



# Week 10 Review



# Calling Conventions Review

- **Calling conventions** define the protocol for calling a function with arguments and getting the return values.
  - **An agreement** between **caller** and **callee**.
  - Some functions are both (nesting, recursion).
- They define things like:
  - How to pass parameters and get return values.
  - Who manages the stack and when?
  - Who is responsible to preserve which registers?

# Passing Arguments

- `x,y,z = some_function(a,b,c)`
- Register-based:
  - **Caller** puts arguments in registers `$a0, $a1, ...`
- Stack-based, **callee**-pops: ← **what we usually use**
  - **Caller** pushes arguments to stack in order A,B,C.
  - **Callee** pops arguments (in order C, B, A).
- Stack-based, **caller**-pops:
  - **Caller** pushes arguments to stack in order A,B,C
  - **Callee** loads argument but does not pop.
  - **Caller** will clear arguments from stack.
- Could combine reg-based and stack-based.

# Returning values

- `x,y,z = some_function(a,b,c)`
- Register-based:
  - **Callee** puts the arguments in registers `$v0, $v1`
- Stack-based: **← what we normally use**
  - **Callee** pushes return value(s) onto the stack in order `z,y,x`.
  - **Caller** pops return values (will get order `x,y,z`)
- In reality, register-based is common since there is seldom a need to return more than a single value.

# Preserving Registers

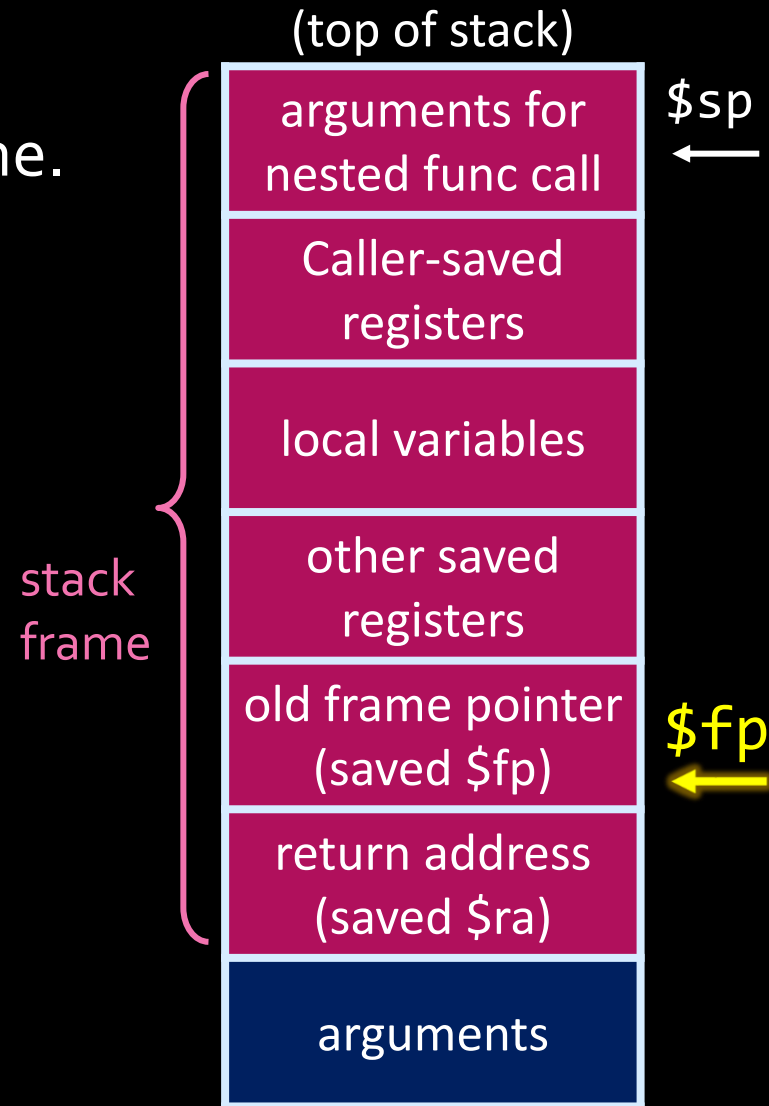
- Registers \$to-\$t9 are **caller-saved**
  - If the **caller** needs their values, save them.
  - Push before calling a function (before arguments)
  - Pop when function returns (after popping ret. val).
  - Also \$a0-\$a3, \$v0, \$v1
- Registers \$s0-\$s7 are **callee-saved**
  - If a **callee** uses them, save them.
  - Push at the beginning, after popping arguments.
  - Pop at the end, before pushing return value.
  - Also \$ra (if using), \$fp (if using)
- **You need to do this to maintain correctness!**

# Preserving Registers

- \$ra must be preserved if calling other functions from inside a function.
- Option 1:
  - Push when calling functions (before arguments)
  - Restore after popping return value
- Option 2 (important when using stack frames):
  - Push early after entering a function
  - Pop just before pushing the return value.
  - Required when using stack frames!
- You can use any option as long as it's correct.

