Prelim 2

Computer Science 3410, Cornell University

Spring 2018, Professor Weatherspoon 3 May 2018

Solutions

- Write your NetID at the top of each physical page of the exam.
- Please turn off and stow away all electronic devices. You may not use them for any reason for the entirety of the exam. Do not bring them with you if you leave the room temporarily.
- This is a closed book and notes examination. You may use the 3-sided reference provided.
- To receive partial credit you must show your work. If you believe a question is open to interpretation, then please ask us about it! Please state any assumptions you make.
- There are 7 problems and 15 pages. Make sure you have the whole exam.
- You have **120 minutes** to complete 129 points. Use your time accordingly.

Problem	Topic	Points
1	Write your name and NetID	1
2	Multiple Choice	10
3	Calling Conventions	20
4	Memory Layout and Assembly	28
5	Caches	24
6	Address Translation	30
7	Synchronization	16
Total		129

Your Name _____

Your NetID _____

1. Write your name and NetID [1 pt]

2. Multiple Choice [10 pts] (parts a–e)

There is only 1 correct answer per question. If you write down multiple answers, we will only grade the first one.

(a) [2 pts] **MIPS.** The function `main` calls the function `power` with the instruction: `jal power`. That `jal` instruction is at address 1000. What happens to `$sp`?

A. Nothing; jal is unrelated to $sp.
B. $sp is set to 1000.
C. $sp is set to 1004.
D. $sp is set to the old $fp.

Correct Answer: **A**, `jal` is unrelated to `$sp`. Instead, `jal` sets `$ra` with the address of the next instruction, which is 4 more than 1000, so that the procedure will know where to return.

(b) [2 pts] **Operating Systems.** Which **one** of the following statements about the operating system is **true**?

A. Multiple copies of OS code reside in physical memory because every process keeps a copy of the kernel in its reserved address space.
B. A programmer can invoke the operating system by using an instruction that will trigger an interrupt.
C. The OS can interrupt user code via a system call.
D. The OS is always actively running on the CPU.
E. The OS uses its own stack when executing a system call on behalf of user code.

Correct Answer: **E**

(c) [2 pts] **Conflict Misses.** You can have a conflict miss in a fully associative cache.

A. True
B. False
C. Cannot be answered with the information given

Correct Answer: **B**

(d) [2 pts] **Asynchronicity.** Which of these exceptional events are asynchronous?

A. Software Exception
B. Hardware Interrupt
C. System Call
D. A and C
E. B and C

Correct Answer: **B**

(e) [2 pts] **Cache Coherence.** A single core machine that supports multiple threads can experience a coherence miss.

A. True
B. False
C. Depends on whether the threads are coarse- or fine-grained.
D. Cannot be answered with the information given

Correct Answer: **False**

3. Calling Conventions [20 pts]

The following assembly is the body of `arraylist_free` from lab compiled for a MIPS processor with a control delay slot:

```
BODY:
    SW   $a0, 28($sp)

    LW   $t0, 0($a0)
    BEQZ $t0, RESET
    NOP

    MOVE $a0,$t0
    JAL free
    NOP

RESET:
    LW $t0, 28($sp)
    SW $0,  4($t0)
    SW $0,  8($t0)
    SW $0,  12($t0)
```

Write the prologue and epilogue for this body. Use the class calling conventions.

The problem attempted to store argument $a0 in the space allocated for it in its parents stack frame. As a result, `SW $a0, 28($sp)` should have been `SW $a0, 24($sp)`. As a result, either solution below was accepted.

```
PROLOGUE:
    ADDIU $sp, $sp,    -24
    SW    $ra, 24($sp)
    SW    $fp, 20($sp)
    ADDIU $fp, $sp,    20
```

```
EPILOGUE:
    LW    $fp, 20($sp)
    LW    $ra, 24($sp)
    ADDIU $sp, $sp,    24
    JR    $ra
    NOP
```

```
PROLOGUE:
    ADDIU $sp, $sp,    -28
    SW    $ra, 28($sp)
    SW    $fp, 24($sp)
    ADDIU $fp, $sp,    24
```

```
EPILOGUE:
    LW    $fp, 24($sp)
    LW    $ra, 28($sp)
    ADDIU $sp, $sp,    28
    JR    $ra
    NOP
```

