

HW#3 Cache Optimization



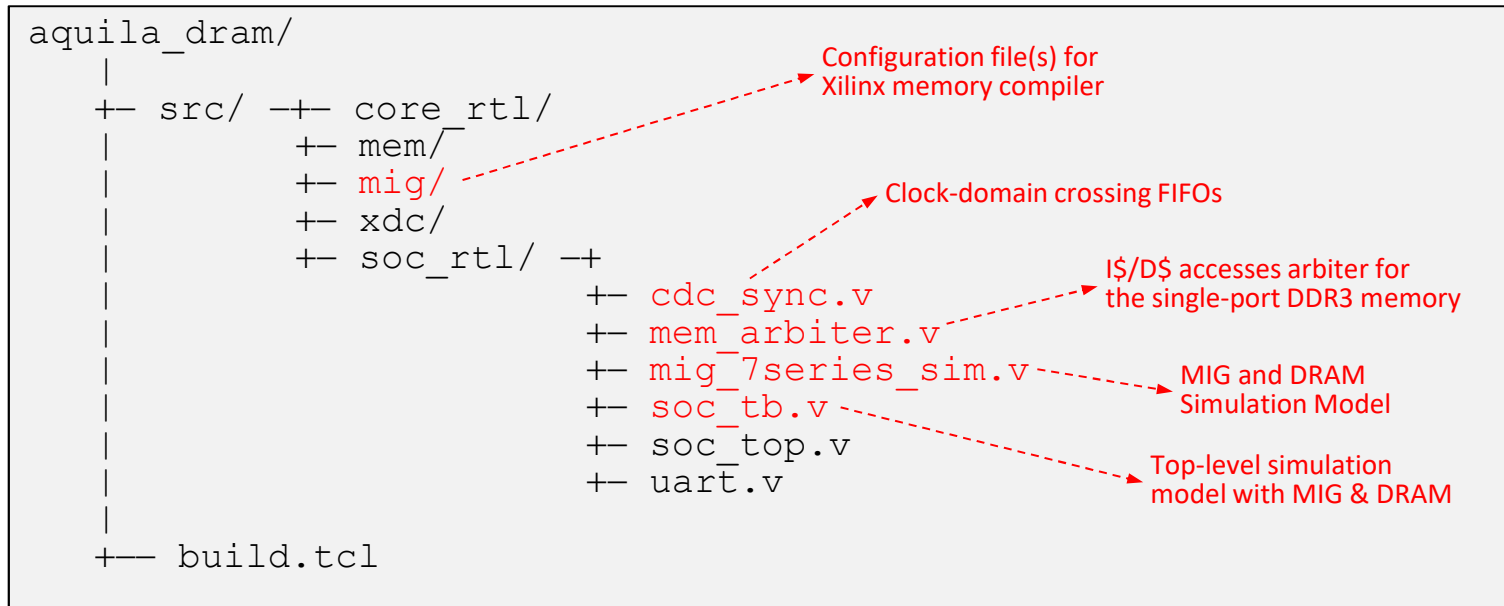
Chun-Jen Tsai
NYCU
11/3/2023

Homework Goal

- ❑ Caches are crucial for microprocessors performance. In this HW, we must analyze and optimize the data cache for the CoreMark benchmark
 - We can configure the DRAM usage of CoreMark
- ❑ Your tasks:
 - Analyze the cache behavior over CoreMark execution
 - Optimize the data cache to improve CoreMark
- ❑ Upload your code & report to E3 by 11/17, 17:00.

Aquila SoC with DRAM Support

- ❑ For this homework, download the new HW workspace `aquila_dram.zip` from E3:



Some Simple Statistics

- ❑ Aquila has a pair of 4-way set associative I/D-caches. Default cache sizes is set to 8KB, the logic usage is
 - 19% usage of LUT
 - 6% usage of LUTRAM
 - 8% usage of FF
 - 25% usage of BRAM

- ❑ With the default DRAM usage and `-DITERATIONS=0`, the CoreMark/MHz are:
 - Running on TCM: 2.008
 - Running on DRAM, 8KB D\$: 2.008
 - Running on DRAM, 4KB D\$: 2.008
 - Running on DRAM, 2KB D\$: 1.993

CoreMark, by default, requires about 2KB ~ 4KB of DRAM space.

