Warm-Up Problem

Write an assembly language MIPS program that takes a value in register $1 and stores the value of its last base-10 digit in register $2.

CS 241 Lecture 4

Procedures

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Branching

- MIPS also comes equipped with control statements.

```
beq $s, $t, i
0001 00ss ssst tttt iiii iiii iiii iiii
```

- Branch on equal. If $s == $t then pc += i * 4. That is, skip ahead i many instructions if $s and $t are equal.

```
bne $s, $t, i
0001 01ss ssst tttt iiii iiii iiii iiii
```

- Branch on not equal. If $s != $t then pc += i * 4. That is, skip ahead i many instructions if $s and $t are not equal.

Example

Write an assembly language MIPS program that places the value 3 in register $2 if the signed number in register $1 is odd and places the value 11 in register $2 if the number is even.

```
lis $8
.word 2
lis $9
.word 3
lis $2
.word 11
div $1, $8
mfhi $3
beq $3, $0, 1
add $2, $9, $0
jr $31
```

Inequality Command

```
slt $d, $s, $t
0000 00ss ssst tttt dddd d000 0010 1010
```

- Set Less Than. Sets the value of register $d to be 1 provided the value in register $s is less than the value in register $t and sets it to be 0 otherwise.
- Note: Again, there is also an unsigned version of this command. See the reference sheet.

Example

Write an assembly language MIPS program that negates the value in register $1 provided it is positive.

```
slt $2, $1, $0
bne $2, $0, 1
sub $1, $0, $1
jr $31
```

Idea: Register 2 is 0 if register 1 is non-negative. Branch if register 2 is not zero. Otherwise negate register 1.

Extended Exercise

Write an assembly language MIPS program that places the absolute value of register $1 in register $2.

Looping

With branching, we can even do looping!

Write an assembly language MIPS program that adds together all even numbers from 1 to 20 inclusive. Store the answer in register $3.

```
lis $2
.word 20
lis $1
.word 2
add $3, $0, $0
add $3, $3, $2 ; line -3
sub $2, $2, $1 ; line -2
bne $2, $0, -3 ; line -1 from here
jr $31
```

Note: semicolons for comments in MIPS assembly

It Works But...

... if you're a good programmer, it should instinctively bother you that we hard-coded −3 in the previous slide. If we were to, say, add a new instruction in between the lines specified by our branching, all our numbers would be incorrect.
How can we fix this?

Our answer will be to use a label:

```
label:  operation commands
```

Labels aren't actually machine code, so don't take words. That means that for beq and bne, labels don't have "line numbers" on their own.

Note: A label at the end of code is allowed. It has the address of what would be the first instruction *after* the program.

Explicit Example

```
sub $3, $0, $0
sample:
add $1, $0, $0
```

- `sample` has the address 0x4, which is the location of `add $1, $0, $0`.

Looping Revisited

```
lis $2
.word 20
lis $1
.word 2
add $3, $0, $0
top:
    add $3, $3, $2
    sub $2, $2, $1
    bne $2, $0, top
jr $31
```

- Note that top in bne is computed by the assembler to be the difference between the program counter and top.
  That is, here it computes (top-PC)/4 which is (0x14 - 0x20)/4 = -3
- (PC is the line number *after* the current line)

Procedures

Let's generalize the above. How do we write procedures/functions in MIPS? Some issues:

- How do we transfer control to and from a procedure?
- What if our procedures call other procedures?
- How to we pass parameters?
- How would recursion work?
- How would we return values?
- What if a procedure wants to use registers that have data already? Calling a function might clobber such data!

Discuss solutions

- What if a procedure wants to use registers that have data already?
- We could reserve registers. But we could run out. Instead, *preserve* registers: save and restore them. But where?

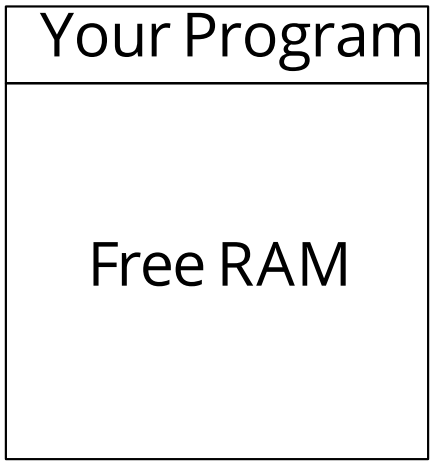
Discuss solutions

- We could reserve registers. But we could run out. Instead, *preserve* registers: save and restore them. But where?
- Well, we have lots of memory in RAM that we can use!
- However, we don’t want our procedures to use the same RAM. They could conflict.
- How can we guarantee that procedures don’t erase each other’s work?

