CS/COE 0447

Functions:
Calling Conventions,
and The Stack

wilkie (with content borrowed from:
Jarrett Billingsley
Dr. Bruce Childers)

Lightning Recap

- We can turn if-else pseudo-code into:

```
if(s0 == 30) {
        block A
}
else {
        block B
}
block C
```

```
  beq s0, 30, blockA
  b blockB
blockA:
    block A
  b blockC
blockB:
    block B
blockC:
    block C
```

Our Alternative Approach

- If you see a conditional branch followed by an unconditional one...

  ...you can **merge them** like so:

invert the condition and **get rid of** the first label.

```
beq s0, 30, if
b else
if:
    block A
  b end
else:
    block B
end:
```

```
bne s0, 30, else

    block A
  b end
else:
    block B
end:
```

Calling Conventions

It’s kind of like computer etiquette.

What is a Calling Convention (CS 449)

- It ensures our programs don't trip over their own feet
- It's **how machine-language functions call one another**.
  - How **arguments** are passed.
  - How **return values** are returned.
  - How control **flows into/out of the function**
  - What **contracts** exist between the call**er** and the call**ee**.
  - **For instance: What registers it will use.**
- Common terms we will be using: Caller v. Callee:

```
void major() {
    minor();
}
```

call**er**

```
void minor() {
    ...
}
```

call**ee**

The Program Counter

- a program's instructions are in memory, so they have **addresses**
- the PC (program counter) holds the address of **the next instruction to run**

time

PC
0x8000
0x8004
0x8008
0x800C
0x8010

0x8000 ? 0x8014

```
top: 0x8000 lw  t0, (s0)
     0x8004 add t0, t0, 1
     0x8008 sw  t0, (s0)
     0x800C add s0, s0, 4
     0x8010 blt s0, s1, top
```

btw what pattern do you notice about these addresses?

Branches and the Program Counter

- The "branch" instructions we have already seen actually simply interact with the program counter.

Instruction	Meaning
`beq a, b, label`	`if(a == b) { $pc = label }`
`bne a, b, label`	`if(a != b) { $pc = label }`
`blt a, b, label`	`if(a < b)  { $pc = label }`
`ble a, b, label`	`if(a <= b) { $pc = label }`
`bgt a, b, label`	`if(a > b)  { $pc = label }`
`bge a, b, label`	`if(a >= b) { $pc = label }`

- Each sets `$pc` to the address of whatever label is given.

Control Flow (Reprise)

- When the caller calls a function, where do we go?
- When the callee's code is finished, where do we go?

```
void fork() {
    knife();
    spoon++;
}
```

caller

```
void knife() {
    spork++;
    spatula--;
}
```

callee

MIPS ISA: The “jal” Instruction

- We **call** functions with **jal: j**ump **a**nd **l**ink

PC `0x8004`

	0x8000	li a0, 10
	0x8004	jal func
	0x8008	li v0, 10
	...	...
func:	0x8C30	li v0, 4
	...	...

What address should go into PC next?

PC `0x8C30`

When **func** returns, where will we go?

ra `0x8008`

This is what **jal** does:

It **jumps** to a new location, and makes a **link** back to the old one in the **ra** (return address) register

and this is ALL it does.

MIPS ISA: The "jr" Instruction

- We return from functions with **jr: j**ump to address in **r**egister

Now we're at the end of **func. ra** still has the proper return address (***thanks to jal***)

PC	0x8C38
ra	0x8008

`jr ra` copies **ra** into **pc.**

PC	0x8008

	Address	Instruction
	0x8000	`li a0, 10`
	0x8004	`jal func`
	0x8008	`li v0, 10`
	...	...
func:	0x8C30	`li v0, 4`
	0x8C34	`syscall`
	0x8C38	`jr ra`

and this is ALL it does.

Don’t Step on Other’s Toes

- let's track **PC** and **ra** as we run this code.

	PC	ra
	`0x8000`	`0x0000`
After `jal fork`:	`0x8020`	`0x8004`
After `jal spoon`:	`0x8040`	`0x8024`
After `jr ra`:	`0x8024`	`0x8024`
After `jr ra`:	`0x8024`	`0x8024`
After `jr ra`:	`0x8024`	`0x8024`
After `jr ra`:	`0x8024`	`0x8024`
After `jr ra`:	`0x8024`	`0x8024`

!

UHHHHHHHH

```
        0x8000  jal  fork
        0x8004  li   v0, 10
           ...  ...
fork:   0x8020  jal  spoon
        0x8024  jr   ra
           ...  ...
spoon:  0x8040  jr ra
```

What’s the Deal

- there's only **one return address register**
- if we call more than **one level deep,** things go horribly wrong
- Could we put it in another register?
  - Then what about three levels deep? four?
    - We just don't have ***enough registers…***
  - ***Pro-tip:*** *Resist any urge to preserve registers in other registers.*
    - Just use memory.

