Operating Systems/Architecture Candidacy Exam FA 2023

Name: _____

Instructions: This exam contains a collection of questions from two areas (*OS* and *Arch*). Each area has 100 points worth of questions. You will have **3 hours** to answer **65** points worth of questions **from each area** for a total of **130 pts.** Indicate which questions you are responding to **by circling the question on the cover sheet**.

Do NOT select more than 65 points worth of questions for an area – additional responses may be penalized.

Please provide answers in the space provided for that question (exams will be scanned into Gradescope); answers outside the provided space may not be graded.

The exam is closed book, so no notes, phones, etc. Good luck!

Questions		**Attempted**	**Score**
OS	Virtual Memory **VM-1** (5 pts); **VM-2** (5 pts); **VM-3** (5 pts); **VM-4** (5 pts); **VM-5** (5 pts)		
	Kernel **K-1** (5 pts); **K-2** (5 pts); **K-3** (5 pts); **K-4** (5 pts); **K-5** (5 pts)		
	Storage Systems **SS-1** (5 pts); **SS-2** (5 pts); **SS-3** (5 pts); **SS-4** (5 pts); **SS-5** (5 pts)		
	Concurrency and Synchronization **CaS-1** (5 pts); **CaS-2** (10 pts); **CaS-3** (5 pts); **CaS-4** (5 pts)		
Arch	Caching **C$-a** (5 pts); **C$-b** (5 pts); **C$-c** (5 pts); **C$-c** (10 pts)		
	Scheduling, Dependencies, and Hazards **SDH-a** (5 pts); **SDH-b** (10 pts); **SDH-c** (10 pts)		
	Performance, Efficiency, and Parallelism **PEP-a** (5 pts); **PEP-b** (5 pts); **PEP-c** (5 pts); **PEP-d** (10 pts)		
	Multicore and Parallel Processors **MaPP-a** (5 pts); **MaPP-b** (10 pts); **MaPP-c** (10 pts)		
Total		/ 130	

Virtual Memory

(**VM-1**: 5 points) Typical CPU architectures support multi-level page tables. What is the main advantage of using multi-level (as opposed to single-level) page tables? Explain your answer.

(**VM-2**: 5 points) Typically, a special "reference" ("access") bit is set by the CPU in a page table entry, whenever there is a first access against a corresponding memory page. What is the benefit of this?

(**VM-3**: 5 points) Aside from leveraging secondary storage (HDD, SSD, etc.) to increase the total memory size, there are other important benefits of virtual memory in modern operating systems. Please describe at least one such benefit.

(**VM-4**: 5 points) How many levels (in the page table) are required for a 64-bit CPU with 4KB pages to map 16GB of (total) virtual memory, assuming that each page table entry in every level occupies 8 bytes? Please show step-by-step how you arrived at the answer.

(**VM-5:** 5 points) What kind of overheads may exist when using multi-level page tables? What is the main benefit of using TLB (translation-lookaside buffer)?

Kernel

(**K-1:** 5 points) Many modern operating systems prefer preemptive schedulers (e.g., round robin) to non-preemptive schedulers (e.g., FIFO). Why is that and what is the main benefit of preemptive schedulers? Explain your answer.

(**K-2:** 5 points) While running in user mode, if an interrupt occurs, x86-64 will automatically switch the current (user) stack to a kernel stack (which is specified in the Task State Segment - TSS). Why should the stack switch be done by the hardware rather than the interrupt handler itself?

[Hint: When a program is interrupted, where is the current program counter stored? Is it going to be placed on a user stack or kernel stack? Why?]

(**K-3:** 5 points) What is the main difference between traps and interrupts? Are system calls closer to traps or interrupts? Explain your answer.

(**K-4:** 5 points) Why do typical CPU architectures support at least two modes of execution: privileged (kernel) and unprivileged (user) modes? Please connect your answer to traps and user-mode execution.

(**K-5:** 5 points) Why does the OS need to create a separate page table for every process? Can the kernel-space portion of the page table be shared across different processes? Please explain why "yes" or why "no". (You can disregard any recent hardware bugs that need special workarounds in that regard.)

