CSc 252: Computer Organization

HW 2

due at 5pm, two days **before** the test

Turn in through GradeScope

Policy Reminders

- You must turn in a single `PDF` to GradeScope. Other file formats will not be accepted. (Sorry, this is necessary for the sanity of the TAs!)
- You are allowed to work with other students on this homework, as we will not be grading it for correctness. However, **each student must turn in their own copy of the homework.**
- **Show your work for all problems.** While we won't be grading for correctness, you will not receive full credit unless you show your work.

  After all, showing your work is required on the test - and homeworks are intended to help you practice for the test!

Required Problems:

2.1(i-j), 2.2(e), 2.3(d), 2.4(d)

Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- `add, addi, sub`
- `beq, bne, j`
- `slt, slti`
- `and, andi, or, ori, nor, xor, xori`
- `sll, srl, sra`
- `lw, lh, lb, sw, sh, sb`
- `la`
- `syscall`

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), do not use them! We want you to learn the **fundamentals** of how assembly language works - you can use fancy tricks after this class is over.

Problem 2.1 - More MIPS

This question assumes the following MIPS code, which sets up memory locations `atsf, dlw, bnsf, epsw, cbq, erie, csx,` and `kcs`. The code then loads the values of *some* of these variables into the indicated MIPS registers. In answering these questions, you can assume this code has already been executed, and that the value of some of the variables are already in the indicated registers.

Each part is independent of the other questions - that is, assume that the program has started over from scratch each time.

Do not modify any `sX` register, unless specifically instructed.

```
.data
atsf:       .word   xxx         # hidden so you can't hard-code values!
dlw:        .word   xxx
bnsf:       .word   xxx
epsw:       .word   xxx
cbq:        .word   xxx
erie:       .word   xxx
csx:        .word   xxx
kcs         .word   xxx

.text
main:
    # set $s0 = kcs
    la   $s0, kcs
    lw   $s0, 0($s0)

    # set $s1 = csx
    la   $s1, csx
    lw   $s1, 0($s1)

    # set $s2 = atsf
    la   $s2, atsf
    lw   $s2, 0($s2)

    # set $s3 = address of dlw
    la   $s3, dlw
```

2.1(a)

Put `csx - kcs - atsf` in register `$s4`

2.1(b)

Put `kcs + csx - atsf` in register `$s4`

2.1(c)

Put `erie` in register `$s4`

2.1(d)

Put `dlw - cbq` in register `$s4`

2.1(e)

If (`kcs == csx`), put `kcs + atsf` in register `$s4`

2.1(f)

Put `kcs - erie` in memory location `csx`

2.1(g)

If (`csx+kcs < erie`), put `csx+kcs` in register `$s4`

2.1(h)

If (`atsf-dlw <= kcs`), decrement `dlw` by one (update the value in memory)!

2.1(i) - Turn in this one

If (`bnsf-atsf > csx-kcs`), store `(bnsf-atsf)` into `cbq`.

2.1(j) - Turn in this one

If (`kcs < erie` and `erie < epsw`), then store the value of `kcs` into `epsw`.

Problem 2.2 - Masking

Masking is a technique that allows certain bits within a word to remain while other bits are set to zero. The idea is to create a mask that has `1`'s in the positions that you wish to remain, and `0`'s elsewhere. For example, if we want to keep bits 31 to 24 within a word but set all other bits to zero, we can use:

```
1111 1111 0000 0000 0000 0000 0000 0000
```

Sometimes, we store the mask as a variable, and load it from memory when we want to use it. This is useful when the mask is complex:

```
.data
mask:    .word    0xF0F0F0F0

.text
         la       $t0, mask
         lw       $t0, 0($t0)
         and      $s1, $s0, $t0
```

However, it is often easier and more efficient to generate the mask from simple instructions.

In each problem below, **first show the 32-bit mask** necessary to mask the bits required. Then give a sequence of instructions which takes a value in `$s0`, masks off the bits required, and stores the result in `$s1`. In all cases, you may only modify the destination register; no other registers should be changed.

The only instructions you are allowed to use are: `and, andi, addi, sll`. Note that, because you are not allowed to use `la` and `lw`, you cannot read from a mask stored in memory; you must construct it using immediate values. (Don't use the `lui` instruction, either.)

HINT: The assembler allows you to use hex values as your immediate values. Don't waste time converting long bit fields to decimal!

