Finite Impulse Response(FIR) Filter

Lecturer: Jia-Ming Lin

Outline

- Background
- Base FIR Architecture
- Calculating Performance
- Design Optimization
  - Loop Unrolling
  - Loop Pipelining
  - Avoid if/else branch condition
- Lab
  - Lab 3-1: Practice with small filter size N=11.
  - Lab 3-2: Performance tuning for larger filter size N=512.

Background

Background

- Applications
  - Signal restoration
    - Reduce the high frequent noise
  - Signal separation
    - Isolate input signal into different parts
- Digital filters
  - Infinite Impulse Filter(IIR)
    - **Advantage**: lower computational complexity; **disadvantage**: unstable
  - Finite Impulse Filter(FIR)
    - **Advantage**: stable; **disadvantage**: higher computational complexity
  - Refer to this video for more theoretical details
- We investigate the methods to speedup FIR algorithm

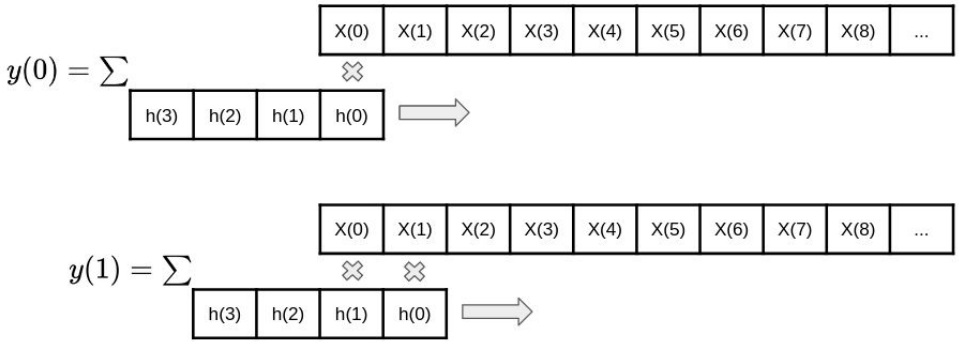
Background

- Definition
  - FIR algorithm, can be computed through the process of 1-D convolution.
  - **N-tap** FIR

$$y(i) = \sum_{j=0}^{N-1} h(j) \cdot X(i-j)$$

h is constant vector over time

  - Example: 4-tap FIR



Background

- Real-world example
  - Signal separation via FIR,
  - Python implementation, Github.
  - **Input**:
    - **Audio(wav)**: data type=int16, 44100 samples/sec.
    - **Filter**:
      - Symmetry
      - Value can be changed for different environments
      - Tap size = 461 entries (461-Tap FIR)
  - **Architecture**:
    - Two low-pass filter, two high pass filter
    - Final output is the average of four FIR outputs

Base FIR Design

Functionally Correct Implementation

```
void fir (data_t *y, data_t x) {


}
```

Streaming function, fir

- Receive/output on sample at a time
  - “**x**” is the input port
  - “**y**” is the output port
- Function is called multiple time
- Input/output data type:
  16-bit, 8-bit, integer, fixed-point etc.

Functionally Correct Implementation

```
void fir (data_t *y, data_t x) {
  coef_t c[N] = {
        0,-10,-9,23,56,63,56,23,-9,-10,0
  };

  static data_t shift_reg[N];

}
```

Components in the function

- "**coef_c**", hard coded coefficient array
- "**shift_reg**", cache for previous samples,
  - Streaming function receives only one sample at a time
  - But we need N=11 consecutive samples to output a result
  - **static**, since data must be persistent across multiple function calls
    - Initial value = 0

Functionally Correct Implementation

```
void fir (data_t *y, data_t x) {
  coef_t c[N] = {
          0,-10,-9,23,56,63,56,23,-9,-10,0
  };

  static data_t shift_reg[N];

  data_t data;
  acc_t acc;
  int i;

}
```

Components in the function

- "**acc**", accumulator for multiple multiplications
  - To prevent numerical overflow
  - e.g. (int8+int8) need 9-bit integer to store the value for correctness
  - Reset to "0" while each function call.

Functionally Correct Implementation

```
void fir (data_t *y, data_t x) {
  coef_t c[N] = {
          0,-10,-9,23,56,63,56,23,-9,-10,0
  };

  static data_t shift_reg[N];

  data_t data;
  acc_t acc;
  int i;

  acc=0;

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

Shift_Accum_Loop: each iteration

- Multiply **one sample** with **one coefficient**
  - N=11 iterations
- "**acc**" stores the running sum
  - How many bits for "**acc**" is needed?
  - Depends on the input data and application requirements.
- Shift values in "**shift_reg**", as FIFO
  - shift_reg[i+1]=shift_reg[i]

Functionally Correct Implementation

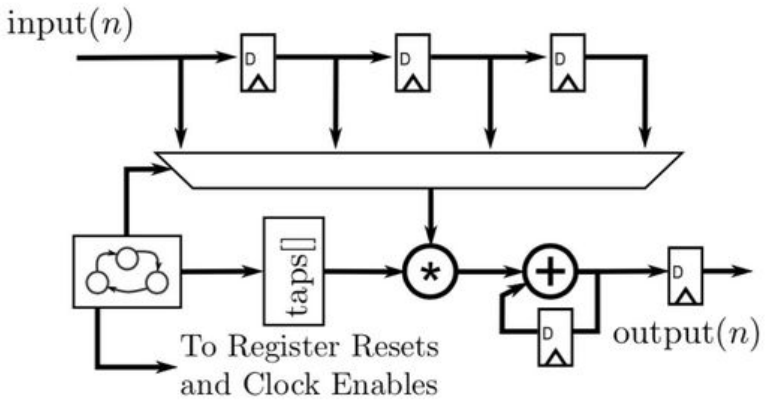
```
void fir (data_t *y, data_t x) {
  coef_t c[N] = {
          0,-10,-9,23,56,63,56,23,-9,-10,0
  };

  static data_t shift_reg[N];

  data_t data;
  acc_t acc;
  int i;

  acc=0;

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



- One multiplier
- N iterations to complete an output
- 4 cycles for each iteration
  - 2 cycles for parallel data read
    - 2 for read coefficient from “tabs[]”
    - 1 for read input
  - 1 for mul, 1 for add

Performance Estimation

Performance of Baseline and Challenges

- How long is the latency for obtaining an output y?
  - 4 cycles for each iteration
  - To output a result, need N iteration
  - Suppose one clock cycle takes 10 ns
  - If N = 461, it takes 4*461*10 = 209920 ns = 20us to obtain an output
- For an input audio sampling rate = 44100 samples/sec.
  - (1/44100) * 10^6 = 22.67 us
  - New sample will come in system in every 22.67 us
- Processing latency = 20 us, is really closed to sampling period = 22.67 us
- Practically, we wish processing latency is as less as possible.

