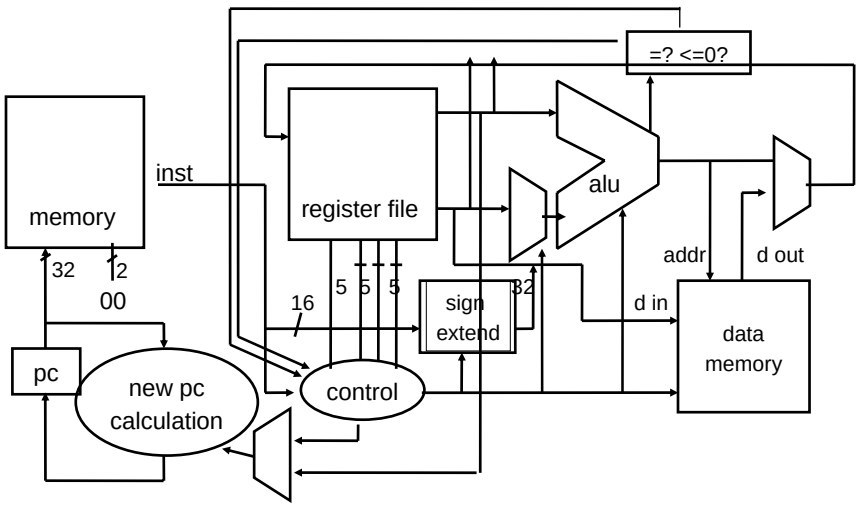
CS 316:
MIPS Assembler

Kavita Bala
Fall 2007
Computer Science
Cornell University

Announcements

- PA 2
  - Logisim seems to be erratic
    - New library will help some (with incrementer)
  - Don't use incrementer 4 times
    - What do we know about addresses and their alignment?
  - Can do delay slot or not for top 4 bits
    - Tell us what it is
    - Will fix in PA 3

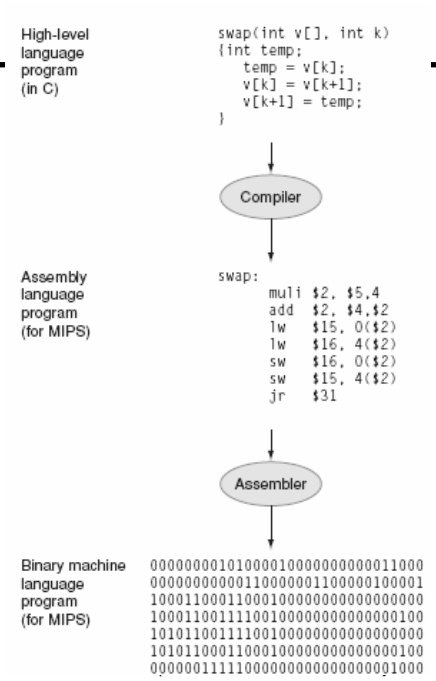
Branch



Examples

- A[12] = h + A[8]

- lw, $t0, 32($s3)
- add $t0, $s2, $t0
- sw $t0, 48($s3)



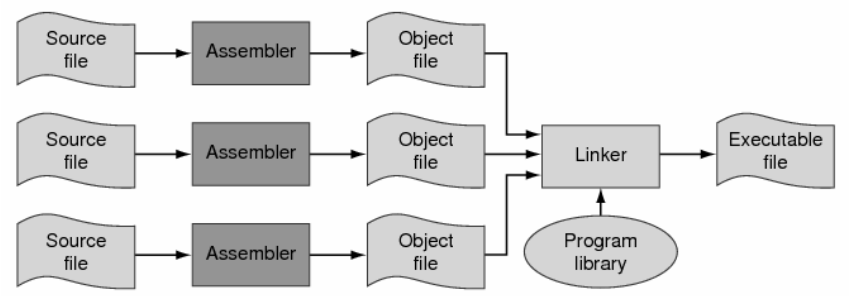


FIGURE A.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

Assembler

- Translates text assembly language to binary machine code
- Input: a text file containing MIPS instructions in human readable form
- Output: an object file (.o file in Unix, .obj in Windows) containing MIPS instructions in executable form

