ELEC 5200-001/6200-001 Computer Architecture and Design

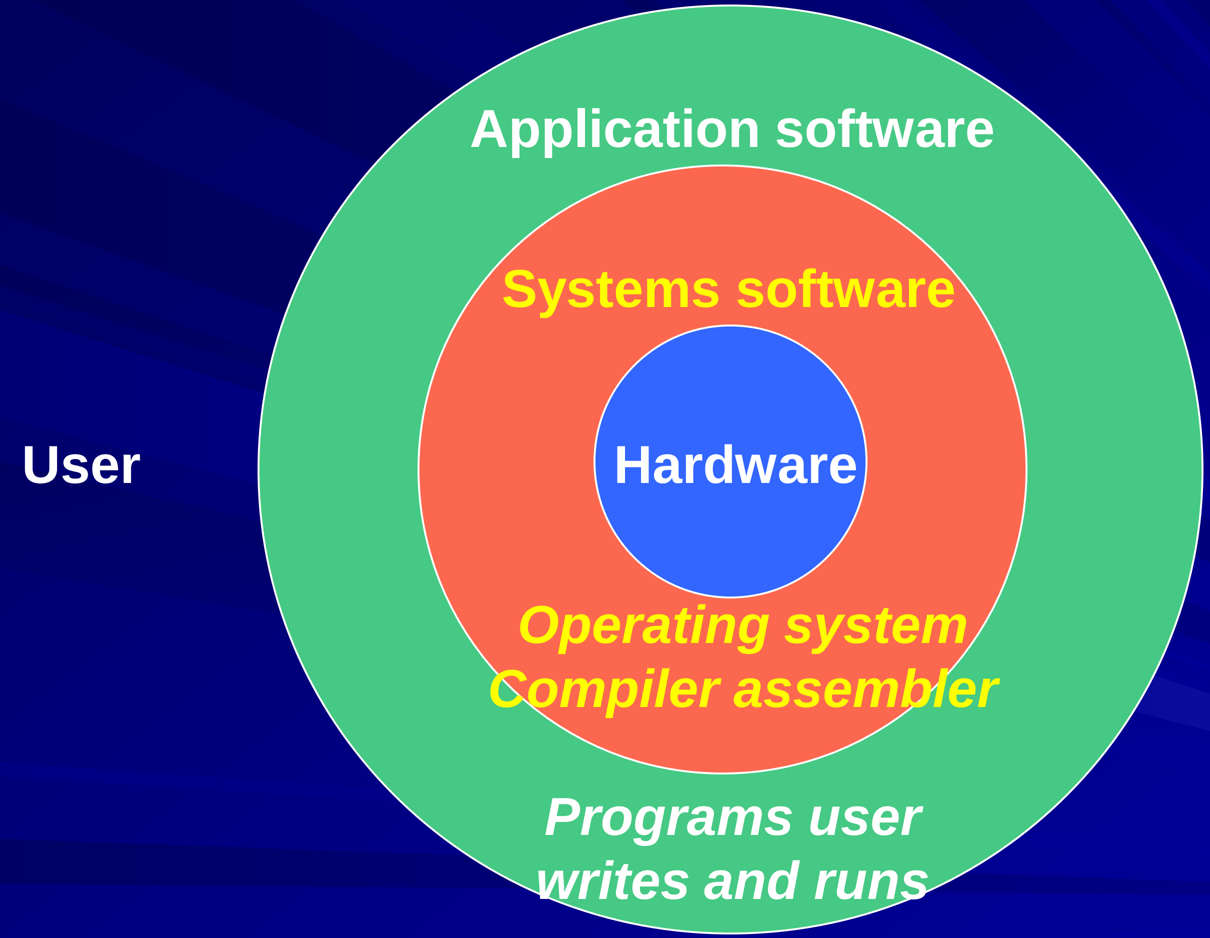
Spring 2019

Compiling and Executing Programs (Chapter 2, Sec 12)

Christopher B. Harris
Assistant Professor
Department of Electrical and Computer Engineering
Auburn University, Auburn, AL 36849

(Adapted from slides by Vishwani D. Agrawal)

