- **Q8:**
  - LRU = 3 means that the page has not been used for the longest amount of time, LRU = 0 means that the page has been the most recently
  - For part 2, the 4 accesses total (across 2.a and 2.b) mentioned happen sequentially not independently
  - For part 3.b, use the same access times as in the previous section 2.b.
  - For part 3.c, it does not follow after 3.a-b (there are again no mappings for the file)
- **Q7.2:** pretend the newline "\n" is the same as a space. This is not an intentional bug.
- **Q4 datapath diagram:** The box for (iii) includes the mux right before it; in other words, ignore the alusrc mux in the datapath.
- **Q5**:
  - fill in the bubbles next to the instruction that must stall as a result of the hazard when the code is run sequentially on the CPU as one program.
  - AND and OR gates have the same delay
  - This is DIFFERENT from your register file in class -- you CANNOT read and write in the same cycle (decode has to be after write back).
- **Q2:** 2.2 There is one bug that is a bug in the old C standard but not C99. Whether or not you notice this particular bug will not be graded
- **Q7.2:** pretend the new line ('\n') is a space (' '). This was not an intentional bug.
- **Q3**:
  - Ooops… the memory address we gave you is not word aligned… Please use it anyway and ignore this silly mistake!
  - You do have to look at the code a little bit to figure out which line gets modified. The hint is just warning you not to try to decode the whole thing

University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Spring 2017 Instructors: Nicholas Weaver, Gerald Friedland 2017-05-09

:(CS61C FINAL :)

After the exam, indicate on the line above where you fall in the emotion spectrum between "sad" & "smiley"...

Last Name	
First Name	
Student ID Number	
CS61C Login	`cs61c-`
*The name of your **SECTION** TA and time*	
Name of the person to your LEFT	
Name of the person to your RIGHT	
*All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet. **(please sign)***	

Instructions (Read Me!)

- This booklet contains 16 numbered pages including the cover page.
- Please turn off all cell phones, smartwatches, and other mobile devices. Remove all hats & headphones. Place your backpacks, laptops and jackets under your seat.
- You have 180 minutes to complete this exam. The exam is closed book; no computers, phones, or calculators are allowed. You may use three handwritten 8.5"x11" page (front and back) crib sheet in addition to the MIPS Green Sheet, which we will provide.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided.
- Scores for Q1-Q3 will be used for Midterm 1 clobbering and scores Q4-Q6 will be used for Midterm 2 clobbering

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Total
Points Possible	15	10	10	10	10	15	10	20	15	5	120

Q1: Many, Many Matrices (15 points)

In this problem, you will be asked to implement operations on a data structure that represents a set of 2D matrices. As the structure is a collection of matrices, it can be conveniently defined as a triple pointer, as such: `int ***matrices`. For simplicity, we will only consider square matrices of integers.

Please fill in the blanks in the functions below to complete the desired implementation. ***You may not need all of the lines provided.***

When completing the functions, you may use the following assumptions:

- All pointers are valid and non-NULL
- All calls to malloc are guaranteed to succeed and do not need to be checked
- All dimension and number counts are valid, meaning that we **never** pass in the dimension value of a matrix that is larger or smaller than the actual dimension
- For multiplication, you may assume that the dimensions align such that the matrix multiplication is valid
- For any matrix `m`, the value at the ith row and jth column is denoted by `m[i][j]`.

You MAY NOT assume that the size of an integer is 4 bytes, the size of a character is 1 byte etc.

Each function has a description of what it does. You may find it helpful to see the example of the product of a matrix and vector shown below.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_{00}b_0 + a_{01}b1 + a_{02}b_2 \\ a_{10}b_0 + a_{11}b1 + a_{12}b_2 \\ a_{20}b_0 + a_{21}b1 + a_{22}b_2 \end{bmatrix}$$

```
// Allocates all memory for a set of “numMatrices” matrices, each with
// dimensions “dim x dim”. Does not initialize the values in each matrix.
int ***allocate(int numMatrices, int dim) {
      int ***matrices = (int ***) malloc(sizeof(int **) * numMatricies);
      for (int i = 0; i < numMatrices; i++) {
            matrices[i] = (int **) malloc(sizeof(int *) * dim);
            for (int j = 0; j < dim; j++) {
                  matrices[i][j] = (int *) malloc(sizeof(int) * dim);
            }
      }
      return matrices;
}
```

```
// Multiplies the kth matrix in the set with the vector provides and
// returns this new vector
int *multiply(int ***matrices, int *vector, int k, int dim) {
      int *product = (int *) malloc(sizeof(int) * dim); // Could use calloc here
      for (int i = 0; i < dim; i++) {     // And not have this loop
            product[i] = 0;
      }
      int **matrix = matrices[k];
      for (int i = 0; i < dim; i++) {
            for (int j = 0; j < dim; j++) {
                  product[i] += vector[j] * matrix[i][j];
            }
      }
      return product;
}

// Free's all memory from the given set of matrices
void freeMatrices(int ***matrices, int numMatrices, int dim) {
      for (int i = 0; i < numMatrices; i++) {
            for (int j = 0; j < dim; j++) {
                  free(matrices[i][j]);
            }
            free(matrices[i]);
      }
      free(matrices);
}
```