Performance of Baseline and Challenges

- Suppose we use *Ns* DSP to process *Ns* multiplications parallelly
  - For the case N = 512 and data type = int16
  - It may take 4 cycles to obtain an output, it’s 512x speedup.
  - However, one 16-bit*16-bit multiplication needs one DSP,
  - 512 multiplications need 512 DSP, which exceeds 280, the DSP limit on PYNQ-Z2
- How about parallely perform Ms multiplication?
  - M is a proper divisor of N
  - Then how much speedup can we achieve?

Design Optimizations

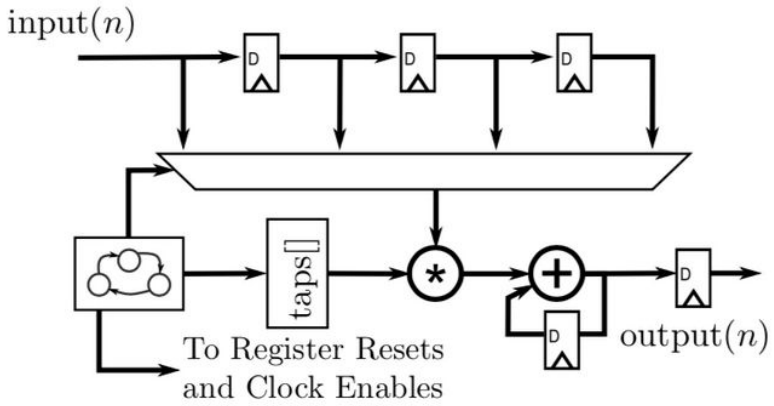
To increase parallelism

- Loop Unrolling
  - Multiple copies of processing elements for higher throughput
- Loop Pipelining
  - Parallel executions of multiple stages
  - Possibly multiple tasks run concurrently
- Avoid if/else branch condition in regular loop

Loop Unrolling

- By default, Vivado HLS synthesize the “**for**” loop in a sequential manner.
- **Advantage**: area efficient architecture
- **Disadvantage**: ignore possible parallelism across loop iterations

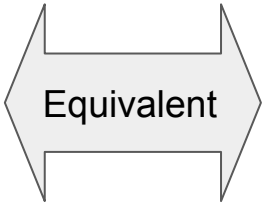
```
Shift_Accum_Loop:
for (i = N - 1; i >= 0; i--) {
    if (i == 0) {
        acc += x * c[0];
        shift_reg[0] = x;
    } else {
        shift_reg[i] = shift_reg[i - 1];
        acc += shift_reg[i] * c[i];
    }
}
```



Loop Unrolling

- Replicate the loop body by some number(factor) of the line.
- Two methods, automatically and manually
- Example: unroll factor = 5
- **Ideally, every copies are executed parallely.**

```
// automatically
for(int i = 0; i < 10; i++){
#pragma HLS UNROLL factor = 5
        …
        Loop Body
        ...
}
```




```
// manually
for(int i = 0; i < 2; i++){
        …
        Loop Body Copy 1
        Loop Body Copy 2
        Loop Body Copy 3
        Loop Body Copy 4
        Loop Body Copy 5
        …
}
```

Note: complete unroll without specifying the factor
i.e. N=10 parallel copy executions.

Loop Unrolling

	External Memory	BRAM	FFs
count	1-4	thousands	millions
size	GBytes	KBytes	Bits
total size	GBytes	MBytes	100s of KBytes
width	8-64	1-16	1
total bandwidth	GBytes/sec	TBytes/sec	100s of TBytes/sec

Memory Hierarchy Recall

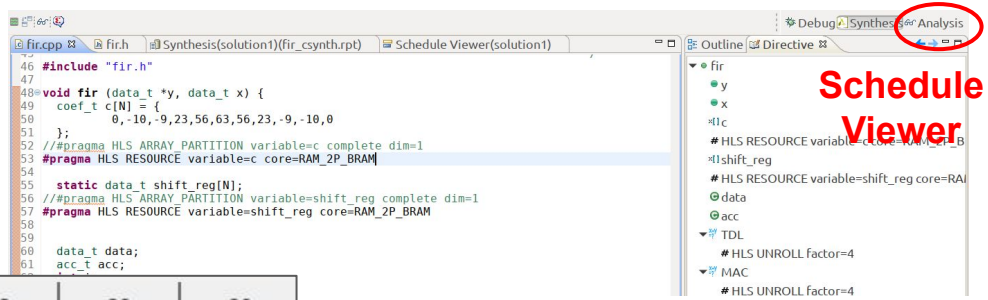
- **Ideally, every copies are executed parallely…**
- Some exceptions…
  - Limited by the shared resources across different copies.
  - Example:

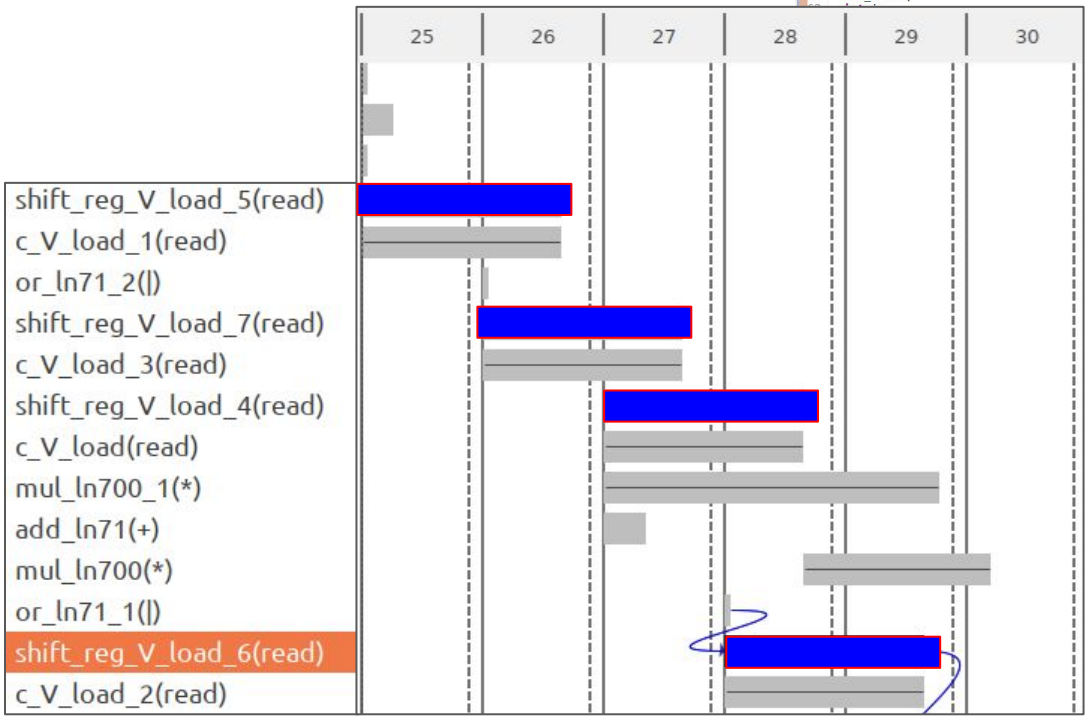
```
data_t shift_reg[N];
...
for(int i = 0; i < 2; i++){
        …
        Loop Body Copy 1
        Loop Body Copy 2
        Loop Body Copy 3
        Loop Body Copy 4
        Loop Body Copy 5
        …
}
```

