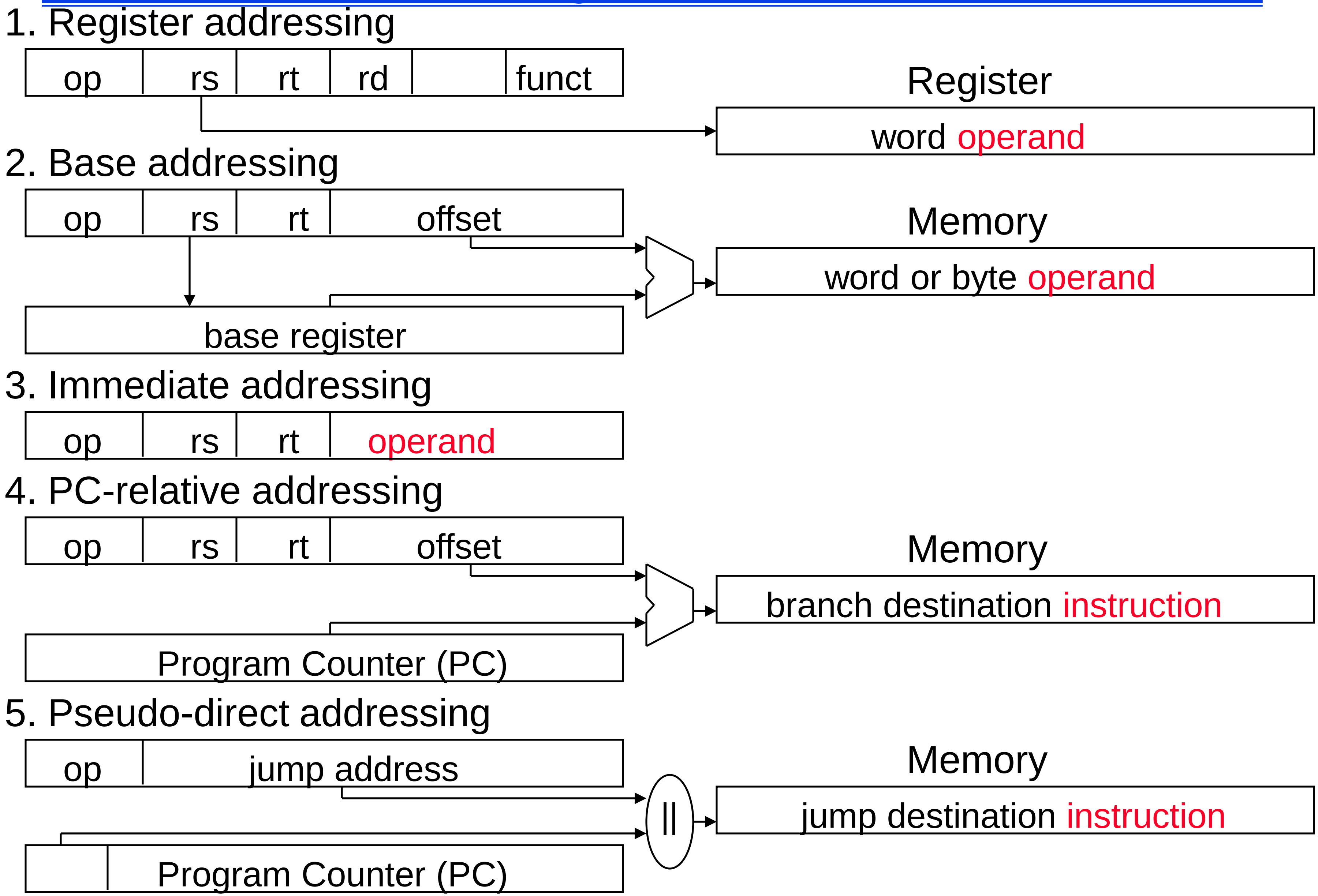
C335
Computer Structures

Assemblers and Linkers

Dr. Liqiang Zhang

Department of Computer and Information Sciences

Review: Addressing Modes Illustrated



Review: MIPS Instructions, so far

Category	Instr	OpC	Example	Meaning
Arithmetic (R & I format)	add	0 & 20	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 & 22	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 4	$s1 = $s2 + 4
	shift left logical	0 & 00	sll $s1, $s2, 4	$s1 = $s2 << 4
	shift right logical	0 & 02	srl $s1, $s2, 4	$s1 = $s2 >> 4 (fill with zeros)
	shift right arithmetic	0 & 03	sra $s1, $s2, 4	$s1 = $s2 >> 4 (fill with sign bit)
	and	0 & 24	and $s1, $s2, $s3	$s1 = $s2 & $s3
	or	0 & 25	or $s1, $s2, $s3	$s1 = $s2 | $s3
	nor	0 & 27	nor $s1, $s2, $s3	$s1 = not ($s2 | $s3)
	and immediate	c	and $s1, $s2, ff00	$s1 = $s2 & 0xff00
	or immediate	d	or $s1, $s2, ff00	$s1 = $s2 | 0xff00
	load upper immediate	f	lui $s1, 0xffff	$s1 = 0xffff0000

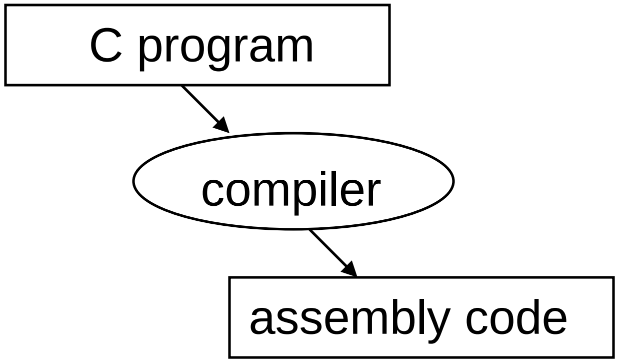
Review: MIPS Instructions, so far

Category	Instr	OpC	Example	Meaning
Data transfer (I format)	load word	23	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	2b	sw $s1, 100($s2)	Memory($s2+100) = $s1
	load byte	20	lb $s1, 101($s2)	$s1 = Memory($s2+101)
	store byte	28	sb $s1, 101($s2)	Memory($s2+101) = $s1
	load half	21	lh $s1, 101($s2)	$s1 = Memory($s2+102)
	store half	29	sh $s1, 101($s2)	Memory($s2+102) = $s1
Cond. branch (I & R format)	br on equal	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	5	bne $s1, $s2, L	if ($s1 !=$s2) go to L
	set on less than immediate	a	slti $s1, $s2, 100	if ($s2<100) $s1=1; else $s1=0
	set on less than	0 & 2a	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1; else $s1=0
Uncond. Jump (J & R format)	jump	2	j 2500	go to 10000
	jump register	0 & 08	jr $t1	go to $t1
	jump and link	3	jal 2500	go to 10000; $ra=PC+4

RISC Design Principles Review

- Simplicity favors regularity
  - fixed size instructions – 32-bits
  - small number of instruction formats
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Good design demands good compromises
  - three instruction formats
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

