IT5002 Tutorial 2

AY 2025/26 Semester 1

Prepared by Michael Yang

Slides adapted from Theodore Leebrant, Prof. Colin and Prof. Aaron

Recap: Bitwise Operations

AND : 1 & 1 = 1 1 & 0 = 0 0 & 0 = 0

OR : 1 | 1 = 1 1 | 0 = 1 0 | 0 = 0

XOR : 1 ^ 1 = 0 1 ^ 0 = 1 0 ^ 0 = 0

NOT : ~1 = 0 ~0 = 1

Left shift : 0b001 << 1 = 0b010

Right shift : 0b010 >> 1 = 0b001

Q1. MIPS bitwise operations

Implement bitwise operations in MIPS. $s0 is mapped to a, $s1 to b, $s2 to c.

For bitwise instructions (e.g. ori, andi) express immediate values in binary.

Bit 31 = MSB. Bit 0 = LSB

(a) Set bits 2, 8, 9, 14 and 16 of **b** to 1. Leave all other bits unchanged.

* Important: `ori` can only be used to set the lower 16 bits

- The higher bits (16-31) cannot be set using `ori`
- They have to be set using `lui`

Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}

(3) ZeroExtImm = { 16{1b'0}, immediate }

Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}	f_{hex}

To set bits, we create a "mask" with 1's in the bit positions we want to set.
Since bit 16 is in the upper 16 bits of the register, we need to use `lui` to set it.

```
lui $t0, 1                            # Set bit 16 of $t0.
ori $t0, $t0, 0b0100001100000100      # Set bits 14, 9, 8 and 2.
or $s1, $s1, $t0
```

(b) Copy over bits 1, 3 and 7 of **b** into **a**, without changing any other bits of **a**

Strategy:

- Extract bits 1, 3 and 7 from **b** (into a temporary register)
- Mask out bits 1, 3 and 7 from **a**
- Transfer the bits over using bitwise OR

```
andi $t0, $s1, 0b0000000010001010  # bits 1,3,7 in b
lui $t1, 0b1111111111111111        # load the upper mask
ori $t1, $t1, 0b1111111101110101   # load the lower mask
and $s0, $s0, $t1                  # mask out 1,3,7 in a
or $s0, $s0, $t0                   # transfer bits using OR
```

(c) Make bits 2, 4 and 8 of **c** the **inverse** of bits 1, 3 and 7 of **b**, without changing any other bits of c.

b: 0 0000 1010

c: 0 0001 1111 (originally)

c: 1 0000 1011 (new)

Strategy:

- Extract the bits 1, 3, and 7 from **b** (into a temporary register)
- Take their inverse using XOR
- Shift the extracted bits to the left by 1
- Clear the bits 2, 4, and 8 from **c**
- Load the extracted bits into **c**

```
xori $t0, $s1, 0b10001010     # Extract the bits, take their inverse
                              # And put the results in $t0

andi $t0, $t0, 0b10001010     # Mask out the other bits in $t0

sll $t0, $t0, 1               # Shift bits to the left by 1

lui $t1, 0b1111111111111111   # Create a mask in $t1
ori $t1, $t1, 0b1111111011101011
and $s2, $s2, $t1             # Use $t1 to mask out bits 2,4,8 in c

or $s2, $s2, $t0              # Load the extracted bits into c
```

Q2. MIPS tracing

```
      add  $t0, $s0, $zero
      lui  $t1, 0x8000
lp:   beq  $t0, $zero, e
      andi $t2, $t0, 1
      beq  $t2, $zero, s
      xor  $s0, $s0, $t1
s:    srl  $t0, $t0, 1
      j    lp
e:
```

Give the final hexadecimal value in $s0 for each of the initial values in $s0:

1. Decimal value 31
2. 0x0AAA AAAA

$s0 is a 31-bit binary sequence.

MSB of $s0 is assumed to be 0 at the start.

$$31 = \overbrace{0000\ 0000\ 0000\ 0000}^{\text{Upper 16 bits}}\ \overbrace{0000\ 0000\ 0001\ 1111}^{\text{Lower 16 bits}}\ \text{(in 32 bits)}$$

Set $t0 = $s0 + 0

```
      add  $t0, $s0, $zero
      lui  $t1, 0x8000
lp:   beq  $t0, $zero, e
      andi $t2, $t0, 1
      beq  $t2, $zero, s
      xor  $s0, $s0, $t1
s:    srl  $t0, $t0, 1
      j    lp
e:
```

Register	Value
$s0	0000 ... 0000 0000 0001 1111
$t0	0000 ... 0000 0000 0001 1111
$t1	?
$t2	?

Upper 16 bits Lower 16 bits

$$31 = \underbrace{0000\ 0000\ 0000\ 0000}_{\text{Upper 16 bits}}\ \underbrace{0000\ 0000\ 0001\ 1111}_{\text{Lower 16 bits}} \text{ (in 32 bits)}$$

Set the upper 16 bits of $t1 to
1000 0000 0000 0000
(Lower bits are set to 0)

```
     add  $t0, $s0, $zero
     lui  $t1, 0x8000
lp:  beq  $t0, $zero, e
     andi $t2, $t0, 1
     beq  $t2, $zero, s
     xor  $s0, $s0, $t1
s:   srl  $t0, $t0, 1
     j    lp
e:
```

Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}	f_{hex}

0000 ... 0000 0000 0001 1111
$s0

0000 ... 0000 0000 0001 1111
$t0

1000 ... 0000 0000 0000 0000
$t1

?
$t2

$$31 = \overbrace{0000\ 0000\ 0000\ 0000}^{\text{Upper 16 bits}}\ \overbrace{0000\ 0000\ 0001\ 1111}^{\text{Lower 16 bits}} \text{ (in 32 bits)}$$

Jump to label 'e' if $t0 is equal to 0. It is not, so continue to next instruction

```
      add  $t0, $s0, $zero
      lui  $t1, 0x8000
lp:   beq  $t0, $zero, e
      andi $t2, $t0, 1
      beq  $t2, $zero, s
      xor  $s0, $s0, $t1
s:    srl  $t0, $t0, 1
      j    lp
e:
```

Register	Value
$s0	0000 ... 0000 0000 0001 1111
$t0	0000 ... 0000 0000 0001 1111
$t1	1000 ... 0000 0000 0000 0000
$t2	?

$$31 = \underbrace{0000\ 0000\ 0000\ 0000}_{\text{Upper 16 bits}}\ \underbrace{0000\ 0000\ 0001\ 1111}_{\text{Lower 16 bits}} \text{ (in 32 bits)}$$

Set $t2 = $t0 & 1 →

```
      add  $t0, $s0, $zero
      lui  $t1, 0x8000
lp:   beq  $t0, $zero, e
      andi $t2, $t0, 1
      beq  $t2, $zero, s
      xor  $s0, $s0, $t1
s:    srl  $t0, $t0, 1
      j    lp
e:
```

Register	Value
$s0	0000 ... 0000 0000 0001 1111
$t0	0000 ... 0000 0000 0001 1111
$t1	1000 ... 0000 0000 0000 0000
$t2	0000 ... 0000 0000 0000 0001

$$31 = \underbrace{0000\ 0000\ 0000\ 0000}_{\text{Upper 16 bits}}\ \underbrace{0000\ 0000\ 0001\ 1111}_{\text{Lower 16 bits}} \text{ (in 32 bits)}$$

```
    add  $t0, $s0, $zero
    lui  $t1, 0x8000
lp: beq  $t0, $zero, e
    andi $t2, $t0, 1
    beq  $t2, $zero, s
    xor  $s0, $s0, $t1
s:  srl  $t0, $t0, 1
    j    lp
e:
```

Branch to label 's' if $t2 is equal to $0.

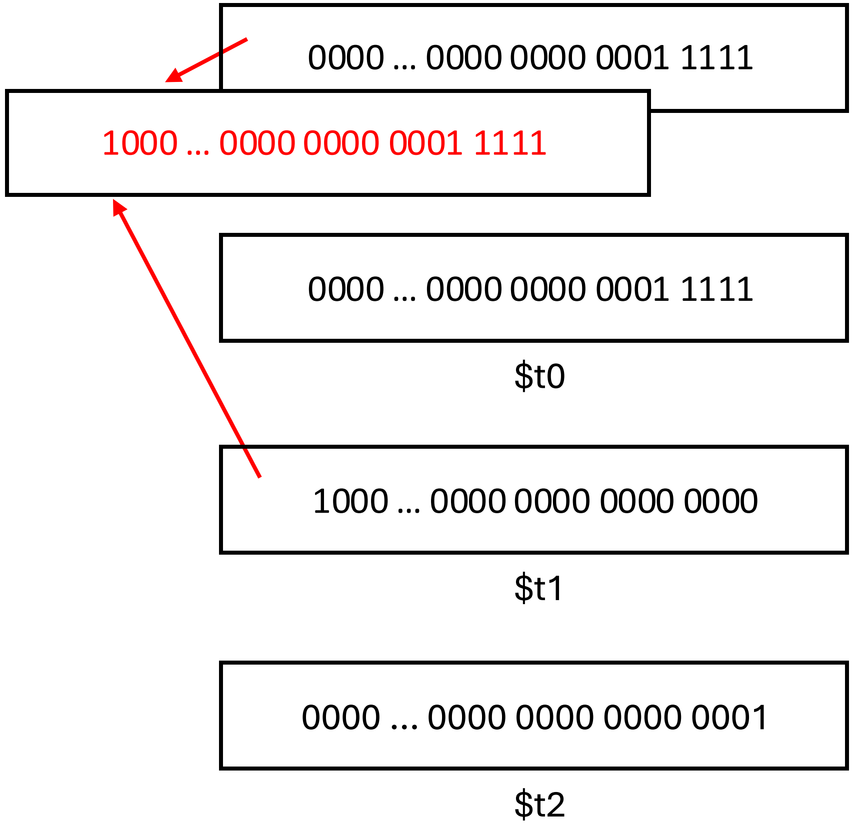
It is not, so continue to next instruction

Register	Value
$s0	0000 ... 0000 0000 0001 1111
$t0	0000 ... 0000 0000 0001 1111
$t1	1000 ... 0000 0000 0000 0000
$t2	0000 ... 0000 0000 0000 0001

$$31 = \overbrace{0000\ 0000\ 0000\ 0000}^{\text{Upper 16 bits}}\ \overbrace{0000\ 0000\ 0001\ 1111}^{\text{Lower 16 bits}} \text{ (in 32 bits)}$$

Set $s0 = $s0 ^ $t1

```
    add  $t0, $s0, $zero
    lui  $t1, 0x8000
lp: beq  $t0, $zero, e
    andi $t2, $t0, 1
    beq  $t2, $zero, s
    xor  $s0, $s0, $t1
s:  srl  $t0, $t0, 1
    j    lp
e:
```



$$31 = \underbrace{0000\ 0000\ 0000\ 0000}_{\text{Upper 16 bits}}\ \underbrace{0000\ 0000\ 0001\ 1111}_{\text{Lower 16 bits}} \text{ (in 32 bits)}$$

```
      add  $t0, $s0, $zero
      lui  $t1, 0x8000
lp:   beq  $t0, $zero, e
      andi $t2, $t0, 1
      beq  $t2, $zero, s
      xor  $s0, $s0, $t1
s:    srl  $t0, $t0, 1
      j    lp
e:
```

Shift the bits in $t0 to the right by 1

The least significant bit is essentially 'discarded'

Register	Value
$s0	1000 ... 0000 0000 0001 1111
$t0	0000 ... 0000 0000 0000 1111
$t1	1000 ... 0000 0000 0000 0000
$t2	0000 ... 0000 0000 0000 0001

$$31 = \overbrace{0000\ 0000\ 0000\ 0000}^{\text{Upper 16 bits}}\ \overbrace{0000\ 0000\ 0001\ 1111}^{\text{Lower 16 bits}} \text{ (in 32 bits)}$$

```
      add  $t0, $s0, $zero
      lui  $t1, 0x8000
lp:   beq  $t0, $zero, e
      andi $t2, $t0, 1
      beq  $t2, $zero, s
      xor  $s0, $s0, $t1
s:    srl  $t0, $t0, 1
      j    lp
e:
```

Unconditional jump to the label 'lp' → `j lp`

Register	Value
$s0	1000 ... 0000 0000 0001 1111
$t0	0000 ... 0000 0000 0000 1111
$t1	1000 ... 0000 0000 0000 0000
$t2	0000 ... 0000 0000 0000 0001

In summary: the loop ends once $t0 becomes 0.
We use $t2 to store whether or not $t0 has bit 0 set.
If $t2 is 0 (i.e. $t0 does not have bit 0 set), then we go to the shift right instruction. Otherwise, we use $t1 to flip the MSB of $s0.
At the end of each iteration of the loop, the bits in $t0 are shifted to the right by 1.

31 = $\overbrace{\text{0000 0000 0000 0000}}^{\text{Upper 16 bits}}\ \overbrace{\text{0000 0000 0001 1111}}^{\text{Lower 16 bits}}$ (in 32 bits)

```
      add  $t0, $s0, $zero
      lui  $t1, 0x8000
lp:   beq  $t0, $zero, e
      andi $t2, $t0, 1
      beq  $t2, $zero, s
      xor  $s0, $s0, $t1
s:    srl  $t0, $t0, 1
      j    lp
e:
```

Register	Value
$s0	1000 … 0000 0000 0001 1111
$t0	0000 … 0000 0000 0000 1111
$t1	1000 … 0000 0000 0000 0000
$t2	0000 … 0000 0000 0000 0001

# of times the MSB of $s0 is flipped = # of '1' bits in $t0

If there are an **odd** number of 1 bits, the final value of the MSB is 1.
If there are an **even** number of 1 bits, the final value of the MSB is 0.

So the final value of $s0 is 1000 … 0000 0001 1111 = 0x8000 001F

If the initial value of $s0 is

0x0AAAAAAA = 0000 1010 1010 1010 1010 1010 1010 1010

There are 2x7=14 '1' bits -> even number of 1 bits

So the MSB of $s0 will still be 0

Therefore the final value of $s0 is still 0x0AAAAAAA

(b) Explain the purpose of the code in one sentence.

The code sets bit 31 of $s0 to 1 if there are odd number of ‘1’ in $s0 initially, or 0 if there are even number of ‘1’.

(This is called the odd parity bit scheme / even parity bit scheme)

Q3. More MIPS tracing

```
        addi $t0, $s0, 0
        addi $t1, $s1, 0
loop:   lw   $t3, 0($t0)
        lw   $t4, 0($t1)
        slt  $t5, $t4, $t3        # line A
        beq  $t5, $zero, skip     # line B
        sw   $t4, 0($t0)
        sw   $t3, 0($t1)
skip:   addi $t0, $t0, 4
        addi $t1, $t1, 4
        bne  $t3, $zero, loop
```

```
        addi $t0, $s0, 0
        addi $t1, $s1, 0
loop:   lw   $t3, 0($t0)
        lw   $t4, 0($t1)
        slt  $t5, $t4, $t3       # line A
        beq  $t5, $zero, skip    # line B
        sw   $t4, 0($t0)
        sw   $t3, 0($t1)
skip:   addi $t0, $t0, 4
        addi $t1, $t1, 4
        bne  $t3, $zero, loop
```

Base address of array A is stored in $s0
Base address of array B is stored in $s1

Example:
$s0 = 0x0000 1000
$s1 = 0x0000 2000

A:	7	4	1	6	0	5	9	0
	0x0000 1000	0x0000 1004	0x0000 1008	0x0000 100C	0x0000 1010	0x0000 1014	0x0000 1018	0x0000 101C
B:	3	4	5	2	1	0	0	9
	0x0000 2000	0x0000 2004	0x0000 2008	0x0000 200C	0x0000 2010	0x0000 2014	0x0000 2018	0x0000 201C

Load value of $s0 and $s1 into $t0 and $t1 respectively

$t0	$t1	$t3	$t4	$t5
0x0000 1000	0x0000 2000	?	?	?

$s0	$s1
0x0000 1000	0x0000 2000

```
        addi $t0, $s0, 0
        addi $t1, $s1, 0
loop:   lw   $t3, 0($t0)
        lw   $t4, 0($t1)
        slt  $t5, $t4, $t3       # line A
        beq  $t5, $zero, skip    # line B
        sw   $t4, 0($t0)
        sw   $t3, 0($t1)
skip:   addi $t0, $t0, 4
        addi $t1, $t1, 4
        bne  $t3, $zero, loop
```

Base address of array A is stored in $s0
Base address of array B is stored in $s1

Example:
$s0 = 0x0000 1000
$s1 = 0x0000 2000

A:	7	4	1	6	0	5	9	0
	0x0000 1000	0x0000 1004	0x0000 1008	0x0000 100C	0x0000 1010	0x0000 1014	0x0000 1018	0x0000 101C

B:	3	4	5	2	1	0	0	9
	0x0000 2000	0x0000 2004	0x0000 2008	0x0000 200C	0x0000 2010	0x0000 2014	0x0000 2018	0x0000 201C

Load Mem[$t0 + 0] into $t3. Load Mem[$t1 + 0] into $t4.

0x0000 1000	0x0000 2000	7	3	?
$t0	$t1	$t3	$t4	$t5

0x0000 1000	0x0000 2000
$s0	$s1

Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}

```
        addi $t0, $s0, 0
        addi $t1, $s1, 0
loop:   lw   $t3, 0($t0)
        lw   $t4, 0($t1)
        slt  $t5, $t4, $t3         # line A
        beq  $t5, $zero, skip      # line B
        sw   $t4, 0($t0)
        sw   $t3, 0($t1)
skip:   addi $t0, $t0, 4
        addi $t1, $t1, 4
        bne  $t3, $zero, loop
```

Base address of array A is stored in $s0
Base address of array B is stored in $s1

Example:
$s0 = 0x0000 1000
$s1 = 0x0000 2000

A:

7	4	1	6	0	5	9	0
0x0000 1000	0x0000 1004	0x0000 1008	0x0000 100C	0x0000 1010	0x0000 1014	0x0000 1018	0x0000 101C

B:

3	4	5	2	1	0	0	9
0x0000 2000	0x0000 2004	0x0000 2008	0x0000 200C	0x0000 2010	0x0000 2014	0x0000 2018	0x0000 201C

Set $t5 to the result of ($t4 < $t3) (1 if true, 0 if false)

$t0	$t1	$t3	$t4	$t5
0x0000 1000	0x0000 2000	7	3	1

$s0	$s1
0x0000 1000	0x0000 2000

Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	0 / $2a_{hex}$

```
      addi $t0, $s0, 0
      addi $t1, $s1, 0
loop: lw   $t3, 0($t0)
      lw   $t4, 0($t1)
      slt  $t5, $t4, $t3     # line A
      beq  $t5, $zero, skip  # line B
      sw   $t4, 0($t0)
      sw   $t3, 0($t1)
skip: addi $t0, $t0, 4
      addi $t1, $t1, 4
      bne  $t3, $zero, loop
```

Base address of array A is stored in $s0
Base address of array B is stored in $s1

Example:
$s0 = 0x0000 1000
$s1 = 0x0000 2000

A:

3	4	1	6	0	5	9	0
0x0000 1000	0x0000 1004	0x0000 1008	0x0000 100C	0x0000 1010	0x0000 1014	0x0000 1018	0x0000 101C

B:

7	4	5	2	1	0	0	9
0x0000 2000	0x0000 2004	0x0000 2008	0x0000 200C	0x0000 2010	0x0000 2014	0x0000 2018	0x0000 201C

Branch to the label 'skip' if $t5 is equal to 0.

0x0000 1000	0x0000 2000	7	3	1
$t0	$t1	$t3	$t4	$t5

0x0000 1000	0x0000 2000
$s0	$s1

It is not, so store the word at $t4 into Mem[$t0 + 0].
Store the word at $t3 into Mem[$t1 + 0].

```
        addi $t0, $s0, 0
        addi $t1, $s1, 0
loop:   lw   $t3, 0($t0)
        lw   $t4, 0($t1)
        slt  $t5, $t4, $t3      # line A
        beq  $t5, $zero, skip   # line B
        sw   $t4, 0($t0)
        sw   $t3, 0($t1)
skip:   addi $t0, $t0, 4
        addi $t1, $t1, 4
        bne  $t3, $zero, loop
```

Set $t0 = $t0 + 4, set $t1 = $t1 + 4

$t0	$t1	$t3	$t4	$t5
0x0000 1004	0x0000 2004	7	3	1

$s0	$s1
0x0000 1000	0x0000 2000

If $t3 is not equal to $0, branch to the label 'loop'

Base address of array A is stored in $s0
Base address of array B is stored in $s1

Example:
$s0 = 0x0000 1000
$s1 = 0x0000 2000

A:

0x0000 1000	0x0000 1004	0x0000 1008	0x0000 100C	0x0000 1010	0x0000 1014	0x0000 1018	0x0000 101C
3	4	1	6	0	5	9	0

B:

0x0000 2000	0x0000 2004	0x0000 2008	0x0000 200C	0x0000 2010	0x0000 2014	0x0000 2018	0x0000 201C
7	4	5	2	1	0	0	9

```
        addi $t0, $s0, 0
        addi $t1, $s1, 0
loop:   lw   $t3, 0($t0)
        lw   $t4, 0($t1)
        slt  $t5, $t4, $t3       # line A
        beq  $t5, $zero, skip    # line B
        sw   $t4, 0($t0)
        sw   $t3, 0($t1)
skip:   addi $t0, $t0, 4
        addi $t1, $t1, 4
        bne  $t3, $zero, loop
```

Base address of array A is stored in $s0
Base address of array B is stored in $s1

Example:
$s0 = 0x0000 1000
$s1 = 0x0000 2000

A:

3	4	1	6	0	5	9	0
0x0000 1000	0x0000 1004	0x0000 1008	0x0000 100C	0x0000 1010	0x0000 1014	0x0000 1018	0x0000 101C

B:

7	4	5	2	1	0	0	9
0x0000 2000	0x0000 2004	0x0000 2008	0x0000 200C	0x0000 2010	0x0000 2014	0x0000 2018	0x0000 201C

(a) What is the purpose of $t1?

0x0000 1004	0x0000 2004	7	3	1
$t0	$t1	$t3	$t4	$t5

0x0000 1000	0x0000 2000
$s0	$s1

To store the **address** of the element in B that will be read

```
        addi $t0, $s0, 0
        addi $t1, $s1, 0
loop:   lw   $t3, 0($t0)
        lw   $t4, 0($t1)
        slt  $t5, $t4, $t3      # line A
        beq  $t5, $zero, skip   # line B
        sw   $t4, 0($t0)
        sw   $t3, 0($t1)
skip:   addi $t0, $t0, 4
        addi $t1, $t1, 4
        bne  $t3, $zero, loop
```

Base address of array A is stored in $s0
Base address of array B is stored in $s1

Example:
$s0 = 0x0000 1000
$s1 = 0x0000 2000

A:	3	4	1	6	0	5	9	0
	0x0000 1000	0x0000 1004	0x0000 1008	0x0000 100C	0x0000 1010	0x0000 1014	0x0000 1018	0x0000 101C
B:	7	4	5	2	1	0	0	9
	0x0000 2000	0x0000 2004	0x0000 2008	0x0000 200C	0x0000 2010	0x0000 2014	0x0000 2018	0x0000 201C

(b) Give the final content of these two arrays.

In one iteration, the program:

- Reads the value of A[i] and B[j] (i is the pointer for A, j is the pointer for B)
- Compare the values: if A[i] > B[j], swap them
- Update the pointers: i=i+1, j=j+1
- Continuation condition: if $t3 (which is A[i]) is not equal to 0, go back to start of the loop

A = {3, 4, 1, 2, 0, 5, 9, 0}
B = {7, 4, 5, 6, 1, 0, 0, 9}

```
      addi $t0, $s0, 0
      addi $t1, $s1, 0
loop: lw   $t3, 0($t0)
      lw   $t4, 0($t1)
      slt  $t5, $t4, $t3        # line A
      beq  $t5, $zero, skip     # line B
      sw   $t4, 0($t0)
      sw   $t3, 0($t1)
skip: addi $t0, $t0, 4
      addi $t1, $t1, 4
      bne  $t3, $zero, loop
```

Base address of array A is stored in $s0
Base address of array B is stored in $s1

Example:
$s0 = 0x0000 1000
$s1 = 0x0000 2000

(c) How many **store word (sw)** operations are performed?

A = {7, 4, 1, 6, 0, 5, 9, 0}
B = {3, 4, 5, 2, 1, 0, 0, 9}

- Store word only occurs when a[i] > b[j]
- Each time, two **sw** operations are incurred
- So in total, 2 x 2 = <u>4</u>

```
                 addi $t0, $s0, 0
                 addi $t1, $s1, 0
PC-32 ->   loop: lw   $t3, 0($t0)
PC-28 ---------> lw   $t4, 0($t1)
PC-24 ---------> slt  $t5, $t4, $t3      # line A
PC-20 ---------> beq  $t5, $zero, skip   # line B
PC-16 ---------> sw   $t4, 0($t0)
PC-12 ---------> sw   $t3, 0($t1)
PC-8  ->   skip: addi $t0, $t0, 4
PC-4  ---------> addi $t1, $t1, 4
PC    ---------> bne  $t3, $zero, loop
```

(d) What is the value (in decimal) of the **immediate** field in the machine code representation of the **bne** instruction?

Branch On Not Equal bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}

(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }

14 bits of immediate[15] (MSB)
Immediate (16 bits)
2x0 bits
PC is a 32-bit address, so we need to extend immediate to 32 bits long

Relative to the bne instruction, the loop label is at address PC-32. (Recall that branch only supports relative jumps w.r.t current PC, up to +/-2^{15} word addresses i.e. +/-2^{17} byte addresses)

But since the new PC is calculated as PC=PC+4+BranchAddr, we need to subtract another 4 byte addresses, i.e. BranchAddr should be –36, so that afterwards PC = PC+4-36 = PC-32

So BranchAddr is –36 (bytes addresses) = **-9** (words addresses)

* Important: Immediate specifies the relative **word** address

```
      addi $t0, $s0, 0
      addi $t1, $s1, 0
loop: lw   $t3, 0($t0)
      lw   $t4, 0($t1)
      slt  $t5, $t4, $t3       # line A
      beq  $t5, $zero, skip    # line B
      sw   $t4, 0($t0)
      sw   $t3, 0($t1)
skip: addi $t0, $t0, 4
      addi $t1, $t1, 4
      bne  $t3, $zero, loop
```

(e) The two lines indicated as “line A” and “line B” represent the translation of a MIPS pseudo-instruction.

Give the corresponding pseudo-instruction.

- $t5 is set to the result of ($t4 < $t3) (1 if true, 0 if false)
- If $t5 is equal to $0, jump to the label 'skip'

In other words,
- If ($t4 < $t3) is false, jump to the label 'skip'
- I.e. if ($t4 >= $t3), jump to the label 'skip'
- I.e. `bge $t4, $t3, skip`
- Branch to 'skip' if $t4 is greater than or equal to $t3

Q4. MIPS instruction encoding

Instruction Encoding	MIPS Code
	# `$s1` stores the result, `$t0` stores a non-negative number
	`addi $s1, $zero, 0` #Inst. address is 0x00400028
`0x00084042`	`loop: srl $t0, $t0, 1`
`0x11000002`	
`0x22310001`	
	`j loop`
	`exit:`

Instruction Encoding	MIPS Code
	# **$s1** stores the result, **$t0** stores a non-negative number
	`addi $s1, $zero, 0` #Inst. address is 0x00400028
`0x00084042`	`loop: srl $t0, $t0, 1`
`0x11000002`	
`0x22310001`	
	`j loop`
	`exit:`

<u>addi $s1, $zero, 0</u>

Opcode = $(8)_{hex}$ = 0b001 000

rt = $s1 = 17 = 0b10001

rs = $zero = 0b00000

SignExtImm = 0

Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}

opcode	rs	rt	imm
001000	00000	10001	0000 0000 0000 0000

Note that although the instruction is written as `addi $rt, $rs, imm` `$rs` comes **before** `$rt` when encoding the instruction!

Instruction Encoding	MIPS Code
	# **$s1** stores the result, **$t0** stores a non-negative number
	addi $s1, $zero, 0 #Inst. address is 0x00400028
0x00084042	**loop: srl $t0, $t0, 1**
0x11000002	
0x22310001	
	j loop
	exit:

<u>addi $s1, $zero, 0</u>

0010 0000 0001 0001 0000 0000 0000 0000

2 0 1 1 0 0 0 0

Ans: 0x2011 0000

001000	00000	10001	0000 0000 0000 0000
opcode	rs	rt	imm

Instruction Encoding	MIPS Code
	# **$s1** stores the result, **$t0** stores a non-negative number
	`addi $s1, $zero, 0` #Inst. address is 0x00400028
`0x00084042`	`loop: srl $t0, $t0, 1`
`0x11000002`	
`0x22310001`	
	`j loop`
	`exit:`

0x1100 0002 =

0001 0001 0000 0000 0000 0000 0000 0010

Opcode is 0x04 = 4_{hex}

This is a `beq` instruction (I-type instruction)

Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}

opcode	rs	rt	imm
0001 00	01 000	0 0000	0000 0000 0000 0010

\$rs = 01000 = 8 = \$t0 \$rt = 00000 = \$zero

Imm = 2

`beq $t0, $zero, 2 / beq $t0, $zero, exit`

Instruction Encoding	MIPS Code
	# **$s1** stores the result, **$t0** stores a non-negative number
	addi $s1, $zero, 0 #Inst. address is 0x00400028
0x00084042	**loop: srl $t0, $t0, 1**
0x11000002	
0x22310001	
	j loop
	exit:

0x2231 0001 =

0010 0010 0011 0001 0000 0000 0000 0001

Opcode is 0x08 = 8_{hex}

This is an `addi` instruction (I-type instruction)

0010 00	10 001	1 0001	0000 0000 0000 0001
opcode	rs	rt	imm

$rs = 10001 = 17 = $s1 $rt = 10001 = $s1

Imm = 1

`addi $s1, $s1, 1`

Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}

Instruction Encoding	MIPS Code
	# **$s1** stores the result, **$t0** stores a non-negative number
	`addi $s1, $zero, 0` #Inst. address is 0x00400028
`0x00084042`	`loop: srl $t0, $t0, 1`
`0x11000002`	
`0x22310001`	
	`j loop`
	`exit:`

J-type instruction:	6 bit opcode	26 bit imm

Recall that we only jump to word-aligned addresses, so the last 2 bits are assumed to be 0, allowing us to specify up to 28 bits of a 32-bit address

The four most significant bits of the address are taken from the current PC value

Instruction Encoding	MIPS Code
	# **$s1** stores the result, **$t0** stores a non-negative number
	`addi $s1, $zero, 0` #Inst. address is 0x00400028
`0x00084042`	`loop: srl $t0, $t0, 1`
`0x11000002`	
`0x22310001`	
	`j loop`
	`exit:`

`j loop`

Address of 'loop' label:

0x0040 0028 + 4 = 0x0040 002C

= 0000 0000 0100 0000 0000 0000 0010 1100

The `j loop` instruction is in the same address 'block' as the loop label

I.e. the 4 most significant bits of the PC are also 0000

Remove the first 4 bits of the target address

Also, remove the last 2 bits (they are implicit in the `imm` value)

Instruction Encoding	MIPS Code
	# **$s1** stores the result, **$t0** stores a non-negative number
	`addi $s1, $zero, 0` #Inst. address is 0x00400028
`0x00084042`	`loop: srl $t0, $t0, 1`
`0x11000002`	
`0x22310001`	
	`j loop`
	`exit:`

`j loop`

~~0000~~ 0000 0100 0000 0000 0000 0010 11~~00~~

Imm = 0000 0100 0000 0000 0000 0010 11

Jump	j	J	PC=JumpAddr	(5)	2_{hex}

0000 10	0000 0100 0000 0000 0000 0010 11

Ans: 0x0810 000B

Instruction Encoding	MIPS Code
	# `$s1` stores the result, `$t0` stores a non-negative number
`0x20110000`	`addi $s1, $zero, 0` #Inst. address is 0x00400028
`0x00084042`	`loop: srl $t0, $t0, 1`
`0x11000002`	`beq $t0, $zero, exit`
`0x22310001`	`addi $s1, $s1, 1`
`0x0810000B`	`j loop`
	`exit:`

(b) Give a simple mathematical expression for the relationship between $t0 and $s1.

- $s1 initialized to 0
- In one iteration of the loop:
    - Shift the bits in $t0 to the right by 1
    - Branch to exit (i.e. stop) if $t0 is equal to 0
    - Otherwise increment $s1 by 1

In other words

- Count the number of 1's in $t0, minus 1
- Count the number of times $t0 needs to be divided by 2 for it to reach 0, minus 1
- A.k.a $\lfloor \log_2 \$t0 \rfloor$ (floor of log base 2 of $t0)

Slides uploaded to https://github.com/michaelyql/IT5002

Email: e1121035@u.nus.edu

Anonymous feedback: https://bit.ly/feedback-michael

(or scan the QR below)