Increasing DRAM Usage of CoreMark

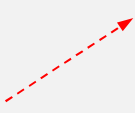
- ❑ In order to do see the effect of D\$ improvement, you **MUST** increase the DRAM Usage of CoreMark
 - Modify the bottom of Makefile as follows:

```
.....  
%.elf: XCFLAGS = -DITERATIONS=0 -DTOTAL_DATA_SIZE=4000 -DUSE_CLOCK=1  
%.elf: $(OBJS) libellibc.a  
    $(LD) $(LDFLAGS) -T$*_$(MEM_TYPE).ld $(OBJS) -lelibc $(LD_SOFT_FP) -o $@  
    $(OD) $(ODFLAGS) $@ > $*.objdump  
    $(SP) $@  
    rm $^
```

- Type “make ddr” to build a coremark.elf for DRAM

- ❑ The result of increasing data size for 8KB D\$:

```
-----  
coremark initializing ... executing, wait about 10 seconds ...  
  
CoreMark Size      : 1333  
Total ticks        : 12044945  
Total time (secs)  : 12.044945  
Iterations/Sec     : 24.906714  
Iterations         : 300  
.....
```

 The number is 25.265013 for execution using TCM.

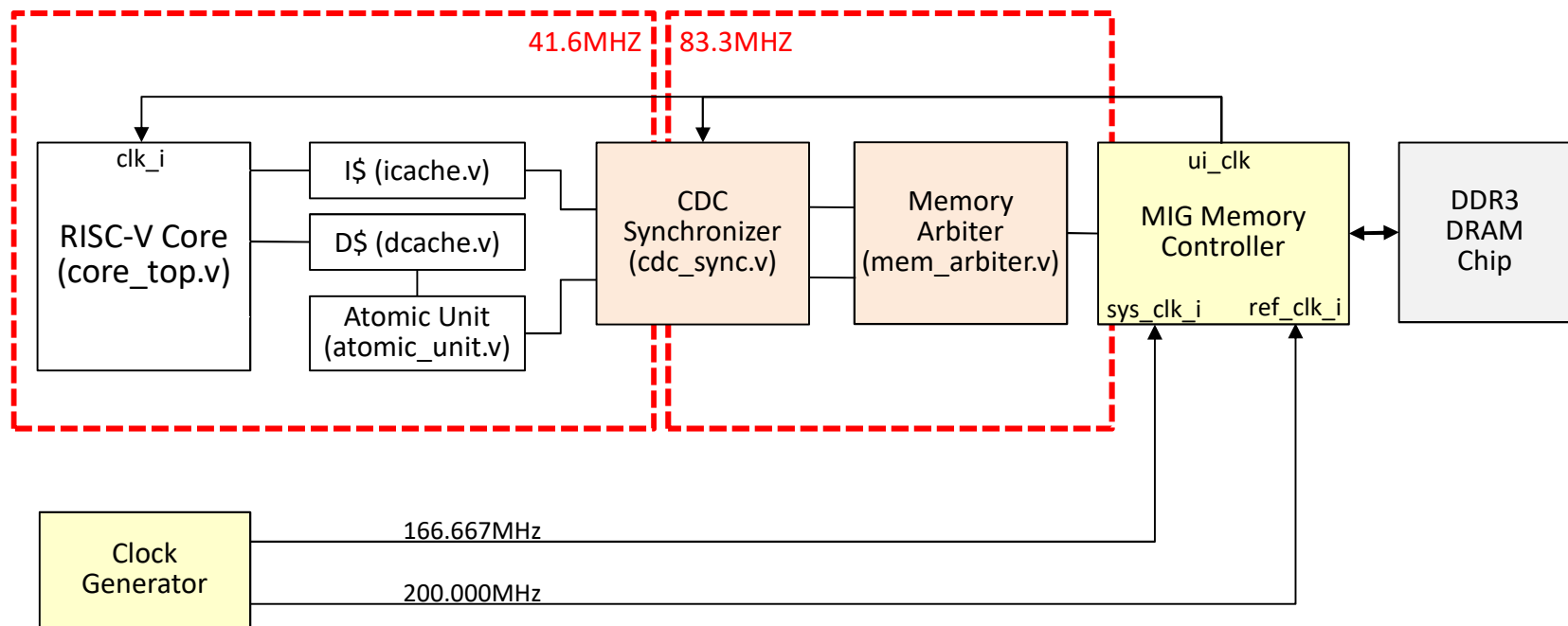
DRAM Interface of Aquila (1/2)

- ❑ The DRAM Chip on Arty is a Micron *MT41K128M16JT-125*
 - The chip is a 16-bit DDR3-1600 IC, but is under-clocked at 333.333 MHz (equivalent to a DDR3-667)
 - The memory controller we use is from Xilinx, called MIG
 - To support 333MHz DRAM clock, the MIG must run at $333/4 = 83.333$ MHz or $333/2 = 166.667$ MHz

- ❑ 83.333 for Aquila on Arty is too high, thus, we choose to use $83.333/2 = 41.6667$ MHz for Aquila

