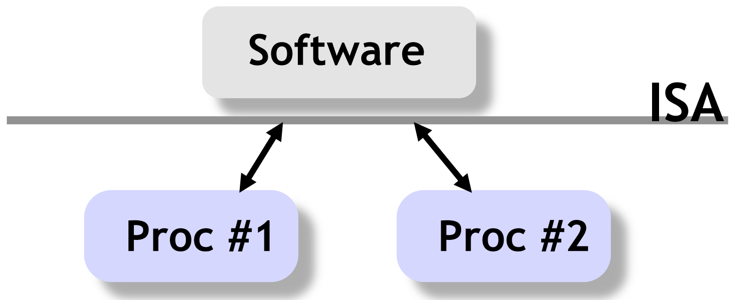
Instruction sets & RISC vs. CISC, Compilers, Assemblers, Linkers, & Loaders, malloc/new & Memory images, and who cares about assembly.

Today's lecture

- **ISA review & history**
- **Compilation process**
- **Types of memory & memory image**
  - Global, automatic (stack), and heap
- **Loading**
- **What is assembly programming good for?**

Instruction Set Architecture (ISA)

- **The ISA is the interface between hardware and software.**



- **The ISA serves as an abstraction layer between the HW and SW**
  - Software doesn't need to know how the processor is implemented
  - Any processor that implements the ISA appears equivalent

A little ISA history

- **1964: IBM System/360, the first computer family**
  - IBM wanted to sell a range of machines that ran the same software
- **1960's, 1970's: Complex Instruction Set Computer (CISC) era**
  - Much assembly programming, compiler technology immature
  - Simple machine implementations
  - Complex instructions simplified programming, little impact on design
- **1980's: Reduced Instruction Set Computer (RISC) era**
  - Most programming in high-level languages, mature compilers
  - Aggressive machine implementations
  - Simpler, cleaner ISA's facilitated pipelining, high clock frequencies
- **1990's: Post-RISC era**
  - ISA complexity largely relegated to non-issue
  - CISC and RISC chips use same techniques (pipelining, superscalar, ..)
  - ISA compatibility outweighs any RISC advantage in general purpose
  - Embedded processors prefer RISC for lower power, cost
- **2000's: Multi-core and Multithreading**

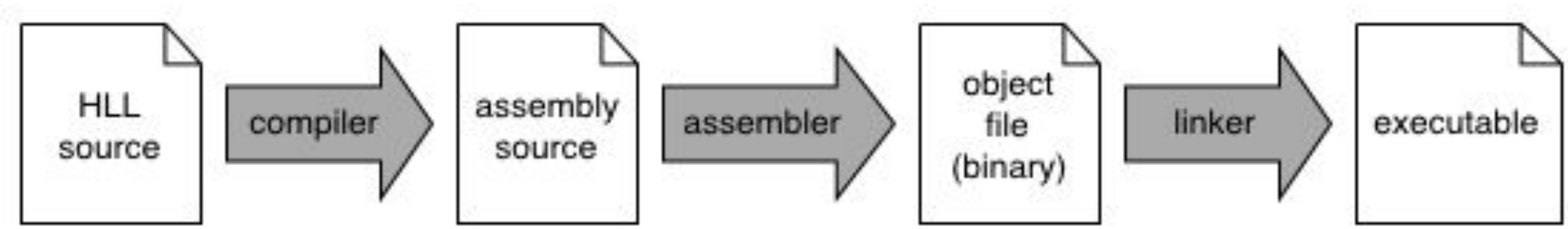
RISC vs. CISC

- **MIPS was one of the first RISC architectures. It was started about 20 years ago by John Hennessy, one of the authors of our textbook.**
- **The architecture is similar to that of other RISC architectures, including Sun's SPARC, IBM's PowerPC, and ARM-based processors.**
- **Older processors used complex instruction sets, or CISC architectures.**
  - Many powerful instructions were supported, making the assembly language programmer's job much easier.
  - But this meant that the processor was more complex, which made the hardware designer's life harder.
- **Many new processors use reduced instruction sets, or RISC architectures.**
  - Only relatively simple instructions are available. But with high-level languages and compilers, the impact on programmers is minimal.
  - On the other hand, the hardware is much easier to design, optimize, and teach in classes.
- **Even most current CISC processors, such as Intel 8086-based chips, are now implemented using a lot of RISC techniques.**

