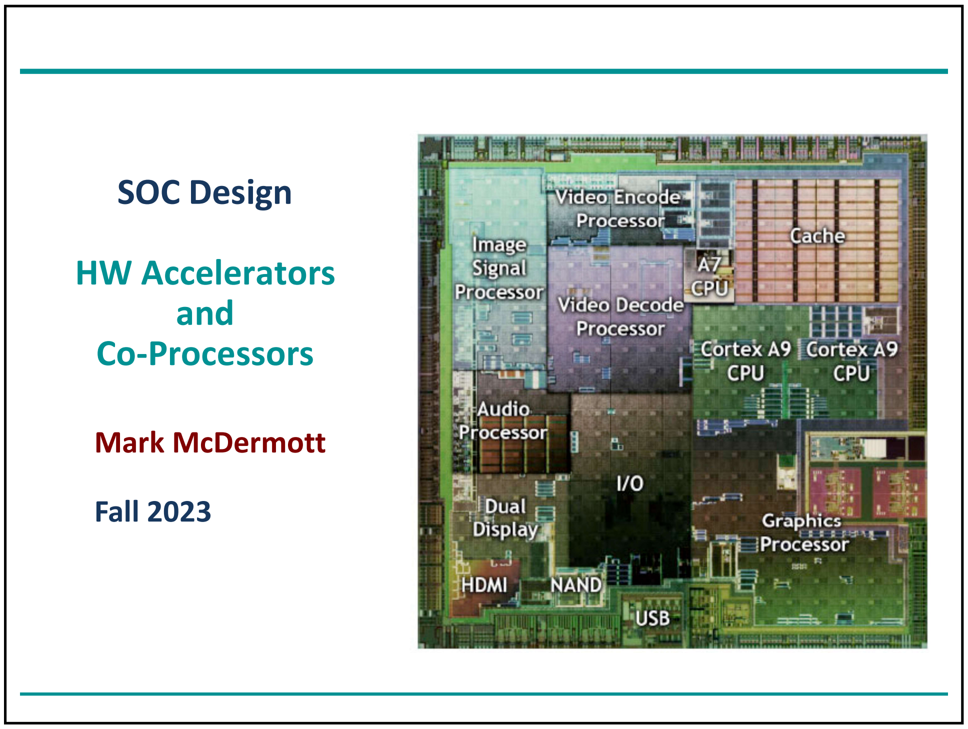
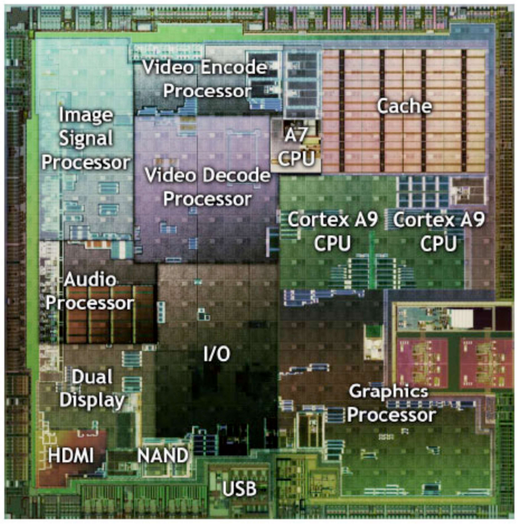


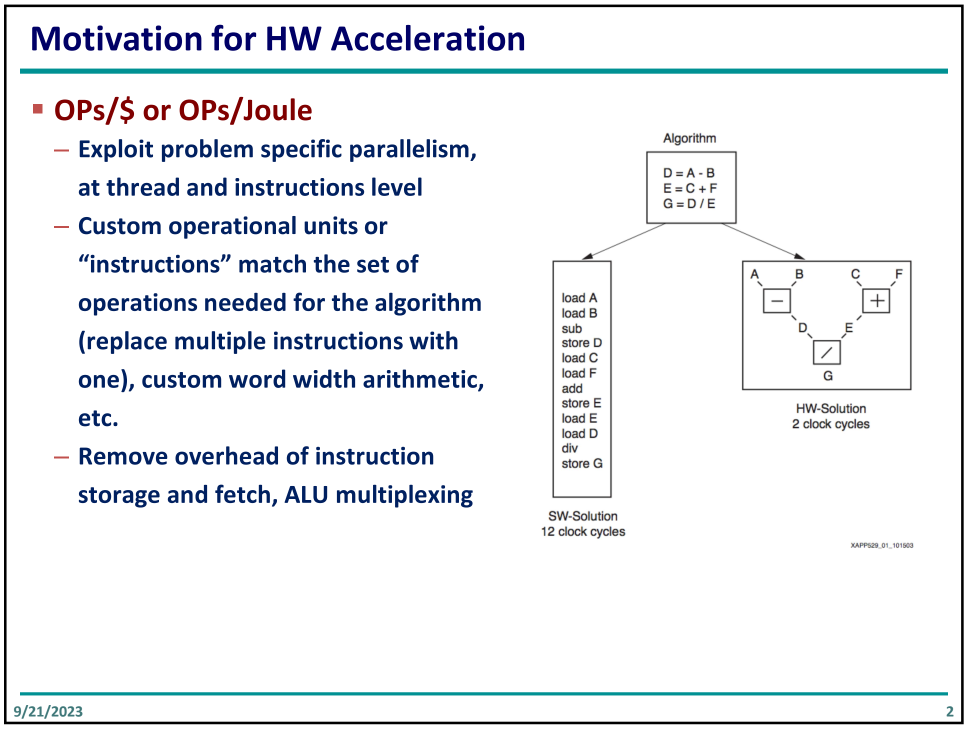
SOC Design

HW Accelerators and Co-Processors

Mark McDermott

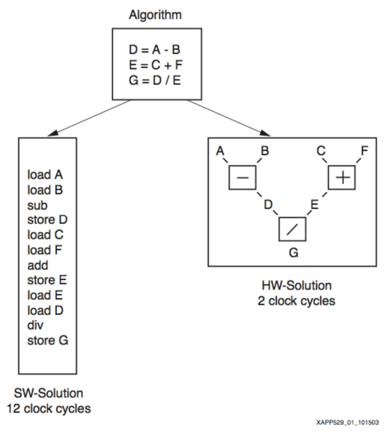
Fall 2023





Motivation for HW Acceleration

- OPs/$ or OPs/Joule
  - Exploit problem specific parallelism, at thread and instructions level
  - Custom operational units or "instructions" match the set of operations needed for the algorithm (replace multiple instructions with one), custom word width arithmetic, etc.
  - Remove overhead of instruction storage and fetch, ALU multiplexing