- Simultaneous 5 reads to “**shift_reg**”
- Usually large array, e.g. “**shift_reg**” is implemented by BRAM
- BRAM has two read ports and one write port
  - Support 2 reads or 1 read 1 write in a cycle
- Only 2 copies can run simultaneously at most.

Loop Unrolling

- Data Hungry



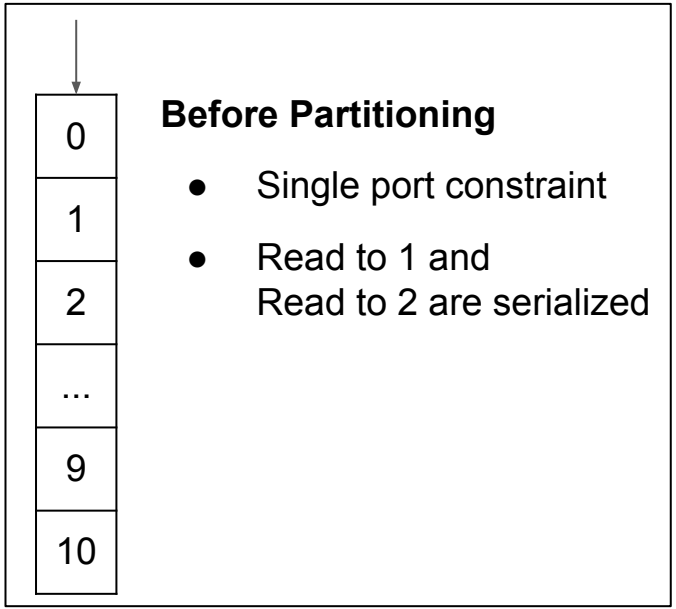


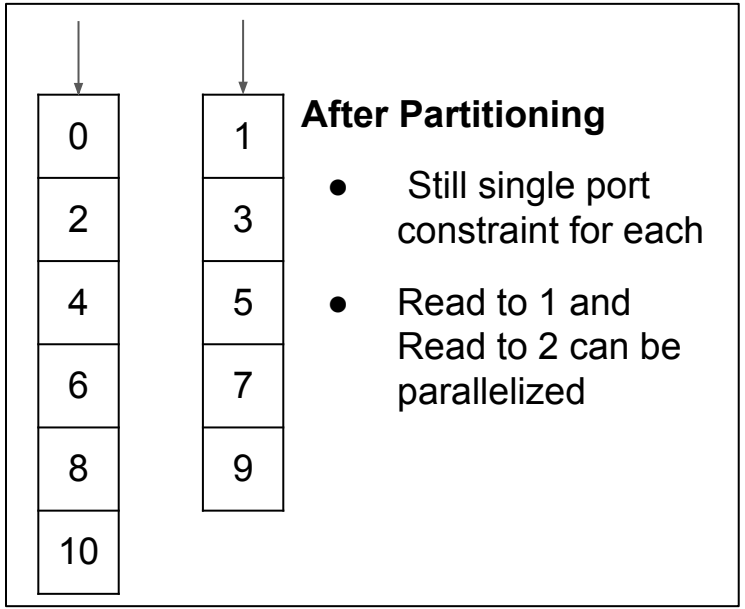
If “**shift_reg**” is RAM_2P_BRAM implemented

At most 2 data read on “**shift_reg**”

Array Partition

- Separate the memory into several partitions
  - To support higher parallelism
- Example, Suppose a single port memory:
  - Only one read or write in a cycle



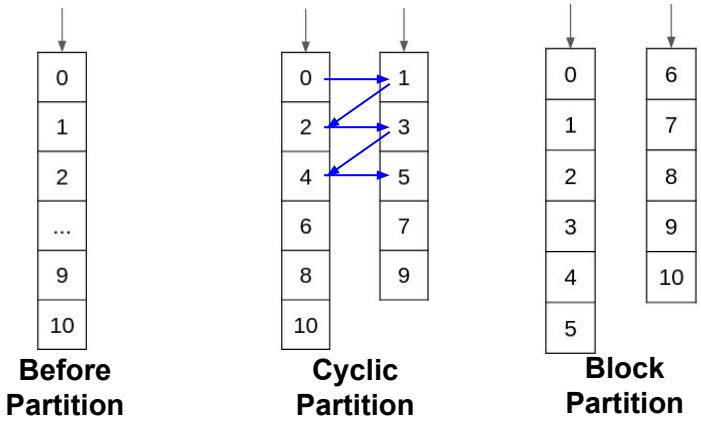


Array Partition

- Implementation in HLS

  data_t shift_reg[N];
  #pragma HLS ARRAY_PARTITION variable=shift_reg factor=K <PARTITION_TYPE>

  - **variable(required)**: array to be partitioned
  - **factor(optional)**: how many partitions are there
  - **PARTITION_TYPE**: cyclic, block, complete



Array Partition

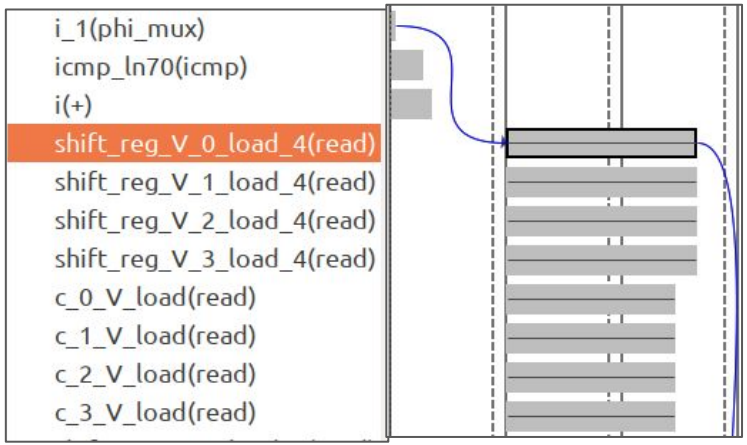
- Implementation in HLS

```
#pragma HLS ARRAY_PARTITION variable=c cyclic factor=4
#pragma HLS ARRAY_PARTITION variable=shift_reg cyclic factor=4
```