Differences between ISA's

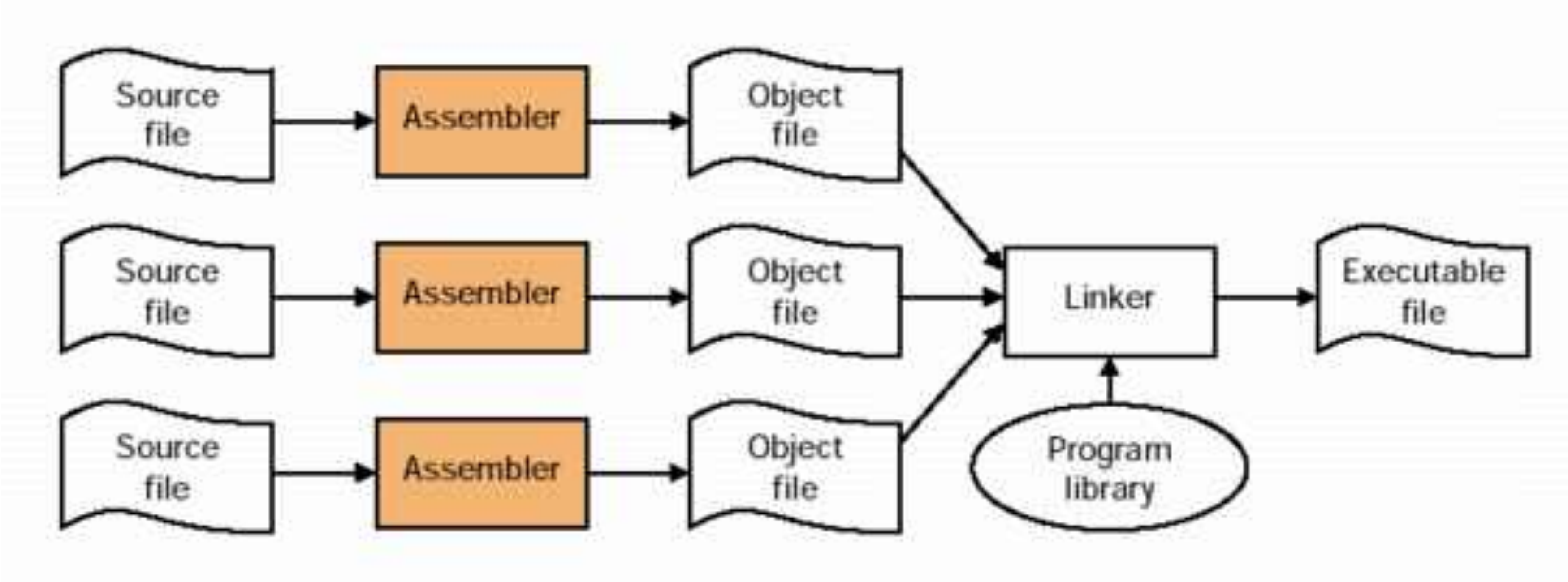
- **Much more is similar between ISA's than different. Compare MIPS & x86:**
  - Instructions:
    - same basic types
    - different names and variable-length encodings
    - x86 branches use condition codes
    - x86 supports (register + memory) -> (register) format
  - Registers:
    - Register-based architecture
    - different number and names, x86 allows partial reads/writes
  - Memory:
    - Byte addressable, 32-bit address space
    - x86 has additional addressing modes
    - x86 does not require addresses to be aligned
    - x86 has segmentation, but not used by most modern O/S's

