Laboratory 5 – Loop Optimization Techniques

I. Loop Unrolling for Array Sum

Objectives

- To implement loop unrolling for performance optimization
- To implement basic array operations and pointer manipulation
- To understand memory addressing in embedded systems

Activity Summary

1. Launch Vitis 2022.2 and create workspace
2. Import hardware platform (XSA file) from Vivado
3. Create loop_unroll_sum applications
4. Implement multiple unrolling strategies (2x, 4x, 8x)
5. Program the FPGA and run applications on Basys3

Introduction

What is Loop Unrolling:

Loop unrolling is a compiler optimization technique that reduces the number of loop iterations by executing multiple loop body operations within a single iteration. Instead of processing one array element per iteration, an unrolled loop processes multiple elements (e.g., 2x, 4x, 8x) simultaneously, thereby reducing loop control overhead including counter increments, conditional checks, and branch instructions. For example, a 4x unrolled loop reduces 1000 iterations to just 250, eliminating 75% of the loop overhead. This optimization improves performance by reducing branch misprediction penalties, enabling better instruction-level parallelism (ILP), and allowing more efficient CPU pipeline utilization, as modern processors can execute independent operations in parallel. However, loop unrolling trades code size for execution speed—the unrolled code becomes larger, which can increase instruction cache pressure and code complexity. The optimal unroll factor (typically 2x-8x) depends on CPU architecture, instruction cache size, available registers, and loop body complexity. Understanding this technique is crucial for performance engineering, especially in domains like image processing, signal processing (DSP), matrix operations, cryptography, and data compression, where it typically achieves 2-5x speedups.

Ref:

Wikipedia - Loop Unrolling: https://en.wikipedia.org/wiki/Loop_unrolling

MIT OpenCourseware - 6.172 Performance Engineering: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/

Specifications

The application demonstrates loop unrolling optimization techniques through performance benchmarking on Xilinx MicroBlaze embedded platform with UART console output.

Functionality:

Basic Array Summation:

- Declares test array of 1000 integer elements
- Initializes array with pattern: arr[i] = i % 100 (values 0-99 repeating)
- Implements standard loop summing all elements (1 element per iteration)
- Records iteration count using global timer_ticks variable
- Expected sum result: 49,500
- Serves as performance baseline (1000 iterations)

2x Loop Unrolling:

- Processes 2 array elements per iteration

- Main loop: sum += arr[i] + arr[i+1] with increment i+=2
- Includes remainder loop to handle odd-sized arrays
- Records iteration count independently
- Target iterations: 500 (50% reduction from baseline)
- Verifies identical sum result as baseline

4x Loop Unrolling:

- Processes 4 array elements per iteration
- Main loop: sum += arr[i] + arr[i+1] + arr[i+2] + arr[i+3] with increment i+=4
- Includes remainder loop for non-divisible array sizes
- Records iteration count independently
- Target iterations: 250 (75% reduction from baseline)
- Verifies identical sum result as baseline

8x Loop Unrolling:

- Processes 8 array elements per iteration
- Main loop: sum += arr[i] + ... + arr[i+7] with increment i+=8
- Includes remainder loop for non-divisible array sizes
- Records iteration count independently
- Target iterations: 125 (87.5% reduction from baseline)
- Verifies identical sum result as baseline

Activities

1. Open and Export Hardware Design

Follow the procedure described in Lab 1 instructions to unzip and open **Lab 5 hardware Files** and export the hardware design files **Top.xsa (including bitstream)**.

As we mentioned in Lab 1 instructions to export the hardware platform and create the XSA file.

1. Click **File → Export → Export Hardware** (Figure 1a)
2. The Export Hardware Platform wizard opens (Figure 1b). Click **Next** to start the process.
3. In the Output window (Figure 1c), select **Include bitstream** to ensure the complete hardware implementation is included, then click **Next**.
4. In the Files window (Figure 1d), specify the XSA filename (default: "top") and export location. The system shows "The XSA will be written to: top.xsa". Click **Next**.
5. Review the summary (Figure 1e) showing that a hardware platform named 'top' will be created as top.xsa with post-implementation model and bitstream. Click **Finish** to complete the export.

The XSA file is now ready for use in Vitis IDE.

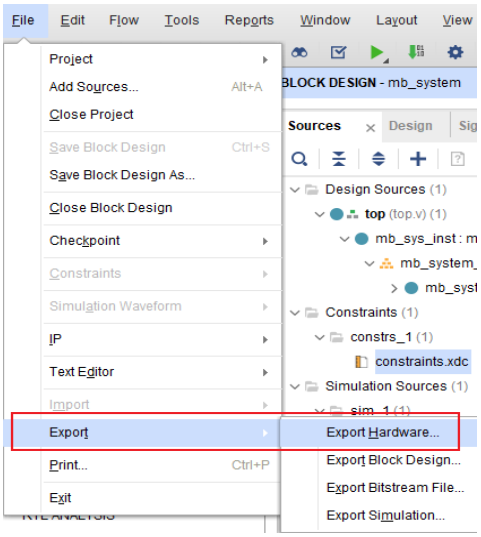

Figure 1(a)

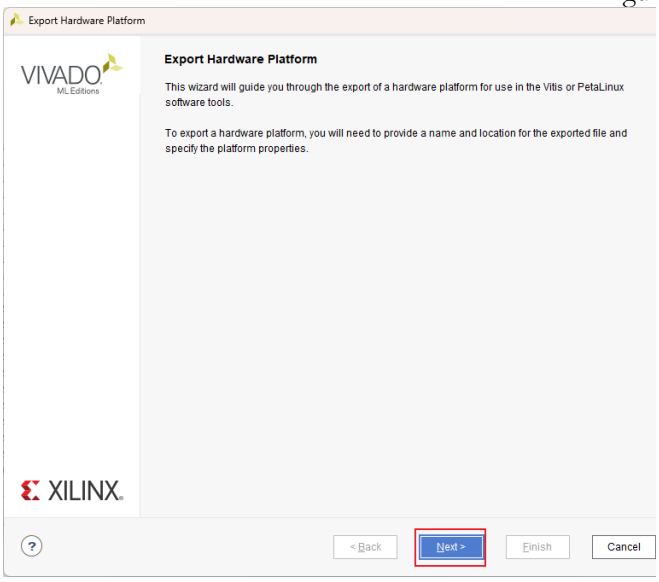

Figure 1(b)

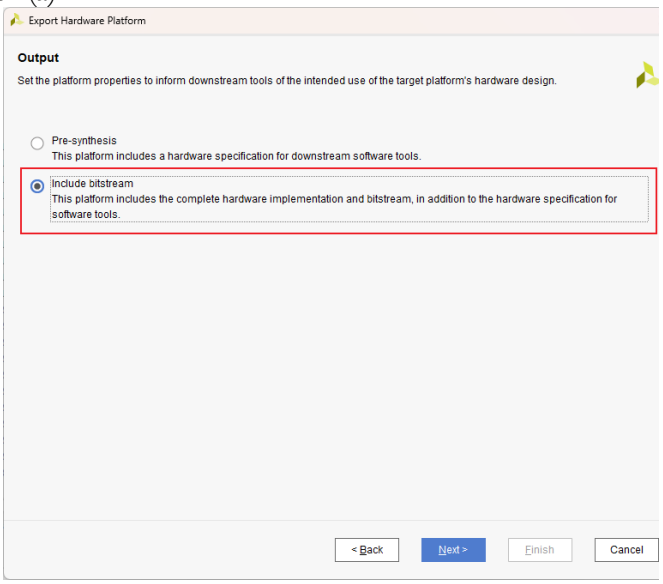

Figure 1(c)