Q2: Memory State Warriors (10 points)

1) Consider the block of code below. You may assume all calls to `malloc` succeed.

```
#define TEAMS 30
char *steph = “curry”;

int main(void) {
      char *kevin = “durant”;
      char klay[9] = “thompson”;
      int wins = 67;

      char **warriors = malloc(3 * sizeof(char *));
      warriors[0] = kevin;
      warriors[1] = steph;
      warriors[2] = (char *) klay;

      char ***nba = malloc(TEAMS * sizeof(char **));
      *nba = warriors;

      return wins;
}
```

For the code on the previous page, please list on what region of memory the following variables/quantities reside right before the function `main` returns. If there are multiple answers, write all possible regions.

Variable/Quantity	Region of Memory
`nba`	Stack
`*nba`	Heap
`**nba`	Heap
`**((*nba) + 1)`	Static
`klay`	Stack
`steph`	Static
TEAMS	Code

2) Consider the block of code below that, when given a value n, returns an array with the integer values from 1 to n. Are there any issues with the function? If yes, then **list all issues** with a brief explanation of each one on the lines below. Furthermore, for each issue you outline, describe how to fix the problem on following line. You may or may not need all lines.

```
int *array_of_n(int n) {
      int arr[n];
      for (int i = 0; i <= n; i++) {
            arr[i] = i + 1;
      }
      return arr;
}
```

1. Problem: The array is allocated on the stack of the local frame, which will be free'd when the function returns

   Solution: Dynamically allocate memory for the array (malloc or calloc)

2. Problem: Accessing the n^{th} element of the array, which is beyond the array's bounds. This could segfault or corrupt other values on the stack.

   Solution: Make limit for the for loop i < n instead of i <= n

3. Problem: Undefined behavior if n is negative

   Solution: Check for negative values of n, and return NULL if so

Q3: Back at it Again with Mipstery (10 points)

Mipstery is known to take in one positive integer as the only argument. Assume **Mipstery** resides at address **0xABCDEEFF**. ***[Hint: do not try to interpret Mipstery immediately (part d) -- parts a-c can be answered without first understanding the entire program!]***

```
Mipstery:  1.    addiu     $t0,  $0,   1
           2.    addu  $s0,  $t0,  $0
           3.    la    $t1,  Mipstery
           4.    lw    $t2,  4($t1)
           5.    li    $t3,  0x7FFF
           6.    and   $t4,  $t3,  $t2
           7.    ori   $t4, $t4, 32768       # 32768 = 2^15
           8.    sll   $t2,  $t2,  20
           9.    or    $t5,   $t2,      $t4
           10.   sw    $t5,   4($t1)
           11.   addiu      a0,   a0,    -1
           12.   bne   $a0,  $0,   Mipstery
           13.   move    $v0,     $s0
           14.   j     done
           … …
```

(a) Which instruction gets modified in the above code? Give the line number.

Line 2

(b) Notice that "la" and "li" (line 3 and line 5) are both pseudo-instructions. Assemble these instructions into TAL as efficiently as possible, recalling specifications for the pseudo-instructions from your project 2.

```
la    $t1,  Mipstery              li    $t3,  0x7FFF
lui   $at, 0xABCD                 ADDIU $t3, $0, 0x7FFF # recall proj2
ori   $t1, $at, 0xEEFF      or:   ORI   $t3, $0, 0x7FFF   specifications
```

(c) Convert the **bne** instruction (line 12 in the MAL code above) into machine code, assuming that "la" gets expanded into 1 instruction and "li" gets expanded into 3 instructions. Express your answer as a hexadecimal number.

The imm offset for bne should be -(12 + 2) = -14. (2 is for 2 expanded instructions)

(d) (***Hard***) In one sentence: What does Mipstery do? Assume $a0 is always positive. [Hint: first convert the modified instruction into hex!]

