McGill

April 2017
Final Examination

Introduction to Computer Systems

COMP 273

April 13, 2017 at 18:30

EXAMINER: Joseph Vybihal ASSOC. EXAMINER: David Meger

STUDENT NAME:		**McGILL ID:**									

INSTRUCTIONS:

EXAM:	CLOSED BOOK ☒ OPEN BOOK ☐
	SINGLE SIDED ☐ PRINTED ON BOTH SIDES OF THE PAGE ☒
	MULTIPLE CHOICE ☐ NOTE: The Examination Security Monitor Program detects pairs of students with unusually similar answer patterns on multiple-choice exams. Data generated by this program can be used as admissible evidence, either to initiate or corroborate an investigation or a charge of cheating under Section 16 of the Code of Student Conduct and Disciplinary Procedures. ANSWER IN BOOKLET ☒ EXTRA BOOKLETS PERMITTED: YES ☒ NO ☐ ANSWER ON EXAM ☐
	SHOULD THE EXAM BE: RETURNED ☒ KEPT BY STUDENT ☐
CRIB SHEETS:	NOT PERMITTED ☒ PERMITTED ☐ one 8 1/2X11 handwritten double-sided sheet Specifications:
DICTIONARIES:	TRANSLATION ONLY ☒ REGULAR ☐ NONE ☐
CALCULATORS:	NOT PERMITTED ☐ PERMITTED (Non-Programmable) ☒
ANY SPECIAL INSTRUCTIONS	

	POINTS	SCORE
SECTION 1: Definitions	20	
SECTION 2: Circuits	30	
SECTION 3: Assembler Programming	30	
SECTION 4: Problems	20	
	100	

SECTION 1: Definitions [20 points]

Question 1.1: TLB 10 points

Answer the following questions about the TLB:

(a) Describe how the addressing works in a TLB.
   a. It is based on modulo arithmetic
      i. Implemented in binary as the lower 2^n bits
      ii. Example 32-bit address in total, modulo uses lower 8-bits
   b. Structure: cache has 8-bit address, 22-bit tag, in-use bit, then the data stored at that address (either 8-bit for data cache, or 32-bits for instruction cache).
      i. Instruction cache: 2 (ignored) + 8 (indexing) + 22 (Tag) + 32-bit data row
      ii. Data cache : 8 (indexing) + 24 (tag) + 8-bit data row
   c. Addressing:
      i. 32-bit address = 8-bit address + 24-bit tag for data cache
      ii. 32-bit address = 2-bit not used + 8-bit address + 22-bit tag for instruction cache. The 2-bit not used at the lower end of the address is due to instructions being 4 bytes long and addressing increments by 4, which is covered by ignoring those 2 lowest bits.
   d. Therefore, for instruction cache: PC → 2+8+20 (as in f above). The 8 is not stored but used as an index into the TLB. The in-use bit is used to determine if there is an instruction in that row. If in-use == 1 then there is data and the 22 bits is compared to the Tag. If the Tag and 22 bits match then the instruction is released from the cache row to the IR.

(b) Give an example showing conflicting instruction addressing
   a. When two addresses in RAM resolve to the same 8-bit modulo address.

(c) Describe all the ways how a TLB implements a faster access technique than regular RAM
   a. Addressing circuitry is smaller since 8-bits and resolves electronically faster
   b. 4-way cache, for example, stores 4 instructions at each address downloading the 4 together resolving the correct instruction in parallel
   c. 4-way cache uses space locality to give quick access to the next instruction
   d. TLB runs at CPU clock speed

Question 1.2: Multi Processing 10 points

(a) Describe how a co-processor could be used to execute multiple instructions at the same time.
   Since the question says co-processor it does not mean multi-CPU and it does not mean multi-Core. It means a FP co-processor as we see in MIPS.

   In MIPS the main processor has an integer ALU while the co-processor has a fixed-point ALU. Each processor (main and co) have their own registers and ALU. If the instruction has mixed numeric types or if two consecutive instructions are math/logic and of different types (one being integer the other fixed-point) then each instruction can be passed to the two processors (main and co) to execute their respective different-type math operation at the same time. Each processor has its own independent clock allowing parallel processing.

