UCCD1133

Introduction to Computer Organisation and Architecture

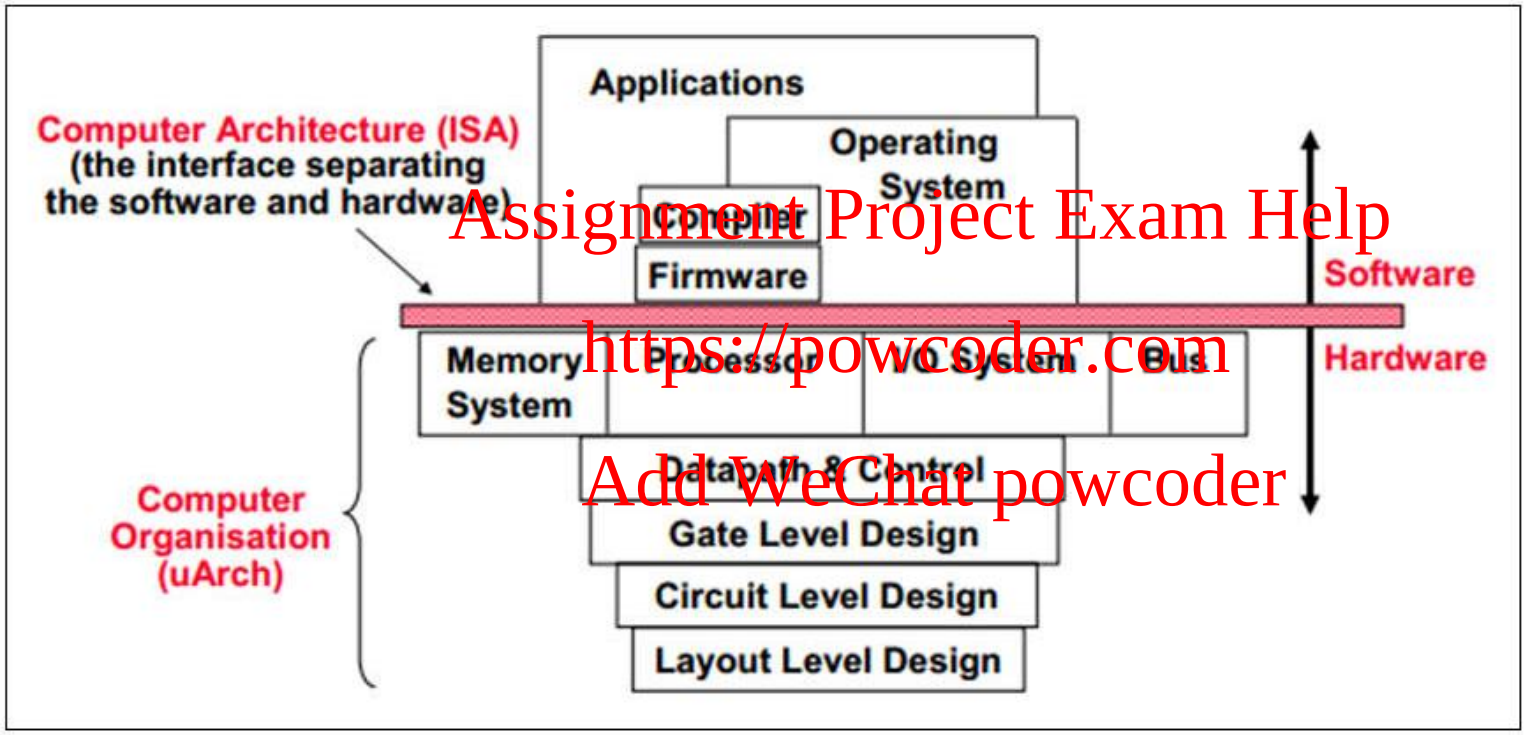
Chapter 5

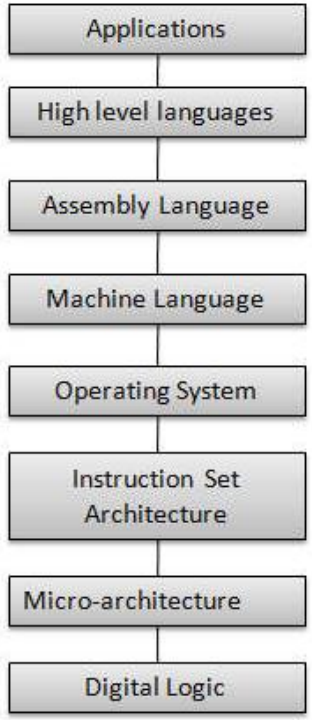
Program Execution

Disclaimer

- This slide may contain copyrighted material of which has not been specifically authorized by the copyright