DRAM Interface of Aquila (2/2)

- ❑ Both instruction and data memory shares the same DRAM, so we must add an arbiter and a CDC synchronizer to the system:



Handling Aquila Memory Requests

- ❑ Unlike for TCM, a memory request for data cache may take multiple cycles to complete
- ❑ Aquila uses single-cycle strobe signals for memory requests
 - The D\$ controller must register the input signals: `p_addr_i`, `p_rw_i`, and `p_byte_enable_i` from the processor during the request cycle for multi-cycle processing

Cache Memory Coding Issue

- ❑ In addition to tag and data of cache blocks, each block should record “valid” and “dirty” flags:
 - Valid – the cache line contains valid data
 - Dirty – the data in the cache line have been modified
- ❑ Memory blocks can be synthesized using LUTRAMs (aka distributed memory) or BRAMs
 - Aquila uses LUTRAMs to store valid bits and dirty bits, and uses BRAM to store TAGs and cache blocks
 - For a small cache, BRAM are used wastefully:
 - BRAMs are allocated in 36-kbit or 18-kbit unit for each cache way
 - Vivado uses same number of BRAMs to synthesize 4 & 8KB D\$s

Implementing 1- or 2-Port Memory

- ❑ A memory block can be implemented using LUTRAMs, Flip-Flops, or BRAMs cells of the FPGA

- ❑ How do you control the implementation methods?

- By proper inference coding style (see next slide)
- Or, use a pragma declaration (may not be honored):

```
(* ram_style = "block" *) reg [0:31] my_ram [127:0];
```

- Type of RAM styles are: `block`, `distributed`, or `ultra`
- Whether the pragma can be satisfied or not depends on:
 - How many accesses on the memory block per clock cycle?
 - How many clocks do you apply to synchronize the accesses?

Inferencing Memory Blocks in FPGA

- ❑ The write port of a memory block should always be synchronous (i.e. updated at clock edge); the read port can be either
 - Asynchronous read port → synthesize into LUTRAM
 - Synchronous read port → synthesize into BRAM

```
reg [DATA_WIDTH-1:0] RAM [1024:0];

assign data_o = RAM[read_addr_i];

always @(posedge clk_i)
begin
    if (we_i)
    begin
        RAM[write_addr_i] <= data_i;
    end
end
```

LUTRAM

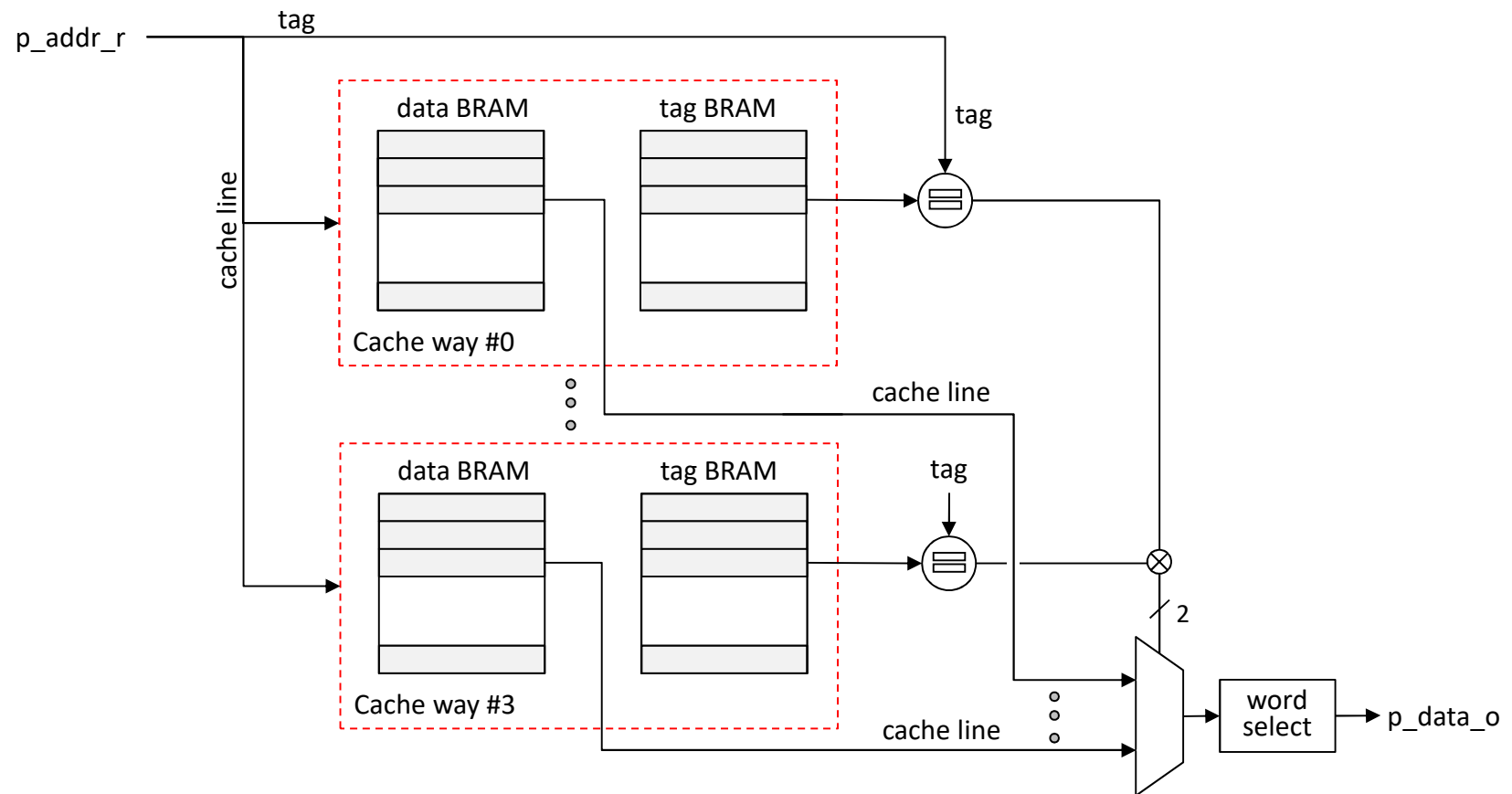
```
reg [DATA_WIDTH-1:0] RAM [1024:0];

always @(posedge clk_i)
begin
    if (en_i)
    begin
        if (we_i)
        begin
            RAM[addr_i] <= data_i;
            data_o <= data_i;
        end
    else
        data_o <= RAM[addr_i];
    end
end
```

