

Final Review

Logic Gates And Boolean Algebra

3/25 - Comparators

- $A == B \rightarrow !(A \text{ xor } B)$
- $A < B \rightarrow !A B$
- $A > B \rightarrow A !B$

2 bit Comparator

A < B when:

$$(A_1 < B_1) \text{ OR } ((A_1 == B_1) \text{ AND } (A_0 < B_0))$$

How to translate this into a boolean expression?

$$\overline{A}_1 \overline{B}_1 + \left(\overline{A}_1 \oplus \overline{B}_1 \cdot \overline{A}_0 \overline{B}_0 \right)$$

- Queens College CSCI 240

A > B when:

$$(A_1 > B_1) \text{ OR } ((A_1 == B_1) \text{ AND } (A_0 > B_0))$$

How to translate this into a boolean expression?

$$A_1 \overline{B}_1 + \left(\overline{A}_1 \oplus \overline{B}_1 \cdot A_0 \overline{B}_0 \right)$$

- Queens College CSCI 240

3 bit comparator

A == B when:

$$(A_2 == B_2) \text{ AND } (A_1 == B_1) \text{ AND } (A_0 == B_0)$$

How to translate this into a boolean expression?

$$\overline{A}_2 \oplus \overline{B}_2 \cdot \overline{A}_1 \oplus \overline{B}_1 \cdot \overline{A}_0 \oplus \overline{B}_0$$

- Queens College CSCI 240

A > B when:

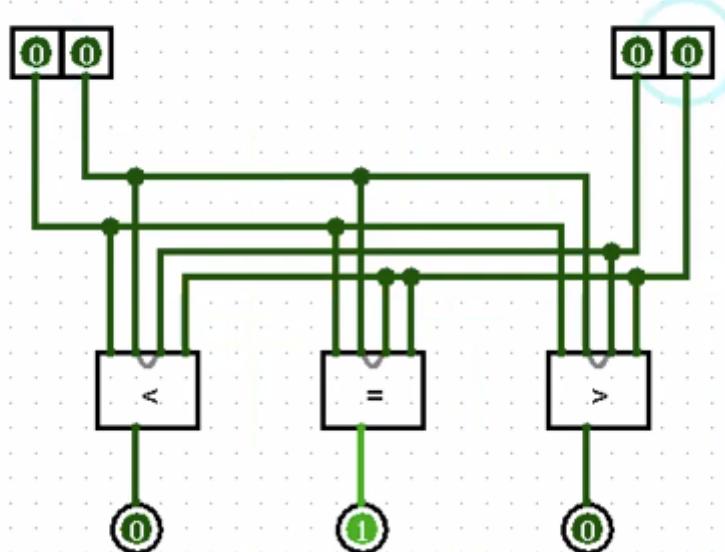
$$(A_2 > B_2) \text{ OR } ((A_2 == B_2) \text{ AND } (A_1 > B_1)) \text{ OR } ((A_2 == B_2) \text{ AND } (A_1 == B_1) \text{ AND } (A_0 > B_0))$$

How to translate this into a boolean expression?

$$(A_2 \bar{B}_2) + (\overline{A_2} \oplus \overline{B}_2 \cdot A_1 \bar{B}_1) + (\overline{A_2} \oplus \overline{B}_2 \cdot \overline{A_1} \oplus \overline{B}_1 \cdot A_0 \bar{B}_0)$$

Copyright 2021 - Queens College CSCI 240

2 Bit Comparator

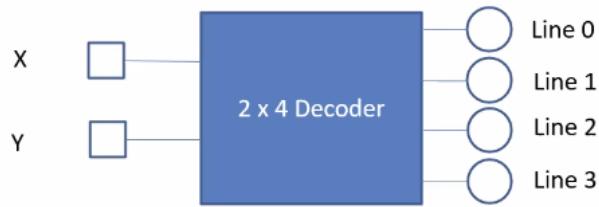


3/23 - Decoders

Definition

"Decodes" binary input into a line selection. Ex: 1 0 = 2, so line 2 would be on. 11 = 3, so line 3 would be on.

"Decodes" binary input into a line selection



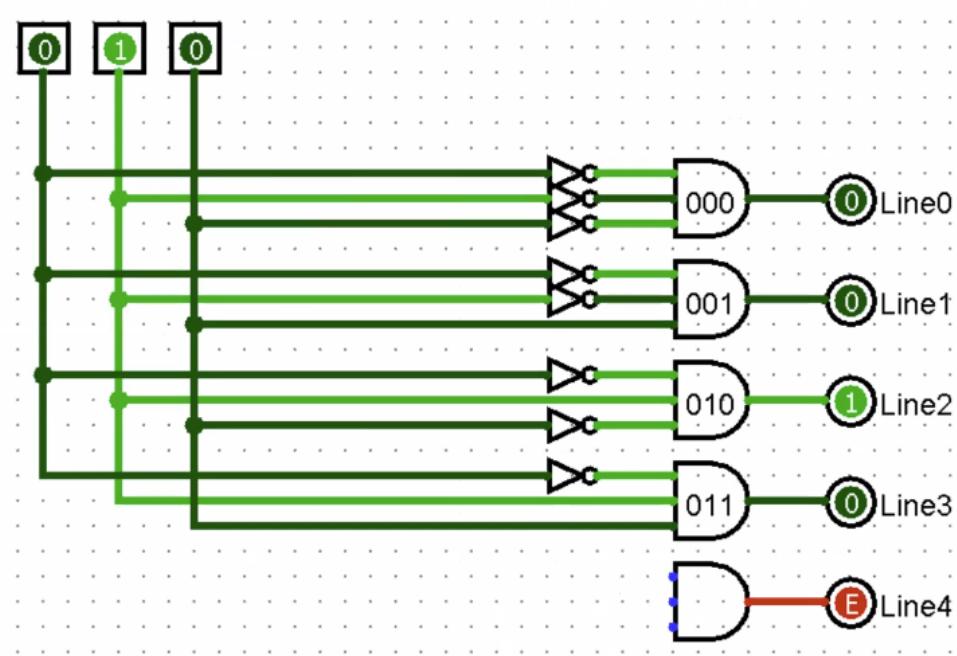
The general format of a decoder is N input lines to 2^N output lines.

i.e.
1 x 2
2 x 4
3 x 8

How to build a decoder

1. Find out how big you want the decoder to be, such as 2x4.
2. Come out with the outputs, line 0, line 1, line 2, line 3, etc
3. Add AND gates linking to each one, 00, 01, 10, 11 etc
4. Left bit is the upper input. If you input a 0, attach a NOT to it. 1 is a straight line.
5. Right most bit is the bottom. Same process 0, NOT. 1, straight wire.
6. Attach the two inputs **X**, **Y**. The most significant bit, **X** is inputted to the top connector of each AND gate. **Y** gets wired to the *bottom*.

Partial 3x8 Decoder

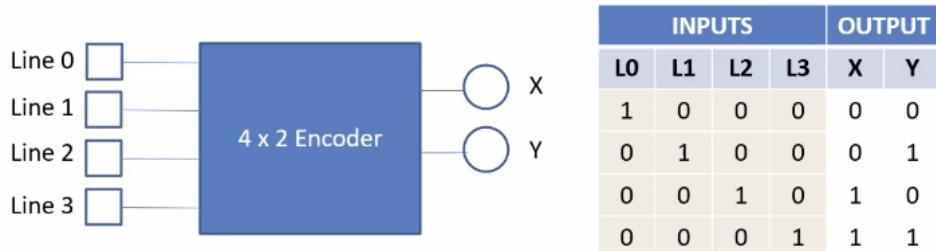


Encoder

Definition Does the opposite of a decoder: takes a line number and "encodes" into a binary number. The format of an encoder is 2^N input lines N to output lines.

Encoders

Does the opposite of Decoder ... takes a line number and
“encodes” it to a binary number

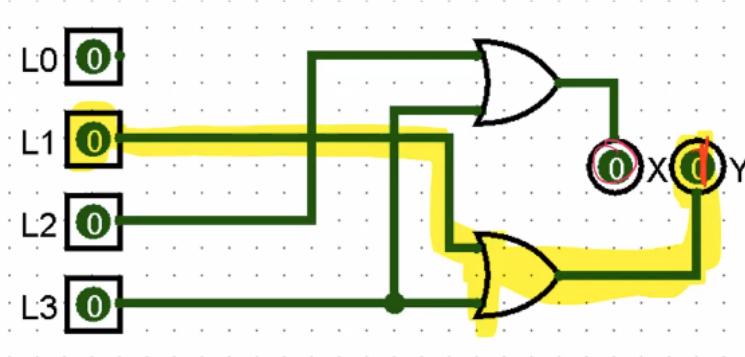


The general format of an encoder is 2^N input lines N to output lines.

i.e. 2×1

$I^{4 \times 2}$
 8×3

4 x 2 Encoder

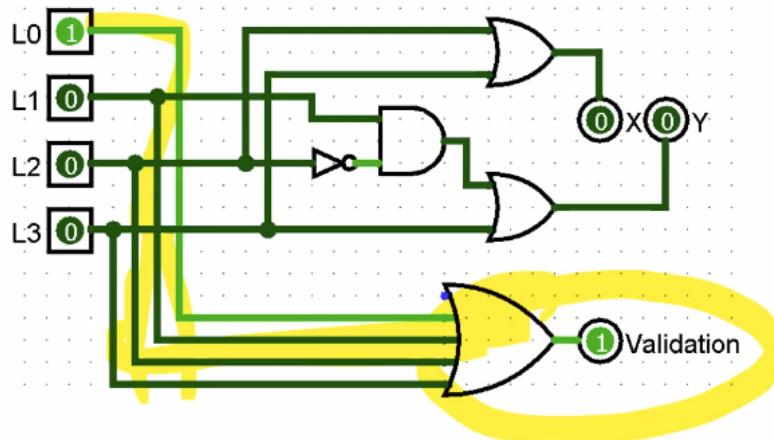


If Line0 = 1, then Output = 00
If Line1 = 1, then Output = 01
If Line2 = 1, then Output = 10
If Line3 = 1, then Output = 11

2 Problems:

What happens if you select L1 and L2? How do i know if L0 was selected vs nothing was selected?
Both are 11. Thus, priority encoder. Prioritizes the highest lineup.

