# Problem 1 (24 points)

Provide **short** answers to the following questions.

A. An operation OP has the following characteristics:

- OP Latency = 7 cycles
- OP Cycles/issue = 2

Derive a lower bound on the computation time (in cycles) for the following computation

```
x[0] OP x[1] OP x[2] OP ... OP x[n-1]
```

Lower bound: $7+(n-2)2$ cycles

B. Consider the following program adding two vectors of length $n$:

```
void vectorsum(double *x, double *y, double *z, int n)
{
  int i;
  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];
}
```

If we assume a CPU that can complete 1 addition every cycle, this computation will take at least $n$ cycles. Assume that $x, y, z$ are all in a cache. The bandwidth between this cache and the CPU is 1 double/cycle. Derive a lower bound on the computation time (in cycles) based on this information.

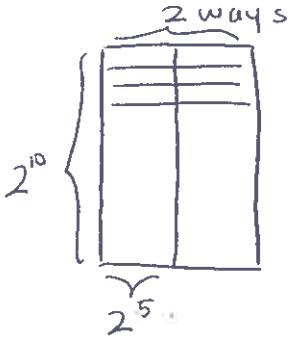
Lower bound: $3n$ cycles

C. What is the advantage of write back caches?

Minimizes writes to memory when there are many writes to the same memory location.

D. A system with 32-bit physical addressing mode, has a 2-way set associative cache with a capacity of 64KB and block size of 32B. Derive how the address is divided:

- How many bits are used for block offset? 5
- How many bits are used for the set index? 10
- How many bits are used for tag? 17



E. For what type of cache will there never be a conflict miss?

Fully associative cache.

F. Why is loop unrolling often useful in improving the performance of numerical code?

It enables other optimizations, for example scalar replacement or reassociation

G. Consider the following c code:

```
double * func(const double N, double *x, int size){
for (i=1; i<size ; i++)
  x[i]=(x[i]/size)*N;
}
```

The previous loop can be optimized as follows:

```
double * func(const double N, double *x, int size){
double tmp = N/size;
for (i=1; i<size ; i++)
  x[i] = x[i]*tmp;
}
```

gcc will not perform this optimization. What could be the reason?

Floating point multiplication is not associative.

H. Two algorithms $A_1$ and $A_2$ solve the same numerical problem and involve only floating point additions and multiplications. $A_1$ has higher performance (measured in flop/s) than $A_2$. Which algorithm is faster and why?

One cannot say. A1 could require more operations and hence be slower.

# Problem 2 (22 points)

Consider the following Matlab function implementing a fast Fourier transform (FFT). The input and output are both vectors of length $n$.

```
% x and y are vectors of length n
function y = fft(x)

n = length(x);

% allocate the result y which is a column vector
y = zeros(n,1);

% base case
if n == 1                          a1 = 0
    y(1) = x(1);                   m1 = 0
    return
end

m = n/2; % ignore in the cost computation

y(1:m) = x(1:m) + x(m+1:n);
y(m+1:n) = x(1:m) - x(m+1:n);      } n adds

% compute_constants(m) returns a list of m constants
% the internal cost of this function can be ignored
y(m+1:n) = y(m+1:n).*compute_constants(m);   } n/2 mults.

t1 = fft(y(1:m));
t2 = fft(y(m+1:n));

y(1:2:n) = t1;
y(2:2:n) = t2;
return
```

The goal is to compute the floating point cost of this function (use the next page):

A. Compute the number of additions (subtractions also count as additions) required for size $n$.

   Number of additions: $n \log_2 n$

B. Compute the number of multiplications for size $n$.

   Number of multiplications: $\frac{1}{2} n \log_2 n$

**Show the derivation.**

The formula $f_k = ca^k + \sum_{i=0}^{k-1} a^i s_{k-i}$ solving $f_0 = c,\ f_k = af_{k-1} + s_k,\ k \geq 1$ may be helpful.