Assembly Language Instructions

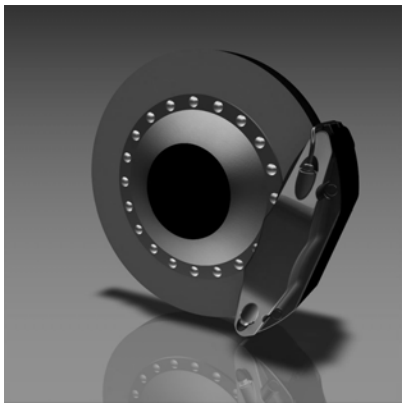
- Arithmetic
  - ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
  - ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLL, SRL, SLLV, SRLV, SRAV, SLTI, SLTIU
  - MULT, DIV, MFLO, MTLO, MFHI, MTHI
- Control Flow
  - BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
  - J, JR, JAL, JALR, BLTZAL, BGEZAL
- Memory
  - LW, LH, LB, LHU, LBU
  - SW, SH, SB
- Special
  - LL, SC, SYSCALL, BREAK, SYNC, COPROC

Assembly Language

- Assembly language is used to specify programs at a low-level
- Will I program in assembly?
  - I did, for kernel hacking
  - For performance (though compilers are getting better)
  - For highly time critical sections
  - For hardware without high level languages

Example: GPU Phong Shader

- Want to compute
  - out = N.L + (R.L)^n



Example: GPU Phong Shader

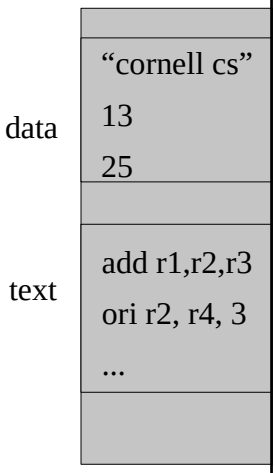
```
ADD R0, c[3], -v[OPOS]      // L-P
DP3  R1, R0, R0             // ||L-P||^2
RSQ R2, R1.W                // 1/||L-P||
MUL R0, R0, R2              // R0 = L

DP3 R3, R0, v[NRML]         // R3 = N.L

….                          // Compute E
DP3 R7, R4, v[NRML]         // E.N
MUL R7, R7, c[6]            // 2 (E.N)
MAD R8, R7, v[NRML], -R4 // 2 (E.N)N-E
DP3 R9, R8, R0              // R.L
LOG R10, R9.x               // LOG (R.L)
MUL R9, c[5].x, R10.z       // n*(LOG(R.L))
EXP R11, R9.z               // (R.L)^n
…
```


Assembly Language

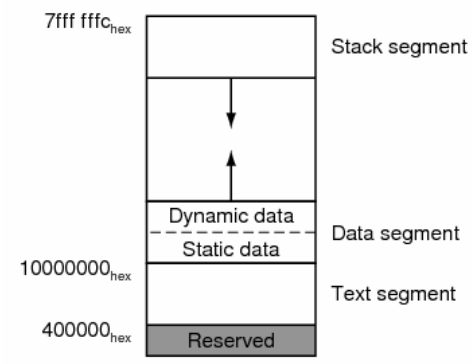
- Assembly language is used to specify programs at a low-level
- What does a program consist of?
  - MIPS instructions
  - Program data (strings, variables, etc)

Program Layout

- Programs consist of segments used for different purposes
  - Text: holds instructions
  - Data: holds statically allocated program data such as variables, strings, etc.



When you run the program



Assembling Programs

```
        .text
        .ent main
main: la $4, Larray
        li $5, 15
        ...
        li $4, 0
        jal exit
        .end main
        .data
Larray:
        .long 51, 491, 3991
```

- Programs consist of a mix of instructions, pseudo-ops and assembler directives
- Assembler lays out binary values in memory based on directives

Example pseudo-ops

- blt = slt and bne
  - blt $s3, $s4, label

  Equivalent to
  - slt $at, $s3, $s4
  - bne $at, $zero, label
- Use register $at (assembler temporary) to compile this

Examples

- gcc –S helloWorld.c
- gcc –S add1To100.c
- gcc –S add1To100Sq.c


```
addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4,  32($29)
sw       $5,  36($29)
sw       $0,  24($29)
sw       $0,  28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu    $14, $14
addiu    $8,  $14, 1
slti     $1,  $8, 101
sw       $8,  28($29)
mflo     $15
addu     $25, $24, $15
bne      $1,  $0, -9
sw       $25, 24($29)
lui      $4,  4096
lw       $5,  24($29)
jal      1048812
addiu    $4,  $4, 1072
lw       $31, 20($29)
addiu    $29, $29, 32
jr       $31
move     $2,  $0
```