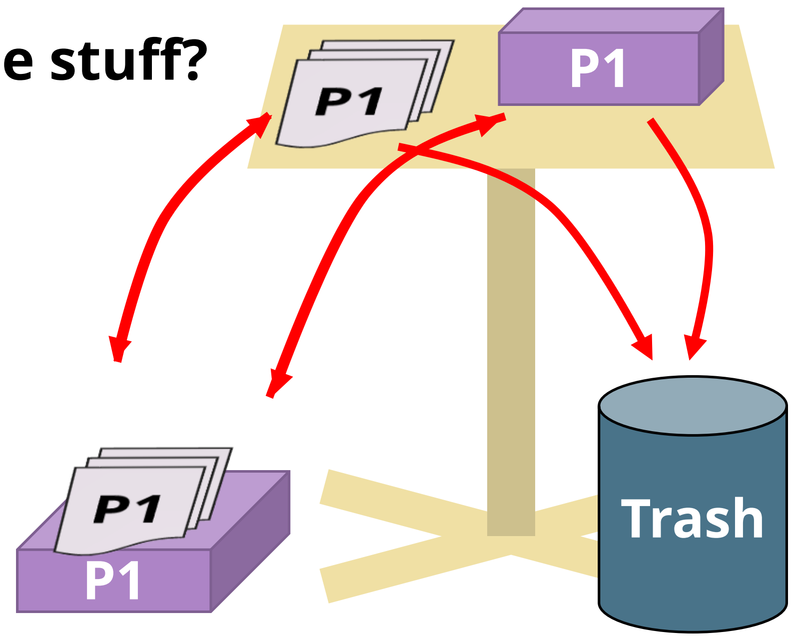
- So **where** do we put things when we don't have room in registers?
  - Memory. But more specifically…

The Stack

It’s just memory.

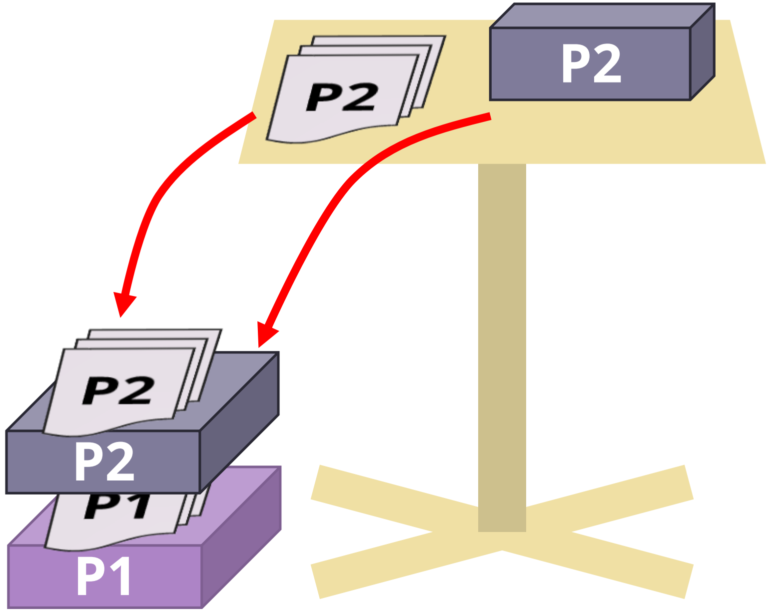
A Busy Desk

- There's a tiny desk that three people have to share
- Person 1 is working at the desk. it's covered in their stuff.
- Person 2 **interrupts them** and needs to do some important work
- **What does person 2 do with the stuff?**
  - Throw it in the trash?
- They **put it somewhere else.**
- When they are done,
  they put it back.
  - According to “etiquette”
    aka “calling convention” ☺



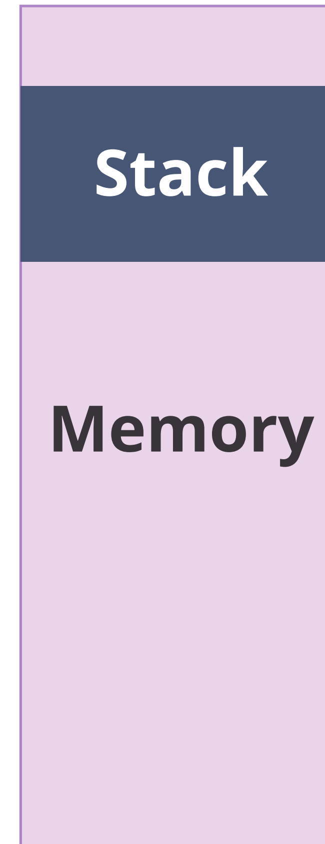
A Very Busy Desk

- So person 2 is working...
- Then person 2 is interrupted by person 3.
- When person 3 is done, person 2 will come back.
- Where do we put person 2's stuff?
  - **on top of the stack of stuff.**
- The desk represents the **registers.**
- The people are **functions.**
- The stack of stuff is... **the stack.**



What is the Stack? ("sp" Register)

- It's an **area of memory** provided to your program by the OS
  - When your program starts, it's already there.
- The stack is holds **information about function calls:**
  - **The return address** to the caller
  - **Copies of registers** that we want to change
  - **Local variables** that can't fit in registers
- How do we access the stack?
  - Through the **stack pointer (sp) register**
  - This register is initialized for you by the OS too.



