MIPS Control Flow

1.) Translate the following abstract branch conditions to minimal MIPS instruction sequences.

- Branch to `exit` if `$s0 < $s1`

```
slt   $t0, $s0, $s1
bne   $t0, $zero, exit
```

- Branch to `next` if `$s0 < 1`

```
slti  $t0, $s0, 1
bne   $t0, $zero, next
```

- Branch to `done` if `$s0 <= $s1`

```
slt   $t0, $s1, $s0
beq   $t0, $zero, done
```

- Branch to `finish` if `$s0 >= 0`

```
slti  $t0, $s0, 0
beq   $t0, $zero, finish
```

2.) Translate the following `strcpy` implementation to MIPS. Assume `$s1` holds the address of the first byte of the array `s1` and `$s2` holds the contents of the pointer variable `s2`.

```
// strcpy:
// char s1[] = Hello!;
// char *s2 = malloc(sizeof(char)*7);
int i=0;
do {
    s2[i] = s1[i];
    i++;
} while (s1[i] != '\0');
s2[i] = '\0';
```

```
      addi $t0, $0,  0
Loop: add  $t1, $s1, $t0  # s1[i]
      add  $t2, $s2, $t0  # s2[i]
      lb   $t3, 0($t1)    # char is
      sb   $t3, 0($t2)    #   1 byte!
      addi $t0, $t0, 1
      addi $t1, $t1, 1
      lb   $t4, 0($t1)
      beq  $t4, $0, Done
      j    Loop
Done: addi $t2, $t2, 1
      sb   $t4, 0($t2)
```

3.) Translate the following algorithm that finds the N^{th} Fibonacci number to MIPS assembly. Assume `$s0` holds `N`, `$s1` holds `fib`, `$t0` holds `i`, and `$t1` holds `j`.

```
int fib = 1, i = 1, j = 1;

if (N==0)      return 0;
else if (N==1) return 1;
N -= 2;

while (N != 0) {
    fib = i + j;
    j = i;
    i = fib;
    N--;
}

return fib;
```

```
      beq  $s0, $0,  Ret0
      addi $t2, $0,  1
      beq  $s0, $t2, Ret1
      subi $s0, $s0, 2
Loop: beq  $s0, $0,  RetF
      add $s1, $t0, $t1
      addi $t0, $t1, 0
      addi $t1, $s1, 0
      subi $s0, $s0, 1
      j    Loop
Ret0: addi $v0, $0,  0
      j    Done
Ret1: addi $v0, $0,  1
      j    Done
RetF: add  $v0, $0,  $s1
Done: jr   $ra
```

4.) Assuming `$a0` and `$a1` hold integer pointers, swap the values they point via the stack and return control.

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
addiu $sp, $sp, -4
lw    $t0, 0($a0)
sw    $t0, 0($sp)
lw    $t0, 0($a1)
sw    $t0, 0($a0)
lw    $t0, 0($sp)
sw    $t0, 0($a1)
addiu $sp, $sp, 4
jr    $ra
```

MIPS Pseudoinstructions

MIPS assemblers recognize several *pseudoinstructions* – assembly code statements that are not actually part of the MIPS ISA – that get converted to short seqeunces of instructions. For each of the following pseudoinstructions, give the shortest possible instruction sequence you can find. Note that the register `$at` ("assembler temporary") is reserved for use as a temporary variable by the assembler.

- `not $dst, $src`

```
nor   $dst, $src, $src
```

- `abs $dst, $src`

```
      slt   $at,  $zero, $src
      bne   $at,  $zero, pos
      sub   $dst, $zero, $src
      j     done
pos:  add   $dst, $zero, $src
done: ...
```

- `bgt $src0, $src1, label`

```
slt   $at, $src1, $src0
bne   $at, $zero, label
```

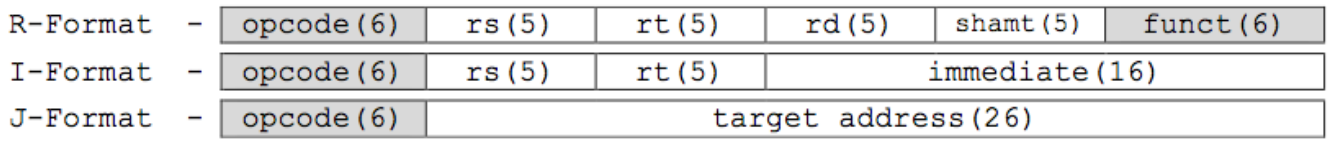
- `li $dst, imm32 # (32b immediate move)`

```
lui   $dst, imm[31:16]
ori   $dst, $dst, imm[15:0]

# Why doesn't the version below work?
lui   $dst, imm[31:16]
addi  $dst, $dst, imm[15:0]
```

MIPS Machine Code

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:



Here the instruction formats are written out by field name and width in bits in parentheses. The first bit of the opcode is the MSB and the other end is the LSB. The fields are used as follows:

`opcode`	indicates operation, or arithmetic family of operations (for `opcode` 0, which is R-type)
`funct`	indicates specific operation within arithmetic family of operations
`rs`, `rt`, `rd`	for R-type, `rs` and `rt` are sources with `rd` as destination – rules vary for other formats!
`shamt`	shift amount for instructions that perform shifts
`immediate`	Relative address or constant
`address`	Absolute address

Practice problem: Decode the instructions a.) `0x02114824` b.) `0x03E00008` c.) `0x292A0020`

a.) `0x02114824` → `and $9, $16, $17` → `and $t1, $s0, $s1`

b.) `0x03E00008` → `jr $31` → `jr $ra`

c.) `0x02114824` → `slti $10, $9, 32` → `slti $t2, $t1, 32`