Priority 4 x 2 Encoder with Validation



What happens to Validation Bit if nothing is selected?

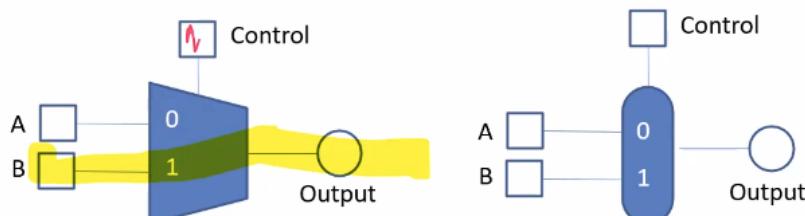
Multiplexer

Allow a line to be shared by several inputs.

Note: A multiplexer is a decoder, but with extra inputs A-D and the outputs moving into an OR gate at the end.

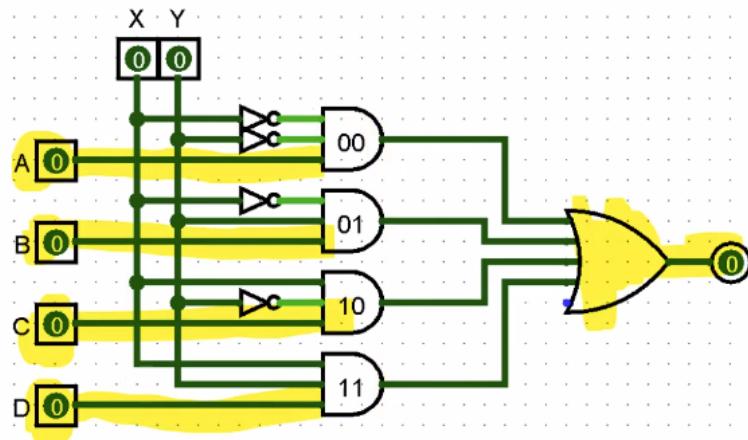
Multiplexers

- Multiplexers allow a line to be shared by several inputs



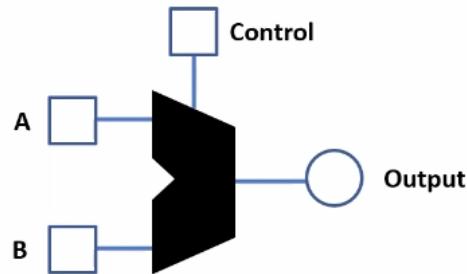
If Control = 0, then Output = A
If Control = 1, then Output = B

4 Line Multiplexer



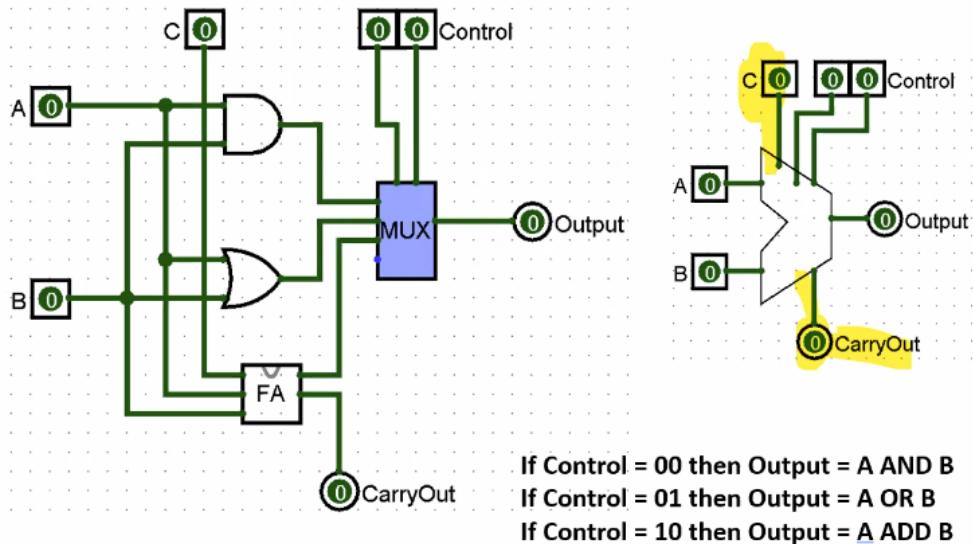
03/28 - Arithmetic Logic Units, ALU

Arithmetic Logic Units - ALUs



If Control = 00 then Output = A AND B
If Control = 01 then Output = A OR B
If Control = 10 then Output = A ADD B
If Control = 11 then Output = A SUBTRACT B

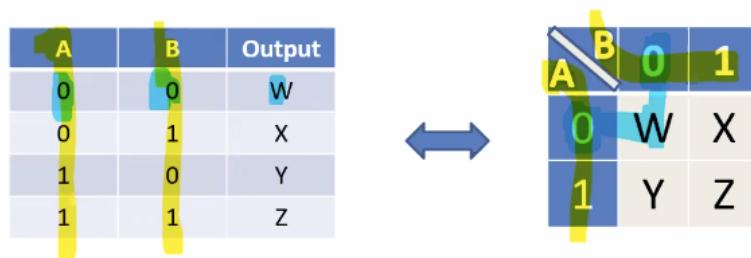
ALU – AND, OR, ADD



Copyright 2020 - Queens College CSCI 240

3/30 K-MAP

For each input, K-maps double in size.



A \ B	0	1
0	1	0
1	0	0

A \ B	0	1
0	1	0
1	0	1

A \ B	0	1
0	1	1
1	1	1

A \ B	0	1
0	1	0
1	1	0

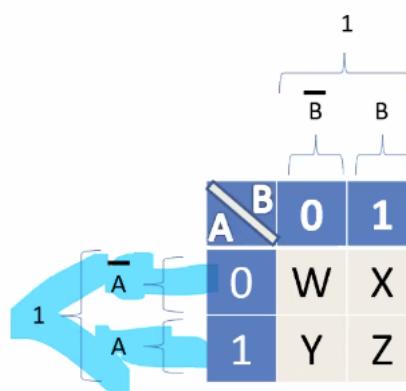
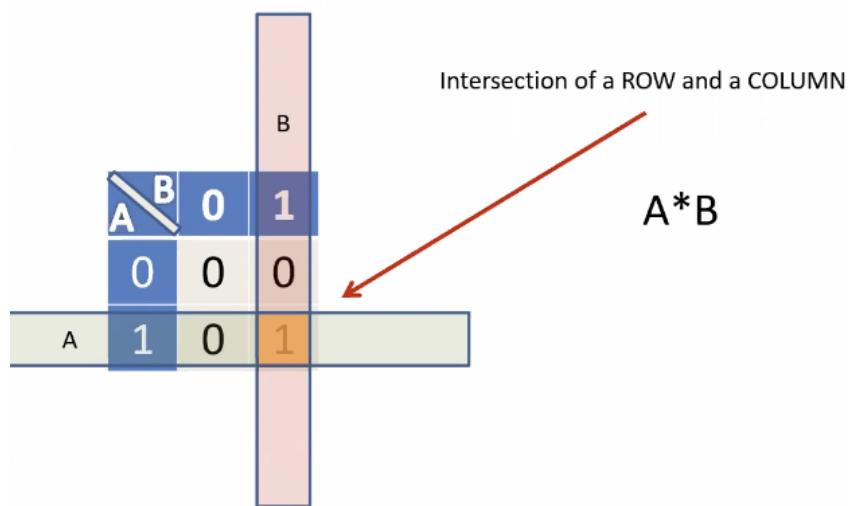
How many groupings of 2^N are there?

