**Student name and email:**

**Student number:**

Exam info

## Exam is written

* There are 100 points in total, which will be used to decide the grade.
* about 75% of the questions/points are written problems.
* about 25% programming exercises; here you can use the tools for development.

## Censorship

There is external censorship.

## Exam Guidelines:

* The exam must be completed at AAU campus; it will be 4 hours.
  + The first two and half hours will be the written part, and the last part will be programming.
* The exam is handed out in paper.
* The programming answers must be submitted in digital exam as a file.
* If you think that the task formulation is deficient / incorrect, then write which additional assumptions / changes you use as a basis for your answer.
* Remember to clearly write the name and study number on the answer.

### Permitted aids:

* Common IT tools such as text editor / word processing, PDF annotation tool, Calculator.
* All tools that are introducing in the course.
* Formats of the entire LC-3 instruction set (page 148 of ICS book).
* The *common.h* and *common\_threads.h* files from the OSTEP book project.

# Problem 1: Data Types, and Operations

1. **Consider two hexadecimal numbers: x434F4D50 and x55544552. What values do they represent for each of the five data types? (4 points)**

|  |  |  |
| --- | --- | --- |
|  | x434F4D50 | x55544552 |
| Unsigned Binary | 1129270608 | |  | | --- | | 1431586130 | |  | |
| 1’s compliment | SAME | SAME |
| 2’s compliment | SAME | SAME |

1. **Add the following 2’s complement binary numbers. Also express the**

**answer in decimal. (6 points)**

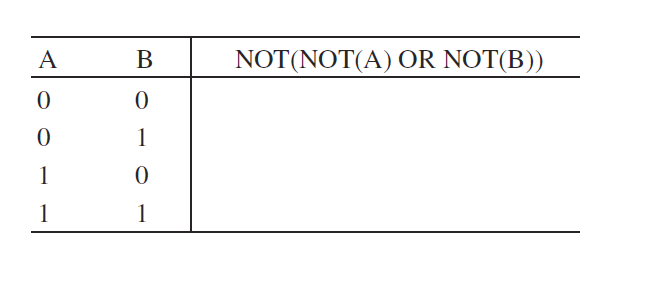
*a.* 01 + 1011=1100=-4

*b.* 11 + 01010101=01010100=-1+85=84

*c.* 0101 + 110=0011=5-2=3

# Problem 2: Digital Logic

1. **Fill in the truth table for the logical expression NOT(NOT(A) OR**

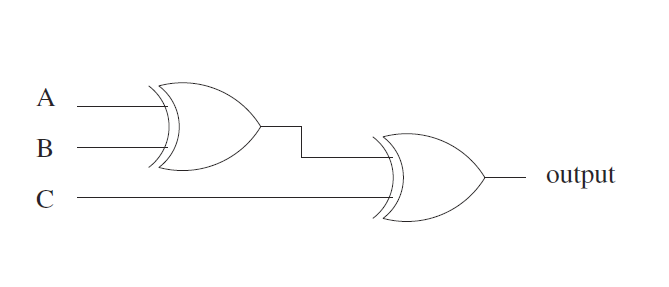
**NOT(B)). What single logic gate has the same truth table?** **(6 points)** ASsa

Answer:

|  |  |  |
| --- | --- | --- |
| A | B | NOT(NOT(A) OR NOT(B)) |
| 1 | 1 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 0 | 0 | 0 |

1. **1.2 The following logic circuits consist of two exclusive-OR gates. Construct the output truth table. (6 points)**

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | Out |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |



# Problem 3: Von Neumann Model

1. **Suppose a 32-bit instruction takes the following format:** **(6 points)**



If there are 225 opcodes and 120 registers,

*1.* What is the minimum number of bits required to represent the OPCODE? 8 BITS

*2.* What is the minimum number of bits required to represent the destination register (DR)? 7 BITS

1. **State the phases of the instruction cycle, and briefly describe what**

**operations occur in each phase.** **(5 points)**

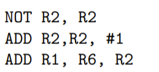
Answer:

You can find the answer the book in 4.3.2

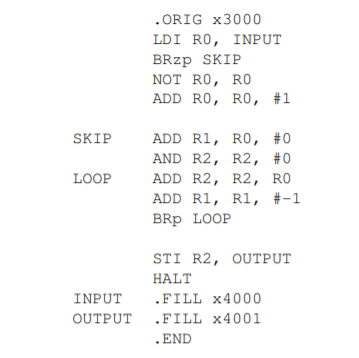
# Problem 4: LC-3 and Assembly

1. **How could one perform the following operation using only three LC-3 instructions? (7 points)**

R1=R6-R1



**b) Consider** **the following program written in LC-3 assembly language:** **(5 points)**



What does the program do?

