Handouts in the front
and back of the room

Assemblers, Linkers, and Loaders

Prof. Kavita Bala and Prof. Hakim Weatherspoon

CS 3410, Spring 2014

Computer Science

Cornell University

See: P&H Appendix A1-2, A.3-4 and 2.12

Goal for Today: Putting it all Together

Brief review of calling conventions

Compiler output is assembly files

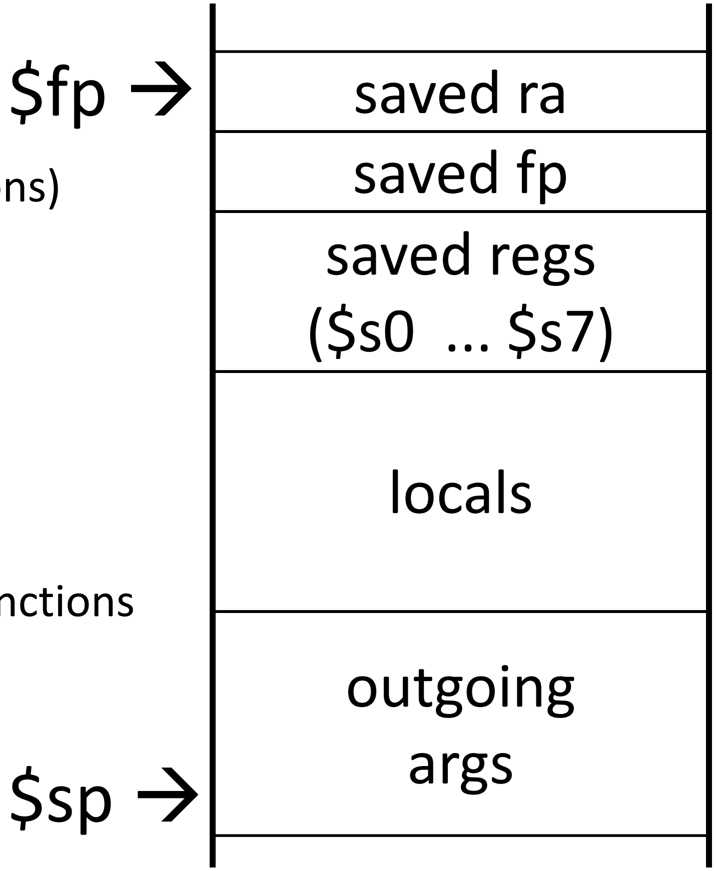
Assembler output is obj files

Linker joins object files into one executable

Loader brings it into memory and starts execution

Recap: Calling Conventions

- first four arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed in parent’s stack frame
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)
  - contains $fp
  - contains local vars (possibly clobbered by sub-functions)
  - contains extra arguments to sub-functions (i.e. argument “spilling)
  - contains space for first 4 arguments to sub-functions
- callee save regs are preserved
- caller save regs are not preserved
- Global data accessed via $gp



$fp →

saved ra

saved fp

saved regs ($s0 ... $s7)

locals

outgoing args

$sp →

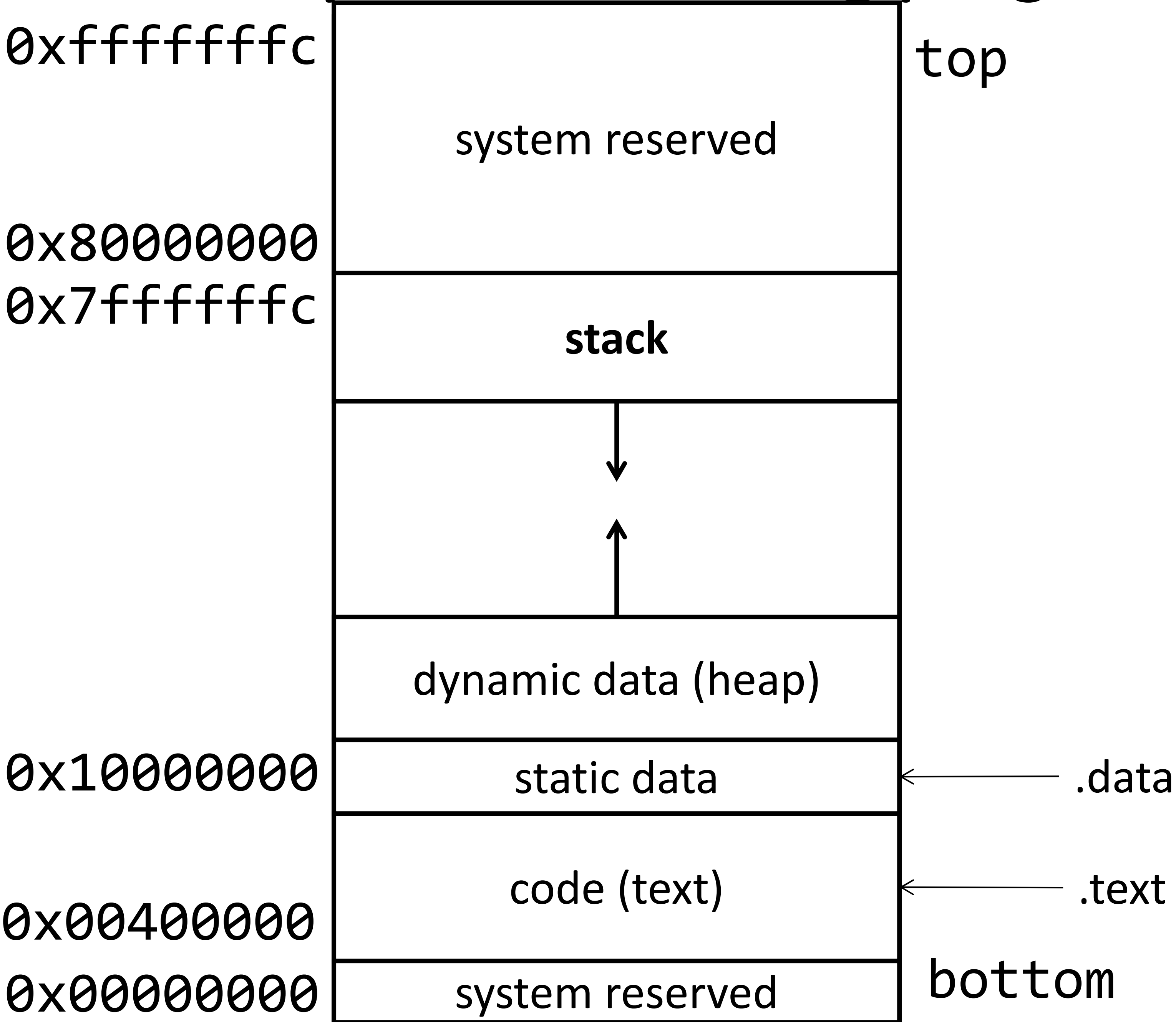
Warning: There is no one true MIPS calling convention.
lecture != book != gcc != spim != web