# Stack Frames

- `$fp` used to manage the stack frame.
  - Useful with local variables.
  - Works with **Caller-pop** arguments.
  - **No need to use it unless we tell you.**
- Calling:
  - Push `$ra`
  - **Push old (caller) `$fp`**
  - `$fp ← $sp`
  - Save registers
- Returning
  - Restore registers
  - `$sp ← $fp`
  - Pop old `$fp`
  - Pop old `$ra`



# Question 1

- Final Exam, Winter 2012:

3. In the space below, write a short assembly language program that is a translation of the program on the right. You can assume that `i` has been placed on the top of the stack, and that the return value should be placed on the stack as well before returning to the calling program. Make sure that you comment your code so that we understand what you're doing. **(10 marks)**

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}
```

- How would you convert this to assembly language?



# Question 1

- You are given an assembly implementation of the function below.
- Implement a main that computes:  
 $\text{series}(42) - \text{series}(35)$

```
int series(int n)
{
    int s = 0;
    int j;
    for (j=1; j <= n; j++)
        s += j;
    return s;
}
```

# Question 1

- Hints:
  - You have to call series twice.
  - You need to preserve the value between calls.
    - Or the result will be wrong.
  - But you don't need to preserve everything!

# Question 2

- What does the following function do?:

```
myfunc:      lw $t0, 0($sp)
              addi $sp, $sp, 4
              addi $t1, $zero, 2
              div $t0, $t1
              mfhi $t0
              beq $t0, $zero, LABEL1
              add $t2, $zero, $zero
              j LABEL2

LABEL1:      addi $t2, $zero, 1
LABEL2:      addi $sp, $sp, -4
              sw $t2, 0($sp)
              jr $ra
```

## Question 2

- We divided  $\$t0$  by 2
  - `div` puts remainder in `HI`
- If remainder is 0  $\rightarrow$  return 1
- if remainder is not 0  $\rightarrow$  return 0
- This is a function that returns 1 if a number is even or 0 if it is odd

# Question 2

Write code to do the following:

- Create an array of integers
  - The last value in the array is zero (to stop the loop)
- Use the function we just saw to count the number of even values in the array.
- Your code must use registers **\$t0-\$t9**

# Question 3

- sign function

```
def sign(i):  
    if(i > 0):  
        result = 1  
    else if(i < 0):  
        result = -1  
    else:  
        result = 0  
    return result
```

# Code for sign(i)

```
# registers: $t0 = i, $t1 = return value
sign:    lw $t0, 0($sp)           # pop i off the stack
        addi $sp, $sp, 4