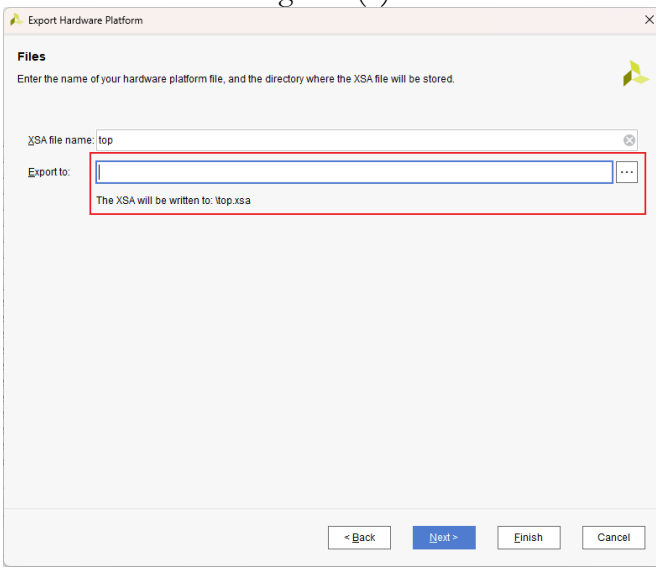

Figure 1(d)

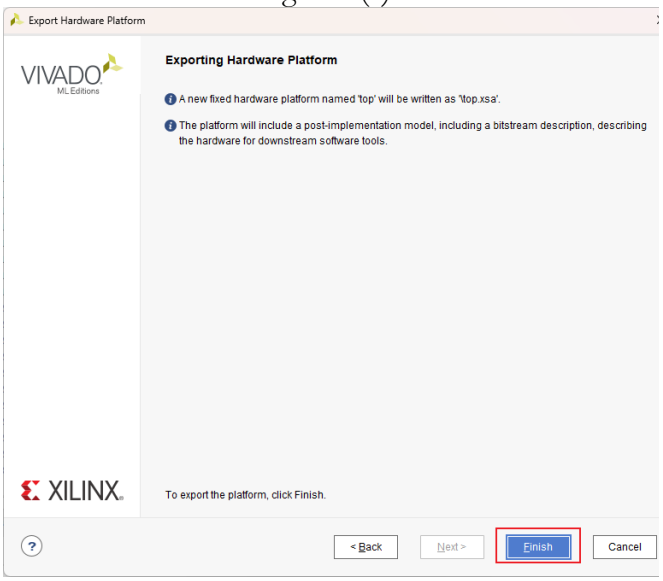

Figure 1(e)

2. **Launch Vitis**

Create the new folder in the PC, named Lab5, then open Vitis and select the file path of Lab5 as the workspace, and select **Launch**. **(Note: If you use Lab PC， please do not create the workspace on the uni-cloud)**

3. **Create a New Application Project** as demonstrated in previous lab exercises.

Based on your images, here's the revised description for creating an application project:

Create a New Application Project

Launch Vitis IDE and follow these steps to create an application project:

Step 1: Launch Vitis IDE The Vitis IDE main interface will open (Figure 2a), displaying the welcome page. In the PROJECT section, click **Create Application Project** to create an application project.

Step 2: Select Platform In the New Application Project wizard (Figure 2b), select **Create a new platform from hardware (XSA)**, then click the **Browse...** button to locate the XSA file exported earlier

from Vivado (the top.xsa file as shown in Figure 2c).

Step 3: Configure Platform On the platform selection page (Figure 2d), browse and select the top.xsa file from your project directory. The XSA file contains the hardware specification exported from Vivado.

Step 4: Configure Application Project On the Application Project Details page (Figure 2e), enter the application project name (such as " **loop_unroll_sum** "). The system will automatically create an associated system project " **loop_unroll_sum** _system" with target processor shown as "microblaze_0".

Step 5: Select Domain On the domain selection page (Figure 2f), the system displays details for the "standalone_microblaze_0" domain, including processor microblaze_0 and 32-bit architecture.

Step 6: Choose Template On the template selection page (Figure 2g), select the **Hello World** template from the embedded software development templates. This template creates a "Hello World" program written in C.

Step 7: Complete Creation Click **Finish** to complete the application project creation. Vitis IDE will create the application project and open the complete development environment, displaying the project structure in the Explorer.

Once created, you can begin developing and debugging your Lab5 embedded application.

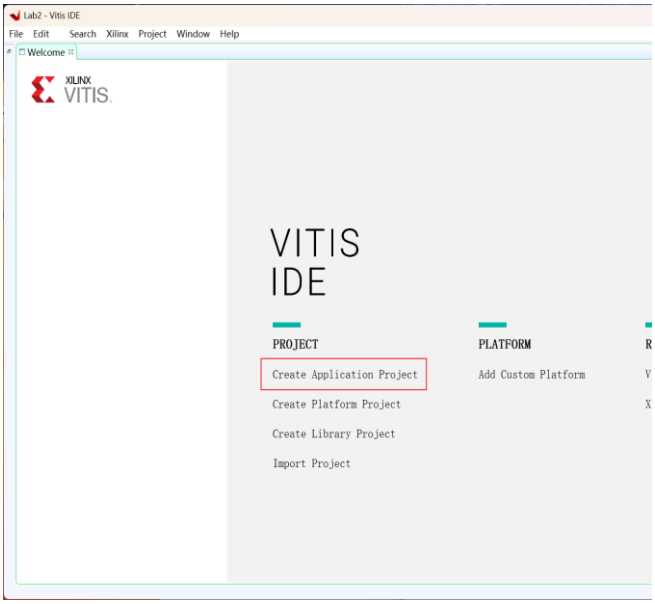

Figure 2(a)

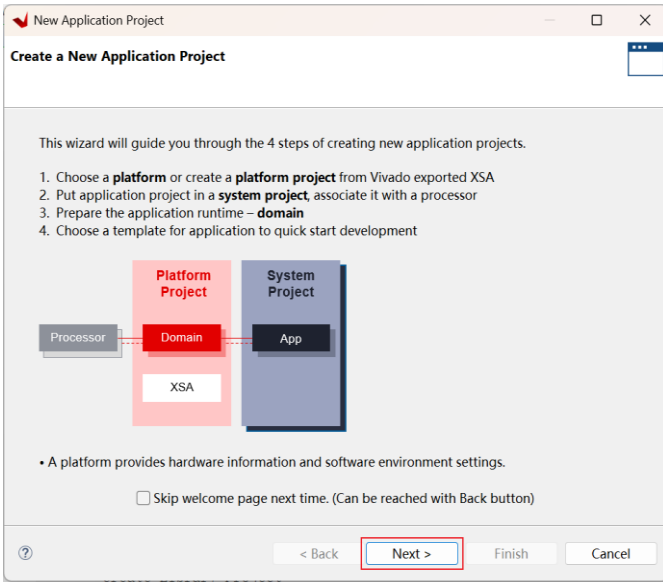

Figure 2(b)

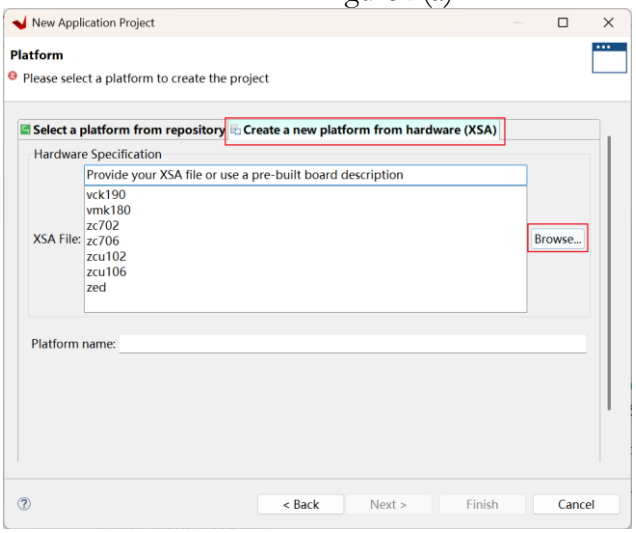

Figure 2(c)

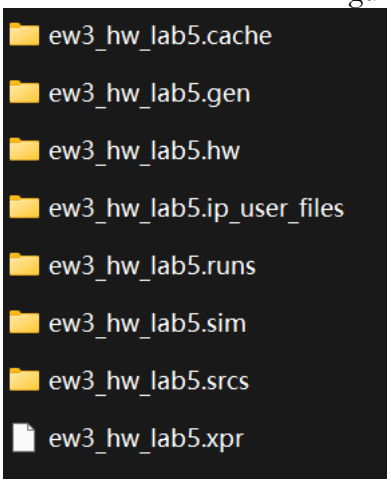

Figure 2(d)

