AES Acceleration

COMP4601 FINAL PRESENTATION

Alyssa Lubrano, z5362292
Linfeng Cai, z5349868
Kavisha Chandraratne, z5473625
Idriz Haxhimolla z5260242,

AGENDA

- Intro
- Recap from Week 7
- Overview and Goals
- Design
  - Software Design
  - Single Core Design
  - Multi Core Design
- Results
  - Software Results
  - Single Core Results
  - Multi Core Results
- Conclusion and Further research

Recap and Overview

RECAP FROM WEEK 7

Subbytes

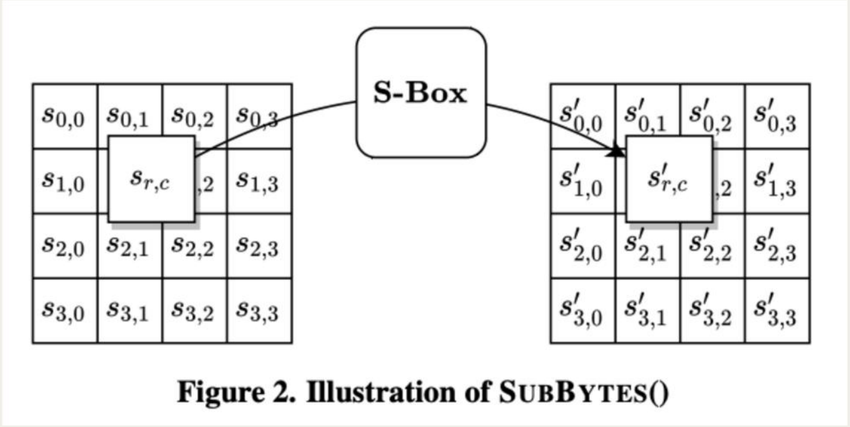


Figure 2. Illustration of SUBBYTES()

ShiftRows

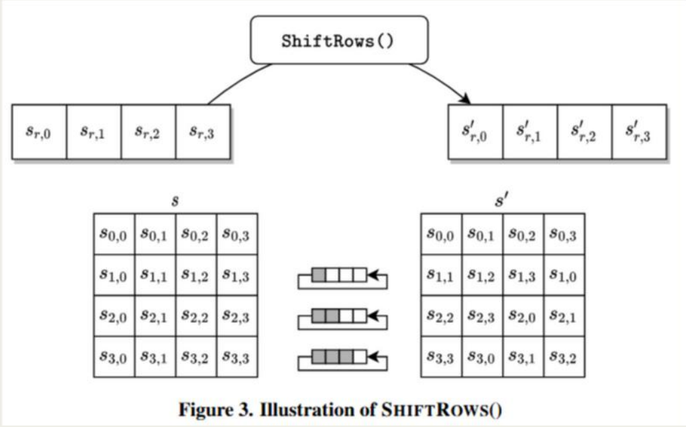
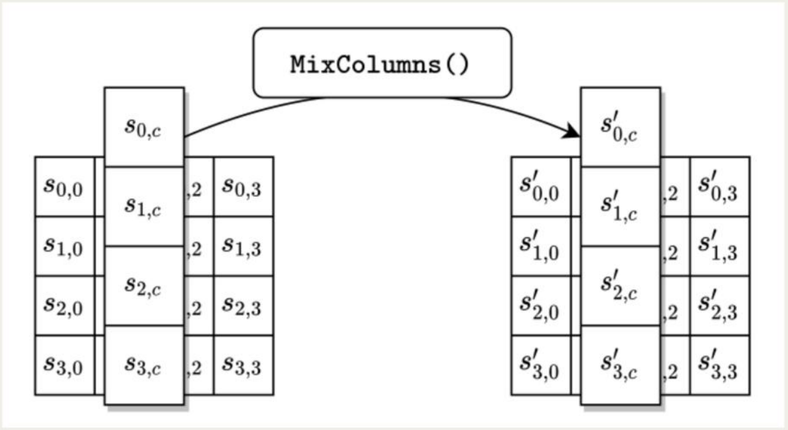


Figure 3. Illustration of SHIFTROWS()

MixColumns



AddRoundKey

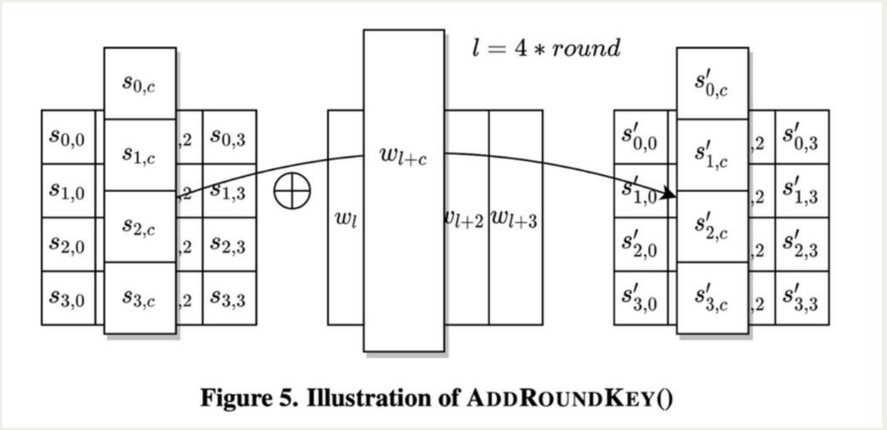


Figure 5. Illustration of ADDROUNDKEY()

OVERVIEW AND GOALS

Functioning Software AES

AIM 1:
A baseline according to the specifications outlined within FIPS 197, document on the implementation of AES

Functioning PS/PL

AIM 2:
Functioning baseline for PS/PL using AXI-lite. Optional extension was testing with AXI-STREAM4 w/ DMA

Optimisation of AES

AIM 3:
Optimise the AES code with pragmas, and/or manual changes to the code

Multicore Extension

AIM 4:
Functioning multicore code with a direct comparison to single core and PS

OVERVIEW AND GOALS

Week 7

Functioning Software AES

AIM 1:
A baseline according to the specifications outlined within FIPS 197, document on the implementation of AES

Week 8

Optimisation of AES

AIM 3:
Optimise the AES code with pragmas, and/or manual changes to the code

Week 8/9/10

Functioning PS/PL

AIM 2:
Functioning baseline for PS/PL using AXI-lite. Optional extension was testing with AXI-STREAM4 w/ DMA

