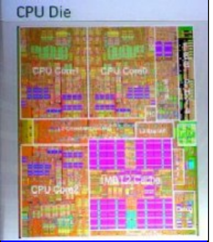
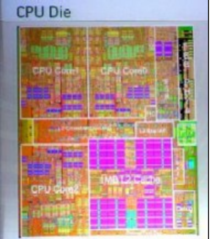


COMP 273
Introduction to Computer Systems

COMP 273

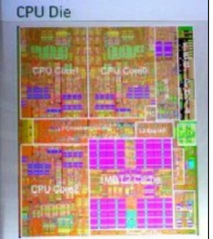
The Assembly Process

Prof. Joseph Vybihal



Announcements

- Project:
  - How is team formation?
- MIPS programming tutorials, next week.
- Last half of course
  - Programming in assembler
  - Special CPU hardware and OS issues
- Assignment #3 end of next week

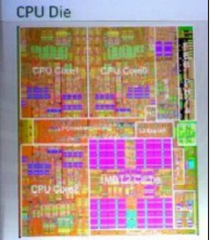


COMP 273
Introduction to Computer Systems

Readings & Activities

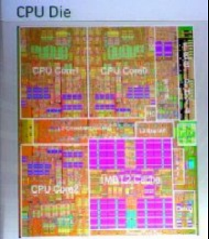
- Download the MIPS interpreter and try to compile and execute either a sample program or the example from this class.

- Web Resources:
  - Google: MARS or SPIM
  - http://pages.cs.wisc.edu/~larus/spim.html
  - http://www.cs.uic.edu/~i366/notes/SPIM%20Examples.html



Part 1

What can an assembly program do?

