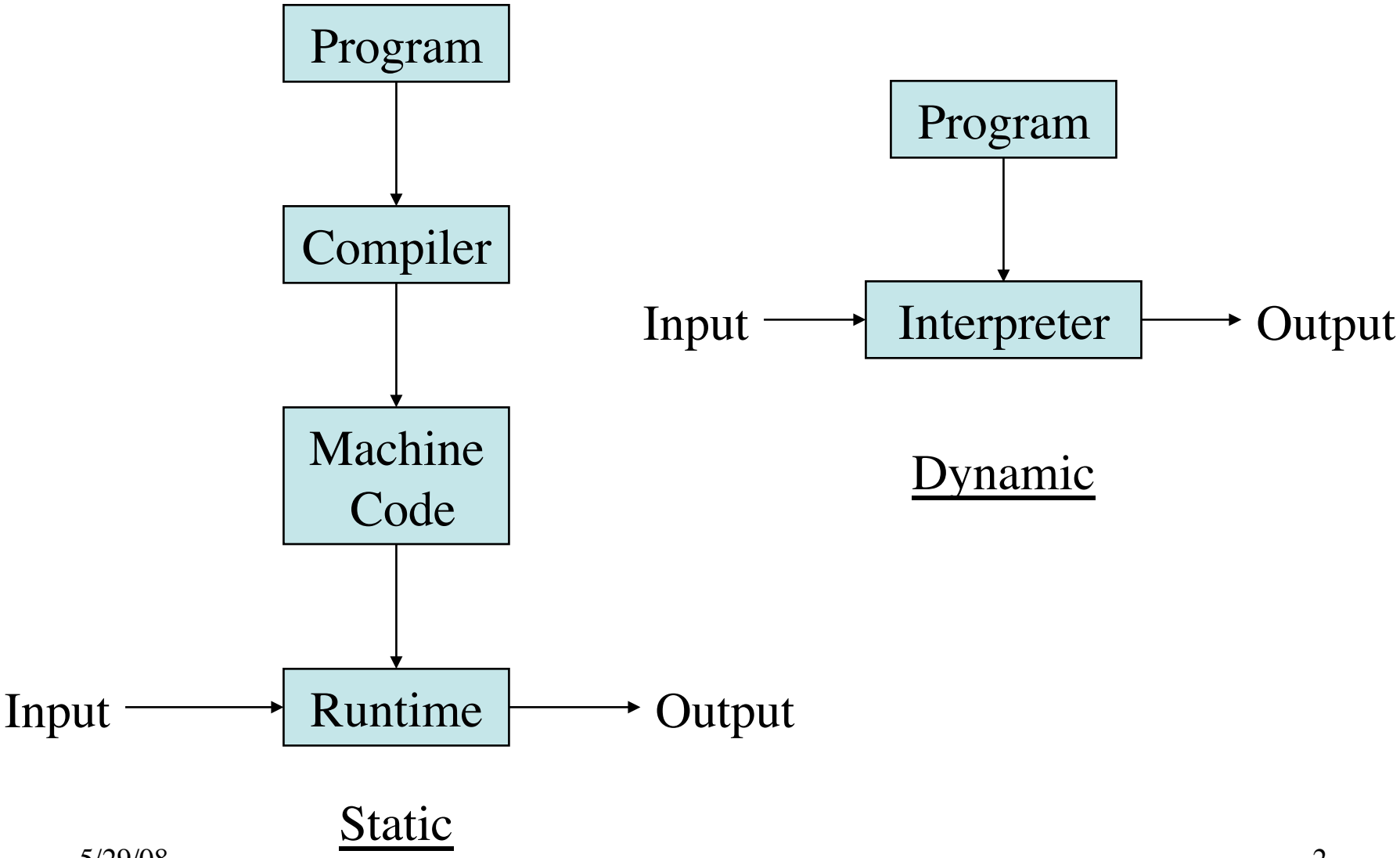
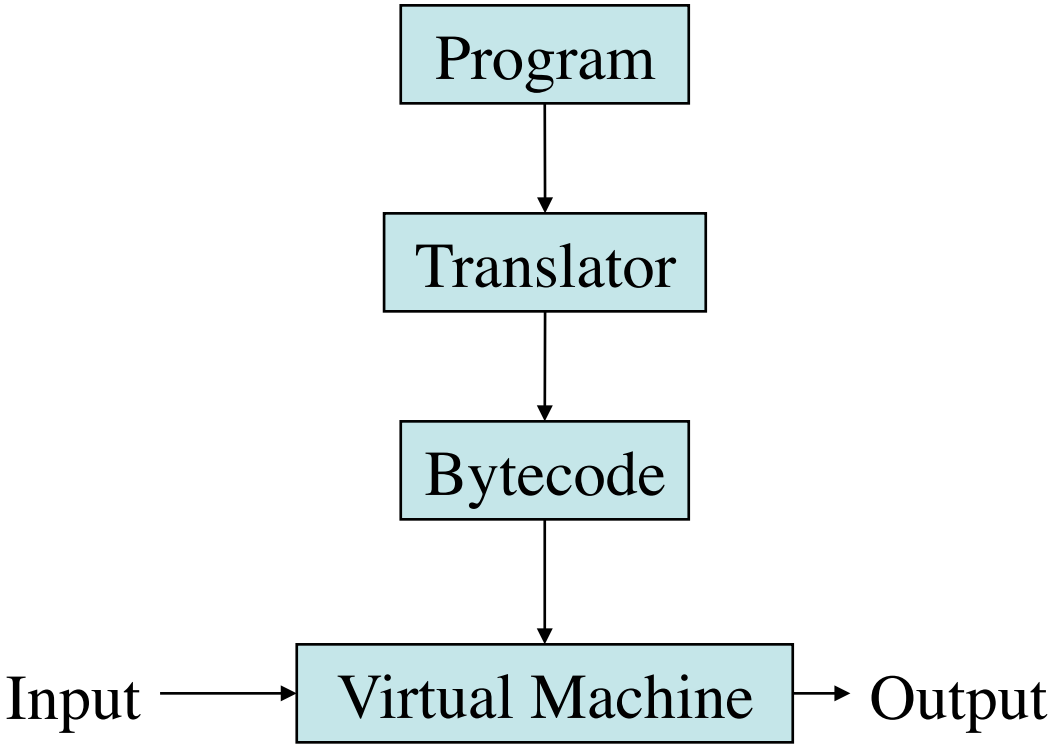
CMPT 300
Operating Systems