2.2(a)

Keep only bits 0 through 13. Do this in one instruction.

2.2(b)

Keep only bits 31 and 28. Do this in three instructions. [1]

2.2(c)

Keep bits 12 through 23. Do this in three instructions. [2]

2.2(d)

Keep bits 0 through 3, and **also** bits 16 through 19. Do this in four instructions.

2.2(e) - Turn in this one

Keep all of the even bits. Do this in four instructions.

[1] If I allowed you another instruction - `lui` - it could be done in two instructions. But that instruction is not allowed, yet.

[2] This can also be done with shifts in three instructions, but remember that this problem requires that you use a mask instead!

Problem 2.3 - Truth Tables and Sum of Products

For each part, convert the truth table into a sum-of-products expression for each of the outputs (W, X, Y, Z).

Then draw a logic network which calculates **only** the Z output from the inputs.

2.3(a)

A	**B**	**C**	W	X	Y	Z
0	0	0	0	0	1	0
0	0	1	0	0	0	0
0	1	0	0	1	0	0
0	1	1	0	0	0	1
1	0	0	0	1	0	1
1	0	1	0	1	0	0
1	1	0	1	0	0	1
1	1	1	1	0	1	1

2.3(b)

A	**B**	**C**	W	X	Y	Z
0	0	0	1	0	1	1
0	0	1	1	0	1	0
0	1	0	1	1	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	1
1	1	0	0	0	1	1
1	1	1	0	1	0	1

2.3(c)

A	**B**	**C**	W	X	Y	Z
0	0	0	0	0	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	0
0	1	1	1	1	0	1
1	0	0	1	1	1	0
1	0	1	0	0	0	1
1	1	0	0	0	1	0
1	1	1	0	1	1	0

2.3(d) - Turn in this one

A	**B**	**C**	W	X	Y	Z
0	0	0	0	1	0	1
0	0	1	0	0	1	0
0	1	0	0	0	1	0
0	1	1	0	1	0	1
1	0	0	0	0	0	1
1	0	1	1	1	0	0
1	1	0	1	1	0	1
1	1	1	1	0	1	0

Problem 2.4 - Loops in MIPS

Convert the following loops from C to assembly. Some variables have already been assigned to registers; for others, you will need to assign them registers yourself. Use `sX` registers for all variables that have names in the C program; use `tX` registers for any other temporaries that you create.

2.4(a)

C code:

```
int sum = 0;    // s0
for (int i=-10; i<10; i++)
{
  sum += i;
}
```

2.4(b)

C code:

```
int val = ... ;    // s2 - assume that some previous code has set it!
while (val > 0)
{
  // HINT: '0' is equal to 0x30    http://www.asciitable.com
  printf("%c", '0'+(val & 0x1));

  // REMINDER: 'div' is not allowed.  Use a shift.
  val /= 2;
}
```

Bonus Question (not required): What does this program do? Why does it have a bug if `$s2 < 0` ?

2.4(c)

C code:

```
int len = ... ;   // s1 - this is set by previous code
int min = ... ;   // s3 - this is set by previous code
for (int i=0; i<len; i++)
{
  if (i >= min)
    printf("%d", i);
}
```

2.4(d) - Turn in this one

C code:

```
int pow  = ... ;   // s0 - this is set by previous code

int prod = 1;      // allocate a register for this, of your choosing
for (int i=0; i<pow; i++)
{
    prod = prod*2;
}
```

EXAMPLES

Example: Problem 2.1(a)

Put `csx - kcs - atsf` in register `$s4`

```
    sub     $s4, $s1, $s0          # s4 = csx - kcs
    sub     $s4, $s4, $s2          # s4 = csx - kcs - atsf
```

Example: Problem 2.1(b)

Put `kcs + csx - atsf` in register `$s4`

```
    add     $s4, $s0, $s1          # s4 = kcs + csx
    sub     $s4, $s4, $s2          # s4 = kcs + csx - atsf
```

Example: Problem 2.1(c)

Put `erie` in register `$s4`

```
    la      $s4, erie              # s4 = &erie
    lw      $s4, 0($s4)            # s4 =  erie
```

Example: Problem 2.1(d)

Put `dlw - cbq` in register `$s4`

```
    lw      $s4, 0($s3)            # s4 =  dlw
    la      $t0, cbq               # t0 = &cbq
    lw      $t0, 0($t0)            # t0 =  cbq
    sub     $s4, $s4, $t0          # s4 =  dlw - cbq
```