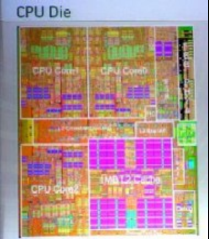


Scope of assembler instructions

All peripherals and system components are accessible by assembly instructions

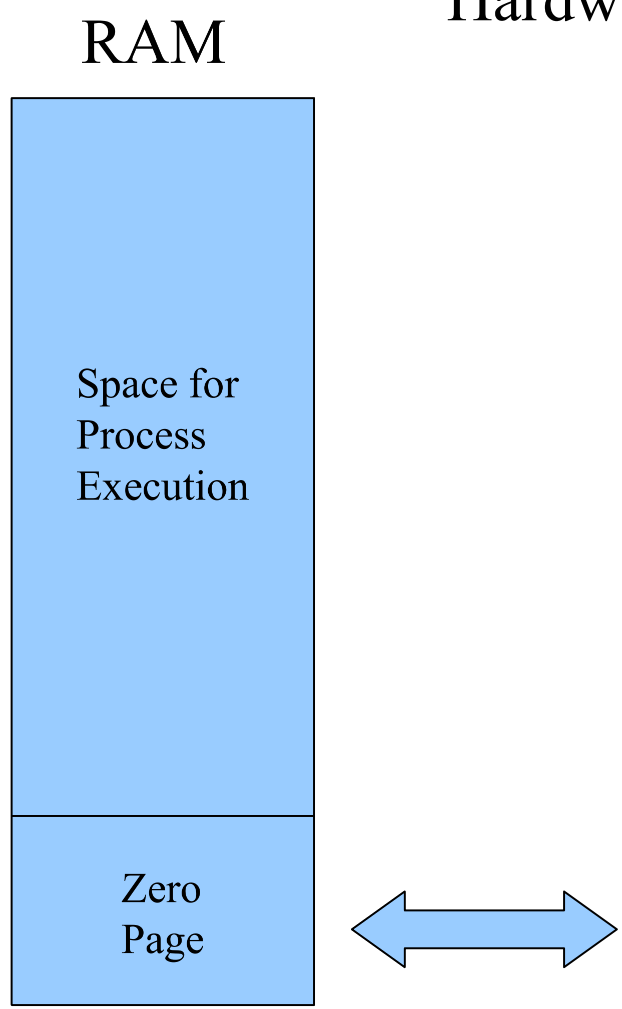
either through RAM

or through some special assembler instruction



Addressable Data Pathways

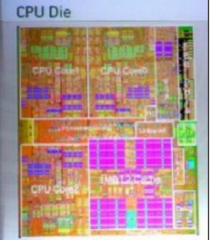
Hardware addressable through *RAM*



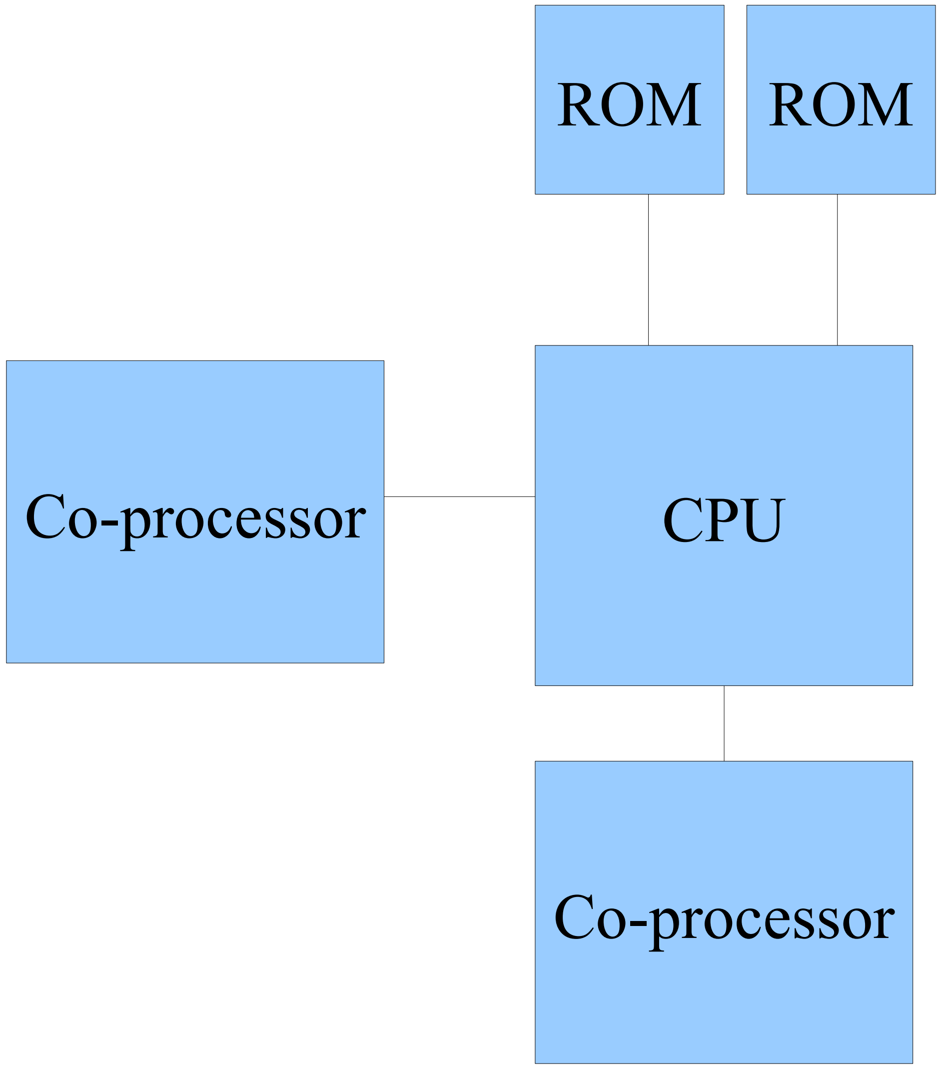
STORE address, value
LOAD register, address

Note:
- Need to know address
- Need to know binary format
- command, status & data registers

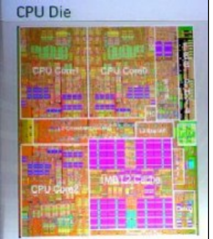
To system board slots & ports



Accessible Data Pathways



Hardware accessible through *special* instructions.

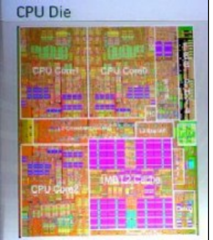


CPU Protection Schemes

Hardware accessible through ***privileged*** instructions.