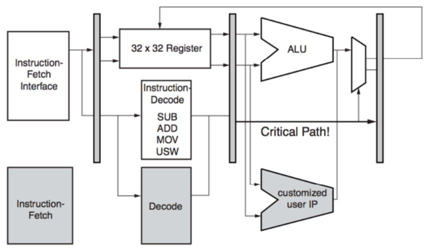
```
D = A - B
E = C + F
G = D / E
```

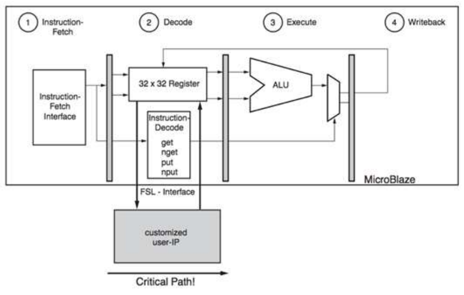
```
load A
load B
sub
store D
load C
load F
add
store E
load E
load D
div
store G
```

Co-Processors

Tightly Coupled Coprocessors

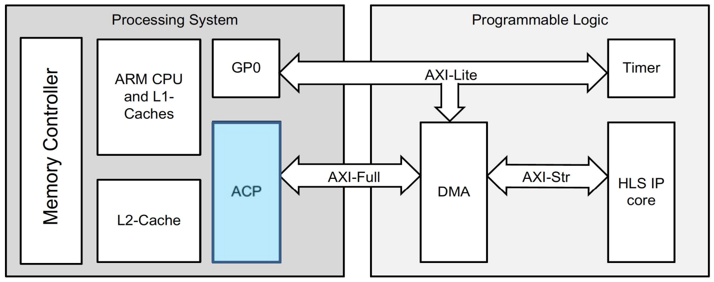
- Integrated with processor control logic
  - Task typically completes in a few cycles
  - Small amounts of data
  - Processor stalls waiting for the coprocessor
  - Communication with coprocessor typically via registers and dedicated control signals





Loosely-Coupled Coprocessors

- **Loosely-Coupled Coprocessors**
  - Used for larger tasks than is the case for tightly-coupled coprocessors
  - Task runs in parallel with main processor
  - May take many cycles per task
  - Large amounts of data that coprocessor may access independent of main processor
  - May or may not use the standard coprocessor interface



https://www.xilinx.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf

Accelerator Coherency Port (ACP)

- Accelerator coherency port (ACP) is a 64-bit AXI slave interface on the SCU that provides an asynchronous cache-coherent access point directly from the PL to the Cortex-A9 MP-Core processor subsystem.

- A range of system PL masters can use this interface to access the caches and the memory subsystem exactly the way the APU processors do to simplify software, increase overall system performance, or improve power consumption.

ACP Usage

- The ACP provides a low latency path between the PS and the accelerators implemented in the PL when compared with a legacy cache flushing and loading scheme. Steps that must take place in an example of a PL-based accelerator are as follows:
  1. The CPU prepares input data for the accelerator within its local cache space.
  2. The CPU sends a message to the accelerator using one of the general purpose AXI master interfaces to the PL.
  3. The accelerator fetches the data through the ACP, processes the data, and returns the result through the ACP.
  4. The accelerator sets a flag by writing to a known location to indicate that the data processing is complete. Status of this flag can be polled by the processor

ACP Caveats

- NOTE: When compared to a tightly-coupled coprocessor, ACP access latencies are relatively long. Therefore, ACP is not recommended for fine-grained instruction level acceleration.

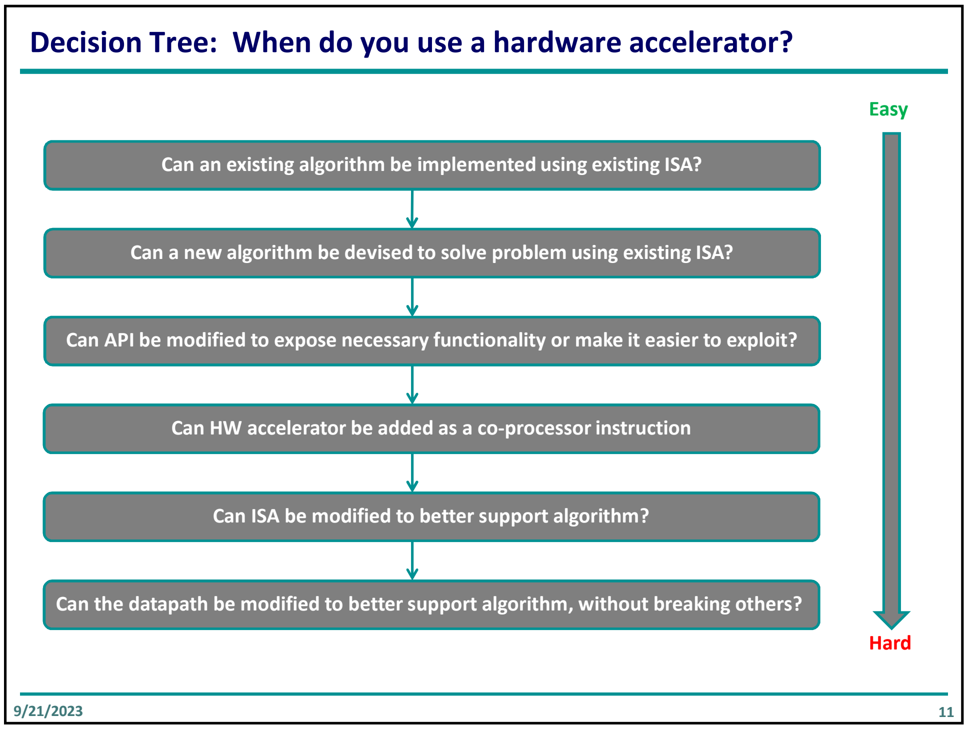
- For coarse-grain acceleration such as video frame-level processing, ACP does not have a clear advantage over traditional memory-mapped PL acceleration because the transaction overhead is small relative to the transaction time and might potentially cause undesirable cache thrashing.