Week 9/10

Multicore Extension

AIM 4:
Functioning multicore code with a direct comparison to single core and PS

The Design

Cipher w/ Pragmas for PL

```
void Cipher(byte in[16], byte out[16], const word w[]) {
    byte state[4][4];


    for (int i = 0; i < 16; i++) {
    #pragma HLS unroll
        state[i % 4][i / 4] = in[i];
    }

    AddRoundKey(state, &w[0]);

    #pragma HLS unroll
    for (int round = 1; round < Nr; round++) {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, &w[round * Nb]);
    }

    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, &w[Nr * Nb]);

    for (int i = 0; i < 16; i++)
    #pragma HLS unroll
        out[i] = state[i % 4][i / 4];
}
```

- Unrolled
  - Allow parallel operation

SubBytes w/ Pragmas for PL

```
void SubBytes(byte state[4][4]) {
    #pragma HLS inline
    for (int r = 0; r < 4; r++) {
    #pragma HLS unroll
        for (int c = 0; c < 4; c++) {
    #pragma HLS unroll
            state[r][c] = sbox[state[r][c]];
        }
    }
}
```

- Inline directive to embed function into caller
  - Reduces overhead
  - Improves latency
- Unrolled
  - Allow parallel operation

Previous iteration: 10-12 Cycles

ShiftRows w/ Pragmas for PL

```
void ShiftRows(byte state[4][4]) {
    #pragma HLS INLINE
    #pragma HLS ARRAY_PARTITION variable=state complete dim=0
    #pragma HLS ARRAY_PARTITION variable=state complete dim=1
    byte temp;

    temp = state[1][0];
    state[1][0] = state[1][1];
    state[1][1] = state[1][2];
    state[1][2] = state[1][3];
    state[1][3] = temp;

    byte t1 = state[2][0];
    byte t2 = state[2][1];
    state[2][0] = state[2][2];
    state[2][1] = state[2][3];
    state[2][2] = t1;
    state[2][3] = t2;

    temp = state[3][3];
    state[3][3] = state[3][2];
    state[3][2] = state[3][1];
    state[3][1] = state[3][0];
    state[3][0] = temp;
}
```

- Inline directive to embed function into caller
  - Reduces overhead
  - Improves latency
- Array_Partition
  - Breaks 2D array into registers
  - Allows parallel data access
- Removal of the nested for loop
  - Similar benefits as unrolling
  - Logic flows better for this case

Previous iteration: 10-12 Cycles

MixColumns w/ Pragmas for PL

```
void MixColumns(byte state[4][4]) {
    #pragma HLS ARRAY_PARTITION variable=state complete dim=0
    #pragma HLS ARRAY_PARTITION variable=state complete dim=1
    for (int c = 0; c < 4; c++) {
    #pragma HLS unroll
        byte a[4], res[4];

        a[0] = state[0][c];
        a[1] = state[1][c];
        a[2] = state[2][c];
        a[3] = state[3][c];

        res[0] = gmul_fast(a[0], 0x02) ^ gmul_fast(a[1], 0x03) ^ a[2] ^ a[3];
        res[1] = a[0] ^ gmul_fast(a[1], 0x02) ^ gmul_fast(a[2], 0x03) ^ a[3];
        res[2] = a[0] ^ a[1] ^ gmul_fast(a[2], 0x02) ^ gmul_fast(a[3], 0x03);
        res[3] = gmul_fast(a[0], 0x03) ^ a[1] ^ a[2] ^ gmul_fast(a[3], 0x02);

        a[0] = state[0][c];
        a[1] = state[1][c];
        a[2] = state[2][c];
        a[3] = state[3][c];
    }
}
```

- Array_Partition
  - Breaks 2D array into registers
  - Allows parallel data access
- Unrolled
  - Allow parallel operation
- Removal of the nested for loop
  - Similar benefits as unrolling
  - Logic flows better for this case

Previous iteration: 10-12 Cycles

AddRoundKey w/ Pragmas for PL

```
void AddRoundKey(byte state[4][4], const word roundKey[4]) {
    #pragma HLS INLINE
    #pragma HLS ARRAY_PARTITION variable=state complete dim=0
    #pragma HLS ARRAY_PARTITION variable=state complete dim=1

    for (int j = 0; j < 4; j++) {
    #pragma HLS UNROLL
        for (int i = 0; i < 4; i++) {
    #pragma HLS UNROLL
            ap_uint<8> k = (roundKey[j] >> (24 - 8 * i)) & 0xFF;
            state[i][j] ^= byte(k);
        }
    }
}
```

- Inline directive to embed function into caller
  - Reduces overhead
  - Improves latency
- Array_Partition
  - Breaks 2D array into registers
  - Allows parallel data access
- Unrolled
  - Allow parallel operation
  - Essentially removes the nested loops