BRAM

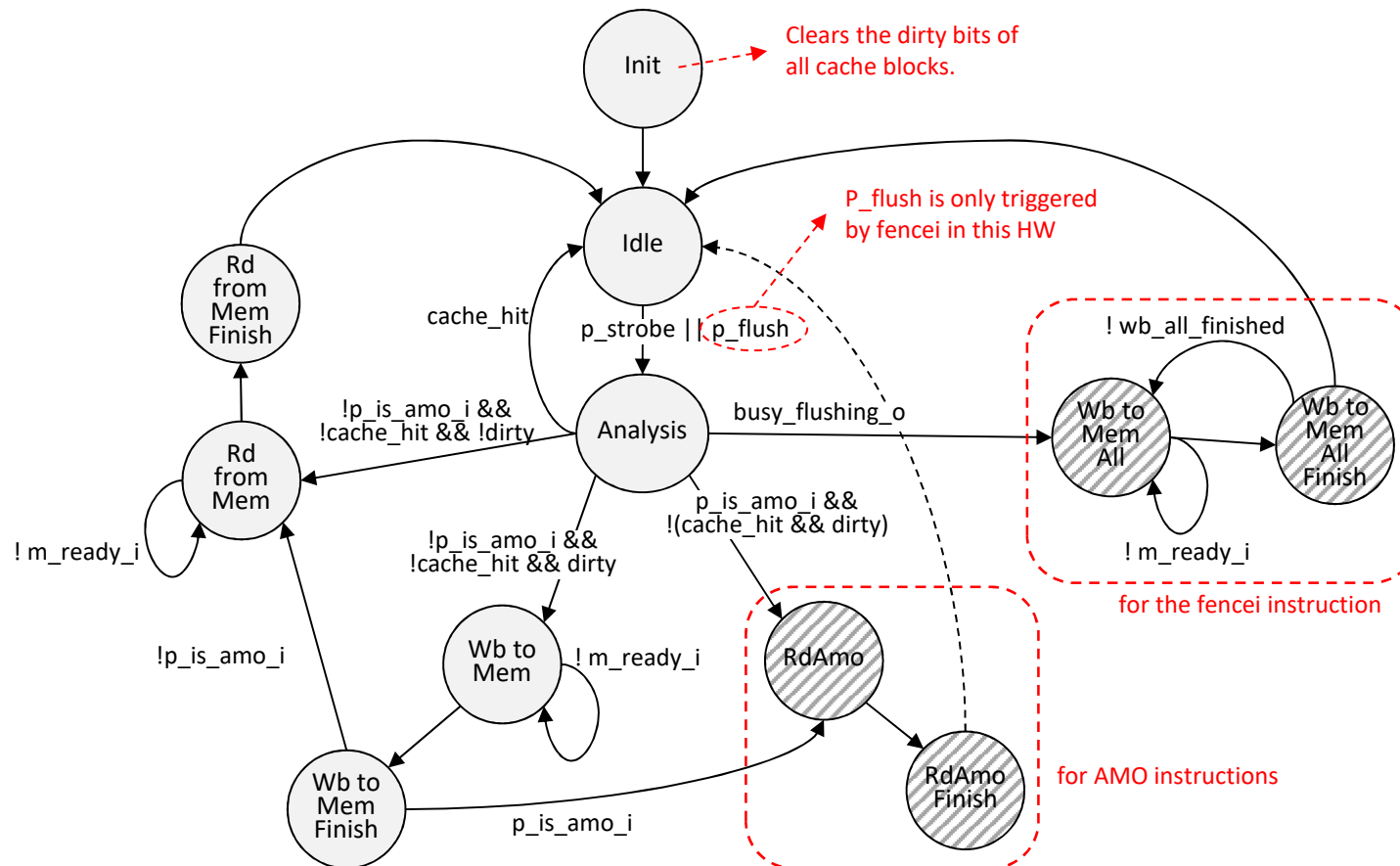
Cache Organization of Aquila SoC

❑ Data flow on cache hit:



Data Cache Controller

- ❑ The FSM of the D-Cache controller is as follows:
 - You can ignore the shaded states for this homework!



Measuring D\$ Performance

- ❑ You should add counters in the D\$ controller to collect the following statistics:
 - Average cache latency for each memory request
 - Read/write latency should be separated
 - Miss/hit latency should be separated
 - Cache hit/miss rates
- ❑ By latency, we mean the #cycles between the `p_strobe_i` and `p_ready_o` signals of the D\$ ports

For Performance Improvements

- ❑ There are a few things you can try to improve the D-Cache performance of Aquila:
 - Change cache ways (2- and 8-way caches are worth trying)
 - Changing the local parameter `N_WAYS` is not enough. Several places in `dcache.v` must be modified for different cache ways.
 - For certain applications, 2-way actually gives better results.
 - Change the cache replacement policy
 - Applying a good pre-fetching algorithm
 - Redesign the cache controller to reduce the stall cycles
 - ...

Resource Usage on the FPGA

- ❑ Always check FPGA utilization after implementation:

The screenshot shows the Vivado 2023.1 IDE. The 'Flow Navigator' on the left has 'Open Implemented Design' circled in red with a '1'. The 'SOURCES' pane shows the design hierarchy. The 'UTILIZATION' report is open at the bottom, showing a table of resource usage. The 'Report Utilization' option is circled in red with a '2'. The 'Hierarchy' pane on the left of the utilization report is circled in red with a '3'.

Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)
soc_top	12295	10563	4641	11218	1077	34	4
Aquila_SoC (aquila_top)	7016	4067	2544	6714	302	34	4
ATOM_U (atomic_unit)	9	162	55	9	0	0	0
CLINT (clint)	143	193	72	143	0	0	0
D_Cache (dcache)	1745	644	636	1713	32	10	0
L_Cache (lcache)	1086	349	342	902	184	8	0
RISCV_CORE0 (core_top)	4023	2715	1725	3937	86	0	4

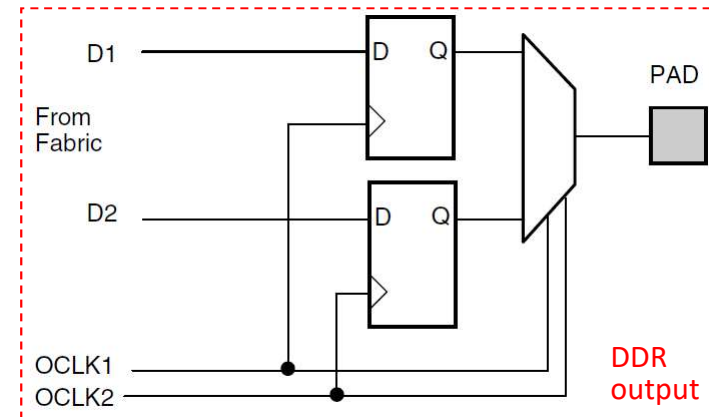
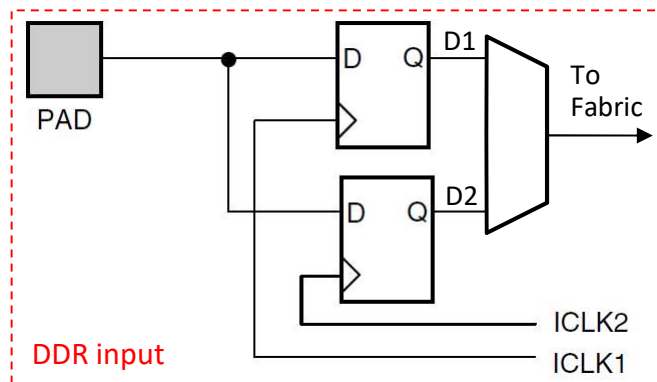
Memory Controller