It computes power of 2.

Q4: When Memory Encounters Branches (10 points)

Consider the new instruction “memory branch equal”:

```
membeq $rs, $rt, label
```

It performs the following operation:

```
if (Mem[R[$rs]] == Mem[R[$rt]]):
      R[$at] ← 1
      PC ← PC + 4 + BranchAddr
Else:
      R[$at] ← 0
      PC ← PC + 4
```

Your task is to modify the single cycle MIPS CPU you have seen in lecture to support `membeq` while maintaining functionality of the rest of the MIPS ISA.

(a) First, assume you are free to modify the wires and gates, but cannot modify the major components (Instruction Memory, Regfile, ALU, Data Memory). Are you able to implement this instruction using our current CPU datapath? In one sentence, explain your answer.

NO, because it requires two individual data memory outputs, but current memory component only supports one input/output port..

(b) Now assume that you CAN modify ONE major component listed in part (a), and you may add to it one new input and one output, called `newInput` and `newOutput`, which you may define as you’d like. You may also modify wiring and gates in areas labelled (i), (ii), (iii) and (modified) in your diagrams packet. Select the correct implementation for the **blanks (i), (ii) and (iii)** from the choices in the packet. **If multiple solutions exist choose the one that uses the least amount of hardware. Also note that two of the control signals have been defined for membeq in part (c) below.**

Select the correct implementations for the given blanks:

(i)	(ii)	(iii)
C	A (B also accepted since we didn’t explicitly extend EQL)	D

(c) List the values of all control signals for the `membeq` instruction in the table below. If the value of a particular signal does not matter, you must put an ‘X’. For the ALUCtr field, write the name of the operation the ALU should execute.

Jump	membeq	RegDst	RegWr	ExtOp	ALUSrc	ALUCtr	MemWr	MemToReg	Branch
0	1	X	1	1	X	XXXX	0	X	0

Q5: Pipelining (10 points)

RegFile Read	RegFile Setup	Register Clk-To-Q	Register Setup	Register Hold	Control Unit	ALU	Adder
200ps	200ps	100ps	100ps	100ps	300ps	300ps	100ps

Shifter	Mux	Or Gate	Memory Read	Memory Write	Extender	Hazard Detection
100ps	100ps	100ps	400ps	400ps	200ps	100ps

See the simplified MIPS Pipeline attached in the packet and labelled "**QUESTION 5.**" This processor can execute the following instructions: `ADD, ADDI, SUB, LW, SW, BEQ.` **Note that branches change the PC in the MEM stage.** Register values must be read during ID. New values are available from the Register File after WB.

1. What is the highest frequency that this CPU can run at?

   1.25 GHz

   a. (Fill out to facilitate partial credit) In what stage is the critical path?

   Mem->Branch->Decode

   b. (Fill out to facilitate partial credit) What is the critical path delay of that path?

   800ps

2. Fill in the bubbles next to lines of code that contain hazards for this program.

   ○ `LW   $s4, 0($s1)`

   ○ `ADDI $s4, $s4, 8` THIS

   ○ `ADD  $s4, $s4, $s3` THIS

   ○ `ADDI $s1, $s1, -4`

   ○ `BEQ  $s4, $zero, DONE //Assume taken` THIS

   ○ `LW   $s4, 0($s1)` THIS

   ○ `SW   $s4, 4($s2)` THIS

   `DONE:`

3. Now, assume we resolve hazards by stalling. How many cycles would the program from part 2 take to execute?

   Number of Cycles: 17 until Fetch of DONE, 18 until WB of BEQ

Q6: Cold Cache, Divine (15 points)

Given the following specifications of the following caches, fill in the blanks with the correct value or term. For each of these scenarios, assume you are working with a 24 bit physical address space on a byte-addressed machine. *Be as precise in your answer as possible* (don't leave variables in your answer).

