MP1 Report

Yicheng Zhou (yz69), Qiran Pang (qpang2)

Part A

(a) This Part contains a simple register chain to delay the input by three cycles to output.

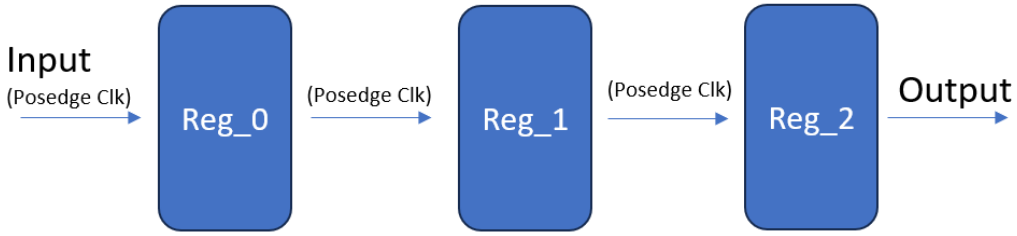


Figure 1. Block diagram for Part A

(b) This Part Consists 1 module and its instantiation in main. This module consists only a register chain as shown above that passes input, without modification, to the next stage.

(c) This design is relatively straight forward, the design process mainly consists of understanding the spec (target waveform), and implementation

(d) The resource utilization is shown in the table below

Resource	Utilization
FF	6
IO	13
LUT	0
Worst Negative Slack (WNS)	6.881
Total Negative Slack (TNS)	0

(e) This part is relatively straight forward, and does not require significant effort.

Part B

(a) This Part contains 2 translator that converts input to the specified format for mode and state. The output is a selector with basic integrated logical operation to select output based on mode and state as shown below.

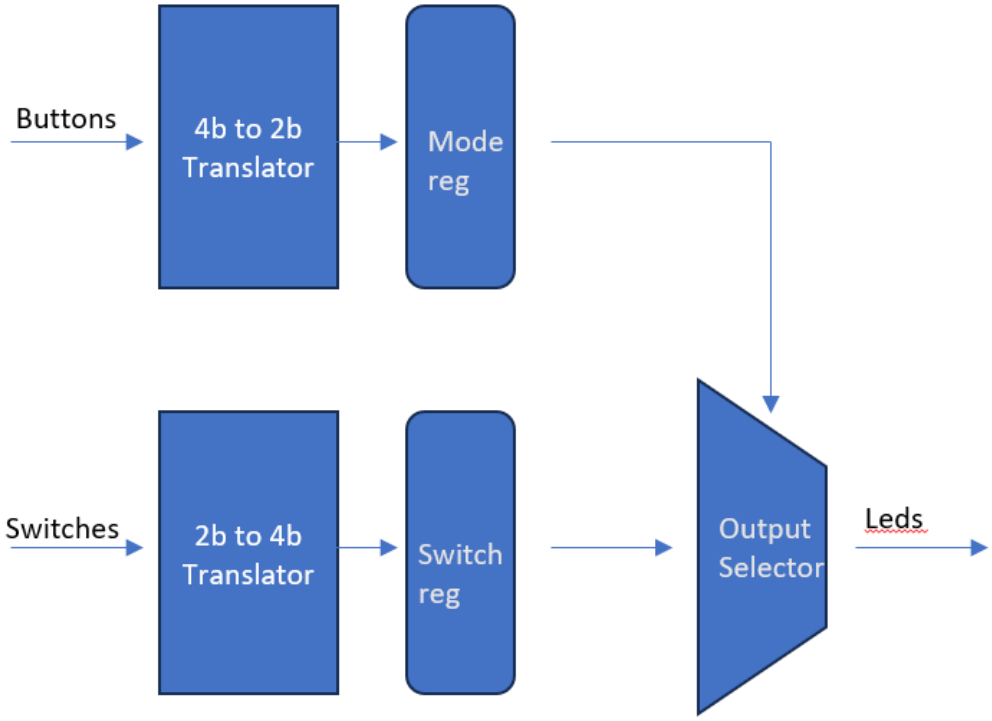


Figure 1. Block diagram for Part B

(b) This Part Consists 1 module and its instantiation in main. This module consists three multiplexors. 2 of which generates the specified mode and state signal based on a predefined look up table, the third selects the output with mode as select, and logical operated state signal as input.

(c) This design is relatively straight forward, the design process mainly consists of understanding the spec (translation table), and implementation

(d) The resource utilization is shown in the table below

Resource	Utilization
FF	6
IO	11
LUT	6
Worst Negative Slack (WNS)	6.864
Total Negative Slack (TNS)	0

(e) This part is relatively straight forward, and does not require significant effort.

