Computer Architecture & Assembly Language 14:332:331

Lecture 4

Addressing Mode, Assembler, Linker

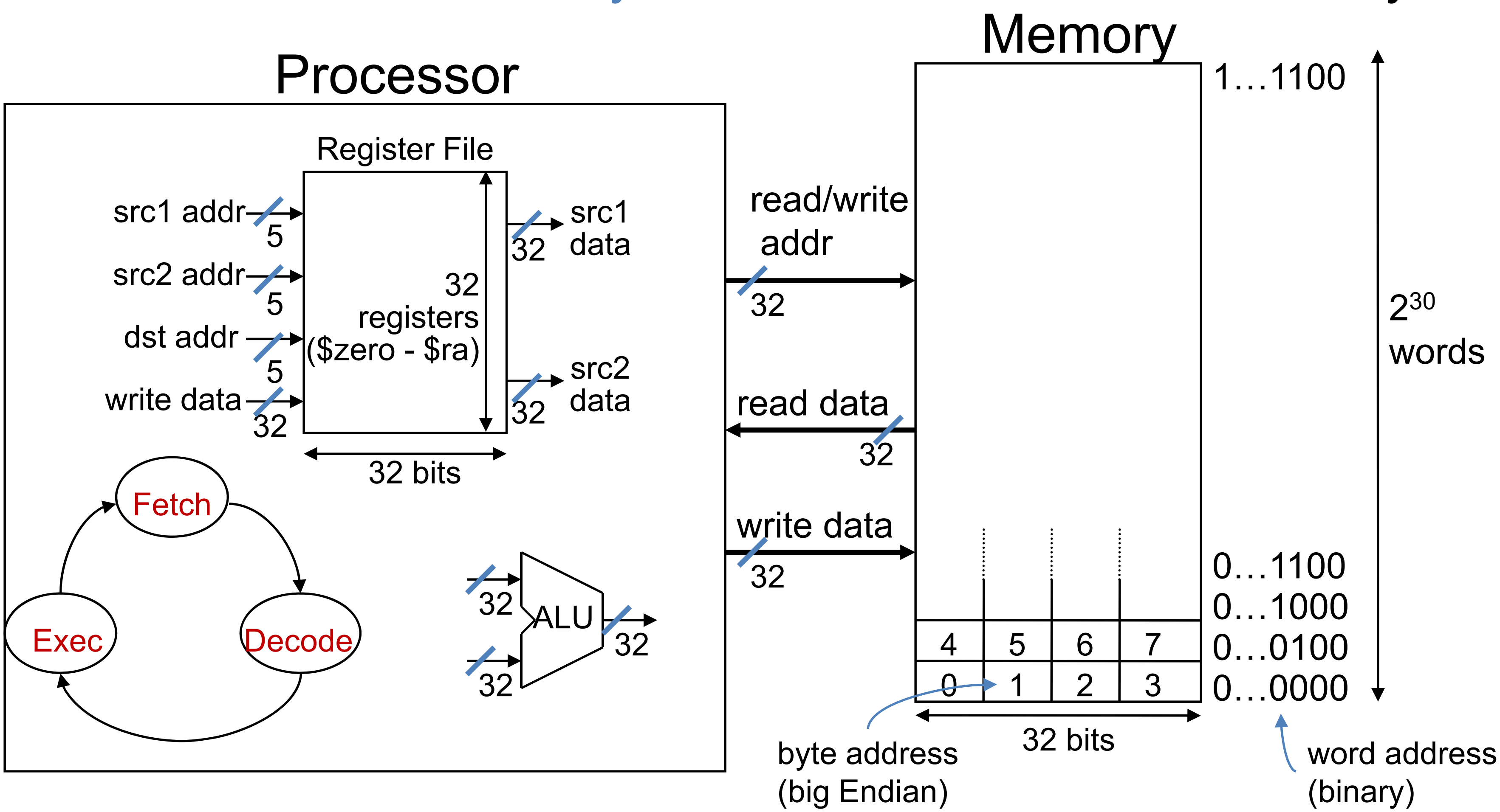
Naghmeh Karimi

Fall 16

Adapted from *Computer Organization and Design*, *5th Edition*, Patterson & Hennessy, © 2013, Elsevier, *and Computer Organization and Design*, *4th Edition*, Patterson & Hennessy, © 2008, Elsevier and Mary Jane Irwin's slides from Penn State University.

Review: MIPS Organization

- Arithmetic instructions – to/from the register file
- Load/store word and byte instructions – from/to memory



Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rt, constant
```

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

`lui $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

String Copy Example

- C code (naïve):
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- MIPS code:

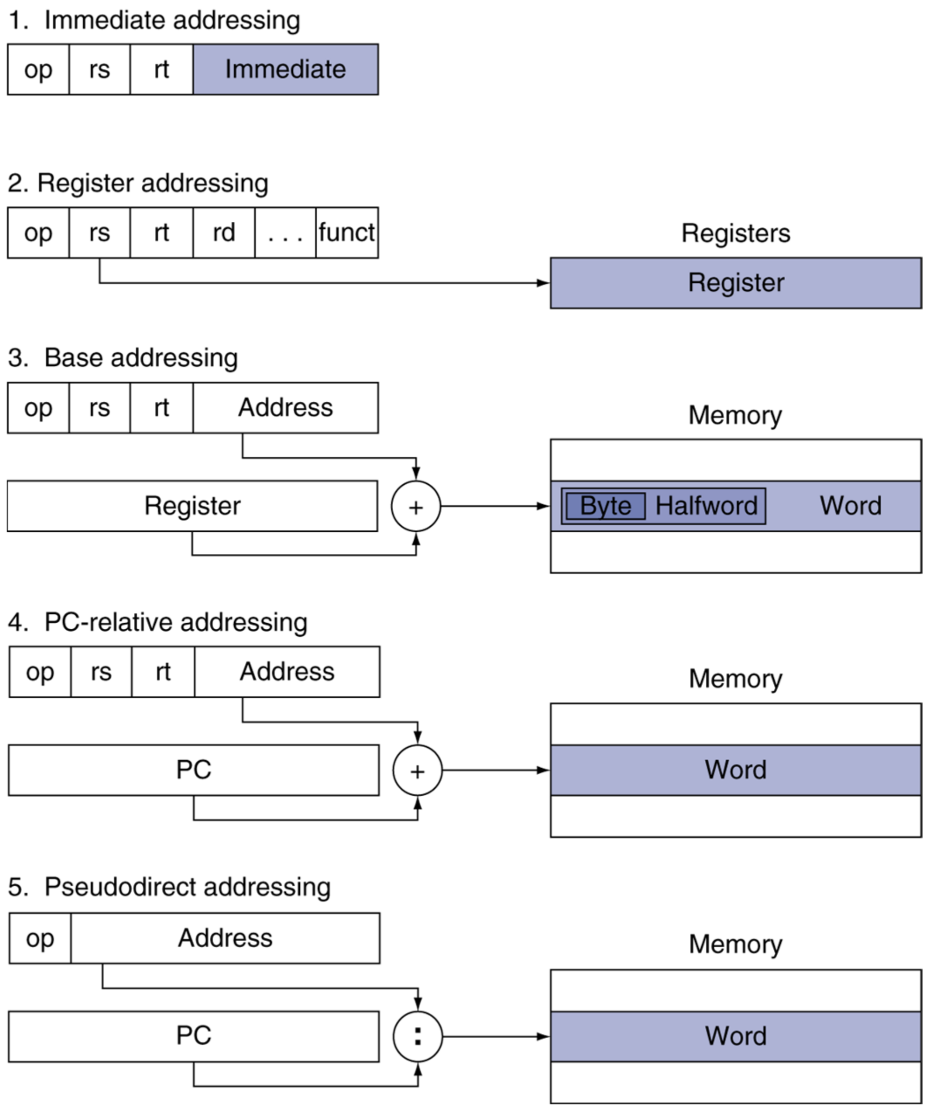
```
strcpy:
      addi $sp, $sp, -4       # adjust stack for 1 item
      sw   $s0, 0($sp)        # save $s0
      add  $s0, $zero, $zero  # i = 0
L1:   add  $t1, $s0, $a1      # addr of y[i] in $t1
      lbu  $t2, 0($t1)        # $t2 = y[i]
      add  $t3, $s0, $a0      # addr of x[i] in $t3
      sb   $t2, 0($t3)        # x[i] = y[i]
      beq  $t2, $zero, L2     # exit loop if y[i] == 0
      addi $s0, $s0, 1        # i = i + 1
      j    L1                 # next iteration of loop
L2:   lw   $s0, 0($sp)        # restore saved $s0
      addi $sp, $sp, 4        # pop 1 item from stack
      jr   $ra                # and return
```

MIPS Addressing Modes

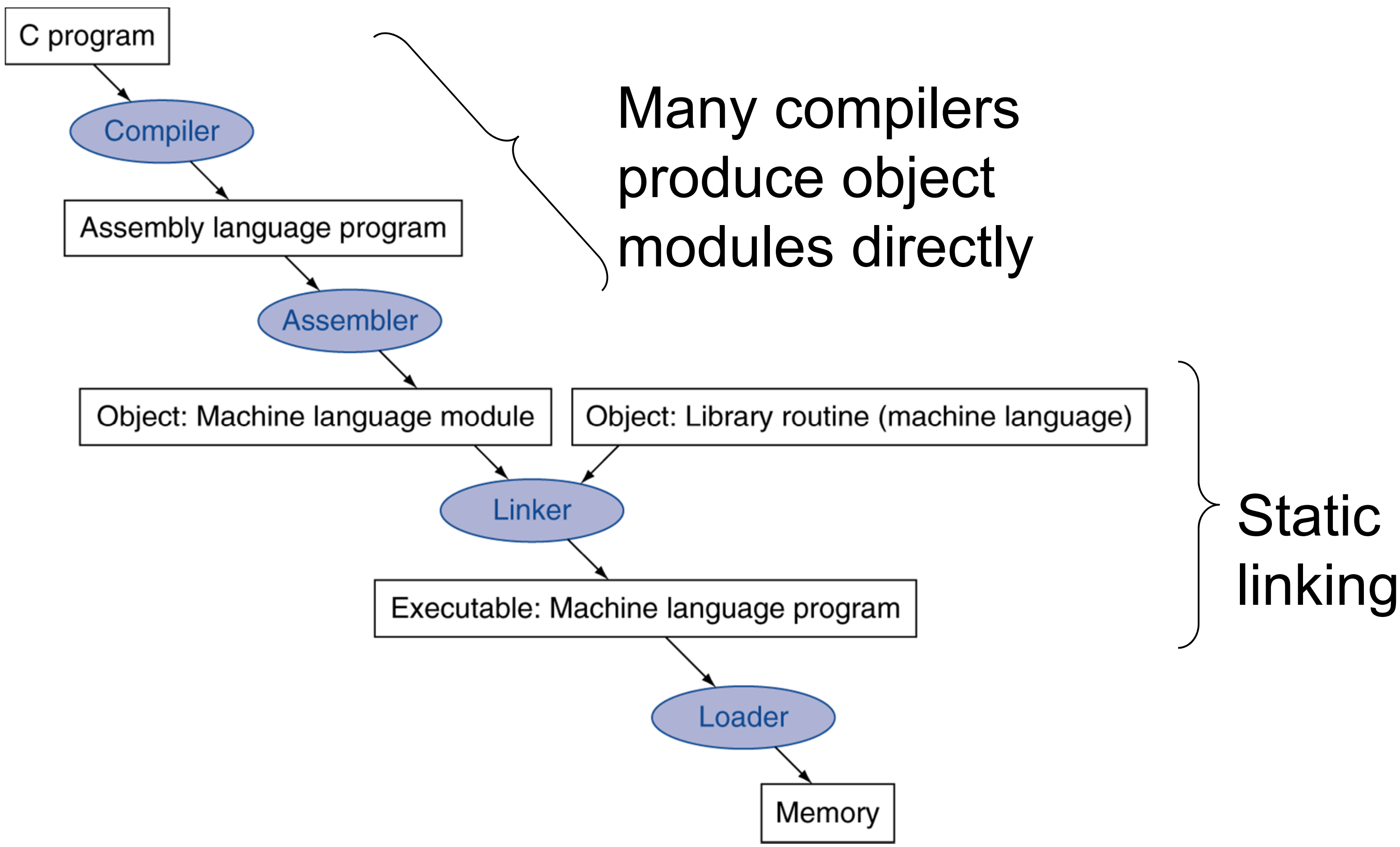
- **Register addressing** – operand is in a register
- **Base (displacement) addressing** – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction
- **Immediate addressing** – operand is a 16-bit constant contained within the instruction

- **PC-relative addressing** –instruction address is the sum of the PC and a 16-bit constant contained within the instruction
- **Pseudo-direct addressing** – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC

Addressing Mode Summary



Translation and Startup



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
  – good – smaller code size, faster execution

Assembler

- Transforms symbolic assembler code into object (machine) code
- Advantages of assembly language
  - Programmer has more control compared to higher level language
  - much easier than remembering instruction 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")
- However, must remember that machine language is the underlying reality

- And, when considering performance, you should count real instructions executed, not code size

Other Tasks of the Assembler

- Determines binary addresses corresponding to all labels
  - keeps track of labels used in branches and data transfer instructions in a symbol table
    - pairs of symbols and addresses
- Converts pseudo-instructions 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 immediates into a load upper immediate followed by an or immediate
- Converts numbers specified in decimal and hexidecimal into their binary equivalents
- Converts characters into their ASCII equivalents

Assembler Pseudoinstructions

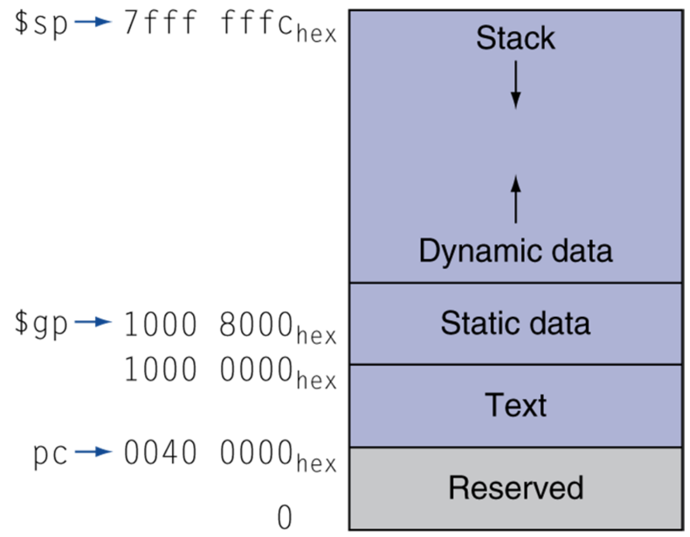
- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1     →  add $t0, $zero, $t1
blt $t0, $t1, L   →  slt $at, $t0, $t1
                     bne $at, $zero, L
```

  - $at (register 1): assembler temporary

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: E.g., malloc in C, new in Java
- Stack: automatic storage



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Typical Object File Pieces

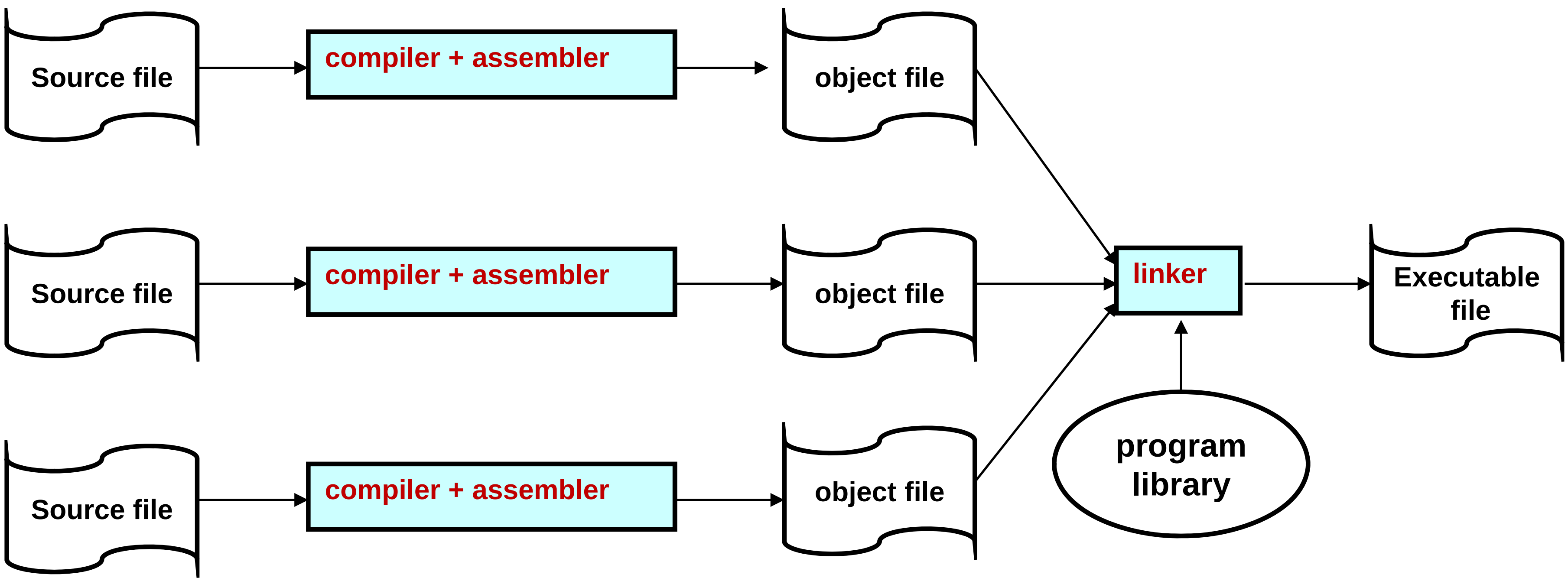
- Object file header: size and position of following pieces
- Text module: assembled object (machine) code
- Data module: data accompanying the code
  - static data - allocated throughout the program
  - dynamic data - grows and shrinks as needed by the program
- Relocation information: identifies instructions (data) that use (are located at) absolute addresses – those that are not relative to a register (e.g., jump destination addr) – when the code and data is loaded into memory
- Symbol table: remaining undefined labels (e.g., external references)
- Debugging information

Object file header	text segment	data segment	relocation information	symbol table	debugging information

Linker

- Takes all of the independently assembled code segments and "stitches" (links) them together
  - Much faster to patch code and recompile and reassemble that patched routine, than it is to recompile and reassemble the entire program
- Decides on memory allocation pattern for the code and data modules of each segment
  - remember, segments were assembled in isolation so each assumes its code's starting location is 0x0040 0000 and its static data starting location is 0x1000 0000
- Absolute addresses must be relocated to reflect the new starting location of each code and data module
- Uses the symbol table information to resolve all remaining undefined labels
  - branches, jumps, and data addresses to external segments

