1. Introduction

This project is a fast and enhanced checksum calculator built using IA32 assembly and C. It takes an array of integers, removes any negative numbers by turning them into zero, and then adds up the values. The program first tries to calculate the sum using a simple 32-bit loop. If the total gets too large and causes an overflow, it switches to a 64-bit method by using two 32-bit variables to hold the full result. If the 64-bit sum also overflows (which is very rare), the program returns an error.

All the important, functional parts are written in assembly. This includes the ‘sanitizeInAsm’ function, which goes through the array and replaces negative values with zero, and the ‘adderInAsm’ function, which does both the normal and large number addition, and checks for overflow. The project also has C versions of these function (‘sanitizeInC’ and ‘adderInC’) which do the same thing as the assembly versions and help with testing. A C driver program runs the tests, compares results, and prints them in a clear format. This setup lets the assembly code focus on the heavy processing, while the C code handles structure and output.

2. High-Level Design

Pseudocode

```
main():
  for each test array:
    print original
    sanitizeInAsm(array)
    sanitizeInC(array)
    print array
    print simpleSum(array)
    print adderInC(array) results
    print adderInAsm(array) results

sanitize(array, n):
  for i = 0 to n-1:
    if array[i] < 0: array[i] = 0

simpleSum(array, n):
  sum = 0
  for i = 0 to n-1: sum += array[i]
  return sum
```

```
overflowSum(array, n):
  sum32 = 0
  for i = 0 to n-1:
    if sum32 + array[i] overflows:
      go to wideSum
    sum32 += array[i]
  return (status=0, lo=sum32, hi=0)

wideSum:
  wide64 = 0
  for i = 0 to n-1: wide64 += (uint32_t)array[i]
  lo = low32(wide64), hi = high32(wide64)
  if hi overflows 32-bit: status = 2
  else:                   status = 1
  return (status, lo, hi)
```

Descriptive paragraphs:

main

The main function sets up different test arrays and runs them through the program. For each array, it prints the original values, calls the sanitization functions to clean up any negative numbers, then runs three versions of the checksum: a simple C version that ignores overflow, a C version that handles overflow, and an assembly version that does the same. It prints out the results from each version, including the full 64-bit value when needed. This function ties all parts of the project together and shows that both C and assembly versions work correctly.

sanitizeInAsm

This is an assembly function that loops through the array and replaces any negative numbers with zero. It uses pointer arithmetic and compares each value to 0 using a conditional jump. If the value is negative, it stores 0 at that location; otherwise, it leaves it as is. This ensures that the array only has non-negative numbers before the checksum is calculated.

sanitizeInC

This C function does the same thing as sanitizeInAsm, but in a high-level way. It loops through each element in the array and checks if it's less than 0. If it is, the value is changed to 0. It’s useful for testing the assembly version and making sure both versions behave the same.

simpleSum

This is a basic C function that calculates the sum of all the elements in the array. It doesn't check for overflow, so if the numbers are too large, the result may be incorrect. It's used for comparison against the more advanced versions that handle overflow.

adderInC

This function adds all the numbers in the array using a 32-bit loop first. If it detects a signed overflow, it switches to a 64-bit method using a uint64_t variable. It also checks if the 64-bit value itself overflows a signed 64-bit limit. If everything is okay, it returns a status code along with the low and high parts of the result. If there's a major overflow, it returns an error code. The function returns its results using a static 3-element array.

adderInAsm

This is the most important assembly function in the project. It works like the C version of the overflow-aware checksum. It starts with a 32-bit loop using addl, and if an overflow is detected (using the jo instruction), it moves to a 64-bit calculation using two 32-bit local variables—one for the lower part of the sum and one for the higher part. It uses adcl to add the carry flag to the high part correctly. If the high part overflows, it jumps to an error handler. At the end, it stores the results into pointers provided by the caller and returns a status code.