(b) Discuss how the system would schedule these multiple instructions (in other words, how would it set it up so that more than one instruction could execute across different co-processors). What extra hardware would we need above the standard MIPS we saw in class. Ignoring the OS, how would the CPU's circuity detect this parallel opportunity?
   a. Scheduling system method & Parallel opportunity circuit detection
      i. Assuming instructions exit the instruction cache one at a time and assuming the opcode for FP instructions are different in number (for example above binary n while integer instructions are below integer n) then the instruction

can be routed to a FP IR register or an Integer IR register as it exits the instruction cache.

ii. This can be detected using the numerical opcode value and the routing criteria. A simple addressing circuit could be used to route the instructions.

iii. At the end of execution (after stage 4) a no-op instruction is automatically loaded into the IR register by the CU. This is in case the instruction exiting the instruction cache is not for the IR. This circuit would exist for both processors. In other words, the IR-CU combination always needs to execute but it can not be guaranteed that an instruction will always be available for that IR-CU. Auto loading it with a no-op after stage 4 would keep it busy, safely.

SECTION 2: Circuits [30 points]

Question 2.1: Bus Competition Circuit 15 points

Assume a 2-bit bus that runs horizontally across the page (call this from east to west). It does not matter for this questions where the bus started from or where it ends. Assume two wires are coming from the north (top of your page) connecting to the bus (each north wire connects to a different bus wire). Assume two wires are coming from the south (bottom of your page) connects to the bus, but at a different location on the left (west), (each south wire connects to a different bus wire).

Design the connection so that only one of the wire pairs (north or south) can use the bus at a time. Usage is based on a first-come-first-serve competition. The north/south wire pair that first accesses the bus blocks the other pair. Assume a signal of zero on both wires indicates the north/south wires do not want to use the bus. Make sure to handle the special case when both wires send their signals at the exact same instant.

You may use basic circuit components: wire, and, or, not, xor, flip-flop.

Question 2.2: Pop Dispenser Machine 15 points

Assume we have machines Q, D, N that can detect the type of coin dropped into a pop dispenser machine. Q, D, and N are sensors with a single wire exiting the sensor. If the sensor detects what it is designed to see the wire emits a 1, otherwise it emits a 0. Q detects quarters (25 cents). D detects dimes (10 cents). N detects nickels (5 cents). Assume we have three additional motors X, Y Z. Each of these motors has a single wire entering the motor. If the wire is 0 then the motor does nothing. If the wire is 1 then the motor moves a can of pop to the customer. The pop at motor X costs $1.25. The pop at motor Y costs $0.30. The pop at Z costs 0.60.

Create a circuit diagram that will run this pop machine properly.

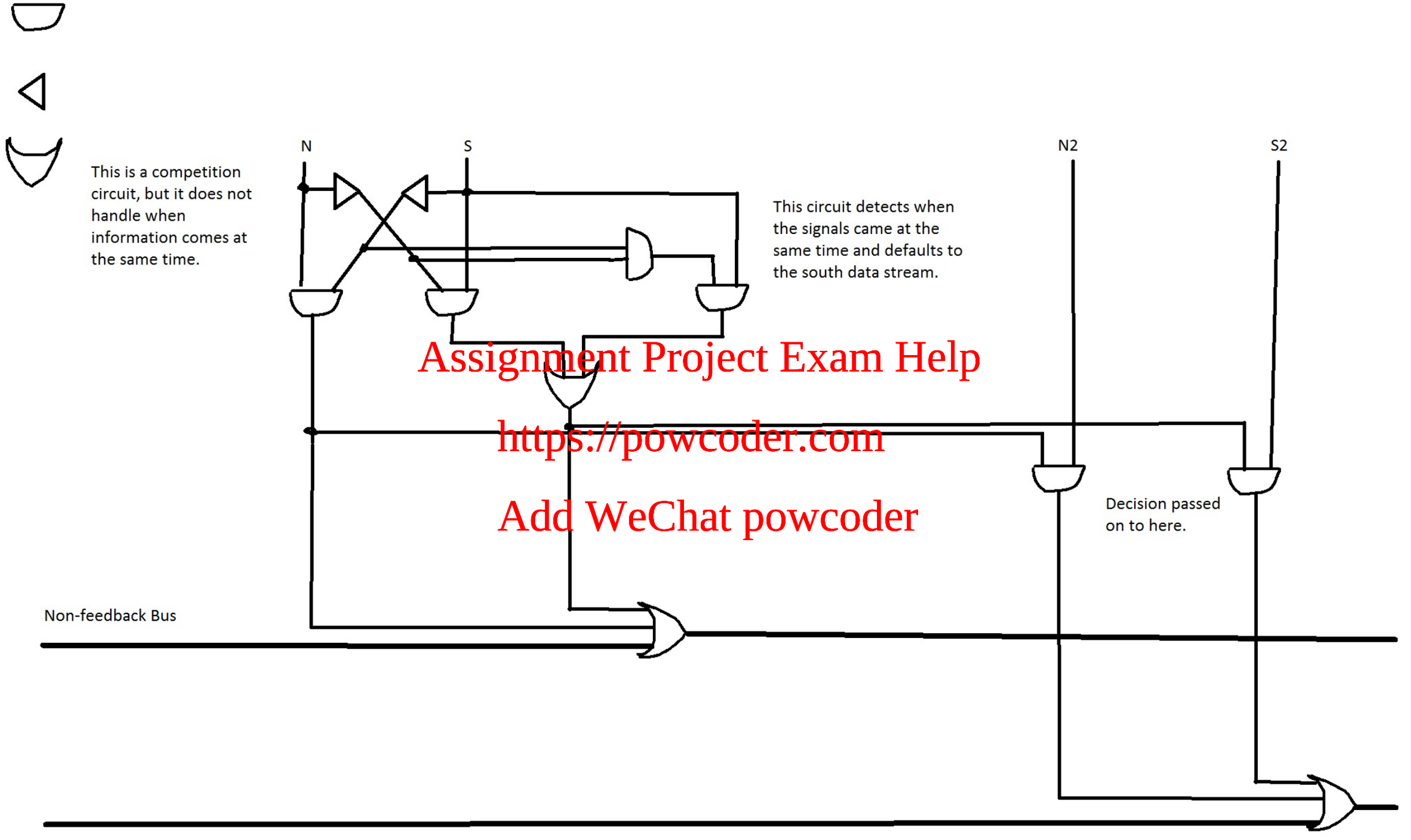
You may use basic circuit components: wire, and, or, not, x or, flip-flop, incrementor.