Example: Problem 2.1(e)

If (`kcs == csx`), put `kcs + atsf` in register `$s4`

```
    bne     $s0, $s1, AFTER_IF     # if (kcs != csx) skip ahead
    add     $s4, $s0, $s2          # if (kcs == csx) s4 = kcs + csx
AFTER_IF:
```

Example: Problem 2.1(f)

Put `kcs - erie` in memory location `$csx`

```
    la      $t0, erie              # t0 = &erie
    lw      $t0, 0($t0)            # t0 =  erie
    sub     $t0, $s0, $t0          # t0 =  kcs - erie
    la      $t1, csx               # t1 = &csx
    sw      $t0, 0($t1)            # csx = kcs - erie
```

Example: Problem 2.1(g)

If (`csx+kcs < erie`), put `csx+kcs` in register `$s4`

```
    add     $t0, $s1, $s0         # t0 =  csx+kcs
    la      $t1, erie             # t1 = &erie
    lw      $t1, 0($t1)           # t1 =  erie
    slt     $t1, $t0, $t1         # t1 = (csx+kcs) < erie
    beq     $t1, $zero, AFTER_IF  # if (csx+kcs >= erie) jump ahead
    add     $s4, $t0, $zero
AFTER_IF:
```

Example: Problem 2.1(h)

If (`atsf-dlw <= kcs`), decrement `dlw` by one (update the value in memory)!

```
    lw   $t0, 0($s3)    # t0 = dlw
    sub  $t1, $s2, $t0  # t1 = atsf-dlw

    slt  $t2, $s0, $t1  # t2 = kcs < (atsf-dlw)
    bne  $t2, $zero, AFTER_IF:   # if (atsf-dlw > kcs) skip over

    addi $t0, $t0, -1   # t0 = dlw-1
    sw   $t0, 0($s3)    # dlw--

AFTER_IF:
```

Example: Problem 2.2(a)

Mask: `0000 0000 0000 0000 0011 1111 1111 1111`

```
    andi  $s1, $s0, 0x3fff
```

Example: Problem 2.2(b)

Mask: `1001 0000 0000 0000 0000 0000 0000 0000`

```
    addi  $s1, $zero, 0x9    # s1 = 1001
    sll   $s1, $s1, 28       # s1 = 1001 0000 0000 0000 0000 0000 0000 0000
    and   $s1, $s0, $s1
```

Example: Problem 2.2(c)

Mask: `0000 0000 1111 1111 1111 0000 0000 0000`

```
    addi   $s1, $zero, 0x0fff
    sll    $s1, $s1, 12
    and    $s1, $s0, $s1
```

Example: Problem 2.2(d)