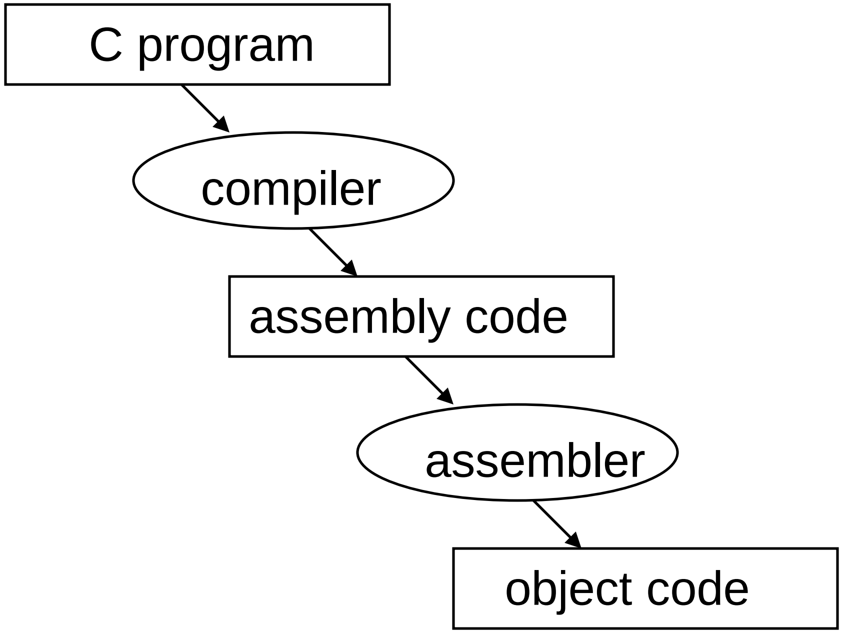
The Code Translation Hierarchy



Compiler

- Transforms the C program into an assembly language program
- Advantages of high-level languages
  - many fewer lines of code
  - easier to understand and debug
- Today's optimizing compilers can produce assembly code nearly as good as an assembly language programming expert and often better for large programs
  - smaller code size, faster execution

The Code Translation Hierarchy



Assembler

- Transforms symbolic assembler code into object (machine) code
- Advantages of assembler
  - much easier than remembering instr's binary codes
  - can use labels for addresses – and let the assembler do the arithmetic
  - can use pseudo-instructions
    - e.g., "move $t0, $t1" exists only in assembler (would be implemented using "add $t0,$t1,$zero")
- When considering performance, you should count instructions executed, not code size

The Two Main Tasks of the Assembler

- Finds the memory locations with labels so the relationship between the symbolic names and their addresses is known
  - Symbol table – holds labels and their corresponding addresses
    - A label is local if the object is used only within the file where its defined. Labels are local by default.
    - A label is external (global) if it refers to code or data in another file or if it is referenced from another file. Global labels must be explicitly declared global (e.g., `.globl main`)
- Translates each assembly language statement by combining the numeric equivalent of the opcodes, register specifiers, and labels