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

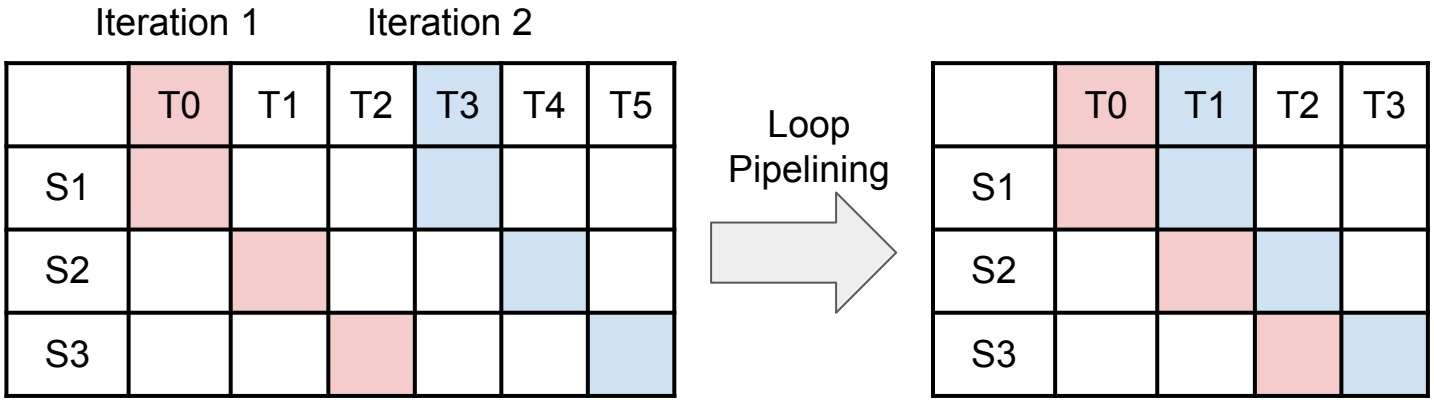
- Parallelize 4 reads to “**shift_reg**”
  - E.g. Read 0, 1, 2, 3 concurrently
- Therefore partition type is cyclic

Schedule Viewer



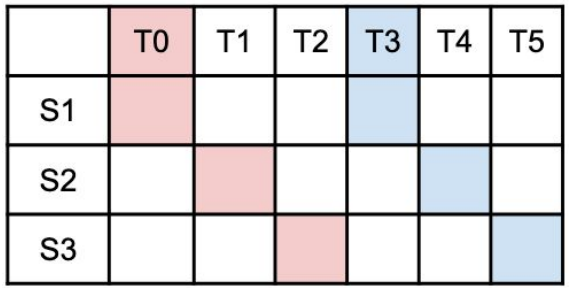
Loop Pipelining

- By default, Vivado HLS synthesize the “**for**” loop in a sequential manner.
  - Second iteration **happen only** when all statements in first iteration are complete.
- It is possible to perform different statements from different iterations in parallel.
  - e.g. at T1, iteration 1 is executing S2, while iteration 2 is executing S1
  - After pipelining, iteration 1 and iteration 2 is executed parallelly.

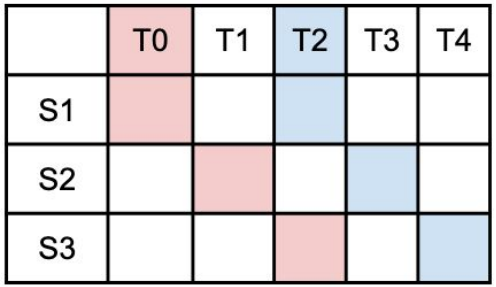


Loop Pipelining

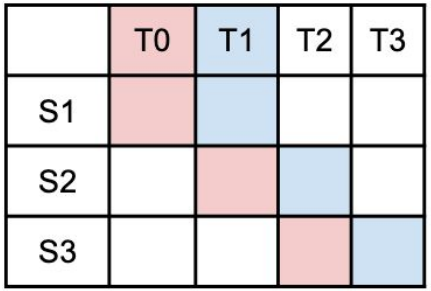
- Initiation Interval (II)
  - # of clock cycles until the next iteration of the loop can start



II = 3



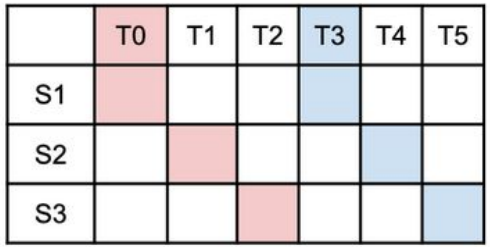
II = 2



II = 1

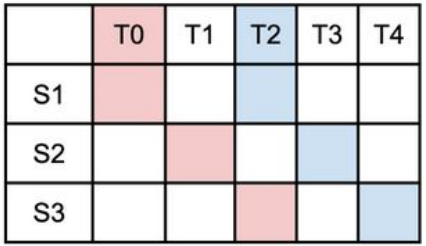
Loop Pipelining

- Calculating # of clock cycles that a “for” loop takes (loop latency)
  - Sequentially
    - (# of clock cycles for an iteration) * (# of iterations)
  - Pipelined
    - (# of clock cycles for an iteration) + (Initiate Interval)*(# of iterations - 1)



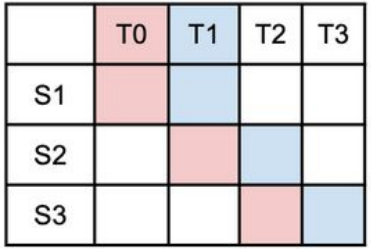
II = 3

Latency = 3*2 = 6



II = 2

Latency = 3+(2*1) = 5



II = 1

Latency = 3+(1*1) = 4

Loop Pipelining

- Implementation in HLS
  - Speedup 2x in this example

Before pipelining conditional branch

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

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

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1292	1292	12.920 us	12.920 us	1292	1292	none

After pipelining conditional branch

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

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

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
527	527	5.270 us	5.270 us	527	527	none

N = 256, clock period = 10 ns

Avoid branching condition in the loop

- if/else statements(control structure) in the loop
  - Statements can only be executed after the condition statement is resolved
- For **regular or predictable loop**, we can have more refactoring opportunities

Before removing conditional branch

```
   for(i = N-1; i >= 0; i--){
#pragma HLS UNROLL factor=16
      if(i == 0){
          data = 0;
          shift_reg[0] = x;
      }else{
          shift_reg[i] = shift_reg[i-1];
          data = shift_reg[i];
      }
      acc += data*c[i];
  }
```

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
801	801	8.237 us	8.237 us	801	801	none

