Run-time Environments

Lecture 8

Status

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
- Next are the back-end phases
  - Optimization
  - Code generation

- We'll do code generation first . . .

Run-time environments

- Before discussing code generation, we need to understand what we are trying to generate

- There are a number of standard techniques for structuring executable code that are widely used

Outline

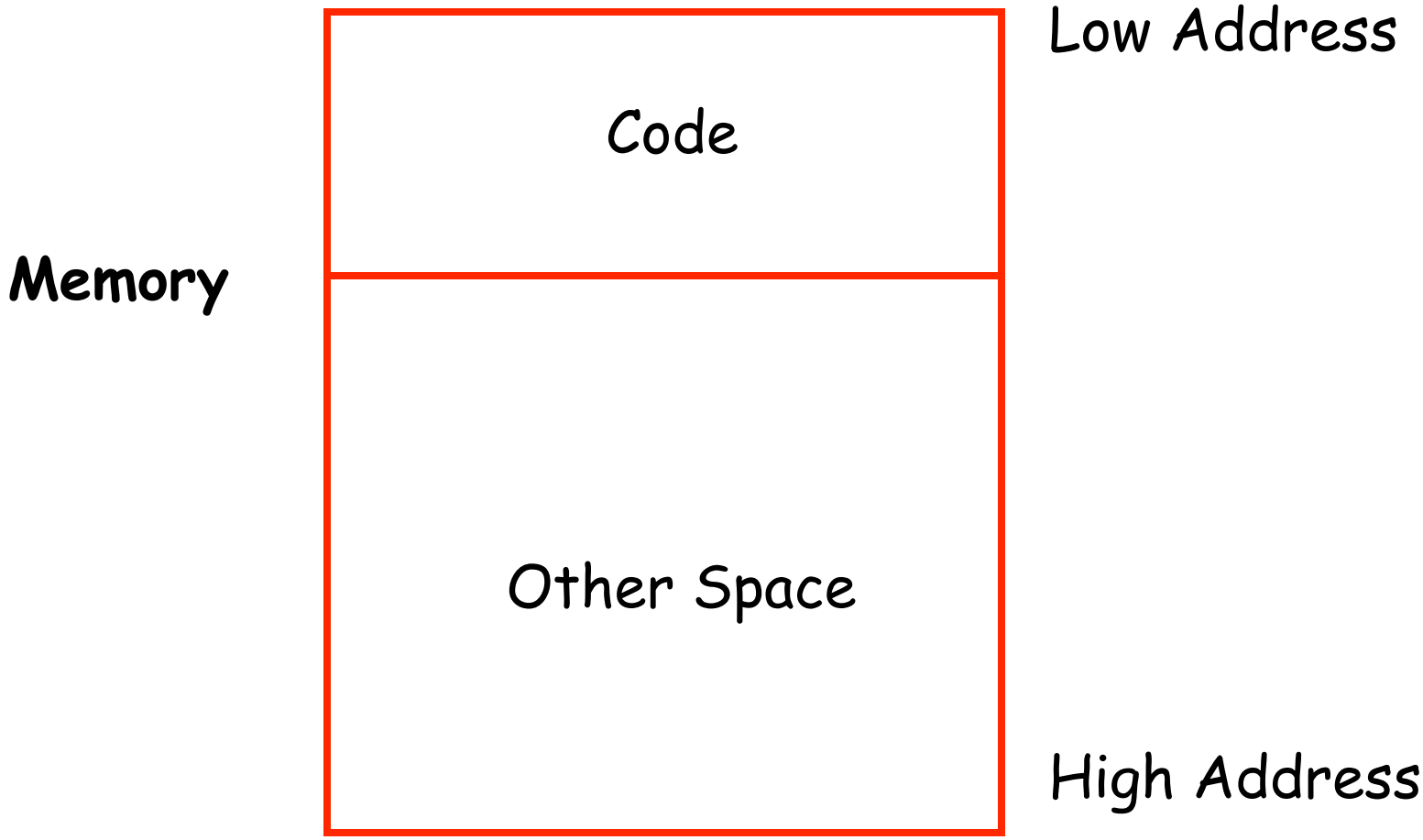
- Management of run-time resources
- Correspondence between static (compile-time) and dynamic (run-time) structures
- Storage organization

Run-time Resources

- Execution of a program is initially under the control of the operating system

- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., "main")

Memory Layout



Notes

- Our pictures of machine organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data
- These pictures are simplifications
  - E.g., not all memory need be contiguous
- In some textbooks lower addresses are at bottom

What is Other Space?

- Holds all data for the program
- Other Space = Data Space

- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area

Code Generation Goals

- Two goals:
  - Correctness
  - Speed

- Most complications in code generation come from trying to be fast as well as correct

Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order

2. When a procedure is called, control eventually returns to the point immediately after the call

Do these assumptions always hold?

Activations

- An invocation of procedure P is an activation of P

- The lifetime of an activation of P is
  - All the steps to execute P
  - Including all the steps in procedures that P calls

Lifetimes of Variables

- The lifetime of a variable x is the portion of execution in which x is defined

- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

The MIPS Architecture

ISA at a Glance

- Reduced Instruction Set Computer (RISC)
- 32 general purpose 32-bit registers
- Load-store architecture: Operands in registers
- Byte Addressable
- 32-bit address space

The MIPS Architecture

32 Register Set (32-bit registers)

Register #	Reg Name	Function
r0	r0	Zero constant
r4-r7	a0-a3	Function arguments
r1	at	Reserved for Operating Systems
r30	fp	Frame pointer
r28	gp	Global memory pointer
r26-r27	k0-k1	Reserved for OS Kernel
r31	ra	Function return address
r16-r23	s0-s7	Callee saved registers
r29	sp	Stack pointer
r8-r15	t0-t7	Temporary variables
r24-r25	t8-t9	Temporary variables
r2-r3	v0-v1	Function return values

The MIPS Architecture

Examples of Native Instruction Set

Instruction Group	Instruction	Function
Arithmetic/ Logic	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
	`addi $s1,$s2,K`	`$s1 = $s2 + K`
Load/Store	`lw $s1,K($s2)`	`$s1 = MEM[$s2+K]`
	`sw $s1,K($s2)`	`MEM[$s2+K] = $s1`
Jumps and Conditional Branches	`beq $s1,$s2,K`	`if ($s1=$s2) goto PC + 4 + K`
	`slt $s1,$s2,$s3`	`if ($s2<$s3) $s1=1 else $s1=0`
	`j K`	`goto K`
Procedures	`jal K`	`$ra = PC + 4; goto K`
	`jr $ra`	`goto $ra`

