Assembling and Linking (credits to Justin Hsia)

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

Two. First to find all label addresses, second for instruction conversion.

The linker resolves issues in relative or absolute addressing?

Absolute addressing.

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

Reduced Instruction Set Computing. Minimal set of instructions leads to many lines of code. Pseudoinstructions are more complex instructions intended to make assembly programming easier for the coder. These are converted to TAL by the assembler.

MIPS Addressing

Assuming the first instruction is at address 0x000 (which is really 0b 00 0000 0000 0000), fill in the fields below. Use decimal for immediate and hex for addresses.

`0x000 beq $s0, $0, Ret0`	`imm = _12__`
`addi $t0, $0, 1`	
`beq $s0, $t0, Ret1`	`imm = _12__`
`subi $s0, $s0, 2`	
`addi $s1, $0, 1`	
`addi $t0, $0, 1`	
`addi $t1, $0, 1`	
`Loop: beq $s0, $0, RetF`	`imm = __9__`
`add $s1, $t0, $t1`	
`addi $t0, $t1, 0`	
`addi $t1, $s1, 0`	
`subi $s0, $s0, 1`	
`j Loop`	`addr = _0x007____`
`Ret0: addi $v0, $0, 0`	
`j Done`	`addr = _0x012____`
`Ret1: addi $v0, $0, 1`	
`j Done`	`addr = _0x012____`
`RetF: add $v0, $0, $s1`	
`Done: jr $ra`	

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

`jr` can reach all 2^{32} addresses while `j` can only reach $2^{26+2} = 2^{28}$.

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

`j`: immediate is 0x000000 to 0x3fffff, so $2^{26} - 1$ words = $2^{28} - 4$ bytes.

`jr`: 0x0000 0000 to 0xffff fffc, so $2^{32} - 4$ bytes = $2^{30} - 1$ instructions.

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 _R_  $a0-$a3 _R_  $t0-$t9 _R_  $s0-$s7 _E_  $sp _N_  $ra _E_`

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 box, draw a possible ordering of the stack just before `bar` calls a function (you may not need all the spaces). Add "(f)" if the register is stored by `foo` and "(b)" if the register is stored by `bar`.

$ra (f)
$s0 (f)
$v0 (f)
$a0 (f)
$t0 (f)
$ra (b)
$s0 (b)
$s1 (b)

$v0 (b)
$a0 (b)
$a1 (b)
$t0 (b)
$t1 (b)

Instruction Conversion Practice

```
# $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!
```

Line 1:	8 00 1000	0 0 0000	2 0 0010	0 0000 0000 0000 0000		
Line 2:	0 00 0000	2 0 0010	4 0 0100	8 0 1000	0 0 0000	42 10 1010
Line 3:	4 00 0100	8 0 1000	0 0 0000	5 0000 0000 0000 0101		
Line 4:	0 00 0000	0 0 0000	2 0 0010	9 0 1001	2 0 0010	0 00 0000
Line 5:	0 00 0000	16 1 0000	9 0 1001	10 0 1010	0 0 0000	33 10 0001
Line 6:	43 10 1011	10 0 1010	8 0 1000	4 0000 0000 0000 0100		
Line 7:	8 00 1000	2 0 0010	2 0 0010	1 0000 0000 0000 0001		
Line 8:	2 00 0010	1 00 0000 0000 0000 0000 0000 0001				