Guest Lecture on Function Calls

Anoop Sarkar

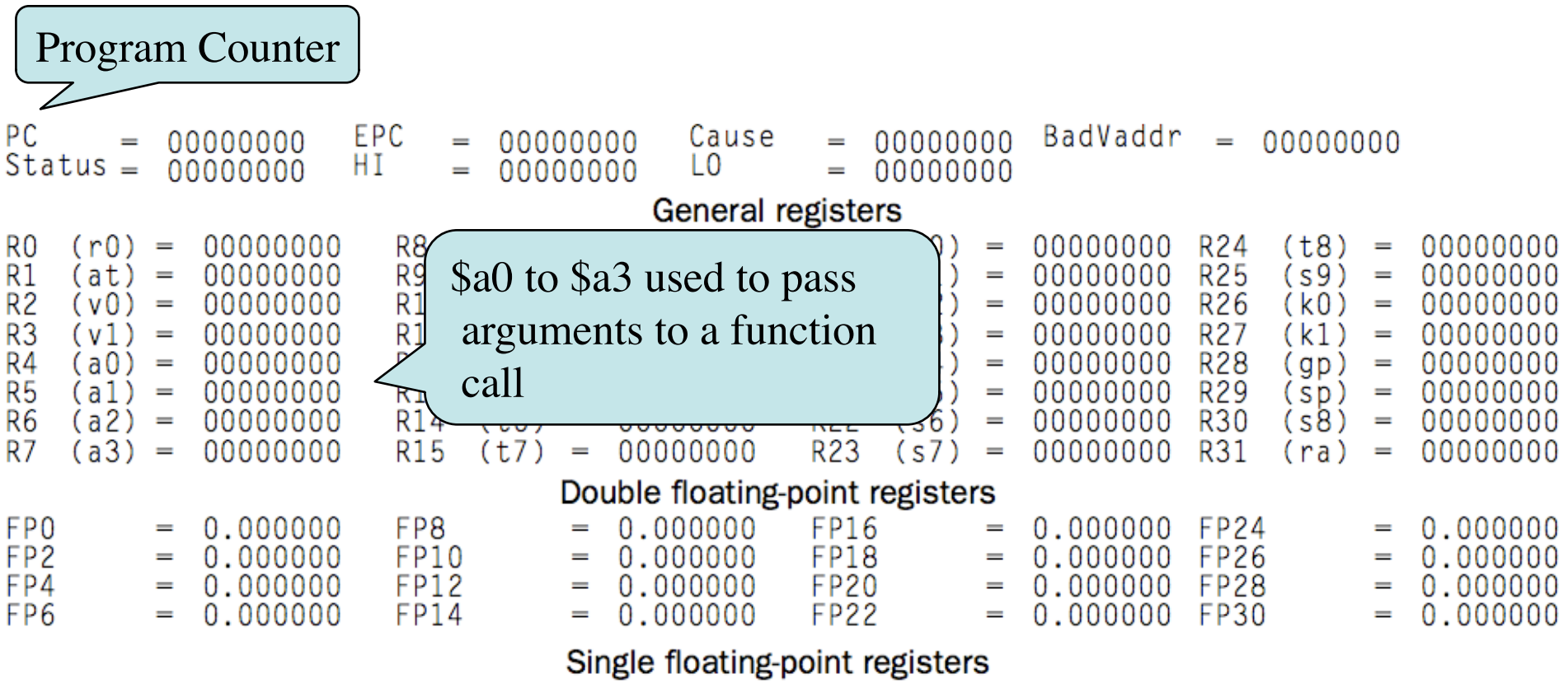
http://www.cs.sfu.ca/~anoop


Program
Compiler
Machine Code
Input
Runtime
Output
Static
Program
Input
Interpreter
Output
Dynamic