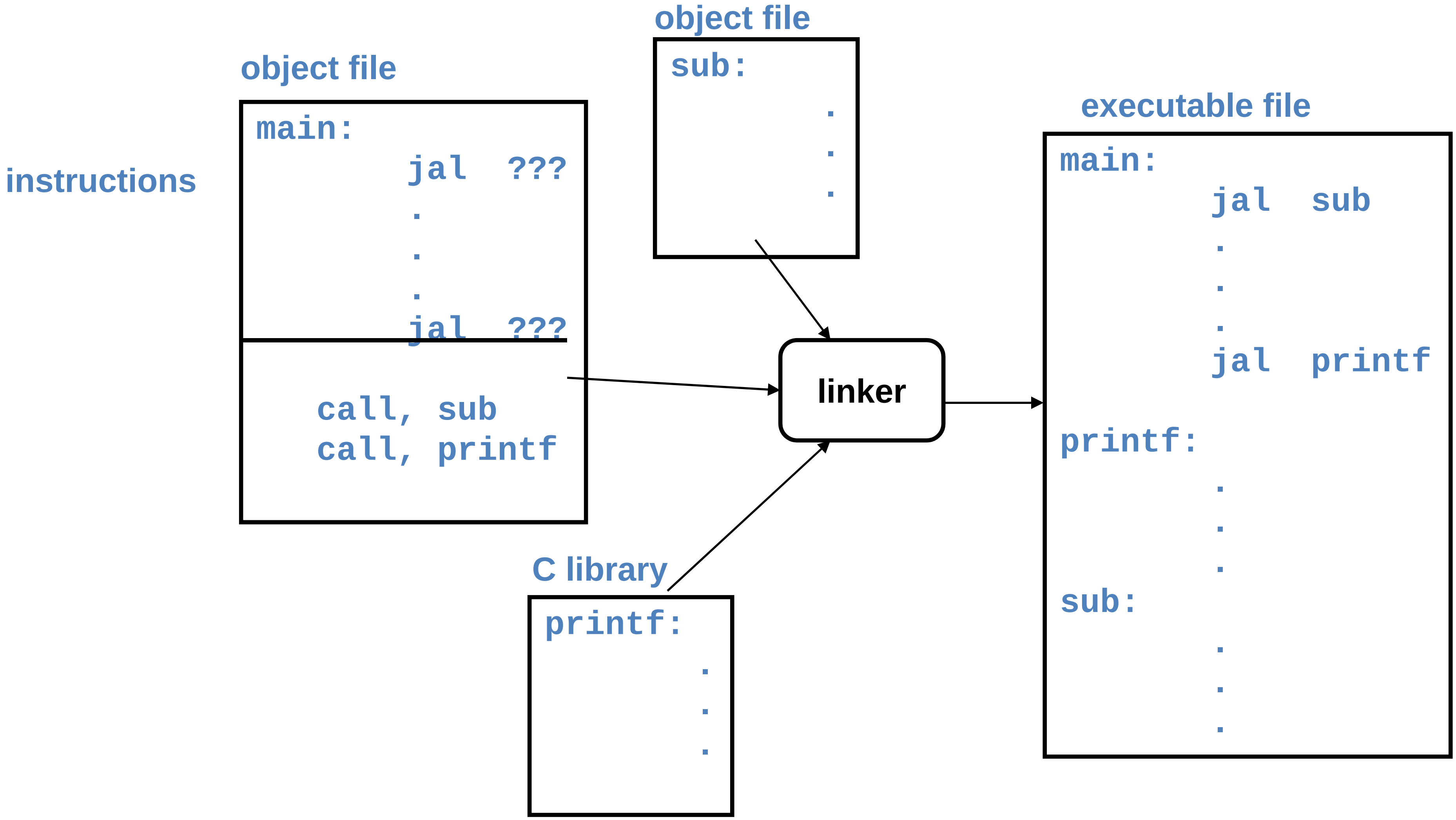
Process that produces an executable file



Loader

- Loads (copies) the executable code now stored on disk into memory at the starting address specified by the operating system
- Initializes the machine registers and sets the stack pointer to the first free location (0x7ffe fffc)
- Copies the parameters (if any) to the main routine onto the stack
- Jumps to a start-up routine (at PC addr 0x0040 0000 on xspim) that copies the parameters into the argument registers and then calls the main routine of the program with a `jal main`

Loader



object file
object file
sub:
. . .
main:
jal ???
. . .
jal ???
call, sub
call, printf
instructions
executable file
main:
jal sub
. . .
jal printf
printf:
. . .
sub:
. . .
linker
C library
printf:
. . .

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

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

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%