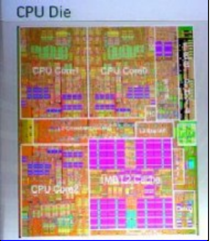
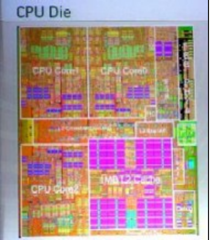



COMP 273
Introduction to Computer Systems

# COMP 273

## Procedures, Run-Time Stack and Complex Data Structures

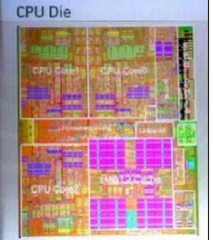
Prof. Joseph Vybihal






# Announcements

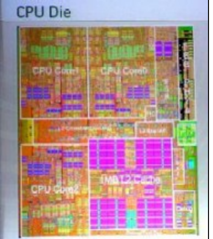
- Ass#3, will be our last assignment, and it will come out by the end of the week.






# At Home

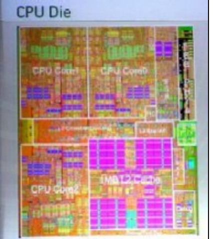
- Write the recursive and non-recursive functions shown in class on your MIPS interpreter. Get them to run and print out their results to the screen. You will need to add a main().
- Web Resources:
  - http://www.cs.sunysb.edu/~cse320/example.html
  - http://www.stewart.cs.sdsu.edu/cs524/fall06/lects/p295_runit.html






# Part 1

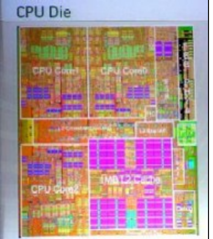
## Procedures and Stacks






# Calling Techniques

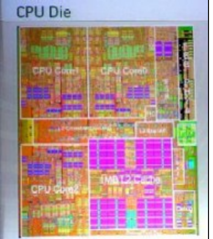
- Register Based
  - Use registers a0 to a4 to pass arguments
  - Use registers v0 and v1 to return values
- Stack Based
  - Passing parameters and returning results using the run-time stack






# Register Based

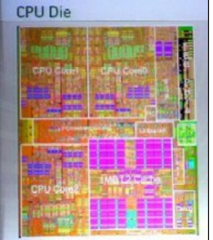
```
Main:# result = calc(a,b,c,d);

      # Setup parameters using $a0 to $a3
      lw $a0, a
      lw $a1, b
      lw $a2, c
      lw $a3, d

      # Call the subroutine
      jal calc  # $ra <-- return address

      # Return values assumed in v0 or v1
      sw $v0, result
```






# Example

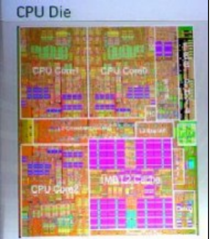
```
int calc(int g, int h, int i, int j)
{
   int f;
   f = (g+h)-(i+j);
   return f;           // return (g+h)-(i+j)
}

# assume: $a0=g, $a1=h, $a2=i, $a3=j
# Extra local variable $s0 = f

Calc: add  $t0,$a0,$a1    # g+h
      add  $t1,$a2,$a3    # i+j
      sub  $s0,$t0,$t1    # subtract the two results
      add  $v0,$s0,$zero # put answer in return register
      jr   $ra            # return
```






# Register Based

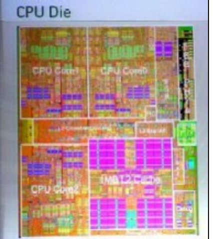
- Benefits:
  - Fast and easy to code
- Drawback:
  - Limited number of registers
  - No local variable simulation






# Run-time stack based

```
Main:# result = calc(a,b,c,d);

     # Setup parameters using stack
     subi $sp, $sp, 16   # make room
     sw $t0, 12($sp)     # assume t0=a
     sw $t1,  8($sp)
     sw $t2,  4($sp)
     sw $t3,  0($sp)

     # Call the subroutine
     jal calc  # $ra <-- return address

     # Return values assumed in v0 or v1
     sw $v0, result
```





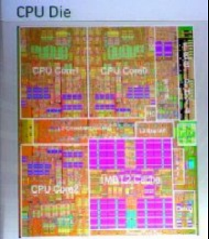

# Example with protection

```
# assume: on stack g, h, i, j
# Extra local variable $s0 = f

Calc: lw    $a0, 12($sp)   # load the parameters
      lw    $a1,  8($sp)
      lw    $a2,  4($sp)
      lw    $a3,  0($sp)
      sub   $sp,$sp,12     # protect 3 registers
      sw    $t1,8($sp)     # stack 3 registers (push)
      sw    $t0,4($sp)
      sw    $s0,0($sp)
      add   $t0,$a0,$a1    # Do work: g+h
      add   $t1,$a2,$a3    # i+j
      sub   $s0,$t0,$t1    # subtract the two results
      add   $v0,$s0,$zero  # put answer in return register
      lw    $s0,0($sp)     # pop stack
      lw    $t0,4($sp)
      lw    $t1,8($sp)
      add   $sp,$sp,28
      jr    $ra            # return
```

Alternatively we could have pushed the result and popped it out in main.