Part B

(a) This Part contains 2 translator that converts input to the specified format for mode and state. The output is a selector with basic integrated logical operation to select output based on mode and state as shown below.

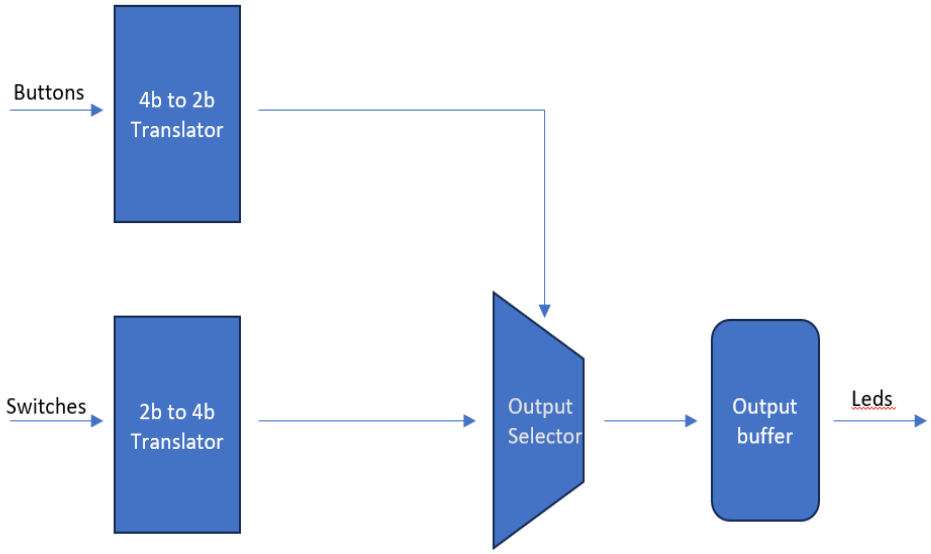


Figure 1. Block diagram for Part B

(b) This Part Consists 1 module and its instantiation in main. This module consists three multiplexors. 2 of which generates the specified mode and state signal based on a predefined look up table, the third selects the output with mode as select, and logical operated state signal as input.

(c) This design is relatively straight forward, the design process mainly consists of understanding the spec (translation table), and implementation

(d) The resource utilization is shown in the table below

Resource	Utilization
FF	6
IO	11
LUT	6
Worst Negative Slack (WNS)	6.864
Total Negative Slack (TNS)	0

(e) This part is relatively straight forward, and does not require significant effort.

Part C

(a) This Part contains Zynq integrated processing system (PS) and AXI GPIO that interfaces with PL I/O pins, with AXI interconnect used to connect and translate between AXI 4 (AXI Full) used by PS and AXI Lite used by AXI GPIO.

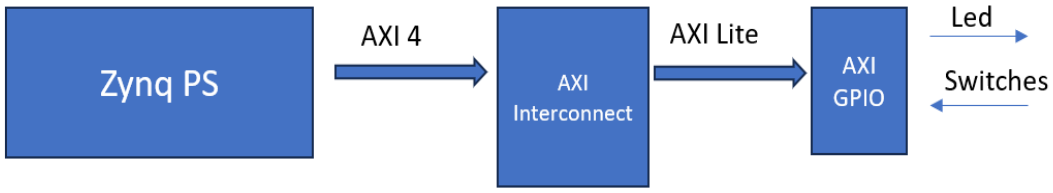


Figure 1. Block diagram for Part C

(b) The Function is mainly implemented in software (PS), where it cycles through mode, but instead of depending on buttons, the software cycles mode based on 1 second delay. The output translation follows the same lookup table as in Part B, but instead is also implemented in software.
Two IPs are utilized here, AXI GPIO, for exposing PL I/O Pins to PS, and AXI interconnect, to translate between AXI Lite and AXI 4.

(c) This design Mainly involved two parts. First we need to build the hardware platform and obtain xsa files, this involved building the basic AXI network and exposing PL I/O Pins to PS. Secondly we use the xsa to build a platform on which we implement the desired functionality with software.

(d) The resource utilization is shown in the table below

Resource	Utilization
FF	642
IO	6
LUT	537
LUT RAM	2
Worst Negative Slack (WNS)	
Total Negative Slack (TNS)	0

(e) The main learning opportunity for us is to walk through the Vivado to Vitis flow for the first time, as when we took ECE385, we were still using altera FPGA and therefore were only exposed to the Quartus embedded workflow.

