COMP4601 PROJECT

AES ENCRYPTION ACCELERATOR

PRESENTED BY:
Alyssa, Linfeng, Kavisha and Idriz

PRESENTED TO:
COMP4601 Cohort

Agenda

01	Problem Specification
02	Project Aims
03	Algorithm Overview
04	Pseudocode
05	Steps 1-3 Findings
06	Opportunities for Acceleration
07	Project Timeline
08	Risks and Contingencies
09	Roles of the Team Members

01 Encryption

02 AES (Advanced Encryption Standard)

03 FPGA Board

04 VivadoHLS

Problem Specification

Produce an FPGA that can take in 128 bit plain text and output its encrypted form using a provided key

Problem Aims

aka to produce a refined and functioning AES encryptor that is both efficient and effective.

Aim # 1

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

Aim # 2

Functioning baseline on FPGA board

Aim # 3

Optimisations for the code

Aim # 4

Analysis as compared to baseline and iterations of optimisations

```
procedure CIPHER(in, Nr, w)
    state ← in                                  // Initialize the state matrix
from input bytes

    state ← ADDROUNDKEY(state, w[0..3]) // Initial round key addition

    for round from 1 to Nr - 1 do
        state ← SUBBYTES(state)          // Byte-wise substitution using
S-box
        state ← SHIFTROWS(state)         // Row-wise cyclic shifts
        state ← MIXCOLUMNS(state)        // Call the separated MixColumns
function
        state ← ADDROUNDKEY(state, w[4*round..4*round + 3])
    end for

    state ← SUBBYTES(state)              // Final round (no MixColumns)
    state ← SHIFTROWS(state)
    state ← ADDROUNDKEY(state, w[4*Nr..4*Nr + 3])

    return state                                // Return the output bytes
end procedure
```

The Operations Pseudocode

AddRoundKey

```
procedure ADDROUNDKEY(state, w)
  for c from 0 to 3 do
      state[0][c] ← state[0][c] ⊕ byte(w[c], 3)
      state[1][c] ← state[0][c] ⊕ byte(w[c], 2)
      state[2][c] ← state[0][c] ⊕ byte(w[c], 1)
      state[3][c] ← state[0][c] ⊕ byte(w[c], 0)
  end for
  return state
end procedure
```

