UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING

MIDTERM EXAMINATION, October 27, 2014
ECE454H1 - Computer Systems Programming
Closed book, Closed note
No programmable electronics allowed
Examiner – C. Amza and D. Yuan

Please Read All Questions Carefully! Please keep your answers as concise as possible.
You do not need to fill the whole space provided for answers.

*There are **16** total numbered pages, 7 Questions.*
You have 2 hours. Budget your time carefully!

Please put your FULL NAME, UTORid, Student ID on THIS page only.

Name: ____

UTORid: ____

Student ID: ____

Grading Page

DO NOT WRITE ON THIS PAGE

	Total Marks	Marks Received
Question 1	10	
Question 2	10	
Question 3	12	
Question 4	13	
Question 5	20	
Question 6	20	
Question 7	15	
Total	100	

Question 1: CPU architectures (10 points)

(1) Consider the following sequence of instructions

```
1 LW    F2, 4(R1)    # 1. load the word at RAM addr (R1+4) into register F2
2 MUL   F0, F1, F2   # 2. F0 = F1 x F2
3 ADD   F5, F0, F4   # 3. F5 = F0 + F4
4 ADD   F10, F1, F6  # 4. F10 = F1 + F6
5 ADD   F12, F13, F7 # 5. F12 = F13 + F7
6 MUL   F8, F9, F7   # 6. F8 = F9 x F7
```

Will this sequence of instructions benefit from out-of-order execution of the processor? If so, how can a processor reorder them to achieve the best performance? If not, why? (6 points)

Yes. The last three instructions do not have any dependencies with the first three, therefore should schedule them after the load instruction since the load might result in a cache miss.

Grading: it is OK to put one or two instructions btw. instruction 2 and 3 as well, but there have to be at least one instruction among 4-6 to be scheduled between 1 and 2.

```
LW    F2, 4(R1)    # 1. load into register F2 ←-----
ADD   F10, F1, F6  # 4. F10 = F1 + F6              |
ADD   F12, F13, F7 # 5. F12 = F13 + F7             |
MUL   F8, F9, F7   # 6. F8 = F9 x F7               |
MUL   F0, F1, F2   # 2. F0 = F1 x F2 --------------- read after write dep.
                                                   ^
ADD   F5, F0, F4   # 3. F5 = F0 + F4 --------------| read after write dep.
```

(2) “The CPI (cycles per instruction) on a processor with only *pipelining* and *branch-prediction (but no other features)* can be lower than 1.” Is it true? Why or why not? (4 points)

Answer: false. Even with perfect branch prediction and pipelining, the CPI cannot be less than 1.

Question 2: Amdahl's law (10 points)

The marketing department at your company has promised your customers that the next software release will show a 2X performance improvement. You have been assigned the task of delivering on that promise. You have determined that only 80% of the system can be improved. How much (i.e., what value of k) would you need to improve this part to meet the overall performance target?

Answer:

```
speedup = oldtime/newtime
        = oldtime/(0.8 x oldtime / k + 0.2 x oldtime)
        = 1/(0.8/k + 0.2) = 2;
1.6/k + 0.4 = 1;
1.6/k = 0.6

k = 2.67
```

Question 3: Alignment (12 points)

Consider the following C struct declaration:

```
typedef struct {
  char a;  // 1 byte
  long b;  // 8 bytes
  float c; // 4 bytes
  char d[3];
  int *e;
  short *f;
} foo;
```

(1). Show how `foo` would be allocated in memory on a 64 bit Linux system. Label *each byte* with the names of the various fields and clearly mark the end of the struct. Use an X to denote space that is allocated in the struct as padding. (6 points)

Answer:
e and f each has 8 bytes (64 bit pointers), therefore the entire struct should be aligned at 8 bytes.

```
|axxxxxxx|bbbbbbbb|cccc|d1d2d3x|eeeeeeee|ffffffff|
^        p+8      p+16 p+20    p+24     p+32
p: multiple of 8
```

(2). Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and clearly mark the end of the struct. Use an X to denote space that is allocated in the struct as padding. (6 points)