Number of additions:

$$a_n = 2a_{n/2} + n \quad \longleftrightarrow \quad A_k = 2A_{k-1} + 2^k, \quad A_0 = 0$$

$$A_k = 2^{k-1} + 2^k, \quad A_0 = 0$$

$$A_k = \sum_{i=0}^{k-1} 2^i 2^{k-i}$$

$$= \sum_{i=0}^{k-1} 2^k = k \cdot 2^k$$

$$a_n = n \log_2(n)$$

Number of multiplications:

$$m_n = 2m_{n/2} + n/2 \quad \longleftrightarrow \quad M_k = 2M_{k-1} + 2^{k-1}, \quad M_0 = 0$$

$$M_k = 2M_{k-1} + 2^{k-1}, \quad M_0 = 0$$

$$M_k = \sum_{i=0}^{k-1} 2^i \cdot 2^{k-i-1}$$

$$= \sum_{i=0}^{k-1} 2^{k-1} = k \cdot 2^{k-1}$$

$$m_n = \frac{1}{2} n \log_2(n)$$

## Problem 3 (24 points)

Consider a direct mapped cache of size 64K with block size of 16 bytes. You will calculate the miss rate for the following code using this cache. Remember that `sizeof(int) == 4`. Assume that the cache starts empty and that the local variables `i, j` and computations involving them take place completely within the registers.

**Note: It helps to draw the cache. Provide some detail so we see how you did it.**

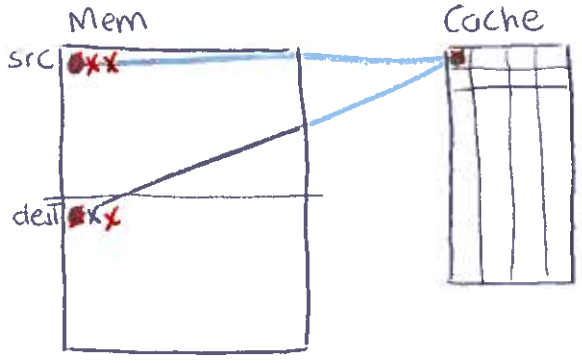
A. Now consider the following code to copy one matrix to another. Assume that the `src` matrix starts at address 0 (i.e., `src[0][0]` is mapped to the first element of the first cache block) and that the `dest` matrix immediately follows it.

```
void copy_matrix(int dest[ROWS][COLS], int src[ROWS][COLS])
{
  int i, j;
  for (i = 0; i < ROWS; i++)
    for (j = 0; j < COLS; j++)
      dest[i][j] = src[i][j];
  }
}
```

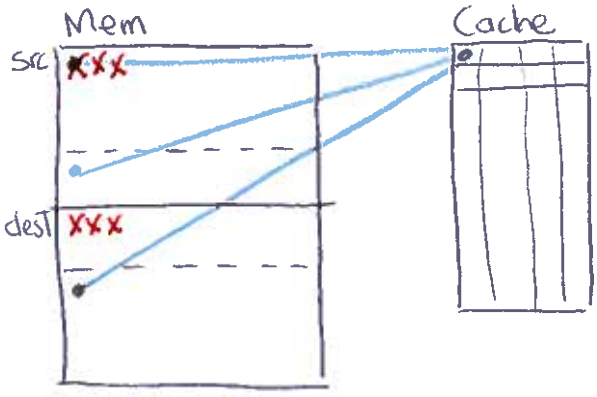
- Every block contains 4 elements
- The cache stores 128 x 128 elements.

1. What is the cache miss rate if ROWS = 128 and COLS = 128?
   Miss rate = 100 %
2. What is the cache miss rate if ROWS = 128 and COLS = 192?
   Miss rate = 25 %
3. What is the cache miss rate if ROWS = 128 and COLS = 256?
   Miss rate = 100 %

* For cases 1 and 3:




* For case 2:




X accesses

— memory - cache mapping.

dest[i][j] and src[i][j] are mapped to the same block.
Miss rate = 100%.

dest[i][j] and src[i][j] are mapped to different blocks.
Total misses = Total accesses / Block size
= 1/4 Total accesses

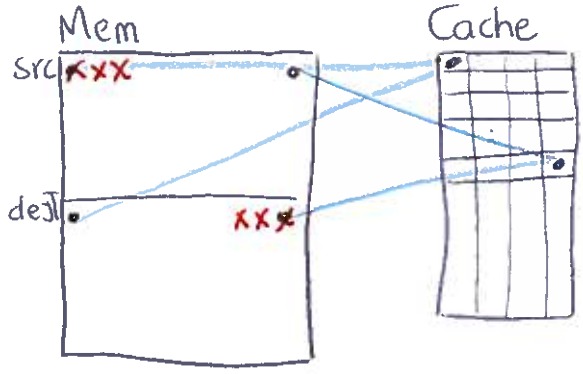
Miss rate = 25%.

