2nd Lab

- You will be asked to write a program that take the user input as string **(two characters only)** and find the integer value of that string. That is similar to implementing int(“56”).
- To complete this assignment, you need to be
  - familiar with the dynamics of ***read string syscall***
  - **parse** every character in the string and **find** its value
  - **calculate** the value of the whole string considering decimal digit weight.
- Build this without using loops - when finished consider using a loop for up to 5 characters, what do you need to change?

Branching (flow control)

Branching

Conditional Branching
HLL: *if, case, loop*
MIPS Assembly: *beq, bne,* bgtz, bltz, bgez, blez

Unconditional Branching
HLL: break, *Function calls!!*
Assembly: j (goto), jal (=call), jr (=return)

Any deviation from sequential instruction execution is Branching!

Branch instructions may ONLY change the program counter

new PC ≠ PC+4

Key Branching Instructions

Other instructions

bne, **bgtz, bltz, bgez, blez**

beq rs, rt, L1 # if (rs == rt) branch to instruction labeled L1

otherwise **proceed** to next instruction

j mytarget # branch to instruction at mytarget

If statement example

HLL code:

```
if (i==j)
  f = g+h
else
  f = g-h
```

f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

Compiled MIPS code:

```
    beq $s3, $s4, if  #start of if stat.
else: sub $s0, $s1,$s2 #else block start
    j   endif # else block end
if:  add $s0, $s1, $s2 # if block start
endif: subsequent instructions after
```

ensures that only one block (if or else) is executed

MIPS Instruction Offset

```
      beq $s3, $s4, if  #start of if stat.
else: sub $s0, $s1,$s2 #else block start
      j   Exit # else block end
if:   add $s0, $s1, $s2 # if block start
endif: subsequent instructions after
```

- The offset is signed (goes up or down)
- The offset represents how many **instructions** is the target far **from the next instruction**.
- 16-bit in branch / 26 bits in jump
- Remember: MIPS has a fixed size instruction (4 bytes)

beq (conditional in general) --> new PC = (PC +4)+ offset × 4*[label value]

```
0x0040000c  0x12740002   8:          beq $s3, $s4, if
0x00400010  0x02328022   9: else:   sub $s0, $s1,$s2
0x00400014  0x08100007  10:           j  endif
0x00400018  0x02328020  11: if:    add $s0, $s1, $s2
0x0040001c  0x24080005  12: endif: li $t0, 5
```

16-bit in branch

26-bit in jump

j instruction

Unconditional branch instruction

- *j* Label # jump to instruction at Label:

2	10000
6 bits	26 bits

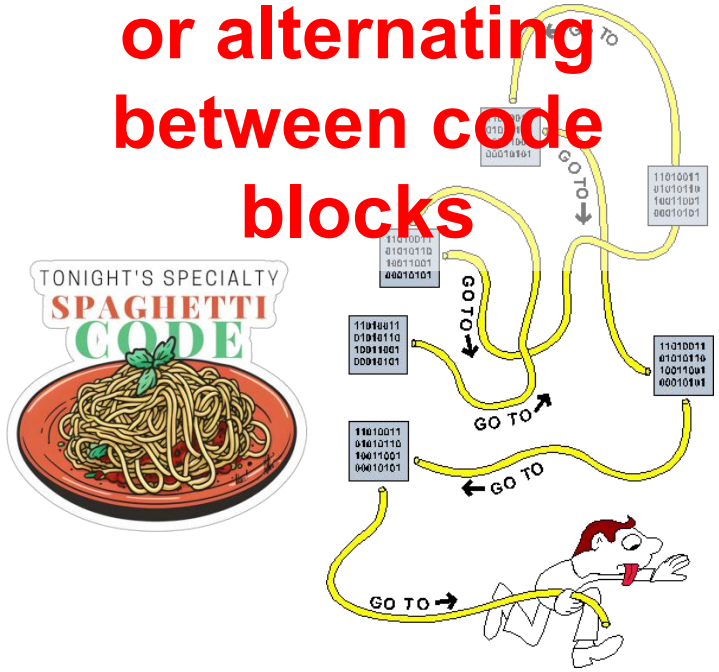
New PC ⟶

PC[31,28]	instruction offset*4

j instruction enables performing **FAR** jumps
26 bits ⟶ ±32M instructions ⟶ ± 128 Mbytes
This is how big could be your if block