Answer: basic idea: put ‘a’ and d[3] together will achieve minimum space allocation.

```
typedef struct {
  long b;  // 8 bytes
  float c; // 4 bytes
  char a;  // 1 byte
  char d[3];
  int *e;
  short *f;
} foo;
```

```
|bbbbbbbb|cccc|ad1d2d3|eeeeeeee|ffffffff|
^        p+8  p+12    p+16     p+24
p: multiple of 8
```

Question 4: Compiler optimization (13 points)

Consider the following code:

```
void bar (int* result) {
  int i;
  *result = 0;
  for (i = 0; i < foo(); i++) {
    *result += data[i]; // data is a global integer array;
  }

  return;
}


int foo () {
   int a = 30;
   int b = 9 - (a / 5);
   int c = b * 4;
   if (c > 10) {
     c = c - 10;
   }
   return c * (60 / a);
}
```

(1) Discuss each optimization that a compiler will perform on this code. Write down the final code after optimizations. (7 points)

1. constant propagation: replace reference to ‘a’ with 30
2. this enables constant folding: b = 3
3. this enables another round of constant propagation: replace ref. to ‘b’ with 3
4. this enables constant folding: c = 12
5. another round of constant propagation: replace ‘c’ with 12
6. dead code elimination: if (c > 10) {c = c - 10}
   if() statement can be removed;
   c = 2;
7. foo() will become: return 4;
8. in bar(): foo() is a loop invariant, should be extracted from the loop
9. compiler can further inline foo() with 4; (it’s ok if students don’t list this optimization, assuming compiler is not aggressive enough to inline functions)

```
void bar (int* result) {
  int i;
  *result = 0;
  for (i = 0; i < 4; i++) {
    *result += data[i]; // data is a global integer array;
  }

  return;
}


int foo () { // will be inlined
   return 4;
}
```

(2) Discuss how you, as a programmer, can rewrite the code for better performance. (6 points)

Answer: get rid of pointer dereference in the loop body --- this will give compiler a better chance to promote the tmp into register.

No need to discuss loop unrolling here.

```
 void bar (int* result) {
   int i;
-  *result = 0;
+  int tmp = 0;
   for (i = 0; i < 24; i++) {
-     *result += data[i];
+    tmp += data[i]; // data is a global integer array;
   }
+  *result = tmp;
   return;
 }
```

Question 5: Loop unrolling (20 points)

Consider the following piece of code we discussed in the lecture:

```
void vsum4(vec_ptr v, int *dest)
{
  int i;                          // i => %edx
  int length = vec_length(v);     // length => %esi
  int *data = get_vec_start(v);    // data => %eax
  int sum = 0;                    // sum => %ecx
  for (i = 0; i < length; i++)
     sum += data[i];
  *dest = sum;
}
```

The loop code is compiled to the following assembly code:

```
.L24:                     # Loop code only:
  add (%eax,%edx,4),%ecx  # sum += data[i]
  inc %edx                # i++
  cmp %esi,%edx           # compare i and length
  jl .L24                 # if i < length goto L24
```

We make the following assumptions of the underlying architecture:

Operation	Latency (cycles)	Issue per cycle	Functional units
Integer (used by add, inc, cmp, and jmp)	1	0.33	3
Load	3	1	1

Note: "Issue per cycle" indicates the minimum number of cycles between two operations. For "load", the machine has 1 functional unit, <u>and it is pipelined</u>, therefore we can issue one load every cycle even if each takes 3 cycles to finish. For load, the result is written to the output register at the end of the 3rd cycle.

In addition, for the following statement (if you unroll the loop):

```
     sum += data[i+N]; // where N is a constant
```

we assume the underlying architecture will need *<u>2 "add" instructions</u>* and one "load". The first "add" will calculate `i+N`, then the "load" will read the content of `data[i+N]`, and the second "add" will add `data[i+N]` to `sum`.

