December 2009
Sample Final Examination

Introduction to Computer Systems

COMP-273

December 15, 2009 at 9:00 – 12:00

Examiner: Joseph Vybihal Assoc Examiner: Michael Langer

Student Name:		**McGill ID:**									

INSTRUCTIONS:

- This is a **CLOSED BOOK** examination.
- You are permitted **TRANSLATION** dictionaries ONLY.
- **STANDARD CALCULATOR** permitted ONLY.
- This examination is **PRINTED ON BOTH SIDES** of the paper
- This examination paper **MUST BE RETURNED**
- You are permitted to write your answers in either **English or French**
- Write your answers in the **exam booklet** providedd
- Attemp all questions, **part marks** will be assigned, show your work.

Grading

Section	Grade	Your Mark
Question 1: Definitions	25	
Question 2: MIPS Program	25	
Question 3: MIPS Peripheral Programming	25	
Question 4: Calculations	25	
Total	100 %	

Bonus: ____ / 2

Question 1: Definitions (25 points)

(A) Two-Pass Assemblers

Answer the following questions: What happens during pass one? What happens during pass two? What is the symbol table and what is it's structure? What's the address of the 5th instruction in an assembler program? What does BEQ $T1, $T2, END assemble into, assuming END is 3 instructions below? What happens to the symbol table at the end of the assembly process?

(B) Stack, Heap, Memory organization

Diagrammatically contrast the standard memory organization of RAM with the MIPS VM memory organization of RAM. Who controls the stack and heap and show MIPS examples of how you can use it? Using the standard model describe and draw how the peripherals are connected to RAM. What hardware exists to help the programmer communicate with the peripherals?

(C) Virtual Memory

Contrast a Page with a Frame. What is a TLB and how does it work? What is a page table and how does it work? Explain how the VM address is converted into a Real address. Identify some of the registers and hardware structures that exist in a CPU to support VM. What is the swap file? What happens when you run a program under VM? When does the VM need a victim and how does it select one?

How it will be graded:
8 points – (A), 1 point for each question, except for BEQ is 3 points
8 points – (B), 2 points for each question
8 points – (C), 1 point for each question
1 point – for answering all the questions

Question 2: MIPS Program (25 points)

Using any MIPS techniques you like, create a complete MIPS program that asks the user for 10 numbers and then performs a Bubble Sort (or a Selection Sort) on those numbers. The program should then display the sorted numbers to the screen. The swap operation must be its own function.

How it will be graded:
5 points – swap
5 points – input 10 numbers
5 points – output sorted numbers
10 points – Bubble Sort

Question 3: MIPS Peripheral Programming (25 points)

Operating System designers often have to write hardware drivers in assembler. For this problem, I am asking you to write portions of a printer's software driver. A driver is a set of functions that the OS or a user's program can call to interface with a device. It is just a regular assembler function. What is special about it is that it is specifically designed to manage and interface a particular device. It is critical that a driver function protect the registers once it is invoked.

Generally speaking, printers are very slow devices compared to CPUs. In class we compared hard disk access with memory access and saw that it took 20 million ns for data to move to and from a hard disk. To compare this to a printer, hard disk access is lightning fast. This complicates printing a little.

This question asks you to write three device functions in MIPS:

- int Aquire_Device ( )
- int Send_Data (int id, char c)
- void Release_Device (int id)

We will imagine that this is a multiprocessing environment and any process may ask for the device at any time. Now, we know that only one device can use the printer at any time. Aquire_Device is the function that determines if the device is available for use by the requesting process. If a process does have control over the device then it can send data to that device using the Send_Data function.

The OS must manage the sharing of the device. Acquire_Device does not interface with the device directly. Instead, this manages the state of the device using internal variables. Assume a word of memory is used to store an ID number. The label for this memory location is ID. This ID is incremented each time a process calls Aquire_Device. The incremented ID number is returned by the function. The process uses the returned number as a ticket number indicating when it is its turn to use the printer.