A \ B	0	1
0	1	1
1	1	0

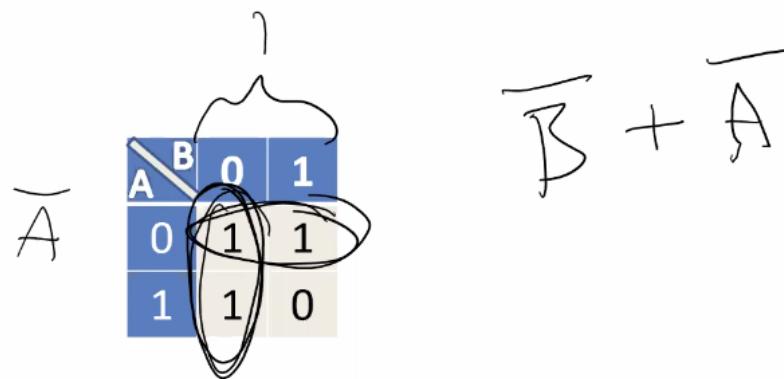
- Look for groups of 2^n
- Diagonals do not count - but you can have groups overlap

Examples of K-Maps

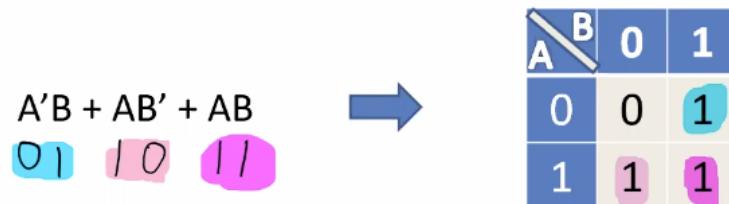
Naming the Groups



Note: if $\neg A$ and A are both intersecting, that means that it doesn't matter if we have A , so we can simplify to 1. $A + \neg A = 1$



Boolean Expression to K-Map



To figure out kmap:

Start with 1 bit. draw a line, and write the reflection. Everything above the line, pad with 0, below, pad with 1.

```

0 0
0 1
-- 
1 1
1 0

```

For 3 bits: Write the numbers out, draw a line, pad with 0s and 1s. Start with **TWO** bits./g

```

0 0 0
0 0 1
0 1 1
0 1 0
-- 
1 1 0
1 1 1
1 0 1
1 0 0

```

	\bar{B}	B	
	$\bar{B}C$	$\bar{B}C$	BC
$A \setminus BC$	00	01	11
	0		
	1		
\bar{A}	\bar{C}	C	\bar{C}

^ How a 3 input k-map would look like.

Simplifying boolean expressions to K-Map

$A'B'C + AB'C$	$\bar{B}C$
0 0 1 1 0 1	
$A \setminus BC$	00 01 11 10
0	0 1 0 0
1	0 1 0 0

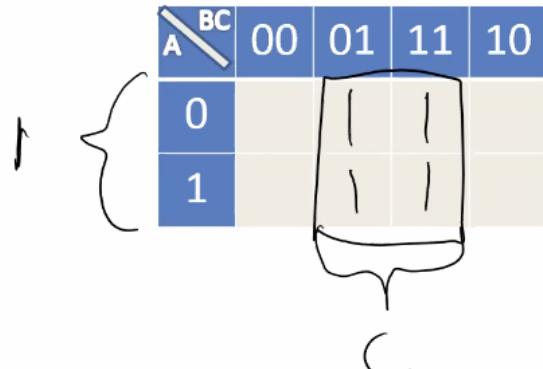
Note - all of this reduces to \bar{A} , since it's the intersection of \bar{A} and 1

$A'B'C' + A'B'C + A'BC + A'BC'$	
0 0 0 0 1 0 1 1 0 1 0	
$A \setminus BC$	00 01 11 10
0	1 1 1 1
1	0 0 0 0

Note - this is simplified to C

$$AB'C + A'B'C + A'BC + ABC$$

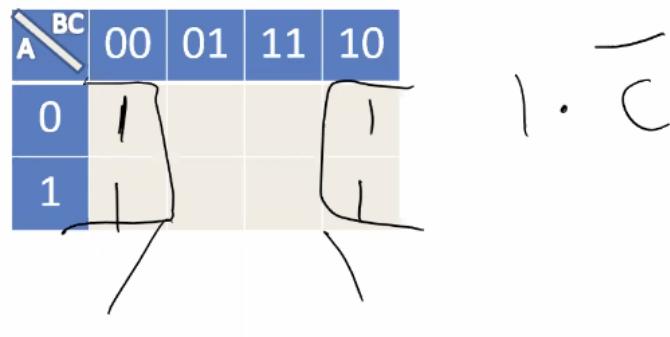
(01 001 011 11)



**BIG NOTE: YOU CAN WRAP AROUND K-MAPS AND IT COUNTS AS A GROUP.
See below.**

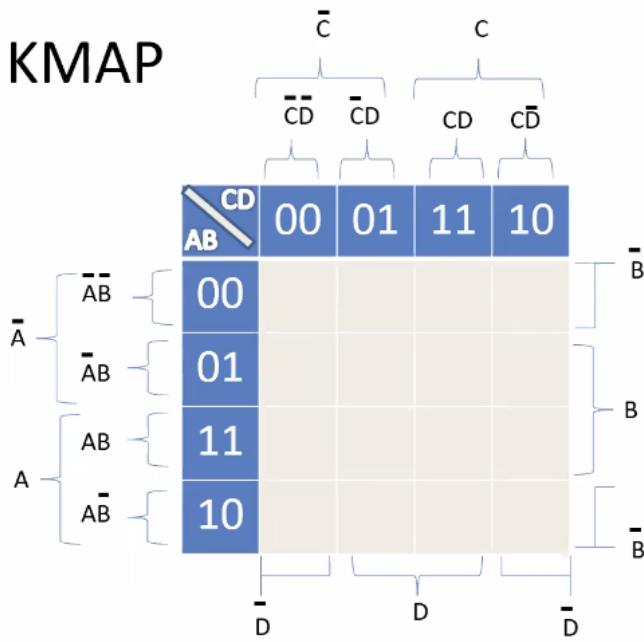
$$A'B'C' + AB'C' + A'BC' + ABC'$$

(000 100 010 110)



