EEL 4712
Midterm 3 – Spring 2019
VERSION 1

Name: _____

UFID: _____

Sign here to give permission for your test to be returned in class, where others might see your score:

_____

IMPORTANT:
• Please be neat and write (or draw) carefully. If we cannot read it with a reasonable effort, it is assumed wrong.
• **As always, the best answer gets the most points.**

COVER SHEET:

Problem#:	Points
1 (3 points)	
2 (3 points)	
3 (3 points)	
4 (3 points)	
5 (3 points)	
6 (3 points)	
7 (3 points)	
8 (3 points)	
9 (6 points)	
10 (3 points)	
11 (3 points)	
12 (8 points)	
13 (12 points)	
14 (5 points)	
15 (6 points)	
16 (26 points)	
17 (4 points)	
18 (3 points)	3

Total:

Regrade Info:

1) (3 points) In general, FPGA synthesis tools convert tri-state buffers into multiplexors. Briefly explain why the tools perform this conversion.

2) (3 points) Write enables for a bus often use a one-hot encoding, where at most one source is assumed to be given access at any time. For a 4-source bus, what component can be added to guarantee this assumption is always true?

3) (3 points) How long does it take for a metastable signal to stabilize (circle one)?
    a. 1 cycle
    b. 10 cycles
    c. 100 cycles
    d. Unknown

4) (3 points) Explain why a dual-flop synchronizer provides a shorter mean-time between failures as the destination clock frequency increases.

5) (3 points) Describe when to use a FIFO instead of a handshake synchronizer.

6) (3 points) Explain why a dual-flop synchronizer *can* be used to synchronize multiple bits when only a single input bit changes.

7) (3 points) FPGA virtualization addresses which of the following problems (circle all that apply):
    a. Productivity
    b. Power consumption
    c. Compile times
    d. Application portability

8) (3 points) What is the name of the emerging area of research that looks at sacrificing accuracy to improve performance and/or energy?

9) a. (3 points) For the MIPS, a function call is usually implemented with what instruction?

b. (3 points) For the MIPS, a return from a function uses what instruction?

10) (3 points) What operation does the ALU perform for a load word instruction?

11) (3 points) Briefly explain what an *immediate* is in a MIPS instruction.

12) (8 points) Write MIPS assembly code that matches the following behavior. Inport0 corresponds to byte address 0xFFF8. Inport1 and Outport corresponds to byte address 0xFFFC. Use $0-$31 for registers instead of the normal MIPS naming convention. Make two columns if necessary.

```
x = Inport0;
y = Inport1;
z = 0;
while (x != y) {
      x++;
      z++;
}
Outport = z;
```

13) (12 points) Create a memory initialization file for the following assembly code. Add comments as necessary. Put a small space between different instruction fields to make it easier to read.

```
        lw $1, 0xFFF8($0)
        addiu $2, $0, 10
        blez $1, ELSE
        addu $3, $0, $2
        j STORE
ELSE:
        addiu $3, $0, 15
STORE:
        sw $3, 0xFFFC($0)
DONE:
        j DONE
```

```
Depth = 256;
Width = 32;
Address_radix = hex;
Data_radix = bin;
% Program RAM Data %
Content
Begin
```

```
End;
```

14) (5 points) Given a solution space with the following implementations, which of the solutions are ***not*** Pareto optimal? If they are all Pareto optimal, state that.
    a. LUTs: 500, DSPs: 15, Time: 18s
    b. LUTs: 1000, DSPs: 5, Time: 18s
    c. LUTs: 1500, DSPs: 22, Time: 15s
    d. LUTs: 2000, DSPs: 20, Time: 12s
    e. LUTs: 10, DSPs: 300, Time: 11s

15) (4 points) a. For the solution space in problem 14, give one new implementation that would prove that none of the previous implementations are Pareto optimal.

    (4 points) b. What implementation from problem 14 would be best for an optimization goal of minimizing LUTs given a time constraint of 15s and a DSP constraint of 20?

16) a. (8 points) For the following code, create a schedule for the provided datapath. Ignore muxes, registers, and other glue logic. Like the examples in class, assume that address calculations are done *without* using the specified resources (i.e., address calculations cost nothing). Do not change the code. Do not unroll or pipeline the loop. List any assumptions.

```
for (int i=0; i < 1000000; i++) {
   a[i] = b[i]*11 + b[i+1]*22 + b[i+2]*33 + b[i+3]*44;
}
```

Datapath
4 multipliers
2 adders
1 comparator
1 memory for b[] (can read 4 elements/cycle)
1 memory for a[] (can write 1 element/cycle)

b. (4 points) What is the execution time in total cycles based on your schedule from part a? Show your work.

c. (4 points) What is the execution time in total cycles after unrolling the loop once (i.e. replicating the datapath).

d. (4 points) For a pipelined implementation of the datapath in part a with no unrolling, what is the approximate execution time in total cycles? Show your work.

e. (4 points) For a pipelined implementation of the datapath in part a after unrolling the loop once, what is the approximate execution time in total cycles? Show your work.

