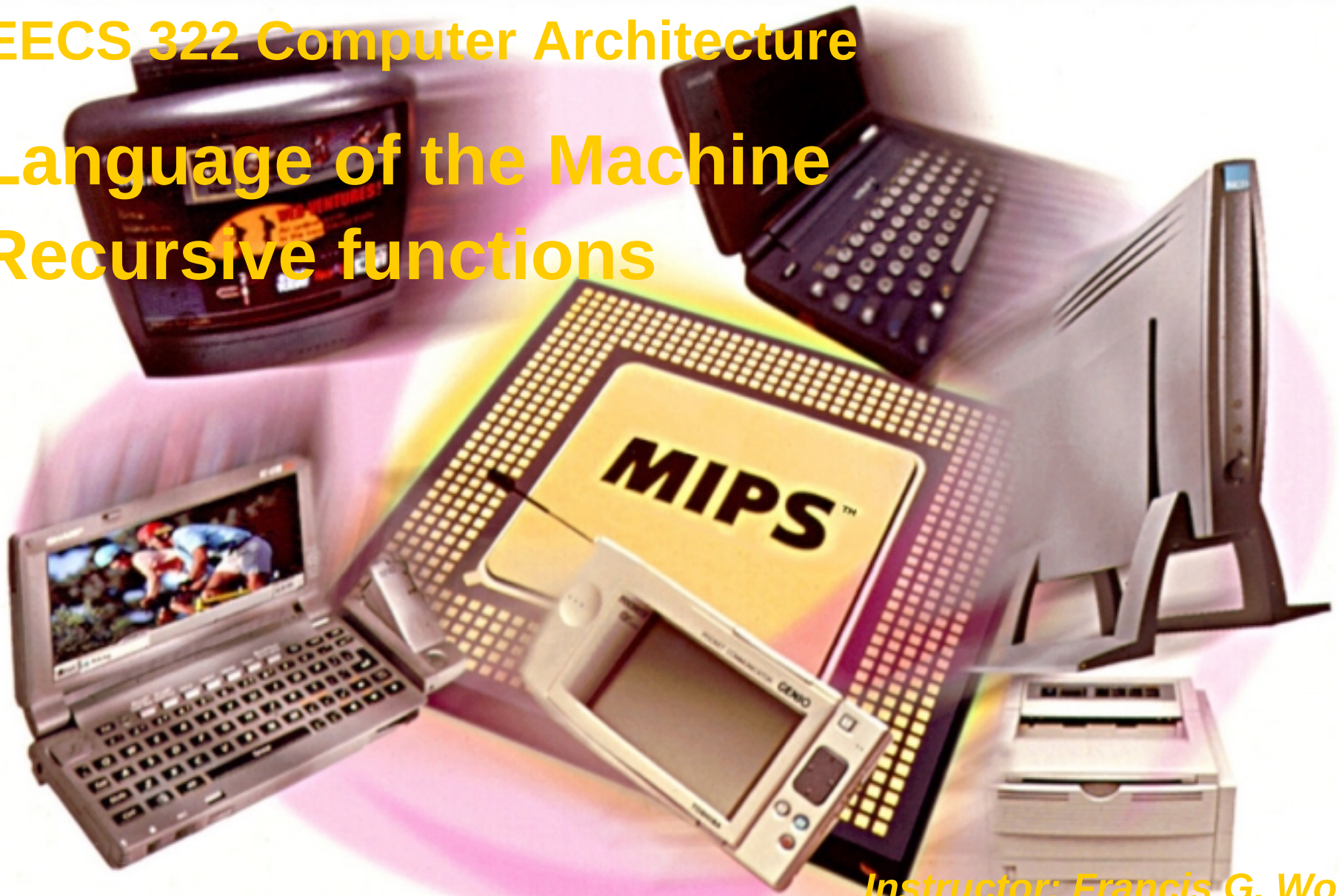
EECS 322 Computer Architecture

Language of the Machine
Recursive functions



Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University