A process can call Send_Data at any time but the function will only send the character to the printer if the process is permitted to use the device. This function knows who is permitted to use the device because it has a word in memory called ActiveID. This memory location stores the ticket ID number of the process who is permitted to use the device. The process who calls this function and does not match this ID is kicked out with a return code of –1. If the processes ID number matches ActiveID then the data is sent to the printer.

Once a process is finished printing it calls the Release_Device function. This function verifies if the id number matches ActiveID. If it does match it then increments ActiveID to permit the next process with the next ticket number in. This function then returns. If the id did not match ActiveID then the function just terminates as well.

To send data to a printer occurs in this manner:

The printer for this questions has two registers: a combined status/command register and a data register. The Statis/Command register is at address 1000 Hex while the data register is at address 1004 Hex. The data register is very simple. Any data placed in their will be sent to the printer for display. You can put in there a Unicode character or an integer number. If it is a Unicode character then it will use only the first 16 bits, 0 to 15. If it is a number then you can use any of the 32 bits.

The status register is a 16-bit register. It contains information on how busy the printer is, what type of data is in the data register, if there were any errors, and the command bits. The register is formatted as follows:

Bit 0 = True if busy (I.E. currently printing something)
Bit 1 = Data type (0 = character, 1 = integer)
Bit 2 = Existence of data (1 = data in data register, 0 = no data or not valid data)
Bit 3-5 = Printer error code (0-no error, 1-paper, 2-ink, 3-data, 4-time out, space for future errors)
Bit 6 = Print (0 = don't print, 1 = print contents of data register)
Bit 7 = Form feed (1 = perform form feed)
Bit 8 = Reset (1 = reset/reboot printer)
Bit 8-F = Not used

To send data, you must first wait for the availability of the printer. Then you must put information into the data register. Then you must tell the printer to print. The program should wait for the printer to finish and return its error code. Then the function returns the error code.

When the printer finishes printing it does the following: sets busy to zero, sets existence of data to zero only if there was an error, set the error code, and clears all the command bits (bits 6, 7 and 8).

<u>Question 4: Calculations</u> (25 points)

All the following questions assume a 1GHz MIPS CPU and assume 1 MIPS instruction takes 4 cycles to run, calculate the following:

1. Given the data and the code you wrote for Question 4, calculate the overhead in cycles your polling and interrupt algorithms would have, individually. (The actual answer is not important here but how you calculated it)
2. Then compute the impact on the CPU by percentage of load.
   - Assume you want to poll 100 times per second
   - Assume you want the interrupt to transfer 100 numbers to the array
     - Suggest an optimization to your original interrupt code (show code)
     - How would the interrupt calculation change if we had DMA (show calculations)? What about the algorithm (show code)?

Note: You do not need to rewrite ALL the code. Just enough to help us understand.

<u>How it will be graded</u>:

5 points – Question 1
5 points – Polling load
5 points – Interrupt load
5 points – Suggested interrupt optimization
5 points – DMA question

Bonus Question

First bonus question: Who is Tom West?
Second bonus question: What's a Microkid?

MIPS operands

Name	Example	Comments
32 registers	$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at, Hi, Lo	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants. Hi and Lo contain the results of multiply and divide.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0 $s1,$epc	$s1 = $epc	Used to copy Exception PC plus other special registers
	multiply	mult $s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; logical AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; logical OR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Logical AND reg, constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Logical OR reg, constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2+100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte unsigned	lbu $s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Compare < constant; two's complement
	set less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Compare less than; natural numbers
	set less than immediate unsigned	sltiu $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	`$f0, $f1, $f2, . . . , $f31`	MIPS floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 – $f6`	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP. multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 – $f6`	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1` = Memory[`$s2` + 100]	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	Memory[`$s2` + 100] = `$f1`	32-bit data to memory
Conditional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if (`$f2` < `$f4`) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if (`$f2` < `$f4`) cond = 1; else cond = 0	FP compare less than double precision

