Computer Science 2021
Spring 2016
Midterm Exam 2 (solutions)
April 15th, 2016
Time Limit: 50 minutes, 3:35pm-4:25pm

- This exam contains 6 pages (including this cover page) and 4 questions. Once we tell you to start, please check that no pages are missing.
- Before starting the exam, you can fill out your name and other information of this page, but don't open the exam until you are directed to start. Don't put any of your answers on this page.
- You may use any textbooks, notes, or printouts you wish during the exam, but you may not use any electronic devices: no calculators, smart phones, laptops, etc.
- You may ask clarifying questions of the instructor or TAs, but no communication with other students is allowed during the exam.
- Please read all questions carefully before answering them. Remember that we can only grade what you write on the exam, so it's in your interest to show your work and explain your thinking.
- Students often find that the quiz questions vary in difficulty. Your best strategy is usually to skim over all the questions, and then start working on the ones that look easiest. We also suggest that you leave time at the end to attempt every question, since we can't give you any partial credit if you leave a question blank.
- By signing below you certify that you agree to follow the rules of the exam, not to share exam material with other students before their exams, and that the answers on this exam are your own work only.

The exam will end promptly at 4:25pm. Good luck!

Your name (print): ______________________________

Your UMN email/X.500: ______________________________`@umn.edu`

Row letter (A-N): ______________ Seat number (101-130): ______________

Sign and date: ______________________________

Question	Points	Score
1	30	
2	20	
3	30	
4	20	
Total:	100	

Afternoon

1. (30 points) Y86-64 instruction design.

   In this question you'll design the hardware for a new Y86-64 instruction in the sequential (SEQ) implementation. The new instruction is the register indirect call instruction "`rcall *`rA". This is similar to an x86-64 instruction like `call *%rax`, and would be used by a C compiler to implement calling a function pointer. Like a normal call it saves the address of the following instruction as a return address by pushing it on the stack, but instead of the address of the function to then jump to being fixed, the address comes from the register rA. The instruction is two bytes long, with an opcode byte of 0x81 and a register specifier byte giving rA, and with 0xF in the rB position.

   Fill in the following table with what operations need to be performed in each of the stages of the SEQ implementation when the instruction is `rcall`. You may not need to put operations in every box.

`rcall *` rA	
Fetch	icode:ifun $\leftarrow$ $M_1[PC]$ rA:rB $\leftarrow$ $M_1[PC + 1]$ valP $\leftarrow$ PC $+ 2$
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[`%rsp`]
Execute	valE $\leftarrow$ valB $+ -8$
Memory	M_8[valE] $\leftarrow$ valP
Write back	R[`%rsp`] $\leftarrow$ valE
PC update	PC $\leftarrow$ valA

   Use the same notation that we did for the existing Y86-64 instructions, with $\leftarrow$ for giving a value to a signal, R[...] for the register file, and M_s[...] for a memory access of size s. Choose the signals you compute from the following:

Stage computed in	Signals
Fetch	icode, ifun, rA, rB, valC, valP
Decode	srcA, srcB, dstE, dstM, valA, valB
Execute	valE, Cnd
Memory	valM
Anywhere appropriate	R[...], M_s[...], PC

2. (20 points) Data dependencies.

   The listing below shows a sequence of Y86-64 instructions from a time-critical part of a program. There are five data dependencies between instructions in this sequence, through registers. Fill in the blanks with details about 5 dependencies: give the number of which instruction (higher numbered) depends on which previous instruction (lower numbered) via which Y86-64 register (that they both access). We've already filled in one for you.

```
iaddq   $8, %r8          # Instruction 1
mrmovq  8(%rax), %rbx    # Instruction 2
addq    %rbx, %rbx       # Instruction 3
subq    %r8, %rbx        # Instruction 4
rrmovq  %rbx, %rcx       # Instruction 5
rmmovq  %rcx, 24(%rax)   # Instruction 6
```

1. Instruction # 4 depends on # 3 via register `%rbx`

[✓ 2.] Instruction # 3 depends on # 2 via register `%rbx`

3. Instruction # 4 depends on # 1 via register `%r8`

4. Instruction # 5 depends on # 4 via register `%rbx`

5. Instruction # 6 depends on # 5 via register `%rcx`

Only one of these dependencies is a load-use hazard that will hurt the performance of the code when running on our pipelined Y86-64 implementation (with forwarding). Mark which dependency is a load-use hazard, by circling its number 1-5.

You can make the code run faster without changing its behavior by rearranging the instructions so that the load-use hazard does not require stalling. Fill in the blanks in the following description:

To avoid load-use stalling, move instruction number 1 to the position immediately after the current instruction number 2 .

3. (30 points) Data cache operations.

For this question we consider the memory system of a small embedded processor. The size of the physical address space is 4K bytes, and the memory is byte-addressable. The single-level cache is 3-way set associative, with a 2-byte block size and 24 total lines.