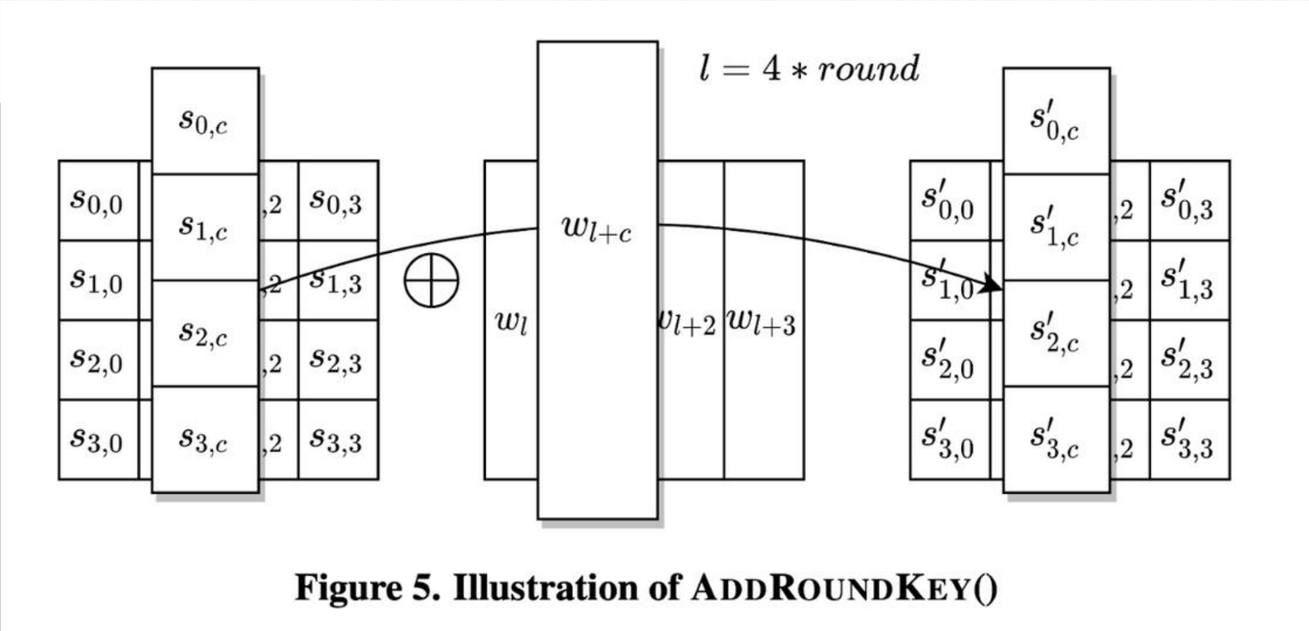


Figure 5. Illustration of AddRoundKey()

Subbytes

```
procedure SubBytes(state, w)
  for c from 0 to 3 do
      state[0][c] ← sbox[0][c]
      state[0][c] ← sbox[1][c]
      state[0][c] ← sbox[2][c]
      state[0][c] ← sbox[3][c]
  end for
  return state
end procedure
```

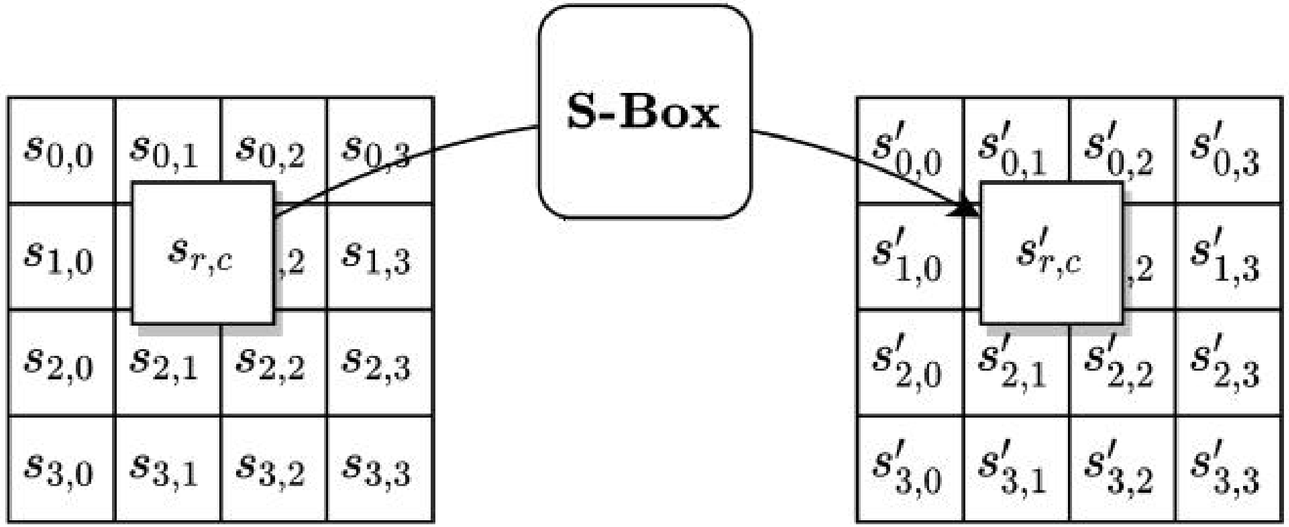


Figure 2. Illustration of SUBBYTES()

ShiftRows

```
procedure SHIFTROWS(state)
  for r from 1 to 3 do
    for shift_count from 1 to r do
      temp ← state[r][0]
      state[r][0] ← state[r][1]
      state[r][1] ← state[r][2]
      state[r][2] ← state[r][3]
      state[r][3] ← temp
    end for
  end for
  return state
end procedure
```