# Run-time stack based

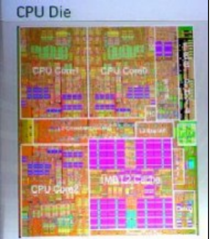
- Benefits:
  - Stack is very large so many parameters can fit
  - Simulates local variables when protection of registers are used
- Drawback:
  - Slows down program by increasing the number of lines of code and the number of move operations






# The Anatomy of a Function

- Calling
  - JAL
  - Passing parameters
    - stacking them or use the $a registers
- Protecting registers locally
  - All registers you use in function get stacked to protect the calling environment
- Returning
  - Restore the calling environment
  - Return value by register, pointer or stack
  - JR



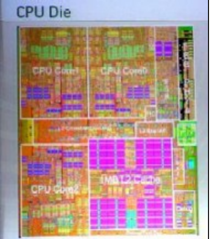



# The Elements

- Double duty for general purpose registers that support procedures:
  - \$a0 - \$a3: Pass parameters
  - \$v0 - \$v1: Return values
  - \$ra : Return address = PC + 4
- The jump-and-link commands
  - jal ProcedureAddress
    - Step 1: save current address into \$ra = PC + 4
    - Step 2: jump to ProcedureAddress
- The Return (jump register) Statement
  - jr \$ra

# By Convention…

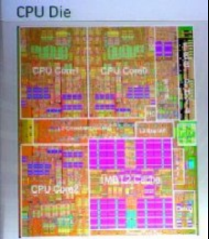
- $t registers not protected by the stack
  - Considered to be temporary registers
- $s registers are protected by the stack
  - Called the SAVED registers
- All other registers are optionally saved by the programmer






# Run-Time Stack

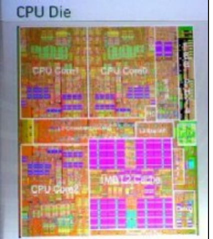
- When calling a procedure/function
  - Local variables are loaded into registers
  - Problematic since registers already have values from the previous function and maybe they are still active and important…
- Solution:
  - Stack the registers you want to use in your procedure, or
  - Simply stack all the registers
- Note:
  - Good policy to save EVERYTHING anyway






# Memory Considerations

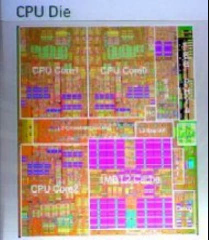
- Protection based
- Register memory based
- Global memory based
- Local memory based
- Combination






# Memory Considerations

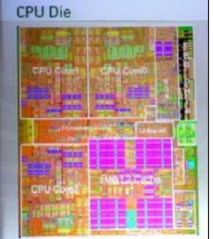
- Protection based
  - Defend the calling environment by saving all the registers after entering the subroutine.
  - Restore the registers on the way out
- Register memory based
- Global memory based
- Local memory based
- Combination




COMP 273
Introduction to Computer Systems

# Memory Considerations

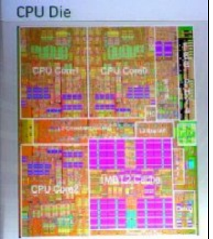
- Protection based
- Register memory based
    - Using the CPU's registers as the only subroutine memory
- Global memory based
- Local memory based
- Combination






# Memory Considerations

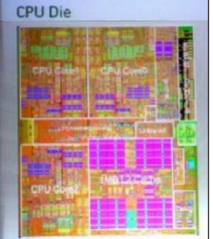
- Protection based
- Register memory based
- Global memory based
    - Using RAM and instructions lw and sw as the only subroutine memory.
- Local memory based
- Combination






# Memory Considerations

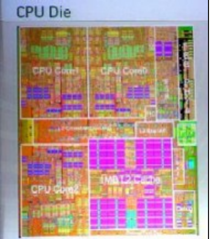
- Protection based
- Register memory based
- Global memory based
- Local memory based
  - Using the run-time stack as the subroutine's only memory
- Combination






# Memory Considerations

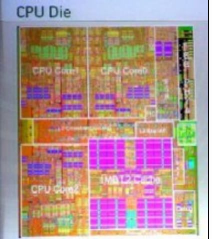
- Protection based
- Register memory based
- Global memory based
- Local memory based
- Combination
    - Using a combination of the above as the subroutine's memory






# Global vs. Local Variables

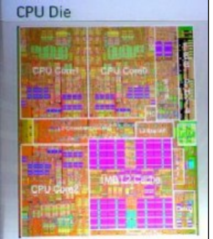
- By definition
  - Global are those items in RAM that are accessible within the entire scope of the program.
  - Local are those items in RAM that are accessible only to the current function
- Implementation
  - Global items are stored in the .data area
  - Local items are stored in the run-time stack
  - Note: CPU registers are considered to be temporary locations






# Simulating Local Variables

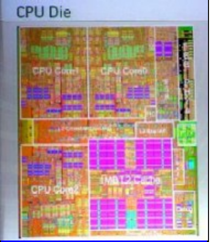
- Int a = 5;
  - Static
    - ADDI $t1,$zero,5
  - Global by .data
    - LA $t1, LABEL
      - "call-by-reference"
      - LW $t2, 0($t1)
  - Local by stack
    - LW $t2, offset($SP)
      - "call-by-value" or value can be a reference
      - Pushed previously, we know offset number





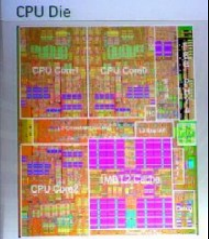

# Reality Check

- The OS defines the max and min address space your program can operate within
- Within this defined space you can access anything anywhere... there is no such thing as local!
- Simulation is the trick and the run-time stack is our tool.
  - Therefore we code our functions strictly to use the R-T Stack and immediate commands.
  - Exceptions:
    - Pointers