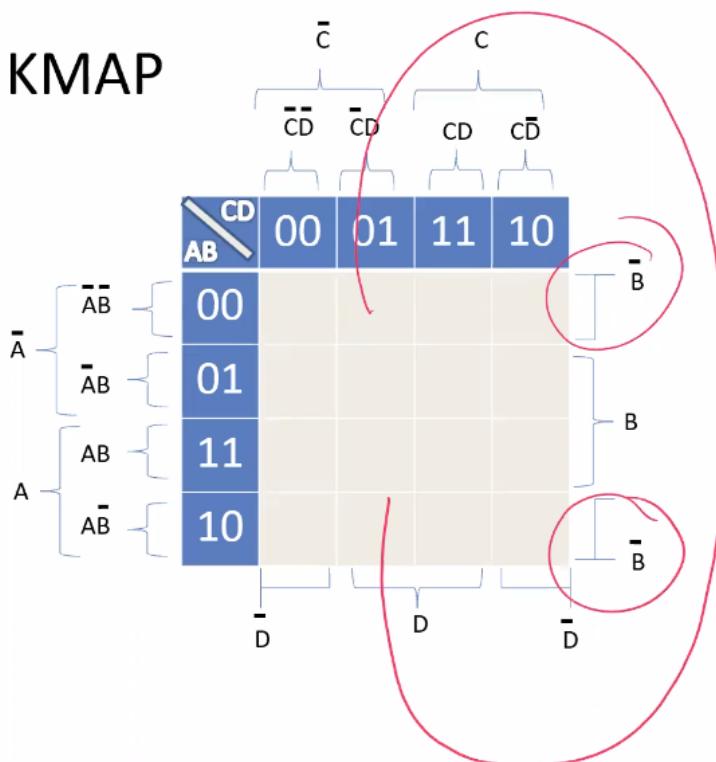
4 Input K-map

4 Input KMAP



Notice the wrap arounds, !D for sides, !B for top/bottom

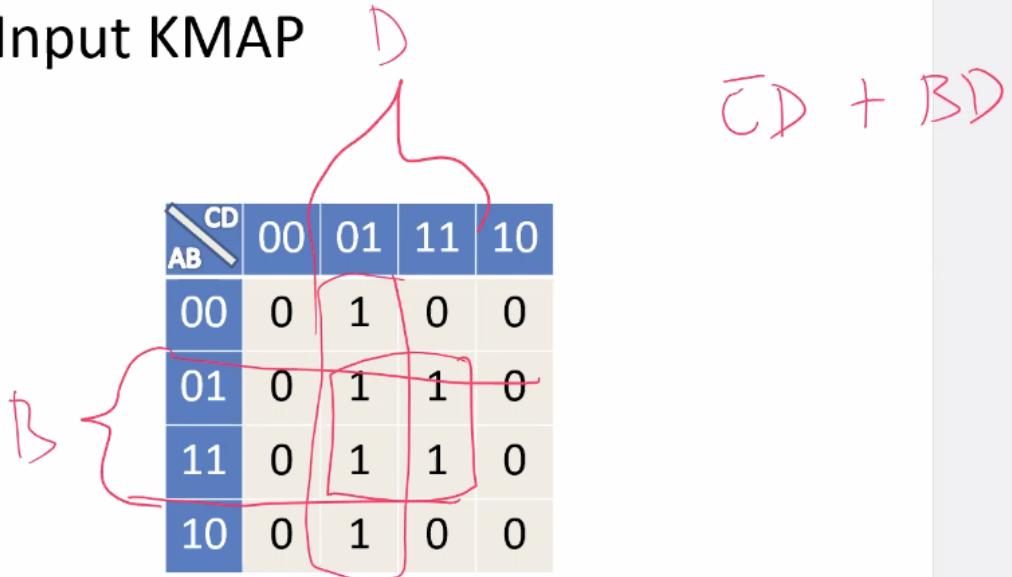
4 Input KMAP



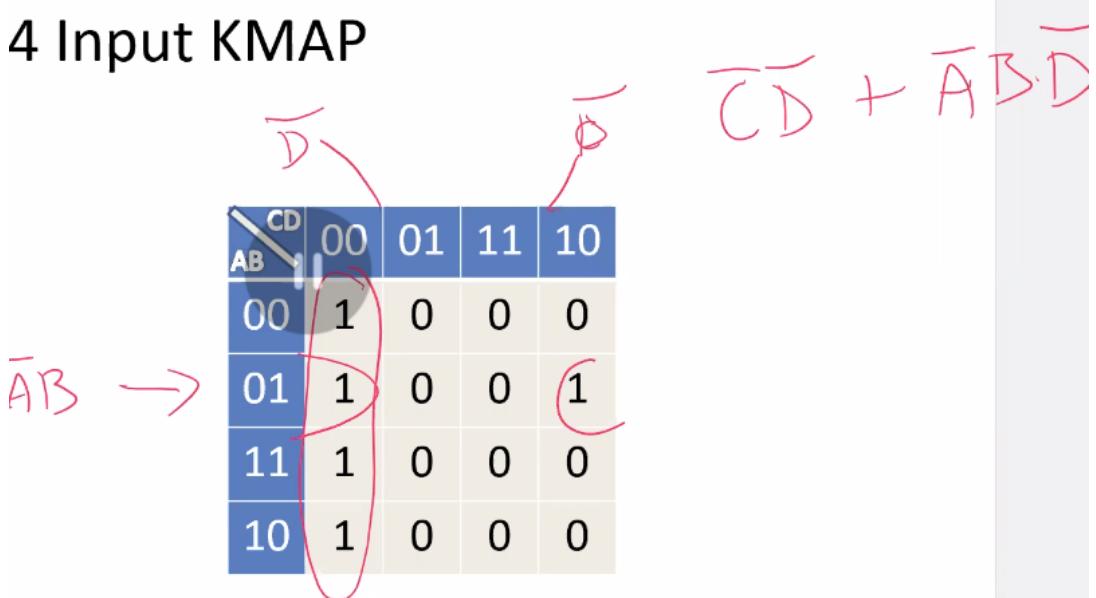
Copyright 2020 - Queens College CSCI 240

4 Input Examples

Input KMAP



4 Input KMAP



MIPS

4/18 - Intro to MIPS

Definition - Microprocessor without Interlocked Pipeline Stages

Primarily used in many embedded systems such as DVRs, routers, printers.

- 32 bit architecture, word size, instructions, mem addresses, registers all 32 bits.

- 32 bit processors take in 32 bits at a time.

```
add $7, $5, $3 #add is the operation, 7 is destination register, 5 and 3 are
source register
#this is $7 = $5 + $3
```

Register is denoted by \$ and is the location in memory where things are stored. \$7 is register 7, the location where the destination is. These are usually:

rs (source), rt(operand), rd (destination).

shamt = shift amount, or sh.

funct: function code (specific R-format instruction)

Decoding MIPS -> Binary

When you have binary, the instructions are 32 bits and are in the following order:

```
op rs rt rd sh func
#instructions are in the order
func $rd, $rs, $rt
```

This code is:

```
add $7, $5, $3
#function, rd, rs, rt
000000 00101 00011 00111 00000 100000
#opcode rs rt rd(destination) sh function
```

Ex 2:

```
and $9, $12, $5
000000 01100 00101 01001 000000 100100
```

Encoding Binary -> MIPS

Ex 1:

```
sub $8, $13, $10
000000 01101 01010 01000 00000 100010
```

R-type instructions:

op	rs	rt	rd	shamt	func
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

op - "opcode" - specifies instruction - for R-Type, opcode is 000000

func – further specification for instruction.

rs, rt, rd – registers used to store operands.

shamt – “shift amount” reserved for shift operations.

Instruction	func
add	100000
and	100100
div	011010
mult	011000
or	100101
sub	100010

R-Type vs I-Type

```
add $7, $5, $3 #R-Type, $7 = $5 + $3 - adds 2 registers together
addi $7, $5, 3 #I-Type: $7 = $5 + 3 adds register and immediate value, constant
```

I-Type Instructions

I-type instructions:

op	rs	rt	imm
6 bit	5 bit	5 bit	16 bit

op - "opcode" - specifies instruction

Instruction	opcode
addi	100000
andi	001100
lw	100011
sw	101011
ori	001101

rs, rt – registers used to store operands.

imm – “immediate value”. Constant built into the Instruction to be used as an operand.

NOTE: IN THIS CASE, RT IS DESTINATION

Examples:

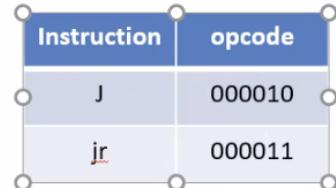
```
andi $6, $3, 15 #op, $rt, $rs, imm
001100 00011 00110 000...01111
addi $10, $13, 5
100000 01101 01010 000...101
```

J-Type Instructions

J-type instructions:

op	target
6 bit	26 bit

op - "opcode" - specifies instruction



target: address to "jump to"

Pseudo instructions

Pseudoinstructions expand the MIPS instruction set but are not part of the MIPS set. They are translated into one or more MIPS instructions when translated to machine code.

```
move $6, $4 = #add $6, $4, $0 -> $0 always contains value 0. Adds $4 to $0,  
then put result into $6  
clear $6 = #add $6, $0, $0 -> adding $0 to $0 and puts $0 into register 6  
mul $6, $5, $4 = #mult $5, $4 -> product is placed into temporary registers due  
to overflow, send lower 32 bits to #mflo $6, higher 32 bits would go to mfhi,  
but we usually don't hit mfhi (>32 bit)
```

Memory

Memory

- Memory is byte addressable (addresses refer to specific bytes)
- 32 bit address to reference 2^{32} addressable locations
- 2^{32} bytes of addressable memory.
- Memory is word-aligned (word addresses must be multiples of 4)
- Words are stored in big-endian format ("big end" first most significant byte is first) as opposed to little-endian, bytes are reversed.

Memory Address				
0	1A	2B	3C	4D
4	33	12	00	00
8	00	00	4F	23
12	23	38	A0	EE
16	65	B3	17	C3

Annotations on the left side of the table show arrows pointing to the first four rows, indicating they are aligned on word boundaries (addresses 0, 4, 8, 12). To the right of the table, handwritten notes explain memory layout:
1 WORD = 32 bits = 4 bytes
 2^{32} bytes = 4 GB
Big Endian (top byte first): 1A 2B 3C 4D
Little Endian (bottom byte first): 00 00 4F 23
Big Endian (top byte first): 23 38 A0 EE
Little Endian (bottom byte first): 65 B3 17 C3

Big-endian format - left to right, Little-ending, right to left

Registers

32 registers in MIPS, preceded by \$, with two formats for addressing:

1. Using register numbers \$0 through \$31
2. Using equivalent names, \$t1, \$sp, etc

Registers

Number	Name	Purpose
\$0	\$zero	0
\$1	\$at	Assembler temporary- reserved by system
\$2-\$3	\$v0-\$v1	Function results
\$4-\$7	\$a0-\$a3	Function arguments
\$8-\$15, \$24, \$25	\$t0-\$t9	Temporary Registers
\$16-\$23, \$30	\$s0-\$s8	Save Registers
\$26-\$27	\$k0-\$k1	Reserved for OS
\$28	\$gp	global pointer - Points to the middle of the 64K block of memory in the static data segment
\$29	\$sp	stack pointer - Points to last location on the stack.
\$31	\$ra	return address

4/20 - MIPS Assembly Programs

File Structure

- ASCII text file, can code in any text editor
- File type ***.ASM**
- Load directly into **PCSPIM**

Data Declarations

- Placed in section: **.data**
- Declares variable names used in program
- Allocate storage in Main Memory
- Assigns initial values into memory

Code

- Placed in section: **.text**
- Contains program code (instructions)
- Sections of code may contain labels for “jumping”

Comments

- # used for comments

Program Template

```
.data      # declare variables in this section
X: .word 5
Y: .word 7
Z: .word 9

.text      # program code in this section
main:     # indicates start of program
    xxxx
    xxxx
Loop1: xxxx
    xxxx
Loop2: xxxx
    xxxx
Print: xxxx
    xxxx

# End of program, leave a blank line afterwards
```

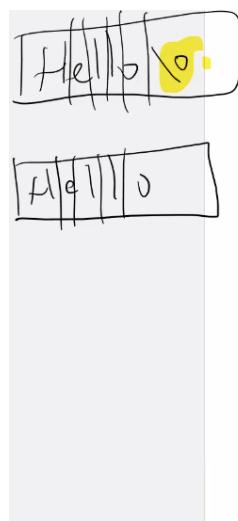
.data section

- X, Y, Z are pointers. Data section initializes values inside memory, **pointing into a block of memory containing 5, 7, 9.**

.text section

- Each line of code is stored in a block of memory, so labels such as main are also **pointers to a line of code.**
- Each program requires a **main** at the beginning of the program.
- **Loops** are used to label certain sections of code to use gotos, loops, if statements, everything that will take the executed line somewhere else in the code.
- **End of program requires a blank line.**

Data Declarations



Data Declarations

- Create storage in main memory for variable of specified type with given label name and initial value(s).
- Each label corresponds to a unique address in memory, which the assembler determines.