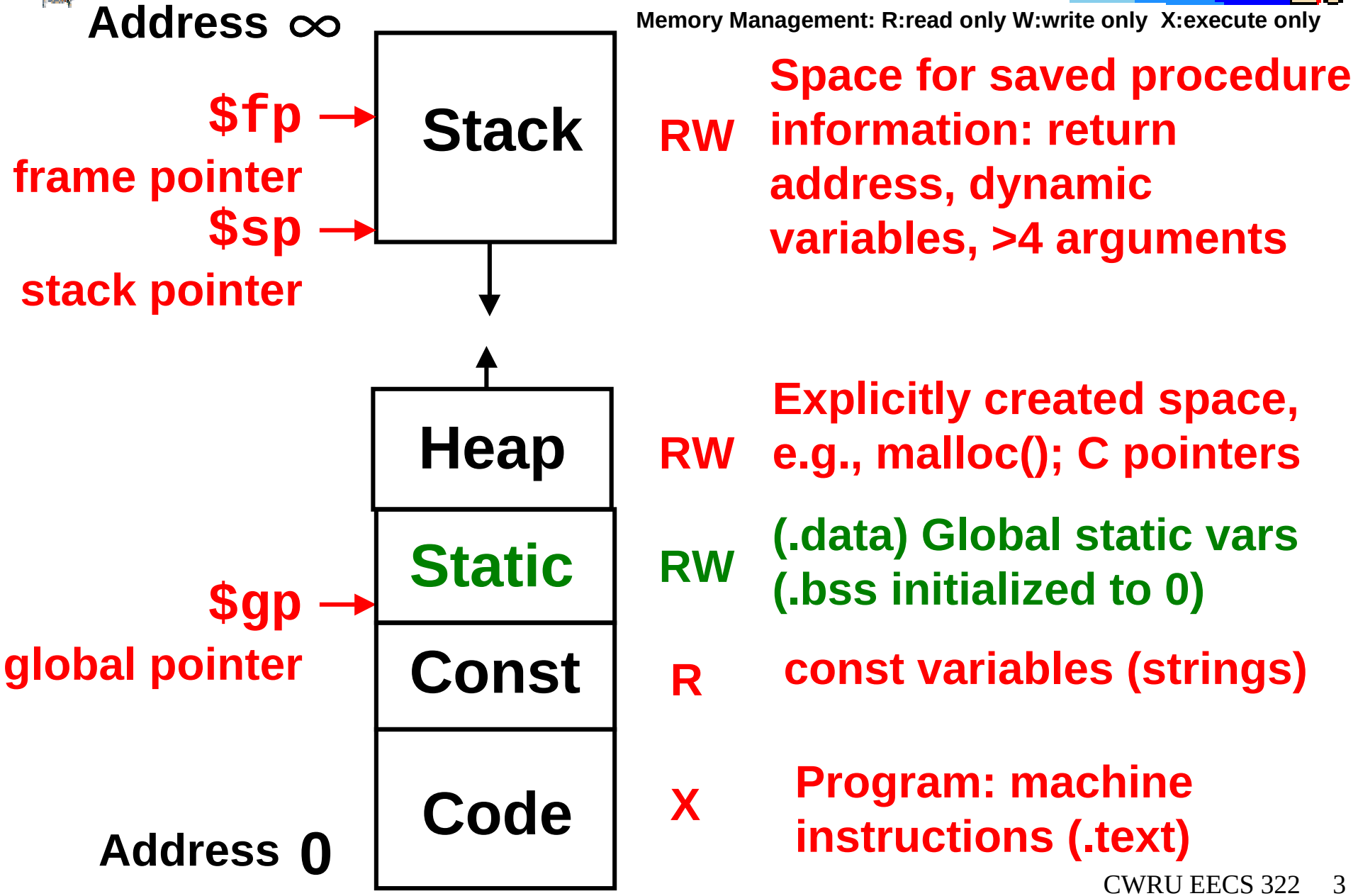
This presentation uses powerpoint animation: please viewshow

Review: Function calling

- Follow calling conventions & nobody gets hurt.