- ❑ Memory controller connects the processing cores to the main memory
 - Typical main memory is composed of DRAM chips
 - Crucial to data-intensive applications

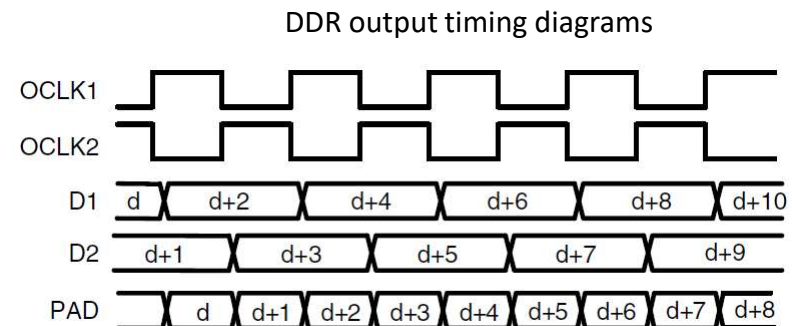
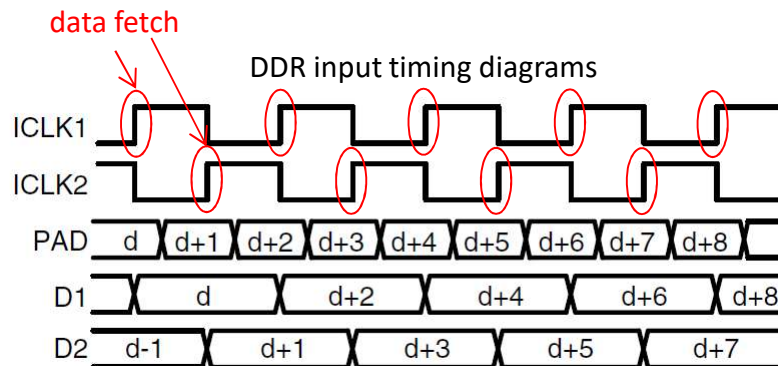
- ❑ Types of DRAM chips
 - SDR – old DRAM that handles one transaction per DRAM clock cycle, up to 133 Mhz
 - DDR – modern DRAM that handles two transactions per DRAM clock cycle, DDR4 can be up to 1.6 GHz

DDRx Memory Controller (1/2)

- We can design a low-speed DRAM controller and connect it to a DRAM chip with generic FPGA user pins



CLK1 and CLK2 are 180° phase shifted



DDRx Memory Controller (2/2)

- ❑ High-speed DDRx memory controller IPs are complicated and requires some dedicated I/O logic
 - Only certain FPGA I/O banks can be used to connect to the high-speed DRAM chips
 - The controller need custom I/O pins to talk to the DRAM chips

- ❑ Xilinx solution for memory controllers
 - Xilinx provides a configurable Memory Interface Generator (MIG) that can be used to generate a memory controller
 - The available DRAM parameters depends on the FPGA family
 - On Kintex devices, DRAM clock up to 800MHz (DDR3-1600)
 - On Artix devices, DRAM clock up to 400MHz (DDR3-800)
 - UltraScale+ devices supports DDR4 chips

