



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
 - **Caller** pushes arguments to stack in order A,B,C.
 - **Callee** pops arguments (in order C, B, A).
- 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:
 - **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

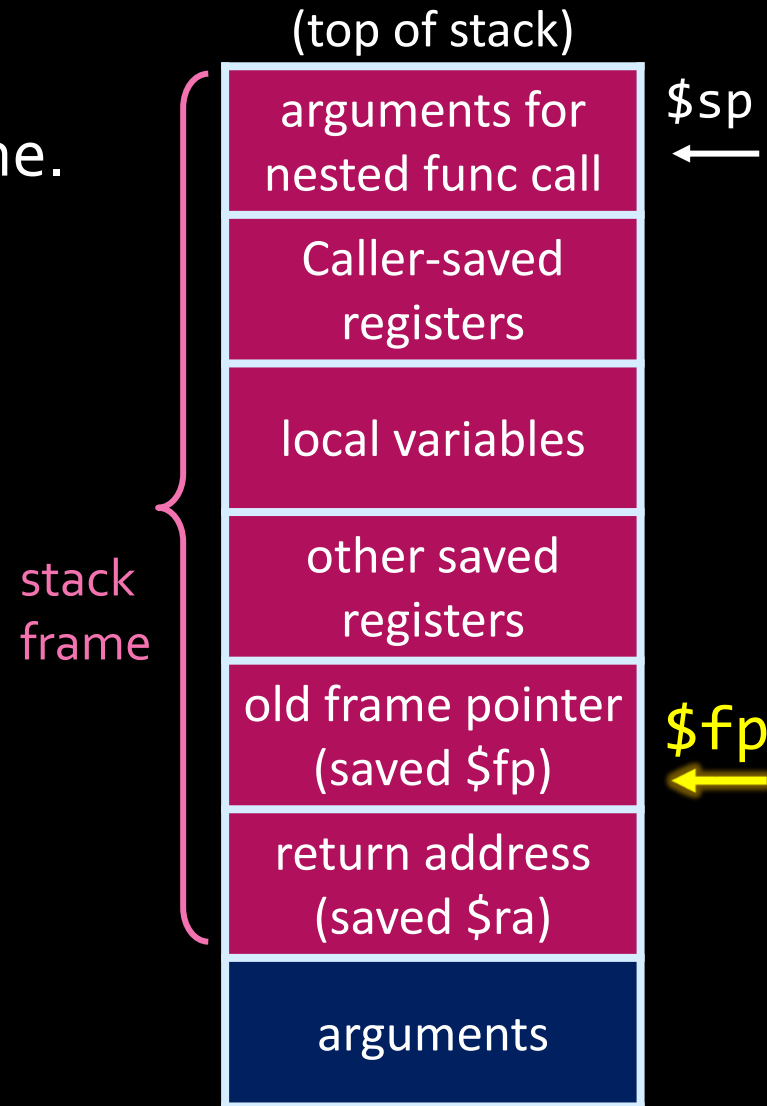
- Registers \$to-\$t9 are **caller-saved**
 - If the **caller** needs their values, save them.
 - Push before calling a function (before arguments)
 - Pop when function returns (after popping ret. val).
 - Also \$a0-\$a3, \$v0, \$v1
- Registers \$s0-\$s7 are **callee-saved**
 - If a **callee** uses them, 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`





Some Practice Questions

Question 1

The following shows the values stored in memory:

Address:	0x5000	0x5002	0x5004	0x5008
Value:	0xB080	0x80B0	0x0000	0x0020

- Is this Little Endian or Big Endian?

Can't tell just from this

Question 2

The following shows the values stored in memory:

Address:	0x5000	0x5002	0x5004	0x5008
Value:	0xB080	0x80B0	0x0000	0x0020

What is the value in \$t1 after the following operation?

```
addi $t0, $zero, 0x5002
lh $t1, 0($t0)
```

- We are loading the 16-bit (half-word) value **0x80B0**
 - **1000000010110000**
- As 16-bit signed, this is a **negative** value (MSB=1)
- lh loads a **signed** 16-bit value (otherwise use lhu)
- So do sign extension: **0xFFFF80B0**

Question 3

- You are the lead architect at a major chip design company
- We want lots and lots of memory
- In fact, we want 256 GB of memory
- Can we do this with MIPS?

Question 3

- You are the lead architect at a major chip design company
- We want lots and lots of memory
- In fact, we want 256 GB of memory
- Can we do this with MIPS?

No!

MIPS is a 32-bit architecture

We can only address 2^{32} bytes = 4GB

But what if we got creative?

Question 3

- You are the lead architect at a major chip design company
- We want lots and lots of memory
- In fact, we want 256 GB of memory
- Can we do this with MIPS?

We could use something different than *byte addressable*

We could use other types of addressing, other than base + offset (e.g., concatenate?)

Question 4

- We want to make our process really fast. Which of the following will likely help and why?
- Use registers instead of stack?
 - Registers are much faster to access, since stack is in memory
- Use function calls?
 - Function calls can lead to overheads from register saving and branch instructions
 - We should avoid using them for small functions

Question 5

- We want to make our processor really fast. Which of the following will likely help and why?
- Have more main memory?
 - No ☹️ Usually makes it slower to access memory
- Add more registers?
 - Yes! Then maybe we can use the stack less
- Instruction set with many more instructions?
 - Not enough information

Question 6

Greater Common Divisor can be implemented using recursion
(where `x % y` is the remainder of `x / y` (known as "modulo"))

```
def gcd(x,y):  
    if y == 0:  
        return x  
    else:  
        return gcd(y, x % y)
```

- What are in AA and BB?
- AA should have `$t0` and BB should have `$ra`

gcd:

```
lw $t1, 0($sp)  
addi $sp, $sp, 4  
lw $t0, 0($sp)  
addi $sp, $sp, 4  
  
bne $t1, $zero, recurse  
addi $sp, $sp, -4  
sw $AA, 0($sp)  
jr $ra
```

recurse:

```
addi $sp, $sp, -4  
sw $ra, 0($sp)  
  
div $t0, $t1  
mfhi $t2  
  
addi $sp, $sp, -4  
sw $t1, 0($sp)  
addi $sp, $sp, -4  
sw $t2, 0($sp)  
  
jal gcd  
  
lw $t0, 0($sp)  
addi $sp, $sp, 4  
  
lw $BB, 0($sp)  
addi $sp, $sp, 4  
  
addi $sp, $sp, -4  
sw $t0, 0($sp)  
  
jr $ra
```