Storage

(**SS-1:** 5 points) Please list at least two examples of what a typical file system inode stores.

(**SS-2:** 5 points) Suppose you have: /usr/bin/ld, which was originally created as a **soft (symbolic)** link to /usr/local/bin/ld (a regular executable file). How many inodes need to be read from the disk when a user executes /usr/bin/ld? Note: If the same inode has to be accessed twice then also count it twice. You can assume that all file paths are within the same file system and no inodes are cached (previously present) in memory. You should also count the topmost (root) inode and assume that it is not cached.

(**SS-3:** 5 points) What is the main advantage of using interrupts rather than polling when dealing with secondary storage (e.g., an HDD)? Explain your answer.

(**SS-4:** 5 points) Please compare fsck-based (file system checker) and journaling approaches to maintain file system metadata consistency. (Which approach is more reliable in terms of metadata consistency? Which approach allows faster recovery? Explain your answer.)

(**SS-5:** 5 points) Please list at least one benefit of using RAID (Redundant Arrays of Inexpensive Disks) and give an example of the corresponding RAID scheme (i.e., explain how it works).

Concurrency and Synchronization

(**CaS-1:** 5 points) Is the following code deadlock-free? Please explain why "yes" or why "no".

Thread 1	Thread 2

`pthread_mutex_lock(&B);`
`// Critical section`
`pthread_mutex_lock(&A);`
`pthread_mutex_unlock(&B);`
`// Another critical section`
`pthread_mutex_unlock(&A);`
	`pthread_mutex_lock(&A);`
	`// Critical section`
	`pthread_mutex_lock(&B);`
	`pthread_mutex_unlock(&A);`
	`// Another critical section`
	`pthread_mutex_unlock(&B);`

(**CaS-2:** 10 points) Suppose we need to solve a producer-consumer problem using a bounded queue, i.e., we only allow at most N elements to be produced. One possible solution that uses semaphores is listed below.

Sender	Receiver

`sem_wait(&sem_send);`
`pthread_mutex_lock(&lock);`
`// append to the queue`
`pthread_mutex_unlock(&lock);`
`sem_post(&sem_recv);`
	`sem_wait(&sem_recv);`
	`pthread_mutex_lock(&lock);`
	`// retrieve from the queue`
	`pthread_mutex_unlock(&lock);`
	`sem_post(&sem_send);`

a) [5pts] Please explain what values *sem_send* and *sem_recv* need to be initialized to assuming that N is the maximum number of elements to be produced at any point of time.

(b) [5pts] Please also explain why pthread_mutex_lock/pthread_mutex_unlock are still needed when manipulating the queue despite having two semaphores already.

(**CaS-3:** 5 points) Is this a correct implementation of a lock? Please explain why "yes" or why "no".

```
struct lock_t {
   size_t value; // Initialized to 0
};


void lock(lock_t *lock) {
    while (lock->value!= 0)
        { } // busy-waiting
    lock->value = 1;
}

void unlock(lock_t *lock) {
    lock->value = 0;

}
```

(**CaS-4:** 5 points) Can you elide/omit locks when running a (lock-based) parallel program on a uniprocessor computer (e.g., the computer only has one core/CPU)? Explain why "yes" or why "no".

Caching

a) (5 pts) Average Memory Access Time (AMAT) is distinct from memory stall latency. Describe a concrete, specific scenario where increasing AMAT could **improve** execution time.

b) (5 pts) On a machine with 64-bit addresses, given a virtually indexed, virtually tagged cache with the following configuration [32KB capacity, 4 way associative, 128 byte blocks], what are the **tag**, **index**, and **block offset** for a memory access to address 0xFEEDEEC5

c) (5 pts) Describe a) the conditions under which an inclusive cache hierarchy would be preferable to an exclusive cache hierarchy, b) the conditions under which an exclusive cache hierarchy would be preferable to an inclusive cache hierarchy, and c) why might companies like Intel use a "non-inclusive" policy (inclusive on insert, inclusion property not maintained thereafter), rather than either exclusive or inclusive policies.

d) (10 pts) For each of the "3 C's" of cache misses, define what that class of misses is and describe how, given a cache configuration and a memory access trace, you would classify whether a miss belongs to that class.