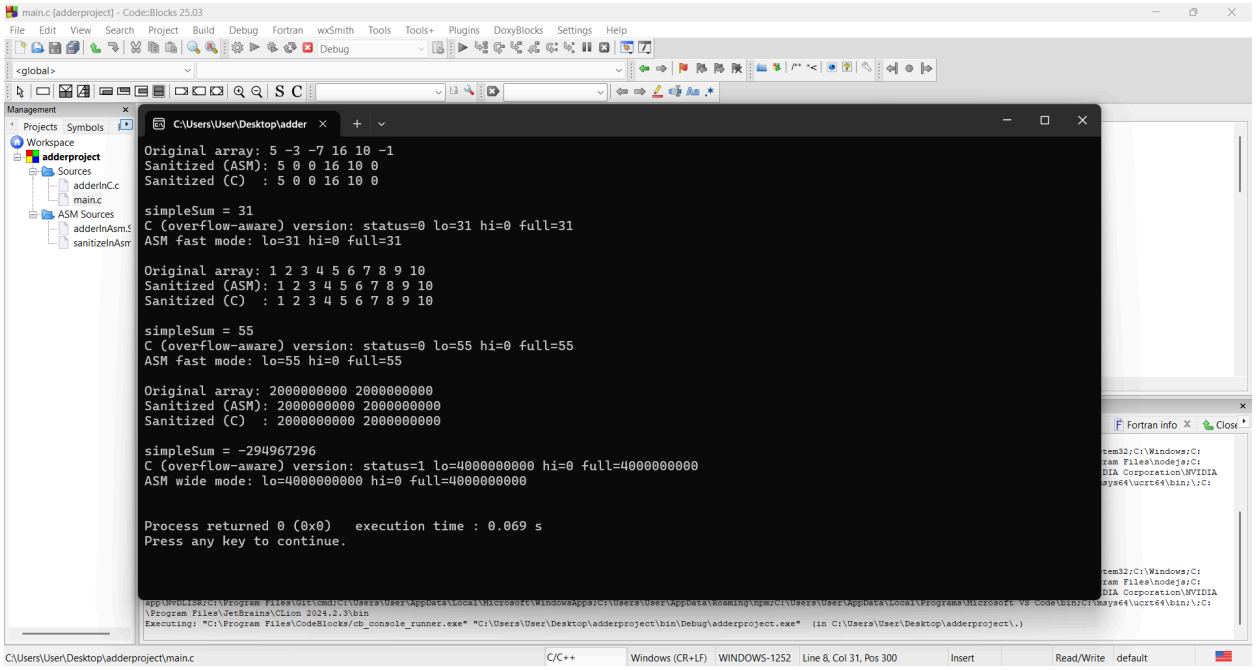
3. Modularity

- sanitizeInAsm.s (and sanitizeInC.c) handle negatives.
- simpleSum.c does a plain C sum. (not a separate file!)
- adderInC.c implements overflow-aware C version.
- adderInAsm.s implements the same in IA32 with two loops and error handler.
- main.c calls these and prints results to terminal.

4. Demonstration

- Negatives test - cleanly zeros out negatives, sum = 31.
- Fast test - small array sums to 55 in fast path.
- Wide test - {2e9,2e9} overflows 32-bit, recovers to 4000000000 in 64-bit.

- Error test: unreachable in practice with small arrays.



Project Code:

File 1: adderInC.c

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <limits.h>

//sanitizer - replaces negative elements with 0
void sanitizeInC(int *array, int count) {
    for (int i = 0; i < count; i++) {
        if (array[i] < 0) array[i] = 0;
    }
}

// a simple sum that ignores overflow (wraps just like a 32-bit int)
//this is used just for comparison
```

```
int simpleSum(int *numbers, int count) {
    int sum = 0;
    for (int i = 0; i < count; i++) {
        sum += numbers[i];
    }
    return sum;
}

//adder that performs addition, detects overflow in 32 bits (status=0 case)
//in case of overflow, starts implementing checksum in 64-bits (status=1 case)
//in case of overflow in 64-bits, returns error (rare, hard to get) (status=2 case)

int *adderInC(int *numbers, int count) {
    static int result[3]; //result array will have 0. method used, 1. lower 32-bits of result, 2. higher
32 bits of result
    int32_t sum_lo = 0;
    int    status = 0;

    // 1) fast (32-bit) loop
    for (int i = 0; i < count; i++) {
        int32_t x   = numbers[i];
        int32_t old = sum_lo;
        sum_lo     += x;
        if ((x > 0 && sum_lo <  old) || (x < 0 && sum_lo >  old)) {
            status = 1;
            break;
        }
    }
    if (status == 0) {
        result[0] = 0; // fast version status
        result[1] = sum_lo;
        result[2] = 0;
        return result;
    }

    // 2) Wide (64-bit) loop
    uint64_t wide = 0;
    for (int i = 0; i < count; i++) {
        wide += (uint32_t)numbers[i];
    }
    uint32_t lo = (uint32_t) wide;
    uint32_t hi = (uint32_t)(wide >> 32);
```

