CSC258 Winter 2016
Lecture 12

Announcements

- Missed Lab 10 last Thursday will be made up this Thursday
- If for a valid reason you cannot make it to Thursday, let me know.

```
int factorial (int n) {
    if (n == 0)
      return 1;
    else
      return n * factorial(n-1);
}
```

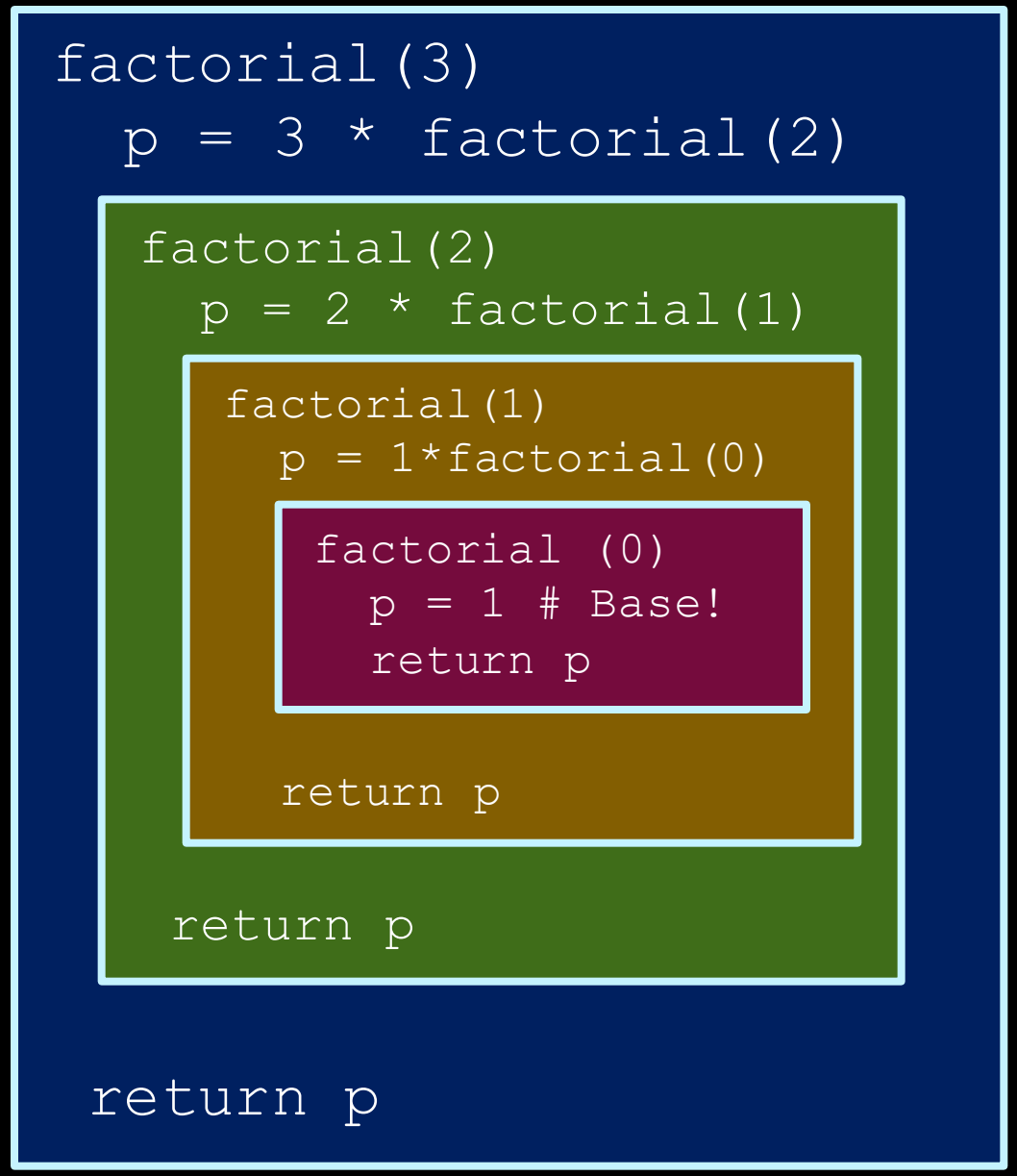
Recursion!