Static/Dynamic

MIPS CPU


Program Counter
PC = 00000000 EPC = 00000000 Cause = 00000000 BadVaddr = 00000000
Status = 00000000 HI = 00000000 LO = 00000000
General registers
$a0 to $a3 used to pass arguments to a function call
R0 (r0) = 00000000 R24 (t8) = 00000000
R1 (at) = 00000000 R25 (s9) = 00000000
R2 (v0) = 00000000 R26 (k0) = 00000000
R3 (v1) = 00000000 R27 (k1) = 00000000
R4 (a0) = 00000000 R28 (gp) = 00000000
R5 (a1) = 00000000 R29 (sp) = 00000000
R6 (a2) = 00000000 R30 (s8) = 00000000
R7 (a3) = 00000000 R15 (t7) = 00000000 R23 (s7) = 00000000 R31 (ra) = 00000000
Double floating-point registers
FP0 = 0.000000 FP8 = 0.000000 FP16 = 0.000000 FP24 = 0.000000
FP2 = 0.000000 FP10 = 0.000000 FP18 = 0.000000 FP26 = 0.000000
FP4 = 0.000000 FP12 = 0.000000 FP20 = 0.000000 FP28 = 0.000000
FP6 = 0.000000 FP14 = 0.000000 FP22 = 0.000000 FP30 = 0.000000
Single floating-point registers

MIPS CPU

Text segments

```
[0x00400000]    0x8fa40000    lw $4, 0($29)              ; 89: lw $a0, 0($sp)
[0x00400004]    0x27a50004    addiu $5, $29, 4           ; 90: addiu $a1, $sp, 4
[0x00400008]    0x24a60004    addiu $6, $5, 4            ; 91: addiu $a2, $a1, 4
[0x0040000c]    0x00041080    sll $2, $4, 2              ; 92: sll $v0, $a0, 2
[0x00400010]    0x00c23021    addu $6, $6, $2            ; 93: addu $a2, $a2, $v0
[0x00400014]    0x0c000000    jal 0x00000000 [main]      ; 94: jal main
[0x00400018]    0x3402000a    ori $2, $0, 10             ; 95: li $v0 10
[0x0040001c]    0x0000000c    syscall                    ; 96: syscall
```

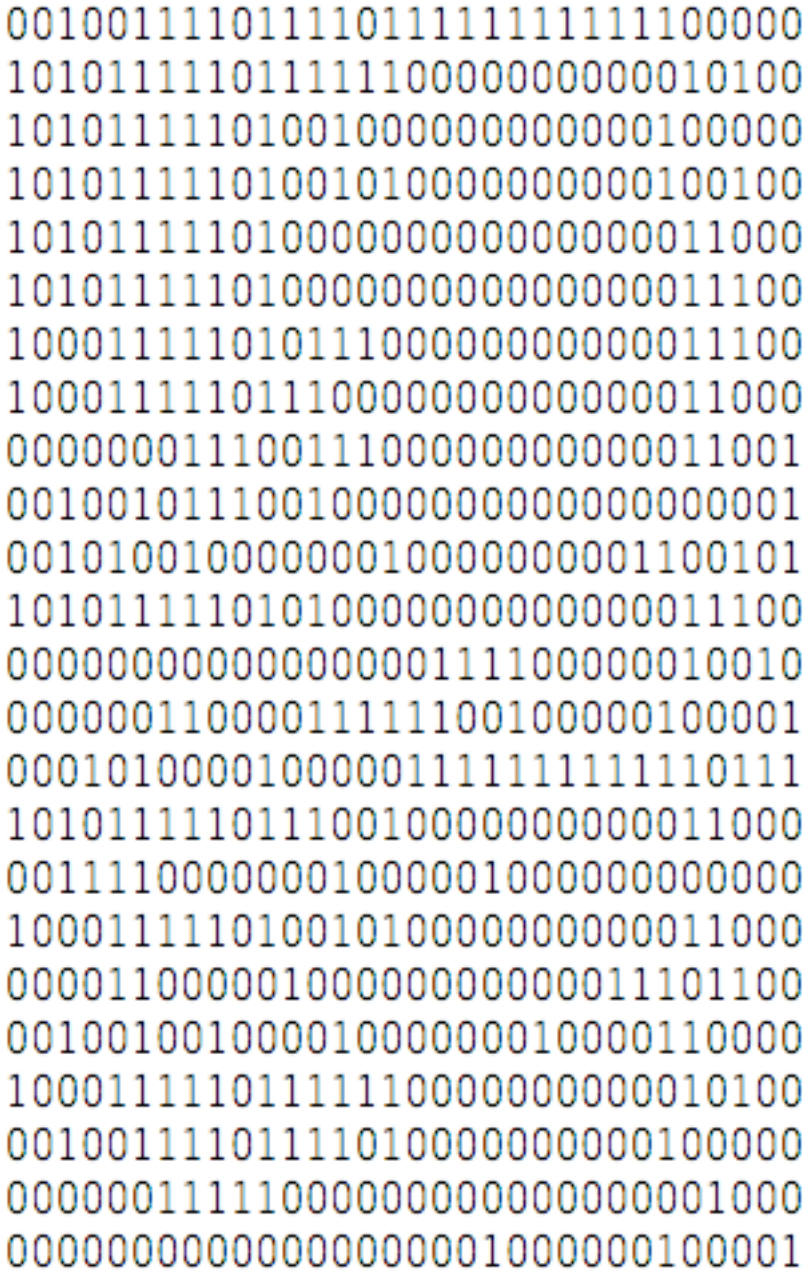
Data segments

```
[0x10000000] ... [0x10010000]   0x00000000
[0x10010004]    0x74706563    0x206e6f69    0x636f2000
[0x10010010]    0x72727563    0x61206465    0x6920646e    0x726f6e67
[0x10010020]    0x000a6465    0x495b2020    0x7265746e    0x74707572
[0x10010030]    0x0000205d    0x20200000    0x616e555b    0x6e67696c
[0x10010040]    0x61206465    0x65726464    0x69207373    0x6e69206e
[0x10010050]    0x642f7473    0x20617461    0x63746566    0x00205d68
[0x10010060]    0x555b2020    0x696c616e    0x64656e67    0x64646120
[0x10010070]    0x73736572    0x206e6920    0x726f7473    0x00205d65
```