        # handle case of i > 0
if_gt:   blez $t0, if_lt
        addi $t1, $zero, 1
        j end

        # handle case of i < 0
if_lt:   beq $t0, $zero, eq
        addi $t1, $zero, -1
        j end

        # handle case of i == 0
eq:      add $t1, $zero, $zero

end:     addi $sp, $sp, -4        # push return value
        sw $t1, 0($sp)
        jr $ra                  # return to caller
```

# Question 3

- Write a function `numPlus(arr, n)` that returns the number of elements larger than zero in `arr` minus the number of negative elements.
- Weird restrictions:
  - Not allowed to use loops.
  - Only allowed one branch, it must be `bne` or `beq`.
  - Not allowed any other branches...
  - ...but can use `sign` function.
- How would you do it in C?

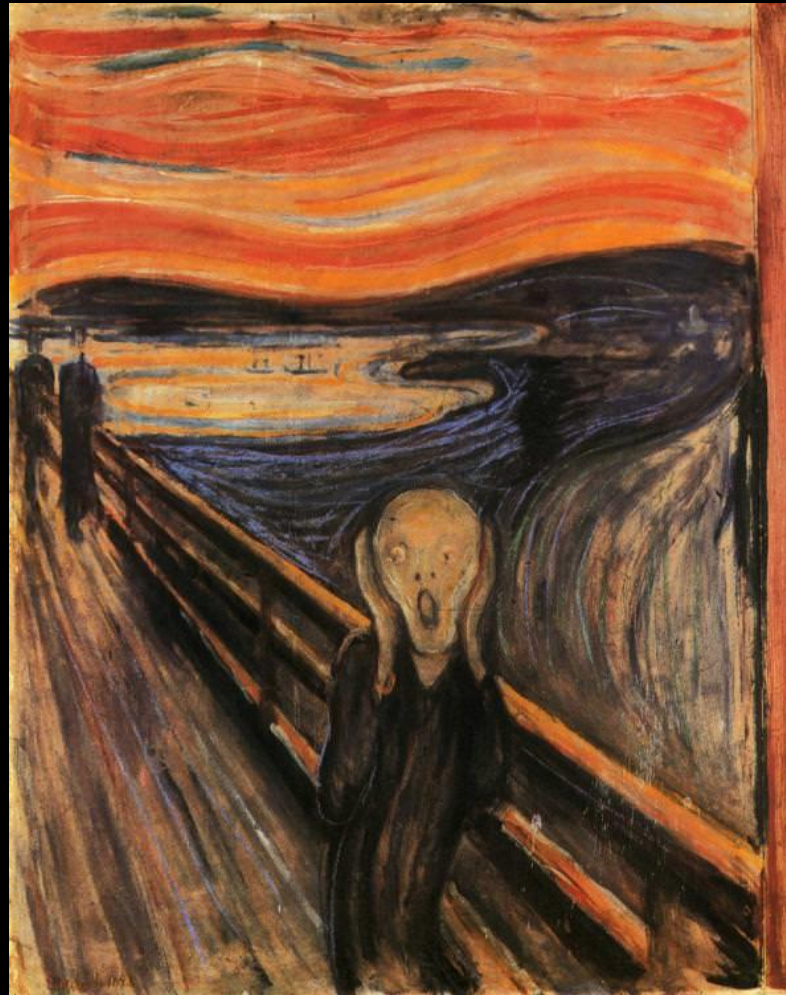


## Question 3

- Since  $\text{sign}(i)$  is  $-1, 0, 1$ , we can compute the sum of  $\text{sign}(\text{arr}[i])$  for all  $i$ .
- Use recursion instead of loops.

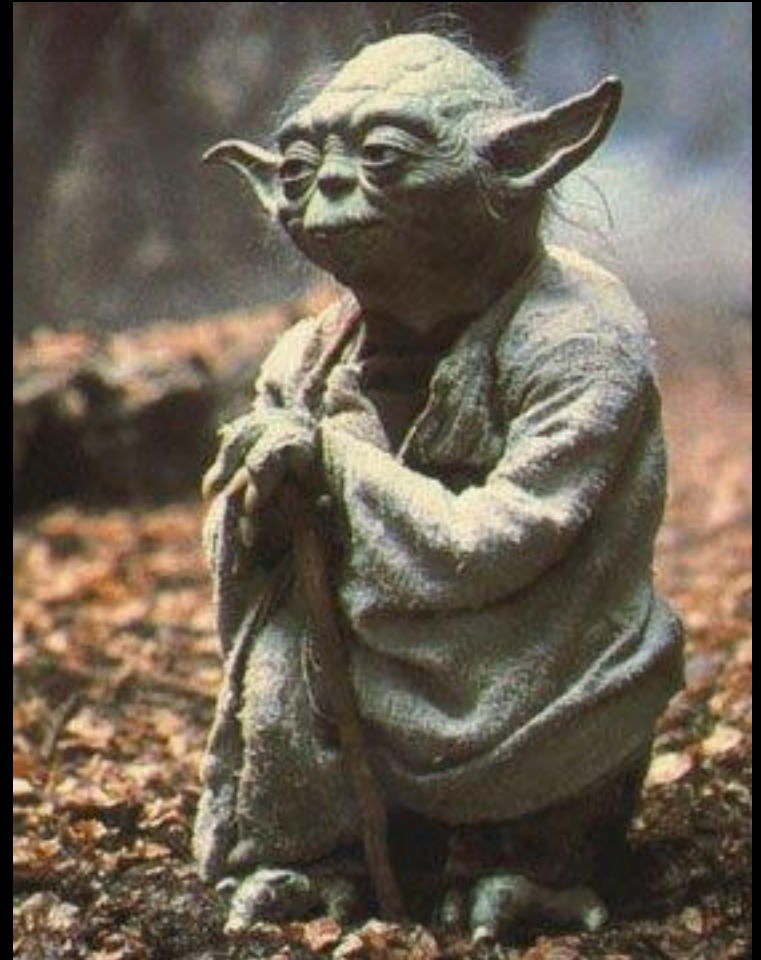
```
int numplus(int arr[], int n)
{
    if (n == 0) // here we use the branch
        return 0;
    return sign(arr[0]) + numplus(arr+1, n-1);
}
```

# Question 3



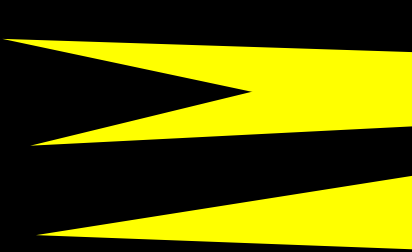
# Question 3

- Don't panic.
- Remember your training!
  - A recursive function is just another function.
  - Save `$ra` and any other registers.
- Use "assembly pseudocode" then convert to assembly



# Question 3

- Strategy:
  - Pop arguments into \$to, \$t1
  - Deal with base case
  - Save \$ra
  - `$t2 = sign(arr[o])`
  - `$t3 = numplus(...)`
  - Compute `$t2+$t3`
  - Restore \$ra
  - Return result



Which registers  
must be saved in  
each case?

# Question 3

- Pop arr, n
- Check for base case  $n == 0$ :
  - Return 0 if n is zero
- Save \$ra
- Push arr, n to save them
- Get arr[0]
- Call sign(arr[0])