- Function Call Bookkeeping:
  - Caller:
    - Arguments $a0, $a1, $a2, $a3
    - Return address $ra
    - Call function jal label # $ra=pc+4;pc=label
  - Callee:
    - Not restored $t0 - $t9
    - Restore caller's $s0 - $s7, $sp, $fp
    - Return value $v0, $v1
    - Return jr $ra # pc = $ra

Review: Program memory layout

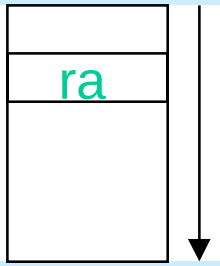


Basic Structure of a Function

Prologue

```
    entry_label:
    addi $sp,$sp,-framesize
    sw   $ra,framesize-4($sp)# save $ra
    save other regs
```

Body ….

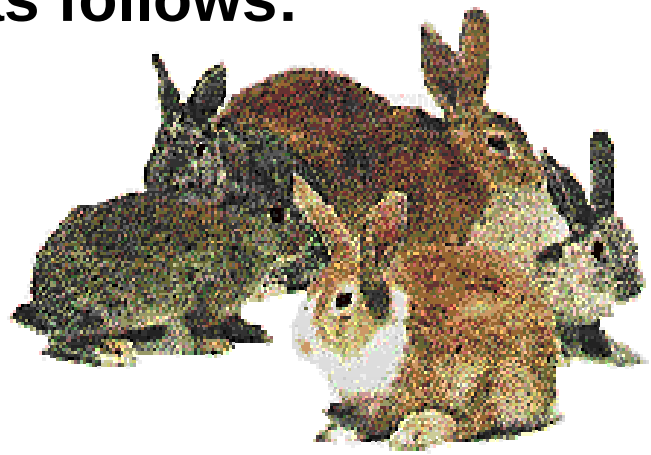


Epilogue

```
    restore other regs
    lw   $ra, framesize-4($sp)#restore $ra
    addi $sp,$sp, framesize
    jr   $ra
```

Recursive functions: Fibonacci Numbers

- **How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the 2nd month on becomes productive.**
  Leonardo Pisano aka Fibonacci (1202, Pisa, Italy)
- **The Fibonacci numbers are defined as follows:**
  - **F(n) = F(n – 1) + F(n – 2),**
  - **F(0) and F(1) are defined to be 1**
- **Re-writing this in C we have:**



```
int fib(int n) {
    if(n == 0) { return 1; }
    if(n == 1) { return 1; }
    return (fib(n - 1) + fib(n - 2));
}
```

Prologue: Fibonacci Numbers

- Now, let's translate this to MIPS!
- Reserve 3 words on the stack: $ra, $s0, $a0
- The function will use one $s register, $s0
- Write the Prologue:

```
fib:
 addi $sp, $sp, -12    # Space for three words
 sw $ra, 8($sp)        # Save the return address
 sw $s0, 4($sp)        # Save $s0
```

Epilogue: Fibonacci Numbers

- Now write the Epilogue:

```
fin:
 lw $s0, 4($sp)          # Restore caller's $s0
 lw $ra, 8($sp)          # Restore return address
 addi $sp, $sp, 12       # Pop the stack frame
 jr $ra                  # Return to caller
```

Body: Fibonacci Numbers

- Finally, write the body. The C code is below. Start by translating the lines indicated in the comments

```
int fib(int n) {
    if(n == 0) { return 1; } /*Translate Me!*/
    if(n == 1) { return 1; } /*Translate Me!*/
    return fib(n - 1) + fib(n - 2);

}
```

```
addi    $v0,$zero,1     # $v0 = 1; return $v0
beq     $a0,$zero,fin   # if (n == 0) goto fin
addi    $t0,$zero,1     # $t0 = 1;
beq     $a0,$t0,fin     # if (n == $t0)goto fin
# Contiued on next slide.  .  .
```