owner. The use of copyrighted materials are solely for educational purpose. If you wish to use this copyrighted material for other purposes, you must first obtain permission from the original copyright owner.

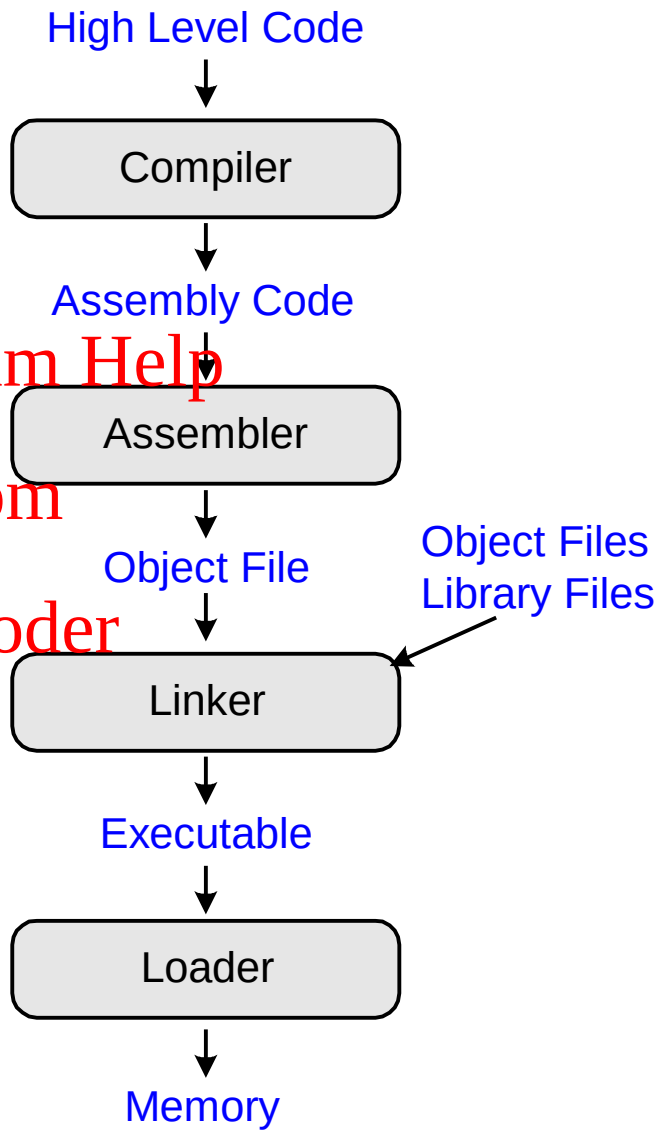
Computer Architecture Levels of Abstraction