Discuss solutions

- How can we guarantee that procedures don't erase each other's work?
- We would need to keep track of the free RAM. The loader helps us out here by letting us know where the start of free RAM is (see next slide).

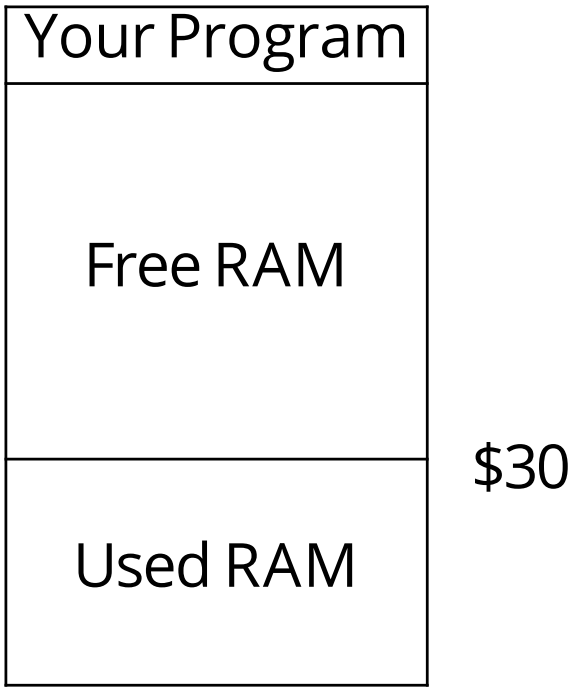
RAM



Register $30 initially points to the very bottom of the free RAM. It can be used as a bookmark to separate the used and unused free RAM **if** we allocate from the one end, and push and pop things like a stack!

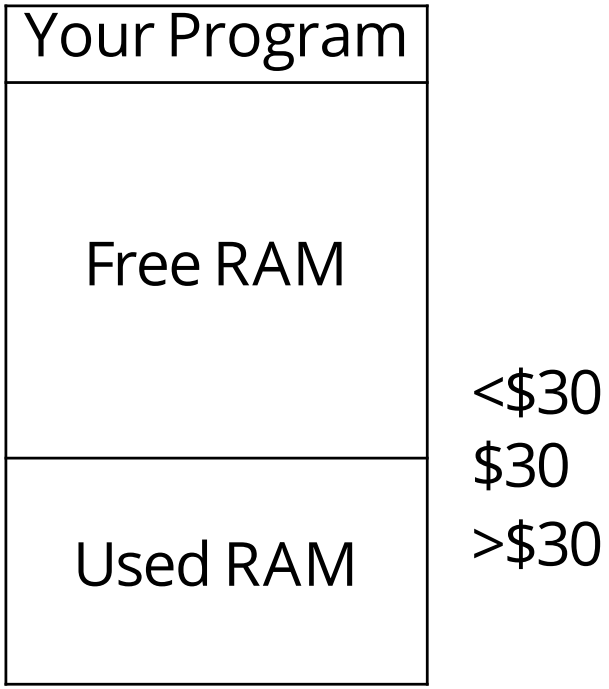
In other words, we will use $30 as a pointer to the top of a stack.

RAM



Really, $30 points to the top of the stack of memory in RAM.