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, 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 **lectures** or **reviews**.
- 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

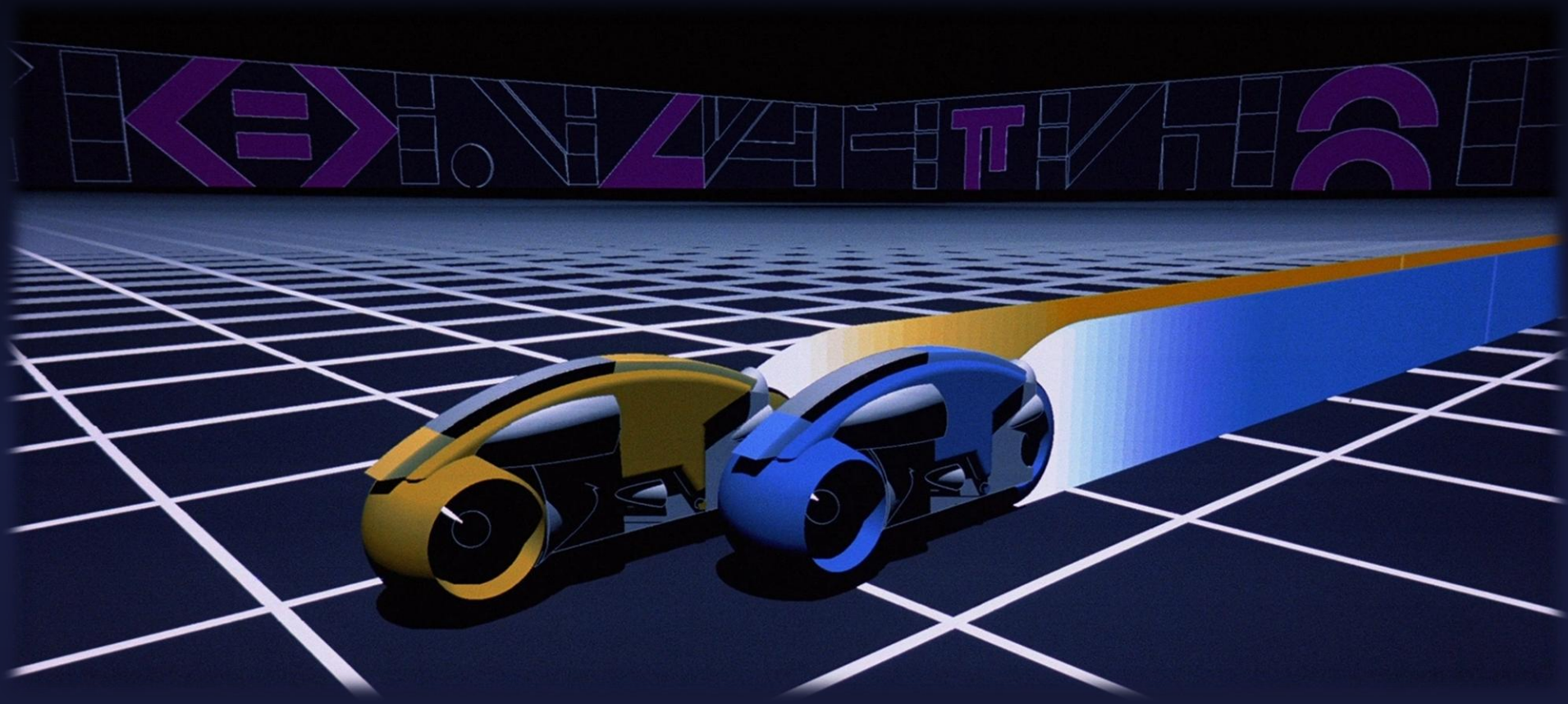
- Analyze a circuit (transistor/comb/sequential)
- Design a circuit
- Design FSM
- How to perform operations on processor
- Control signals
- Translate to/from assembly
- Write a program
- How much will this circuit cost / how fast is it?
- And more

To Do Well

- **Study and practice:**
 - Review videos, slides, your notes
 - Review labs
 - Review quizzes
 - Solve old exams
- **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.

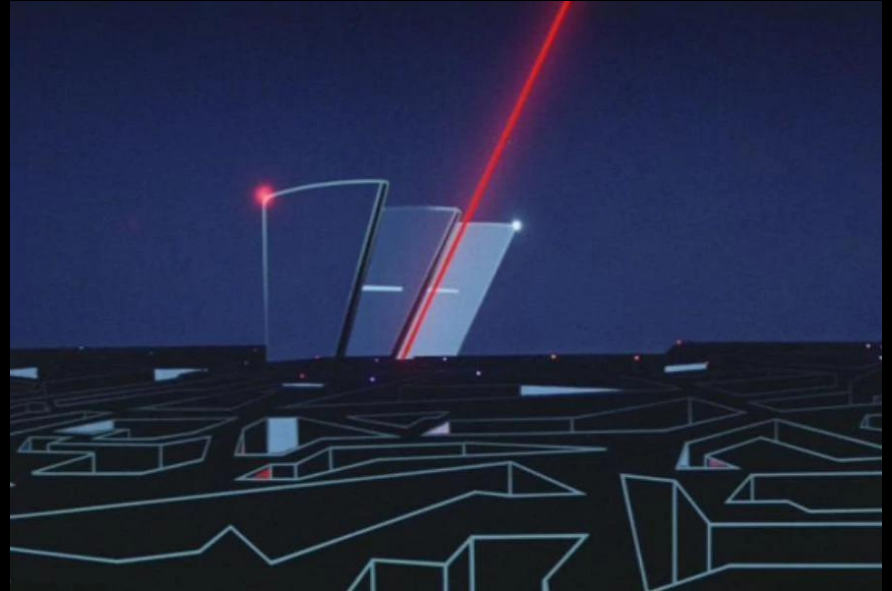


Talking To Hardware



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)?



How to we communicate with I/O devices?

Let's learn some concepts!

- Events
- Memory-mapped I/O
- Polling vs. Interrupts

When do we interact with an I/O device?