return: Fibonacci Numbers

- Almost there, but be careful, this part is tricky!

```
int fib(int n) {
    . . .
    return (fib(n - 1) + fib(n - 2));
}
```

```
sw   $a0,0($sp)      # Need $a0 after jal
addi $a0,$a0, -1     # $a0 = n - 1
jal  fib             # fib($a0)
add  $s0,$v0,$zero   # $s0 = fib(n-1)
lw   $a0,0($sp)      # Restore original $a0 = n
addi $a0,$a0, -2     # $a0 = n – 2
jal  fib             # fib($a0)
add  $v0,$s0,$v0     # fib(n-1) + fib(n-2)
```

return: $s1 improvement?

- Can we replace the `sw $a0,0($sp)`
- with `add $s1,$a0,$zero`
- in order to avoid using the stack?

```
add  $s1,$a0,$zero   # was sw  $a0,0($sp)
addi $a0,$a0, -1     # $a0 = n - 1
jal  fib             # fib($a0)
add  $s0,$v0,$zero   # $s0 = fib(n-1)
                     # was lw  $a0,0($sp)
addi $a0,$s1, -2     # $a0 = n – 2
jal  fib             # fib($a0)
add  $v0,$s0,$v0     # fib(n-1) + fib(n-2)
```

return: $s1 improvement...

- Can we replace the `sw $a0,0($sp)`
- with `add $s1,$a0,$zero`
- in order to avoid using the stack?
- We did save one instruction so far, a plus!
- By convention, all $s registers must be preserved for the caller.
- Thus, modifying $s1 will confuse the caller.
- and therefore we would have to add another lw and sw for $s1 in the prologue and epilog.
- The saving of instruction, requires 2 new instructions; resulting in a net gain of minus one!

return: $t1 improvement?

- Can we replace the `sw $a0,0($sp)`
- with `add $t1,$a0,$zero`
- in order to avoid using the stack?

- We did save one instruction so far, a plus!
- By convention, all $t registers are not preserved for the caller.
- and therefore we would have to add another lw and sw for $t1 to the stack.

Here's the complete code: Fibonacci Numbers

```
fib:
addi $sp, $sp, -12
sw   $ra, 8($sp)
sw   $s0, 4($sp)

addi $v0, $zero, 1
beq  $a0, $zero, fin
addi $t0, $zero, 1
beq  $a0, $t0, fin
sw   $a0, 0($sp)
addi $a0, $a0, -1
jal  fib
add $s0, $v0, $zero
```

```
lw   $a0, 0($sp)
addi $a0, $a0, -2
jal  fib
add  $v0, $v0, $s0

fin: # epilog
lw   $s0, 4($sp)
lw   $ra, 8($sp)
addi $sp, $sp, 12
jr   $ra
```

Time Complexity: recursive

- **The Fibonacci numbers are defined as follows:**
  - **F(n) = F(n – 1) + F(n – 2),**
  - **F(0) and F(1) are defined to be 1**

- **Let T(n) be the number of steps required to calculate F(n).**
- **Then we can set up a recurrence relation for T(n)**
- **Note that call=return=compare=arithmetic=1 time unit**
- **Step 1: initial call F(n) =1**
- **In F: 1. If n<=1 ret 1 =2**
  2. else ret F(n-1)+F(n-2) =6+T(n-1)+T(n-2)

The approximate solution is T(n) > O(1.5^n)
exponential growth! want to avoid that!