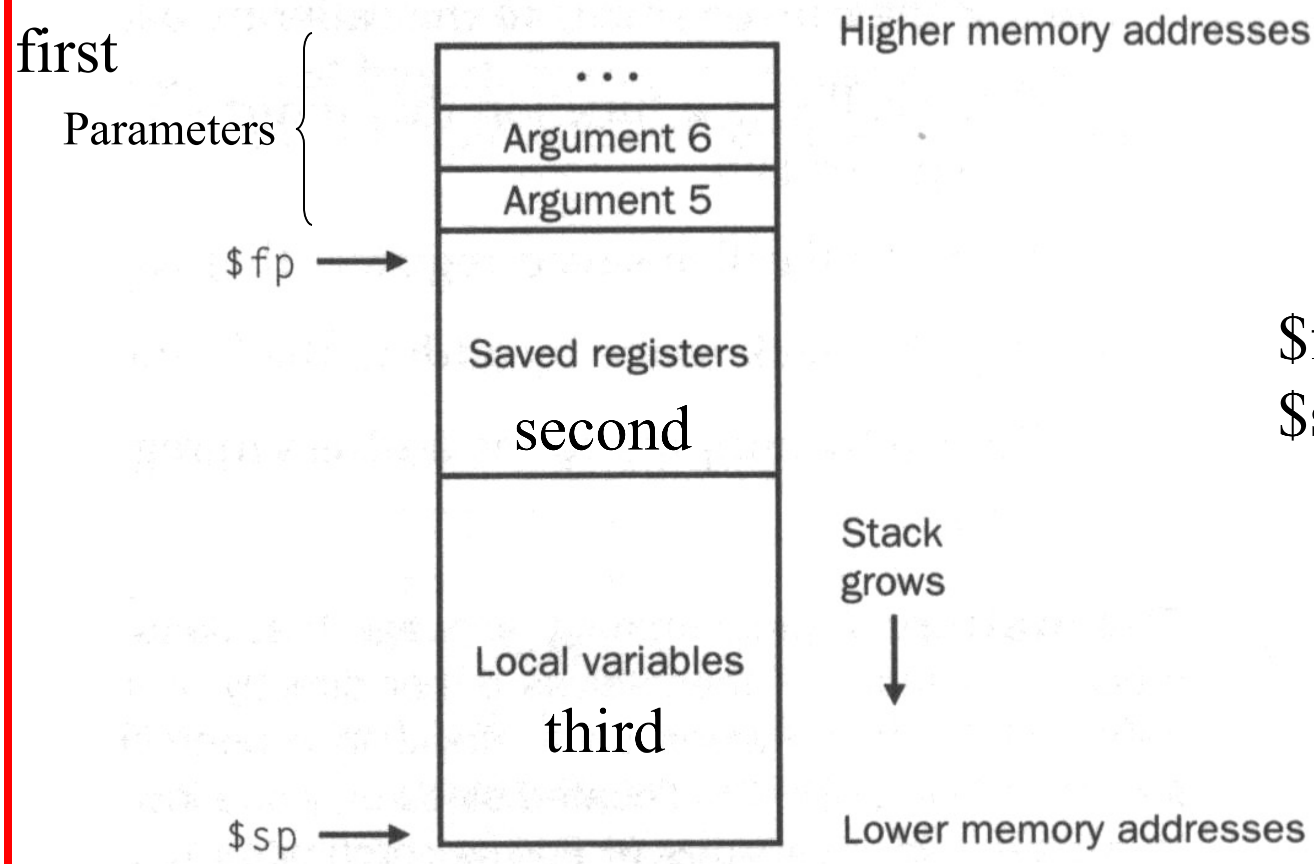





# Run-time stack anatomy

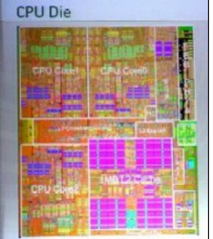



# $sp and $fp




$fp – start
$sp - top

Implementation rules: do not let variables refer past $fp.






