The last page has a summary of information about the MIPS.

1. Consider the C declarations:

```
int arr[]= {1, 3, 5, 7, 9, 11, 13, 15, 17};
int k;
```

Suppose these are global variables, and the program is compiled so the memory location where the array `arr` begins (its base address) has the label name `arr` in the data segment and `k` has the label name `k`. Give MIPS assembly code which a compiler might generate for the following statements which use the array. You can assume the compiler doesn't perform any optimization.

   a. `printf("%d", arr[5]);`

   b. `k= arr[5]++;`

   c. `k= arr[(k + 5) * 3];`

2. Consider the C declarations:

```
typedef struct {
  int month;
  int day;
  int year;
} Date;

Date d;
Date may[31];
int n;
```

Suppose these are global variables, and the program is compiled so the memory location where the array `may` begins (its base address) has the label name `may` in the data segment, `d` has the label name `d`, and `n` has the label name `n`. Give MIPS assembly code which a compiler might generate for the following statements which use these declarations. You can assume the compiler doesn't perform any optimization.

   a. `d.year= 2008;`

   b. `printf("%d", d.month + d.day);`

   c. `n= may[15].year;`

3. Consider the C declarations:

```
int x, y;
int *p;
int **q;
```

Suppose these are global variables, and the program is compiled so the memory location where each variable is located has a label which is the same as its name. Give MIPS assembly code which a compiler might generate for the following statements which use these declarations. You can assume the compiler doesn't perform any optimization.

a. `p= &x;`

b. `*p= 13;`

c. `printf("%d", *p);`

d. `q= &p;`

e. `y= **q;`

f.
```
if (p != NULL)
  *p= 212;
```

g.
```
if (q != NULL && *q != NULL)
  **q= 212;
```

4. Give the MIPS assembly code which would be generated for leaving or returning from a function, which will pop its return address from the stack, and return to the caller. Remember that the stack pointer register is `$sp`. Assume that the function doesn't have any local variables or parameters.

5. Translate the following programs into MIPS assembly language, as if you yourself were a (nonoptimizing) C compiler, and trace their execution.

a.
```
#include <stdio.h>

void f(void) {
  printf("%d", 1);
}

void g(void) {
  f();
}

int main() {
  f();
  g();
  return 0;
}
```

b.
```
#include <stdio.h>

void f(void) {
  int a= 5;
  printf("%d", a);
}

int main() {
  int a= 3;

  printf("%d", a);
  f();

  return 0;
}
```

6. Trace through the following MIPS assembly code, describe what it does, and write a C program which does the same thing (i.e., a C program which could have produced the assembly code when compiled).

```
      .text
f:    sw      $ra, ($sp)
      sub     $sp, $sp, 12
      sw      $a0, 8($sp)
      lw      $t0, 8($sp)
      beqz    $t0, done
      move    $a0, $t0
      li      $v0, 1
      syscall
      sub     $t0, $t0, 2
      sw      $t0, 4($sp)
      move    $a0, $t0
      jal     f
done: add     $sp, $sp, 12
      lw      $ra, ($sp)
      jr      $ra

main: li      $sp, 0x7ffffffc
      li      $a0, 6
      jal     f
      li      $v0, 10
      syscall
```

7. Give the sum (in binary) of the two binary numbers $10010101_2 + 100111_2$.

8. Assuming 8 bits are use to store integers, convert the decimal number -21_{10} to binary, using two's–complement.

- The text segment is where a program's code is stored, and the data segment is where its global and static data are stored. The `.text` directive tells the assembler to put whatever follows in the text segment, and the `.data` directive tells the assembler to put whatever follows in the data segment.

- Registers `$t0` through `$t9` are general–purpose registers for computation, `$a0` through `$a3` are used to pass parameters to functions, and `$v0` and `$v1` are used to return a value from a function.

- **Only** load and store instructions access memory, since the MIPS has a load/store architecture.

- The memory addressing modes are:
  - immediate– a constant value is the operand
  - contents of register– a register name in parentheses means the register contains the memory address of the operand
  - indexed– a register name in parentheses preceded by a constant value *c* means the memory address that is *c* bytes past the memory address in the register is where the operand is
  - label– a label means that label (the memory location with that label) is the address of the operand

- The load and store instructions:

  `li`: the load immediate instruction loads a constant value (the second operand) into a register

  `lw`: the load word instruction loads a word from a memory address (specified by the second operand) into a register

  `la`: the load address instruction loads the memory address that the by the second operand specifies (not the contents of that memory location) into a register

  `sw`: the store word instruction stores the contents of its first register operand into a memory address (specified by the second operand)

- The `move` instruction transfers a value from one register (its second operand) to another one (its first register operand).

- Some branch and jump instructions:

  `beqz`: the branch if equal to zero instruction jumps to a memory location (the second operand, which is a label) if its first register operand contains zero

  `bnez`: the branch if not equal to zero instruction jumps to a memory location (the second operand, which is a label) if its first register operand does not contain zero

  `j`: the jump instruction unconditionally jumps to a label (its only operand)

  `jr`: the jump register instruction unconditionally jumps to whatever memory location its register operand contains (`$ra` is often used)

  `jal`: the jump and link instruction saves the address of the next instruction into the `$ra` register and then jumps to a memory address (which is specified by a label)

- System calls are executed by loading a code value in register `$v0` that indicates which system call to perform, arguments the system call should operate upon (if any) in registers `$a0` through `$ad30`, and invoking the `syscall` instruction. Some system calls:

system call name	code	argument	effect
`print_int`	1	an integer in `$a0`	prints the integer
`print_str`	4	the address of a null–terminated string in `$a0`	prints the string
`exit`	10	none	quits the program