After removing conditional branch

```
  TDL:for(i = N-1; i >=0; i--){
#pragma HLS UNROLL factor=16
      shift_reg[i] = shift_reg[i-1];
  }
  shift_reg[0] = x;


  MAC:for(i = N-1; i >=0 ; i--){
#pragma HLS UNROLL factor=16
      acc += shift_reg[i]*c[i];
  }
```

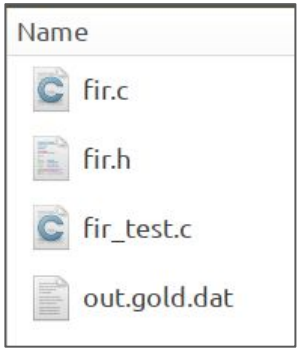
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
706	706	7.260 us	7.260 us	706	706	none

Here, N = 256

Labs

Lab 3-1: Practice with small filter size N=11

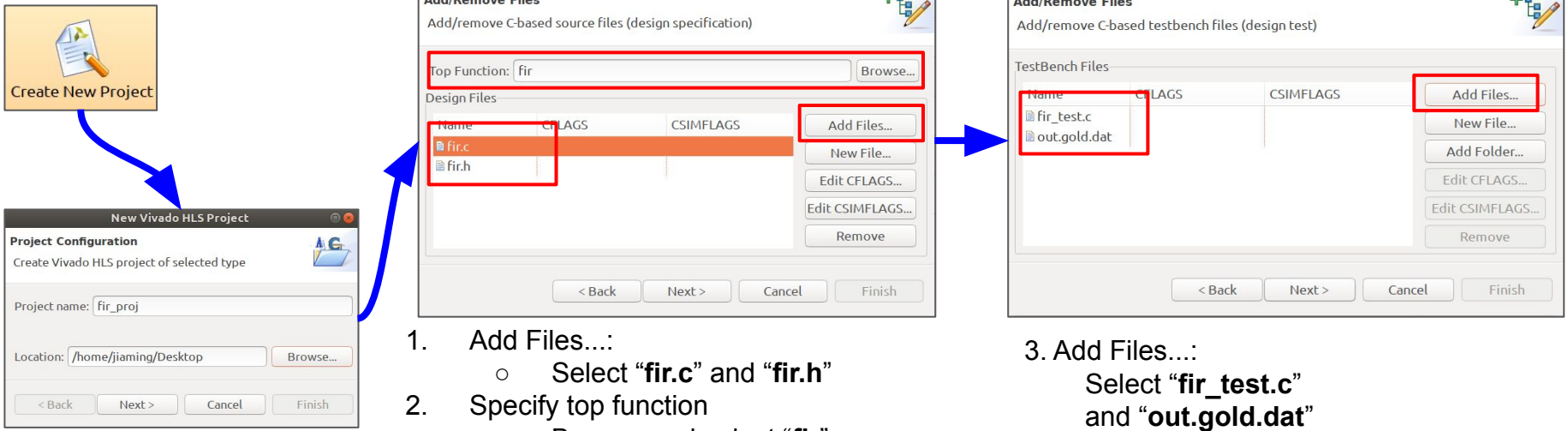
- Prepare lab files
  - Download the files from here.
  - Four files in the package,
    - **(1) fir.c**: design implementation.
    - **(2) fir.h**: header file, define data types, functions
    - **(3) fir_test.c**: compare results from hardware and software
    - **(4) out.gold.dat**: golden data
      - Column 1: sample index
      - Column 2: input sample value
      - Column 3: output of FIR




```
0 1 0
1 2 -10
2 3 -29
3 4 -25
4 5 35
5 6 158
6 7 337
7 8 539
8 9 732
```

Lab 3-1: Practice with small filter size N=11

- Open Vivado HLS 2020.1
- Create New Project
  - Add the downloaded files into project

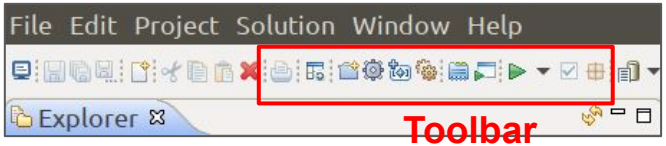


1. Add Files...:
   - Select “**fir.c**” and “**fir.h**”
2. Specify top function
   - Browse and select “**fir**”

3. Add Files...:
   Select “**fir_test.c**”
   and “**out.gold.dat**”

Lab 3-1: Practice with small filter size N=11

- C-Simulation and C Synthesis
  - On the toolbar, click 



Analysis Tab

Synthesis Report for 'fir'

General Information

Date: Wed Mar 10 14:55:56 2021
Version: 2020.1 (Build 2897737 on Wed May 27 20:21:37 MDT 2020)
Project: fir_proj
Solution: solution1
Product family: zynq
Target device: xc7z020-clg400-1

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.978 ns	1.25 ns

Latency

Summary

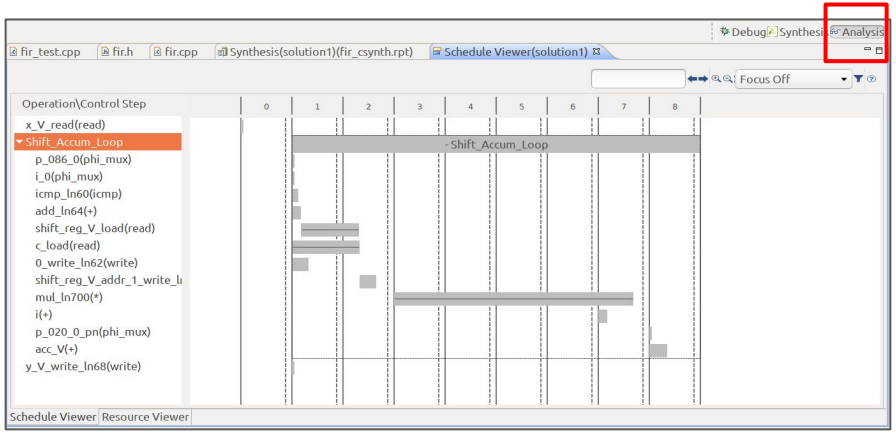
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
34	89	0.340 us	0.890 us	34	89	none

Detail

Instance

Loop

Synthesis Report



Schedule Viewer

Lab 3-1: Practice with small filter size N=11

- Try the optimization techniques learned today
  - Loop Unrolling
  - Loop Pipelining
  - Avoid branch condition
- Try to read the synthesis report and schedule viewer

Lab 3-2: Performance tuning for larger filter size N=512.

