Bridge of Life Education

Advanced SOC Design

Memory Architecture

Jiin Lai

Topics

- FPGA Memory (Recap)
- DDR Memory
- Memory Restructure
- Memory Access Transformation
- Memory-based Shifter
- Memory Hierarchy

FPGA Memory Memory

Topics

- CLB/LUT: Distributed RAM (64x1b or 32x2b)
- CLB: Flip-flops
- Block RAM - FIFO
- Ultra RAM
- HBM
- DDR Memory

	UltraScale+	UltraScale	7 Series	Spartan-6
Block RAM	36Kb	36Kb	36Kb Block RAM	18Kb Block RAM
UltraRAM	288Kb	-	-	-
High Bandwidth Memory	16GB	-	-	-
External Max Data Rate	2667Mb/s	2400Mb/s	1866Mb/s	800Mb/s
Serial Memory	HMC Bandwidth Engine 2 Bandwidth Engine 3	HMC Bandwidth Engine 2 Bandwidth Engine 3	Bandwidth Engine 2	

CLB Resources

- 8 x 6-input LUT
- 16 x flip-flops
- 8-bit carry-chain CARRY8
- Wide-multiplexers combine LUTs to FMUX- 7, 8, 9 inputs, 55 inputs
- A LUT in SLICEM
  - A loop-up table
  - 64-bit distributed RAM
  - 32-bit shift registers

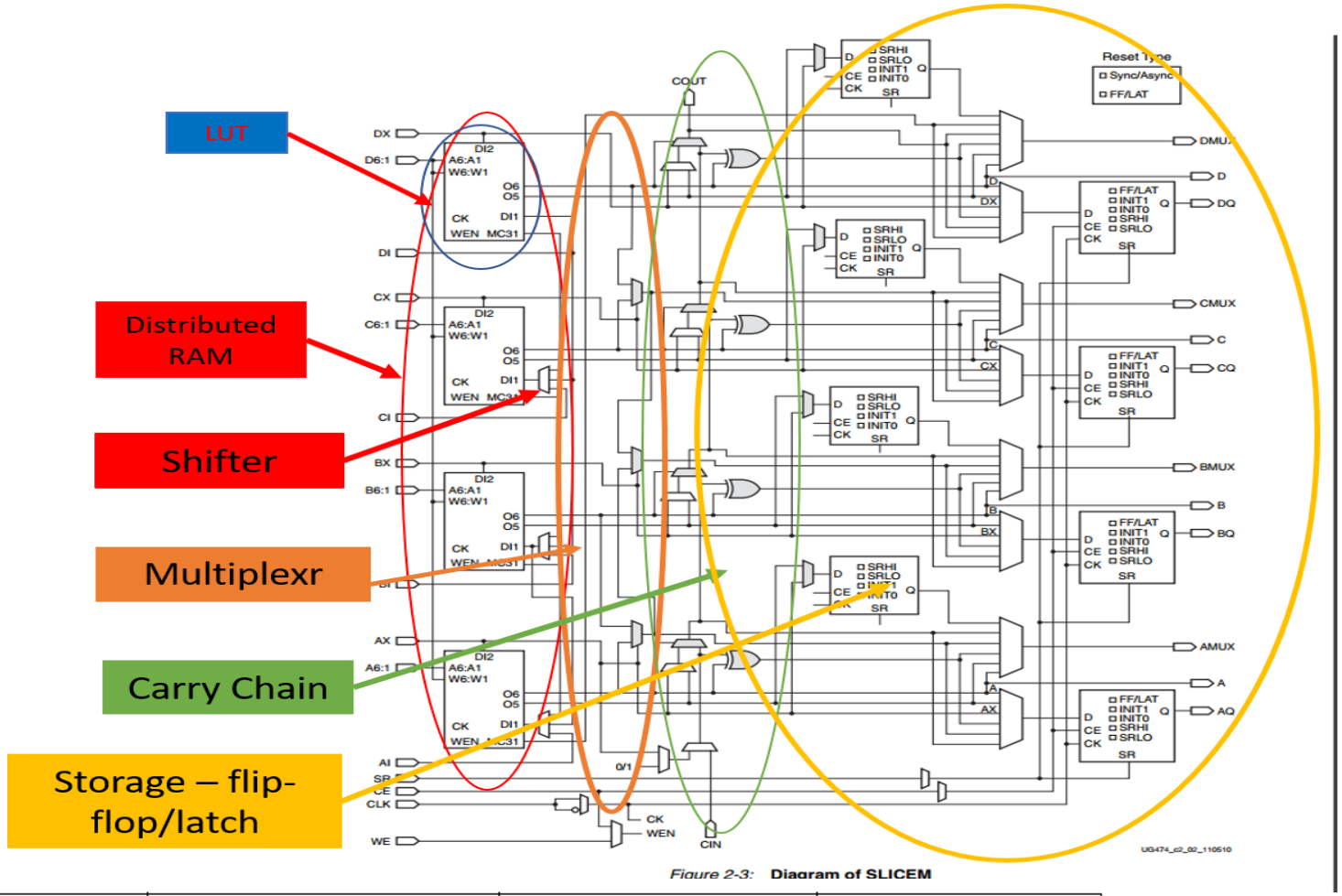


Figure 2-3: **Diagram of SLICEM**

CLB Slice	LUTs	Flip-Flops	Arithmetic and Carry Chains	Wide Multiplexers	Distributed RAM	Shift Registers
SLICEL	8	16	1	F7, F8, F9	N/A	N/A
SLICEM	8	16	1	F7, F8, F9	512 bits	256 bits

LUT as Distributed RAM, ROM

- One LUT - 64x1 or 32x2
- SliceM = 8 LUT => 512bit
- Combine multiple Slice-M

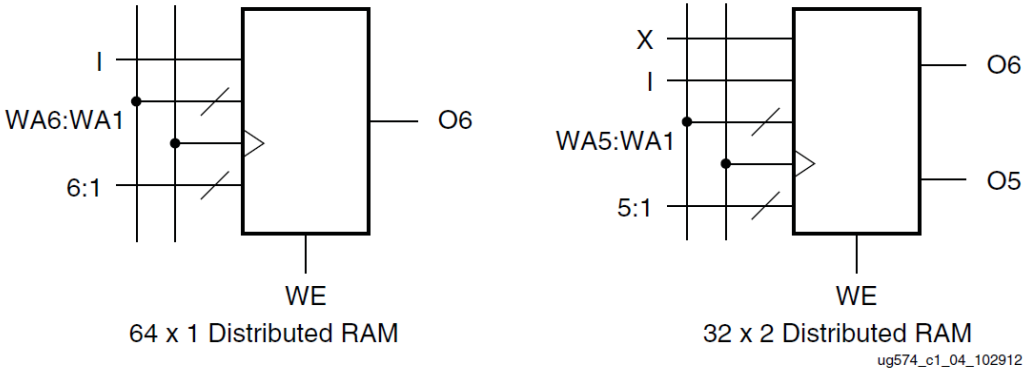


Figure 1-4: **Distributed RAM Using Look-Up Tables**

Table 2-2: **ROM Configuration**

ROM	Number of LUTs
64 x 1	1
128 x 1	2
256 x 1	4
512 x 1	8

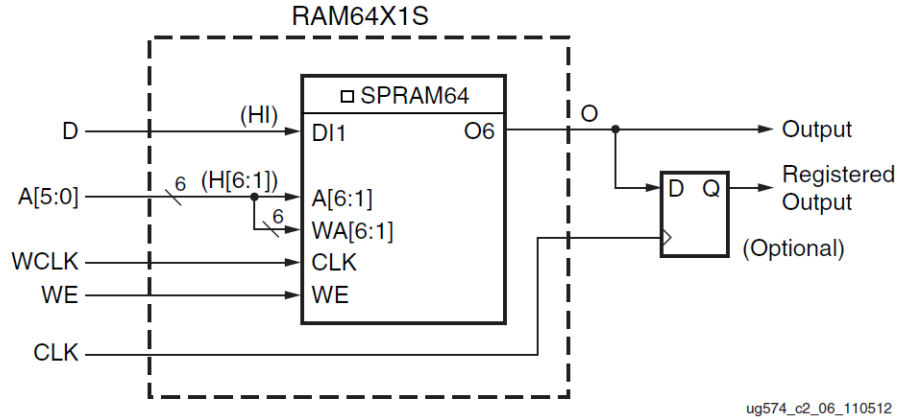
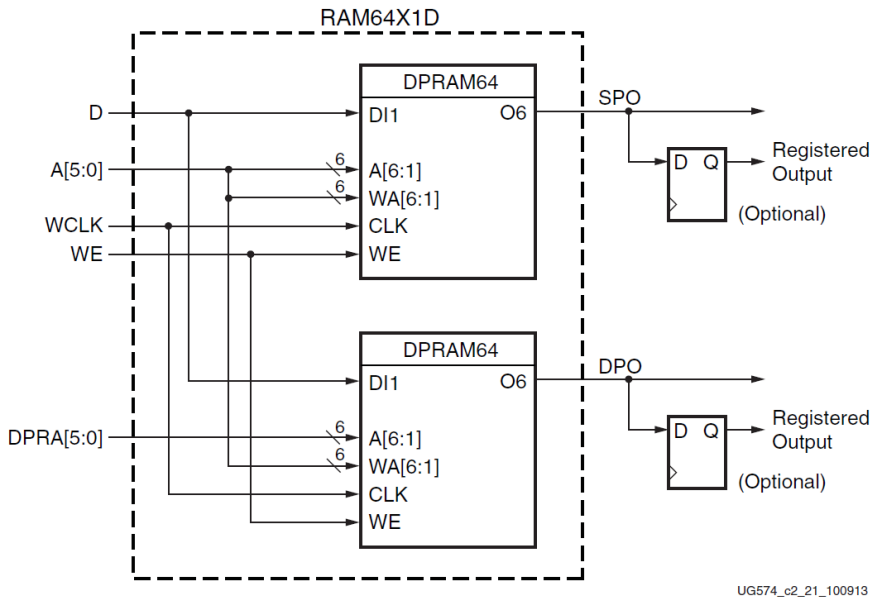


Figure 2-5: **Distributed RAM (RAM64X1S)**



Distributed RAM Configurations

Distributed RAM Configurations (SLICEM): port * depth * width <= 512 bit

- Single-port 32 x (1 to 16)-bit RAM
- Dual-port 32 x (1 to 8)-bit RAM
- Quad-port 32 x (1 to 4)-bit RAM
- Simple dual-port 32 x (1 to 14)-bit RAM
- Single-port 64 x (1 to 8)-bit RAM
- Dual-port 64 x (1 to 4)-bit RAM
- Quad-port 64 x (1 to 2)-bit RAM
- Octal-port 64 x 1-bit RAM
- Simple dual-port 64 x (1 to 7)-bit RAM
- Single-port 128 x (1 to 4)-bit RAM
- Dual-port 128 x 2-bit RAM
- Quad-port 128 x 1-bit RAM
- Single-port 256 x (1 to 2)-bit RAM
- Dual-port 256 x 1-bit RAM
- Single-port 512 x 1-bit RAM

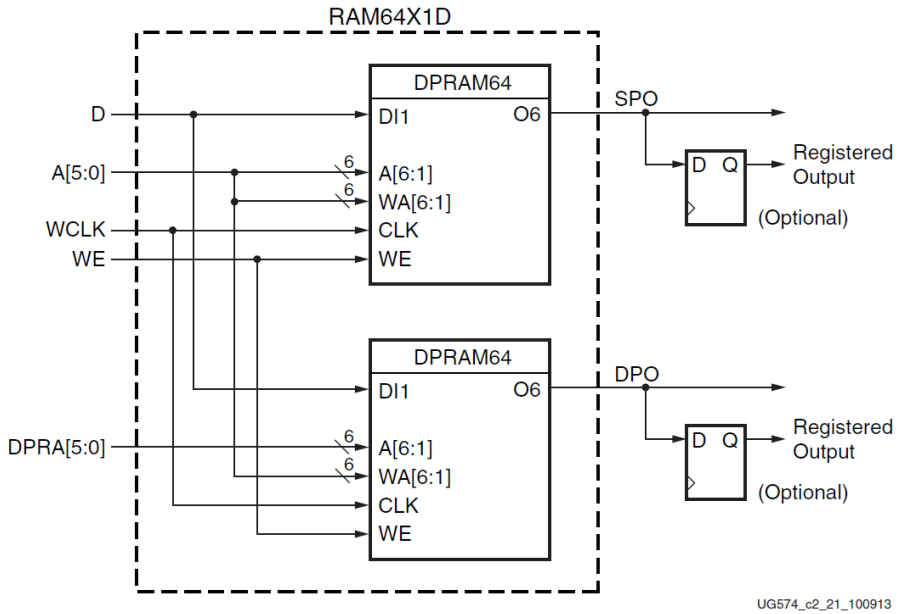


Figure 2-6: **64x1 Dual-Port Distributed RAM (RAM64X1D)**

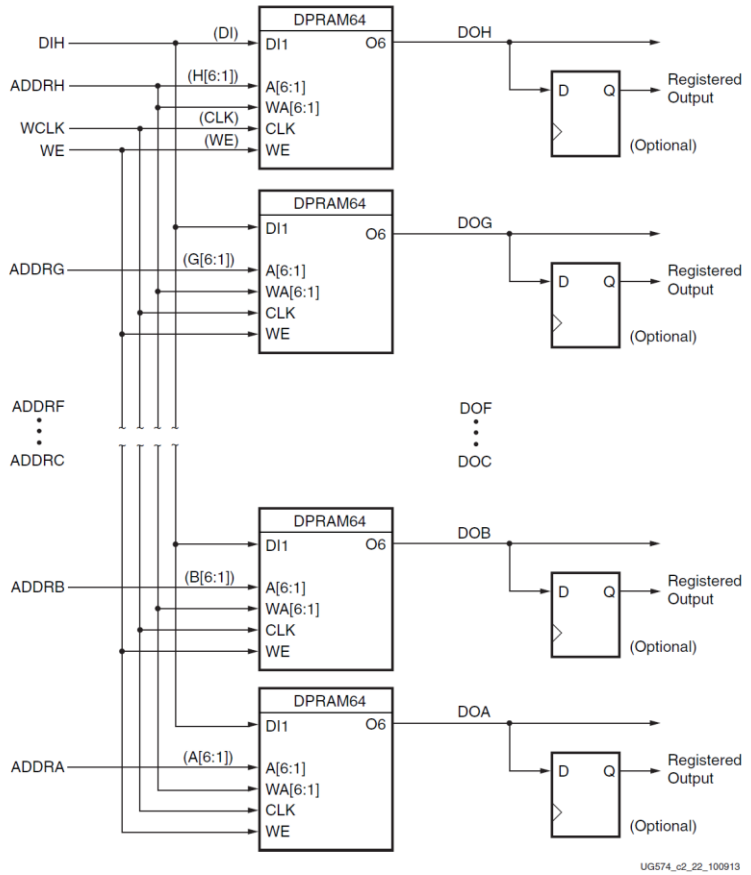


Figure 2-7: **64x1 Octal Port Distributed RAM**

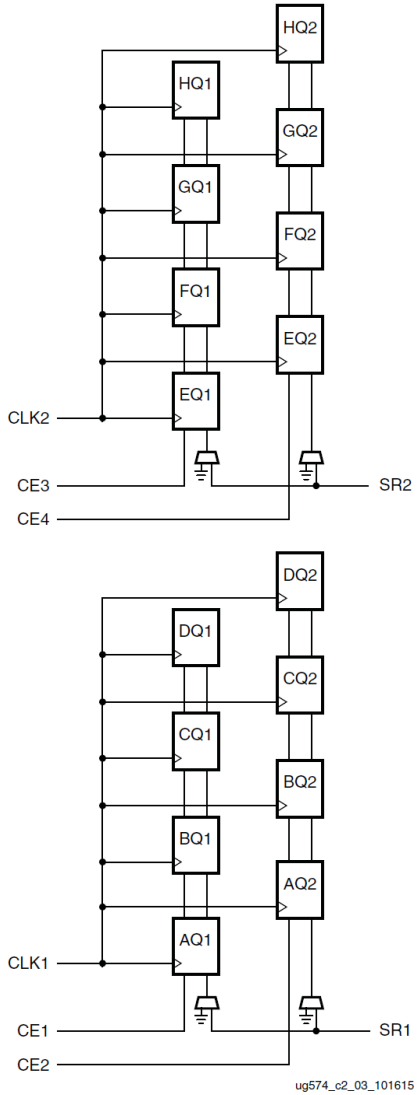
Storage Elements (Flip-flop)

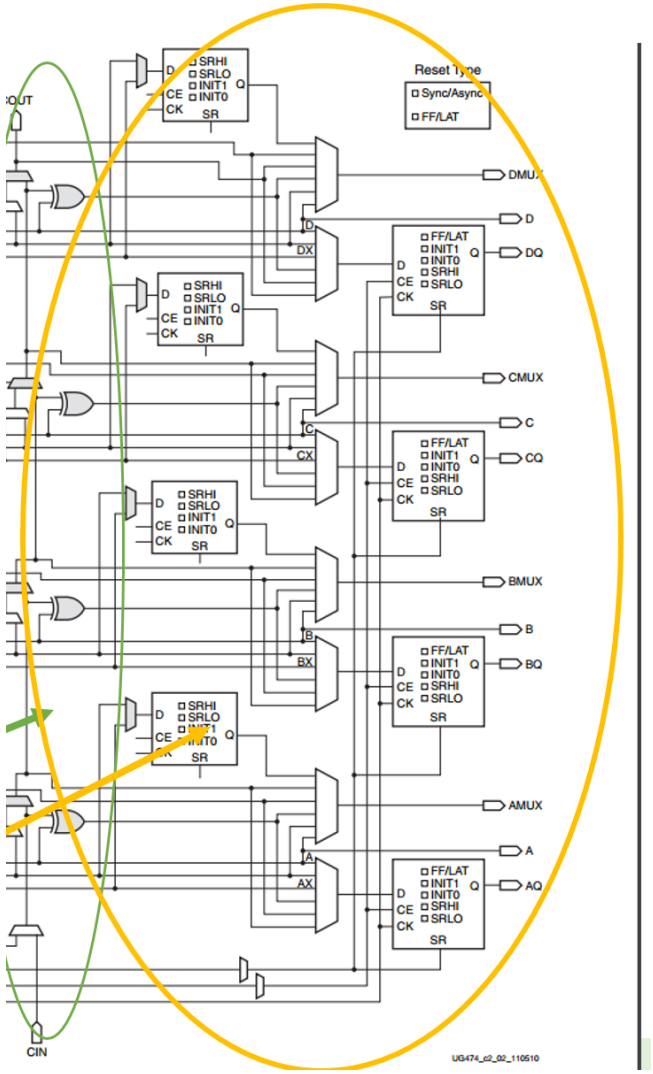
Input Sources:

- The LUT O6 output.
- The LUT O5 output.
- A direct input from the CLB inputs that bypasses the LUT (BYP). CLB X input for Q1, CLB I input for Q2.
- The carry XOR result.
- The carry cascade output (CO) at that bit.
- The local one of the seven wide multiplexers (FMUX). This is not available on the bottom LUT A.

