EEL 4712
Midterm 3 – Spring 2018
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 (4 points)	
4 (3 points)	
5 (3 points)	
6 (6 points)	
7 (3 points)	
8 (3 points)	
9 (3 points)	
10 (3 points)	
11 (3 points)	
12 (8 points)	
13 (12 points)	
14 (5 points)	
15 (8 points)	
16 (24 points)	
17 (6 points)	**6**

Total:

Regrade Info:

1) (3 points) **True/false**. During FPGA synthesis, a bus with 4 sources connected by 4 tristates gets replaced by an 8x1 mux.

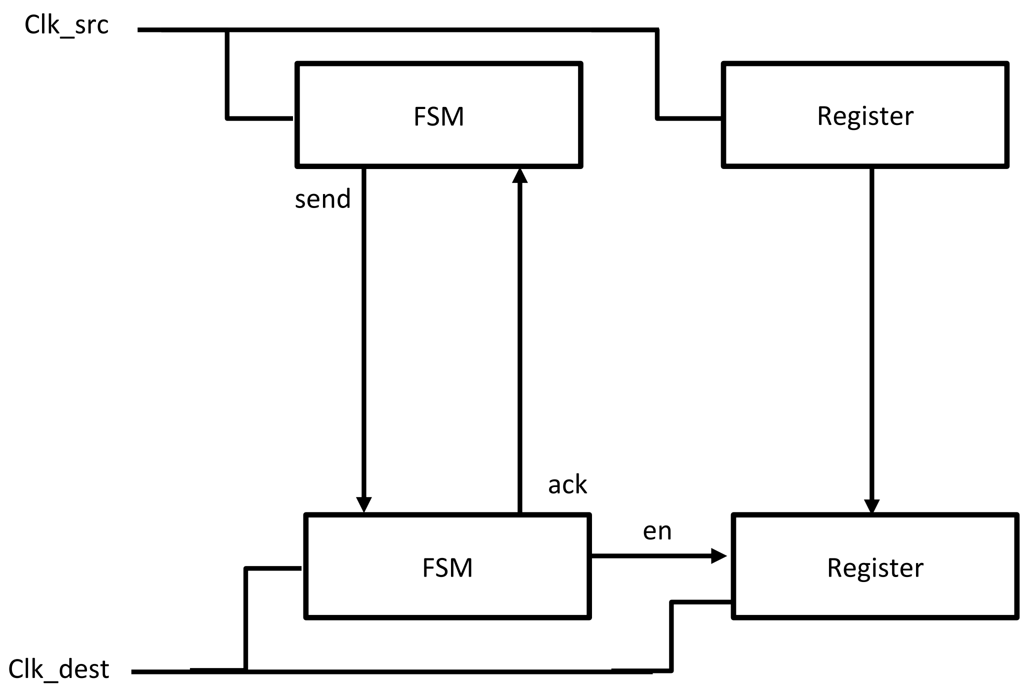
2) (3 points) What causes a flip-flop output to become metastable?

3) (4 points) What are the two situations where metastability is unavoidable?

4) (3 points) You are designing a circuit that feeds data to a pipeline from an external memory running on a different clock. What type of synchronizer should you use for this signal?

5) (3 points) You are designing a circuit that occasionally transfers a multi-bit control signal across clock domains. What type of synchronizer should you use for this signal?

6) (6 points) Complete the handshake architecture shown below by adding flip flops to the send and ack signals. Connect each flip flop to the appropriate clock.



7) (3 points) A MIPS Jump and Link instruction stores a return address into what register?

8) (3 points) What value is always stored in MIPS register 0?

9) (3 points) MIPS jump instructions specify a target using an instruction index instead of a byte address. How does the datapath convert this instruction index into a byte address? (Describe high-level functionality, not specific control signals)

10) (3 points) What MIPS instructions set the HI and LO registers?

11) (3 points) What range of byte addresses get mapped to the RAM inside of the memory entity?

12) (8 points) Write MIPS assembly code that matches the following behavior. Inport0 corresponds to byte address 0xFFF8. Outport corresponds to byte address 0xFFFC. Use $r0-$r31 for registers. Make two columns if necessary.

```
x = inport0;
if (x > 10) {
      y = 10;
else
      y = 15;
}
outport = y;
```

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 $r1, 0x000F($r0)
        addiu $r2, $r1, 0x7
        addiu $r3, $r0, 0xA
        xor $r4, $r2, $r3
        sw $r4, 0xFFFC($r0)
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. Area: 500 LUTs, Time: 18s
   b. Area: 700 LUTs, Time: 20s
   c. Area: 2000 LUTs, Time: 12s
   d. Area: 2500 LUTs, Time: 17s
   e. Area: 3000 LUTs, Time: 8s

15) (4 points) a. For the solution space in problem 14, what tradeoff (a-e) would be best if the optimization goal was to minimize execution time with a LUT constraint of 2500 LUTs?

(4 points) b. What tradeoff would be best for an optimization goal of minimizing LUTs given a time constraint of 10s?

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 < 10000; i++) {
   a[i] = b[i]*10 + b[i+1]*20 + b[i+2]*30 + b[i+3]*40;
}
```

<u>Datapath</u>
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) (6 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

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset
6	5	5	16

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 $vAddr_{1..0} \neq 0^2$ 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

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	immediate
6	5	5	16

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.

Exclusive OR — XOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110
6	5	5	5	5	6

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

Purpose:

To do a bitwise logical Exclusive OR

Description: `rd ← rs XOR rt`

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd.*

Restrictions:

None

Operation:

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

Store Word **SW**

31 26	25 21	20 16	15 0
SW 101011	base	rt	offset
6	5	5	16

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 $vAddr_{1..0} \ne 0^2$ 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**

31 26	25 0
J 000010	instr_index
6	26

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.