The compilation process



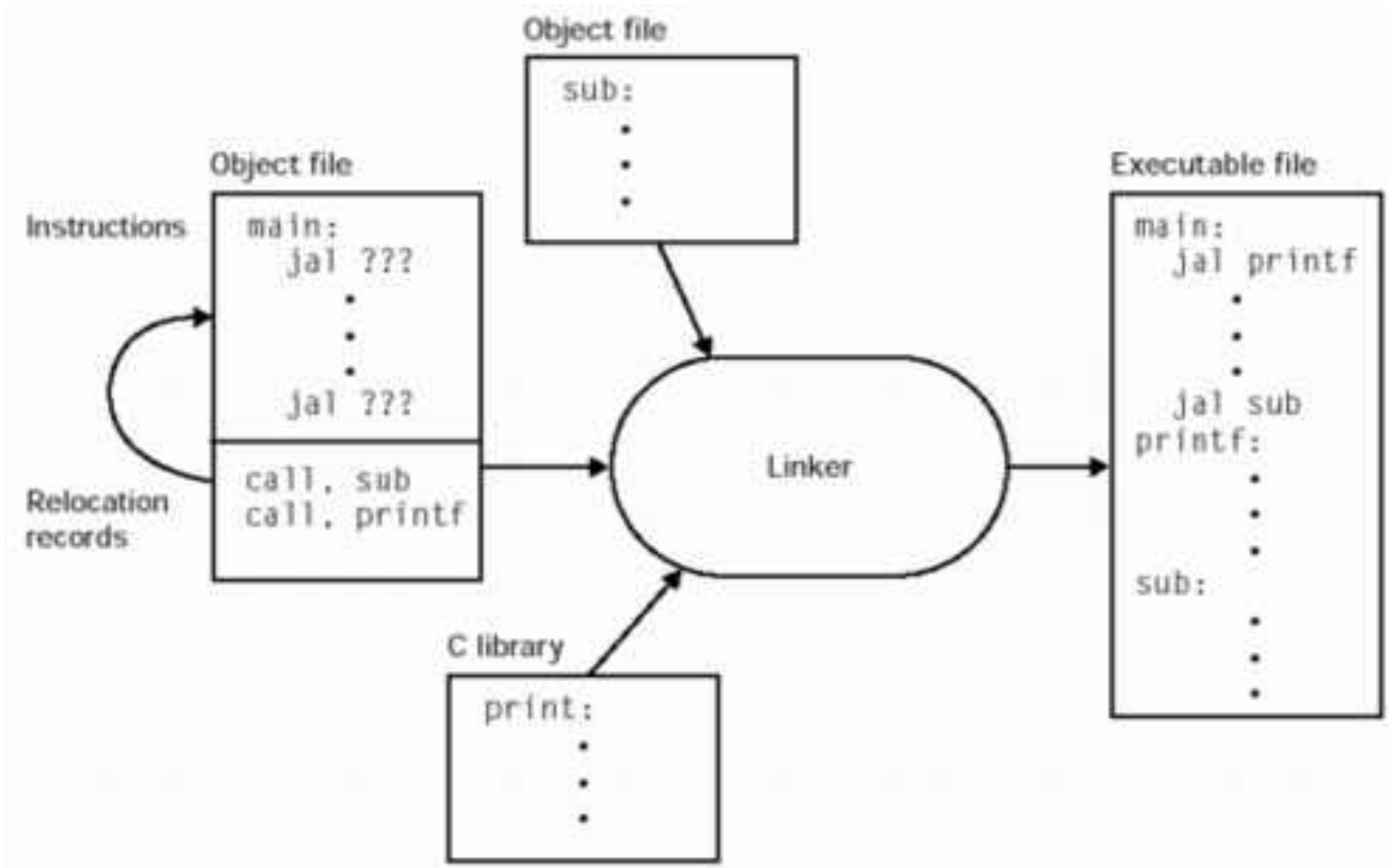
- **To produce assembly code:** `gcc -S test.c`
  - produces `test.s`
- **To produce object code:** `gcc -c test.c`
  - produces `test.o`
- **To produce executable code:** `gcc test.c`
  - produces `a.out`