Control Signals

- Clock (2 sets)
- Clock Enable (4 sets)
- Initialization – SR set/reset (Synchronous/Asynchronous)



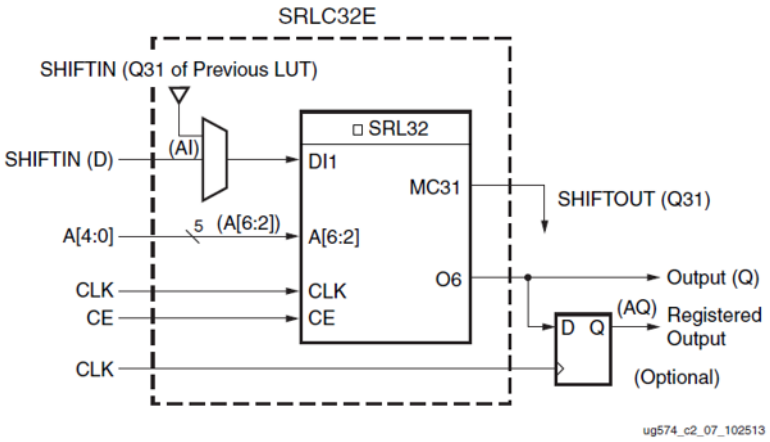


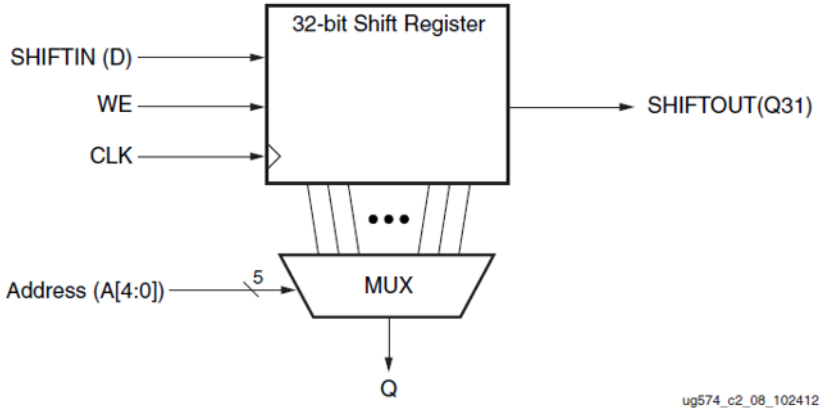
Shift Registers (SLICEM only)

- SLICEM = 256bit

Usages

- Synchronous FIFO
  - dataflow stream channel
- Delay or Latency compensation
  - balance latency difference for re-converged paths
- Content-Addressable Memory
- Cascade to form larger shift registers





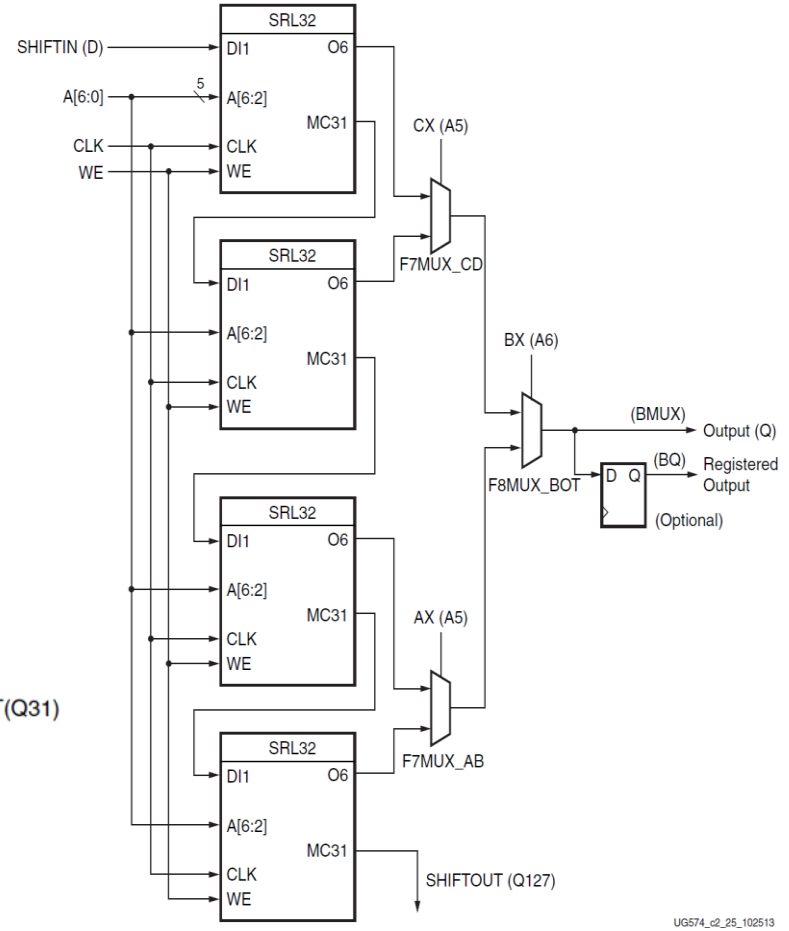


Figure 2-13: **128-bit Shift Register Configuration**

Block RAM

- **36Kb storage area + 2 completely independent access ports (each 18kb)**
- 36Kb: Width: 36K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18, 1K x 36, 512 x 72 (if simple Dual-port)
- 18Kb: Width: 16K x 1, 8K x 2, 4K x 4, 2K x 9, 1K x 18, 512 x 36 (used as SDP)
- Memory content can be initialized by configuration bitstream
- Optional Output Registers (better timing)
- Independent Read and Write Port Width Selection
- **Simple Dual-Port BlockRAM**, **True Dual port**
- Cascadable Block RAM (more depth)
- Byte-Wide Write Enable (for MCU)
- Error Correction (reliability)
- FIFO (built in sequencing, flag … save CLB resource)
- Read-during-write function

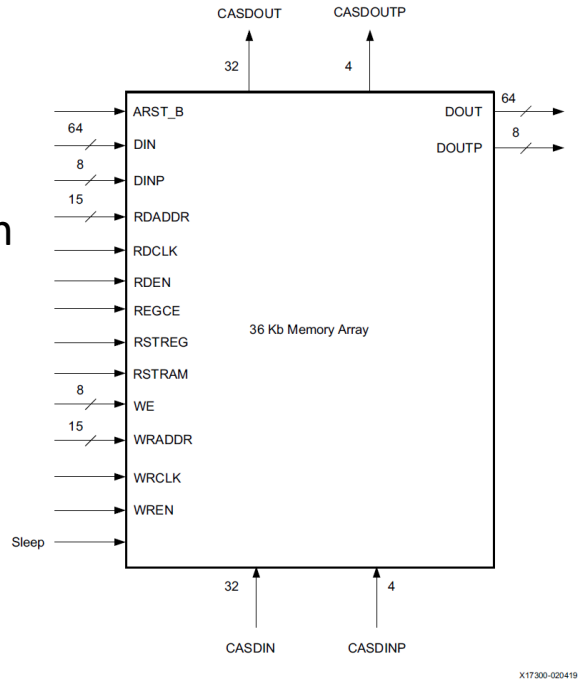


Figure 1-6: **RAMB36 Usage in a Simple Dual-Port Data Flow**

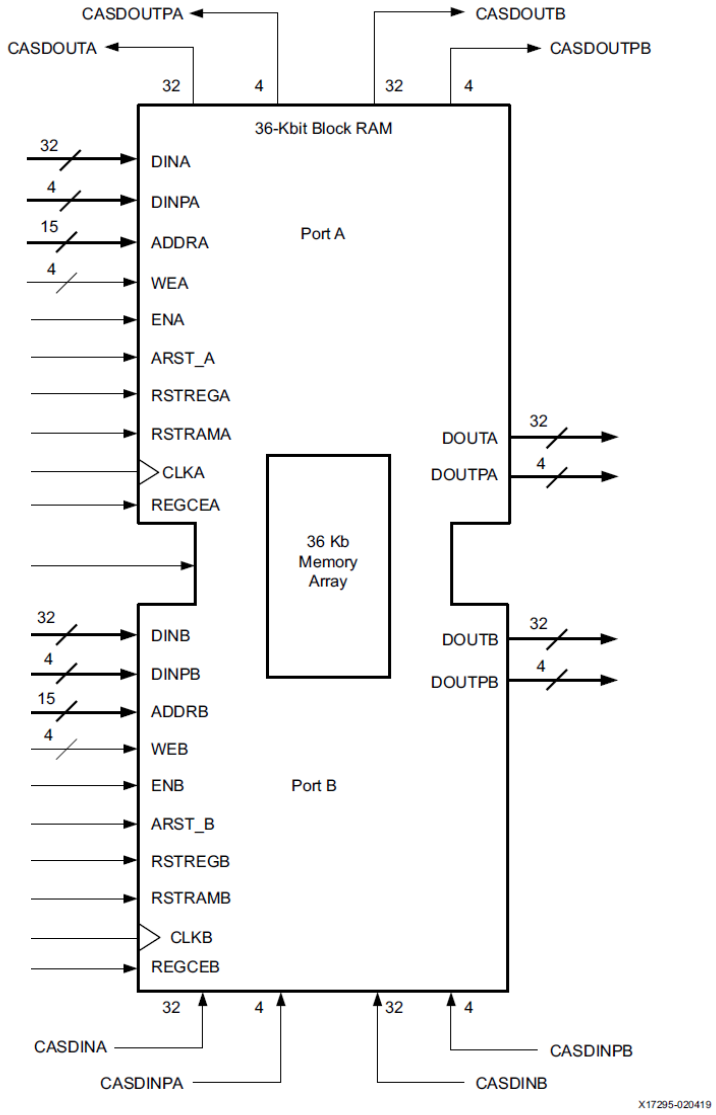


Figure 1-1: **RAMB36 Usage in a True Dual-Port Data Flow**

https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf

Optional Output Registers

```
#pragma HLS bind_storage variable=xxx type=RAM_1P impl=bram latency=2
```

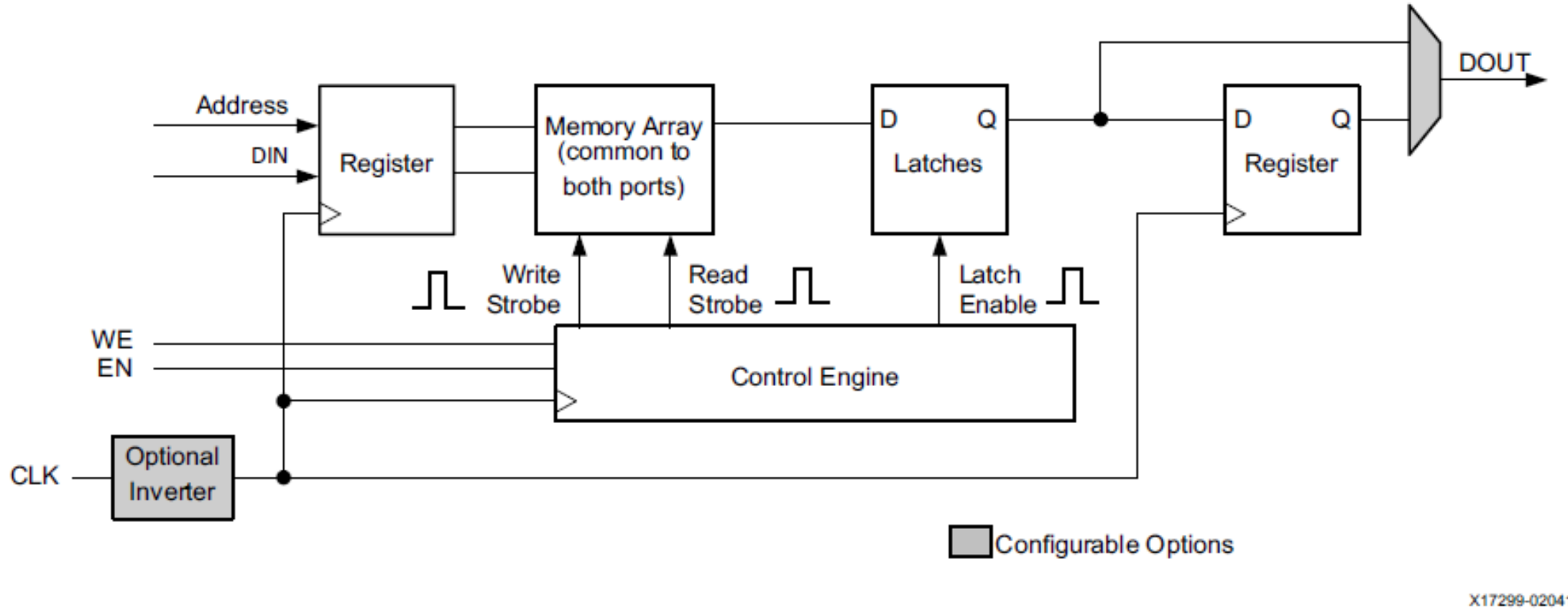


Figure 1-5: **Block RAM Logic Diagram (One Port Shown)**

Cascadable Block RAM

- Standard Data Out Cascade
  - Impact final clock-to-out performance
- Data Out cascade in Pipeline Mode
  - Last stage select register out
- Cascaded signals have dedicated interconnects within a block RAM column

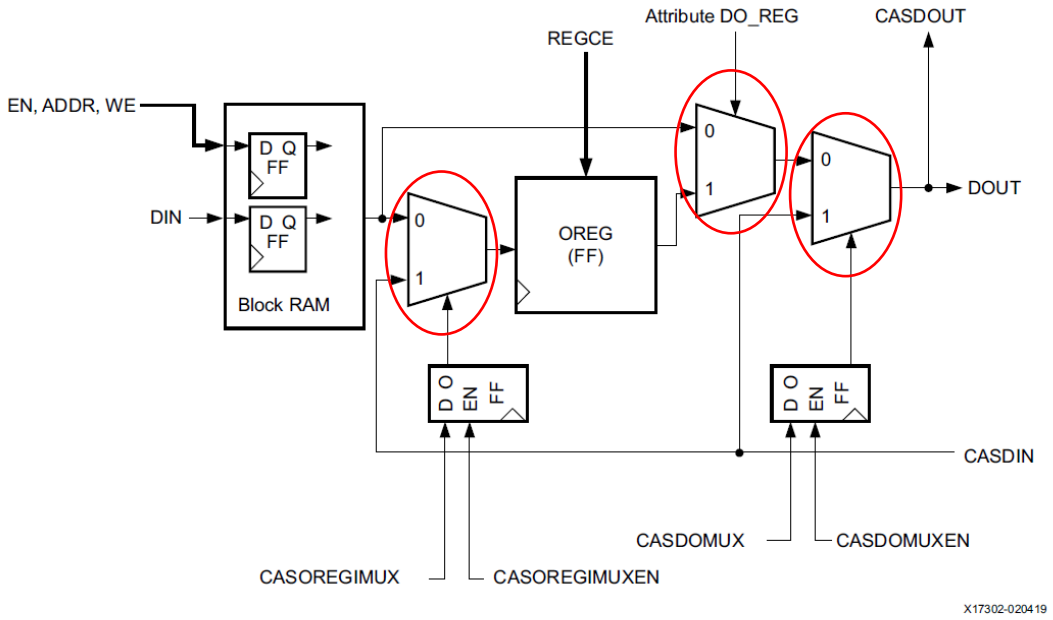


Figure 1-8: **Cascade Functional Diagram**

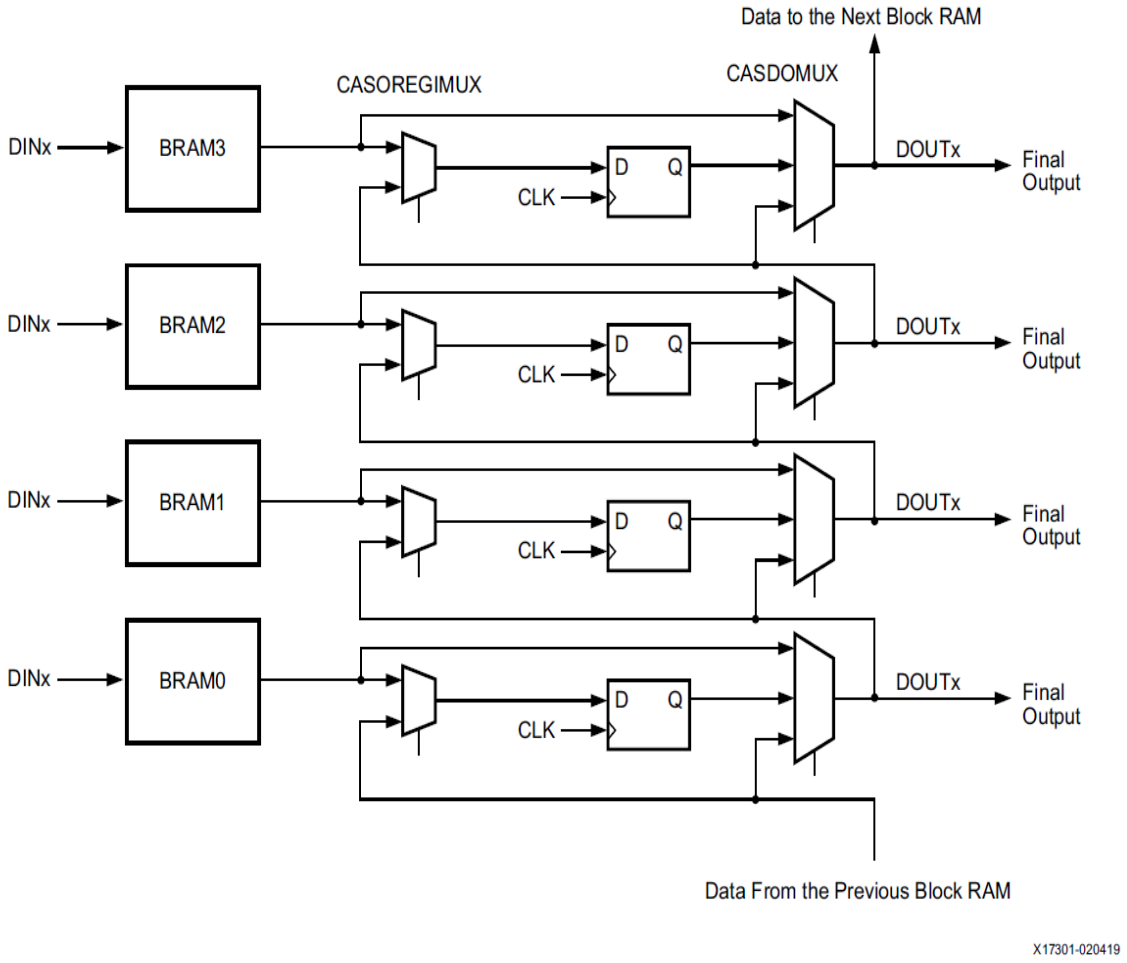


Figure 1-7: **High-level View of the Block RAM Cascade Architecture**

Error Correction Code- Single bit correction , Double-bit detection

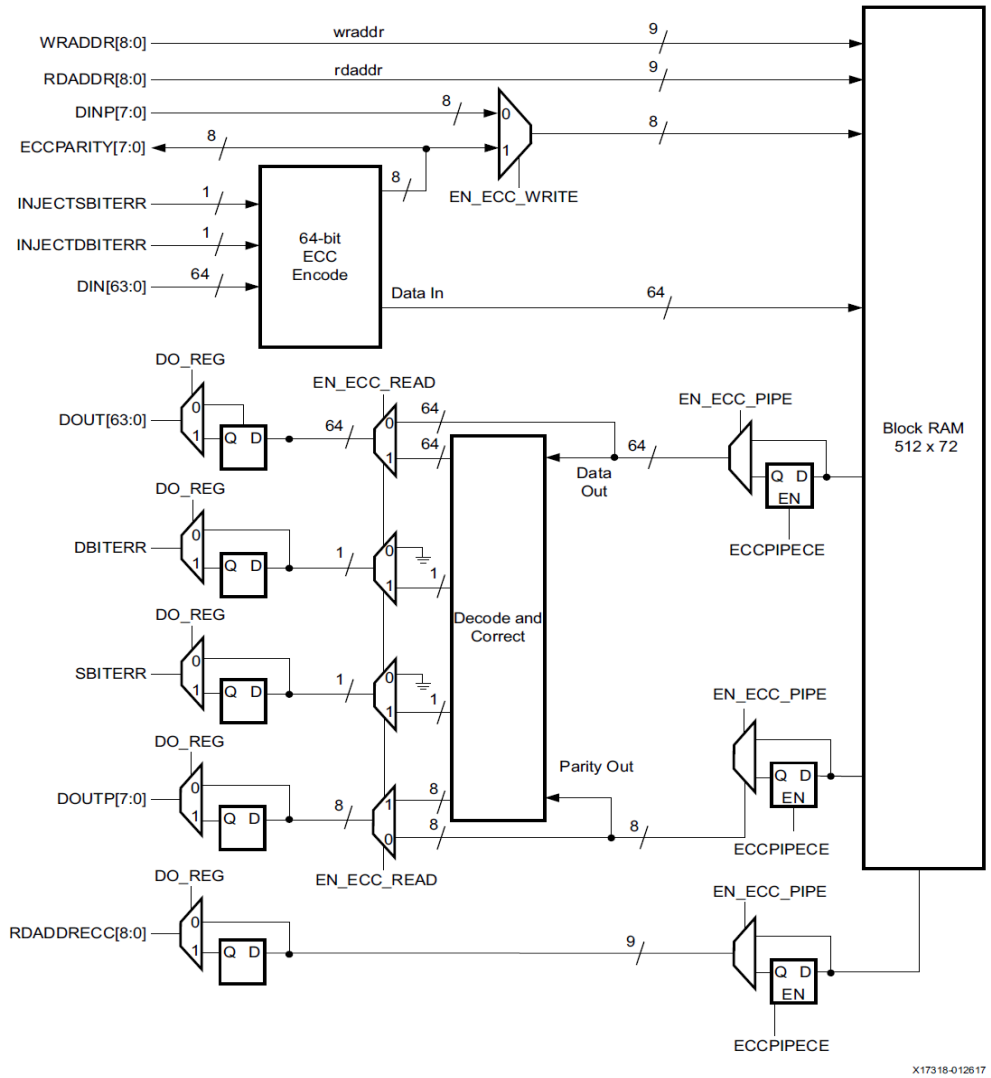


Figure 1-32: **Top-Level View of Block RAM ECC**

FIFO

- Common-clock or independent-clock
  - Avoid ambiguity, glitches, meta-stability
  - Pass data across different clock domains
- Standard or first-word fall-through (FWFT)
- Port A as FIFO read port, Port B as FIFO write port
- 18Kb (FIFO18E2): 4Kx4, 2Kx9, 1Kx18, 512 x36
- 36Kb (FIFO36E2): 8Kx4, 4Kx9, 2Kx18, 1Kx35, 512 x72