Service	System call code	Arguments	Result
`print_int`	1	`$a0` = integer	
`print_float`	2	`$f12` = float	
`print_double`	3	`$f12` = double	
`print_string`	4	`$a0` = string	
`read_int`	5		integer (in `$v0`)
`read_float`	6		float (in `$f0`)
`read_double`	7		double (in `$f0`)
`read_string`	8	`$a0` = buffer, `$a1` = length	
`sbrk`	9	`$a0` = amount	address (in `$v0`)
`exit`	10		

Remaining MIPS I	Name	Format	Pseudo MIPS	Name	Format
exclusive or (*rs* ⊕ *rt*)	xor	R	move	move	rd,rs
exclusive or immediate	xori	I	absolute value	abs	rd,rs
nor (¬(*rs* ∨ *rt*))	nor	R	not (¬*rs*)	not	rd,rs
shift right arithmetic	sra	R	negate *(signed or unsigned)*	neg*s*	rd,rs
shift left logical variable	sllv	R	rotate left	rol	rd,rs,rt
shift right logical variable	srlv	R	rotate right	ror	rd,rs,rt
shift right arith. variable	srav	R	mult. & don't check oflw *(signed or uns.)*	mul*s*	rd,rs,rt
			multiply & check oflw *(signed or uns.)*	mulo*s*	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder *(signed or unsigned)*	rem*s*	rd,rs,rt
load halfword unsigned	lhu	I	load immediate	li	rd,imm
store halfword	sh	I	load address	la	rd,addr
load word left *(unaligned)*	lwl	I	load double	ld	rd,addr
load word right *(unaligned)*	lwr	I	store double	sd	rd,addr
store word left *(unaligned)*	swl	I	unaligned load word	ulw	rd,addr
store word right *(unaligned)*	swr	I	unaligned store word	usw	rd,addr
branch on less than zero	bltz	I	unaligned load halfword *(signed or uns.)*	ulh*s*	rd,addr
branch on less or equal zero	blez	I	unaligned store halfword	ush	rd,addr
branch on greater than zero	bgtz	I	branch	b	Label
branch on ≥ zero	bgez	I	branch on equal zero	beqz	rs,L
branch on ≥ zero and link	bgezal	I	branch on ≥ *(signed or unsigned)*	bge*s*	rs,rt,L
branch on < zero and link	bltzal	I	branch on > *(signed or unsigned)*	bgt*s*	rs,rt,L
jump and link register	jalr	R	branch on ≤ *(signed or unsigned)*	ble*s*	rs,rt,L
return from exception	rfe	R	branch on < *(signed or unsigned)*	blt*s*	rs,rt,L
system call	syscall	R	set equal	seq	rd,rs,rt
break *(cause exception)*	break	R	set not equal	sne	rd,rs,rt
move from FP to integer	mfc1	R	set greater or equal *(signed or unsigned)*	sge*s*	rd,rs,rt
move to FP from integer	mtc1	R	set greater than *(signed or unsigned)*	sgt*s*	rd,rs,rt
FP move (<u>s</u> *or* <u>d</u>)	mov.*f*	R	set less or equal *(signed or unsigned)*	sle*s*	rd,rs,rt
FP absolute value (<u>s</u> *or* <u>d</u>)	abs.*f*	R	set less than *(signed or unsigned)*	sle*s*	rd,rs,rt
FP negate (<u>s</u> *or* <u>d</u>)	neg.*f*	R	load to floating point (<u>s</u> *or* <u>d</u>)	l.*f*	rd,addr
FP convert (<u>w</u>, <u>s</u>, *or* <u>d</u>)	cvt.*ff*	R	store from floating point (<u>s</u> *or* <u>d</u>)	s.*f*	rd,addr
FP compare un (<u>s</u> *or* <u>d</u>)	c.xn.*f*	R			