Recursion in Assembly

what recursion really is in hardware

Example: factorial(int n)

- Basic pseudocode for recursive factorial:
  - Base Case (n == 0)
    - return 1
  - Get factorial(n-1)
    - Store result in "product"
  - Multiply product by n
    - Store in "result"
  - Return result




```
int factorial(int n) {
    if (n==0)
      return 1;
    else
      return n*fact(n-1);
}
```

Before writing assembly, we need to know explicitly where to store values

```
int factorial (int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

Need to store …

- the value of n
- the value of n – 1
- the value factorial(n-1)
- the return value: 1 or n*factorial(n-1)

Design decision #1: store values in registers

```
int factorial(int n) {
   if (n==0)
     return 1;
   else
     return n*fact(n-1);
}
```

Does it work?

- store n in $t0
- store n-1 in $t1
- store factorial(n-1) in $t2
- store return value in $t3

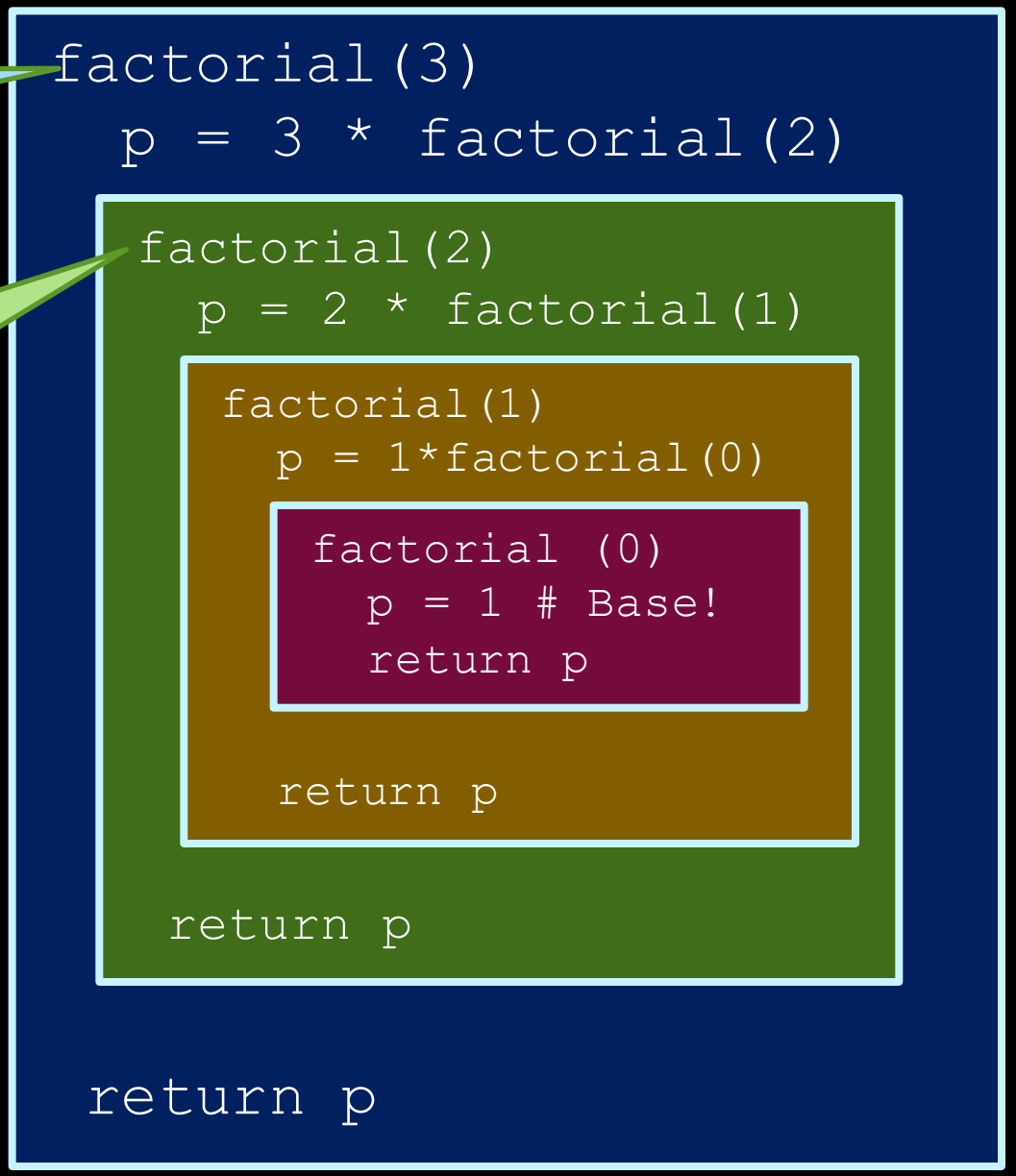
No, it doesn't work.

Store **n=3** in $to

Store **n=2** in $to, the stored 3 is overwritten, lost!

Same problem for $t1, t2, t3

- store n in $to
- store n-1 in $t1
- store factorial(n-1) in $t2
- store return value in $t3



A register is like a laundry basket -- you put your stuff there, but when you call another function (person), that person will use the same basket and take / mess up your stuff.

And yes, the other person will guarantee to use the **same** basket because … the other person is **YOU**! (because recursion)

So the correct design decision is to use __**Stack**__ .

Each recursive call has its own space for storing the values

Stores **n=2** for factorial (2)

Stores **n=3** for factorial (3)

Stacking Laundry Baskets are ideal for sorting laundry in small spaces.

Two useful things about stack

1. It has a lot of space
2. Its LIFO order (last in first out) is suitable for implementing recursions

LIFO order & recursive calls