In the following table, all numbers are given in hexadecimal. The content of the cache is as follows (V = Valid, B0 = Byte 0, B1 = Byte 1):

3-way set associative cache												
Index	Tag	V	B0	B1	Tag	V	B0	B1	Tag	V	B0	B1
0	29	1	33	5D	4E	1	4A	EA	94	0	5A	18
1	E3	1	14	4D	7C	0	C6	36	A9	1	74	4E
2	1A	1	36	38	D4	1	F2	7D	F3	1	4C	3A
3	56	1	F7	74	05	1	ED	BE	52	0	56	68
4	47	0	21	9A	79	0	A2	34	94	1	D7	51
5	0A	1	3C	39	D8	1	11	6B	EF	1	FD	01
6	89	0	22	E2	3B	1	3A	86	33	1	35	65
7	32	0	1A	3B	15	1	D8	19	50	1	76	50

(a) Please indicate, by labeling the following diagram, the fields in an address that would be used to determine the following:

CO The cache offset
CI The cache set index
CT The cache tag

CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO

(b) For the given physical address, 0xF35, indicate the cache entry accessed and the cache byte value returned in hex. Indicate whether a cache miss occurs. If there is a cache miss, enter "unknown" for "Cache Byte Returned".

First, write the physical address in the same format as above, putting one bit per box:

1	1	1	1	0	0	1	1	0	1	0	1

Then, compute the following parameters of the cache access:

Parameter	Value
Cache Offset (CO)	0x1
Cache Index (CI)	0x2
Cache Tag (CT)	0xF3
Cache Hit? (Y/N)	Y
Cache Byte Returned	0x3A

(c) Counting the cache blocks as well as the tags and the valid bits, what is the total size of this cache in bits? (Please use decimal for the rest of this question.)

$(1 + 8 + 16) \times 24 = 600$ bits

(d) The L3 cache of a computer has a hit time of 15 cycles and a miss penalty of 300 cycles. Suppose that the hit rate for this cache, for a given application, is 95%. Considering just this cache, what is the average memory access time in cycles? (Recall that *average memory access time* $=$ *hit time* $+$ *miss rate* $\times$ *miss penalty*)

$15 + 0.05 \times 300 = 15 + 15 = 30$ cycles

Suppose we can improve the hit rate of the application to 98%. What is the new average memory access time?

$15 + 0.02 \times 300 = 15 + 6 = 21$ cycles

Notice how a small improvement in the hit rate makes a large (about 30%) improvement in performance.

4. (20 points) Optimizing cache usage.

You are designing a meal plan for a small university's dormitory, Gopher Hall. It has 4 floors with 8 rooms on each floor. The software will run on a machine with a 256-byte direct-mapped data cache with 32-byte blocks. You are implementing a prototype of your software that records the meal expenses (Breakfast, Lunch and Dinner) for each student in the dormitory each day. The C structures you are using are:

```
struct meal_expenses {
    float BLD_meals[3];
    int student_ID;
};

struct meal_expenses Gopher_Hall[4][8];
int i, j, k;
```

You have to decide between two alternative implementations of the routine that initializes the array `Gopher_Hall`. You want to choose the one with the better cache performance. You can assume:

- `sizeof(int) = 4` and `sizeof(float) = 4`
- `Gopher_Hall` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `Gopher_Hall`.
- Variables `i`, `j` and `k` are stored in registers.

(a) What percentage of the writes in the following initialization code will miss in the cache?

```
for (i=0; i<4; i++){
    for (j=0; j<8; j++) {
        Gopher_Hall[i][j].student_ID = 0;
    }
}
for (i=0; i<4; i++){
    for (j=0; j<8; j++) {
        for (k=0; k<3; k++) {
            Gopher_Hall[i][j].BLD_meals[k] = 0;
        }
    }
}
```

Total number of misses in the first loop: **16**

There are a total of 32 accesses. They are sequential, and two structures fit in each cache block, so the pattern will be an alternation in which one access misses, and then the next hits the same block. In total there will be 16 hits and 16 misses.

Total number of misses in the second loop: **16**

Because the cache is not large enough to hold the entire array, the second loop doesn't get any benefit from the first loop. The cache pattern is similar: a block of data is read in on a miss, and then all subsequent accesses to that structure and the next are hits in the same block. Thus there are again 16 misses.

Overall miss rate for writes to `Gopher_Hall`: **25%**

From the previous parts, the total number of misses is $16 + 16 = 32$. The total number of accesses is 32 from the first loop and $32 \times 3 = 96$ in the second loop, or alternatively 4 fields in each of 32 structures; $32 + 96 = 4 \times 32 = 128$. Thus the miss rate is $32/128 = 25\%$.

(b) What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<4; i++){
    for (j=0; j<8; j++) {
        for (k=0; k<3; k++) {
            Gopher_Hall[i][j].BLD_meals[k] = 0;
        }
        Gopher_Hall[i][j].student_ID=0;
    }
}
```

Miss rate for writes to `Gopher_Hall` is: **12.5%**

The total number of accesses is the same as in part (a), 128. But now the accesses form one sequential pass instead of two, which is more efficient. Each miss brings the contents of two adjacent structures into the cache, just like in either of the loops in (a). So there is only a total of 16 misses, for a rate of $16/128 = 12.5\%$.