Cache 1:		Cache 2:	
Type:	Direct Mapped	Type:	8-way set associative
Size:	256B	Size:	32KiB
# of Sets:	32	# of Sets:	64
Tag Bits:	16	Tag Bits:	12
Index Bits:	5	Index Bits:	6
Offset Bits:	3	Block Size:	64B

Cache 1: You get the index bits from taking the log base 2 of the number of sets, so you get 5. The offset bits are everything that's not the index and tag bits, so that's 3. In a direct mapped cache, all the total number of blocks are the total number of sets, so there's 32 blocks. The block size is 2^3=8B. Thus, the total size is 8*32=256B

Cache 2: You get the offset bits by taking the log base 2 of the block size, so you get 6. The index bits are everything that's not the tag and offset bits, so that's 6. The number of sets is 2^(index bits) so that's 64 sets. To figure out what kind of cache it is, you need to observe that the size is 32KiB = 2^15B, and that since each block is 64=2^6B, then there are 2^15/2^6 = 2^9 blocks. Since there are 2^6 sets, then each set needs to hold 2^9/2^6=2^3 blocks in them, and thus, there are 8 ways.

Consider the following C code:

```
#define ARRAY_SIZE 128*1024
double a[ARRAY_SIZE];
int i, j;
void normalize(int vect_size){
      for (i = 0; i < ARRAY_SIZE, i+=vect_size) {
            double squared_sum = 0.0;
            for (j = 0; j < vect_size; j += 1) {   // InnerLoop1
                  squared_sum += a[i+j] * a[i+j];
            }
            rooted_sum = sqrt(squared_sum);
            for (j = 0; j < vect_size; j += 1) {   // InnerLoop2
                  a[i+j] /= rooted_sum;
            }
      }
}
```

Once again, assume we are working with 24 bit physical addresses and the byte-addressed machine from above. Consider a 256 B direct mapped cache with 32B blocks, whose valid bits are all set to 0. Doubles are 8 bytes, and the doubles stored in the array a are double-word aligned (e.g. at addresses 0, 8, 16…).

a) Suppose that we set `vect_size` to be 32 such that our vectors have 32 components (numbers) in them. We execute `normalize(32);` Over the entire execution, what is the **hit rate** of:

Inner Loop1: ⅞

Block size is 32B which holds 4 doubles. We compulsory miss the first double and then we get 7 hits. The cycle repeats and the hit rate never changes.

Inner Loop2: 100%

Given that loop 1 occurs, the four requested doubles in the block will be in the cache when loop 2 runs. Thus all the accesses will hit.

b) The cache is reinitialized (it is cold again). What is the smallest value of `vect_size` such that we have the **maximum** possible **hit** rate of Inner Loop1 over the entire execution of `normalize(size);`? What is the maximum possible hit rate you found?

`vect_size`: 1

Because we pull in the entire block size when we make one array access, having a vect_size of 1 will cause you to have a compulsory miss and then a hit, and the next 3 iterations of loop 1 will hit.

maximum possible hit rate: ⅞

Idea to take away: in this problem, our hit rate is capped by the block size.

c) Given that you are allowed to change a single parameter about the cache or the code, from the following options, what can you do to increase the hit rate of Inner Loop1? Circle all answers you believe are correct.

i) Increase cache size to be the array size
ii) Make the cache fully associative
iii) Double the block size to 64B
iv) Change the line `squared_sum += a[i+j] * a[i+j];` to be

```
double val = a[i+j];
squared_sum += val * val;
```

As stated above, the only solution is to increase the block size since that is what determines our hit rate for small values of vect_size.

Q7: Performance Programming (10 points)

1) For each snippet of code below, **fill in the circles for all outputs** that could be printed after running the code. You may assume that the function `printf` is atomic and that OpenMP parallelizes "for" loops using a blocking scheme as opposed to an adjacent scheme where threads process lower index values first. For both parts, assume there are **2 threads used to parallelize the loop**.

a.

```
#pragma omp parallel for
for (int i = 0; i < 4; i++) {
      printf("%d ", i);
}
```

Outputs (Fill in ALL possible outputs):

- ● 0 1 2 3
- ○ 0 1 3 2
- ● 0 2 1 3
- ● 2 0 1 3
- ○ 2 1 3 0
- ○ None of the outputs listed are possible

We know that one thread will do indices 0 and 1, while the other will do indices 2 and 3, and the threads will process the indices in that order. This gives us the following possible orderings:

1. T1, T1, T2, T2 → 0, 1, 2, 3
2. T1, T2, T1, T2 → 0, 2, 1, 3
3. T1, T2, T2, T1 → 0, 2, 3, 1
4. T2, T2, T1, T1 → 2, 3, 0, 1
5. T2, T1, T2, T2 → 2, 0, 3, 1
6. T2, T1, T1, T2 → 2, 0, 1, 3

Thus, only "0 1 3 2" and "2 1 3 0" are not possible.

b.

```
int x = 0;
#pragma omp parallel for
for (int i = 1; i < 3; i++) {
      x += i;
      printf("%d ", x);
}
```