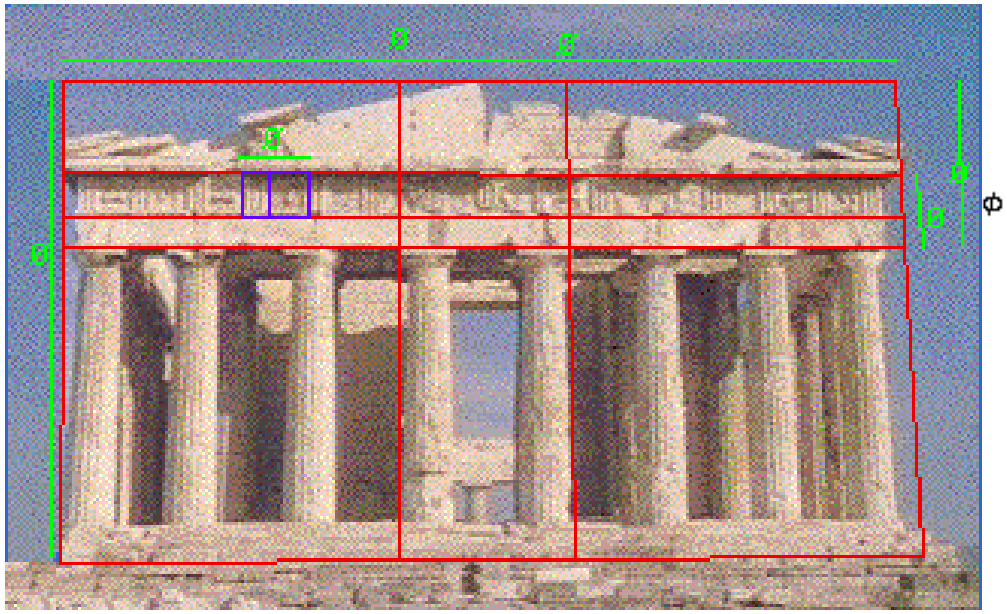
Time Complexity: non-recursive

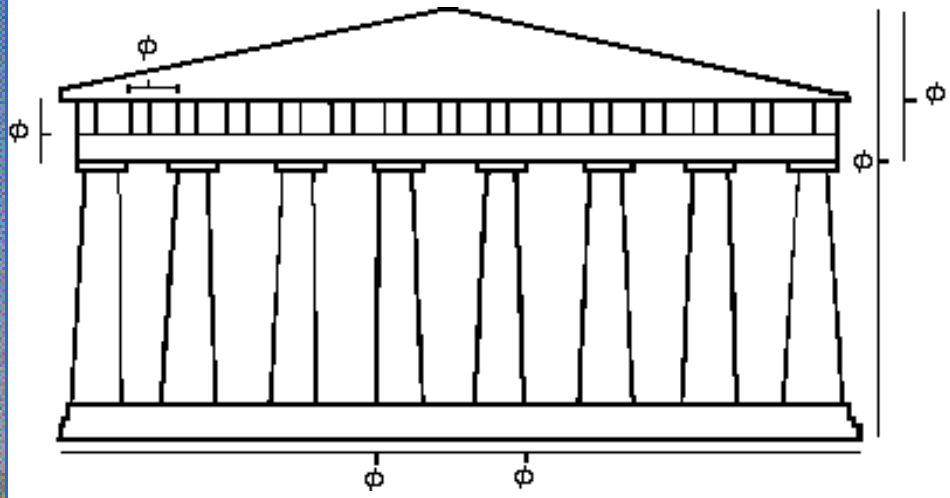
- non-recursive version
  - t1=1; t2=2;
  - for(i=2; i > n; i++) { t3=t2+t1; t1=t2; t2=t3; } return t3;
  - T(n) = 7(n-1)+4 = 7n - 3 = O(7n - 3) = O(n)
  - O(n) Linear in Time! better than the recursive version!
  - This shows that the algorithm applied to the target architecture has the greatest impact on performance issues.
  - C/C++ compilers are not the answer to everything.
  - Thinking out the problem beforehand is.

