Application Processors and the Aquila SoC

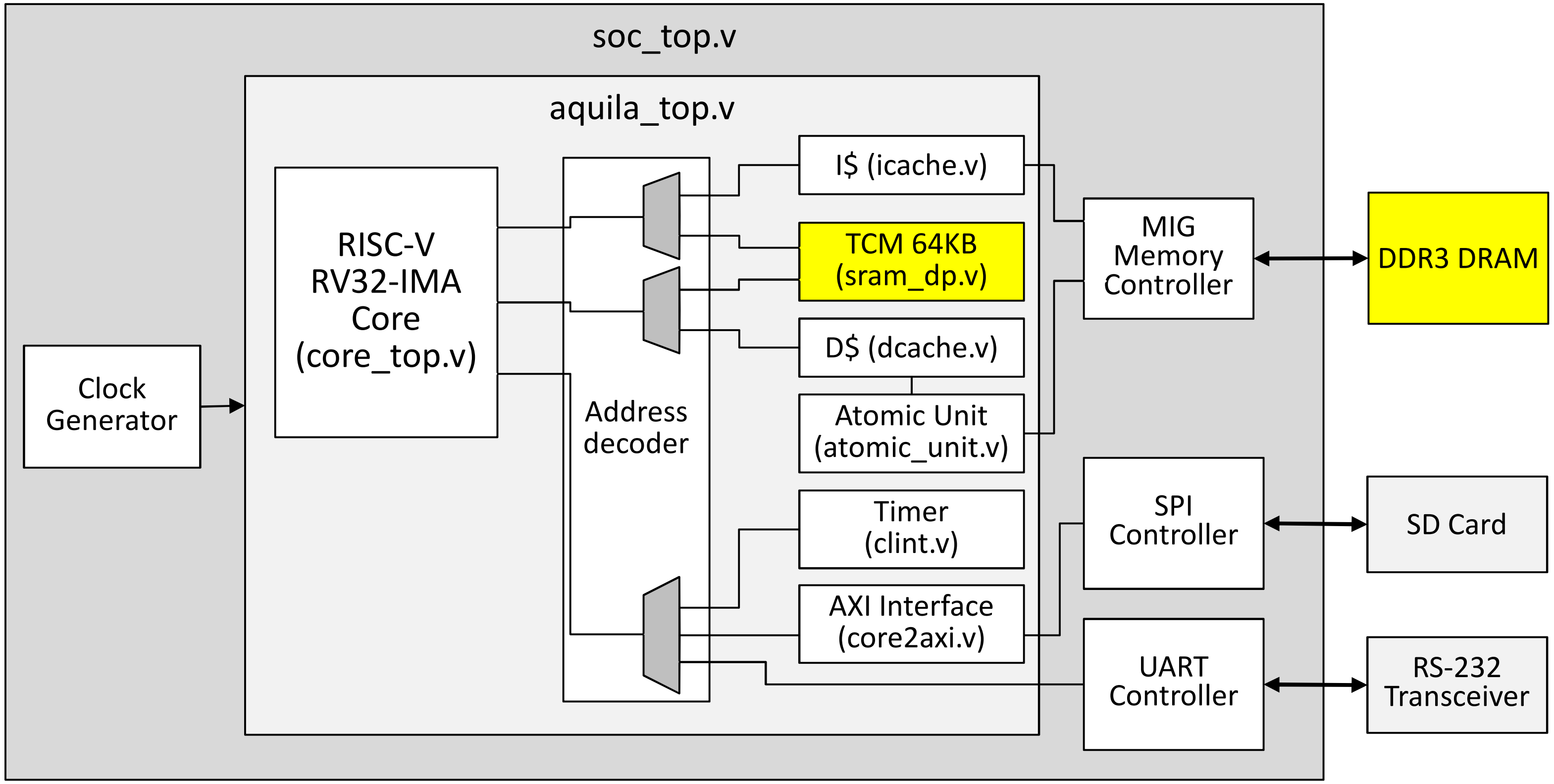
Chun-Jen Tsai
NYCU
10/27/2023

Aquila – An Open Source RISC-V

- The spec of the complete version of Aquila SoC:
  - RV32IMA + Zicsr + Zifencei ISA-compliant
  - Embedded 64KB tightly-coupled on-chip memory (TCM)
  - L1 4-way set associative data and instruction caches
  - CLINT for standard timer interrupts
  - CSRs and complete M-, U-, and S-mode support
  - SV32 Memory Management Unit
  - SD card I/O support
  - Ethernet support
  - The RTL model written in Verilog

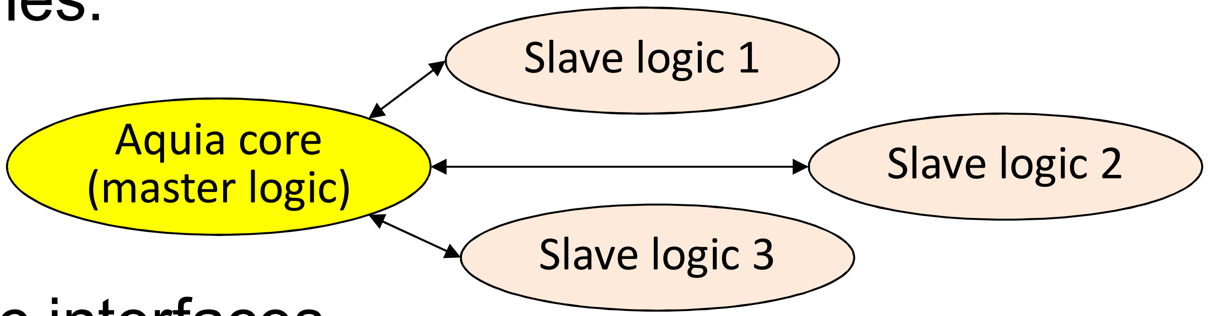
SoC Interfaces (1/2)

- Aquila core can be used as:
  - A reusable IP (using AXI4 bus protocol)
  - A proprietary SoC (using proprietary bus protocol)