The SPIM Assembler

Examples of Pseudo-Instruction Set

Instruction Group	Syntax	Translates to:
Arithmetic/ Logic	neg $s1, $s2	sub $s1, $r0, $s2
	not $s1, $s2	nor $17, $18, $0
Load/Store	li $s1, K	ori $s1, $0, K
	la $s1, K	lui $at, 152 ori $s1, $at, -27008
	move $s1, $s2	
Jumps and Conditional Branches	bgt $s1, $s2, K	slt $at, $s1, $s2 bne $at, $0, K
	sge $s1, $s2, $s3	bne $s3, $s2, foo ori $s1, $0, 1 beq $0, $0, bar foo: slt $s1, $s3, $s2 bar:

***Pseudo Instructions**: translated to native instructions by Asser*

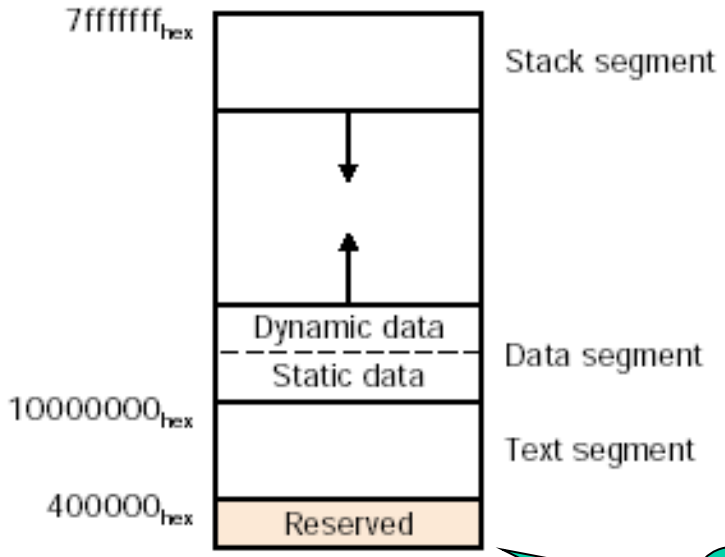
The SPIM Assembler

Examples of Assembler Directives

Group	Directive	Function
Memory Segmentation	.data <addr>	Data Segment starting at
	.text <addr>	Text (program) Segment
	.stack <addr>	Stack Segment
	.ktext <addr>	Kernel Text Segment
	.kdata <addr>	Kernel Data Segment
Data Allocation	x: .word <value>	Allocates 32-bit variable
	x: .byte <value>	Allocates 8-bit variable
	x: .ascii "hello"	Allocates 8-bit cell array
Other	.globl x	x is external symbol

***Assembler Directives**:*
Provide assembler additional info to generate machine code

The MIPS Architecture
Memory Model



32-bit byte addressable address space

Procedure Linkage Approach I

- Problem
  - procedure must determine where to return after servicing the call
- Solution: Architecture Support
  - Add a jump instruction that saves the return address in some place known to callee
    - MIPS: jal instruction saves return address in register $ra
  - Add an instruction that can jump to return address
    - MIPS: jr instruction jumps to the address contained in its argument register

Computing Integer Division (Procedure Version)

Iterative C++ Version

```
int a = 0;
int b = 0;
int res = 0;
main () {
  a = 12;
  b = 5;
  res = 0;
  div();
  printf("Res
}
void div(voi
  while (a >=
    a = a - b
    res ++;
  }
}
```

C++

```
            .data
x:          .word   0
y:          .word   0
res:        .word   0
pf1:        .asciiz "Result = "
pf2:        .asciiz "Remainder = "
            .globl    main
            .text
main:                                       # int main() {
                                            #  uses registers sx unused
            la          $s0, x
            li          $s1, 12
            sw          $s1, 0($s0)
            la          $s0, y
            li          $s2, 5
            sw          $s2, 0($s0)
            la          $s0, res            #   res = 0;
            li          $s3, 0
            sw          $s3, 0($s0)
            jal         d                   #   div();
            lw          $s3, 0($s0)
            la          $a0, pf1            #   printf("Result = %d \n");
            li          $v0, 4              #   //system call to print_str
            syscall
            move        $a0, $s3
            li          $v0, 1              #   //system call to print_int
            syscall
            la          $a0, pf2            #   printf("Remainder = %d \n");
            li          $v0, 4              #   //system call to print_str
            syscall
            move        $a0, $s1
            li          $v0, 1              #   //system call to print_int
            syscall
            jr          $ra                 #   return // TO Operating System
```

Function Call

MIPS Assembly Language

Computing Integer Division (Procedure Version)

Iterative C++ Version

```
int a = 0;
int b = 0;
int res = 0;
main () {
  a = 12;
  b = 5;
  res = 0;
  div();
  printf("Res = %d
}
void div(void) {
  while (a >= b) {
    a = a - b;
    res ++;
  }
}
```

C++

MIPS Assembly Language

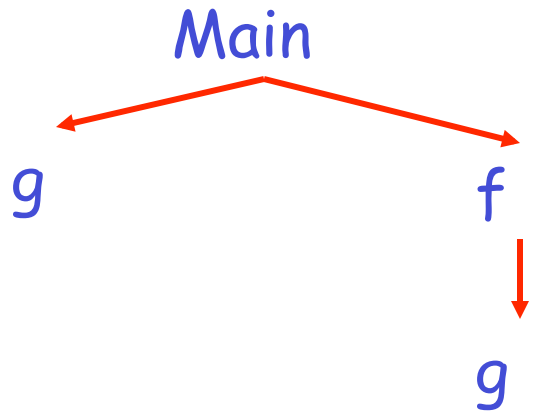
```
# div function
# PROBLEM: Must save args and registers before using them
d:                                              # void d(void) {
                                                #   // Allocate registers for globals
            la      $s0, x                      #   // x in $s1
            lw      $s1, 0($s0)
            la      $s0, y                      #   // y in $s2
            lw      $s2, 0($s0)
            la      $s0, res                    #   // res in $s3
            lw      $s3, 0($s0)
while:      bgt      $s2, $s1, ewhile           #   while (x <= y) {
            sub     $s1, $s1, $s2               #     x = x - y
            addi    $s3, $s3, 1                 #     res ++
            j       while                       #   }
ewhile:                                         #   // Update variables in memory
            la      $s0, x
            sw      $s1, 0($s0)
            la      $s0, y
            sw      $s2, 0($s0)
            la      $s0, res
            sw      $s3, 0($s0)
enddiv:     jr       $ra                        #   return;
                                                # }
```

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```



Example 2

```
Class Main {
   g() : Int { 1 };
   f(x:Int):  Int { if x = 0 then g() else f(x - 1) fi};
   main(): Int {{f(3); }};
}
```

What is the activation tree for this example?