Closed form

- $F(n) = 1/\sqrt{5}\,(\Phi^n - \hat{\Phi}^n)$
- where the golden ratio $\Phi = (1+\sqrt{5})/2$ and $\hat{\Phi} = 1-\Phi$

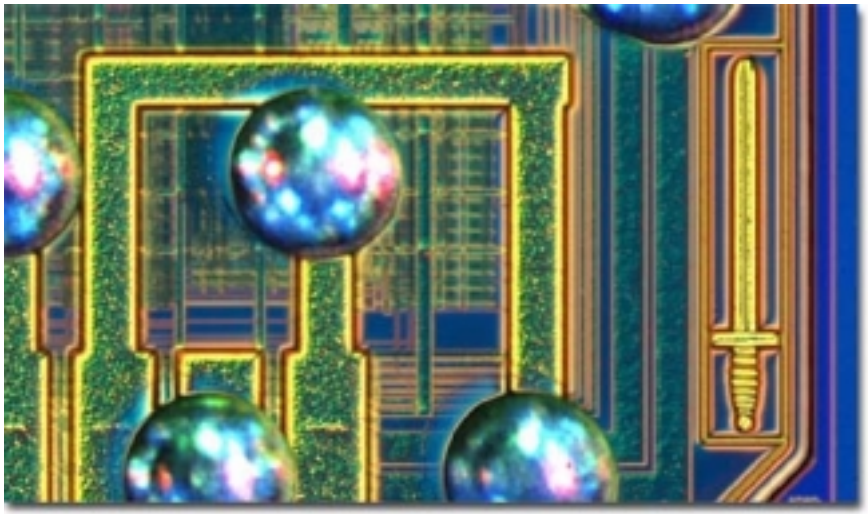
- Even from the time of the Greeks, the golden proportion has been used in architecture. The most famous is the Parthenon build circa 430 BC.





Signatures and Silicon Art

- Just as the great architects, place their hidden Φ signature, so too do computer designers.



The “Pentium Killer” Macintosh G3 chips were code-named "Arthur" as in Camelot, and the sword represents Excalibur.

Motorola/IBM PowerPC 750

MIPS R10000 Processor

Argument Passing greater than 4

- C code fragment

```
g=f(a, b, c, d, e);
```

- MIPS assembler

```
addi    $sp, $sp, -4
sw      $s4, 0($sp)   # push e
add     $a3, $s3, $0  # register push d
add     $a3, $s2, $0  # register push c
add     $a1, $s1, $0  # register push b
add     $a0, $s0, $0  # register push a
jal     f             # $ra = pc + 4
add     $s5, $v0, $0  # g=return value
```

Argument Passing Options

- **2 common choices**
  - **"Call by Value": pass a copy of the item to the function/procedure**
  - **"Call by Reference": pass a pointer to the item to the function/procedure**
- **Single word variables passed by value**
- **Passing an array? e.g., a[100]**
  - **Pascal--call by value--copies 100 words of a[] onto the stack: inefficient**
  - **C--call by reference--passes a pointer (1 word) to the array a[] in a register**

Memory Allocation

- int *sumarray(int x[ ], int y[ ])
- adds two arrays and puts sum in a third array
- 3 versions of array function that
  - Dynamic allocation (stack memory)
  - Static allocation (global memory)
  - Heap allocation (malloc, free)

- Purpose of example is to show interaction of C statements, pointers, and memory allocation

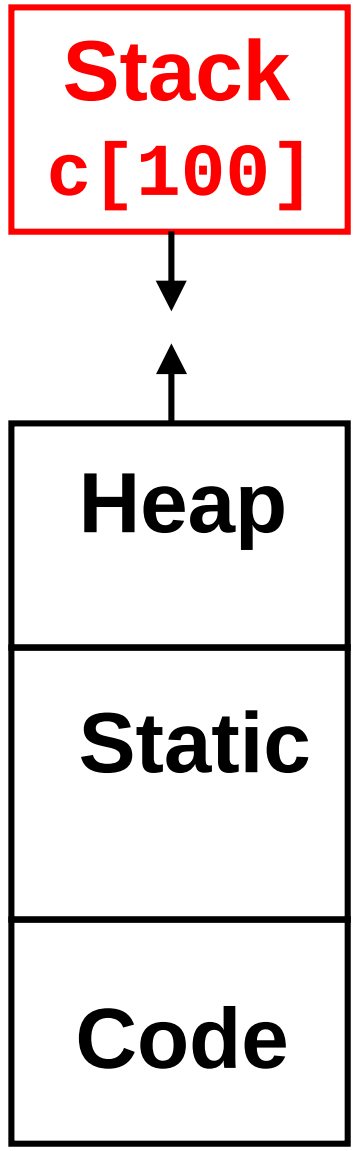
Dynamic Allocation

- **Caller provides temporary work space**

```
int f(int x[100], y[100], ) {
    int c[100];
    sumarray(x, y, c);
    . . .
```

