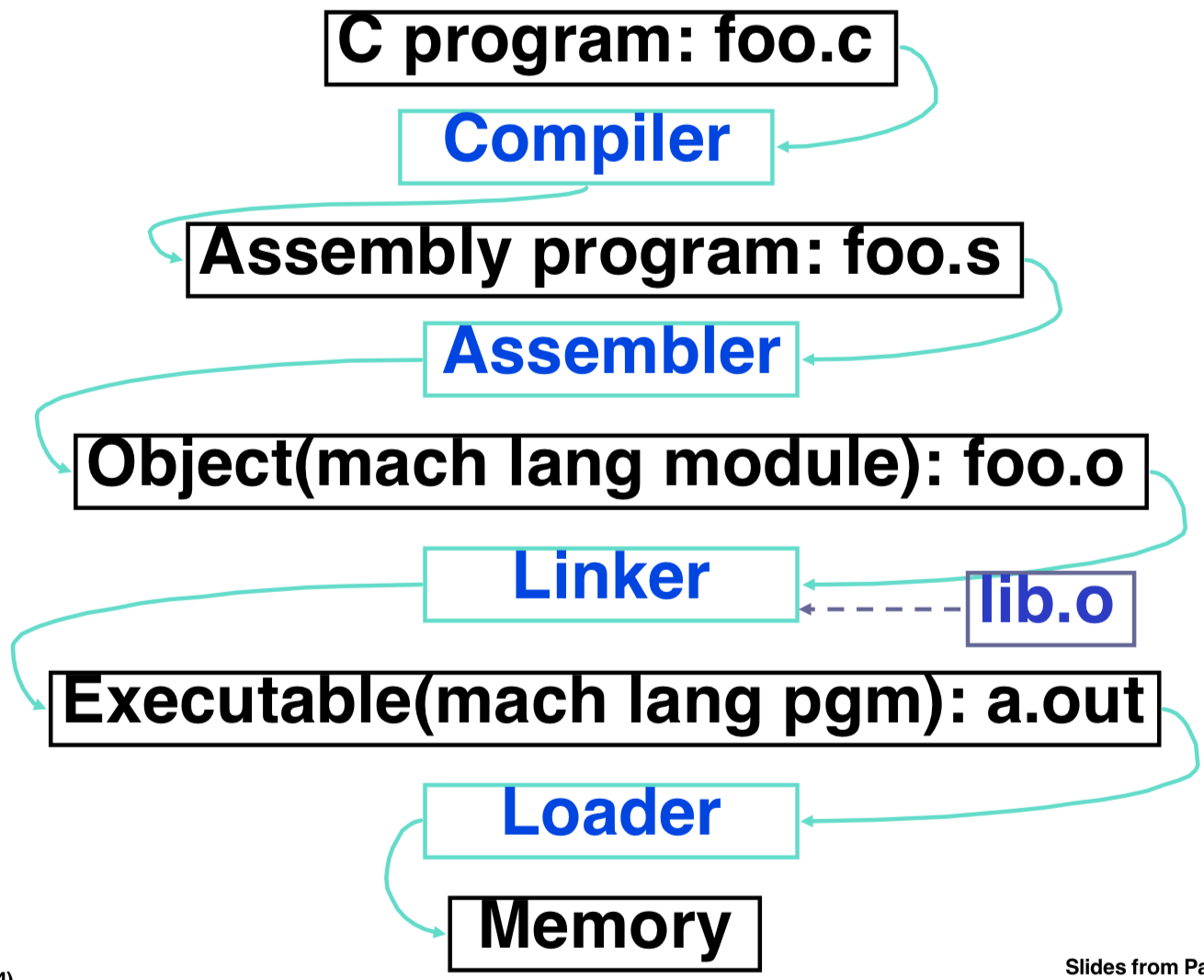
**Program**

Input: High-Level Language Code (e.g., C, Java)

Output: Assembly Language Code (e.g., MIPS)

Note: Output may contain *pseudoinstructions*



1. Reads and uses **directives**

2. Replaces pseudoinstructions

3. Produces machine language

4. Creates Object File

**Assembler**

**1. Assembler Directives**

Give directions to assembler, but does not produce machine instructions

.text: Subsequent items put in user text (instructions) segment

.data: Subsequent items put in user data segment

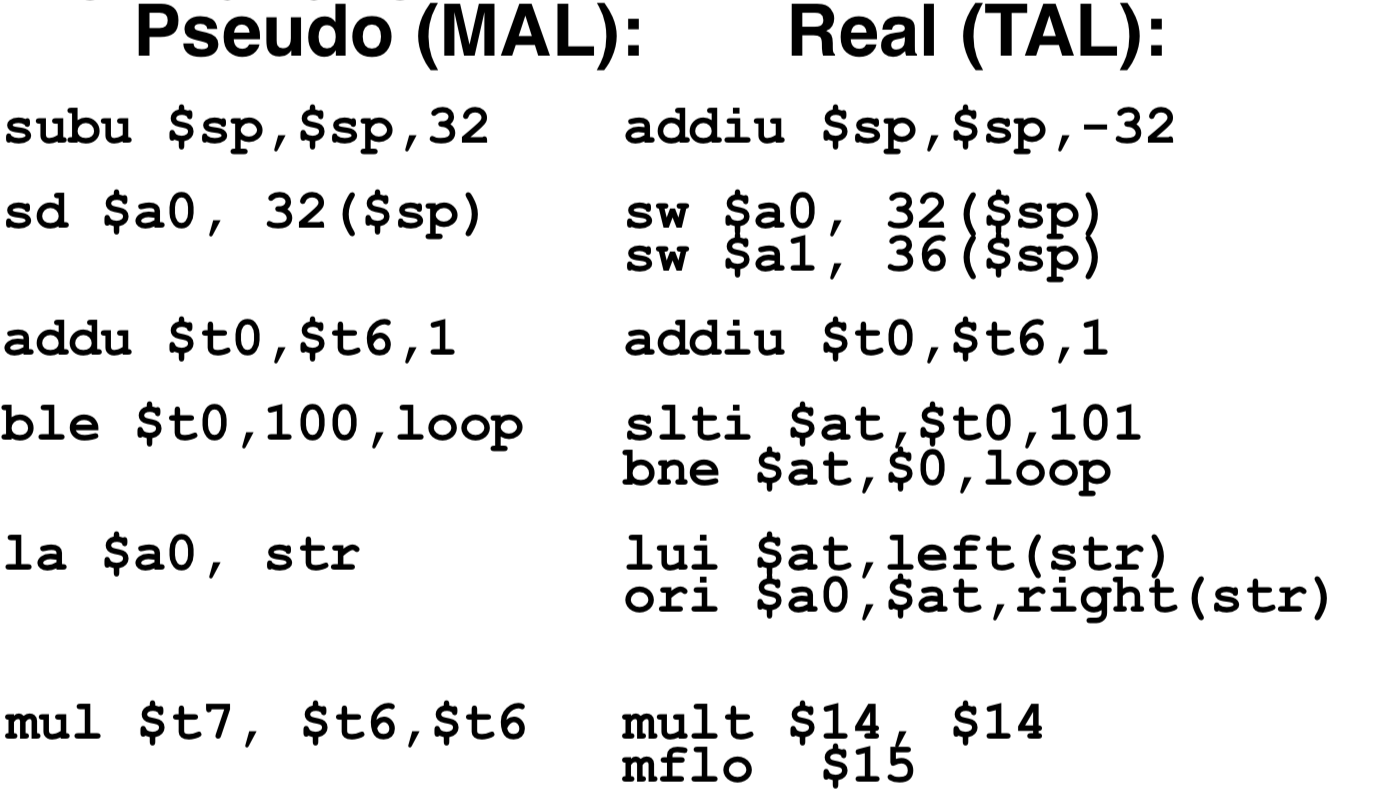
.globl sym: declares sym global and can be referenced from other files

.asciiz str: Store the string str in memory and null-terminate it

.word w1...wn: Store the n 32-bit quantities in successive memory words

**2. Pseudoinstruction Replacement**

Asm. treats convenient variations of machine language instructions as if real instructions



**3. Producing Machine Language**

Easily handled

* Simple instructions for Assembler

Arithmetic, Logical, Shifts, and so on.

All necessary info is within the instruction already.

* Branches

PC-Relative. Once pseudoinstructions are replaced by real ones, we know by how many instructions to branch.

Can’t be determined yet, must wait to see where this code will appear in final program.

* Jumps (j and jal)

Jumps require absolute addresses

* References to data

la gets broken up into lui and ori

These will require the full 32-bit address of the data.

