arm

Accelerate Image Processing Using FPGA Hardware

© 2021 Arm

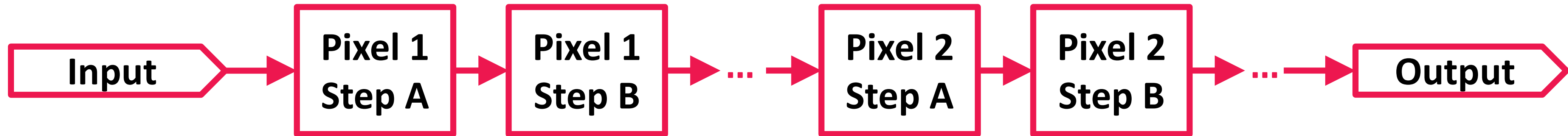
Learning Outcomes

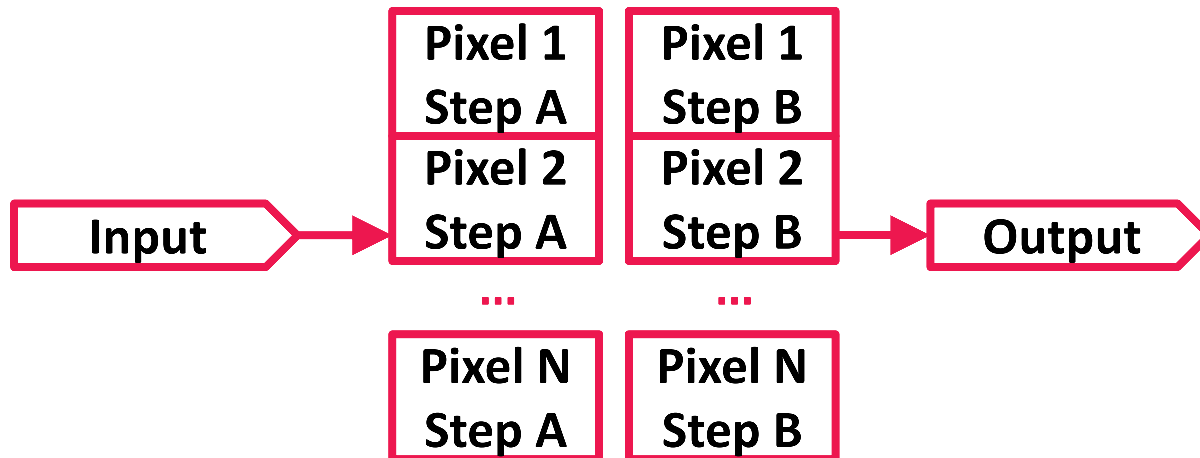
At the end of this module, you will be able to:

- Explain the purpose of hardware acceleration and give examples of its implementation.
- Give examples on how to modularize hardware to achieve hardware acceleration.
- Compare and contrast the advantages and disadvantages of hardware acceleration vs using Neon technology.

© 2021 Arm

Parallel Image Processing





- SIMD processor
- DSP
- GPU
- Customized logic

© 2021 Arm
arm

Hardware Acceleration

- Hardware acceleration is the use of hardware to perform some functions instead of running on a general-purpose CPU.
- The hardware is designed for specific functions to increase performance and/or reduce resource consumption.
- It is a tradeoff between flexibility and efficiency.
- Examples of hardware acceleration:
  - GPUs
  - Fixed-function implemented on FPGAs
  - Fixed-function implemented on ASICs