Software in a Computer



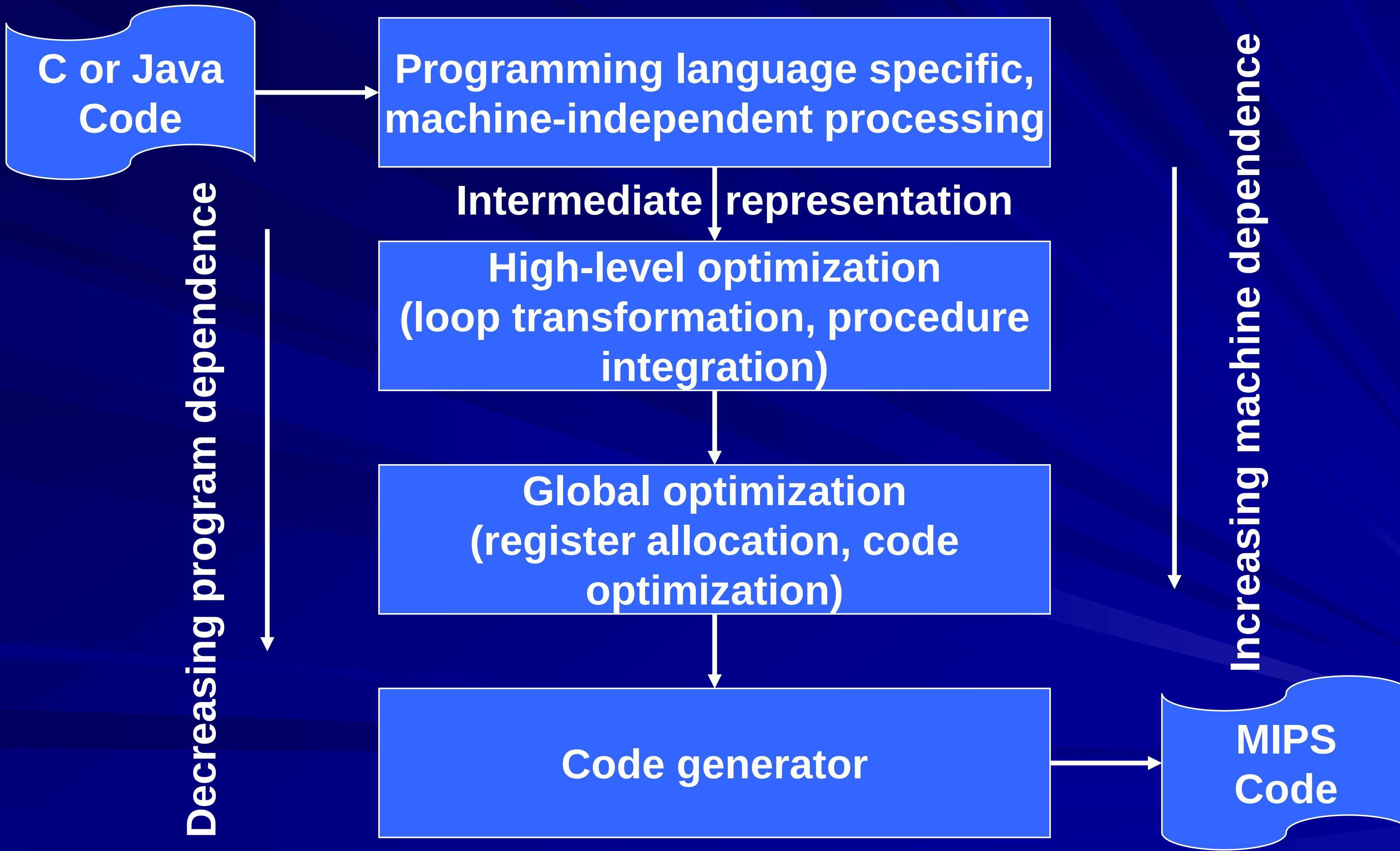
Memory and Registers

Policy of Register Usage (Conventions)

Name	Register number	Usage
`$zero`	0	the constant value 0
`$at`	1	reserved for use by assembler
`$v0-$v1`	2-3	values for results and expression evaluat
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Register 1 ($at) reserved for assembler, 26-27 for operating system

Compiler is a Multi-Pass Program



Compiler Optimizes Code

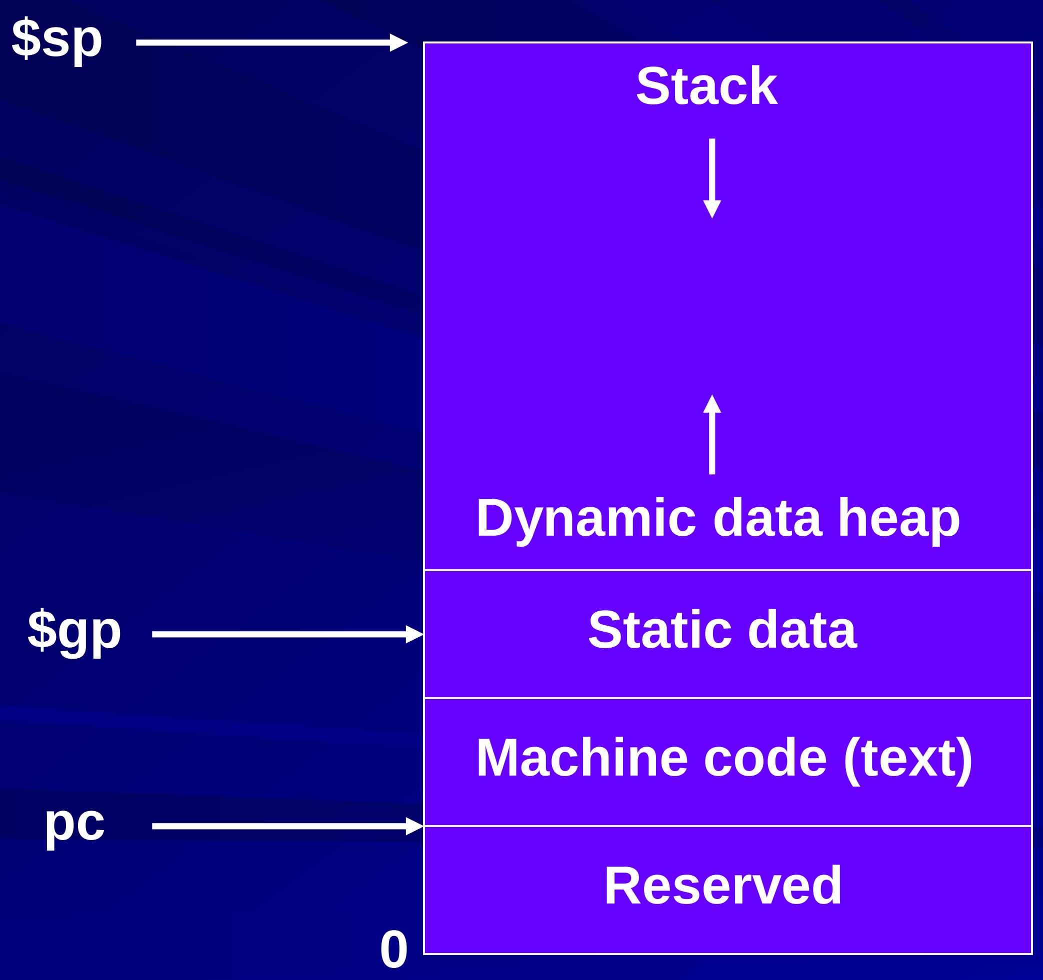
- Minimize number of machine instructions
  - **Example: x[i] = x[i] + 4, memory address for x[i] is generated only once, saved in a register, and used by lw and sw – *Common subexpression elimination*.**
  - **Local optimization within a block of code.**
  - **Global optimization across blocks.**
  - **Global register allocation.**
- Strength reduction. Example: replace integer multiply by 2^k with shift left.
- Unnecessary instructions. A value not used in the later part of program may not be stored in memory (eliminate sw).