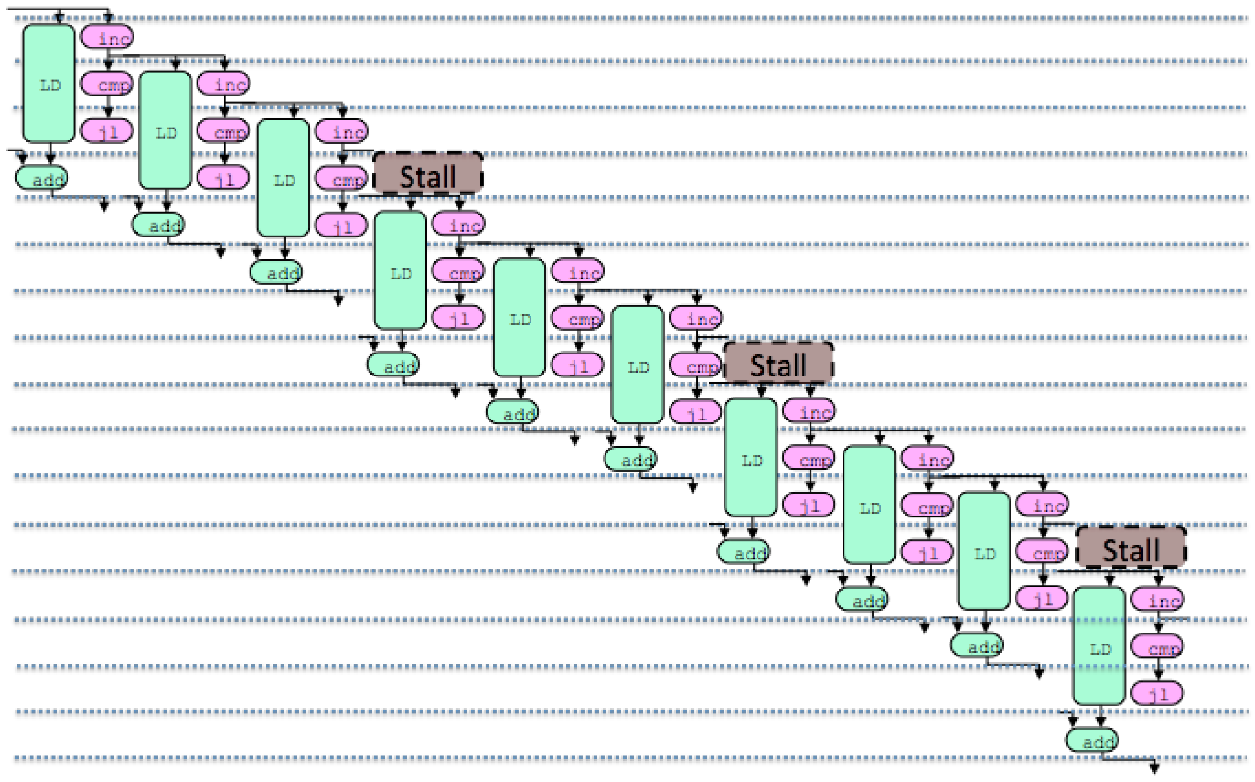
The machine is superscalar (i.e., can issue UNLIMITED number of different instructions each cycle), pipelined, has enough registers for any loop unrolling, and has accurate branch prediction.

You can make additional assumptions of the machine as long as they do not conflict with the above assumptions. In that case write down your assumptions.

(A). Please identify all the data dependencies between two consecutive iterations of the loop (without any loop unrolling). (4 points)