- Pop result into \$t2
- Pop arr, n to restore them
- Push \$t2 to save it (it contains the result of sign(...))
- Call numplus(arr+4, n-1) ← note arr+4 (in C it would be +1)
- Pop return value into \$t3
- Pop \$t2 to restore it
- Restore \$ra
- Return  $t2 + t3$



# Week 11: Odds and Ends



# We're Almost Done

- We're pretty much done with Assembly!
  - Arithmetic and logical operations
  - Branches for loops and conditions
  - Memory
  - Functions
  - Stack
  - Calling conventions
- Today, just a few more odds and ends
  - But before that...

# Final Exam

- Worth 33% of grade.
- Get at least 30% of exam marks to pass course.
  - Doing that is quite easy.
- 3 hours (TBD), 5-6 questions.
- No big surprises: you would see pretty much the same kinds of questions you already saw in quizzes/labs/project.
- Watch this 3-minute video with good advice:  
<https://www.youtube.com/watch?v=OVxL36yYQMs>



# What to Bring

- **Arrive early**
- **Bring:**
  - Pen, pencil + eraser. Bring at least 2 or 3 for spare.
  - Your **T-card** or **government ID with photo**.
  - A bottle of water is fine.
- **Do not bring:**
  - No electronics like phone, smartwatch, calculator...
    - You will have to leave them in your bag at the front.
  - No material, books, summary sheets – closed book exam!
  - No paper of your own
    - There is sufficient space for drafts on exam booklet.
  - No hats (we will ask you to remove them!)
- UCheck red screen, sick, covid, etc.? **Do not come!**
  - Follow UCheck procedures, report to ACORN, ask for deferral.

# Exam Covers

- Everything.
  - Anything in pre-recorded lectures or review lectures.
- Types of questions?
  - Anything you saw in past quizzes or reviews.
  - Conceptual questions.
  - “What happens if” questions
  - Do or design something questions.
  - Write program
