More MIPS programming
(Loops/IO/Subroutines)

CS230 Tutorial 04
First: Labels and Branch instructions

- Labels give a name to a place in the code
  - Tab at the beginning of lines in your code to make labels more readable

```
begin:  addi $3, 0, 1
        sub $3, $3, $1
```

- Branch instructions perform a test (think if statements):
  - If the test succeeds, we go to the given label
  - If it fails, we continue to the next instruction in order
Branch instructions

- There are two branch instructions
  - `beq $s, $t, label` branch to label if $s$ equals $t$
  - `bne $s, $t, label` branch to label if $s$ not equals $t$

- `slt (set less than)`
  - `slt $d, $s, $t` sets $d$ to 1 if $s$ less than $t$, otherwise it sets $d$ to 0
  - Note: `slt` can cover all of $<$, $\leq$, $>$, $\geq$ by reordering $s$ and $t`
Memory instructions

- To read and write from memory we use the `lw` and `sw` instructions.
  - `sw $t, i($s)` stores the value in `$t` in the memory address `$s + i`
  - `lw $t, i($s)` loads the value in the memory address `$s + i` into `$t`

- The `array` frontend is another way to run your MIPS program, like `twoints`
- Run `array` with `/u/cs230/pub/array program.mips`
- `array` asks for a number (which will go in `$2`), and then that many more numbers. Those numbers will be stored in memory starting at the address that will go in `$1`
Loops: example

Now that we know more about branch and memory instructions, let’s put them together!
Follow along with this example question:

Write a program that sums the elements of the array and places the result in $s_3$. You can assume that the result $x$ satisfies $-(2^{31}) < x < 2^{31}$. 
Array sum in a loop: example

Write a program that sums the elements of the array and places the result in $3$. You can assume that the result $x$ satisfies $-(2^{31}) < x < 2^{31}$.

```assembly
$1 -$ mem location of start of array
; $2 -$ num elements in the array
; we want to sum the array and store in $3$
add $3, $0, 0 ; store 0 in $3$ (before starting sum, sum is 0)

; compute the sum in this loop
loop: beq $2, $0, end ; if there's no elements in the array, done
lw $4, 0($1) ; load the word in mem location in $1 into $4
; this corresponds to the first elt in the array
add $3, $3, $4 ; add current elt to the sum
addi $2, $2, -1 ; decrease number of elts left to process
addi $1, $1, 4 ; now corresponds to address of next elt in array
beq $0, $0, loop ; this is unconditional jump
; equivalent to 'j', but binasm doesn't recognize 'j'

end: jr $31
```
Subroutines: the stack

● Subroutine: think function (for ex., from python or racket)
  ○ Need to make sure regular code can keep running once the function is done

● THE STACK (memory for the program)
  ○ The address of the next free spot on the stack is kept in $30$.
  ○ Subtract from $30$ to make room for us to store new stuff (increase the stack)
    ■ When we’re done with that stuff, we add back to $30$ to clean up after ourselves
      (decrease the stack)
    ■ Subtracting makes the stack larger (-4 adds one more spot)
Subroutines: the stack

Lets work through an example.

$1 = 0x0000001A
$2 = 0x00000005
$3 = 0x000000FF

```
addi $30, $30, -8
sw $1, 0($30)
sw $2, 4($30)
lw $3, 0($30)
addi $30, $30, 8
```

Example stack
Subroutines: the stack

Lets work through an example.

Example stack

\begin{align*}
\$1 &= 0x0000001A \\
\$2 &= 0x00000005 \\
\$3 &= 0x000000FF \\
\end{align*}

\begin{align*}
\text{addi } &\$30, \$30, -8 \\
\text{sw } &\$1, 0(\$30) \\
\text{sw } &\$2, 4(\$30) \\
\text{lw } &\$3, 0(\$30) \\
\text{addi } &\$30, \$30, 8
\end{align*}
Subroutines: the stack

Let's work through an example.

