Midterm Review

1 Number Systems

1. Convert the following decimal numbers into 8-bit binary numbers and two-digit hexadecimal numbers.

Decimal	8-bit Binary	2-digit Hexadecimal
22_{10}		
100_{10}		

Solution:

Decimal	8-bit Binary	2-digit Hexadecimal
22_{10}	00010110_2	16_H
100_{10}	01100100_2	64_H

2. Convert the following decimal numbers into 8-bit binary numbers in sign-magnitude, one's complement, and two's complement format.

Decimal	8-bit Binary Sign-Magnitude	8-bit Binary One's Complement	8-bit Binary Two's Complement
-33_{10}			
-115_{10}			

Solution:

Decimal	8-bit Binary Sign-Magnitude	8-bit Binary One's Complement	8-bit Binary Two's Complement
-33_{10}	10100001_2	11011110_2	11011111_2
-115_{10}	11110011_2	10001100_2	10001101_2

3. Convert the following 8-bit binary (two's complement) numbers into decimal numbers.

8-bit Binary (Two's Complement)	Decimal
11010101_2	
10100011_2	

Solution:

8-bit Binary (Two's Complement)	Decimal
11010101_2	-43_{10}
10100011_2	-93_{10}

2 MIPS Machine Language (9 points)

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R	opcode (31–26)	rs (25–21)	rt (20–16)	rd (15–11)	shamt (10–6)	funct (5–0)
I	opcode (31–26)	rs (25–21)	rt (20–16)	immediate (15–0)		
J	opcode (31–26)	address (25–0)				

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

FLOATING-POINT INSTRUCTION FORMATS

FR	opcode (31–26)	fmt (25–21)	ft (20–16)	fs (15–11)	fd (10–6)	funct (5–0)
FI	opcode (31–26)	fmt (25–21)	ft (20–16)	immediate (15–0)		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

According to the MIPS reference data shown above, answer the following questions:

1. What is the 32-bit instruction (in binary) of the instruction `slt $t0, $s0, $a0`? (5 points)

   Solution: According to MIPS Reference Data, the instruction `slt` is R-type.

   - `op` (6 bits): 0_H or 000000_2
   - `rs` (5 bits): `$s0` or 16_{10} or 100000_2
   - `rt` (5 bits): `$a0` or 4_{10} or 00100_2
   - `rd` (5 bits): `$t0` or 8_{10} or 01000_2
   - `shamt` (5 bits): 00000_2
   - `funct` (6 bits): $2a_H$ or 101010_2

   Thus, the 32-bit machine instruction is 000000 10000 00100 01000 101010.

2. What is the 32-bit instruction (in binary) of the instruction `sb $s2, -12($sp)`? (4 points)

   Solution: According to MIPS Reference Data, the instruction `sb` is I-type.

   - `op` (6 bits): 28_H or 101000_2
   - `rs` (5 bits): `$sp` or 29_{10} or 11101_2
   - `rt` (5 bits): `$s2` or 28_{10} or 10010_2
   - `imm` (16 bits): -12_{10} or 1111111111110100_2

   Thus, the 32-bit machine instruction is 101000 11101 10010 1111111111110100.

3. Consider the following code fragment where number of the left column are addresses of instructions **in decimal**.

```
0020 |      add  $v0, $zero, $zero
0024 |loop: lb   $t1, 0($t0)
0028 |      beq  $t1, $zero, done
0032 |      addi $v0, $v0, 1
0036 |      addi $t0, $t0, 1
0040 |      j    loop
0044 |done: jr   $ra
```

   (a) What is the 32-bit machine instruction in binary of the instruction `beq $t1, $zero, done`?

   Solution: According to MIPS Reference Data, the instruction `beq` is I-type.

   - `op` (6 bits): 4_H or 000100_2
   - `rs` (5 bits): `$t1` or 9_{10} or 01001_2
   - `rt` (5 bits): `$zero` or 0_{10} or 00000_2
   - `Imm` (16 bits): The formula to calculate the address of the destination (PC) is as follows:

$$\text{PC} = \text{PC} + 4 + (\text{SignExt}(\text{Imm}) << 2)$$

Note that logically shift left by 2 is the same as multiply by 4. The instruction `beq` is located at the address 28_{10} and the destination (`done:`) is located at the address 44_{10}. Thus

$$\begin{aligned}
\text{PC} &= \text{PC} + 4 + (\text{SignExt}(\text{Imm}) * 4)\\
44 &= 28 + 4 + (\text{SignExt}(\text{Imm}) * 4)\\
44 &= 32 + (\text{SignExt}(\text{Imm}) * 4)\\
12 &= \text{SignExt}(\text{Imm}) * 4\\
3 &= \text{SignExt}(\text{Imm})\\
3 &= \text{Imm}
\end{aligned}$$

Thus, the 32-bit machine instruction is 000100 01001 00000 0000000000000011.