MIPS Register Conventions

r0	$zero	zero
r1	$at	assembler temp
r2	$v0	function return values
r3	$v1	
r4	$a0	function arguments
r5	$a1	
r6	$a2	
r7	$a3	
r8	$t0	**temps** **(caller save)**
r9	$t1	
r10	$t2	
r11	$t3	
r12	$t4	
r13	$t5	
r14	$t6	
r15	$t7	

r16	$s0	**saved** **(callee save)**
r17	$s1	
r18	$s2	
r19	$s3	
r20	$s4	
r21	$s5	
r22	$s6	
r23	$s7	
r24	$t8	**more temps** **(caller save)**
r25	$t9	
r26	$k0	reserved for kernel
r27	$k1	
r28	$gp	global data pointer
r29	$sp	stack pointer
r30	$fp	frame pointer
r31	$ra	return address

Anatomy of an executing program



0xfffffffc top

system reserved

0x80000000

0x7ffffffc

stack

dynamic data (heap)

0x10000000

static data ← .data

code (text) ← .text

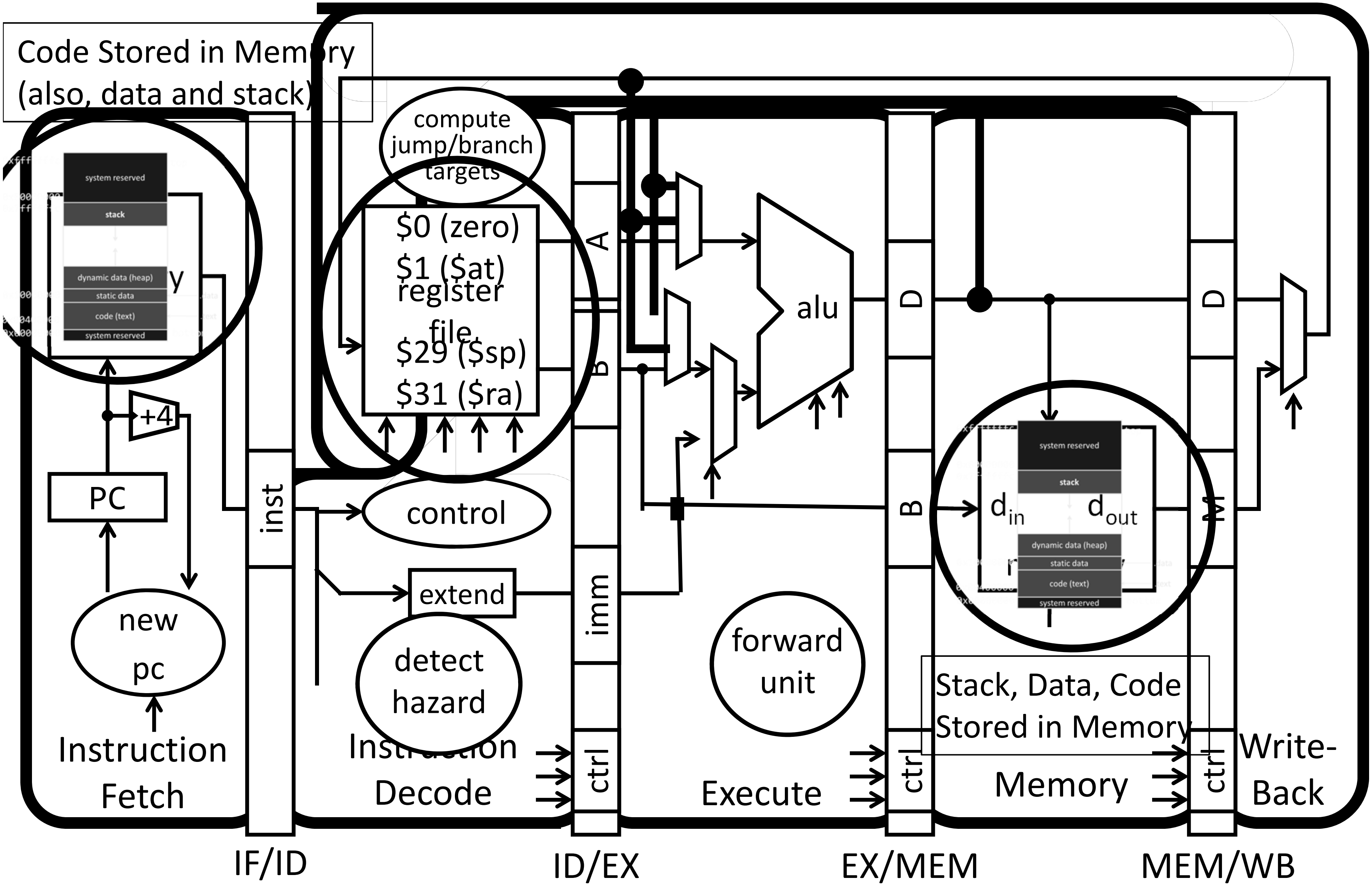
0x00400000

bottom

0x00000000

system reserved

Anatomy of an executing program



Code Stored in Memory
(also, data and stack)

Stack, Data, Code
Stored in Memory

Takeaway

We need a calling convention to coordinate use of registers and memory. Registers exist in the Register File. Stack, Code, and Data exist in memory. Both instruction memory and data memory accessed through cache (modified harvard architecture) and a shared bus to memory (Von Neumann).