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```

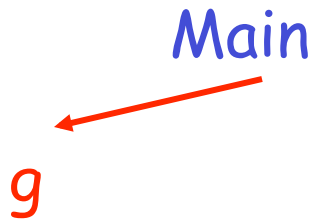
Main

Stack

Main

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```



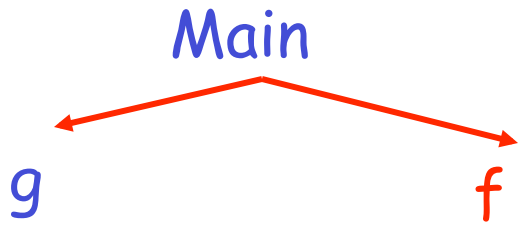
Stack

Main

g

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```



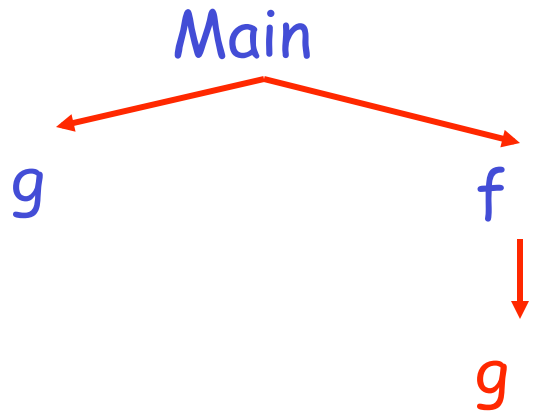
Stack

Main

f

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```



Stack

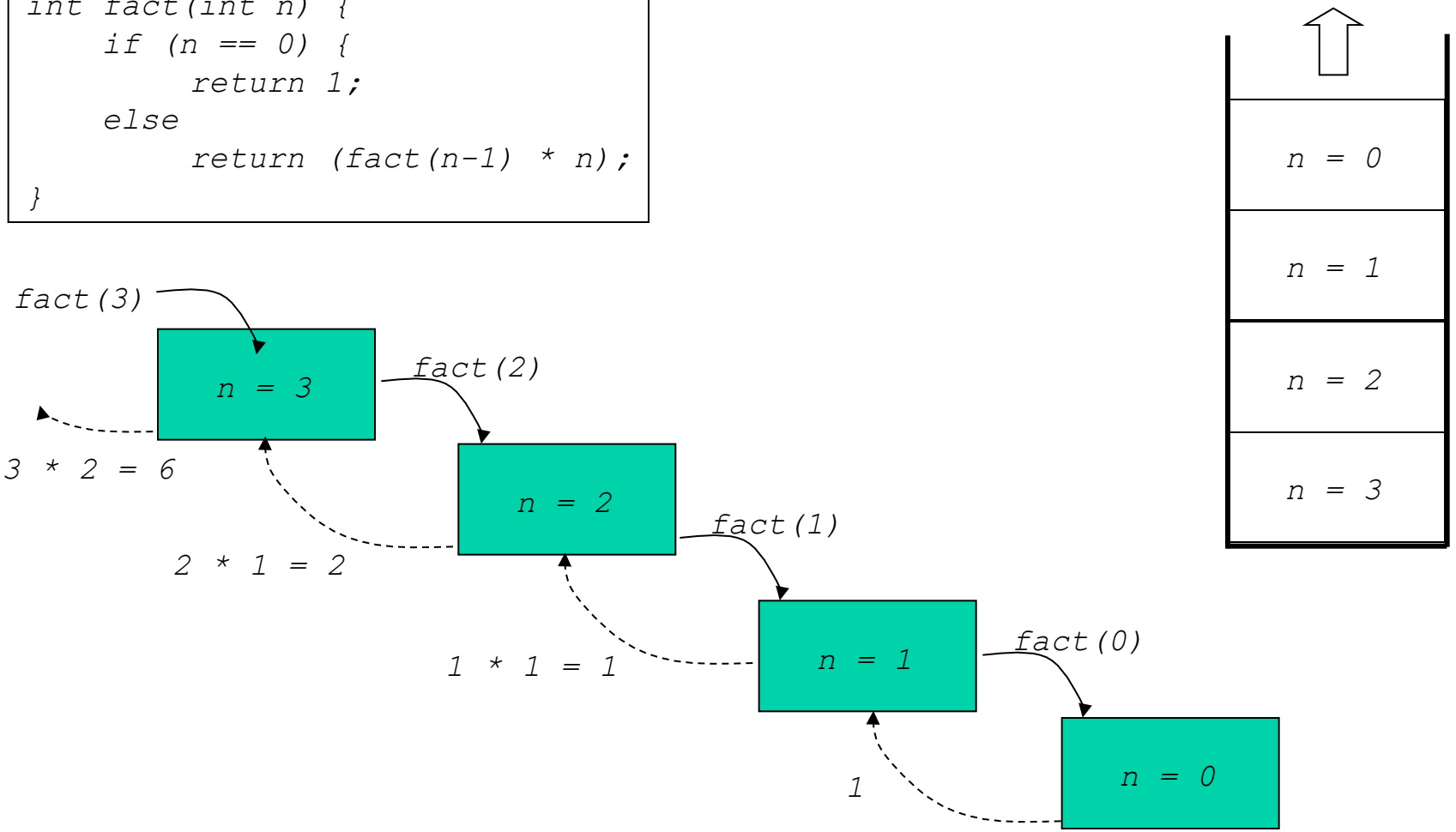
Main

f

g

Recursion Basics

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



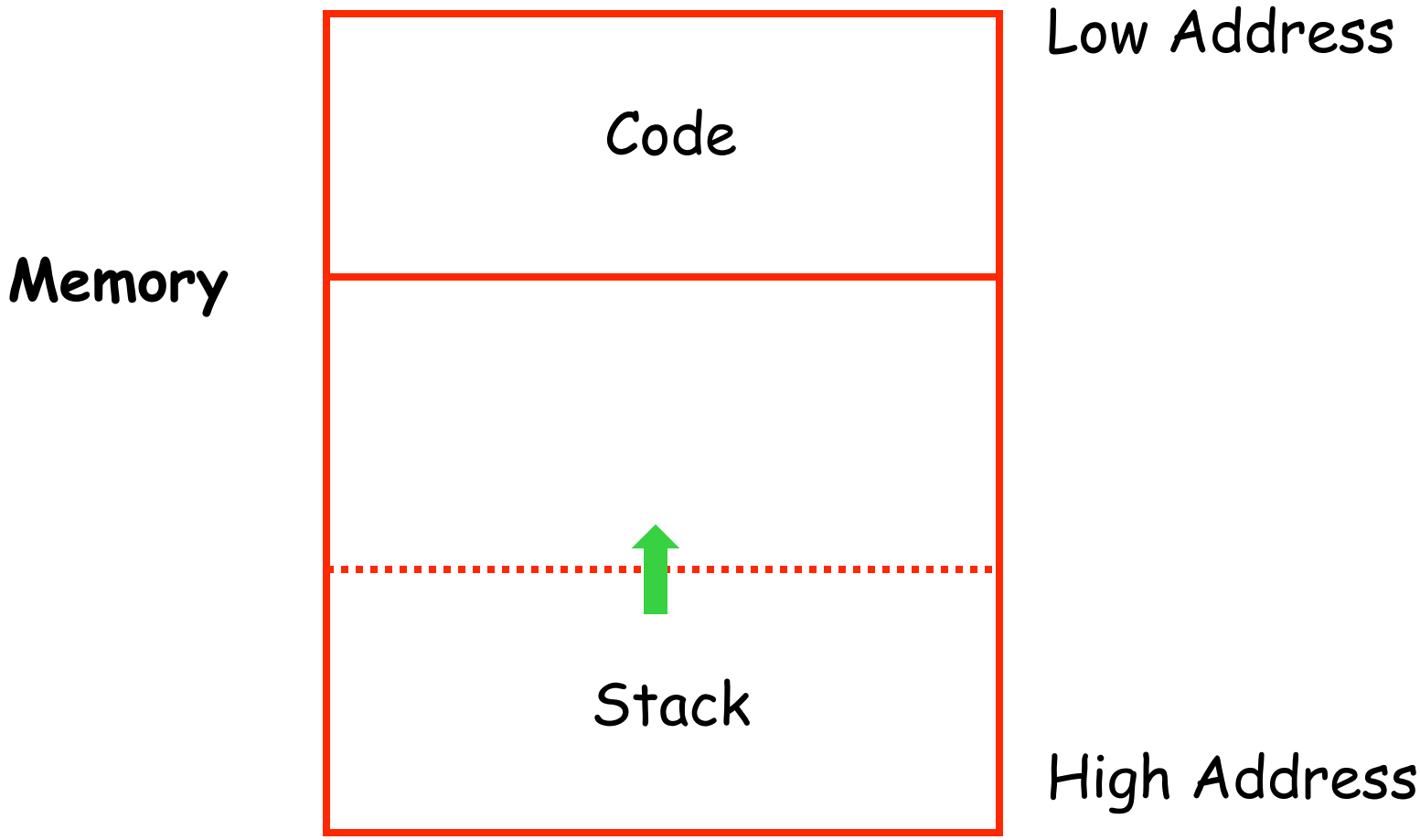
Pending Problems With Linkage Approach I

- Registers shared by all procedures
  - procedures must save/restore registers (use stack)
- Procedures should be able to call other procedures
  - save multiple return addresses (use stack)
- Lack of parameters forces access to globals
  - pass parameters in registers
- Recursion requires multiple copies of local data
  - store multiple procedure activation records (use stack)
- Need a convention for returning function values
  - return values in registers

Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