Previous iteration: 10-12 Cycles

Cipher Function for PS, and data for PS/PL

```
void Cipher(byte in[16], byte out[16], const word w[]) {
    byte state[4][4];

    for (int i = 0; i < 16; i++)
        state[i % 4][i / 4] = in[i];

    AddRoundKey(state, &w[0]);

    for (int round = 1; round < Nr; round++) {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, &w[round * Nb]);
    }

    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, &w[Nr * Nb]);

    for (int i = 0; i < 16; i++)
        out[i] = state[i % 4][i / 4];
}
```

```
// auto generated 10 packets long input data
const byte input_data[10][16] = {
    { 0x54, 0x68, 0x65, 0x20, 0x71, 0x75, 0x69, 0x63, 0x6B, 0x20, 0x62, 0x72, 0x6F, 0x77, 0x6E, 0x20 }, // "The quick brown "
    { 0x66, 0x6F, 0x78, 0x20, 0x6A, 0x75, 0x6D, 0x70, 0x73, 0x20, 0x6F, 0x76, 0x65, 0x72, 0x20, 0x74 }, // "fox jumps over t"
    { 0x68, 0x65, 0x20, 0x6C, 0x61, 0x7A, 0x79, 0x20, 0x64, 0x6F, 0x67, 0x2E, 0x20, 0x54, 0x68, 0x65 }, // "he lazy dog. The"
    { 0x20, 0x71, 0x75, 0x69, 0x63, 0x6B, 0x20, 0x62, 0x72, 0x6F, 0x77, 0x6E, 0x20, 0x66, 0x6F, 0x78 }, // " quick brown fox"
    { 0x20, 0x6A, 0x75, 0x6D, 0x70, 0x73, 0x20, 0x6F, 0x76, 0x65, 0x72, 0x20, 0x74, 0x68, 0x65, 0x20 }, // " jumps over the "
    { 0x6C, 0x61, 0x7A, 0x79, 0x20, 0x64, 0x6F, 0x67, 0x2E, 0x20, 0x54, 0x68, 0x65, 0x20, 0x71, 0x75 }, // "lazy dog. The qu"
    { 0x69, 0x63, 0x6B, 0x20, 0x62, 0x72, 0x6F, 0x77, 0x6E, 0x20, 0x66, 0x6F, 0x78, 0x20, 0x6A, 0x75 }, // "ick brown fox ju"
    { 0x6D, 0x70, 0x73, 0x20, 0x6F, 0x76, 0x65, 0x72, 0x20, 0x74, 0x68, 0x65, 0x20, 0x6C, 0x61, 0x7A }, // "mps over the laz"
    { 0x79, 0x20, 0x64, 0x6F, 0x67, 0x2E, 0x20, 0x54, 0x68, 0x65, 0x20, 0x71, 0x75, 0x69, 0x63, 0x6B }, // "y dog. The quick"
    { 0x20, 0x62, 0x72, 0x6F, 0x77, 0x6E, 0x20, 0x66, 0x6F, 0x78, 0x20, 0x6A, 0x75, 0x6D, 0x70, 0x73 }  // " brown fox jumps"
};
```

```
byte key[16] = {
    0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F, 0x70
};
```

```
for (int packet_index = 0; packet_index < NUM_PACKETS; packet_index++) {
    // Encode packet_index into 4 words
    uint32_t a = (packet_index >> 0) & 0xFF;
    uint32_t b = (packet_index >> 8) & 0xFF;
    uint32_t c = (packet_index >> 16) & 0xFF;
    uint32_t d = (packet_index >> 24) & 0xFF;
```


Note: The input data method is different in the final submission. Results are still the same

SOFTWARE DESIGN

PS

```
int ps() {
    xil_printf("Beginning of Software AES Results...\r\n");
    XTime start, end;
    XTime_GetTime(&start);

    byte in[16];
    byte out[16];

    byte key[16] = {
        0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F, 0x70
    };

    word w[44];
    KeyExpansion(key, w);

    for (int i = 0; i < 10; i++) {
        memcpy(in, input_data[i], 16);

        Cipher(in, out, w);

        xil_printf("Packet %d's Result: %02X%02X%02X%02X_%02X%02X%02X%02X_%02X%02X%02X%02X_%02X%02X%02X%02X\r\n",
            i, out[0], out[1], out[2], out[3], out[4], out[5], out[6], out[7], out[8], out[9], out[10], out[11], out[12], out[13], out[14], out[15]);
    }

    XTime_GetTime(&end);
    u64 cycles = end - start;
    double seconds = ((double) cycles) / COUNTS_PER_SECOND;

    xil_printf("End of the Software AES Results.\r\n This process took %llu cycles, aka %.6f seconds\r\n", cycles, seconds);

    return 0;
}
```


Note: The input data is different in the final submission so code looks slightly different

PL Single Core

```c
int pl_singlecore() {
    xil_printf("Beginning of SINGLE Core Hardware AES Results...\r\n");
    XTime start, end;
    XTime_GetTime(&start);

    XSimple_add adder;

    if (XSimple_add_Initialize(&adder, simple_add_baseaddrs[0]) != XST_SUCCESS) {
        xil_printf("Initialisation of the SINGLE core has failed :(\r\n");
        return -1;
    }

    for (int i = 0; i < 10; i++) {
        uint32_t a = (input_data[i][0] << 24) | (input_data[i][1] << 16) | (input_data[i][2] << 8) | (input_data[i][3]);
        uint32_t b = (input_data[i][4] << 24) | (input_data[i][5] << 16) | (input_data[i][6] << 8) | (input_data[i][7]);
        uint32_t c = (input_data[i][8] << 24) | (input_data[i][9] << 16) | (input_data[i][10] << 8) | (input_data[i][11]);
        uint32_t d = (input_data[i][12] << 24) | (input_data[i][13] << 16) | (input_data[i][14] << 8) | (input_data[i][15]);

        XSimple_add_Set_a(&adder, a);
        XSimple_add_Set_b(&adder, b);
        XSimple_add_Set_c(&adder, c);
        XSimple_add_Set_d(&adder, d);
        XSimple_add_Start(&adder);

        // staller
        while(!XSimple_add_IsDone(&adder));

        uint32_t res_a = XSimple_add_Get_res_a(&adder);
        uint32_t res_b = XSimple_add_Get_res_b(&adder);
        uint32_t res_c = XSimple_add_Get_res_c(&adder);
        uint32_t res_d = XSimple_add_Get_res_d(&adder);

        xil_printf("Packet %d's Result: %08X_%08X_%08X_%08X\r\n", i, res_d, res_c, res_b, res_a);

    }
    XTime_GetTime(&end);
    u64 cycles = end - start;
    double seconds = ((double) cycles) / COUNTS_PER_SECOND;

    xil_printf("End of the SINGLE core Hardware AES Results.\r\n This process took %llu cycles, aka %.6f seconds\r\n", cycles, seconds);

    return 0;
}
```


Note: The input data is different in the final submission so code looks slightly different

