Name Solution

Computer Organization
EE 3755
Midterm Examination

Wednesday, 30 October 2013, 8:30–9:20 CDT

Problem 1		(21 pts)
Problem 2		(15 pts)
Problem 3		(25 pts)
Problem 4		(11 pts)
Problem 5		(11 pts)
Problem 6		(11 pts)
Problem 7		(6 pts)
Exam Total		(100 pts)

Alias They Know

Good Luck!

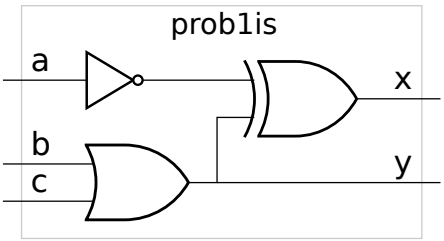
Problem 1: [21 pts] Consider the module below:

```
module prob1is(x,y,a,b,c);
   input a, b, c;   output x, y;
   wire  a, b, c, x, y;          <-- HINT: Declare types for parts b and c.
   assign x = !a ^ y;
   assign y =  b || c;
endmodule
```

(*a*) Draw a logic diagram corresponding to the module. Don't optimize.

☑ Logic diagram of `prob1is`.

Solution:



(*b*) Complete the module below so that it performs the same operation as `prob1is` but is in explicit structural form.

☑ Explicit structural form of `prob1is`. ☑ Don't forget to declare types.

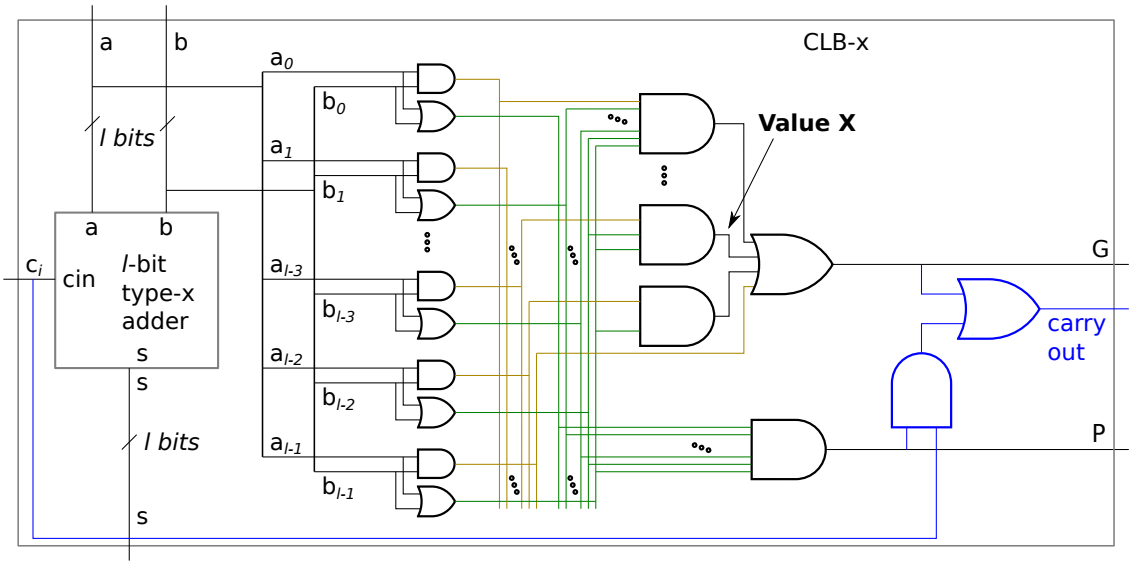
```
module prob1es(x,y,a,b,c);
   input a, b, c;    output x, y;

 // SOLUTION
   wire   a, b, c, x, y, na;
   not n1(na,a);
   or o1(y,b,c);
   xor x1(x,na,y);

endmodule
```

(*c*) Complete the module below so that it performs the same operation as `prob1is` but is in synthesizable behavioral form.

☑ Synthesizable behavioral form of `prob1is`. ☑ Don't forget to declare types.

```
module prob1sb(x,y,a,b,c);
   input a, b, c;    output x, y;

 // SOLUTION
   wire  a, b, c;
   reg   x, y;

   always @ ( a or b or c ) begin
      y =  b || c;  // Note that y must be assigned before x.
      x = !a ^ y;
   end


endmodule
```

Problem 2: [15 pts] Appearing below is the generic carry lookahead block from the CLA lecture slides. Remember that it is part of a hierarchical carry lookahead adder.



(*a*) The output of one of the AND gates is labeled *Value X*. Suppose $l = 16$. Find inputs to the module such that Value X is `1` but all of the other inputs to the generate OR gate are `0`.

☑ Value of module inputs needed so that Value X will be `1`.

Tracing the inputs of the AND gate back to module inputs `a` and `b`, we find that we need to set $a = e000_{16}$ and $b = 2000_{16}$. Specifically, we need to set a_{l-3} and b_{l-3} to `1` and either (but not both) a_{l-2} or b_{l-2} to `1` and either a_{l-1} or b_{l-1} to `1`. For bits $l-4$ to 0 the easy thing to do is setting them all to zero.