B. Now consider the following two implementations (one is on the next page) of a horizontal flip and copy of the matrix. Again assume that the `src` matrix starts at address 0 and that the `dest` matrix immediately follows it.

```
void copy_n_flip_matrix1(int dest[ROWS][COLS], int src[ROWS][COLS])
{
  int i, j;
  for (i = 0; i < ROWS; i++)
    for (j = 0; j < COLS; j++)
      dest[i][COLS - 1 - j] = src[i][j];
  }
}
```

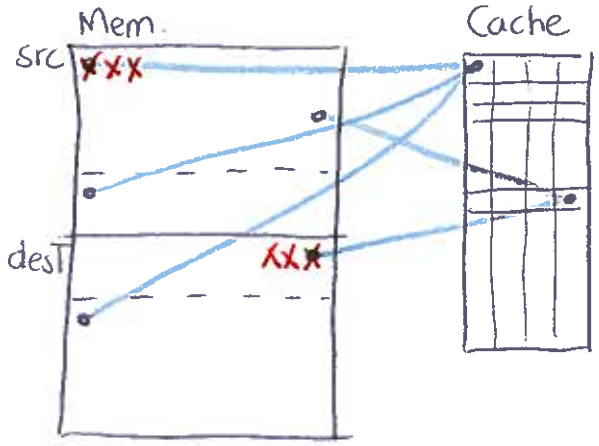
1. What is the cache miss rate if ROWS = 128 and COLS = 128?
   Miss rate = 25 %
2. What is the cache miss rate if ROWS = 128 and COLS = 192?
   Miss rate = 25 %

* For case 1:




dest[i][cols-1-j] and src[i][j] are mapped to different blocks.
Miss rate = 25%.

* For case 2:

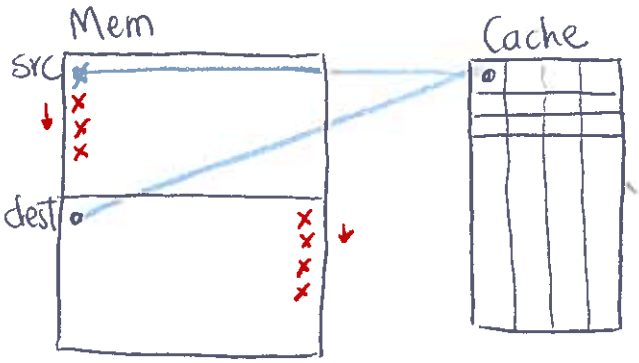



dest[i][col-1-j] and src[i][j] are mapped to different blocks.
Miss rate = 25%.

```
void copy_n_flip_matrix2(int dest[ROWS][COLS], int src[ROWS][COLS])
{
  int i, j;
  for (j = 0; j < COLS; j++)
    for (i = 0; i < ROWS; i++)
      dest[i][COLS - 1 - j] = src[i][j];
  }
}
```

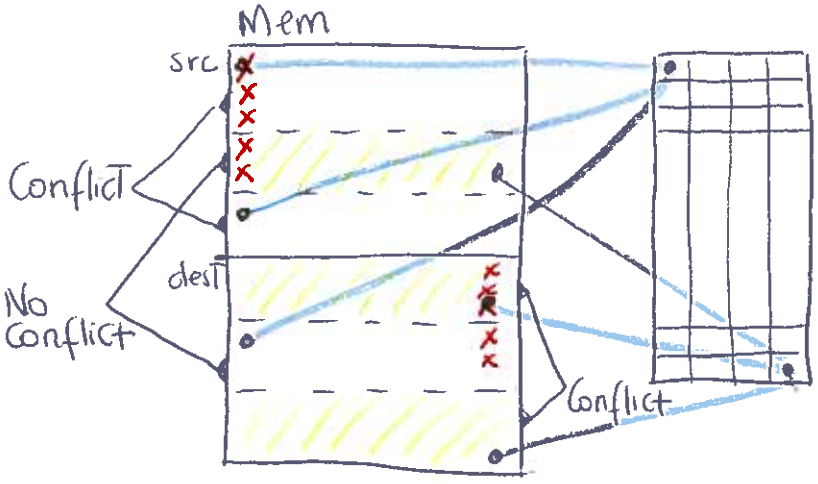
1. What is the cache miss rate if ROWS = 128 and COLS = 128?
   Miss rate = 25 %
2. What is the cache miss rate if ROWS = 128 and COLS = 192?
   Miss rate = 75 %

* For case 1:




There is no conflict
Miss rate = 25%.