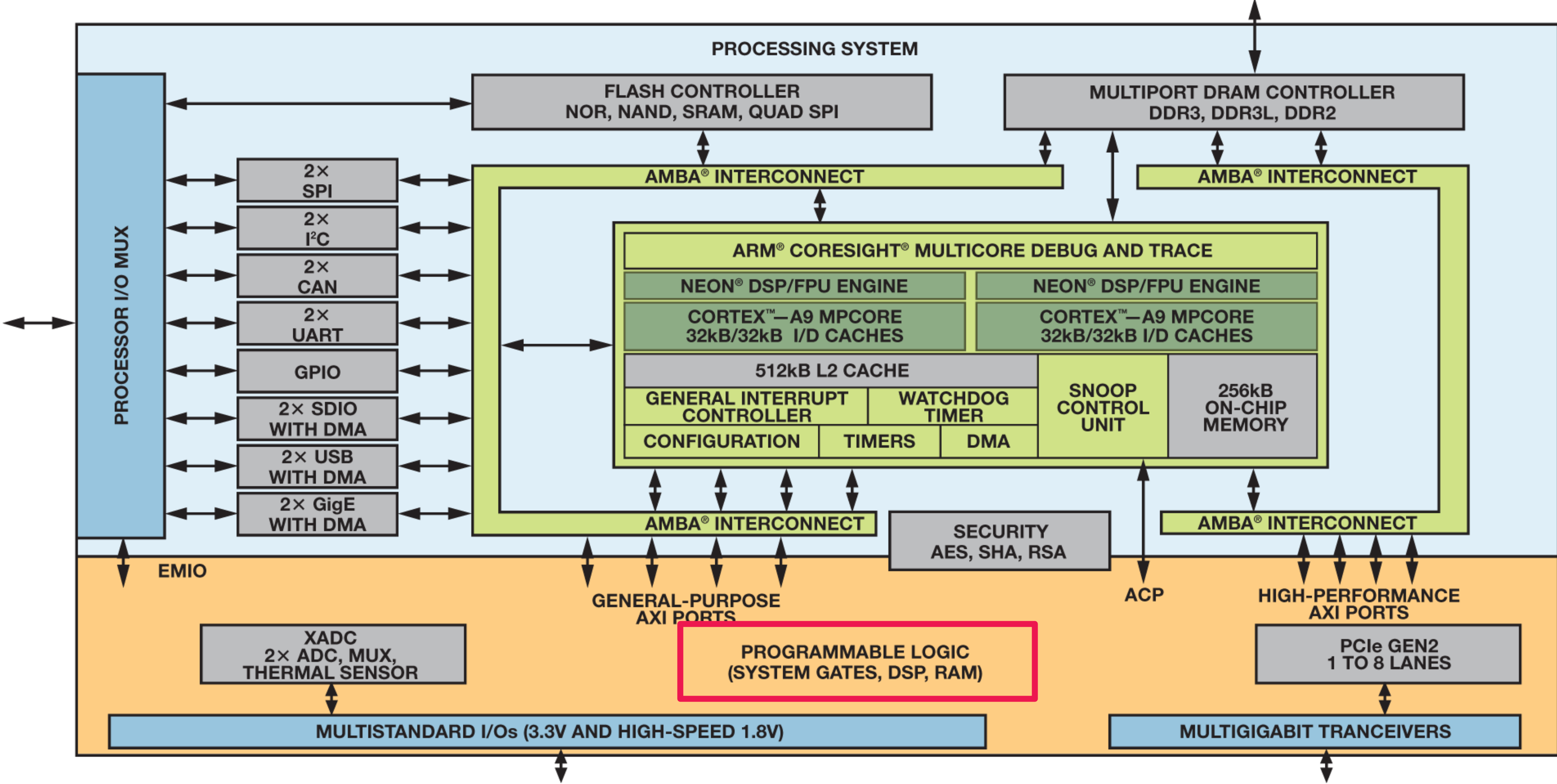
© 2021 Arm

Example of Hardware Acceleration

- Graphics Processing Unit (GPU) is one of the well-known examples of hardware acceleration.
- The name GPU was popularized by Nvidia in 1999.
  Before that, some implementations of graphics display accelerator had been used in PCs and game consoles.
- Nowadays, GPUs become usually one of the top 2 computing modules in computers and mobile devices to perform 2D and 3D graphics. Considering the parallel computing ability of GPUs, they are even used to do general-purpose computing.

