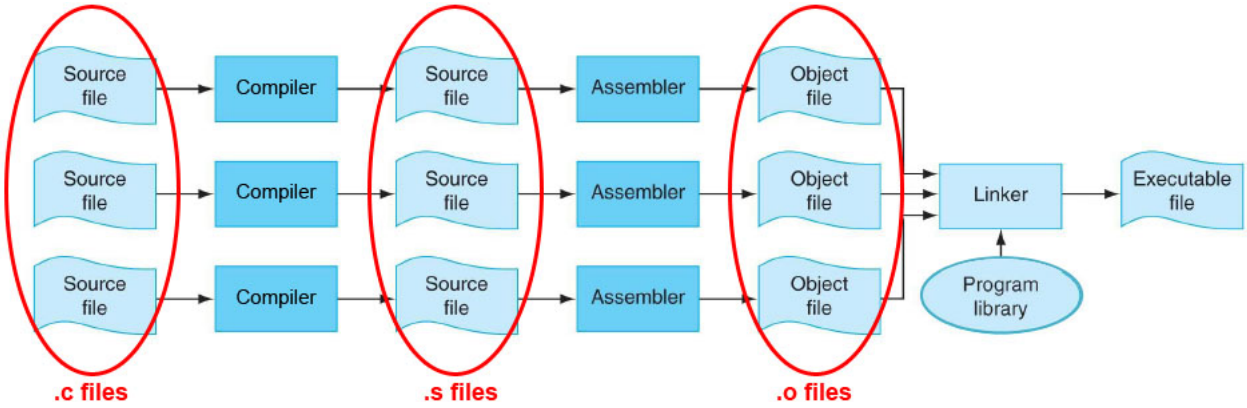
Assembling and Linking



Assembler: Converts pseudoinstructions to MIPS instructions. Uses the `$at` register. Converts assembly language into an object file containing:

1) Header – size and position of other parts
2) Text segment – machine language code
3) Static data segment
4) Relocation information – instructions and data words using absolute addressing
5) Symbol table – matches symbols/labels with addresses
6) Debugging information

Linker: Combines independently assembled machine language programs and resolves all remaining undefined labels from the relocation information and symbol table. Result is the executable file.

Review Questions:

How many passes over assembly code does an assembler have to make and why?

The linker resolves issues in relative or absolute addressing?

What does RISC stand for? How is this related to pseudoinstructions?

MIPS Addressing

As a quick reminder, the program counter register (`$PC`) stores the address of the instruction being executed (because code also sits in memory). This register cannot be accessed directly.

Here we reiterate the difference between *relative* addressing and *absolute* addressing. An instruction that uses absolute addressing, such as `j`, stores directly into `$PC`. An instruction that uses relative addressing, such as `beq`, uses the `immediate` as an offset to **`$PC+4`**.

Addresses are byte-addressed, meaning the addresses of neighboring words in memory are different by 4 (addresses always end with `0b00` in binary). Branching and jump statements take this into account by automatically multiplying addresses by 4 (shifting left by 2):

```
$PC = ($PC+4) + 4*immediate
```

Assuming the first instruction is at address 0x000 (which is really 0x00000), fill in the fields below. Use decimal for immediate and hex for addresses.

```
0x000 beq  $s0, $0,  Ret0          imm = _____
      addi $t0, $0,  1
      beq  $s0, $t0, Ret1          imm = _____
      subi $s0, $s0, 2
      addi $s1, $0,  1
      addi $t0, $0,  1
      addi $t1, $0,  1
Loop: beq  $s0, $0,  RetF          imm = _____
      add $s1, $t0, $t1
      addi $t0, $t1, 0
      addi $t1, $s1, 0
      subi $s0, $s0, 1
      j    Loop                    addr = _____
Ret0: addi $v0, $0,  0
      j    Done                    addr = _____
Ret1: addi $v0, $0,  1
      j    Done                    addr = _____
RetF: add  $v0, $0,  $s1
Done: jr   $ra
```

Which instruction can reach more addresses, `j` or `jr`?

What is the maximum jump distance of `j` and `jr`?

MIPS Calling Conventions

When calling a function in MIPS, who needs to save the following variables to the stack? Answer **R** for the caller, **E** for the callee, or **N** for neither.

`$v0-$v1`	`$a0-$a3`	`$t0-$t9`	`$s0-$s7`	`$sp`	`$ra`

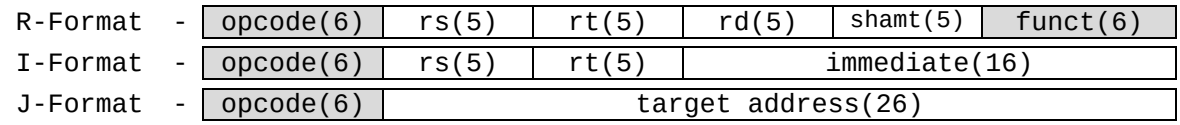
Now assume our function `foo` calls another function `bar`, which is known to call other functions. `foo` takes one argument and uses `$t0` and `$s0`. `bar` takes two arguments, returns an integer, and uses `$t0-$t1` and `$s0-$s1`.

In the boxes below, draw a possible ordering of the stack just before `bar` calls a function (you may not need all the spaces). The top of the left box is the address of `$sp` when `foo` is first called and the top of the right box follows directly after the bottom of the left box. Add "(f)" if the register is stored by `foo` and "(b)" if the register is stored by `bar`. The first one is written in for you.

`$ra (f)`

MIPS Instruction Formats Reference

Instructions are represented as bits, same as everything else! All instructions fit in a word (32 bits). In order to cover all the different instructions, there are 3 different instruction types:



Instruction Conversion Practice

opcode	funct	Instruction
0	33	addu
8	--	addi
4	--	beq
5	--	bne
35	--	lw
43	--	sw
0	42	slt
10	--	slti
2	--	j
0	0	sll

Register Name	Register Num
$zero	0
$v0-$v1	2-3
$a0-$a3	4-7
$t0-$t7	8-15
$s0-$s7	16-23
$t8-$t9	24-25
$sp	29
$ra	31

Convert the following MIPS code into instruction format (use decimal and only convert to binary if you have time). The box for opcode is shown, demarcate the other fields on your own. Fill unused fields with zeros.

```
# $s0 -> int * (address)
# $a0 -> int
0x000  addi $v0, $0,  0
Loop:  slt  $t0, $v0, $a0
       beq  $t0, $0,  Done
       sll  $t1, $v0, 2
       addu $t2, $s0, $t1
       sw   $t0, 4($t2)
       addi $v0, $v0, 1
       j    Loop
Done:  # done!
```