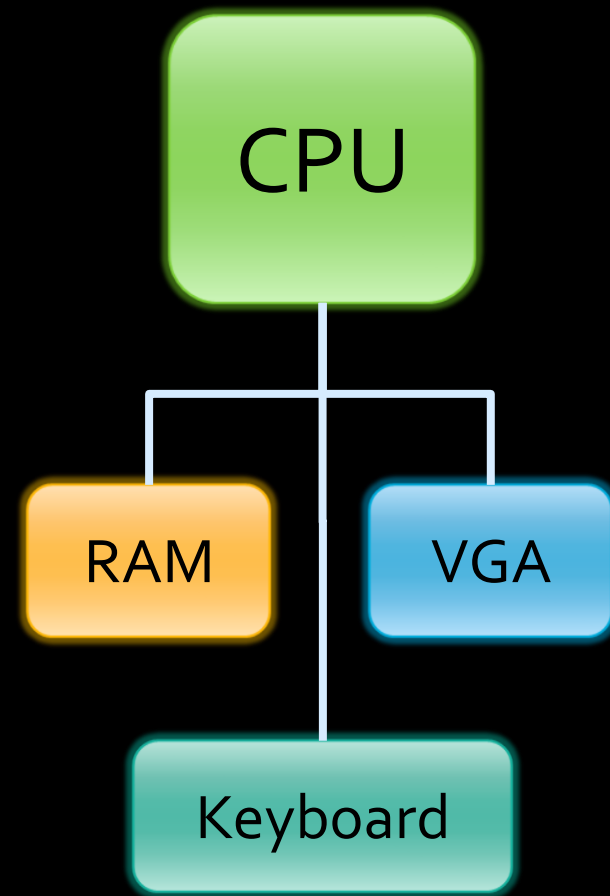
- When an "event" occurs
- An "event" is something that happens, and the system or program may need to respond to it
 - A mouse button click
 - A mouse move
 - A keyboard press
 - A timer firing
 - A network message arriving
 - A disk read/write completing
 - An error in the program!

When do we interact with an I/O device?

- When an "event" occurs
- An "event" is something that happens, and the system or program may need to respond to it
 - A mouse button click
 - A mouse move
 - A keyboard press
 - A timer firing
 - A network message arriving
 - A disk read/write completing
 - An error in the program!

Memory Mapped I/O

- How can we communicate with devices?
- We could use special registers or memory in the devices themselves!
- We can still use memory addresses to read/write from them
- These 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.



Polling

- We can check if a certain event has occurred by “polling” a special memory-mapped register
- For example, if a key was pressed
- We read memory in a loop until status has changed
 - `while (*status == 0) sleep(1);`
- Once an event has occurred (status of register has changed), we can react to the situation

Downside of polling?

- The program is stuck in a loop
- A lot of wasted processing cycles
- What could we do instead?

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

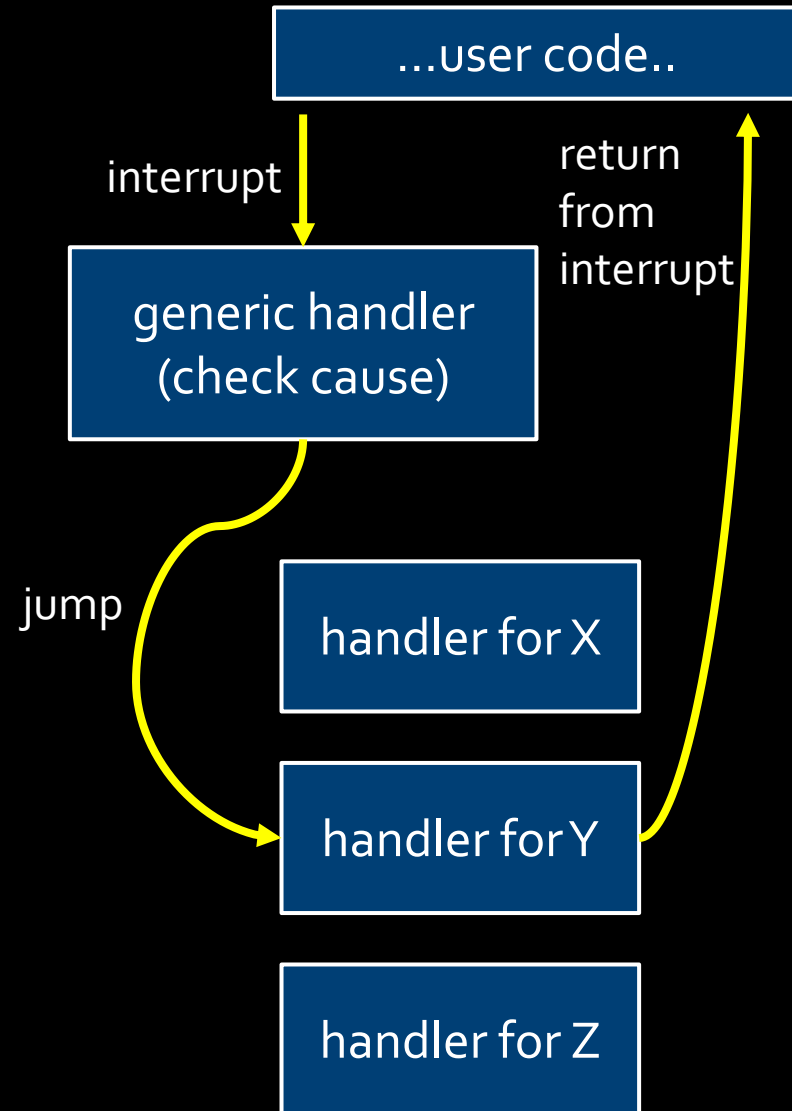
- Reasons for interrupts/exceptions:
 - Device I/O ← interrupt (external to CPU)
 - Invalid instruction (can't decode!)
 - Arithmetic overflow (add with overflow).
 - Divide by zero.
 - Unaligned access to memory
 - System calls ← traps (internal to CPU, deliberate)
- exceptions
(internal to CPU,
unexpected)