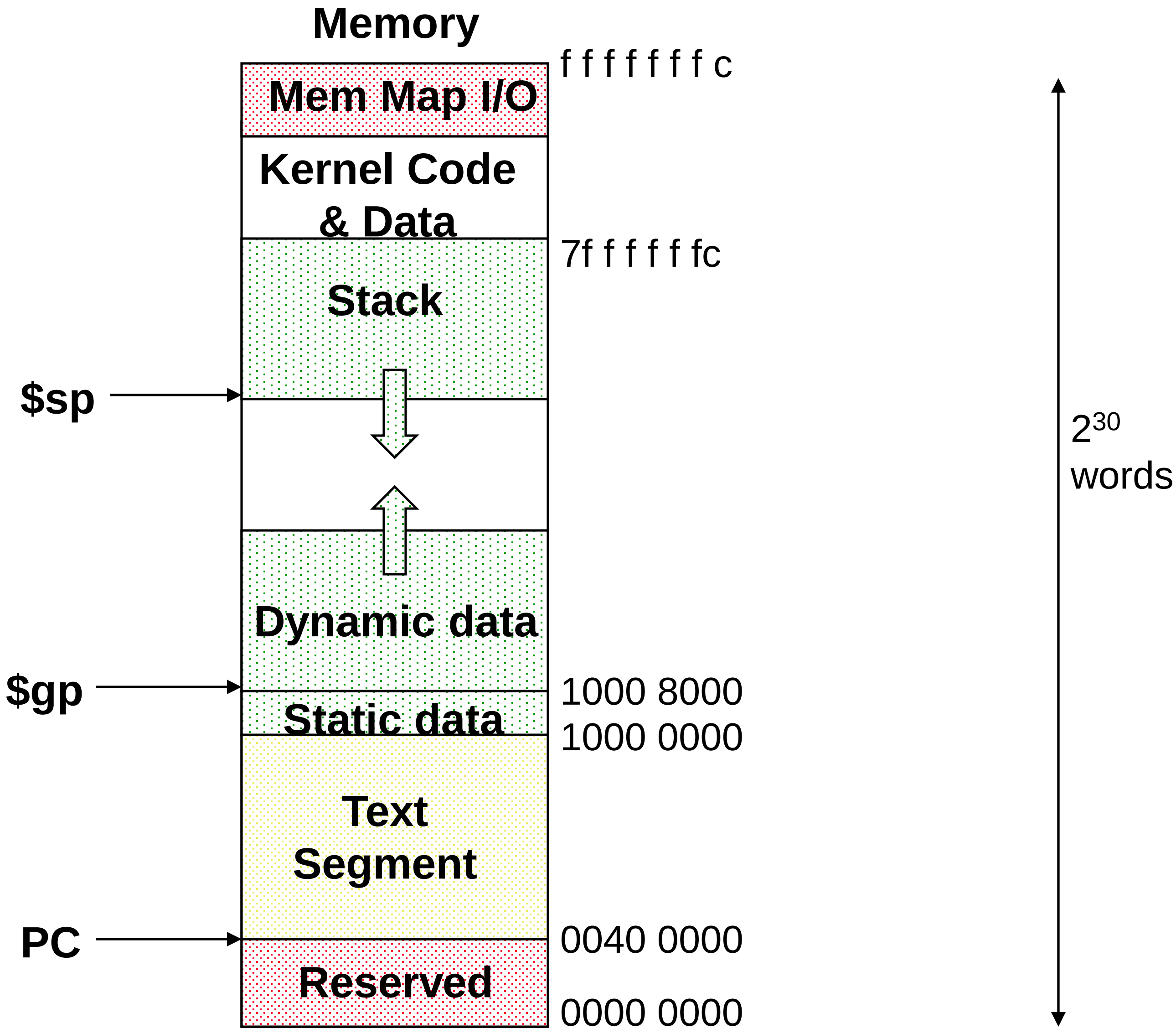
Other Tasks of the Assembler

- Converts pseudo-instr's to legal assembly code
  - register $at is reserved for the assembler to do this
- Converts branches to far away locations into a branch followed by a jump
- Converts instructions with large immediate into a `lui` followed by an `ori`
- Converts numbers specified in decimal and hexadecimal into their binary equivalents and characters into their ASCII equivalents
- Deals with data layout directives (e.g., `.asciiz`)
- Expands macros (frequently used sequences of instructions)