Sum 1 to 100 Square

```
#include <stdio.h>

int main (int argc, char* argv[]) {

   int count = 0;
   int i = 0;
   for (i = 0; i < 100; i++) { count += i*i; }
   printf ("The sum from 0 .. 100 is %d\n", count);
}
```


MIPS Registers

- Return address: $31 (ra)
- Stack pointer: $29 (sp)
- First four arguments: $4-$7 (a0-a3)

```
        .text
        .align    2
        .globl    main
main:
        subu      $sp, $sp, 32
        sw        $ra, 20($sp)
        sd        $a0, 32($sp)
        sw        $0,  24($sp)
        sw        $0,  28($sp)
loop:
        lw        $t6, 28($sp)
        mul       $t7, $t6, $t6
        lw        $t8, 24($sp)
        addu      $t9, $t8, $t7
        sw        $t9, 24($sp)
        addu      $t0, $t6, 1
        sw        $t0, 28($sp)
        ble       $t0, 100, loop
        la        $a0, str
        lw        $a1, 24($sp)
        jal       printf
        move      $v0, $0
        lw        $ra, 20($sp)
        addu      $sp, $sp, 32
        jr        $ra

        .data
        .align    0
str:
        .asciiz   "The sum from 0 .. 100 is %d\n"
```

References

- Global labels
  - External: can be referenced from outside
  - In example
    - Main
- Local labels

Object File Generation

- A program is made up of code and data from several object files
- Each object file is generated independently
- Assembler starts at some PC address, e.g. 0, in each object file, generates code as if the program were laid out starting out at location 0x0
- It also generates a symbol table, and a relocation table
  - In case the segments need to be moved

Object file

- Header
  - Size and position of pieces of file
- Text Segment
  - instructions
- Data Segment
  - Static data
- Relocation Information
  - Instructions and data that depend on absolute addresses
- Symbol Table
  - External and unresolved references
- Debugging Information

Forward References

- Local labels can have forward references
- Two-pass assembly
  - Do a pass through the whole program, allocate instructions and lay out data, thus determining addresses
  - Do a second pass, emitting instructions and data, with the correct label offsets now determined

Forward References

- One-pass (or backpatch) assembly
  - Do a pass through the whole program, emitting instructions, emit a 0 for jumps to labels not yet determined, keep track of where these instructions are
  - Backpatch, fill in 0 offsets as labels are defined
- Pros and cons
  - Faster
  - But need to hold whole program in memory
  - Have to do largest branch possible

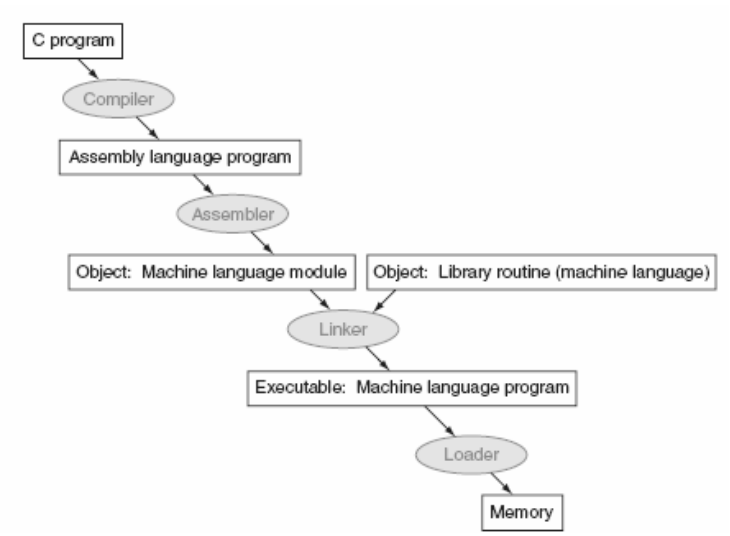
Handling Forward References

- Example:
  - bne $1, $2, L
    sll $0, $0, 0
    L: addiu $2, $3, 0x2
- The assembler will change this to
  - bne $1, $2, +1
    sll $0, $0, 0
    addiu $7, $8, $9