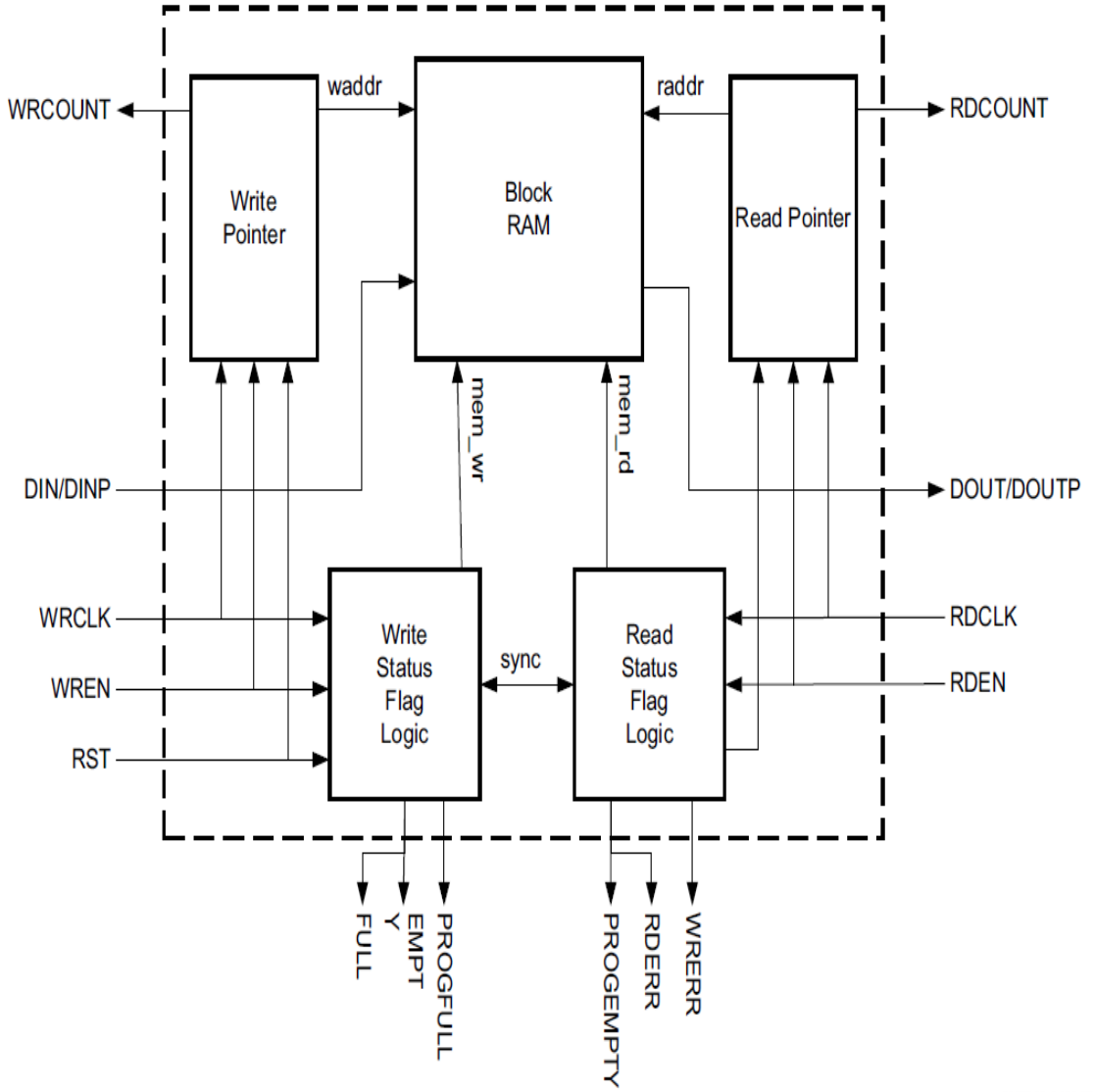


Figure 1-20: **Top-Level View of FIFO in Block RAM**

Cascade FIFO

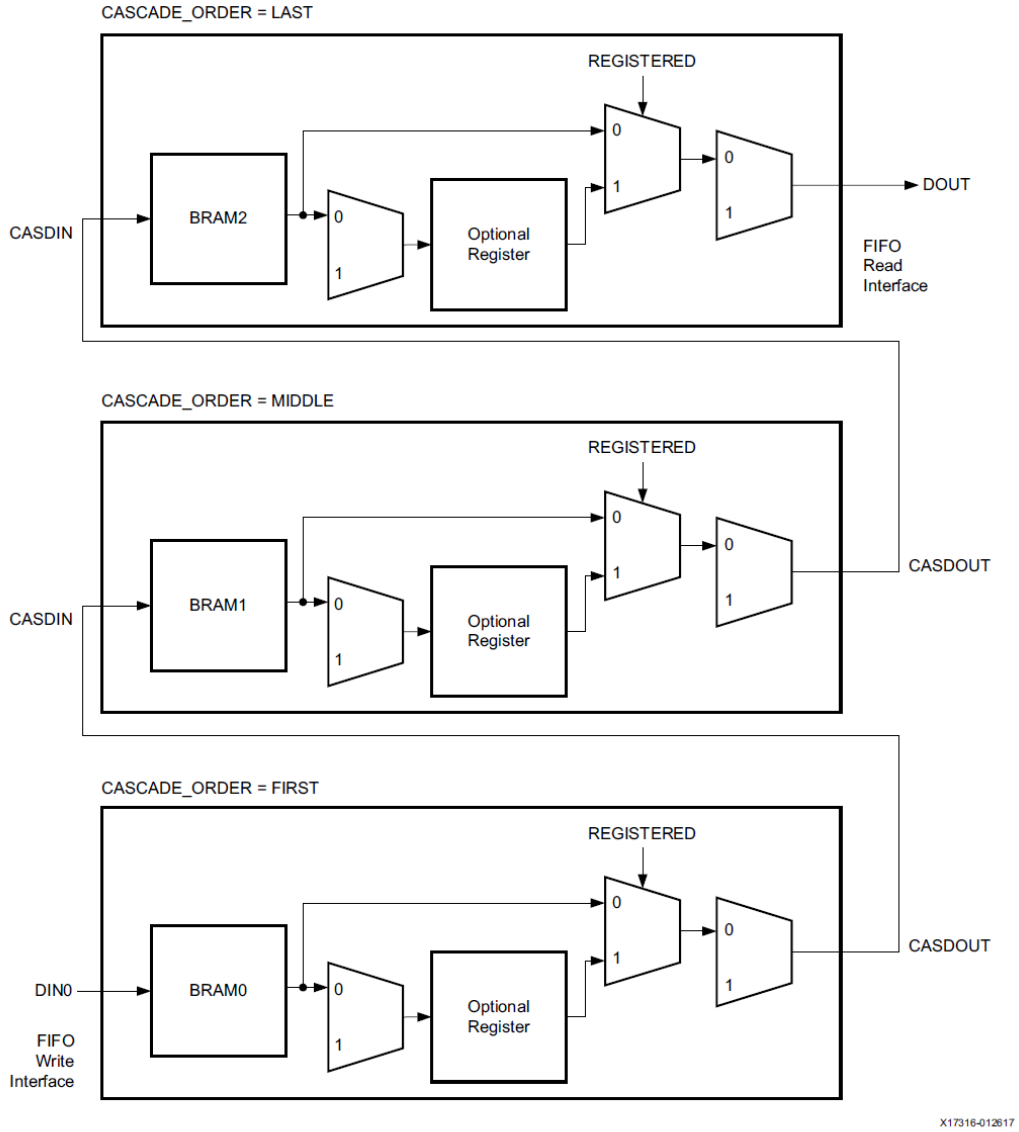
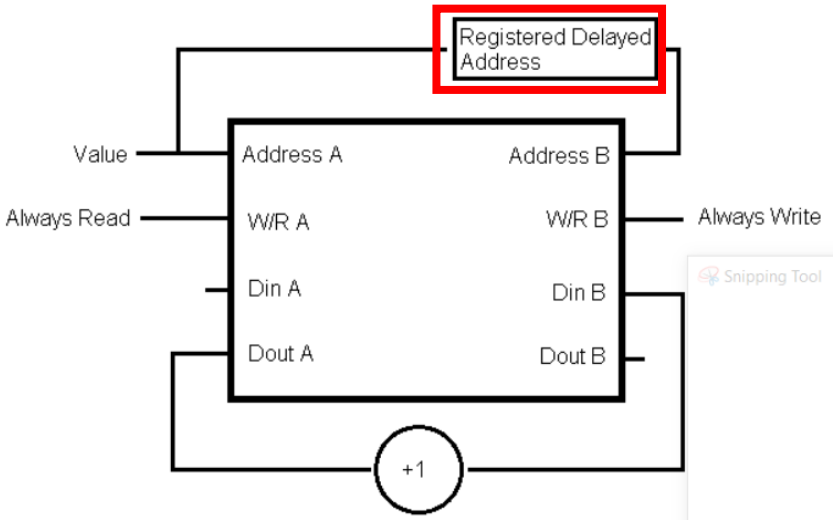


Figure 1-30: **FIFO Serial Cascade**

Creative Use of BlockRAM

- Read-Modify-Write, One Operation Per Clock
  - Dual-port – PortA as the read port, PortB as the write port
  - Use one common clock for both ports
- Shift Registers (FIFO)



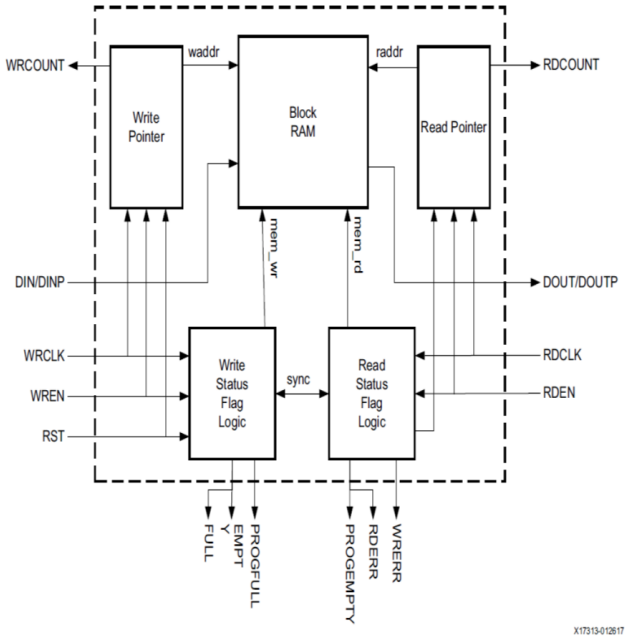


Figure 1-20: **Top-Level View of FIFO in Block RAM**

wp335 - https://docs.xilinx.com/v/u/en-US/wp335

Creative Use of BlockRAM : State Machines

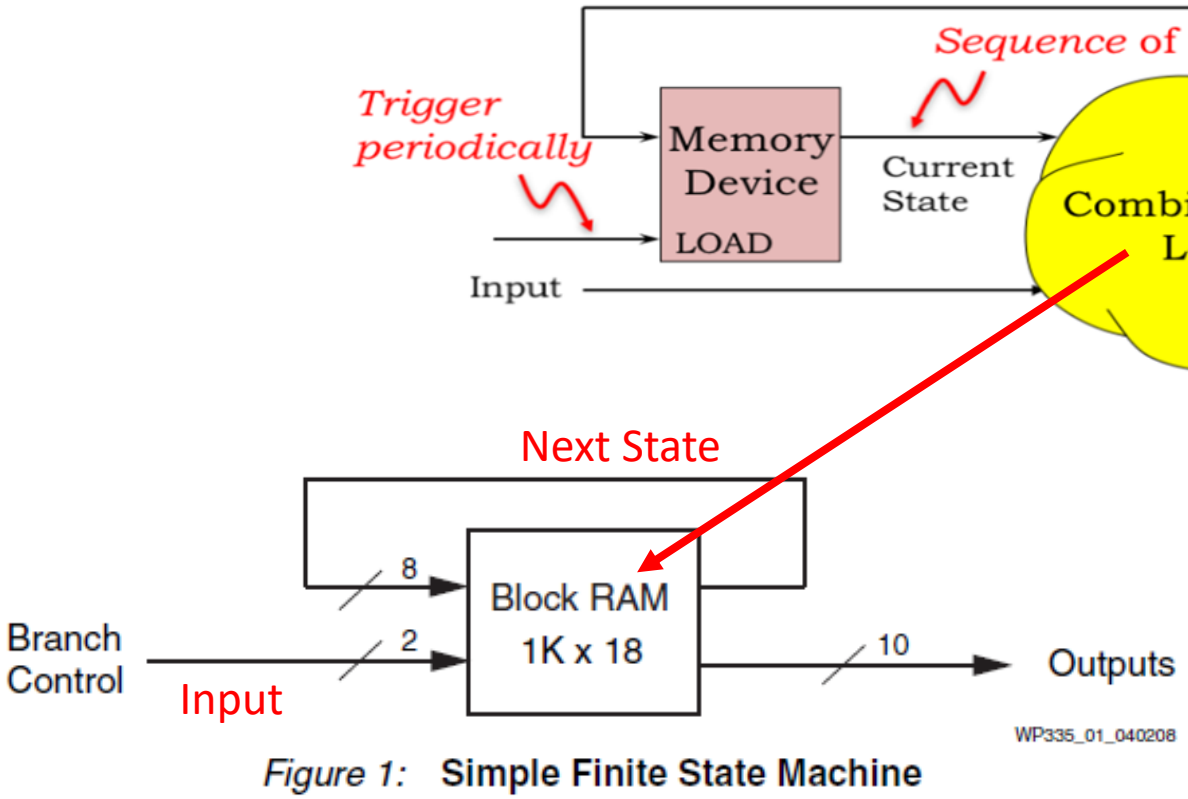


Figure 1: **Simple Finite State Machine**

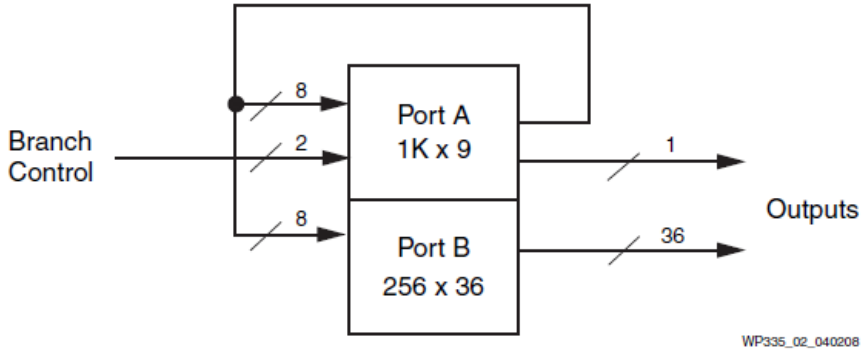


Figure 2: **FSM with Additional Outputs**

wp335 - https://docs.xilinx.com/v/u/en-US/wp335

UltraRAM

- 288kb (8xBRAM) – Single Dual port 4K x 72
- No FIFO
- Single clock – 2 ports
  - Each port can operate read/write independently
  - Port A take precedence over Port B
- Cascadable from 288kb to 300Mb – replace external SRAM
- ECC
- Optional pipeline flip-flops (input, output)
- Initialized to all 0's

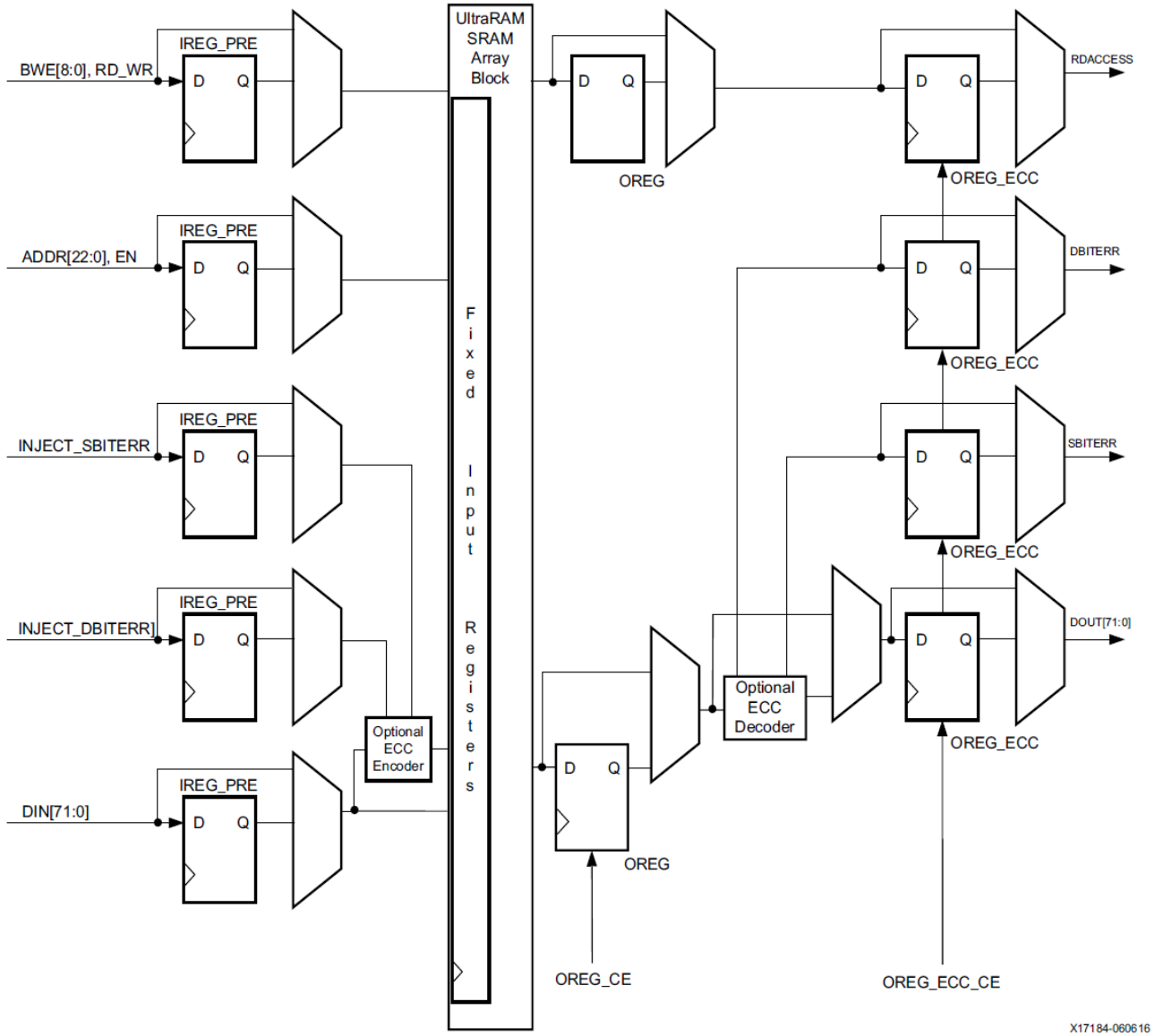


Figure 2-3: **Simplified Single UltraRAM Block Diagram without Cascade (One Port shown)**

UltraRAM https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf
https://www.xilinx.com/products/technology/memory.html

Cascade UltraRAMs to build deep memory

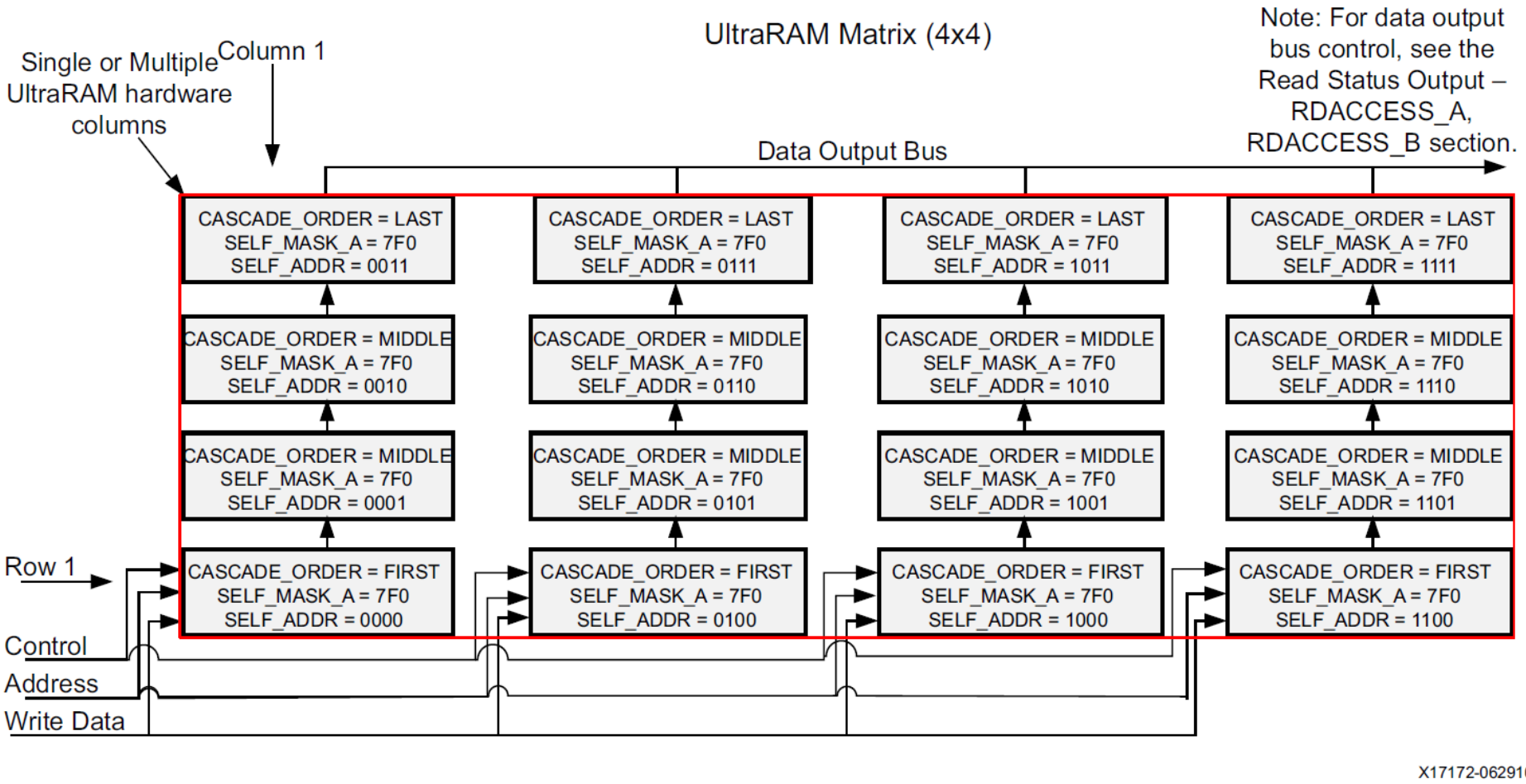


Figure 2-7: **4x4 UltraRAM Matrix**

HBM

HBM – High Bandwidth Memory

Memory bandwidth does not scale with application

- Network bandwidth: 10Gb -> 25Gb -> 40Gb -> 100Gb -> 400Gb -> 800Gb (proposed)
- Video: 2K -> 4K -> 8K
- FPGA DSP slice from 2,000 -> 12,000
- But DDR3 -> DDR4 only 2X bandwidth improvement

