2. Loops and Inequalities

slt $a, $b, $c is equal to $a = 1 if $b < $c

```
for (i = 0; i < j; i++)
{
      a += i;
}

// i = $a0, j = $a1, a = $v0

i = 0;
do
{
      a += i;
      i--;
} while (i >= j)
```

```
      add $a0, $0, $0
Loop: slt $t0, $a0, $a1
      beq $t0, $0, End
      add $v0, $v0, $a0
      addiu $a0, $a0, 1
      j Loop
End:

      add $a0, $0, $0
Loop: add $v0, $v0, $a0
      addi $a0, $a0, -1
      slt $t0, $a0, $a1
      beq $t0, $0, Loop
```

While loop is very similar to for loop in MIPS. Notice that the do-while loop is shorter because it doesn't the extra jump at the end to reiterate; checks are done at the end of each cycle of the loop.

3. >=, <=, >, < Comparisons

Inequality	C	MIPS

>	`if ($s0 > $s1)`
	`  goto a;`
	`else  goto b;`
		`slt $t0, $s1, $s0`
		`bne $t0, $0, a`
		`j b`

<	`if ($s0 < $s1)`
	`  goto a;`
	`else  goto b;`
		`slt $t0, $s0, $s1`
		`bne $t0, $0, a`
		`j b`

>=	`if ($s0 >= $s1)`
	`  goto a;`
	`else  goto b;`
		`slt $t0, $s0, $s1`
		`beq $t0, $0, a`
		`j b`

<=	`if ($s0 <= $s1)`
	`  goto a;`
	`else  goto b;`
		`slt $t0, $s1, $s0`
		`beq $t0, $0, a`
		`j b`

Various permutations of slt and bne/beq allow us to represent all the inequalities with only three commands. Simple hardware means the onus is on the programmer/compiler to generate correct code.

Other variants of slt include slti, sltu, and sltiu.

4. Functions and jal

To pass parameters to a function, copy values or registers into $a0-$a3. Anything beyond that is stored on the stack.
To call a function, use jal to go to the location in memory of the function code. Jal (jump and link)

stores the address of the next place in memory in $ra and moves to the label that you specify.

To get back, we need to jr (jump register) to the stored address in $ra.

Return values from a register are found in $v0-$v1 or the stack.

5. Nested Functions and storing variables on the stack

After one jal, what's to prevent the nested function from overwriting $ra and the input registers? Nothing! The compiler or the programmer must store $ra and all parameters passed into the the function on the stack before calling the nested function. That nested function must also restore $sp to what it was before the nested function was called and cannot change $s0-$s7. $a0-$a3, $t0-$t9, and $v0-$v1 can (and usually will) be overwritten after calling a function.

6. Solved Problems

True/False:

1. Functions must always store $ra and $a0-$a3 on the stack if it calls a nested function that will change those registers.
2. Decrementing the stack pointer's value means moving “forward” to allocate free space and incrementing means moving “backwards” and extracting stored values.
3. C translates rightward shifts (>>) as srl.
4.

Translate the following C into MIPS:

```
// assume ptr is the address of a0[4]
// num → $s0, num2 → $s1, dest → $s2

int num = *(ptr);
char num2 = (char) (ptr[8] >> 24);
if (num == num2)    dest = num + num2;
else                dest = num - num2;
```

```
lw $s0, 16($a0)
lbu $s1, 48($a0)
beq $s0, $s1, equal
subu $s2, $s0, $s1
j end
equal: addu $s2, $s0, $s1
end:
```

```
// Nth_Fibonacci(N)
// N → $s0, fib → $s1
// i → $t0, j → $t1
if(N==0) return 0;
else if(N==1) return 1;

N-=2;
int fib=1, i=1, j=1;
while(N!=0)
{
      fib = i+j;
      j = i;
      i = fib;
      N--;
}
return fib;
```

```
zero: bne $s0 $0 one
      addi $s1 $0 0
      j done
one:  addi $s1 $0 1
      bne $s0 $s1 init
      j done
init: addi $s0 $s0 -2
      addi $t0 $0 1
      addi $t1 $0 1
loop: beq $s0 $0 done
      add $s1 $t0 $t1
      addi $t1 $t0 0
      addi $t0 $s1 0
      subi $s0 $s0 1
      j loop
done:
```

7. More Problems (solutions will be on *next* discussion's handout)