Two tables are used to help assembly and later resolution of addresses

**1st Table: Symbol Table**

*List of “items” in this file that may be used by this and other files.*

* Labels: function calling
* Data: anything in the .data section; variables which may be accessed across files

First Pass: record label-address pairs

Second Pass: produce machine code

Result: can jump to a label later in code without first declaring it

**2nd Table: Relocation Table**

*Line numbers of “items” for this file which need the address filled in (or fixed up) later.*

* Any label jumped to: j or jal
* Internal (i.e., label inside this file)
* External (including lib files)
* Any absolute address of piece of data

Such as used by the la pseudo-instruction: la $destination,label

**Object File Format**

object file header: size and position of the other pieces of the object file

text segment: the machine code

data segment: binary representation of the data in the source file

relocation table: identifies lines of code that need to be “handled”

symbol table: list of this file’s labels and data that can be referenced

debugging information

**Link Editor/Linker**

*Combines several object (.o) files into a single executable (“linking”)*

Enables Separate Compilation of files

Changes to one file do not require recompilation of whole program

Code in file called a **module**

Step 1: Take text segment from each .o file and put them together.

Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.

Step 3: Resolve References

Go through Relocation Table and handle each entry using the Symbol Table

If not found, search library files (for example, for printf)

Once absolute address is determined, fill in the machine code appropriately

Four Types of Addresses

* PC-Relative Addressing (beq, bne): never fix up (never “relocate”)
* Absolute Address (j, jal): always relocate
* External Reference (usually jal): always relocate
* Symbolic Data Reference (often lui and ori, for la): always relocate

Linker assumes **first word of first text segment is at address 0x00000000.**

Linker knows:

length of each text and data segment

ordering of text and data segments

Linker calculates:

absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

Output of linker: executable file containing text and data (plus header)

May not have library object files resolved if dynamically loaded

**Loader**

*Executable files are stored on disk. When one is to be run, loader’s job is to load it into memory and start it running.*

In reality, loader is the operating system (OS) [loading is one of the OS tasks]

* Reads executable file’s header to determine size of text and data segments
* Creates new address space for program large enough to hold text and data segments, along with a stack segment
* Copies instructions and data from executable file into the new address space
* Copies arguments passed to the program onto the stack
* Initializes machine registers

Most registers cleared, but stack pointer assigned address of 1st free stack location

* Jumps to start-up routine that copies program’s arguments from stack to registers and sets the PC

If main routine returns, start-up routine terminates program with the exit system call

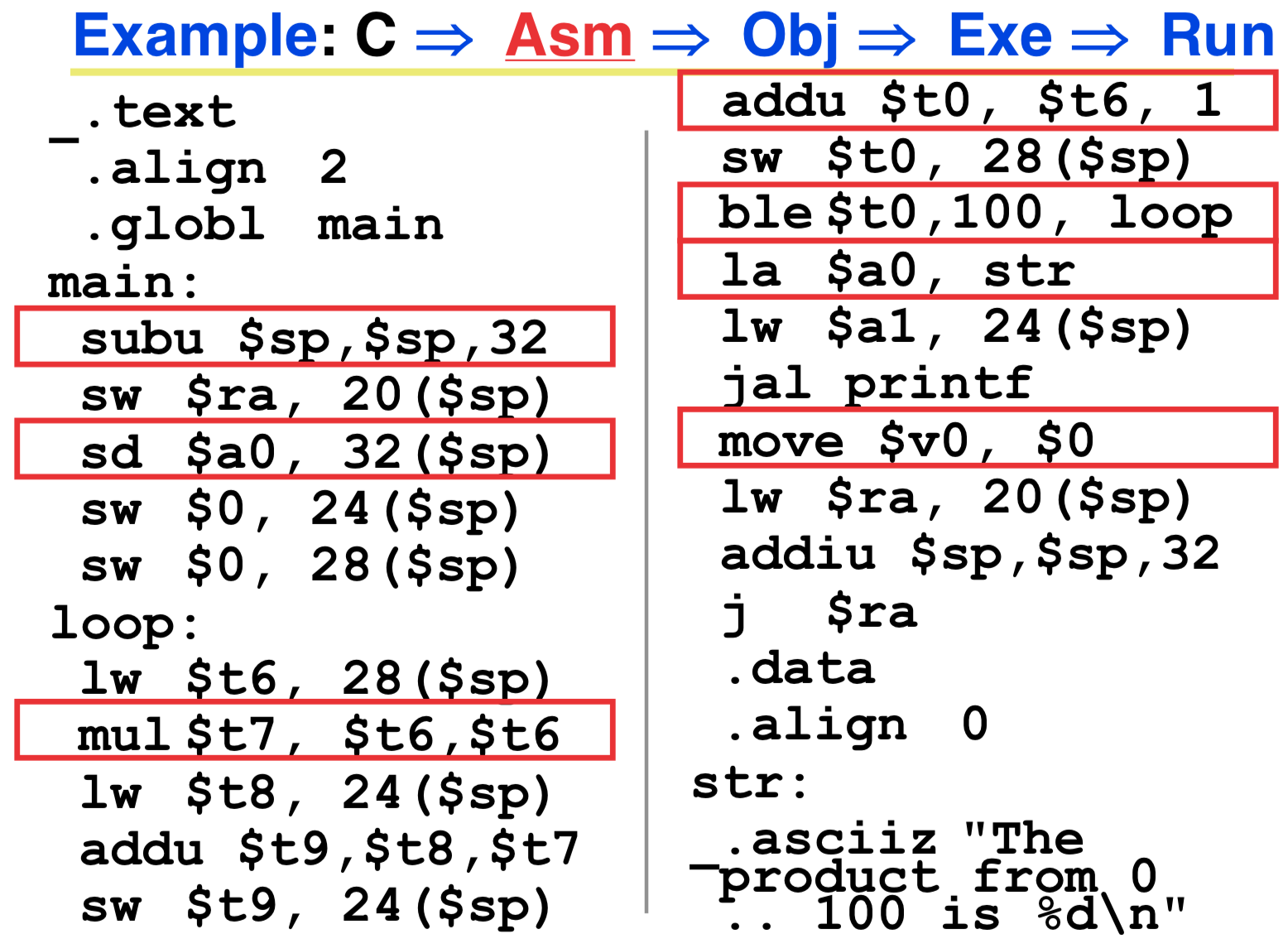
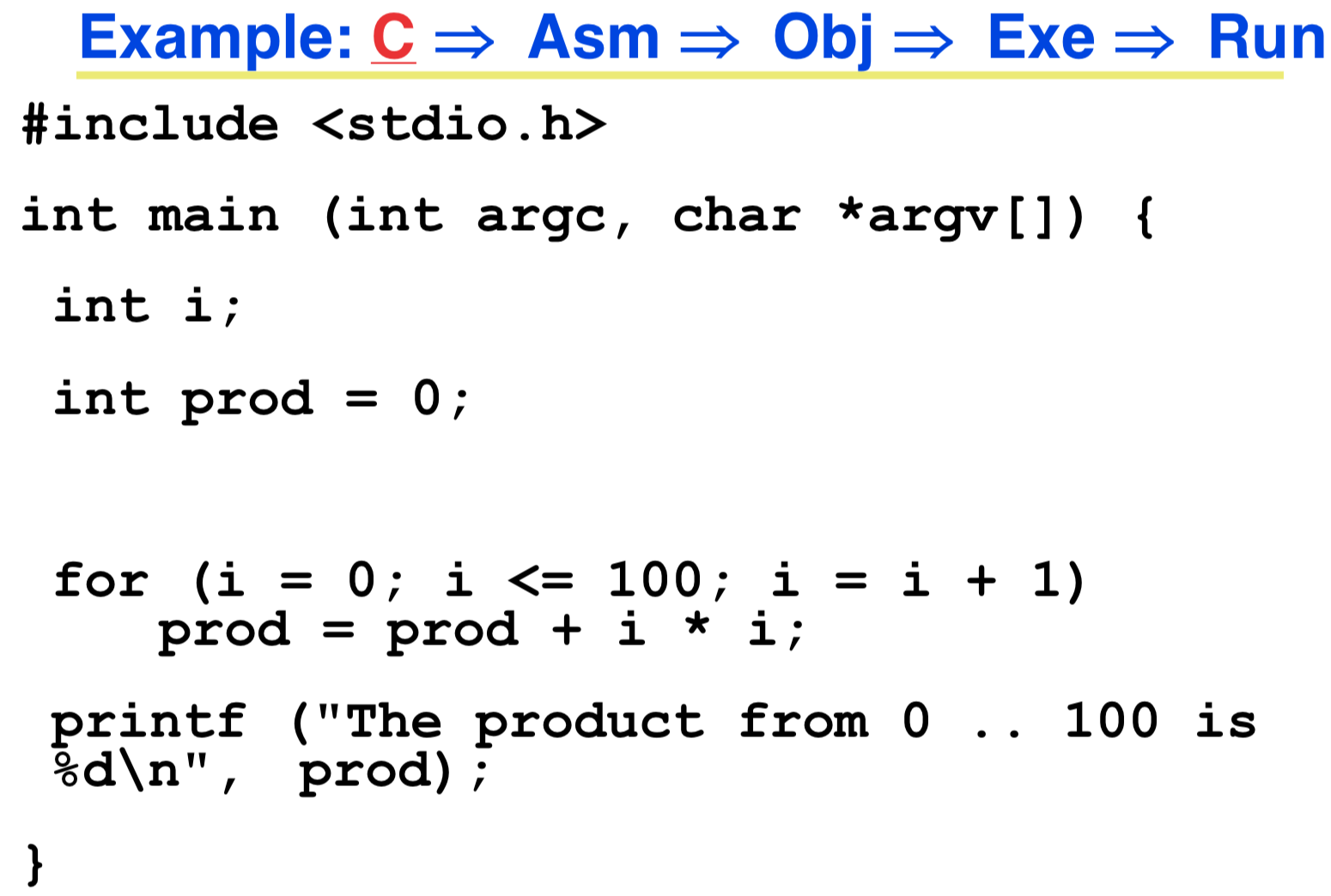
**Dynamic Linking**