Registers

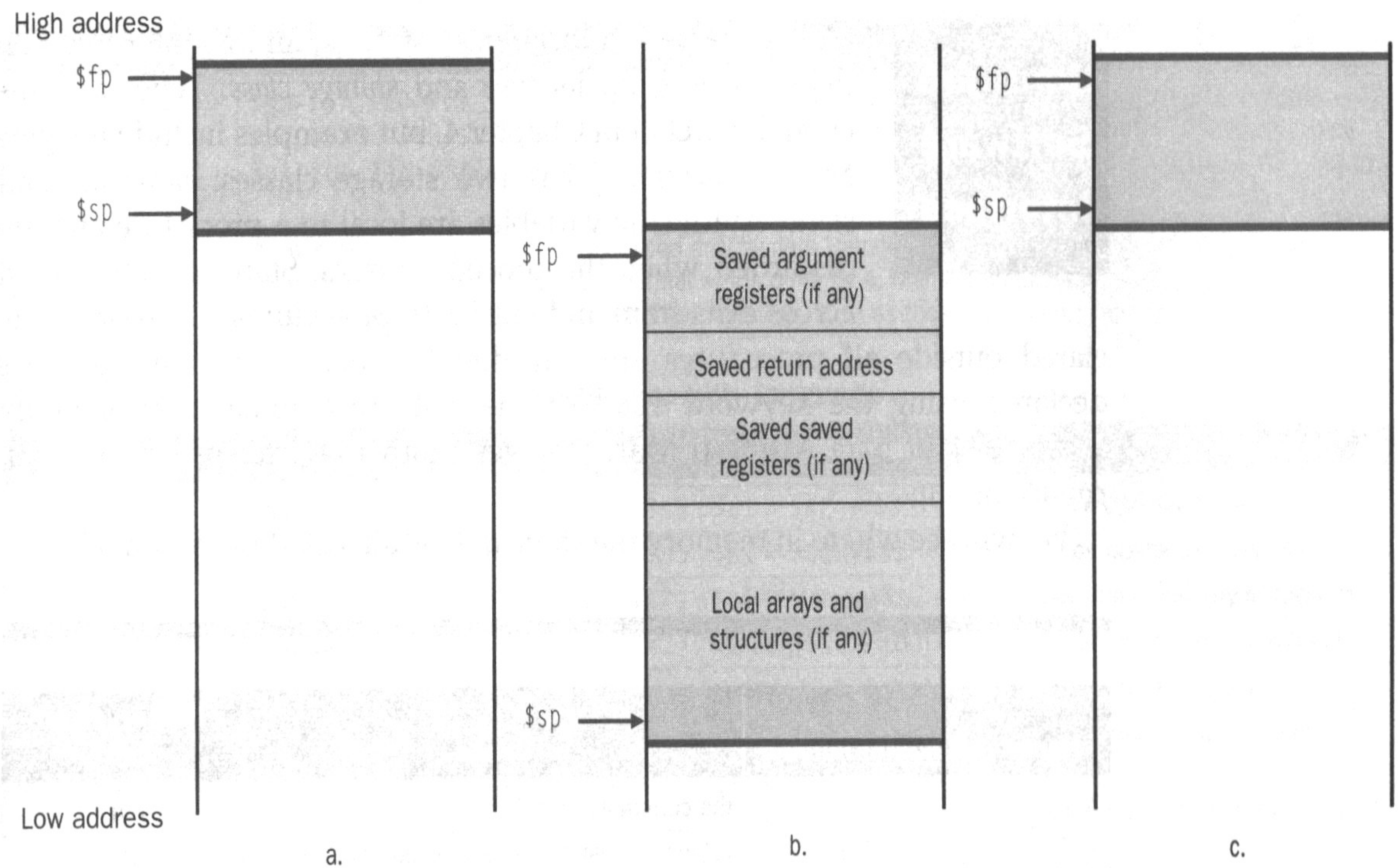
(global)

Dynamic
Stack
(local / free)

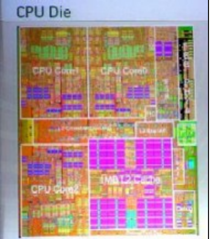
.data

(global)

# Using the run-time stack to help us… (creating a block var)




$fp is used to provide a constant offset when programs move the $sp midway between entry and exit procedures.



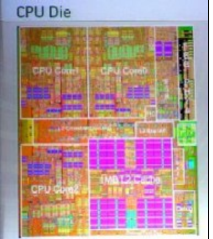



# Run-Time Stack Functionality

- \$sp is used to point to a buffer in RAM
  - The Buffer is assumed to be a block of free memory that will be treated like a stack
  - \$sp is the top of the stack
  - PUSH = \$sp – 4 (!)
  - POP = \$sp + 4
- \$sp is already setup and ready for you when your program starts

# Recursive Procedures

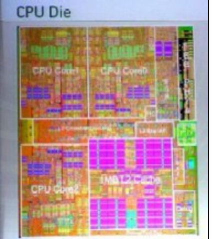
- Problems:
  - Limited number of registers
    - Multiple calls
    - Parameters for each call
    - Local variables
  - Only one $ra registers
    - We need that for each return statement






# Example

```
int fact(int n)
{
    if (n<1) return 1;
    else return n * fact(n-1);
}

# assume $a0 = n

Fact: subi $sp,$sp,8      # make room for 2 registers
      sw   $ra,4($sp)
      sw   $a0,0($sp)
      slt  $t0,$a0,1      # if (n<1)
      beq  $t0,$zero,L1
      addi $v0,$zero,1    # return 1
      addi $sp,$sp,8      # pop 2 from stack (not restored)
      jr   $ra
L1:   subi $a0,$a0,1      # n-1
      jal  Fact           # recursion
  *   lw   $a0,0($sp)     # when we return back
      lw   $ra,4($sp)
      addi $sp,$sp,8
      mul  $v0,$a0,$v0    # n*fact(n-1)
      jr   $ra
```