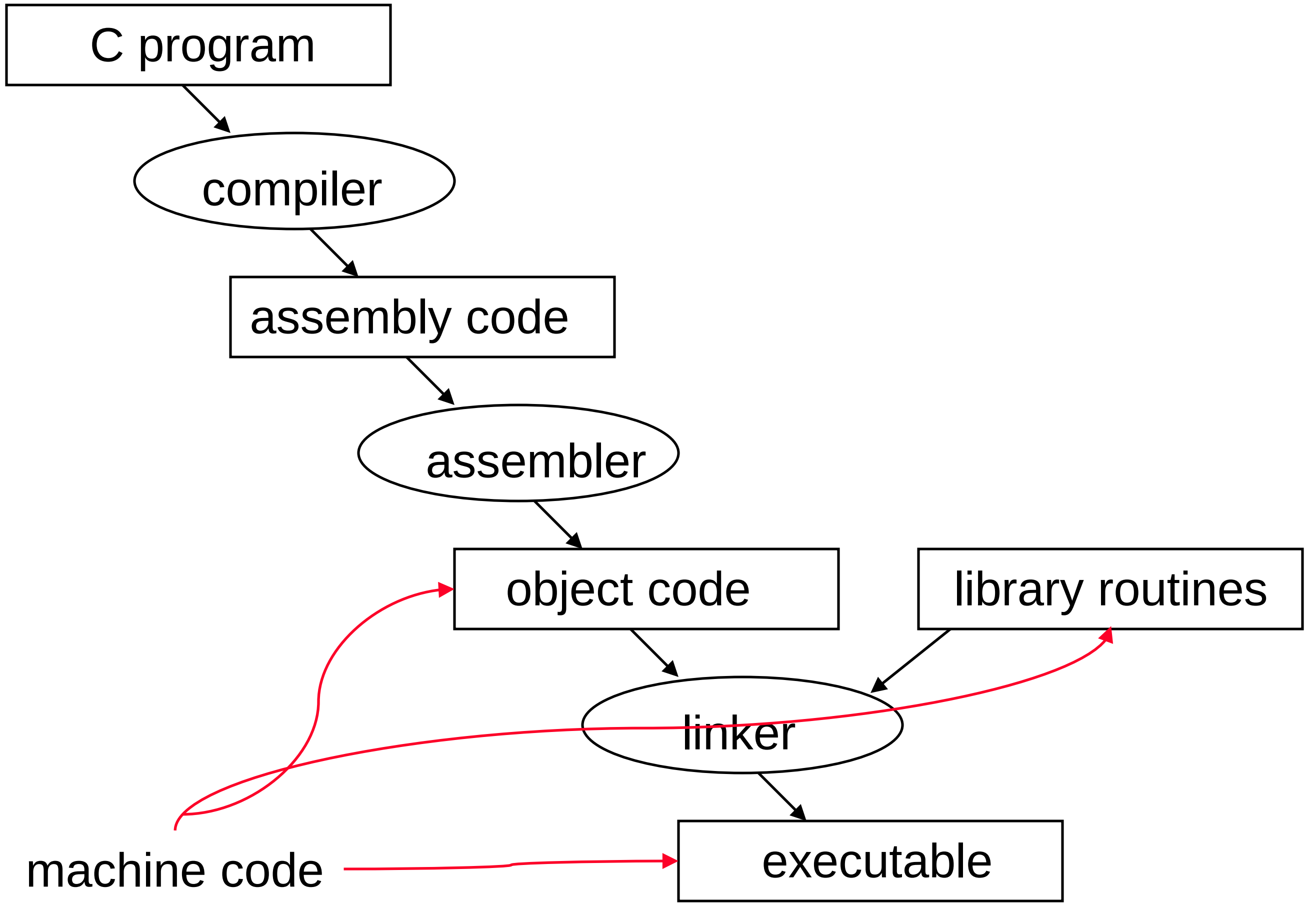
Typical Object File Pieces

- Object file header: size and position of the following pieces of the file
- Text (code) segment (`.text`): assembled object (machine) code
- Data segment (`.data`): data accompanying the code
  - static data - allocated throughout the program
  - dynamic data - grows and shrinks as needed
- Relocation information: identifies instructions (data) that use (are located at) absolute addresses – *not* relative to a register
  - on MIPS only `j`, `jal`, and some loads and stores (e.g., `lw $t1, 100($zero)`) use absolute addresses
- Symbol table: remaining undefined labels (external references to labels in other object files or libraries)
- Debugging information