Format for declaration:

label: storage_type value(s)

STORAGE TYPES	
.word	Stores a 32 bit word in memory
.ascii	Stores a string in memory <u>without a null terminator</u>
.asciz	Stores a string in memory <u>with a null terminator</u>
.space	Reserves a block of space in memory

,wurj

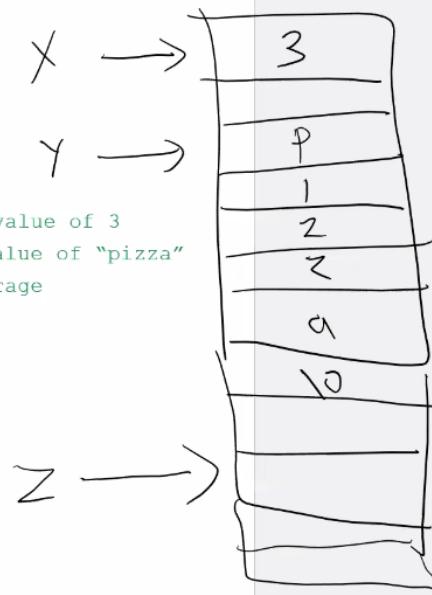
32 bit

Data Declarations

Examples:

```
X: .word 3          # creates a integer with value of 3  
Y: .asciiz "pizza" # creates a string with value of "pizza"  
Z: .space 40        # creates 40 bytes of storage
```

NOTE: labels always followed by colon



Loading in MIPS

Load Address

Looks at a label and returns the address it is pointing to. **Taking pointer's memory address.**

Load Address

Load Address – loads memory address into a register

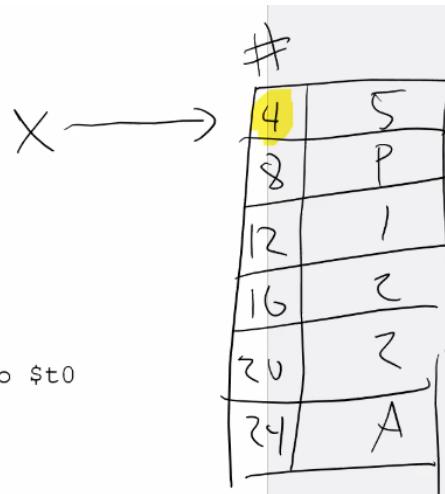
syntax:

la dest_register, variable label

example:

la \$t0, X # loads address of X into \$t0

$$A \quad \$t0 = 4$$



Load Word

Loads the address, then passes the address into the load word. Gives the word/value of the address.

Load Word

Load Word – loads value from memory into a register.

syntax:

`lw dest_register, 0(source_register)`

`source_register` contains memory address of value in main memory.

example:

`lw $s0, 0($t0)`

loads value (stored at memory address $\$t0$) into $\$s0$

$$\# \$s0 = 5$$

#	4	5
8	P	
12	I	
16	C	
20	Z	
24	A	

Load Immediate

Load Immediate

Load Immediate – loads constant value into a register.

syntax:

`li dest_register, value`

example:

`li $s0, 1` #loads value 1 into $\$s0$

$$\# \$s0 = 1$$

Sample Program using Loads

```
.data
X: .word 5
Y: .word 7

.text
main:

#load X into $s0
la $t0, X #loads memory address of X into $t0
lw $s0, 0($t0) #loads value stored at X into $s0

#load Y into $s1
la $t1, Y #loads memory address of Y into $t1
lw $s1, 0($t1) #loads value stored at Y into $s1
```

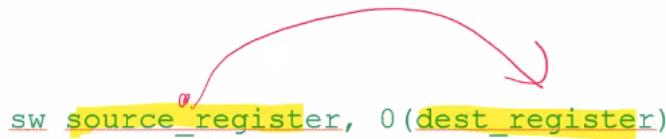
```
#note: You can re-use $t0, it is a temp register
```

Store Word

Store Word

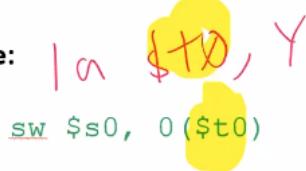
Store Word – Stores (aka “save”) value from register into memory.

syntax:

 sw source_register, 0(dest_register)

dest_register contains memory address storage location in main memory.

example:

 sw \$s0, 0(\$t0)

stores value from \$s0 into memory address at \$t0

NOTE: must use Load Address first to define memory address to save to

Exercise with Store Word

```
.data
A: .word 9
B: .word 7
C: .space 4

.text
main:

#load A into $s5
la $t0, A #loads memory add of A to $t0
lw $s5, 0($t0) #loads value stored at A to $s5
#load B into $s6
la $t1, B
lw $s6, 0($t1)
#add A and B and place sum into $s7
add $s7, $s5, $s6 #$s7 = $s5 + $s6
#save $s7 into memory at C
la $t2, C #loads memory address of C to $t2
sw $s7, 0($t2) #saves value from $s7 to memory address
```

04/25 - System Calls

System calls are used to communicate with the system for reading from keyboard or writing to the screen.

System calls require some parameter to be passed in an argument register (i.e. \$a0) and service code to be passed via \$v0

SYSCODES

System Call Codes

Service	Code	Argument(s)	Result
Print Integer	1	\$a0 = integer value	(none)
Print Float	2	\$f12 = float value	(none)
Print Double	3	\$f12 = float value	(none)
Print String	4	\$a0 = address of string	(none)
Read Integer	5	(none)	\$v0 = value read
Read Float	6	(none)	\$f0 = value read
Read Double	7	(none)	\$f0 = value read
Read String	8	\$a0 = address where string to be stored. \$a1 = number of characters to read + 1.	(none)
Memory Allocation	9	\$a0 = number of bytes of storage desired	\$v0 = address of memory block
Exit (end of program)	10	(none)	(none)
Print Character	11	\$a0 = integer value	
Read Character	12	(none)	\$v0 = character read

Copyright 2020 - Queens College CSCI 240

The order to do a system call is:

```
#print integer 15, note li stands for load immediate
li $v0, 1 #loads code 1, which is print integer into $v0. $v0 contains call
code
li $a0, 15 #loads immediate 15 into $a0
syscall #prints 15

#print character 'x' (ASCII 120)
li $v0, 11 #load 11 (code for print char) into $v0, call code
li $a0, 120 #load immediate 120 into $a0, 120 is ASCII for x
syscall #prints 'x' - sees $v0 is 11, so we want to print char. sees $a0, 120,
prints ascii 120
```

Print String

```

.data
String1: .asciiz "Fried Chicken" #string1 is a pointer to a block of memory

.text
main:
    li $v0, 4 #load 4 (Code for print STRING) into $v0
    la $a0, String1 #load address of string1 into $a0
    syscall #prints "Fried Chicken"

```

Read Integer

```

li $v0, 5 #load 5 (read int) to $v0
syscall #starts capturing keyboard values into $v0 until <CR> encountered.
add $s0, $v0, $0 #copies value from $v0 into $s0

```

Exit Program

```

li $v0, 10 #load 10 (code for Exit) into $v0
syscall #exits program

```

Hello World!

```

.data
str = .asciiz "Hello World" #hello world string

.text
main:
    li $v0, 4 #load 4 (print string sys call)
    la $a0, str #loads address for str
    syscall #prints "Hello World"
    li $v0, 10 #load exit code 10
    syscall #exit program

```

Simple Addition Calculator

```

.data
Str1: .asciiz "Enter first number"
Str2: .asciiz "enter second number"
Str3: .asciiz "The sum is:"

.text
main:
    #print string "enter first number"
    li $v0, 4
    la $a0, Str1
    syscall

    #read integer into $s0
    li $v0, 5 #read int
    syscall
    add $s0, $v0, $0

```

```

#print "enter second number"
li $v0, 4 #print string
la $a0, Str2
syscall
#read integer into $s1
li $v0, 5 #read int
syscall
add $s1, $v0, $0

#add $s0 and $s1 into $s2
add $s2, $s0, $s1

#print string "The sum is"
li $v0, 4
la $a0, Str3
syscall

#print integer stored in $s2raar
li $v0, 1 #syscall 1 is print int
move $a0, $s2 #copies contents of $s2 into $a0
syscall

```

04/27 - Conditional Branching

- By default, instructions execute sequentially
- Conditional Branching alters the instruction sequence if certain conditions are met.

Branch on Equal, BEQ - branches to a specific instruction at a diff memory location when two register contains equal values

Branch Not Equal, BNE compares if != to each other.

```

#syntax
beq reg1, reg2, label #branch equal
bne reg1, reg2, label #branch not equal
#if $t0==$s0 go to else
beq $t0, $s0, else
#if $t0 != $s0 go to else
bne $t0, $s0, else

```

Set on Less Than (SLT) - Compares two contents in registers and tests if less than. **IF TRUE - RETURN 1, 0 OTHERWISE**

Set on Less than Immediate same as SLT, but using an immediate rather than a var.

```

slt dest_reg, reg1, reg2
#dest_reg = 1 if reg1 < reg2
#dest_reg = 0 if reg1 >= reg2
# <- slt immediate ->
slti dest_Reg, reg1, imm

```

BRANCHING INSTRUCTIONS

Branching Pseudo-instructions

Pseudo-Instruction	Syntax	Expansion
Branch if Greater Than	bgt reg1, reg2, label	slt \$at, reg2, reg1 bne \$at, \$0, label
Branch if Less Than	blt reg1, reg2, label	slt \$at, reg1, reg2 bne \$at, \$0, label
Branch if Greater Than or Equal	bge reg1, reg2, label	slt \$at, reg1, reg2 beq \$at, \$0, label
Branch if Less Than or Equal	ble reg1, reg2, label	slt \$at, reg2, reg1 beq \$at, \$0, label
Branch if zero	beqz reg1, label	beq reg1, \$0, label
Branch if Equal to Immediate	beq reg1, Imm, label	ori \$at, \$0, Imm beq \$reg1, \$at, label
Branch if Not Equal to Immediate	Bne reg1, Imm, label	ori \$at, \$0, Imm beq \$reg1, \$at, label