Answer:

Squares the value at x4000 and stores it in 4001

# Problem 5: Process and Disk

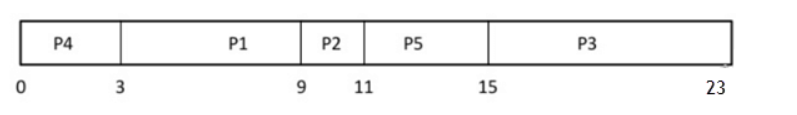
1. **Consider the following table of arrival time and burst time for five processes P1, P2, P3, P4 and P5.** **(10 points)**

| **Process** | **CPU Time** | **Arrival Time** |
| --- | --- | --- |
| P1 | 6 ms | 2 ms |
| P2 | 2 ms | 5 ms |
| P3 | 8 ms | 1 ms |
| P4 | 3 ms | 0 ms |
| P5 | 4 ms | 4 ms |

Compute the response time and turnaround time for each process using shortest job first scheduling.

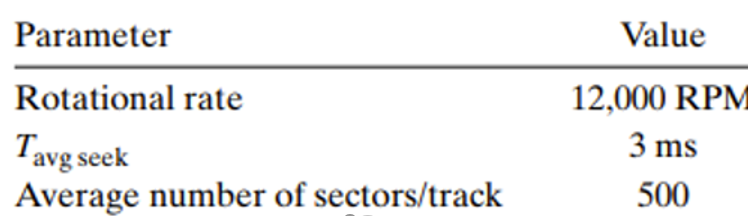
Answer:

Here is the time order of execution for each process. You can compute turnaround time using this.



Response time = 0 +3+9+11+15/5

1. **Estimate the average time (in ms) to access a sector on the following disk: (5 points)**



Answer:

Here an example when we consider with following parameter:  
  
Rotational rate = 15,000 RPM  
  
Tavg seek =4 ms  
  
Average number of sectors/track = 800

And the answer is following:

Rotational rate = 15000 RPM = (60/15000) = 4 × 10⁻³ s

Rotation time = 1/2 × Rotational rate = 1/2 × 4 × 10⁻³ = 2 × 10⁻³ s

Rotation time = 2 × 10⁻³ × 1000 ms = 2 ms

Transfer time = Rotational rate x (1/Average number of sectors/track) = 4 × 10⁻³ x 1/800 = 5 × 10⁻⁶

Transfer time = 5 × 10⁻⁶ × 1000 ms = 0.005 ms

Average time to access a Sector on the disk = Average seek time + Rotation time + transfer time

Average time to access a Sector on the disk = 4ms + 2ms + 0.005 ms = 6.005 ms

The average time to access a Sector on the disk is 6.005 ms

# Problem 6: Virtual memory

* 1. **Consider a machine with 64 MB physical memory and a 32-bit virtual address space. If the page size is 4KB, what is the approximate size of the page table?** **(10 points)**  
       
     Answer:

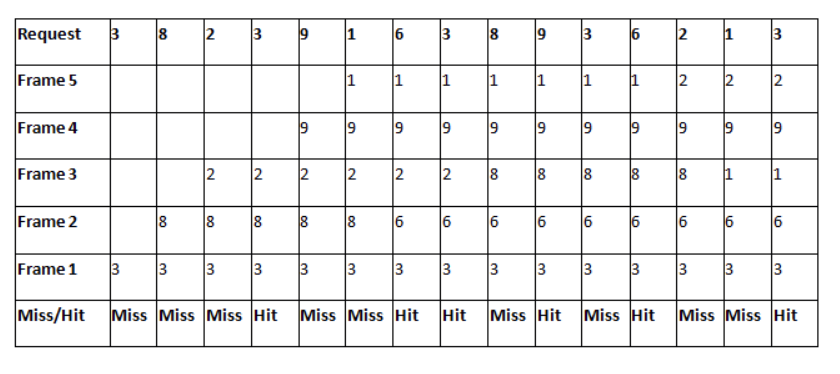
Number of pages =2^32/4KB=2^20=2^20 as we need to map every possible virtual address.  
  