MIPS (spim) Memory Allocation



The Code Translation Hierarchy



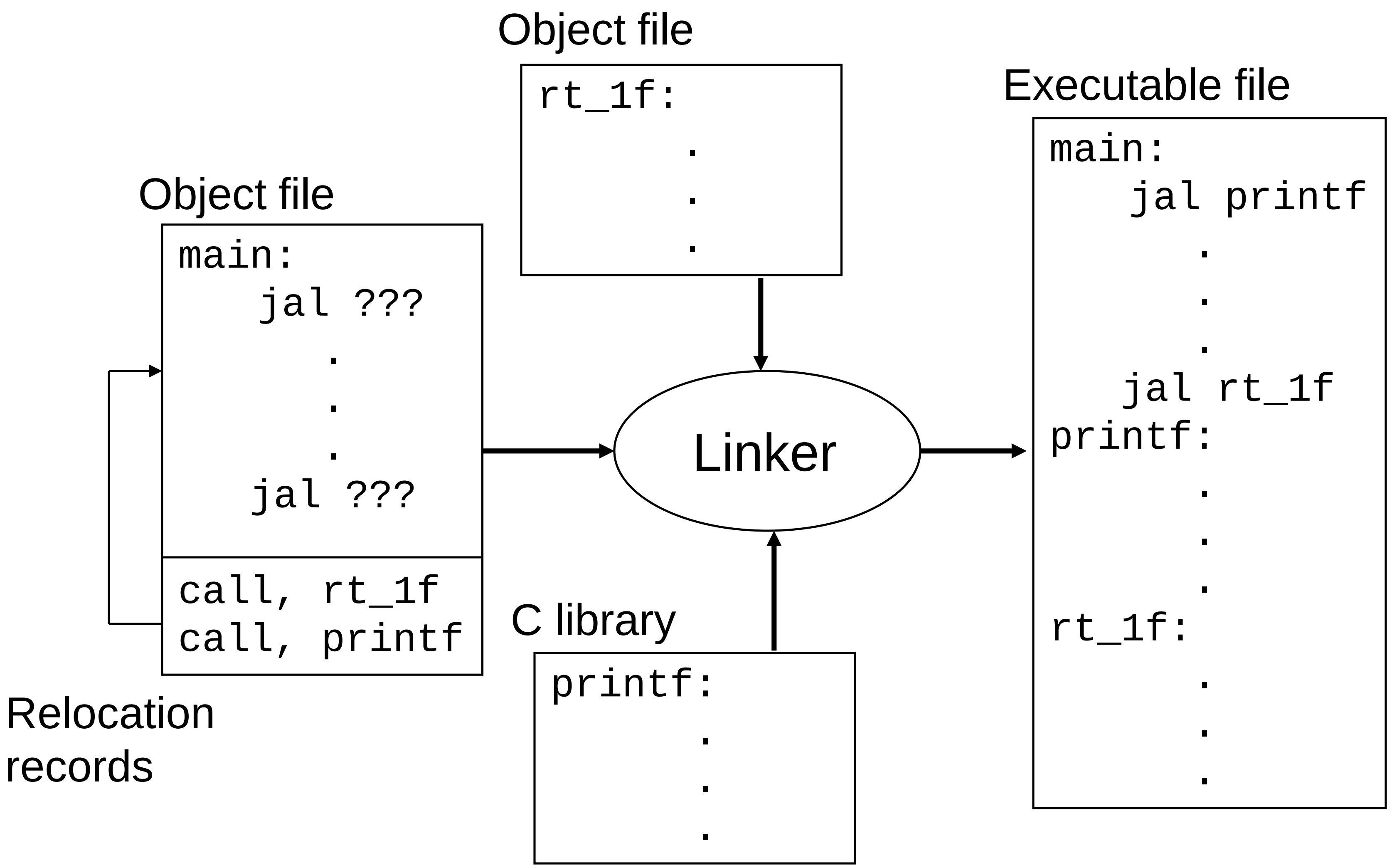
Linker

- Takes all of the independently assembled code segments and "stitches" (links) them together
  - Faster to recompile and reassemble a patched segment, than it is to recompile and reassemble the *entire* program