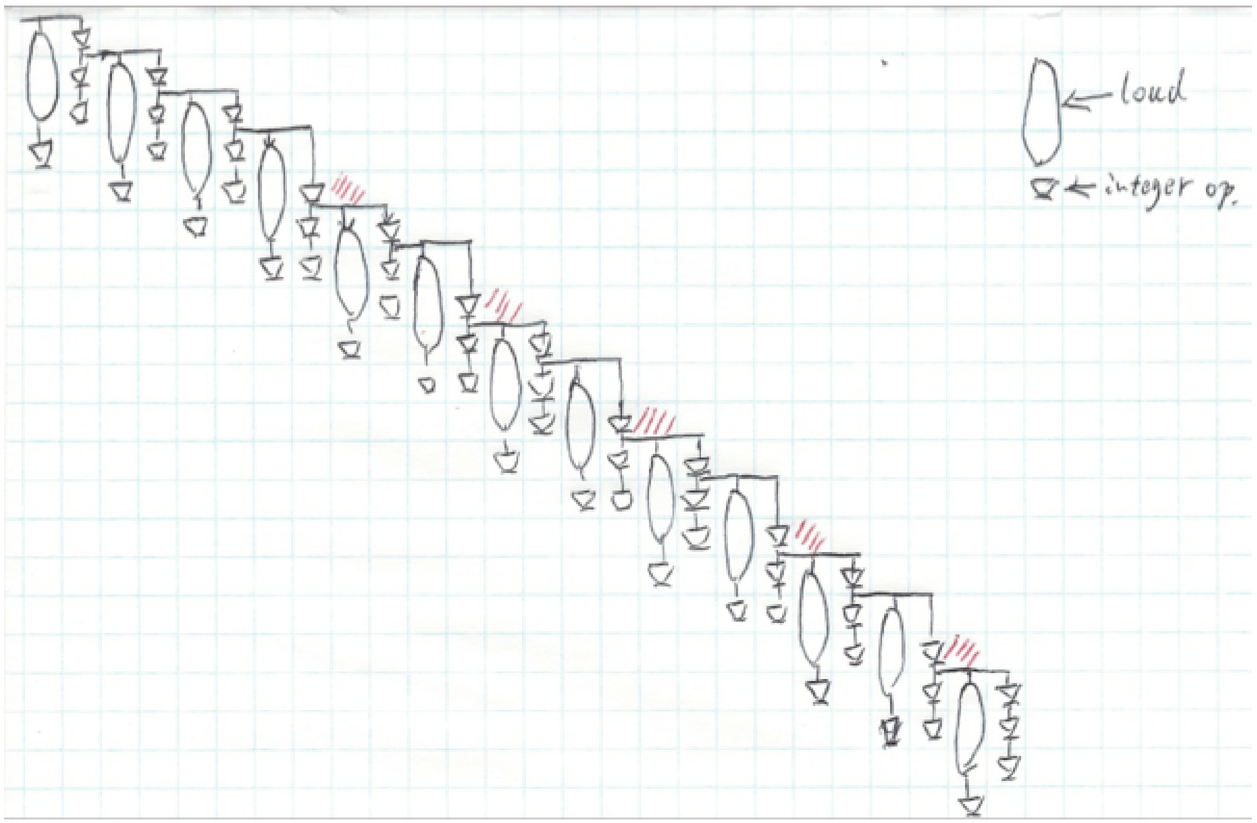
i++ (inc) depends on ‘i’ from the previous iteration;
load data[i] depends on ‘i’ from the previous iteration;
sum depends on ‘sum’ from the previous iteration;

(B). Based on the definitions from our lectures, what is the CPE for this function? Explain how you get this CPE based on the dependencies formed between the loop iterations and the number of functional units. Draw diagrams if necessary. (8 points)



In every three iterations, forced to stall once.
Every 4 cycles produce 3 elements: CPE = 1.33