COMP 273

Introduction to Computer Systems

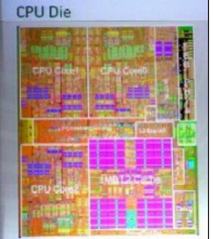
# The Anatomy of a Function

Stacking

Reverse order

| | | | | |
|---|---|---|---|---|
| **Saving registers** | | | | |
| | sort: | addi | $sp,$sp, -20 | # make room on stack for 5 registers |
| | | sw | $ra, 16($sp) | # save $ra on stack |
| | | sw | $s3,12($sp) | # save $s3 on stack |
| | | sw | $s2, 8($sp) | # save $s2 on stack |
| | | sw | $s1, 4($sp) | # save $s1 on stack |
| | | sw | $s0, 0($sp) | # save $s0 on stack |
| **Procedure body** | | | | |
| Move parameters | | move | $s2, $a0 | # copy parameter $a0 into $s2 (save $a0) |
| | | move | $s3, $a1 | # copy parameter $a1 into $s3 (save $a1) |
| Outer loop | | move | $s0, $zero | # i = 0 |
| | for1tst: | slt | $t0, $s0, $s3 | # reg $t0 = 0 if $s0 ≥ $s3 (i ≥ n) |
| | | beq | $t0, $zero, exit1 | # go to exit1 if $s0 ≥ $s3 (i ≥ n) |
| Inner loop | | addi | $s1, $s0, -1 | # j = i - 1 |
| | for2tst: | slti | $t0, $s1, 0 | # reg $t0 = 1 if $s1 < 0 (j < 0) |
| | | bne | $t0, $zero, exit2 | # go to exit2 if $s1 < 0 (j < 0) |
| | | add | $t1, $s1, $s1 | # reg $t1 = j * 2 |
| | | add | $t1, $t1, $t1 | # reg $t1 = j * 4 |
| | | add | $t2, $s2, $t1 | # reg $t2 = v + (j * 4) |
| | | lw | $t3, 0($t2) | # reg $t3 = v[j] |
| | | lw | $t4, 4($t2) | # reg $t4 = v[j + 1] |
| | | slt | $t0, $t4, $t3 | # reg $t0 = 0 if $t4 ≥ $t3 |
| | | beq | $t0, $zero, exit2 | # go to exit2 if $t4 ≥ $t3 |
| Pass parameters and call | | move | $a0, $s2 | # 1st parameter of swap is v (old $a0) |
| | | move | $a1, $s1 | # 2nd parameter of swap is j |
| | | jal | swap | # swap code shown in Figure 3.24 |
| Inner loop | | addi | $s1, $s1, -1 | # j = j - 1 |
| | | j | for2tst | # jump to test of inner loop |
| Outer loop | exit2: | addi | $s0, $s0, 1 | # i = i + 1 |
| | | j | for1tst | # jump to test of outer loop |
| **Restoring registers** | | | | |
| | exit1: | lw | $s0, 0($sp) | # restore $s0 from stack |
| | | lw | $s1, 4($sp) | # restore $s1 from stack |
| | | lw | $s2, 8($sp) | # restore $s2 from stack |
| | | lw | $s3,12($sp) | # restore $s3 from stack |
| | | lw | $ra,16($sp) | # restore $ra from stack |
| | | addi | $sp,$sp, 20 | # restore stack pointer |
| **Procedure return** | | | | |
| | | jr | $ra | # return to calling routine |

Popping

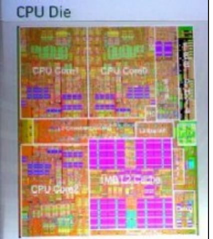





# Question

- Using an array, implement your own stack.

(Do this as a discussion)






```
1.  int x = 10;
2.
3.  int add(int x, int y[]) {
4.     int a;
5.     int b[10];
6.     .....
7.     return a;
8.  }
9.
A. void main() {
B.     int c;
C.     int d[5];
D.     c = add(5, d);
E. }
```

R-T Stack

add

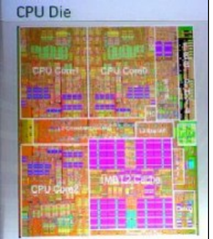
X, *y, a, b[10]
Return to D.

main

Int c, d[5]
Return to OS

.DATA

Global x = 10




COMP 273
Introduction to Computer Systems

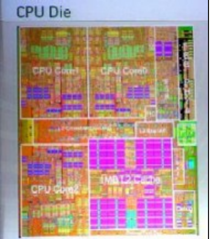
```
1.  Int fact(int n) {
2.          If (n <= 1) return 1;
3.          Else return n * fact(n-1);
4. }

A. Main
B. {
C.  fact(3);
D. }
```

~~N = 1~~
~~Return to 3.~~

~~N = 2~~
~~Return to 3.~~

~~N = 3~~
~~Return to C.~~

Return to OS

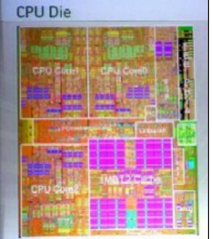





# Calling a subroutine

- Make space on the stack
- Push your arguments
- JAL to your subroutine

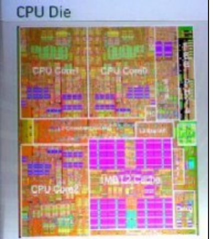
```
> subi $sp, $sp, amountOfSpace
> sw $r, offset($sp)
> JAL subroutineName
```






# Example

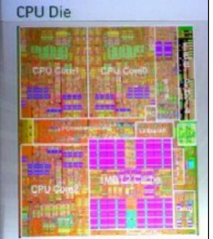
```
Main:# result = calc(a,b,c,d);

     # Setup parameters using stack
     subi $sp, $sp, 16   # make room
     sw $t0, 12($sp)     # assume t0=a
     sw $t1,  8($sp)
     sw $t2,  4($sp)
     sw $t3,  0($sp)

     # Call the subroutine
     jal calc  # $ra <-- return address

     # Return values assumed in v0 or v1
     sw $v0, result
```






# A subroutine

- Entrance
    - Make space on the stack
    - Push the registers you plan to use
    - Move arguments into registers
- DO THE SUBROUTINE BODY
- Exit
    - Move the result into $v0, $v1
    - Pop all saved registers and arguments
    - Remove stack space > addi $sp,$sp,amount
    - Return > jr $ra

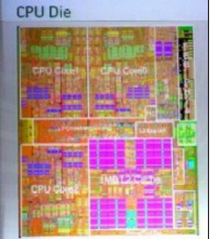





# Example with protection

```
# assume: on stack g, h, i, j
# Extra local variable $s0 = f

Calc: lw    $a0, 12($sp)   # load the parameters
      lw    $a1,  8($sp)
      lw    $a2,  4($sp)
      lw    $a3,  0($sp)
      sub   $sp,$sp,12     # protect 3 registers
      sw    $t1,8($sp)     # stack 3 registers (push)
      sw    $t0,4($sp)
      sw    $s0,0($sp)
      add   $t0,$a0,$a1    # Do work: g+h
      add   $t1,$a2,$a3    # i+j
      sub   $s0,$t0,$t1    # subtract the two results
      add   $v0,$s0,$zero  # put answer in return register
      lw    $s0,0($sp)     # pop stack
      lw    $t0,4($sp)
      lw    $t1,8($sp)
      add   $sp,$sp,28
      jr    $ra            # return
```

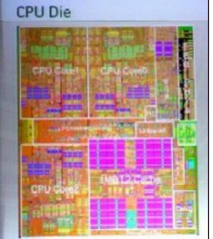
Alternatively we could have pushed the result and popped it out in main.