PL Multi Core

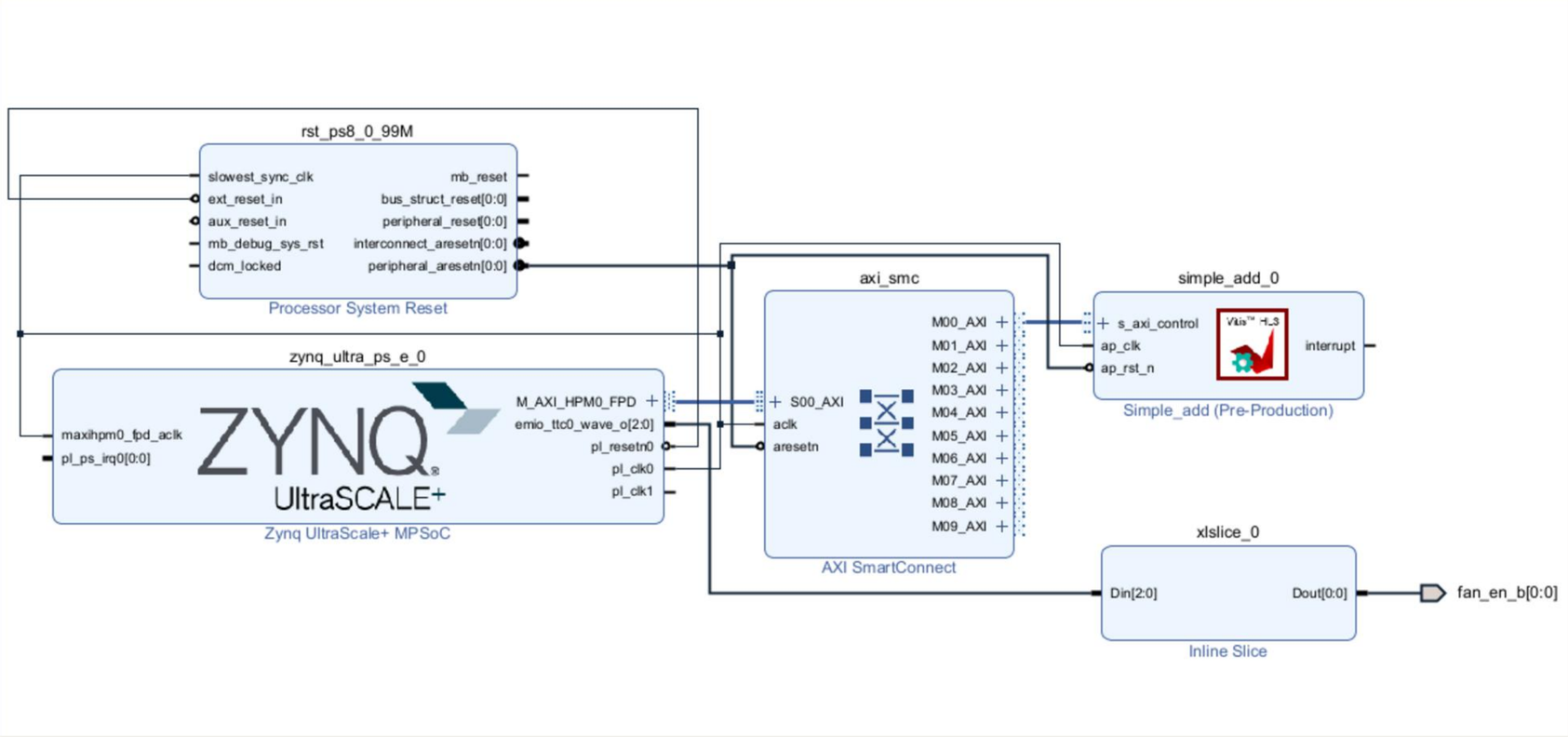
```c
int pl_10core() {
    xil_printf("Beginning of MULTI Core Hardware AES Results...\r\n");
    XTime start, end;
    XTime_GetTime(&start);

    XSimple_add adder[10];
    CoreStatus core_status[10];

    for (int i = 0; i < 10; i++) {
        core_status[i].in_use = false;
        core_status[i].packet_index = -1;
    }

    int packets_started = 0;
    int packets_finished = 0;
    uint32_t packet_results[10][4] = {0};

    for (int i = 0; i < 10; i++) {
        if (XSimple_add_Initialize(&adder[i], simple_add_baseaddrs[i]) != XST_SUCCESS) {
            xil_printf("Initialisation of MULTI core '%d' has failed :(\r\n", i);
            return -1;
        }
    }
```

```c
while (packets_finished < 10) {
    for (int i = 0; i < 10 && packets_started < 10; i++) {
        if (core_status[i].in_use == false) {

            uint32_t a = (input_data[packets_started][0] << 24) | (input_data[packets_started][1] << 16) |
                         (input_data[packets_started][2] << 8) | (input_data[packets_started][3]);
            uint32_t b = (input_data[packets_started][4] << 24) | (input_data[packets_started][5] << 16) |
                         (input_data[packets_started][6] << 8) | (input_data[packets_started][7]);
            uint32_t c = (input_data[packets_started][8] << 24) | (input_data[packets_started][9] << 16) |
                         (input_data[packets_started][10] << 8) | (input_data[packets_started][11]);
            uint32_t d = (input_data[packets_started][12] << 24) | (input_data[packets_started][13] << 16) |
                         (input_data[packets_started][14] << 8) | (input_data[packets_started][15]);

            XSimple_add_Set_a(&adder[i], a);
            XSimple_add_Set_b(&adder[i], b);
            XSimple_add_Set_c(&adder[i], c);
            XSimple_add_Set_d(&adder[i], d);
            XSimple_add_Start(&adder[i]);

            core_status[i].in_use = true;
            core_status[i].packet_index = packets_started;
            packets_started++;
        }
    }

    for (int i = 0; i < 10; i++) {
        if (core_status[i].in_use == true && XSimple_add_IsDone(&adder[i])) {
            int packet = core_status[i].packet_index;
            packet_results[packet][0] = XSimple_add_Get_res_a(&adder[i]);
            packet_results[packet][1] = XSimple_add_Get_res_b(&adder[i]);
            packet_results[packet][2] = XSimple_add_Get_res_c(&adder[i]);
            packet_results[packet][3] = XSimple_add_Get_res_d(&adder[i]);

            core_status[i].in_use = false;
            packets_finished++;
        }
    }
}
```