© 2021 Arm

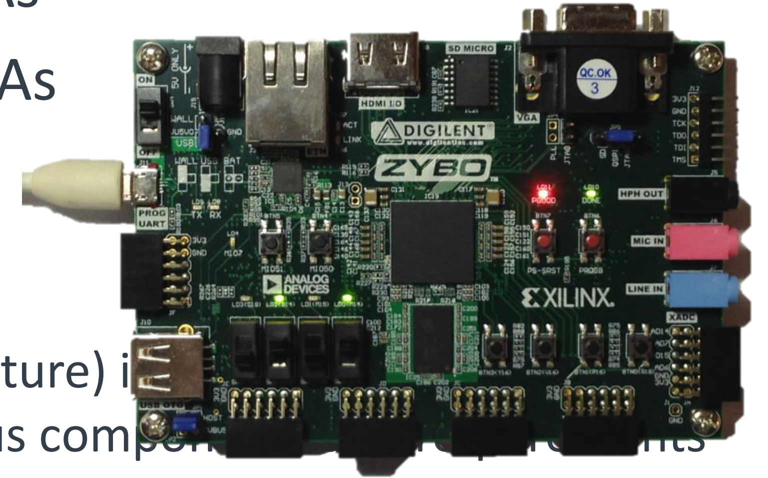
Zynq-7000 Platform



© 2021 Arm
arm

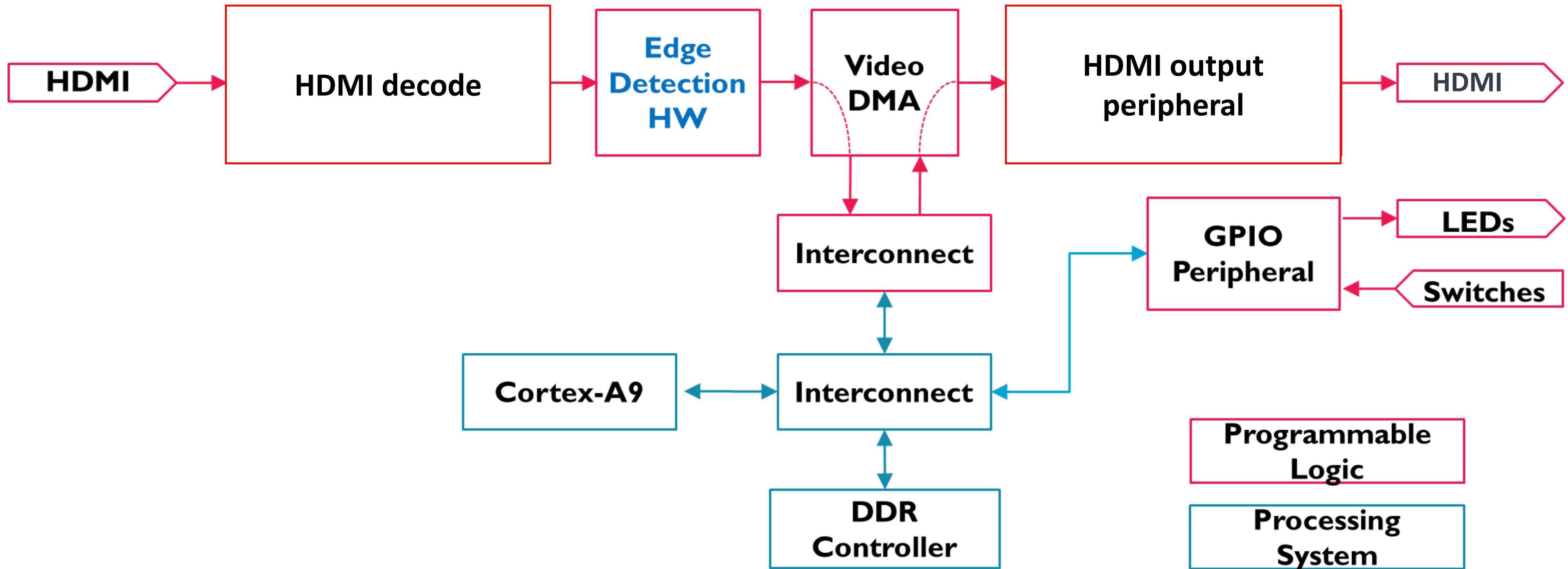
FPGA on Zynq platforms

- Zynq specifically integrates Arm Cortex CPUs and Xilinx FPGAs
- Need of description and connection between CPUs and FPGAs
  - AXI protocol is one of the standard protocols for bus connections
  - Increase the re-usability and lead to low development cost
- AXI4 protocol (Advanced eXtensible Interface 4)
  - 4th generation of Arm AMBA (Advanced Microcontroller Bus Architecture) i
  - Provide high-bandwidth, low-latency, highly flexible design for various comp
  - For re-usability and scalability