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

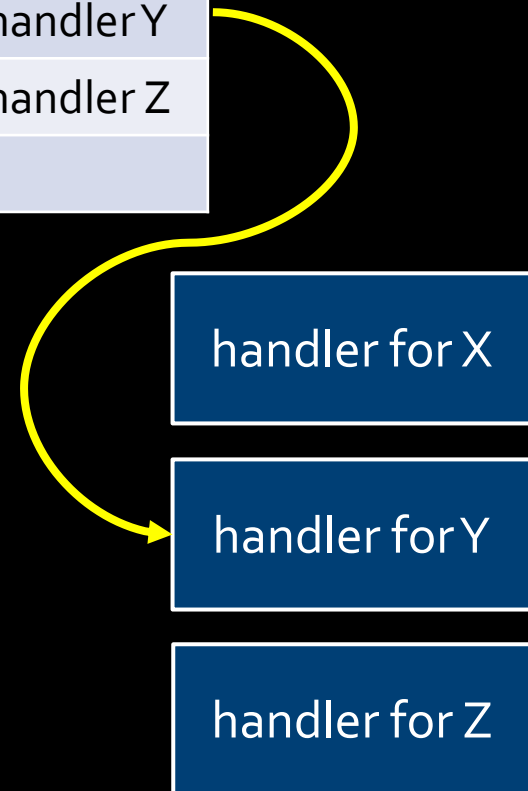
- (not related to previous “polling”)
- CPU branches to generic handler code for all exceptions.
- Handler checks the cause of the exception and branches to specific code depending on the type of exception.
- **This is what MIPS uses.**



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



MIPS Interrupt Handling

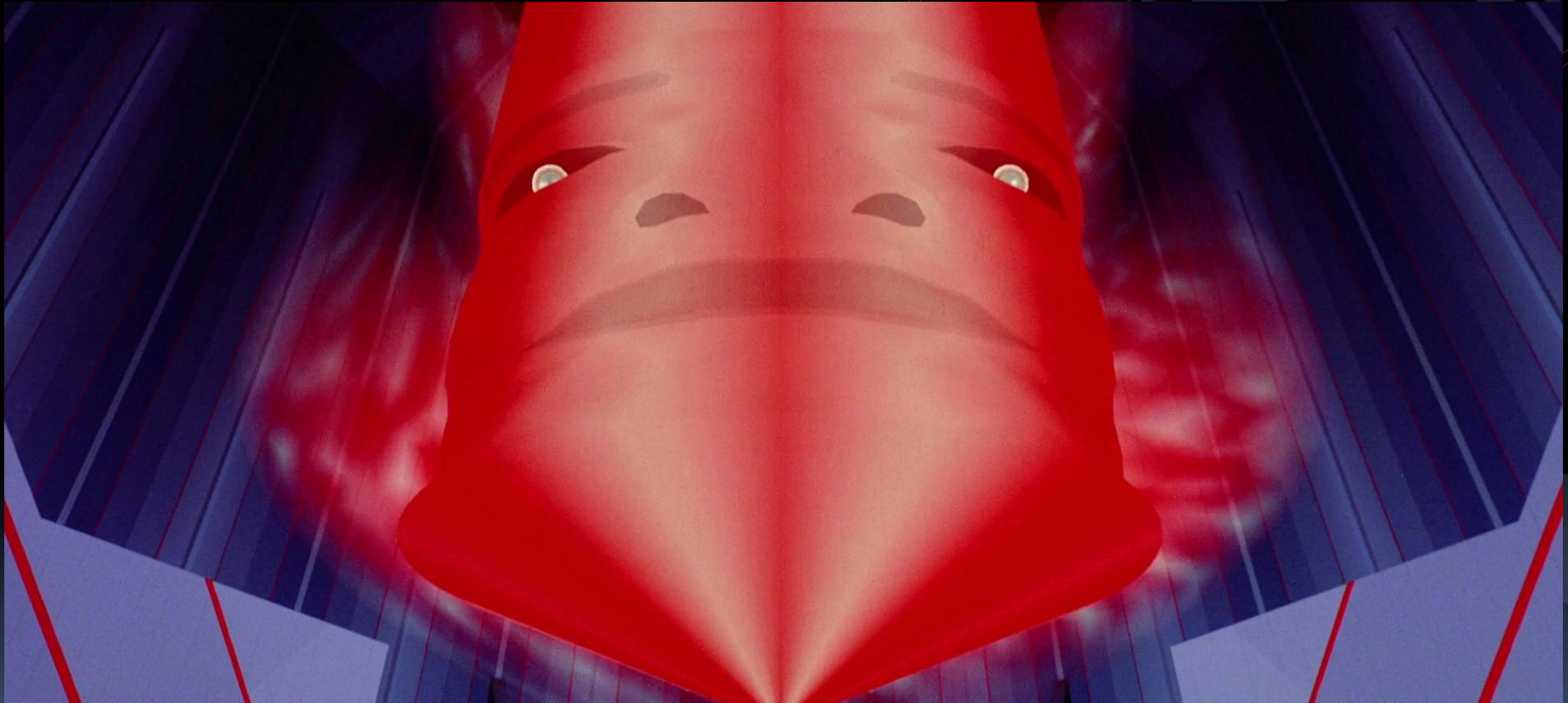
- MIPS has polled interrupt handling: jumps to exception handler code, based on the value in the **cause register** (not part of reg file).
- But what happens after?
- Depends on cause:
- If program can continue, we want to return to the original program. But how?
 - PC has been lost during the jump 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

Caches

When the CPU misses in the cache

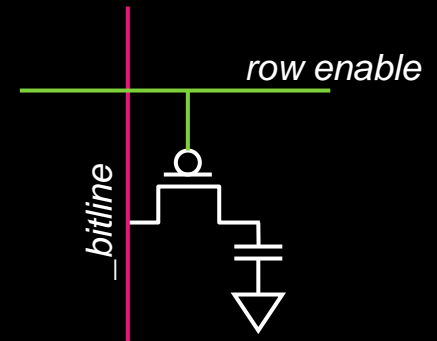


Summary: Memory vs Registers

- Memory houses most of the data values being used by a program.
 - And the program instructions themselves!
- Registers are for local / temporary data stores, meant to be used to execute an instruction.
 - Registers are can host memory between instructions (like scrap paper for a calculation).
 - Some have special purpose or used to control execution, like the stack pointer register

Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
 - Whether the capacitor is charged or discharged indicates storage of 1 or 0
 - 1 capacitor
 - 1 access transistor
- Capacitor keeps leaking
 - DRAM cell loses charge over time
 - DRAM cell needs to be refreshed