```
factorial(2)
  p = 2 * factorial(1)
    factorial(1)
      p = 1*factorial(0)
        factorial (0)
          p = 1 # Base!
          return p

      return p

  return p
```

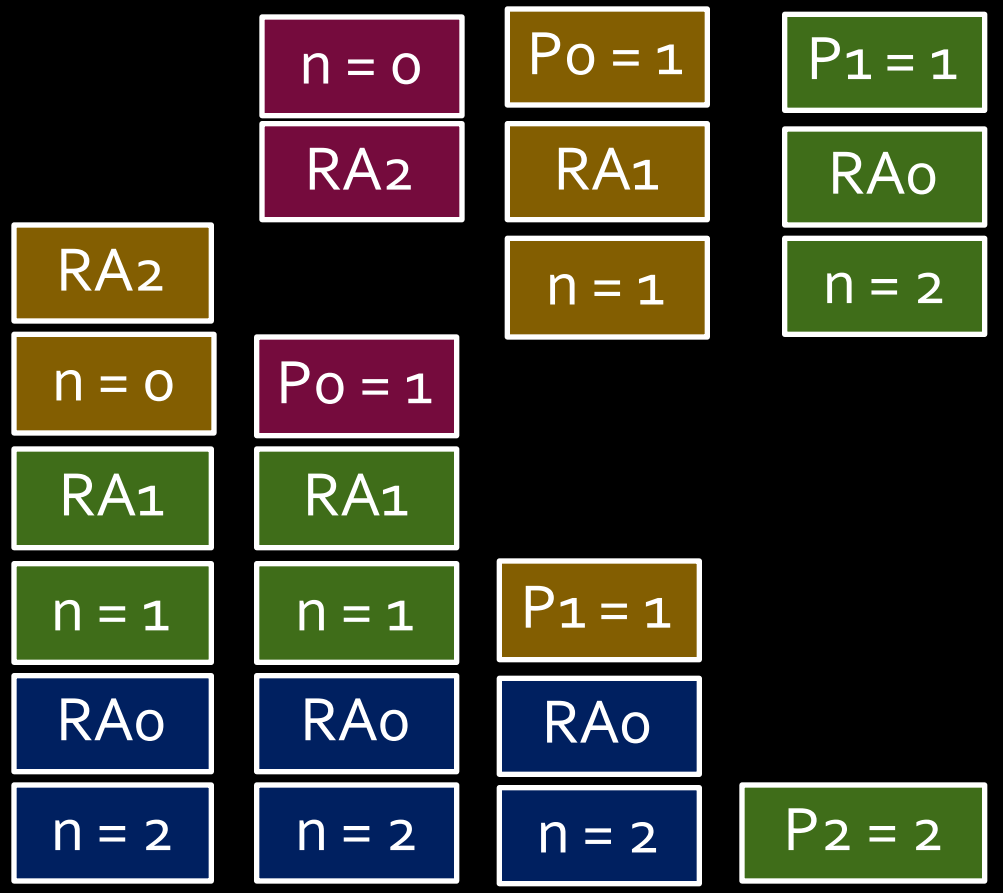
Note: Everybody is getting the **correct** basket because of LIFO!

Design decisions made,
now let’s actually write the
assembly code

LIFO order & recursive calls