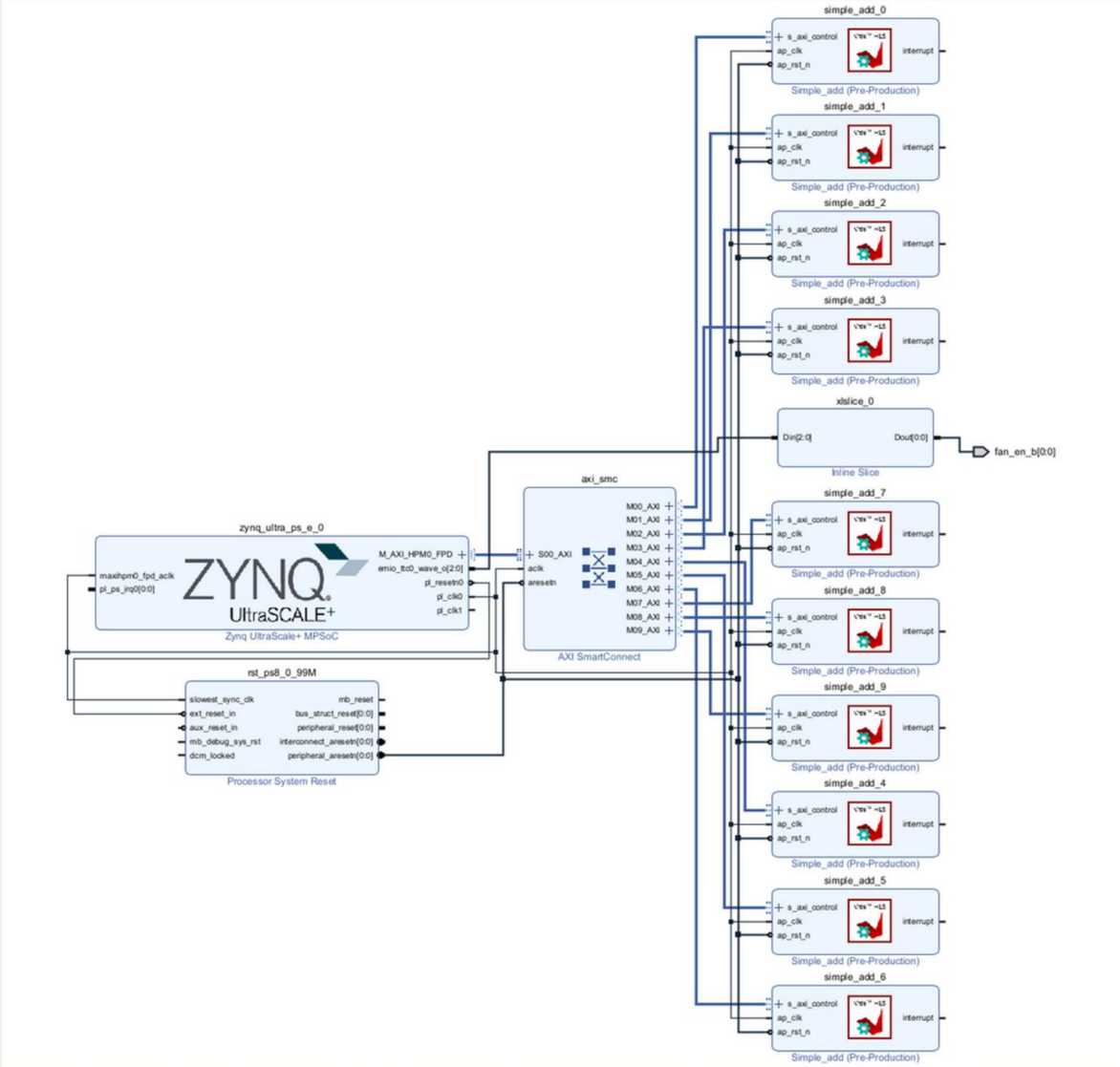
```c
    for (int i = 0; i < 10; i++) {
        xil_printf("Packet %d's Result: %08X_%08X_%08X_%08X\r\n", i, packet_results[i][3], packet_results[i][2], packet_results[i][1], packet_results[i][0]);
    }

    XTime_GetTime(&end);
    u64 cycles = end - start;
    double seconds = ((double) cycles) / COUNTS_PER_SECOND;

    xil_printf("End of the MULTI core Hardware AES Results.\r\n This process took %llu cycles, aka %.6f seconds\r\n", cycles, seconds);

    return 0;
}
```

Note: The input data is different in the final submission so code looks slightly different

PL Single Core



PL Multi Core



The Results

Single core Comparison overview

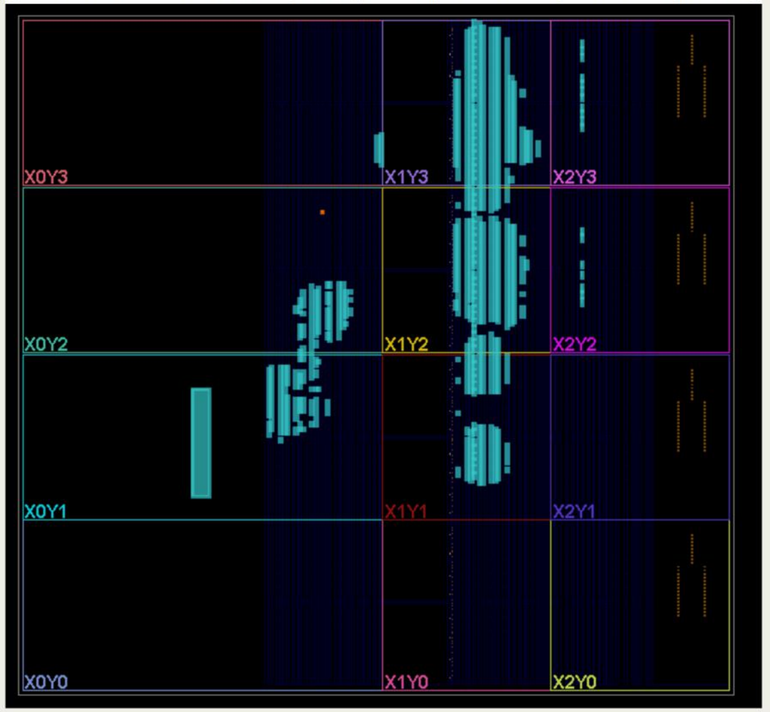
Metric	Optimized	Unoptimized
Platform	PL (RTL via AXI4-Lite) (100Mhz)	PS(A53 Cortex Vitis bare-metal) (553Mhz)
Latency (Cycles)	123	513
Latency (Effective)	110 ns	1401 ns
Iteration Interval(II)	12	1 (Key loops)
BRAM Usage	8	3
LUT Usage	3087	4617
FF Usage	610	885
S-box ROMs	8 instantiated	8 instantiated

NAME	II	LATENCY
simple_add	124	123
Cipher	105	105
Cipher_Pipeline_VITIS_LOOP_301_1	17	18

Utilisation

Resource	Used	Available	Utilisation
LUTs	1865	117120	1.59%
Registers	2011	234240	0.86%
BRAM (RAMB18)	17	288	5.90%
DSPs	0	1248	0.00%
IOs	1	189	0.53%
Clock BUFG_PS	1	96	1.04%

Site Type	Used	Fixed	Prohibited	Available	Util%
CLB LUTs	1865	0	0	117120	1.59
LUT as Logic	1855	0	0	117120	1.58
LUT as Memory	10	0	0	57600	0.02
LUT as Distributed RAM	8	0			
LUT as Shift Register	2	0			
CLB Registers	2011	0	0	234240	0.86
Register as Flip Flop	2011	0	0	234240	0.86
Register as Latch	0	0	0	234240	0.00
CARRY8	5	0	0	14640	0.03
F7 Muxes	5	0	0	58560	<0.01
F8 Muxes	0	0	0	29280	0.00
F9 Muxes	0	0	0	14640	0.00