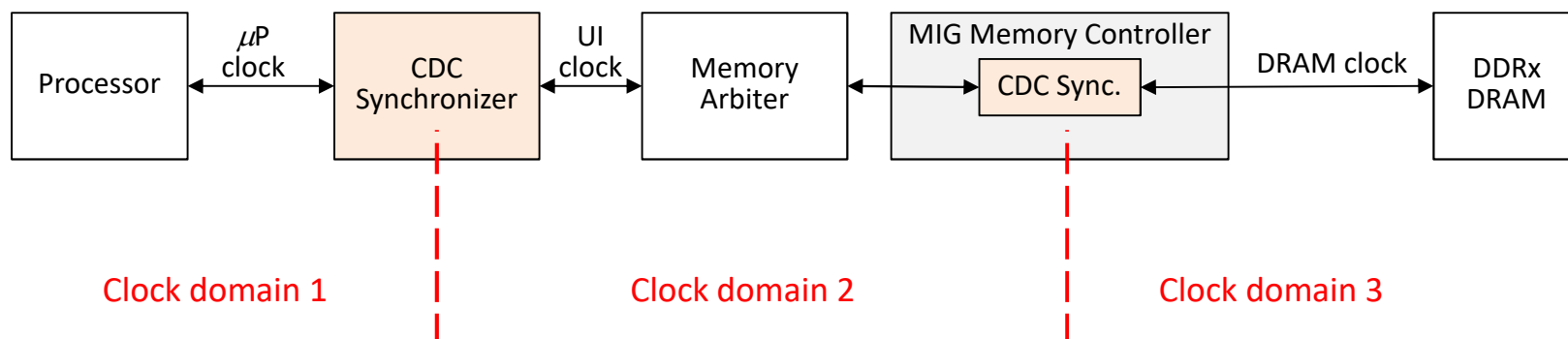
MIG Interface on the Processor Side

- ❑ MIG supports two types of processor side interfaces:
 - AXI interface – easier to use if your processor has AXI-compatible memory ports
 - Native interface – close to the real DRAM chip interface, more efficient to use, but your logic must handle the DRAM burst re-ordering and the large access block issues.

- ❑ For this HW, we choose to use the native MIG interface
 - Aquila has I-Cache and D-Cache so we always access the DRAM on a block basis (128-bit at a time)
 - Burst ordering issue is not hard to handle, we do that in the memory arbiter