ONLY Use J to build conditional logic (if and loop)

Do NOT use jump for function calls or alternating between code blocks



Adding array elements using a loop

```
1   .data
2   array:      .word 5, 10, 15     # Integer array with three elements
3   .text
4   .globl main
5   main: la $t4, array    # Load the base address of the array into $t4
6         li $t5, 0      # Initialize Loop counter
7         li $t3, 0       # Initialize Sum of array elements
8         li $t6, 3      # Load the number of elements (3) into $t6
9   loop: lw $t0, 0($t4) # Load the element at offset 0
10        add $t3, $t3, $t0  # Add the current element to the sum
11        addi $t4, $t4, 4   # Increment by 4 bytes to move to the next element
12        addi $t5, $t5, 1      # Increment the loop counter
13  Loop? bne $t5, $t6, loop      # Branch to loop if counter is not equal to 3
14        move $a0, $t3           # Move the sum to $a0 for printing
15        li $v0, 1               # Load the print integer syscall code
16        syscall
17        # Exit the program
18        li $v0, 10              # Load the exit syscall code
19        syscall
```

Initialize

Process and update counter

Key MIPS *Conditional* Branching Instructions

Instruction	RTL	Notes
beq $rs, $rt, imm	*if(R[$rs] = R[$rt]) PC ← PC + 4 + $SignExt_{18b}$({imm, 00})*	*(I-format)*
bne $rs, $rt, imm	*if(R[$rs] != R[$rt]) PC ← PC + 4 + $SignExt_{18b}$({imm, 00})*	*(I-format)*
blez $rs, imm	*if(R[$rs] <= 0) PC ← PC + 4 + $SignExt_{18b}$({imm, 00})*	*Signed comparison (I-format)*
bgtz $rs, imm	*if(R[$rs] > 0) PC ← PC + 4 + $SignExt_{18b}$({imm, 00})*	*Signed comparison (I-format)*
slt $rd, $rs, $rt	*R[$rd] ← R[$rs] < R[$rt]*	*Signed comparison (R-format)*
sltu $rd, $rs, $rt	*R[$rd] ← R[$rs] < R[$rt]*	*Unsigned comparison (R-format)*

Pseudo branching instructions!!

BLT is not a basic MIPS instruction, Why ?

Design Decision

too complicated to implement → would stretch the clock cycle time
Solution: use two basic instructions to execute its logic :)

Set Instructions

```
slt rd, rs, rt        # if (rs < rt) rd = 1; else rd = 0;
slti rt, rs, const    #if (rs < constant) rt = 1; else rt = 0;
```

BLT →

```
slt $t0, $s1, $s2   # if ($s1 < $s2) → t0 = 1
bne $t0, $zero, L   #   branch to L if $S1<$S2
```

Yes, you can write blt in the editor
But, it will be translated to two instructions