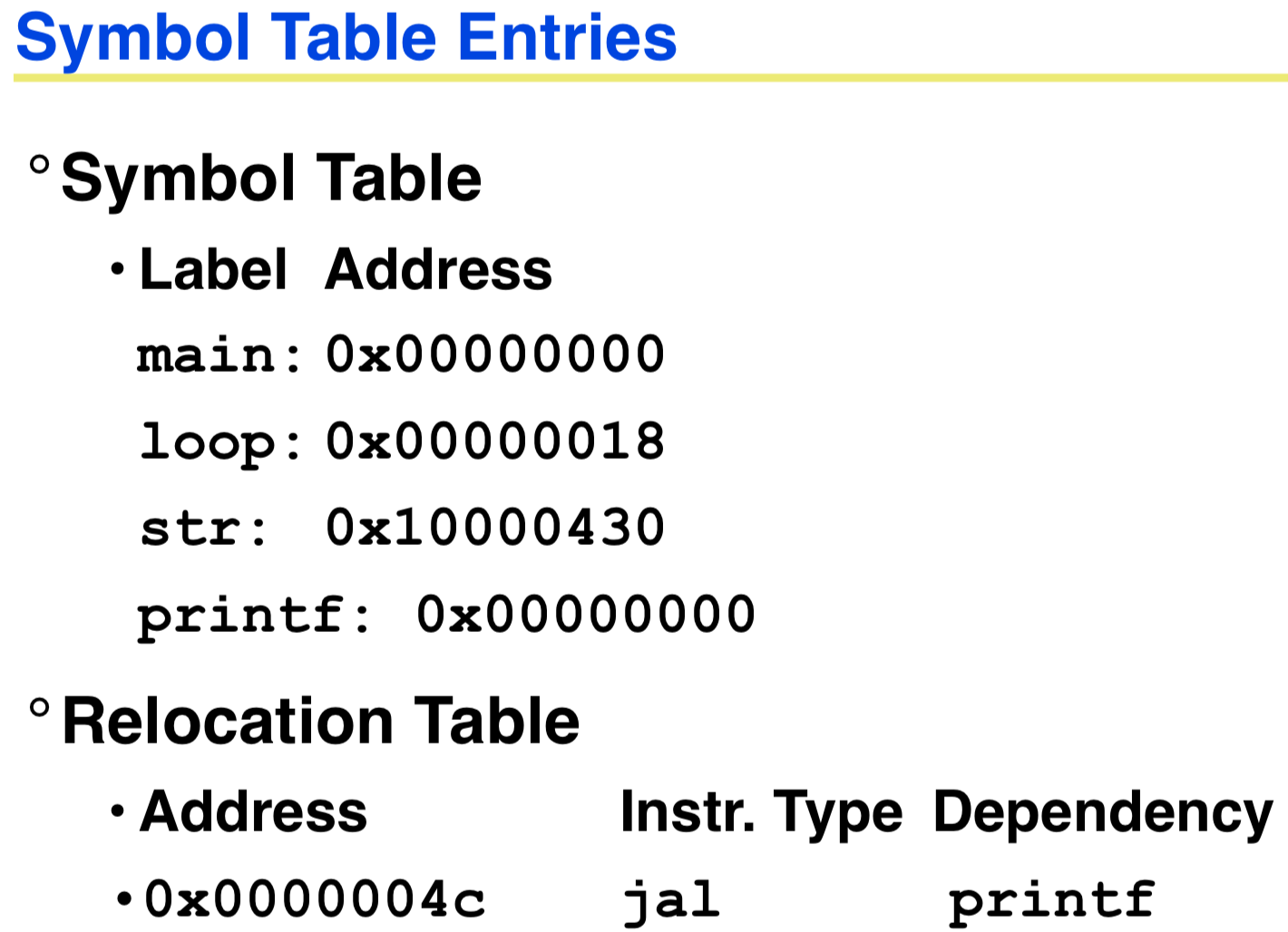
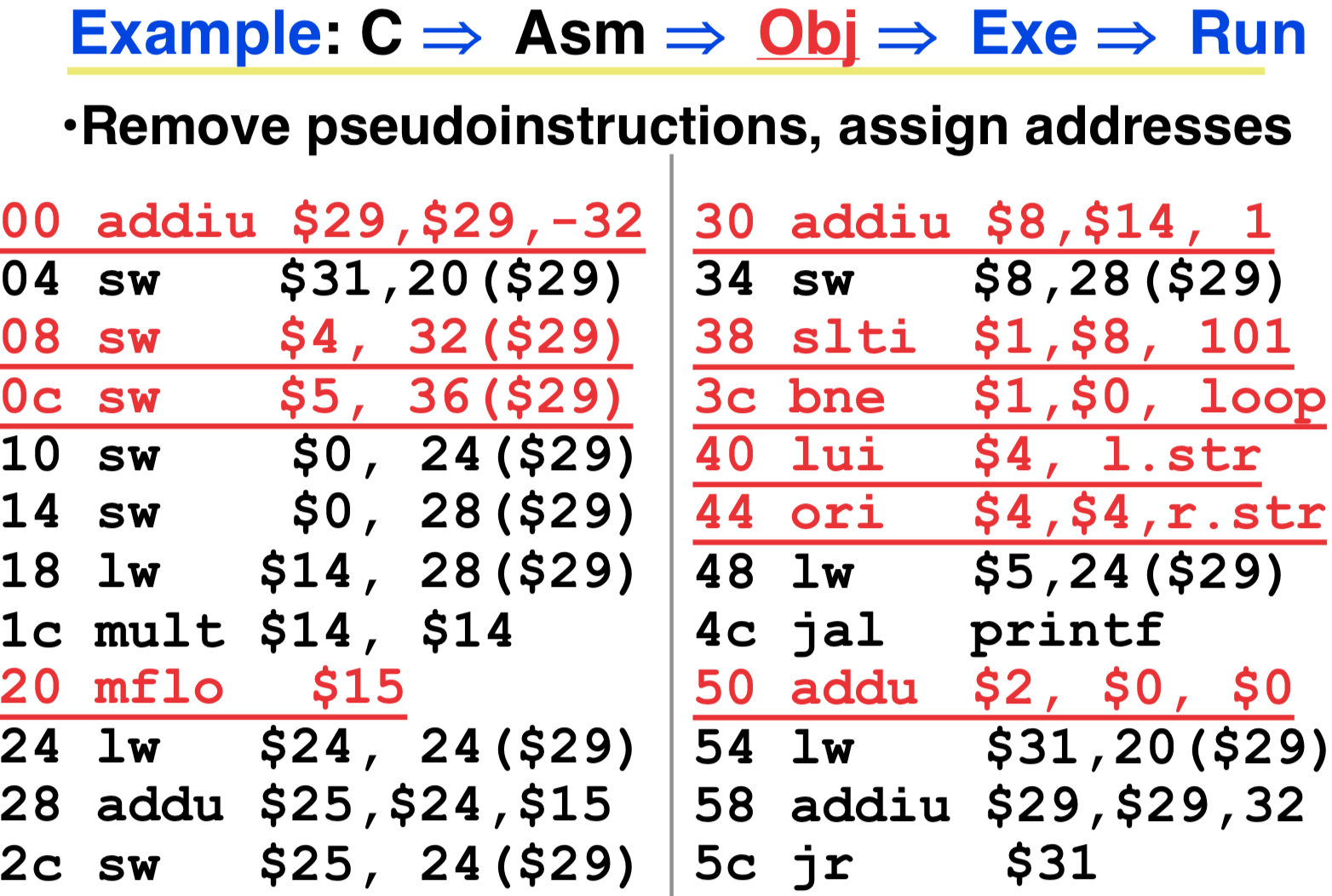
Some operating systems allow “dynamic linking”