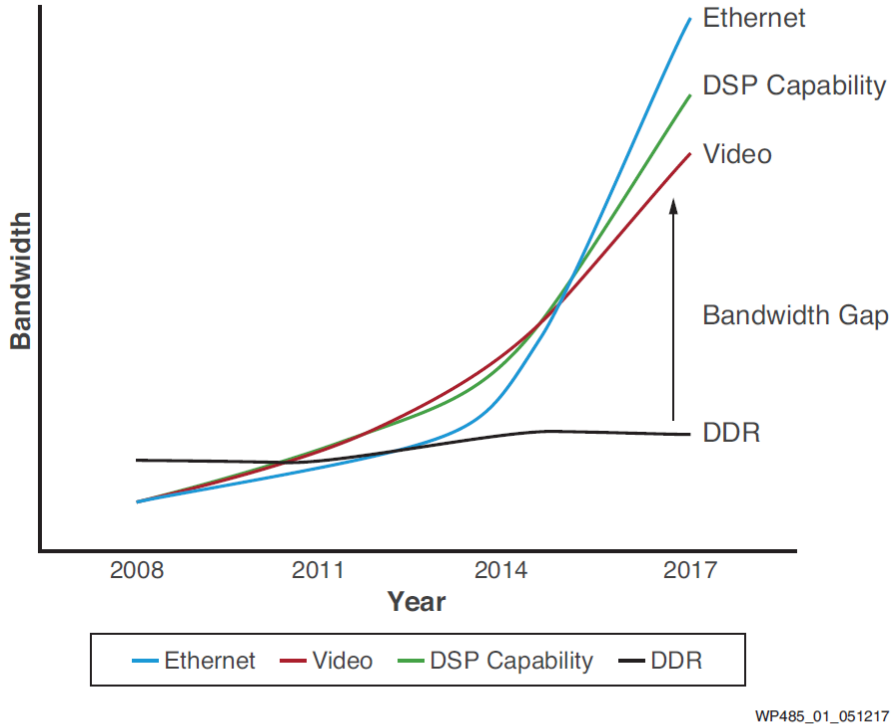


Figure 1: **Relative Memory Bandwidth Requirements**

HBM – High Bandwidth Memory

- Stacked multiple smaller DDR memory using TSV
- Place FPGA & DRAM in the same package – chip-on-wafer-on-substrate (CoWoS)
  1. Dense microbump
  2. IO is 20X smaller area
  3. 1024 DQ pins uses half of I/O silicon area of DDR
  4. much shorter distance, less capacitor, 4X less power per bit

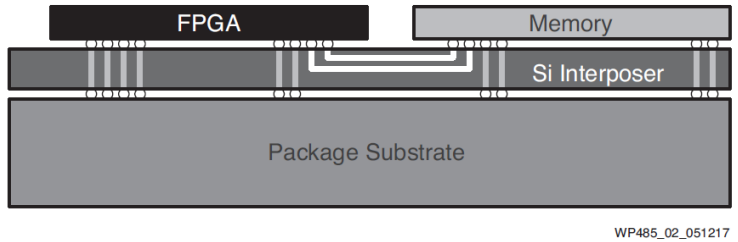
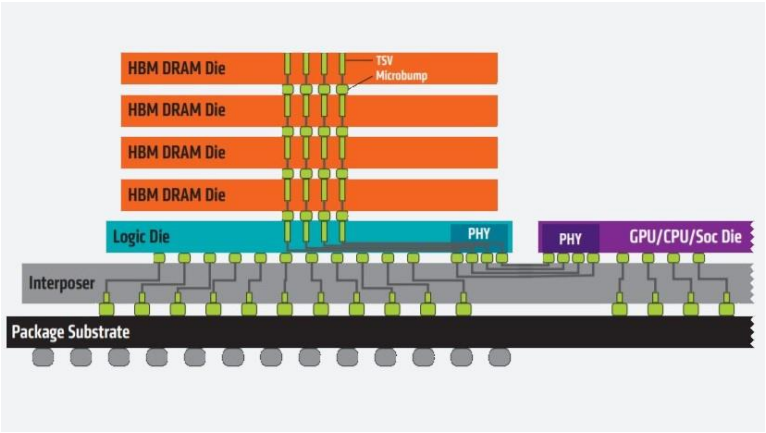


Figure 2: **TSMC CoWoS Assembly Allows for Thousands of Very Small Wires Connecting Adjacent Die**

Table 1: **Comparison of Key Features for Different Memory Solutions**

	DDR4 DIMM	RLDRAM-3	HMC	HBM
Description	Standard commodity memory used in servers and PCs	Low latency DRAM for packet buffering applications	Hybrid memory cube serial DRAM	High bandwidth memory DRAM integrated into the FPGA package
Bandwidth	21.3GB/s	12.8GB/s	160GB/s	460GB/s
Typical Depth	16GB	2GB	4GB	16GB
Price / GB	$	$$	$$$	$$
PCB Req	High	High	Med	None
pJ / Bit	~27	~40	~30	~7
Latency	Medium	Low	High	Med



https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf

AXI Interface to access HBM

- 8GB HBM memory
- 32 HBM Banks/Pseudo Channels (PC) – each 256MB
- 32 AXI channels (PC) communicate with FPGA using segmented crossbar switch
- 14.375 GB/s max theoretical bandwidth per PC
- 460 GB/S (32 * 14.375 GB/s per PC)
- Note: 14.375GB is less than 19.25GB/s for a DDR channel
  - Us multiple AXI masters efficiently into the HBM subsystem.
- 32 AXI channels / AXI switch network
- 256bit data width
- Flexible Addressing
- RAMA (Random Access Master Attachment) to improve throughput
  - Random access across multiple HBM banks
  - Techniques: Resizing burst, Multiple outstanding requests, response re-ordering

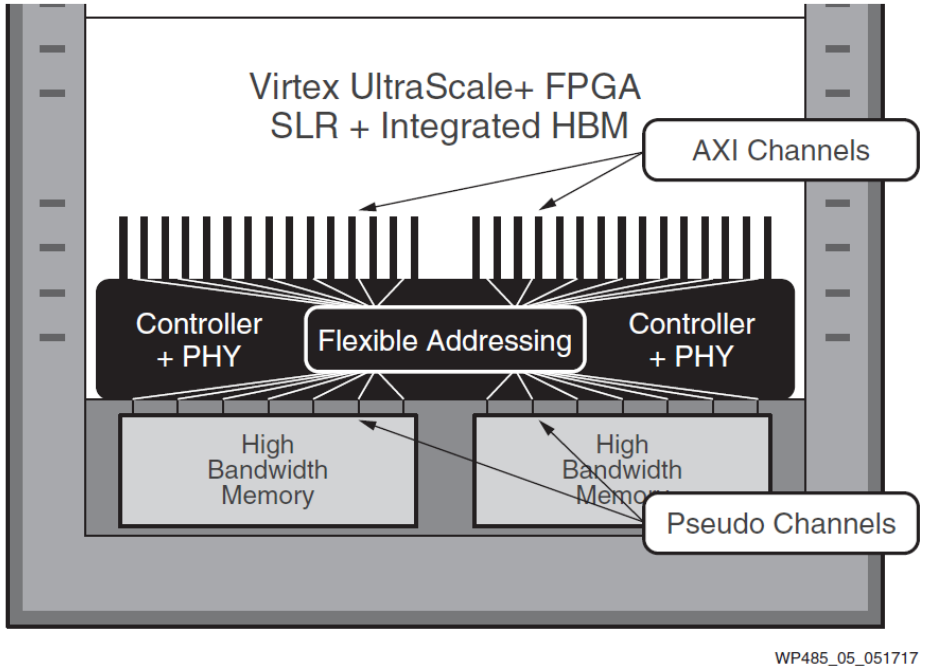
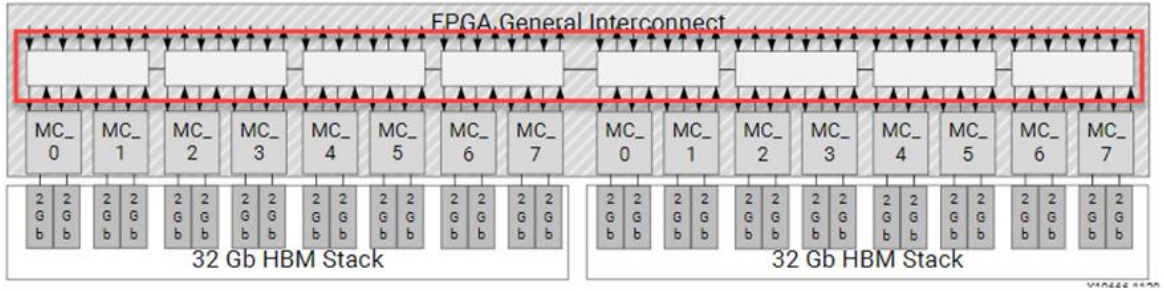


Figure 5: **AXI Interfaces (to User Logic) and HBM Pseudo Channels (to HBM Stacks)**

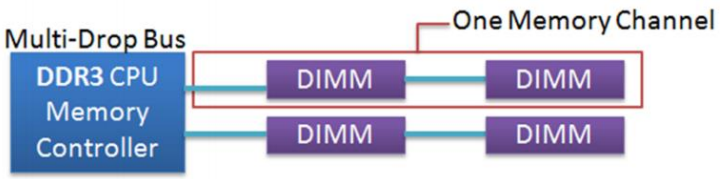


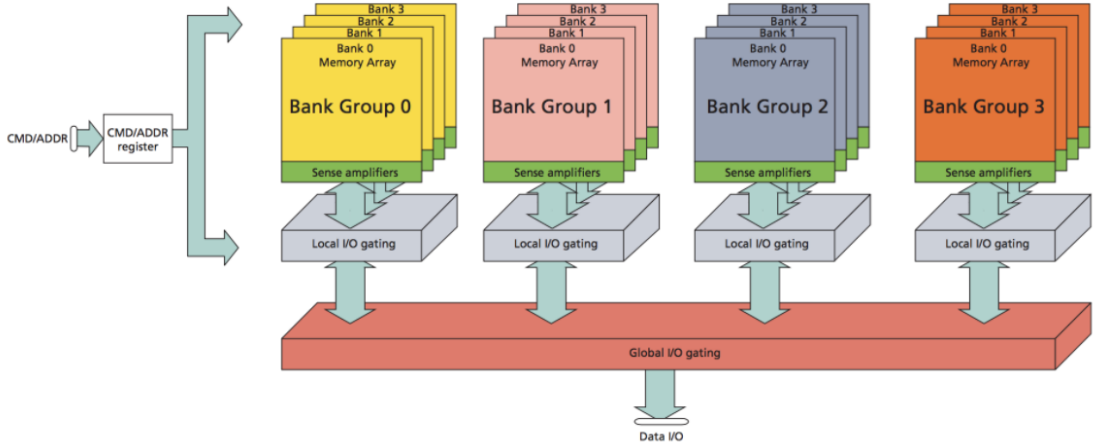
https://github.com/Xilinx/Vitis-Tutorials/tree/2022.1/Hardware_Acceleration/Feature_Tutorials/07-using-hbm
https://docs.xilinx.com/r/en-US/pg310-rama/IP-Facts

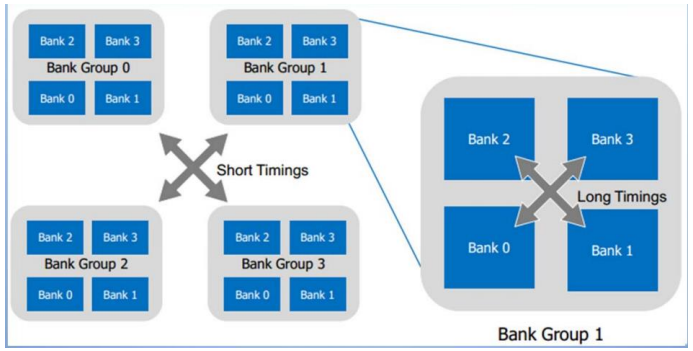
DDR Memory

DRAM Configuration

- Rank (CS), DIMM, Channel
- Bank, Bank Group
  - Number of banks: 8 – 16
  - Bank Group: 2-4
  - Separate activation, read, write or refresh underway in each of bank group
  - Increase memory bandwidth and efficiency for small granularity of access
- Page registers – keep track which page is activated, ready for read/write access







DDR3/DDR4 Memory Explained

From the perceptive of performance

- Device Behavior
  - Multi-bank / page registers - bank interleave, open-page
  - Pipeline Burst access
  - Read/write turn-around
- Controller features
  - Interface to be compatible with memory device specification
  - Features for optimizing bus utilization

DDR4 specification: http://www.softnology.biz/pdf/JESD79-4_DDR4_SDRAM.pdf

DRAM Configuration

Device 4Gb x 8

- Number of Row Address bits: A0-A15 = 15 bits => Total number of row = 2^15 = 32K
- Number of Column Address bits: A0-A9 = 10 bits => Number of columns per row = 1K
- Width of each column = 8 bits
- Number of Bank Groups = 4
- Number of Banks = 4

- Total DRAM Capacity =
  - Num.Rows x Num.Columns x Width.of.Column x Num.BankGroups x Num.Banks
  - 32K x 1K x 8 x 4 x 4 = 4Gb
- Page-size (assume 64-bit DRAM channel)
  - Num.Columns x 64-bit = 1K x 8B = 8KB

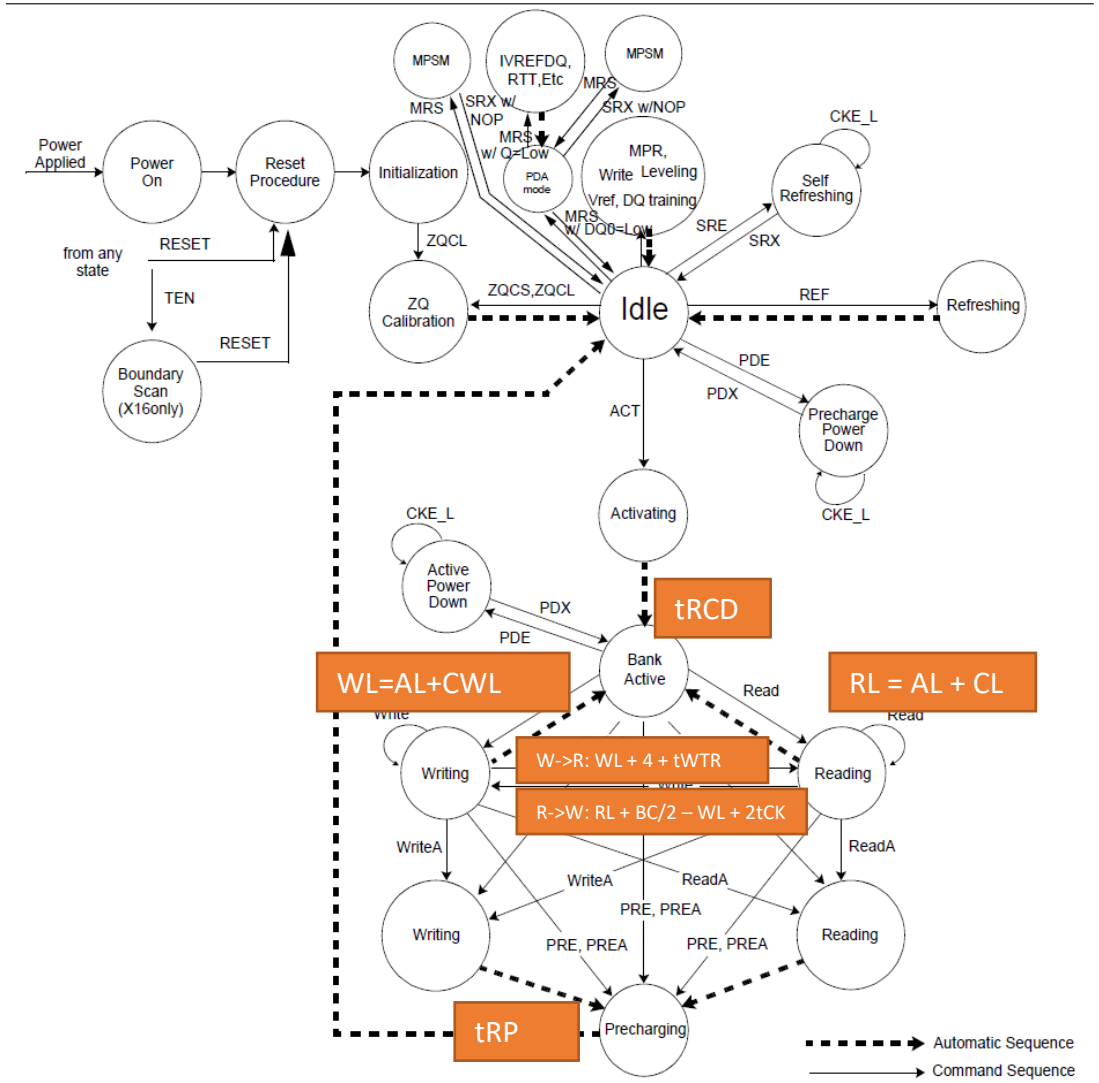
https://www.systemverilog.io/ddr4-basics#:~:text=DRAM%20Page%20Size&text=Page%20size%20is%20essentially%20the,4K%20bits%20(or%20512B) .