* For case 2:




There are no conflicts in $\frac{2}{6}$ of the total number of accesses, and hence, 1 of every 4 of these accesses is a miss.

There are conflicts in $\frac{4}{6}$ of the total number of accesses, and hence, every of these accesses is a miss.

# Problem 4 (30 points)

We consider a computer with a direct-mapped cache with 256 sets. Each cache block fits 16 bytes, i.e., 2 doubles. Also, the computer has 4KB pages and one TLB with 256 entries. Consider the following program which performs some computation on one column of a 2D array.

```
typedef double data[512][512]

// K is the (positive) number of iterations
// *res will contain the result; ignore in the analysis
void important_algorithm(data a, int col, int K, double *res)
{
  int i, j;
  for (i = 0; i < K; i++)
    for (j = 0; j < 512; j++)
      access_compute(a[j][col], res);
}
```

* The cache stores 512 elements.

A. Determine

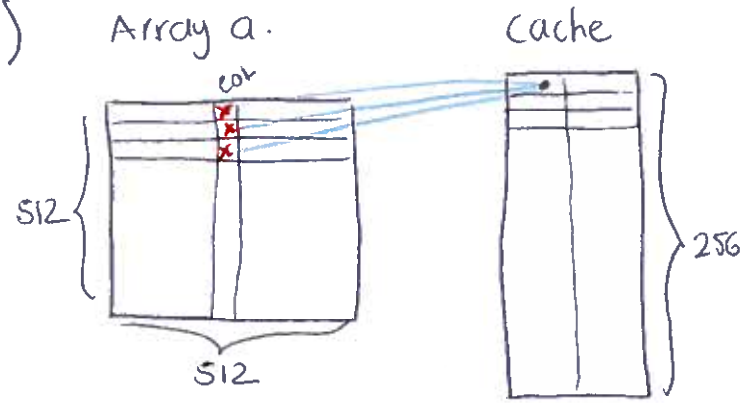
(a) the number of cache misses, → 512 K.

(b) the types of occuring cache misses, → Cold and conflict.

(c) the number of TLB misses → 512 K.

of the above program. Only consider the array a in this analysis. For simplicity you can assume that the column of a that you consider is cache-aligned, i.e., `a[0][col]` is mapped to the first element of the first cache block. Assume an empty cache and empty TLB. Again, it helps to draw the cache. Also, show some detail so we know how you did it.

(a)




All a[j][col] are mapped to the same block.

(c)

TLB accesses jump 512 doubles or 4 kB. Therefore, each access is a new page.

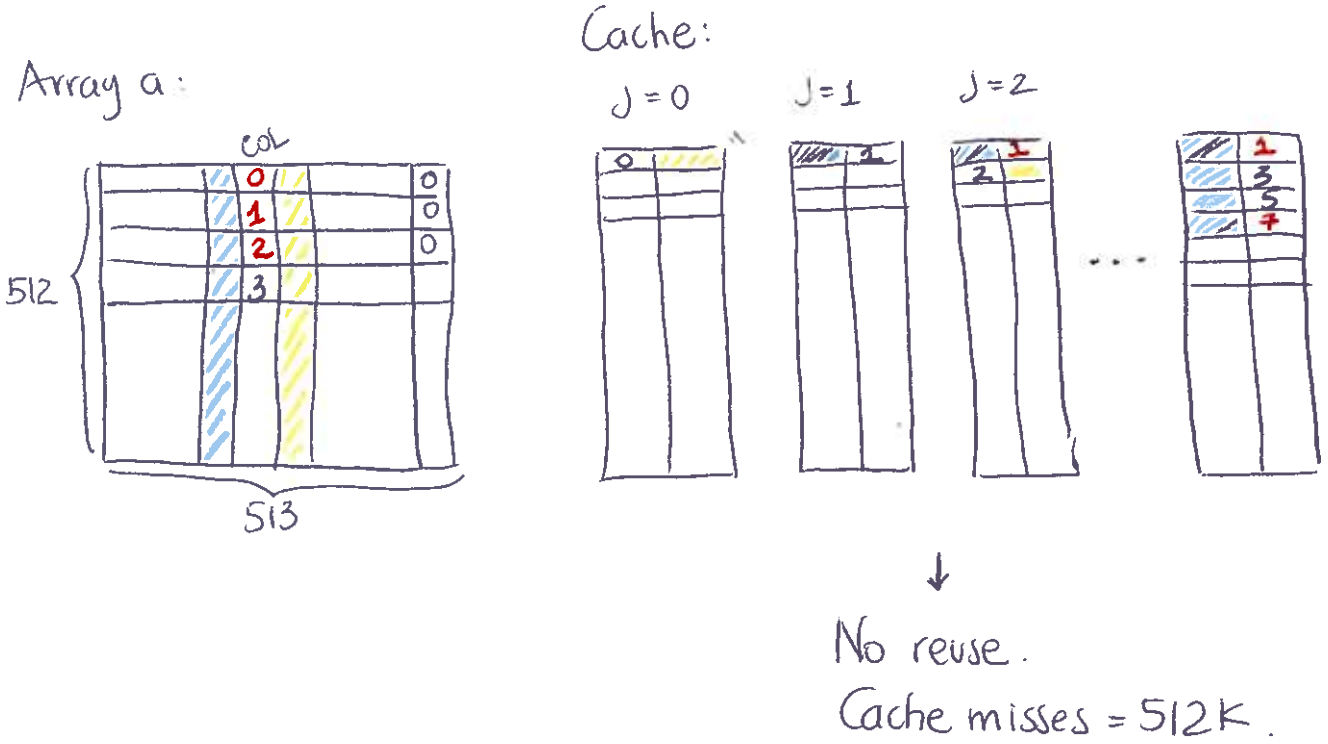
B. Because the programmer was not happy with the performance, she came up with an idea to change the way the data is stored. Specifically, she introduced a new data type

```
typedef betterdata[512][513]
```