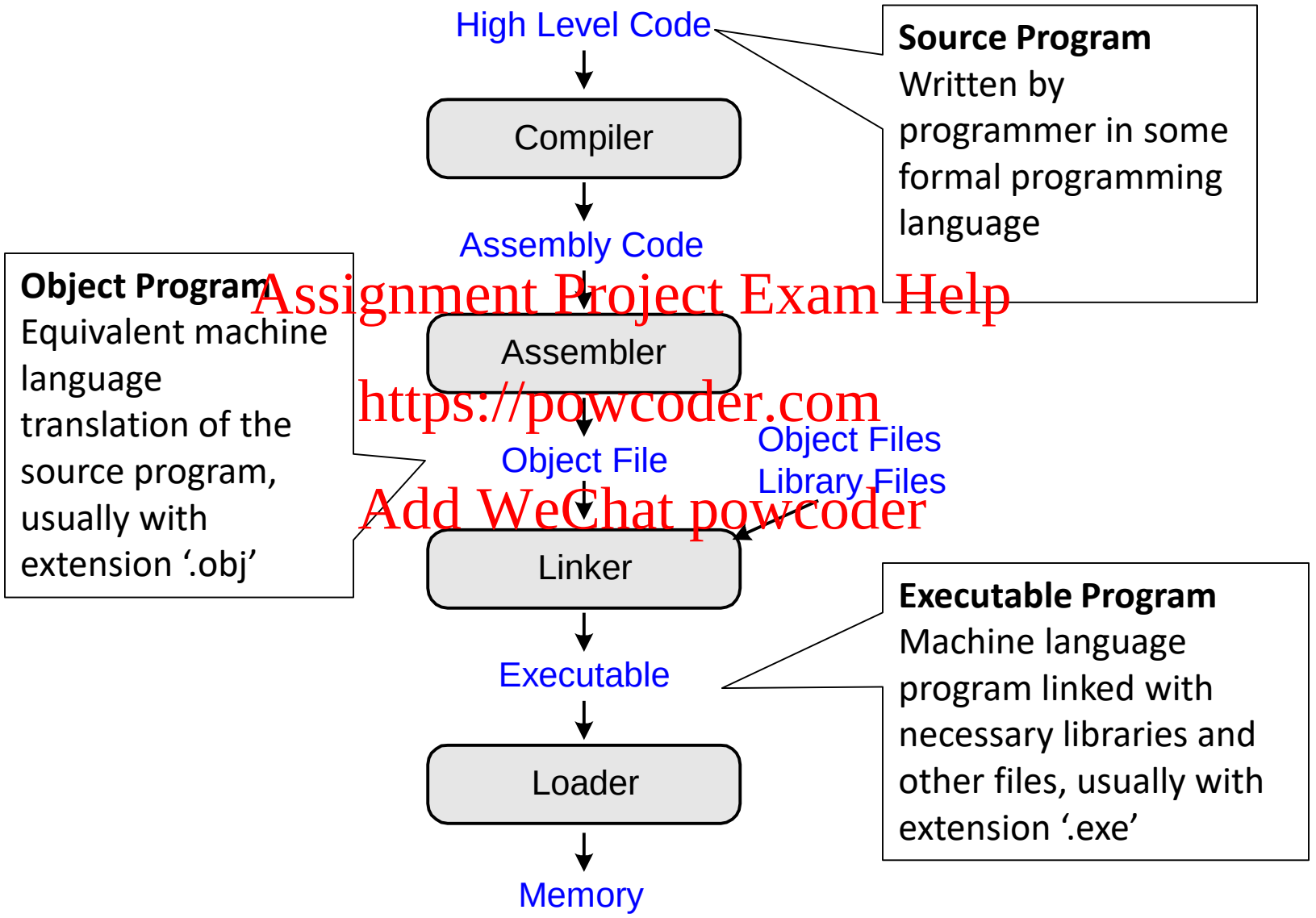


How to Compile & Run a Program

1. **C**ompiler
2. **A**ssembler
3. **L**inker
4. **L**oader



How to Compile & Run a Program



Compiler

- Compiler converts a single HLL file into a single assembly file.
- Example:
  - Input: `foo.c` (C code)
  - Output: `foo.s` (MIPS)

- Programming in HLL
  - Allow the programmer to think in a more natural language for their intended use
    - Fotran for scientific computation
    - Cobol for business programming
    - Lisp for symbol manipulation
    - Java for web programming
  - Improve programmer productivity – more understandable code that is easier to design, debug and validate
  - Improve program maintainability
  - Allow programs to be independent of the computer (compilers and assemblers can translate high-level language programs to the binary instructions of any machine)
  - Emergence of optimizing compilers that produce very efficient assembly code optimized for the target machine

Compiler

- Most programmers don't write assembly code for a number of reasons, including:
  - Readability: HLL's are clearer than assembly.
  - Portability: Assembly code is ISA specific.
  - Productivity: One line of HLL code often takes many lines of assembly, and programmers can write a fixed number of lines of code a day.