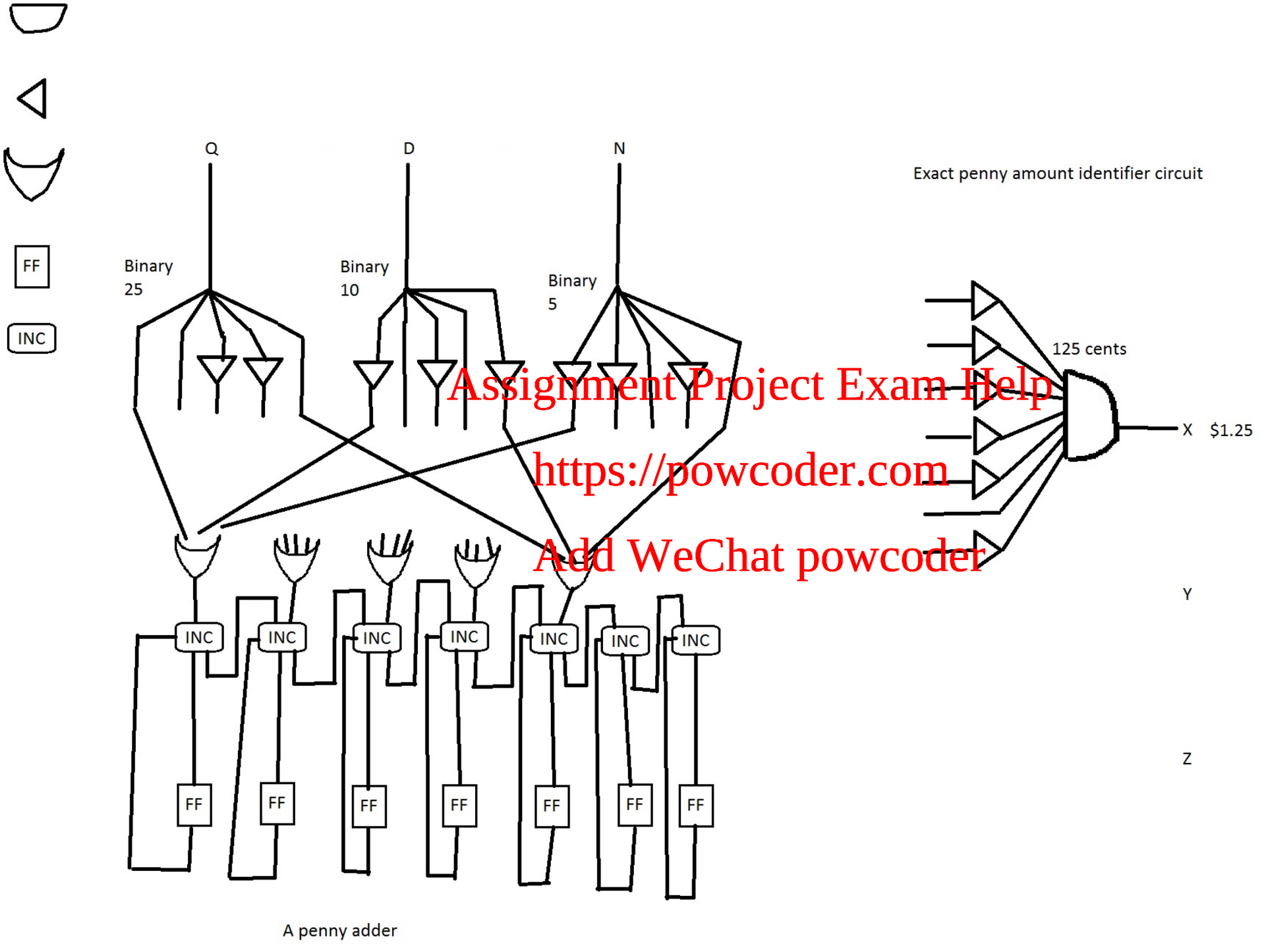
SECTION 3 : Assembler Programming [30 points]