Revised Memory Layout



Activation Records

- On many machines the stack starts at high-addresses and grows towards lower addresses
- The information needed to manage one procedure activation is called an activation record (AR) or frame
- If procedure F calls G, then G's activation record contains a mix of info about F and G.

What is in G's AR when F calls G?

- F is "suspended" until G completes, at which point F resumes. G's AR contains information needed to resume execution of F.

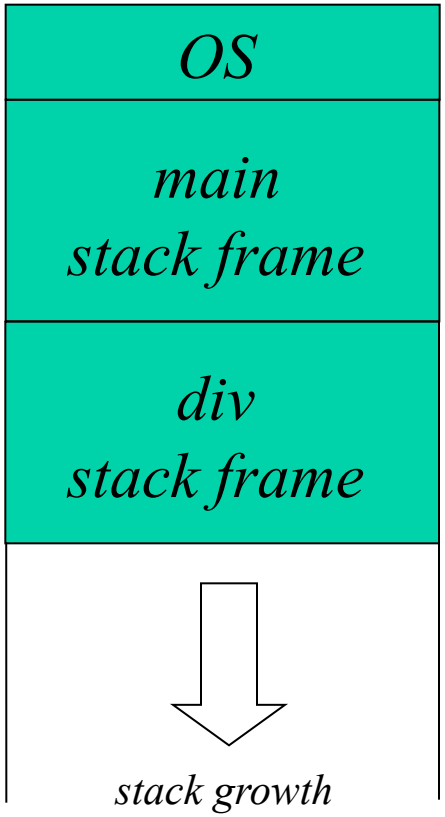
- G's AR may also contain:
  - Actual parameters to G (supplied by F)
  - G's return value (needed by F)
  - Space for G's local variables

The Contents of a Typical AR for G

- Space for G's return value
- Actual parameters
- Pointer to the previous activation record
  - The control link points to AR of caller of G
- Machine status prior to calling G
  - Contents of registers & program counter
  - Local variables
- Other temporary values

Solution: Use Stacks of Procedure Frames

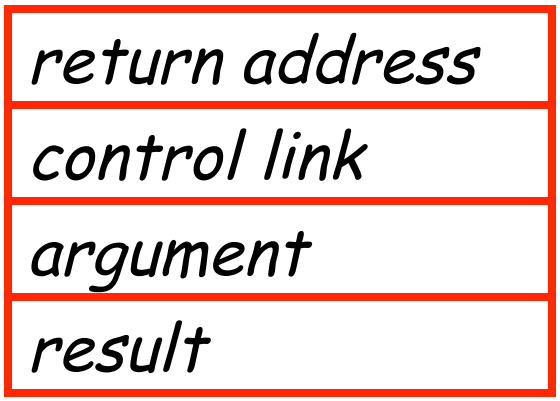
- Stack frame contains:
  - Saved arguments
  - Saved registers
  - Return address
  - Local variables



