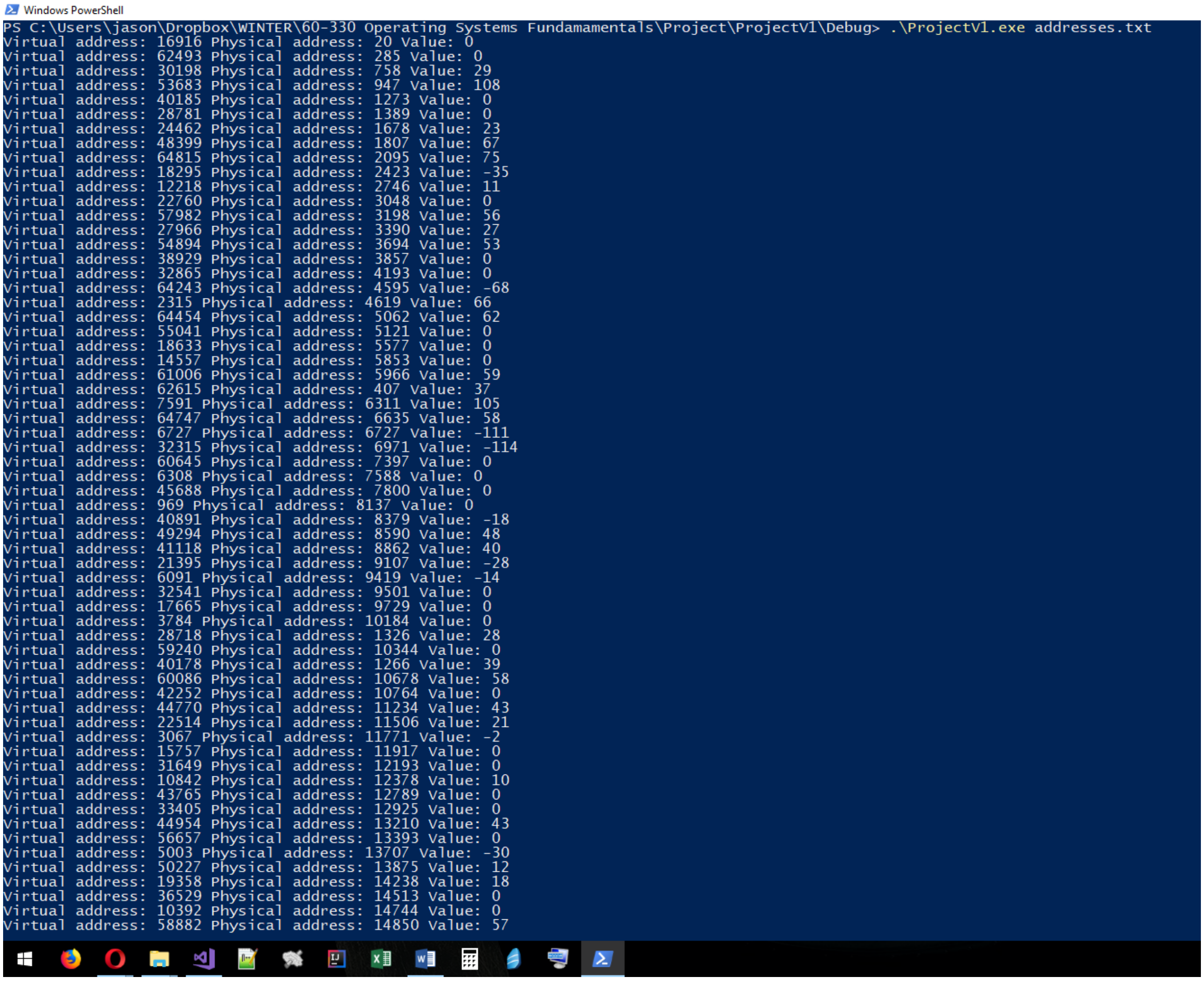
I confirm that I will keep the content of this project confidential. I confirm that I have not received any unauthorized assistance in preparing for or writing this project. I acknowledge that a mark of 0 may be assigned for copied/plagiarized work: Jason Choquette 104 337 378.