Unconditional Branching

Similar to "goto" in C++

Jump -> **goto**

Jump and Link - copies the address of the next instruction into register \$ra and then jumps to the address stored in LABEL

Jump Register - Jumps to the address stored in a register

```

j LABEL #syntax
j loop #jumps to loop

#<-Jump and Link ->
#ex: pretend address of print is 3
jal print
#then $ra = 4, then jumps to print
jr $ra #jumps to $ra, which would be register 4

```

Exercise 1

```
.data
A: .word 8
B: .word 3
C: .space 4

.text
main:
#load A into $s1
la $t0, A
lw $s1, 0($t0)
#load B into $s2
la $t0, B
lw $s2, 0($t0)
#if A<B goto WRITEB
blt $s1, $s2, WRITEB
#write A to memory at C
la $t0, C #$t0 = address of C
sw $s1, 0($t0) #write A to memory address at C
#goto END
j END
WRITEB:
#write B to memory at C
la $t0, C #$t0 address of C
sw $s2, 0($t0) #write B to memory at C
#goto END:

END:
```

Calculator

```
.data
Str1: .asciiz "First #"
Str2: .asciiz "2nd #"
Str3: .asciiz "Enter operation + - * /"

.text
main:
#print String "Enter first #"
li $v0, 4 #load 4 (Code for print STRING) into $v0
la $a0, Str1 #load address of string1 into $a0
syscall #prints "first number"

#read int into $s0
li $v0, 5 #read int
syscall
add $s0, $v0, $0

#print String into Enter 2nd
li $v0, 4
la $a0 Str2
syscall
```

```

#read int into $s1
li $v0, 5
syscall
add $s1, $v0, $0

#print string enter operaiton
li $v0, 4
la $a0, Str3
syscall

#read char into $s2 +(43), -(45), *(42), /(47)
li $v0, 12 #read char
syscall
move $s2, $v0

#if + goto ADD
beq $s2, 43, ADD #branch eq to immediate, $s2 -> 43 ascii
#if - goto SUB
beq $s2, 45, SUB
#if * goto MULT
beq $s2, 42, MULT
#if / goto DIV
beq $s2, 47, DIV

ADD:
#$s0 add $s1 into $s2
add $s2, $s0, $s1
#Print int stored in $s2
li $v0, 1 #1 is syscode for print INT
move $a0, $s2
syscall
j END

SUB:
#$s0 sub $s1 into $s2
sub $s2, $s0, $s1
li $v0, 1 #1 is syscode for print INT
move $a0, $s2
syscall
j END

MULT:
mul $s2, $s0, $s1
li $v0, 1 #1 is syscode for print INT
move $a0, $s2
syscall
j END

DIV:
div $s2, $s0, $s1
li $v0, 1 #1 is syscode for print INT
move $a0, $s2
syscall
j END

END:
li $v0, 10 #load exit code 10
syscall #exit program

```

05/02 - Control Structures

A **control structure** determines the order in which statements are executed.

- Sequential: Default
- Repetition: Loops
- Selection: Choose between 2 or more paths.

If.. Then

IF ... THEN

C++:

```
if ( X < 5 )
{
    Z = X;
}
```

MIPS:

```
slti $t0, $s0, 5      # $t0 = (X < 5)?
beq $t0, $0, ENDIF
add $s2, $s0, $0

ENDIF:
```

ASSUME:

```
X = $s0
Y = $s1
Z = $s2
```

Copyright 2020 - Queens College CSCI 240

IF ... THEN

C++:

```
if ( X < Y )
{
    Z = X;
}
```

MIPS:

```
slt $t0, $s0, $s1      # $t0 = ($s0 < $s1)
beq $t0, $0, ENDIF
add $s2, $s0, $0

ENDIF:
```

ASSUME:

```
X = $s0
Y = $s1
Z = $s2
```

Copyright 2020 - Queens College CSCI 240

If.. Else

IF ... ELSE

C++:

```
if ( x < y )
{
    Z = X;
}
else
{
    Z = Y;
}
```

MIPS:

```
slt $t0, $s0, $s1      # $t0 = (x < y) ?
beq $t0, $0, ELSE
add $s2, $s0, $0
j ENDIF
```

ELSE: add \$s2, \$s1, \$0

ENDIF:

ASSUME:

```
X = $s0
Y = $s1
Z = $s2
```

Copyright 2020 - Queens College CSCI 240

The diagram illustrates the translation of an if-else statement from C++ to MIPS assembly. Handwritten annotations in red and blue highlight specific parts of the code. In the C++ section, a red oval encloses the conditional block, and a red arrow points to the 'F' label below it. In the MIPS section, a red oval encloses the 'beq' instruction, and a red arrow points to the 'F' label below it. A blue oval encloses the 'add' instruction under the 'ELSE:' label, and a blue arrow points to the 'ENDIF:' label below it. A purple box labeled 'ASSUME' contains the initial values: X = \$s0, Y = \$s1, Z = \$s2.

While Loops

1. Test condition at top
2. if condition fails
 - 1. Branch to the first instruction after the loop body.
3. If condition passes:
 - 1. continue to first instruction inside loop body
 - 2. jump to top of loop

WHILE LOOPS

C++:

```
total = 0;
i = 0
while (i < 5)
{
    total += i;
    i++;
}
```

MIPS:

```
li $s1, 0
li $s0, 0
LOOP: slti $t0, $s0, 5      # $t0 = (i < 5) ?
beq $t0, $0, ENDLOOP
add $s1, $s1, $s0
addi $s0, $s0, 1
j LOOP
```

ENDLOOP:

ASSUME:

```
$s0 = i
$s1 = total
```

Copyright 2020 - Queens College CSCI 240

The diagram illustrates the translation of a while loop from C++ to MIPS assembly. Handwritten annotations in red and blue highlight specific parts of the code. In the C++ section, a red oval encloses the loop body, and a red arrow points to the 'F' label below it. In the MIPS section, a red oval encloses the 'beq' instruction under the 'LOOP:' label, and a red arrow points to the 'F' label below it. A blue oval encloses the 'add' instruction under the 'LOOP:' label, and a blue arrow points to the 'j LOOP' label below it. A purple box labeled 'ASSUME' contains the initial values: \$s0 = i, \$s1 = total.

Do While Loop

1. Execute Loop Body Once
2. At end of loop body, test condition
3. If condition passes
 1. jump to first instruction inside loop body
4. If condition fails
 1. continue to next instruction outside loop body

DO WHILE LOOPS

C++:

```
total = 0;
i = 0;

do{
    total += i;
    i++;
} while (i < 5)
```

MIPS:

```
li $s1, 0
li $s0, 0

TOP: add $s1, $s1, $s0
      addi $s0, $s0, 1
      slti $t0, $s0, 5
      bne $t0, $0, TOP
```

b1b = (i < 5)?

ASSUME:
\$s0 = i
\$s1 = total

Copyright 2020 - Queens College CSCI 240

Short Circuit Evaluation &&

Short Circuit Evaluation &&

C++:

```
if ( X >= 50 && X < 100 )
{ F Y = X; T } F
```

MIPS:

```
# X >= 50
slti $t0, $s0, 50
bne $t0, $0, ENDIF # B1D = (X < 50)

# X < 100
slti $t0, $s0, 100
beq $t0, $0, ENDIF

# Y = X
add $s1, $s0, $0
```

X >= 50 ↗
X < 50 ↘

ASSUME:
\$s0 = X
\$s1 = Y

ENDIF:

Copyright 2020 - Queens College CSCI 240

Think of it as double negatives, if $X < 50 == \text{true}$, that means it's not ≥ 50 , so endif

Short Circuit Evaluation ||

Short Circuit Evaluation ||

C++:

```
if ( X < 50 || X >= 100 )
{
    Y = X;
}
```

MIPS:

```
# X < 50
slti $t0, $s0, 50
bne $t0, $0, IF
# X >= 100
slti $t0, $s0, 100
bne $t0, $0, ENDIF
# Y = X
add $s1, $s0, $0
```

ASSUME:

```
$s0 = X
$s1 = Y
```

Copyright 2020 - Queens College CSCI 240

Exercise - Guess a Number

```
,data
Str1: .ascii "Guess #"
Str2: .asciiz "Congrats"
Str3: .asciiz "Wrong"
SECRET: .word 42

.text
main:
    #load Secret into $s0
    la $t0, SECRET
    lw $s0, 0($t0)

    Guess: #print string "guess"
    li $v0, 4
    la $a0, Str1
    syscall

    #read int input into $s1
    li $v0, 5
    syscall
    move $s1, $v0

    #compare input to secret, if eq, goto win
    beq $s0, $s1, WIN

    #print wrong!
    li $v0, 4
    la $a0, Str3
    syscall
```

```

#goto guess
j GUESS

WIN: #print congrats
li $v0, 4
la $a0, Str2
syscall
#End program
li $v0, 10
syscall

```

5/4 - Data Structures

Addressing Modes

An addressing mode is a method of addressing (locating) operands for an instruction

MIPS supports:

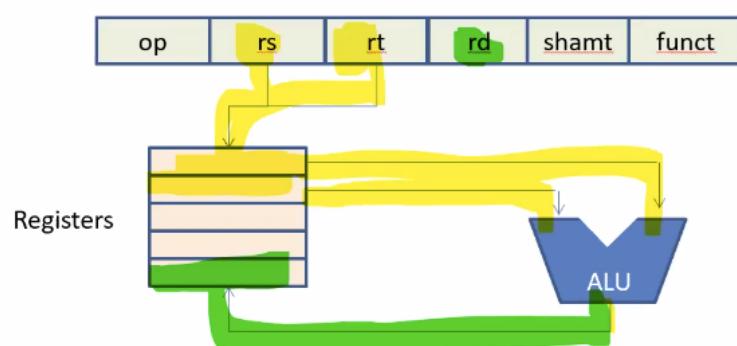
- Register Addressing
- Immediate Addressing
- Base Displacement Addressing

Register Addressing

Register Addressing

- Operands are stored in a register
- Must retrieve from register before using
- Example: add \$t0, \$s0, \$s1

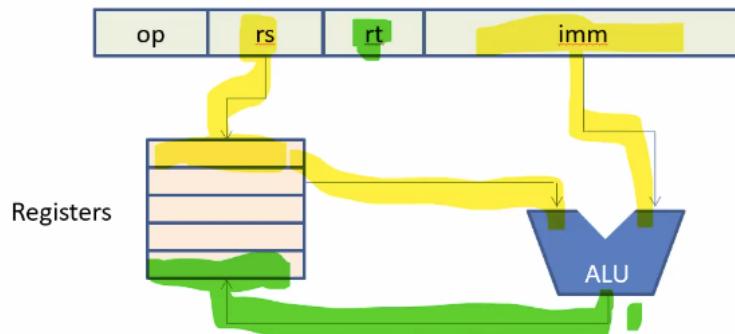
$$(\text{d}) \quad (\text{s}) \quad (\text{t})$$



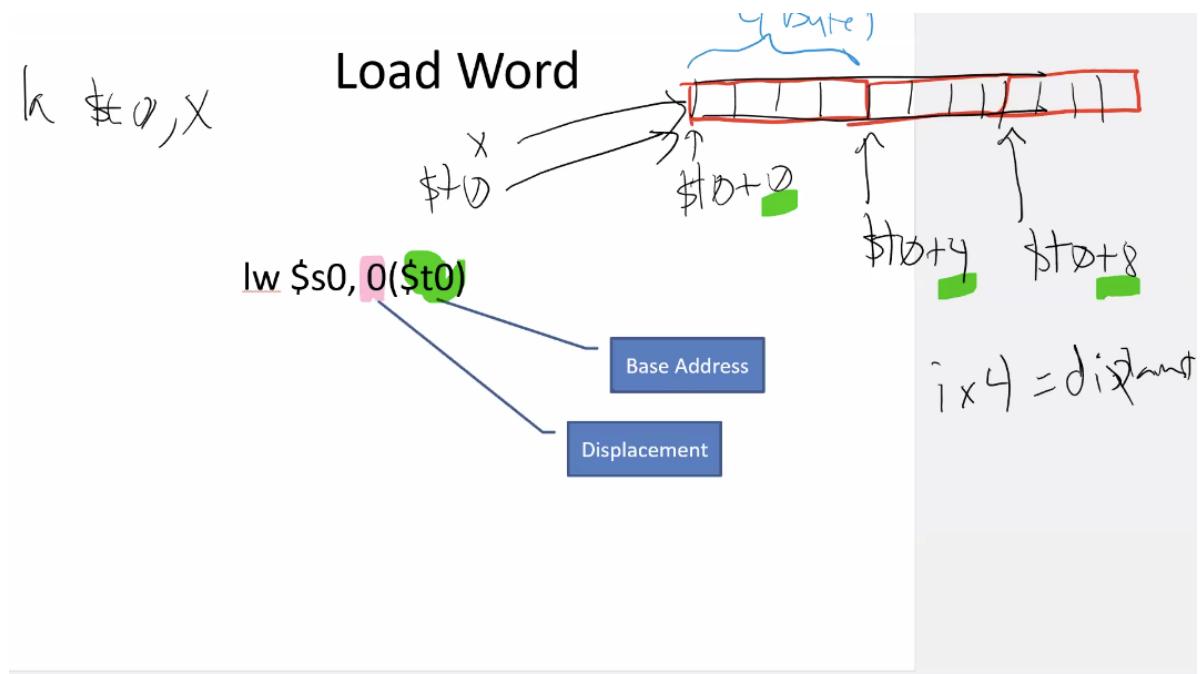
Immediate Addressing

- Operand is an immediate value
- Value is part of the instruction
- Example: addi \$t0, \$s0, 5

(+) rs imm

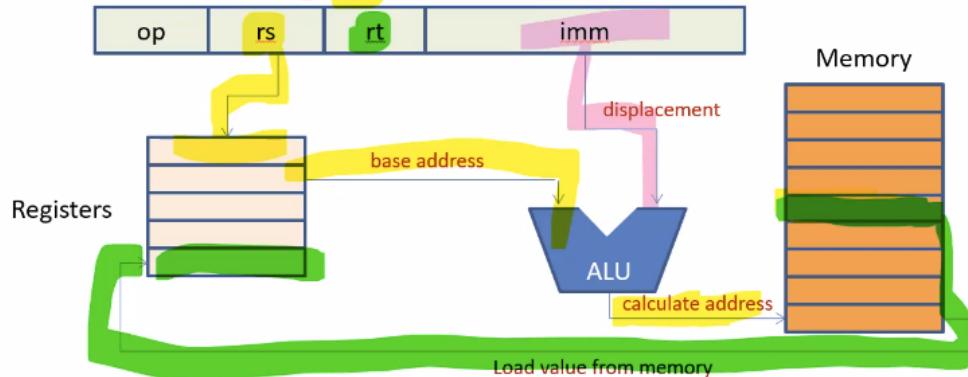


Base Displacement Addressing



Base Displacement Addressing

- Operand is a memory location
- Memory address is a base + displacement (must calculate address)
- Must load value from memory
- Example: `lw $s0, 0($t0)`



Loading values from memory

`la $t0, A` $\#\$t0 = \text{address of } A$

`lw $s0, 12($t0)`

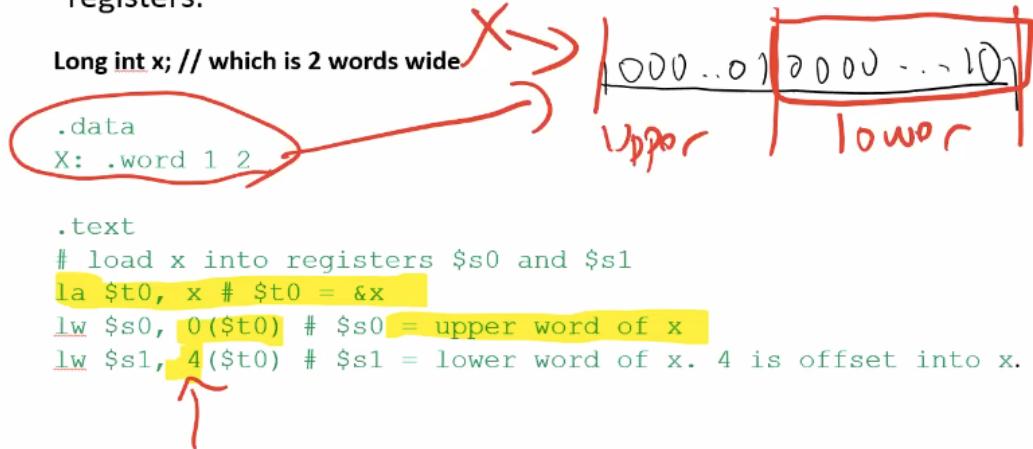
A	Memory Address				
	0	1A	2B	3C	4D
	4	33	12	00	00
	8	00	00	4F	23
	12	23	38	A0	EE
	16	65	B3	17	C3

Exercise: How to retrieve value stored at memory address 12 in terms of A?

Long data type - 64bit value

Long data type - 64 bit values

- For variables larger than 1 word (32 bits), label refers to first word of variable
- Need several `lw` statements to load variable into several registers.