- Some reasons to write code in assembly in the real world:

  - To exploit hardware features that have no analogues in HLL
    - there are some things that cannot be expressed in HLL (*e.g.*, I/O, accesses to special registers).

  - Programs with speed- and size-critical, in particular, embedded apps
    - Humans can out code compilers in certain circumstances (performance and code size).

Example Program: C Code

```
int f, g, y;  // global
  variables



int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);

  return y;
}



int sum(int a, int b) {
  return (a + b);
}
```

Example Program: MIPS Assembly

```
int f, g, y;  // global variables


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);

  return y;
}


int sum(int a, int b) {
  return (a + b);
}
```

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4     # stack frame
  sw   $ra, 0($sp)      # store $ra
  addi $a0, $0, 2       # $a0 = 2
  sw   $a0, f           # f = 2
  addi $a1, $0, 3       # $a1 = 3
  sw   $a1, g           # g = 3
  jal  sum              # call sum
  sw   $v0, y           # y = sum()
  lw   $ra, 0($sp)      # restore $ra
  addi $sp, $sp, 4      # restore $sp
  jr   $ra              # return to OS
sum:
  add  $v0, $a0, $a1    # $v0 = a + b
  jr   $ra              # return
```

Assembler

- The assembler convert the assembly language code into an object file containing machine language code.
- An assembler
  - Reads and uses assembler directives, such as
    - `.text`: subsequent items put in user text segment
    - `.data`: subsequent items put in user data segment
    - `.global sym`: declares `sym` as global label
  - Replace pseudoinstructions
  - Assigns instruction addresses
  - Record the addresses of the symbols (labels and global variable names) in the symbol table.
  - Create a relocation table to list the items that needs the address later.
    - Example: `lui $4, l.str`
    - the `l.str` is a label from somewhere, but don't know the value yet.
  - Produce the machine code and stored it in the object file.
- Example:
  - Input: `foo.s` (MIPS)
  - Output: `foo.obj` (Object file)

Assembler

- The translation process has two major parts.

  - During the **first pass**, the assembler
    - Reads each line of an assembly file.
    - If a line begins with a label, records in its ***symbol table*** the name of the label and the address of the memory that the instruction occupies.

  - During the **second pass**, the assembler
    - Use the symbol table to produces ***machine code***.
    - The process is similar to Chapter 4.
    - Instructions and data words that reference an external symbol defined in another file cannot be completely assembled.

Example Program: Symbol Table

```
0x00400000        main:   addi $sp, $sp, -4    # stack frame
0x00400004                sw   $ra, 0($sp)     # store $ra
0x00400008                addi $a0, $0, 2      # $a0 = 2
0x0040000C                sw   $a0, f          # f = 2
0x00400010                addi $a1, $0, 3      # $a1 = 3
0x00400014                sw   $a1, g          # g = 3
0x00400018                jal  sum             # call sum
0x0040001C                sw   $v0, y          # y = sum()
0x00400020                lw   $ra, 0($sp)     # restore $ra
0x00400024                addi $sp, $sp, 4     # restore $sp
0x00400028                jr   $ra             # return to OS
0x0040002C          sum:  add  $v0, $a0, $a1   # $v0 = a + b
0x00400030                jr   $ra             # return
```


Symbol	Address

Example Program: Symbol Table