SoC Circuit Block Interfaces (2/2)

- Aquila SoC uses several interfaces to integrate slave circuit modules:

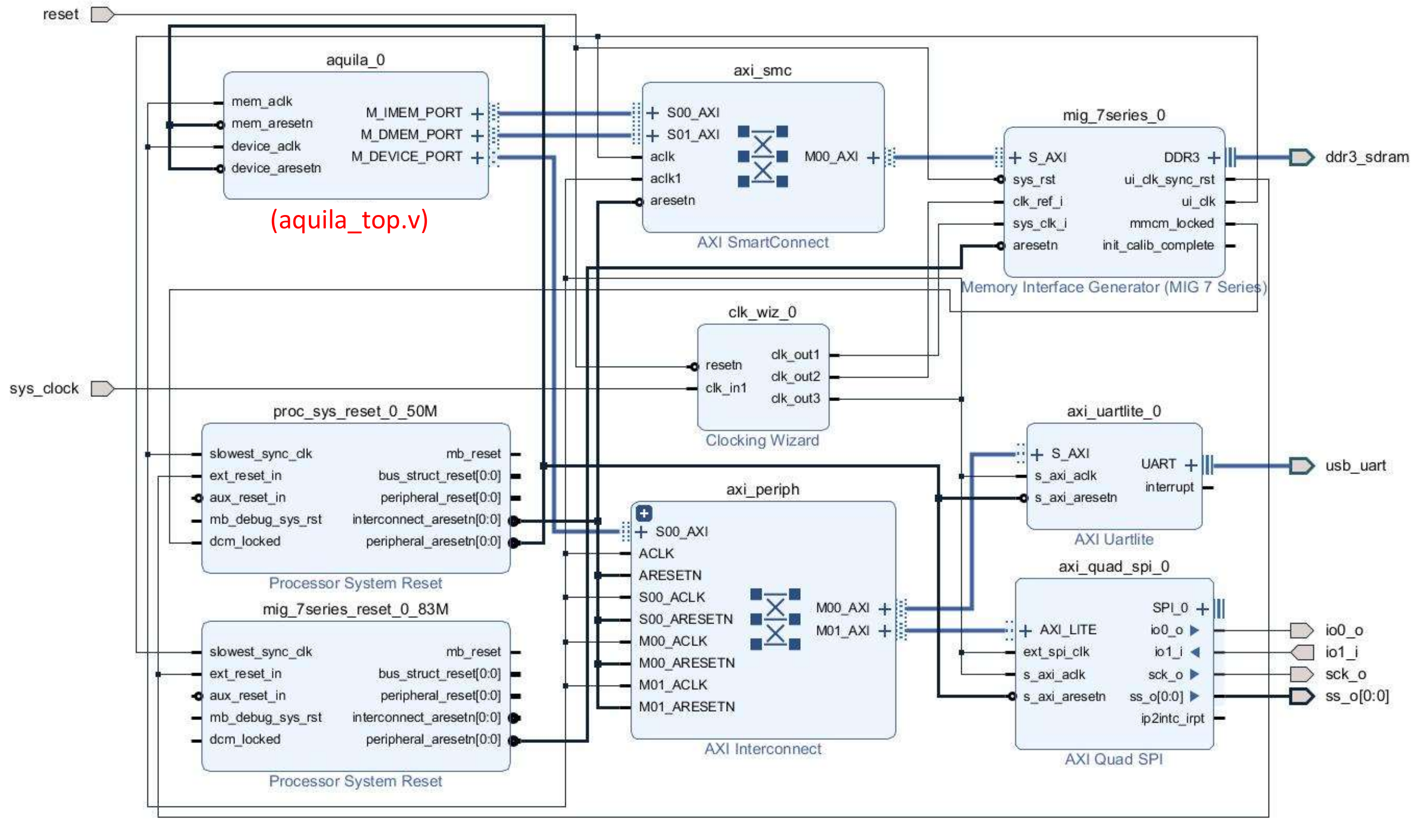


- Custom logic interfaces
  - Native MIG memory controller interface
  - cache controller interface