Compilers and Assemblers

Next Goal

How do we compile a program from source to assembly to machine object code?

Big Picture

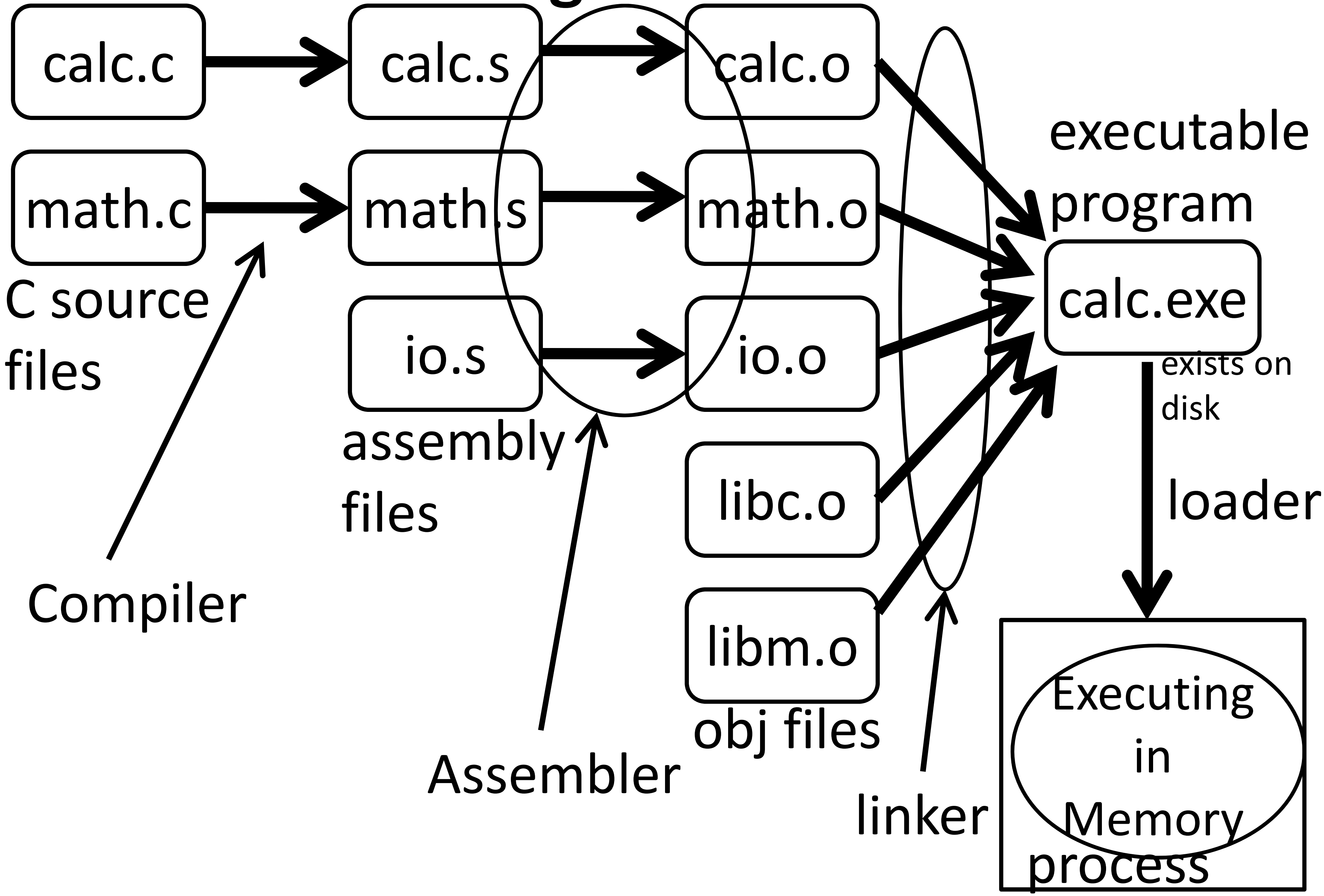
Compiler output is assembly files

Assembler output is obj files

Linker joins object files into one executable

Loader brings it into memory and starts execution

Big Picture


calc.c
math.c
C source files
Compiler
calc.s
math.s
io.s
assembly files
Assembler
calc.o
math.o
io.o
libc.o
libm.o
obj files
linker
executable program
calc.exe
exists on disk
loader
Executing in Memory
process

Next Goal

How do we (as humans or compiler) program on top of a given ISA?

Assembler

Translates text *assembly language* to binary machine code

Input: a text file containing MIPS instructions in human readable form

```
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```

Output: an object file (.o file in Unix, .obj in Windows) containing MIPS instructions in executable form