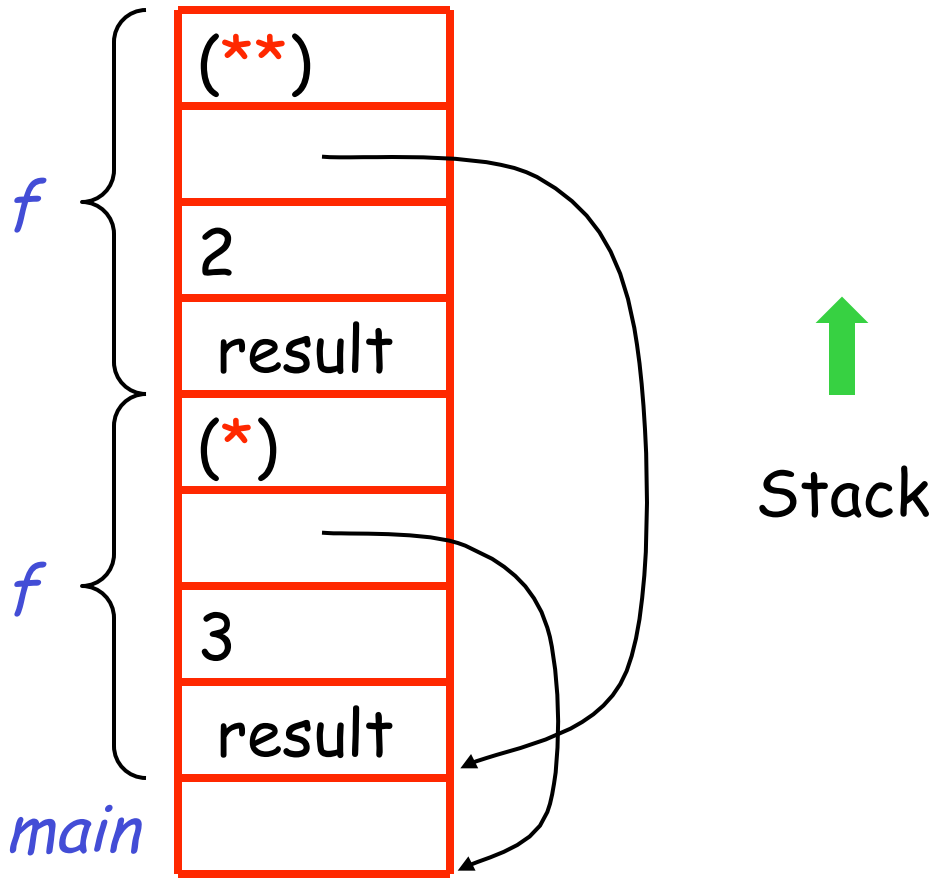
Example 2, Revisited

```
Class Main {
   g() : Int { 1 };
   f(x:Int):Int {if x=0 then g() else f(x - 1)(**)fi};
   main(): Int {{f(3); (*) }};}
```

AR for f:



Stack After Two Calls to f



Notes

- main has no argument or local variables and its result is never used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of f
  - The return address is where execution resumes after a procedure call finishes

- This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.

The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record

Thus, the AR layout and the code generator must be designed together!

Discussion

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame

- There is nothing magic about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation

Discussion (Cont.)

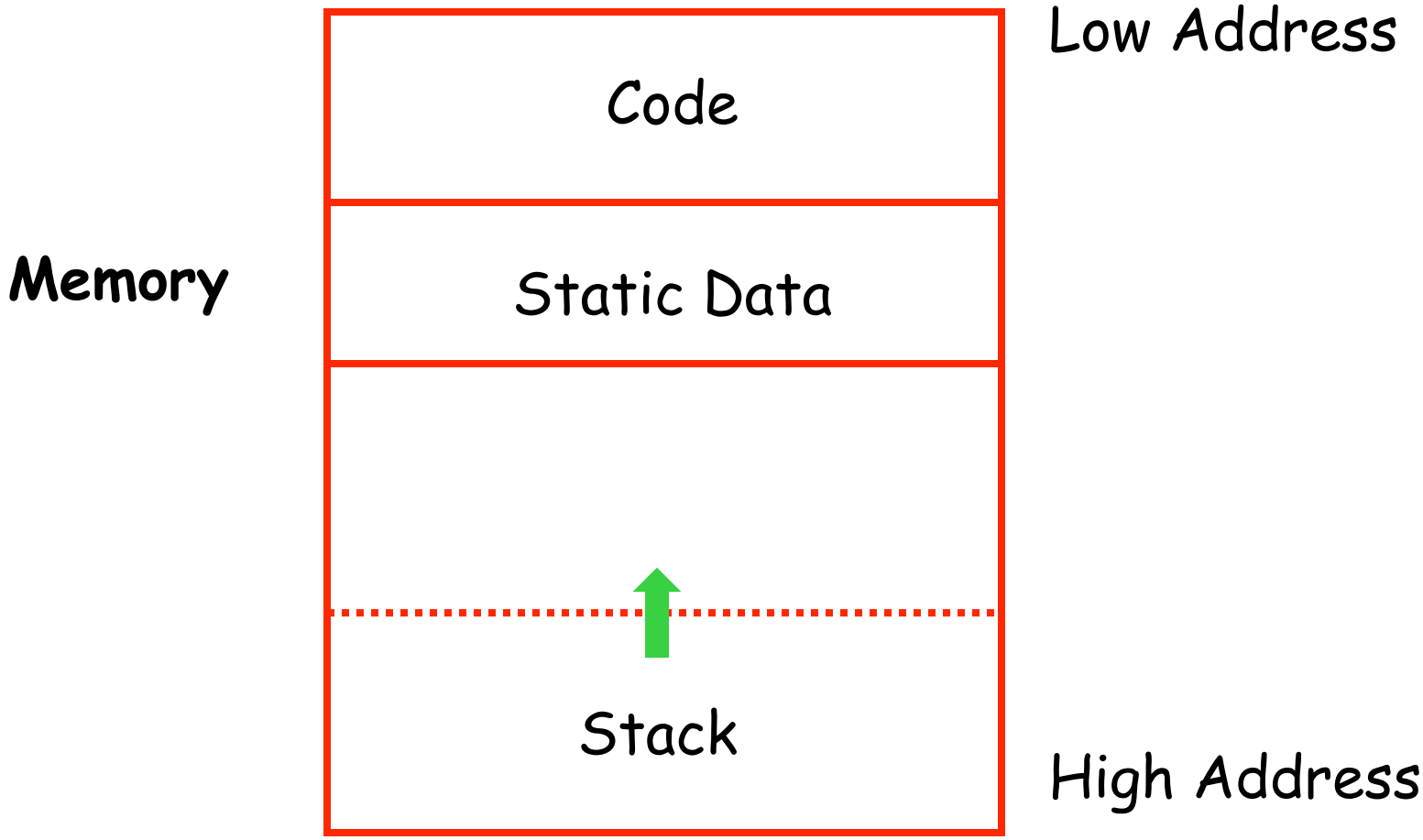
- Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments

Globals

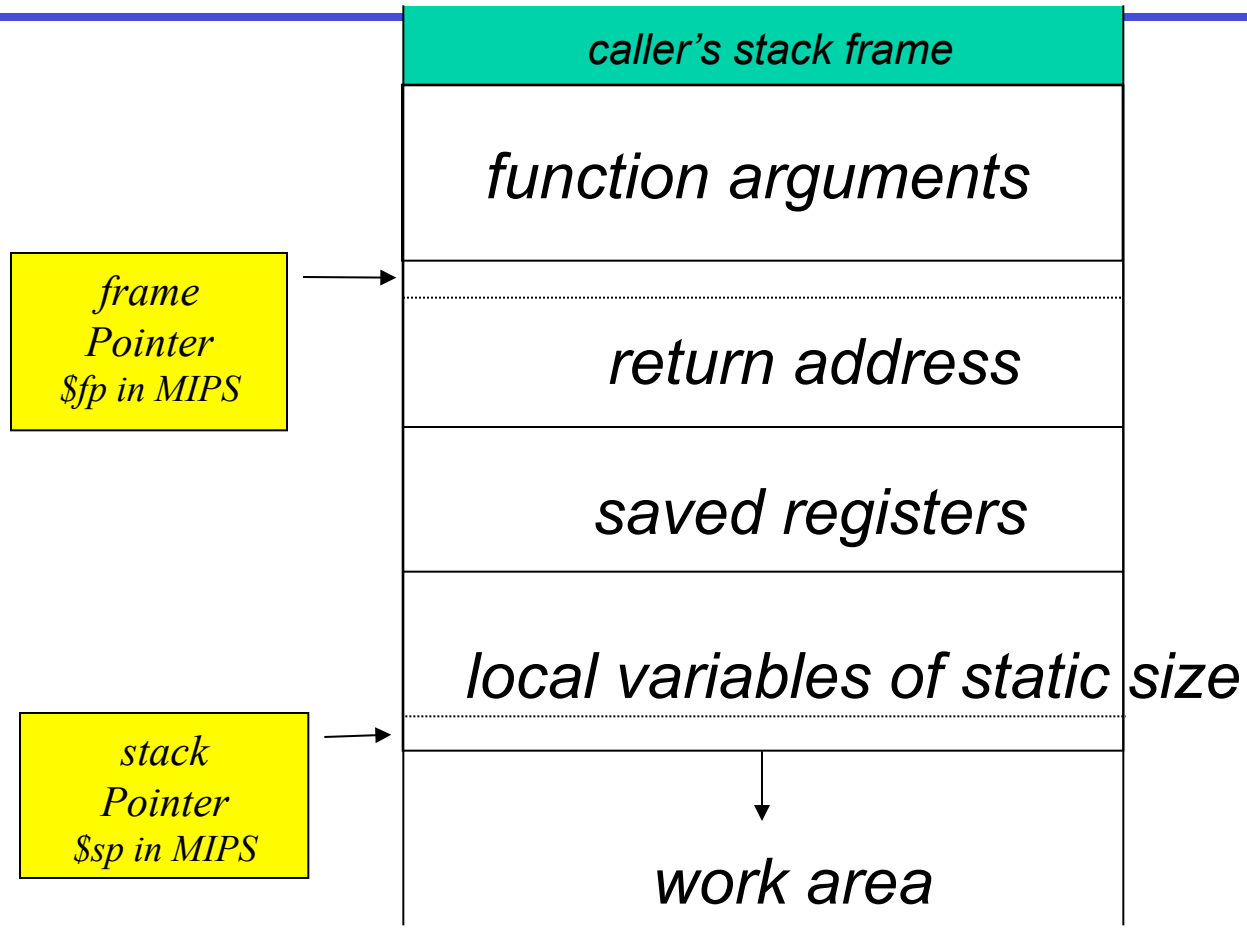
- All references to a global variable point to the same object
  - Can't store a global in an activation record

- Globals are assigned a fixed address once
  - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values

Memory Layout with Static Data



Anatomy of a Stack Frame



Contract: Every function must leave the stack the way it found it

Example: Function Linkage using Stack Frames

```
int x = 0;
int y = 0;
int res = 0;
main () {
  x = 12;
  y = 5;
  res = div(x,y);
  printf("Res = %d",res);
}
int div(int a,int b) {
  int res = 0;
  if (a >= b) {
    res = div(a-b,b) + 1;
  }
  else {
    res = 0;
  }
  return res;
}
```

- *Add return values*
- *Add parameters*
- *Add recursion*
- *Add local variables*

Example: Function Linkage using Stack Frames

```
div:        sub         $sp, $sp, 28          # Alloc space for 28 byte stack frame
            sw          $a0, 24($sp)          # Save argument registers
            sw          $a1, 20($sp)          # a in $a0
            sw          $ra, 16($sp)          # Save other registers as needed
            sw          $s1, 12($sp)          # Save callee saved registers ($sx)
            sw          $s2, 8($sp)
            sw          $s3, 4($sp)                       # No need to save $s4, since not used
            li          $s3, 0
            sw          $s3, 0($sp)                       # int res = 0;
                                              # Allocate registers for locals
            lw          $s1, 24($sp)          #   a in $s1
            lw          $s2, 20($sp)          #   b in $s2
            lw          $s3, 0($sp)                       #   res in $s3

if:         bgt         $s2, $s1, else        # if (a >= b) {
            sub         $a0, $s1, $s2         #
            move        $a1, $s2
            jal         div                   #
            addi        $s3, $v0, 1                       #   res = div(a-b, b) + 1;
            j           endif                 # }
else:       li          $s3, 0                # else { res = 0; }
endif:
            sw          $s1, 32($sp)          #   deallocate a from $s1
            sw          $s2, 28($sp)          #   deallocate b from $s2
            sw          $s3, 0($sp)                       #   deallocate res from $s3
            move        $v0, $s3              # return res

            lw          $a0, 24($sp)          # Restore saved registers
            lw          $a1, 20($sp)          # a in $a0
            lw          $ra, 16($sp)          # Save other registers as needed
            lw          $s1, 12($sp)          # Save callee saved registers ($sx)
            lw          $s2, 8($sp)
            lw          $s3, 4($sp)                       # No need to save $s4, since not used
            addu        $sp, $sp, 28          # pop stack frame
enddiv:     jr          $ra                   # return;
#
```

MIPS: Procedure Linkage Summary

- First 4 arguments passed in $a0-$a3
- Other arguments passed on the stack
- Return address passed in $ra
- Return value(s) returned in $v0-$v1
- Sx registers saved by callee
- Tx registers saved by caller