- ACP is therefore optimal for medium-grain acceleration, such as block-level crypto accelerator and video macro-block level processing.

Hardware Acceleration

Common HW Acceleration Applications

- Graphics
- Data Compression/Decompression
- Data Streaming: Audio/Video Encoding/Decoding, Network, I/O
- Image sensing and processing
- Logic Simulation
- Data Encryption: RSA, DES, AES
- FFT, DCT, EXP, LOG, …
- Neuronal Networks
- Neuromorphic



Decision Tree: When do you use a hardware accelerator?

Easy

Can an existing algorithm be implemented using existing ISA?

Can a new algorithm be devised to solve problem using existing ISA?

Can API be modified to expose necessary functionality or make it easier to exploit?

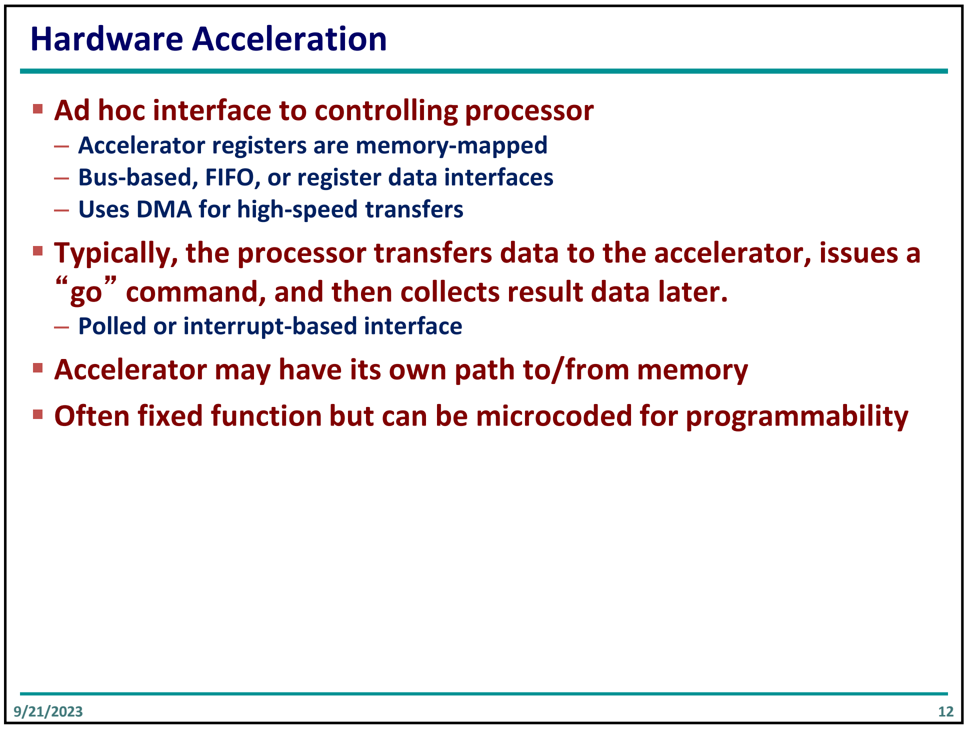
Can HW accelerator be added as a co-processor instruction

Can ISA be modified to better support algorithm?

Can the datapath be modified to better support algorithm, without breaking others?

Hard

9/21/2023 11



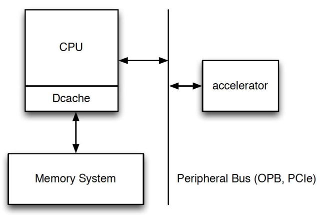
Hardware Acceleration

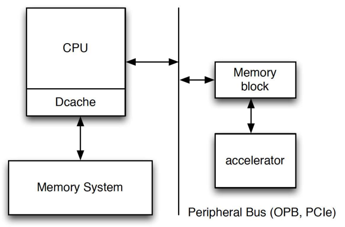
- Ad hoc interface to controlling processor
  - Accelerator registers are memory-mapped
  - Bus-based, FIFO, or register data interfaces
  - Uses DMA for high-speed transfers
- Typically, the processor transfers data to the accelerator, issues a "go" command, and then collects result data later.
  - Polled or interrupt-based interface
- Accelerator may have its own path to/from memory
- Often fixed function but can be microcoded for programmability

9/21/2023 12

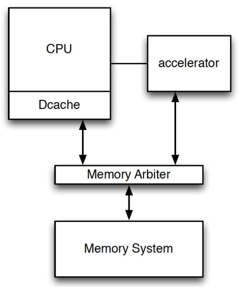
Hardware Accelerator Topologies

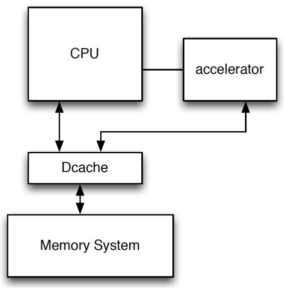
Accelerator appears as a device on a bus





Accelerator is tightly coupled into the processor memory system





Hardware Accelerator Interface: Interrupts or Polling?

- **Polling interfaces usually require the processor to read a memory-mapped register to determine the state of the accelerator.**
  - Can the accelerator accept new input data?
  - Is the accelerator done with its current task?
  - Has the accelerator generated an error condition?

- **Polling interfaces offer minimal latency between the setting of a condition on the accelerator and its discovery by the controlling processor.**
  - But processor isn't doing useful work while it polls…

Hardware Accelerator Interface: Interrupts or Polling?