State Machine

Transaction flow

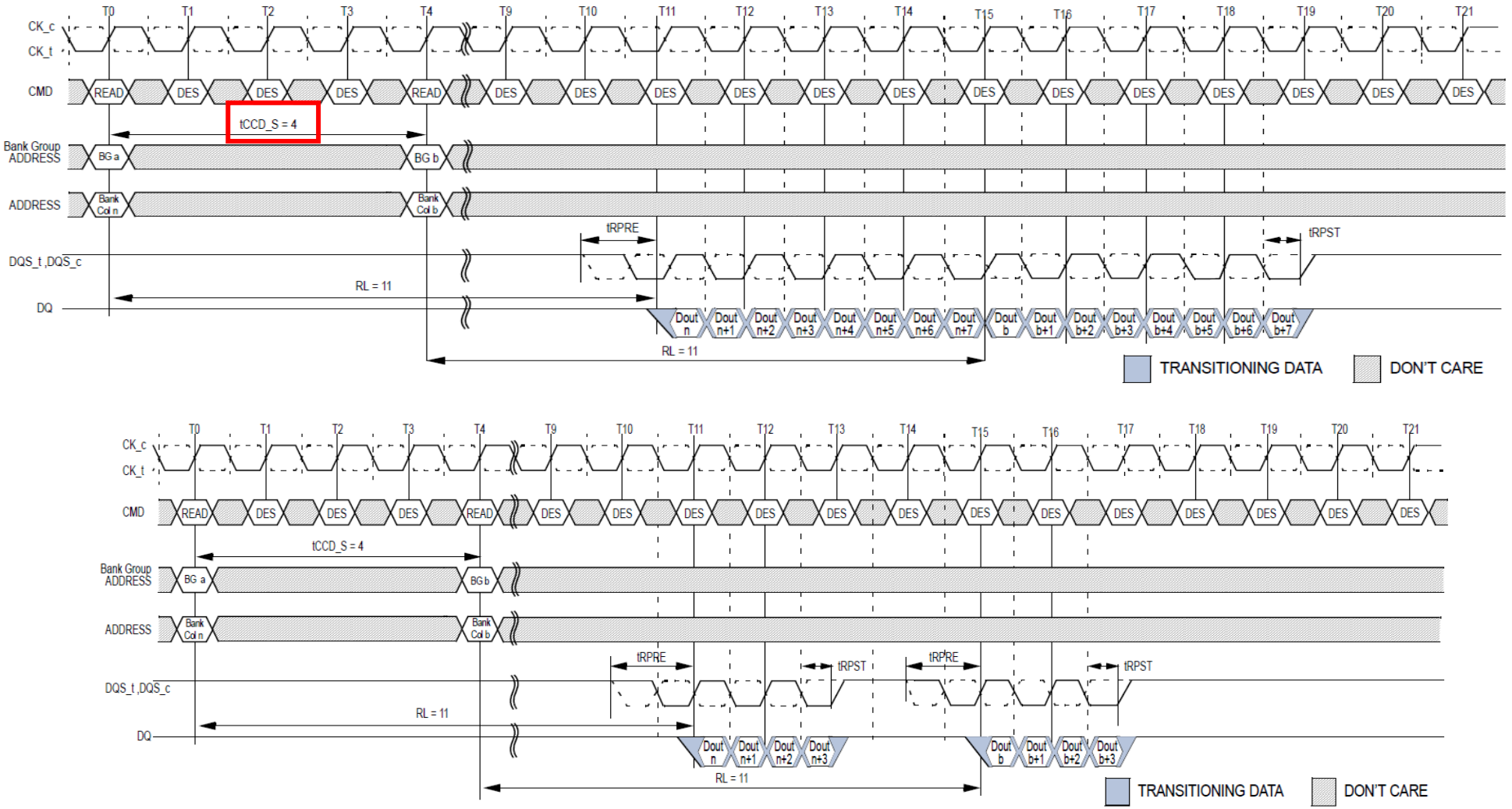
- Start Page :
  - accessed bank is idle
  - Idle -> Activate (tRCD) -> Read/Write
- On Page:
  - accessed bank is already activated
  - Bank Active -> Read/write
- Off Page:
  - accessed bank is different from current activated bank
  - Bank Active -> Precharge (tRP) -> Activate -> Read/Write

3.1 Simplified State Diagram

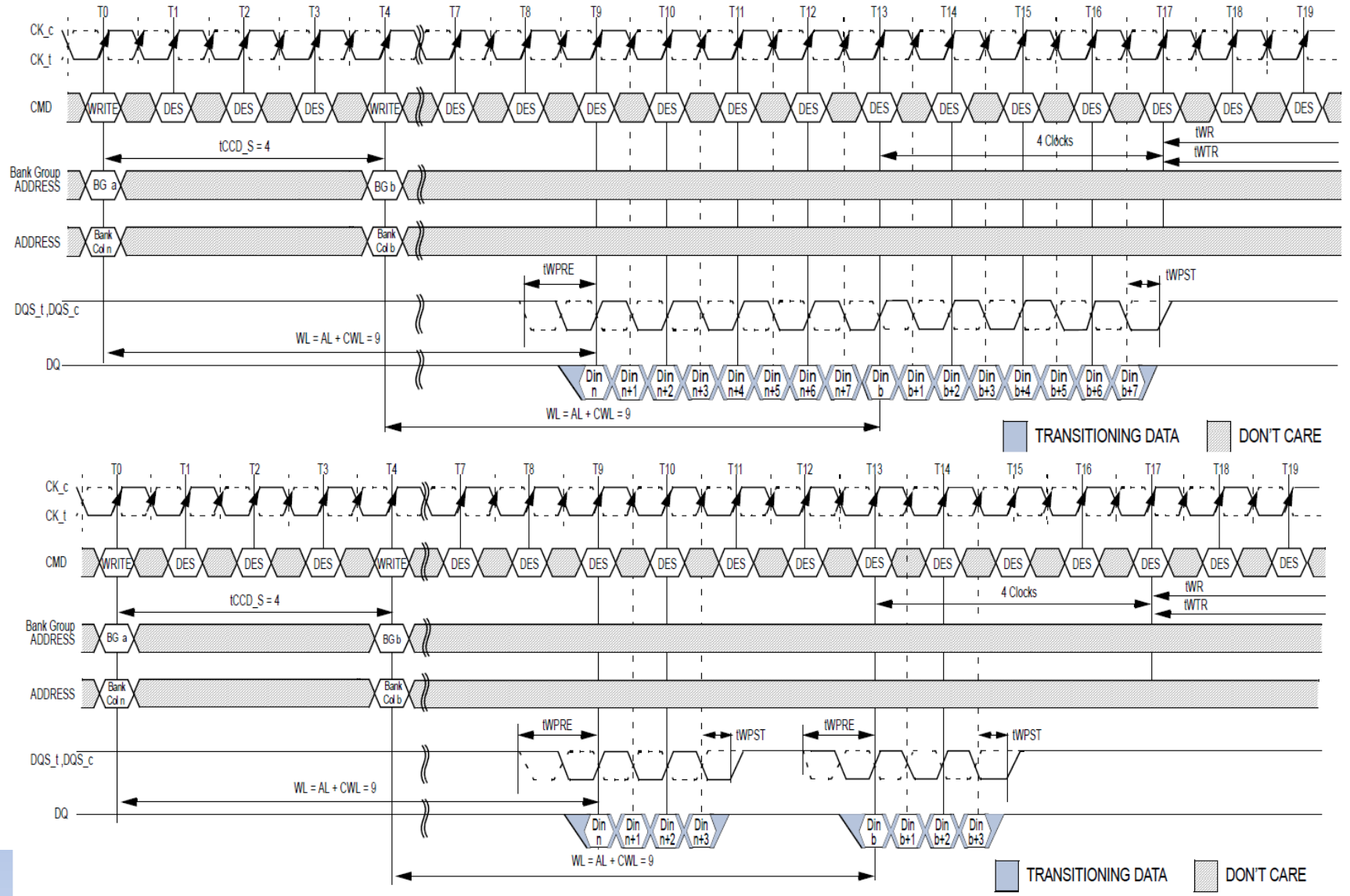


Abbreviation	Function	Abbreviation	Function	Abbreviation	Function
ACT	Activate	Read	RD,RDS4, RDS8	PDE	Enter Power-down
PRE	Precharge	Read A	RDA, RDAS4, RDAS8	PDX	Exit Power-down
PREA	PRECHARGE All	Write	WR, WRS4, WRS8 with/without CRC	SRE	Self-Refresh entry
MRS	Mode Register Set	Write A	WRA,WRAS4, WRAS8 with/without CRC	SRX	Self-Refresh exit
REF	Refresh, Fine granularity Refresh	RESET_n	Start RESET procedure	MPR	Multi Purpose Register
TEN	Boundary Scan Mode Enable				

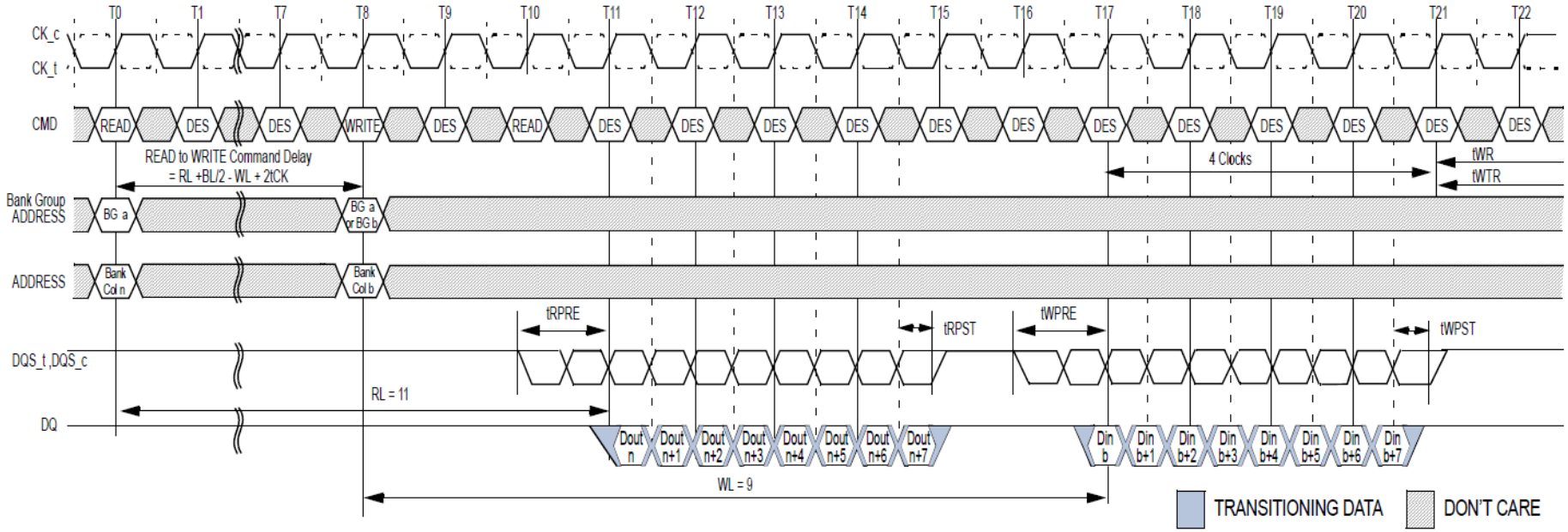
Consecutive Read BL8, BL4



Consecutive WRITE BL8, BC4



READ(BL8) to WRITE(BL8) Turn-around

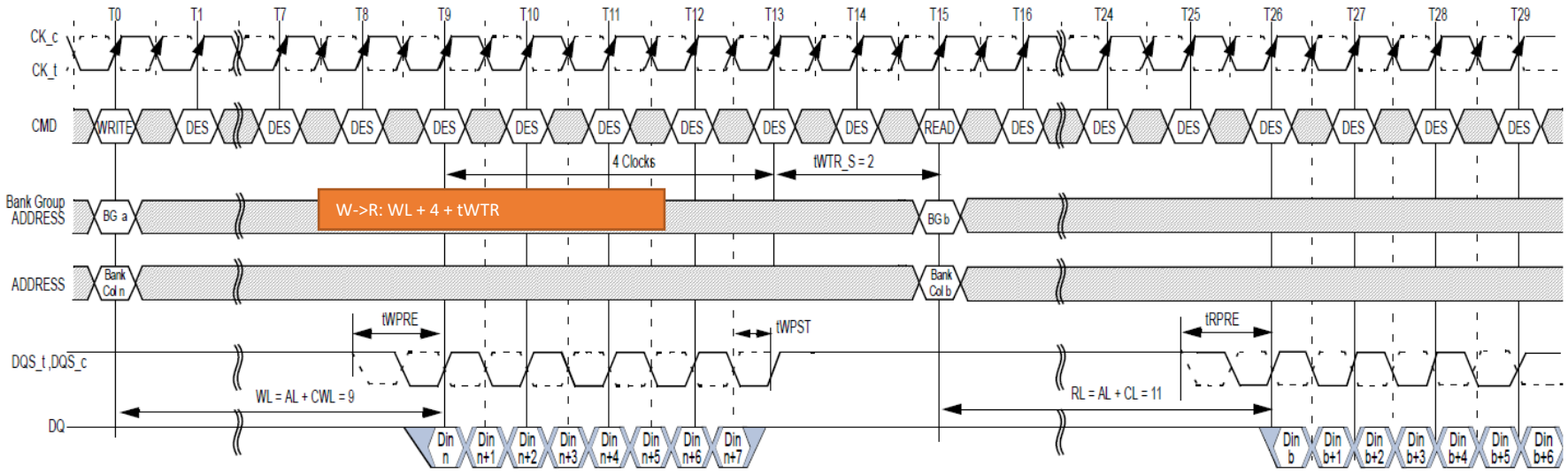


NOTE :

1. BL = 8, RL = 11 (CL = 11, AL = 0), Read Preamble = 1tCK, WL =9 (CWL = 9, AL = 0), Write Preamble = 1tCK
2. DOUT n = data-out from column n, DIN b = data-in to column b.
3. DES commands are shown for ease of illustration; other commands may be valid at these times.
4. BL8 setting activated by either MR0[A1:A0 = 0:0] or MR0[A1:A0 = 0:1] and A12 = 1 during READ command at T0 and WRITE command at T8.
5. CA Parity = Disable, CS to CA Latency = Disable, Read DBI = Disable, Write DBI = Disable, Write CRC = Disable.

Figure 79 — READ (BL8) to WRITE (BL8) with 1tCK Preamble in Same or Different Bank Group

Write to Read Turn-around



4/(13+4) < 25% utilization

NOTE:

1. BC = 4, AL = 0, CWL = 9, CL = 11, Preamble = 1tCK
2. DIN n = data-in to column n(or column b). DOUT b = data-out from column b.
3. DES commands are shown for ease of illustration; other commands may be valid at these times.
4. BL8 setting activated by either MR0[A1:A0 = 0:0] or MR0[A1:A0 = 0:1] and A12 = 1 during WRITE command at T0 and READ command at T15.
5. CA Parity = Disable, CS to CA Latency = Disable, Write DBI = Disable.
6. The write timing parameter (tWTR_S) are referenced from the first rising clock edge after the last write data shown at T13.

Figure 114 — WRITE (BL8) to READ (BL8) with 1tCK Preamble in Different Bank Group

DDR3/DDR4 Memory Controller Functions

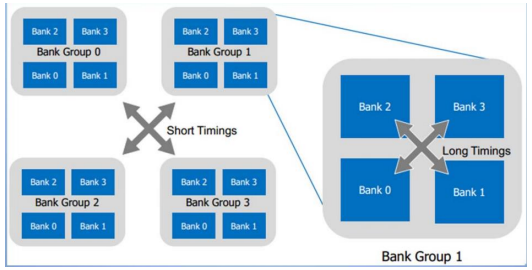
- Meet DRAM protocol and Timing requirement
- Command queues/state machine for each concurrent bank operation
  - Interleave DRAM command for multiple transactions
- Re-order transactions （read over write）
- Open page/closed page policy (AutoPrecharge)
  - Look ahead and predict offpage
  - Heuristic: idle for certain time, automatically precharge
- Refresh scheduling - opportunistic refresh
- Optimize bus utilization and subject to latency requirement

Application Optimization

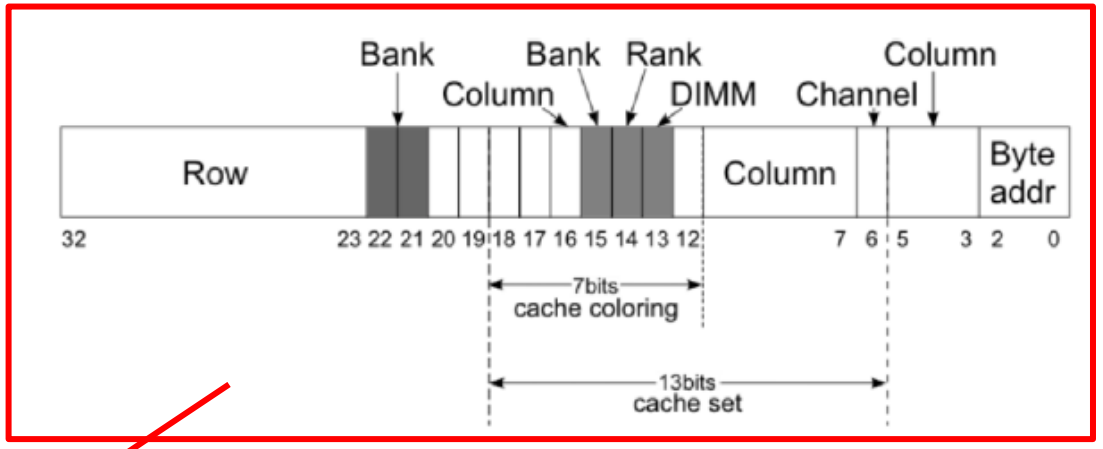
- Considering DRAM access latency – deeper transaction pipeline
- Minimize read-write turn-around (minimize read/write dependency)
- Carefully design the data structure/addressing scheme
  - Take advantage on-page & bank-interleave (bank-group)
  - Know DRAM controller address map (page size)
- Local buffering to resolve conflict
  - Minimize random, short access
  - Streamline DRAM access – longer read/write burst
  - Resolve read-write turn-around

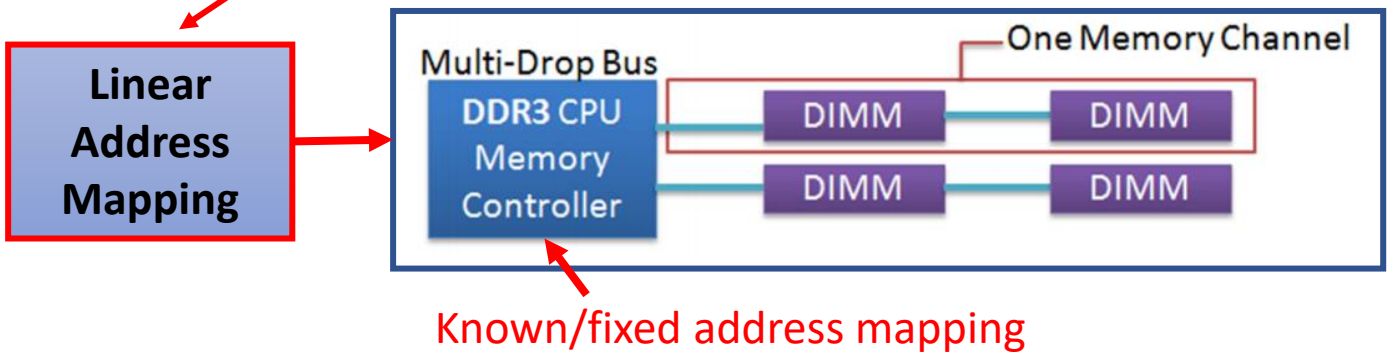
Adaptive Linear Address mapping:
http://epub.vgu.edu.vn/bitstream/dlibvgu/90/1/Adaptive%20linear%20address%20map%20for%20bank%20interleaving%20in%20DRAMs.pdf

Adaptive Linear Address Mapping



- Eliminate bank interfere by linear address mapping
  - Multiple Kernels
  - Non-linear access pattern
- Example
  - CPU cacheline size matches DRAM burst 8 = 64B
  - Channel A[6] for interleaving
  - What if your dataset size 128B
    - Channel interleaves at 128B





https://www.researchgate.net/publication/319451254_A_Software_Memory_Partition_Approach_for_Eliminating_Bank-level_Interference_in_Multicore_Systems

Some Ground Rules for Memory

- Fast means Small and Expensive
- Locality is Good (Temporal, Spatial)
- Data Movement not Free
- Memory Performance Matters (Arithmetic Intensity) - # operations/ # bytes

Memory Allocation Among BRAM, UltraRAM, DDR/HBM

Memory Comparison (Xilinx XCVU9P – AWS F1 Instance)

	BRAM	UltraRAM	DDR
Access Time	1T	3T	Read:52T, Write: 32T
Size	9.5MB	33.8 MB	64GB

- BRAM as L1 Cache, UltraRAM as L2 Cache
  - Graph Processing
- Use on-chip memory (BRAM/UltraRAM) to
  - Reduce DDR row conflict
  - Adapt burst length
- Different memory allocation scheme for different algorithm, by analyzing *access frequency, lifetime, data size* -> Area for innovation

Optimizing Use of Different Types of Memory for FPGAs in High Performance Computing https://par.nsf.gov/servlets/purl/10278004
"Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching"

Memory Restructure

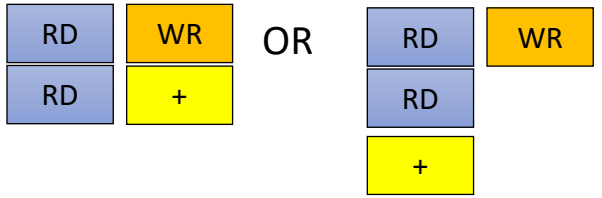
ARRAY_PARTITION

ARRAY_RESHAPE

Array: Performance Bottleneck

- Array are targeted to a default RAM, optionally a FIFO

```
Void  foo_top ( … ) {
  …
  for( i = 2; I < N; i++)
    mem[i] = mem[i-1] + mem[i-2];
    …
 }
```



Even with a dual-port RAM, we cannot schedule all read and write in one cycle

- HLS allows arrays to be partitioned and reshaped

Array and RAM selection

- If no RAM resource is selected, HLS determine the RAM to use
  - Use a Dual-port if it improves throughput
  - Else it will use a single port
- BRAM and LUTRAM selection
  - If the user specifies the RAM target (e.g. RAM_1P_BRAM, or RAM_1P_LUTRAM), HLS will obey the target
    - If LUTRAM is selected, HLS reports registers not BRAM
  - If user does not specify, RTL synthesis will determine BRAM or LUTRAM