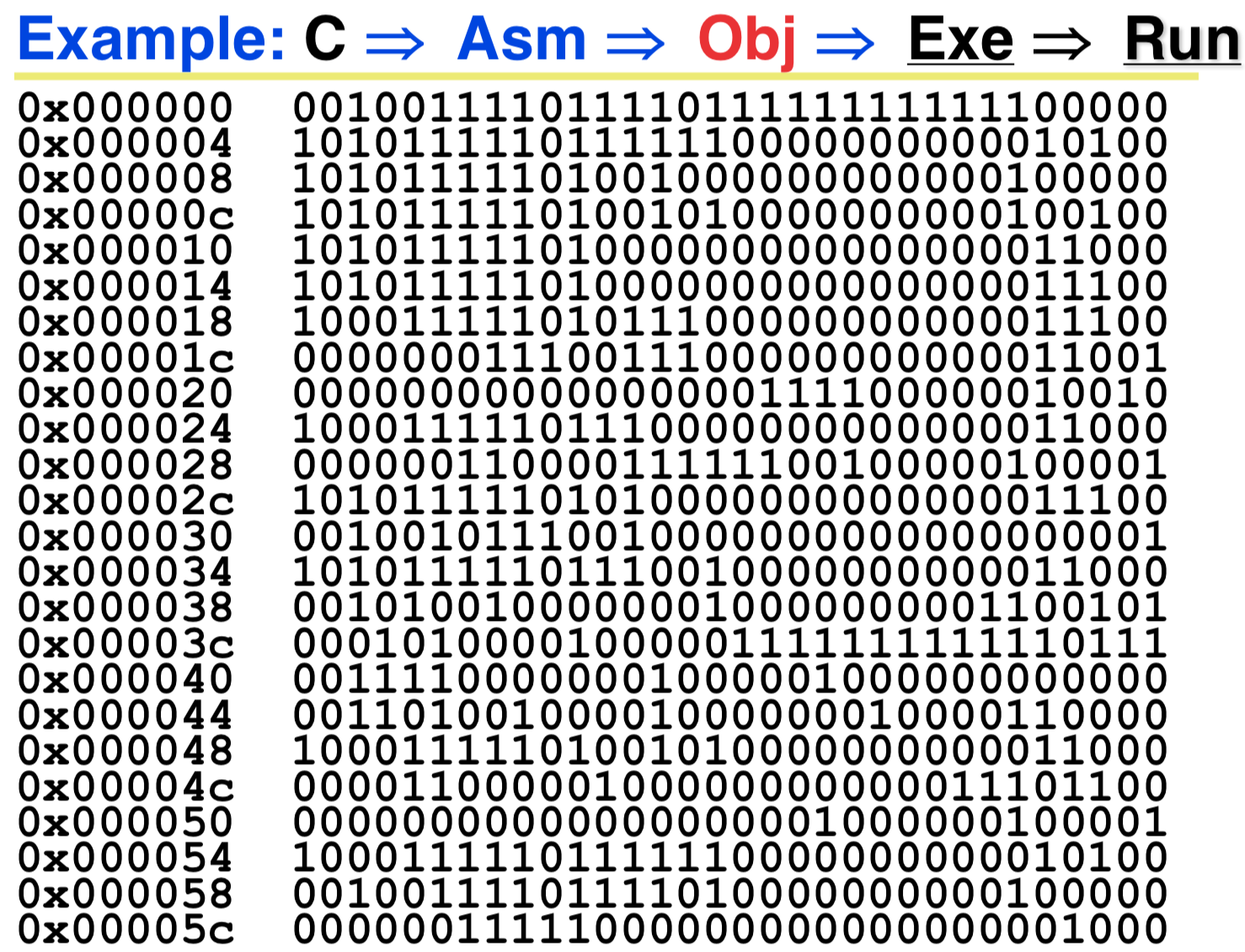
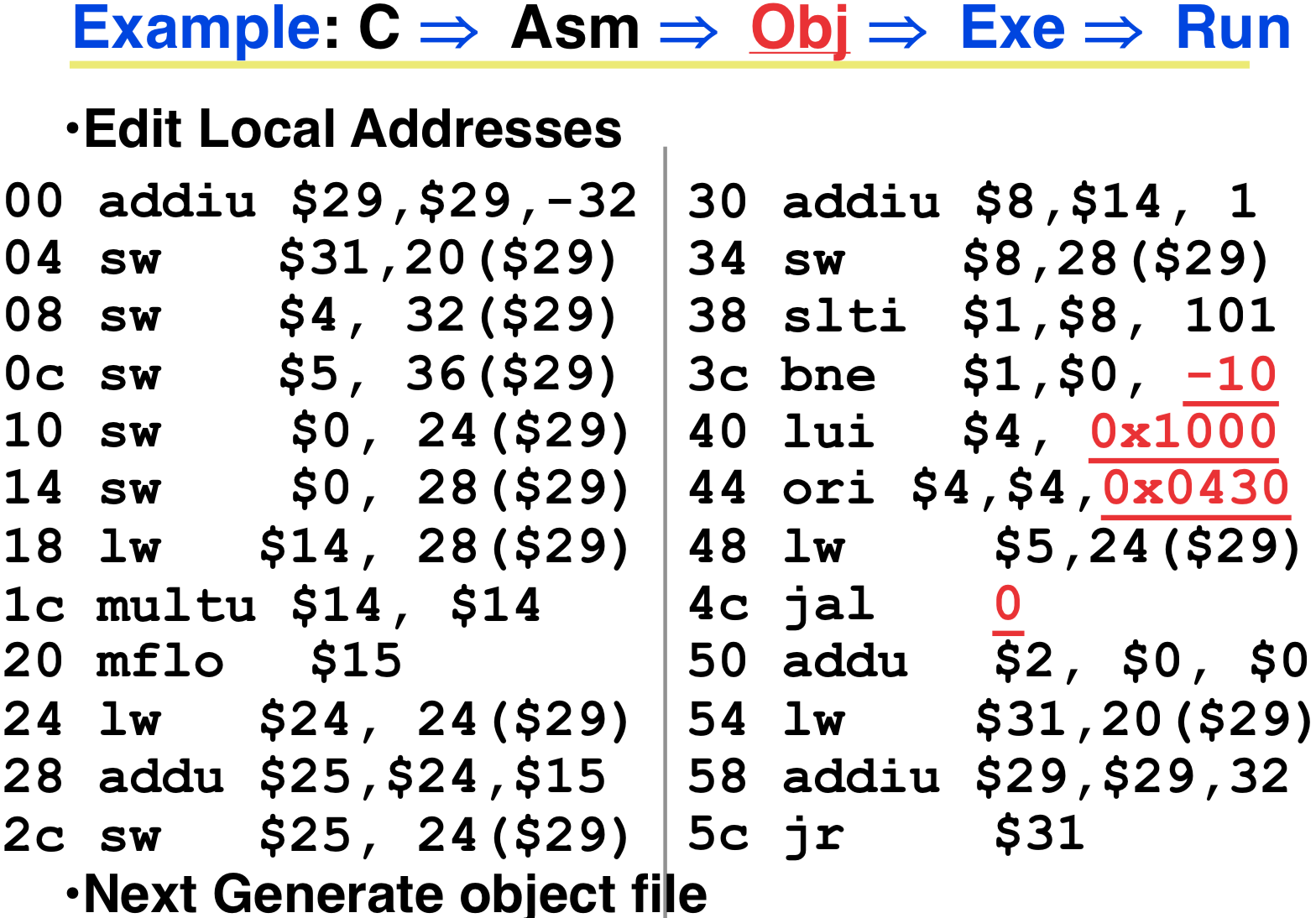
**Both the loader and the linker are part of the operating system** - so modules can be linked and loaded at runtime