Lower Boundary	- lowest allowable address reference
Upper Boundary	- highest allowable address reference
CPU Status register	- Privileged bit

- The boundary registers serve to protect the CPU from errors:
  - Set at the limits of your process space
  - Cannot access other processes directly
  - Cannot access the OS directly
  - Cannot access the Zero Page directly
- Only privileged processes can access system resources
  - In other words, have access to privileged instructions

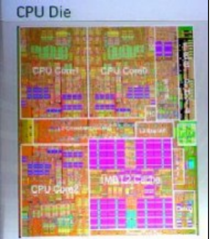


COMP 273
Introduction to Computer Systems

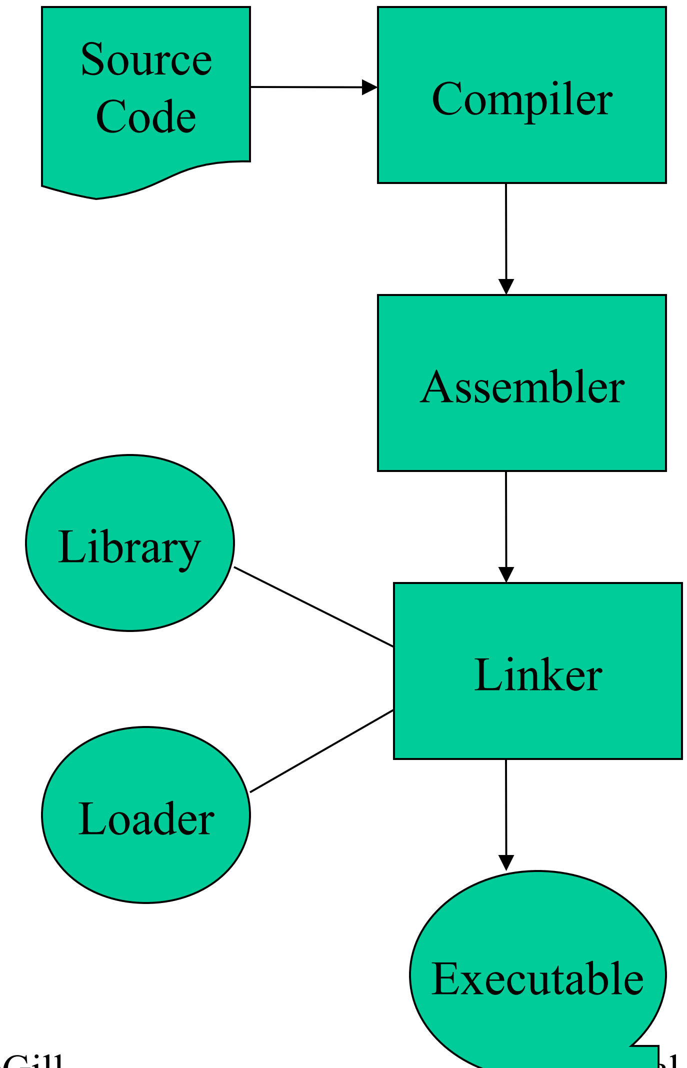
Part 2

From Source Code to Executable

COMP 273
Introduction to Computer Systems



Compiling



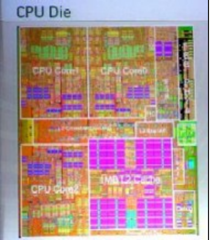
- Verifies syntax
- Conversion to <u>assembly</u> code

- Verifies syntax
- Conversion to <u>machine</u> code

The .o file in C

- Verifies library calls actually exist
- Merges library code into program
- Special OS functions added (e.g. loader)

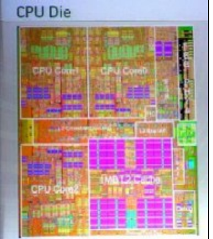
- A complete machine compatible file
- Assumes CPU and OS interface match