```
0x00400000        main:  addi $sp, $sp, -4    # stack frame
0x00400004               sw   $ra, 0($sp)     # store $ra
0x00400008               addi $a0, $0, 2      # $a0 = 2
0x0040000C               sw   $a0, f          # f = 2
0x00400010               addi $a1, $0, 3      # $a1 = 3
0x00400014               sw   $a1, g          # g = 3
0x00400018               jal  sum             # call sum
0x0040001C               sw   $v0, y          # y = sum()
0x00400020               lw   $ra, 0($sp)     # restore $ra
0x00400024               addi $sp, $sp, 4     # restore $sp
0x00400028               jr   $ra             # return to OS
0x0040002C        sum:   add  $v0, $a0, $a1   # $v0 = a + b
0x00400030               jr   $ra             # return
```


Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Object File Format

- Object file header: size and position of the other pieces of the object file
- Text segment: the machine code for the routines (instructions) in the source file. Maybe unexecutable because of unresolved references.
- Data segment: binary representation of the data in the source file. The data also may be incomplete because of unresolved references to labels in other files.
- Relocation information: identifies instructions and data that depend on absolute addresses
  - j, jal, and some loads and stores (e.g. lw $t1, 100($zero)) use absolute addresses
  - These references must change if portions of the program are moved in memory.
- Symbol table: addresses with external labels in the source file and lists unresolved reference.
- Debugging information: contains a concise description of the way the program was compiled for debugging purpose.

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

A UNIX assembler produces an object file with 6 distinct sections.

Linker

- Most programs are very large and consist of several modules. Even small programs use existing code provided by the programming environment called libraries.

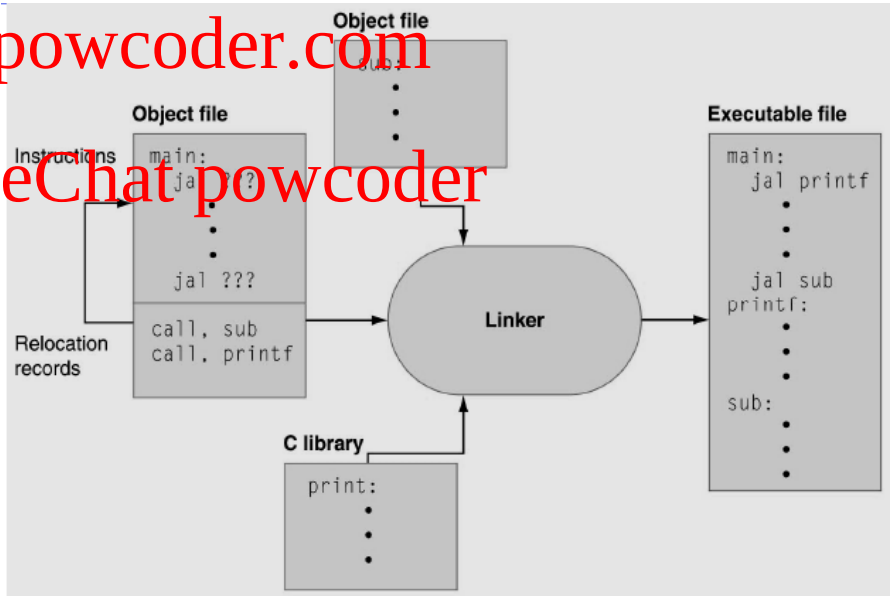
- Assembler only sees one compiled program at a time. It's the job of the linker to link them together.

- A linker combines several object files into a single executable and resolves absolute addresses.
  - It calculates the absolute address of each label to be jumped to (internal or external) and each piece of data being referenced.

- It enables separate compilation of files
  - Changes to one file do not require recompiling of whole program

Linker

- A linker
  - Merge text / data segments
  - Resolve reference
    - Search for reference (data or label) in all symbol tables
    - If not found, search library files (for example, `printf`)
    - Determine the absolute memory addresses
    - Fill in the absolute address in the machine code
- Example:
  - Inputs: foo.obj, lib.obj
  - Output: abc.exe



Executable File Format

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	**Address**	**Instruction**
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008
Data segment	**Address**	**Data**
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Example: Procedure A Object File