BIND_STORAGE

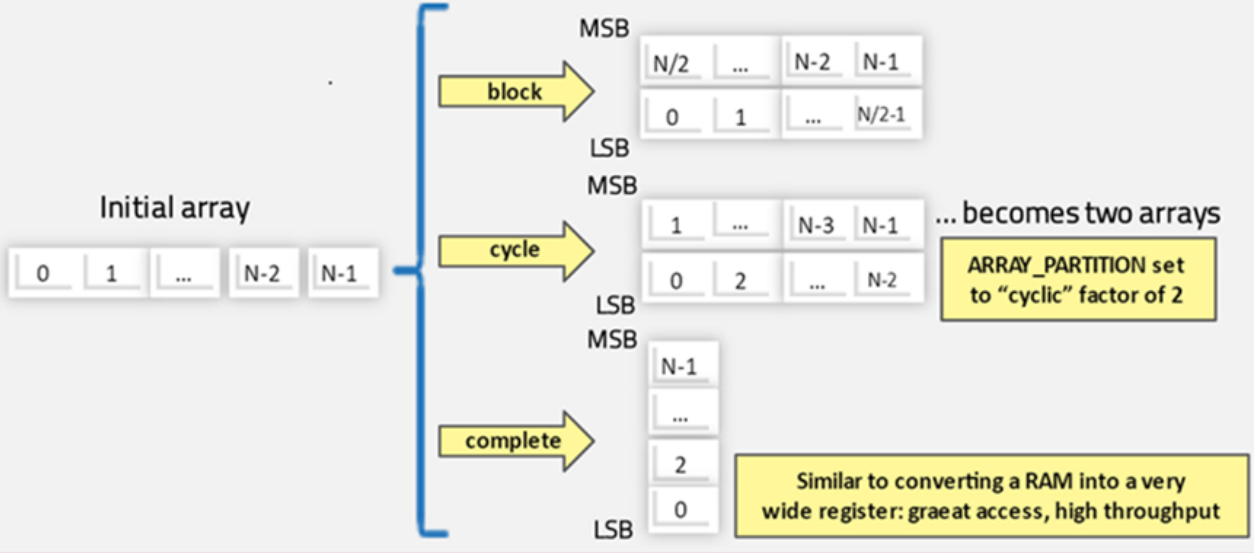
- Specify a variable with a specific memory (type)

```
#pragma HLS bind_storage variable=<variable> type=<type>  [ impl=<value> latency=<int> ]
```

- Type=<type>: ***fifo, ram_1p, ram_1wnr, ram_2p, ram_s2p, ram_t2p, rom_1p, rom_2p, rom_np.***
- impl=<value>: ***bram, bram_ecc, lutram, uram, uram_ecc, and srl***
- Latency=<int>: default -1, let Vitis HLS choose the latency

ARRAY-PARTITION

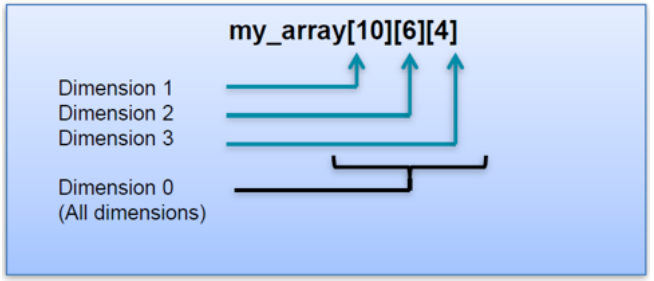
- Break an array into smaller elements
  - If the factor is not an integer multiple => last array has fewer elements
  - inherit the same resource target
- Effectively increase the read and write ports
  - Improve the throughput of the design
- Requires more memory instances or registers



ARRAY-PARTITION

#pragma HLS array_partition variable=<name> <type> factor=<int> dim=<int>

- variable=<name>: Specifies the array variable to be partitioned.
- <type>:
  - Cyclic: interleaving elements into smaller arrays
  - Block: consecutive elements into smaller array
  - Complete: partition into individual registers. Default type
- factor=<int>: Specifies the number of smaller arrays that are to be created.
- dim=<int>: Specifies which dimension of an array to partition.
  - If not specified, dimension zero is assume
  - Dimension 0 means all dimensions. all dimensions of a multi-dimensional array are partitioned with the specified type and factor options
  - Any non-zero value partitions only the specified dimension.



#pragma HLS array_partition variable=my_array complete dim=3

my_array[10][6][4] → partition dimension 3 →
my_array_0[10][6]
my_array_1[10][6]
my_array_2[10][6]
my_array_3[10][6]

#pragma HLS array_partition variable=my_array complete dim=1

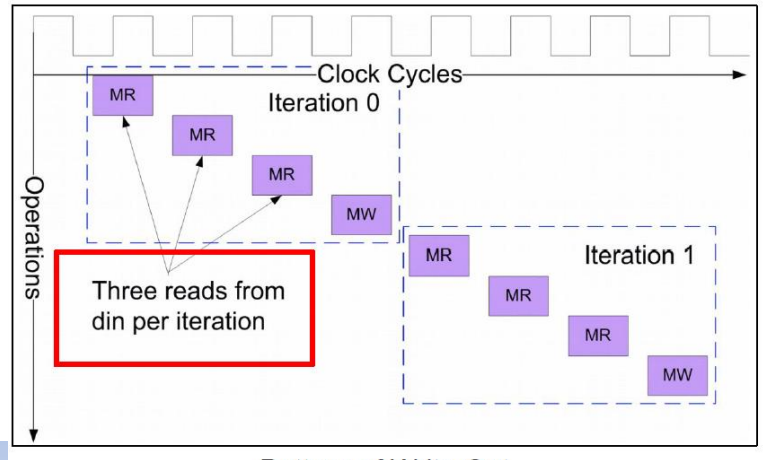
my_array[10][6][4] → partition dimension 1 →
my_array_0[6][4]
my_array_1[6][4]
my_array_2[6][4]
my_array_3[6][4]
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]

#pragma HLS array_partition variable=my_array complete

my_array[10][6][4] → partition dimension 0 → 10x6x4 = 240 registers

Interleaving using ARRAY_PARTITION

```
// Orignal Code
void interleave(  ap_int<8> x_in[NUM_WORDS],
                  ap_int<8> y[NUM_WORDS / 3],
                  bool load) {
  static ap_int<8> x[NUM_WORDS];
  int idx = 0;

 if (load)
    LOAD:  for (int i = 0; i < NUM_WORDS; i += 1)
                  x[i] = x_in[i];
 else
    WRITE: for (int i = 0; i < NUM_WORDS; i += 3)
                  // requires three separate reads from "x"
                  y[idx++] = x[i] + x[i + 1] + x[i + 2]; }
```



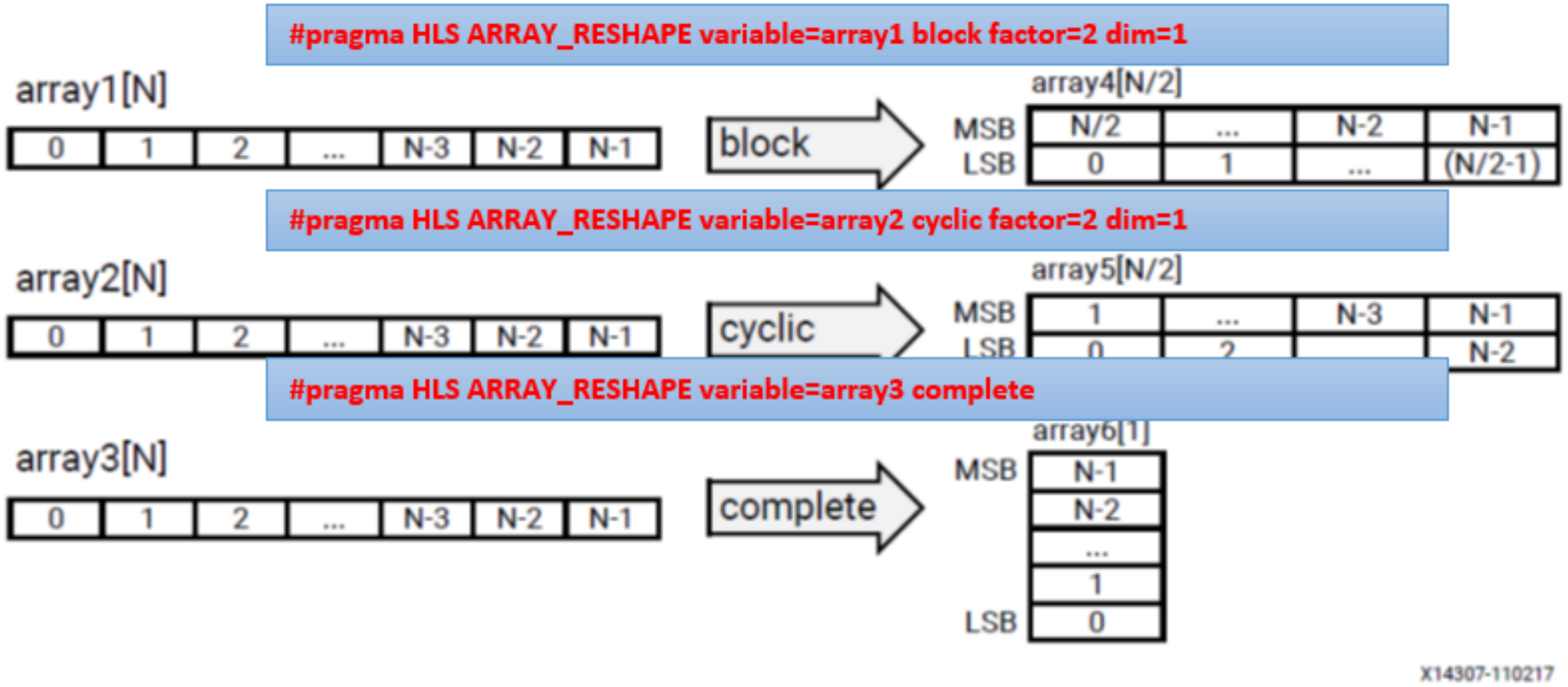
Pattern of Write Out

```
void interleave(  ap_int<8> x_in[NUM_WORDS],
                  ap_int<8> y[NUM_WORDS / 3],
                  bool load) {
#pragma HLS RESOURCE variable=x_in core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=y core=RAM_1P_BRAM

  static ap_int<8> x[NUM_WORDS];
#pragma HLS RESOURCE variable=x core=RAM_1P_BRAM
#pragma HLS ARRAY_PARTITION variable=x cyclic factor=3 dim=1

  int idx = 0;

  if (load)
     LOAD:  for (int i = 0; i < NUM_WORDS; i += 1)
                  #pragma HLS PIPELINE II=1
                  x[i] = x_in[i];
  else
     WRITE: for (int i = 0; i < NUM_WORDS; i += 3)
                  #pragma HLS PIPELINE II=1
                  y[idx++] = x[i] + x[i + 1] + x[i + 2];
}
```

ARRAY_RESHAPE

Perform

1. Partition: Partition the array into a smaller array based on <type>,<factor>,<dim>
2. Concatenate elements of arrays by **increasing bit widths.**

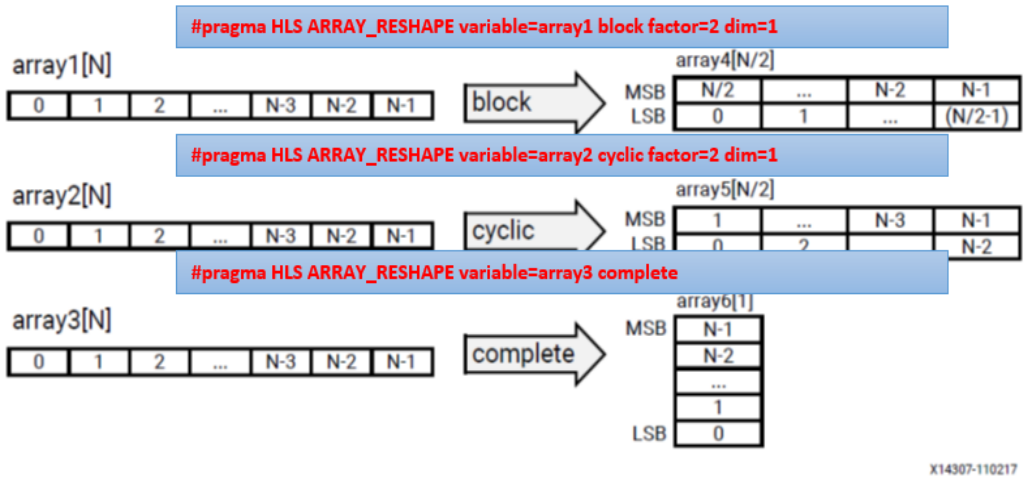


ARRAY_RESHAPE

Syntax

#pragma HLS array_reshape variable=<name> <type> factor=<int> dim=<int>

- <name>: Required argument that specifies the array variable to be reshaped
- <type>:
  - Cyclic: interleaving elements into smaller arrays
  - Block: consecutive elements into smaller array, splits the array into <N> equal blocks
  - Complete: partition into individual registers. Default type
- factor=<int>: Specifies the number of smaller arrays that are to be created.
- dim=<int>: Specifies which dimension of a multidimentional array to partition.
  - If a value is 0, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
  - Any non-zero value partitions only the specified dimension

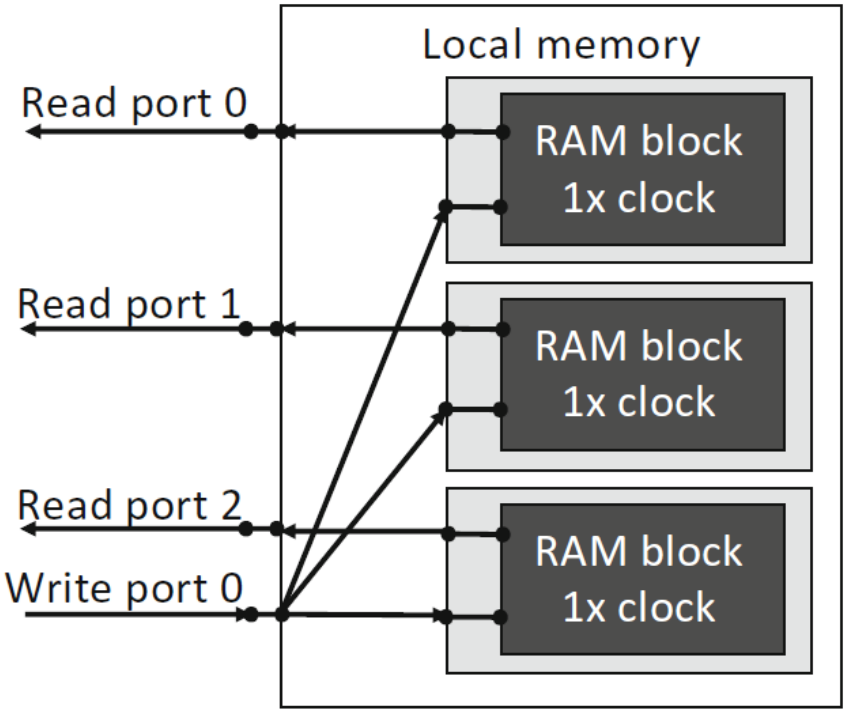


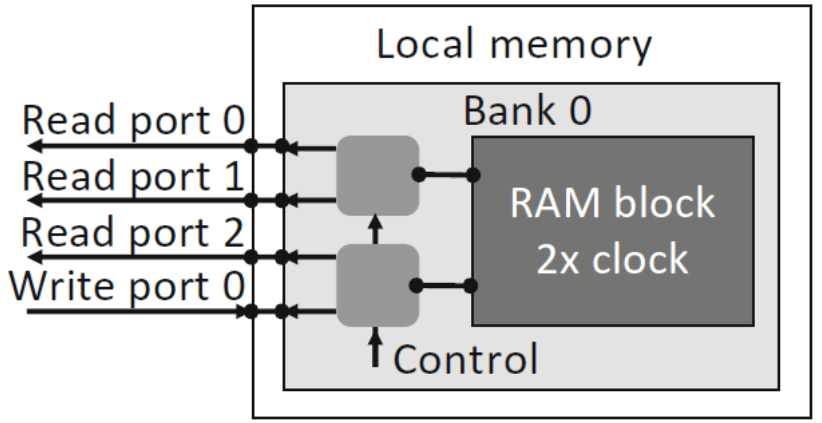
Reshape v.s. Partition

- Both are useful for increasing the memory or data bandwidth
- Reshape
  - Simply increase the width of the data word
  - Does not increase the number of memory ports
- Partition
  - Increase the memory ports, thus more I/O to deal with
  - **Use it only if you have to use independent addressing**

Increase Parallel Data Access

- Increase number of banks (partition, reshape)
- Memory replication
- Use Multi-Pumping





Multi-Pumping for Resource Reduction in FPGA High-Level Synthesis https://janders.eecg.utoronto.ca/pdfs/andrewdate13.pdf

Using Two-port BRAM

```
unsigned int v1_buffer[BUFFER_SIZE];   // Local memory to store vector1
unsigned int v2_buffer[BUFFER_SIZE];   // Local memory to store vector2
unsigned int vout_buffer[BUFFER_SIZE]; // Local Memory to store result
vadd:
      for (int j = 0; j < chunk_size; j++) {
        #pragma HLS UNROLL FACTOR=2
         vout_buffer[j] = v1_buffer[j] + v2_buffer[j];
      }
```

Using Multiple DDR Banks

- U50 device uses HBM memory. It provides 32 memory channels

```
void accel ( int *in1,  int *in2, const int Iter, int *output)
{
   #pragma HLS INTERFACE m_axi depth=10   port=in1    offset=slave   bundle=gmem   num_write_outstanding=300
   #pragma HLS INTERFACE m_axi depth=10   port=in2    offset=slave   bundle=gmem    num_write_outstanding =300
   #pragma HLS INTERFACE m_axi depth=10   port=output offset=slave bundle=gmem2  num_write_outstanding=300
   #pragma HLS INTERFACE s_axilite       port=in1              bundle=control
   #pragma HLS INTERFACE s_axilite       port=in2              bundle=control
   #pragma HLS INTERFACE s_axilite       port=Iter             bundle=control
   #pragma HLS INTERFACE s_axilite       port=output             bundle=control
   #pragma HLS INTERFACE s_axilite       port=return           bundle=control
```

Memory Port Width Widening

Auto port width widening – pipeline loop is fixed

```
dot_product_2(const uint32_t* a, const uint32_t* b, uint64_t* res, const int size, const int reps) {

loop_reps: for (int i = 0; i < reps; i++) {
   dot_product_outer: for (int j = 0; j < size; j += DATA_WIDTH) {
      dot_product_inner: for (int k = 0; k < DATA_WIDTH; k++) {
         res[j + k] = a[j + k] * b[j + k];
} }  }     }
```

Interface pragma based

```
void dot_product_5(const uint32_t* a, const uint32_t* b, uint64_t* res, const int size, const int reps) {
#pragma HLS INTERFACE m_axi port = a bundle = gmem0 max_widen_bitwidth = 512
#pragma HLS INTERFACE m_axi port = b bundle = gmem1 max_widen_bitwidth = 256
#pragma HLS INTERFACE m_axi port = res bundle = gmem0 max_widen_bitwidth = 512
```

Using data type : ap_uint<>, e.g. ap_unit<512>

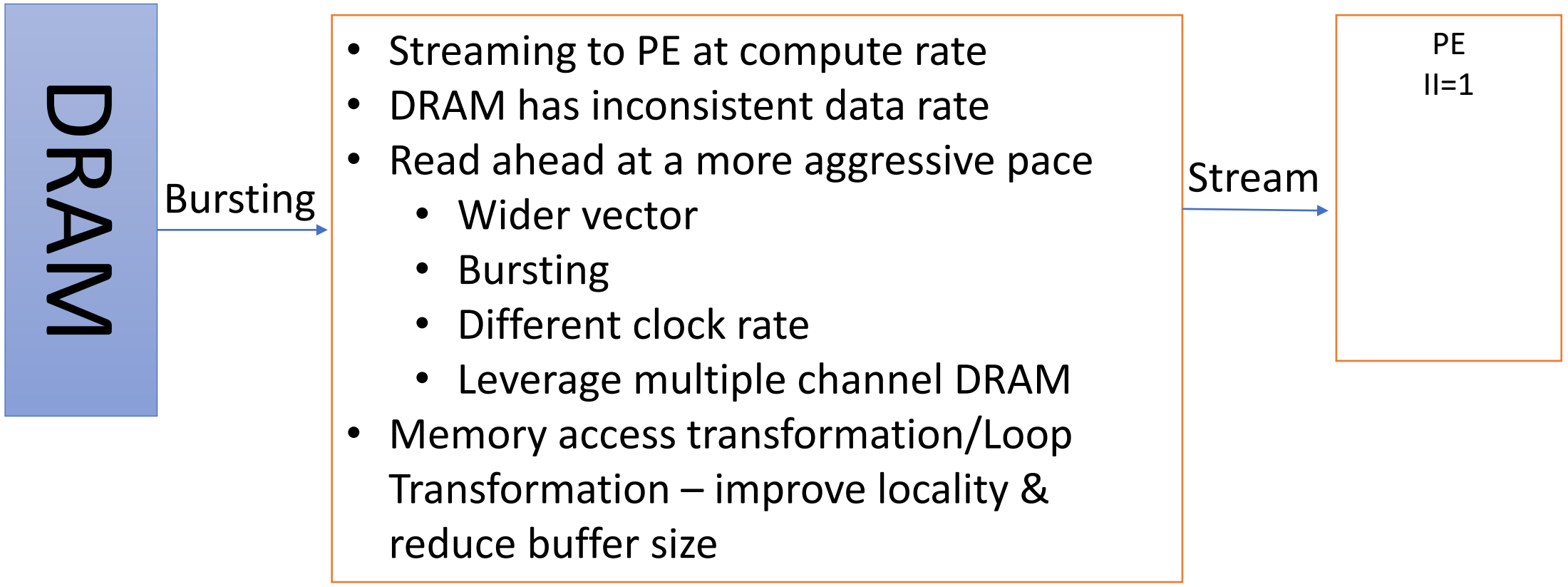
```
#define DATAWIDTH 512
typedef ap_uint<DATAWIDTH> uint512_dt;
uint512_dt v1_local[BUFFER_SIZE];
uint512_dt result_local[BUFFER_SIZE];
```