- Change N = **512** in the header file
  - 512-tap FIR filter
  - Ignore C-simulation check with testbench, we don't have the data yet.
- **To get today's full credit, try your best to suppress my result.**
  - Lower latency
  - Reasonable resource consumption
- Hints:
  - Apply all the optimization techniques we learned today
  - **Relation**:
    Behavior "Shift" on "shift_reg" → array partition
  - Look into "Schedule Reviewer" to check the schedule is exactly matched with your thoughts
  - *Considering "symmetry" property of filter(optional)

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)	
min	max	min	max	min	max
129	129	1.290 us	1.290 us	129	129

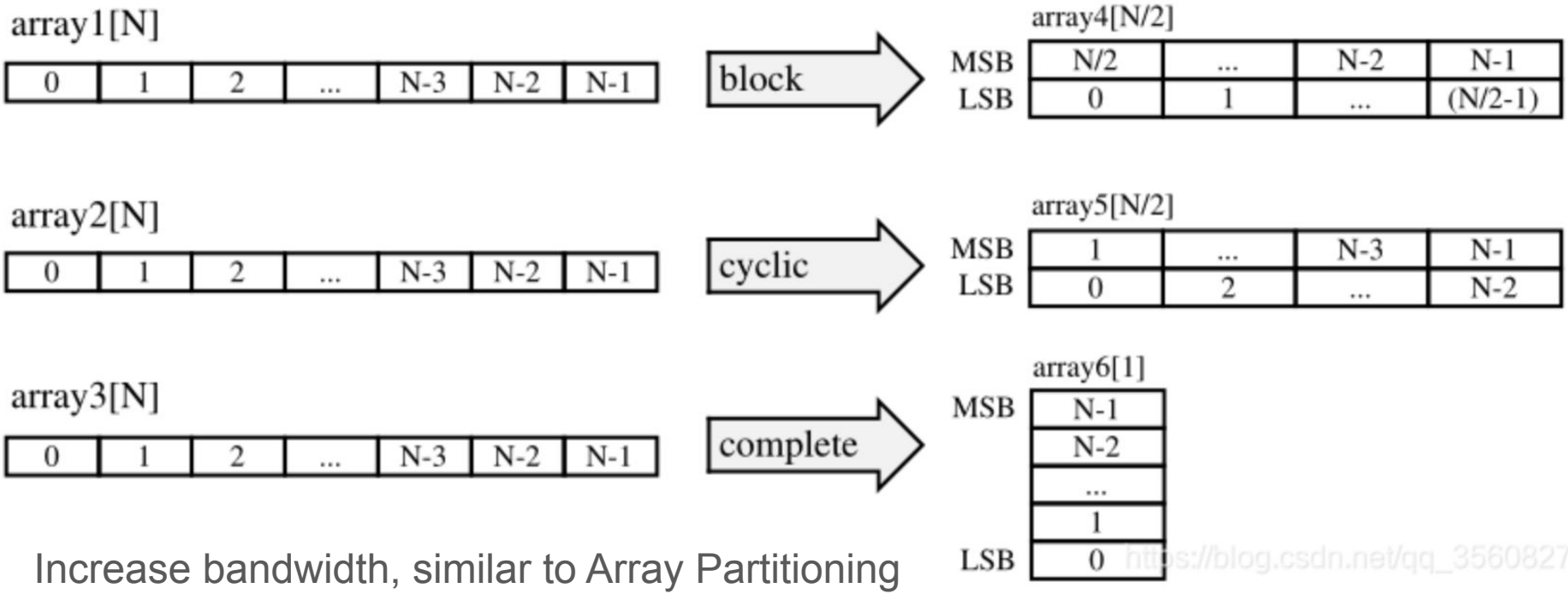
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	13	-	-	-
Expression	-	1	0	15562	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	0	-	270	72	0
Multiplexer	-	-	-	216	-
Register	-	-	16963	-	-
Total	0	14	17233	15850	0
Available	280	220	106400	53200	0
Utilization (%)	0	6	16	29	0

Summary

- Understanding the algorithm, application workload
  - Then we can design good accelerator.
- Optimization Techniques
  - Loop Unrolling, Loop Pipelining, Avoid branch condition
- When considering parallelization, don’t forget the memory bandwidth support.
- Next time
  - Introduction to the interfaces(AXI streaming), so that we can port the design on PYNQ-z2

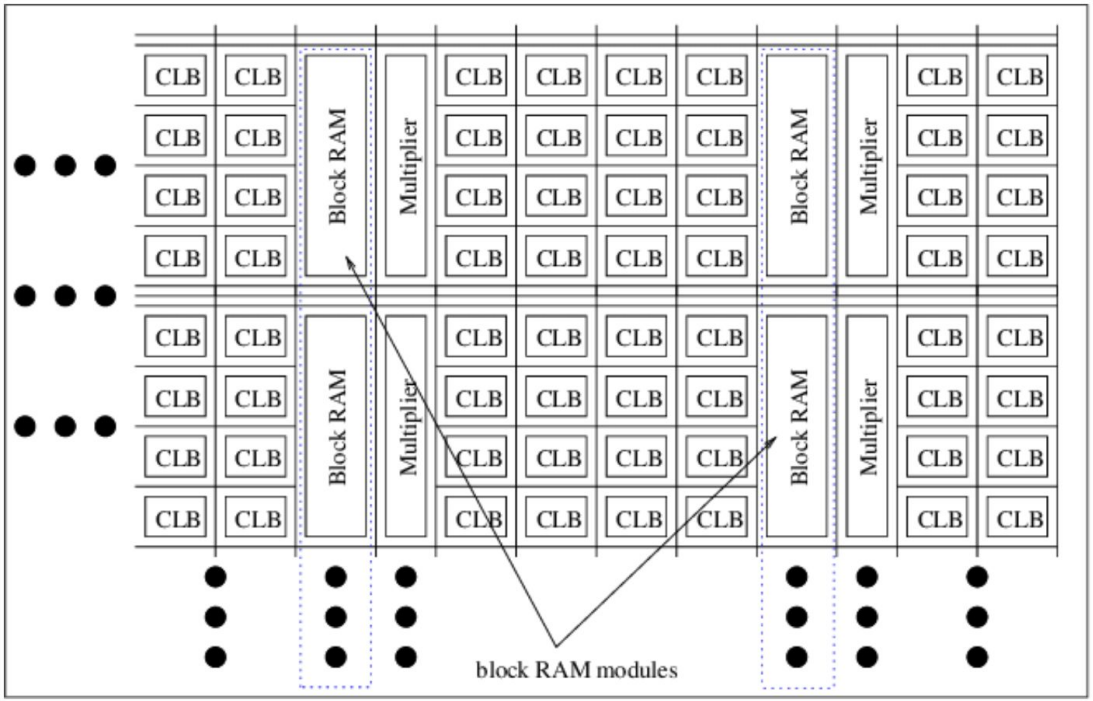
Appendix: Array Reshape v.s. Array Partition