So, we need 2^20  entries in the page table. Physical memory being 64 MB, a physical address must be 26 bits and a page address  needs 26−12=14 address bits. So, each page table entry must be at least 14 bits. So the answer 2^20 \* 14.

* 1. **Consider the page reference string:** **(5 points)**

**3, 8, 2, 3, 9, 1, 6, 3, 8, 9, 3, 6, 2, 1, 3**

Find the number of page faults with LRU, where 5 page frames are present.

Answer:



Number of Page Faults = 9  
Number of Hits = 6

# Problem 7.1: Deadlock (part 1)

## a) Discuss why the following code might lead to a deadlock (3 points)

* Describe a possible execution path that leads to a deadlock.

sem\_t a; sem\_t b; sem\_t c; sem\_t d;  
sem\_init(&a, 0, 1); sem\_init(&b, 0, 1); sem\_init(&c, 0, 1); sem\_init(&d, 0, 1);

|  |  |  |
| --- | --- | --- |
| Thread 1 | Thread 2 | Thread 3 |
| sem\_wait(&a);  sem\_wait(&b);  sem\_wait(&c);  // critical section  sem\_post(&c);  sem\_post(&b);  sem\_post(&a); | sem\_wait(&b);  sem\_wait(&c);  sem\_wait(&d);  // critical section  sem\_post(&d);  sem\_post(&c);  sem\_post(&b); | sem\_wait(&c);  sem\_wait(&d);  sem\_wait(&a);  // critical section  sem\_post(&a);  sem\_post(&d);  sem\_post(&c); |

Answer:

According to the definition, to have a deadlock, the following four conditions must hold at the same time:

1. Mutual exclusion: threads take exclusive control of resources.
   1. In this example, the threads are using semaphores, and once calling the wait function threads get exclusive access to the semaphores.
2. Hold-and-wait: threads hold resources while waiting for additional resources.
   1. In this example, all threads call three wait functions one after each other on different semaphores. For example, once Thread1 gets access to semaphore a, it will hold the access while trying to get access to semaphores b and c.
3. No pre-emption: resources cannot be forcibly removed from threads holding them.
   1. In this example, we are using semaphores, which according to the definition, once a thread gets hold of a semaphore, it cannot be forced to release it.
4. Circular wait: there is a circular chain of threads such that each thread holds one more resource that are being requested by the next thread in the chain.
   1. In this example, we can see that the following circular wait happens: T1 holds a, which is wanted by T3, while T3 holds c, which is wanted by T1.

A possible execution flow that leads to a deadlock can be the one described in the point 4 above, as visualized in the table below. At the end of this execution, both T1 and T3 cannot progress anymore since they are waiting for each other’s resources, i.e., semaphore.

|  |  |  |
| --- | --- | --- |
| T1  Starts execution  Gets hold of a  Gets hold of b  <interrupt>  Wakes up  Tries to get hold of c, but gets blocked since T3 holds it | T2 | T3  Starts execution  Gets hold of c  Gets hold of d  Tries to get hold of a, but gets blocked since T1 holds it  <goes to Sleep> |

## b) Rewrite the pseudocode of the table to guarantee the absence of deadlock (2 points)

* Discuss which of the four deadlock conditions did you avoid here to eliminate the deadlock.

To guarantee the absence of deadlock, we need to eliminate one of the four conditions mentioned above. In this solution, we can remove the circular wait. A way to do this, is by defining a total order of the semaphores and make sure that all threads try to get hold of the semaphores following this order.