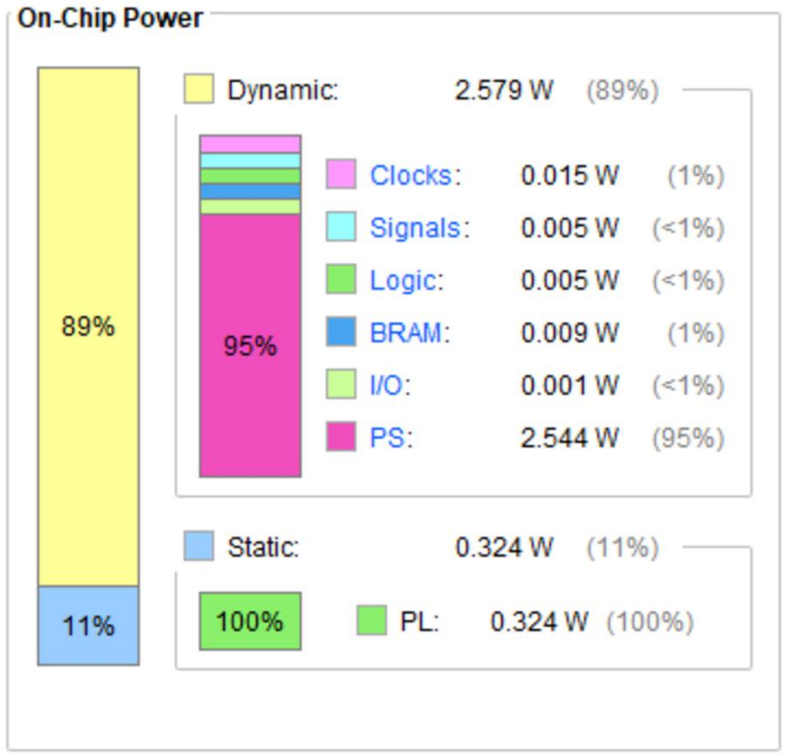
Power Consumption

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**2.903 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**31.7°C**
Thermal Margin:	53.3°C (22.7 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	2.3°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



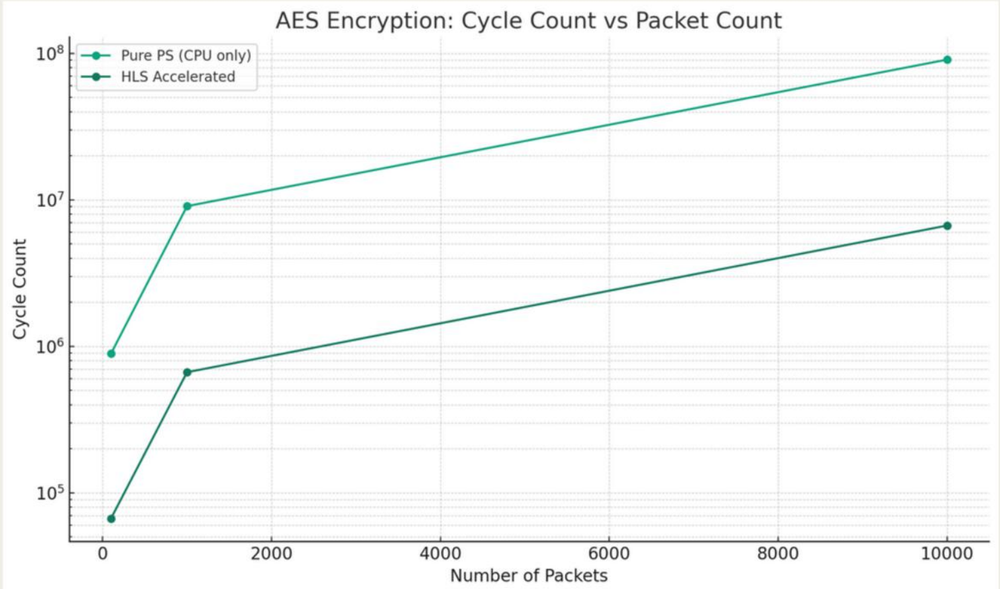
PS Dominates Power Consumption

- 95% of dynamic power (2.544 W) is consumed by the Processing System (PS), showing minimal activity in the programmable logic. Shifting tasks to PL could improve efficiency.

Low Thermal Load

- The junction temperature is just 31.7 °C with a 53.3 °C thermal margin, indicating ample headroom for additional cores or increased performance without overheating.

Speedup and Acceleration



Packet Count	Software Time (µs)	Hardware Time (µs)	Speedup (×)
100	8928.37	669.47	13.3×
1000	90554.2	6662.8	13.6×
10000	905431.87	66753.09	13.6×

PL Single Core Findings

Hard ware accelerated:

- Through fully unrolling and inlining all internal loops
  - Unroll: increase parallel paths
  - Pipeline: maximize throughput (new input every cycle)
  - Significantly reduced clock cycles (513 ↠ 123)
- 100 Mhz PL clock
  - Drastically reduced cycle ↠ improvement in effective latency
    - 1401 ns (513 cycles) ↠ 110 ns (123cycles)

PL Multi Core Findings

10 core implementation

- Take double the time of software AES core
  - With same throughput
  - Significant time spent in AXI-lite used for input and output communication
  - Disregarding data transfer between PS-PL
    - Algorithm does show improved
      - performance
      - Throughput

Conclusions and Further Research

CONCLUSIONS AND FURTHER RESEARCH

HLS
conclusion

- Based C implementation of AES on the reference provided by National Institute of Standards and Technology (US).
- Imported the C code into Vivado HLS for hardware synthesis.
- Applied HLS directives to reduce latency from ~500 cycles to 123 cycles.
- Exported the optimised HLS design as an IP block into Vivado.