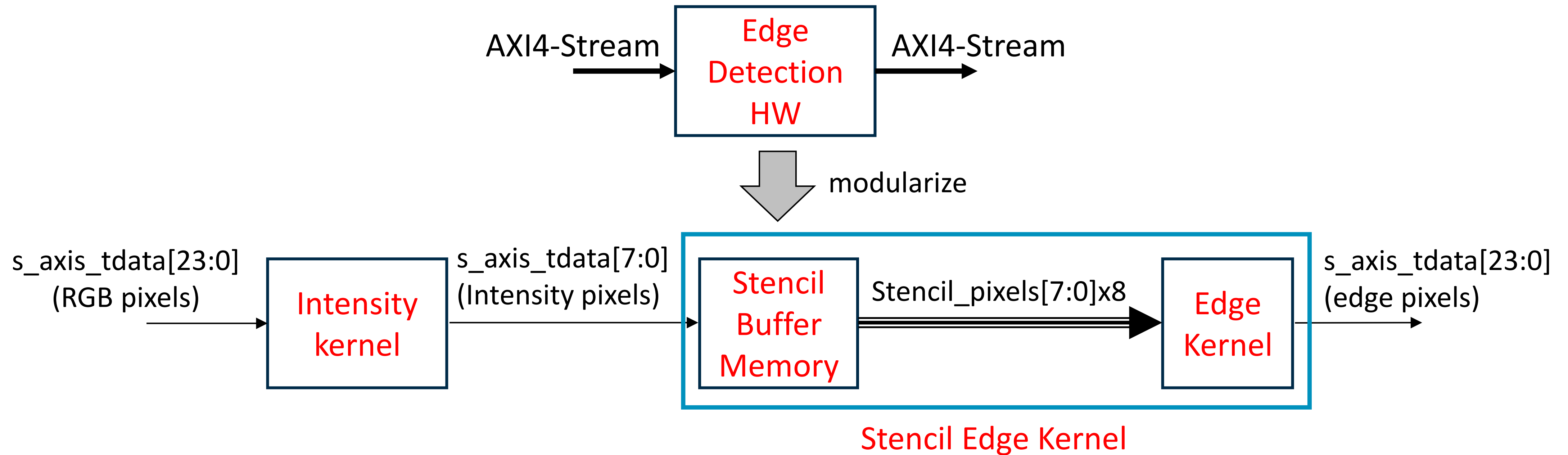
© 2021 Arm
arm

Offload Image-Processing Algorithm from CPU to FPGA



© 2021 Arm
arm

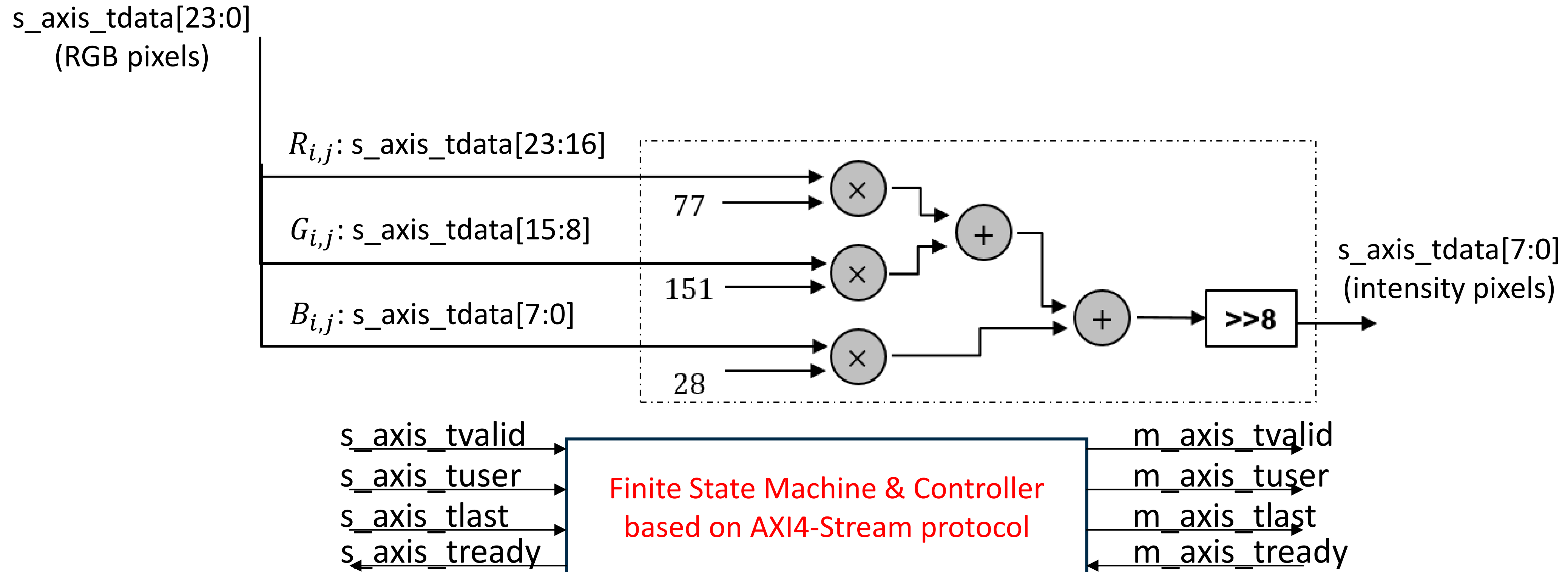
Modularize Hardware



- Video Stream Processing
  - AXI4 Stream protocol would be suitable
  - Using a standard protocol leads to design re-usability
- We split "Edge Detection HW" into two(or three) modules
  - Intensity kernel (computes intensity of each pixel, or converts RGB pixels to grey-scale ones)
  - Stencil Edge Kernel (computes edge values of specific pixels)
    - Stencil Buffer Memory (buffers intensity of each pixel)
    - Edge Kernel (actual kernel for edge values)

© 2021 Arm

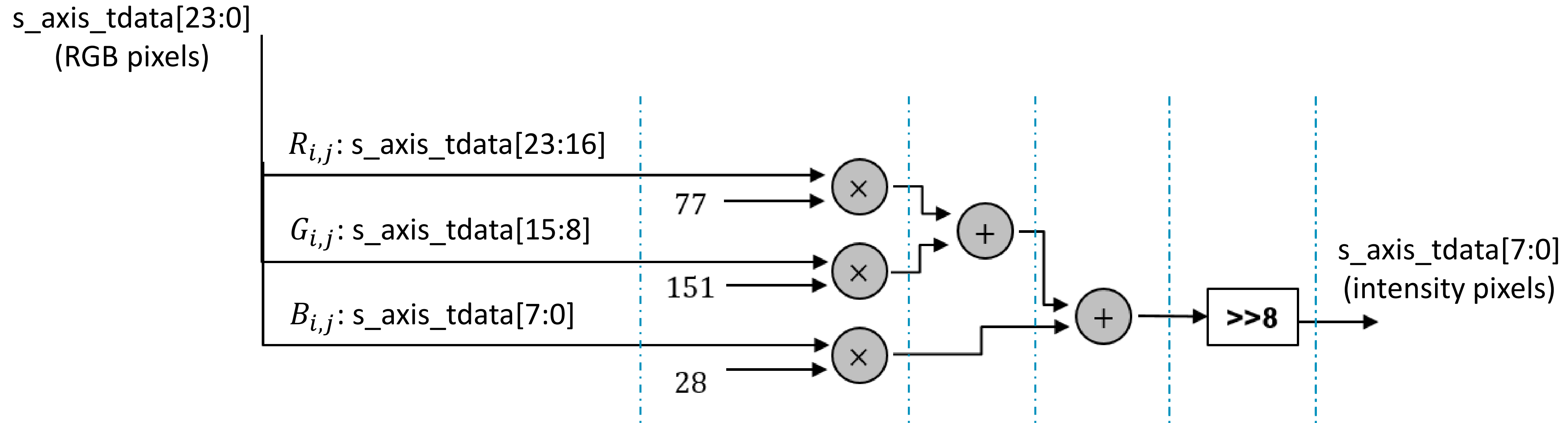
1st Module: Intensity kernel



- Pipeline structure for intensity computation
  - In our design, we get an intensity pixel per cycle after 5cycle latency
- Separate data path and control path