```
lw $a0, xxx($gp)
jal B
…
```

refer to the address of data word X

instruction that refer to the addresses of procedures B

Procedure A needs to find the address for the variable labeled X to put in the load instruction and to find the address of procedure B to place in the `jal` instruction.

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20 hex	
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	…	…	
Data segment	0	(X)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	–	
	B	–	

Example: Procedure B Object File

```
sw $a1, yyy($gp)
jal A
...
```

refer to the address of data word Y

instruction that refer to the address of procedures A

Procedure B needs the address of the variable labeled Y for the store instruction and the address of procedure A for its `jal` instruction.

Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	sw $a1, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Example: Executable File

Procedure A

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	-	
	B	-	

Procedure B

Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	sw $a1, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	-	
	A	-	

linker

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, -8000 hex($gp)
	$0040\ 0004_{hex}$	jal 40 0100 hex
	...	...
	$0040\ 0100_{hex}$	sw $a1, -7980 hex($gp)
	$0040\ 0104_{hex}$	jal 40 0000 hex
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

* `$gp` is initialized to $1000\ 8000_{hex}$
* Recall that MIPS instructions are word aligned, so `jal` drops the right two bits to increase the instruction's address range. Thus, it use 26 bits to create a 28-bit byte address. Hence, the actual address in the lower 26 bits of the `jal` instruction in this example is $10\ 0040_{hex}$, rather than $40\ 0100_{hex}$.

Static vs Dynamically Linked Libraries

- **Statically Linked Library:** fastest way to call library routines.

  Problems:
  - if the library has new release, the program still uses the old version (unless recompile if we have the source files)