PS vs HLS
conclusion

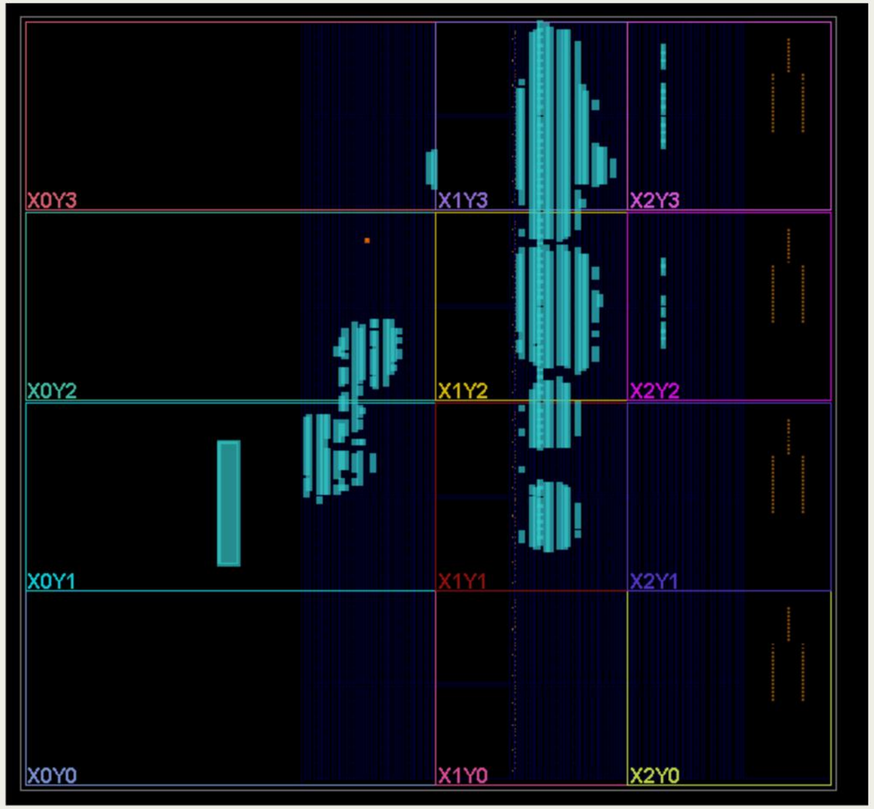
- The optimised AES IP was integrated into Vivado and connected to the Processing System via AXI Lite.
- While AXI Lite is designed for control rather than high-throughput data, it was sufficient for validating and benchmarking the system.
- The full encryption process was offloaded to the FPGA, yielding a 13× speedup over the software-only version.
- This performance gain remained consistent across input sizes from 10 to 10,000 packets.

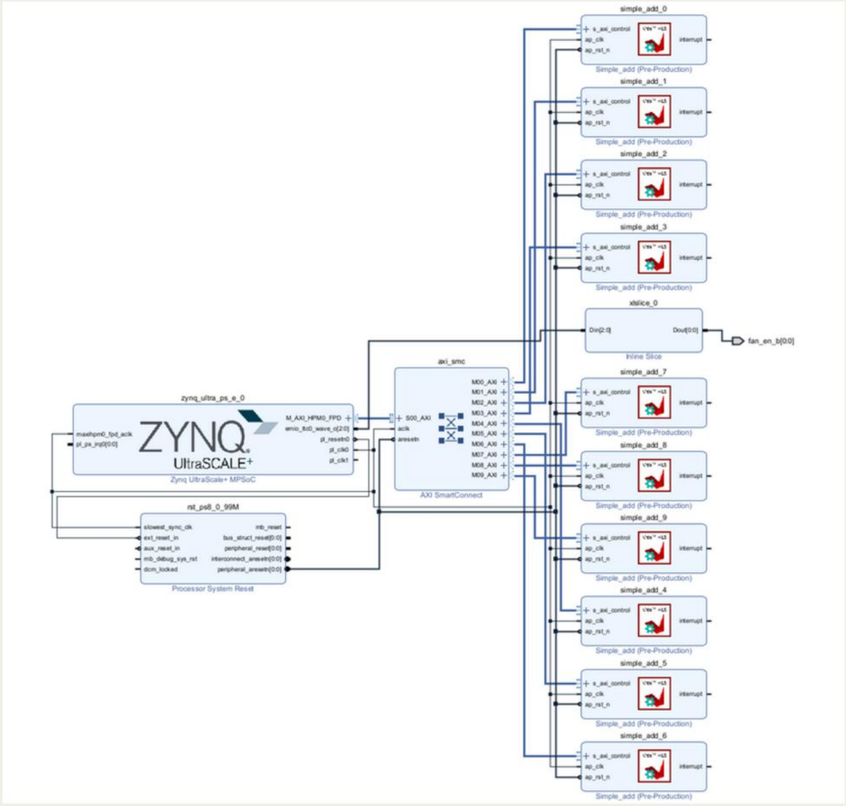
PS ↠ PL Speedup Achieved!