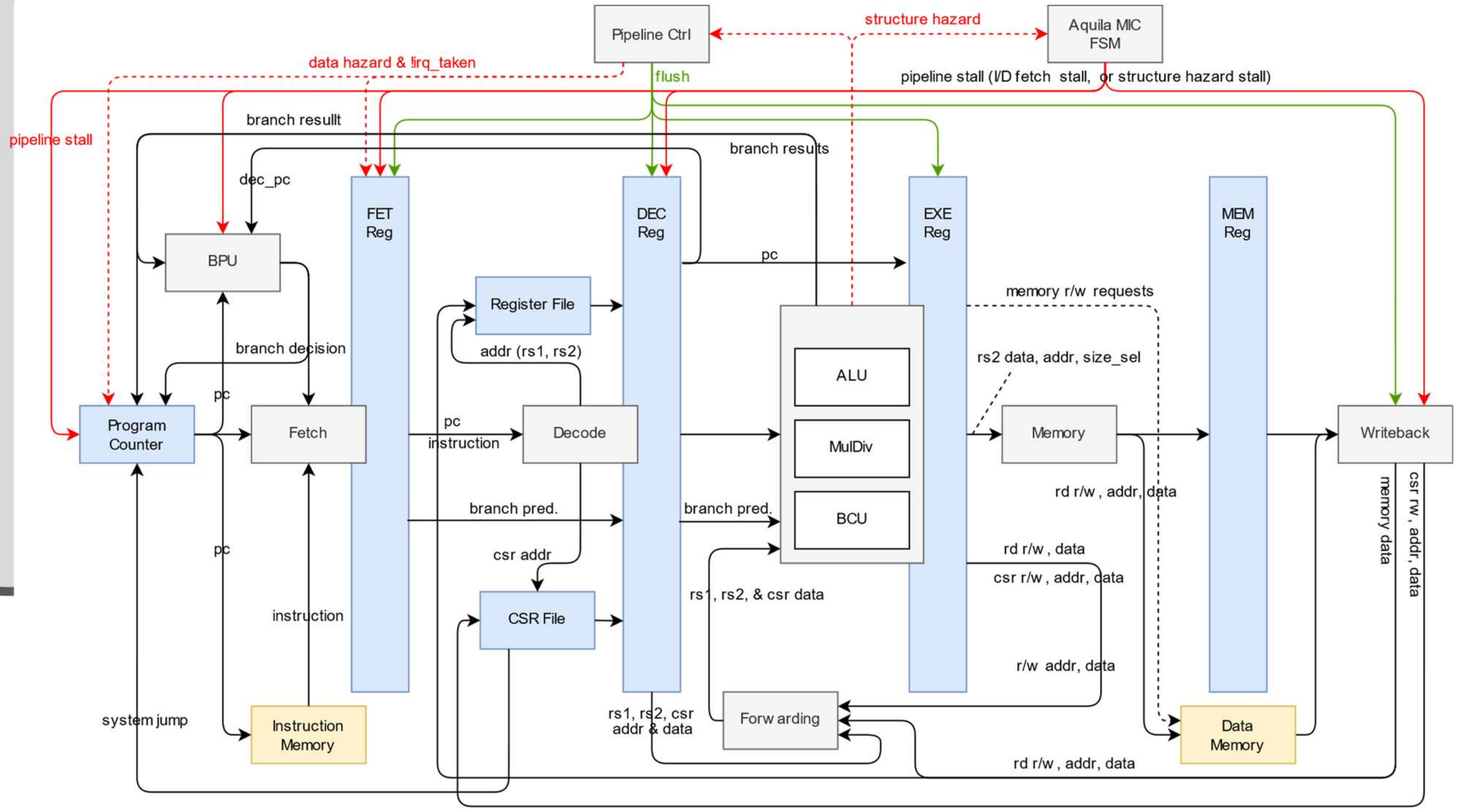
- General-purpose bus interfaces
  - Minimal synchronous handshaking bus: Timer, UART controller
  - Industry-standard bus (AXI): SPI controller for SD card

Aquila as an AXI Reusable IP

- Aquila can also be in a full-AXI infrastructure
  - AXI bus is designed by ARM to ease complex SoC integration
  - Vivado has an GUI-based IP Integrator tool for this purpose:



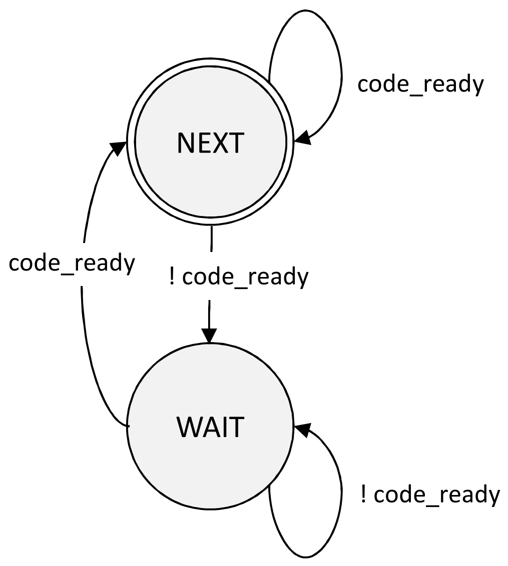
Aquila Pipeline Organization



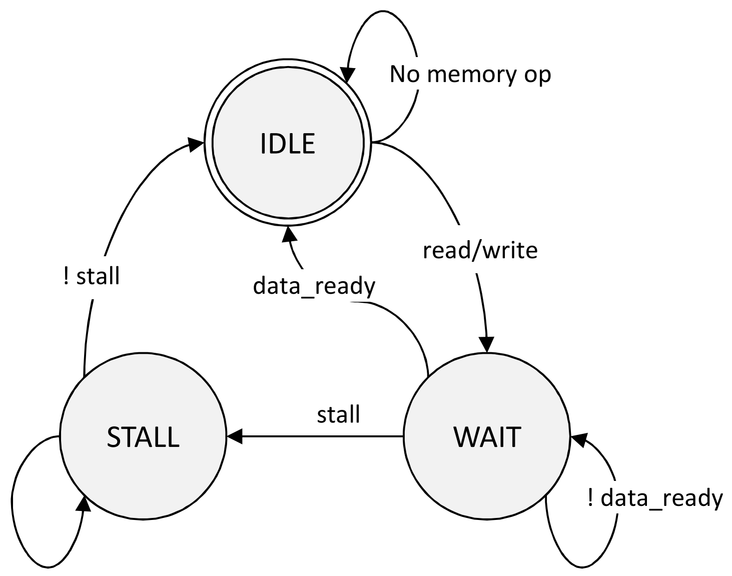
Aquila Memory Interface Controllers

- Memory interface controllers (MIC) in `core_top.v` controls the instruction & data fetch states:

stall = stall_instr_fetch || stall_from_exe



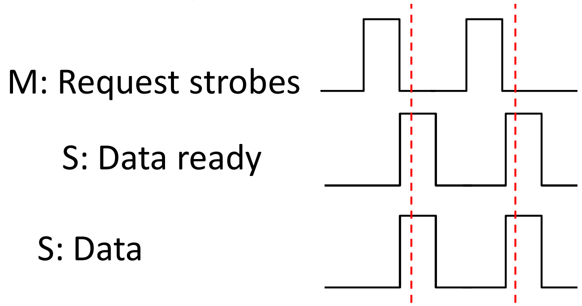
Instruction fetch controller



Data access controller

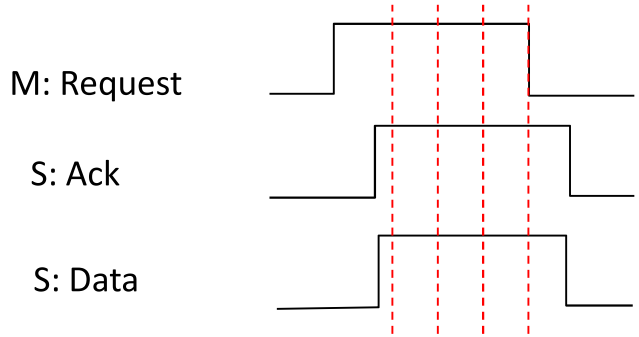
Memory Access Signaling

- Aquila MIC uses the strobing protocol
- Strobing protocol: single-cycle request (read)



M means master, S means slave.