MIPS Compiler Example

- C code: f = (g+h) – (i+j)
- Compiler assigns variables f, g, h, i and j to registers $s0, $s1, $s2, $s3 and $s4
- Uses two temporary registers, $t0 and $t1, to produce the following MIPS assembly code:

```
add $t0, $s1, $s2       # reg $t0 contains g+h
add $t1, $s3, $s4       # reg $t1 contains i+j
sub $s0, $t0, $t1       # reg $s0     = $t0 – $t1
                                      = (g+h) – (i+j)
```

Registers $sp and $gp

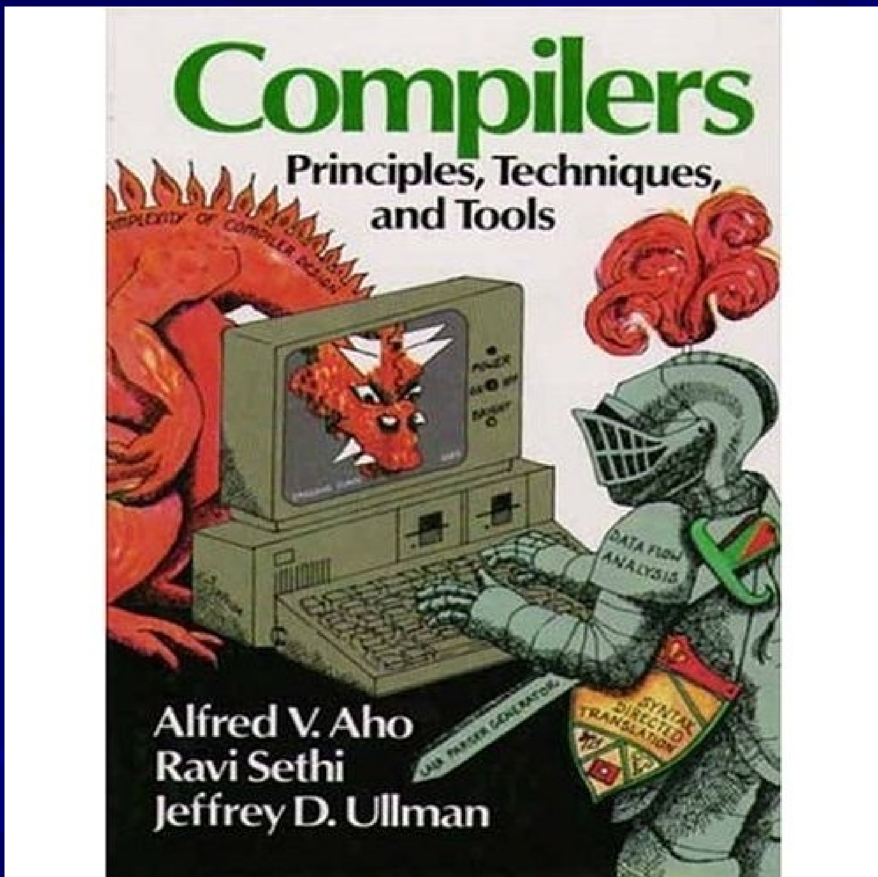


Temp. registers saved before procedure call

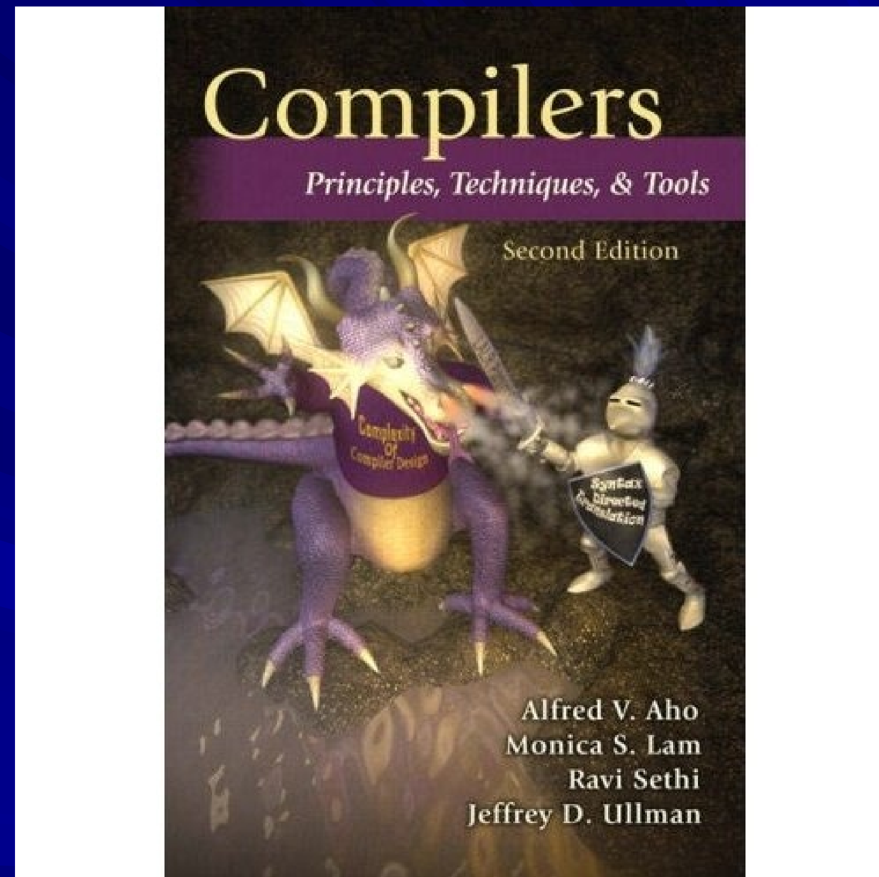
$s0-$s7 saved in stack by procedure

Global variables in program

Dragon Books on Compilers



Addison-Wesley 1986



Pearson Education 2006

Beyond Textbooks: Papers Available on Course Website

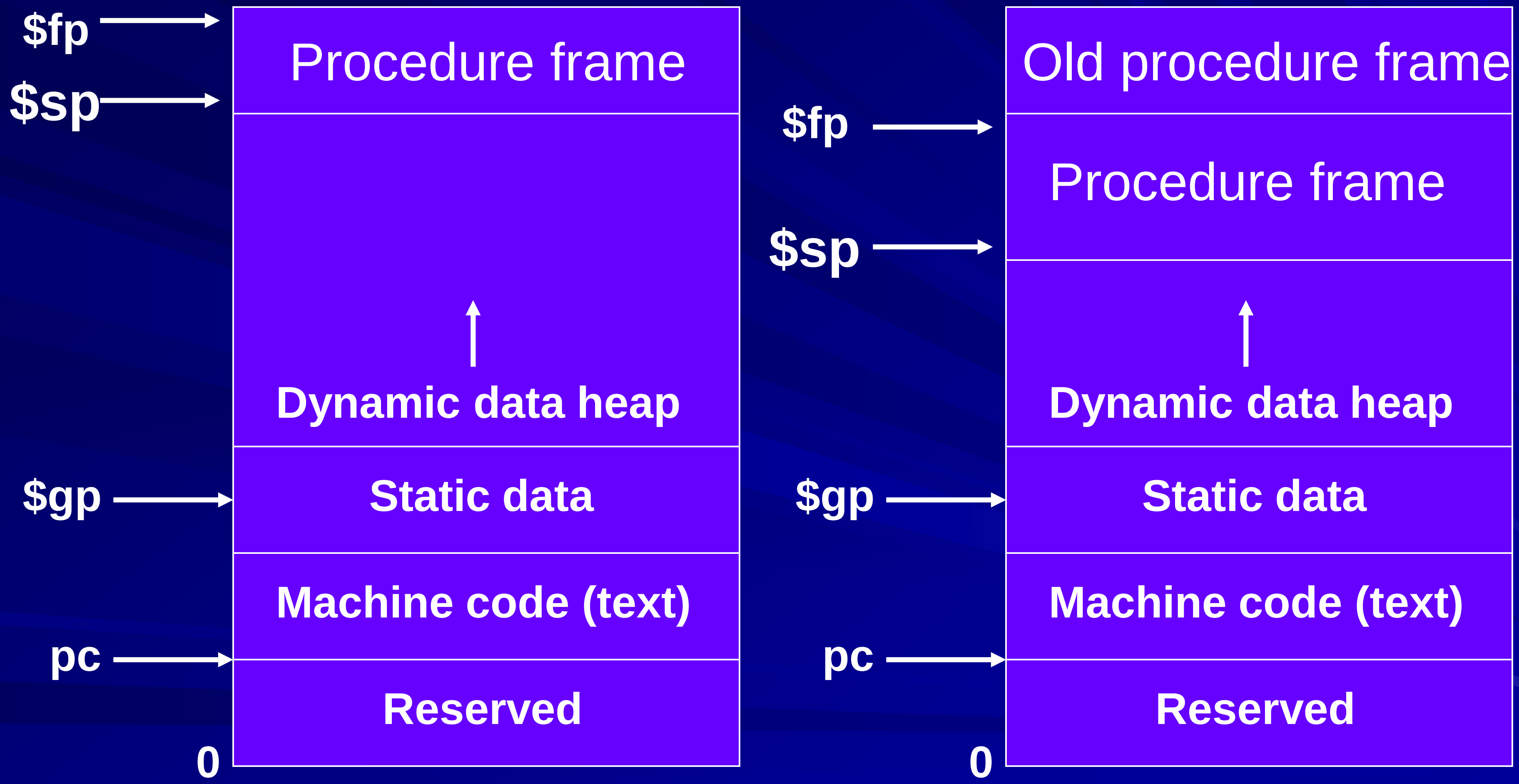
- M. Hall, D. Padua and K. Pingali, “Compiler Research: The Next 50 Years,” *CACM*, vol. 52, no. 2, pp. 60-67, Feb. 2009.
- J. Larus and G. Hunt, “The Singularity System,” *CACM*, vol. 53, no. 8, pp. 72-79, Aug. 2010.
- S. V. Adve and H.-J. Boehm, “Memory Models: A Case for Rethinking Parallel Languages and Hardware,” *CACM*, vol. 53, no. 8, pp. 90-101, Aug. 2010.
- G. L. Steele Jr., “An Interview with Frances E. Allen,” *CACM*, vol. 54, no. 1, pp. 39-45, Jan. 2011.