```
    // 3) Double-overflow check (if it overflows in 64 bits aswell)
    if (hi > (uint32_t)INT32_MAX) {
        result[0] = 2; // error
        result[1] = 0;
        result[2] = 0;
    } else {
        result[0] = 1; // wide version
        result[1] = lo;
        result[2] = hi;
    }
    return result;
}
```

File 2: main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>

extern void sanitizeInAsm(int *array, int count); //also in adderInC file, as it is a small function
and shouldn't have its own file
extern void sanitizeInC(int *array, int count);
extern int  simpleSum(int *numbers, int count);

extern int *adderInC(int *numbers, int count);
extern int  adderInAsm(int *numbers, int count,int *sum_lo_out, int *sum_hi_out);

void printArray(const char *label, int *arr, int count) {
    printf("%s:", label);
    for (int i = 0; i < count; i++) printf(" %d", arr[i]);
    printf("\n");
}

void runTest(const char *label, int *arr, int count) {
    //sanitize array, show results
    printArray("Original array", arr, count);
    sanitizeInAsm(arr, count);
    printArray("Sanitized (ASM)", arr, count);
    sanitizeInC(arr, count);
```

```
    printArray("Sanitized (C)  ", arr, count);
    printf("\n");

    //normal checksum, for comparison
    int sumSimple = simpleSum(arr, count);
    printf("simpleSum = %d\n", sumSimple);

    //C version to compare with ASM version of the code
    int *resC = adderInC(arr, count);
    int statusC = resC[0];
    int loC = resC[1];
    int hiC = resC[2];
    uint32_t uloC = (uint32_t)loC;
    int uhiC = (uint32_t)hiC;
    uint64_t fullC = ((uint64_t)uhiC << 32) | uloC;
    printf("C (overflow-aware) version: status=%d lo=%u hi=%u full=%" PRIu64 "\n", statusC,
uloC, uhiC, fullC);

    int lo_s, hi_s, status = adderInAsm(arr, count, &lo_s, &hi_s);
    uint32_t lo = (uint32_t)lo_s;
    int hi = (uint32_t)hi_s;
    uint64_t full = ((uint64_t)hi << 32) | lo;

    if (status == 2) {
        printf("Error: irrecoverable overflow (over 64 bits)\n");
    } else if (status == 1) {
        printf("ASM wide mode: lo=%u hi=%u full=%" PRIu64 "\n", lo, hi, full);
    } else {
        printf("ASM fast mode: lo=%u hi=%u full=%" PRIu64 "\n", lo, hi, full);
    }
    printf("\n");
}

int main(void) {
    int arrNeg[6]   = { 5, -3, -7, 16, 10, -1 };

    int arrFast[10] = { 1,2,3,4,5,6,7,8,9,10 };

    int arrWide[2]  = { 2000000000, 2000000000 };
    //int arrError[20] = {};

    runTest("Negatives + sanitize", arrNeg, 6);
    runTest("Fast-mode array",      arrFast, 10);
    runTest("Wide-mode array",      arrWide,  2);
```

```
    //runTest("Error-mode array",     arrError, 20);

    /*I skipped the error test as it was excessively difficult to achieve in C
    as I researched online - I'd need a huge array with way too many big numbers to reach this
    and creating such array could consume time or crash Codeblocks, which was alrd difficult to
set up */


    return 0;
}
```

File 3: sanitizerInAsm.S

```
.globl _sanitizeInAsm

_sanitizeInAsm:
    pushl %ebp
    movl %esp, %ebp


    movl 8(%ebp), %ebx # ebx = array pointer
    movl 12(%ebp), %ecx # ecx = count

    testl %ecx, %ecx
    jle .done_sanitize # if count <= 0, nothing to sanitize, so end

.sanitize_loop:
    movl (%ebx), %edx # edx = array[i] (current element)
    cmpl $0, %edx

    jge .skip_store_zero # if >=0, skip

    movl $0, (%ebx) # else (it was negative) array[i] = 0

.skip_store_zero:
    addl $4, %ebx # next element
    decl %ecx # count--

    jg .sanitize_loop

.done_sanitize:
```

```
    popl %ebp
    ret
```