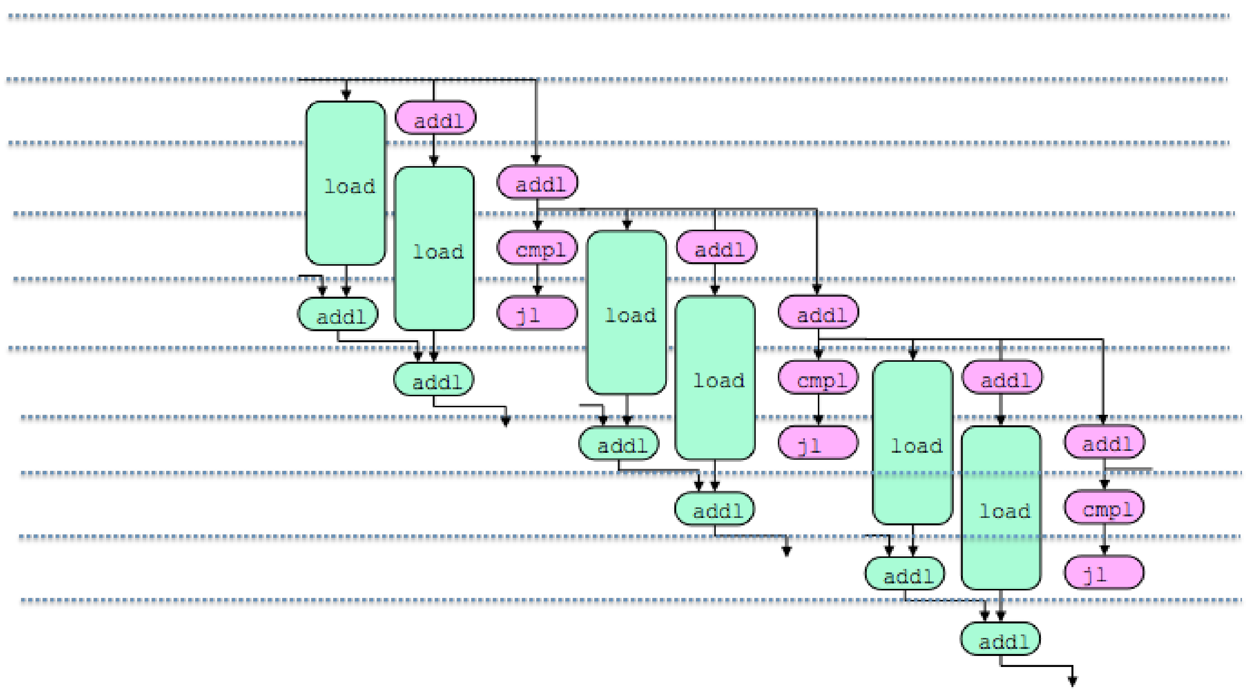
Note: in the 4th iteration above, if you assume the underlying HW is aggressive and does not stall the load, but only the inc/cmp/jl, you might get the following:



Every 2nd iterations, stall for 1 cycle.
Every 3 cycles produce 2 elements: CPE = 3/2 = 1.5

We gave full mark for both solutions.

(C). Can the CPE from B) be optimized by rewriting the code? If so, how, and what is the improved the CPE (explain your answer)? If not, why? (8 points)
Yes: unrolling 2 times can get the CPE to 1.
Cannot get the CPE to < 1 since there is only one load FU.



Question 6. Cache Miss Rates. (20 Points)

After watching the presidential election you decide to start a business in developing software for electronic voting. The software will run on a machine with a 1024-byte direct-mapped data cache with 64 byte blocks.

You are implementing a prototype of your software that assumes that there are 7 candidates. The C- structures you are using are:

```
struct vote {
  int candidates[7];
  char valid;
};
struct vote vote_array[16][16];
register int i, j, k;
```

You have to decide between two alternative implementations of the routine that initializes the array `vote_array`.

You want to choose the one with the better cache performance.

You can assume:

- `sizeof(int) = 4`
- `sizeof(char) = 1`
- `vote_array` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `vote_array`.

Variables `i, j` and `k` are stored in registers.

In the following, you are asked to provide miss rates as fractions. Please show your work i.e., write each miss rate as a fraction showing explicitly (the number of misses) / (the total number of accesses) in the given code fragment, for each fraction. You can then simplify the fraction.

(A). What fraction of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
  for (j=0; j<16; j++) {
    vote_array[i][j].valid=0;
  }
}

for (i=0; i<16; i++){
  for (j=0; j<16; j++) {
    for (k=0; k<7; k++) {
      vote_array[i][j].candidates[k] = 0;
    }
  }
}
```

Miss rate in the first loop: ___(16x8) / (16 x 16)__= 1/2_
Miss rate in the second loop: ___(16 x 8) / (16 x 16 x 7) __ = 1/14
Overall miss rate for writes to `vote_array`: __(16 x16) / (16 x 16 x 8)__ = 1/8

(B). What fraction of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
  for (j=0; j<16; j++) {
    for (k=0; k<7; k++) {
      vote_array[i][j].candidates[k] = 0;
    }
    vote_array[i][j].valid=0;
  }
}
```

Miss rate for writes to `vote_array`: _(16 x 8) / (16 x 16 x 8)__= 1/16

(C). Which of the two versions of the code A or B do you believe will run faster ? Please justify briefly.