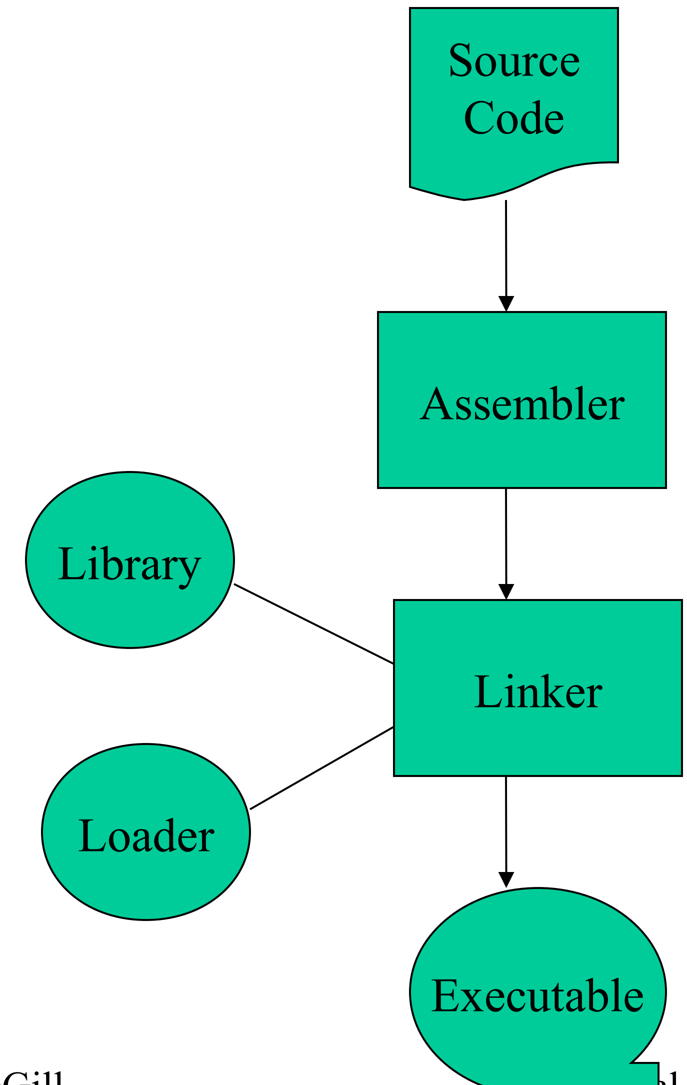
Program only execute when...

- Source code version same as compiler
- Machine code output same as CPU instruction format on computer
- Program’s Loader version same as OS API on computer

- If any not true, then program does not work

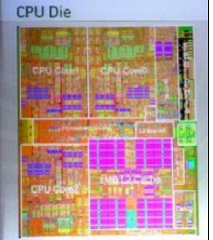


Assembly Programming



The compiler does not exist. Instead the programmer produces the assembly code.

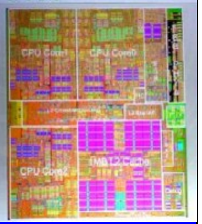
The rest is the same



COMP 273
Introduction to Computer Systems

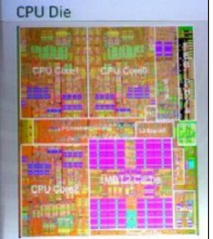
Part 3

The Assembler



Assembler Directives

Besides the assembly programming language, the assembler uses <u>directives</u> to help control the assembling process and to help the programmer is various ways