MIPS Compiler Example

- C code: f = (g+h) – (i+j)
- Compiler assigns variables f, g, h, i and j to registers $s0, $s1, $s2, $s3 and $s4
- Uses two temporary registers, $t0 and $t1, to produce the following MIPS assembly code:

```
add $t0, $s1, $s2      # reg $t0 contains g+h
add $t1, $s3, $s4      # reg $t1 contains i+j
sub $s0, $t0, $t1      # reg $s0    = $t0 – $t1
                                    = (g+h) – (i+j)
```

Register $fp (Frame Pointer)



Compiling a Procedure Call

- Consider a program that uses register $t0 and calls a procedure *proc*
- Assembly code of program with procedure call:

```
                .
                .
addi    $sp, $sp, - 4     # adjust stack pointer
sw      $t0, 0($sp)       # save $t0
jal     proc              # call proc
lw      $t0, 0($sp)       # restore $t0
addi    $sp, $sp, 4       # pop 1 word off stack
                .
                .
```

Compiling a Procedure

- Consider a procedure *proc* that uses register $s0
- Assembly code:

```
proc :
addi        $sp, $sp, - 4        # adjust stack pointer
sw          $s0, 0($sp)          # save $s0
                    .
            Assembly code of proc
                    .
lw          $s0, 0($sp)          # restore $s0
addi        $sp, $sp, 4          # pop 1 word off stack
jr          $ra                  # return
```

Software

Compiler → **Assembler** →

Application software, a program in C:

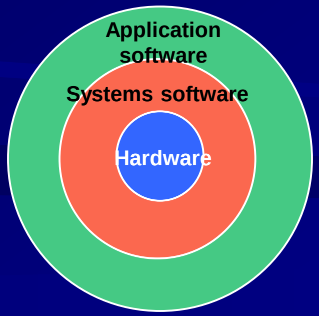
```
swap (int  v[ ], int  k)
{int  temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

MIPS compiler output, assembly language program:

```
swap;
        muli    $2,     $5, 4
        add     $2,     $4, $2
        lw      $15,    0 ($2)
        lw      $16,    4 ($2)
        sw      $16,    0 ($2)
        sw      $15,    4 ($2)
        jr      $31
```

MIPS binary machine code:

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```



See pages 123-124

MIPS Assembler Example

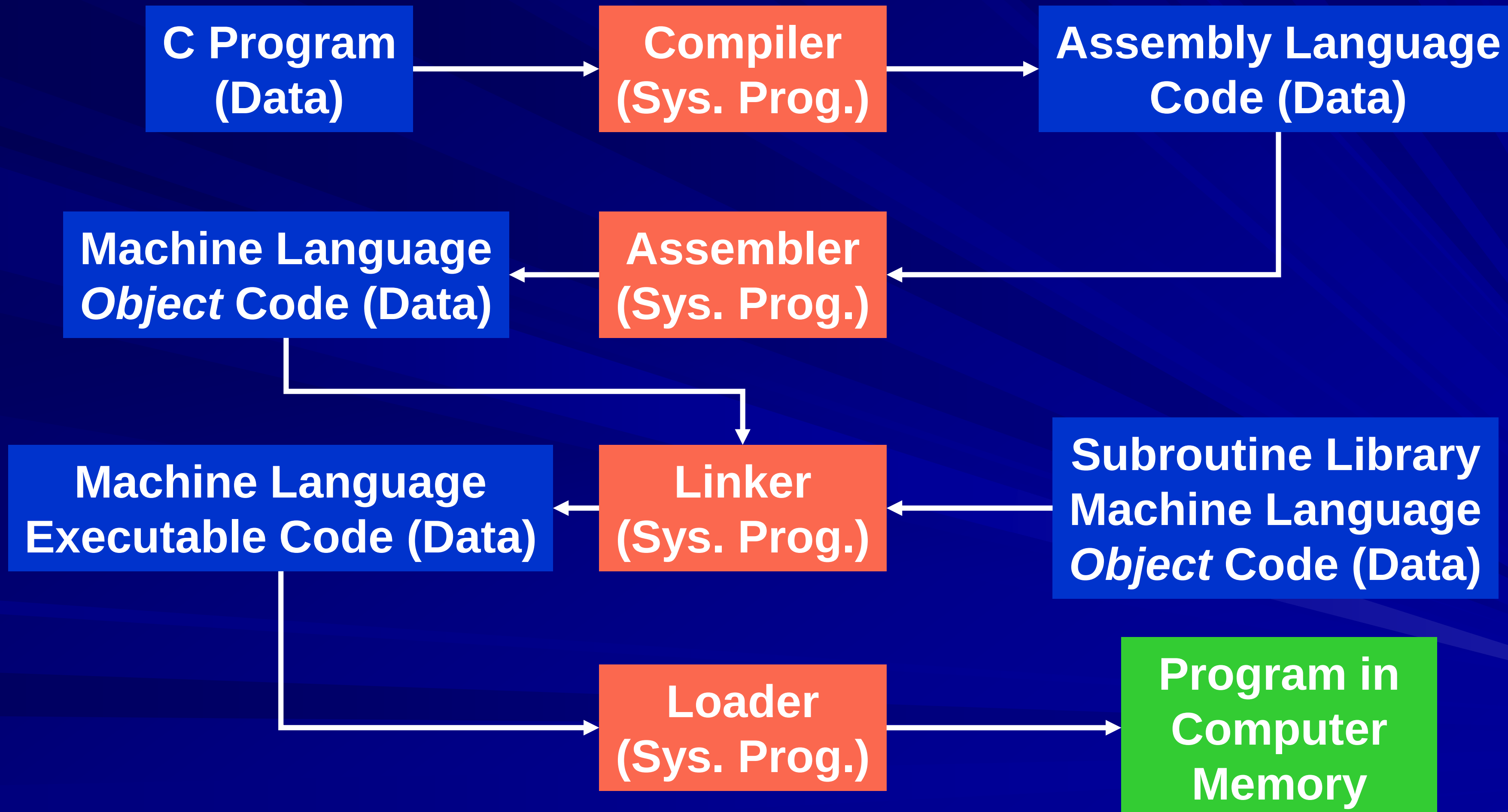
- Use register convention:
  - – Registers 16-23 are \$s0 through \$s7
  - – Registers 8-15 are \$t0 through \$t7
- Machine code: three 32-bit words from assembly code of slide 7

add 000000 10001 10010 01000 00000 100000

add 000000 10011 10100 01001 00000 100000

sub 000000 01000 01001 10000 00000 100010

Translating a Program to Executable Code



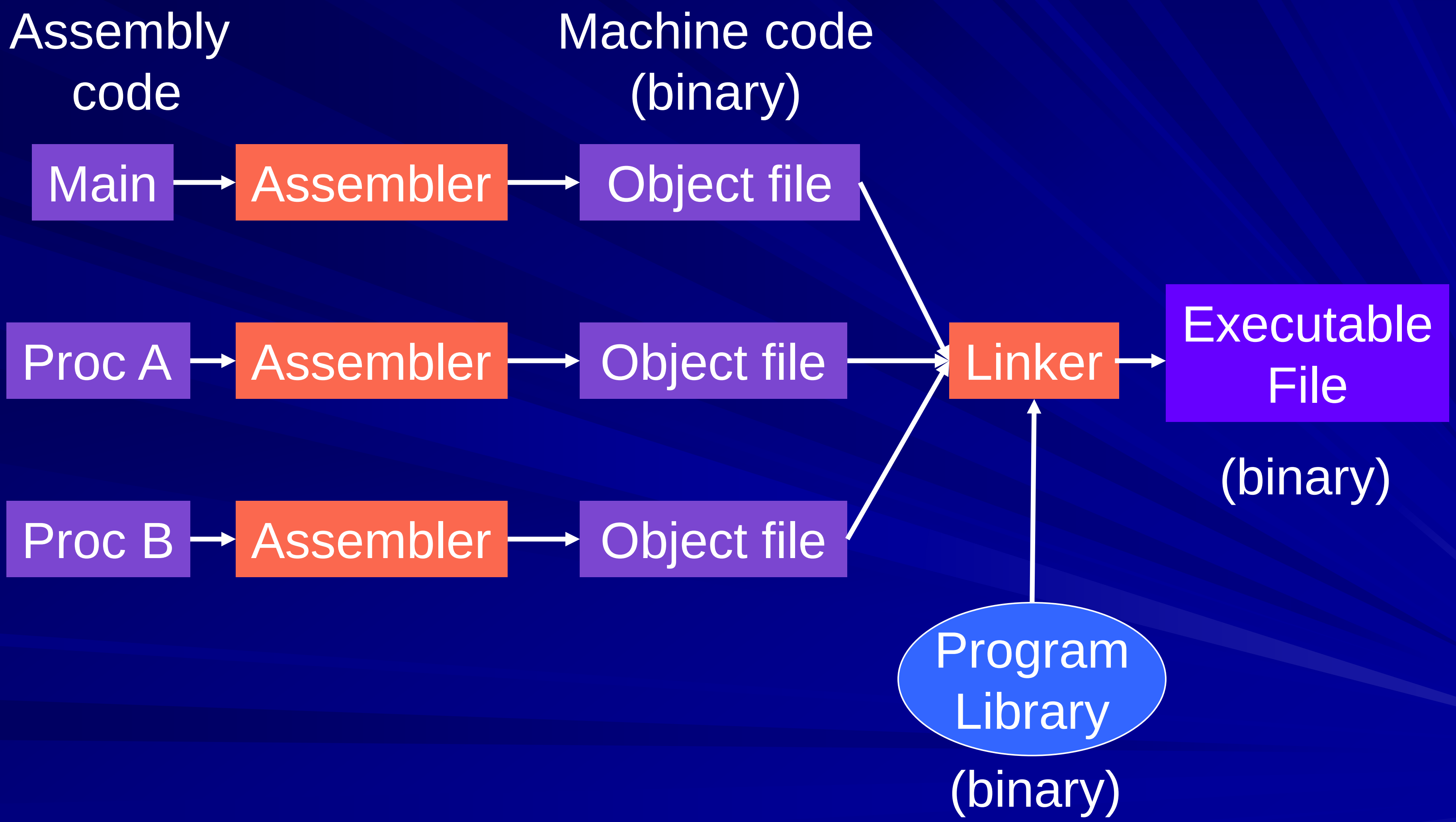
Compiler

- System program
- Inputs:
  - Programming language code, e.g., a C program
  - Instruction set (including pseudoinstructions)
  - Memory and register organization
- Output: Assembly language code
- Compiler's function
  - Specific to ISA and machine organization
  - Assigns variables to registers
  - Translates C into assembly language
  - Saves and restores registers \$s0 through \$s7 when compiling a subroutine (or procedure) that uses them

Assembler

- System program
- Input: Assembly language code
- Output: Machine language code (*binary* or *object code*) contains:
  - Machine instructions
  - Data
  - Symbol table
- Converts pseudoinstructions into core instructions using register $at
- Converts assembly code into machine code
- Creates a symbol table – labels and locations in code

Assembler and Linker



Symbols

- Program names and labels are symbols.