+4 for calculating the right stack frame size
+2 point per correct line

-2 no JR $ra
-2 for various errors such as transposed $ra and $fp on stack, not correctly update fp in prologue, no include NOP in epilogue, not correctly reset $sp in epilogue, not decrement $sp in prolgue, not store/load frame pointer, stored one argument to stack frame with stack size 28, etc.

4. Memory Layout and Assembly [28 pts] (parts a–b)

Suppose you have the following lines of code:

```
#define MAX_SIZE
int x;

int main()
{
    x = 5;
    int a = 6;
    int *b = (int*)malloc(MAX_SIZE);
    *b = a+x;
}
```

(a) [14 pts] **Memory layout.** Where are the following program components located? Choose one of the following regions of virtual memory in the full system layout:

A. Stack
B. Heap
C. Data
D. Text
E. None of the above

C `x`
A `a`
A `b`
B the address that `b` points to
D `malloc()`
D `main()`
E `MAX_SIZE`

+2 per correct answer

(b) [14 pts] **Assembly.** Translate the four lines of code in the `main` function (**no** prologue/epilogue needed) using the class calling convention. If the problem does not provide enough information for full translation in addressing, make reasonable assumptions consistent with the general memory layout.

Problem tests for knowledge of two things: difference between stack, global, and heap memory access, as well as how the memory is laid out in general. Sample solution optimizes away stack access, but could just use `$sp` offsets.

Assumption: x is located 0 bytes from `$gp`

Assumption: `MAX_SIZE` is 4 for 4 bytes
Assumption: Use callee-save $s registers for `a` and `x`. Alternatively, could have used caller-save $t registers, but would have needed to restore value after call to `malloc`

```
LI $s0, 5             # $s0 = x
SW $s0, 0($gp)        # x = 5
LI $s1, 6             # $s1 = a
MOVE $a0,MAX_SIZE     # int is 4 bytes
JAL malloc            # call malloc function
ADDU $t0, $s0, $s1    # $t0 = a + x; Note, this instruction is in delay slot of JAL
SW $t0, 0($v0)        # *b = a + x; $v0 is the address of b and
                      # $v0 is the address of memory that malloc returned
```

+2 for per line, or if equivalent but different implementation works
-1 if used `$s` or `$t` registers not according to calling conventions
-1.5 not loading 5 into a register
-1.5 not using global pointer ($gp) to store 5 in memory
-2 for varous problems such as not loading 6 into a register, not calling `JAL malloc`, not calculation `a+x`, not storing result into the `b*` pointer, etc.

5. Caches [24 pts] (parts a–d)

A program runs on a 1024 byte cache that has been just flushed. The table below lists the first memory addresses accessed, in order, and the results of the cache lookup. Answer the following questions about the cache. When asked for numeric responses, assume that the only valid answers are powers of 2.

Address	Cache Behavior
0x8000	MISS
0x801F	HIT
0x8020	MISS
0x803F	HIT
0x8040	MISS
0x8100	MISS
0x8000	HIT
0x8200	MISS
0x8000	HIT
0x8300	MISS
0x8000	HIT
0x8400	MISS
0x8000	MISS
0x8420	MISS
0x8020	HIT
0x8240	MISS
0x8040	HIT
0x8440	MISS
0x8040	HIT

(a) [6 pts] What is the length of a cache line? Justify your answer briefly.

32 bytes

Since the cache starts cold and address 0x801F hits, it must be part of the cache line from 0x8000. The next address 0x8020 misses, so it must be the start of the next cache line. Since the same happens between 0x803F and 0x8040, that cache line must end at 0x803F. Thus, a cache line is 32 bytes long.