- **C calling convention means above the same as**

```
    sumarray(&x[0], &y[0], &c[0]);
```



Optimized Compiled Code

```
void sumarray(int a[],int b[],int c[]) {
  int i;
  for(i=0;i<100;i=i+1)
     c[i] = a[i] + b[i];
}
```

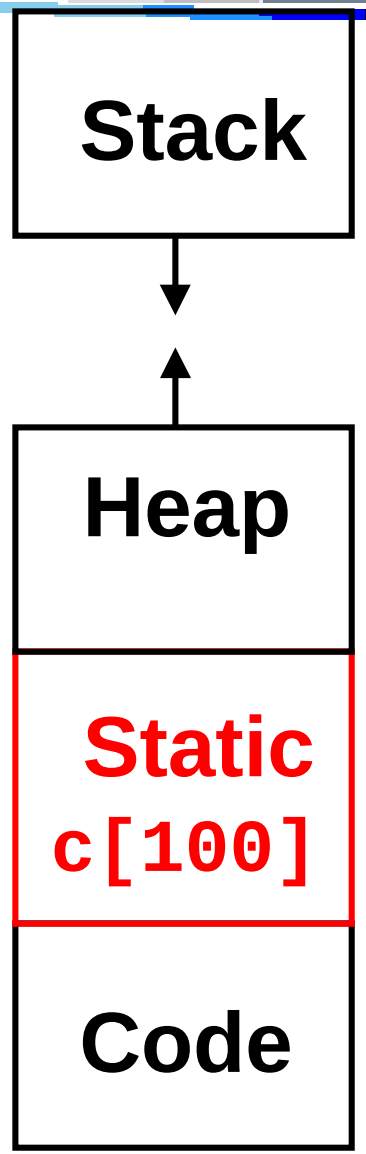
```
      addi  $t0,$a0,400    # beyond end of a[]
Loop: beq   $a0,$t0,Exit   # if (i==sizeof(int)* 100)
      lw    $t1, 0($a0)    # $t1=a[i]
      lw    $t2, 0($a1)    # $t2=b[i]
      add   $t1,$t1,$t2    # $t1=a[i] + b[i]
      sw    $t1, 0($a2)    # c[i]=a[i] + b[i]
      addi  $a0,$a0,4      # $a0++
      addi  $a1,$a1,4      # $a1++
      addi  $a2,$a2,4      # $a2++
      j     Loop
Exit: jr    $ra
```

Static allocation (scope: private to function only)

- Static declaration

```
int *sumarray(int a[],int b[]) {
    int i;
    static int c[100];

    for(i=0;i<100;i=i+1)
        c[i] = a[i] + b[i];
    return c;
}
```



- Compiler allocates once for function, space is reused by function
  - On re-entry will still have old data
  - Can not be seen by outside functions

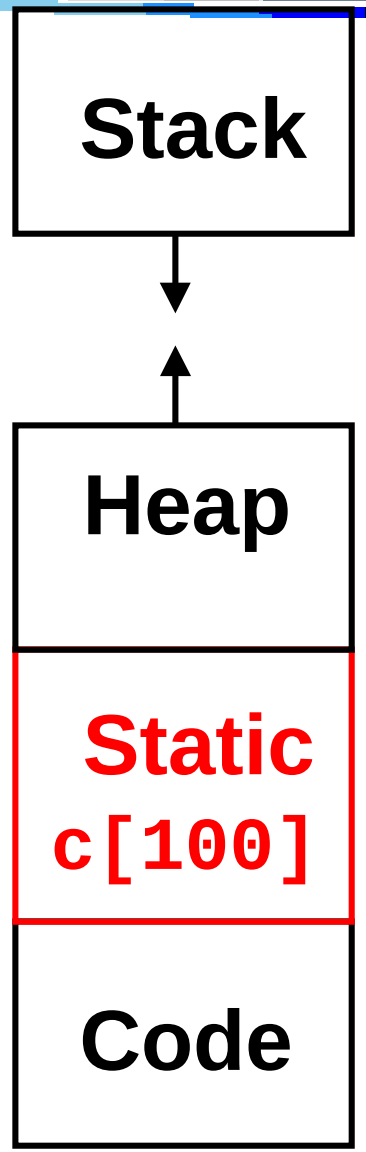
Alternate Static allocation (scope: public to everyone)