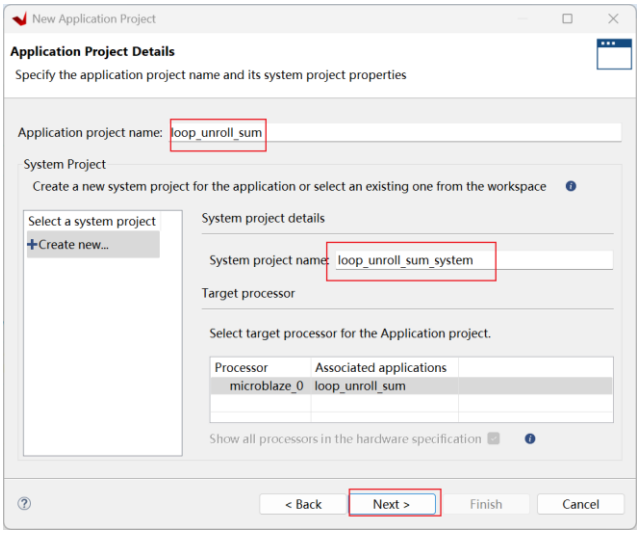

Figure 2(e)

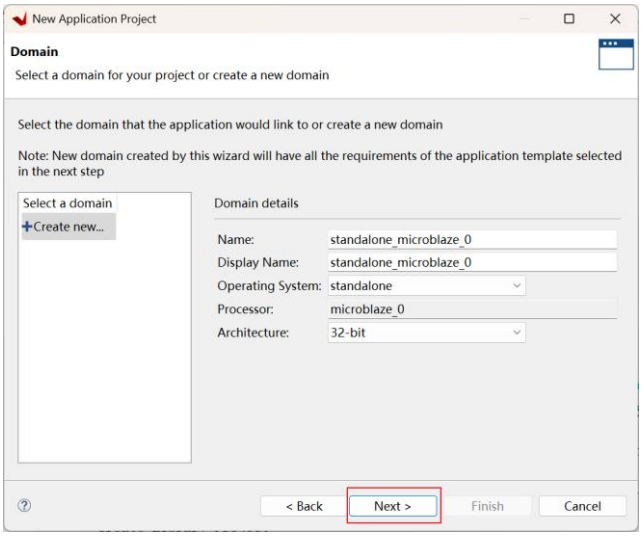

Figure 2(f)

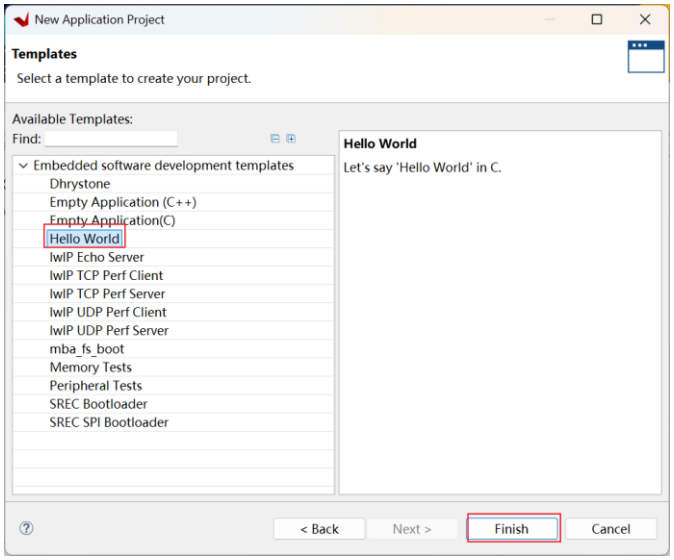

Figure 2(g)

4. Create and Rename the Source File

- As shown in the Figure 3, In **Project Explorer**, expand the **loop_unroll_sum → src** folder.
- Right-click on helloworld.c, select **Rename…**, and rename the file to loop_unroll_sum.c.

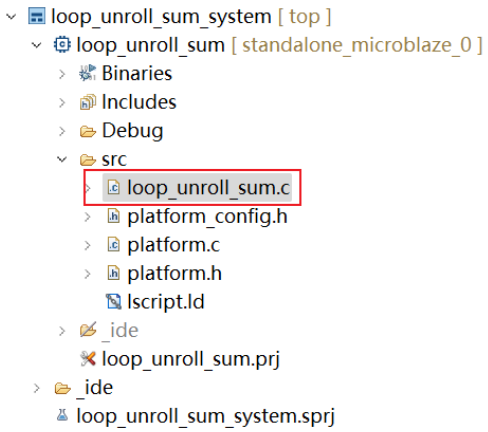

Figure 3

5. Include Required Header Files

- Open loop_unroll_sum.c.
- Remove the following two lines of code:

  print("Hello World\n\r");

  print("Successfully ran Hello World application");

- Add the code shown in Figure 4(a)-(e).

Figure 4(a): Header files and function declarations

Figure 4(b): Main function

Figure 4(c-e): Function implementations

```c
#include "xil_printf.h"    // Xilinx printf function for UART output
#include "platform.h"      // Platform initialization/cleanup functions
#include "xparameters.h"   // Hardware parameter definitions


#define TEST_SIZE 1000     // Test array size: 1000 elements

volatile unsigned int timer_ticks = 0;    // Loop iteration counter
int sum_basic(int *arr, int size);        // Basic summation (no optimization)
int sum_unrolled_2x(int *arr, int size);  // 2x loop unrolling summation
int sum_unrolled_4x(int *arr, int size);  // 4x loop unrolling summation
int sum_unrolled_8x(int *arr, int size);  // 8x loop unrolling summation
void benchmark_sum();                     // Main benchmark execution function
```

Figure 4(a)

```c
int main() {
    // ========== Platform Initialization ==========
    init_platform();                    // Initialize hardware platform

    // ========== Display Program Title ==========
    xil_printf("     LOOP UNROLLING OPTIMIZATION DEMO\r\n");

    // ========== Run Benchmark Test ==========
    benchmark_sum();                    // Execute complete benchmark suite

    // ========== Display Completion Message ==========
    xil_printf("\r\n[INFO] Test completed. Reset to run again.\r\n");

    // ========== Platform Cleanup ==========
    cleanup_platform();                 // Clean up platform resources

    return 0;                          // Program completed successfully
}
```

Figure 4(b)

```c
int sum_basic(int *arr, int size) {
    int sum = 0;                        // Initialize accumulator
    for (int i = 0; i < size; i++) {    // Standard for loop
        sum += arr[i];                  // Add current element
        timer_ticks++;                  // Increment counter to track iterations
    }
    return sum;                         // Return total sum
}

int sum_unrolled_2x(int *arr, int size) {
    int sum = 0;                        // Initialize accumulator
    int i;                              // Loop variable declared outside for remainder handling
    // Process 2 elements per iteration
    // Main loop: processes 2 elements at once
    for (i = 0; i < size - 1; i += 2) { // Increment by 2 each iteration
        sum += arr[i] + arr[i+1];       // Accumulate 2 elements at once
        timer_ticks++;                  // Count as 1 iteration (processing 2 elements)
    }
    // Handle remainder
    // Remainder loop: processes leftover elements (if size is odd)
    for (; i < size; i++) {             // Continue from previous loop index
        sum += arr[i];                  // Process final element
        timer_ticks++;
    }
    return sum;
}
```

Figure 4(c)

```c
int sum_unrolled_4x(int *arr, int size) {
    int sum = 0;
    int i;

    // Process 4 elements per iteration
    // Main loop: processes 4 elements at once
    for (i = 0; i < size - 3; i += 4) { // Increment by 4, ensure at least 4 elements available
        sum += arr[i] + arr[i+1] + arr[i+2] + arr[i+3];  // Accumulate 4 elements
        timer_ticks++;                  // Count as 1 iteration (processing 4 elements)
    }
    // Handle remainder
    // Remainder loop: processes 0-3 leftover elements
    for (; i < size; i++) {
        sum += arr[i];
        timer_ticks++;
    }
    return sum;
}

int sum_unrolled_8x(int *arr, int size) {
    int sum = 0;
    int i;
    // Process 8 elements per iteration
    // Main loop: processes 8 elements at once
    for (i = 0; i < size - 7; i += 8) { // Increment by 8, ensure at least 8 elements available
        sum += arr[i] + arr[i+1] + arr[i+2] + arr[i+3] +
               arr[i+4] + arr[i+5] + arr[i+6] + arr[i+7];  // Accumulate 8 elements
        timer_ticks++;                  // Count as 1 iteration (processing 8 elements)
    }
    // Handle remainder
    // Remainder loop: processes 0-7 leftover elements
    for (; i < size; i++) {
        sum += arr[i];
        timer_ticks++;
    }
    return sum;
}
```