- In executable code, a symbol is a memory address.
- In object code, a symbol's final value (address) has not been determined:
  - Internal symbols (statement labels) have relative addresses.
  - External symbols (called subroutine names, library procedures) are unknown (unresolved).
- See example on page 126 of textbook (page 110 in old version).

Linker

- System program.
- Inputs: Program and procedure libraries, all in machine code.
- Output: Executable machine code.
- Linker functions:
  - Links the machine code of procedures from a library.
  - Sets memory addresses of data variables for procedures.
  - Sets procedure addresses in the calling program code.

Loader

- System program
- Inputs: Machine code and data from disc
- Output: Set up program and data in memory
- Loader functions:
  - Read executable code and data from disc to computer memory
  - Initialize registers and set stack pointer to first free location
  - Transfer control to a *start-up* routine that calls the main routine of the program

Recursive and Nested Programs

The following convention is understood and used by all calling (caller) and called (callee) programs.

Preserved	Not preserved
Saved reg. $s0 - $s7	Temp. reg. $t0 - $t9
Stack pointer reg. $sp	Argument reg. $a0 - $a3
Return addr. Reg. $ra	Return value reg. $v0 - $v1
Stack above the stack pointer	Stack below the stack pointer

When Callee becomes a Caller

- Saving and restoring of saved and temporary registers is done same as described before.
- May reuse argument registers ($a0 - $a3); they are saved and restored as necessary.
- Must reuse $ra; its content is saved in memory and restored on return.

Example: Program→Callee A→Callee B

Main program