© 2021 Arm
arm

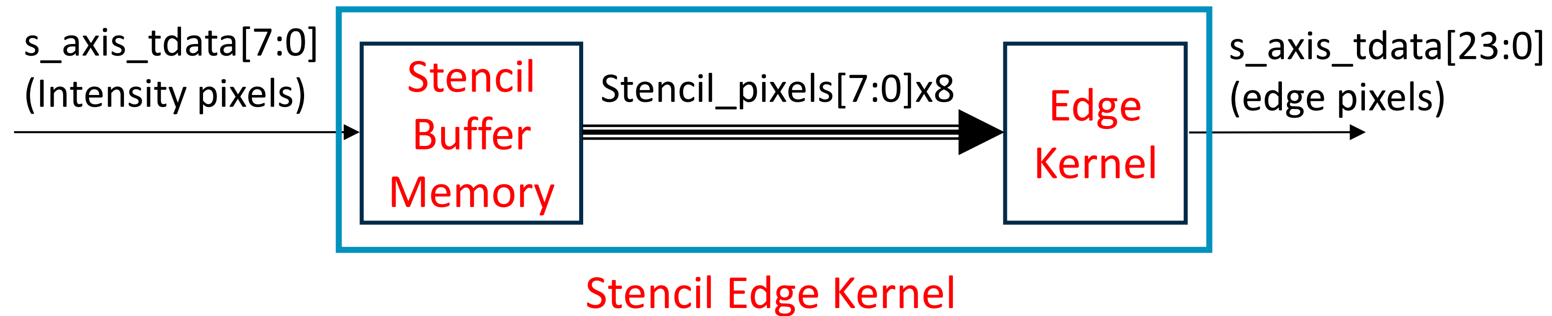
1st Module: Intensity kernel - Pipeline



- In our design, we put pipeline register in the above dotted line
  - 5 pipeline registers lead to 5 cycle latency
  - # of pipeline registers could be reduced in accordance with clock frequency