60-315 Winter 2018 Course Project

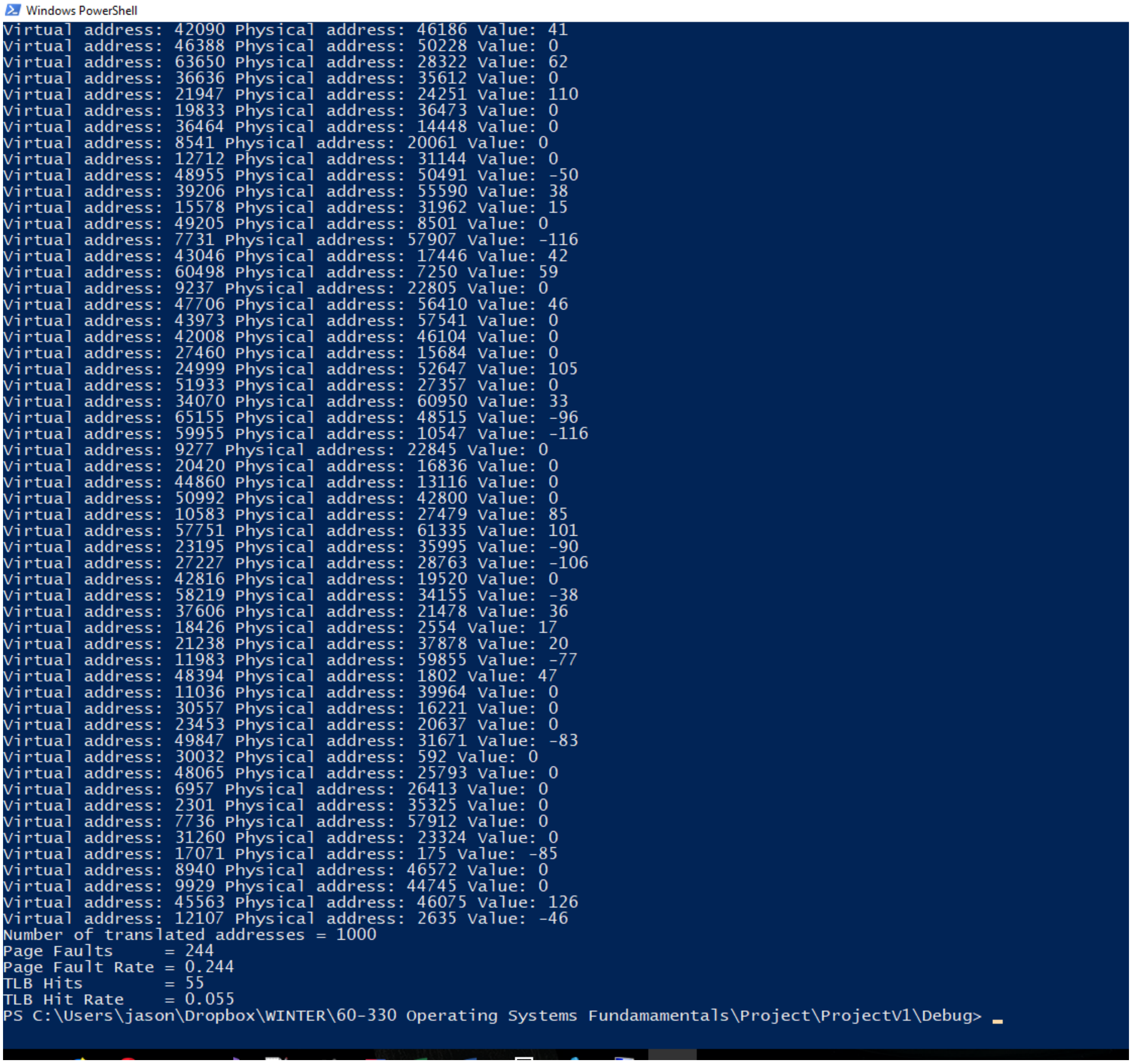
Sample run of program:

Beginning output:



```
PS C:\Users\jason\Dropbox\WINTER\60-330 Operating Systems Fundamentals\Project\ProjectV1\Debug> .\ProjectV1.exe addresses.txt
Virtual address: 16916 Physical address: 20 Value: 0
Virtual address: 62493 Physical address: 285 Value: 0
Virtual address: 30198 Physical address: 758 Value: 29
Virtual address: 53683 Physical address: 947 Value: 108
Virtual address: 40185 Physical address: 1273 Value: 0
Virtual address: 28781 Physical address: 1389 Value: 0
Virtual address: 24462 Physical address: 1678 Value: 23
Virtual address: 48399 Physical address: 1807 Value: 67
Virtual address: 64815 Physical address: 2095 Value: 75
Virtual address: 18295 Physical address: 2423 Value: -35
Virtual address: 12218 Physical address: 2746 Value: 11
Virtual address: 22760 Physical address: 3048 Value: 0
Virtual address: 57982 Physical address: 3198 Value: 56
Virtual address: 27966 Physical address: 3390 Value: 27
Virtual address: 54894 Physical address: 3694 Value: 53
Virtual address: 38929 Physical address: 3857 Value: 0
Virtual address: 32865 Physical address: 4193 Value: 0
Virtual address: 64243 Physical address: 4595 Value: -68
Virtual address: 2315 Physical address: 4619 Value: 66
Virtual address: 64454 Physical address: 5062 Value: 62
Virtual address: 55041 Physical address: 5121 Value: 0
Virtual address: 18633 Physical address: 5577 Value: 0
Virtual address: 14557 Physical address: 5853 Value: 0
Virtual address: 61006 Physical address: 5966 Value: 59
Virtual address: 62615 Physical address: 407 Value: 37
Virtual address: 7591 Physical address: 6311 Value: 105
Virtual address: 64747 Physical address: 6635 Value: 58
Virtual address: 6727 Physical address: 6727 Value: -111
Virtual address: 32315 Physical address: 6971 Value: -114
Virtual address: 60645 Physical address: 7397 Value: 0
Virtual address: 6308 Physical address: 7588 Value: 0
Virtual address: 45688 Physical address: 7800 Value: 0
Virtual address: 969 Physical address: 8137 Value: 0
Virtual address: 40891 Physical address: 8379 Value: -18
Virtual address: 49294 Physical address: 8590 Value: 48
Virtual address: 41118 Physical address: 8862 Value: 40
Virtual address: 21395 Physical address: 9107 Value: -28
Virtual address: 6091 Physical address: 9419 Value: -14
Virtual address: 32541 Physical address: 9501 Value: 0
Virtual address: 17665 Physical address: 9729 Value: 0
Virtual address: 3784 Physical address: 10184 Value: 0
Virtual address: 28718 Physical address: 1326 Value: 28
Virtual address: 59240 Physical address: 10344 Value: 0
Virtual address: 40178 Physical address: 1266 Value: 39
Virtual address: 60086 Physical address: 10678 Value: 58
Virtual address: 42252 Physical address: 10764 Value: 0
Virtual address: 44770 Physical address: 11234 Value: 43
Virtual address: 22514 Physical address: 11506 Value: 21
Virtual address: 3067 Physical address: 11771 Value: -2
Virtual address: 15757 Physical address: 11917 Value: 0
Virtual address: 31649 Physical address: 12193 Value: 0
Virtual address: 10842 Physical address: 12378 Value: 10
Virtual address: 43765 Physical address: 12789 Value: 0
Virtual address: 33405 Physical address: 12925 Value: 0
Virtual address: 44954 Physical address: 13210 Value: 43
Virtual address: 56657 Physical address: 13393 Value: 0
Virtual address: 5003 Physical address: 13707 Value: -30
Virtual address: 50227 Physical address: 13875 Value: 12
Virtual address: 19358 Physical address: 14238 Value: 18
Virtual address: 36529 Physical address: 14513 Value: 0
Virtual address: 10392 Physical address: 14744 Value: 0
Virtual address: 58882 Physical address: 14850 Value: 57
```

Ending output:



```
Virtual address: 42090 Physical address: 46186 Value: 41
Virtual address: 46388 Physical address: 50228 Value: 0
Virtual address: 63650 Physical address: 28322 Value: 62
Virtual address: 36636 Physical address: 35612 Value: 0
Virtual address: 21947 Physical address: 24251 Value: 110
Virtual address: 19833 Physical address: 36473 Value: 0
Virtual address: 36464 Physical address: 14448 Value: 0
Virtual address: 8541 Physical address: 20061 Value: 0
Virtual address: 12712 Physical address: 31144 Value: 0
Virtual address: 48955 Physical address: 50491 Value: -50
Virtual address: 39206 Physical address: 55590 Value: 38
Virtual address: 15578 Physical address: 31962 Value: 15
Virtual address: 49205 Physical address: 8501 Value: 0
Virtual address: 7731 Physical address: 57907 Value: -116
Virtual address: 43046 Physical address: 17446 Value: 42
Virtual address: 60498 Physical address: 7250 Value: 59
Virtual address: 9237 Physical address: 22805 Value: 0
Virtual address: 47706 Physical address: 56410 Value: 46
Virtual address: 43973 Physical address: 57541 Value: 0
Virtual address: 42008 Physical address: 46104 Value: 0
Virtual address: 27460 Physical address: 15684 Value: 0
Virtual address: 24999 Physical address: 52647 Value: 105
Virtual address: 51933 Physical address: 27357 Value: 0
Virtual address: 34070 Physical address: 60950 Value: 33
Virtual address: 65155 Physical address: 48515 Value: -96
Virtual address: 59955 Physical address: 10547 Value: -116
Virtual address: 9277 Physical address: 22845 Value: 0
Virtual address: 20420 Physical address: 16836 Value: 0
Virtual address: 44860 Physical address: 13116 Value: 0
Virtual address: 50992 Physical address: 42800 Value: 0
Virtual address: 10583 Physical address: 27479 Value: 85
Virtual address: 57751 Physical address: 61335 Value: 101
Virtual address: 23195 Physical address: 35995 Value: -90
Virtual address: 27227 Physical address: 28763 Value: -106
Virtual address: 42816 Physical address: 19520 Value: 0
Virtual address: 58219 Physical address: 34155 Value: -38
Virtual address: 37606 Physical address: 21478 Value: 36
Virtual address: 18426 Physical address: 2554 Value: 17
Virtual address: 21238 Physical address: 37878 Value: 20
Virtual address: 11983 Physical address: 59855 Value: -77
Virtual address: 48394 Physical address: 1802 Value: 47
Virtual address: 11036 Physical address: 39964 Value: 0
Virtual address: 30557 Physical address: 16221 Value: 0
Virtual address: 23453 Physical address: 20637 Value: 0
Virtual address: 49847 Physical address: 31671 Value: -83
Virtual address: 30032 Physical address: 592 Value: 0
Virtual address: 48065 Physical address: 25793 Value: 0
Virtual address: 6957 Physical address: 26413 Value: 0
Virtual address: 2301 Physical address: 35325 Value: 0
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
Number of translated addresses = 1000
Page Faults     = 244
Page Fault Rate = 0.244
TLB Hits        = 55
TLB Hit Rate    = 0.055
PS C:\Users\jason\Dropbox\WINTER\60-330 Operating Systems Fundamamentals\Project\ProjectV1\Debug> _
```