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR

  method foo() { new Bar }

  The Bar value must survive deallocation of foo's AR

- Languages with dynamically allocated data use a heap to store dynamic data

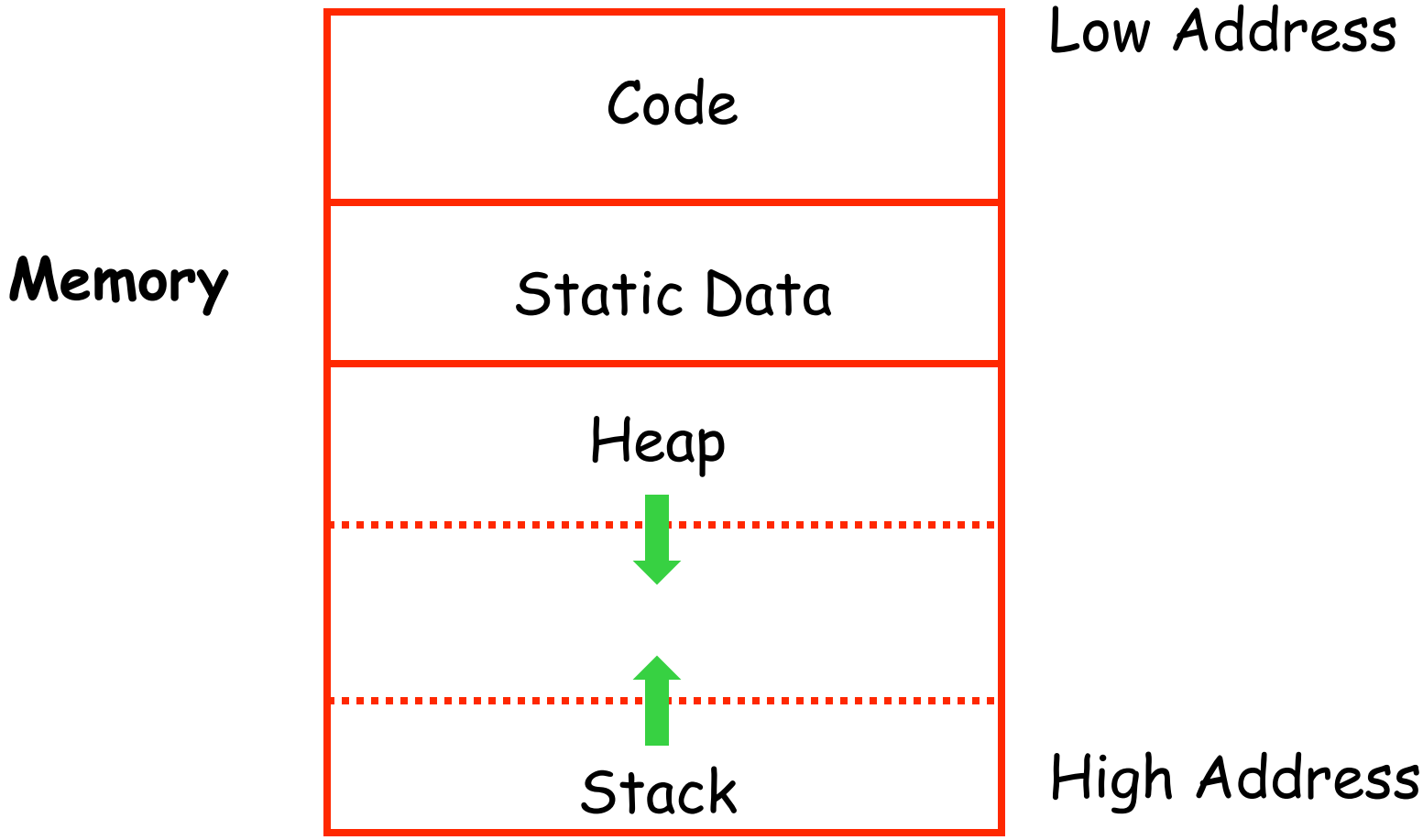
Notes

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by *malloc* and *free*

Notes (Cont.)

- Both the heap and the stack grow
- Must take care that they don't grow into each other
- Solution: start heap and stack at opposite ends of memory and let the grow towards each other

Memory Layout with Heap



Data Layout

- Low-level details of machine architecture are important in laying out data for correct code and maximum performance

- Chief among these concerns is alignment

Alignment

- Most modern machines are (still) 32 bit
  - 8 bits in a byte
  - 4 bytes in a word
  - Machines are either byte or word addressable
- Data is *word aligned* if it begins at a word boundary
- Most machines have some alignment restrictions
  - Or performance penalties for poor alignment

Alignment (Cont.)

- Example: A string

  "Hello"

  Takes 5 characters (without a terminating \0)

- To word align next datum, add 3 "padding" characters to the string

- The padding is not part of the string, it's just unused memory

END

Computing Integer Division
Iterative C++ Version

MIPS/SPIM Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
    while (a >= b) {
      a = a - b;
      result ++;
    }
  }
}
```

C++

MIPS Assembly Language

```
                .data                       # Use HLL program as a comment
x:              .word       12              # int x = 12;
y:              .word       4               # int y = 4;
res:            .word       0               # int res = 0;
                .globl      main
                .text
main:           la          $s0, x          # Allocate registers for globals
                lw          $s1, 0($s0)     #   x in $s1
                lw          $s2, 4($s0)     #   y in $s2
                lw          $s3, 8($s0)     #   res in $s3
while:          bgt         $s2, $s1, endwhile  # while (x >= y) {
                sub         $s1, $s1, $s2   #   x = x - y;
                addi        $s3, $s3, 1     #   res ++;
                j           while           # }
endwhile:
                la          $s0, x          # Update variables in memory
                sw          $s1, 0($s0)
                sw          $s2, 4($s0)
                sw          $s3, 8($s0)
```

Computing Integer Division
Iterative C++ Version

MIPS/SPIM Version
Input/Output in SPIM

```
int a = 12;
int b = 4;
int result = 0;
main () {
    while (a >= b
      a = a - b;
      result ++;
    }
  }
  printf("Result =
\n");
}
```

C++

MIPS Assembly Language