# Part 2

## Over-sized Data
## & Local Variables

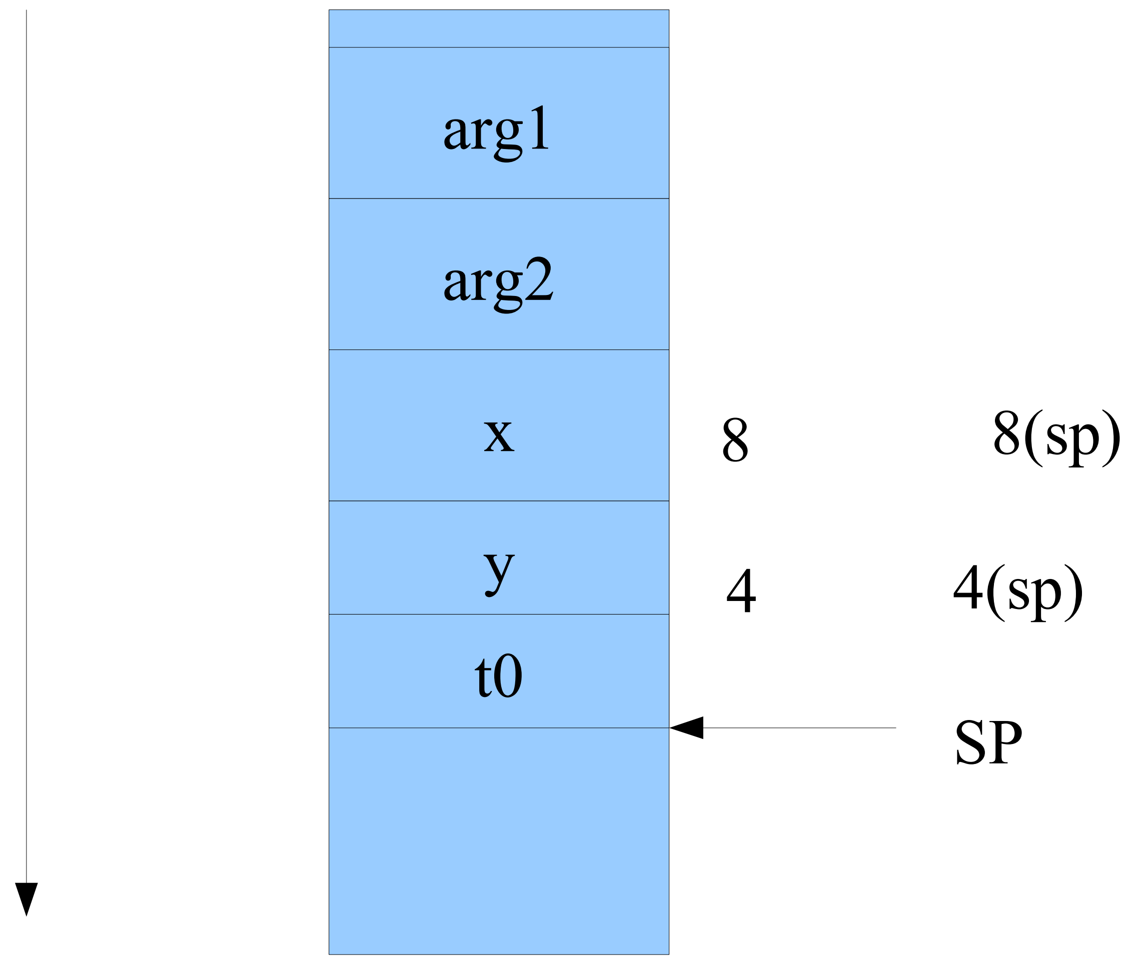


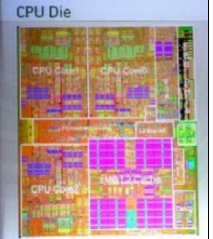



# Local Var

- Use a register
  - Int x; ---------- $t0
  - Int a,b,c,d,e,f,g; ----- $t0,t1,t2.t3…
  - Int y;
    - Subi sp,sp,4

COMP 273
Introduction to Computer Systems






| |
|---|
| Cell n |
| |
| |
| Cell 2 |
| Cell 1 |
| Cell 0 |
| x |
| t0 |

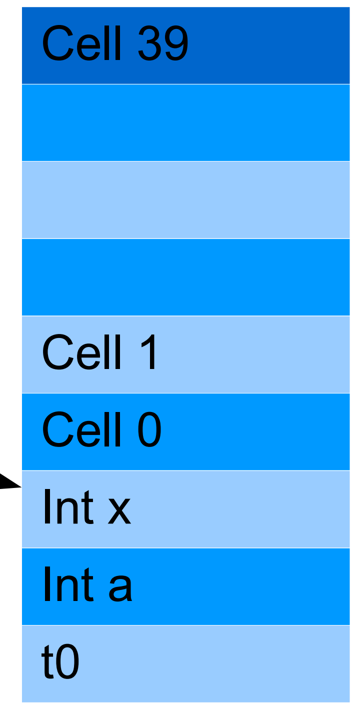
8(sp)