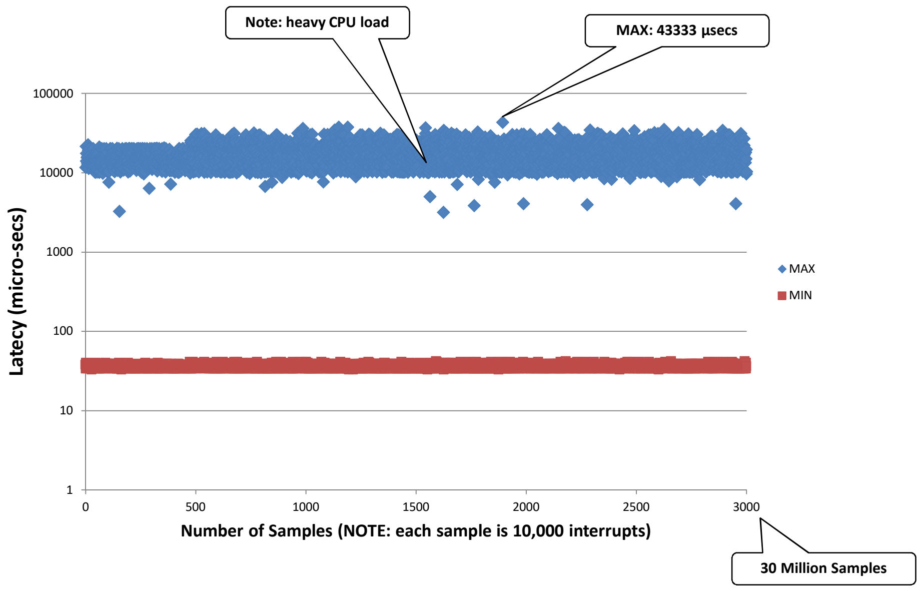
- **Interrupt-based interfaces allow the accelerator to signal conditions to the controlling processor.**
  - Interrupt latency is longer than is achievable via the polling method.
  - But the processor can more easily proceed with other work while the accelerator is busy with a task.

- **Interrupts are more efficient for coarse grained parallelism (i.e., larger tasks with looser and less frequent synchronization requirements)**

- **Interrupts may not work for real-time control tasks with tight schedules**

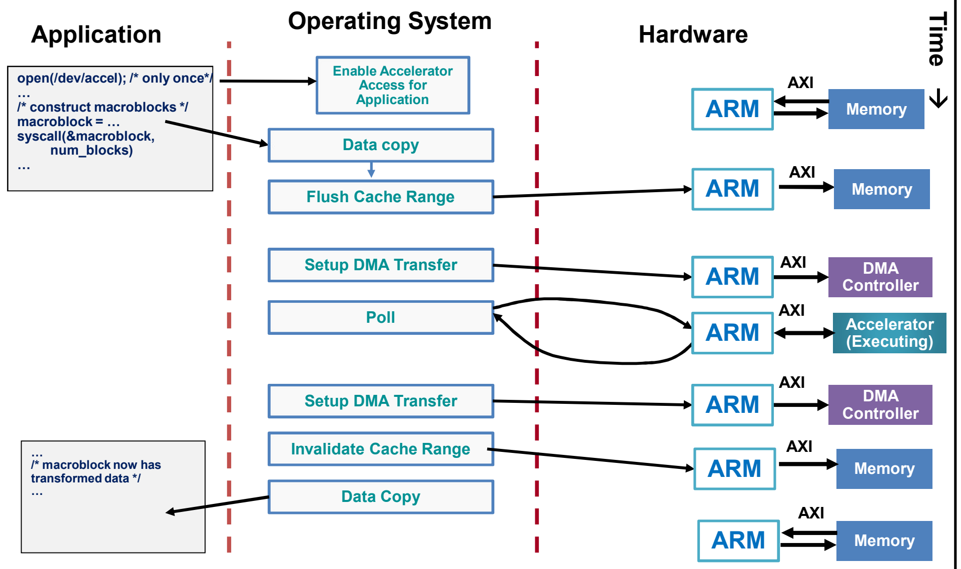
9/21/2023 15

Linux Interrupt latency measurement results



9/21/2023 16

Typical CPU → Accelerator Transaction

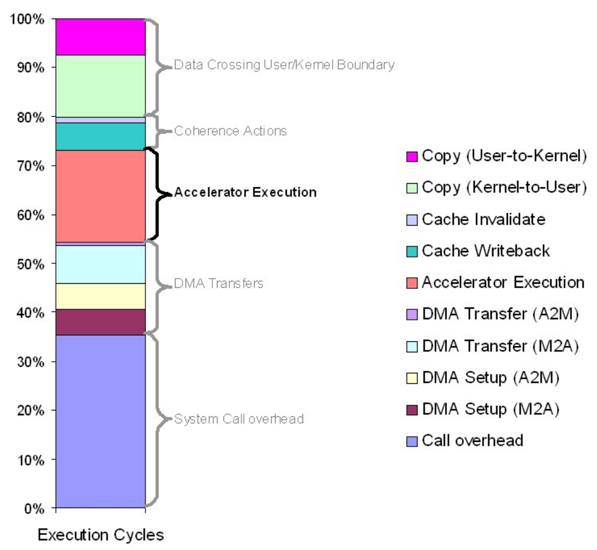


```
open(/dev/accel); /* only once*/
…
/* construct macroblocks */
macroblock = …
syscall(&macroblock,
        num_blocks)
…
```

```
…
/* macroblock now has
transformed data */
…
```

Device Driver Access Cost

System Call Execution Breakdown



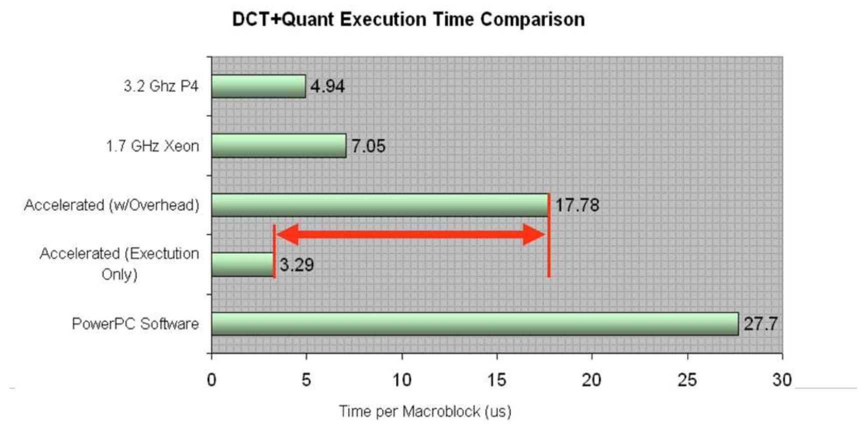
Accelerator Speedup

- Assume loop is executed n times.

```
Speedup = n(t_CPU - t_accel)
              = n(t_CPU - (t_in + t_exec + t_out))
```

$$\text{Speedup} = n(t_{CPU} - t_{accel}) = n(t_{CPU} - (t_{in} + t_{exec} + t_{out}))$$