  - Describe something

# Examples

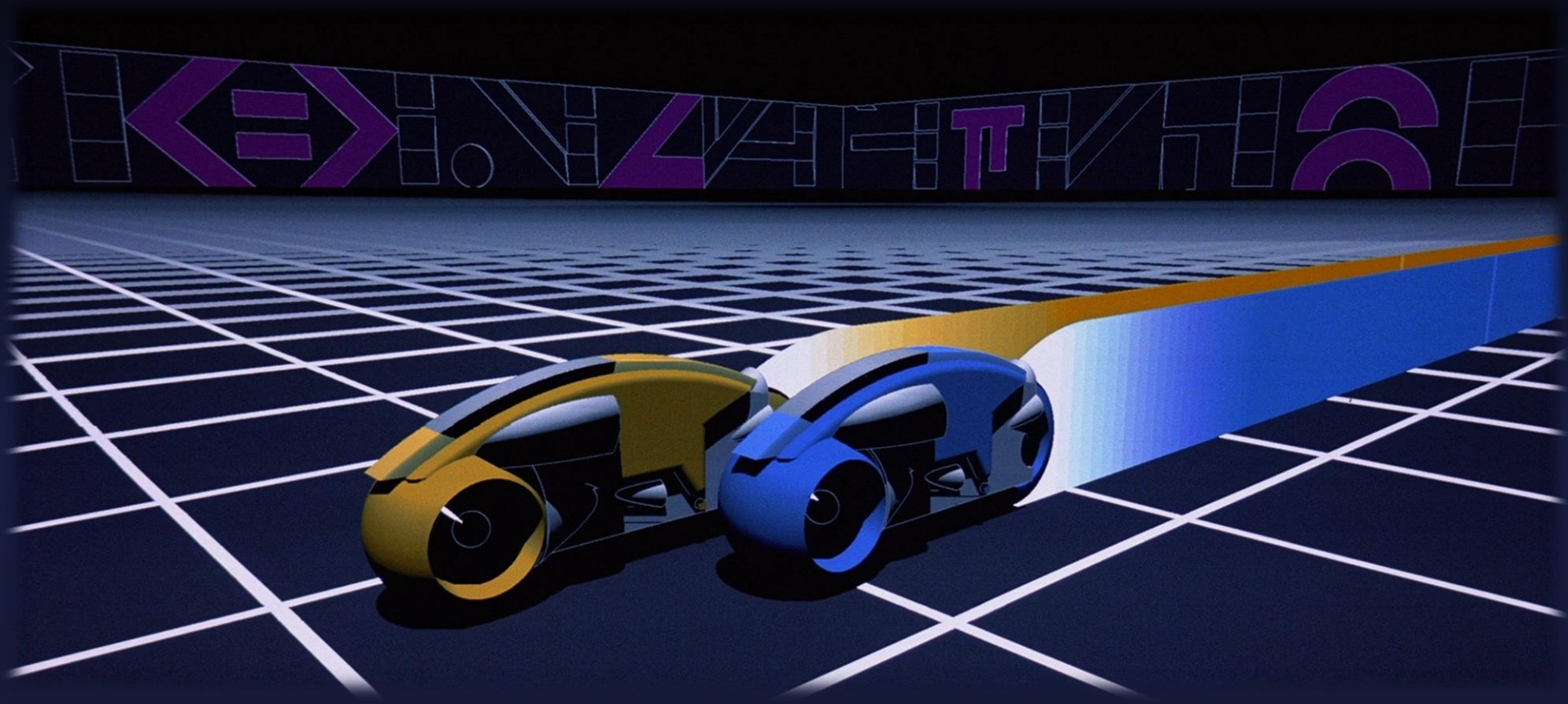
- Analyze a circuit (transistor/comb/sequential)
- Design a circuit
- Design FSM
- How to perform operations on processor
- Control signals
- Translate to/from assembly
- Write a program
- How much will this circuit cost / how fast is it?
- And more

# To Do Well

- **Study and practice:**
  - Review videos, slides, your notes
  - Review labs
  - Review quizzes
  - Solve old exams (will be released)
- **Pay attention, do the work:**
  - Don't just skim over slides and "yeah, I get it"
  - Re-solve questions in lectures, slides, quizzes
  - Test yourself: re-solve review questions and quizzes.
- **Develop processes for different kinds of problems:**
  - Programming in Assembly? Write pseudocode and translate
  - Circuit design? Build truth table then k-map.

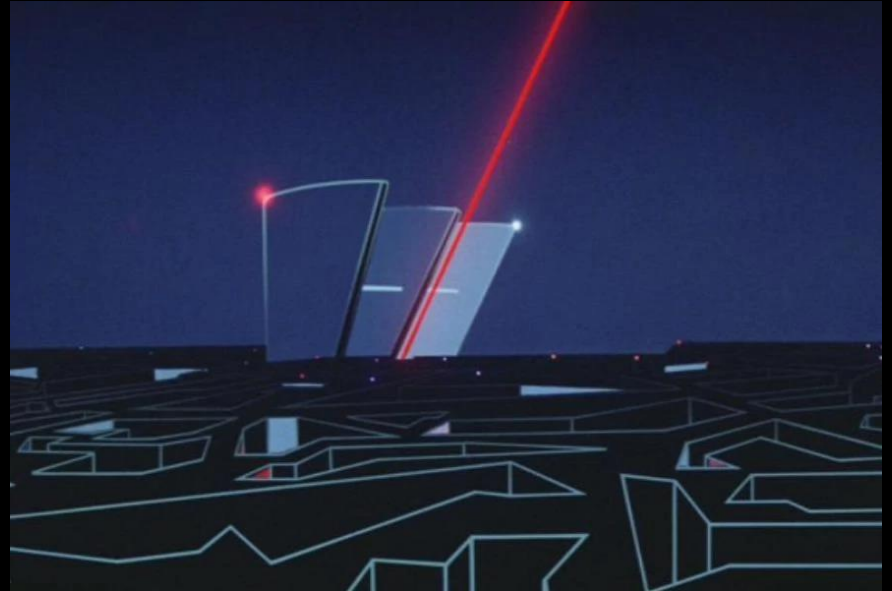


# Talking To Hardware



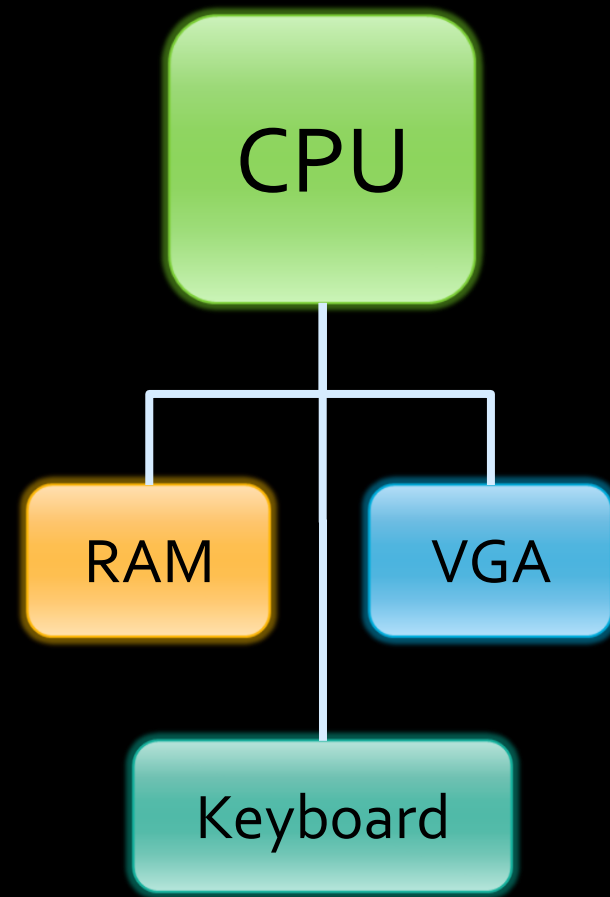
# Input and Output

- There is a world outside the CPU
  - Display
  - Hard drives, SSD
  - Keyboard, mouse
  - Network cards
  - ... and much more.
- How do we communicate with this hardware?
- How do we do I/O (Input/Output)?



# Memory Mapped I/O

- Certain memory addresses don't go to RAM.
- Instead, memory controller sends them to device registers.
  - Write to control a device.
  - Read to get data or device status.
- Often works with **polling**:
  - Example: to know if an operation is finished, we read memory in a loop until status is "finished".
  - `while (*status == 0) sleep(1);`



# Interrupts

- Alternative to polling.
- Devices **interrupt** the processor to signal important events
  - Operation completed, error, and so on.
- Interrupts are special signals that go from devices to the CPU.
- When an interrupt occurs, the CPU stops what it is doing and jumps to an **interrupt handler** routine