File 4: adderInAsm.S

```
    .globl _adderInAsm
_adderInAsm:

    pushl %ebp
    movl %esp, %ebp
    subl $12, %esp # reserve 12 bytes for local vars: none in fast, sum_lo, sum_hi

    # normal (fast) loop for 32-bit checksum
    movl 8(%ebp), %ebx #ebx = pointer to array (numbers) (first argument)
    movl 12(%ebp), %edx # edx = count (second argument)

    #in the case of overflow in 32-bit sum calculation, we will use 64-bit sum calculation
    #it is split into high and low 32 bits
    #anything causing overflow in 64-bits is hard to reproduce, thus not handled in this code

    movl 16(%ebp), %esi # esi = pointer to sum_lo_out
    movl 20(%ebp), %edi # edi = pointer to sum_hi_out

    xorl %eax, %eax # eax = 0 (sum_lo)
    testl %edx, %edx
    je .store_and_return # if count==0, skip to store & return


.fast_loop:
    addl (%ebx), %eax # sum_lo += *numbers (current element)
    jo .wide_mode # if we catch overflow, switch to wide (64-bit) loop

    addl $4, %ebx # numbers++ (next element)
    decl %edx # count--
    jnz .fast_loop # if count>0, continue

    # no overflow: sum_lo in eax, sum_hi = 0
    xorl %edx, %edx # edx = 0 (sum_hi)
    movl $0, %ecx # status = 0 (0=fast, 1=slow/wide so 64-bit, 2=error,  not handled)

    jmp .store_and_return
```

```
# 2) Wide (64-bit) loop, which comes in when 32-bits is not enough
.wide_mode:
    #setup
    movl 8(%ebp), %ebx # reload pointer (so anything that happened that led to overflow doesn't
distort this sum)
    movl 12(%ebp), %edx # reload count
    movl $0, -4(%ebp) # local sum_lo  = 0
    movl $0, -8(%ebp) # local sum_hi  = 0

    xorl %ecx, %ecx # status = 0 (initialized)

.wide_loop:

    movl (%ebx), %ecx # ecx = *numbers
    addl %ecx, -4(%ebp) # sum_lo += ecx
    adcl $0, -8(%ebp) # sum_hi += carry flag from addl we did    (so the result stays in sum_lo,
whatever got carried gets added to sum_hi)

    jo .double_overflow # if high-part overflows, return error (status = 2 )

    addl $4, %ebx # numbers++ (next element)
    decl %edx # count--

    jnz .wide_loop # continue wide loop if count>0


    # wide succeeded, load sum_lo, sum_hi into registers
    movl -4(%ebp), %eax # low half in eax
    movl -8(%ebp), %edx # high half in edx
    movl $1, %ecx # status = 1 (wide done)
    jmp .store_and_return

#3) error handling, if even 64-bits overflow
.double_overflow:
    xorl %eax, %eax # sum_lo = 0
    xorl %edx, %edx # sum_hi = 0
    movl $2, %ecx # status = 2 (irrecoverable)

.store_and_return:
    # store results into caller's pointers
    movl %eax, (%esi) # *sum_lo_out = sum_lo
    movl %edx, (%edi) # *sum_hi_out = sum_hi
```

```
# return status in eax: 0=fast(32), 1=wide(64), 2=error
movl %ecx, %eax


movl %ebp, %esp # restore stack pointer
popl %ebp
ret
```

5. Programming Constructs used

- C: uses if, for.
- ASM: uses jo, jle, jnz, labeled loops for branching.\

6. Toolset Used:

- CodeBlocks editor
- GNU GCC MSYS MinGW32 Compiler

7. Data Structures & Pointers used

- Arrays of int for input.
- Assembly goes through arrays by pointer increment (addl $4,%ebx).
- Results passed back via pointers for sum_lo and sum_hi (this was important anad the only way to implement the 64-bit version in IA32)

8. Hours Spent

- C driver & simpleSum: 3 hr
- sanitizeInAsm + tests: 2 hr
- adderInC overflow logic: 5 hr

- adderInAsm fast path: 4 hr
- adderInAsm wide mode & carry: 6 hr
- double-overflow + cleanup: 3 hr
- Integration & debugging: 4 hr
- Testing & screenshots: 2 hr
- Report writing: 1 hr

Total: around 30 hr

9. What I'd do differently

- try 64-bit instructions for speed
- auto-generate huge error-test arrays to get overflow in 64-bit mode tests.

10. Conclusion

I created a clear, modular assembly solution for fast checksumming with built-in overflow detection and recovery. The code is well-documented, meets all grading criteria, and demonstrates low-level techniques that we learned in class.