```
factorial(n=2)
  r = factorial(1)
    factorial(n=1)
      r = factorial(0)
        factorial(n=0)
          p = 1 # Base!
          return p #P0
      p = n * r;  # RA2
    return p #P1
  p = n * r; # RA1
return p # P2
```

```
int x = 2;
int y = factorial(x)
print(y) # RA0
```



Actions in factorial (n)

Before making the recursive call

- pop argument n
- push argument n-1 (arg for recursive call)
- push return address (remember where to return)
- make the recursive call

After finishing the recursive call

- pop return value from recursive call
- pop return address
- compute return value
- push return value (so the upper call can get it)
- jump to return address

factorial(int n)

- Pop n off the stack
  - Store in $t0
- If $t0 == 0,
  - Push return value 1 onto stack
  - Return to calling program
- If $t0 != 0,
  - Push $t0 and $ra onto stack
  - Calculate n-1
  - Push n-1 onto stack
  - Call factorial

  - ...time passes...
  - Pop the result of factorial (n-1) from stack, store in $t2
  - Restore $ra and $t0 from stack
  - Multiply factorial (n-1) and n
  - Push result onto stack
  - Return to calling program

n → $t0

n-1 → $t1

fact(n-1) → $t2

factorial(int n)

```
fact:       lw $t0, 0($sp)
            addi $sp, $sp, 4
            bne $t0, $zero, not_base
            addi $t0, $zero, 1
            addi $sp, $sp, -4
            sw $t0, 0($sp)
            jr $ra
not_base:   addi $sp, $sp, -4
            sw $t0, 0($sp)
            addi $sp, $sp, -4
            sw $ra, 0($sp)
            addi $t1, $t0, -1
            addi $sp, $sp, -4
            sw $t1, 0($sp)
            jal fact
```

n → $t0

n-1 → $t1

fact(n-1) → $t2

- Pop n off the stack
  - Store in $t0
- If $t0 == 0,
  - Push return value 1 onto stack
  - Return to calling program
- If $t0 != 0,
  - Push $t0 and $ra onto stack
  - Calculate n-1
  - Push n-1 onto stack
  - Call factorial
  - Pop the result of factorial (n-1) from stack, store in $t2
  - Restore $ra and $t0 from stack
  - Multiply factorial (n-1) and n
  - Push result onto stack
  - Return to calling program

factorial(int n)

```
lw $t2, 0($sp)
addi $sp, $sp, 4
lw $ra, 0($sp)
addi $sp, $sp, 4
lw $t0, 0($sp)
addi $sp, $sp, 4
mult $t0, $t2
mflo $t3
addi $sp, $sp, -4
sw $t3, 0($sp)
jr $ra
```

n → $t0

n-1 → $t1

fact(n-1) → $t2

- Pop n off the stack
  - Store in $t0
- If $t0 == 0,
  - Push return value 1 onto stack
  - Return to calling program
- If $t0 != 0,
  - Push $t0 and $ra onto stack
  - Calculate n-1
  - Push n-1 onto stack
  - Call factorial
  - Pop the result of factorial (n-1) from stack, store in $t2
  - Restore $ra and $t0 from stack
  - Multiply factorial (n-1) and n
  - Push result onto stack
  - Return to calling program

Recursive programs

- How do we handle recursive programs?
  - Still needs base case and recursive step, as with other languages.
  - Main difference: Maintaining register values.
    - When a recursive function calls itself in assembly, it calls `jal` back to the beginning of the program.
    - Between different level of recursive calls, things are passed through the stack
      - Function arguments, return addresses, return values

```
int factorial (int x) {
    if (x==0)
        return 1;
    else
        return x*fact(x-1);
}
```

Recursive programs

- Use of stack
  - Before recursive call, store the register values that you use onto the stack, and restore them when you come back to that point.
  - Store $ra as one of those values, to remember where each recursive call should return.

```
int factorial (int x) {
    if (x==0)
        return 1;
    else
        return x*fact(x-1);
}
```

There is no recursion in hardware

- It's just a linear sequence of assembly instructions, where you jump to the beginning of the program over and over again.
- While sensibly store and retrieve remembered values from the stack

Translated recursive program (part 1)

```
main:       addi   $t0, $zero, 10      # call fact(10)
            addi   $sp, $sp, -4        #   by putting 10
            sw     $t0, 0($sp)         #   onto stack
            jal    factorial           # result will be
            ...                        #   on the stack
```

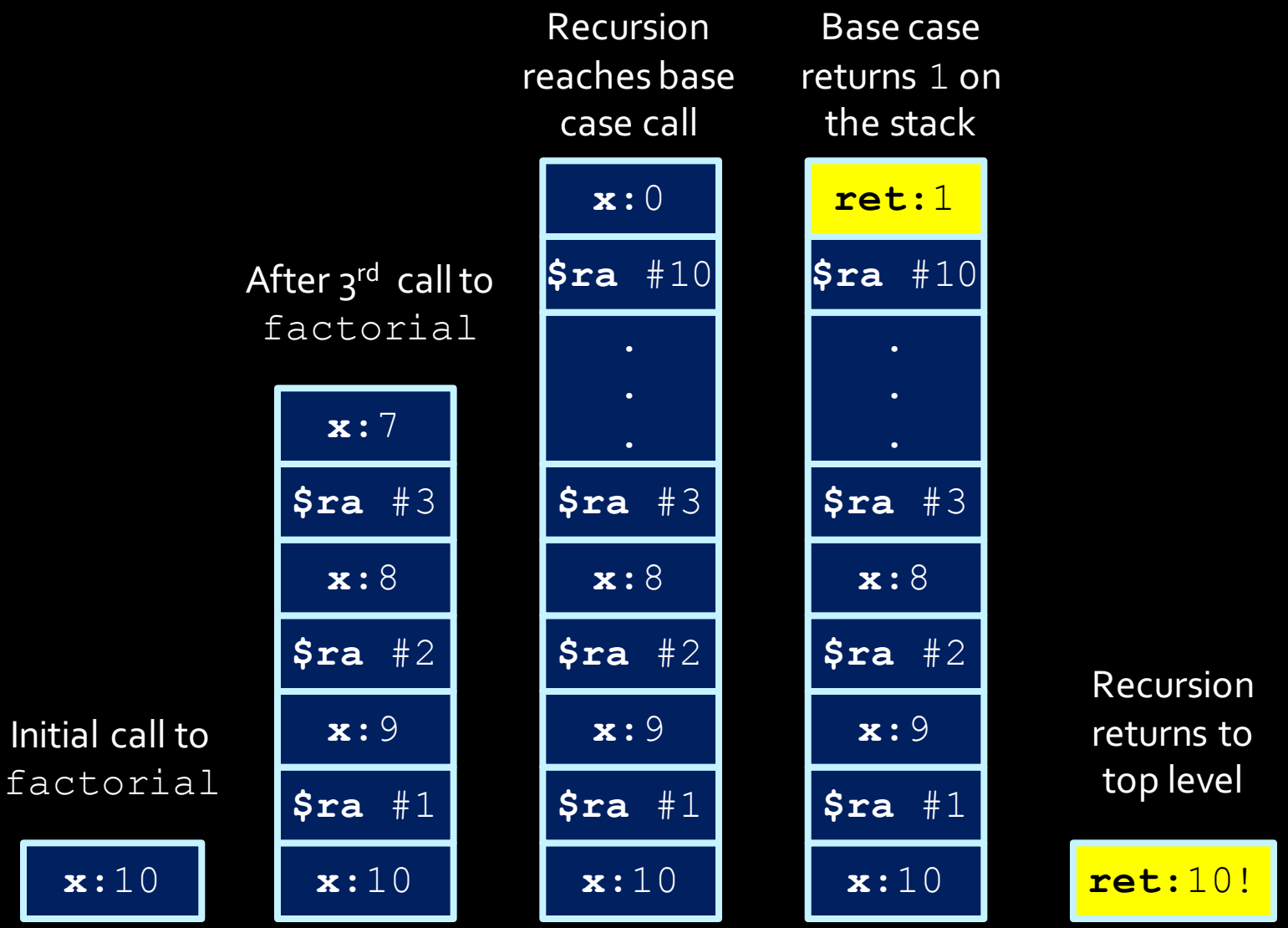
```
factorial:  lw     $a0, 4($sp)         # get x from stack
            bne    $a0, $zero, rec     # base case?
base:       addi   $t0, $zero, 1       # put return value
            sw     $t0, 4($sp)         #   onto stack
            jr     $ra                 # return to caller
rec:        addi   $sp, $sp, -4        # store return
            sw     $ra, 0($sp)         #   addr on stack
            addi   $a0, $a0, -1        # x--
            addi   $sp, $sp, -4        # push x on stack
            sw     $a0, 4($sp)         #   for rec call
            jal    factorial           # recursive call
```

Translated recursive program (part 2)

```
(continued from part 1)
            lw     $v0, 0($sp)          # get return value
            addi   $sp, $sp, 4          #   from stack
            lw     $ra, 0($sp)          # restore return
            addi   $sp, $sp, 4          #   address value
            lw     $a0, 0($sp)          # restore x value
            addi   $sp, $sp, 4          #   for this call
            mult   $a0, $v0             # x*fact(x-1)
            mflo   $t0                  # fetch product
            addi   $sp, $sp, -4         # push product
            sw     $t0, 0($sp)          #   onto stack
            jr     $ra                  # return to caller
```

- Note: `jal` always stores the next address location into `$ra`, and `jr` returns to that address.

Factorial stack view



You can't recur too much

The stack is NOT of infinite size, so there is always a limit on the number of recursive calls that you can make.

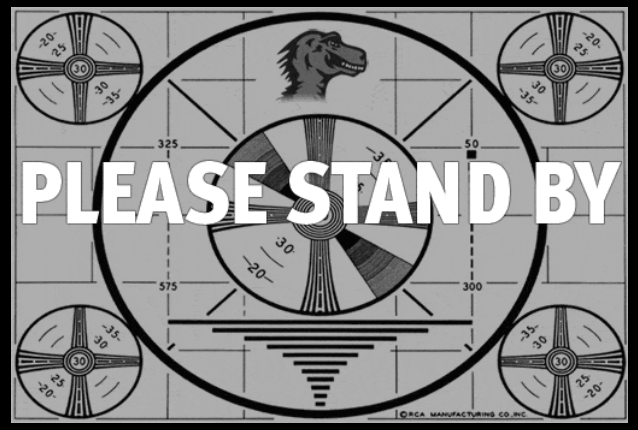
When exceeds that limit, you get a stack overflow, all content of the stack will be dumped.

Bureaublad — bash — 80×24

```
    state = deepcopy(state, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 147, in deepcopy
    y = copier(x, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 238, in _deepcopy_dict
    y[deepcopy(key, memo)] = deepcopy(value, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 147, in deepcopy
    y = copier(x, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 211, in _deepcopy_list
    y.append(deepcopy(a, memo))
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 147, in deepcopy
    y = copier(x, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 211, in _deepcopy_list
    y.append(deepcopy(a, memo))
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 143, in deepcopy
    cls = type(x)
RuntimeError: maximum recursion depth exceeded while calling a Python object
Mac-Pro-van-Mathias:Desktop Mathias$
```

Interrupts and Exception

A note on interrupts



- Interrupts take place when an external event requires a change in execution.
  - Example: arithmetic overflow, system calls (`syscall`), Ctrl-C, undefined instructions.
  - Usually signaled by an external input wire, which is checked at the end of each instruction.
  - High priority, override other actions

A note on interrupts

- Interrupts can be handled in two general ways:
  - Polled handling: The processor branches to the address of interrupt handling code (interruption handler), which begins a sequence of instructions that check the cause of the exception, i.e., need to ask around to figure out what type of exception.

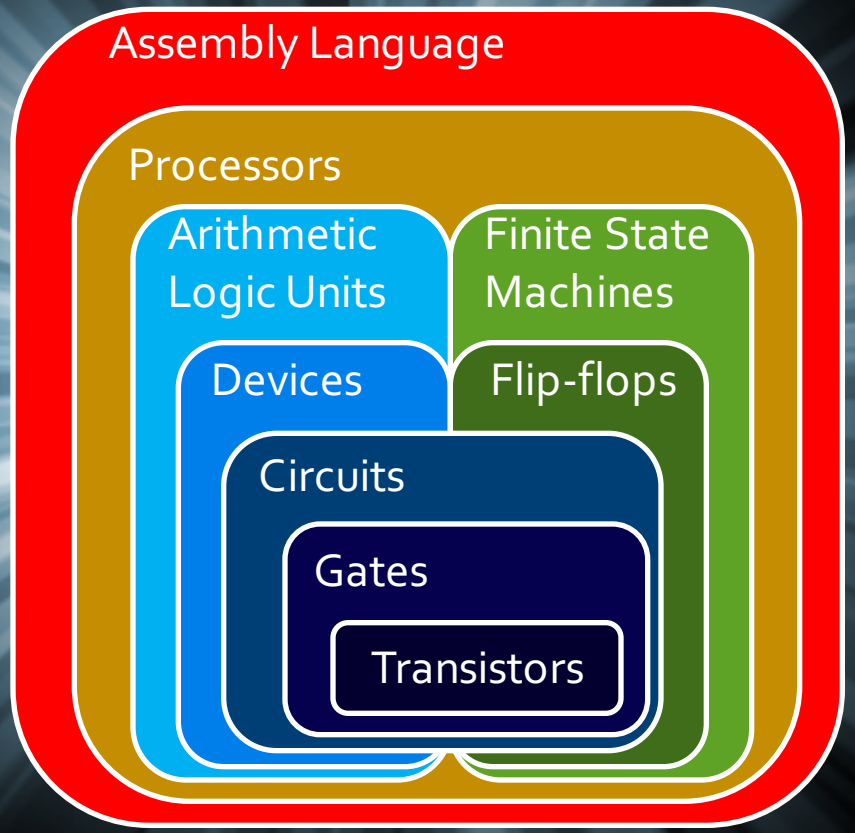
    → This is what MIPS uses.
  - Vectored handling: The processor can branch to a different address for each type of exception. Each exception address is separated by only one word. A jump instruction is placed at each of these addresses for the handler code for that exception. So no need to ask around.

Interrupt handling

- In the case of polled interrupt handling, the processor jumps to exception handler code, based on the value in the cause register (see table).
  - If the original program can resume afterwards, this interrupt handler returns to program by calling `rfe` instruction.
  - Otherwise, the stack contents are dumped and execution will continue elsewhere.