Best Practices for Designing with M_AXI Interfaces

https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Best-Practices-for-Designing-with-M_AXI-Interfaces

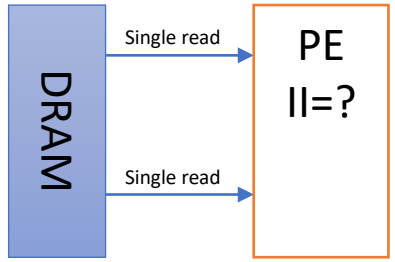
Memory Access Transformation
Optimize the efficiency of off-chip memory access

Memory Buffering



Memory Access Extraction

```
void PE(const int A[N], int B[N/2]) {
for (int i = 0; i < N/2; i++) {
  #pragma HLS PIPELINE
  // Issues N/2 memory requests of size 1
  B[i] = A[i] + A[N/2 + i];
}
```



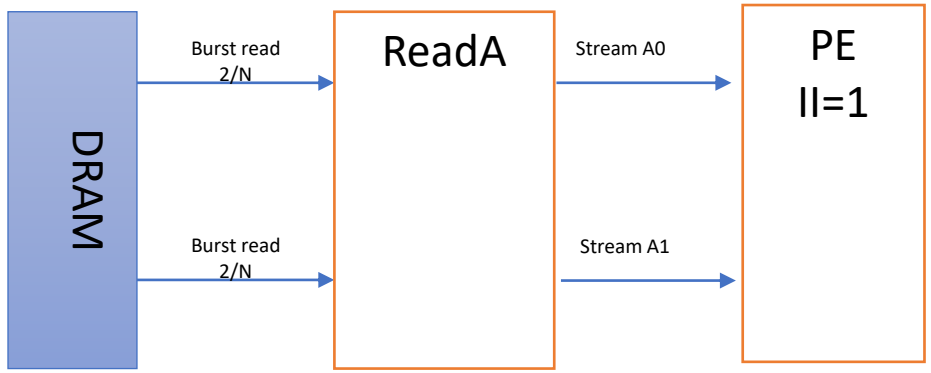
Issues:

- Access the same interface multiple times within the same pipeline
- Single memory access
- Interleave among two different pages, may cause page break

Memory Access Extraction

A Solution:

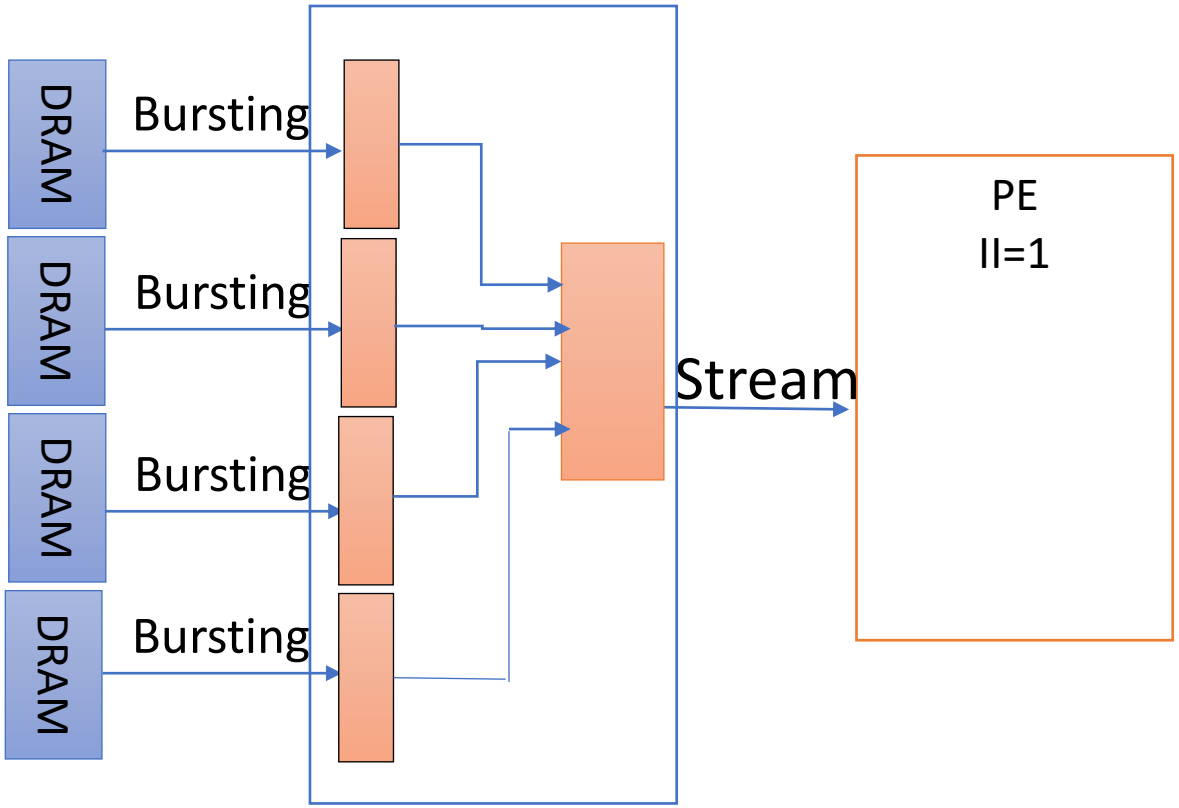
- Separate compute and memory access. Optimize separately
- Allow for burst access
- Prefetch A0 (N/2 elements)
- PE access A0, A1 stream in parallel
- The fast on-chip buffer performs data layout transformation

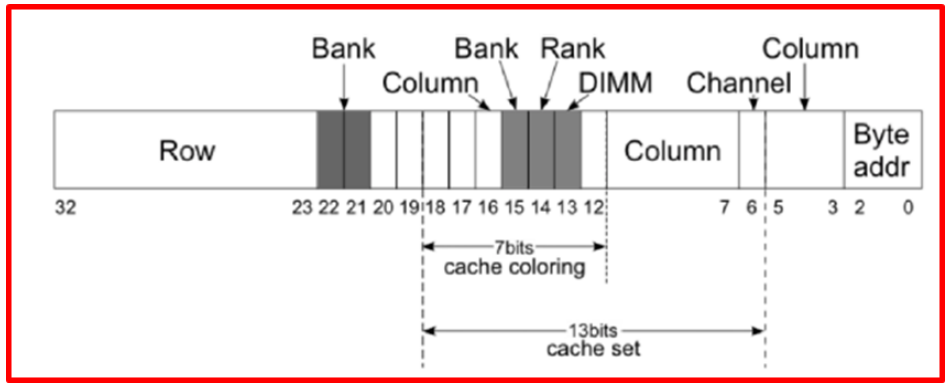



```
void PE(hls::stream<int>& A0,
        hls::stream<int>& A1,
        int B[N/2] ) {
  for (int i = 0; i < N/2; i++)
     B[i]  = A0.read() + A1.read();
}
```

```
Void ReadA(const int A[N],
           hls::stream<int>& A0,
           hls::stream<int>& A1 ) {
  int buffer[N/2];
  for (int i = 0; i < N/2; i++) {
     #pragma HLS PIPELINE
     buffer[i] = A[i];  }   // Issues 1 memory requet of size N/2;
   for (int i = 0; i < N/2; i++) {
     A0.write(buffer[i]);        // Sends to PE
     A1.write( A[N/2 + i]);     // Issue 1 memory request of size N/2
   }
}
```

Memory Stripping





- Use multiple memory bank to increase bandwidth
- Stripping/Interleave across multi-channel memory
- Join or split data stream from multiple memory interfaces

Memory Cache

```
dout_t array_mem_bottleneck(din_t mem[N]) {
    dout_t sum=0;
    int i;
    SUM_LOOP:for(i=2;i<N;++i)
      sum += mem[i] + mem[i-1] + mem[i-2];
    return sum;
}
```

iteration							tmp2->	tmp1->	tmp0
i=2				mem[2]	mem[1]	mem[0]	mem[i]->	mem[i-1]->	mem[i-2]
i=3			mem[3]	mem[2]	mem[1]				
i=4		mem[4]	mem[3]	mem[2]					

Observe – each iteration only read a new item

1. tmp2 <- mem[i] save new memory item into tmp2
2. tmp0 <- tmp1 shift tmp1 into tmp0
3. tmp1 <- tmp2 shift tmp2 into tmp1

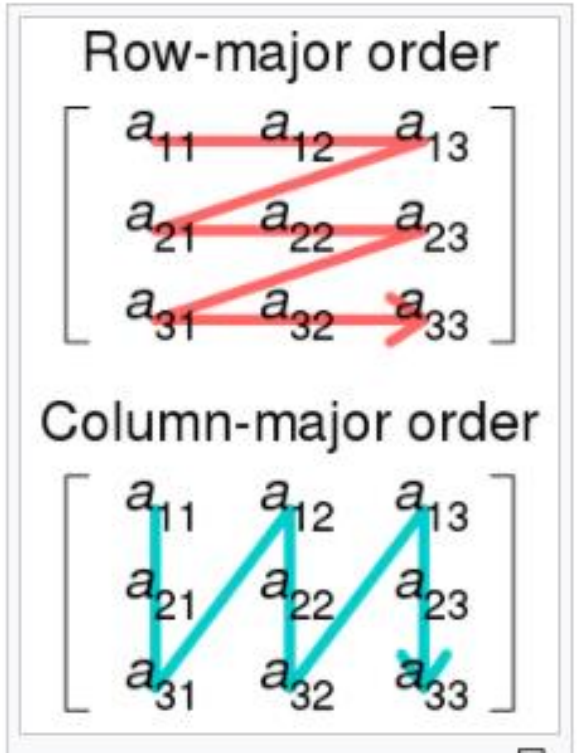
```
dout_t mem_bottleneck_resolved(din_t mem[N]) {

  din_t tmp0, tmp1, tmp2;
  dout_t sum=0;
  int i;

  tmp0 = mem[0];
  tmp1 = mem[1];
  SUM_LOOP: for (i = 2; i < N; i++) {
    tmp2 = mem[i];
    sum += tmp2 + tmp1 + tmp0;
    tmp0 = tmp1;
    tmp1 = tmp2;
  }

  return sum;
}
```

Loop Optimization with Memory Access Transformation

https://en.wikipedia.org/wiki/Loop_optimization

Access Array Elements in Storage Order

- Prefer Sequential access
  - CPU Caching exploit spatial locality (Cache Line), Temporal local (Way)
  - SIMD on vectors of data
- Row-major order are arranged consecutively along the row
- Column-major order are arranged consecutively along the column
- C/C++ : 2-D arrays are stored in row-major order.




```
for (i = 0; i < 999; i++)
      for (j = 0; j < 999; j++)
         m[i][j] = m[i][j] + (m[i][j] * m[i][j]);
```

row major order run time: 0.067300

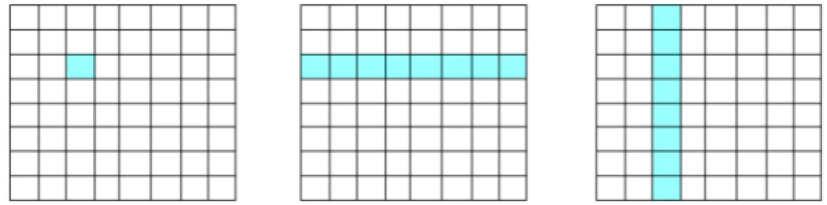
```
for (j = 0; j < 999; j++)
      for (i = 0; i < 999; i++)
         m[i][j] = m[i][j] + (m[i][j] * m[i][j]);
```

column major order run time = 0.136622

Data Reuse – Loop Interchange

- Reduction on a[][]
  - Loop-carried dependency
  - Multi-cycle operator to prevent II=1
  - Prevent parallelism even with inner loop unroll
- Reorder loop to remove Reduction
  - No Loop-carried dependency
  - Fully use SIMD
  - Cache B element
  - Linear access on C

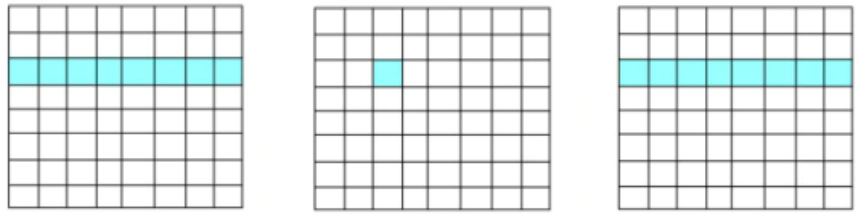
```
for(i=0;i<N;i++)
   for(j=0;j<N;j++)
     for(k=0;k<N;k++)
       a[i][j] +=b[i][k] * c[k][j];
```



A matrix B matrix C matrix

One iteration of loop j in ijk ordering (reduction)

```
for(i=0;i<N;i++)
   for(k=0;j<N;j++)
     for(j=0;k<N;k++)
       a[i][j] +=b[i][k] * c[k][j];
```



A matrix B matrix C matrix

One iteration of loop k in ikj ordering

https://github.com/bol-edu/2022-spring-nthu/tree/main/Lab_B_Vitis_Tutorials/Loop_Reorder

Loop Interchange + Unroll

- Nested loop with data Recursion
- Swaps the order of execution of two nested loops
- What if M >> N, considering Loop overhead
- Unroll to mitigate Loop overhead when M >> N

```
for (i=0; i<N; i++) {
  for (j=0; j<M; j++) {
    a[j][i] = a[j][i] + a[j][i-1];
} }
```

```
for (j=0; i<M; i++) {
  for (i=0; i<N; j++) {
    a[j][i] = a[j][i] + a[j][i-1];
} }
```

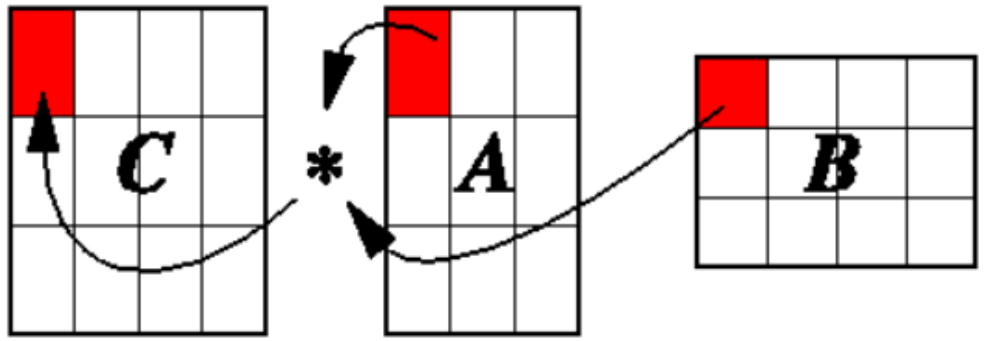
```
for (j=0; j<M; j+=4) {
  for (i=0; i<N; i++) {
    a[j+0][i] = a[j+0][i] + a[j+0][i-1];
    a[j+1][i] = a[j+1][i] + a[j+1][i-1];
    a[j+2][i] = a[j+2][i] + a[j+2][i-1];
    a[j+3][i] = a[j+3][i] + a[j+3][i-1];
  }
}
```

https://techpubs.jurassic.nl/manuals/0640/developer/OrOn2_PfTune/sgi_html/ch07.html
https://www.cita.utoronto.ca/~merz/intel_c10b/main_cls/mergedProjects/optaps_cls/common/optaps_perf_optstrats.htm

Cache Blocking

- Dividing large array into smaller blocks of memory to fit on-chip memory (b1, b2)
- Data block size, buffer size, number of times data is reused
- Use
  - Loop splitting &
  - Loop interchange,
  - And many way to do the block

```
for (ii=0; ii<m; ii+=b1) {
  for (kk=0; kk<l; kk+=b2) {
    for (j=0; j<n; j++) {
      for (i=ii; i<MIN(ii+b1-1,m); i++) {
        for (k=kk; i<MIN(kk+b2-1,l); k++) {
          c[i][j] += a[i][k]*b[k][j];
} } } } }
```



https://www.intel.com/content/www/us/en/developer/articles/technical/cache-blocking-techniques.html?wapkw=loop%20interchange%20cache%20blocking

Manipulate Data Structure to Optimize Memory Use

- Array of Structure - AOS
- Dot Product
  - dx += v.x * p;
- AoS can not apply SIMD
- #pragma AGGREGATE variable =

```
typedef struct {
  float          x, y, z;
  int            a, b, c;
} Vertex;
Vertex   Vertices[NumOfVertices];  // AOS
```

https://www.intel.com/content/www/us/en/developer/articles/technical/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.html?wapkw=data%20structure%20

Manipulate Data Structure to Optimize Memory Use

- Structure of Array - SOA
- Swizzle elements of AoS to SoA format before SIMD
  - On-the-fly or pre-processing
  - Better data is store in SoA format in the beginning
- SoA more efficient use of memory bandwidth
- But uses more independent memory-stream (address generation)

```
typedef struct {                    // SOA – Structure of Array
  float x[NumOfVertices], y[NumOfVertices], z[NumOfVertices];
  int   a[NumOfVertices], b[NumOfVertices], c[NumOfVertices];
} VerticesList;
VerticeList  Vertices;
```

https://www.intel.com/content/www/us/en/developer/articles/technical/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.html?wapkw=data%20structure%20

Manipulate Data Structure to Optimize Memory Use

- Hybrid SoA
- Data is organized to enable more efficient vertical SIMD computation
- Simpler/less address generation than AoS
- Fewer streams, which reduces page misses
- Use of fewer prefetch, due to fewer streams
- Efficient buffer packing of data elements that are used concurrently

```
NumOfGroup = NumOfVertices/SIMwidth;
typedef struct {                    // Hybrid SoA
        float x[SIMDwidth], y[SIMDwidth], z[SIMDwidth];
}  VerticesCorrdList;
typedef struct{
        int a[SIMDwidth]; b[SIMDwidth], c[SIMDwidth];
}  VerticesColorList;
VerticesCoordList   VerticesCoord[NumOfGroups];
VerticesColorList    VerticesColor[NumOfGroups];
```

xxxxxxxx yyyyyyyy zzzzzzzz

aaaaaaaa bbbbbbbb cccccccc

https://www.intel.com/content/www/us/en/developer/articles/technical/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.html?wapkw=data%20structure%20

Prefetch – Double-Buffering

- Caches –
  - exploit temporal and spatial locality by keeping recently used data in the cache and also bringing the nearby data into the cache.
  - But it can not hide cache miss latency
- Prefetch If the memory access pattern of an application is static
- Use double-buffered prefetching avoids stalls

```
// A[N][K] x B[K][M] ; A,B,C : double
for (int n = 0; n < N; ++n)
  for (int m = 0; m < M; ++m) {
     double acc = C[n][m];
     for (int k = 0; k < K; ++k) {
       #pragma PIPELINE
       acc += A[n][k] * B[k][m]; }
     C[n][m] = acc;
}  }
```

```
// A[N][K] * B[K][M]
// CB[K],C2B[K] : column prefetch buffers from B[][]
// RB[K],R2B[K] : row prefetch buffers from A[][]
// prefetch CB, RB
// Convert to perfect loop
for (k = 0; k < K; ++k) {
                RB[k] = A[0][k];
                CB[k] = B[k][0];
}
for (int n = 0; n < N; ++n) {
  for (int m = 0; m < M; ++m) {
    for(int k = 0; k < K; ++k) {
      if( k == 0 ) { acc = C[n][m]; }
      acc += ((n%2)== 0) ? RB[k] : R2B[k]) *
              ((m%2)== 0) ? CB[k] : C2B[k]);
      if( m%2 == 0) {   // prefetch column buffer
        C2B[k] = B[k](m+1)%M];
      } else {
        CB[k] = B[k][(m+1)%M];
      }
      if((n%2) == 0 & m ==0) { // prefetch row buffer
        R2B[m] = A[n+1][k];
      } else {
        RB[m] = A[n+1][k];;
      }
      if( k == (K-1)) {
        C[n][m] = acc;  }
} } }
```


https://www.researchgate.net/publication/336588625_Application-Specific_Memory_Access_by_Prefetching_via_High_Level_Synthesis