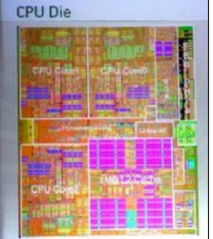


Assembler Directives

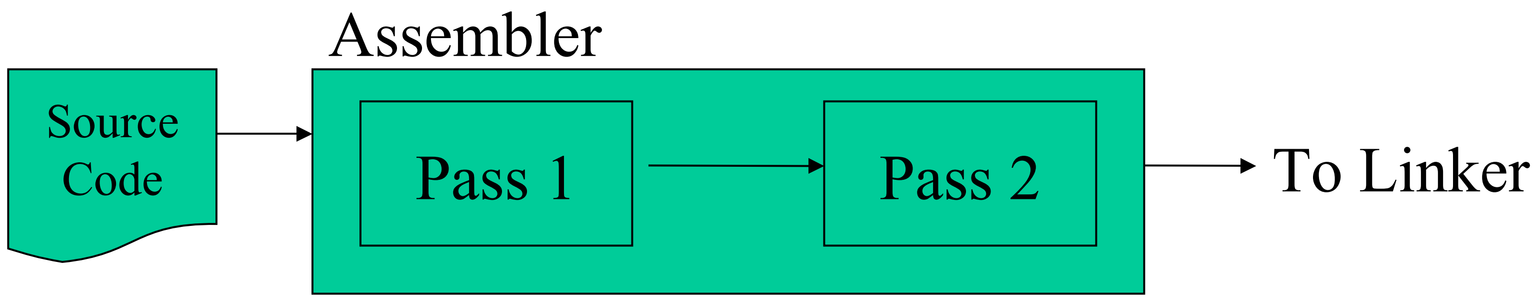
- Comments #

- Directives `.COMMAND`
  - `.text` → source code segment
  - `.data` → data segment

- Labels `LABEL:` or `LABEL`
  - `LABEL:` → a label used in a reference-definition
  - `LABEL` → a reference to the label-definition

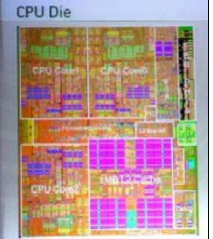


Two-Pass Assembly



- Pass 1: Build Symbol Table
  - o Identify all labels
  - o Determine offset address of all identifiers

- Pass 2: Build Machine Language Program
  - Basic 1-to-1 map between assembly and machine code
  - With help from the symbol table
  - Remember all MIPS instructions are 32-bits long



Assembly: Pass 1

Building the Symbol Table

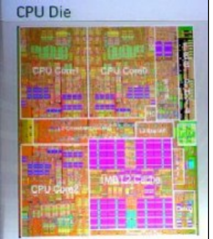
Label	Address
_start	x
Sum	10

Symbol Table

1. Scan source from top to bottom
2. Count the number of bytes in each instruction passed, use this to determine how many bytes down we are from the beginning of the program
3. If a label is encountered and it is not already in the Symbol Table, insert it and the offset from the beginning of the program

Where:

- x is called the *Base* address. It points to the first instruction in the program and it is undetermined. Its true value is given by the OS after the program is launched when the actual location in RAM where the program is placed is known.
- 10, or any number after x, is called the *offset* and with x determines the true location of the label in RAM (e.g. label = x + offset)

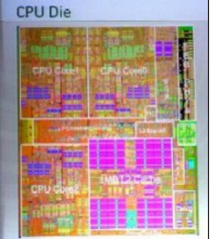


COMP 273
Introduction to Computer Systems

Create a Symbol Table Example

```
##
## length.a - prints out the length of character
## string "str".
##
##      t0 - holds each byte from string in turn
##      t1 - contains count of characters
##      t2 - points to the string
##

################################################
#                                              #
#               text segment                   #
#                                              #
################################################

        .text
        .globl __start
__start:                # execution starts here
        la $t2,str      # t2 points to the string
        li $t1,0        # t1 holds the count
nextCh: lb $t0,($t2)    # get a byte from string
        beqz $t0,strEnd # zero means end of string
        add $t1,$t1,1   # increment count
        add $t2,1       # move pointer one character
        j nextCh        # go round the loop again

strEnd: la $a0,ans      # system call to print
        li $v0,4        # out a message
        syscall

        move $a0,$t1    # system call to print
        li $v0,1        # out the length worked out
        syscall

        la $a0,endl     # system call to print
        li $v0,4        # out a newline
        syscall

        li $v0,10
        syscall         # au revoir...
```

```
################################################
#                                              #
#               data segment                   #
#                                              #
################################################

        .data
str:    .asciiz "hello world"
ans:    .asciiz "Length is "
endl:   .asciiz "\n"

##
## end of file length.a
```



Assembly: Pass 2

The Machine Code Generator

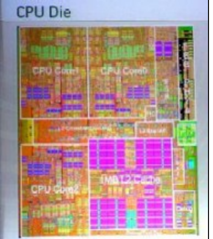
Address	Instruction
x	

Assembly Template

1. Scan source from top to bottom
2. Note that one line of assembly is one line of machine
3. For each line generate the machine version and insert into the template
   - Each COMMAND has its own bit pattern for op-code & format for instruction
   - Fill-out format with arguments provided in COMMAND

<u>Note</u>:

- The first instruction address is x
- All subsequent addresses are offsets
- “Instruction” stores the binary version of the original assembler instruction



Converting an Instruction

(add,sub,and, or, slt) Op-Code

0	rs	rt	rd	shamt	funct
31–26	25–21	20–16	15–11	10–6	5–0

Bit positions

a. R-type instruction

load store

Loc = base + address i.e. offset

35 or 43	rs	rt	address
31–26	25–21	20–16	15–0

Bit positions

b. Load or store instruction

base

43 – source
35 - dest

breq

4	rs	rt	address
31–26	25–21	20–16	15–0

Bit positions

c. Branch instruction

Equal?