  - When the program runs, the *entire* library is loaded including unused parts (library can be larger than the main program)

- An alternative is **dynamically linked libraries (DLL),** common on Windows & UNIX platforms

  - Library routines are not linked and loaded until a routine is called during execution.
  - Only the required routine is linked.

- Use of a dynamic linker-loader.
  - Find the desired routine, remap it, and “link” it to the calling routine

- Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system.
  - But, advantages more than disadvantages.

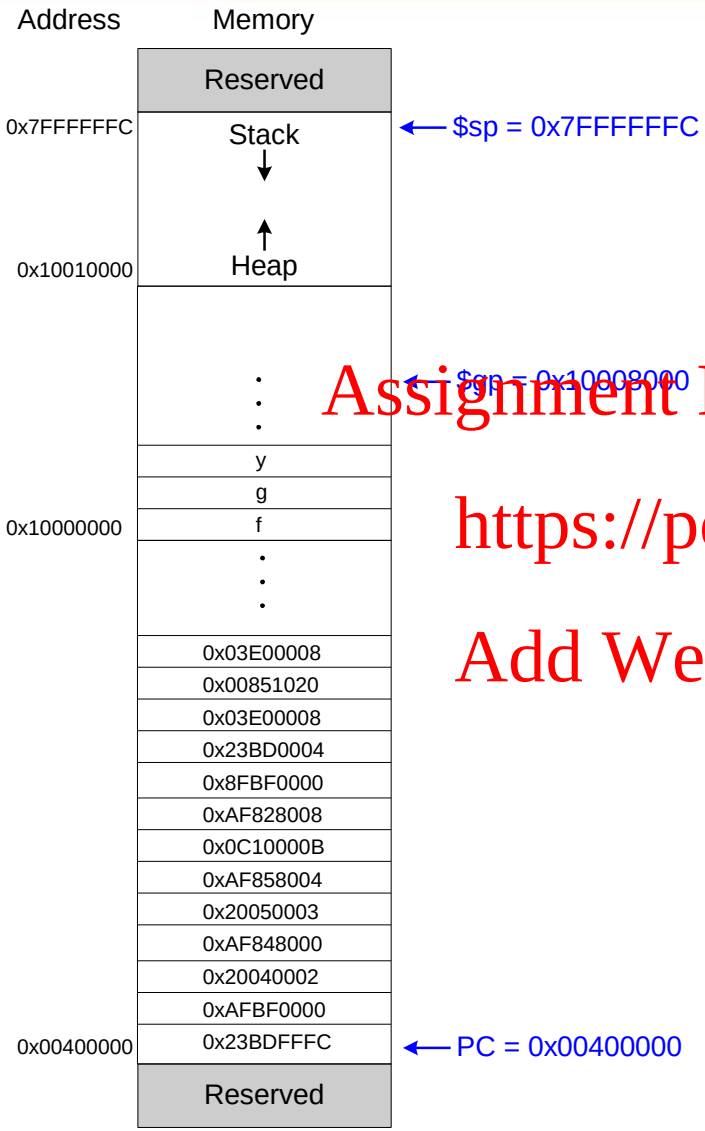
Dynamically Linked Libraries

- Space/time issues
  - + Storing a program requires less disk space
  - + Sending a program requires less time
  - + Executing two programs requires less memory (if they share a library)
  - – At runtime, there's time overhead to do link
- Upgrades
  - + Replacing one file upgrades every program that uses that library
- Microsoft's Windows relies extensively on dynamically linked libraries, and it is also the default when executing programs on UNIX systems today.

Loader

- A program that links without error can be run.
- Before being run, the program resides in a file on secondary storage, such as a disk. It needs to be brought into memory, usually by operating system, to start running.
- The operating system loads a program by:
  - Reads the executable file's header to determine the size of the text and data segments.
  - Creates a new address space for the program.
  - Copies the instruction and data from executable file into the new address space
  - Copies arguments passed to the program onto the stack
  - Initializes the machine registers
    - Most registers are cleared.
    - Sets `$gp` to 0x1000 8000 (the middle of the global data segment)
    - Sets `$sp` to 0x7FFF FFFC (the top of dynamic data segment)
  - Jump to the beginning of the program
    - Perform a `jal` 0x0040 0000.
- Example:
  - Input: abc.exe
  - Output: <program is run>