Accessing the Stack (animated)

- Let's say **sp** starts at the address **`0xF000`**
- We want to **push** something on the stack
- The first thing we'll do is **move sp to the next available slot**
- Clearly, that's the *previous* address
  - **Subtract** 4 from **sp** (why 4?)
- Then, we can store something in **the memory that sp points to.**

	...	...
	0xF008	0x00000000
	0xF004	0x00000000
sp →	0xF000	0x00000000
	0xEFFC	0xC0DEBEEF

MIPS: Accessing the Stack (animated)

- Say `ra = 0xC0DEBEEF`
- First: move the stack pointer down (up?):

```
sub sp, sp, 4
```

- Then, store **`ra`** at the address that **`sp`** holds.

```
sw  ra, (sp)
```

- Now the value in **`ra`** is **saved** on the stack, and **we can get it back later.**
  - And we can store as many return addresses as we want!

	...	...
	0xF008	0x00000000
	0xF004	0x00000000
sp →	0xF000	0x00000000
	0xEFFC	0xC0DEBEEF

MIPS: Going the Other Direction (animated)

- Now we wanna **pop** the value off the stack and put it back in **`ra`**
- We do the same things, but in reverse

```
lw  ra, (sp)
```

- Then, we move the stack pointer…
  up? down? whatever

```
add sp, sp, 4
```

- Now we got back the old value of **`ra`**!
- And **`sp`** is back where it was before!

`ra` `0xC0DEBEEF`

	...	...
	0xF008	0x00000000
	0xF004	0x00000000
	0xF000	0x00000000
sp →	0xEFFC	0xC0DEBEEF

MIPS: Simplifying: "push" and "pop" pseudo-ops

- The push and pop operations always look and work the same
- Since you'll be using them in most functions, we shortened em!
- If you write **push ra** or **pop ra**, it'll do these things for you!
  - Thank goodness.

```
push ra   =   subi sp, sp, 4
              sw   ra, (sp)

pop  ra   =   lw   ra, (sp)
              addi sp, sp, 4
```

These are **pseudo-ops:** fake instructions to shorten common tasks

These can be used with ANY register, not just **ra!**

Some Steel-Toed Boots

sp ➡ `0x8004`

	PC	ra
	0x8000	0x0000
After **jal fork**:	0x8020	0x8004

Then we **push ra on the stack!**

	PC	ra
After **jal spoon**:	0x8040	0x802C
After spoon **jr ra**:	0x802C	0x802C

Then we **pop ra off the stack!**

	PC	ra
Before fork **jr ra**:	0x8034	0x8004
After fork **jr ra**:	0x8004	0x8004

```
0x8000  jal   fork
fork:
0x8020  push  ra
0x8028  jal   spoon
0x802C  pop   ra
0x8034  jr    ra
spoon:
0x8040  jr    ra
```

Writing a Simple Function

- Writing a function follows a simple structure:

1. Give it a name (label).
2. Save **ra** to the stack.
3. Do whatever.
4. Load ra from the stack.
5. Return!

```
spoon:
    push ra
    your code goes here
    pop  ra
    jr   ra
```

- Push everything you may need. Pop it back at the end.

Extending: Preserving other Registers

- Writing a function follows a simple structure:

1. Give it a name (label). **`spoon:`**

2a. Save **ra** to the stack. `push ra`

2b. Save **s0** to the stack. `push s0`

3. Do whatever. *`your code using s0 goes here`*

4a. Load s0 from the stack. `pop s0`

4b. Load ra from the stack. `pop ra`

5. Return! `jr ra`

```
spoon:
    push ra
    push s0
    your code using s0 goes here
    pop  s0
    pop  ra
    jr   ra
```

- Push everything you may need. Pop it back **in reverse order** at the end. **(We will look at why we do this and when soon!)**

When to “push” and “pop”

- Treat pushes and pops like the **{ braces }** around a function.

```
spoon:
    push ra # {
    # 800 instructions
    # so much stuff omg
    pop  ra # }
    jr   ra
```

pushes come at the beginnings of functions

pops come at the end

That is it, seriously, don't make it more complicated

never push or pop anywhere else please

Calling Conventions: Arguments and Return Values

Functions that can now do things!