and copied using the function

```
void copy(data a, betterdata b)
{
  int i, j;
  for (i = 0; i < 512; i++)
    for (j = 0; j < 512; j++)
      b[i][j] = a[i][j];
    b[i][512] = 0;
}
```

This technique is called zero-padding. Determine the number of cache misses when the program `important_algorithm` is applied to this new data type (`betterdata` instead of `data`). Again, it helps to draw the cache. Also, show some detail so we know how you did it. `a[0][col]` is again mapped to the first element in the first cache block.



C. Since the performance of the zero-padded version was still not as good as expected, the programmer decides to buy a computer with twice the cache size (1024 doubles instead of 512). Three cache options are available. Determine in each case the number of cache misses of `important_algorithm` (still working with `betterdata`):

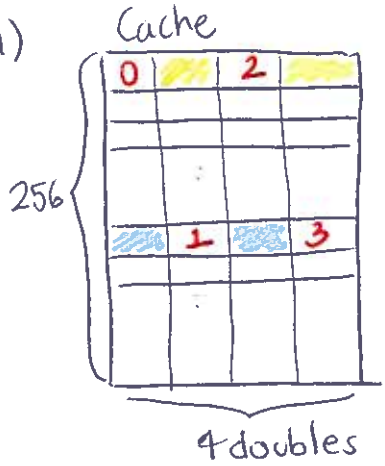
(a) Twice the cache block size → 512K

(b) Twice the number of sets → 512

(c) Two-way set associative → 512

Again, it helps to draw the cache. Also, show some detail so we know how you did it.

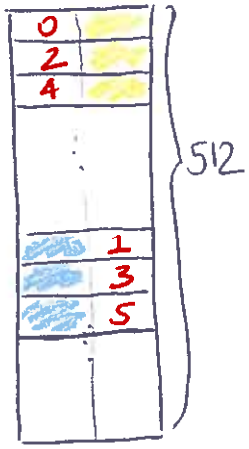
(a)




Same scenario: 2 elements map to the same block. The block is overwritten every time.

Cache misses: 512K

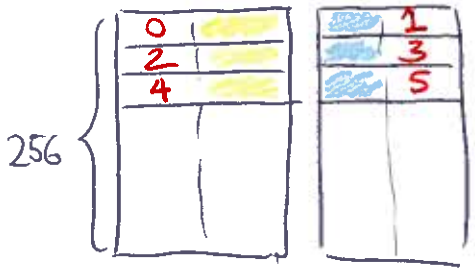
(b)




Blocks don't overlap. 1 block per element of Col. There will be cache misses in first iteration

Cache misses = 512

(c)




2 consecutive elements of col are mapped to the same set index, but they can be stored in the 2 ways.

Cache misses = 512.

D. With the (proper choice of) new cache the number of cache misses got indeed reduced. But by how much did the number of TLB misses get reduced and why?

The number of TLB misses does not change. Each element of col is mapped to a different page.