Example Program: In Memory



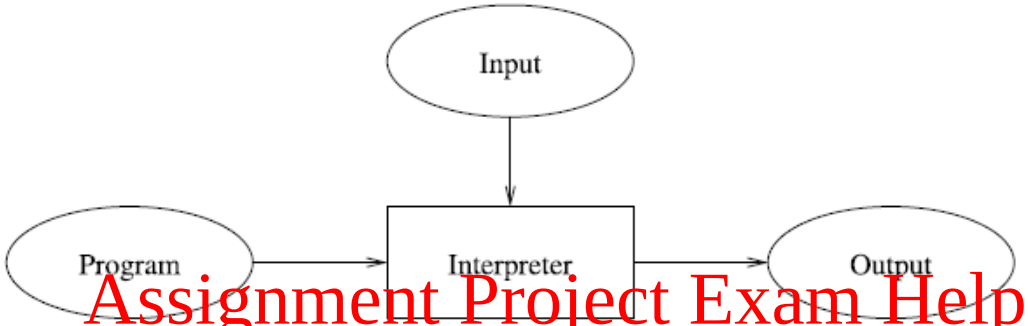
Interpreter vs compiler

- Interpreter and compiler are both translator
  - convert high level language commands into machine code.

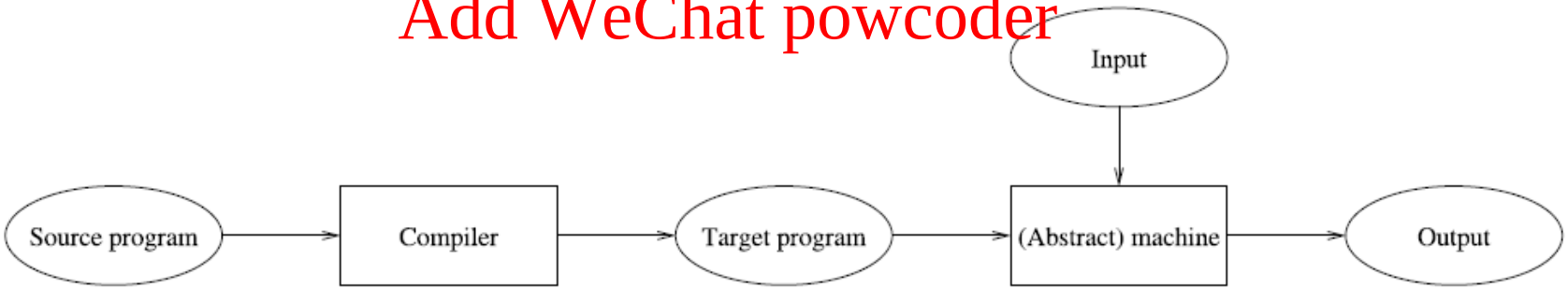
- A compiler translate the whole code into machine code before running the program.
  - Compiled programming languages include Java, Fortran and C++.

- An interpreter translate one instruction at a time. The CPU execute each instruction before the interpreter moves on to the next instruction.
  - Unlike compilation, interpretation has
    - No linking
    - No object code generated
    - Source statements executed line by line
  - Interpreted languages include JavaScript, PHP, Python and Ruby.

Interpreter vs compiler



Interpretation in one stage.



Compilation and execution in two stages.

Advantages and Disadvantages of Compiler

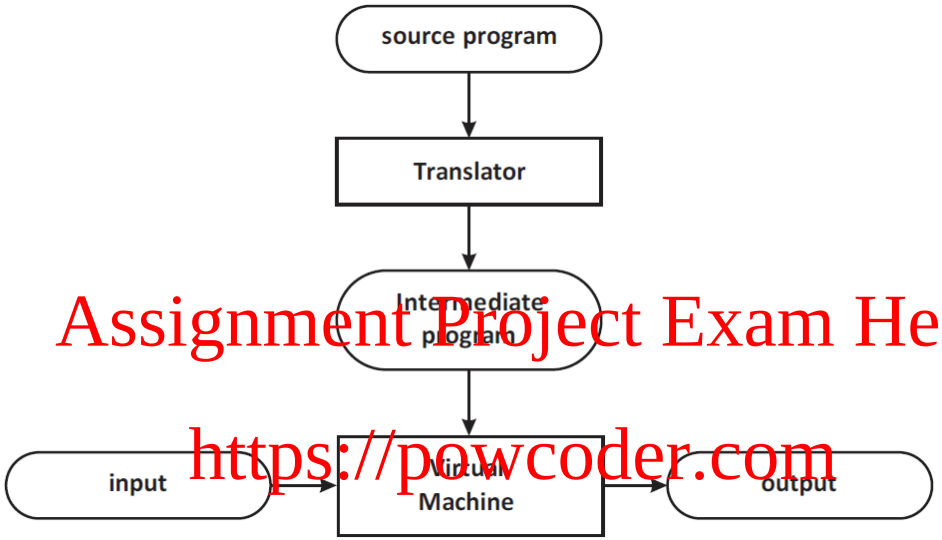
- Advantages
  - Faster Execution
  - Single file to execute
  - Compiler can do better diagnosis of syntax and semantic errors, since it has more info than an interpreter (Interpreter only sees one line at a time)
  - Can find syntax errors before run program
  - Compiler can optimize code
- Disadvantages
  - Harder to debug
  - Takes longer to change source code, recompile and relink

Advantages and Disadvantages of Interpreter

- Advantages
  - Easier to debug
  - Faster development time
  - Interpreter closer to high-level, so can give better error message
  - Provides instruction set independence: can run on any machine

- Disadvantages
  - Slower execution times
  - No optimization
  - Need all of source code available
  - Source code larger than executable for large systems
  - Interpreter must remain installed while the program is interpreted

Hybrid System



- Nearly all Java language processors conform to this hybrid system.
- Java programs are compiled to an intermediate form call bytecodes. Bytecode programs are executed by a Java virtual machine.