Outputs (Fill in ALL possible outputs):

- ● 1 2
- ● 2 2
- ● 1 3
- ● 3 3
- ● 1 1
- ○ None of the outputs listed are possible

There is a race condition on the variable x. We have to consider the following possible scenarios:

a. Thread 1 and 2 concurrently execute the incrementing of x, and both load in the value 0 for x. Then, Thread 1 runs fully, setting x to 1 and then printing 1. Thread 2 will then load 2 into x (as it did x = 0 + 2), so it prints 2. This makes the output: 1 2. As note, we could also get the output 2 1 using a similar idea, but this was not an option.
b. Thread 1 runs fully, then thread 2 runs fully: this means thread 1 will print 1, as we incremented x by 1 and thread 2 will print 3, as we incremented x by 2. The output here would be: 1, 3
c. Thread 1 runs fully, then thread 2 runs fully: similar to the last scenario, we print 2 and then 3, making the output: 2, 3
d. Thread 1 runs, switched before print statement, then thread 2 runs. Then we print from thread 1 and thread 2 (order doesn't matter here, as x is already 3). The value of x for thread 1 would be with both increments, so the output would be: 3, 3
e. Thread 2 runs, switched before print statement, then thread 1 runs, then we print from thread 2. Same as (c), so we get: 3, 3
f. Thread 1 and 2 simultaneously update x, but 2 is slightly after (both load x = 0 during the increment), so the value 2 is loaded into x (i.e. the x += 1 was overwritten). This means that the output would be: 2, 2
g. Thread 1 and 2 simultaneously update x, but 1 is slightly after (both load x = 0 during the increment), so the value 1 is loaded into x (i.e. x += 2 was overwritten). This means the output would be: 1,1

Thus, all options are valid.

2) The goal of the piece of code below is to print “0 1 2”. **Fill in one** of the following options regarding the code and its functionality when there are 3 threads used to parallelize the loop.

```
#pragma omp parallel for
for (int i = 0; i < 3; i++) {
      #pragma omp critical
      printf(“%d\n”, i);
}
```

○ Always works, faster than serial

● Sometimes Works

○ Always works, slower than serial

○ Never Works

The critical section here ensures that only one thread will execute the print at a time, but it does not enforce an ordering amongst the threads. Thus, the critical section is useless (printf was given to be atomic). This means that the output depends on the ordering in which the threads execute, which is non-deterministic, so it may be that the ordering causes “0 1 2” to print (runs Thread 1, Thread 2, and then Thread 3), but this may not be the case (i.e. “2 1 0” could be printed if the reverse order occurs). Therefore, this code sometimes works.

Q8: I have virtually no memory of this… (20 points)

We have 2 machines, A and B, both with a **4 entry TLB, 4 KiB pages, 16 MiB physical memory, and 4 GiB (32b address space) of virtual memory**. They both have the same TLB and Page Table state, shown below. The only difference between these 2 machines is that, to store additional data beyond the capacity of physical memory, machine **A uses a hard disk drive and machine B uses a SSD**. On both machines, we run the same program that has the following memory accesses made in the order given below.

Accesses:
R 0xA0000008
W 0xA0004FFC
W 0xA0001234

TLB:

VPN	PPN	Valid Bit	Dirty Bit	LRU
0x80003	0x649	1	0	0
0xA0001	0x61F	1	0	1
0xBFFEC	0x613	1	1	2
0xBFFEB	0x612	1	1	3

Page Table: *shown starting from index 0xA0000.*

PPN	Valid Bit	Dirty Bit
0x609	1	1
0x61F	1	0
0x620	0	0
0x625	1	1
0x629	0	0
0x62B	1	0
0x62E	0	0

1. Translate the virtual address 0xA0000008 to its corresponding physical address.

   0x609008

2. Let us now analyze how our accesses progress through memory.
   a. Consider the possible access progressions below. Next to each access, write the letter corresponding to its access progression.
      a. TLB Hit, neither Page Hit nor Page Fault
      b. TLB Hit, Page Hit
      c. TLB Hit, Page Fault
      d. TLB Miss, neither Page Hit nor Page Fault
      e. TLB Miss, Page Hit
      f. TLB Miss, Page Fault

   R 0xA0000008 E **W 0xA0004FFC** F **W 0xA0001234** A

b. Given the following times to access each component, calculate the latency of **W 0xA0002FFC** on Machine A and Machine B. Remember to include in your calculations, the time to translate addresses, the time to get the corresponding data at that address, and the time to update any necessary caches and/or tables. Hardware accesses do **NOT** happen in parallel (e.g. accessing the TLB and physical memory at the same time is not possible, one must happen first and then the next). There is no data cache for physical memory.