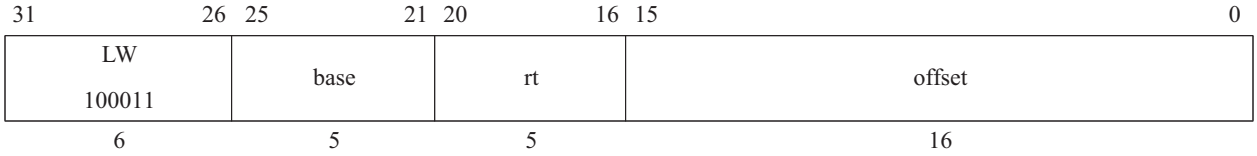
17) a. (2 points) Create a VHDL type *my_array* for a one-dimensional unconstrained array where each element is 16 bits.

b. (2 points) Create a signal *array1* that is an instance of *my_array* with 100 total elements.

18) (3 points) Free points just because.

Category	OpCode (Hex)	Function (Hex)	Instruction	Example	Meaning	MIPS?	Comments	IsSigned = false
Arithmetic	0x00	0x21	**add - unsigned**	addu $s1, $s2, $s3	$s1 = $s2 + $s3	yes	Adds two registers and stores the result in a register	
	0x09		**add immediate unsigned**	addiu $s1, $s2, IMM	$s1 = $s2 + IMM	yes	Adds a register and a sign-extended immediate value and stores the result in a register	
	0x00	0x23	**sub unsigned**	subu $s1, $s2, $s3	$s1 = $s2 - $s3	yes	Subtracts two registers and stores the result in a register	
	0x10 (not MIPS)		**sub immediate unsigned**	subiu $s1, $s2, IMM	$s1 = $s2 - IMM	no	Subtracts IMM from a register and stores the result in a register	
	0x00	0x18	**mult**	mult $s, $t	$LO= $s * $t	yes	Multiplies $s by $t and stores the result in $LO.	
	0x00	0x19	**mult unsigned**	multu $s, $t	$LO= $s * $t	yes	Multiplies $s by $t and stores the result in $LO.	
Logical	0x00	0x24	**and**	and $s1, $s2, $s3	$s1 = $s2 and $s3	yes	The contents of s3 are combined with the contents of s2 in a bitwise logical AND operation. The result is placed into s1.	
	0x0C		**andi**	andi $s1, $s2, IMM	$s1 = $s2 and IMM	yes	The 16-bit immediate is zero-extended to the left and combined with the contents of s2 in a bitwise logical AND operation. The result is placed into s1.	yes
	0x00	0x25	**or**	or $s1, $s2, $s3	$s1 = $s2 or $s3	yes	The contents of s3 are combined with the contents of s2 in a bitwise logical OR operation. The result is placed into s1.	
	0x0D		**ori**	ori $s1, $s2, IMM	$s1 = $s2 or IMM	yes	The 16-bit immediate is zero-extended to the left and combined with the contents of s2 in a bitwise logical OR operation. The result is placed into s1.	yes
	0x00	0x26	**xor**	xor $s1, $s2, $s3	$s1 = $s2 xor $s3	yes	Combine the contents of s3 and s2 in a bitwise logical exclusive OR operation and place the result into s1.	
	0x0E		**xori**	xori $s1, $s2, IMM	$s1 = $s2 xor IMM	yes	Combine the contents of s2 and the 16-bit zero-extended immediate in a bitwise logical exclusive OR operation and place the result into s1.	yes
	0x00	0x02	**srl -shift right logical**	srl $s1, $s2, H	$s1 = $s2 >> H	yes	These logical shifts append 0s. Where H is the number of shifts desired. H is an 5-bit immediate value stored in bits 10-6 of the instruction register.	
	0x00	0x00	**sll -shift left logical**	sll $s1, $s2, H	$s1 = $s2 << H	yes	These logical shifts append 0s. Where H is the number of shifts desired. H is an 5-bit immediate value stored in bits 10-6 of the instruction register.	
	0x00	0x03	**sra -shift right arithmetic**	sra $s1, $s2, H	$s1 = $s2 >> H	yes	These arithmetic shifts, if signed bit is 1, will append 1s. If signed bit is 0, will append 0s. H is an 5-bit immediate value stored in bits 10-6 of the instruction register.	
	0x00	0x2A	**slt -set on less than signed**	slt $s1,$s2, $s3	$s1=1 if $s2 < $3 else $s1=0	yes	Compare the contents of s3 and s2 as signed integers and record the Boolean result of the comparison in s1. If s2 is less than s3 the result is 1 (true), otherwise 0 (false). The arithmetic comparison does not cause an Integer Overflow exception.	
	0x0A		**slti -set on less than immediate signed**	slti $s1,$s2, IMM	$s1=1 if $s2 < IMM else $s1=0	yes	Compare the contents of s2 and the 16-bit signed immediate as signed integers and record the Boolean result of the comparison in s1. If s2 is less than immediate the result is 1 (true), otherwise 0 (false). The arithmetic comparison does not cause an Integer Overflow exception.	