What we understand

```
#include <stdio.h>

int main (int argc, char *argv[]) {
      int i;
      int sum = 0;
      for (i = 0; i <= 100; i++)
          sum = sum + i * i;
      printf (”Sum from 0..100 = %d\n", sum);
}
```



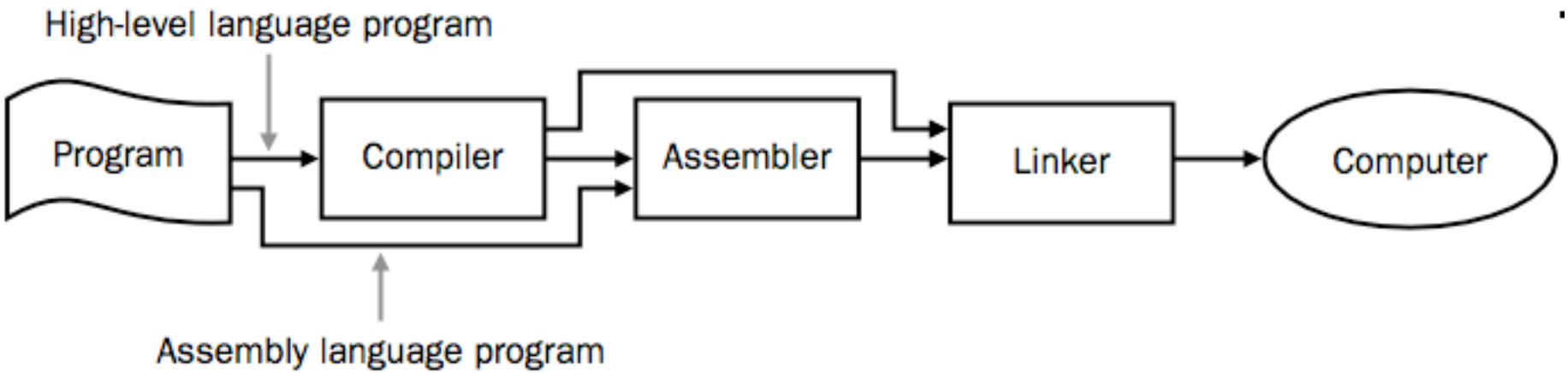
Conversion into machine instructions

MIPS
machine language
code

Assembly language

```
 .text
.align 2
.globl main
main:
  subu $sp, $sp, 32
  sw $ra, 20($sp)
  sd $a0, 32($sp)
  sw $0, 24($sp)
  sw $0, 28($sp)
loop:
  lw $t6, 28($sp)
  mul $t7, $t6, $t6
  lw $t8, 24($sp)
  addu $t9, $t8, $t7
  sw $t9, 24($sp)
  addu $t0, $t6, 1
  sw $t0, 28($sp)
  ble $t0, 100, loop
  la $a0, str
  lw $a1, 24($sp)
  jal printf
  move $v0, $0
  lw $ra, 20($sp)
  addu $sp, $sp, 32
  jr $ra
.data
.align 0
str:
.asciiz "The sum from 0 .. 100 is %d\n"
```