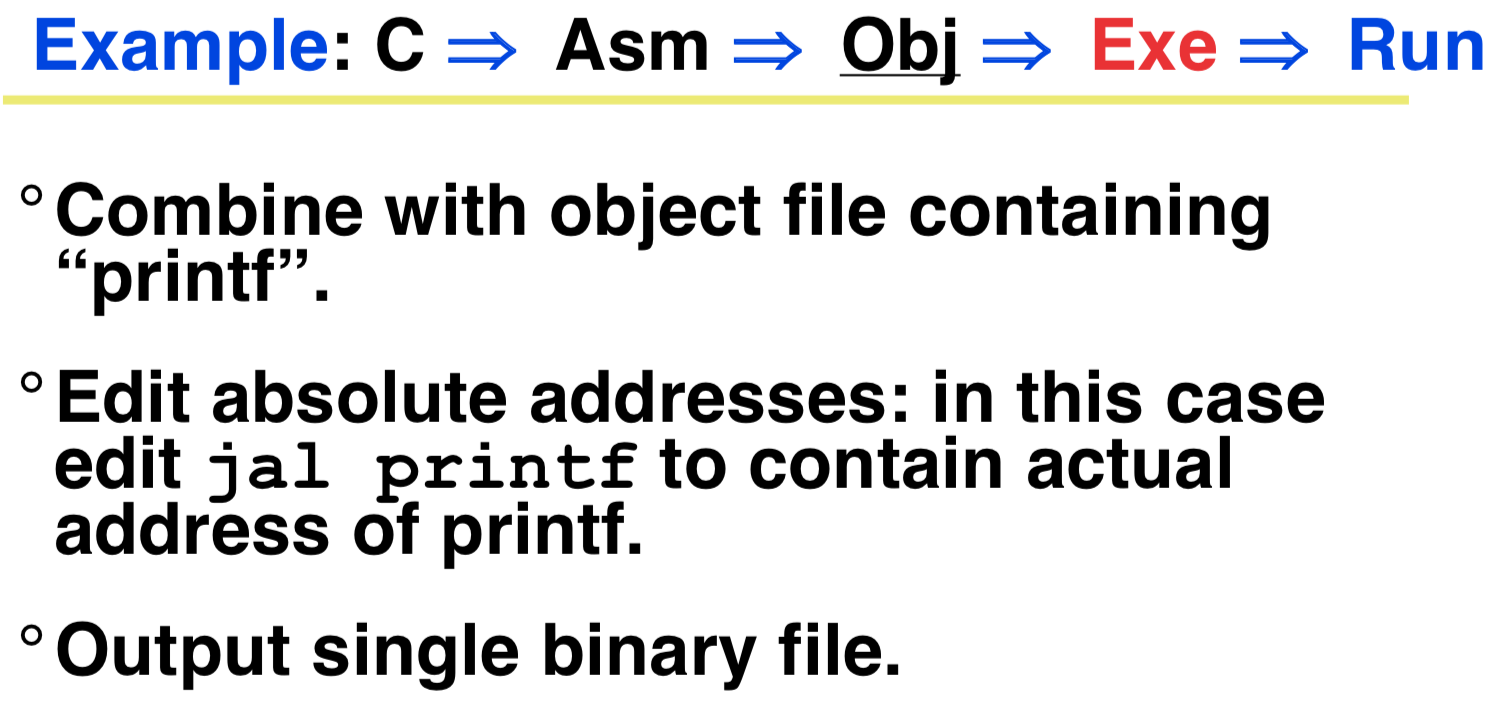
**If a module is needed and already loaded, it need not be loaded again**

[ Called DLLs ]