My tlb hits are off by 1 (compared to correct.txt). Not sure why? I tried to figure it out but couldn't see what I did wrong?

Since the logical addresses were random, the tlb hit rate is rather low. In reality, during program execution, the CPU will be reading instructions and data where the addresses are within a limited range (perhaps as low as 4096 kb for most stacks). Therefore, more frames and pages will be located on the tlb and the hit rates will be higher. Complier optimization will also help to create a "tighter" instruction and data variability. Since this was a purely academic exercise with a software-implemented tlb, the effects were not seen in the statistics.

Program analysis

I attempted the two-person project since the single person project was fairly simple. The design of the program is straight-forward:

The main function reads in the addresses.txt file, validates the input and then calls a function called process_addresses() (function prototypes and global variables and constants will be provided at the end of the report.).

Once the call returns, the statistics of the program and address translations are printed to console.

Below is the main function, process_addresses function and print_main_results function:

```
int main(const int argc, char *argv[])
{

        char address[MAX_BUFFER_SIZE];
        backing_store_name = "BACKING_STORE.bin";
        int number_of_translated_addresses = 0;
        tlb_hits = 0;

        if (argc != 2)
        {
                fprintf(stderr,"Usage: ./a.out [input file]\n");
                return -1;
        }

        if(!valid_file_input(backing_store = fopen(backing_store_name, "rb"),
                backing_store_name)) return -1;
        if(!valid_file_input(address_file  = fopen(argv[1], "r"),
                argv[1])) return -1;

        process_addresses(address, address_file, &number_of_translated_addresses);

        print_main_results(number_of_translated_addresses);

        // close the input file and backing store
        fclose(address_file);
        fclose(backing_store);

        return 0;
}
```

```
void process_addresses(FILE *address_file, int * number_of_translated_addresses)
{
        char address[MAX_BUFFER_SIZE];

        // read through the input file and output each logical address
        while ( fgets(address, MAX_BUFFER_SIZE, address_file) != NULL)
        {
                // get the physical address and value stored at that address
                find_page(atoi(address));
                *number_of_translated_addresses += 1;
        }
}
```

```c
void print_main_results(const int number_of_translated_addresses)
{
        // calculate and print out the stats
        printf("Number of translated addresses = %d\n", number_of_translated_addresses);
        const double pf_rate  = page_faults / (double)number_of_translated_addresses;
        const double tlb_rate = tlb_hits    / (double)number_of_translated_addresses;

        printf("Page Faults     = %d\n",   page_faults);
        printf("Page Fault Rate = %.3f\n", pf_rate);
        printf("TLB Hits        = %d\n",   tlb_hits);
        printf("TLB Hit Rate    = %.3f\n", tlb_rate);
}
```

The print_main_results() function is straight-forward so I will discuss the implementation of the process_addresses() function.

The function reads through the addresses.txt file and for each logical address in the file, a function called find page is called, and the number of addresses counter is incremented. The find_page() function searches the tlb and page table for a page. If the page is not found in either of those data structures, a backing store (long-term storage) is searched for on-demand-paging.