- Handshaking protocol: multi-cycle handshaking (read)



M means master, S means slave.

Pipeline Controller

- Pipeline Controller sends out flush signals and data hazard signals to different pipeline stages

- Different stages are flushed on different conditions:
  - Fetch – mis-branch, sys_jump†, or fence_i
  - Decode – mis-branch, sys_jump, fence_i, load-use hazard, unsupported instruction
  - Execute – sys_jump or fence_i
  - Memory – No register to flush
  - Writeback – sys_jump

† sys_jump means an exception or interrupt happens!

Fetch Stage

- Receives instructions from the instruction memory, and pass it to the Decode unit

- Functions
  - Delay the instruction upon stall
  - Send “NOP” instruction upon flush, invalid instruction
  - Branch prediction is often considered as part of the Fetch stage tasks
  - Detect page fault exceptions when we have MMU
  - Pass exceptions to later stages

Decode Stage

- Extracts different fields from the instruction and sets control and immediate value signals accordingly
  - Very tedious but straightforward to implement

- Functions:
  - Decode instructions
  - Detect data hazard
    - Happens the '`rd`' of the last instruction is the same as either the '`rs1`' or '`rs2`' of the current instruction
  - Pass exceptions to later stages

Execute Stage

- The Execute carries out data manipulation. The critical path of a processor usually involves this stage
  - On FPGAs, the critical path may gets a little bit tricky
- Functions:
  - Single-cycle ALU operations
  - Multi-cycle ALU operations
  - Branch comparison calculations (using exclusive comparators)
  - Branch address computation (using exclusive adder)
  - Sent out memory read/write requests
  - May generate ALU exceptions
  - Pass exceptions to later stages

Multi-Cycle Instructions

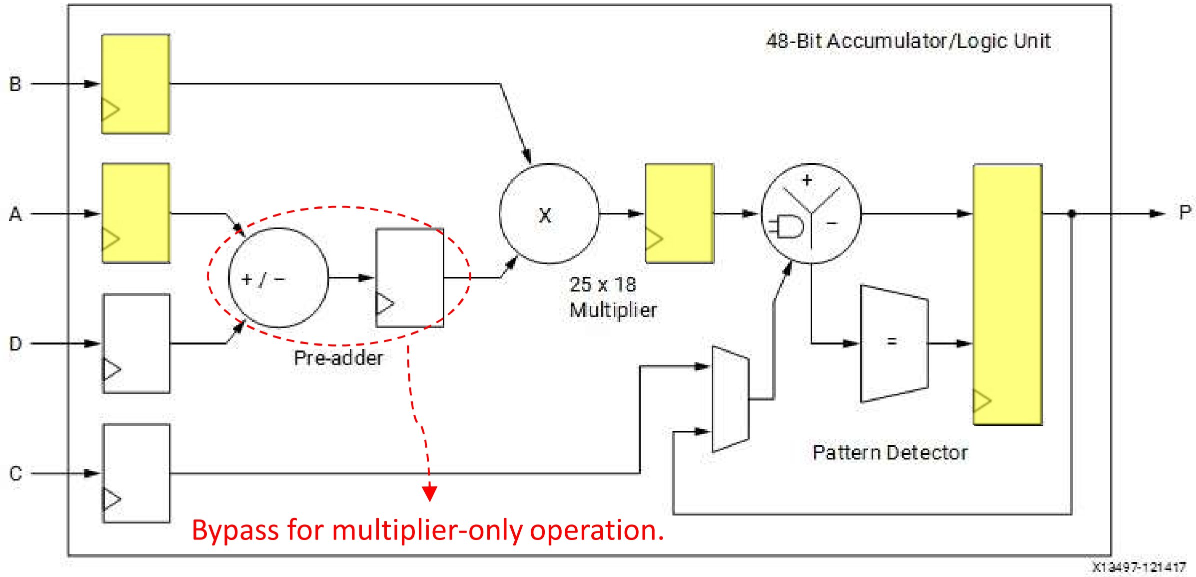
- RISC-V Ext. M is in the MulDiv module
  - This module is more of a template for multi-cycle instructions
  - The FSM for multi-cycle multiply/division:

```
always @(*)
begin
   case (S)
      S_IDLE:
         S_nxt = (req_i)? (is_a_zero | is_b_zero)? S_DONE : S_CALC : S_IDLE;
      S_CALC:
         S_nxt = (is_calc_done)? S_SIGN_ADJUST : S_CALC;
      S_SIGN_ADJUST:
         S_nxt = S_DONE;
      S_DONE:
         S_nxt = (stall_i)? S_STALL : S_IDLE;
      S_STALL:
         S_nxt = (stall_i)? S_STALL : S_IDLE;
      default:
         S_nxt = S_IDLE;
   endcase
end
```

Result cannot be returned during a pipeline stall!