Question 3.1: Linked-list traversal 15 points

Write the following function in MIPS assembler:

```
int len_list(node_ptr)
```





Assume a linked list has already been created for you in the computer's memory with a head pointer referencing the first node in the list. Every node has two fields. The first field is type integer and the second field is a pointer to the next node in the list. The last node in the list has its next pointer referencing the value null to indicate the end of the list.

The function len_list takes the head pointer as parameter and counts the number of nodes in the list. Once that it done it returns the count. Make sure to pass parameters and return values correctly (following standard MIPS practices).

The data structure shape is similar to:

```
Struct Node {
        int data;
        struct node *next;
}
```

Where all the types are in MIPS. 32-bit integer and 32-bit address.
The function is passed a pointer to the list (either the head of the list or any other position – it does not matter because it behaves the same way). The pointer points to the first address of the node.

Assume $a0 has this parameter.

```
        Move    $s0, $a0        # this will be our traversal pointer
Loop:
        Lw      $s1, 0($s0)     # data from node loaded into s1
        Lw      $s0, 4($s0)     # pointer has been updated to the next node
        Beq     $s0, $zero, End # is s0 is the null pointer end the loop
        J       Loop
End:
```

This is the basic structure of the solution. Students should pass parameters using MIPS technique. They calculate and return the length as well.

Question 3.2: Peripheral Device Interfacing [15 points]

Assume a proximity sensor is connected to your MIPS computer. Proximity sensors return integer numbers at each sensor clock tick (this is different from your CPU and System clock ticks). The integer number the sensor returns is a measurement in centimetres. It measures the distance between the sensor and any object directly in front of the sensor. The interface card has a data register, a status register, and a command register. The data register is 16 bits and starts at address 100 hex. The status register is 8 bits and starts at address 102 hex. The command register is 8 bits and starts at address 103 hex.

The data register is populated automatically by the sensor. The command register is set to zero automatically by the interface card. When a programmer changes the command register's value, the command associated with the value is instantly initiated by the interface card and then the interface card resets the command register back to zero. The command register works with integer numbers. Integer 0 means the command register is ready for a command. Integer 1 sets the sensor in continuous measurement mode. Integer 2 sets the sensor into Non-Overwrite mode. Integer 3 turns off the sensor. Integer 4 turns on the sensor. The status register's bit 0 indicates if the sensor is on (1) or off (0). Bit 1 indicates new data true (1) false (0). Bit 2 indicates sensor mode, continuous (1) Non-Overwrite (0). Bits 6 and 7 store a two-digit positive integer error code. If these two bits are zero then the sensor is running normally, any other values then something is wrong.

The Non-Overwrite mode uses the Status register's new data bit to indicate that new data has arrived and does not put a new value into the data register until after the programmer clears the new data bit (sets it back to zero). The continuous mode does not use the new data bit from the status

register. It instead continuously copies data from the sensor into the data register regardless of the programmer.

Write two functions:

```
Void getContinuous(int array{}, int arraylength)
Void getNonOverwrite(int array[], int array length)
```

The above two functions sets the peripheral card to the correct mode and then attempts to optimally copy arraylength number of values from the interface card to the array taking into consideration the special nature of each mode.