Loc = sign ext, sift 2, add to PC

d. Jump Instruction

2	address

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

Example: Convert this…

Example: add $t1, $t2, $t3

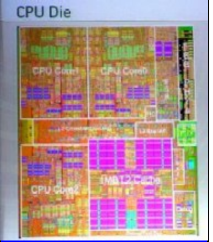
Answer: 000000 00010 00011 00001 00000 100000

Note how the integer binary number is used to indicate the register number.

The above ADD command is a register command – it's arguments use only registers.

The command ADDI would be used to mix registers with integers… addi $t1, $t2, 1 ← more later…

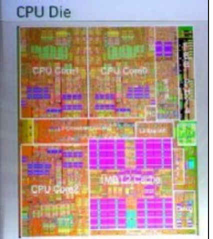
addi opcode = 001000



COMP 273
Introduction to Computer Systems

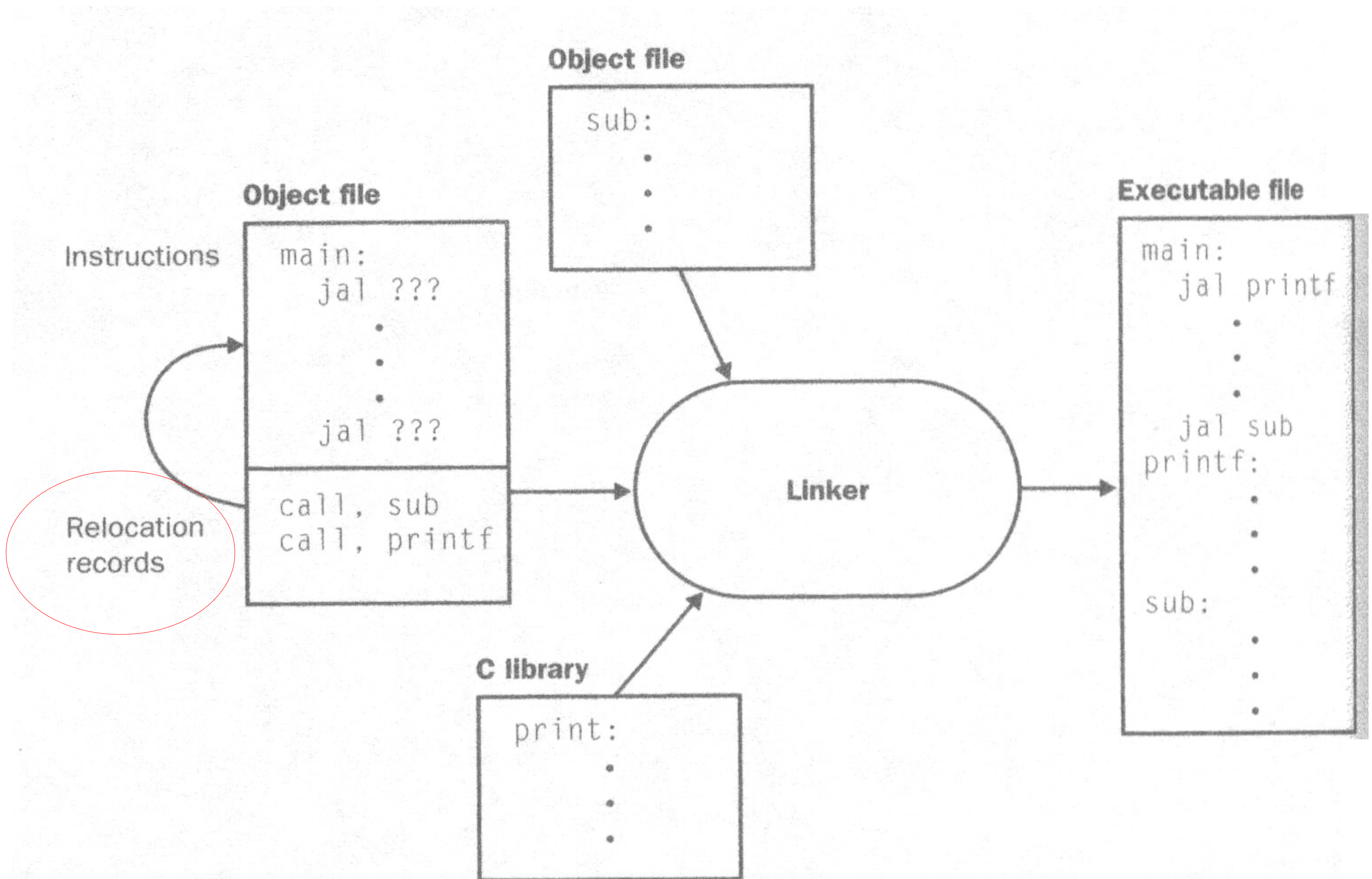
Part 4

The Linker

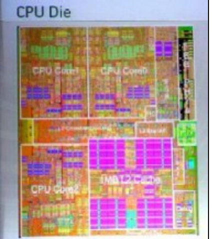


COMP 273
Introduction to Computer Systems

A Linker Example

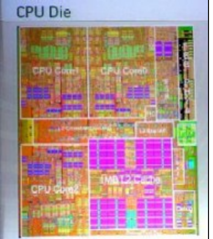


gcc –c file.c
gcc –o outfile file.o



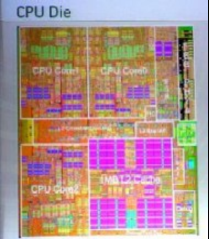
Basic Operations

- Searches the language library to find routines used by the program and inserts them at the end of the machine file
- Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
- Resolves references among files
- Ensures that there are no undefined labels



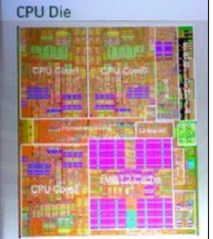
Part 5

Loading & Loaders



Definitions

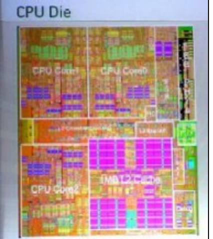
- Loading:
  - The responsibility of putting a program into memory (RAM) and notifying the OS of its presence.
  - The OS responsibility will be to pass control of the CPU to the program at a later time.
- Loader:
  - A special subroutine that knows how to put the program into RAM and notify the OS of its presence.
  - Three types of OS:
    - Some OS expect a *Loader Function* to be inserted as the first instruction of your program. This is done by the linker.
    - Other OS have this function built-in and require your program to simply call the built-in function.
    - Some OS do not need this at all and take care of the entire process internally. But requires a data record for parameters.



The Unix Loading Algorithm

1. Read file header to determine CODE and DATA size requirements
2. Find space in RAM for program and set BASE address.
3. Copy instructions and data from file into RAM.
4. Copy any run-time arguments onto run-time stack.
5. Assign SP to next free stack position and IP/PC to first instruction
6. Clear registers

<u>Note</u>: Steps 5 and 6 may be controlled directly by OS and not the loader.



