000000,0.000000,-1.00000
{0.000000,-0.923879,-0.707107,0.382683,1.000000,0.382683
{0.000000,-0.707107,-1.000000,-0.707106,0.000001,0.70710
{0.000000,-0.382683,-0.707107,-0.923879,-1.000000,-0.923
};
```

```
void DFT_trigo_tbl( data_t real[16], data_t imag[16])
{
    cmpt_t tmpReal[16], tmpImag[16];

    DFT_LOOP: for (int i = 0; i < 16; i++) {
        tmpReal[i] = 0.0;
        tmpImag[i] = 0.0;

        DFT_MAC: for (int j = 0; j < 16; j++) {
            tmpReal[i] += real[j] * cos_tbl[i][j];
            tmpImag[i] -= real[j] * sin_tbl[i][j];
        }
    }

    WB: for (int i = 0; i < 16; i++) {
        real[i] = (data_t)tmpReal[i];
        imag[i] = (data_t)tmpImag[i];
    }
}
```

Comparison

Original

Performance Estimates

⊟ **Timing (ns)**

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.63	1.25

⊟ **Latency (clock cycles)**

⊟ Summary

Latency		Interval		
min	max	min	max	Type
17250	19298	17250	19298	none

⊟ Detail

⊞ Instance

⊞ Loop

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	86
FIFO	-	-	-	-
Instance	16	203	12511	20494
Memory	0	-	128	16
Multiplexer	-	-	-	423
Register	-	-	893	-
Total	16	203	13532	21019
Available	280	220	106400	53200
Utilization (%)	5	92	12	39

Table Look-Up

Performance Estimates

⊟ **Timing (ns)**

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.02	1.25

⊟ **Latency (clock cycles)**

⊟ Summary

Latency		Interval		
min	max	min	max	Type
3138	3138	3138	3138	none

⊟ Detail

⊞ Instance

⊞ Loop

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	95
FIFO	-	-	-	-
Instance	-	10	696	1422
Memory	2	-	128	16
Multiplexer	-	-	-	196
Register	-	-	320	-
Total	2	10	1144	1729
Available	280	220	106400	53200
Utilization (%)	~0	4	1	3

Applied Pipeline Architecture

- See, an HLS function DFT_pipe() in https://github.com/HirokiNakahara/FPGA_lecture/blob/master/Lec11_Prac_HLS_design/dft_hls.cpp
  - Achieved II=1
  - Fortunately, array partition (dim=2) is automatically done
  - Carefully read Vivado HLS Console!!

```
INFO: [XFORM 203-502] Unrolling all sub-loops inside loop 'DFT_LOOP' (lec_11_1/dft_hls.cpp:51) in function 'DFT_pipe' for p
INFO: [XFORM 203-501] Unrolling loop 'DFT_MAC' (lec_11_1/dft_hls.cpp:56) in function 'DFT_pipe' completely.
INFO: [XFORM 203-102] Partitioning array 'sin_tbl' in dimension 2 automatically.
INFO: [XFORM 203-102] Partitioning array 'cos_tbl' in dimension 2 automatically.
```

```
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-61] Pipelining loop 'DFT_LOOP'.
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 87.
```

Bitwidth Optimization

- Apply a half-precision (16 bit) floating point
- Include <hls_half.h>
- To reduce the HW resource

Overall Performance and HW Resources

Original

Performance Estimates

⊟ **Timing (ns)**

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.63	1.25

⊟ **Latency (clock cycles)**

⊟ Summary

Latency		Interval		
min	max	min	max	Type
17250	19298	17250	19298	none

⊟ Detail

⊞ Instance

⊞ Loop

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	86
FIFO	-	-	-	-
Instance	16	203	12511	20494
Memory	0	-	128	16
Multiplexer	-	-	-	423
Register	-	-	893	-
Total	16	203	13532	21019
Available	280	220	106400	53200
Utilization (%)	5	92	12	39

After Optimizations

Performance Estimates

⊟ **Timing (ns)**

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.61	1.25

⊟ **Latency (clock cycles)**

⊟ Summary

Latency		Interval		
min	max	min	max	Type
146	146	146	146	none

⊟ Detail

⊞ Instance

⊞ Loop

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	68
FIFO	-	-	-	-
Instance	-	124	6200	4712
Memory	0	-	560	132
Multiplexer	-	-	-	267
Register	0	-	2016	226
Total	0	124	8776	5405
Available	280	220	106400	53200
Utilization (%)	0	56	8	10

Conclusion

- Introduce a DFT
- Comparison of various optimizations
- Applied to optimizations
  - Achieved more faster and smaller architecture

Exercise

1. (Mandatory) Compared with an Unrolling version of DFT design with respect to performance and resources
2. (Optional) Implement the DFT design on your ZYBO board

Send a report by a PDF file to OCW-i

Deadline is 7th, Aug., 2020 JST PM 13:20

(At the beginning of the lecture)