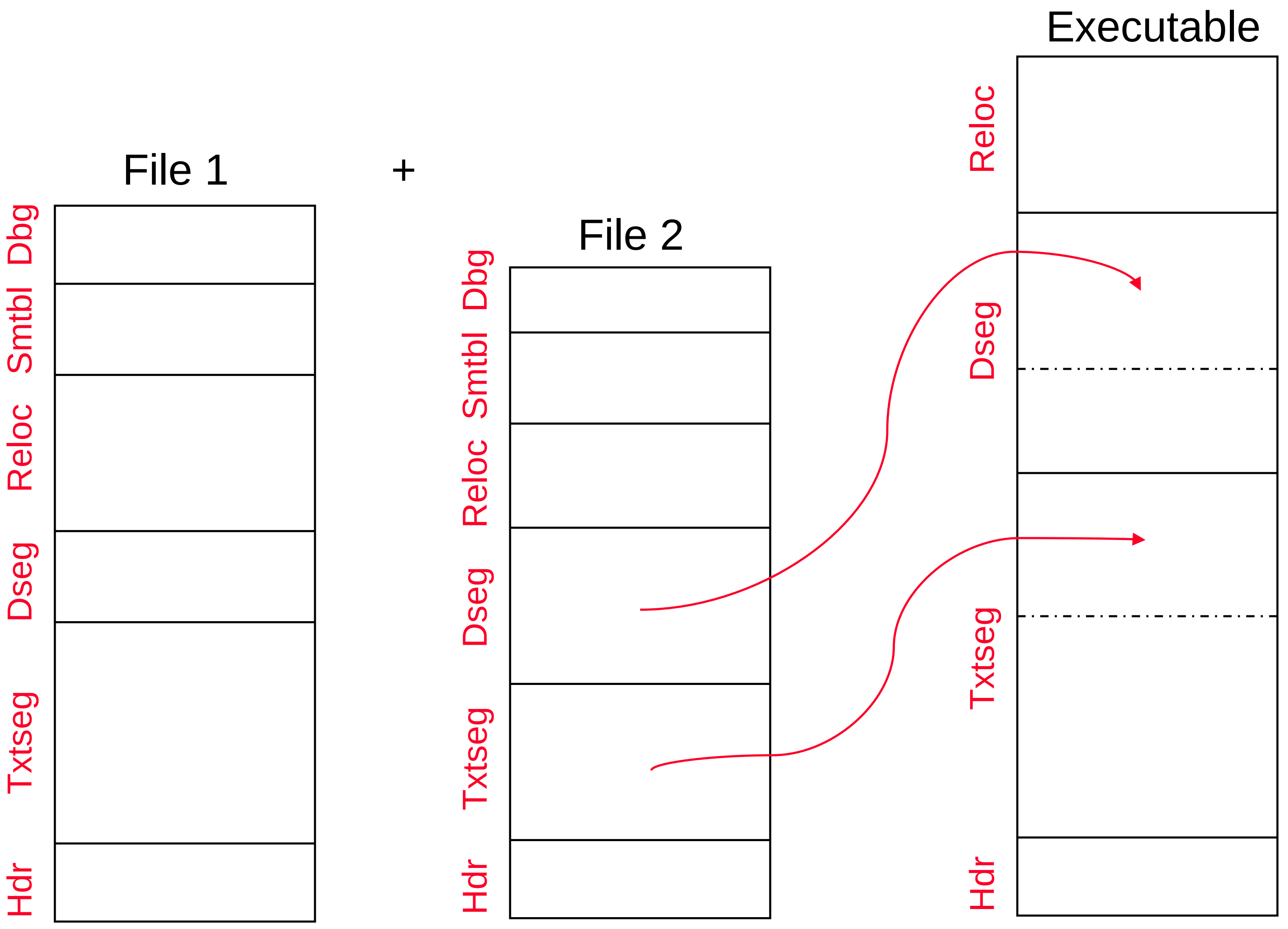
1. Decides on memory allocation pattern for the code and data modules of each segment
   - Remember, segments were assembled in isolation so each has assumed its code's starting location is 0x0040 0000 and its static data starting location is 0x1000 0000
2. Relocates absolute addresses to reflect the new starting location of the code segment and its data module
3. Uses the symbol tables information to resolve all remaining undefined labels
   - branches, jumps, and data addresses to/in external segments

- Linker produces an executable file

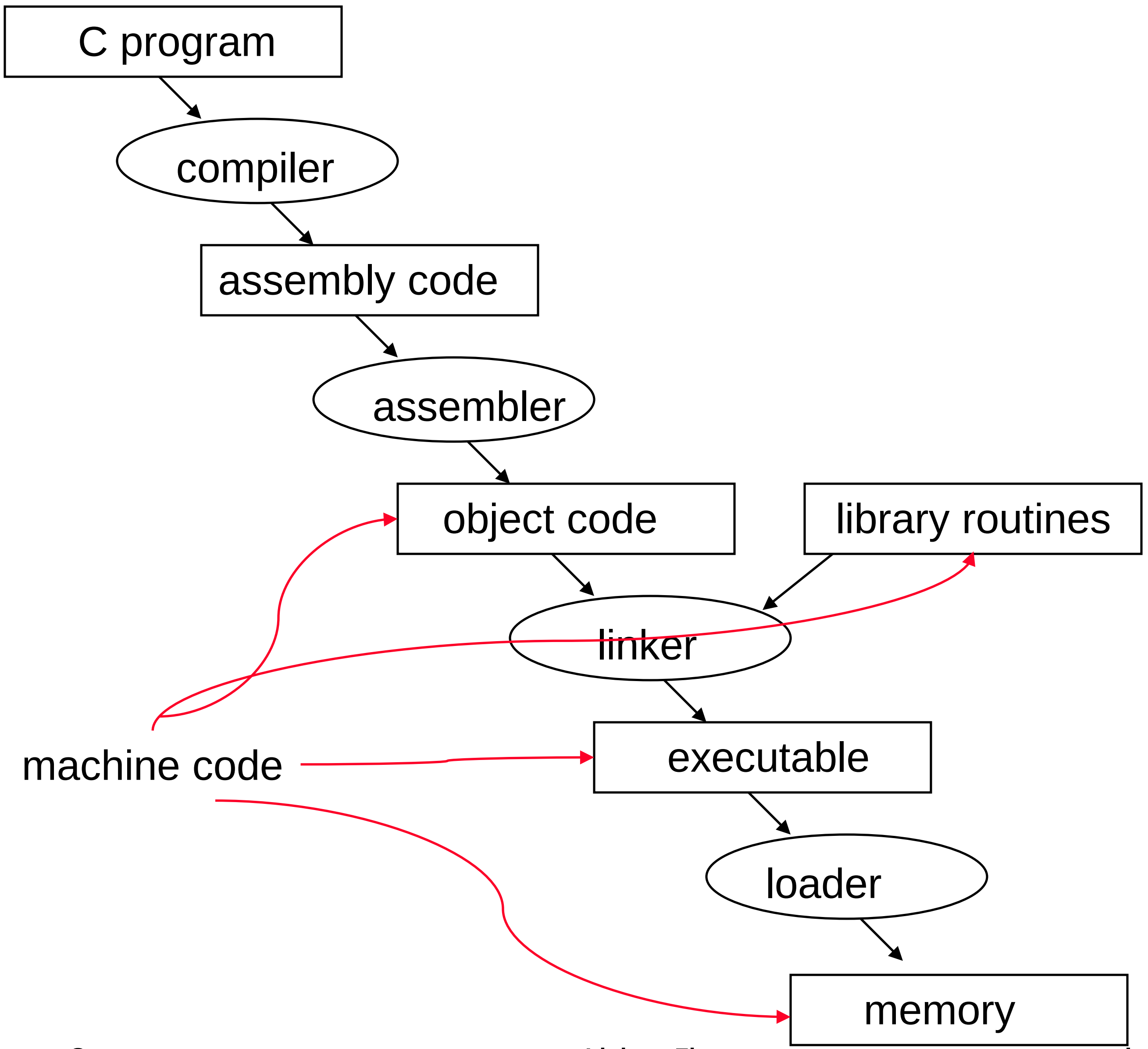
Linker Code Schematic



Linking Two Object Files



The Code Translation Hierarchy



Loader

- Loads (copies) the executable code now stored on disk into memory at the starting address specified by the operating system
- Copies the parameters (if any) to the main routine onto the stack
- Initializes the machine registers and sets the stack pointer to the first free location (0x7fff fffc)
- Jumps to a start-up routine (at PC addr 0x0040 0000 on spim) that copies the parameters into the argument registers and then calls the main routine of the program with a `jal main`

Dynamically Linked Libraries

- Statically linking libraries mean that the library becomes part of the executable code
  - It loads all the routines in the library that are called anywhere in the executable even if those calls are not executed.
  - What if a new version of the library is released ?
- Dynamically linked libraries (DLL) – library routines are not linked and loaded until a routine is called during execution
  - The first time the library routine called, a dynamic linker-loader must
    - find the desired routine, remap it, and “link” it to the calling routine
    - DLLs require extra space for dynamic linking information, but do not require the all the routines to be copied or linked