Accessing TLB (us)	Accessing Physical Memory (us)	Accessing HDD (us)	Accessing SSD (us)
0.001	0.1	3000	450

TLB Miss + Page Fault => Access TLB + Access PM (page table) + Access Disk/SSD + Update PM with data + Update Page Table (PM) + Update TLB = 3000.302 us (Disk) or 450.302 (SSD)

A: 3000.302us
B: 450.302us

3. Both machines now starts running a new program that reads all of a 1 MiB file, which is stored entirely on disk. Assume that this program's page table has no virtual memory mappings for this file.
   a. The file is memory mapped through the virtual memory system, and the program simply touches every page in the file (e.g. reads 1 byte from each page). How many page faults would occur from reading this file?
   2^20 B (file) / 2^12 B (per page) = 2^8 pages, each is a fault

   2^8

   b. How long would it take to process all of the page faults from part (a) on both systems? Use the same access times as in the previous section 2.b. You may leave your answer as an unsimplified expression, and may use PF in place of your answer to part (a).

   A: PF * 3000.302

   B: PF * 450.302

   c. Now assume the file is laid out contiguously on disk and the program instructs the operating system to load the entire 1 MiB file in a single request. If there is a sustained transfer rate of 20 MiB/us for both systems, how long does this take to process for system A? How long for system B?
   Time to find first page + Time to continuously read in file = Latency from Part 2 + 1 MiB / (20MiB/us) = 3000.302 + 0.05 AND 450.302 + 0.05

   A: 3000.352us
   B: 450.352us

Q9: I/O (15 points)

An important advantage of interrupts over polling is the ability of the processor to perform other tasks while waiting for communication from an I/O device. Suppose that a **1 GHz processor** needs to read 1000 bytes of data from a particular I/O device. The I/O device supplies 1 byte of data every 0.02 ms. The time to process the data and store it in a buffer is negligible.

a) Assume a polling iteration takes 60 cycles. If the processor detects that a byte of data is ready through polling:

1. How many cycles does it take for the I/O device to supply 1 byte of data?

   _20,000__

2. How many polling iterations does it take to read 1 byte of data? (round up to an integer)

   ___334____

3. How many cycles does it take to read the 1000 bytes of data?

   _20,040,000___

b) If instead, the processor is interrupted when a byte is ready, and the processor spends the time between interrupts on another task, how many cycles of this other task can the processor complete while the I/O communication is taking place? The overhead for handling an interrupt is 2000 cycles.

18,000,000

c) The advantage of polling however arises when data rates become very large so that the interrupt overhead becomes substantial and at some point the system simply can't keep up. What is the data arrival time (in ms) at which point an interrupt-driven I/O scheme on this computer can't keep up with the data coming in? The overhead for handling an interrupt is 2000 cycles.

2000 / 1GHz = 0,002ms_____

Q10: Potpourri (5 points)

a) One byte of data is encoded in Hamming ECC with single-error correction. The result is the 12-bit number **`0xdd3`**. What was the original byte of data? [Remember to check that the data is correct!]
1111 1101 0011
1 2

1110 0011

b) Fill in the bubble for all statements that are True:

- ● RAID 5 has better fault tolerance than RAID 0
- ○ RAID 5 has better fault tolerance than RAID 1 (nope, same)
- ● RAID 0 can store data more compactly than RAID 1 (no redundancy)
- ● RAID 5 can store data more compactly than RAID 1 (more efficient)
- ○ RAID 0 reads faster than RAID 1 (either choice is fine, multi-small reads are better for RAID1 while single reads are better for RAID 0),
- ○ RAID 5 writes faster than RAID 1 (nope, Raid5 needs 2x read first)

c) Two machines are being considered for purchase. The SuperDuper machine has a Mean Time to Failure (MTTF) of 24 hours and a Mean Time to Repair (MTTR) of 6 hours, while the UnbelievablyGreat machine has a MTTF of 4 hours and a MTTR of 1 hour.

A. What's the availability of the SuperDuper Machine?

0.8

B. What's the availability of the UnbelievablyGreat Machine?

0.8

d) In which step of CALL is machine code first generated?

- ○ Compiler
- ○ Assembler
- ○ Linker
- ○ Loader