Multiplier on FPGA (1/2)

- On Xilinx FPGAs, multiplier can be implemented using LUTs or DSP48 slices
  - Aquila has both implementations
- For DSP48 multiplier, you can specify the #stages
  - There are three sets of registers along the combinational path
  - Each set can be bypassed for a long combinational path



Multiplier on FPGA (2/2)

- To use DSP slices for 32×32 multiplication, four slices will be used:

$$((a{<<}16)+b) \times ((c{<<}16)+d) = (a\times c){<<}32 + (a\times d){<<}16 + (b\times c){<<}16 + b\times d.$$

- The number of pipeline stages can be inferred using the following code patterns:

```
always @(posedge clk)
begin
    m_r <= v1*v2;
end
```

one-stage

```
always @(posedge clk)
Begin
    v1_r <= v1;
    v2_r <= v2;
    m_r <= v1_r*v2_r;
end
```

two-stage

```
always @(posedge clk)
Begin
    v1_r <= v1;
    v2_r <= v2;
    m0_r <= v1_r * v2_r;
    m_r <= m0_r
end
```

three-stage

Memory Stage

- The memory stage performs the following tasks:
  - Prepare 8-bit or 16-bit data for 32-bit buses
  - Setup the byte-selection signals
  - Detect memory alignment exceptions
  - Write a word to the data memory
- Aquila only supports aligned accesses for 16- or 32-bit data since non-aligned access takes multiple cycles
  - Generate misaligned exception if necessary
- Passing prior signals to the writeback stage
  - Including PC, instruction validity & exception signals
  - These signals are registered at the input of the writeback stage, instead of the output of the memory stage

Writeback Stage

- The Writeback unit performs physical write back to registers
  - The source may come from the Execute unit (through the Memory unit) or the data memory
  - The destination may be the register file, the CSR file, or the Forwarding unit

- Functions:
  - Data write back to registers
  - Perform sign-extension if necessary
  - Handles exception

Atomic Unit

- Atomic unit sits between the data cache and the DDRx memory controller, intercepting all memory requests
  - Each hardware thread (HART) has an ID

- Functions:
  - If the memory request is not from atomic instructions, bypass
  - If it is a lock-based AMO request, execute the operations
  - If it is a lock-free LR/SC request, manage the reservation table (one entry per HART):
    - Register the reservation for LR by its hart ID
    - Check the reservation and allow the first SC to cancel all other ID’s reservations

Application Processors

- An application processor is an SoC contains:
  - Processor cores (ISA-based CPU/GPU/DSP)
  - Application-specific accelerators (none-ISA based)
    - audio/video codecs
    - image processing cores
    - machine-learning/AI cores
    - other application-specific cores
  - On-chip memories
  - Clock-domain infrastructure components

- ISA does not define an application processor

User-Defined Instructions

- An application processor based on the RISC-V ISA may define its own application-specific instructions
  - For 32-bit ISA, two of the 7-bit opcode are reserved for users: 0001011 (custom-0) and 0101011 (custom-1)
  - The rest of the 25-bit pattern can be defined arbitrarily
    - It is recommended to stick to the R/I/S formats

	31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
R	funct7	rs2	rs1	func3	rd	opcode
I	imm[11:0]		rs1	func3	rd	opcode
S	imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode

  - Programming of user-defined instructions can be done using inline assembly

Integration Issues of Multiple Cores

- Different cores often runs at different clock rates, and must exchange data and control information
- Hardware synchronization mechanisms
  - Atomic operations for heterogeneous cores
  - Data sharing
    - Unified coherent cache
    - Multi-port memory
- Clock-Domain Crossing (CDC) schemes
  - Shift registers
  - Asynchronous FIFOs

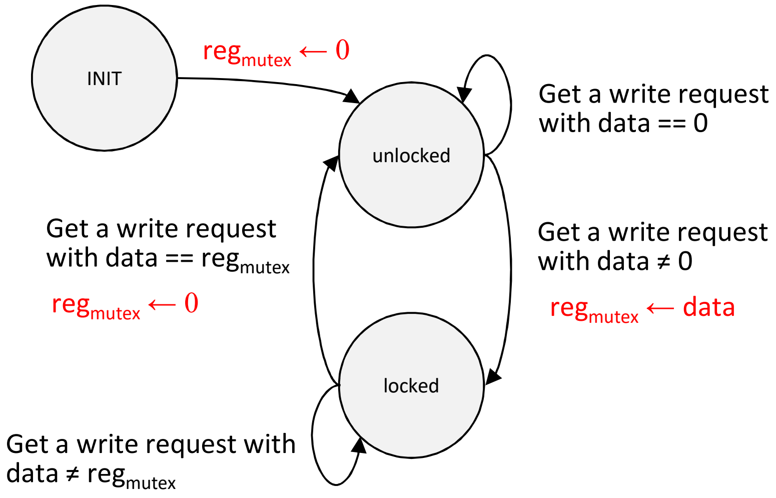
Atomicity cross Heterogeneous Cores

- The atomic instructions defined in ISA does not work across heterogeneous cores
- To guarantee atomic test-and-set, we can either use a coherent bus protocol, or hardware mutexes
- Atomic bus protocols can support
  - Bus locking
  - Multi-cycle read-and-write transactions (RMW cycle)

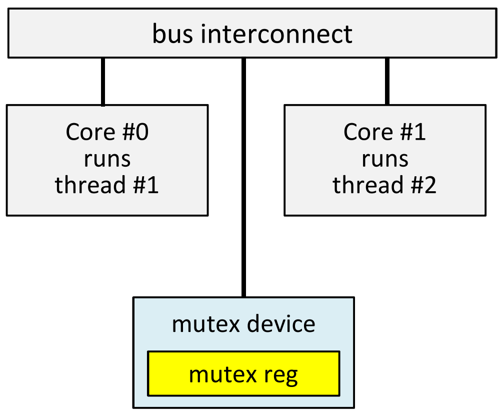
Hardware Mutex

- A HW mutex is a special memory device that "conditionally" accepts write requests
  - Suitable for synchronization between heterogeneous cores
  - Drawback: limited number of mutexes

- Hardware mutex controller:

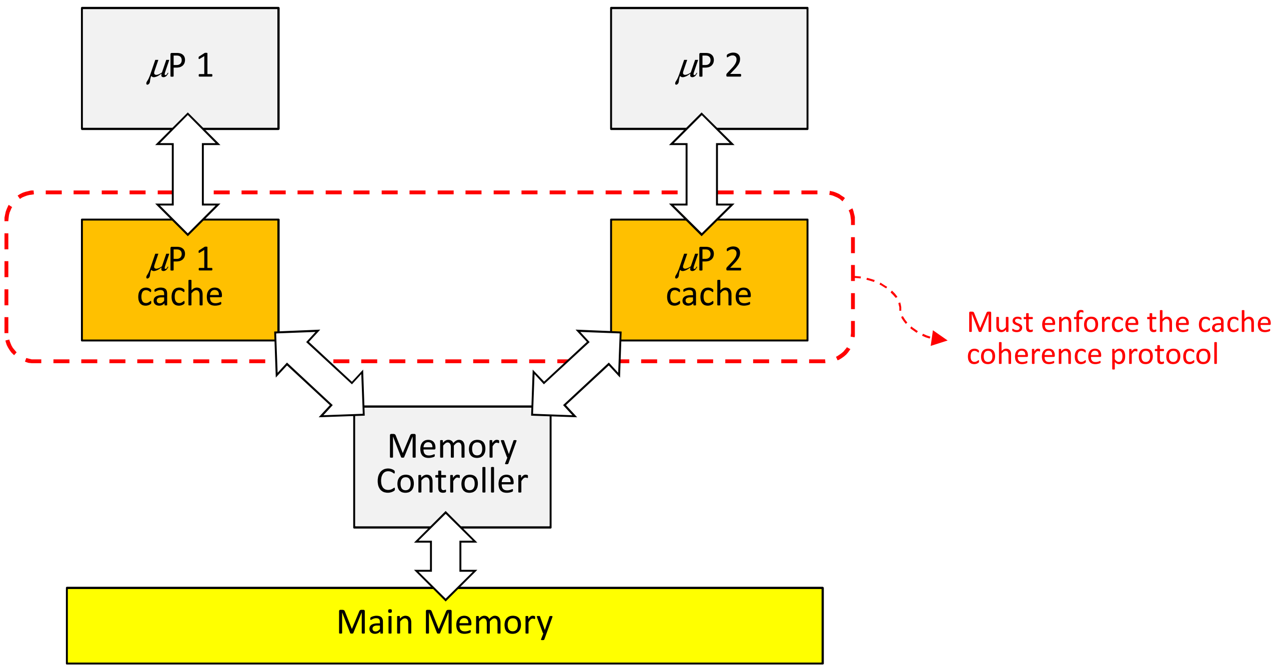


- System diagram:



Cache Coherence Interconnect

- If each processor has its own data cache, there would be data coherence issues when two μPs share data:



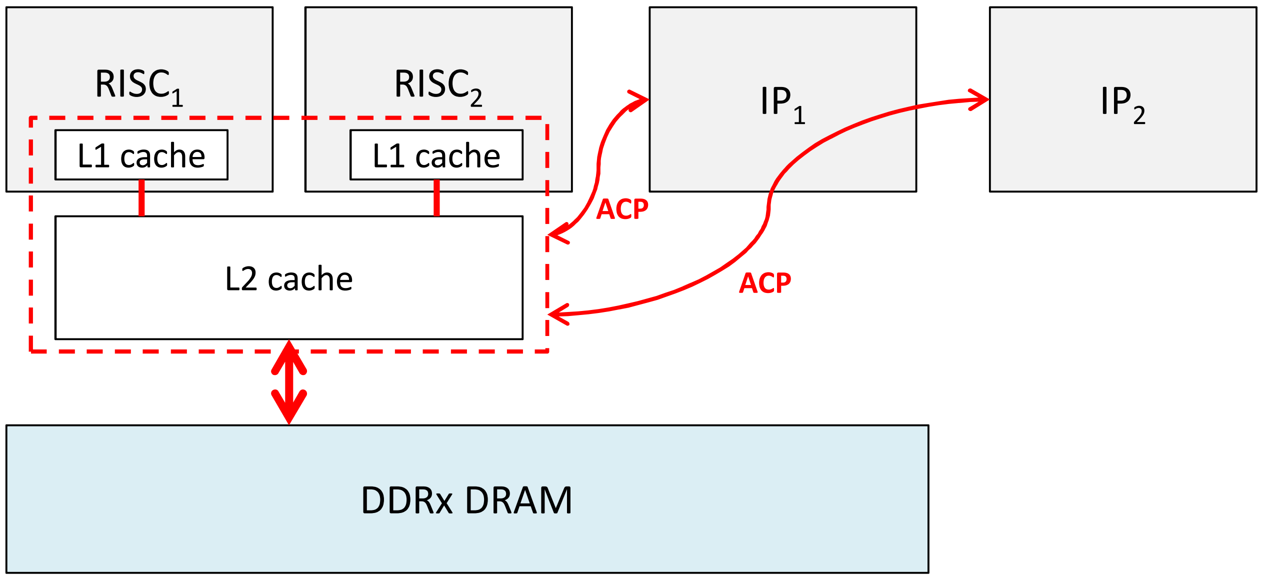
Cache Coherence Protocols

- If only a centralized cache controller is used, there is no cache coherence issue
  - When multiple cache controllers are used, there are two ways to guarantee cache coherence: snooping- and directory-based
- Snooping-based
  - All cache controllers of all processors monitor the bus traffics
  - A controller updates/invalidates its cache lines if they are modified by other controllers
- Directory-based
  - The shared cache line’s state flags is in a common directory
  - A cache controller, when accessing a shared cache line must check the directory to ensure cache coherency