(b) What is the 32-bit machine instruction in binary of the instruction `j loop`?

Solution: According to MIPS Reference Data, the instruction `j` is J-type.

- `op` (6 bits): 2_H or 000010_2
- `address` (26 bits): The destination is the high 4 bits of the program counter (PC) of the jump instruction and appended it by the address field shift left by 2. Since the address the `j` instruction is 40_{10}, the top 4-bit of the address is 0000_2. Which is the same as the top 4-bit of the destination (24_{10}). Note that we check this just to make sure that the destination is not out-of-range. If the destination is in range, to calculate the address field, we use the following formula:

$$\begin{aligned}
\text{PC} &= \text{address} << 2\\
\text{PC} &= \text{address} * 4\\
24 &= \texttt{address} * 4\\
6 &= \texttt{address}
\end{aligned}$$

Thus, the 32-bit machine instruction is 000010 00000000000000000000000110.

3 MIPS Assembly Language

1. Consider the following assembly program:

```
.data
    x: .word 0x5
.text
    la   $a0, x
    lw   $a0, 0($a0)
    j    _inc1
    addi $v0, $zero, 10
    syscall
_inc1:
    addi $v0, $a0, 1
    ja   $ra
```

The above program can be assembled correctly. Will this program terminate properly? Why?

Solution: No. We did not use `jal` to call the function `_inc1`. Therefore, when we use `jr $ra`, it will jump to an unknown address.

2. Consider the following assembly program:

```
.data
        str: .asciiz "Computer Organization and Assembly Language\0"
.text
        la   $s0, str
loop:   lb   $a0, 0($s0)      # Set the character to be printed
        beq  $a0, $zero, done
        addi $v0, $zero, 11   # Syscall 11: print a character
        syscall               # Print the character in $a0
        addi $s0, $s0, 2
        j    loop
done:   addi $v0, $zero, 10   # Syscall 10: exit
        syscall               # Exit program
```

What is the output of the above program?

Solution:

```
Cmue raiainadAsml agae
```

Note that the above program prints every other character of the string `str`.

4 MIPS Assembly Language

Consider the main program of the following MIPS assembly code below:

```
.data
        str:    .asciiz  "Computer Organization"
        buffer: .space 100
.text
        la   $a0, str          # Set input address for _toUpper
        la   $a1, buffer       # Set output buffer for _toUpper
        jal  _toUpper          # Call toUpper
        la   $a0, buffer       # Set the string to be printed
        addi $v0, $zero, 4     # Syscall 4: print string
        syscall                # Print the string buffer
        addi $v0, $zero, 10    # Syscall 10: exit
        syscall                # Exit program
```

Implement the function `_toLower` that converts every lowercase character in an input string to uppercase character and store the result in an output string. This function should take two arguments, `$a0` is the address of an input string, and `$a1` is an address of an output buffer. No return value for this function. Implement this function that satisfies all calling conventions discussed in class. **Note** the lowercase character `'a'` is 97 in decimal, `'b'`, is 98 in decimal, and so on. The lowercase character `'z'` is 122 in decimal.

Solution:

```
_incChar:
        addi $t0, $zero, 97         # 'a' is 97
        addi $t1, $zero, 127        # 'z' is 122
loop:   lb   $t2, 0($a0)            # $t2 = c (a character)
        beq  $t2, $zero, done       # if c == \0, done
        slt  $t3, $t2, $t0          # $t3 will be 1 if c < 97
        bne  $t3, $zero, copyChar   # c < 97, not a lowercase. Just copy
        slt  $t3, $t1, $t2          # $t3 will be 1 if 122 < c
        bne  $t3, $zero, copyChar   # c > 122, not a lowercase, Just copy
        addi $t2, $t2, -32          # Convert c to lowercase
copyChar:
        sb   $t2, 0($a1)            # Store the character to buffer
        addi $a0, $a0, 1            # Increase input address by 1
        addi $a1, $a1, 1            # Increase buffer address by 1
        j    loop                   # Go back to loop
done:   lb   $zero, 0($a1)          # Store null character to buffer
        jr   $ra                    # Go back to caller
```