COMP 273

Introduction to Computer Systems

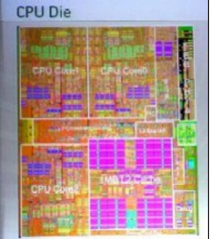
Part 6

Run-Time Environment

Understanding the language

```
##
## length.a - prints out the length of character
## string "str".
##
##       t0 - holds each byte from string in turn
##       t1 - contains count of characters
##       t2 - points to the string
##

#################################################
#                                               #
#               text segment                    #
#                                               #
#################################################

        .text
        .globl __start
__start:                # execution starts here
        la $t2,str      # t2 points to the string
        li $t1,0        # t1 holds the count
nextCh: lb $t0,($t2)    # get a byte from string
        beqz $t0,strEnd # zero means end of string
        add $t1,$t1,1   # increment count
        add $t2,1       # move pointer one character
        j nextCh        # go round the loop again

strEnd: la $a0,ans      # system call to print
        li $v0,4        # out a message
        syscall

        move $a0,$t1    # system call to print
        li $v0,1        # out the length worked out
        syscall

        la $a0,endl     # system call to print
        li $v0,4        # out a newline
        syscall

        li $v0,10
        syscall         # au revoir...

#################################################
#                                               #
#               data segment                    #
#                                               #
#################################################

        .data
str:    .asciiz "hello world"
ans:    .asciiz "Length is "
endl:   .asciiz "\n"

##
## end of file length.a
```

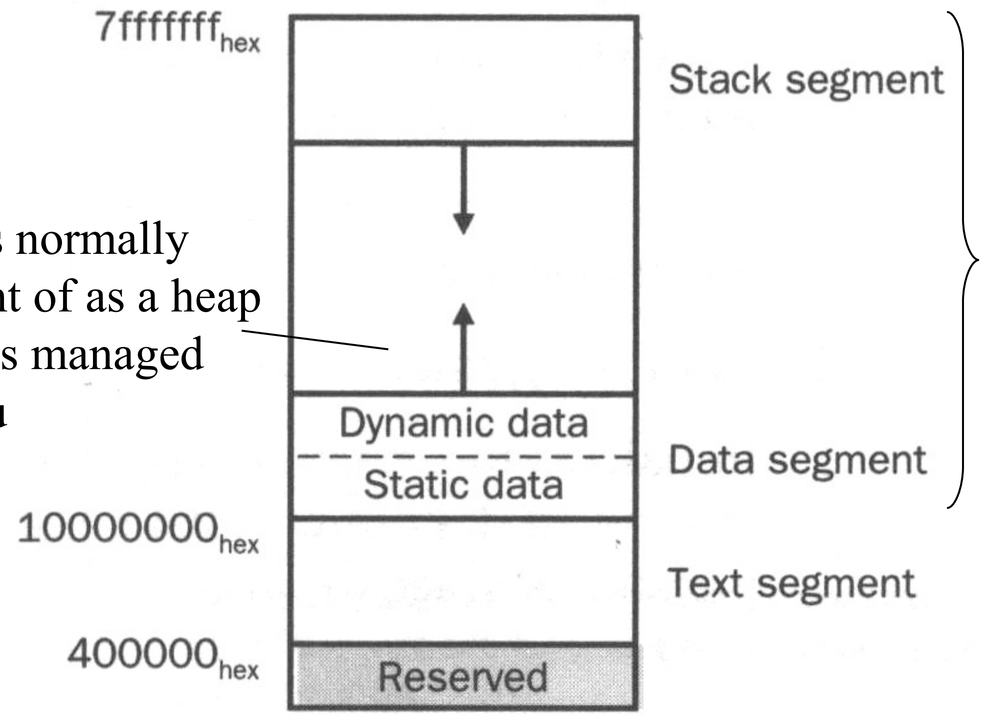
COMP 273
Introduction to Computer Systems



Virtual Memory Usage

Bottom of stack

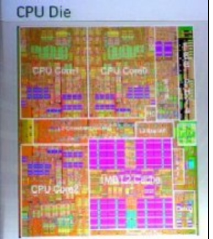
SP register for top of stack



This is normally thought of as a heap but it is managed by you

Stack and data segments are actually shared space and can *crash* (overlap)

Real memory vs. Virtual Memory



Program File Format

Under UNIX Convention

- The *object file header* describes the size and position of the other pieces of the file.
- The *text segment* contains the machine language code for routines in the source file. These routines may be unexecutable because of unresolved references.
- The *data segment* contains a binary representation of the data in the source file. The data also may be incomplete because of unresolved references to labels in other files.
- The *relocation information* identifies instructions and data words that depend on absolute addresses. These references must change if portions of the program are moved in memory.
- The *symbol table* associates addresses with external labels in the source file and lists unresolved references.
- The *debugging information* contains a concise description of the way in which the program was compiled, so a debugger can find which instruction addresses correspond to lines in a source file and print the data structures in readable form.

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information