  - This routine handles the interrupt and returns to the original code.



# Exceptions

- An **exception** is like an interrupt that comes from **inside** the CPU.
  - Often because the program has an error.
  - Sometimes on purpose.
- The mechanism is similar, the **difference is semantic**.



# Traps



- A **trap** is an exception triggered not by an error, but deliberately by a **trap instructions**
  - **syscall** in MIPS
- Used to send system calls to the operating system
  - Interacting with the user, and exiting the program.
  - Service code goes in  $\$v0$
  - Arguments in  $\$a0$ ,  $\$a1$ , etc

# Examples

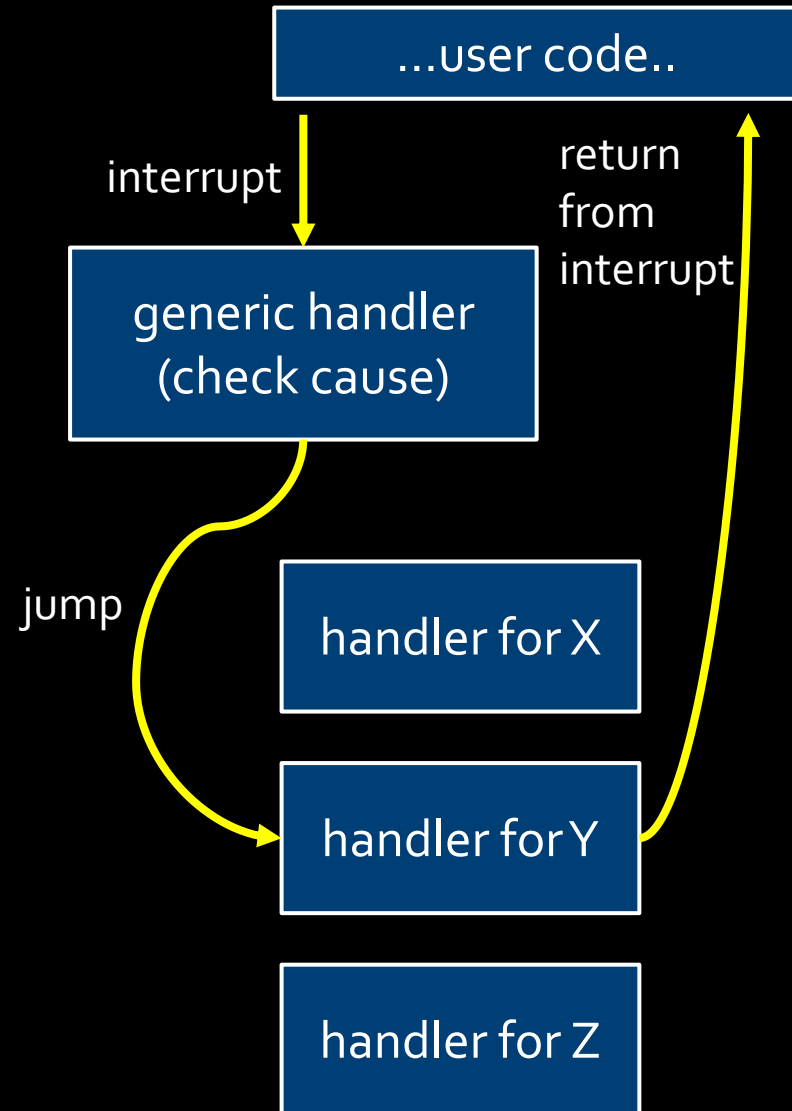
- Reasons for interrupts/exceptions:
    - Device I/O ← interrupt (external to CPU)
    - Invalid instruction (can't decode!)
    - Arithmetic overflow (add with overflow).
    - Divide by zero.
    - Unaligned access to memory
    - System calls ← traps (internal to CPU, deliberate)
- exceptions  
(internal to CPU,  
unexpected)

# The Interrupt Handler

- Just a piece of assembly code.
  - Almost like a function that can be called at **any time**.
  - Must not cause error (no one else to handle them...)
  - Must save and restore all registers.
- 1. **Determine the cause** using special registers.
- 2. **Handle** what needs to be handled:
  - Read/write from device.
  - Deal with memory issues.
  - Terminate program.
- 3. **Return** to original code.

# Polled Handling

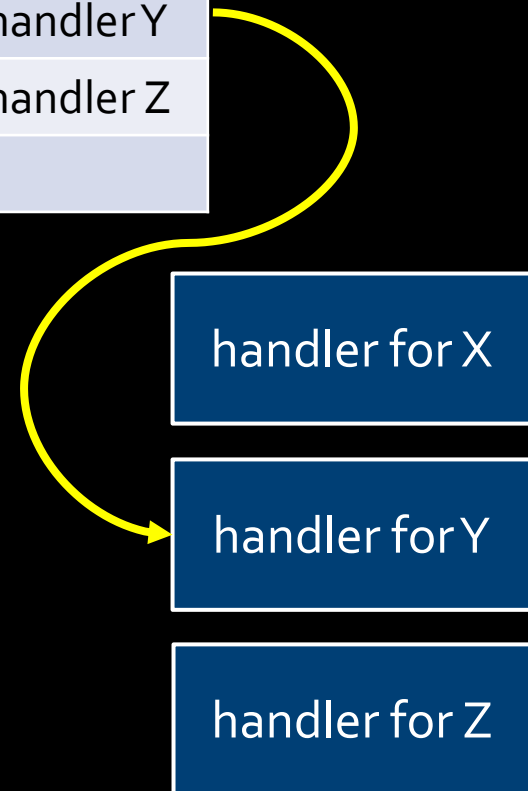
- (not related to previous “polling”)
- CPU branches to generic handler code for all exceptions.
- Handler checks the cause of the exception and branches to specific code depending on the type of exception.
- **This is what MIPS uses.**