```
.
.
addi  $a0, $zero, 3
jal A
.
.
```

Procedure A

```
.
.
addi  $sp, $sp, -8
sw  $ra, 4($sp)
sw  $a0, 0($sp)
addi  $a0, $zero, 7
jal B
lw  $a0, 0($sp)
lw  $ra, 4($sp)
addi $sp, $sp, 8
.
.
jr  $ra
```

Procedure B

```
.
.
.
jr  $ra
```

Example: A Recursive Procedure

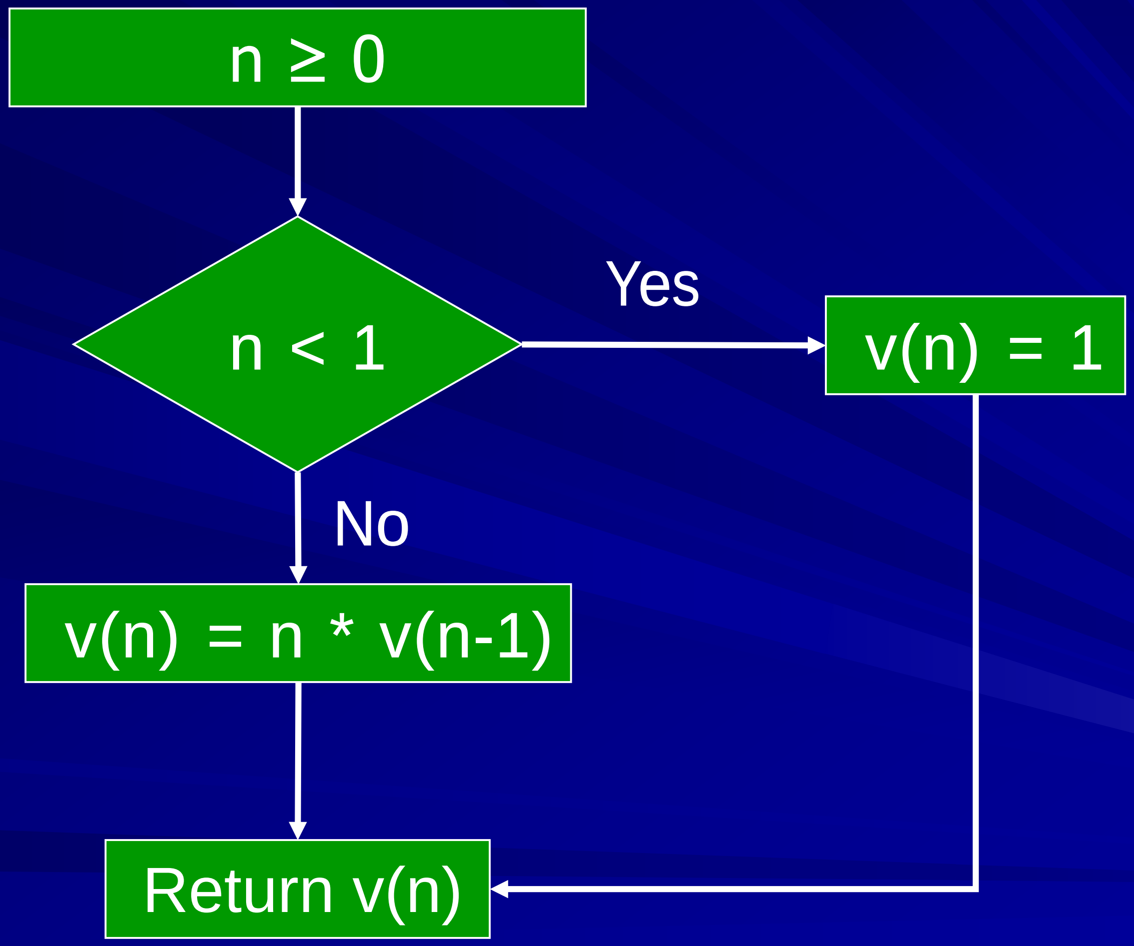
```
Int     fact (n)
{
        if (n < 1)      return (1);
                else    return (n * fact (n-1));
}
```

This procedure returns factorial of integer n, i.e.,

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 = n \times (n-1)!$$

Boundary case, 0! = 1

Flowchart of *fact*



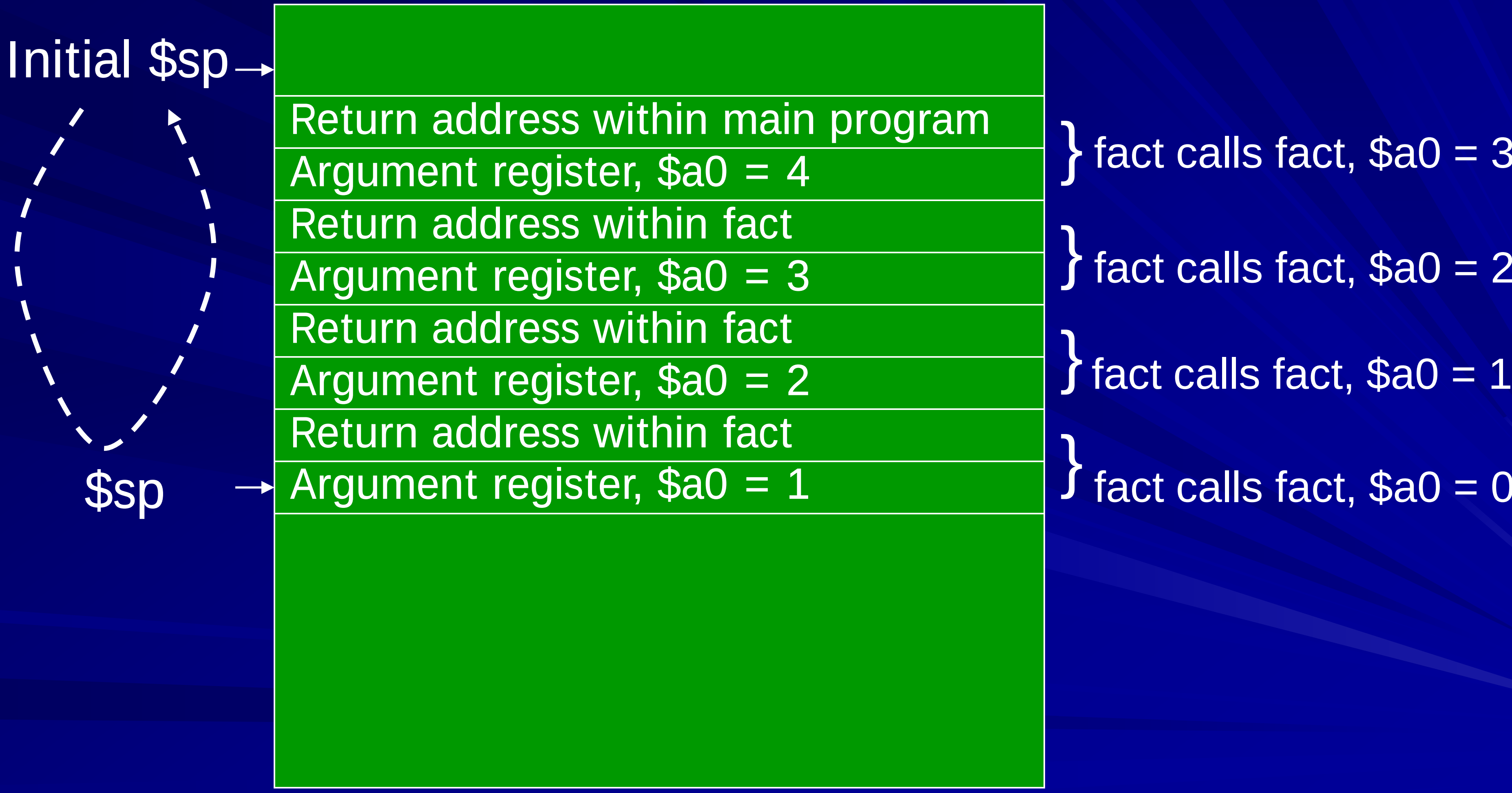
Example: Compute 4!

	n = 4	
Main program calls fact, $a0 = 4		Returns $v0 = $v0 * $a0 = 24
	v(4) = 4 * v(3)	
fact calls fact, $a0 = 3		Returns $v0 = $v0 * $a0 = 6
	v(3) = 3 * v(2)	
fact calls fact, $a0 = 2		Returns $v0 = $v0 * $a0 = 2
	v(2) = 2 * v(1)	
fact calls fact, $a0 = 1		Returns $v0 = $v0 * $a0 = 1
	v(1) = 1 * v(0)	
fact calls fact, $a0 = 0		Returns $v0 = 1
	v(0) = 1	

What to Save?

- Recursive procedure will call itself. So,
  - will reuse return address register $ra
  - will change argument register $a0
- These two registers must be saved.

Saved Registers for 4! Example



Assembly Code for *fact*