DRAM – Dynamic Random Access Memory

Array of Values

3455434
43543
98734
0
847
42
873909
1729

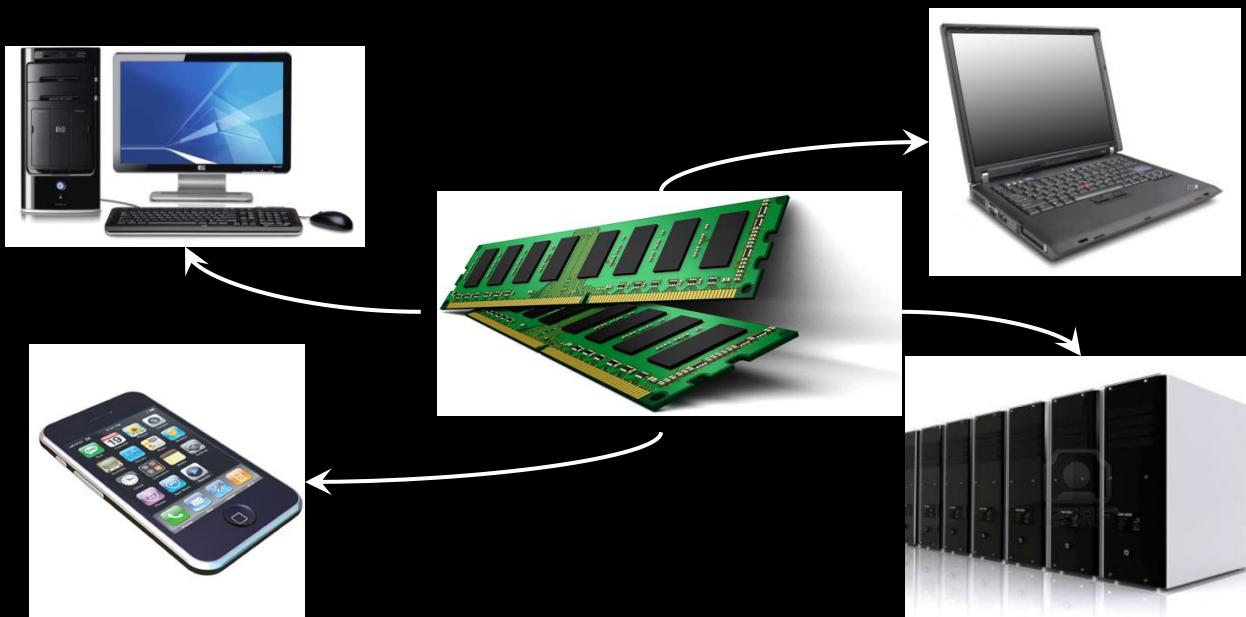
READ *address*

WRITE *address, value*

Accessing any location takes the same amount of time

Data needs to be constantly refreshed

DRAM in Today's Systems

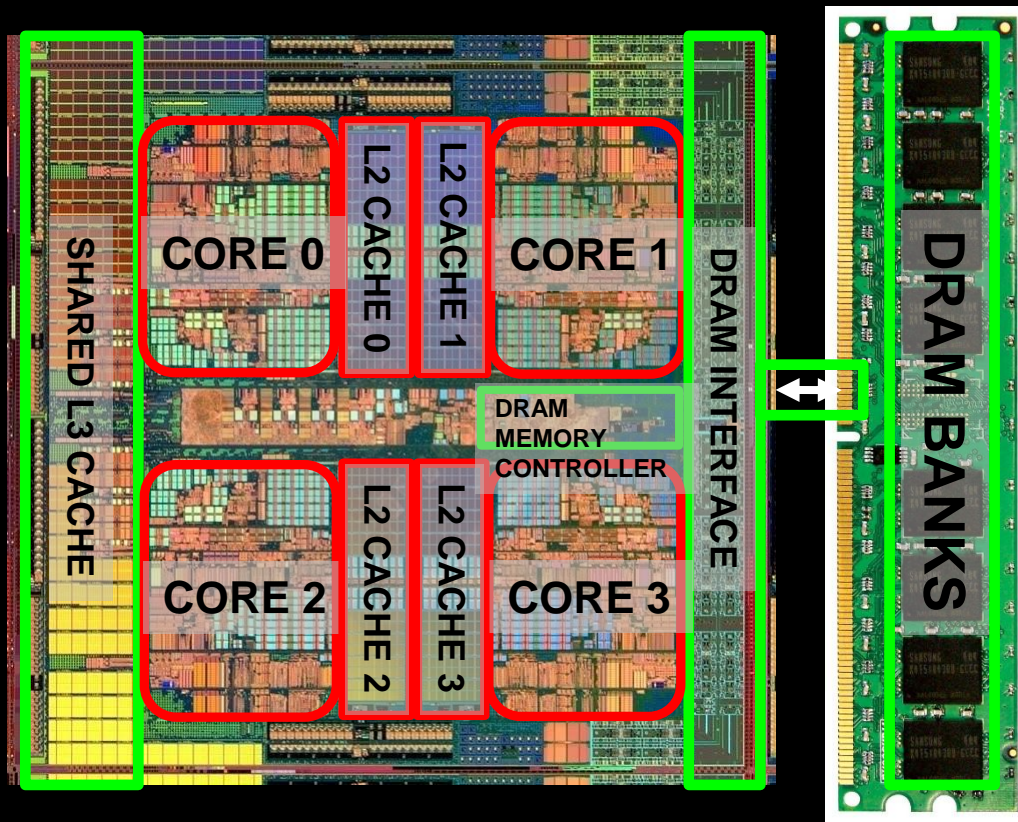


Why DRAM? Why not some other memory?