Example stack

$1 = \text{0x0000001A}$
$2 = \text{0x00000005}$
$3 = \text{0x000000FF}$

```plaintext
addi $30, $30, -8
sw $1, 0($30)
sw $2, 4($30)
lw $3, 0($30)
addi $30, $30, 8
```
Subroutines: the stack

Lets work through an example.

Example stack

$1 = 0x0000001A
$2 = 0x00000005
$3 = 0x000000FF

addi $30, $30, -8
sw $1, 0($30)
sw $2, 4($30)
lw $3, 0($30)
addi $30, $30, 8
Subroutines: the stack

Lets work through an example.

Example stack

$1 = 0x0000001A$
$2 = 0x00000005$
$3 = 0x0000001A$

addi $30, $30, -8
sw $1, 0($30)
sw $2, 4($30)
lw $3, 0($30)
addi $30, $30, 8

$30 = 0x0000AAE8$
Subroutines: the stack

Let's work through an example.

Example stack

$1 = 0x0000001A
$2 = 0x00000005
$3 = 0x0000001A

addi $30, $30, -8
sw $1, 0($30)
sw $2, 4($30)
lw $3, 0($30)
addi $30, $30, 8
Subroutines: the stack

● When making a subroutine, we keep $31 on the stack
  ○ When we use jal to call a function, it puts the address of itself in $31
    ■ If we don’t save $31 before using jal, the old value of $31 will be lost
    ■ We need that value to know where to go back (jr $31) to!
    ■ This means a function call looks somewhat like this:
      ```
      addi $30, $30, -4
      sw $31, 0($30)
      jal label_for_function
      lw $31, 0($30)
      addi $30, $30, 4
      ```

● Remember if you use other registers, make sure you also save them on the stack before you call a function. It might overwrite those registers!
Quick note about IO

To write to standard output use ASCII and `sw` to 0xffff000c, for example

```assembly
lis $10
.word 0xffff000c
addi $9, $0, 0x4B ; 0x4B is ASCII for ‘K’
sw $9, 0($10)
```
Subroutines/Loops: example

Now that we know more about subroutines, let’s do an example!
Follow along with this example question:

Write a program that, for each element of the array, prints a 1 to standard output if that number is odd, and a 0 if it is even. This program should include a function that is called on each element of the array.
Subroutines/Loops

Write a program that, for each element of the standard output if that number is odd, and a program should include a function that is called the array.

```assembly
subroutines/loops example: solution
lis $18
.word 0xFFFF000C ; set up stdout in $18
add $16, $0, $1 ; we do these saves since registers $16-$31 must
add $17, $0, $2 ; be preserved by the subroutine, so this means we
; just dont need to revalidate them later
; apparently, this is supposed to be in ur course notes

loop: beq $17, $0, end ; if we're out of array elts, we're done
 ; this is always the recipe for calling a subroutine
addi $30, $30, -4 ; make room on the stack
sw $31, 0($30) ; save return address on the stack in new spot
jal isOdd ; call the subroutine
lw $31, 0($30) ; restore the return address
addi $30, $30, 4 ; reset the stack pointer

addi $17, $17, -1 ; decrement number of elts in array
addi $16, $16, 4 ; move to next elt in array
beq $0, $0, loop ; unconditional jump

end: jr $31

; subroutine code!
isOdd: addi $2, $0, 2 ; store 2 in $2 ($2 is 0 now b/c subroutine)
    lw $3, 0($16) ; store current array elt

    div $3, $2 ; divide current array elt by 2
    mfhi $3 ; check out the remainder
    addi $3, $3, 0x30 ; 0x30 is the start of digit chars in ASCII
        ; so it's '0' if it's 0, '1' if it's 1
    sw $3, 0($18) ; stdout
    jr $31 ; return to regular code (after subroutine call)
```

17
Assignment reminders

- Always test your code on the university servers before you hand it in
- Hand in the .asm file (*not the .mips file*)