A one-one translation from assembly to machine code

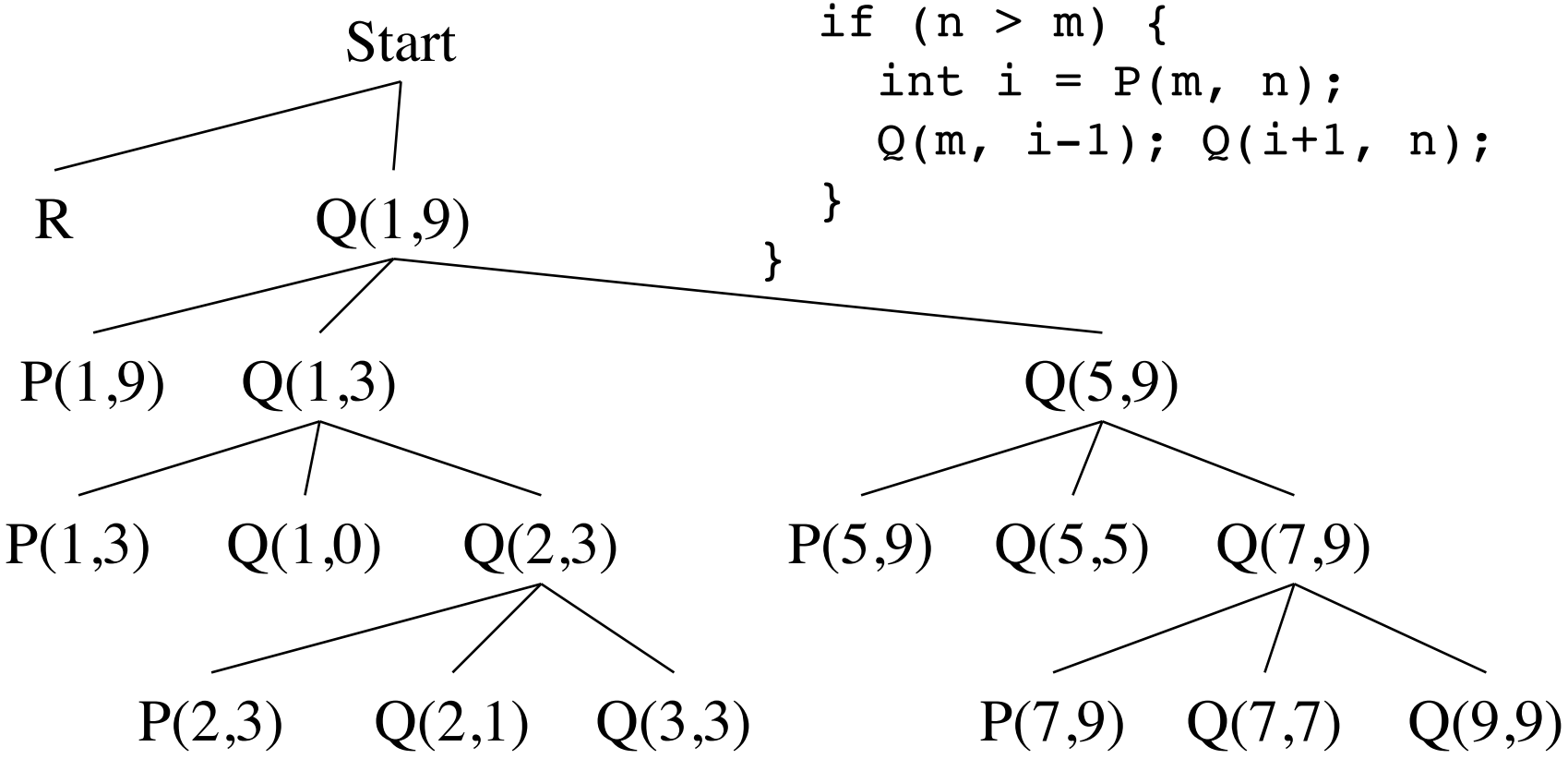

High-level language program
Program
Compiler
Assembler
Linker
Computer
Assembly language program

Function Calls: Activation Trees

- An activation of a function is a particular invocation of that function
- Each activation will have particular values for the function parameters
- Each activation can call another activation before it becomes inactive
- The sequence of function calls can be represented as an *activation tree*

Activation Tree

```
int Q(int m, int n) {
  if (n > m) {
    int i = P(m, n);
    Q(m, i-1); Q(i+1, n);
  }
}
```



Problems with Functions

- Recursive functions
- If a function has local variables, and if it calls another function: local variables must be restored after control returns back
- Function can access non-local (global) variables
- Parameter passing into a function

Activation Records

- Information for a single execution of a function is called an *activation record* or *procedure call frame*
- A frame contains:
  - Temporary local register values for caller
  - Local data
  - Snapshot of machine state (important registers)
  - Return address
  - Link to global data
  - Parameters passed to function
  - Return value for the caller