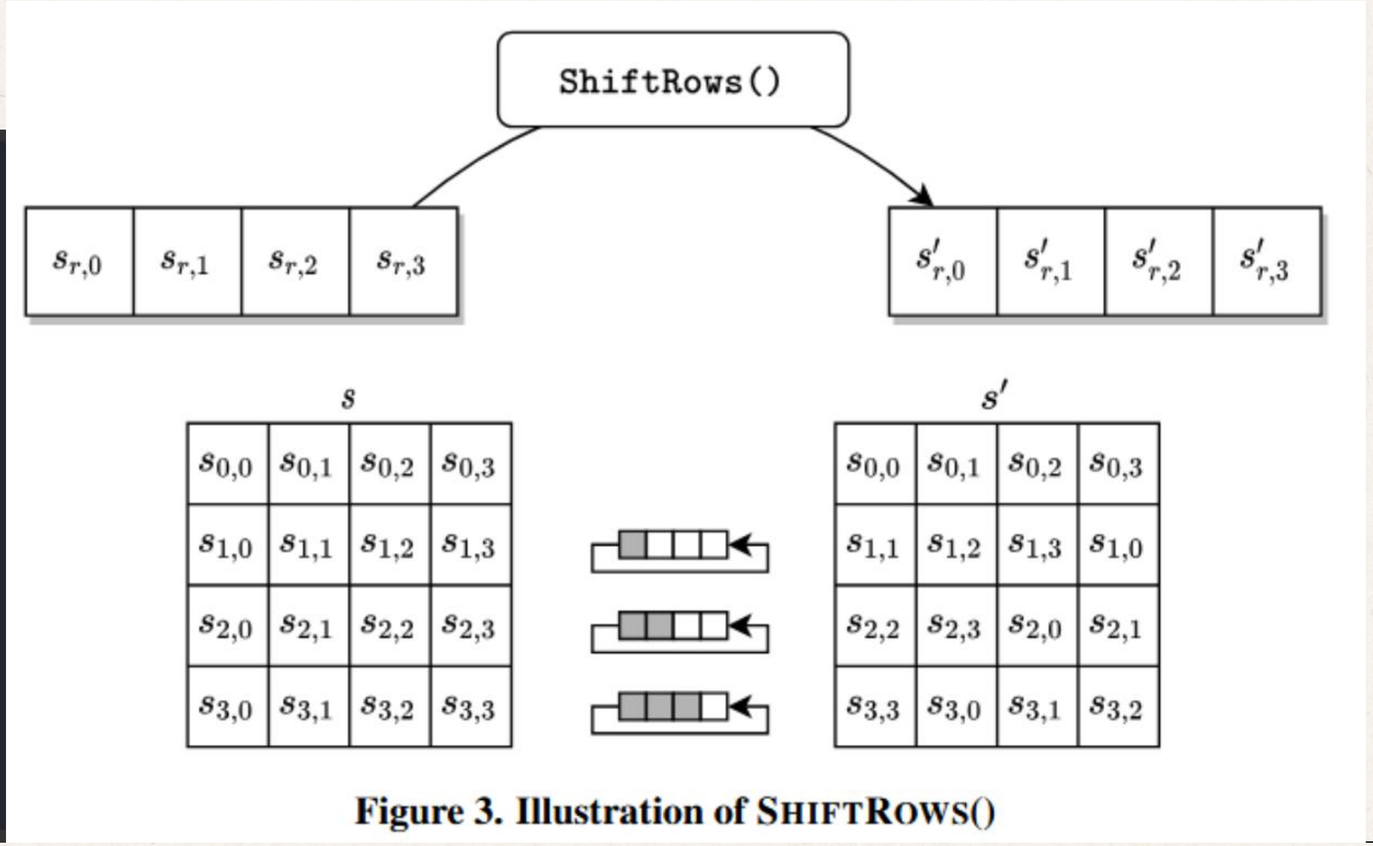


Figure 3. Illustration of SHIFTROWS()

MixColumns

```
procedure MIXCOLUMNS(state)
    for c from 0 to 3 do                    // For each column
        s0 ← (02 • state[0][c]) ⊕ (03 • state[1][c]) ⊕ state[2][c] ⊕
state[3][c]
        s1 ← state[0][c] ⊕ (02 • state[1][c]) ⊕ (03 • state[2][c]) ⊕
state[3][c]
        s2 ← state[0][c] ⊕ state[1][c] ⊕ (02 • state[2][c]) ⊕ (03 •
state[3][c])
        s3 ← (03 • state[0][c]) ⊕ state[1][c] ⊕ state[2][c] ⊕ (02 •
state[3][c])

        state[0][c] ← s0
        state[1][c] ← s1
        state[2][c] ← s2
        state[3][c] ← s3
    end for

    return state
end procedure
```