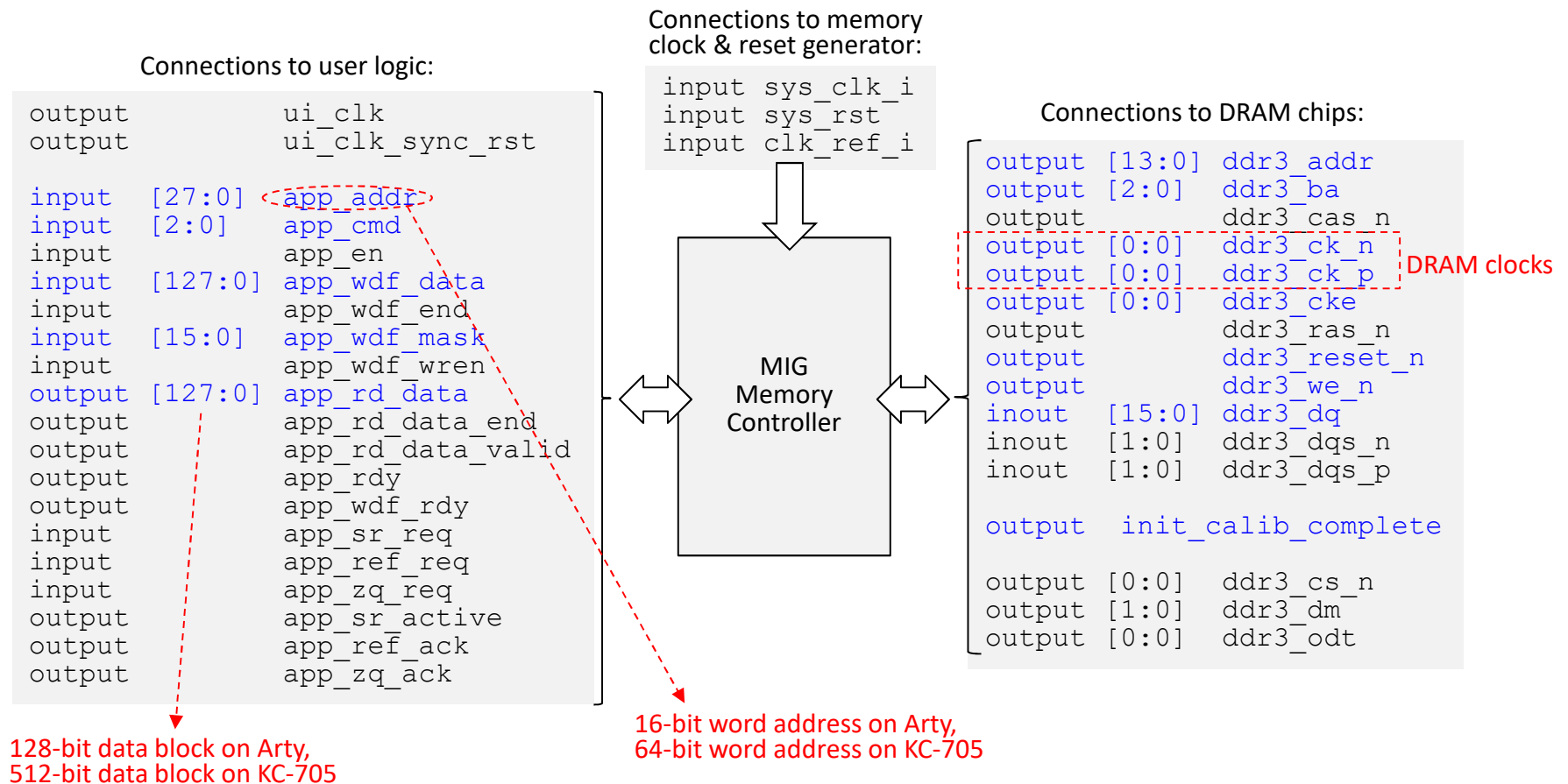
Clock Domain Crossing of MIG

- ❑ The memory controller generated by MIG is a cross-clock domain IP
 - On DRAM side, it runs at `sys_clk` rate (166.667MHz on Arty)
 - On processor side, it runs at `ui_clk` rate (83.333MHz on Arty)
- ❑ If `ui_clk` is too high for the processor, we must produce a slower clock for the processor core
 - In this case, a CDC synchronizer module must be used to connect the processor to the memory controller:



DRAM Native Interface

- Two clock domains of MIG: DRAM & UI (user interface)



Block-based I/O of Memory Controller

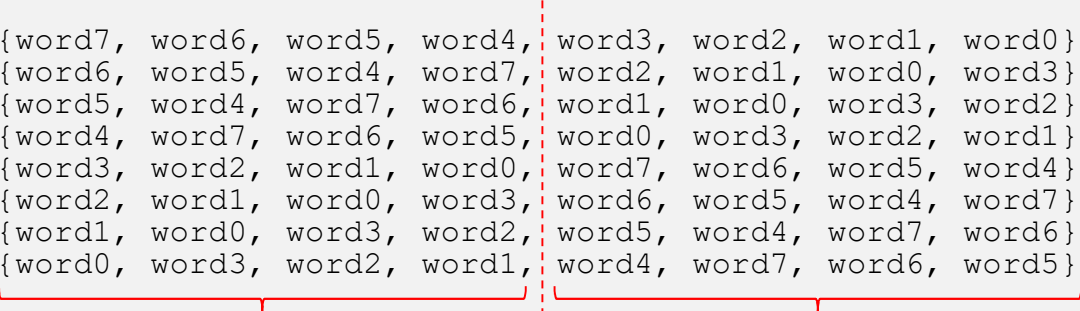
- ❑ DRAM chips typically operates on a row at one time, each read/write operation is for a row of memory cells
 - The memory controller will read/write a large block at one time

- ❑ On Arty, MIG read/write 128-bit data at a time
 - You specify the 16-bit starting word addresses, the memory controller will read 128-bit data that contains the data in the same row of DRAM cells
 - For writing, a mask can be used to specify the words you want to modify

Data Reordering of Transaction Data

- ❑ MIG is hardwired to read/write 8-word burst each time
 - However, DRAM chips output 4-word wrapping burst each time
 - The least significant word contains the `[app_addr]` data
 - For efficiency, a read burst returns data out-of-order
- ❑ On Arty, the following logic is used to re-order the data back to normal order (not really necessary for Aquila):

```
always @(posedge clk_i) begin
    if (rst_i) read_data <= {128{1'b0}};
    else if (read_data_valid_i)