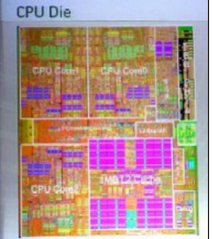
SP

```
Int fun() {
Int a;
Struct abc
{
      Int x;
      Char name[40];
} *p;
```

P

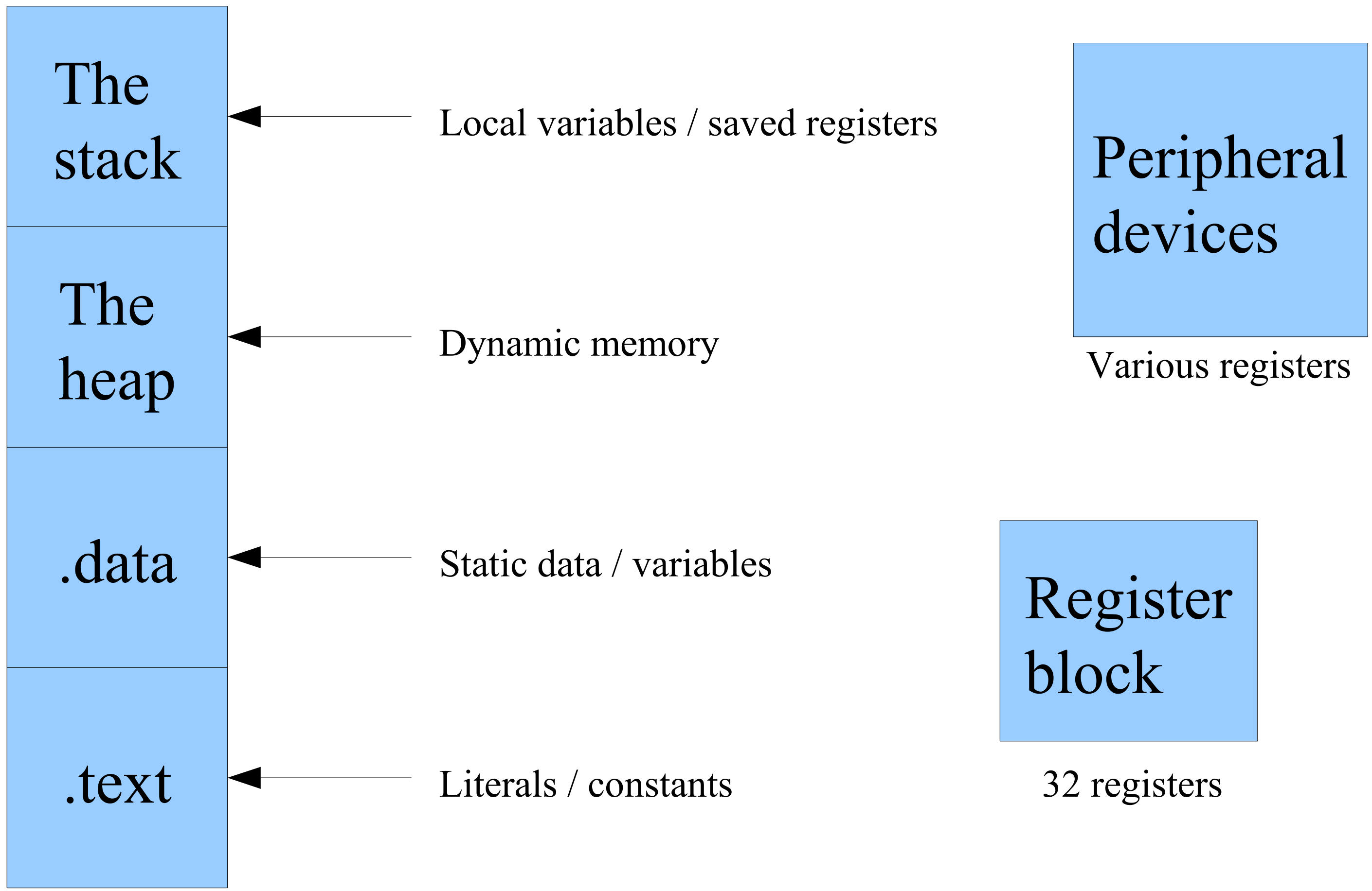
P = 8(sp)

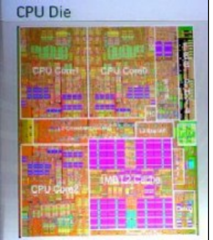







# Where data lives in MIPS

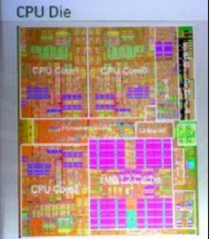







# Examples

- How do we create and use each of these types of memory
  - Literals
  - Static variables
  - Local variables
  - Static data structures (array, linked list)
    - Oversized data (structures)
  - Local data structures (array)
  - Dynamic memory (syscall)
  - Call-by-value vs. call-by-reference

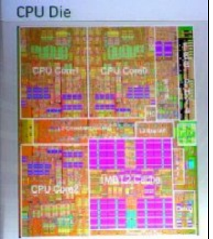





Draw

# Memory Overview

- CPU
  - Registers
    - General purpose
    - Specific purpose
  - Cache (general & pipelined versions)
- RAM
  - Reserved zero page
  - General purpose RAM
  - Protected OS Space
- Peripheral Cards
  - Specific purpose registers
    - In some cases true mini-CPU (graphic cards)
  - Buffers
    - In some cases true RAM (graphic cards)





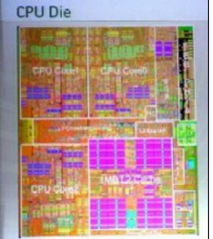

# Data Types

- Character
- Integer
- Fixed Point
- Pointers
  - Call-by-value
  - Call-by-reference
  - Scope
    - Global
    - Local
    - Cross-scope

In memory:
- Location?
- Format?
- Implementation?