13x

Speedup
with Acceleration

Potential for More Acceleration with Multiple Cores

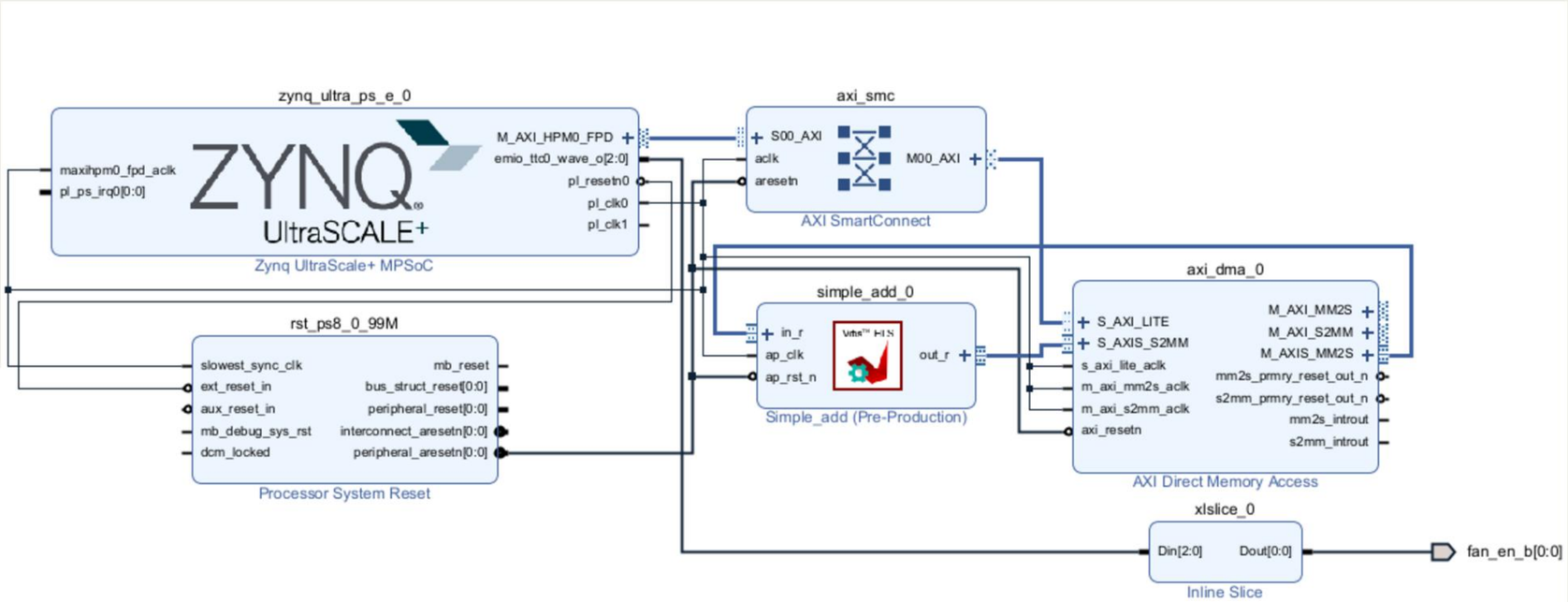




- Parallel execution through core-level scheduling can achieve significantly higher speedups.
- The multi-core design using AXI-Lite performed worse than the single-core implementation, highlighting the limitations of AXI-Lite.

Accelerating Throughput: AXI4-Stream with DMA Integration

```
xil_printf("Initialising AXI DMA...\r\n");

XAxiDma_Config *cfg = XAxiDma_LookupConfig(DMA_DEV_ID);
if (!cfg) {
    xil_printf("No config found for DMA\r\n");
    return XST_FAILURE;
}

status = XAxiDma_CfgInitialize(&AxiDma, cfg);
if (status != XST_SUCCESS) {
    xil_printf("DMA Init failed: %d\r\n", status);
    return XST_FAILURE;
}

if (!XAxiDma_HasSg(&AxiDma)) {
    xil_printf("SG mode is disabled. Proceeding with simple mode.\r\n");
} else {
    xil_printf("ERROR: Scatter-Gather mode enabled. This code only supports simple mode.\r\n");
    return XST_FAILURE;
}
```



- Potential for significantly higher speedup with DMA + more cores.

CONCLUSIONS AND FURTHER RESEARCH

Pure Software
conclusion

- Easy to set up thanks to ample resources

Single Core
conclusion

- considerably faster than the pure software

Multi Core
conclusion

- surprisingly less efficient than single core
- AND less efficient than pure software

AXI-Stream + DMA
further research

- could try using AXI-stream w/ DMA
- Last year's group didn't find any acceleration with it
- BUT they didn't try multicore
- AXI-lite not optimised for Multicore, but maybe DMA is?

Thank you!

ANY QUESTIONS?

Alyssa Lubrano, z5362292
Linfeng Cai, z5349868
Kavisha Chandraratne, z5473625
Idriz Haxhimolla, z5260242

```c
int ps() {
    xil_printf("Initialising (Software AES)...\r\n");

    XTime start, end;
    XTime_GetTime(&start);

    byte key[16] = {
        0xCA, 0xFE, 0xBA, 0xBE,
        0xCA, 0xFE, 0xBA, 0xBE,
        0xCA, 0xFE, 0xBA, 0xBE,
        0xCA, 0xFE, 0xBA, 0xBE
    };
    word w[44];
    KeyExpansion(key, w);

    for (int packet_index = 0; packet_index < NUM_PACKETS; packet_index++) {
        byte in[16] = {0};

        // Encode packet_index into the input block (first 4 bytes)
        in[0] = (packet_index >> 0) & 0xFF;
        in[1] = (packet_index >> 8) & 0xFF;
        in[2] = (packet_index >> 16) & 0xFF;
        in[3] = (packet_index >> 24) & 0xFF;

        byte out[16];
        Cipher(in, out, w);

        if (PRINT) {
            xil_printf("Packet %d → Result: ", packet_index);
            for (int i = 0; i < 16; i++) {
                xil_printf("%02X", out[i]);
                if ((i + 1) % 4 == 0) xil_printf("_");
            }
            xil_printf("\r\n");
        }
    }

    XTime_GetTime(&end);
    u64 cycles = end - start;
    double seconds = ((double)cycles) / COUNTS_PER_SECOND;

    char cycles_str[32];
    format_commas(cycles, cycles_str);

    xil_printf("Software AES completed in %s cycles (%.6f seconds)\r\n", cycles_str, seconds);
    return 0;
}
```

```c
int pl_singlecore() {
    xil_printf("Initialising single core...\r\n");

    XTime start, end;
    XTime_GetTime(&start);

    XSimple_add adder;
    int status = XSimple_add_Initialize(&adder, simple_add_baseaddrs[0]);
    if (status != XST_SUCCESS) {
        xil_printf("Init failed for core 0\r\n");
        return -1;
    }

    for (int packet_index = 0; packet_index < NUM_PACKETS; packet_index++) {
        // Encode packet_index into 4 words
        uint32_t a = (packet_index >> 0) & 0xFF;
        uint32_t b = (packet_index >> 8) & 0xFF;
        uint32_t c = (packet_index >> 16) & 0xFF;
        uint32_t d = (packet_index >> 24) & 0xFF;

        XSimple_add_Set_a(&adder, a);
        XSimple_add_Set_b(&adder, b);
        XSimple_add_Set_c(&adder, c);
        XSimple_add_Set_d(&adder, d);
        XSimple_add_Start(&adder);

        while (!XSimple_add_IsDone(&adder));

        uint32_t r0 = XSimple_add_Get_res_a(&adder);
        uint32_t r1 = XSimple_add_Get_res_b(&adder);
        uint32_t r2 = XSimple_add_Get_res_c(&adder);
        uint32_t r3 = XSimple_add_Get_res_d(&adder);

        if (PRINT) {
            xil_printf("Packet %d → Result: %08X_%08X_%08X_%08X\r\n",
                    packet_index, r3, r2, r1, r0);
        }
    }

    XTime_GetTime(&end);
    u64 cycles = end - start;
    double seconds = ((double)cycles) / COUNTS_PER_SECOND;

    char cycles_str[32];
    format_commas(cycles, cycles_str);

    xil_printf("Hardware Accelerated AES single core completed in %s cycles (%.6f seconds)\r\n", cycles_str, se
    xil_printf("All %d packets processed on single core.\r\n", NUM_PACKETS);

    return 0;
}
```