Passing Arguments

- If we have a function in a higher level language…

```
v0            a0         a1
int gcd(int a, int b) {
  while(a != b) {
    if(a > b)
      a -= b;
    else
      b -= a
  }
  return a;
}
```

We use particular registers to pass arguments and return values.

We already know how to return. How?

For this, just *put the value you want to return in **v0** before jr ra.*

The "a" and "v" Registers

- **a0-a3** are the **argument registers**
- **v0-v1** are the **return value registers**
  - This is just a **convention,** there's nothing special about them
- To call a function…
  - You **put its arguments in the a registers** before doing a jal
- Once control is inside the callee…
  - The arguments are just "there" in the a registers.
  - Cause they are.
  - They didn't go anywhere…

Let's Write a Function

- let's write a function like this:

```
int add_nums(int x, int y) {return x + y;}
```

- inside of our **add_nums** asm function...
  - which register represents **x**?
  - which register represents **y**?
  - which register will hold the sum that we return?

```
add_nums:
    add v0, a0, a1
    jr  ra
```

Let’s Call That Function

- Now let's make **main** do this:

```
print(add_nums(3, 8))
```

- How do we set 3 and 8 as the arguments?
- How do we call **add_nums**?
- Afterwards, which register holds the sum?
- So how can we print that value?
- *Why* do syscalls put the number of the syscall in **v0**?
  - Well what do you get when you cross an elephant and a rhino?
    - Hell if I know (it’s a bad, confusing decision. Sorry.)

```
li   a0, 3
li   a1, 8
jal  add_nums
move a0, v0
li   v0, 1
syscall
```

Calling Conventions: Saved and Unsaved Registers

It’s about how to trust functions to not step on your toes.

An Innocent Looking Function

- Let's make a variable and a function to change it

```
.data
  counter: .word 0
.text
increment:
  la  t0, counter
  lw  t1, (t0)
  add t1, t1, 1
  sw  t1, (t0)
  jr  ra
```

then we can call it

```
main:
  jal increment
  jal increment
  jal increment
```

Everything's Fine Right?

- Let's write a loop that calls it ten times in a row
- So we need a loop counter ('*i*' in a for loop)

```
  li t0, 0  # our counter
loop_begin:
  jal increment
  add t0, t0, 1
  blt t0, 10, loop_begin
loop_end:
```

(another way to write a for loop)

If we run this, it only increments the variable **once.** ☹

Why? let's put a **breakpoint** on blt and see what it sees.

Another Piece of the Calling Convention Puzzle

- When you call a function, **it's allowed to change some registers**
- But other registers **must be left exactly as they were**

functions are required to put these registers **back the way they were before they were called.**

Saved	Unsaved
`s0-s7`	`v0-v1`
`sp`	`a0-a3`
`ra*`	`t0-t9`

anyone can change these. after you call a function, **they might have totally different values from before you called it.**

*ra is a little weird cause it's kinda "out of sync" with the other saved regs but you DO save and restore it like the others

When You Call a Function...

- After a `jal`, **you have no idea what's in these registers.**

```
...
jal   increment
...
```

Unsaved
`v0-v1`
`a0-a3`
`t0-t9`

could be nonsense!
garbage! bogus!

Why It Broke

- If we look at this code again…

```
  li t0, 0
loop_begin:
  jal increment
  add t0, t0, 1
  blt t0, 10, loop_begin
loop_end:
```

t0 is our loop counter and everything's fiiiine.

uh oh.

WHAT IS IN t0 NOW??

Instead, this is a great place to use an **s register.**

Fixing Our Function (Abiding by Convention)

- If we use an s register...

```
    li s0, 0
loop_begin:
    jal increment
    add s0, s0, 1
    blt s0, 10, loop_begin
loop_end:
```

s0 is our loop counter and everything's fiiiine.

uh oh.

oh whew, we used an s register, it's fine.

but s registers aren't magic. **they don't do this automatically.**

The "s" Register Terms of Service

- If you want to use an s register...
- You must **save and restore it,** just like **ra.**

```
my_func:
    push ra
    push s0

        code that uses s0! it's fine! we saved it!

    pop  s0
    pop  ra
    jr   ra
```

moving the papers off the desk

putting the papers back

the pops happen
in **reverse order!**

Things To Consider:

- You must **always** pop the same number of registers that you push.
- To make this simpler for yourself… **make a label before the pops.**
  - Then you can leave the function by jumping/branching there.
- Again: *only push at the top of the function and only pop at the bottom*.
  - Never anywhere else!

```
my_func:
  push ra
  push s0
  ...
  bge ...
  b exit_func
  ...
exit_func:
  pop  s0
  pop  ra
  jr   ra
```