We will look at the find_page() function next:

```c
bool find_page(const int logical_address)
{
        // obtain the page number and offset from the logical address
        const int page_number  = ((logical_address & ADDRESS_MASK) >> 8);
        const int offset       =  (logical_address & OFFSET_MASK);
        bool page_fault_raised = FALSE;
        bool page_found        = FALSE;
        int frame_number       = -1;

        // first try to get page from TLB
        search_tlb(page_number, &page_found, &frame_number);

        // if the frame number was not found in tlb, search the page_table...
        if(!page_found)
            search_page_table(page_number, &frame_number, &page_fault_raised);

        if(page_fault_raised)
        {
            read_from_store(page_number);
        frame_number = first_available_frame - 1; // set the frameNumber to the current firstAvailableFrame index
            tlb_insert(page_number, frame_number);
        }

        // frame number and offset used to get the signed value stored at that address
        const signed char value = physical_memory[frame_number][offset];
        printf("Virtual address: %d Physical address: %d Value: %d\n", logical_address, (frame_number << 8) | offset, value);

        return page_fault_raised;
}
```

This function first obtains the page number and offset from the logical address and then sets some flags. Once those are completed, we search the simulated tlb for the page number. If the page is not found, search the page table.

Again, if the page is not found in the page table, a page fault is raised and a search is made in the backing store.

Through one of these functions, the page will be found and the tlb will be updated. Once updated the result is printed to the console.

This process is repeated for each logical address in the file.

Next, we look at the search functions:

- Search_tlb
- Search_page_table

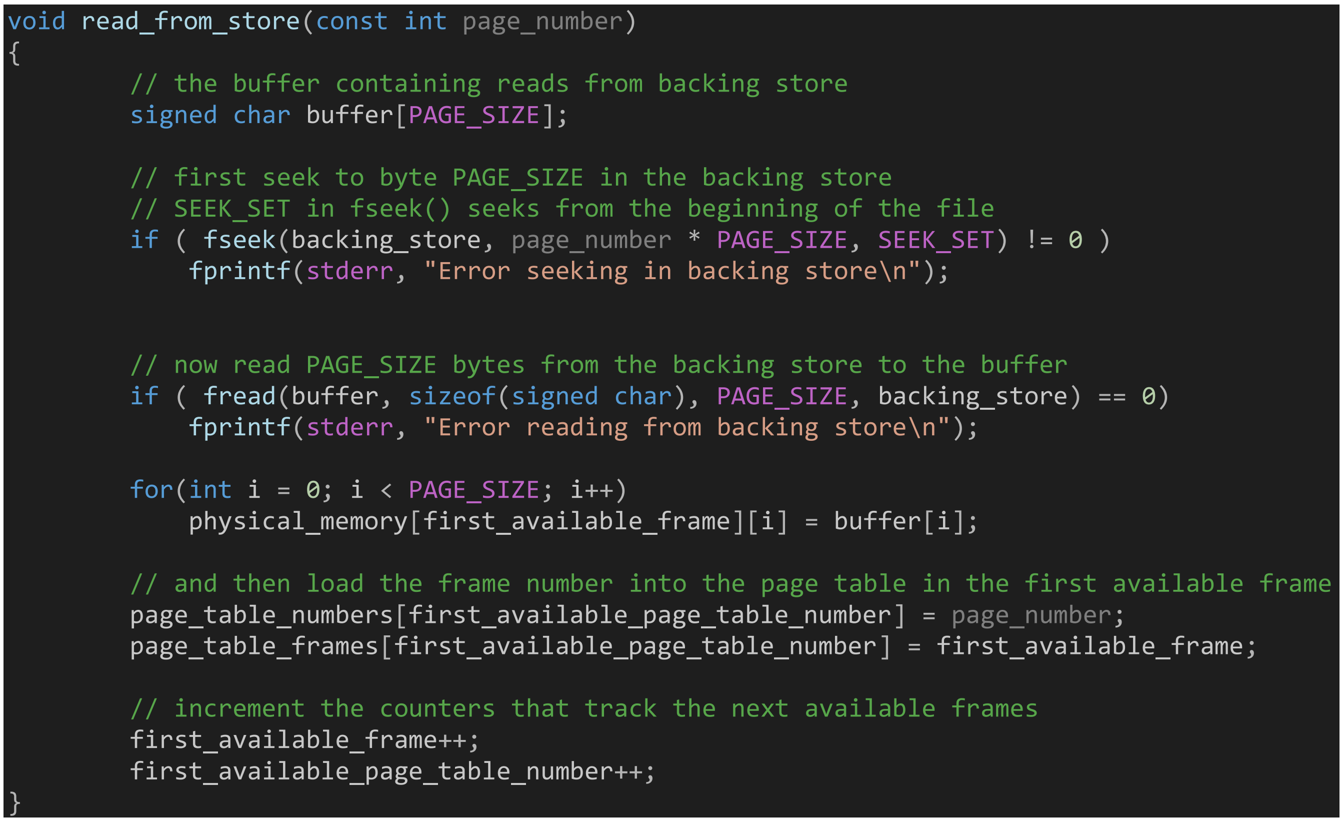
```
void search_tlb(const int page_number, bool * page_found, int * frame_number)
{
    for (int i = 0; i < TLB_SIZE; i++)
        if(tlb_page_number[i] == page_number)
        {
             *frame_number = tlb_frame_number[i];
             tlb_hits++;
             *page_found = TRUE;
        }
}
```

```
void search_page_table(const int page_number, int * frame_number, bool * page_fault_raised)
{
    for (int i = 0; i < first_available_page_table_number; i++)
        if(page_table_numbers[i] == page_number)

        // if the page is found in those contents
        *frame_number = page_table_frames[i];     // extract the frame number

        if(*frame_number == -1)
        {
            *page_fault_raised = TRUE;
            page_faults++;
        }
}
```