Storage Allocation for Functions

- Static Allocation
  - Layout all storage for all data objects at compile time
  - Essentially every variable is stored globally
  - But the compiler can still control local activation and de-activation of variables
  - Very restricted recursion is allowed
  - Fortran 77

Storage Allocation for Functions

- Stack Allocation √
  - Storage for recursive functions is organized as a stack: last-in first-out (LIFO) order
  - Activation records are associated with each function activation
  - Activation records are pushed onto the stack when a call is made to the function
  - Size of activation records can be fixed or variable

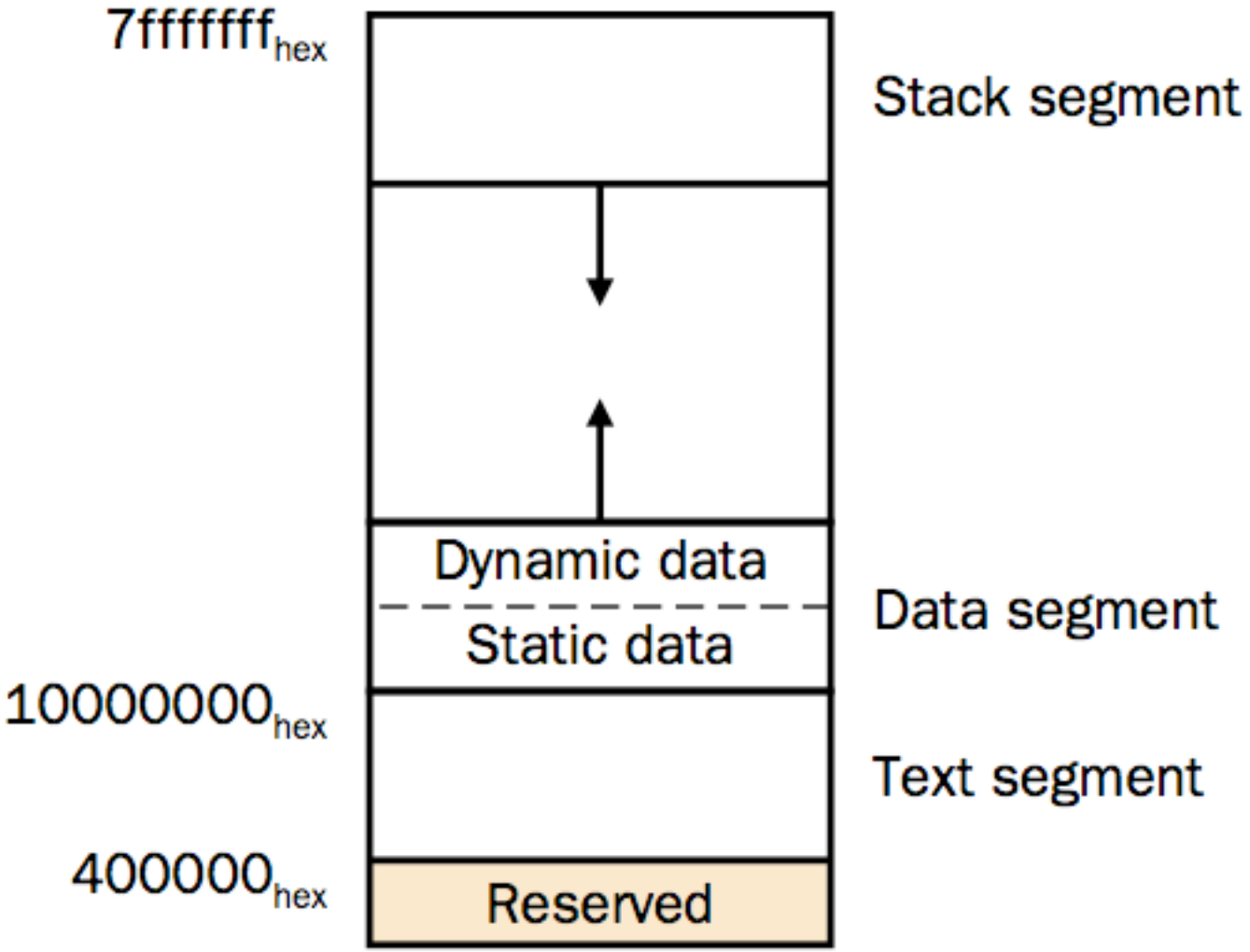
Storage Allocation for Functions

- Stack Allocation √
  - Sometimes a minimum size is required
  - Variable length data is handled using pointers
  - Locals are deleted after activation ends
  - Caller locals are reinstated and execution continues
  - C, Pascal and most modern programming languages