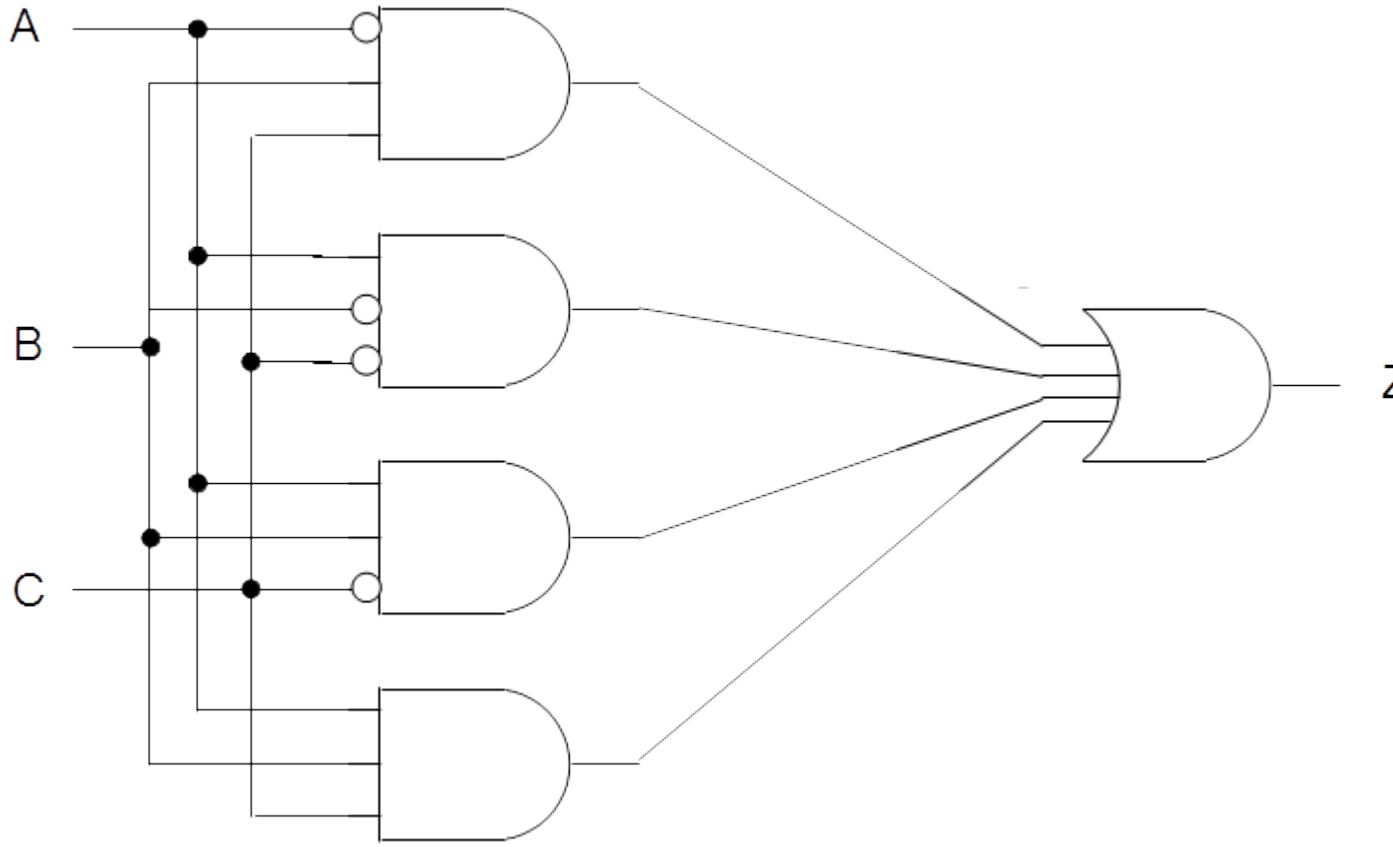
Keep bits 0 through 3, and **also** bits 16 through 19. Do this in four instructions.

Mask: `0x000F 000F`

```
addi  $s1, $zero, 0x000F
sll   $s1, $s1,   16
addi  $s1, $s1,   0x000F
and   $s1, $s0,   $s1
```