Heterogeneous Sharing of Caches

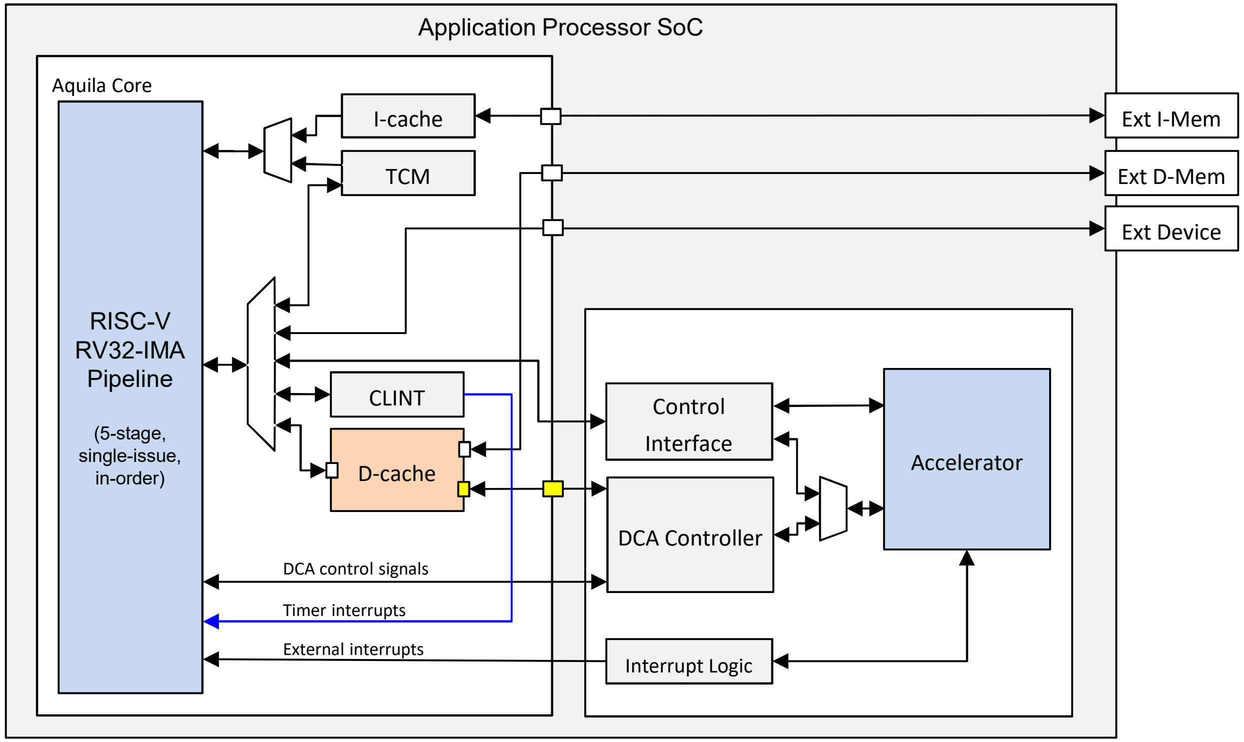
- For heterogeneous cores to shares a cache:
  - General-purposes coherent bus protocols (e.g. AXI ACP)
  - Custom-made Direct-Cache Access controllers (DCA)

- Example: ARM AXI bus has the Accelerator Coherence Port (ACP) for accelerators to share the CPU cache



Direct Cache Access (DCA)

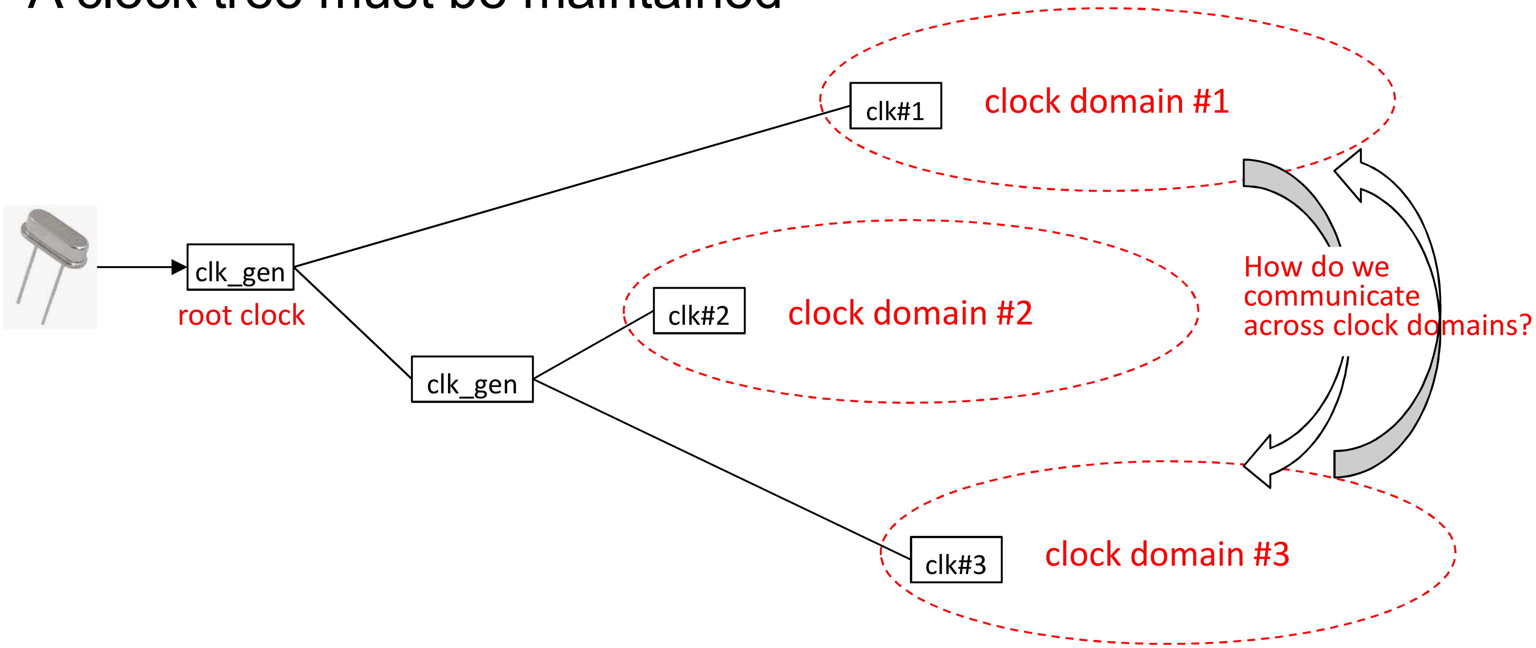
- The DCA concept was proposed by Intel†
  - Simulations show that transferring incoming packets directly into the CPU cache reduces misses by 11%



† R. Huggahalli, R. Iyer, S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O," *Proc. of 32nd Int. Symp. on Computer Architecture (ISCA'05)*, Jun. 4-8, 2005.