```
                .data                                   # Use HLL program as a comment
x:              .word       12                          # int x = 12;
y:              .word       4                           # int y = 4;
res:            .word       0                           # int res = 0;
pf1:            .asciiz     "Result = "
                .globl      main
                .text
main:           la          $s0, x                      # Allocate registers for globals
                lw          $s1, 0($s0)                 #   x in $s1
                lw          $s2, 4($s0)                 #   y in $s2
                lw          $s3, 8($s0)                 #   res in $s3
while:          bgt         $s2, $s1, endwhile          # while (x >= y) {
                sub         $s1, $s1, $s2    #   x = x - y;
                addi        $s3, $s3, 1                 #   res ++;
                j           while                       # }
endwhile:
                la          $a0, pf1                    # printf("Result = %d \n");
                li          $v0, 4                      # //system call to print_str
                syscall
                move        $a0, $s3
                li          $v0, 1                      # //system call to print_int
                syscall
                la          $s0, x                      # Update variables in memory
                sw          $s1, 0($s0)
                sw          $s2, 4($s0)
                sw          $s3, 8($s0)
```

SPIM Assembler Abstractions

- Symbolic Labels
  - Instruction addresses and memory locations
- Assembler Directives
  - Memory allocation
  - Memory segments
- Pseudo-Instructions
  - Extend native instruction set without complicating arquitecture
- Macros

The MIPS Architecture
Main Instruction Formats

–*R Format*

opcode *6 bits*	*rs* *5 bits*	*rt* *5 bits*	*rd* *5 bits*	*shamt* *5 bits*	*funct* *6 bits*

–*I Format*

opcode *6 bits*	*rs* *5 bits*	*rt* *5 bits*	*Address/Immediate* *16 bits*

–*J Format*

opcode *6 bits*	*rs* *5 bits*	*rt* *5 bits*	*Address/Immediate* *16 bits*

MIPS Data Paths

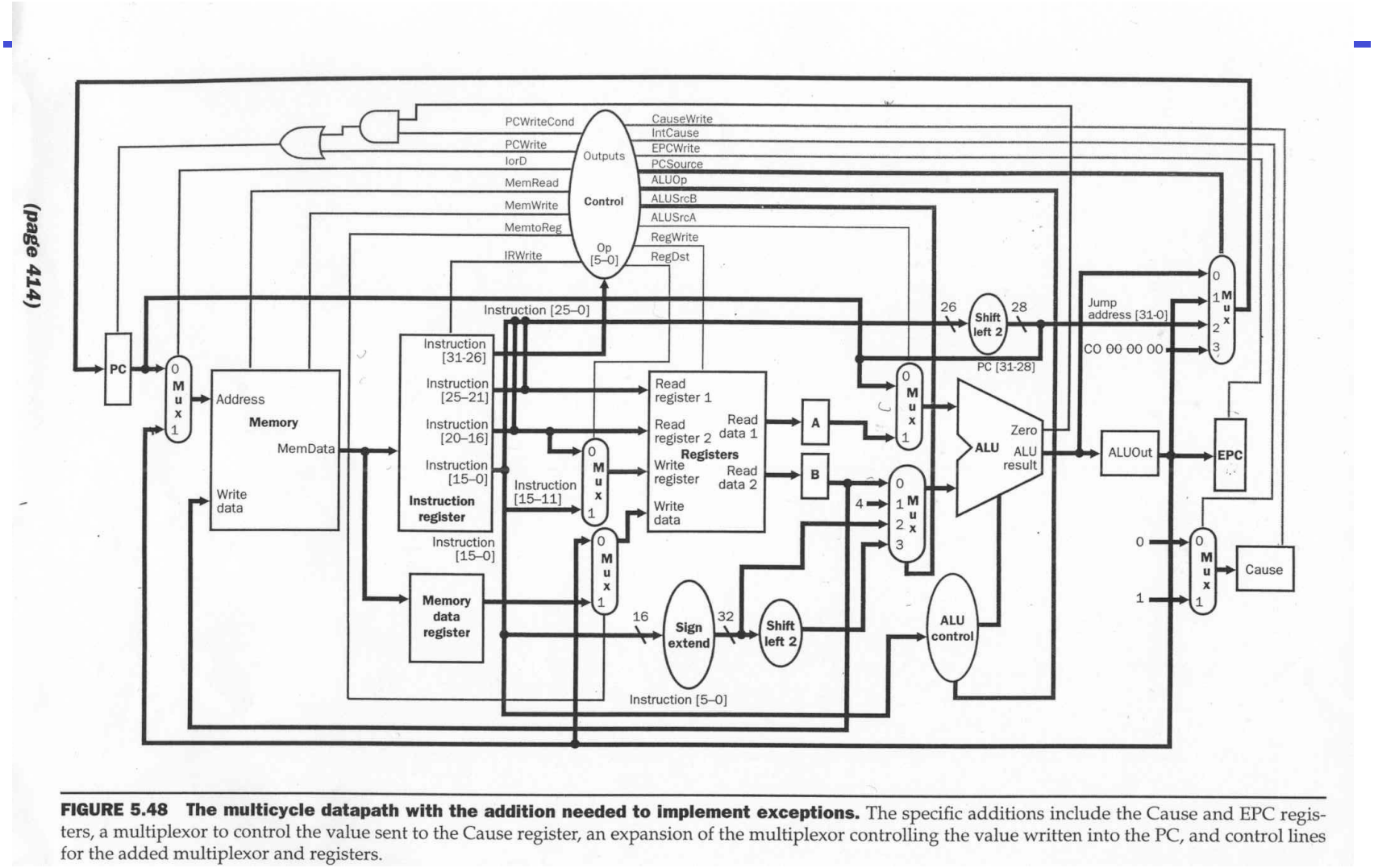
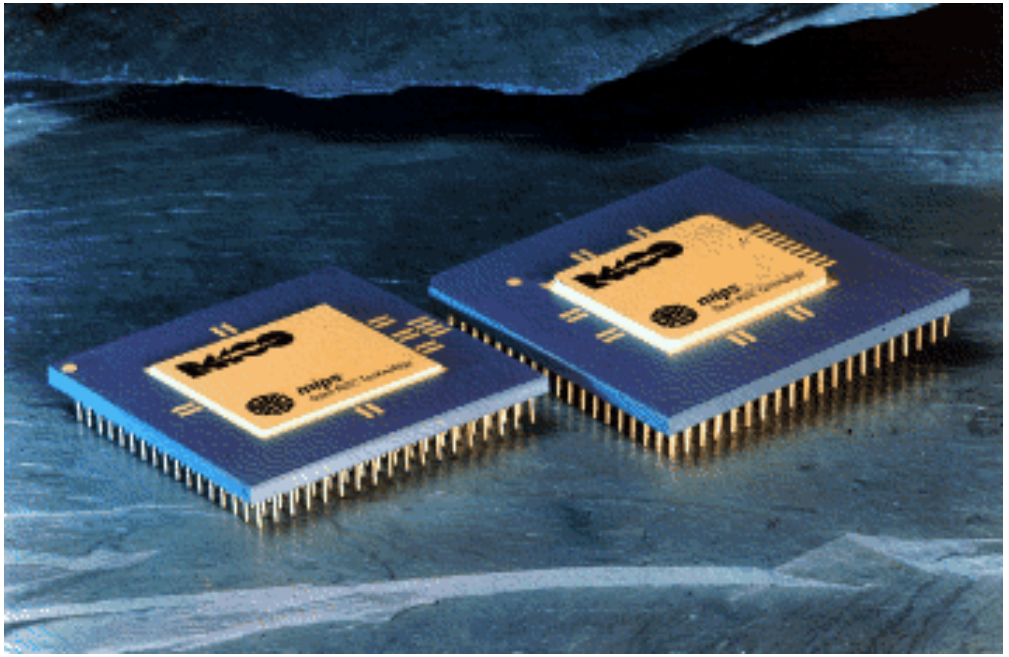


FIGURE 5.48 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers.

Mips Packaging



Handy MIPS ISA References

- *Appendix A: Patterson & Hennessy*
- *SPIM ISA Summary on class website*
- *Patterson & Hennessy Back Cover*