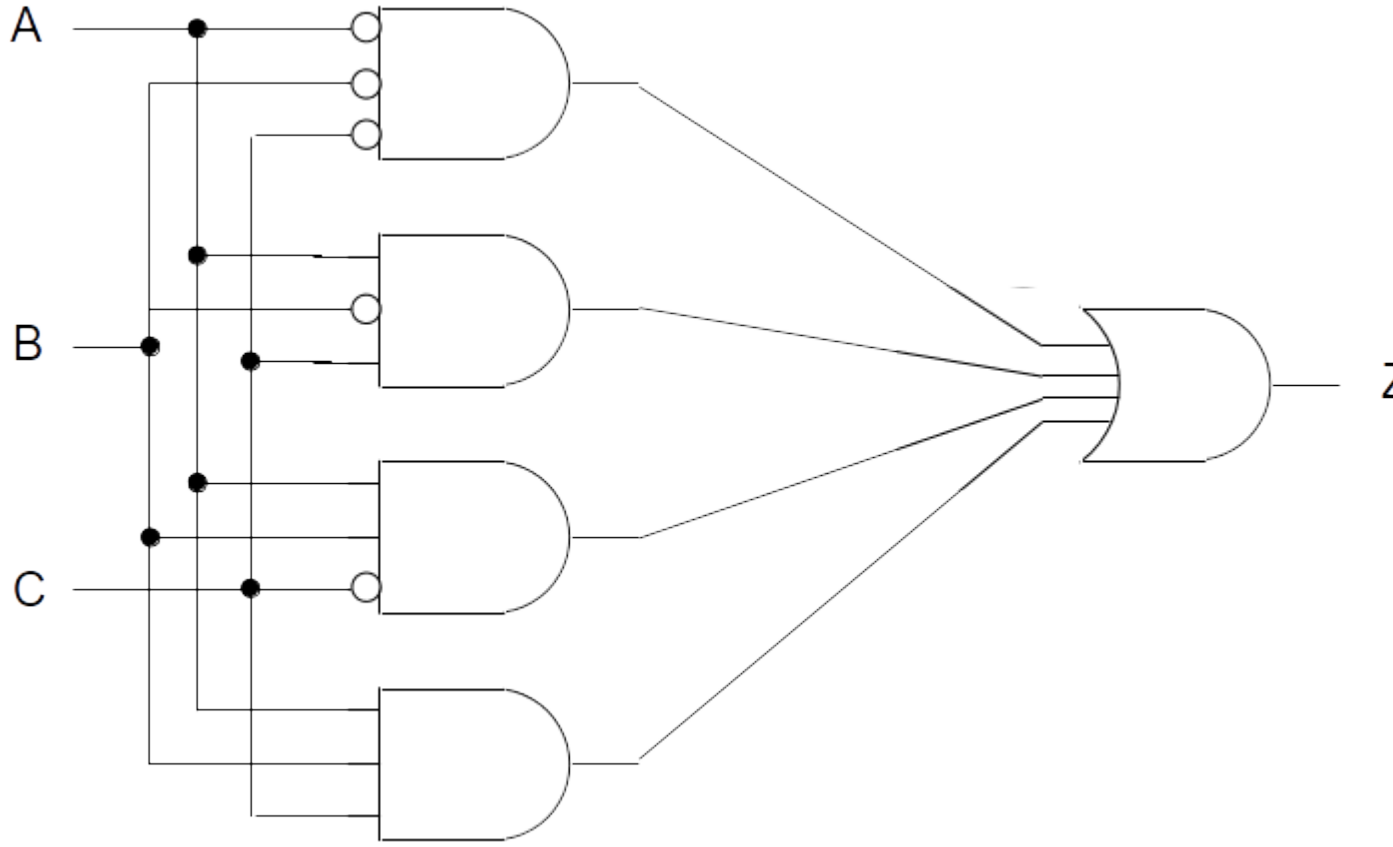
Example: Problem 2.3(a)

$W = A\ B\ \overline{C} + A\ B\ C$
$X = \overline{A}\ B\ \overline{C} + A\ \overline{B}\ \overline{C} + A\ \overline{B}\ C$
$Y = \overline{A}\ \overline{B}\ \overline{C} + A\ B\ C$
$Z = \overline{A}\ B\ C + A\ \overline{B}\ \overline{C} + A\ B\ \overline{C} + A\ B\ C$



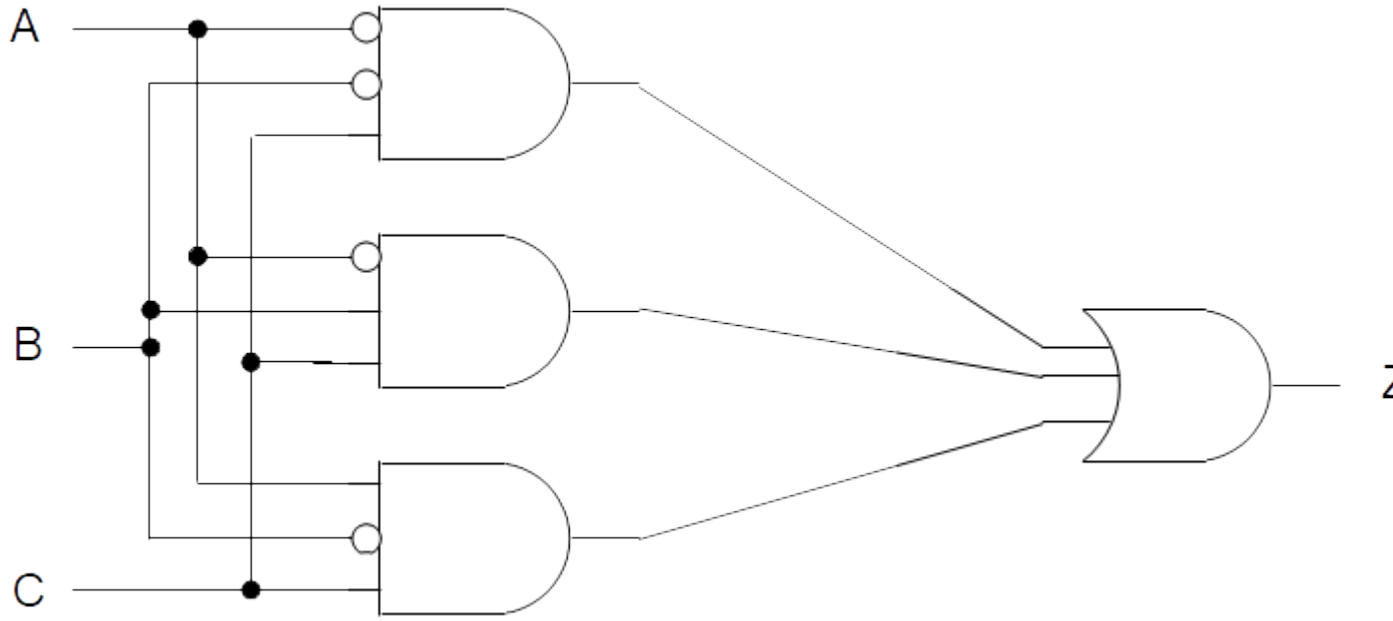
Example: Problem 2.3(b)