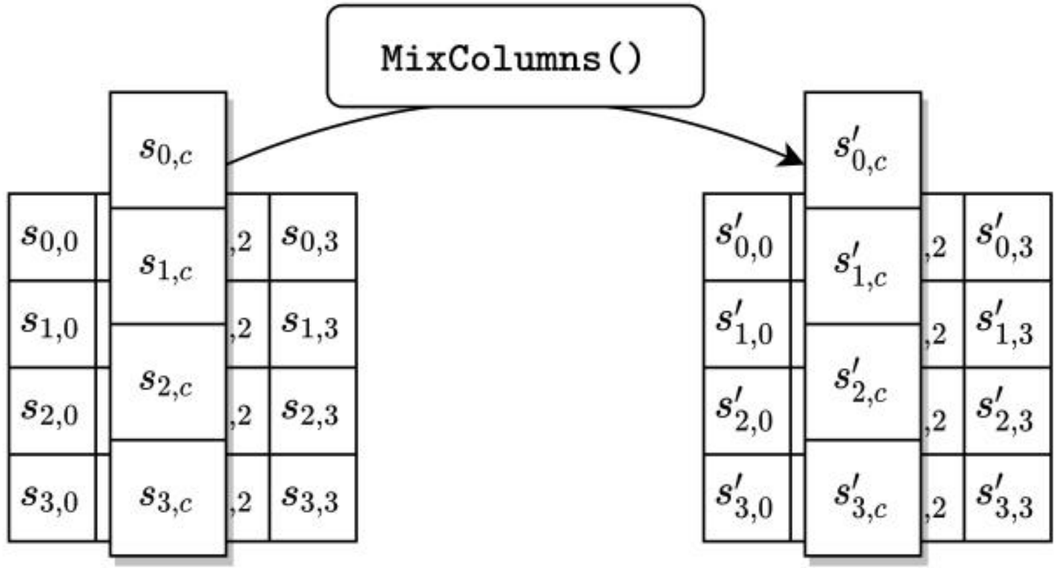
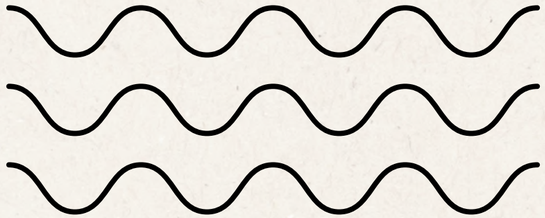


Figure 4. Illustration of MIXCOLUMNS()



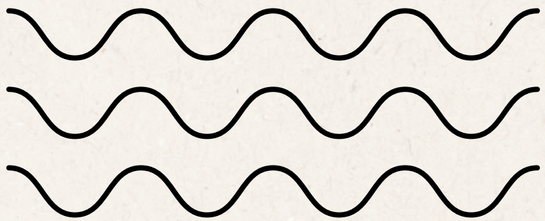
Synthesis Report

- Performance Estimates

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.453 ns	1.25 ns

Latency (cycles)		Latency (Absolute)		Interval (cycles)	
min	max	min	max	min	max
2266	2506	22.660 us	25.060 us	2265	2506

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	2	-	615	2243	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	15	-
Register	-	-	3	-	-
Total	2	0	618	2258	0
Available	2060	2800	607200	303600	0
Utilisation	~0	0	~0	~0	0



Synthesis Report

- Utilisation Estimates

Potential for Acceleration

Briefly discuss the key dates for the project.

Limitations of Sequential Execution

- Traditional AES implementations are designed for sequential execution on CPUs.
- Optimised for general-purpose instruction sets (ISA), not dataflow hardware.

- As a result:
  - Operations like SubBytes, MixColumns, and KeyExpansion execute serially.
  - Multiple nested loops, data dependencies, and conditional logic.
  - Entire encryption takes hundreds of clock cycles, even for 16-byte input.

- This makes AES an ideal candidate for hardware acceleration:
  - Highly regular structure.
  - Deterministic execution.
  - Amenable to pipelining and parallelism.