Signed vs. Unsigned

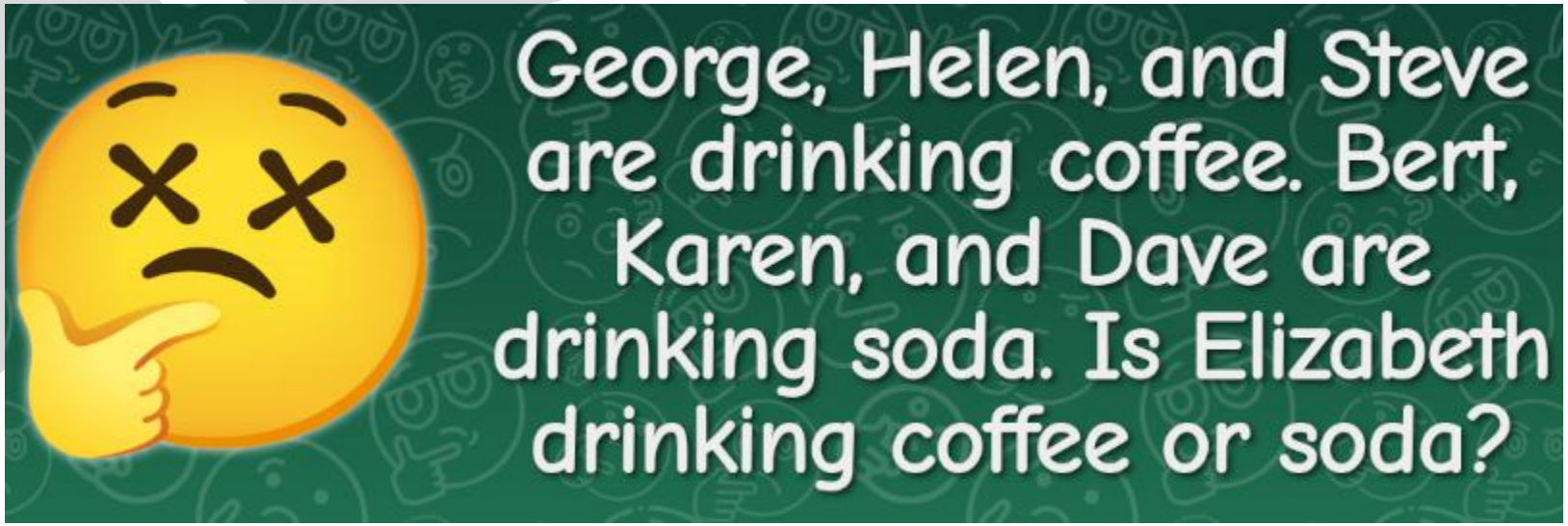
- Signed comparison: ***slt***, ***slti***
- Unsigned comparison: ***sltu***, ***sltui***
- ***Example***
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001

```
slt  $t0, $s0, $s1   # signed
```

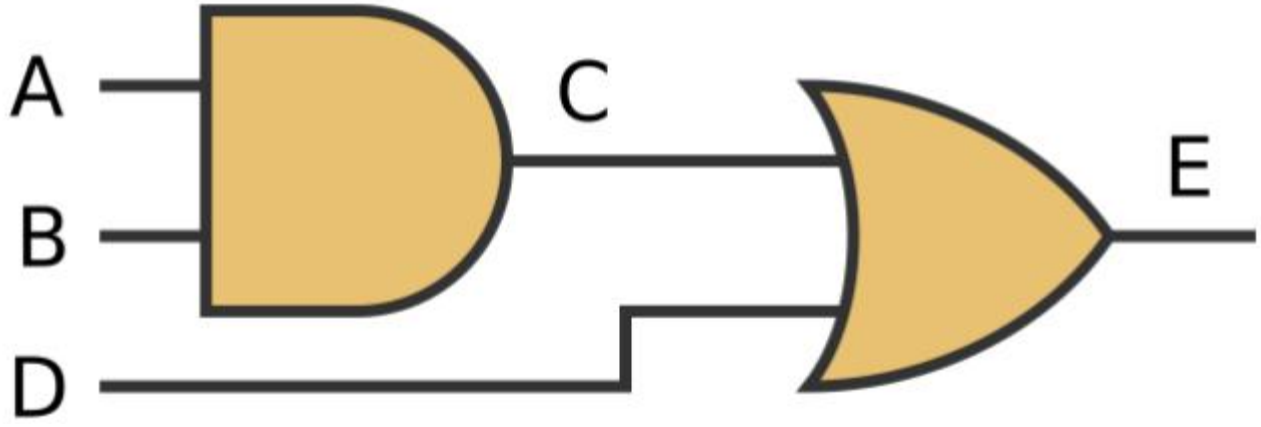
  - –1 < +1 → $t0 = 1

```
sltu $t0, $s0, $s1   # unsigned
```