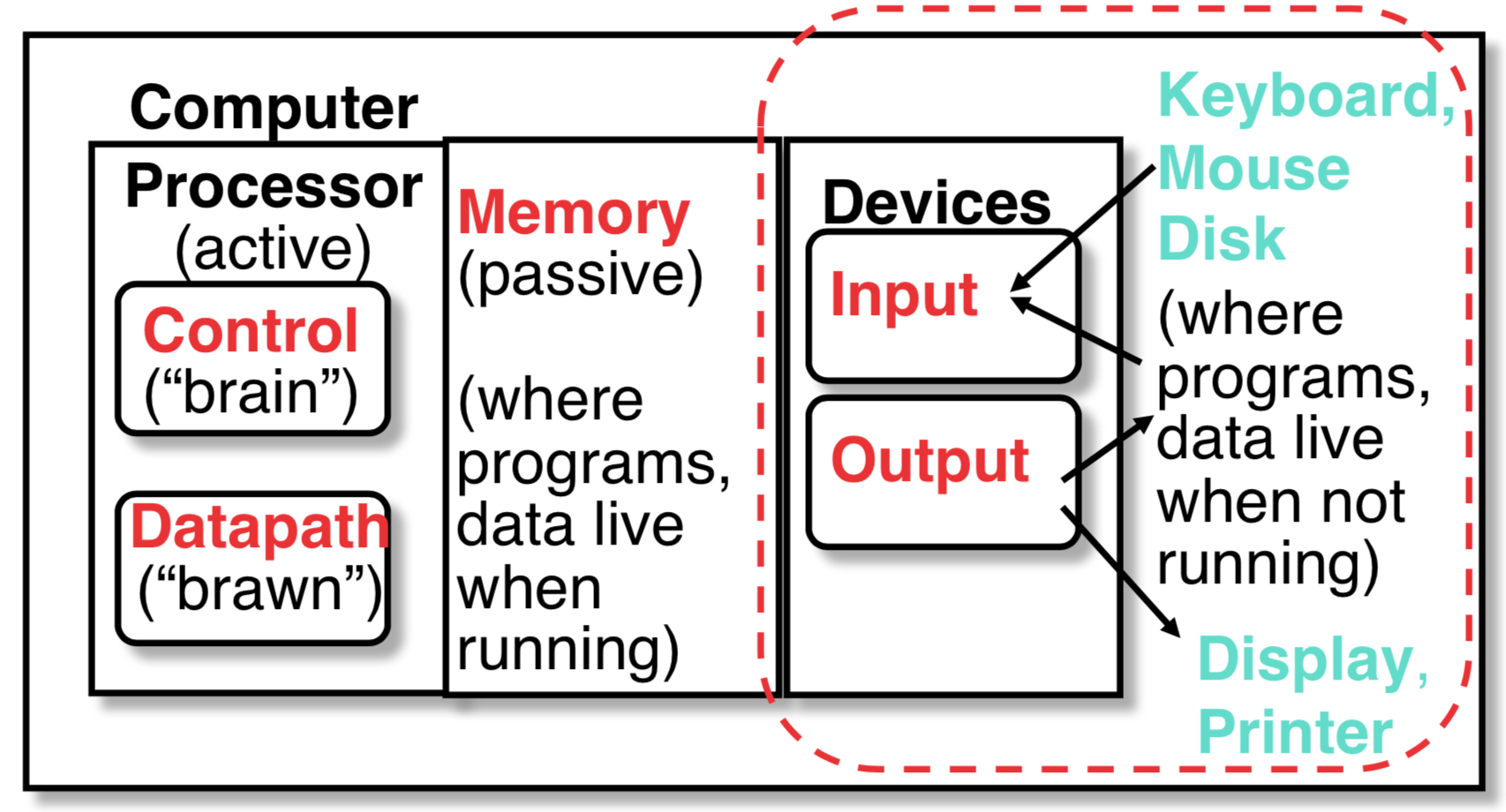






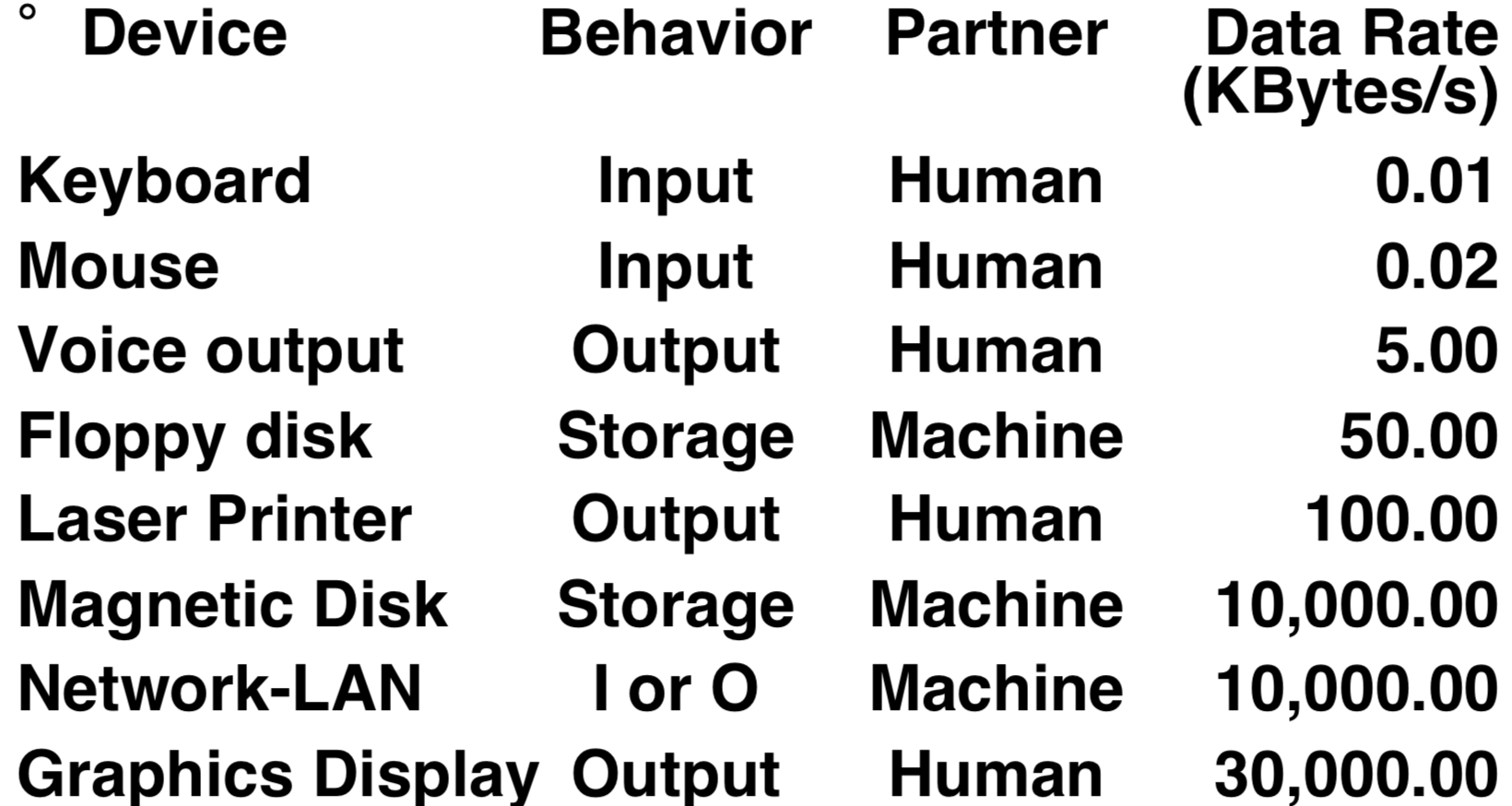


**Input/Output: Polling and Interrupts**

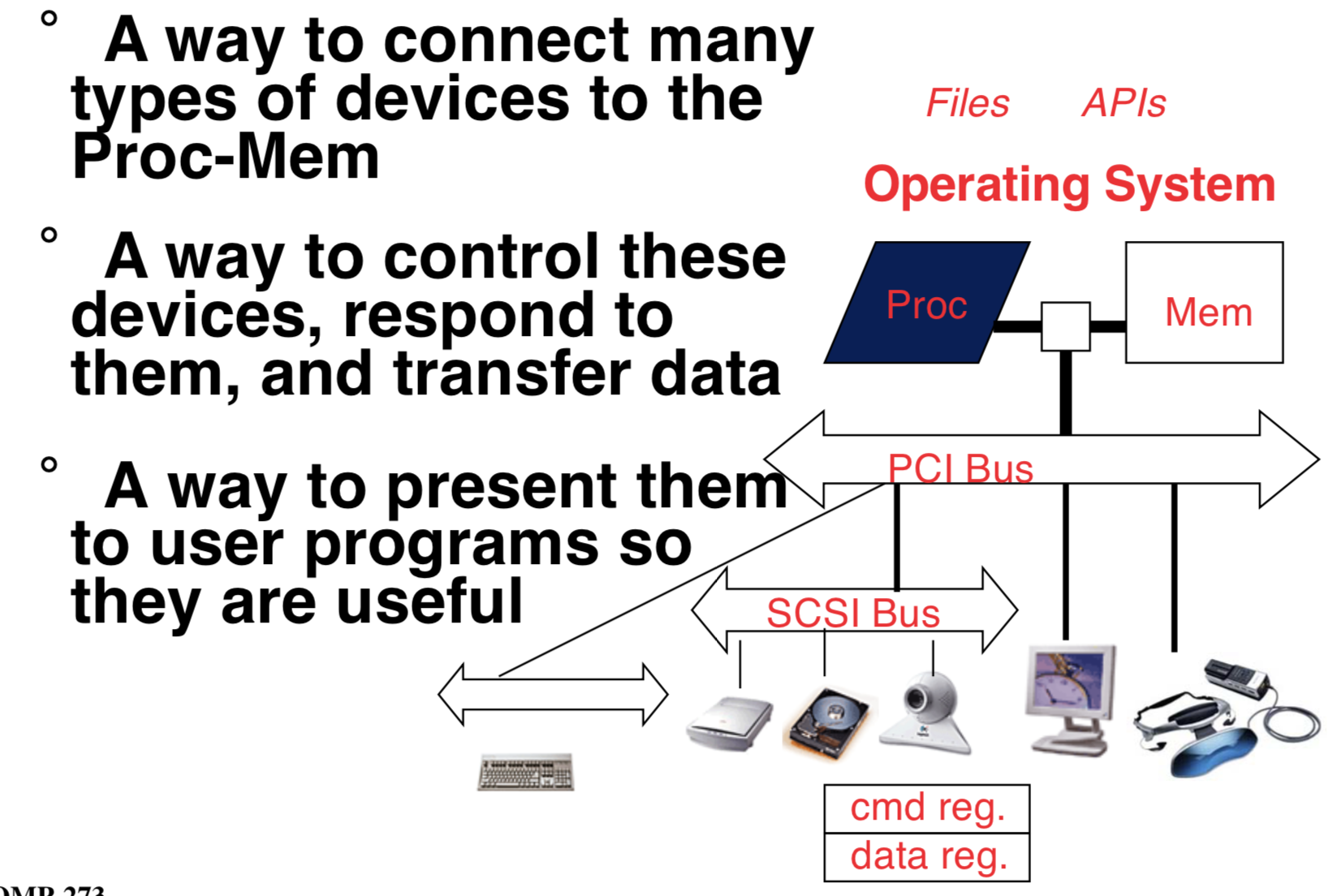


*I/O is how humans interact with computers*

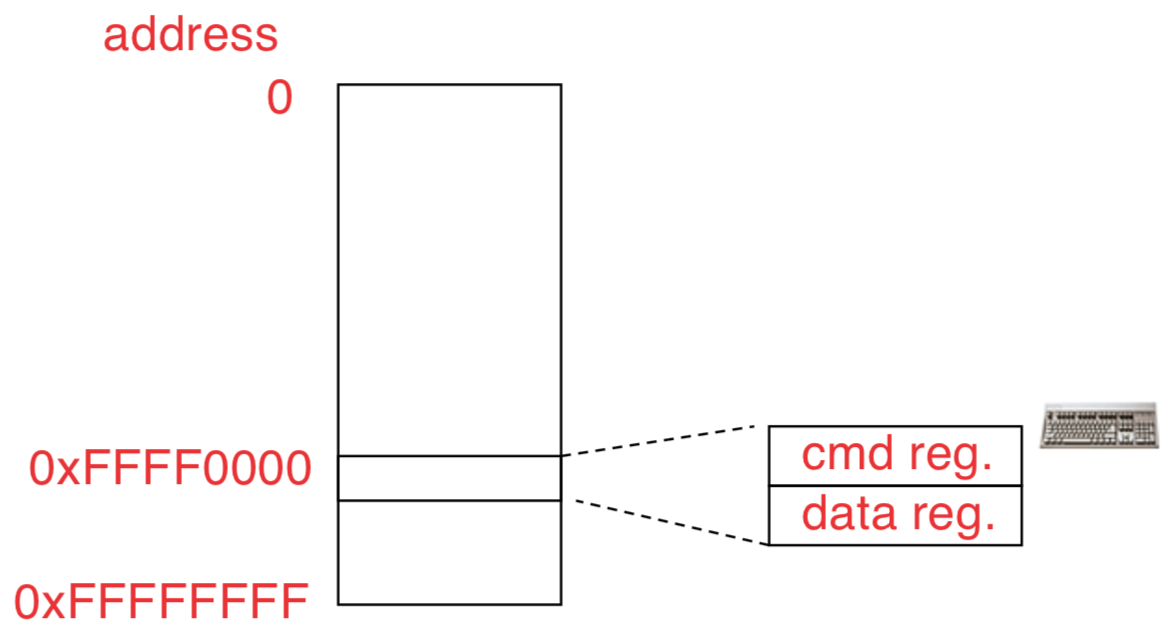
I/O Speed: bytes transferred per second



What do we need to make I/O work



**Instruction Set Architecture for I/O**

* What must the processor do for I/O?
* Input: reads a sequence of bytes
* Output: writes a sequence of bytes
* Some processors have special input and output instructions
* Alternative model (used by MIPS):
* Use loads for input, stores for output
* Called **“Memory Mapped Input/Output”**
* A portion of the address space dedicated to communication paths to Input or Output devices (no memory there)

**Memory Mapped I/O (Polling)**

**Processor-I/O Speed Mismatch**

1GHz microprocessor can execute 1 billion load or store instructions per second, or 4,000,000 KB/s data rate. But I/O devices data rates range from 0.01 KB/s to 30,000 KB/s.