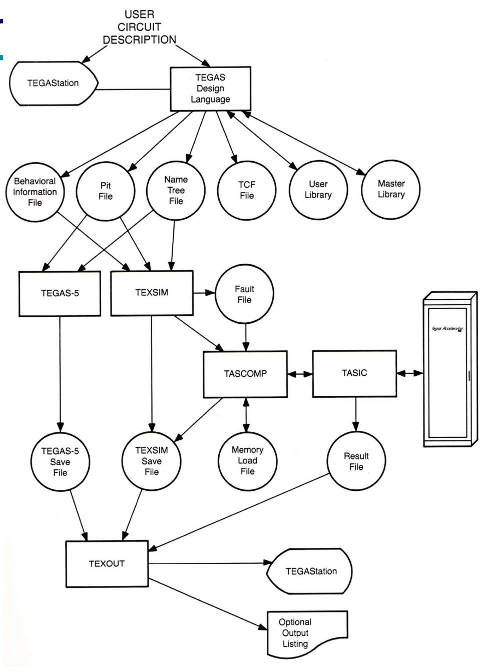
- Compare accelerated system to non-accelerated system:



Page: 19

Logic Simulation Accelerator

- 2 hours to compile 64K gate design
  - No incremental compile
- 75 I/O pins
- 500+ observation points
- 30 minutes to download compiled descriptors to accelerator
- 11 seconds to simulate 2000 µSec of sim-time
- 3-4 hours to unload accelerator data
  - Pins & observation points
- Only marginally faster than SW simulation
  - Amdahl's Law at work….



9/21/2023 20

Single-threaded vs Multi-threading

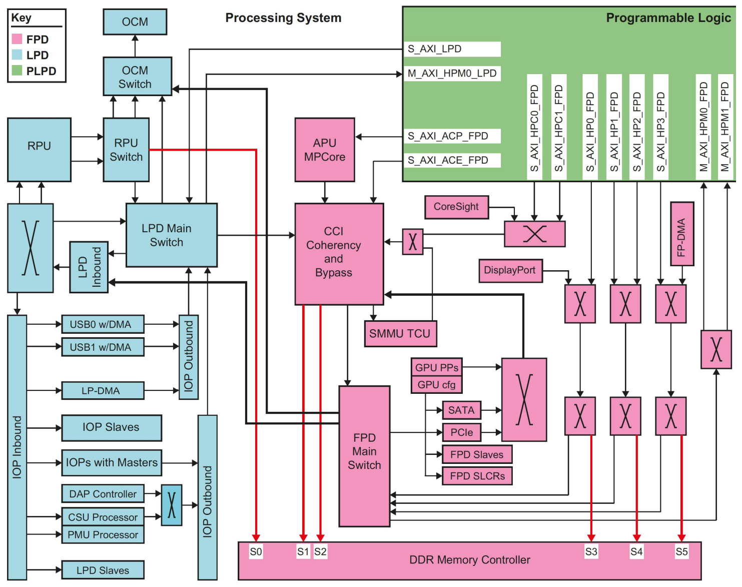
- One critical factor is the available parallelism in the application:
  - Single-threaded/blocking: CPU waits for accelerator;
  - Multithreaded/non-blocking: CPU continues to execute along with accelerator.
- For multithread, CPU must have some useful work to do while accelerators perform some tasks.
  - Software environment must also support multi- threading.

Caching Issues with Accelerators

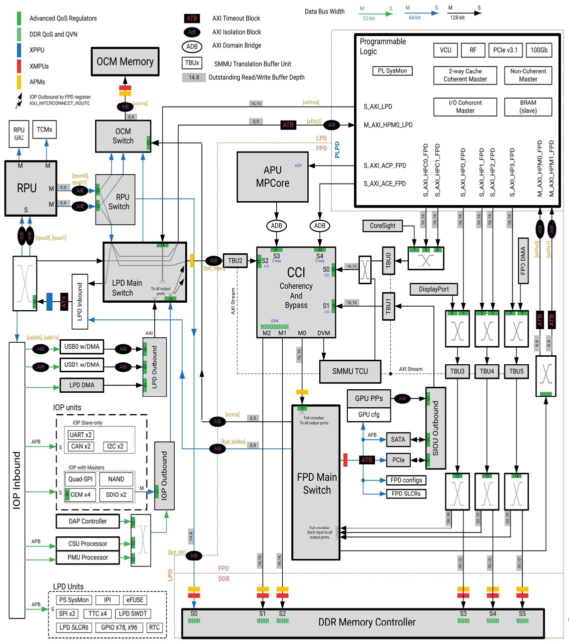
- Main memory provides the primary data transfer mechanism to the accelerator.
- Programs must ensure that caching does not invalidate main memory data.
  - CPU reads location S.
  - Accelerator writes location S.
  - CPU writes location S.
    - Program will not see proper value of S stored in the cache
- Many CPU buses implement test-and-set atomic operations that the accelerator can use to implement a semaphore. This can serve as a highly efficient means of synchronizing inter-process Communications (IPC)

Configuring an Accelerator in the Ultra96 FPGA

PS-PL Interconnect



Detailed PS Bussing Block Diagram

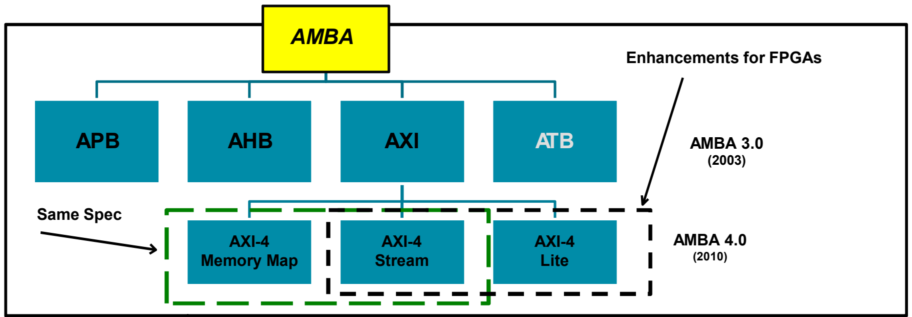


XILINX ➤ ALL PROGRAMMABLE.