Step 1 - SubBytes

Restructured for Parallelism

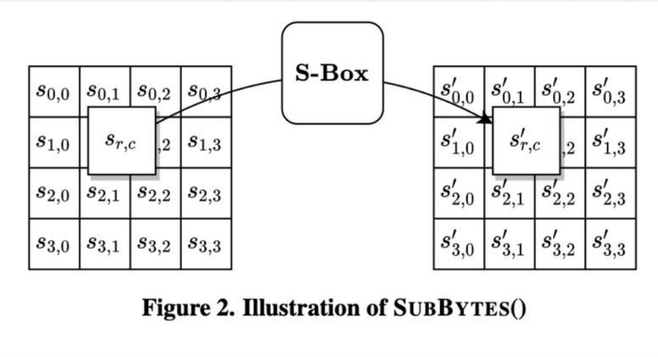


Figure 2. Illustration of SUBBYTES()

Acceleration

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

```
void SubBytes_fast(byte state[4][4]) {
#pragma HLS INLINE
#pragma HLS UNROLL
    state[0][0] ^= sbox[state[0][0]];
    state[0][1] ^= sbox[state[0][1]];
    state[0][2] ^= sbox[rstate[0][2]];
    state[0][3] ^= sbox[state[0][3]];
    state[1][0] ^= sbox[state[1][0]];
    state[1][1] ^= sbox[rstate[1][1]];
    state[1][2] ^= sbox[state[1][2]];
    state[1][3] ^= sbox[state[1][3]];
    state[2][0] ^= sbox[state[2][0]];
    state[2][1] ^= sbox[state[2][1]];
    state[2][2] ^= sbox[state[2][2]];
    state[2][3] ^= sbox[state[2][3]];
    state[3][0] ^= sbox[state[3][0]];
    state[3][1] ^= sbox[state[3][1]];
    state[3][2] ^= sbox[state[3][2]];
    state[3][3] ^= sbox[state[3][3]];
}
```

Original Implementation:

- Two nested loops (4x4 state).
- Each byte substitution happens sequentially.
- Latency: ~16 cycles.

Optimised Version:

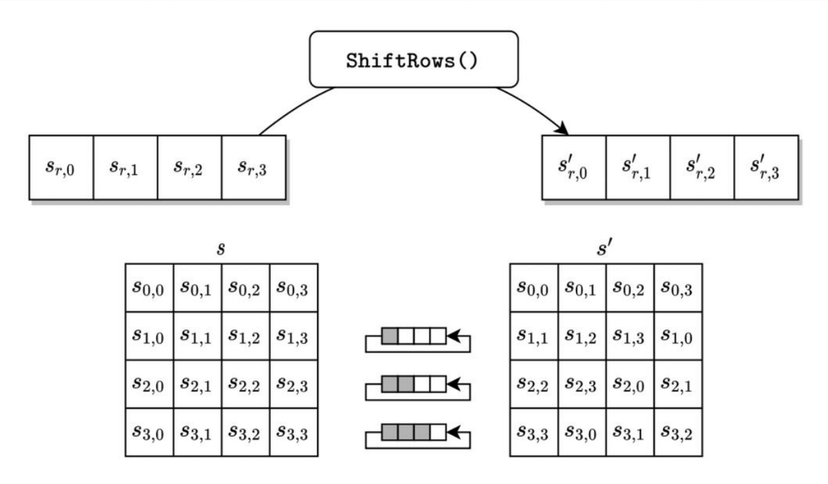
- Applied #pragma HLS UNROLL on both loops.
- Used #pragma HLS INLINE to flatten into calling function.
- Entire transformation executes in 1 clock cycle.

Impact:

- Latency reduced from 16 → 1 cycle.
- Area increase due to instantiating 16 S-boxes in parallel.

Step 2 - ShiftRows

Direct Swaps over Indexing



Acceleration

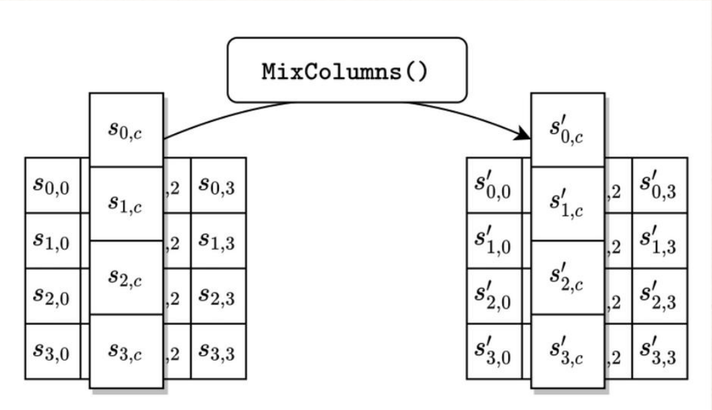
```c
// the whole thing can run in one clock cycle
void ShiftRows(byte state[4][4]) {

    byte tmp[4];

    // Row 1: shift left by 1
    for (int c = 0; c < 4; c++) tmp[c] = state[1][(c + 1) % 4];
    for (int c = 0; c < 4; c++) state[1][c] = tmp[c];

    // Row 2: shift left by 2
    for (int c = 0; c < 4; c++) tmp[c] = state[2][(c + 2) % 4];
    for (int c = 0; c < 4; c++) state[2][c] = tmp[c];

    // Row 3: shift left by 3
    for (int c = 0; c < 4; c++) tmp[c] = state[3][(c + 3) % 4];
    for (int c = 0; c < 4; c++) state[3][c] = tmp[c];
}
```

```c
void ShiftRows_fast(byte state[4][4]) {
#pragma HLS INLINE

    byte tmp;

    // Row 1: shift left by 1
    tmp = state[1][0];
    state[1][0] = state[1][1];
    state[1][1] = state[1][2];
    state[1][2] = state[1][3];
    state[1][3] = tmp;

    // Row 2: shift left by 2
    tmp = state[2][0];
    state[2][0] = state[2][2];
    state[2][2] = tmp;
    tmp = state[2][1];
    state[2][1] = state[2][3];
    state[2][3] = tmp;

    // Row 3: shift left by 3 (== right by 1)
    tmp = state[3][3];
    state[3][3] = state[3][2];
    state[3][2] = state[3][1];
    state[3][1] = state[3][0];
    state[3][0] = tmp;
}
```

- Original Implementation:
  - Performed left shifts using modulo arithmetic.
  - High overhead from array indexing and modulo operations.
  - Latency: ~10–12 cycles.
- Optimised Version:
  - Replaced loops with direct byte swaps.
  - Removed modulo operations completely.
  - Fully inlined.
- Impact:
  - Latency reduced to 1 clock cycle.
  - Negligible area overhead.

Step 3 - MixColumns

Unrolling and Fast Multiplication



Acceleration

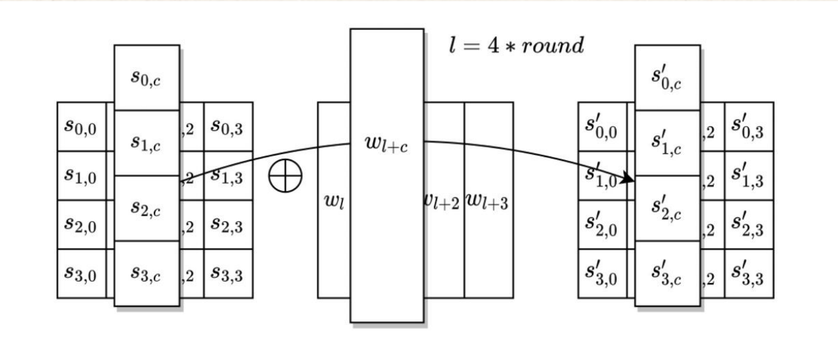
```c
void MixColumns(byte state[4][4]) {

    for (int c = 0; c < 4; c++) {
        byte a[4], res[4];
        for (int r = 0; r < 4; r++) a[r] = state[r][c];

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

        for (int r = 0; r < 4; r++) state[r][c] = res[r];
    }
}
```

```c
// one cycle for the whole thing including gmul_fast and xtime
void MixColumns_fast(byte state[4][4]) {
#pragma HLS INLINE

    for (int c = 0; c < 4; c++) {
#pragma HLS UNROLL

        byte a0 = state[0][c];
        byte a1 = state[1][c];
        byte a2 = state[2][c];
        byte a3 = state[3][c];

        state[0][c] = gmul_fast(a0, 0x02) ^ gmul_fast(a1, 0x03) ^ a2 ^ a3;
        state[1][c] = a0 ^ gmul_fast(a1, 0x02) ^ gmul_fast(a2, 0x03) ^ a3;
        state[2][c] = a0 ^ a1 ^ gmul_fast(a2, 0x02) ^ gmul_fast(a3, 0x03);
        state[3][c] = gmul_fast(a0, 0x03) ^ a1 ^ a2 ^ gmul_fast(a3, 0x02);

    }
}
```

- Original Implementation:
  - Performed left shifts using modulo arithmetic.
  - High overhead from array indexing and modulo operations.
  - Latency: ~10–12 cycles.
- Optimised Version:
  - Replaced loops with direct byte swaps.
  - Removed modulo operations completely.
  - Fully inlined.
- Impact:
  - Latency reduced to 1 clock cycle.
  - Negligible area overhead.