Input: device may not be ready to send data as fast as the processor loads it

Also, might be waiting for human to act

Output: device may not be ready to accept data as fast as processor stores it

**🡪 Processor Checks Status before Acting**

Path to device generally has 2 registers:

• ***Control Register***, says it’s OK to read/write (I/O ready)

• ***Data Register***, contains data

Processor reads from *Control Register* in loop, waiting for device to set Ready bit in Control reg (0🡪1) to say it’s OK. Processor then loads from (input) or writes to (output) data register

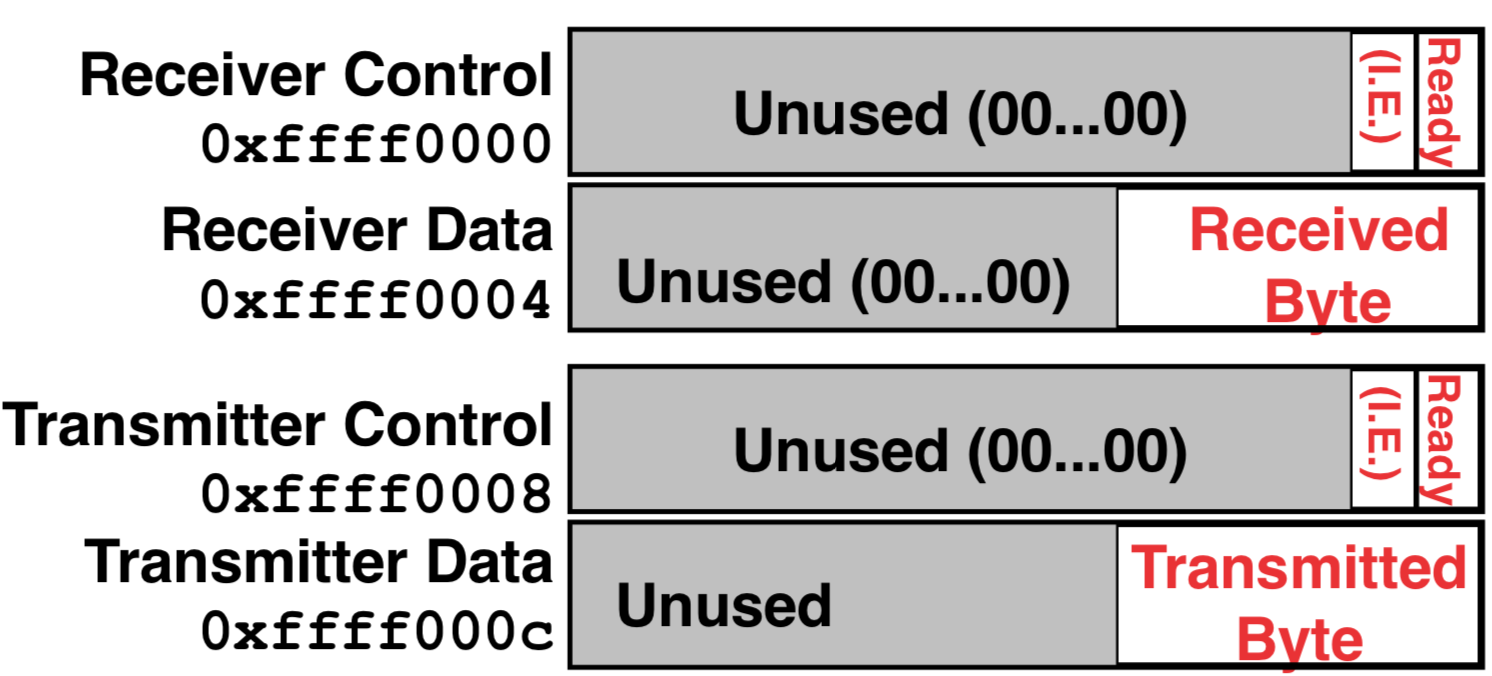
Load from or Store into Data Register resets Ready bit (1🡪0) of Control Register

MARS I/O Simulation

**MARS simulates 1 I/O device: memory- mapped terminal (keyboard + display)**

• Read from keyboard **(receiver**); 2 device regs

• Writes to terminal (**transmitter**); 2 device regs

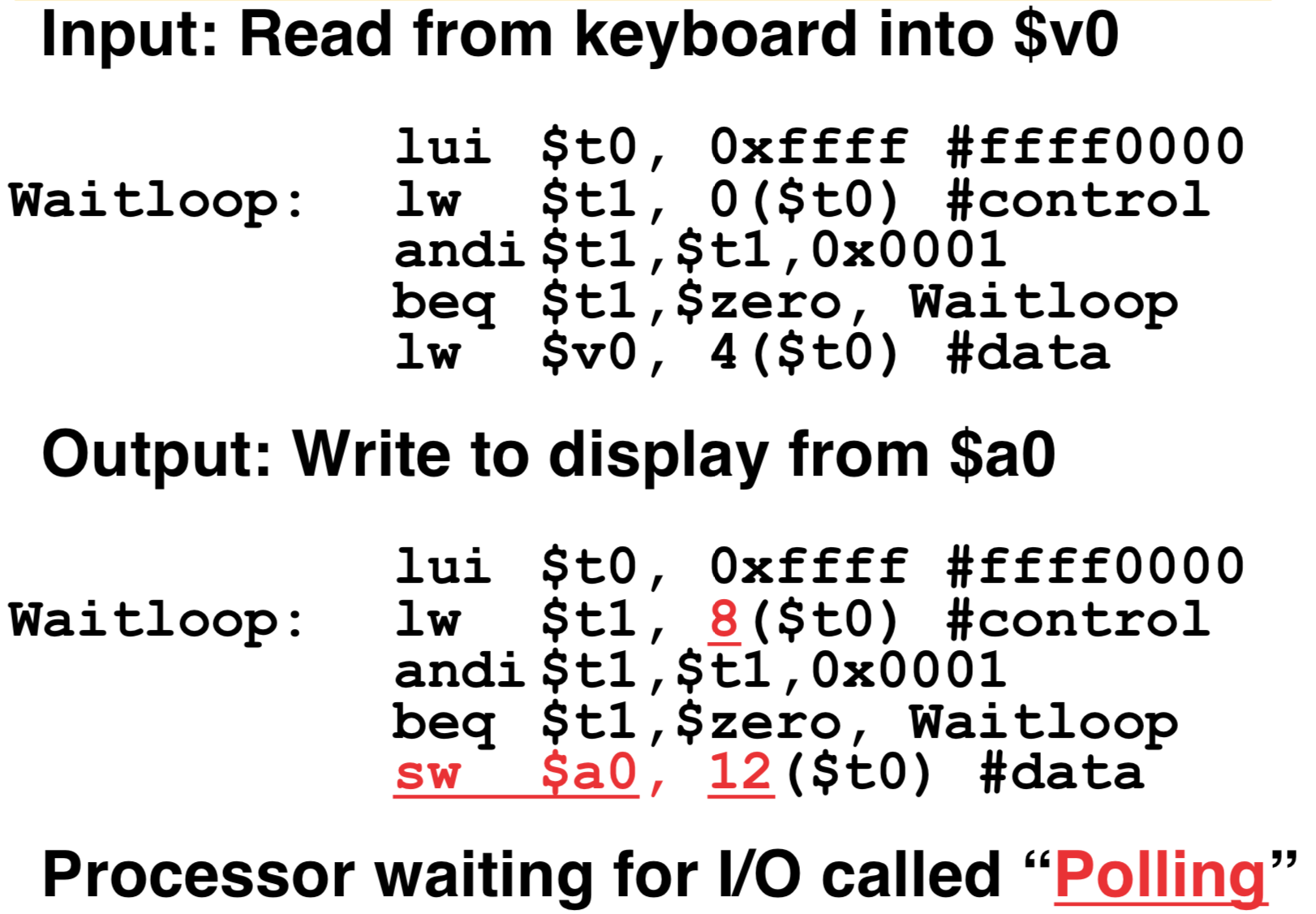


Control register rightmost bit (0): Ready

* Receiver: Ready==1 means character in Data Register not yet been read; 1 🡪 0 when data is read from Data Reg
* Transmitter: Ready==1 means transmitter is ready to accept a new character; 0 🡪1 Transmitter still busy writing last char

Data register rightmost byte has data

* Receiver: last char from keyboard; rest = 0
* Transmitter: when write rightmost byte, writes char to display



Cost of Polling

The following example used dated estimates, but it conveys the general idea. Assume for a processor with a 1GHz clock it takes 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning). Determine % of processor time used for polling.

* Mouse: polled 30 times/sec so as not to miss user movement

Mouse Polling, Clocks/sec = 30 \* 400 = 12000 clocks/sec

% Processor for polling: 12\*103/1\*109 = 0.0012%

🡪 Polling mouse little impact on processor

* Floppy disk: transfers data in 2-Byte units and has a data rate of 50 KB/second. No data transfer can be missed.