	0x0B		**sltiu- set on less than immediate unsigned**	sltiu $s1,$s2, IMM	$s1=1 if $s2 < IMM else $s1=0	yes	Compare the contents of s2 and the sign-extended 16-bit immediate as unsigned integers and record the Boolean result of the comparison in s1. If s2 is less than immediate the result is 1 (true), otherwise 0 (false). Because the 16-bit immediate is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range. The arithmetic comparison does not cause an Integer Overflow exception	
	0x00	0x2B	**sltu - set on less than unsigned**	sltu $s1,$s2, $s3	$s1=1 if $s2 < $s3 else $s1=0	yes	Compare the contents of s3 and s2 as unsigned integers and record the Boolean result of the comparison in s1. If s3 is less than s2 the result is 1 (true), otherwise 0 (false). The arithmetic comparison does not cause an Integer Overflow exception.	
	0x00	0x10	**mfhi -move from Hi**	mfhi $s1	$s1= HI	yes	The two instructions that follow an MFHI instruction must not be instructions that modify the HI register: DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MTHI, MULT, MULTU. If this restriction is violated, the result of the MFHI is undefined.	
	0x00	0x12	**mflo -move from LO**	mflo $s1	$s1= LO	yes	The two instructions that follow an MFLO instruction must not be instructions that modify the LO register: DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MTLO, MULT, MULTU. If this restriction is violated, the result of the MFLO is undefined.	
Load/Store	0x23		**load word**	lw $s1, offset($s2)	$s1 = RAM[$s2+offset]	yes	The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the register length if necessary, and placed in s1. The 16-bit signed offset is added to the contents of s2 to form the effective address	yes (will be chang
	0x2B		**store word**	sw $s1, offset($s2)	RAM[$s2+offset] = $s1	yes	The least-significant 32-bit word of register s1 is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of s2 to form the effective address.	yes (will be chang
Branch	0x04		**branch on equal**	beq $s1,$s2, TARGET	if $s1=$s2, PC += 4+TARGET	yes	Branches to the specified label when the contents of s1 equal the contents of s2, or it can branch when the contents of s1 equal the immediate value.	
	0x05		**branch not equal**	bne $s1,$s2, TARGET	if $s1/=$s2, PC += 4+TARGET	yes	Branches to the specified label when the contents of s1 do not equal the contents of s2, or it can branch when the contents of s1 do not equal the immediate value.	
	0x06		**Branch on Less Than or Equal to Zero**	blez $s1, TARGET	if $s1 <= 0, PC += 4+TARGET	yes	Branches to the specified label when the contents of s1 are less than or equal to zero. The program must define the destination.	
	0x07		**Branch on Greater Than Zero**	bgtz $s1, TARGET	if $s1 > 0, PC += 4+TARGET	yes	Branches to the specified label when the contents of s1 are greater than zero.	
	0x01		**Branch on Less Than Zero**	bltz $s1, TARGET	if $s1 < 0, PC += 4+TARGET	yes	Branches to the specified label when the contents of s1 are less than zero. The program must define the destination.	
	0x01		**Branch on Greater Than or Equal to Zero**	bgez $s1, TARGET	if $s1 >= 0, PC += 4+TARGET	yes	Branches to the specified label when the contents of s1 are greater than or equal to zero.	
Unconditional Jump								
	0x02		**jump to address**	j TARGET	PC = TARGET	yes	This is a way of always jumping to an address given in the	
	0x03		**jump and link**	jal TARGET	$ra = PC+4 and PC = TARGET	yes	This is the equivalent of the call instruction. The return address is stored in $ra ($31)	
	0x00	0x08	**jump register**	jr $ra	PC = $ra	yes	This can be used to jump to an address held in a register. For example the address held in $ra($31) after a jal is the return address.	