  - +4,294,967,295 > +1 → $t0 = 0



MIPS Logic Instructions



Logical Operations

Sections 2.6

- Instructions for ***bitwise*** manipulation
- Bit is the smallest data unit (used in real applications)

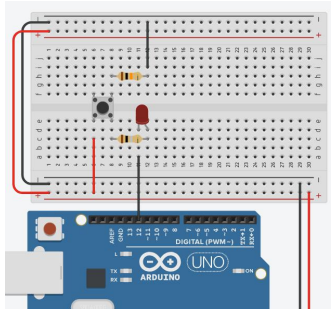
Operation	C	Java	MIPS
Shift left	<<	<<	*sll*
Shift right	>>	>>>	*srl*
Bitwise AND	&	&	*and, andi*
Bitwise OR	|	|	*or, ori*
Bitwise NOT	~	~	*nor*

Number 1	1	0	1	0	1
Number 2	1	1	1	0	0
AND	1	0	1	0	0
OR	1	1	1	0	1
XOR	0	1	0	0	1

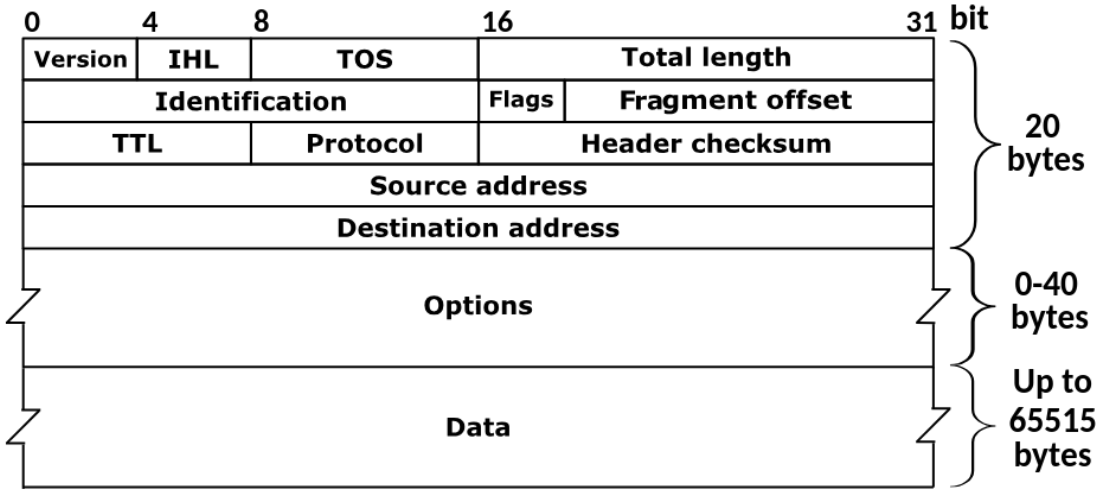
Logical Instructions (only register operands)

Instruction	*RTL*	*Notes*
sll $rd, $rt, shamt	*R[$rd] ← R[$rt] << shamt*	***sll*** by *i* bits multiplies by 2^i
srl $rd, $rt, shamt	*R[$rd] ← R[$rt] >> shamt*	***srl*** by *i* bits divides by 2^i (unsigned)
sra $rd, $rt, shamt	*R[$rd] ← R[$rt] >> shamt*	*Signed right shift*
and $rd, $rs, $rt	*R[$rd] ← R[$rs] & R[$rt]*	
or $rd, $rs, $rt	*R[$rd] ← R[$rs] | R[$rt]*	
xor $rd, $rs, $rt	*R[$rd] ← R[$rs] ^ R[$rt]*	
nor $rd, $rs, $rt	*R[$rd] ← !(R[$rs] | R[$rt])*	