Another technology: SRAM

- SRAM:
 - ~6 transistors
 - Retains data bits in its memory as long as power is being supplied
 - Used in caches (holds a small amount of data)
 - Faster access times and more expensive
- Comparatively, DRAM has:
 - 1 transistor + 1 capacitor
 - Must be periodically refreshed to retain their data which increases the power usage
 - Used in main memory (holds much more data)
 - Slower access times and cheaper

Memory in a Modern System



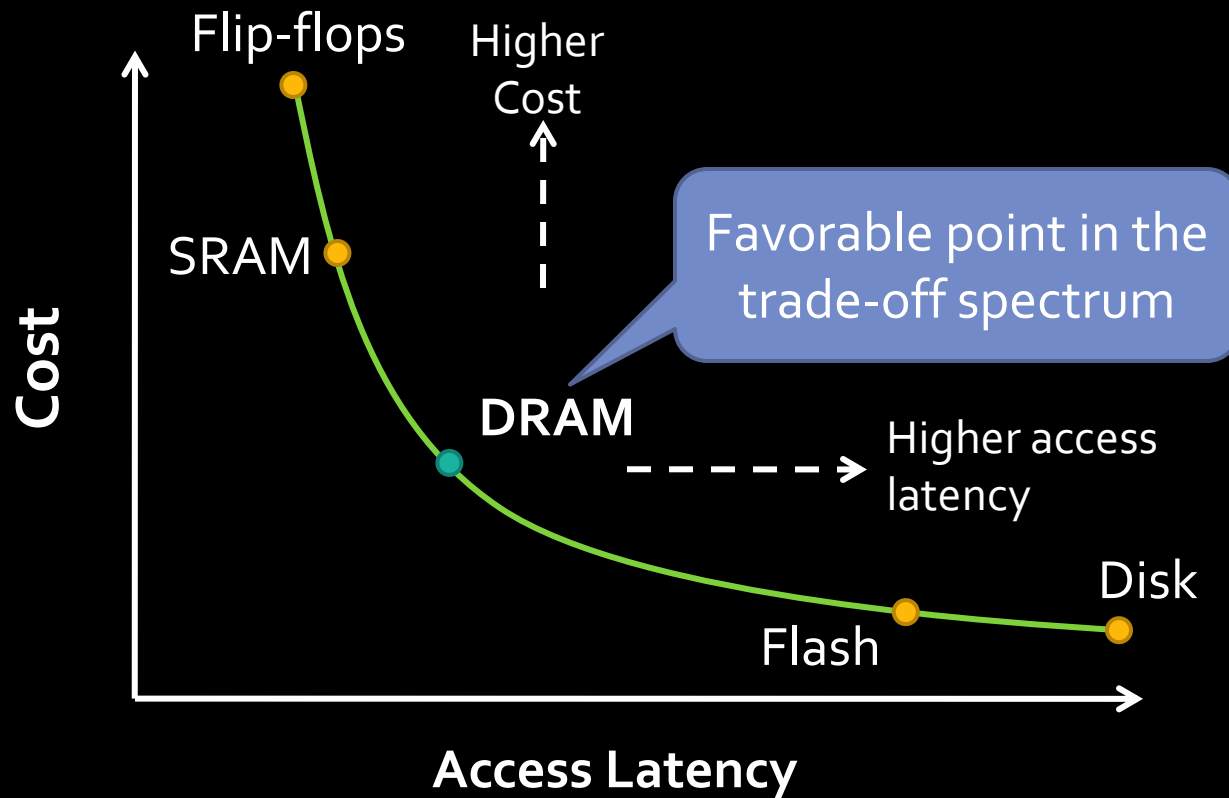
Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
 - Bigger → Takes longer to determine the location
- Faster is more expensive
 - Memory technology: SRAM vs. DRAM
- Higher bandwidth is more expensive
 - Need more banks, more ports, higher frequency, or faster technology

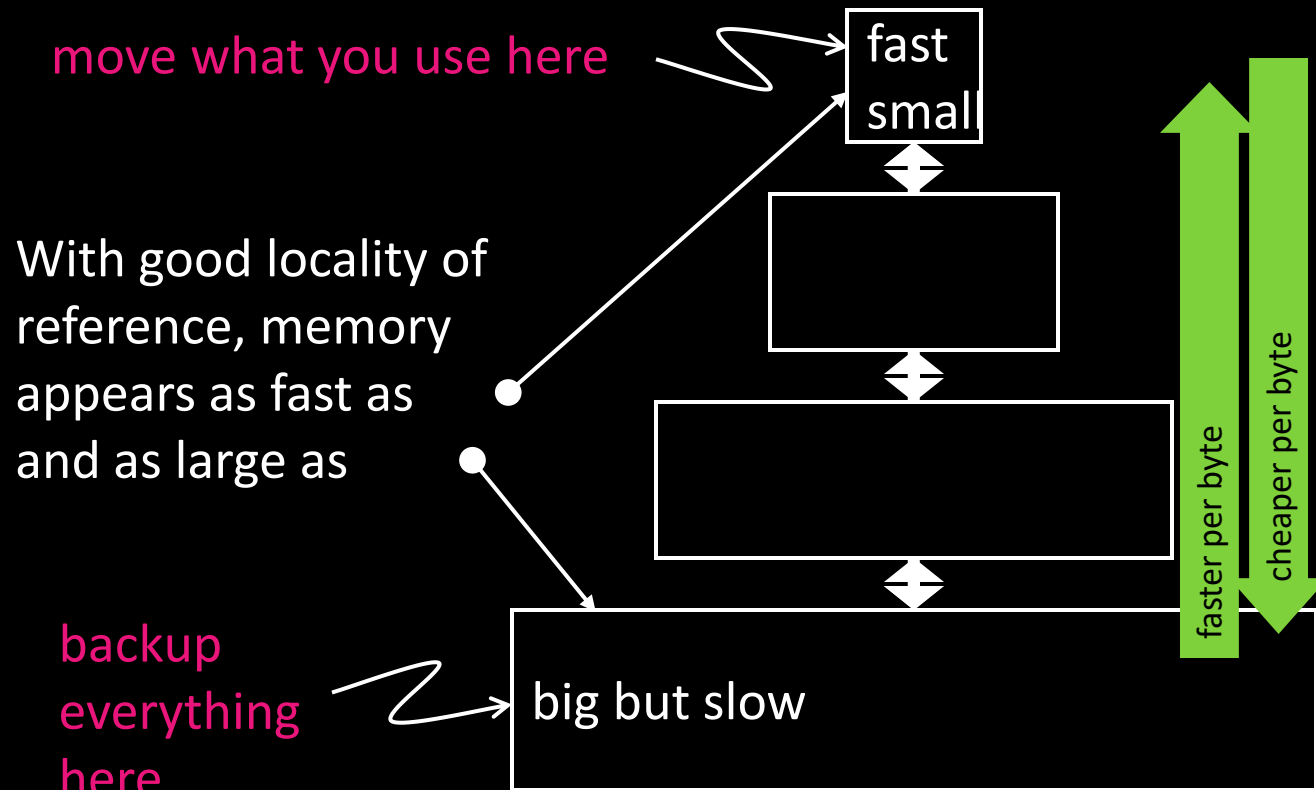
Why DRAM?



The Memory Hierarchy

- We want both fast and large
- But we cannot achieve both with a single level of memory
- Idea: **Have multiple levels of storage** (progressively bigger and slower as the levels are farther from the processor) and **ensure most of the data the processor needs is kept in the fast(er) level(s)**

The Memory Hierarchy



Memory Hierarchy as Food

(In terms of access speed)

- **Registers**: food in your mouth, ready for chewing
- **Cache**: food on your plate
- **Memory**: food in your fridge
- Hard disk: grocery store down the street
- Network: the farm

Locality

- One's recent past is a very good predictor of his/her near future.
- Temporal Locality: If you just did something, it is very likely that you will do the same thing again soon
 - since you are here today, there is a good chance you will be here again and again regularly
- Spatial Locality: If you did something, it is very likely you will do something similar/related (in space)
 - every time I find you in this room, you are probably sitting close to the same people

Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast
- Anticipation: the data will be accessed again soon
- Temporal locality principle
 - ▣ Recently accessed data will be again accessed in the near future

Caching Basics: Exploit Spatial Locality

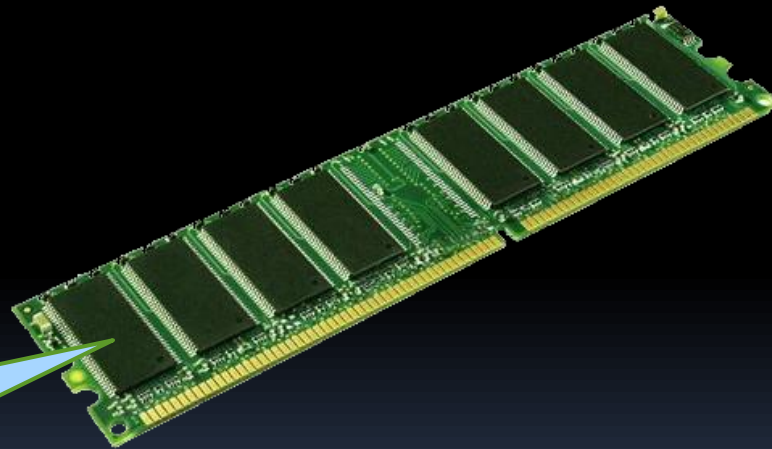
- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
 - Logically divide memory into equal size blocks
 - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon
- Spatial locality principle
 - Nearby data in memory will be accessed in the near future
 - E.g., sequential instruction access, array traversal

Examples of Locality

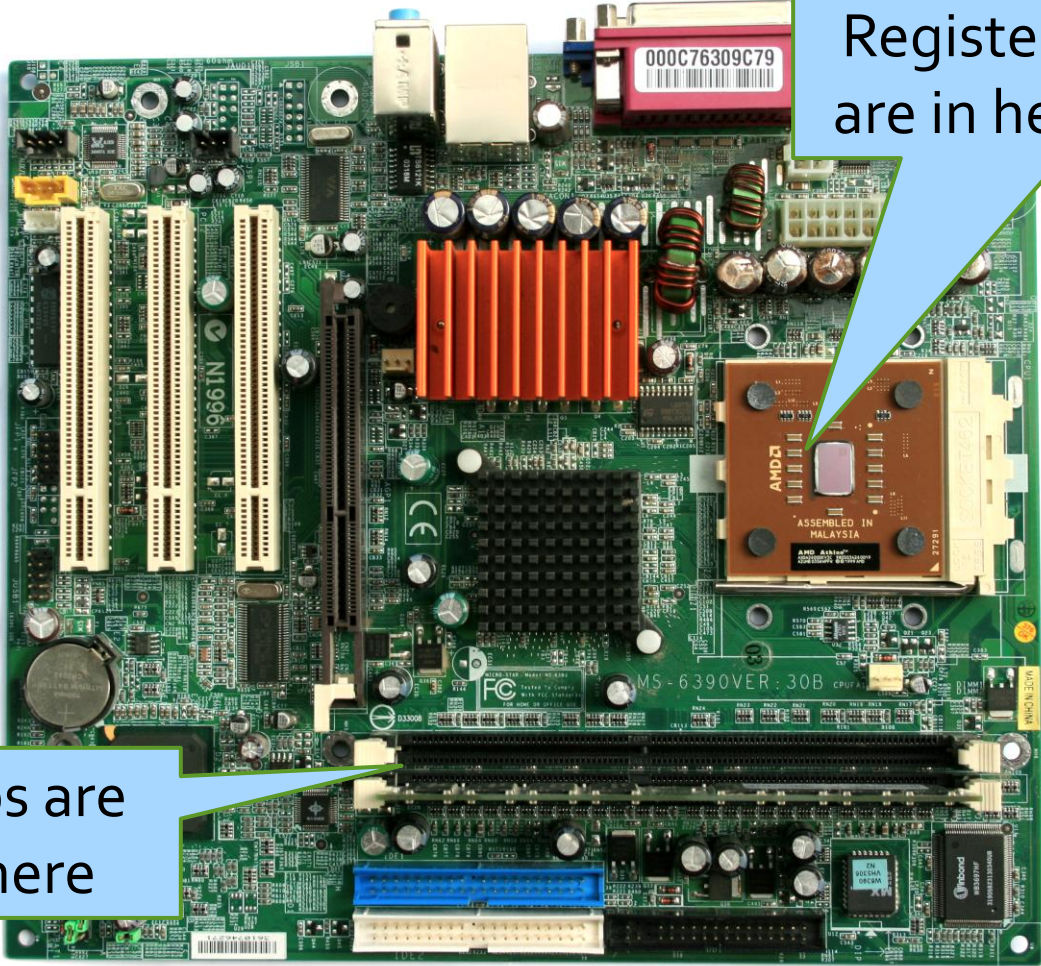
- “Iterating over an array” exhibits both temporal and spatial locality.
- “Executing code” often exhibits temporal and spatial locality.
- “Accessing items from a dictionary” does not: the items in the dictionary may not be close to each other in memory.
- Linked lists and other dynamically allocated structures can also cause locality problems.



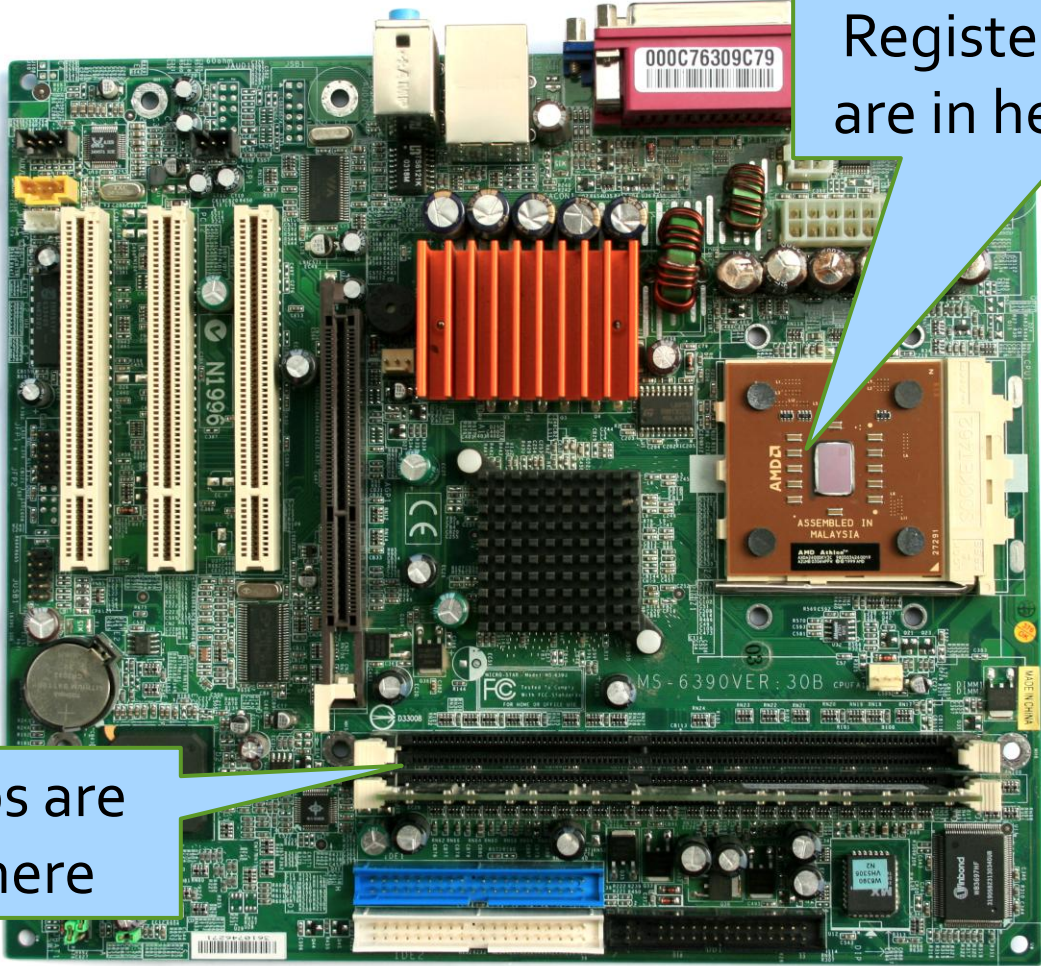
Registers and
caches are in here



Memory is in here



Registers and caches
are in here in the CPU



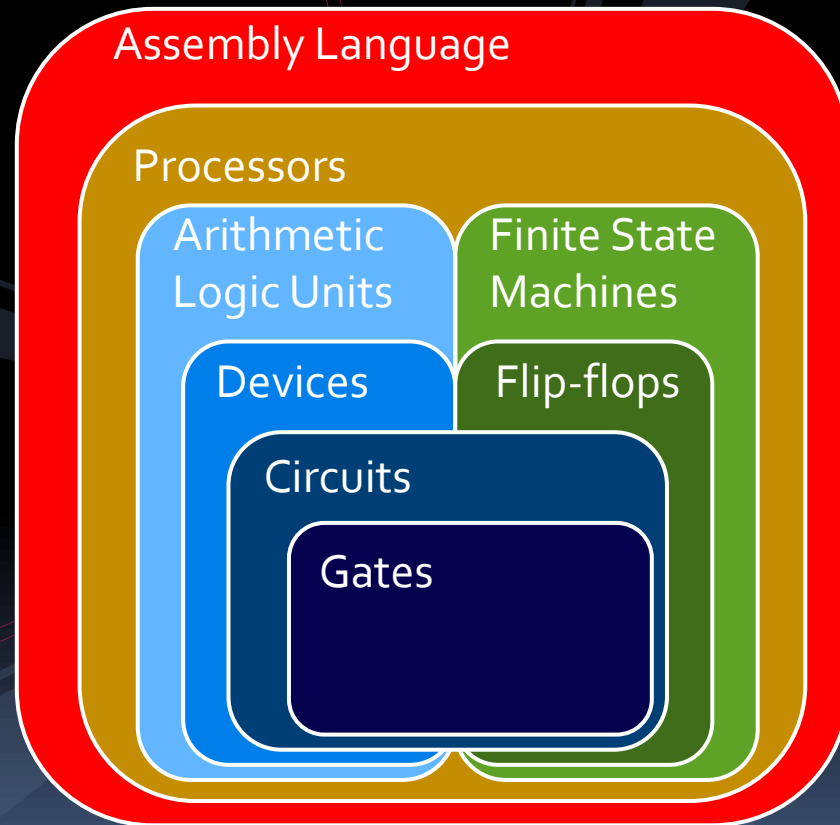
Memory chips are
plugged in here

Examples of Locality

- “Iterating over an array” exhibits both temporal and spatial locality.
- “Executing code” often exhibits temporal and spatial locality.
- “Accessing items from a dictionary” does not: the items in the dictionary may not be close to each other in memory.
- Linked lists and other dynamically allocated structures can also cause locality problems.

■ 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 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.



Research as an UG?

- Send me an email if you are interested in my areas of research (www.embarclab.com)
- Choose a semester for research when can spend >20 hours/week
- Typically, penultimate year before you graduate is a good time to start