```
00100000000001010000000000001010
00000000000001010010100001000000
00100000101001010000000000001111
```

Assembly Language

Assembly language is used to specify programs at a low-level

Will I program in assembly?

A: I do...

- For CS 3410 (and some CS 4410/4411)
- For kernel hacking, device drivers, GPU, etc.
- For performance (but compilers are getting better)
- For highly time critical sections
- For hardware without high level languages
- For new & advanced instructions: rdtsc, debug registers, performance counters, synchronization, ...

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)

Assembler

Assembler:

Input:

assembly instructions

+ psuedo-instructions

+ data and layout directives

Output:

Object file

Slightly higher level than plain assembly

e.g: takes care of delay slots

(will reorder instructions or insert nops)

Assembler

Assembler:

Input:

assembly instructions

+ psuedo-instructions

+ data and layout directives

Output:

Object File

Slightly higher level than plain assembly

e.g: takes care of delay slots

(will reorder instructions or insert nops)

MIPS Assembly Language Instructions

Arithmetic/Logical

- 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

Memory Access

- LW, LH, LB, LHU, LBU, LWL, LWR
- SW, SH, SB, SWL, SWR

Control flow

- BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
- J, JR, JAL, JALR, BEQL, BNEL, BLEZL, BGTZL

Special

- LL, SC, SYSCALL, BREAK, SYNC, COPROC

Assembler

Assembler:

Input:

assembly instructions

+ psuedo-instructions

+ data and layout directives

Output:

Object file

Slightly higher level than plain assembly

e.g: takes care of delay slots

(will reorder instructions or insert nops)

Pseudo-Instructions

Pseudo-Instructions

NOP # do nothing

- SLL r0, r0, 0

MOVE reg, reg # copy between regs

- ADD R2, R0, R1 # copies contents of R1 to R2

LI reg, imm # load immediate (up to 32 bits)

LA reg, label # load address (32 bits)

B label # unconditional branch

BLT reg, reg, label # branch less than

- SLT r1, rA, rB # r1 = 1 if R[rA] < R[rB]; o.w. r1 = 0
- BNE r1, r0, label # go to address label if r1!=r0; i.t. rA < rB

Assembler

Assembler:

Input:

assembly instructions

+ psuedo-instructions

+ data and layout directives

Output:

Object file

Slightly higher level than plain assembly

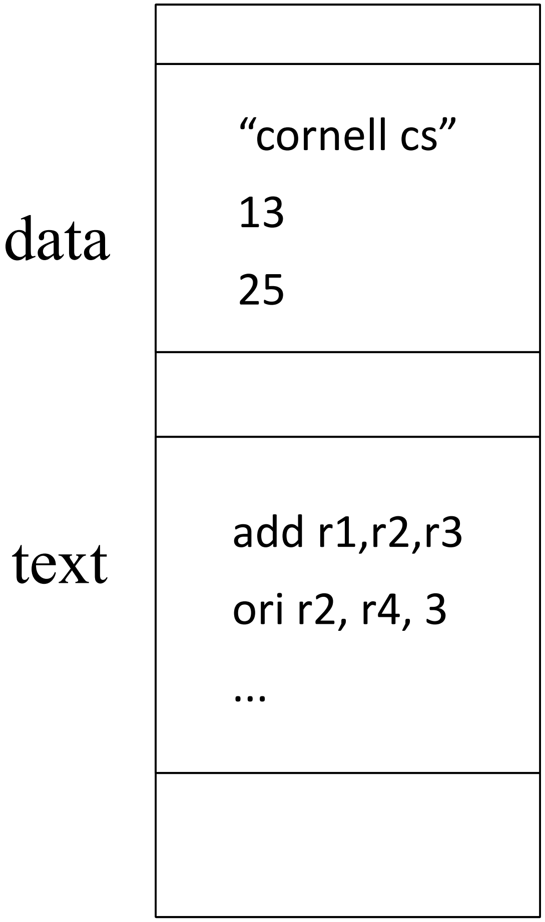
e.g: takes care of delay slots

(will reorder instructions or insert nops)

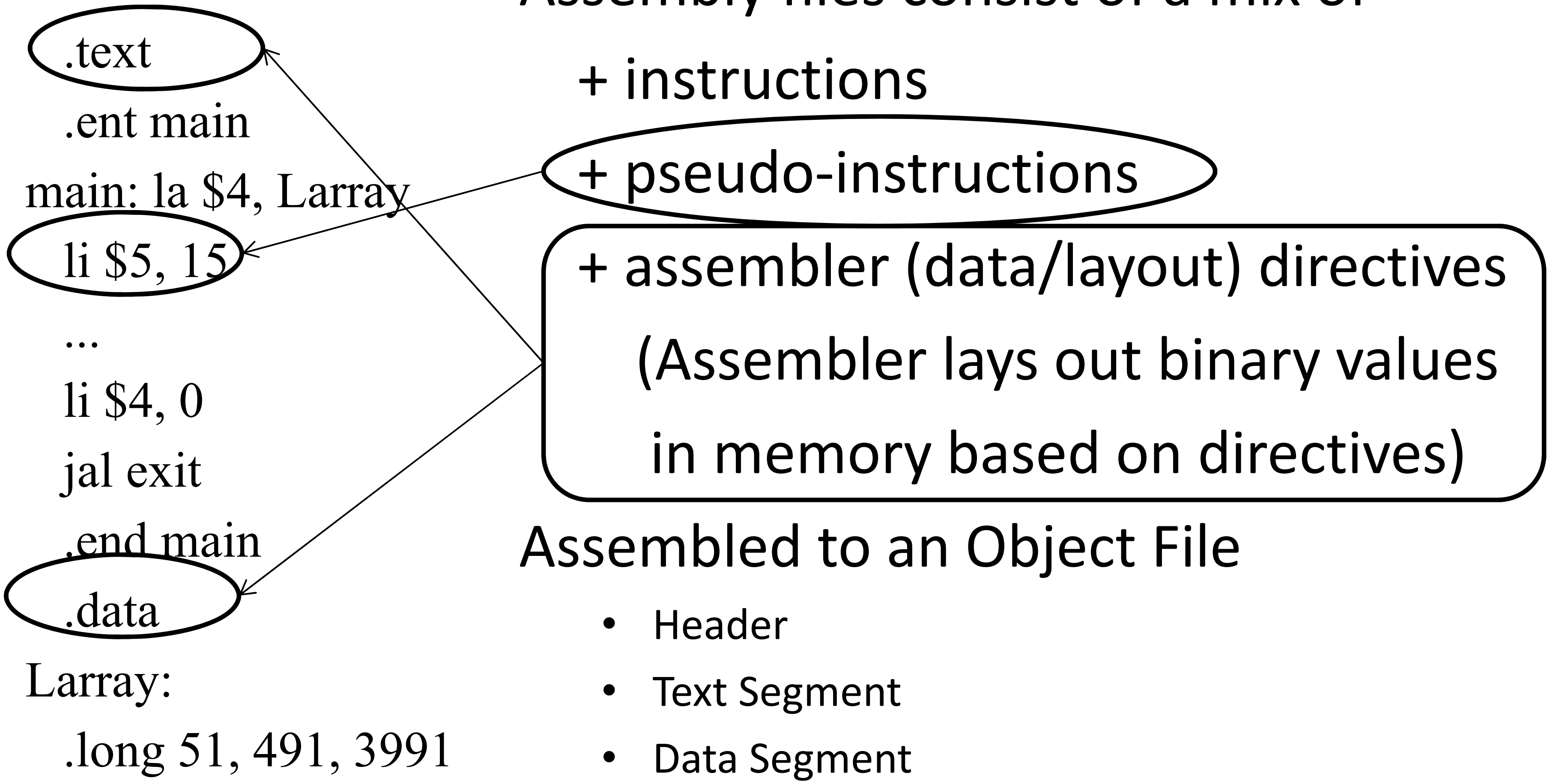
Program Layout

Programs consist of segments used for different purposes

- Text: holds instructions
- Data: holds statically allocated program data such as variables, strings, etc.



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
```