Scheduling, Dependencies, and Hazards

For a) and b) Consider the following RISC-style (in this case, MIPS) assembly-level representation of a function **FOO(int *N)** (Note that, in MIPS, except for SW[writes to memory] and BEQ/BNE[PC update only] the leftmost identifier is the destination register)

```
1 |FOO:  XOR    $t0, $t0, $t0     ; $t0 ← 0
2 |       ADD    $v0, $0, $0       ; $v0 ← 0
3 |       ADDI   $t1, $0, 1        ; $t1 ← 1
4 |       BEQ    $a0, $0, DONE     ; if  N==0, done
5 |       LW     $a1, 0($a0)       ; $a1 ← *N
6 |       BEQ    $a1, $0, DONE     ; if *N==0, done
7 |FL1:  ADD    $t2, $0, $v0      ; $t2 ← $v0
8 |       ADD    $v0, $v0, $t1     ; $v0 ← $v0 + $t1
9 |       ADD    $t1, $0, $t2      ; $t1 ← $t2
10|       ADDI   $t0, $t0, 1       ; $t0 ← $t0 + 1
11|       BNE    $t0, $a1, FL1     ; if $t0 != *N, loop
12|DONE: JR     $ra               ; return $v0
```

a) (5 pts) List all data dependencies (RAW) and anti-dependencies (WAW/WAR) among the instructions in the sequence starting at line 7 and ending at line 11, inclusive.

b) (10 pts) Assuming a standard 5-stage (**F**etch-**D**ecode-e**X**ecute-**M**emory-**W**riteback) in-order processor pipeline with all viable forwarding paths and both hazard detection and branch resolution in stage **D**, schedule the instructions executed from a call to FOO(N) where N!=0 and (*N)=1 through cycle 16. Assume perfect branch prediction and that all memory accesses are hits. Indicate that an instruction completes a particular pipeline stage in a given cycle with the capital letter (FDXMW) associated with that pipeline stage. Indicate an instruction being stalled in a particular cycle with a lower-case letter of the pipeline stage that instruction is currently stalled in. Indicate value forwarding with a vertical arrow between the source pipeline stage and the destination stage, in the cycle in which forwarding occurs.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
1|**XOR**	F	D	X	M	W												
2|ADD																	
3|ADDI																	
4|BEQ																	
5|LW																	
6|BEQ																	
7|**ADD**																	
8|ADD																	
9|ADD																	
10|ADDI																	
11|BNE																	
12|**JR**																	

c) (10 pts) Consider a 2-level branch predictor with a shared global history and per-branch 2-bit {N,n,t,T} saturating hysteresis counters for taken/not-taken decisions, i.e. tables are selected via the global history FIFO value and each table has PC-indexed counters. Assume that the global history is of length 3 and is currently [N,N,N]. Assume that all PC-indexed counters experience no aliasing and are initially in state "t".

Consider the execution of the nested loop

```
for (i = 0; i < 8; ++i) {
      for (j = 0; j < 4; ++j) {
            C[i] += A[i][j];
      } // Branch Inner
}  // Branch Outer
```

Assume that branches "Inner" and "Outer" are the only branches and occur at the end of their loop structures, as indicated. Compute the overall branch prediction rate for "Inner" over the course of the entire loop nest; show work.

Performance, Efficiency, and Parallelism

a) (5 pts) List at least three key limiting factors to exploiting ILP in an in-order superscalar pipeline, including those related to scalability of issue width and pipeline depth.

b) (5 pts) The "Denver" mobile processor dynamically generated VLIW bundles of ARM code in HW. While this was shown to be somewhat slower than an equivalent-issue-width dynamically scheduled processor, this performance limitation was considered part of an acceptable tradeoff. Discuss why this might have been the case (i.e. what tradeoff metrics were likely considered) and what limitations may apply to this judgement.

c) (5 pts) The energy required to perform a computation involves both the energy to obtain operands and the energy to compute upon them. In modern designs, data movement energy represents an increasing proportion of total execution energy. Discuss impediments, in a traditional shared cache multiprocessor, to performing computation locally to where data resides.

d) (10 pts) GPUs can provide substantially higher throughput on certain codes than CPUs. However, they have much worse individual thread latency for most computations. List at least three prerequisite properties of a computation for a GPU to plausibly offer substantial performance benefits over a multi-threaded multi-processor.