Load Word LW



Format: `LW rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To load a word from memory as a signed value

Description: `rt ← memory[base+offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

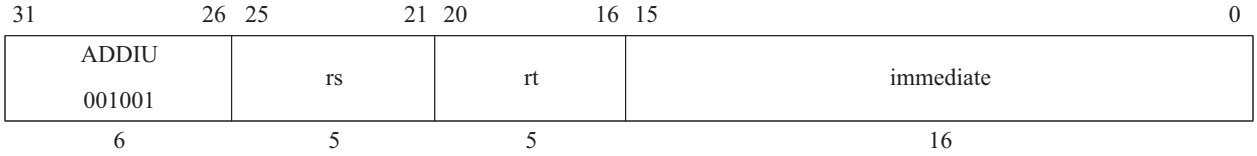
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]← memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Add Immediate Unsigned Word — ADDIU



Format: `ADDIU rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To add a constant to a 32-bit integer

Description: `rt ← rs + immediate`

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

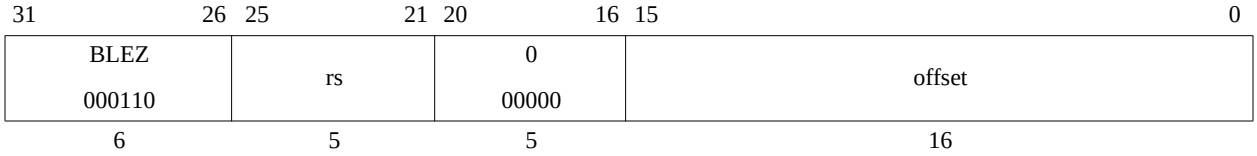
Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

Branch on Less Than or Equal to Zero — BLEZ



Format: `BLEZ rs, offset` **MIPS32 (MIPS I)**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs ≤ 0 then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:     target_offset ← sign_extend(offset || 0^2)
       condition ← GPR[rs] ≤ 0^GPRLEN
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

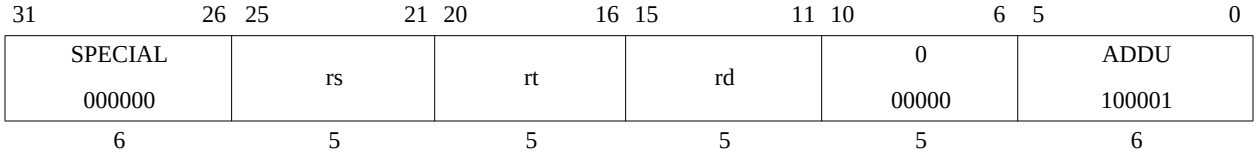
Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.

Add Unsigned Word — ADDU



Format: `ADDU rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To add 32-bit integers

Description: `rd ← rs + rt`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

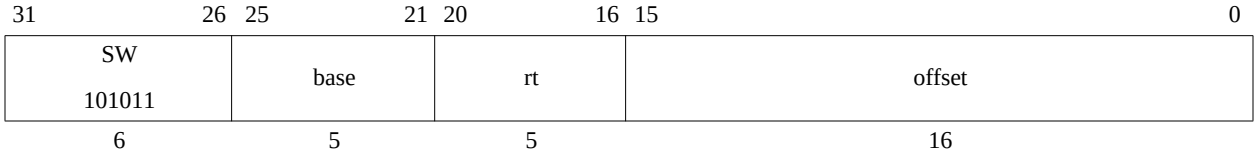
Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

Store Word SW



Format: `SW rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To store a word to memory

Description: `memory[base+offset] ← rt`

The least-significant 32-bit word of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

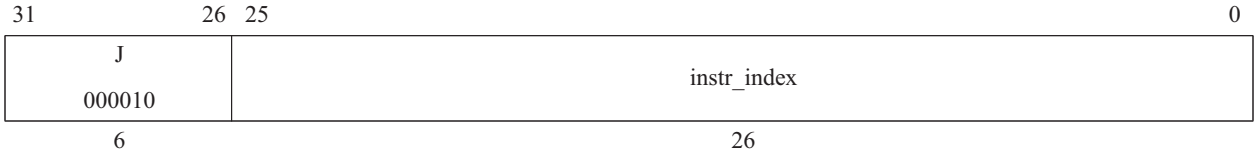
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Jump J



Format: `J target` **MIPS32 (MIPS I)**

Purpose:

To branch within the current 256 MB-aligned region

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:
I+1:PC ← PC_GPRLEN..28 || instr_index || 0^2
```

Exceptions:

None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the jump instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.