Assembly files consist of a mix of

+ instructions

+ pseudo-instructions

+ assembler (data/layout) directives

(Assembler lays out binary values in memory based on directives)

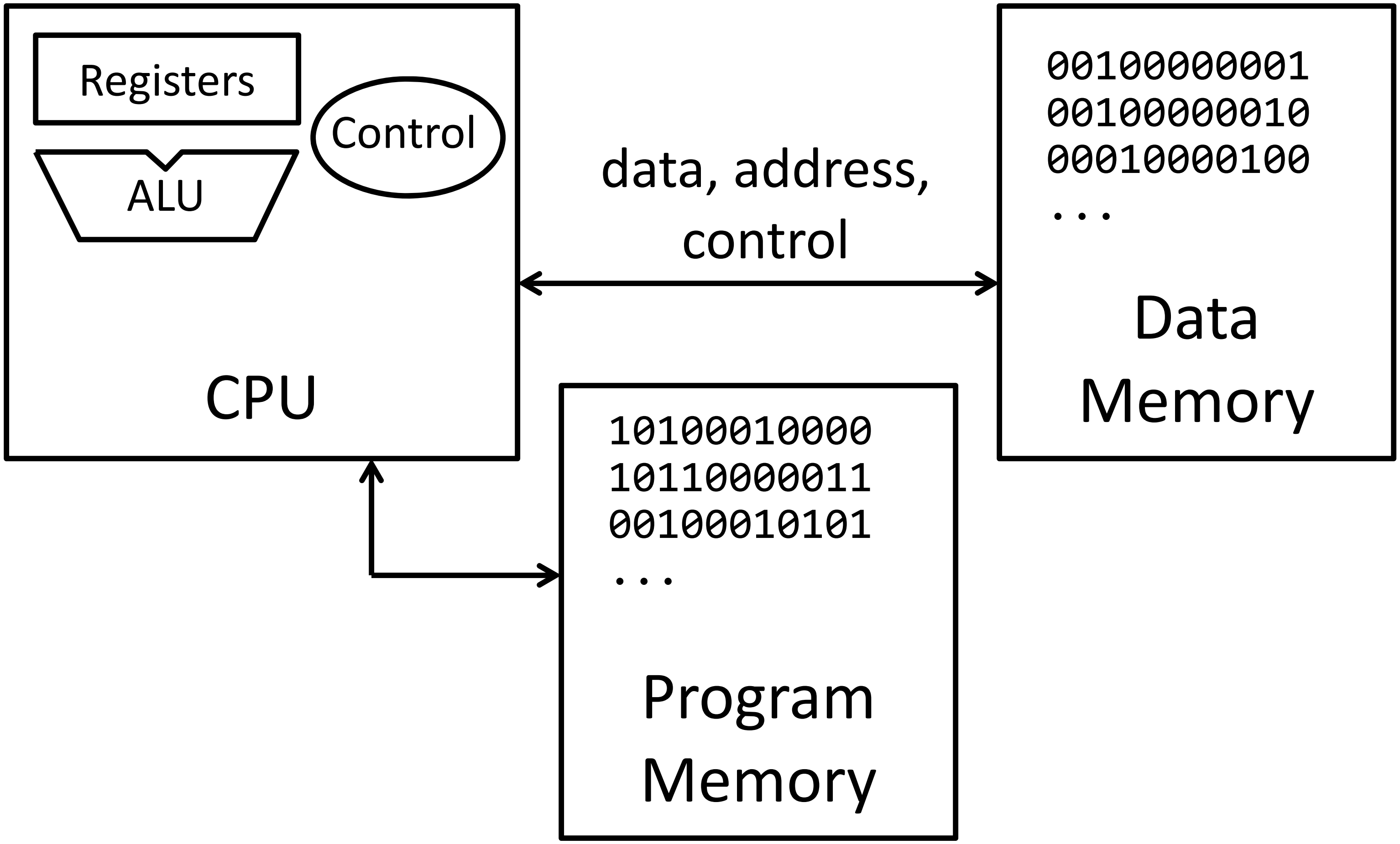
Assembled to an Object File

- Header
- Text Segment
- Data Segment
- Relocation Information
- Symbol Table
- Debugging Information

Assembling Programs

Assembly with a but using (modified) Harvard architecture

- Need segments since data and program stored together in memory



Takeaway

Assembly is a low-level task

- Need to assemble assembly language into machine code binary. Requires
  - Assembly language instructions
  - *pseudo-instructions*
  - And Specify layout and data using *assembler directives*

- Today, we use a modified Harvard Architecture (Von Neumann architecture) that mixes data and instructions in memory

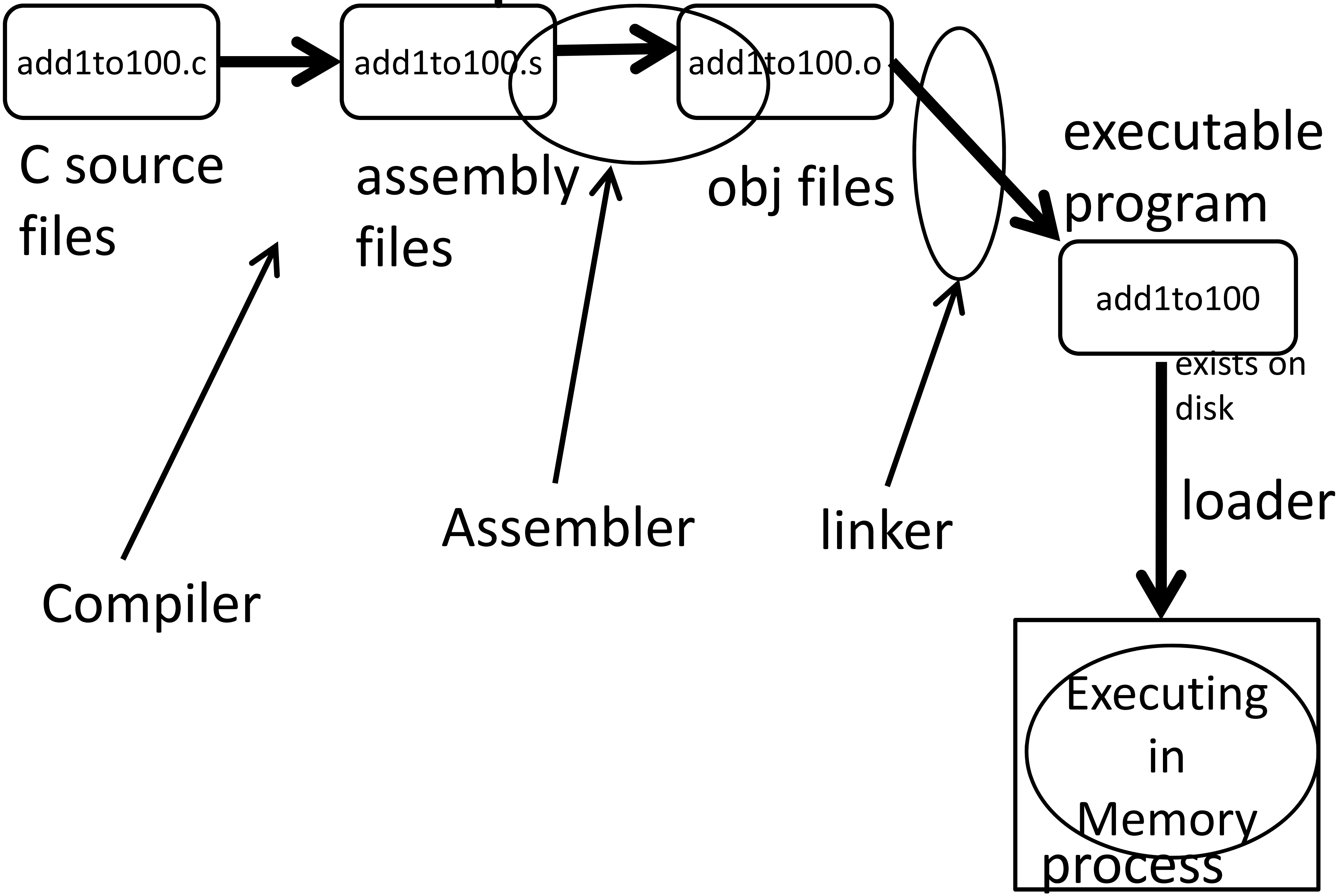
  … but kept in separate *segments*

  … and has separate caches

Next Goal

Put it all together: An example of compiling a program from source to assembly to machine object code.

Example: Add 1 to 100



add1to100.c

C source files

add1to100.s

assembly files

add1to100.o

obj files

executable program

add1to100

exists on disk

loader

Executing in Memory

process

Compiler

Assembler

linker

Example: Add 1 to 100

```
int n = 100;

int main (int argc, char* argv[ ]) {
                int i;
                int m = n;
                int sum = 0;

                for (i = 1; i <= m; i++)
                    sum += i;

                printf ("Sum 1 to %d is %d\n", n, sum);
}
```

```
export PATH=${PATH}:/courses/cs3410/mipsel-linux/bin:/courses/cs3410/mips-sim/bin
or
setenv PATH ${PATH}:/courses/cs3410/mipsel-linux/bin:/courses/cs3410/mips-sim/bin
```

```
# Compile
[csug03] mipsel-linux-gcc –S add1To100.c
```

Example: Add 1 to 100

```
        .data
        .globl  n
        .align  2
n:      .word   100
        .rdata
        .align  2
$str0: .asciiz
           "Sum 1 to %d is %d\n"
        .text
        .align  2
        .globl  main
main:   addiu   $sp,$sp,-48
        sw      $31,44($sp)
        sw      $fp,40($sp)
        move    $fp,$sp
        sw      $4,48($fp)
        sw      $5,52($fp)
        la      $2,n
        lw      $2,0($2)
        sw      $2,28($fp)
        sw      $0,32($fp)
        li      $2,1
        sw      $2,24($fp)
```

```
$L2:    lw      $2,24($fp)
        lw      $3,28($fp)
        slt     $2,$3,$2
        bne     $2,$0,$L3
        lw      $3,32($fp)
        lw      $2,24($fp)
        addu    $2,$3,$2
        sw      $2,32($fp)
        lw      $2,24($fp)
        addiu   $2,$2,1
        sw      $2,24($fp)
        b       $L2
$L3:    la      $4,$str0
        lw      $5,28($fp)
        lw      $6,32($fp)
        jal     printf
        move    $sp,$fp
        lw      $31,44($sp)
        lw      $fp,40($sp)
        addiu   $sp,$sp,48
        j       $31
```

Example: Add 1 to 100

```
          .data
          .globl    n√
          .align    2
n:        .word     100
          .rdata
          .align    2
$str0:    .asciiz
             "Sum 1 to %d is %d\n"
          .text
          .align    2
          .globl    main√
main:     addiu          $sp,$sp,-48
          sw             $31,44($sp)
prologue  sw             $fp,40($sp)
          move           $fp,$sp
          sw        $a0  $4,48($fp)
          sw        $a1  $5,52($fp)
          la        $v0  $2,n
          lw             $2,0($2)  $v0=100
          sw             $2,28($fp) m=100
          sw             $0,32($fp) sum=0
          li             $2,1
          sw             $2,24($fp) i=1
```

```
$L2:      lw        $v0  $2,24($fp) i=1
          lw        $v1  $3,28($fp) m=100
          slt            $2,$3,$2  if(m < i)
          bne            $2,$0,$L3 100 < 1
          lw             $3,32($fp) v1=0(sum)
          lw             $2,24($fp) v0=1(i)
          addu           $2,$3,$2 v0=1(0+1)
          sw             $2,32($fp) sum=1
          lw             $2,24($fp) i=1
          addiu          $2,$2,1  i=2 (1+1)
          sw             $2,24($fp)  i=2
          b              $L2
$L3:      la        $a0  $4,$str0   str
  printf  lw        $a1  $5,28($fp) m=100
          lw        $a2  $6,32($fp) sum
          jal            printf
          move           $sp,$fp
epilogue  lw             $31,44($sp)
          lw             $fp,40($sp)
          addiu          $sp,$sp,48
          j              $31
```

Example: Add 1 to 100

```
# Assemble
[csug01] mipsel-linux-gcc –c add1To100.s

# Link
[csug01] mipsel-linux-gcc –o add1To100 add1To100.o
${LINKFLAGS}
# -nostartfiles –nodefaultlibs
# -static -mno-xgot -mno-embedded-pic
-mno-abicalls -G 0 -DMIPS -Wall

# Load
[csug01] simulate add1To100
Sum 1 to 100 is 5050
MIPS program exits with status 0 (approx. 2007
instructions in 143000 nsec at 14.14034 MHz)
```

Globals and Locals

Variables	Visibility	Lifetime	Location
Function-Local i, m, sum	w/in func	func invocation	stack
Global n, str	whole prgm	prgm execution	.data
Dynamic A	? Anywhere that has a ptr	b/w malloc and free	heap

```
int n = 100;
int main (int argc, char* argv[ ]) {
        int i, m = n, sum = 0;  int* A = malloc(4*m + 4);
        for (i = 1; i <= m; i++) { sum += i; A[i] = sum; }
        printf ("Sum 1 to %d is %d\n", n, sum);
}
```

Globals and Locals

Variables	Visibility	Lifetime	Location
Function-Local i, m, sum	w/in func	func invocation	stack
Global n, str	whole prgm	prgm execution	.data
Dynamic A **C Pointers can be trouble**	Anywhere that has a ptr	b/w malloc and free	heap

Globals and Locals

Variables	Visibility	Lifetime	Location
Function-Local i, m, sum	w/in func	func invocation	stack
Global n, str	whole prgm	prgm execution	.data
Dynamic A	Anywhere that has a ptr	b/w malloc and free	heap

C Pointers can be trouble

```
int *trouble()
{ int a; …; return &a; }
```

“addr of” something on the stack!

Invalid after return

```
char *evil()
{ char s[20]; gets(s); return s; }
```

Buffer overflow

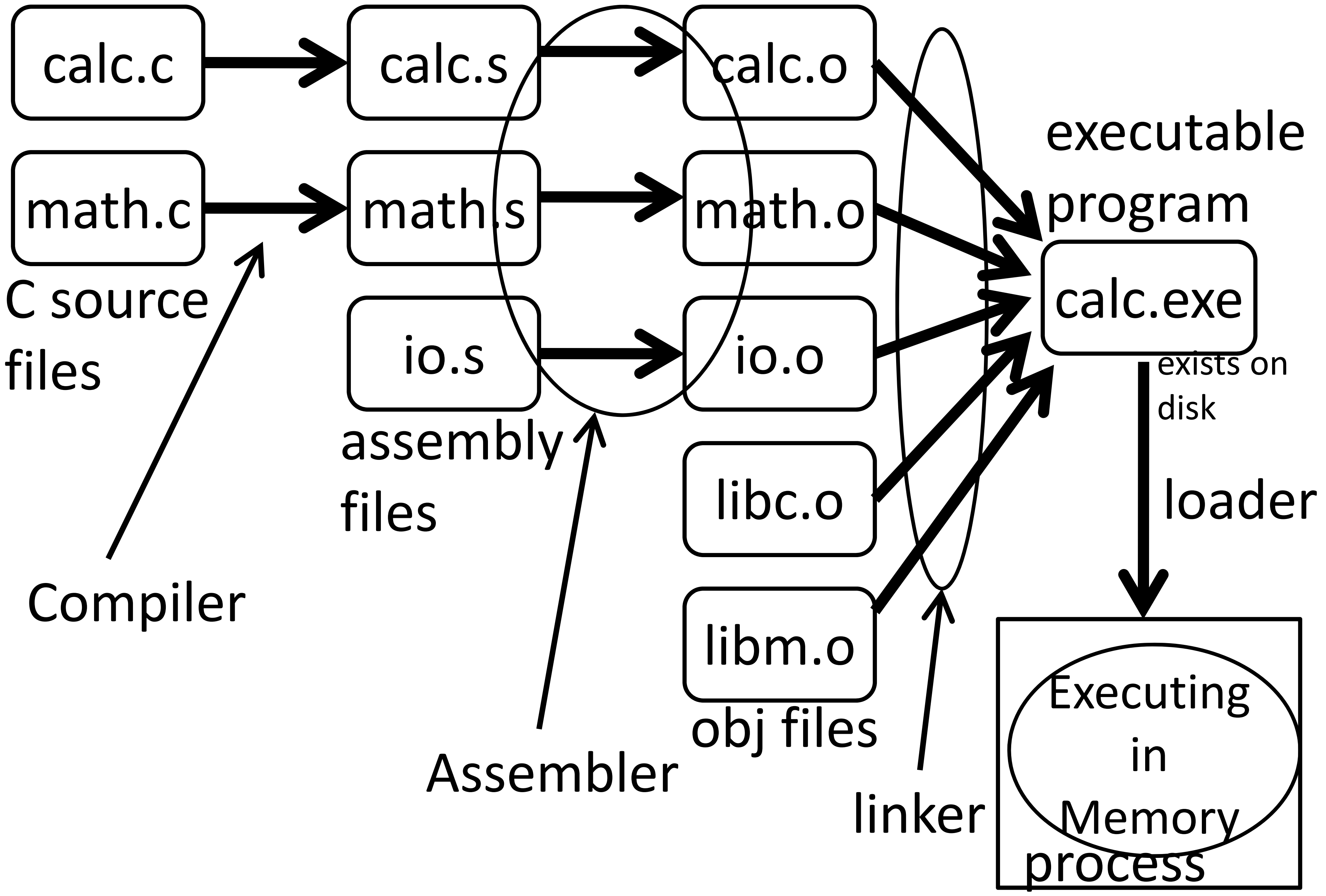
```
int *bad()
{ s = malloc(20); … free(s); … return s; }
```