Figure 4(d)

```c
void benchmark_sum() {
    // ========== Variable Declarations ==========
    static int test_array[TEST_SIZE];   // Static array to avoid stack overflow
    unsigned int basic_ops, unroll2_ops, unroll4_ops, unroll8_ops;  // Iteration counts
    int sum1, sum2, sum3, sum4;         // Computed sums from each method
    // ========== Initialize Test Data ==========
    // Initialize array with pattern: arr[i] = i % 100
    // Content: 0,1,2,...,99, 0,1,2,...,99 (repeats 10 times)
    // Expected sum: (0+1+...+99) × 10 = 4950 × 10 = 49,500
    for (int i = 0; i < TEST_SIZE; i++) {
        test_array[i] = i % 100;
    }
    // ========== Output Test Description ==========
    xil_printf("         UNDERSTANDING LOOP UNROLLING\r\n");
    xil_printf("\r\nLoop unrolling processes multiple elements per iteration\r\n");
    xil_printf("to reduce loop overhead (i++, comparison, branch).\r\n");
    xil_printf("\r\nTest: Array of %d elements (values 0-99)\r\n", TEST_SIZE);
    xil_printf("Methods: Basic, 2x, 4x, 8x unrolling\r\n");
    xil_printf("         RUNNING BENCHMARK\r\n");

    // ========== Test 1: Basic Implementation ==========
    xil_printf("[1/4] BASIC... ");      // Display progress
    timer_ticks = 0;                    // Reset counter
    sum1 = sum_basic(test_array, TEST_SIZE);  // Execute basic summation
    basic_ops = timer_ticks;            // Save iteration count
    xil_printf("Done (%u iterations)\r\n", basic_ops);  // Display result

    // ========== Test 2: 2x Unrolling ==========
    xil_printf("[2/4] 2x UNROLLED... ");
    timer_ticks = 0;                    // Reset counter
    sum2 = sum_unrolled_2x(test_array, TEST_SIZE);  // Execute 2x unrolled summation
    unroll2_ops = timer_ticks;          // Save iteration count
    xil_printf("Done (%u iterations)\r\n", unroll2_ops);

    // ========== Test 3: 4x Unrolling ==========
    xil_printf("[3/4] 4x UNROLLED... ");
    timer_ticks = 0;
    sum3 = sum_unrolled_4x(test_array, TEST_SIZE);
    unroll4_ops = timer_ticks;
    xil_printf("Done (%u iterations)\r\n", unroll4_ops);

    // ========== Test 4: 8x Unrolling ==========
    xil_printf("[4/4] 8x UNROLLED... ");
    timer_ticks = 0;
    sum4 = sum_unrolled_8x(test_array, TEST_SIZE);
    unroll8_ops = timer_ticks;
    xil_printf("Done (%u iterations)\r\n", unroll8_ops);

    // ========== Results Output ==========
    xil_printf("         RESULTS\r\n");


    // ========== Correctness Verification ==========
    // Verify that all methods produce identical results
    // This is critical: optimization should never change program correctness
    xil_printf("\r\n--- Correctness Check ---\r\n");
    xil_printf("Basic: %d, 2x: %d, 4x: %d, 8x: %d\r\n", sum1, sum2, sum3, sum4);

    // Compare all results for equality
    if (sum1 == sum2 && sum2 == sum3 && sum3 == sum4) {
        xil_printf("Status: [PASS] All methods identical!\r\n");  // Pass: all results match
    } else {
        xil_printf("Status: [FAIL] Mismatch detected!\r\n");      // Fail: results differ
    }

    // ========== Iteration Count Analysis ==========
    // Display iteration count for each method and percentage relative to baseline
    xil_printf("\r\n--- Iteration Count ---\r\n");
    xil_printf("Method          Iterations    %% of Basic\r\n");
    xil_printf("-----------------------------------------\r\n");
    xil_printf("Basic           %4u          100%%\r\n", basic_ops);  // Baseline: 100%

    // Calculate percentage: (unrolled_iterations × 100) / baseline_iterations
    xil_printf("2x Unroll       %4u          %u%%\r\n",
               unroll2_ops, (unroll2_ops * 100) / basic_ops);  // Expected: ~50%
    xil_printf("4x Unroll       %4u          %u%%\r\n",
               unroll4_ops, (unroll4_ops * 100) / basic_ops);  // Expected: ~25%
    xil_printf("8x Unroll       %4u          %u%%\r\n",
               unroll8_ops, (unroll8_ops * 100) / basic_ops);  // Expected: ~12%

    // ========== Speedup Factor Calculation ==========
    // Calculate speedup factor for each unrolling method
    // Formula: speedup = baseline_iterations / unrolled_iterations
    // Multiply by 100 to preserve 2 decimal places in integer arithmetic
    unsigned int reduction_2x = (basic_ops * 100) / unroll2_ops;  // Expected: 200 (2.00x)
    unsigned int reduction_4x = (basic_ops * 100) / unroll4_ops;  // Expected: 400 (4.00x)
    unsigned int reduction_8x = (basic_ops * 100) / unroll8_ops;  // Expected: 800 (8.00x)

    xil_printf("\r\n--- Speedup ---\r\n");
    // Display speedup in format: integer_part.decimal_part×
    // Example: 200 → 2.00×
    xil_printf("2x: %u.%02ux, 4x: %u.%02ux, 8x: %u.%02ux\r\n",
               reduction_2x/100, reduction_2x%100,    // 2x speedup factor
               reduction_4x/100, reduction_4x%100,    // 4x speedup factor
               reduction_8x/100, reduction_8x%100);   // 8x speedup factor

    xil_printf("  DEMONSTRATION COMPLETE\r\n");
}
```

Figure 4(e)

Note: This comprehensive C program for Xilinx MicroBlaze FPGAs demonstrates loop unrolling optimization techniques, showing how compiler optimizations can significantly improve performance without altering program correctness. The code implements four distinct versions of an array summation algorithm: a baseline sum_basic() function that processes one element per iteration in the traditional manner, and three progressively unrolled variants (sum_unrolled_2x(), sum_unrolled_4x(), and sum_unrolled_8x()) that process 2, 4, and 8 elements per iteration respectively by expanding the loop body to include multiple accumulation operations (e.g., sum += arr[i] + arr[i+1] + arr[i+2] + arr[i+3] for 4x unrolling). Each implementation includes careful remainder handling to process leftover elements when the array size isn't perfectly divisible by the unroll factor, and uses a volatile timer variable to count actual loop iterations rather than execution time, providing a clear metric for comparison. The benchmark initializes a 1000-element test array with values 0-99 (repeating pattern) and executes all four summation methods sequentially, demonstrating that while the basic version requires 1000 iterations with full loop control overhead (counter increment, conditional comparison, branch instruction), the unrolled versions achieve dramatic reductions—approximately 500 iterations for 2x (50% reduction), 250 for 4x (75% reduction), and 125 for 8x (87.5% reduction)—yet all produce identical sums (49,500), proving optimization preserves correctness. The program generates streamlined educational output organized into three main sections: the introduction explains the fundamental concept of loop unrolling and why it improves performance by reducing per-iteration overhead (increment operations, boundary checks, conditional branches); the benchmark execution displays real-time progress with iteration counts for each method; and the results section provides correctness verification, iteration count analysis with percentage calculations relative to baseline, and speedup factor computations showing the theoretical versus actual performance gains. The output emphasizes that optimization should never change program behavior, with all methods producing identical results while demonstrating clear performance improvements through reduced iteration counts.

6. Once you have completed writing the loop_unroll_sum.c code, Vitis IDE will display the complete development environment. The project structure shows both the system project and application project. First, you need to program the FPGA with the hardware design by right-clicking on the Pointer_basics project and selecting Program Device from the context menu(Figure 5a). This step configures the FPGA with the hardware bitstream containing the MicroBlaze processor and other peripherals. Next, to build the application, right-click on the Pointer_basics project and select Build Project from the context menu (Figure 5b). This compiles the C source code into an executable file (.elf) that can run on the MicroBlaze processor.

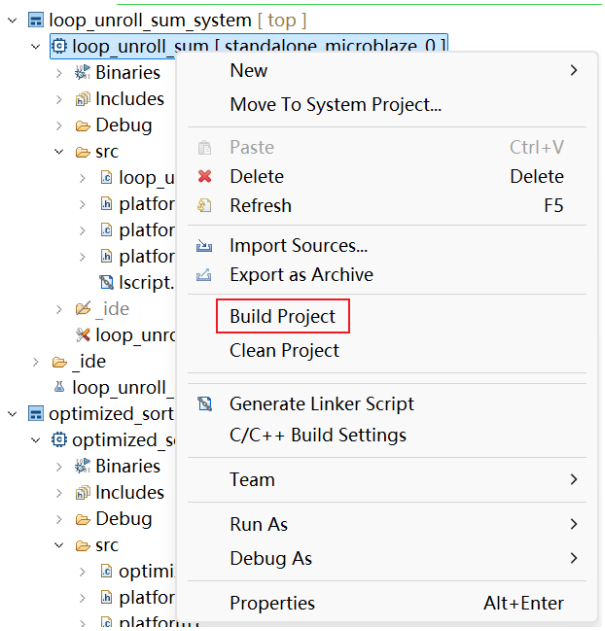

Figure 5(a)

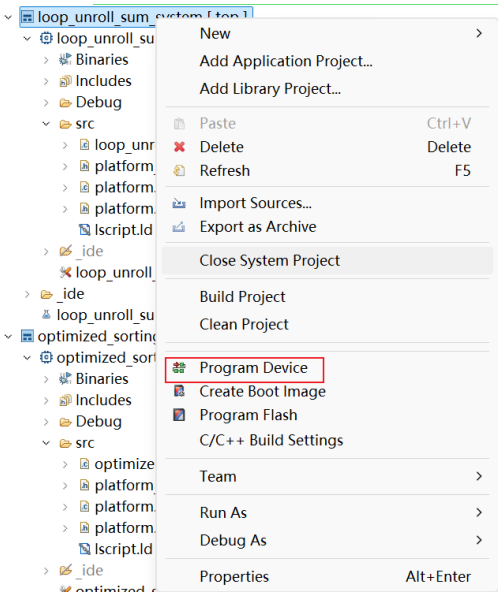

Figure 5(b)

7. To view the output and run the application, follow **Section 4 Steps 5-8** of Lab 1 instructions (also illustrated in **Figure 6(a)-(d)**): open the Terminal view, configure the Serial Terminal settings, and set up the run/debug configuration. **Figure 7(a)-(b)** shows what the Console should look like when the application is successfully run.

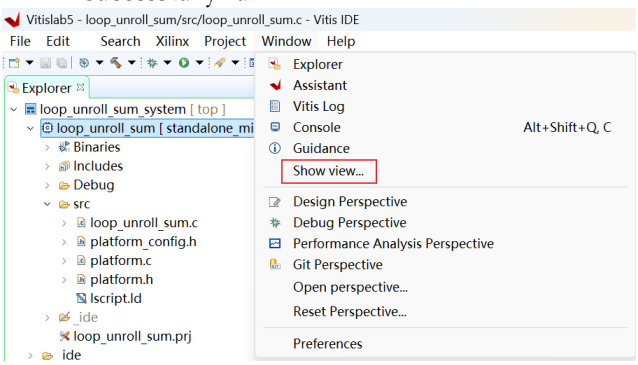

Figure 6(a)

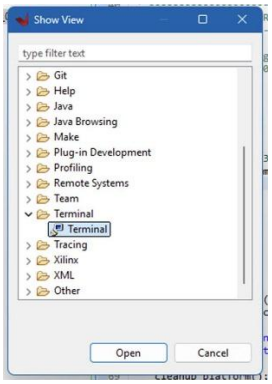

Figure 6(b)

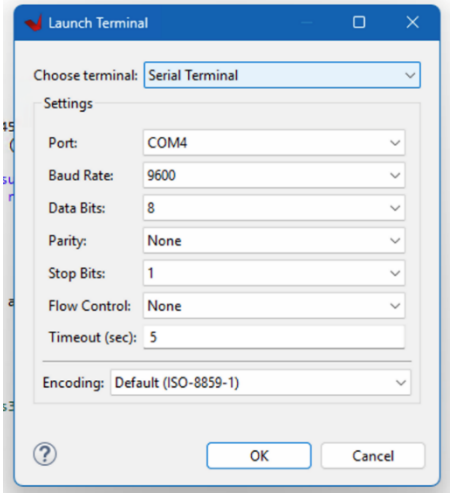

Figure 6(c)

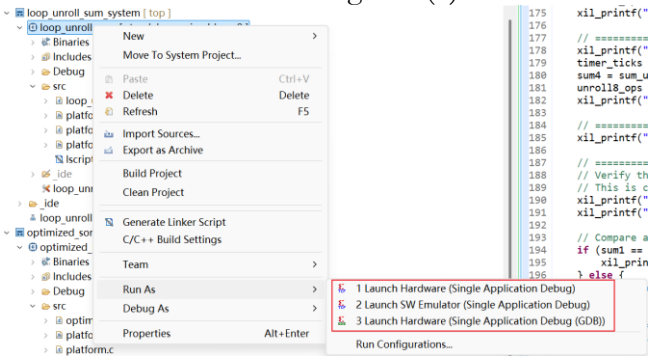

Figure 6(d)

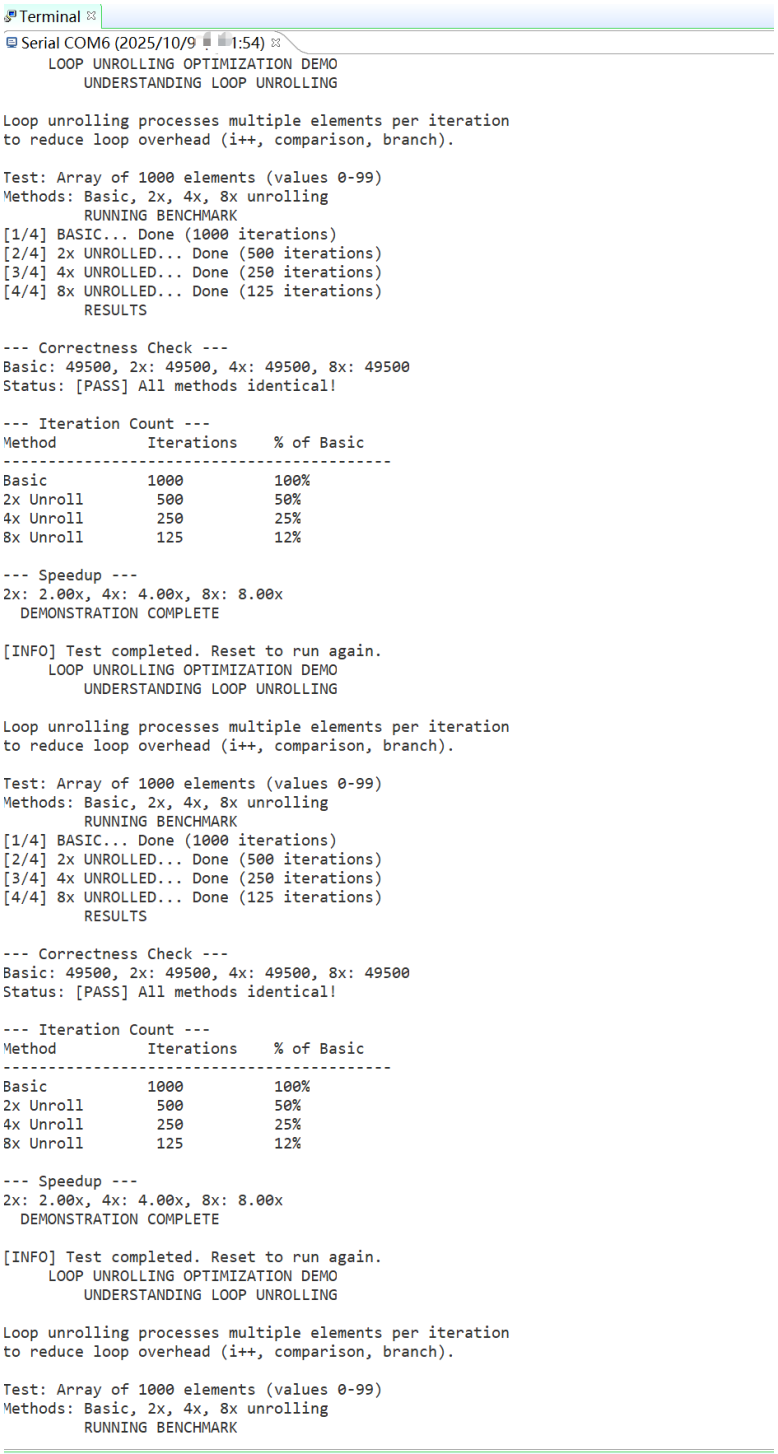
Terminal
Serial COM6 (2025/10/9 1:54)
LOOP UNROLLING OPTIMIZATION DEMO
UNDERSTANDING LOOP UNROLLING
Loop unrolling processes multiple elements per iteration
to reduce loop overhead (i++, comparison, branch).
Test: Array of 1000 elements (values 0-99)
Methods: Basic, 2x, 4x, 8x unrolling
RUNNING BENCHMARK
[1/4] BASIC... Done (1000 iterations)
[2/4] 2x UNROLLED... Done (500 iterations)
[3/4] 4x UNROLLED... Done (250 iterations)
[4/4] 8x UNROLLED... Done (125 iterations)
RESULTS
--- Correctness Check ---
Basic: 49500, 2x: 49500, 4x: 49500, 8x: 49500
Status: [PASS] All methods identical!
--- Iteration Count ---
Method Iterations % of Basic
Basic 1000 100%
2x Unroll 500 50%
4x Unroll 250 25%
8x Unroll 125 12%
--- Speedup ---
2x: 2.00x, 4x: 4.00x, 8x: 8.00x
DEMONSTRATION COMPLETE
[INFO] Test completed. Reset to run again.
LOOP UNROLLING OPTIMIZATION DEMO
UNDERSTANDING LOOP UNROLLING
Loop unrolling processes multiple elements per iteration
to reduce loop overhead (i++, comparison, branch).
Test: Array of 1000 elements (values 0-99)
Methods: Basic, 2x, 4x, 8x unrolling
RUNNING BENCHMARK
[1/4] BASIC... Done (1000 iterations)
[2/4] 2x UNROLLED... Done (500 iterations)
[3/4] 4x UNROLLED... Done (250 iterations)
[4/4] 8x UNROLLED... Done (125 iterations)
RESULTS
--- Correctness Check ---
Basic: 49500, 2x: 49500, 4x: 49500, 8x: 49500
Status: [PASS] All methods identical!
--- Iteration Count ---
Method Iterations % of Basic
Basic 1000 100%
2x Unroll 500 50%
4x Unroll 250 25%
8x Unroll 125 12%
--- Speedup ---
2x: 2.00x, 4x: 4.00x, 8x: 8.00x
DEMONSTRATION COMPLETE
[INFO] Test completed. Reset to run again.
LOOP UNROLLING OPTIMIZATION DEMO
UNDERSTANDING LOOP UNROLLING
Loop unrolling processes multiple elements per iteration
to reduce loop overhead (i++, comparison, branch).
Test: Array of 1000 elements (values 0-99)
Methods: Basic, 2x, 4x, 8x unrolling
RUNNING BENCHMARK

Figure 7(a)

```
[1/4] BASIC... Done (1000 iterations)
[2/4] 2x UNROLLED... Done (500 iterations)
[3/4] 4x UNROLLED... Done (250 iterations)
[4/4] 8x UNROLLED... Done (125 iterations)
         RESULTS

--- Correctness Check ---
Basic: 49500, 2x: 49500, 4x: 49500, 8x: 49500
Status: [PASS] All methods identical!

--- Iteration Count ---
Method          Iterations    % of Basic
-----------------------------------------
Basic           1000          100%
2x Unroll        500          50%
4x Unroll        250          25%
8x Unroll        125          12%

--- Speedup ---
2x: 2.00x, 4x: 4.00x, 8x: 8.00x
  DEMONSTRATION COMPLETE

[INFO] Test completed. Reset to run again.
     LOOP UNROLLING OPTIMIZATION DEMO
         UNDERSTANDING LOOP UNROLLING

Loop unrolling processes multiple elements per iteration
to reduce loop overhead (i++, comparison, branch).

Test: Array of 1000 elements (values 0-99)
Methods: Basic, 2x, 4x, 8x unrolling
         RUNNING BENCHMARK
[1/4] BASIC... Done (1000 iterations)
[2/4] 2x UNROLLED... Done (500 iterations)
[3/4] 4x UNROLLED... Done (250 iterations)
[4/4] 8x UNROLLED... Done (125 iterations)
         RESULTS

--- Correctness Check ---
Basic: 49500, 2x: 49500, 4x: 49500, 8x: 49500
Status: [PASS] All methods identical!

--- Iteration Count ---
Method          Iterations    % of Basic
-----------------------------------------
Basic           1000          100%
2x Unroll        500          50%
4x Unroll        250          25%
8x Unroll        125          12%

--- Speedup ---
2x: 2.00x, 4x: 4.00x, 8x: 8.00x
  DEMONSTRATION COMPLETE

[INFO] Test completed. Reset to run again.
```

Figure7(b)

8. Loop unrolling optimization offers several key advantages and disadvantages:

Advantages:

- Fewer loop control instructions are executed, reducing the overhead of counter increments, conditional checks, and branch operations
- Better CPU pipeline utilization as the processor can execute more independent operations in parallel
- Reduced branch misprediction penalties since fewer branch instructions are executed overall

Disadvantages:

- Increased code size due to replicated loop body instructions, which can impact memory usage
- Higher instruction cache pressure that may cause more cache misses and potentially offset performance gains in memory-constrained systems

Applications: This technique finds widespread use in performance-critical domains including image and video processing, digital signal processing (DSP), matrix operations, cryptography, and data compression algorithms, where it typically achieves speedups ranging from 2x to 5x depending on the specific workload and hardware architecture.

II. Pipeline Application on Array

Objectives

- To implement basic and pipelined array processing functions
- To understand the concept of overlapping operations
- To measure pipeline efficiency improvements
- To visualize pipelining stages through LED patterns

Activity Summary

1. Create a new application project called software_pipeline
2. Implement multiple pipelining strategies
3. Program the FPGA and run pipeline benchmarks
4. Analyze performance improvements from overlapping operations

Introduction

What is Pipeline:

Pipeline is a fundamental computer architecture technique that improves processor performance by dividing instruction execution into sequential stages (fetch, decode, execute, memory access, write-back), allowing multiple instructions to be processed simultaneously at different stages. Like an assembly line, while one instruction executes, another decodes, and another fetches, enabling the CPU to complete one instruction per clock cycle instead of waiting for each to finish completely. This overlapping significantly increases throughput and CPU utilization. However, pipeline efficiency can be disrupted by hazards such as branch instructions, data dependencies, and resource conflicts, requiring techniques like branch prediction and forwarding to maintain performance. Understanding pipelines is essential for optimization, as programming patterns like loop unrolling can improve pipeline efficiency by reducing branches and increasing instruction-level parallelism.

Ref

Wikipedia - Instruction Pipelining: https://en.wikipedia.org/wiki/Instruction_pipelining

Specifications

The application demonstrates software pipelining optimization techniques through correctness verification and performance comparison on Xilinx MicroBlaze embedded platform, showing how manual pipeline organization can improve throughput for array transformation operations with UART console output.

Basic Array Transformation:

- Declares input array of 500 integer elements
- Initializes with pattern: (i * 7 + 3) % 100 for varied test data
- Implements standard loop: Load → Compute (multiply by 2, add 5) → Store
- Each iteration completes fully before next begins
- No overlap between Load/Compute/Store stages
- Serves as correctness baseline

Simple Software Pipelining:

- Reorganizes same computation into 3-stage pipeline
- **Prologue**: Initializes Load and Compute for first element
- **Kernel**: Overlaps Load(i), Compute(i), and Store(i-1) operations
- **Epilogue**: Completes final Store operation
- Pipeline registers (temp0, result0) track data between stages
- Demonstrates basic overlapping execution concept

Advanced Software Pipelining:

- Extends to 4-stage pipeline with split computation
- **Stages**: Load → Compute1 (multiply) → Compute2 (add) → Store
- Three pipeline registers manage data flow
- More operations can overlap simultaneously
- Demonstrates multi-stage pipeline organization

Complex Transformation (Basic):

- More compute-intensive baseline for comparison
- **Operations**: Square → Add original → Divide by 4 → Mask to byte
- Sequential execution: all steps complete per iteration
- Higher computation density better shows pipelining benefits

Complex Transformation (Pipelined):

- 5-stage pipeline: Load → Square → Add → Divide → Store
- Each stage processes different iteration concurrently
- Maximum overlap of 5 iterations at steady state
- Four pipeline registers (stage0-stage3) manage complex data flow
- Demonstrates advanced pipelining for multi-step computations

Activities

1. As shown in Figure 8, Create a new application with the name **software_pipeline** and rename **helloworld.c** to **software_pipeline.c**.

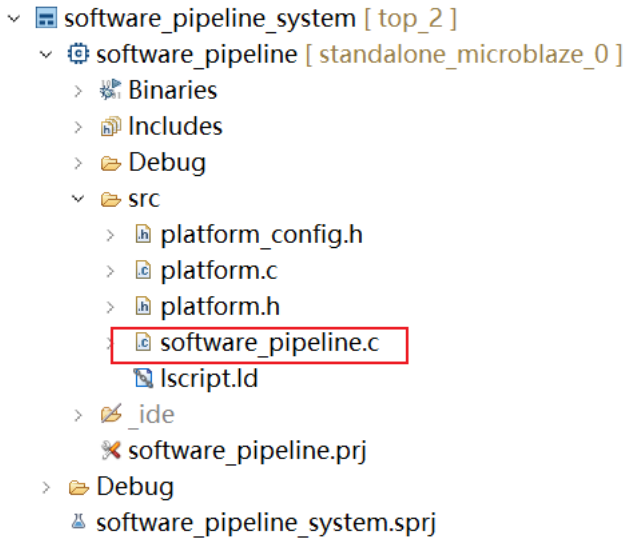

Figure 8

2. Write the code as shown in Figure 9(a)-(e).

Figure 9(a): Header files and function declarations

Figure 9(b): Main function

Figure 9(c-e): Function implementations

```c
#include "xil_printf.h"    // Xilinx printf function for UART output

#include "platform.h"      // Platform initialization/cleanup functions

#include "xparameters.h"   // Hardware parameter definitions

// ========== Macro Definitions ==========
#define ARRAY_SIZE 500     // Test array size: 500 elements

// ========== Global Variables ==========
static volatile unsigned int cycle_count = 0;  // Cycle counter

void transform_basic(int *input, int *output, int size);
// Basic sequential transformation (no pipelining)
// Baseline implementation for correctness comparison
// Each iteration: Load -> Compute -> Store (no overlap)

void transform_pipelined(int *input, int *output, int size);
// Simple 3-stage software pipeline
// Overlaps Load, Compute, and Store operations
// Demonstrates basic prologue-kernel-epilogue structure

void transform_advanced_pipeline(int *input, int *output, int size);
// Advanced 4-stage pipeline
// Splits computation into two separate stages
// More complex pipeline with better potential overlap

void complex_transform_basic(int *input, int *output, int size);
// Complex sequential computation baseline
// Multi-step operation: Square -> Add -> Divide -> Mask
// Sequential execution for comparison

void complex_transform_pipelined(int *input, int *output, int size);
// Complex 5-stage pipeline
// Overlaps all computation steps across iterations
// Demonstrates advanced pipelining for multi-step operations

void benchmark_pipelining();
// Main benchmark function
// Executes all transformations and verifies correctness
// Displays results via UART console
```

Figure 9(a)

```c
int main() {
    init_platform();                    // Initialize hardware (cache, UART)
                                        // Enables instruction and data caches
                                        // Sets up UART for console output

    xil_printf("\r\n=== Software Pipelining Demo ===\r\n");
    xil_printf("Starting automated benchmark...\r\n\r\n");

    benchmark_pipelining();             // Run all tests
                                        // Executes transformations and verifies results

    xil_printf("\r\nBenchmark Complete!\r\n");

    cleanup_platform();                 // Cleanup resources
                                        // Disables caches
    return 0;                           // Program success
}
```

Figure 9(b)

```c
void transform_basic(int *input, int *output, int size) {
    for (int i = 0; i < size; i++) {
        int temp = input[i];           // Load: Read from input array
        int result = temp * 2 + 5;     // Compute: Multiply by 2 and add 5
        output[i] = result;            // Store: Write to output array
    }
}


void transform_pipelined(int *input, int *output, int size) {
    if (size == 0) return;             // Handle empty array

    // Prologue: Fill pipeline
    // Initialize first iteration to start pipeline
    int temp0 = input[0];              // Load first element
    int result0 = temp0 * 2 + 5;       // Compute first result

    // Kernel: Steady state with overlapping operations
    // All pipeline stages active simultaneously
    for (int i = 1; i < size; i++) {
        int temp1 = input[i];          // Load next (iteration i)
        int result1 = temp1 * 2 + 5;   // Compute next (iteration i)
        output[i-1] = result0;         // Store previous (iteration i-1)

        // Advance pipeline
        // Move data forward through pipeline stages
        temp0 = temp1;
        result0 = result1;
    }

    // Epilogue: Drain pipeline
    // Complete final store operation
    output[size-1] = result0;
}

void transform_advanced_pipeline(int *input, int *output, int size) {
    if (size <= 1) {                   // Handle edge cases
        if (size == 1) output[0] = input[0] * 2 + 5;
        return;
    }

    // Prologue: Initialize all stages
    int stage0_data = input[0];        // Load first
    int stage1_data = stage0_data * 2; // Compute1 first
    int stage2_data = stage1_data + 5; // Compute2 first

    // Kernel: All stages active
    for (int i = 1; i < size; i++) {
        int new_load = input[i];       // Load iteration i
        int new_compute1 = new_load * 2;                // Compute1 iteration i
        int new_compute2 = stage1_data + 5;             // Compute2 iteration i-1
        output[i-1] = stage2_data;                      // Store iteration i-2

        // Advance pipeline
        // Each stage receives data from previous stage
        stage0_data = new_load;
        stage1_data = new_compute1;
        stage2_data = new_compute2;
    }

    // Epilogue
    // Store final result
    output[size-1] = stage2_data;
}


void complex_transform_basic(int *input, int *output, int size) {
    for (int i = 0; i < size; i++) {
        int temp = input[i];           // Load input
        int step1 = temp * temp;       // Square
        int step2 = step1 + temp;      // Add original
        int step3 = step2 >> 2;        // Divide by 4
        output[i] = step3 & 0xFF;      // Mask to byte and store
    }
}
```

Figure 9(c)

```c
void complex_transform_pipelined(int *input, int *output, int size) {
    if (size == 0) return;              // Handle empty array

    // Prologue
    // Fill all pipeline stages
    int stage0 = input[0];              // Load first
    int stage1 = stage0 * stage0;       // Square first
    int stage2 = stage1 + stage0;       // Add first
    int stage3 = stage2 >> 2;           // Divide first

    // Kernel: Five iterations overlapping
    // Each stage processes different iteration
    for (int i = 1; i < size; i++) {
        int new_load = input[i];        // Load iteration i
        int new_square = new_load * new_load;           // Square iteration i
        int new_add = stage1 + stage0;                  // Add iteration i-1
        int new_div = stage2 >> 2;                      // Divide iteration i-2
        output[i-1] = stage3 & 0xFF;                    // Store iteration i-3

        // Advance all stages
        // Data flows through all stages
        stage0 = new_load;
        stage1 = new_square;
        stage2 = new_add;
        stage3 = new_div;
    }

    // Epilogue
    // Complete final store
    output[size-1] = stage3 & 0xFF;
}
```

Figure 9(d)

```c
void benchmark_pipelining() {
    // Static arrays avoid stack overflow
    // Total memory: 5 arrays × 500 elements × 4 bytes = 10 KB
    static int input_array[ARRAY_SIZE];
    static int output_basic[ARRAY_SIZE];
    static int output_pipeline[ARRAY_SIZE];
    static int output_advanced[ARRAY_SIZE];
    static int output_complex[ARRAY_SIZE];

    // Initialize input: pattern (i * 7 + 3) % 100
    // Provides varied test data (values 0-99)
    for (int i = 0; i < ARRAY_SIZE; i++) {
        input_array[i] = (i * 7 + 3) % 100;
    }

    xil_printf("=== Software Pipelining Demo ===\r\n");
    xil_printf("Array size: %d elements\r\n", ARRAY_SIZE);
    xil_printf("\r\nRunning transformations...\r\n");

    // Execute all transformations

    transform_basic(input_array, output_basic, ARRAY_SIZE);
    xil_printf("Basic Transform: Complete\r\n");  // Sequential baseline

    transform_pipelined(input_array, output_pipeline, ARRAY_SIZE);
    xil_printf("Simple Pipeline: Complete\r\n");  // 3-stage pipeline

    transform_advanced_pipeline(input_array, output_advanced, ARRAY_SIZE);
    xil_printf("Advanced Pipeline: Complete\r\n"); // 4-stage pipeline

    complex_transform_pipelined(input_array, output_complex, ARRAY_SIZE);
    xil_printf("Complex Pipeline: Complete\r\n");  // 5-stage pipeline

    // Verify correctness
    xil_printf("\r\n=== Correctness Verification ===\r\n");
    xil_printf("Checking first 5 elements:\r\n");

    int errors = 0;  // Error counter
    for (int i = 0; i < 5 && i < ARRAY_SIZE; i++) {
        // Display values for manual inspection
        xil_printf("  [%d] Input=%d, Basic=%d, Pipeline=%d, Advanced=%d\r\n",
                   i, input_array[i], output_basic[i],
                   output_pipeline[i], output_advanced[i]);

        // Check if pipelined result matches basic result
        if (output_basic[i] != output_pipeline[i]) {
            errors++;  // Count mismatches
        }
    }

    // Display verification result
    if (errors == 0) {
        xil_printf("\r\nPipeline results MATCH basic implementation\r\n");
    } else {
        xil_printf("\r\nPipeline results DIFFER from basic implementation\r\n");
    }

   // Remind user about timing limitations
   xil_printf("\r\nNote: Install timer hardware for cycle-accurate timing\r\n");
}
```

Figure 9(e)

Note: This C program for Xilinx MicroBlaze microprocessors demonstrates software pipelining, an advanced compiler optimization technique that overlaps operations from consecutive loop iterations to maximize CPU pipeline utilization and hide instruction latency. The code implements multiple array transformation functions illustrating the progression from basic to sophisticated pipelining: transform_basic() represents conventional sequential execution where each iteration loads an input element, performs computation (multiply by 2, add 5), and stores the result before moving to the next element, creating pipeline stalls as the CPU waits for each operation to complete. In contrast, transform_pipelined() introduces fundamental software pipelining with three distinct phases—a prologue that primes the pipeline, a kernel that achieves steady-state operation by overlapping Load(i), Compute(i), and Store(i-1) operations simultaneously, and an epilogue that drains the pipeline. The transform_advanced_pipeline() extends this to a four-stage pipeline where operations from four consecutive iterations execute concurrently with careful management of stage variables acting as pipeline registers. For computationally intensive operations, complex_transform_pipelined() implements a five-stage pipeline (Load → Square → Add → Divide → Store) maintaining multiple concurrent stages, effectively keeping functional units busy rather than idle. The fundamental advantage is reducing instruction latency impact: while one iteration waits for memory load, another's computation proceeds and a third's store executes, maintaining continuous productive work across CPU execution units. The benchmark processes a 500-element array initialized with pattern data, executes all transformation variants, and performs correctness verification by comparing outputs—critical since pipelining must preserve program semantics while changing execution order. Console output displays completion status and sample results from the first five elements, verifying that despite different execution patterns, all implementations produce identical results. Although the code includes a cycle counter variable, it notes that accurate performance measurement requires dedicated hardware timer peripherals since software-based timing on embedded systems is imprecise due to cache effects and measurement overhead. The educational value lies in exposing how compilers achieve instruction-level parallelism on pipelined architectures: modern CPUs contain multiple execution units that can operate simultaneously, but sequential code often leaves these idle; software pipelining restructures loops to explicitly overlap independent operations, trading increased register pressure for reduced execution latency. This technique proves particularly crucial in DSP applications processing continuous data streams, scientific computing with matrix operations, and embedded systems maximizing throughput from limited resources.

3. Upon successful implementation of this code the output shown in Figure 10 would be observed on the console.

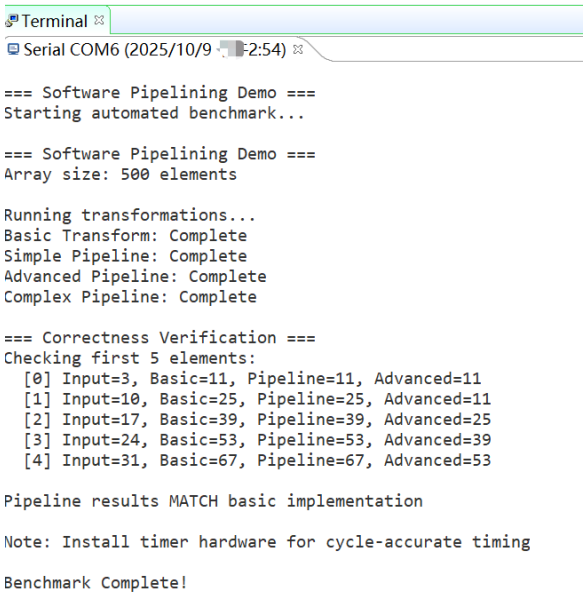

Figure 10

4. Pipeline offers several key advantages and disadvantages:

Advantages:

- Increased throughput by overlapping execution of multiple loop iterations simultaneously, allowing different operations (Load, Compute, Store) from consecutive iterations to execute concurrently
- Better resource utilization as multiple functional units (ALU, memory units, multipliers) remain busy rather than sitting idle waiting for sequential operations to complete
- Reduced instruction latency impact by hiding memory access delays and computation delays through concurrent execution of independent operations from different iterations
- Improved instruction-level parallelism (ILP) enabling the processor to maximize pipeline efficiency by keeping execution units continuously productive

Disadvantages:

- Increased code complexity with explicit prologue, kernel, and epilogue phases requiring careful management of pipeline stage variables and data flow
- Higher register pressure as pipeline stages demand additional registers to hold intermediate values, potentially exhausting available registers and forcing memory spills
- Limited applicability to loops with data dependencies or irregular control flow, where operations cannot be safely overlapped without violating program semantics
- Debugging difficulty due to overlapped execution making it harder to trace errors and inspect program state during development

Applications:

This technique finds widespread use in performance-critical domains including digital signal processing (DSP filters and FFT operations), image and video processing (convolution and codec operations), scientific computing (matrix operations and simulations), cryptography (block ciphers and hash functions), embedded systems (sensor processing and motor control), and data compression algorithms, where it typically achieves speedups ranging from 2x to 5x depending on the specific workload, loop characteristics, and hardware architecture.

References:

You can go to the lab5 reference in the learn section, or directly copy the link to your browser:

1. Wikipedia - Loop Unrolling: https://en.wikipedia.org/wiki/Loop_unrolling
2. MIT OpenCourseware - 6.172 Performance Engineering: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/
3. Wikipedia - Instruction Pipelining: https://en.wikipedia.org/wiki/Instruction_pipelining