        case(addr_o[2:0])
            3'h0: read_data <= {word7, word6, word5, word4, word3, word2, word1, word0};
            3'h1: read_data <= {word6, word5, word4, word7, word2, word1, word0, word3};
            3'h2: read_data <= {word5, word4, word7, word6, word1, word0, word3, word2};
            3'h3: read_data <= {word4, word7, word6, word5, word0, word3, word2, word1};
            3'h4: read_data <= {word3, word2, word1, word0, word7, word6, word5, word4};
            3'h5: read_data <= {word2, word1, word0, word3, word6, word5, word4, word7};
            3'h6: read_data <= {word1, word0, word3, word2, word5, word4, word7, word6};
            3'h7: read_data <= {word0, word3, word2, word1, word4, word7, word6, word5};
        endcase
end
```



2nd 4-word wrapping burst 1st 4-word wrapping burst

2-to-1 Memory Arbitration

- ❑ Since Aquila has two memory ports (I-Mem & D-Mem) that accesses the DRAM, a 2-to-1 multiplexor must be used to share the single memory controller port
- ❑ For Aquila, instruction fetches have higher priority over data accesses

Your Homework

- ❑ The goal of this homework is to analyze and improve the data cache
 - Note that you shall increase the data size of CoreMark to 8000, or you will not get runtime cache misses on an 8KB D\$
- ❑ Write a report:
 - Analyze the data cache behavior for the CoreMark program
 - Describe the improvements you have done to the data cache
- ❑ Note: it is possible that your work turns out to degrade the performance. You can still discuss why your idea does not work and get a good grade