RAM



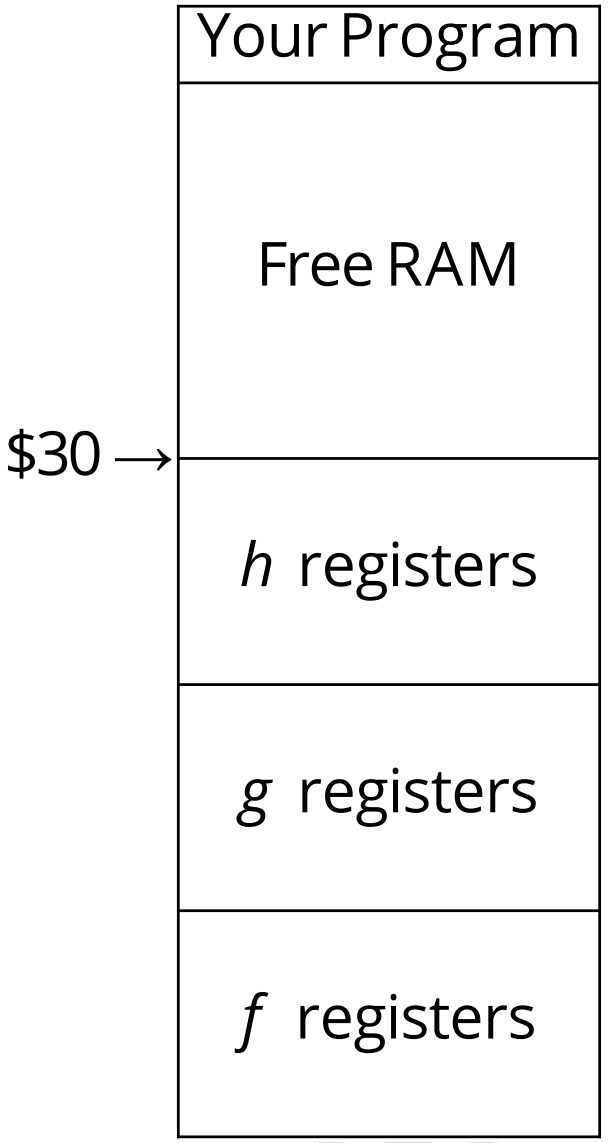
Because our program is at zero, the stack grows from *high memory* to *low memory*, so pushing involves *reducing* the value of $30. (This is confusing, but you get used to it.)

Example

Suppose procedures *f* , *g* and *h* are such that:

f calls procedure *g*
g calls procedure *h*
h returns
g returns
f returns.

Then, your RAM with this model will look like it does on the right



Previous Example

In the previous example:

- Calling procedures pushes more registers onto the stack and returning pops them off.
- This is a stack, and we call $30 our **stack pointer**.
- We can also use the stack for local storage if needed in procedures. Just reset $30 before procedures return.
- Note that the MIPS standard guideline often will use $29 for this purpose. (We use it differently)

Template for Procedures

Template for a procedure *f* that modifies registers $1 and $2:

```
f:
  sw $1, -4($30)  ; Push registers we modify
  sw $2, -8($30)
  lis $2          ; Decrement stack pointer
  .word 8
  sub $30, $30, $2
    ; Insert procedure here
  add $30, $30, $2 ; Assuming $2 is still 8
  lw $2, -8($30)    ; Pop = restore
  lw $1, -4($30)
    ; Uh oh!          How do we return?
```

Returning

- There is a problem with returning:

```
main:
lis $8
.word f ; Recall f is an address
jr $8 ; Jump to the first line of f
(NEXT LINE)
```

- Once *f* completes, we really want to jump back to the line labelled above as (NEXT LINE), i.e., set the program counter *back* to that line. How do we do that?

A New Command!

```
jalr $s
0000 00ss sss0 0000 0000 0000 0000 1001
```

- Jump and Link Register. Sets $31 to be the PC and then sets the PC to be $s. Accomplished by temp = $s then $31 = PC then PC = temp.
- Be careful about the ordering of these steps. Read the reference carefully!

- A new problem: jalr will overwrite register $31. How do we return to the loader from main after using jalr? What if procedures call each other?

A New Command!

- A new problem: jalr will overwrite register $31. How do we return to the loader from main after using jalr? What if procedures call each other?
- We need to save this register first!

Main Changes

```
main:
    lis $8
    .word f
    sw $31, -4($30) ; Push $31 to stack
    lis $31 ; Use $31 since it's been saved
    .word 4
    sub $30, $30, $31
    jalr $8 ; Overwrites $31
    lis $31 ; Use $31 since we'll restore it
    .word 4
    add $30, $30, $31
    lw $31, -4($30) ; Pop $31 from stack
    jr $31 ; Return to loader
```

Procedure Changes

```
f:
    sw $1, -4($30) ; Push registers we
                   ; will modify
    sw $2, -8($30)
    lis $2
    .word 8
    sub $30, $30, $2 ; Decrement stack
                     ; pointer
    ; Insert procedure here
    add $30, $30, $2 ; Assuming $2 is
                     ; still 8
    lw $2, -8($30) ; Pop registers to
                   ; restore
    lw $1, -4($30)
    jr $31 ; New line!
```