Clock Domain Crossing

- A complex SoC usually covers several clock domains
  - A clock tree must be maintained



  - The parameters of each clock must be set properly in the constraint files for the EDA tools to do static timing analysis

Clock Generation

- For FPGAs, there are several ways to generate a clocks based on an oscillator/clock inputs
  - By user logics – not recommended
  - Digital Clock Managers (DCM) – old DLL-style clock synthesizer
  - Phase-Lock Loops (PLL) – precise but less flexible synthesizer
  - Multi-Mode Clock Manager (MMCM) – combines the advantages of DCM and PLL
- Verilog code to invoke MMCM:

```
clk_wiz_0 Clock_Generator(
    .clk_in1(sysclk_i),   // Input clock from the oscillator
    .clk_out1(clk),       // System clock for the Aquila SoC
    .clk_out2(clk_166M), // Clock input to the MIG Memory controller
    .clk_out3(clk_200M)   // DRAM Reference clock for MIG
);
```

Clock Constraints

- If you use MMCM to generate clocks, the constraints for each clock will be automatically generated
  - Don't set constraints yourself, or you get design rule violation

- If you use user-logic to generate clocks, then you must specify the clock parameters in a constraint file:
  - For the clocks from the oscillator (i.e. the root clock):

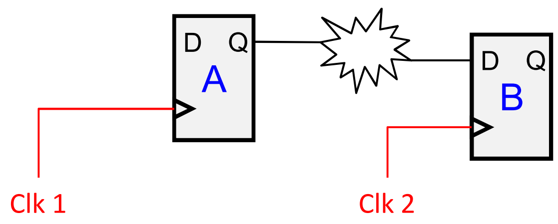
```
create_clock -name sysclk_i -period 10.000 -waveform {0.000 5.000} [get_ports sysclk_i];
```

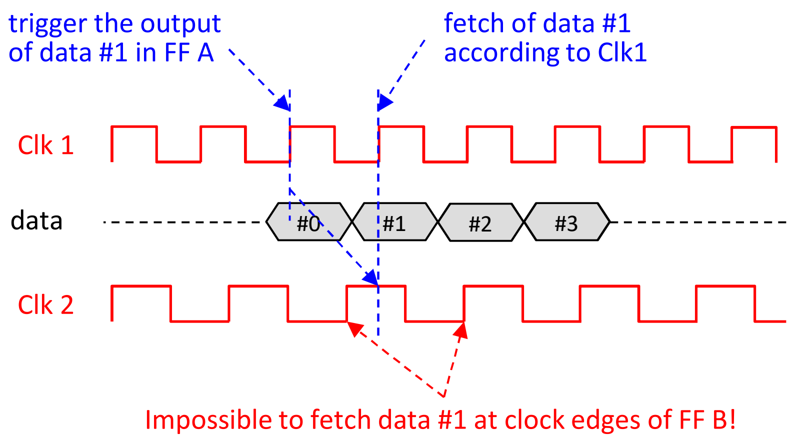
  - For the derived clocks from user logic (clock multiplier/divider)

```
create_generated_clock -period 24.000 -waveform {0.000 12.000} -name clk -source [get_pins usr_clock/clk_o]
```

CDC Issues

- When two flip-flops are driven by different clocks, data exchange can cause problems (a.k.a. meta-stability)

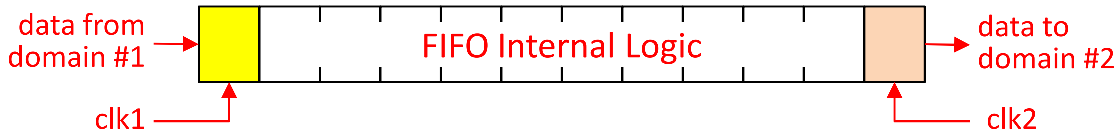




- Some simple solutions can be found in the white paper: https://zipcpu.com/blog/2017/10/20/cdc.html

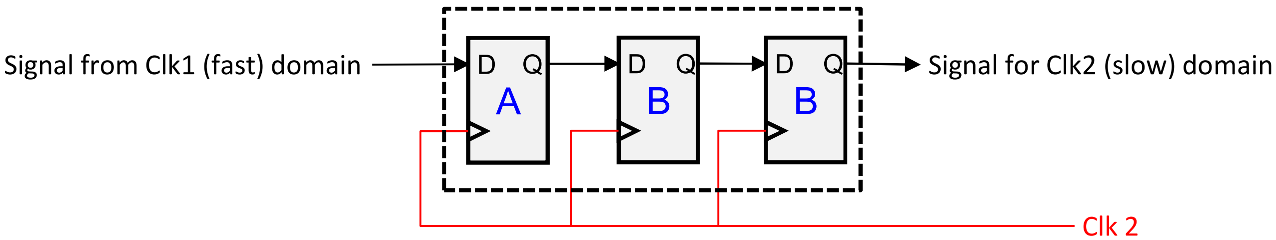
Passing Signals across Domains

- For general signal crossing, use asynchronous FIFOs:
  - BRAM on FPGA has asynchronous dual memory ports
  - The design of head/tail pointers is not trivial†



  - Xilinx has a "fifo_generator" that creates asynchronous FIFOs

- For 1-bit signals, shift registers can be used:



† C. E. Cummings and P. Alfke, "Simulation and Synthesis Techniques for Asynchronous FIFO design," Synopsys Users Group Meetings, San Jose, CA, 2002.

Discussions

- Application Processor design is the key differentiator for smartphone processors
  - Many processor are based on exactly the same ARM core, but they are different processors (e.g., from MediaTek, Qualcomm, or Samsung)
- For best performance, some accelerators must be tightly integrated with the μarch of the processor core
  - Knowing the guts of the μarch is important to design a high performing application processor