- Static declaration

```
int c[100];
int *sumarray(int a[],int b[]) {
    int i;
    static int c[100];

    for(i=0;i<100;i=i+1)
          c[i] = a[i] + b[i];
    return c;
}
```

- The variable scope of c is very public and is accessible to everyone outside the function



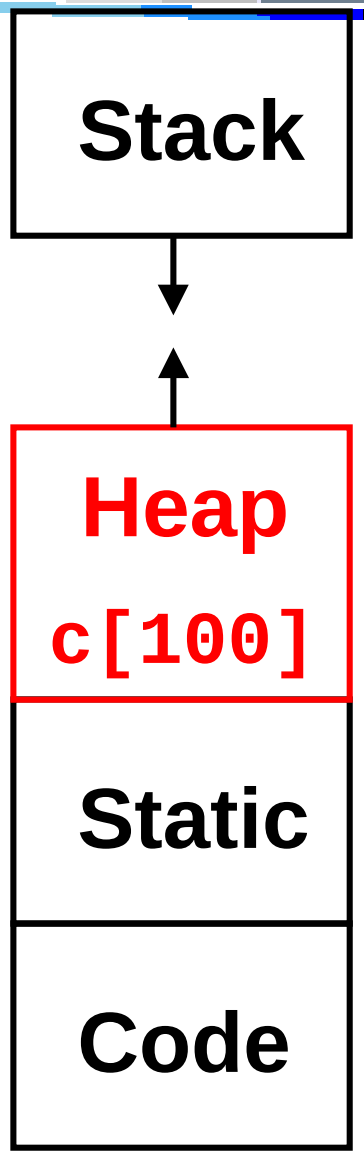
Heap allocation

- **Solution: allocate c[] on heap**

```
int * sumarray(int a[],int b[]) {
    int i;
    int *c;
    c = (int *) malloc(100);

    for(i=0;i<100;i=i+1)
        c[i] = a[i] + b[i];
    return c;
}
```

- **Not reused unless freed**
  - **Can lead to memory leaks**
  - **Java, Scheme have garbage collectors to reclaim free space**



Lifetime of storage & scope

- automatic (stack allocated)
  - typical local variables of a function
  - created upon call, released upon return
  - scope is the function
- heap allocated
  - created upon malloc, released upon free
  - referenced via pointers
- external / static
  - exist for entire program

What about Structures?

- Scalars passed by value (i.e. int, float, char)
- Arrays passed by reference (pointers)
- Structures by value (struct { … })
- Pointers by value
- Can think of C passing everything by value, just that arrays are simply a notation for pointers

Register Names as Numbers (page A-23)

• Register Names	Register No.
–$zero	$0
–$at (reserved for assembler)	$1
–(Return) Value registers ($v0,$v1)	$2 - $3
–Argument registers ($a0-$a3)	$4 - $7
–Temporary registers ($t0-$t7)	$8 - $15
–Saved registers ($s0-$s7)	$16 - $22
–Temporary registers ($t8-$t9)	$23 - $24
– $k0,$k1 (reserved for OS kernel)	$26, $27
–Global Pointer ($gp)	$28
–Stack Pointer ($sp)	$29
–Frame Pointer ($fp), or $t10	$30
–Return Address ($ra)	$31