Allocated on the heap

But freed (i.e. a dangling ptr)

(Can’t do this in Java, C#, ...)

Example #2: Review of Program Layout



Example #2: Review of Program Layout

calc.c

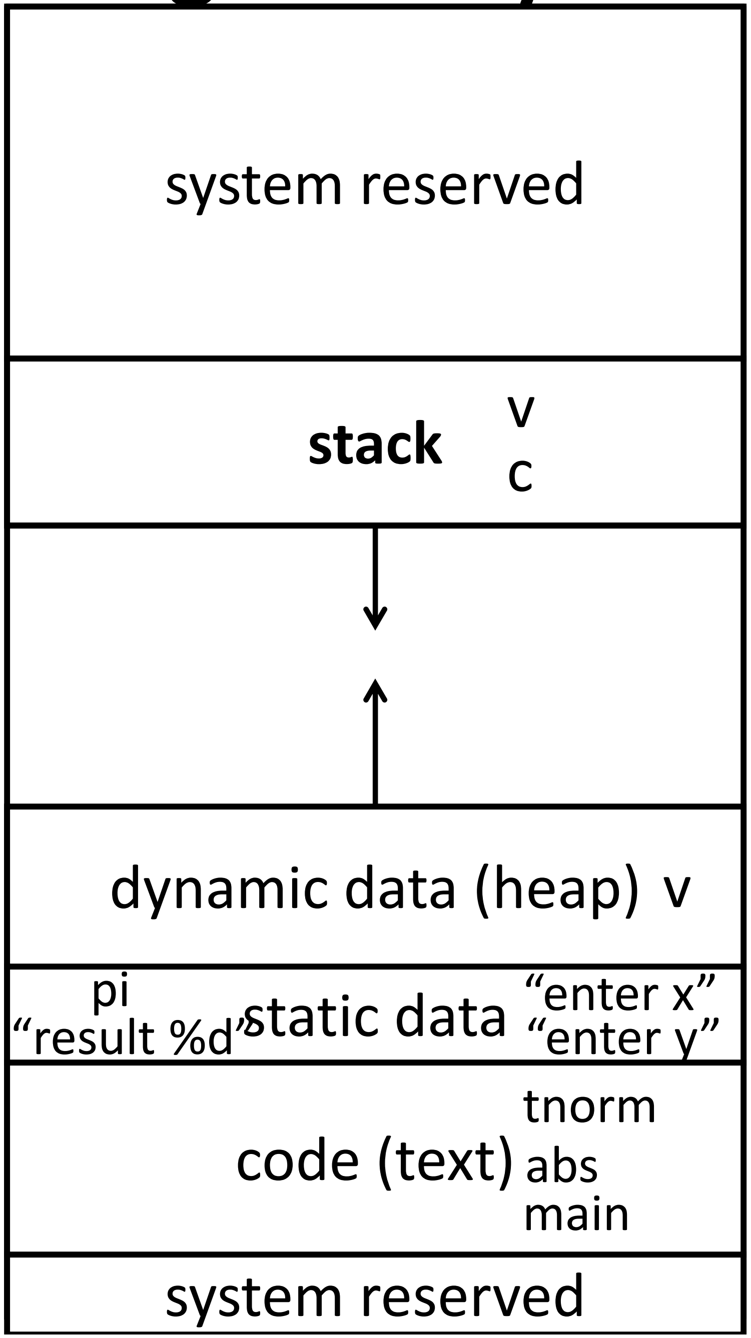
```
vector* v = malloc(8);
v->x = prompt(“enter x”);
v->y = prompt(“enter y”);
int c = pi + tnorm(v);
print(“result %d”, c);
```

math.c

```
int tnorm(vector* v) {
 return abs(v->x)+abs(v->y);
}
```

lib3410.o

```
  global variable: pi
  entry point: prompt
  entry point: print
  entry point: malloc
```



system reserved

stack v c

dynamic data (heap) v

pi “result %d” static data “enter x” “enter y”

code (text) tnorm abs main

system reserved

Takeaway

Compiller produces assembly files

- (contain MIPS assembly, pseudo-instructions, directives, etc.)

Assembler produces object files

- (contain MIPS machine code, missing symbols, some layout information, etc.)

Linker produces executable file

- (contains MIPS machine code, no missing symbols, some layout information)

Loader puts program into memory and jumps to first instruction

- (machine code)

Recap

Compiler output is assembly files

Assembler output is obj files

Next Time

Linker joins object files into one executable

Loader brings it into memory and starts execution

Administrivia

Upcoming agenda

- PA1 due two days ago

- PA2 available and discussed during lab section this week
- PA2 Work-in-Progress due Monday, March 17th
- PA2 due Thursday, March 27th

- HW2 available next week, due before Prelim2 in April

- Spring break: Saturday, March 29th to Sunday, April 6th