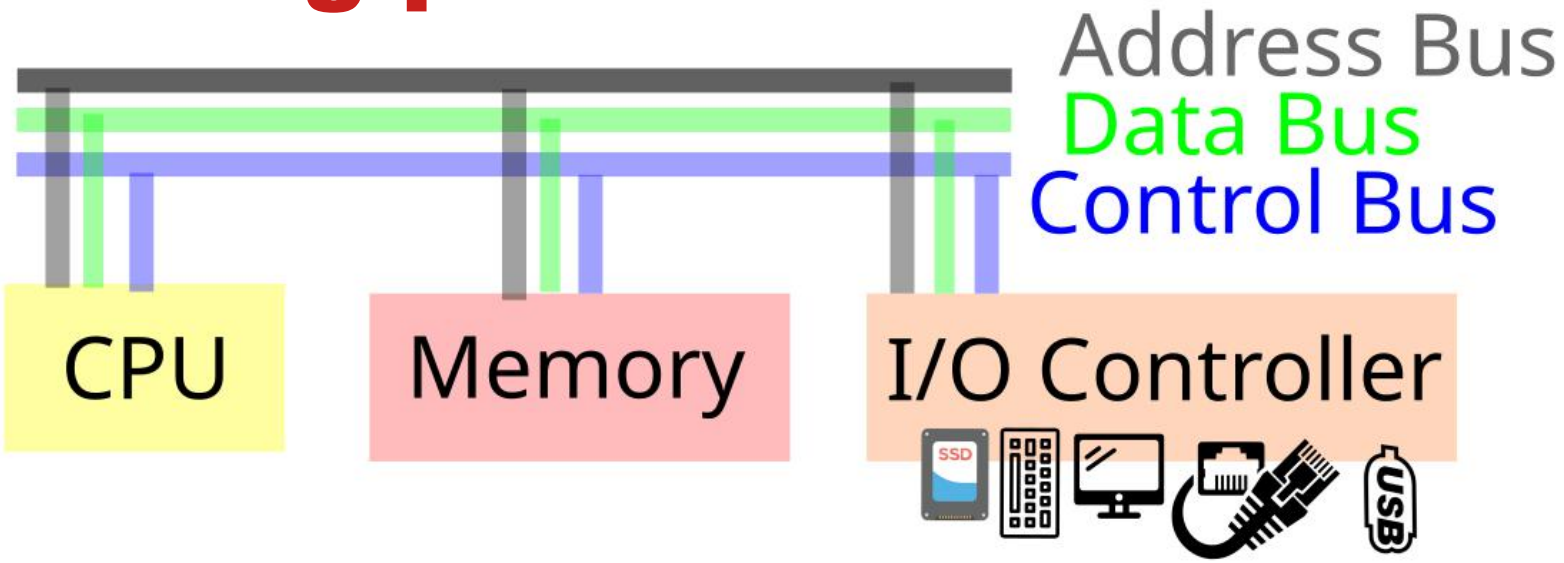
Relevant Applications

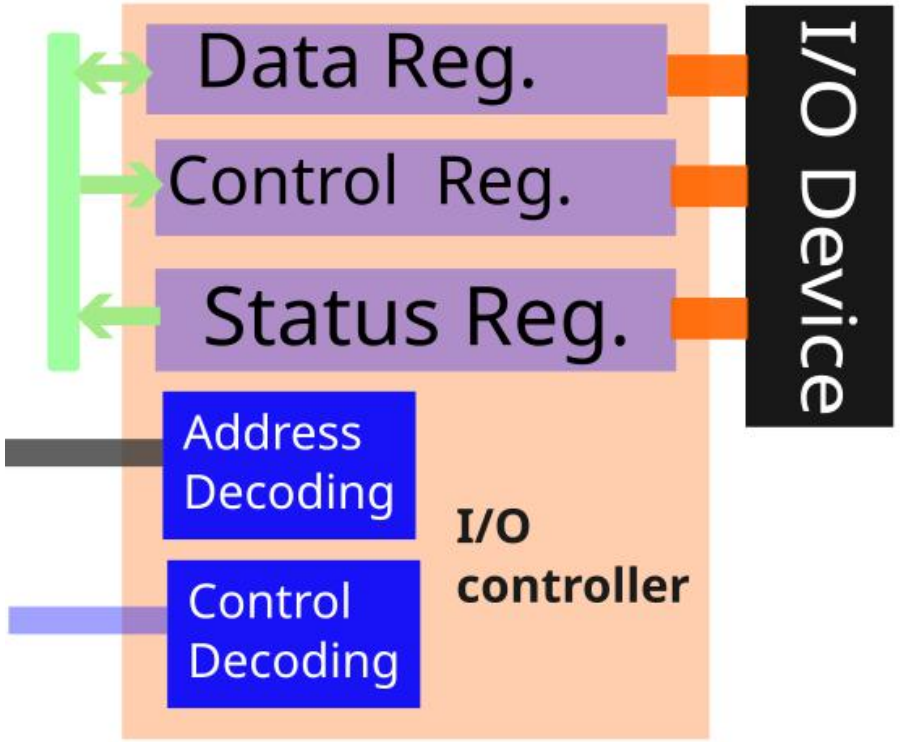


- ✓ **Embedded Systems**: Bitwise operations are heavily used in embedded systems for controlling hardware. For example, setting, clearing, toggling, or checking the status of individual bits in hardware registers.
- ✓ **Fast Arithmetic Operations**
  - ***sll*** one position is same x2
  - ***slr*** one position is the same as /2
- ✓ **Networking and Communication Protocols**
- ✓ **Data compression and encryption**