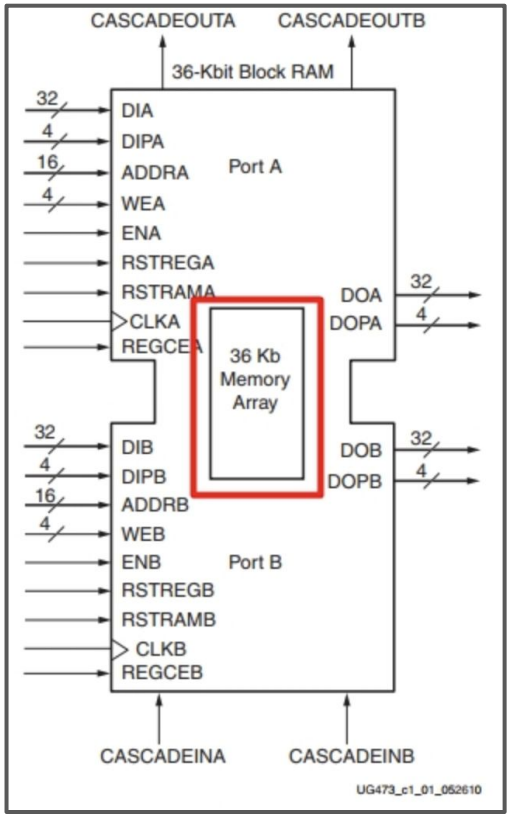
Array Reshape



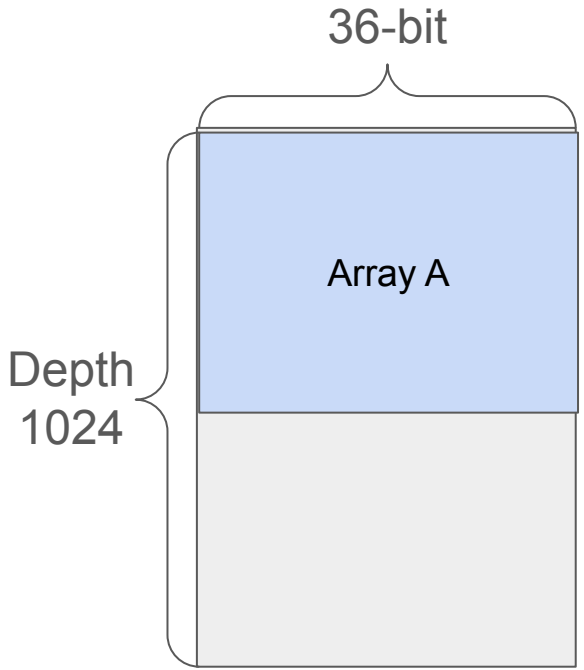
Increase bandwidth, similar to Array Partitioning
And has higher BRAM utilization

Block RAM in FPGA



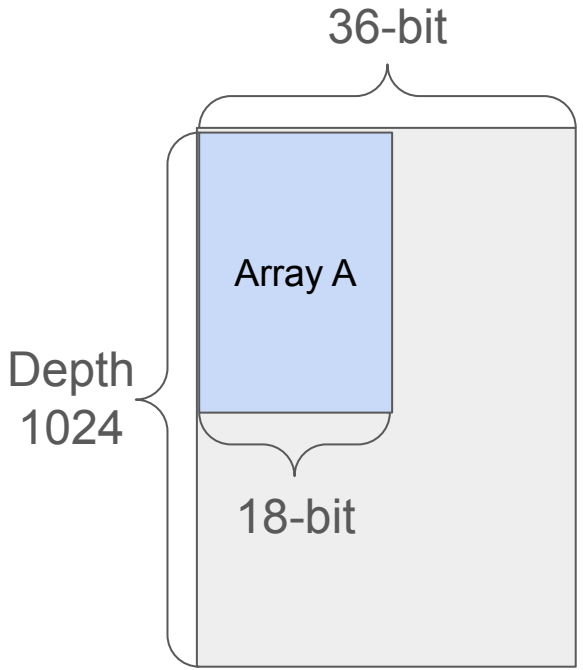


Mapping Data on Block RAM



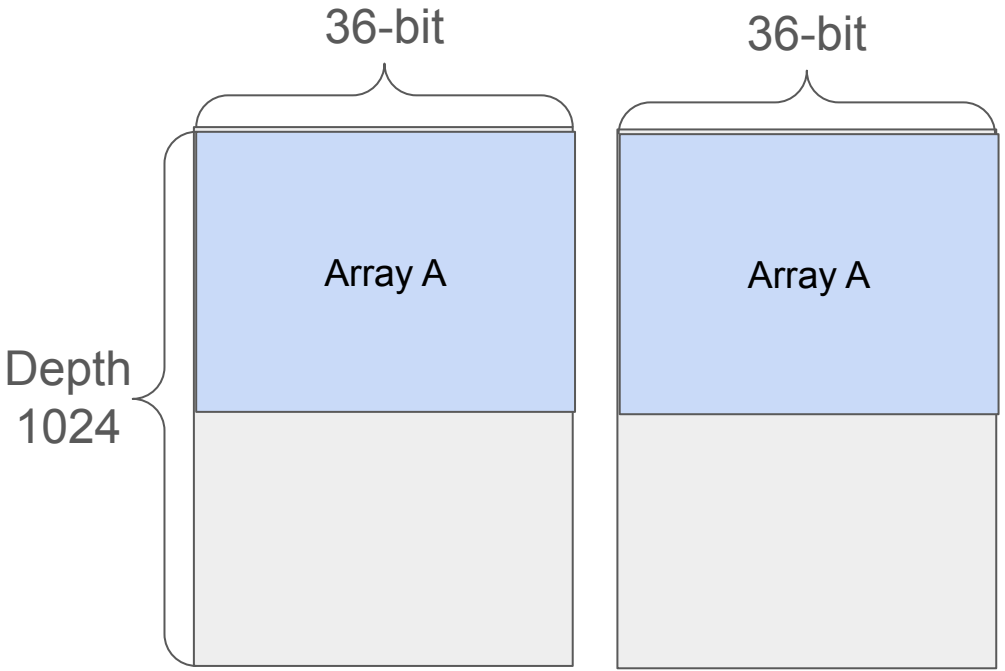
- For array A
  - Width = 36-bit
  - Depth = 512