Correct answer: +4
Reasonable explanation: +2

(b) [6 pts] Is this a direct-mapped cache? Justify your answer briefly.

No

The cache is 1024 bytes long so if it's direct mapped, consecutive accesses to 1024 byte offsets should always miss. However, this does not happen for the accesses to 0x8440 and 0x8040.

Correct answer: +2
Reasonable explanation: +4
Answering “yes”: 0

(c) [8 pts] What is the set-associativity of this cache? Justify your answer briefly. Assume that the replacement policy is FIFO (First in, First out).

Four-way

The access to 0x8400 evicts the element from 0x8000 since accesses to 0x8000 were hitting until the access to 0x8400 was made. By inspecting the accesses to 0x8100, 0x8200, 0x8300 and 0x8400, it is clear that the cache must be either 2 or 4 way associative. However, it cannot be 2 way since then it would be expected that the last access to 0x8040 would have been evicted by the immediately preceding accesses to 0x8240 and 0x8440.

Correct answer: +4
Reasonable explanation: +4
2-way set associative : +3
2-way set associative no explanation : 0
1-way set associative, direct mapped: 0

(d) [4 pts] Assume that the addresses listed are the lower bits of a full 32 bit address. What bits of the address correspond to the tag for this cache? Justify briefly.

Bits 8-31.

The lower 5 bits are the cache line address and the next three select one of the eight 4-way sets. The remaining 24 bits are the tag.

Rubric: Depends on the answer of the previous question, all or nothing.
-2 for minor off-by-one error in calculation, but correct field sizes

6. Address Translation [30 pts] (parts a–j)

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to 4-byte words.
- Virtual addresses are 24 bits wide.
- Physical addresses are 20 bits wide.
- The page size is 16kB (i.e.).
- The TLB is 4-way set associative with 16 total entries. Remember, the TLB is just like a cache except that it stores the Physical Page Number associated with a Virtual Page Number instead of the Data associated with an Address.

(a) [2 pts] What is the largest size (in bytes) of the virtual address space for a process?

24 bit virtual address means 2^{24} bytes = 16 MB.

(b) [2 pts] How large (in bytes) is memory/DRAM?

20 bit physical address means 2^{20} addressable bytes = 1 MB.

(c) [2 pts] How large (in bits) is the page offset?

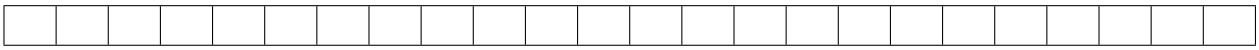
16KB = 2^{14} bytes, so 14 bits for the page offset

(d) [2 pts] How many pages are there in DRAM?

1 MB memory / 16KB per page = 2^{20} / 2^{14} = 2^4 = 64 pages

(e) [4 pts] The box below shows the format of a virtual address.

23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



Indicate (by specifying the bit indices) the fields that would be used to determine the following:

VPO The virtual page offset: 13-0

VPN The virtual page number: 23-14

+2 per correct answer

+1 per incorrect answer, but consistent with parts b and c

(f) [4 pts] The box below shows the same format for a virtual address.

23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Using the same virtual address above, indicate (by specifying the bit indices) the fields that would be used to determine the following:

TLBI The TLB index: 15-14

TLBT The TLB tag: 23-16

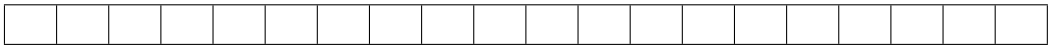
+2 per correct answer

+1 per incorrect answer, but consistent with part c

+1 per correct answer, but inconsistent with part c

(g) [4 pts] The box below shows the same format for a physical address.