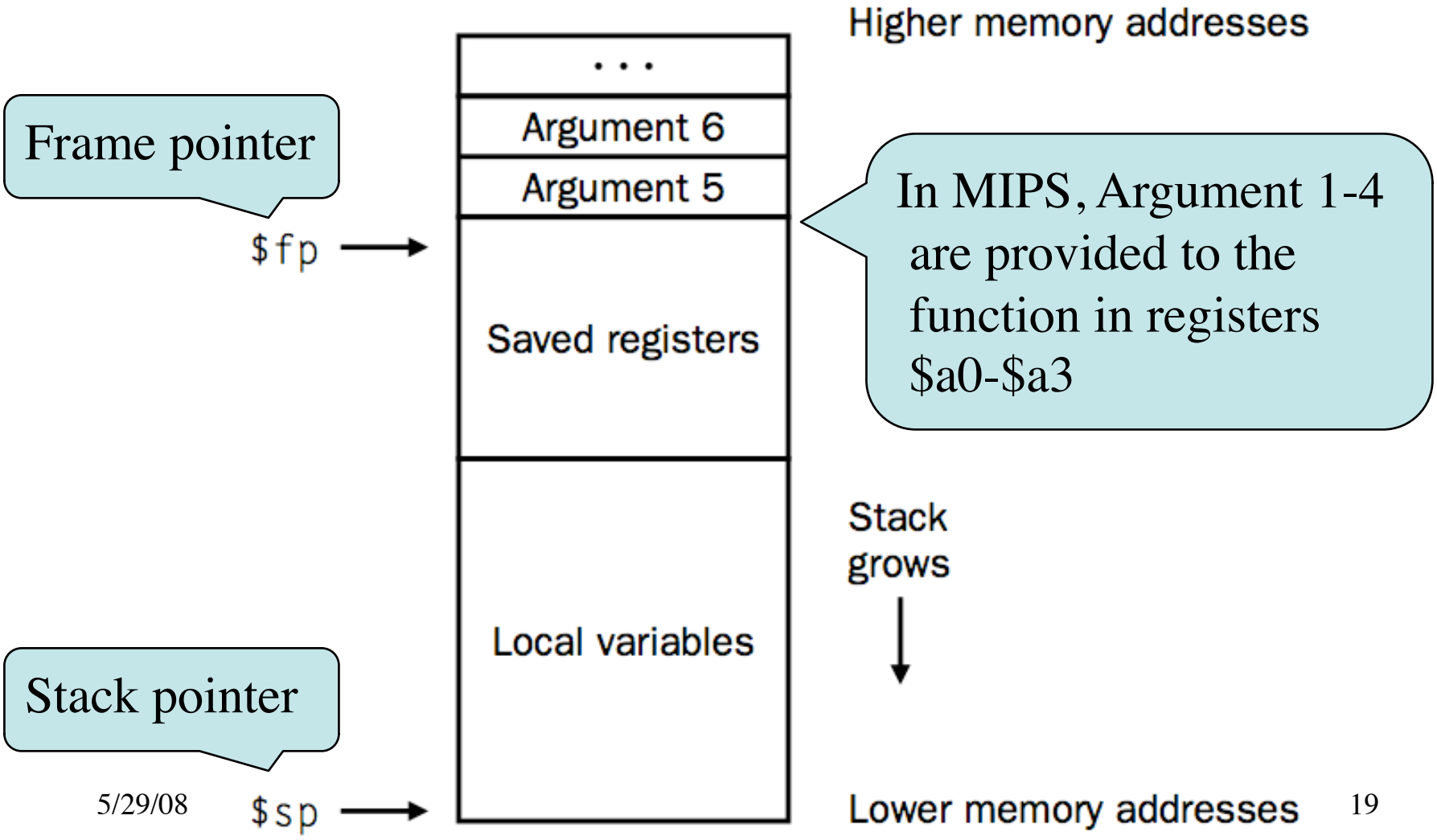
Storage Allocation for Functions

- Heap Allocation
  - In some special cases stack allocation is not possible
  - If local variables must be retained after the activation ends
  - If called activation outlives the caller
  - Anything that violates the last-in first-out nature of stack allocation e.g. closures in Lisp and other functional programming languages