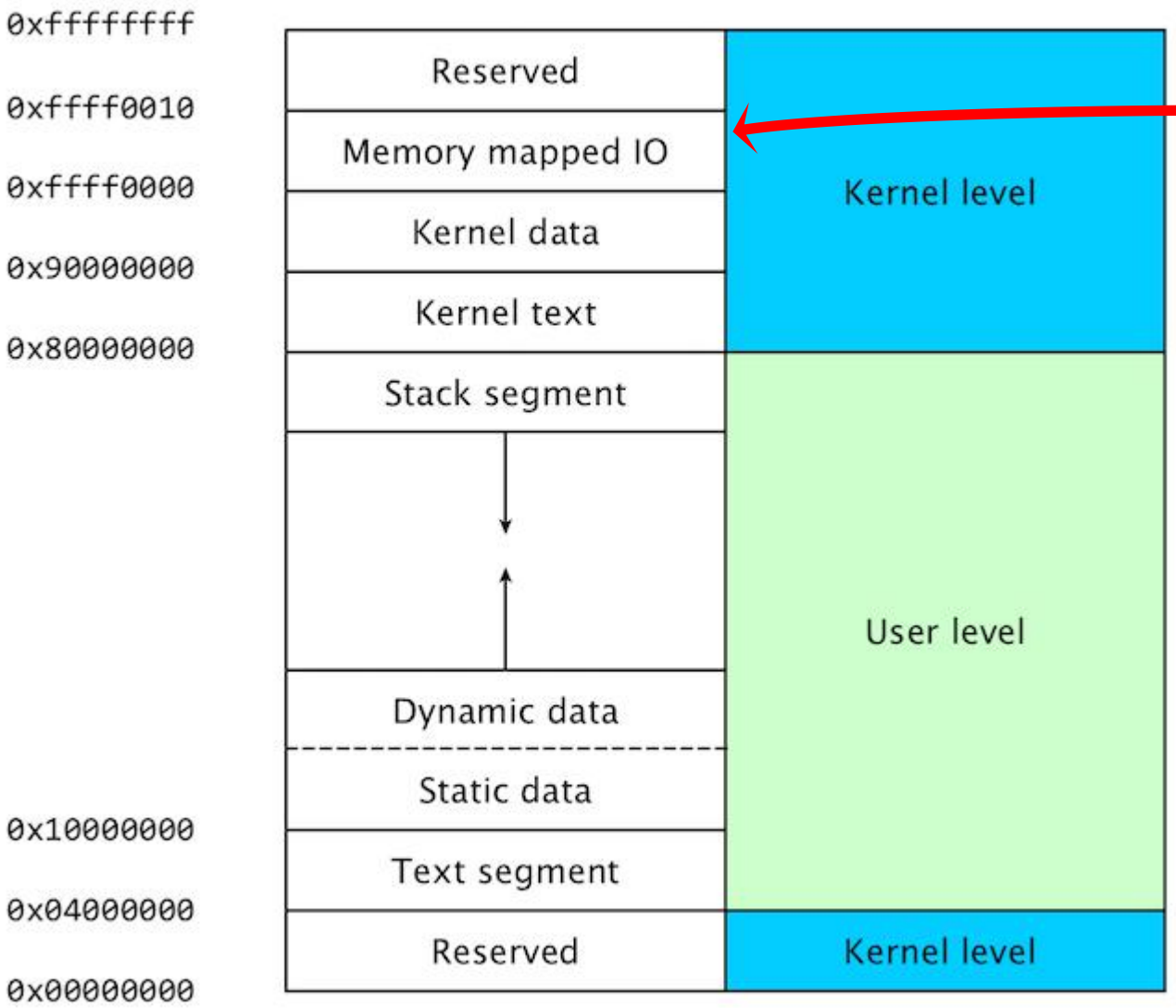
I/O big picture





- I/O device appears as a set of special-purpose registers to the CPU
  - **Status registers:** usually read-only to determine the state of the device
  - **Configuration/control registers** → configuration registers are usually read-only while control can be read/write
  - **Data registers** are used to read data from or send data to the I/O device

MIPS I/O addressing



- MIPS uses **memory mapped I/O addresses**
  - registers have memory addresses
- MIPS I/O uses memory instructions (e.g., lw, lb, ..) to transfer data between I/O registers (RISC)
  - Intel has in and out instructions (CISC)

AND Example: Keyboard Interaction

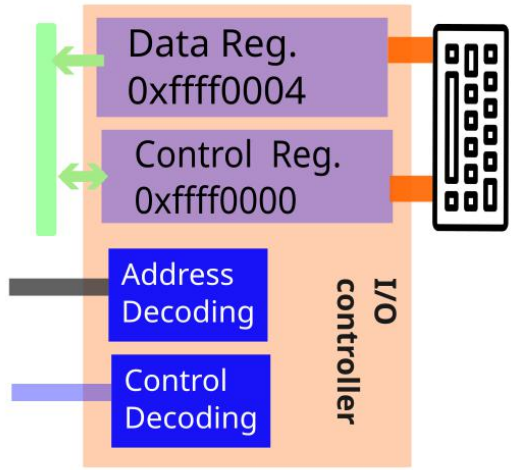
Receiver control register
@0xffff0000

Bit 0 (R) is the **ready bit:**

- set to 1 by the keyboard controller when a key is pressed
- cleared automatically when the receiver data register is read