Data fits in:
– Registers
– Instructions
– RAM/Cache

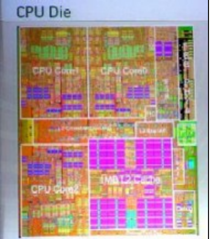
Difference between data & instructions?
– Sequencer
ASCII & Data?






# Data Structures

- Topics:
  - Memory location?
    (local + global, parameters)
  - Format?
  - Implementation?
- Data Structures (C in Assembler):
  - Arrays
    - Stacks & Heaps + malloc
  - Structs
  - Objects





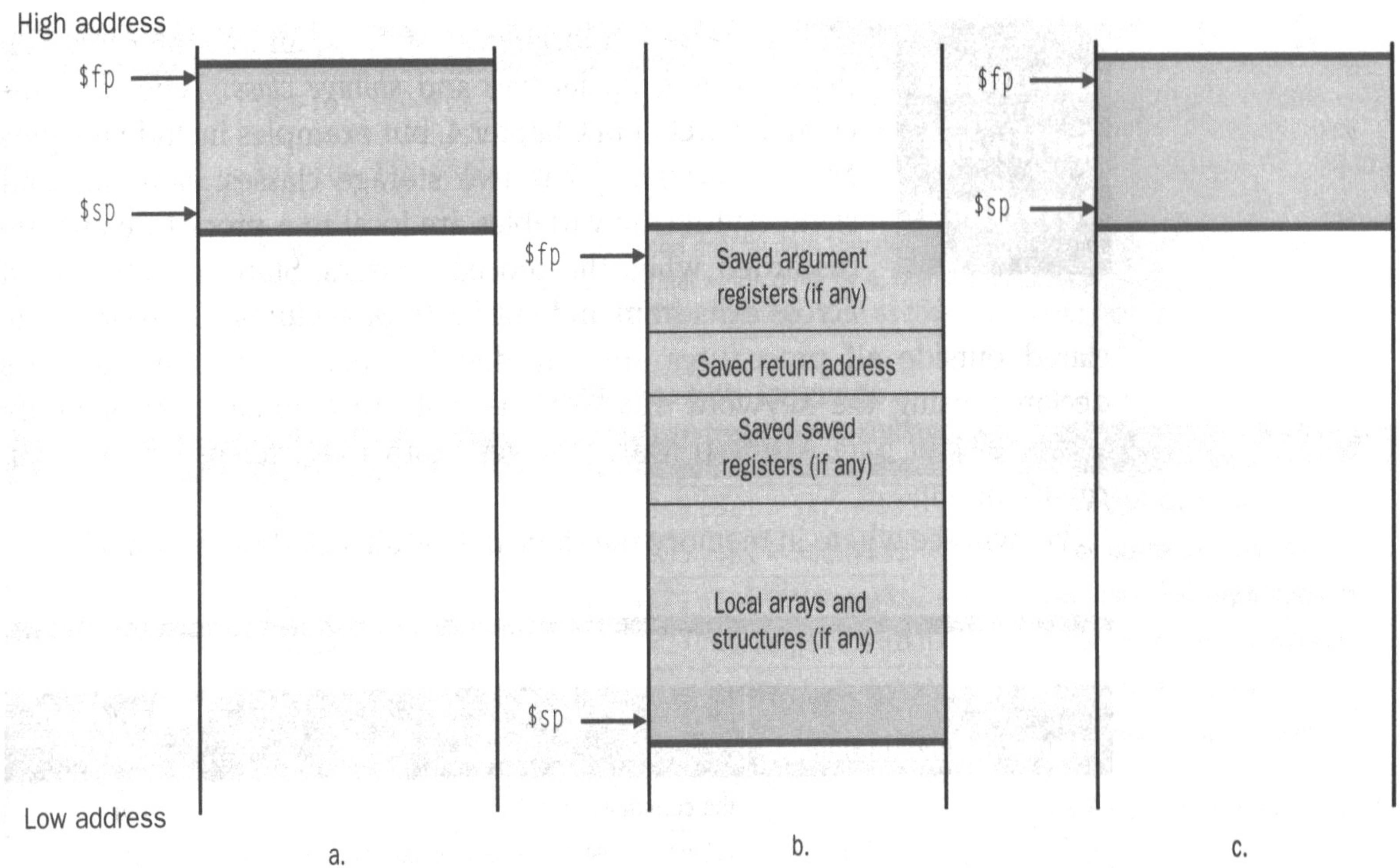

# Over-sized Local Variables

- What do we do with data that do not fit in a register? How do we pass them?
  - Arrays
  - Structures
  - Objects
- Solution:
  - Pointers (i.e. don’t pass them, send a pointer)
  - Local variables:
    - Make space / define space on stack

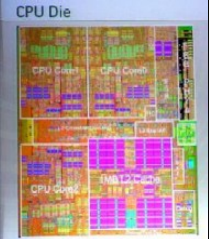
# Examples

- Array
  - Global
  - Local
- C Struct statement and access
- True object and method representation and access

# Using the run-time stack to help us… (creating a block var)




$fp is used to provide a constant offset when programs move the $sp midway between entry and exit procedures.






# Example

- Pseudo malloc using the run-time stack
  - Of a struct
  - What about block variables?

- Different ways of representing variables? Data?

Discuss…