Part D

(a) This design shows how the Zynq Processing System works together with a custom Caesar accelerator using an AXI DMA engine. The processor sets up the DMA through a simple AXI-Lite interface, telling it where to read and write data in DDR memory. The DMA then takes care of moving the data: it pulls input from DDR, streams it into the Caesar accelerator for processing, and writes the results back into DDR. The interconnect blocks handle the communication paths, while the reset controller keeps everything starting in sync. Finally, the DMA signals the processor with an interrupt once the job is done.

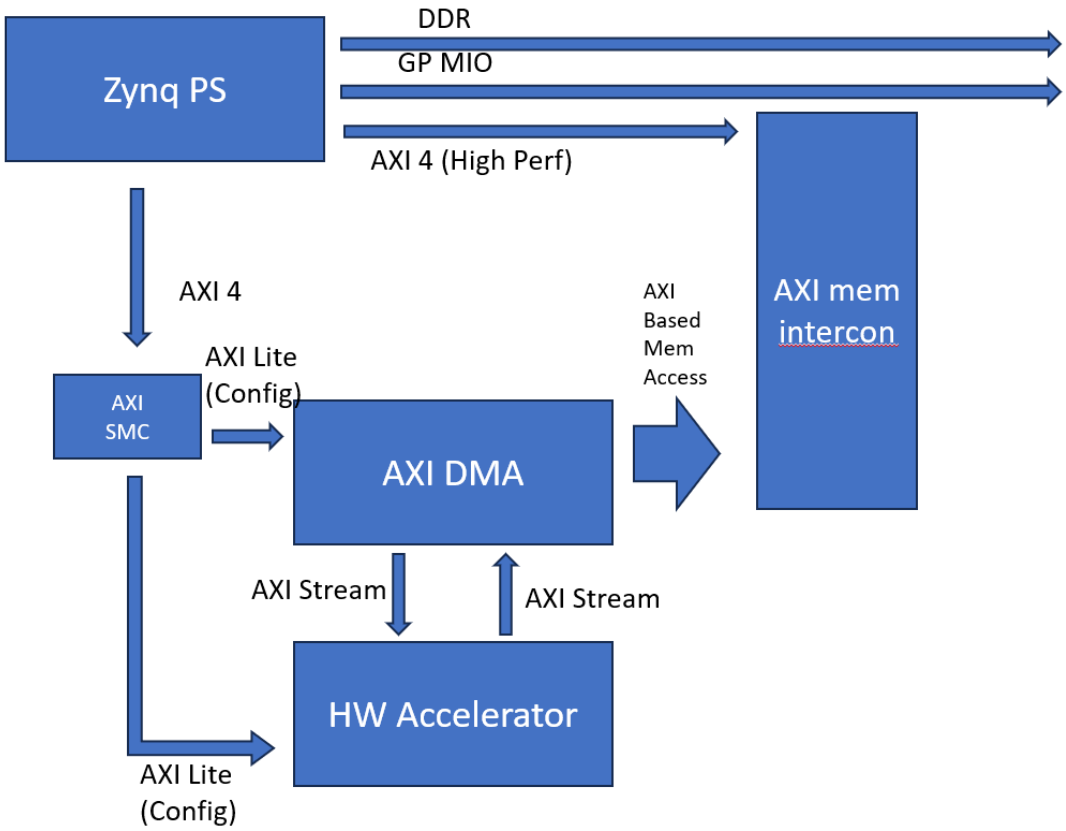


Figure 1. Block diagram for Part B

(b) The following is a per module description of the design

1. Zynq PS: the ARM CPU subsystem, which provides DDR controller, multiple AXI ports and interrupt controller for the system. In this design, it runs software, configures the DMA and accelerator through AXI Lite, Provides DDR memory space for I/O buffers and handles interrupts when DMA transfers finish.

2. AXI DMA: Bridge between DDR and accelerator, or between AXI Full and AXI Stream. It contains two main datapaths, MM2S (memory to stream) and S2MM (Stream to memory).
3. HW Accelerator: The RTL Core that provides Caesar cipher acceleration on hardware, receive stream data from DMA and outputs results to DMA.
4. AXI MEM INTERCON: an AXI interconnects that routs DMA to DDR controller via HP ports, manages data width adaptation and protocol compliances, in this case it exposes DDR controller to PL masters like AXI DMA.
5. AXI SMC: Newer generation AXI interconnects, it automatically adapts between AXI full and AXI Lite.

(c) The design flow starts on the hardware side in Vivado. First, the Zynq Processing System is configured with DDR and AXI ports enabled. Using block automation, the AXI DMA is added and connected to the PS through AXI-Lite (for control) and the HP port (for memory access). The custom Caesar accelerator is then inserted, with its AXI-Stream interfaces linked to the DMA. SmartConnect and Interconnect blocks are created automatically to handle routing, and the reset controller is included to synchronize startup. Once the block diagram is complete, HDL is generated, synthesized, and exported to SDK/Vitis.

On the software side, the ARM processor configures the DMA by programming its registers over AXI-Lite. Input and output buffers are allocated in DDR, and transfer descriptors are set up. The software starts the MM2S channel to send data into the accelerator and the S2MM channel to capture results back into memory. Once the DMA finishes, it raises an interrupt, which the software handles to verify and use the processed data. This flow separates control and data movement, letting the processor focus on setup and results while the DMA and accelerator handle high-speed processing.

One thing to note is that with in the accelerator, instead of pipelining the combinational logic, we instead have set the timing constraint to multi-cycle, which allowed us to reach no timing violation.

(d) The resource utilization is shown in the table below

Resource	Utilization
FF	5870
LUT	4429
LUT RAM	200

BRAM	2
Total Negative Slack (TNS)	0

(e) I learned how the PS, DMA, and accelerator work together through memory and streaming paths. The trickiest part was understanding the different AXI interfaces and how data actually flows between them. Setting up block automation and connections in Vivado was straightforward, while configuring DMA registers in software required more care.

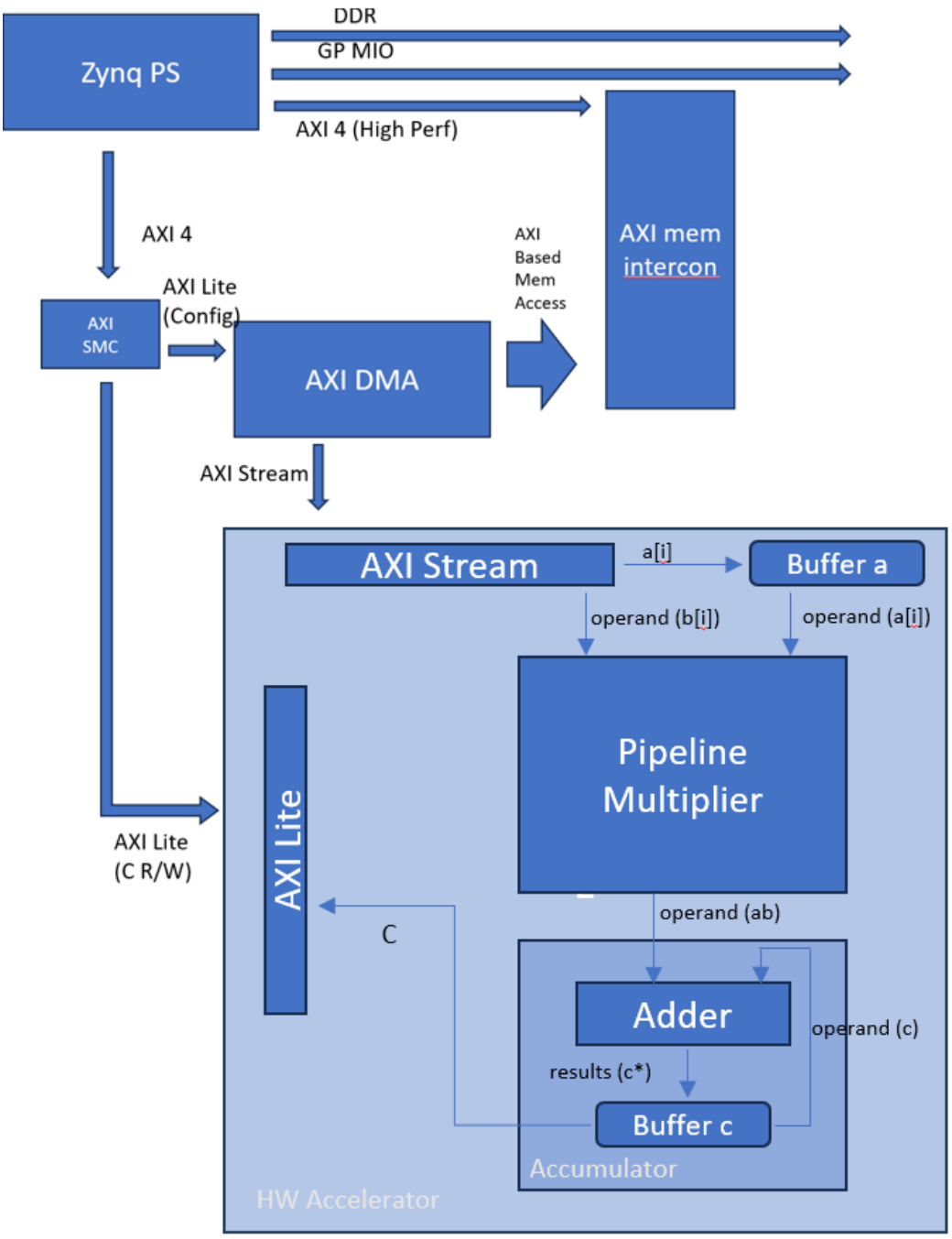
Additional Questions

(a) i. The accelerator used two protocols because neither use case requires the capabilities of AXI 4 (Full), and implementing AXI 4 for all three not only significantly increases the difficulty of the design, it also wastes resources in hardware. AXI Lite functions similarly like memory mapped registers, which is perfect for config registers, while AXI stream functions like a unilateral data dump, which is perfect for data consumer that does not require handshaking.

ii. It supports Burst-Based transaction, Multiple Outstanding Transactions, Out of Order completions, Unaligned Data Transfers, Quality of Service supports, and Atomic Operations. We used AXIL instead of AXI4 because for config registers, we don't expect frequent and high-speed changes, therefore, we don't need bursts, Multiple Outstanding Transactions. And We don't expect the channel to be busy (configs for DMA and Accelerator), there fore QoS and Out of Order completion would not be useful, additionally, since these are config, which mostly functions as write only register for PS, we don't require Atomic Operations as well. Therefore, implementing full AXI4 would only serve to increase design complexity and resource utilization with no real benefits.

iii. the interface between AXI DMA to AXI Mem intercom uses full AXI-MM protocol to allow PL to interact with DDR controller in PS, as well as AXI GP port on PS (but this is due to standardized PS interface, rather than functional needs).

(b) i. the design is shown below:


Zynq PS
DDR
GP MIO
AXI 4 (High Perf)
AXI mem intercon
AXI 4
AXI Lite (Config)
AXI SMC
AXI DMA
AXI Based Mem Access
AXI Stream
AXI Lite (C R/W)
AXI Stream
a[i]
Buffer a
operand (b[i])
operand (a[i])
Pipeline Multiplier
AXI Lite
C
operand (ab)
Adder
operand (c)
results (c*)
Buffer c
Accumulator
HW Accelerator

ii. The pseudo code is shown below:

```
// ---------- Setup ----------
map AXIL_DOT_PRODUCT_REG    // 32-bit running dot product register
map DMA_CTRL_REGS          // DMA control/status registers

// Optionally reset accumulator
AXIL_write(AXIL_DOT_PRODUCT_REG, 0)

// Prepare interleaved input buffer [a[0], b[0], a[1], b[1], ...]
buffer_in = allocate(2*N words)
for i from 0 to N-1:
   buffer_in[2*i]   = a[i]
   buffer_in[2*i+1] = b[i]
flush_cache(buffer_in)   // if cache enabled

// ---------- Configure and Start DMA ----------
DMA_write(S2MM_CONTROL, RESET)
DMA_write(S2MM_CONTROL, RUN)

DMA_write(S2MM_SRC_ADDR, phys_addr(buffer_in))
DMA_write(S2MM_LENGTH,   2*N*4)    // each 32-bit word is 4 bytes

// ---------- Wait for DMA Transfer to Complete ----------
while not (DMA_read(S2MM_STATUS) & DMA_DONE):
   sleep_small()      // short CPU delay to avoid busy loop

// ---------- Fixed Delay for Pipeline Drain ----------
cycles_to_wait = L
sec_to_wait    = cycles_to_wait / f_clk
us_to_wait     = sec_to_wait * 1e6
usleep(us_to_wait )  // cycles to wait for all data to clear the pipeline

// ---------- Final Dot Product ----------
final_c = AXIL_read(AXIL_DOT_PRODUCT_REG)

// ---------- Optional: Host can preload/overwrite C ----------
AXIL_write(AXIL_DOT_PRODUCT_REG, some_new_value)
```