Total order: a < b < c < d

With this order in mind, we can rewrite the source code below by placing sem\_wait(&a) as the first instruction in Thread 3.

sem\_t a; sem\_t b; sem\_t c; sem\_t d;  
sem\_init(&a, 0, 1); sem\_init(&b, 0, 1); sem\_init(&c, 0, 1); sem\_init(&d, 0, 1);

|  |  |  |
| --- | --- | --- |
| Thread 1 | Thread 2 | Thread 3 |
| sem\_wait(&a);  sem\_wait(&b);  sem\_wait(&c);  // critical section  sem\_post(&c);  sem\_post(&b);  sem\_post(&a); | sem\_wait(&b);  sem\_wait(&c);  sem\_wait(&d);  // critical section  sem\_post(&d);  sem\_post(&c);  sem\_post(&b); | sem\_wait(&a);  sem\_wait(&c);  sem\_wait(&d);  // critical section  sem\_post(&d);  sem\_post(&c);  sem\_post(&a); |

Programming part

# Problem 7.1: Deadlock (part 2)

## c) Implement the deadlock-free solution solved in 7.1b (5 points)

* Implement the program of the deadlock-free pseudocode that has 3 threads, 4 semaphores, and one shared variable initialized to 0.
* In the critical section, each thread simply increases the shared variable by 1.
* Print the final value of the shared variable in the main thread after each thread is done running.

# Problem 7.2: Parallel sum (7 points)

Implement a multithreaded program that calculates the sum of an array in parallel with **two threads**. Each thread has an id, i.e., 0 and 1, so we will call them T0 and T1.

T0 is responsible for calculating the sum of the first half of the array, while T1 is responsible for the sum of the second half. Each thread gets the id as an argument and returns the partial sum as a value.

Use the code template below as a starting point and complete the TODO parts with your own code. Notice that the program takes one argument in the command line, which defines the size of the array. The array then gets filled in with random numbers.

// *TODO: include the needed libraries*volatile int array\_size;  
volatile int\* array; // shared global array  
  
void \*mythread(void \*arg) {  
 int id = (int) arg;  
  
 // *TODO: YOUR CODE HERE* // *IMPLEMENT THE LOGIC OF THE THREAD*}  
  
int main(int argc, char \*argv[]) {  
 if (argc != 2) {  
 fprintf(stderr, "usage: main-first <arraysize>\n");  
 exit(1);  
 }  
 array\_size = atoi(argv[1]);  
 srand(time(NULL));  
 array = (int \*) malloc(array\_size \* sizeof(int));  
 if (array == NULL) {  
 fprintf(stderr, "Could not allocate memory\n");  
 exit(1);  
 }  
  
 for (int i = 0; i < array\_size; i++) {  
 array[i] = rand() % 100;  
 }  
  
 // *TODO: YOUR CODE HERE* // *CREATE THE TWO THREADS WITH ID 0 AND 1* // *MAKE THE THREAD WITH ID 0 CALCULATE AND RETURN THE SUM OF THE FIRST HALF OF THE ARRAY: STORE THE RESULT IN A VARIABLE sum1* // *MAKE THE THREAD WITH ID 1 CALCULATE AND RETURN THE SUM OF THE SECOND HALF OF THE ARRAY: STORE THE RESULT IN A VARIABLE sum2* // *CALCULATE THE TOTAL SUM AS sum = sum1 + sum2* // *PRINT THE TOTAL SUM* return 0;  
}

# Problem 7.3: Ordering threads (8 points)

Use the code below as a starting point to create three threads A, B, C and make thread A to call the function *first()*, thread B to call the function *second(),* and thread C to call the function *third().*

The threads should all start one after each other, you should not control the order of the threads’ execution.

Implement the necessary synchronization mechanisms within the functions so that no matter the order of the threads’ execution, the output is always the string “first-second-third”.

// *TODO: include the needed libraries*// *TODO: declare any necessary global variable if needed*// *TODO: implement the logic so that this function is the first to print*void \*first(void \*arg) {  
  
 printf("first-");  
  
}  
  
// *TODO: implement the logic so that this function is the second to print*void \*second(void \*arg) {  
  
 printf("second-");  
  
}  
  
// *TODO: implement the logic so that this function is the last to print*void \*third(void \*arg) {  
  
 printf("third\n");  
  
}  
  
int main(int argc, char \*argv[]) {  
  
 // *TODO: YOUR CODE HERE* // *CREATE THE THREE THREADS, EACH WILL CALL ONE OF THE THREE FUNCTIONS ABOVE* // *WAIT FOR THE THREADS TO COMPLETE AND EXIT* // NOTE: EVERY TIME THE CODE GETS EXECUTED THE OUTPUT SHOULD ALWAYS BE THE SAME: first-second-third  
  
 return 0;  
}