$W = \overline{A}\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + \overline{A}\ B\ \overline{C}$
$X = \overline{A}\ B\ \overline{C} + \overline{A}\ B\ C + A\ B\ C$
$Y = \overline{A}\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + A\ \overline{B}\ C + A\ B\ \overline{C}$
$Z = \overline{A}\ \overline{B}\ \overline{C} + A\ \overline{B}\ C + A\ B\ \overline{C} + A\ B\ C$



Example: Problem 2.3(c)

$W = \overline{A}\ \overline{B}\ C + \overline{A}\ B\ \overline{C} + \overline{A}\ B\ C + A\ \overline{B}\ \overline{C}$
$X = \overline{A}\ \overline{B}\ C + \overline{A}\ B\ C + A\ \overline{B}\ \overline{C} + A\ B\ C$
$Y = \overline{A}\ \overline{B}\ \overline{C} + \overline{A}\ B\ \overline{C} + A\ \overline{B}\ \overline{C} + A\ B\ \overline{C} + A\ B\ C$
$Z = \overline{A}\ \overline{B}\ C + \overline{A}\ B\ C + A\ \overline{B}\ C$



Example: Problem 2.4(a)

```
    # Remember: sum is in s0.
    #
    # We'll decide to store i in t0, and other temporaries in t1.

    addi $s0, $zero, 0            # sum = 0
    addi $t0, $zero, -10          # i   = -10

LOOP:
    slti $t1, $t0, 10             # t1 = (i < 10)
    beq  $t1, $zero, AFTER        # if (i >= 10) break

    add  $s0, $s0, $t0            # sum += i

    addi $t0, $t0, 1              # i++
    j LOOP
AFTER:
```

Example: Problem 2.4(b)

```
    # Remember: val is in s2.  It has been initialized to *some* value before
    # this code runs.
    #
    # We'll decide to use t1 for all temporaries.

LOOP:
    slt  $t1, $zero, $s2          # t1 = (0 < val)
    beq  $t1, $zero, AFTER        # if (val <= 0) break

    andi $t1, $s2, 0x1            # t1 = (val & 0x1)
    addi $a0, $t1, 0x30           # t1 = 0x30 + (val & 0x1)
    addi $v0, $zero, 11           # print_char('0' + (val & 0x1))
    syscall

    sra  $s2, $s2, 1              # s2 /= 2

    j LOOP
AFTER:
```

Bonus Question: This program prints out the bits of `$s2`, in reverse order (that is, from LSB to MSB). It has a bug if `$s2 < 0` because it will never print out anything; the loop ends immediately.

Example: Problem 2.4(c)

```
      # remember: s1 contains len, s3 contains min
      #
      # s0 will contain i
      # t0 will be used for various temporaries

      addi  $s0, $zero, 0                # i = 0

LOOP:
      slt   $t0, $s0, $s1                # t0 = (i < len)
      beq   $t0, $zero, AFTER            # if (i >= len) break

      slt   $t0, $s0, $s3                # t0 = (i < min)
      bne   $t0, $zero, SKIP             # if (i < min) skip over

      addi  $v0, $zero, 1                # print_int(i)
      add   $a0, $s0, $zero
      syscall

SKIP:
      addi  $s0, $s0, 1                  # i++
      j     LOOP

AFTER:
```