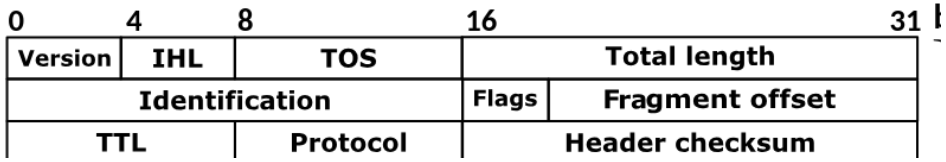
Receiver data register @
0xffff0004

The low 8 bits of this register contain the ASCII/ISO code of the last key that was pressed




```
14 getChar:
15      lui   $t0, 0xffff      # address of Keyboard control register  0xffff 0000
16      li    $t1, 1            # ready bit MASK (least significant bit) 0x0000 0001
17 key_wait:
18      lbu   $t2, ($t0)        # Read keyboard control register  at  xffff 0000
19      and   $t2, $t2, $t1     # Apply ready bit  mask
20      beqz $t2, key_wait      # 0 no key press --> busy waiting , 1 a key is pressed
21      lbu   $v0, 4($t0)       # load RECEIVER_DATA to $v0
```

Multi-bit AND Operations



- Useful to **checking (testing)** the value of ***specific*** bit(s)

STEPS:

1 The data is loaded to a register $t2

2 A ***mask*** (test pattern) is loaded in an arbitrary register $t1

3- ***and*** mask with data and store in result

4- check result and take actions

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

Applications:

- Is a button pressed?
- read a sensor from multiple bits

For AND operation:
The mask has “1”s at test bit locations and “0”s otherwise.

Multi-bit Example

0X7800 = 0B 0111 1000 0000 0000

```
4   .text
5   main:
6       lw $t0, 0xFFFFFFF0 # Read the I/O register (at address 0xFFFFFFF0)
7       lw $t1, 0x7800      # Mask to extract bits 11-14
8       and $t1, $t0,$t1   # test by MASK and register
9       srl $t1, $t1, 10   # Shift the result to the right by 10 bits
10      move $a0, $t1     # Move the value of $t1 to $a0 for printing
11      li $v0, 1         # Load the print integer syscall code
12      syscall         # Print the integer reading
13      li $v0, 10    # Load the exit syscall code
14      syscall           # Exit the program
```

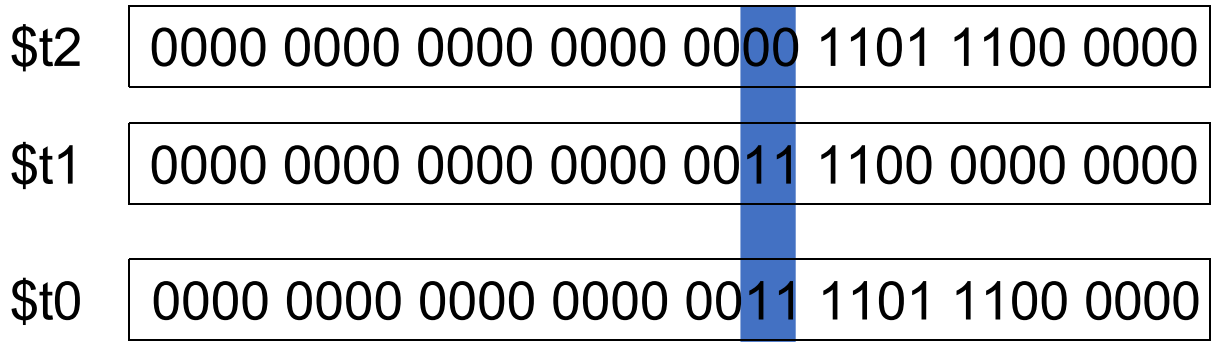
NOTE: MIPS use same memory instructions to read I/O ports (RISC instruction set)

OR Operation

- Useful to **set** one or more bits to 1 in a word and leave others unchanged
  - e.g., switch a LED on, set on a HW switch, set a configuration bit in an I/O control register

STEPS:

1 The port is read to register $t2
2 relevant bits are set in another register ($t1)
3 OR two registers
4 send the result to the target port



NOT Operations

- Useful to **invert** bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS use NOR instruction for not
  - a NOR 0 == ***NOT*** (a) *(remember: A or Zero = A)*
  - ***NOR*** *$t0, $t1, $zero ($t0 = not($t1))*

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111