PS-PL Interconnect

Interface Name	Description	Master	Slave
S_AXI_ACP_FPD	The Accelerator Coherency Port (ACP) provides a coherent path between the PL and the APU's Level 2 cache.	PL	PS FPD
S_AXI_ACE_FPD	The AXI Coherency Extension (ACE) can access system memory and the local memory of the APU, via the Cache Coherent Interconnect (CCI); sharing up-to-date information.	PL	PS FPD
S_AXI_HPC0_FPD	Each High-Performance Coherent (HPC) port is directly connected to the CCI and System Memory Management Unit (SMMU).	PL	PS FPD
S_AXI_HPC1_FPD		PL	PS FPD
S_AXI_HP0_FPD	These High-Performance (HP) ports pass through the SMMU and are connected to the interconnect's central switch in the FPD. They are connected to three dedicated ports on the DDR controller. However, sharing exists with the DisplayPort and Full-Power DMA (FP-DMA).	PL	PS FPD
S_AXI_HP1_FPD		PL	PS FPD
S_AXI_HP2_FPD		PL	PS FPD
S_AXI_HP3_FPD		PL	PS FPD
M_AXI_HPM0_FPD	High-Performance port from the FPD to the PL.	PS FPD	PL
M_AXI_HPM1_FPD		PS FPD	PL
S_AXI_LPD	A high-performance path from the PL to the LPD. This port can access the RPU when the FPD is powered down.	PL	PS LPD
M_AXI_HPM0_LPD	Low-latency high-performance port that interfaces the LPD to the PL.	PS LPD	PL

AMBA



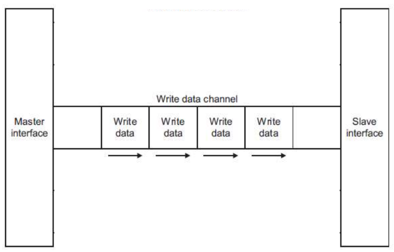
Interface	Features
Memory Map / Full (AXI4)	Traditional Address/Data Burst (single address, multiple data)
Streaming (AXI4-Stream)	Data-Only, Burst
Lite (AXI4-Lite)	Traditional Address/Data—No Burst (single address, single data)

XILINX ALL PROGRAMMABLE.

The AXI Interface—AXI4-Stream

- No address channel, no read and write, always just master to slave
- Effectively an AXI4 “write data” channel
- Unlimited burst length
  - AXI4 max 256
  - Note: AXI4-Lite does not burst
- Virtually same signaling as AXI Data Channels
- Protocol allows merging, packing, width conversion
- Supports sparse, continuous, aligned, unaligned streams

AXI4-Stream Transfer



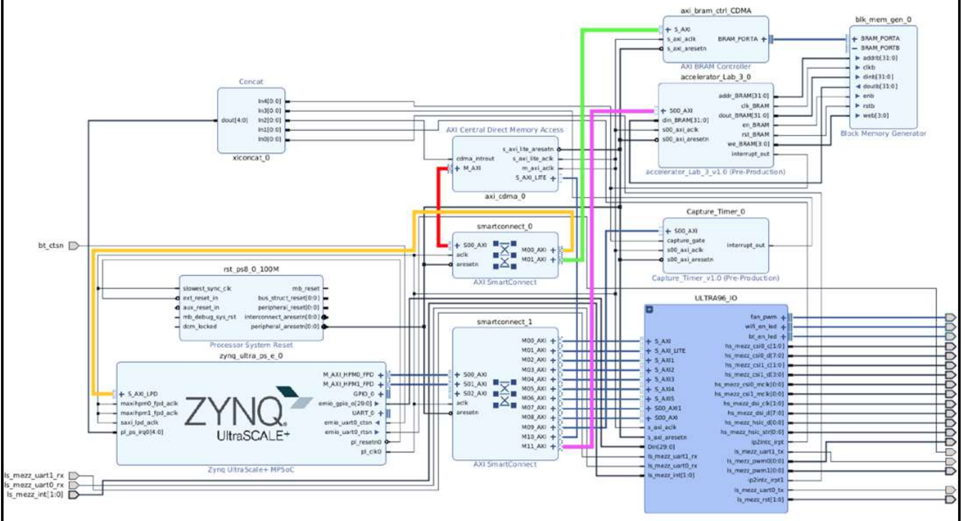
XILINX ALL PROGRAMMABLE.

DMA Engine Overview

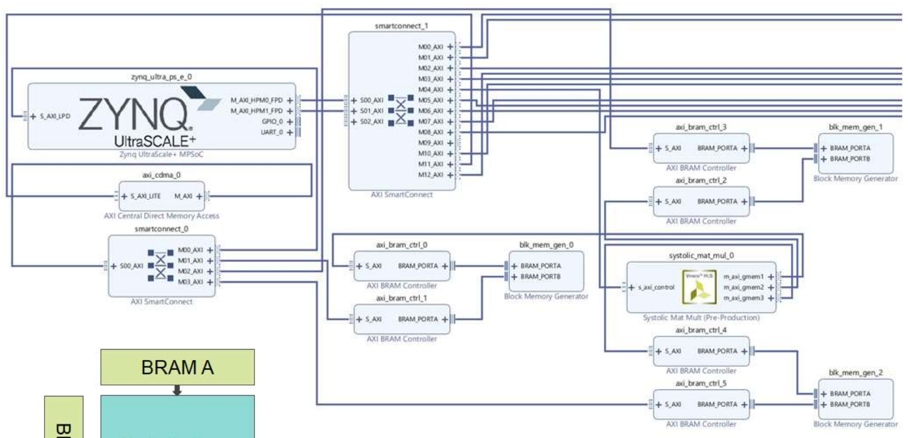
- DMA controller (DMAC) uses a 64-bit AXI provider interface operating at the CPU_2x clock rate to perform DMA data transfers to/from system memories and PL peripherals.
- Transfers are controlled by the DMA instruction execution engine. The DMA engine runs on a small instruction set that provides a flexible method of specifying DMA transfers.
  - This method provides greater flexibility than the capabilities of DMA controller methods.
- Program code for the DMA engine is written by software into a region of system memory that is accessed by the controller using its AXI provider interface.
  - DMA engine instruction set includes instructions for DMA transfers and management instructions to control the system.