```
fact:
        addi    $sp, $sp, -8      # adjust stack for two items
        sw      $ra, 4($sp)       # save return address of caller
        sw      $a0, 0($sp)       # save caller supplied argument n

        slti    $t0, $a0, 1       # $t0 = 1, if n ≤ 1, i.e., n = 0
        beq     $t0, $zero, L1    # go to L1, if n ≥ 1

        addi    $v0, $zero, 1     #  return $v0 = 1 to caller

        addi    $sp, $sp, 8       # no need to restore registers, since
                                          none was changed, but must
                                          restore stack pointer
        jr      $ra               # return to caller instruction after jal
```

Assembly Code Continued

```
L1:   addi $a0, $a0, -1        # set $a0 to n-1
      jal  fact                # call fact with argument n-1

      lw  $a0, 0($sp)          # on return from fact, restore n
      lw  $ra, 4($sp)          # restore return address
      addi  $sp, $sp, 8        # adjust stack pointer

      mul  $v0, $a0, $v0       # n! = n × (n-1)!

      jr  $ra                  # return to caller
```

Execution of *fact* for n ≥ 1

Call Sequence	Caller	#a0	$ra	Returned $v0
1	Main Program	n	PC+4	n×(n-1)!
2	fact	n-1	L1+8	(n-1)×(n-2)!
3	fact	n-2	L1+8	(n-2)×(n-3)!
	fact	.		
	fact	.		
	fact	.		
n-2	fact	3	L1+8	3×2 = 6
n-1	fact	2	L1+8	2×1 = 2
n	fact	1	L1+8	1×1 = 1
n+1	fact	0	L1+8	1

Summary

- A user's program is processed by several system programs:
  - Compiler
  - Assembler
  - Linker
  - Loader
  - Start-up routine: begins program execution and at the end of the program issues an *exit* system call.