These functions loop through each data structure until a match is found. For the search_page_table(), if the page is not found the page fault is raised.
After these function calls, a check is made to see a page fault was raised and if it was, a search is made in the backing store.

```
if(page_fault_raised)
{
    read_from_store(page_number);
    frame_number = first_available_frame - 1;
    tlb_insert(page_number, frame_number);
}
```

Here, we read from the store and then insert the page number and frame number into the tlb.
We look at these functions next.

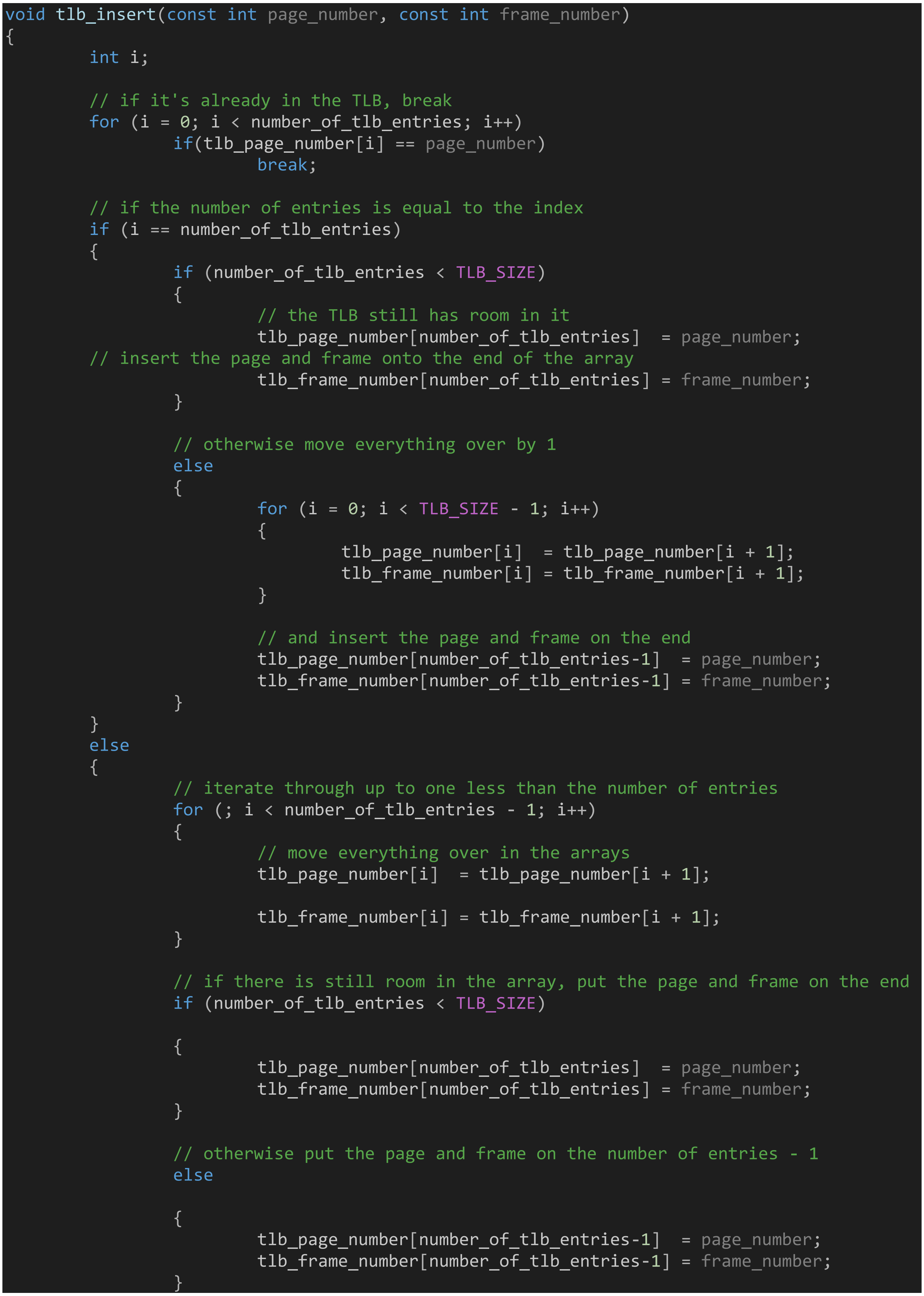


```
void read_from_store(const int page_number)
{
        // the buffer containing reads from backing store
        signed char buffer[PAGE_SIZE];

        // first seek to byte PAGE_SIZE in the backing store
        // SEEK_SET in fseek() seeks from the beginning of the file
        if ( fseek(backing_store, page_number * PAGE_SIZE, SEEK_SET) != 0 )
            fprintf(stderr, "Error seeking in backing store\n");


        // now read PAGE_SIZE bytes from the backing store to the buffer
        if ( fread(buffer, sizeof(signed char), PAGE_SIZE, backing_store) == 0)
            fprintf(stderr, "Error reading from backing store\n");

        for(int i = 0; i < PAGE_SIZE; i++)
            physical_memory[first_available_frame][i] = buffer[i];

        // and then load the frame number into the page table in the first available frame
        page_table_numbers[first_available_page_table_number] = page_number;
        page_table_frames[first_available_page_table_number] = first_available_frame;

        // increment the counters that track the next available frames
        first_available_frame++;
        first_available_page_table_number++;
}
```