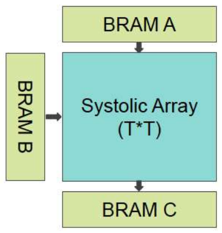
https://www.fpgadeveloper.com/2014/08/using-the-axi-dma-in-vivado.html/

Basic Dual Ported w/CDMA



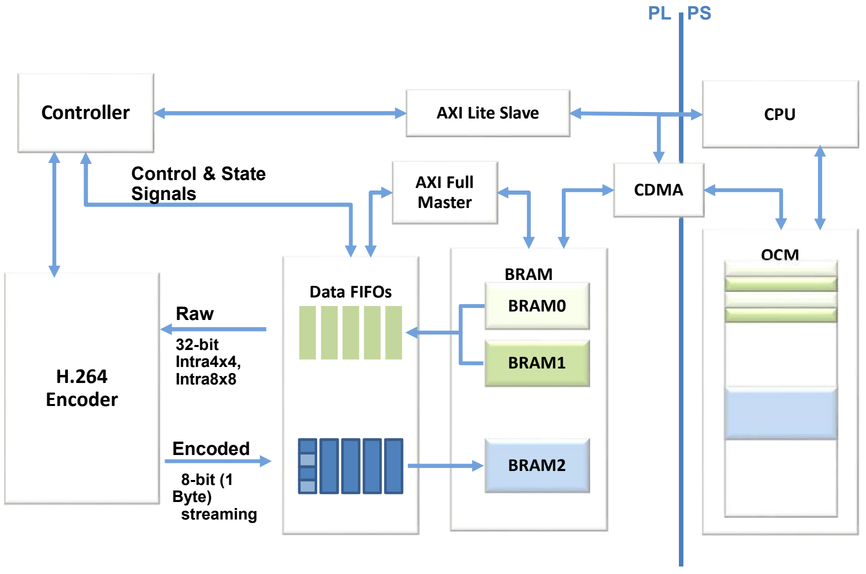
Multiple BRAM blocks





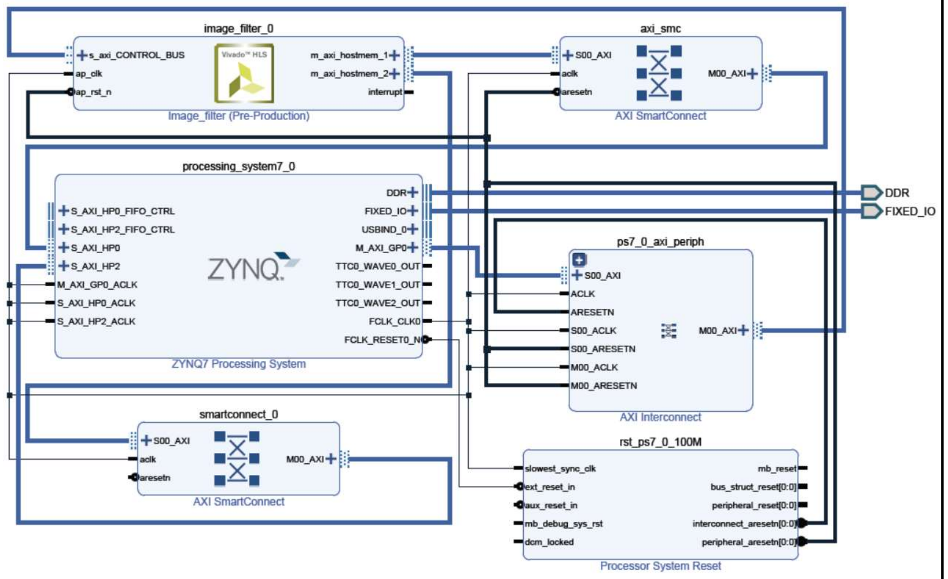
9/21/2023 31

FIFOs for buffering

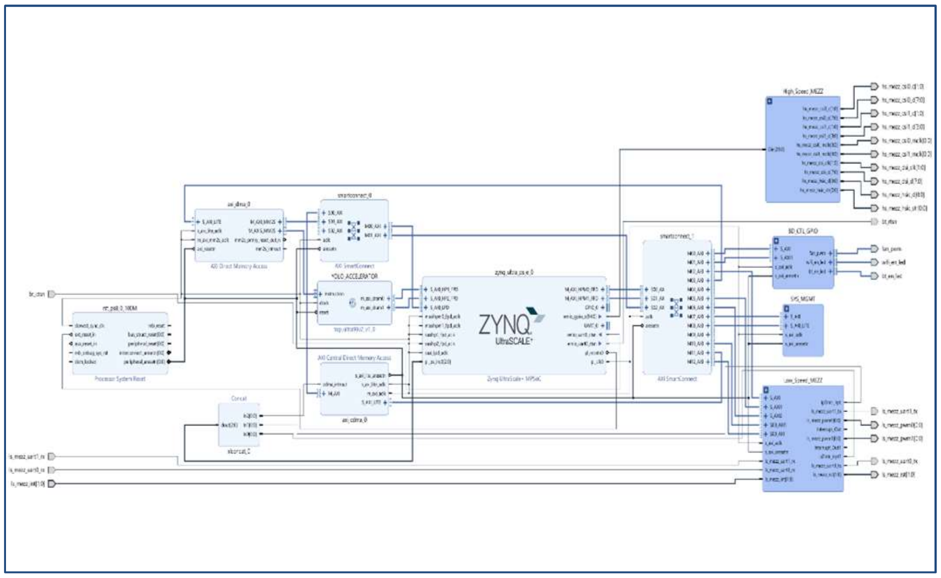


9/21/2023 32

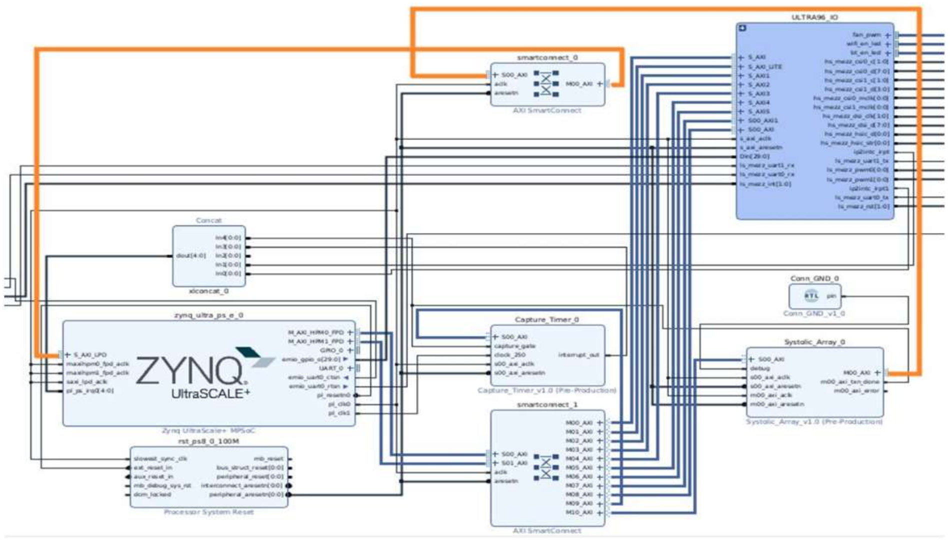
Master Connection



Master connection to DRAM



AXI Master Connection to OCM



Design flow suggestions

- **Baseline the software**
  - Use WC datasets to profile the time spent in each function.
  - Do the results make sense? If not, fix your profiling techniques…
  - Generate a matrix of candidates that could be accelerated.
    - Rank order, based on factors that are critical in your system.
- **Determine "Speeds & Feeds" in your system**
  - I/O, memory, disk, CPU, algorithmic, busses, etc.?
  - Which ones can you do anything about?
  - Is latency an issue or is it throughput or both?
- **Finalize data representations**
  - Is fixed point sufficient? How many bits?
  - What about conversion overhead between Application and Accelerator
- **Build a debug "nest" for the accelerator hardware.**
  - Build test pattern generation and stimulus system for block level debug
  - Build controllability and observability hooks into your design.
    - Doesn't come for free.