The code in B will run faster because of lower miss rate, and also less loop overhead and smaller code footprint (so possibly better use of the instruction cache as well).

Question 7. Dynamic Memory Allocation (15 Points)

Consider an allocator with the following specification:

- Uses a single explicit free list.
- All memory blocks have a size that is a multiple of 8 bytes and is at least 16 bytes.
- All headers, footers, and pointers are 4 bytes in size
- Headers consist of a size in the upper 29 bits, a bit indicating if the block is allocated in the lowest bit, and a bit indicating if the previous block is allocated in the second lowest bit.
- Allocated blocks consist of a header and a payload (no footer)
- Free blocks consist of a header, two pointers for the next and previous free blocks in the free list, and a footer at the end of the block.
- All freed blocks are immediately coalesced.
- The heap starts with 0 bytes, never shrinks, and only grows large enough to satisfy memory requests.
- The heap contains only allocated blocks and free blocks. There is no space used

for other data or special blocks to mark the beginning and end of the heap.

- When a block is split, the lower part of the block becomes the allocated part and the upper part becomes the new free block.
- Any newly created free block (whether it comes from a call to free, the upper part of a split block, or the coalescing of several free blocks) is inserted at the beginning of the free list.
- All searches for free blocks start at the head of the list and walk through the list in order.
- If a request can be fulfilled by using a free block, that free block is used. Otherwise the heap is extended only enough to fulfill the request. If there is a free block at the end of the heap, this can be used along with the new heap space to fulfill the request.
- The allocator uses a first fit placement policy.

A) Given that the block size must be a multiple of 8 bytes, what is the minimum block size allowable under this scheme?

As per the problem statement the minimum blocks size allowable is 16 bytes because we need to store the header and also next and previous pointers whenever a block is free, so, at least 12 bytes, which rounded to a multiple of 8 bytes is 16 bytes.

B) Simulating Malloc

Below you are given a series of memory requests. You are asked to show what the heap looks like after each request is completed using a first fit placement policy. The heap is represented as a series of boxes, where each box is a single block on the heap, and the bottom of the heap is the leftmost box. Each malloc in a sequence of malloc's will search the existing blocks on the heap for a fit, and if no fit is found, will add a new block at the end of the current block sequence. In each block, you should write the total size (including headers and footers) of the block in bytes and either 'f' or 'a' to mark it as free or allocated, respectively. For example, the following heap contains an allocated block of size 16, followed by a free block of size 40.

16a	40f

Now, if we run a malloc(16) instruction, the heap will be like this:

16a	24a	16f

Assume that the heap is empty before the first instruction is run. You do not necessarily have to use all the boxes provided for the heap. The boxes corresponding to the first two instructions are already filled in to help you.

1. ptr1 = malloc(32);
2. ptr2 = malloc(16);
3. ptr3 = malloc(16);
4. ptr4 = malloc(40);
5. free(ptr3);
6. free(ptr1);
7. ptr5 = malloc(16);
8. free(ptr4);
9. ptr6 = malloc(48);
10. free(ptr2);

The Heap grows in this direction: →

1.	40a					
2.	40a	24a				
3.	40a	24a	24a			
4.	40a	24a	24a	48a		
5.	40a	24a	24f	48a		
6.	40f	24a	24f	48a		
7.	24a	16f	24a	24f	48a	
8.	24a	16f	24a	72f		
9.	24a	16f	24a	56a	16f	
10.	24a	40f	56a	16f		

C) Given that the block size must be a multiple of 8 bytes, determine the total amount of memory (in bytes), wasted due to internal fragmentation in the whole heap, at the end of the execution of all 10 instructions above (please show and explain all your work).

We count both headers and padding as wasted space within allocated nodes i.e., internal fragmentation.

Then, at the end of the execution of all 10 instructions, the heap contains only two allocated blocks of sizes 24 and 56.

The total internal fragmentation is 2 * (4 for header + 4 for padding) = 2 * 8 = 16 bytes.

This page is blank. You can use it as an overflow page for your answers.