Since we know what page we are looking for we directly find the page in the backing store using fseek(). Then we store address into the simulated physical memory.
Next is the function tlb_insert().

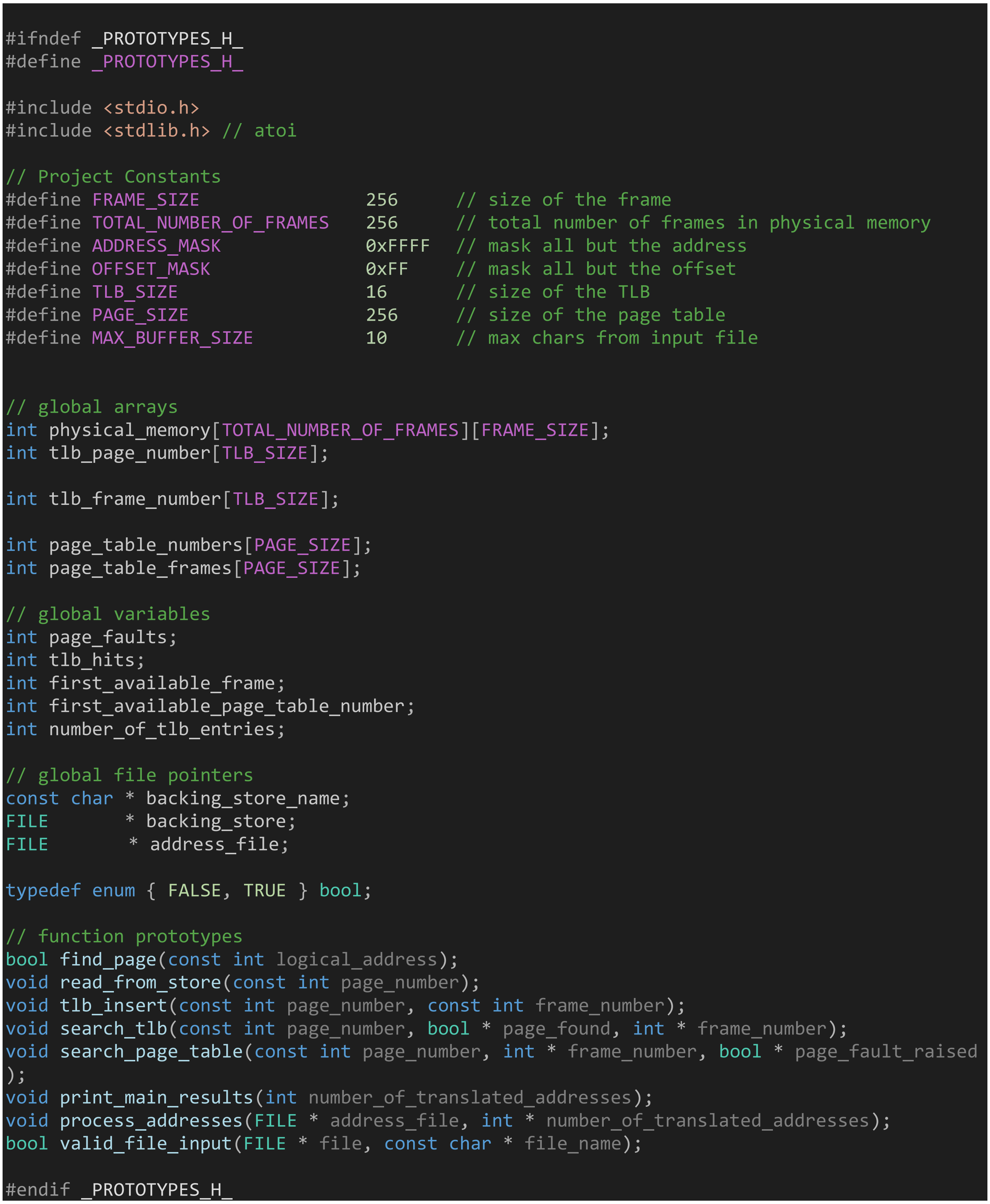
The tlb_insert() function uses a FIFO design to replace the last element of the array which has been shifted on each tlb insert from beginning to end. So, the last element is the oldest.