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

We are done!


Assembly Language
Processors
Arithmetic Logic Units
Finite State Machines
Devices
Flip-flops
Circuits
Gates
Transistors

Given enough silicon, phosphorus and boron, you are now able to build a computer!

Final Exam Review

Time & Location

Time: April 11, 9am-12pm

Location: IB-110

No aid.

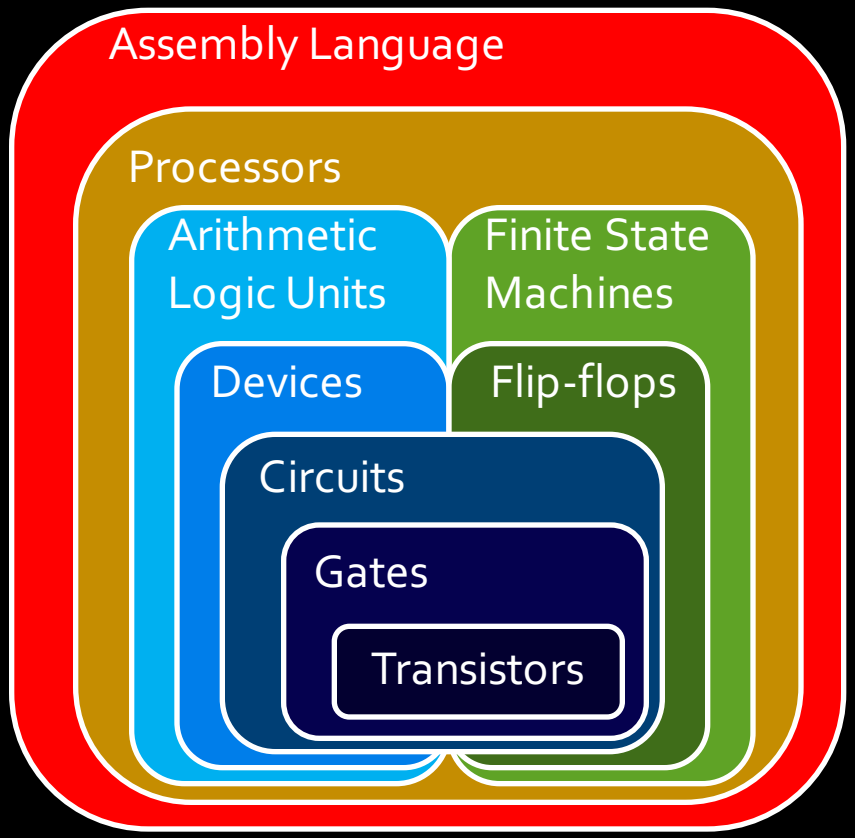
Bring your student card

Pre-exam office hours

- March 28~April 1
  - Monday, Tuesday, Friday
  - 4-6pm
- April 4~April 8
  - Monday, Tuesday, Thursday, Friday
  - starting 3pm

Coverage

- Everything
- Slightly more weight on the second half (after midterm)
- No Verilog questions
- No Booth algorithm



Types of questions

- Short answer: conceptual questions
- Read combination circuit
- Design combinational circuit
- Read sequential circuit, draw waveform
- Design sequential circuit, FSM
- Set Datapath signals
- Write assembly code according to requirement
- Read assembly code, see what it does
- Translate between assembly and machine code
- Translate between assembly and C code.

How to study for final exam

1. Review lecture / tutorial slides
2. Review what you did for labs
3. Review quizzes
4. Practice with past exams
   - One with solution posted on course page
   - More in old exam repository
5. Whenever confused, ask on Piazza, or come to **office hours**

Important Lessons

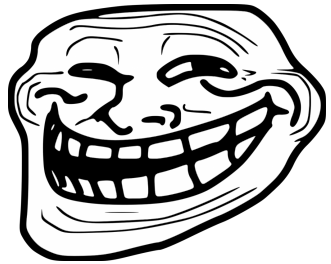
- Pay attention to **details**
  - Don't just skim over the slides and feel "got the idea".
- Understand very well what you did in the **labs**
- Understand very well the **quiz** questions

Sample questions

1. Who messed up Agent Smith's laundry basket when only one basket is used? **(2 marks)**

2. Name the two students who demoed in class the "human flip flop". **(2 marks)**

3. What did Bruce Lee say about the flow of electricity? **(2 marks)**



Quiz Bonus

3 bonus marks if you get >= 18 points

2 bonus marks if you get >= 10 points

1 bonus mark if you get >= 1 point

Name	Points
Chris C.	**29**
Elijah M.	**29**
Alexei F.	23
Jay G.	23
Shayan G.	22
Brandon A. M.	21
Eric C.	21
Gang Z.	21
James T.	21
Joseph C.	21
Raj S.	21
Alexander K.	20
Frank Y.	20
Jailani D.	20
Michael Z.	20
Ramy E.	20

See you in office hours!