# Vectored Handling

- Assign a unique id (number) for each device and interrupt/exception type (example from 0 to 255).
- Set up a table containing the address of the specific interrupt handler for every possible id.
- On interrupt with type X, the CPU gets the address from row X of the table and branches to the address.
- **This is what x86 uses.**

int #	hanlder address
...	...
53	handler X
54	handler Y
55	handler Z
...	



# MIPS Interrupt Handling

- MIPS has polled interrupt handling: jumps to exception handler code, based on the value in the **cause register** (not part of reg file).
- But what happens after?
- Depends on cause:
- If program can continue, we want to return to the original program. But how?
  - PC has been lost during the jump to exception handler
  - CPU stored original PC in EPC register. **rfe** instruction copies it back to PC.
- If program cannot continue, OS terminates it
  - Can dump stack to file or screen to help in debugging.

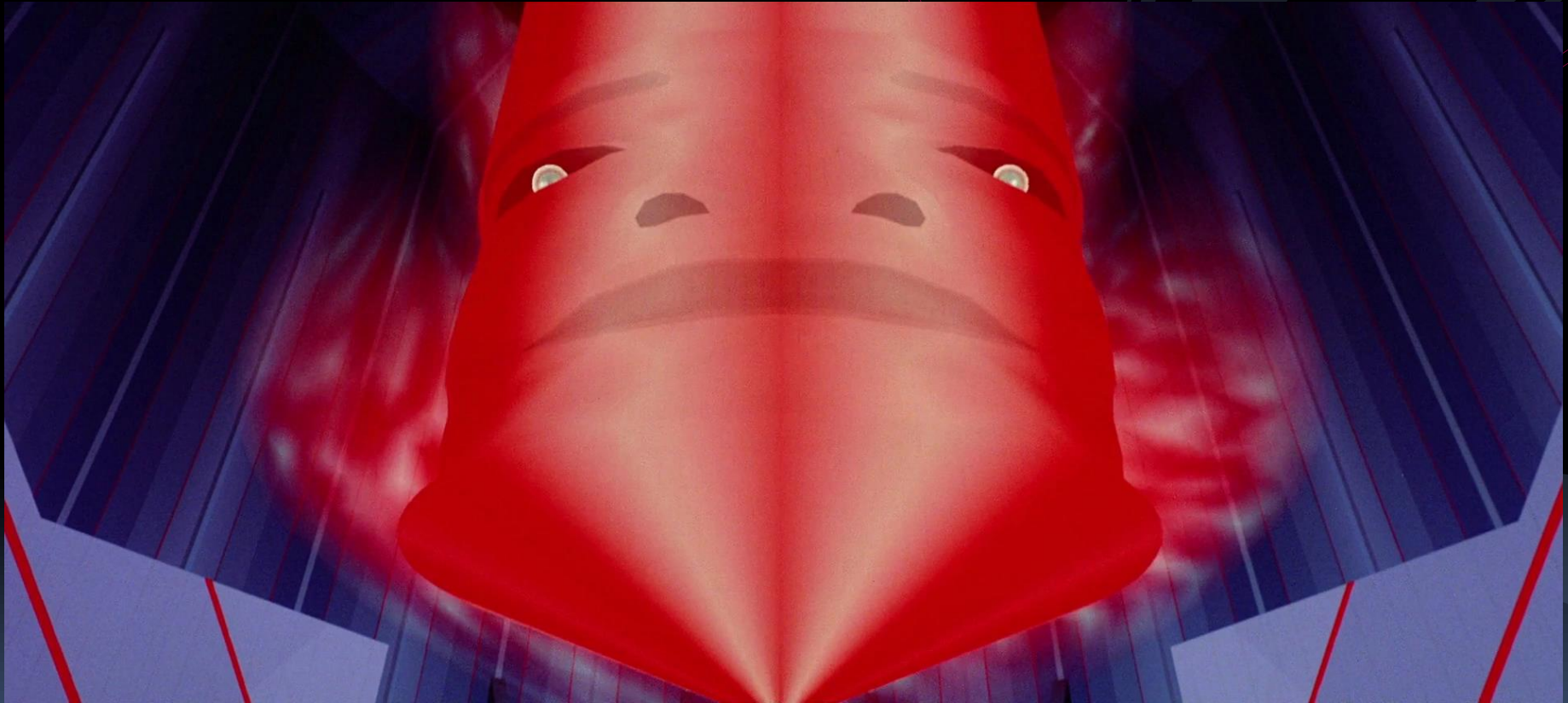
0 (INT)	external interrupt.
4 (ADDRL)	address error exception (load or fetch)
5 (ADDRS)	address error exception (store).
6 (IBUS)	bus error on instruction fetch.
7 (DBUS)	bus error on data fetch
8 (Syscall)	Syscall exception
9 (BKPT)	Breakpoint exception
10 (RI)	Reserved Instruction exception
12 (OVF)	Arithmetic overflow exception

# Coordination

- Talking with hardware is a lot of work.
  - What if you change your hardware, do you need to change every program?
  - Should we duplicate code (e.g., for handling keyboard) in every program that needs it?
    - In the older days of DOS → **yes!** And it was hard.
  - Who will manage all the different programs on the computer and offer them I/O services?
- We need some sort of **master control program** to coordinate all this...



# ■ The Operating System



# The Operating System

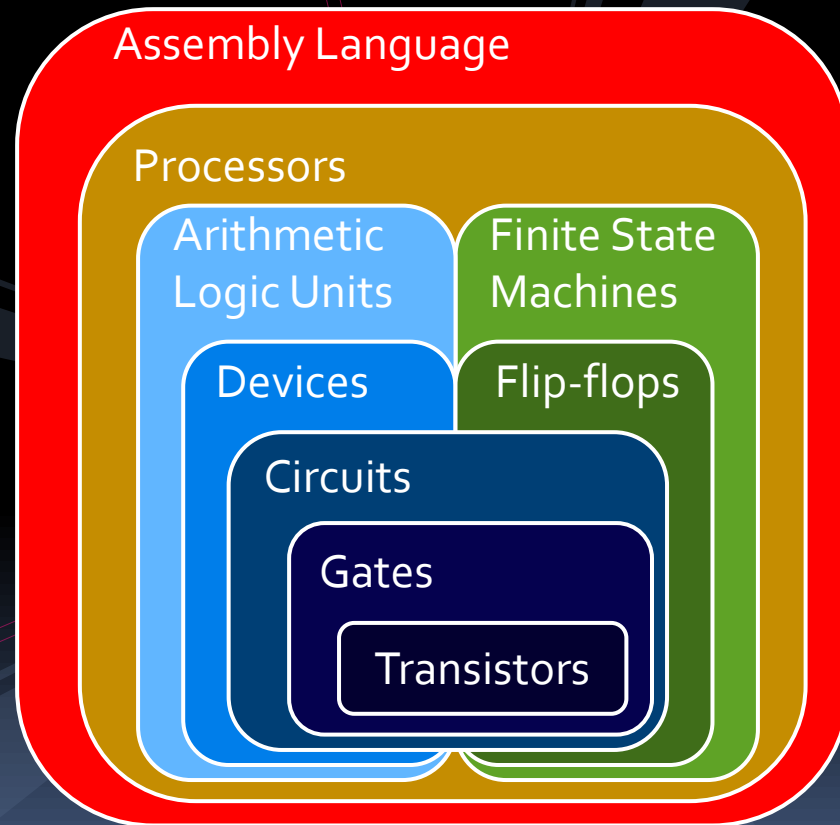