Memory-based Shifter

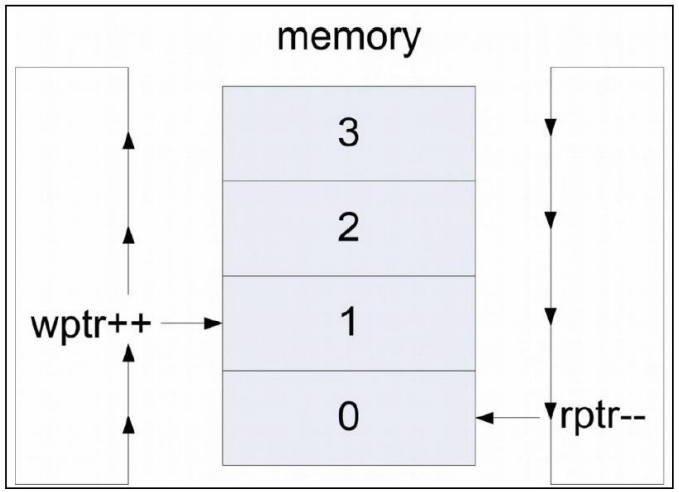
Memory-based Shift Register

- Impractical to use registers
- Read and write to the same memory => Fully rolled

```
void shift_reg(dType din, dType dout[N_REGS]) {

  static dType regs[N_REGS];
#pragma HLS RESOURCE variable=regs
core=RAM_1P_BRAM

SHIFT:
  for (int i = N_REGS - 1; i >= 0; i--) {
    if (i == 0)
      regs[i] = din;
    else
      regs[i] = regs[i - 1];
  }

WRITE:
  for (int i = 0; i < N_REGS; i++) {
    dout[i] = regs[i];
  }
}
```

Pointer-based Circular Buffer

- Use read/write pointer to implement circular buffer
- Read/write pointers in registers allows concurrent updates
- Shift in data -> advance write pointer
- Read data with index




```
// shift_reg.cpp
void circular_shift_reg(dType din,
dType dout[N_REGS]) {

  static circular_shift<dType,
N_REGS> regs;

SHIFT:
  regs << din;

WRITE:
  for (int i = 0; i < N_REGS; i++) {
                dout[i] = regs[i];
  }
}
```

```
// circular_shift.h
template <typename T, int N>
class circular_shift {
 private:
  T mem[N];
  ap_int<ADDRESS_BITWIDTH+1> wptr;
  ap_int<ADDRESS_BITWIDTH+1> rptr;

 public:
  circular_shift() {
#pragma HLS RESOURCE variable=mem core=RAM_1P_BRAM
   T dummy;
    wptr = 0;
    rptr = 0;
    for (int i = 0; i < N; i++) mem[i] = dummy;
  }

  void operator<<(T data) {
    mem[wptr] = data;
    wptr++;
    if (wptr == N) wptr = 0;
  }

  T operator[](ap_uint<ADDRESS_BITWIDTH> idx) {
    rptr = (wptr - 1 - idx);
    if (rptr < 0) rptr = rptr + N;
    return mem[rptr];
  }
```

Techniques in memory access

- Loop transformation
  - improve temporal, and spatial locality
- Scratch-pad memory
  - User-defined explicit data transfer
- Prefetch
  - Eliminate memory latency, allow Burst transfer
- Hash Address / Interleave
  - Minimize memory bank conflict
  - Improve irregular problem size/performance characteristics caused by alias
- Reduce Address expression complexity

https://www.academia.edu/19796541/Constructing_Application_Specific_Memory_Hierarchies_on_FPGAs?email_work_card=view-paper

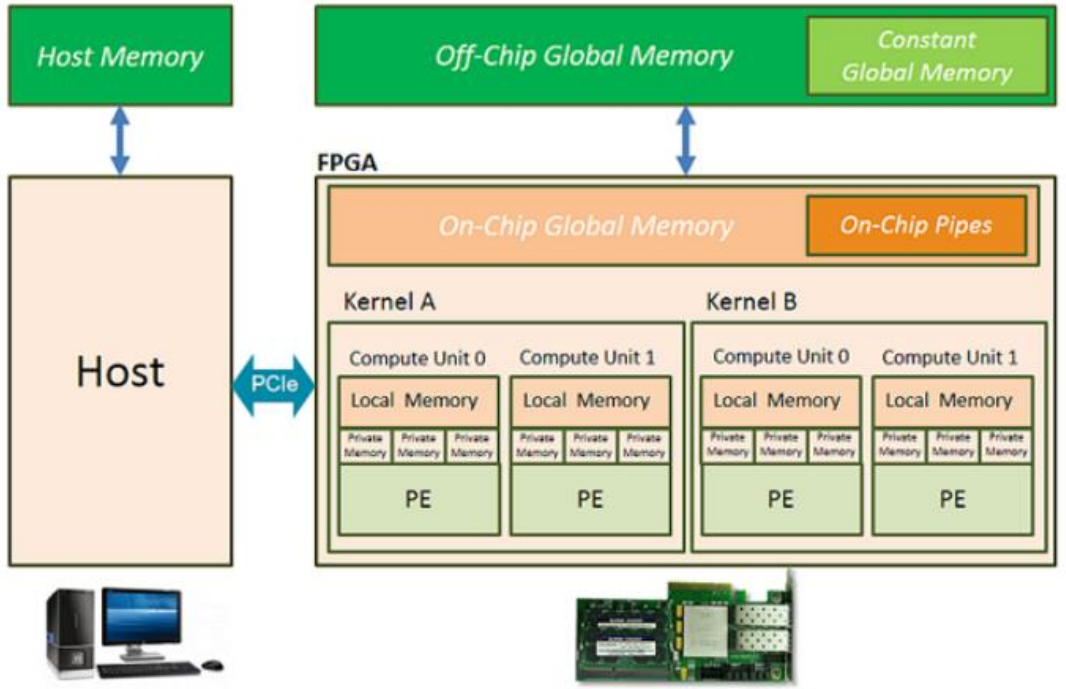
Memory Hierarchy

https://www.academia.edu/19796541/Constructing_Application_Specific_Memory_Hierarchies_on_FPGAs?email_work_card=view-paper

OpenCL Memory Hierarchy

- Host Memory
  - Directly accessible from the host processor.
  - Transfer to FPGA global memory for Kernel
- (Constant) Global Memory
  - Memory attached to FPGA device.
  - Accessible to both host and FPGA device.
  - Handshake protocol:
    - host memory->global memory,
    - kernel access global memory,
    - global memory->host memory
  - Host responsible for the allocation and de-allocation of global memory.
- Local Memory (BlockRAM/UltraRAM)
  - Local to a single compute unit. (CU)
  - Accessible by multiple PEs (work-items)
  - Un-ordered between work-items (synchronization by barrier, and fence)
- Private Memory (Registers,BlockRAM)
  - Private to an individual work-item (PE) in FPGA.

Figure: OpenCL Memory Model



Specify memory bank: https://xilinx.github.io/XRT/master/html/opencl_extension.html
https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/bop1504034296858.html

Memory Hierarchy CPU v.s. FPGA

- Processor
  - Register -> L1 -> L2 -> L3 -> DDR (vertical)
  - Application adjusted to fit these size
  - Cache replacement policy
  - Scratch-pad memory (user controlled, embedded processor
  - Prefetching
- FPGA – user-defined explicit data transfer
  - Wider memory buffer – horizontal, tremendous on-chip bandwidth
  - Dual port memory – transfer data between DDR and parallel with operation, different clock.
  - Similar to scratch pad memory, customize access pattern
- Processor access vertically, FPGA access horizontally parallel (Wider Bus)

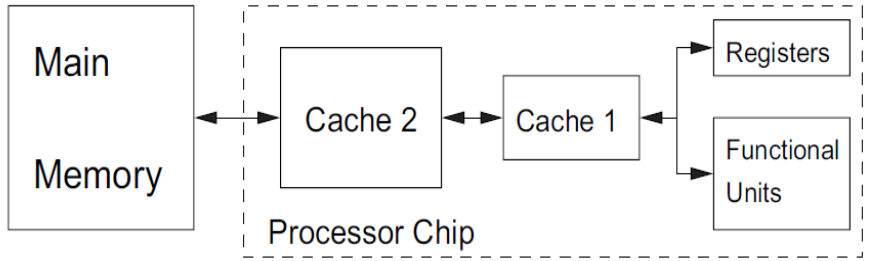


Fig. 2. Memory hierarchy on a typical processor

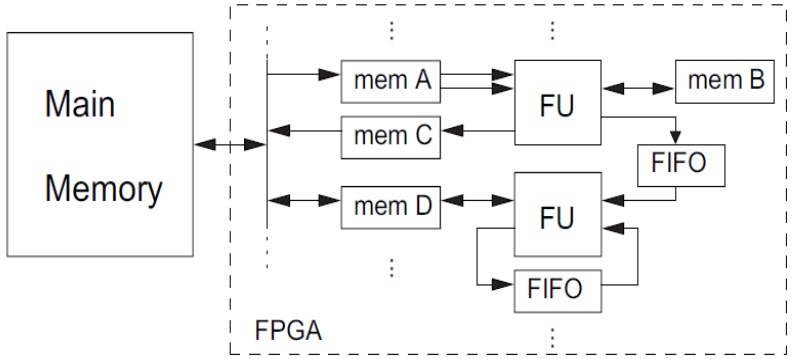


Fig. 3. Example memory hierarchy on an FPGA (FU = Functional Unit)

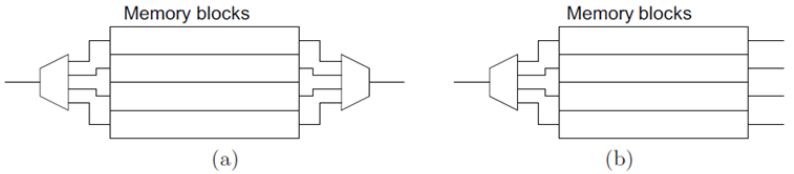


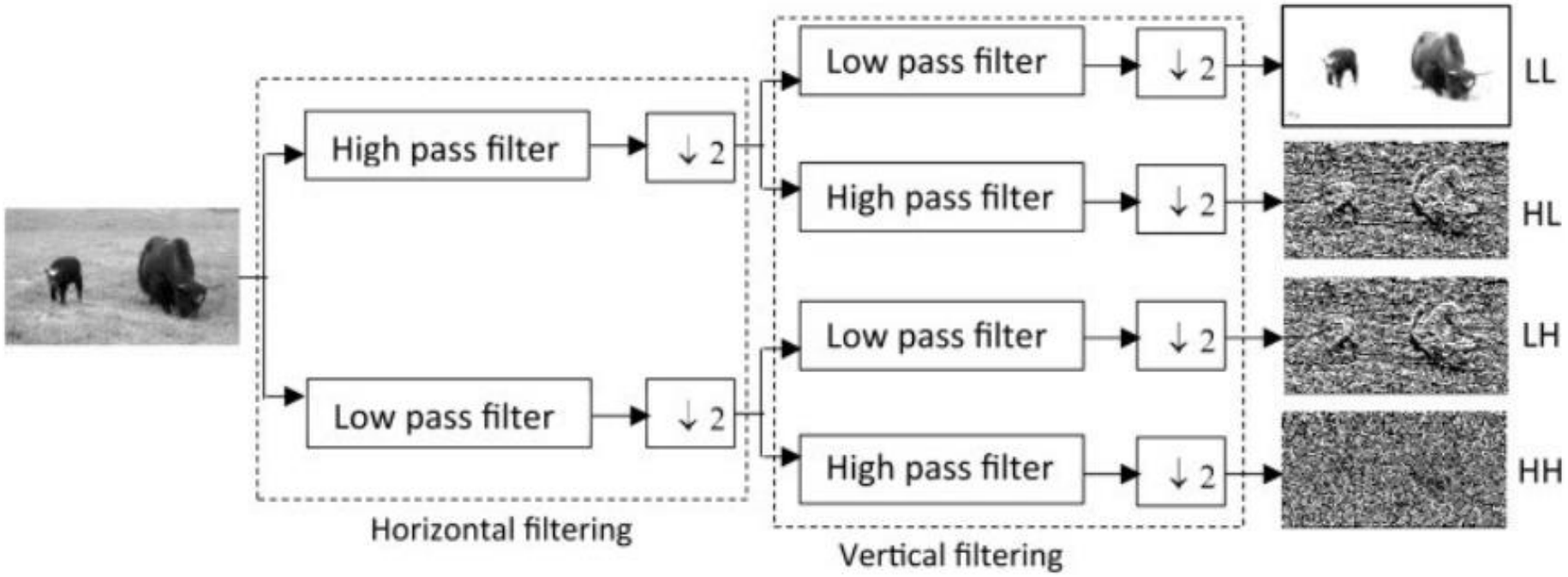
Fig. 4. Multiple memory blocks combined with (de)multiplexers to form larger buffers (a) can be accessed in parallel (b).

Techniques in memory access

- Loop transformation
  - improve temporal, and spatial locality
- Scratch-pad memory
  - User-defined explicit data transfer
- Prefetch
  - Eliminate memory latency, allow Burst transfer
- Hash Address / Interleave
  - Minimize memory bank conflict
  - Improve irregular problem size/performance characteristics caused by alias
- Reduce Address expression complexity

https://www.academia.edu/19796541/Constructing_Application_Specific_Memory_Hierarchies_on_FPGAs?email_work_card=view-paper

An Example: 2D DWT – Discrete Wavelet Transformation



Decomposition of an image 2-D discrete wavelet transform (2-D DWT). Source — Parida et al. , 2017. Wavelet based transition region extraction for image segmentation. Future Computing and Informatics Journal.

https://medium.com/@koushikc2000/2d-discrete-wavelet-transformation-and-its-applications-in-digital-image-processing-using-matlab-1f5c68672de3

Steps to Construct Memory Hierarchy

1. On-chip memories added for intermediate datasets. FIFO is used if meet requirements.
2. Buffers inserted for external memory to kernel. Single copy transaction of all data, copied at start the execution and a single transfer to external memory at the end. Separate I/O module. -> adapt to different platform.
3. IO operation are split into smaller prefetch and store operation. only a small amount of data in the buffers is alive. Synchronization between data transfers and compute operation.
4. Buffers resized by remapping – hash/cache translate the indices of the arrays into the new address, e.g. strided mapping. -> scalable with on-chip memory size.
5. If needed, partition buffer to allow parallel access
6. External memories (step 3) merged to form one main memory with different base address. I/O module does arbitration between transfers.
7. Address expression optimization. (step 1, 2, 4, 6)

https://www.academia.edu/19796541/Constructing_Application_Specific_Memory_Hierarchies_on_FPGAs?email_work_card=view-paper

Example: 2D Discrete Wavelet Transformation (DWT)

Baseline Implementation

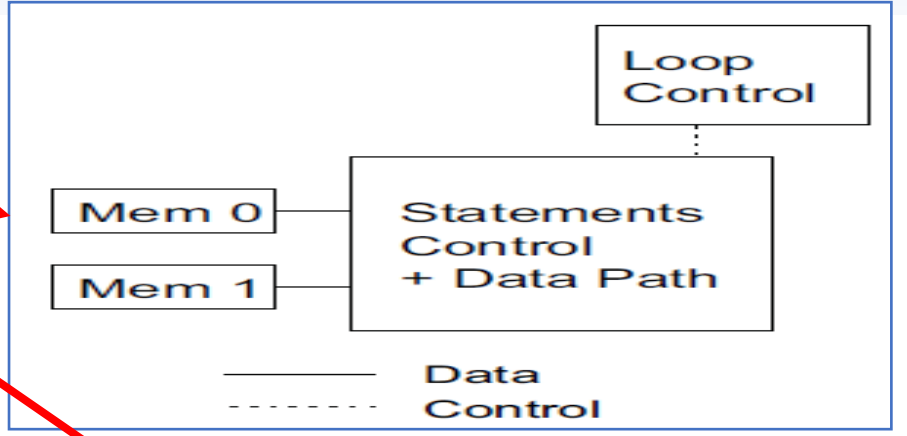
1 data locality optimized by loop transformation => Line-based

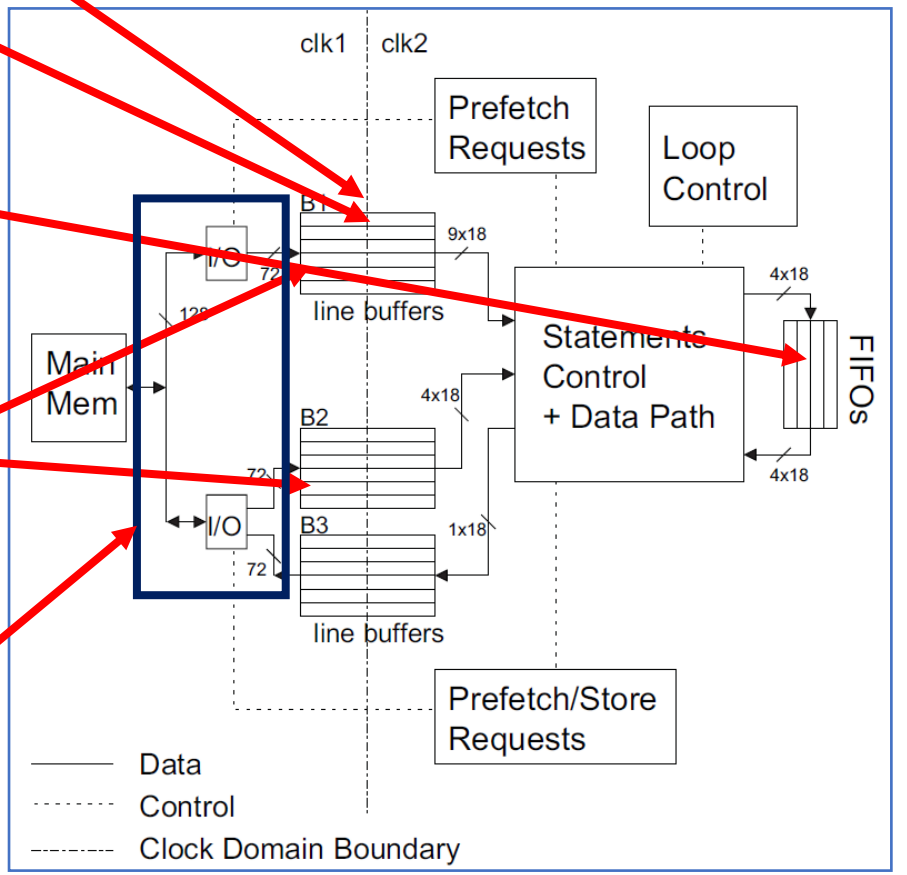
2. Replace with FIFO, access with push/pop, more data elements accessed in parallel

3. FIFO inserted from vertical to horizontal wavelet transformation. In parallel with memory IO access.
4. Align image lines to multiple of 512

5. Mem 0/Mem 1 original size, reduced to Line Buffer B1 (16), B2 (8), B3 (8). Address expressions use modulo expression.
6. B1, B2 split into parallel accessible buffers of one line to increase the on-chip bandwidth.

7. Mem0, Mem1 transfer scheduling by queue of prefetch and store requests (Preferch/Synchronization)
8. Mem0, Mem1 Merged into Main memory with address scheme





https://www.academia.edu/19796541/Constructing_Application_Specific_Memory_Hierarchies_on_FPGAs?email_work_card=view-paper

Reference

- Register promotion: https://dl.acm.org/doi/pdf/10.1145/258915.258943
- https://github.com/Inference-and-Optimization/Design-of-FPGA-Based-Computing-Systems-with-OpenCL/blob/master/CH5-Exploiting-the-Memory-Hierarchy.md
- High-Performance Sparse Linear Algebra on HBM-Equipped https://scholar.google.com.tw/scholar_url?url=https://www.csl.cornell.edu/~zhiruz/pdfs/spmv-fpga2022.pdf&hl=en&sa=X&ei=rC1KY6GzMuSN6rQPiPKJwA0&scisig=AAGBfm2EQnsD7WHaPrn5acxChTY_OfcwAA&oi=scholarr
- 2D Discrete Wavelet Transformation https://medium.com/@koushikc2000/2d-discrete-wavelet-transformation-and-its-applications-in-digital-image-processing-using-matlab-1f5c68672de3
- Double Pump
  - https://support.xilinx.com/s/question/0D52E00006hpfBlSAI/double-pumped-asymmetric-sdp-bram-inference?language=ja
  - https://www.researchgate.net/profile/Arda-Yurdakul/publication/262398113_Efficient_Implementations_of_Multi-pumped_Multi-port_Register_Files_in_FPGAs/links/53eb57160cf28f342f453275/Efficient-Implementations-of-Multi-pumped-Multi-port-Register-Files-in-FPGAs.pdf
- Intel Advisor Cookbook: https://www.intel.com/content/www/us/en/develop/documentation/advisor-cookbook/top.html
- https://www.intel.com/content/www/us/en/develop/documentation/advisor-cookbook/top/optimize-memory-access-patterns.html