© 2021 Arm
arm

2nd Module: Stencil Edge Kernel

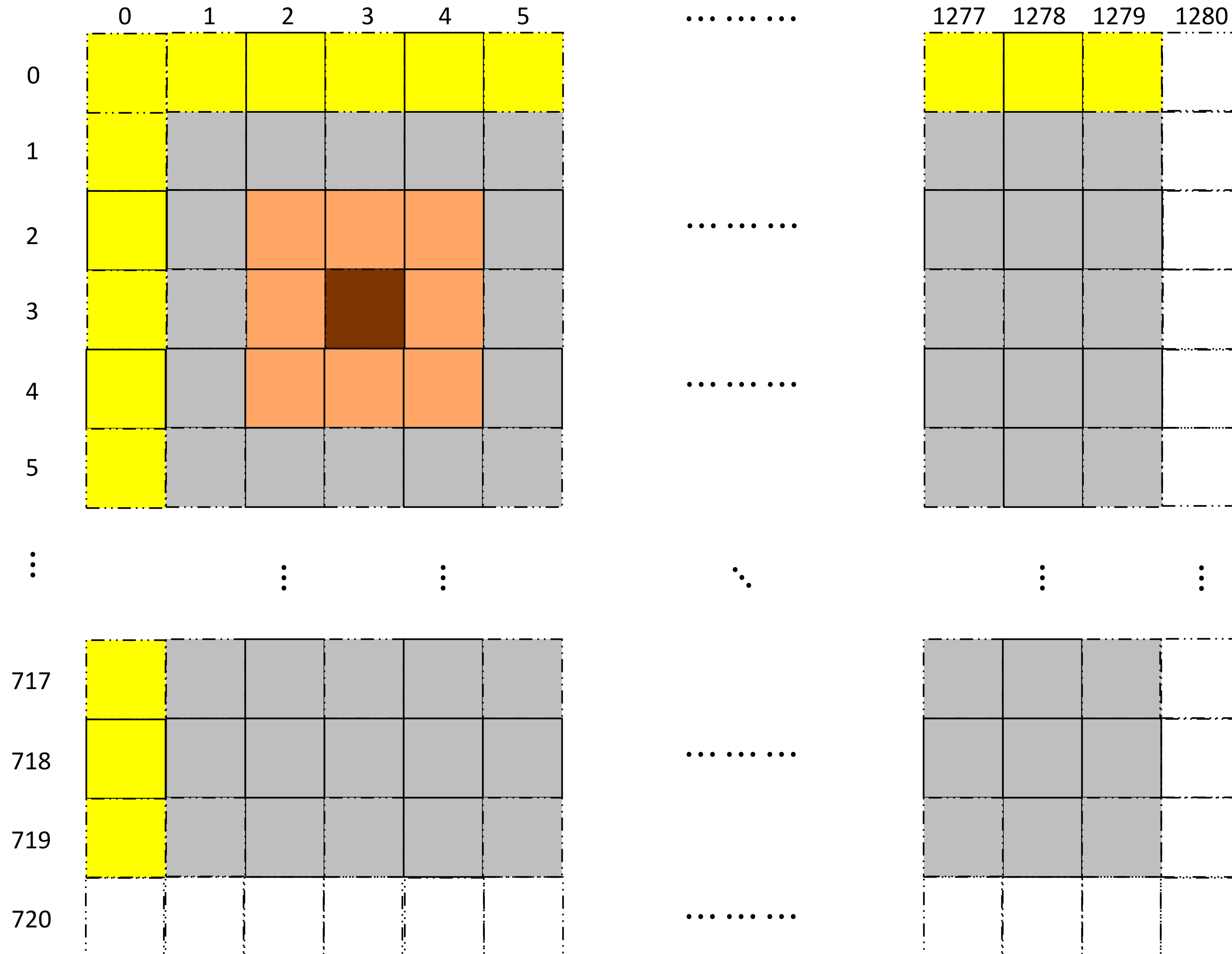


- Split "Stencil Edge Kernel" Module into two smaller modules to separate functions (buffering & computation)
- Stencil Buffer Memory is useful for Stencil Kernel i.e. Stencil computation in stream processing

© 2021 Arm

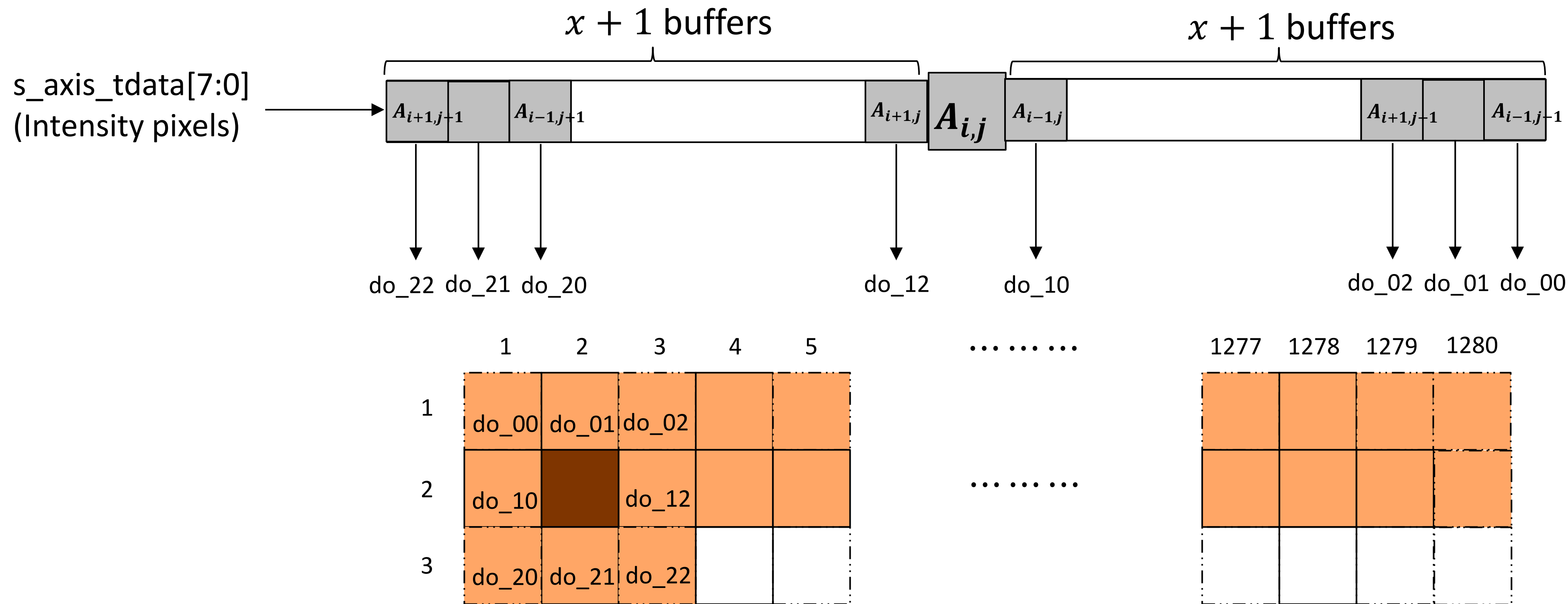
Stencil Computing

Computation involving 8 neighboring pixels is called "Stencil Computation"



© 2021 Arm
arm

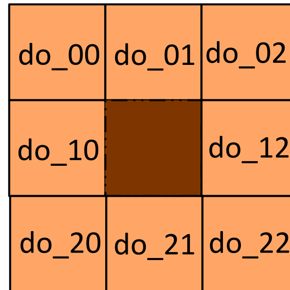
2-1st Module: Stencil Buffer Memory

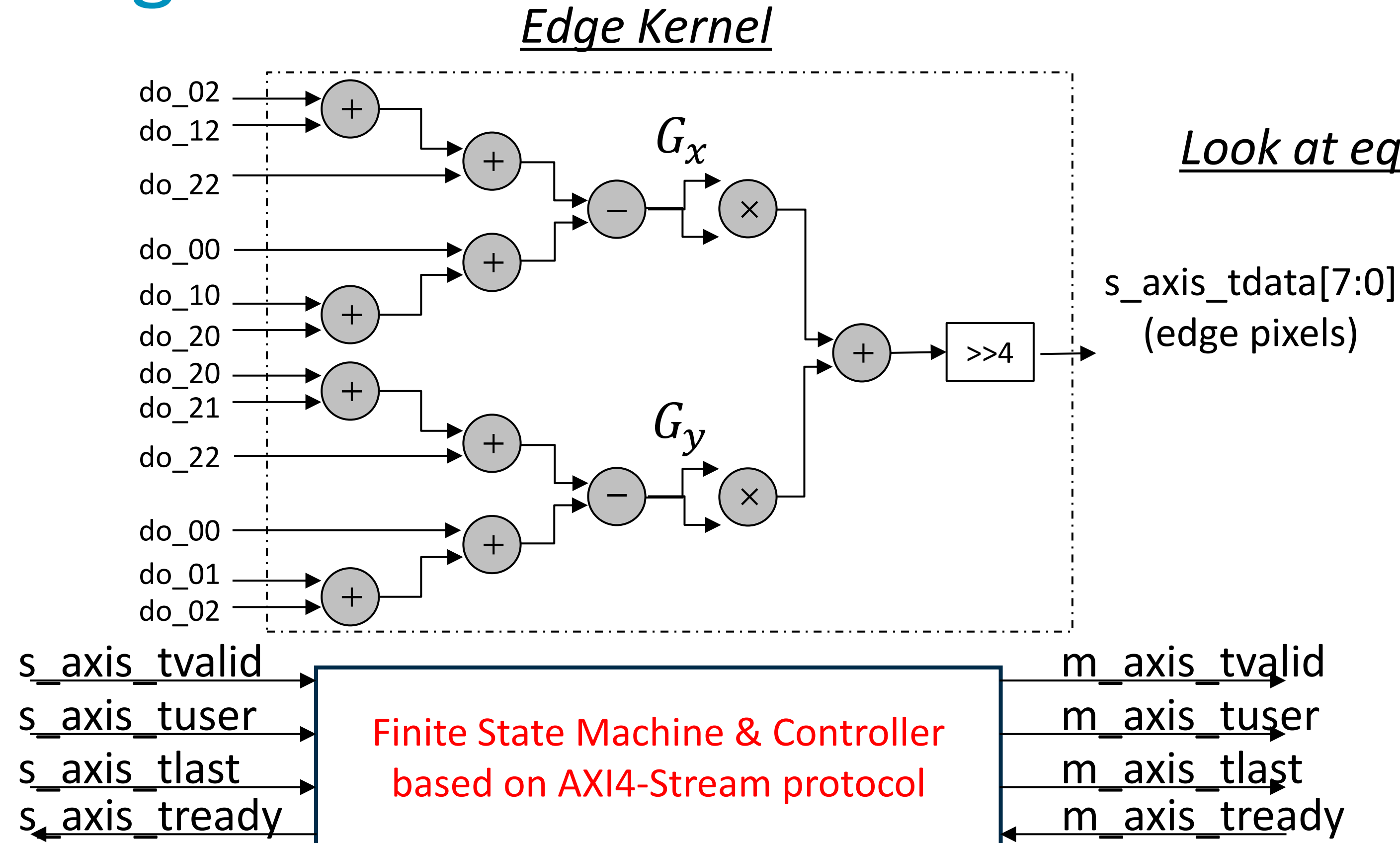


- In video streaming, one pixel comes in every 1 cycle
- For example, in order to compute edge pixel (2, 2), you need to buffer the above light-blue pixels
  - In case of 1280x720 resolution, we need $2x + 3 = 2{,}563$ buffers ($\because x = 1{,}280$)
- Get valid data for edge kernel module after 1 cycle latency
  - Our policy is that we slack computing peripheral pixels because they are not so important after all

© 2021 Arm
arm

2-2nd Module: Edge Kernel



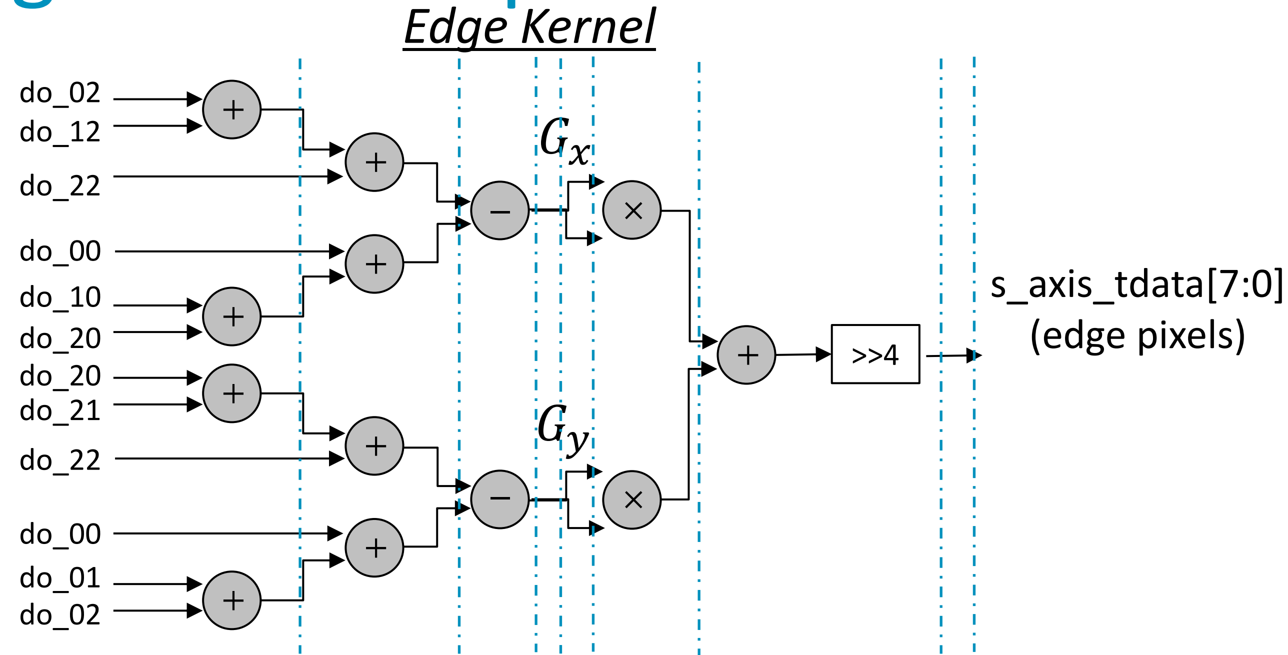


Look at eq(2)&(3) in slide No.8

- Pipeline structure for intensity computation
  - In our design, we get an edge pixel per cycle after 8cycle latency
- Separate data path and control path

© 2021 Arm
arm

2-2nd Module: Edge Kernel - Pipeline



- In our design, we put pipeline registers in above dotted line
  - 8 pipeline registers lead to 8 cycle latency
  - # of pipeline registers could be reduced in accordance with clock frequency
- Some operations deal with bit width adjustment, data saturation and you name it

© 2021 Arm

Performance Measurement

- Global timer embedded in Zynq can be used to measure the computing time consumption.

```
#include <stdio.h>
#include "xparameters.h"
#include "xtime_l.h"

int main() {
    XTime tStart, tEnd;

    /*** Retain the start time ***/
    XTime_GetTime(&tStart);
    /* ==============================================
       Operations whose processing time you measure
       ==============================================*/
    /*** Retain the end time ***/
    XTime_GetTime(&tEnd);

    /*** Calculate clock cycles and processing time ***/
    printf("Output took %llu clock cycles.\n", 2*(tEnd - tStart));
    printf("Output took %.2f us.\n",
            1.0 * (tEnd - tStart) / (COUNTS_PER_SECOND/1000000));

    return 0;
}
```


© 2021 Arm

Performance Measurement

- Custom accelerator for edge detection has a 14-cycle latency.
- Given the main clock of AXI-based system in FPGA 125MHz (or 8ns/cycle), the processing time T_{proc} applying edge detection to a single 1280x720 frame
    - $T_{proc} = 8\text{ns} \cdot \{14\text{cycles} + (1280 \cdot 720)\text{cycles}\} \approx 7.37\text{ms/frame}$

- In theory, the framerate can be achieved:
- ${}^{1\text{frame}}/_{7.4\text{ms}} \approx 135\text{FPS}$

© 2021 Arm
arm

Custom Hardware Resource Utilization

Module	LUTs (17,600)	Registers (35,200)	BRAMs (60)	DSPs (80)
Whole	5,090 (29.0%)	7,085 (20.2%)	16.5 (27.5%)	0
VDMA	2,595	3,441	14	0
Edge Detect HW	1,079	1,771	0	0

- Total utilization of LUTs, Registers, and BRAMs is approx. 20-30%
  - VDMA is responsible for almost or more than the half of it
- Custom accelerator ‘edge_detect_hw’ takes up less than 25% of the overall system with respect to LUTs and registers.

© 2021 Arm
arm

Neon vs. FPGA Offload

- Neon advantages
  - Easy programming & debug
  - Fully coherent with CPU, no cache maintenance operations
  - Part of Arm arch - no hardware or software integration required
  - Ecosystem support off-the-shelf, no porting required

- FPGA advantages
  - Runs parallel with CPU, few CPU cycles required
  - More ‘real time’ - no OS/cache variability
  - Fixed function or limited codec support
  - Potentially higher performance (e.g. 1080p Full HD video)

© 2021 Arm