Frequency of Polling Floppy = 50 KB/s /2B = 25K polls/sec

Floppy Polling, Clocks/sec = 25K \* 400 = 10,000,000 clocks/sec

% Processor for polling: 10\*106/1\*109 = 1%

🡪 OK if not too many I/O devices

* Hard disk: transfers data in 16-Byte chunks and can transfer at 16 MB/second. Again, no transfer can be missed.

Frequency of Polling Disk = 16 MB/s /16B = 1M polls/sec

Disk Polling, Clocks/sec= 1M \* 400 = 400,000,000 clocks/sec

% Processor for polling: 400\*106/1\*109 = 40%

🡪 Unacceptable

Wasteful to have processor spend most of its time “spin-waiting” for I/O to be ready

Solution: use *exception mechanism* to help I/O. *Interrupt* program when I/O ready, return when done with data transfer

**I/O interrupt**

An I/O interrupt is like overflow exceptions except:

* An I/O interrupt is “**asynchronous**” with respect to instruction execution
* I/O interrupt is not associated with any instruction, but it can happen in the middle of any given instruction
* I/O interrupt does not prevent any instruction from completion
* More information needs to be conveyed

[Definitions for Clarification]

* Exception: signal marking that something “out of the ordinary” has happened and needs to be handled
* Interrupt: asynchronous exception
* Trap: synchronous exception

[ Instruction Set Support for I/O Interrupt ]

* Save the PC for return

But where?

* Where to go when interrupt occurs?

MIPS defines location: **0x80000080**

* Determine cause of interrupt?

MIPS has ***Cause Register***, 4-bit field (bits 5 to 2) gives cause of exception

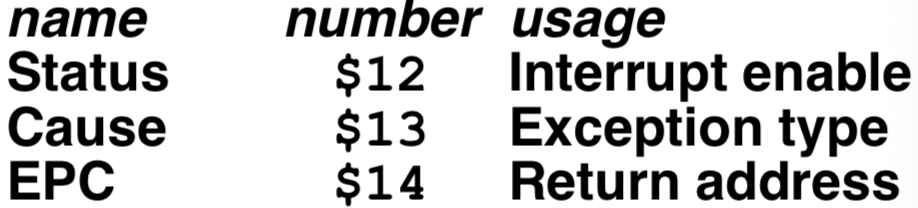
* Portion of MIPS architecture for interrupts called ***“coprocessor 0”***

Coprocessor 0 Instructions

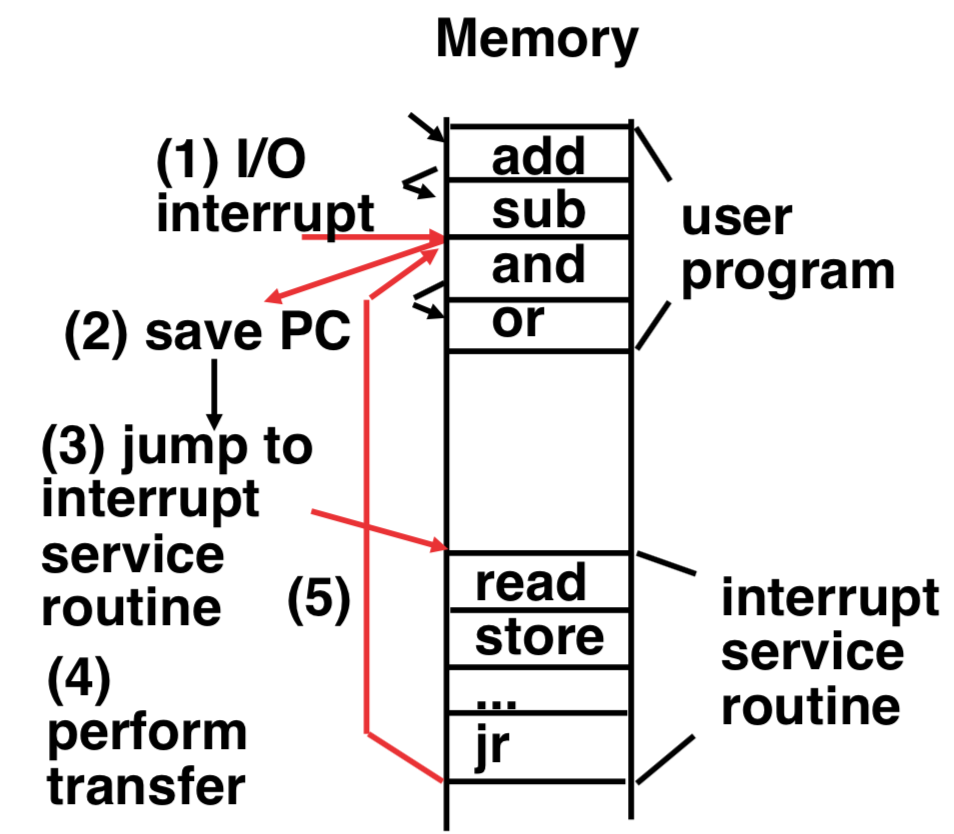
**Data transfer: lwc0, swc0**

**Move: mfc0, mtc0**

Coprocessor 0 Registers:



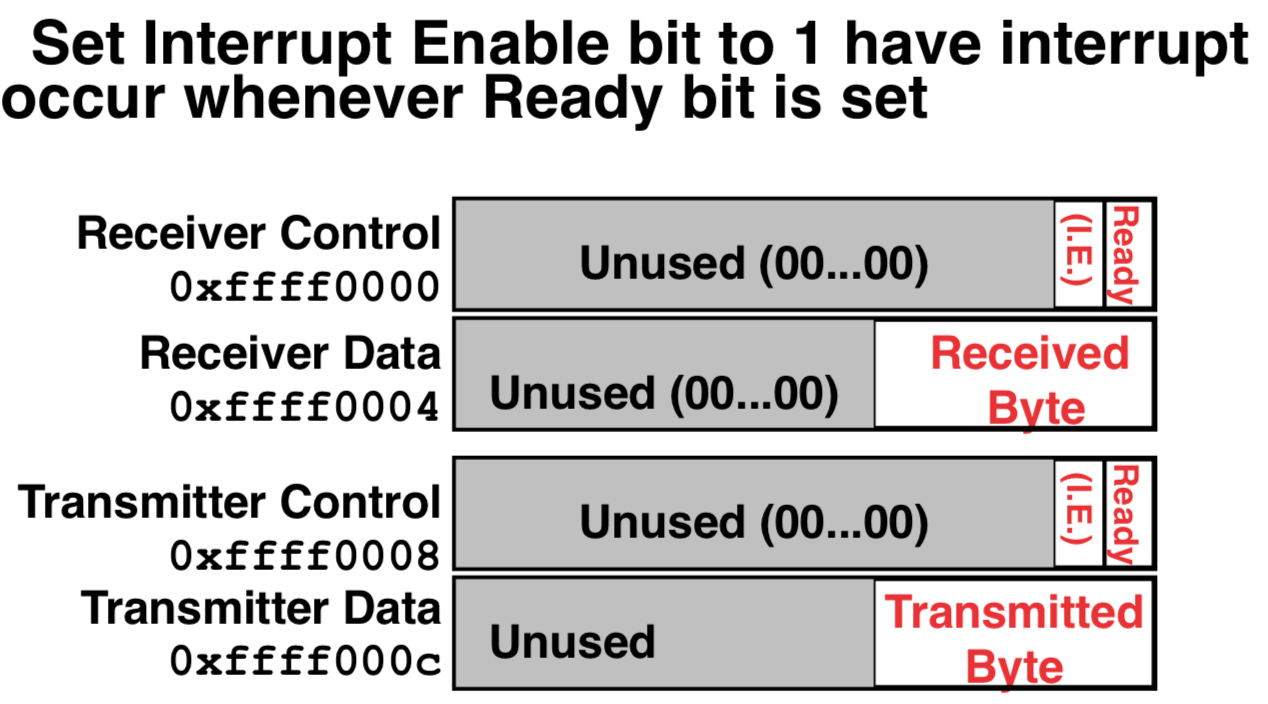
Interrupt Driven Data Transfer



SPIM I/O Simulation: Interrupt Driven I/O

*I.E. stands for Interrupt Enable.*

Set Interrupt Enable bit to 1 have interrupt occur whenever Ready bit is set



Find the % of processor consumed if the hard disk is only active 5% of the time. Assuming 500 clock cycle overhead for each transfer, including interrupt:

• Disk Interrupts/sec = 16 MB/s /16B = 1M interrupts/sec

• Disk Interrupts, Clocks/sec = 1M \* 500 = 500,000,000 clocks/sec

• % Processor for during transfer: 500\*106/1\*109= 50%

• Disk active 5% 🡪 5% \* 50% = 2.5% busy