At start the sensor is off and the status register values cannot be trusted until the sensor is turned on. You must handle error checking. You cannot overwrite data until you are sure it is safe to do so.

SECTION 4: Problems [20 points]

Question 4.1: Assembling 5 points

Convert the following assembler instructions into machine language:

(a) lw $1,10($2)
(b) and $1, $2, $3

Question 4.2: CPU and System Comparisons A=5 points, B=5 points, C=5 points

A. You have a 2 GHz CPU, your RAM runs at 333 MHz, your hard drive writes at 100 characters per second and is connected to your Zero Page. Assume all other values are 1. You need to save a 1024-byte file:
   a. How long will it take to save the file?
      i. Basic operation given that file is in RAM and we want to move it to disk. There is no mention of blocks so we are in byte mode.
      ii. Moving by byte mode looks like this: (1) RAM to CPU, (2) CPU to device, (3) wait for device to write byte, (4) repeat until done
      iii. Steps (1) (2) and (4) are assumed to be 1 tick.
         1. The CPU is slowed down by the RAM so only the RAM speed is important = 333 MHz = students can use 1000 to represent the Hertz to make the math easier or properly 1024 = 0.000000003 per byte per second
      iv. Step (3) is 100 characters (bytes) per second = 0.01 secs per byte
      v. Moving a single byte = 0.01 sec + 3 ticks = 0.010000009
      vi. Moving all the bytes multiple by 1024.
   b. If you had the option to upgrade the computer with either a RAM that runs at 666 MHz or upgrade the hard drive to 300 characters per second (you cannot upgrade both) what would be faster? Prove this.
      i. The above calculation would need to be repeated twice to find out which gave the lower value.
         1. RAM = 0.010000003
         2. Drive = 0.003333336
      ii. The drive upgrade would be better

B. Compare the number of ticks it would take for 10 instructions to be executed in a standard CPU vs a pipeline CPU. Assume for the standard CPU the instructions are in RAM and the PC is already pointing to the first instruction. Assume for the pipeline CPU the instructions are already in the cache and the PC points to the first instruction. Assume both CPUs run at 2GHz. Assume that the RAM runs at 1 GHz. Assume that each stage of the pipeline requires

5 clock ticks. Justify your answer. If any other circuits are required then their ticks are represented by a variable.

a. Classical: PC -> MAR -> AR -> RAM[AR] -> DR -> MBR -> IR -> CU Times 10
    i. CPU speeds: PC -> MAR, MBR -> IR, IR -> CU
        1. 2 GHz and 3 operations = 3/2,000,000,000 = M
    ii. RAM speeds: MAR -> AR, AR -> RAM[AR], RAM[AR] -> DR, DR -> MBR
        1. 1 GHz and 4 operations = 4/1,000,000,000 = N
    iii. CU speed not needing RAM
        1. 2 GHz and x operations = (x/2,000,000,000) * a% = 0
        2. Where "a" is the fraction of instructions not needing RAM
    iv. CU speed with RAM load
        1. 1 GHz and y operations = (y/1,000,000,000) * b% = P
        2. Where "b" is the fraction of instruction needing RAM
    v. Add all the above for one instruction and Multiply the above 10 times
        1. 10 * (M + N) + O + P
b. Pipeline: 4 stages at 5 ticks per stage
    i. Instructions not needing RAM 4 * 5 = 20 ticks per instruction = M
    ii. Instructions needing RAM
        1. 4 * 5 + RAM access = 4 * 5 + CPU/RAM speed = 4 * 5 + 2 per instruction = N
        2. Final total
            a. M * a% + N * b%
            b. Where "a" and "b" are the fraction of instructions not needing RAM and needing RAM respectively.

C. Provide your own numbers and calculations, (a) show why a DMA circuit impacts the CPU less than Interrupts. (b) Show why Interrupts impact the CPU less than polling. (c) Show a case where polling would be better than interrupts.

a. Student needs to show how a peripheral operation contains the following steps: (1) check device status and give command, (2) wait for the device to finish, (3) copy information to RAM.
b. DMA = DMA setup and Interrupt when it is complete. Copy to RAM CPU does not do.
c. Interrupt = command device, the interrupt handler copies data to RAM
d. Polling = command device, wait for device, copy data to RAM