Multicore and Parallel Processors

a) (5 pts) Assume a four-core system, each core with two SMT (Simultaneous Multi-threading AKA hyperthreading) threads, wherein each core has a private L1 followed by a shared bus to main memory. Assume that all cache lines are initially invalid. Assume that there exist two lines FOO and BAR that map to different sets in the cache and variables A, B, C, and D that reside in FOO and BAR.

Using the **MESI** protocol, simulate the coherence state of cache lines FOO and BAR under the following thread to core and variable to cache line mappings:
Thread to core – T0:Core 0, T1: Core 1, T2: Core 2, T3: Core 3, T4: Core 0, T5: Core 1
Variable to cache line – A and B in FOO, C and D in BAR

OPERATION	FOO				BAR			
<Initial>	Core 0	Core 1	Core 2	Core 3	Core 0	Core 1	Core 2	Core 3
	I	I	I	I	I	I	I	I
T0: Load A								
T1: Load B								
T0: Store A								
T3: Load C								
T3: Store A								
T4: Load B								
T2: Store D								
T0: Load B								
T5: Store B								

b) (10 pts) Consider a program where the initial (sequential) version's execution time is 95% parallelizable and 5% serial. Assume that the program is rewritten to generate a parallel version of the code and that doing so introduces i) an additional initialization overhead equivalent to 5% of the original total execution time and ii) a synchronization overhead = (0.5% of the original execution time * $\log_2$**(N)**) where N is the number of cores that the parallel program runs on. You may assume that the parallel portion can only run on a power-of-2 number of cores. Further, the behavior of the program has changed in the following manner: Due to caching effects, code in the parallelizable portion now has 0.25x its previous CPI and code in the serial portion has 0.5x its previous CPI.

1) Express the speedup of the parallel program, relative to the original version, as a function of (N)

2) For what value of N is this speedup maximized and what is the maximum value? What would a naïve application of Amdahl's law (considering only a parallelization, by N, of the parallelizable portion) predict this speedup to be for this same N?

(note: you do not need to reduce formulas/expressions to their simplest form)

c) (10 pts) Common atomic memory primitive in modern processors include 1) the load-linked/store-conditional (LL/SC) instruction pair and 2) the compare-and-swap instruction (CAS).

1) Using an LL/SC instruction pair, write an assembly function (in generic 3-address code similar to RISC-V / MIPS) that takes a memory location A and a value X and atomically increments memory at A by X.
2) Assume a CAS instruction with syntax "CAS $rd, $rt, ($rs)" where rs is the address, rt is the comparison value, rd is the swap value, and rd overwritten by the value read from ($rs). Using the CAS instruction, write an assembly function (in generic 3-address code similar to RISC-V / MIPS) that takes the address of a lock variable, where 0 is unlocked and 1 is locked, and implements a lock acquire.

Scratch paper (not graded)

Scratch paper (not graded)

MIPS-32 instructions, derived from P&H COD and the MARS help information. MIPS has 32 (mostly) general purpose registers – see next page for details and naming conventions. "imm_x" indicates a signed immediate represented in x bits and "imm_{xz}" represents an unsigned immediate. Updated registers, *if any*, are indicated in bold.

INSTRUCTION	Description
add $**rd**, $rs, $rt	REG[$rd] = REG[$rs] + REG[$rt]
addi $**rt**, $rs, imm_{16}	REG[$rt] = REG[$rs] + imm
addiu $**rt**, $rs, imm_{16}	REG[$rt] = REG[$rs] + imm; ! ovrflw
addu $**rd**, $rs, $rt	REG[$rd] = REG[$rs] + REG[$rt]; !ovr
sub $**rd**, $rs, $rt	REG[$rd] = REG[$rs] – REG[$rt]
subu $**rd**, $rs, $rt	REG[$rd] = REG[$rs] – REG[$rt]; !ovr
mul $**rd**, $rs, $rt	REG[$rd] = (REG[$rs]*REG[$rt]}[31:0]
mult $rs, $rt	HI|LO = REG[$rs] * REG[$rt]
multu $rs, $rt	HI|LO = us(REG[$rs]) * us(REG[$rt])
div $rs, $rt	$rs/$rt; HI = quot; LO = rem
divu $rs, $rt	us($rs)/us($rt); HI = quot; LO = rem
and $**rd**, $rs, $rt	REG[$rd] = REG[$rs] & REG[$rt]
andi $**rt**, $rs, imm_{16z}	REG[$rt] = REG[$rs] & imm
lui $**rt**, imm_{16z}	REG[$rt][31:16]=imm;[15:0]=0
nor $**rd**, $rs, $rt	REG[$rd] = REG[$rs] NOR REG[$rt]
or $**rd**, $rs, $rt	REG[$rd] = REG[$rs] OR REG[$rt]
ori $**rt**, $rs, imm_{16z}	REG[$rt] = REG[$rs] OR imm
xor $**rd**, $rs, $rt	REG[$rd] = REG[$rs] ^ REG[$rt]
xori $**rt**, $rs, imm_{16z}	REG[$rt] = REG[$rs] ^ imm
sll $**rd**, $rs, SHAMT	REG[$rd] = REG[$rs] << SHAMT
sra $**rd**, $rs, SHAMT	REG[$rd] = REG[$rs] >>> SHAMT
srl $**rd**, $rs, SHAMT	REG[$rd] = REG[$rs] >> SHAMT
lb $**rt**, imm_{16}($rs)	Load byte from MEM[REG[$rs]+imm]
lbu $**rt**, imm_{16}($rs)	Load us(byte) from MEM[REG[$rs]+imm]
lw $**rt**, imm_{16}($rs)	Load 32bits from MEM[REG[$rs]+imm]
sb $rt, imm_{16}($rs)	Store REG[$rt][7:0] to MEM[REG[$rs]+imm]
sw $rt, imm_{16}($rs)	Store REG[$rt]to MEM[REG[$rs]+imm]
slt $**rd**, $rs, $rt	REG[$rd] = REG[$rs] < REG[$rt]
slti $**rt**, $rs, imm_{16}	REG[$rt] = REG[$rs] < REG[$rt]
sltiu $**rt**, $rs, imm_{16}	REG[$rt] = REG[$rs] us(<) imm
sltu $**rd**, $rs, $rt	REG[$rd] = REG[$rs] us(<) REG[$rt]
beq $rs, $rt, imm_{16}	Branch to label if REG[$rs]=REG[$rt]
bne $rs, $rt, imm_{16}	Branch to label if REG[$rs]!=REG[$rt]
j imm_{26z}	Jump to label
jal imm_{26z}	Jump to label and set $31 = ret. addr
_jalr $rs, $rd	Jump to REG[$rs] and set $rd to ret
jr $rs	Jump to REG[$rs]

ll	$**rt**, imm_{16}($rs)	As LW, but set linked status for line
sc	$**rt**, imm_{16}($rs)	As SW, **but set $rt to 1 if succeeded, 0 if failed;**

MIPS Register Mnemonics & ABI Reference

$zero	0	Constant 0
$at	1	Reserved for assembler
$v0, $v1	2, 3	Function return values
$a0 - $a3	4 – 7	Function argument values
$t0 - $t7	8 – 15	Temporary (caller saved)
$s0 - $s7	16 – 23	Temporary (callee saved)
$t8, $t9	24, 25	Temporary (caller saved)
$k0, $k1	26, 27	Reserved for OS Kernel
$gp	28	Pointer to Global Area
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Production (availability) & **C**onsumption in an FDXMW pipeline, by instruction class

Inst. Class	**F**	**D**	**X**	**M**	**W**
Load			**C**		**P**
Store			**C**(addr)	**C**(value)	
ALU			**C**	**P**	
Branch		**C**			
J*r		***C*(**for JR**)**	***P***(Return address for JALR)		