The purpose of a linker

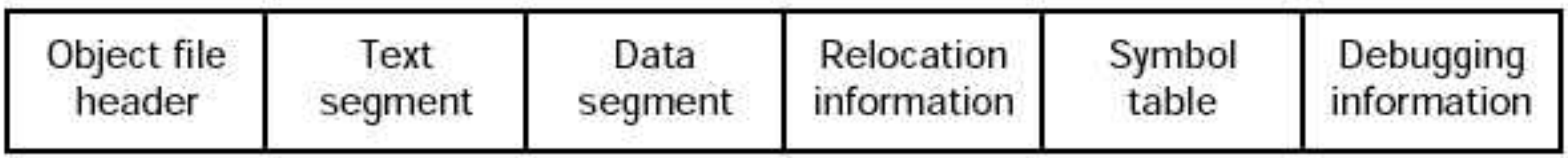


- The linker is a program that takes one or more object files and assembles them into a single executable program.
- The linker resolves references to undefined symbols by finding out which other object defines the symbol in question, and replaces placeholders with the symbol's address

What the linker does



Object File Formats



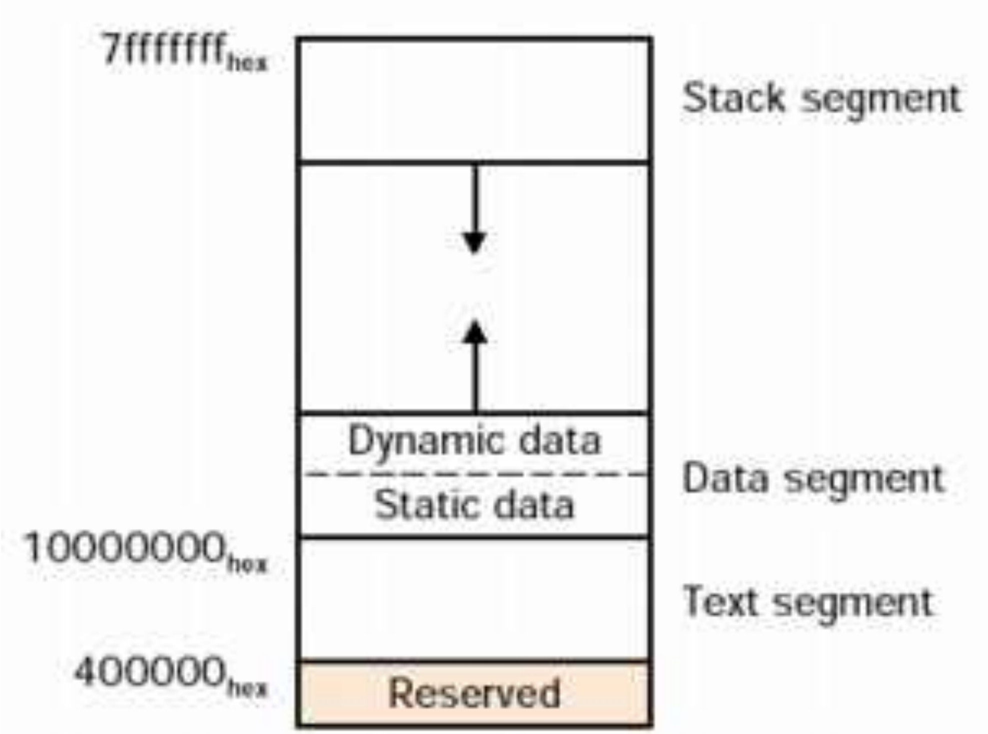
The three types of memory

```
int array1[100];
int an_int_with_a_value = 100;

void
a_function() {
   int array2[100];
   int *array3 = (int *) malloc(100 * sizeof(int));

   /* function contents … */
}
```

MIPS memory image



Loader

- **Before we can start executing a program, the O/S must load it:**

- **Loading involves 5 steps:**
  1. Allocates memory for the program's execution.
  2. Copies the text and data segments from the executable into memory.
  3. Copies program arguments (*e.g.*, command line arguments) onto the stack.
  4. Initializes registers: sets $sp to point to top of stack, clears the rest.
  5. Jumps to start routine, which: 1) copies main's arguments off of the stack, and 2) jumps to main.

Whither Assembly Language

Inline assembly Example

```
intadd(int a, int b) { /* return a + b */
  int ret_val;
  __asm("add %2, %0, %1", a, b, ret_val);
  return(ret_val);
}
```