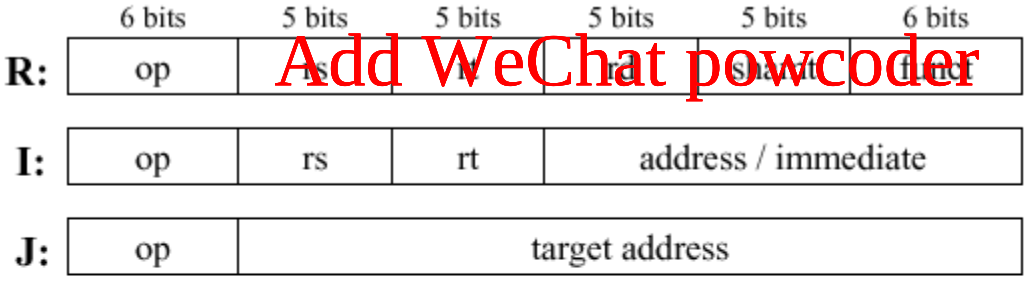
BONUS QUESTIONS [1 point each]

B1: Name all the women from the novel.

B2: What was unique about the computer they were building?

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
	subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
	add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
	add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
	subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
	add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
	Move fr. copr. reg.	mfc0 $1,$epc	$1 = $epc	Used to get exception PC
	multiply	mult $2,$3	Hi, Lo = $2 ¥ $3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $2,$3	Hi, Lo = $2 ¥ $3	64-bit unsigned product in Hi, Lo
	divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
	divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient and remainder
	Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
	Move from Lo	mflo $1	$1 = Lo	Use to get copy of Lo
Logical	and	and $1,$2,$3	$1 = $2 & $3	3 register operands; logical AND
	or	or $1,$2,$3	$1 = $2 | $3	3 register operands; logical OR
	and immediate	andi $1,$2,100	$1 = $2 & 100	Logical AND register, constant
	or immediate	ori $1,$2,100	$1 = $2 | 100	Logical OR register, constant
	shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
	shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
Data transfer	load word	lw $1,100($2)	$1 = Memory[$2+100]	Data from memory to register
	store word	sw $1,100($2)	Memory[$2+100] = $1	Data from register to memory
	load upper imm.	lui $1,100	$1 = 100 x 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100	Equal test; PC relative branch
	branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100	Not equal test; PC relative
	set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0	Compare less than; 2's complement
	set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0	Compare < constant; 2's comp.
	set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0	Compare less than; natural numbers
	set l.t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0	Compare < constant; natural
Unconditional jump	jump	j 10000	go to 10000	Jump to target address
	jump register	jr $31	go to $31	For switch, procedure return
	jump and link	jal 10000	$31 = PC + 4; go to 10000	For procedure call



op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-2^{15})
immediate: constants for immediate instructions

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		

FIGURE A.17 System services.

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

Remaining MIPS I	Name	Format	Pseudo MIPS	Name	Format
exclusive or ($rs \oplus rt$)	xor	R	move	move	rd,rs
exclusive or immediate	xori	I	absolute value	abs	rd,rs
nor ($\neg(rs \vee rt)$)	nor	R	not ($\neg rs$)	not	rd,rs
shift right arithmetic	sra	R	negate *(signed or unsigned)*	negs	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.)*	muls	rd,rs,rt
			multiply & check oflw *(signed or uns.)*	mulos	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)*	rems	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.)*	ulhs	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)*	bges	rs,rt,L
branch on < zero and link	bltzal	I	branch on > *(signed or unsigned)*	bgts	rs,rt,L
jump and link register	jalr	R	branch on ≤ *(signed or unsigned)*	bles	rs,rt,L
return from exception	rfe	R	branch on < *(signed or unsigned)*	blts	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)*	sges	rd,rs,rt
move to FP from integer	mtc1	R	set greater than *(signed or unsigned)*	sgts	rd,rs,rt
FP move *(s or d)*	mov.f	R	set less or equal *(signed or unsigned)*	sles	rd,rs,rt
FP absolute value *(s or d)*	abs.f	R	set less than *(signed or unsigned)*	slts	rd,rs,rt
FP negate *(s or d)*	neg.f	R	load to floating point *(s or d)*	l.f	rd,addr
FP convert *(w, s, or d)*	cvt.f.f	R	store from floating point *(s or d)*	s.f	rd,addr
FP compare un *(s or d)*	c.xn.f	R			

Name	Register Number	Usage	Preserved on call
$zero	0	the constant value 0	n.a.
$at	1	reserved for the assembler	n.a.
$v0-$v1	2-3	value for results and expressions	no
$a0-$a3	4-7	arguments (procedures/functions)	yes
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t8-$t9	24-25	more temporaries	no
$k0-$k1	26-27	reserved for the operating system	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes