Problem 3:
\[ c = a + b; \]
\[ d = c - b; \]
\[ a = c + d; \]

**Accumulator:**
load addrA
add addrB
store addrC
sub addrB
store addrD
add addrC
store addrA

**Memory-memory:**
add addrC, addrA, addrB
sub addrD, addrC, addrB
add addrA, addrC, addrD

**Stack**
push addrA
push addrB
add
pop addrC
push addrC
push addrB
sub
pop addrD
push addrC
push addrC
add
pop addrA

Above code an be optimized

**Load-store:**
load $r0, addrA
load $r1, addrB
add $r2, $r0, $r1
sub $r3, $r2, $r1
add $r0, $r2, $r3
store $r2, addrC
store $r3, addrD
store $r0, addrA

* 3-bytes to encode 3-operand instructions*
(1 byte for opcode, 2 bytes for register names)
* 4-bytes to encode load-store instructions

### Memory analysis

<table>
<thead>
<tr>
<th>Code</th>
<th>Memory</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>7*3=21</td>
<td>7*4=28</td>
</tr>
<tr>
<td>Mem-Mem</td>
<td>7*3=21</td>
<td>3*12=36</td>
</tr>
<tr>
<td>Stack</td>
<td>9*3+3=30</td>
<td>9*4=36</td>
</tr>
<tr>
<td>Load-store</td>
<td>5<em>4+3</em>3=29</td>
<td>5*4=20</td>
</tr>
</tbody>
</table>

In terms of code bandwidth alone, memory-memory architectures and accumulator architectures are the most efficient.

In this example, both accumulator and load-store architectures are equally efficient in terms of total memory bandwidth.

Mem-Mem architectures are common where on-chip storage is limited and instruction code density is very important. While accumulator architectures seem as efficient as load-store architecture, for large code sequences the single accumulator will become a bottleneck.

### Problem 4:

**MIPS:**

```assembly
add $v0, $zero, $zero          # freq = 0
add $t0, $zero, $zero          # i = 0
addi $t8, $zero, 400           # $t8 = 400
outer: add $t4, $a0, $t0       # $t4 = address of a[i]
lw $t4, 0($t4)                 # $t4 = a[i]
add $s0, $zero, $zero          # x = 0
addi $t1, $zero, 400           # j = 400
inner: add $t3, $a0, $t1       # $t3 = address of a[j]
lw $t3, 0($t3)                 # $t3 = a[j]
bne $t3, $t4, skip            # if (a[i] != a[j]) skip x++
addi $s0, $s0, 1               # x++
skip: add $t1, $t1, -4         # j--
bne $t1, $zero, inner         # loop if j != 0
slt $t2, $s0, $v0              # $t2 = 0 if x > freq
bne $t2, $zero, next           # skip freq = x if
add $v0, $s0, $zero            # freq = x
next: add $t0, $t0, 4           # i++
bne $t0, $t8, outer            # loop if i != 400
```

**PowerPC:**

```assembly
add $v0, $zero, $zero          # freq = 0
add $t0, $zero, $zero          # i = 0
```
addi $t8, $zero, 400     # $t8 = 400
add $t7, $a0, $zero     # keep track of $a[i] with update addressing
outer: lwu $t4, 4($t7)     # $t4 = $a[i]
addi $ctr, $zero, 100     # j = 100
add $t6, $a0, $zero     # keep track of $a[j] with update addressing
inner: lwu $t3, 4($t6)     # $t3 = $a[j]
bne $t3, $t4, skip     # if ($a[i] != $a[j]) skip $x++
addi $s0, $zero, $zero     # $x = 0
addi $t6, $a0, $zero     # keep track of $a[j] with update addressing
inner: lwu $t3, 4($t6)     # $t3 = $a[j]
bne $t3, $t4, skip     # if ($a[i] != $a[j]) skip $x++
addi $s0, $s0, 1     # $x++
skip: bc inner, $ctr!=0     # j--, loop if j!=0
slt $t2, $s0, $v0     # $t2 = 0 if $x >= freq
bne $t2, $zero, next     # skip freq = $x if
addi $t0, $t0, 4     # $i++
next: add $v0, $s0, $zero     # freq = $x
bne $t0, $t8, outer     # loop if $i != 400

Problem 5:
a: 0110 1111 1011 1010
b: 1001 1100 0110 0110
pi: 1111 1111 1111 1111
gi: 0000 1100 0010 0010
P: 1 1 1 1
G: 0 1 1 1

Problem 6:
Using C1, the average CPI for I1 is \((.4 \times 2 + .4 \times 3 + .2 \times 5) = 3\),
and the average CPI for I2 is \((.4 \times 1 + .2 \times 2 + .4 \times 2) = 1.6\). Thus,
with C1, I1 is
\[
\frac{(6 \times 10^9 \text{ cycles/second})/(3 \text{ cycles/instruction})}{(3 \times 10^9 \text{ cycles/second})/(1.6 \text{ cycles/instruction})} = 16/15
\]
times as fast as I2.

Using C2, the average CPI for I2 is \((.4 \times 2 + .2 \times 3 + .4 \times 5) = 3.4\),
and the average CPI for I2 is \((.4 \times 1 + .4 \times 2 + .2 \times 2) = 1.6\). So
with C2, I2 is faster than I1 by factor of
\[
\frac{(6 \times 10^9 \text{ cycles/second})/(1.6 \text{ cycles/instruction})}{(3 \times 10^9 \text{ cycles/second})/(3.4 \text{ cycles/instruction})} = 17/16.
\]

For the rest of the questions, it will be necessary to have the CPIs
of I1 and I2 on programs compiled by C3. For I1, C3 produces programs
with CPI \((.6 \times 2 + .15 \times 3 + .25 \times 5) = 2.9\). I2 has CPI
\((.6 \times 1 + .15 \times 2 + .25 \times 2) = 1.4\).

The best compiler for each machine is the one which produces programs
with the lowest average CPI. Thus, if you purchased either I1 or I2, you would use C3.

Then performance of I1 in comparison to I2 using their optimal compiler (C3) is
\[
\frac{(6 \times 10^9 \text{ cycles/second})}{(2.9 \text{ cycles/instruction})} / \frac{(3 \times 10^9 \text{ cycles/second})}{(1.4 \text{ cycles/instruction})} = 28/29.
\]
Thus, I2 has better performance and is the one you should purchase.

**Problem 7:**
The first option reduces the number of instructions to 80%, but increases the time to 120%. Thus it will take: 0.8 * 1.2 = 0.96 as much time as the initial case.

The second option removes 20%/2 = 10% of the instructions and increases the time taken to 110%. Therefore it will take 0.9 * 1.1 = 0.99 times as much time as the initial case.

Therefore, the first option is the faster of the two, and it is faster than the orginal, so you should have hardware automatically do garbage collection.

**Problem 8:**
objdump is a program for displaying various information about object files. For instance, it can be used as a disassembler to view executable in assembly form. It is part of the GNU binutils for fine-grained control over executable and other binary data.

```
push %ebp
mov %esp,%ebp
sub $0x8,%esp
and $0xffffffff0,%esp
mov $0x0,%eax
add $0xf,%eax
add $0xf,%eax
shr $0x4,%eax
shr $0x4,%eax
sub %eax,%esp    ; above save context
movl $0x0,0xfffffff8(%ebp)   ; initialize sum to 0
movl $0x0,0xfffffffc(%ebp)   ; initialize i to 0
0x804838e: cmpl $0x17,0xfffffffc(%ebp)  ; loop condition check to see
      ; whether to continue loop
      ; break out of loop
jg 80483a3 <main+0x3f>
mov 0xffffffffc(%ebp),%eax     ; load i into register eax
lea 0xffffffff8(%ebp),%edx     ; load address of sum into register edx
add %eax,(%edx)                 ; sum = sum + i
```
lea 0xffffffffc(%ebp),%eax ; load address of i into eax
incl (%eax) ; i++;
jmp 804838e <main+0x2a>

0x80483a3: mov 0xffffffff8(%ebp),%eax ; store sum to eax
    leave
    ret

With -O3:

    push %ebp
    xor %eax,%eax
    mov %esp,%ebp
    sub $0x8,%esp
    xor %edx,%edx
    and $0xffffffff0,%esp
    sub $0x10,%esp
    add %edx,%eax
    inc %edx
    cmp $0x17,%edx
    jle 8048380 <main+0x10>
    leave
    ret
    nop
    nop

The compiler optimizes the loop and determines that it can determine sum without executing the loop. Also with the -O3 optimization the compiler attempts to maintain most values in registers as opposed to memory.

The other instructions set up context to start the program and are part of standard libraries to load a program.
1-bit Shifter

2-bit Shifter

Similarly 4, 8 bit shifters

Top-level

IN[15:0]