Step 4 - AddRoundKey

Flattening Nested Loops



Acceleration

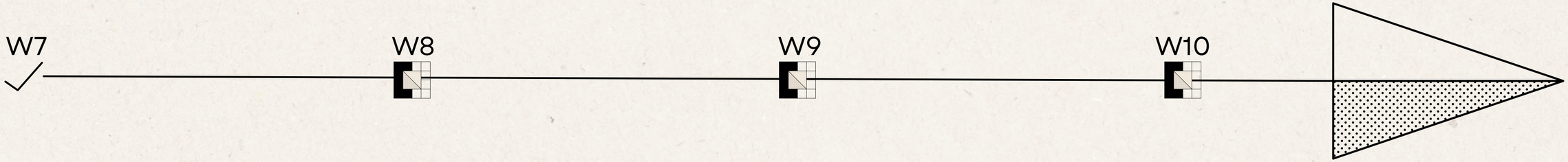
```
void AddRoundKey(byte state[4][4], const word roundKey[4]) {

    for (int c = 0; c < 4; c++) {
        word keyWord = roundKey[c];
        for (int r = 0; r < 4; r++)
            state[r][c] ^= (keyWord >> (24 - 8 * r)) & 0xFF;
    }
}
```

```
void AddRoundKey_fast(byte state[4][4], const word roundKey[4]) {
#pragma HLS INLINE
#pragma HLS UNROLL
    state[0][0] ^= (roundKey[0] >> 24) & 0xFF;
    state[0][1] ^= (roundKey[1] >> 24) & 0xFF;
    state[0][2] ^= (roundKey[2] >> 24) & 0xFF;
    state[0][3] ^= (roundKey[3] >> 24) & 0xFF;
    state[1][0] ^= (roundKey[0] >> 16) & 0xFF;
    state[1][1] ^= (roundKey[1] >> 16) & 0xFF;
    state[1][2] ^= (roundKey[2] >> 16) & 0xFF;
    state[1][3] ^= (roundKey[3] >> 16) & 0xFF;
    state[2][0] ^= (roundKey[0] >> 8) & 0xFF;
    state[2][1] ^= (roundKey[1] >> 8) & 0xFF;
    state[2][2] ^= (roundKey[2] >> 8) & 0xFF;
    state[2][3] ^= (roundKey[3] >> 8) & 0xFF;
    state[3][0] ^= (roundKey[0]) & 0xFF;
    state[3][1] ^= (roundKey[1]) & 0xFF;
    state[3][2] ^= (roundKey[2]) & 0xFF;
    state[3][3] ^= (roundKey[3]) & 0xFF;
}
```

- Original Implementation:
  - Nested loops extract bytes from 32-bit round keys using shifts and masks.
  - Latency: ~16 cycles.
- Optimised Version:
  - Unwrapped key extraction logic into explicit byte variables.
  - Removed all loops and masks.
  - Fully inlined into main cipher.
- Impact:
  - Latency reduced to 1 cycle.
  - Minimal increase in logic area.

Timeline

Briefly discuss the key dates for the project.



Milestone 1

AES encryption core w/a C testbench

AES encryption core running on Zynq ARM processor (PS)

Milestone 2

Connect the HLS core (PL) to the ARM processor on the Zynq SoC (PS)

Milestone 3

Optimize your HLS core

Milestone 4

Compare and report on findings

Risks and Contingencies

- Disjoined logic and failing code
- Time constraints
- Failure to complete tasks
- Insignificant acceleration
- Breaking the board
-Mental stress

Team Roles

Linfeng

Overall Algo and MixedColumn

Kavisha

Overall Algo and ShiftRows

Idriz

Overall Algo and ShiftRows

Alyssa

Overall Algo and AddRoundKey

Thank you!

ANY QUESTIONS?