Arrays

Arrays

C++:

```
int a[5] = {2, 1, 5, 4, 3};
```

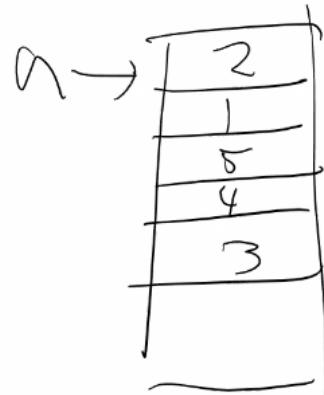
MIPS:

```
.data  
a: .word 2 1 5 4 3
```

```
.text  
la $t0, a # $t0 = &a  
lw $s0, 0($t0) # a[0]  
lw $s1, 4($t0) # a[1]  
lw $s2, 8($t0) # a[2]
```

$i \times 4$

$a[i]$



Arrays

Imm

What about $A[i]$?

lw \$s0, i (\$t0)

MIPS:

```
# $s0 = &a  
# $s1 = i  
# $s2 = a[i]
```

```
la $s0, a      # base address  
sll $t0, $s1, 2 # $t0 = i * 4  
add $t1, $s0, $t0 # $t1 = address of a[i]  
lw $s2, 0($t1) # load a[i]
```

$$t_0 = s_1 \times 2^2$$

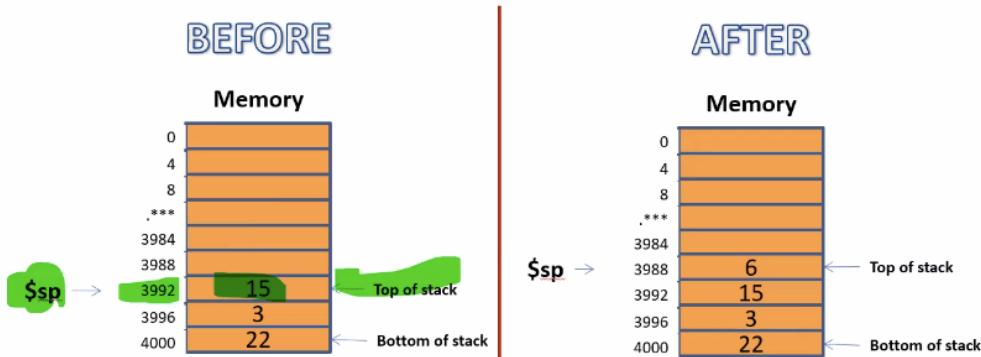


Stack Memory

Pushing onto Stack

EXAMPLE: Push 6 onto stack

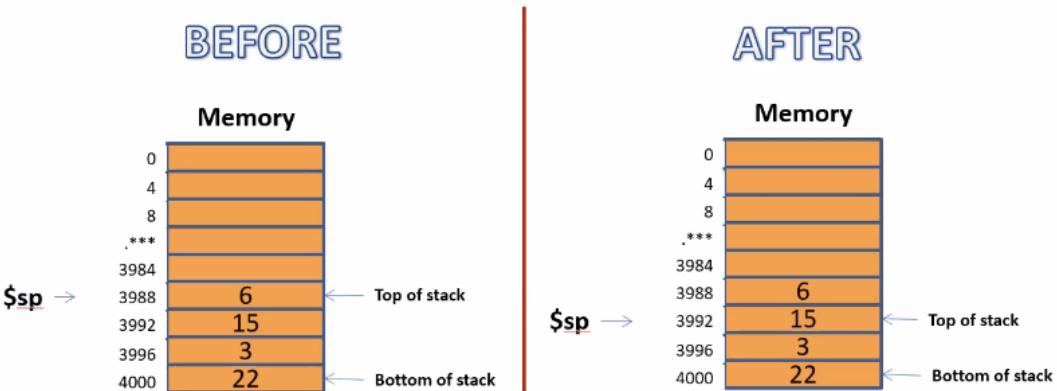
```
li $t0, 6      # $t0 = 6  
addi $sp, $sp, -4 # $sp -= 4  
sw $t0, 0($sp) # push $t0 onto stack.
```



Pop from Stack

EXAMPLE: Pop 6 from stack

```
lw $t0, 0($sp)      # pop 6 from stack into $t0  
addi $sp, $sp, 4     # move $sp to new top of stack
```



Exercise - SumArray

```
.data  
A: .word 5 4 3 2 1 #array 5,4,3,2,1  
Str: .asciiz "The sum is:"  
  
.text  
main:  
#set $s0 to beginning of A  
la $s0, A  
#(index = $s1) initialize $s1=0  
li $s1, 0  
#(Sum = $s3) initialize $s3=0  
li $s3, 0  
  
LOOP:  
#while index < 5  
slti $t0, $s1, 5 #$t0 = (index < 5)?  
beq $t0, $0, ENDLOOP #if condit not met, endloop  
#load A[index] into $s2  
sll $t0 $s1, 2 #$t0 = i*4  
add $t1, $s0, $t0 #$t1 = address of a[i]  
lw $s2, 0($t1) #load a[i]  
#add $s2 to Sum  
add $s3, $s3, $s2 #s3 = s3+s2  
#increment index  
add $s1, $s1, 1 #s1 = s1 + 1  
#goto top of loop  
j LOOP  
  
ENDLOOP:
```

```
#print str "sum is"
li $v0, 4
la $a0, Str
syscall
#print int stored in $s3
li $v0, 1
move $a0, $s3
syscall
```

5/9 Functions

jal - jump and link jumps to memory address indicated in the instruction label and saves the address of the next instruction into \$ra (\$31)

jr - jump register - jumps to the address in the register used by this instruction.

Registers

Registers



\$a0-\$a3

- Used to pass arguments (aka parameters) to the function

\$v0-\$v1

- Used to pass return values from the function.

\$t0-\$t9

- Used to store temporary values. Function Caller cannot assume the values are unchanged after a function call.

\$s0-\$s7

- Used to store “save” values. Function must guarantee that the values are the same upon return from function call.

Function Flow Control

Function Flow Control

```
1 *****
2 add $a0, $t0, $0
3 jal PRINT
4 *****
5 *****
6 add $a0, $t0, $0
7 jal PRINT
8 *****
9 *****
10 *****
11 *****
```

41 PRINT: li \$v0, 1
42 syscall
43 jr \$ra

\$ra

- What is the content of \$ra at line 4?
- What is the content of \$ra at line 5?
- What is the content of \$ra at line 9?

Copyright 2020 - Queens College CSCI 240

PrintArray

```
.data
A: .word 5 4 3 2 1

.text
main:
#set $s0 to beginning of Array A
la $s0, A
#index = $s1) initialize $s1=0
li $s1, 0
#(Sum = $s3) Initialize $s3=0
li $s3, 0

LOOP:
#while index < 5
slt $t0, $s1, 5 #t0 = (index < 5)?
beq $t0 $0, ENDLOOP #if index =5, break

#load A[index] into $a0
sll $t0, $s1, 2 #to = 1*4
add $t1, $s0, $t0 #t1 = address a[i]
lw $a0, 0($t1) #load a[i]

#call PRINT function
jal PRINT
#increment index
addi $s1, $s1, 1 #s1=s1+1
#goto top of loop
j LOOP

ENDLOOP:
#end program
li $v0, 10
```

```

syscall
PRINT:
#print integer stored in $a0
li $v0, 1
syscall #note, a0 has already been filled in before print
#return
jr $ra

```

5/11 Palindrome, High Low

Palindrome

Exercise

Palindrome.asm

A program that prints contents of an integer array forward and backward.

- 1) Set initial index to zero
- 2) Select value at A[index]
- 3) Call print function to print char value
- 4) Push A[index] onto stack
- 5) Increment index
- 6) Repeat steps 3-5 for all items in array
- 7) Print all char values in stack

```

.data
A: .word 76 73 86 69
.text
main:
# Set $s0 to beginning of Array A
La $s0, A    # $s0 = address of A
#index = $s1) Initialize $s1=0
Li $s1, 0    # $s1 = 0

LOOP1: #while index < 4
Slti $t0, $s1, 4          # $t0 = (index < 4)?
beq $t0, $0, ENDLOOP1    # If condition not met, break loop
#load A[index] into $a0
sll $t0, $s1, 2            # $t0 = i * 4
add $t1, $s0, $t0          # $t1 = address of a[i]
lw $a0, 0($t1)             # load a[i]

```

```

#call PRINT function
Jal PRINT

#increment Stack pointer
Addi $sp, $sp, -4    # $sp -=4
#push A[index] onto stack
Sw $a0, 0($sp)        # push $a0 onto stack.

#increment index
Addi $s1, $s1, 1        # i++
#goto top of LOOP1
J LOOP1

ENDLOOP1:   # Initialize $s1=0
Li $s1, 0    # $s1 = 0

LOOP2:   #while index < 4
Slti $t0, $s1, 4          # $t0 = (index < 4)?
beq $t0, $0, ENDLOOP2      # breaks loop if condition not met.
#print character stored in top of stack via PRINT function
Lw $a0, 0($sp)        # $a0 = character on top of stack
Jal PRINT

#decrement Stack pointer
Addi $sp, $sp, 4        # moves stack pointer
#increment index
Addi $s1, $s1, 1        # i++
#goto top of LOOP2
J LOOP2

ENDLOOP2:
#end program
Li $v0, 10  # 10 = syscode for exit
syscall

PRINT: #print character stored in $a0
Li $v0, 11  # 11 = syscode for PRINT Character
syscall
#return
Jr $ra

```

HighLow

```

.data
STR1: .asciiz "Guess a number"
STR2: .asciiz "Congratulations"
STR3: .asciiz "Higher"
STR4: .asciiz "Lower"
SECRET: .word 42

.text
main:
# Load SECRET into $s0
La $t0, SECRET

```

```

lw $s0, 0($t0)
GUESS: # Load String "Guess a number" into $a0
La $a0, STR1
    # Call PRINT function
Jal PRINT
    # Capture input into $s1
Li $v0, 5
syscall
move $s1, $v0

    # If SECRET = INPUT, goto WIN
Beq $s0, $s1, WIN
    # If SECRET < INPUT, goto LOW
Blt $s0, $s1, LOW
    # If SECRET > INPUT, goto HIGH
Bgt $s0, $s1, HIGH

WIN:   # Print Congratulations via function
La $a0, Str2
jal PRINT
# End program
Li $v0, 10
syscall

HIGH:  # Print "Higher" via function
La $a0, STR3
jal PRINT
# Goto GUESS
J GUESS

LOW:   # Print "Lower" via function
La $a0, STR4
jal PRINT
# Goto GUESS
J GUESS

PRINT: #print String address stored in $a0
Li $v0, 4
syscall
    #return
Jr $ra

```