Mapping Data on Block RAM



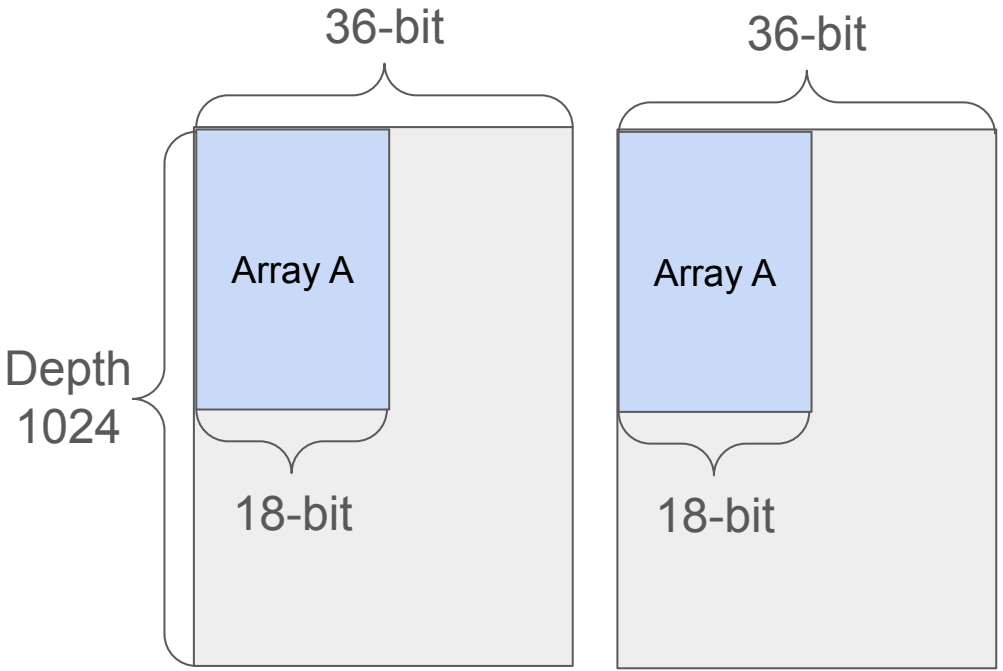
- For array A
  - Width = 18-bit
  - Depth = 512

Mapping Data on Block RAM



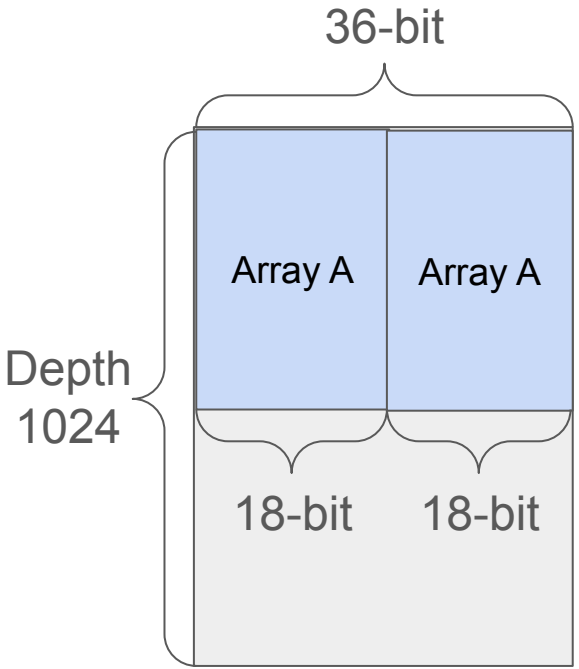
- For array A
  - Width = 72-bit
  - Depth = 512
- Combine two BRAMs to obtain higher width.

Mapping Data on Block RAM, using Array Partitioning



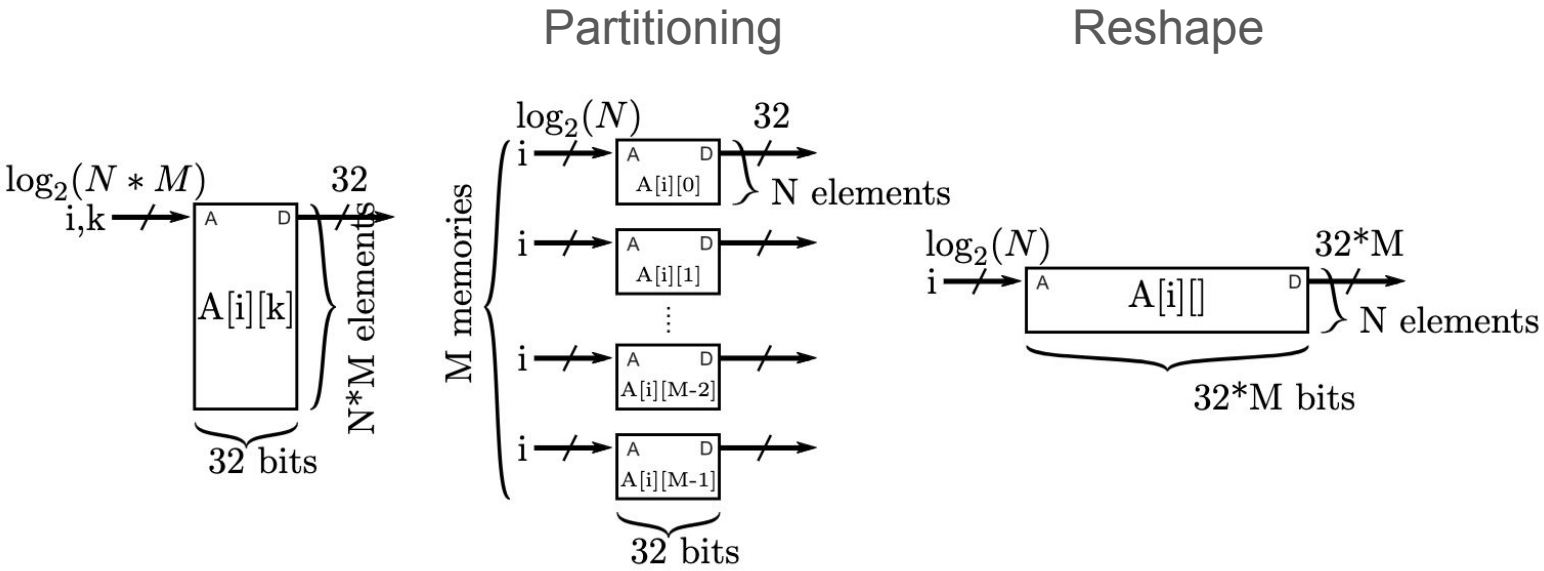
- For array A[512][2]
  - Data type = ap_int<18>
  - Total width = 36-bit
  - Depth = 512
- Using Array Partitioning
- Cost #BRAM = 2

Mapping Data on Block RAM, using Array Reshape



- For array A[512][2]
  - Data type = ap_int<18>
  - Total width = 36-bit
  - Depth = 512
- Using Array Reshape
- Cost #BRAM = 1

Advantage of Partitioning over Reshape



Partitioning is more flexible than Reshape,

- Allow to access to different addresses concurrently