- The **operating system** is the program that manages all the other programs.
  - Loads, runs, and stops programs.
  - Coordinates multiple programs simultaneously.
  - Abstracts hardware and I/O, offers services.
- Programs invoke the OS to:
  - Read/write from files.
  - Write to screen.
  - Run other programs
  - More...
- Invoking OS **system calls** is done via **traps**.



Learn more  
in C69

# ■ We've Covered So Much...

Started at the bottom, now at the top!





# ...and there's So Much More!

## Processor Hardware

- Caches
- Floating point unit (FPU)
- Pipelining
- Superscalar CPUs
- Out-of-order execution
- Register renaming
- VLIW architectures
- Multicore

## Systems Software

- Operating systems
- Compilers
- Optimization techniques
- Automatic parallelization
- Virtual machines
- Concurrency
- Data structures
- Cache-obliviousness

# We Are Done!

You can build and  
program computers!

You understand how  
they work.

There is no magic  
here, except the  
magic of engineering.



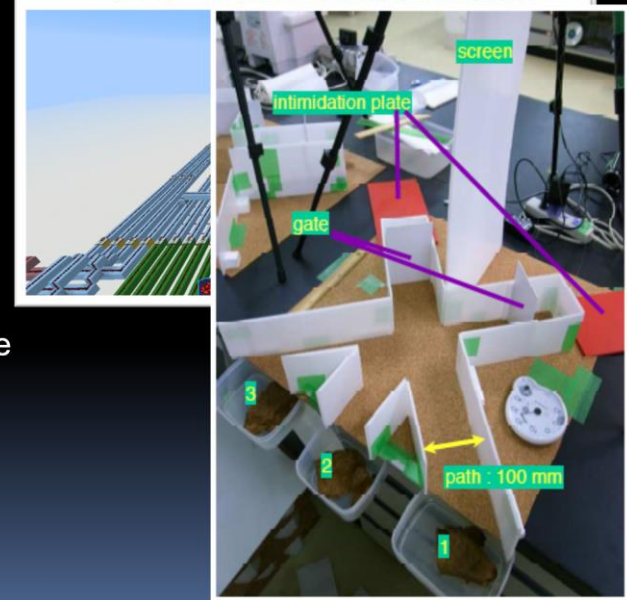
# One More Thing

- Will we even need to build a computer?
  - Maybe!
- Remember crab-puter from Wo1 intro lecture?
  - Ah, memories...

## Build a Computer!

- ...from LEGO
- ...in Minecraft
- ...from living crabs\*

\* Do not build a computer from live crabs. It is unethical.





# One Last Thing

- Thank your TAs!
  - They've been working hard behind the scenes.
- Calling future Tas!
  - Want to TA B58?
  - Be the change you want to see.
  - You can help improve/shape the future of the course.
  - Ask your current TAs what they think!
- Course Evaluations
  - They are anonymous.
  - I do actually read them.
  - I do actually care.
  - They do actually make an impact.
  - No, I won't bribe you





Now We Are Done

Good luck and stay safe!

**Thank you all!**