Design Suggestions

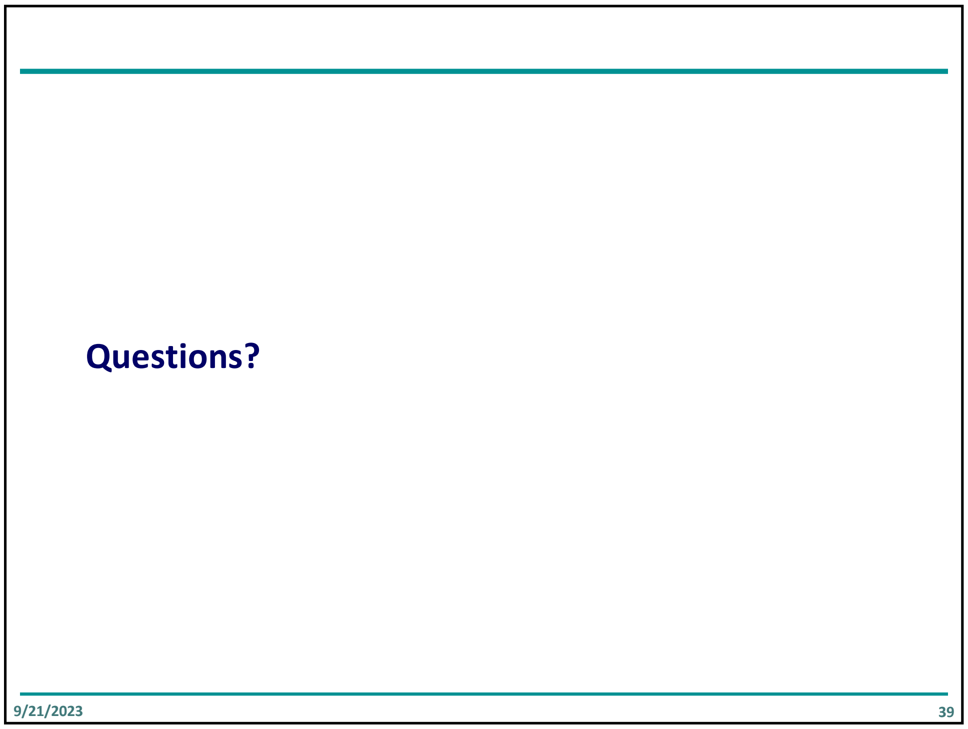
- Crawl before you run.
  - Build, test and profile blocks using your new debug nest.
    - Don't try to debug the top level first.
  - Use data streams from the software to confirm functionality and performance (if possible)
- Remember the following debug wisdom:
  - Disprove all theories.
  - Firmware is not always wrong, but it is mostly wrong
  - Hardware is not always perfect either.
    - Crosstalk in FPGAs can cause weird problems
  - If you must use someone else's IP, then make sure you get the verification IP too.
- Remember the following about using design tools:
  - GIGO – garbage in garbage out.
  - Synthesis is a constraint design system.
  - Spend time writing good constraints –> spend less time debugging.

9/21/2023 37

Design Suggestions

- Register bits make great controllability nodes
  - "tie" nodes to high or low when debugging.
  - recompiling is not free.
- Register bits make great observability nodes
- Use at least one preprogrammed register as a "Canary in the coal mine".
- There are a number of free widgets in the Xilinx IP Repo that can help with your design or debugging.
  - Test them in a separate nest to prove that they are doing what they advertise.
    - Then integrate the widget into your design.

9/21/2023 38



Questions?

9/21/2023

39