Handling Forward References

- Final machine code
  - 0X14220001 # bne
    0x00000000 # sll
    0x24620002 # addiu

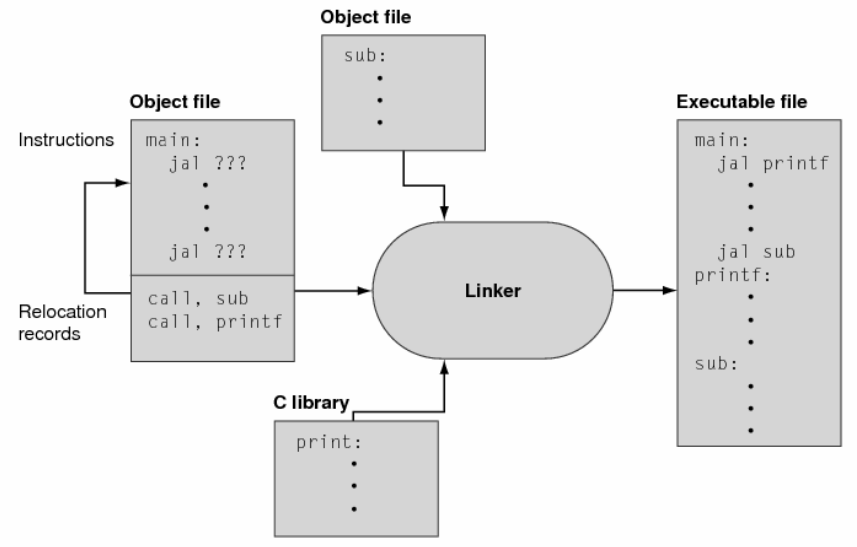
From Assembly to Running



Separate Compilation

- Separately compiling modules and linking them together removes the need to recompile the whole program every time something changes
- Need to just recompile a small module
- A linker coalesces object files together to create a complete program

Linker



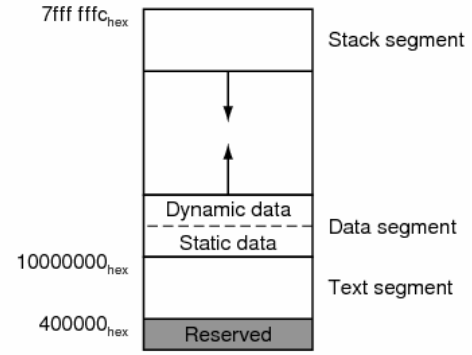
Linkers

- Combine object files into an executable
  - Resolve symbols
  - Creates final executable
  - Stores entry point in executable so processor knows where to start executing
- End result: a program on disk, ready to execute

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	—	
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	—	

When you run the program

- jal are easy
- gp will be explained next time



jal addresses

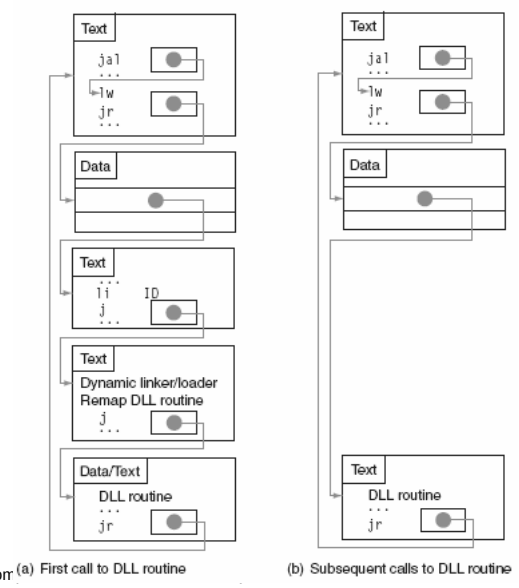
Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

Linkers

- Static linkers
  - Doesn't reflect changes
  - Whole library (big)
- Dynamic linkers
  - Dynamically linked libraries (dlls)
  - Integrate code at runtime
    - One copy of shared library in memory
    - But linked it all, so still quite large

Linkers

- Dynamic linkers
  - Dynamically linked libraries (dlls)
  - Lazy



(a) First call to DLL routine (b) Subsequent calls to DLL routine

Loaders

- Reads executable from disk
- Loads code and data into memory
- Initializes registers, stack, arguments
- Jumps to entry-point
- Part of the Operating System (OS)