See code on next page.



```c
void tlb_insert(const int page_number, const int frame_number)
{
        int i;

        // if it's already in the TLB, break
        for (i = 0; i < number_of_tlb_entries; i++)
                if(tlb_page_number[i] == page_number)
                        break;

        // if the number of entries is equal to the index
        if (i == number_of_tlb_entries)
        {
                if (number_of_tlb_entries < TLB_SIZE)
                {
                        // the TLB still has room in it
                        tlb_page_number[number_of_tlb_entries]  = page_number;
        // insert the page and frame onto the end of the array
                        tlb_frame_number[number_of_tlb_entries] = frame_number;
                }

                // otherwise move everything over by 1
                else
                {
                        for (i = 0; i < TLB_SIZE - 1; i++)
                        {
                                tlb_page_number[i]  = tlb_page_number[i + 1];
                                tlb_frame_number[i] = tlb_frame_number[i + 1];
                        }

                        // and insert the page and frame on the end
                        tlb_page_number[number_of_tlb_entries-1]  = page_number;
                        tlb_frame_number[number_of_tlb_entries-1] = frame_number;
                }
        }
        else
        {
                // iterate through up to one less than the number of entries
                for (; i < number_of_tlb_entries - 1; i++)
                {
                        // move everything over in the arrays
                        tlb_page_number[i]  = tlb_page_number[i + 1];

                        tlb_frame_number[i] = tlb_frame_number[i + 1];
                }

                // if there is still room in the array, put the page and frame on the end
                if (number_of_tlb_entries < TLB_SIZE)

                {
                        tlb_page_number[number_of_tlb_entries]  = page_number;
                        tlb_frame_number[number_of_tlb_entries] = frame_number;
                }

                // otherwise put the page and frame on the number of entries - 1
                else

                {
                        tlb_page_number[number_of_tlb_entries-1]  = page_number;
                        tlb_frame_number[number_of_tlb_entries-1] = frame_number;
                }
        }

        // if there is still room in the arrays, increment the number of entries
        if (number_of_tlb_entries < TLB_SIZE)

                number_of_tlb_entries++;
}
```

PROJECT PROTOTYPES, GLOBALS AND CONSTANTS



```c
#ifndef _PROTOTYPES_H_
#define _PROTOTYPES_H_

#include <stdio.h>
#include <stdlib.h> // atoi

// Project Constants
#define FRAME_SIZE                   256     // size of the frame
#define TOTAL_NUMBER_OF_FRAMES       256     // total number of frames in physical memory
#define ADDRESS_MASK                 0xFFFF  // mask all but the address
#define OFFSET_MASK                  0xFF    // mask all but the offset
#define TLB_SIZE                     16      // size of the TLB
#define PAGE_SIZE                    256     // size of the page table
#define MAX_BUFFER_SIZE              10      // max chars from input file


// global arrays
int physical_memory[TOTAL_NUMBER_OF_FRAMES][FRAME_SIZE];
int tlb_page_number[TLB_SIZE];

int tlb_frame_number[TLB_SIZE];

int page_table_numbers[PAGE_SIZE];
int page_table_frames[PAGE_SIZE];

// global variables
int page_faults;
int tlb_hits;
int first_available_frame;
int first_available_page_table_number;
int number_of_tlb_entries;

// global file pointers
const char * backing_store_name;
FILE       * backing_store;
FILE       * address_file;

typedef enum { FALSE, TRUE } bool;

// function prototypes
bool find_page(const int logical_address);
void read_from_store(const int page_number);
void tlb_insert(const int page_number, const int frame_number);
void search_tlb(const int page_number, bool * page_found, int * frame_number);
void search_page_table(const int page_number, int * frame_number, bool * page_fault_raised
);
void print_main_results(int number_of_translated_addresses);
void process_addresses(FILE * address_file, int * number_of_translated_addresses);
bool valid_file_input(FILE * file, const char * file_name);

#endif _PROTOTYPES_H_
```