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



Indicate (by specifying the bit indices) the fields that would be used to determine the following:

PPO The physical page offset: 13-0

PPN The physical page number: 19-14

+2 per correct answer

+1 per incorrect answer, but consistent with part c

+1 per correct answer, but inconsistent with part c

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 32 pages are as follows:

TLB			
Index	Tag	PPN	Valid
0	03	B	1
	06	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	06	4	1
	0B	1	1
2	2A	A	0
	11	1	0
	1F	8	1
	06	5	1
3	06	3	1
	3F	F	0
	10	D	0
	32	0	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	10	6	0
01	8	1	11	7	0
02	9	1	12	8	0
03	A	1	13	3	0
04	6	0	14	D	0
05	3	0	15	B	0
06	1	0	16	9	0
07	8	0	17	6	0
08	2	0	18	C	1
09	3	0	19	4	1
0A	1	1	1A	F	0
0B	6	1	1B	2	1
0C	A	1	1C	0	0
0D	D	0	1D	E	1
0E	E	0	1E	5	1
0F	D	1	1F	3	1

(h) [2 pts] Complete the **Virtual Address**: `0x19E37C`, by filling in one bit per box:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	1	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0

0001 1001 1110 0011 0111 1100

+2 for correct answer

(i) [6 pts] **Address translation.** For the above virtual address, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs. If there is a page fault, enter "–" for "PPN".

Parameter	Value
VPN	0x067
TLB Index	0x3
TLB Tag	0x19
TLB Hit? (Y/N)	N
Page Fault? (Y/N)	Y
PPN	0x—

+1 per correct answer or consistent with prior answers

(j) [2 pts] **Physical address.** Write physical address format (one bit per box). Also, write physical address in hex below boxes. If you had a page fault in the previous

part, you may not be able to fill in all the boxes.

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

page fault

+2 correct or consistent answer

+1 correct or consistent answer, but no/incorrect hex address (in the case of TLB hit or no page fault

NetID: _____

7. Synchronization [16 pts] (parts a–b)

Consider the following possible implementations of `mutex_lock(int *lock_addr)`, a function that obtains a lock located at address `lock_addr`. The lock is free when the value in memory is 0, taken when 1. `SC` returns 1 if successful. This function is meant to be called before entering a *critical section* of code.

```
        A.                      B.                      C.
      LI $t1, 1         try:  LI $t1, 1         try:  LI $t1, 1
try:  LL $t0, 0($a0)          LL $t0, 0($a0)          LL $t0, 0($a0)
      BNEZ $t0, try           SC $t1, 0($a0)          BNEZ $t0, try
      SC $t1, 0($a0)          BEQZ $t1, try           SC $t1, 0($a0)
      BEQZ $t1, try                                   BEQZ $t1, try


        D.                      E.                      F.
try:  ADDIU $t1, $t1, 1   try:  LI $t1, 1         try:  LL $t0, 0($a0)
      LL $t0, 0($a0)            LL $t0, 0($a0)          BNEZ $t0, try
      BNEZ $t0, try             SC $t1, 0($a0)          ADDIU $t1, $t0, 1
      SC $t1, 0($a0)                                    SC $t1, 0($a0)
      BEQZ $t1, try                                     BEQZ $t1, try
```

(a) [10 pts] Of the above versions of `mutex_lock`, which ones are functionally *correct*? Select all apply.

C and F

+5 for correct answer

-3 for each incorrect answer

(b) [6 pts] Of the correct implementations, which **one** would be your *first* choice and why?

Your argument why is key to getting points.

F is good because the spinning portion more quickly checks whether the lock is free. F is bad because when you finally see that the lock is free you have more instructions before you perform the store which means more chances that the lock could be grabbed by someone else.

C is good/bad for the opposite reasons of F. Also okay to make a power argumnet that an add might be more expensive than a load immediate. Really, any intelligent and correct answer will be accepted.

+2 chose one of the correct implementations

+4 reasonable explanation about the correct implementation

+2 reasonable explanation about the incorrect implementation

+0 wrong implementation and explanation was off or no explanation