Run-time Memory



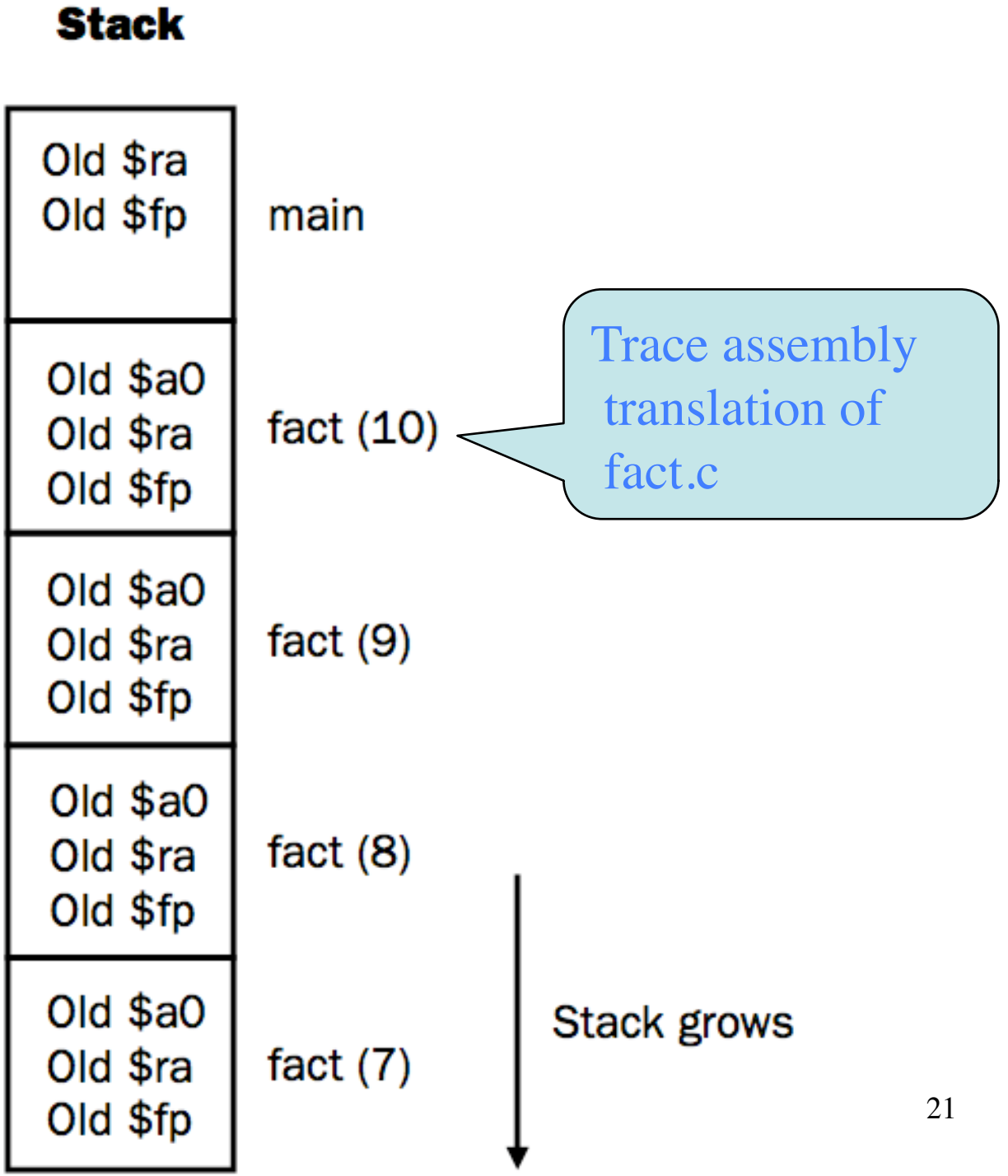
Stack frame


Higher memory addresses
. . .
Argument 6
Argument 5
Frame pointer
$fp
Saved registers
In MIPS, Argument 1-4 are provided to the function in registers $a0-$a3
Stack grows
Local variables
Stack pointer
$sp
Lower memory addresses

```
#include <stdio.h>

main ()
{
    int n = 10;
    printf("The factorial of 10 is %d\n", fact(n));
}

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


Stack
Old $ra
Old $fp
main
Old $a0
Old $ra
Old $fp
fact (10)
Old $a0
Old $ra
Old $fp
fact (9)
Old $a0
Old $ra
Old $fp
fact (8)
Old $a0
Old $ra
Old $fp
fact (7)
Stack grows
Trace assembly translation of fact.c

Parameter Passing Conventions

- Differences based on:
  - The parameter represents an r-value (the rhs of an expr)
  - An l-value (a location where a value can be stored)
  - Or the text of the parameter itself
- Call by Value
  - Each parameter is evaluated
  - Pass the r-value to the function
  - No side-effect on the parameter

Parameter Passing Conventions

- Call by Reference
  - Also called call by address/location
  - If the parameter is a name or expr that is an l-value then pass the l-value
  - Else create a new temporary l-value and pass that
  - Typical example: passing array elements a[i]

Parameter Passing Conventions

- Copy Restore Linkage
  - Pass only r-values to the called function (but keep the l-value around for those parameters that have it)
  - When control returns back, take the r-values and copy it into the l-values for the parameters that have it
  - Fortran
- Call by Name
  - Function is treated like a macro (a #define) or in-line expansion
  - The parameters are literally re-written as passed arguments (keep caller variables distinct by renaming)

Summary

- The CPU provides run-time support for functions
- Compilers exploit this support when dealing with code generation for function calls
- Activation records for each function invocation
- Storage allocation on the “stack” for activation records
- Stack allocation is the easiest methodology for function calls (even recursive function calls)