Grading Note: This problem confused many students. I'm sure every student thoroughly understood that the output of an AND gate is `1` *iff all its inputs are* `1` *and that the output of an OR gate is* `0` *iff all its inputs are* `0`*. Perhaps some students need to think more about how an l-bit quantity like* `a` *connects to the different gates. It's not difficult, but one should not think about such things for the first time during an exam.*

(*b*) Notice that the module does not have a carry out signal. Add logic to the module above to compute a carry out signal. (See next part.)

☑ Logic for carry out. ☑ Do not add new inputs to the module.

Solution appears above in blue. Notice that the carry out is conveniently generated from the propagate, generate, and carry-in signals.

(*c*) The carry out signal from the previous part isn't needed in real life. Why not? What if it were used?

☑ Why doesn't the CLB need a carry out signal?

Because the `P` and `G` signals are intended for external carry generation logic, which will generate the carry signal. If the module generated the carry signal there would be no reason to have `P` and `G` outputs.

☑ What would be the problem with using the carry out signal?

It would result in a slower adder (assuming lookahead carry generation logic (CGL)). The `P` and `G` values are available after 3 gate delays (using the faster model), but the carry out we generate would not be available until 2 gate delays after the carry in. The last carry out would have to wait 2 gate delays for each block, which would be longer than the delay for lookahead CGL.

Problem 3: [25 pts] A population count module based on the Homework 3 Problem 2 solution appears below.

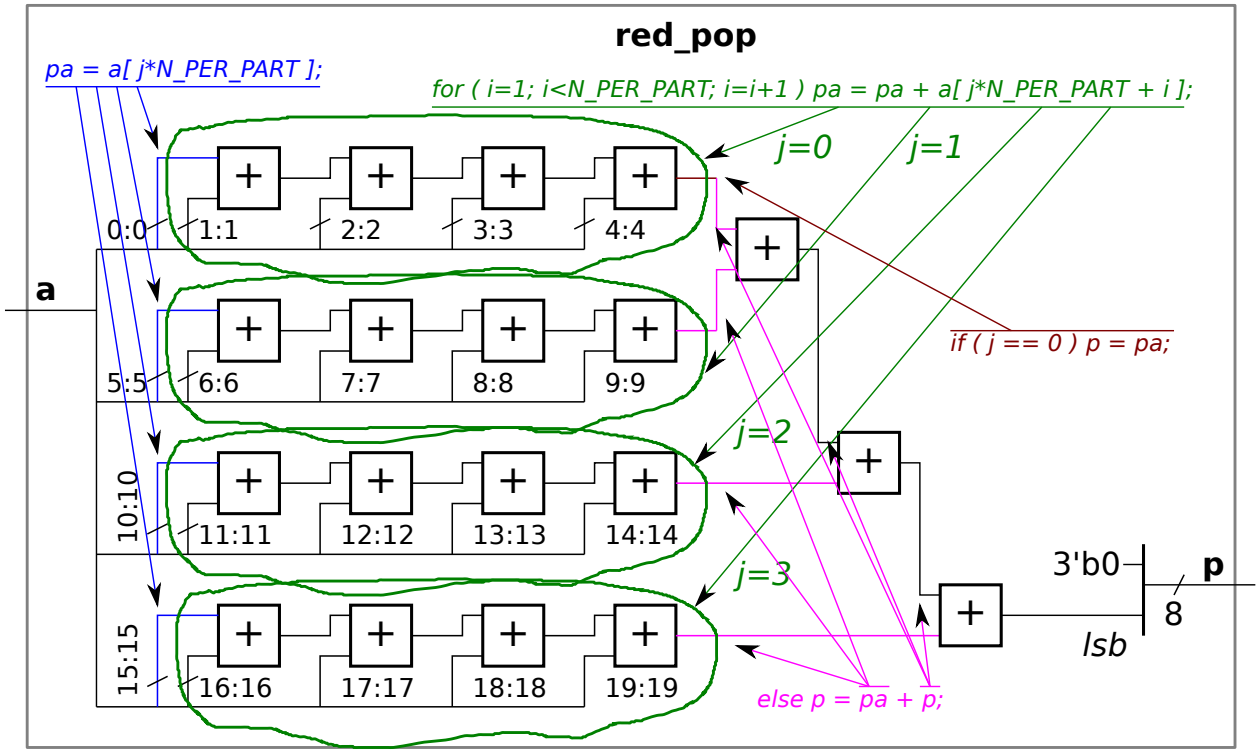
```
module red_pop(p,a);
   parameter N = 20;
   parameter M = 4;   // Number of parts.
   parameter N_PER_PART = N/M;
   input wire [N-1:0] a;        output reg [8:0] p;
   reg [8:0]    pa;             integer      i, j;

   always @( a ) begin
      for ( j=0; j<M; j=j+1 ) begin
         pa = a[ j*N_PER_PART ];
         for ( i=1; i<N_PER_PART; i=i+1 ) pa = pa + a[ j*N_PER_PART + i ];
         if ( j == 0 ) p = pa; else p = p + pa;
      end
   end
endmodule
```

(*a*) Sketch the logic that would be inferred for this module (the logic that would be synthesized without optimization). Show adders as a box with a + inside.

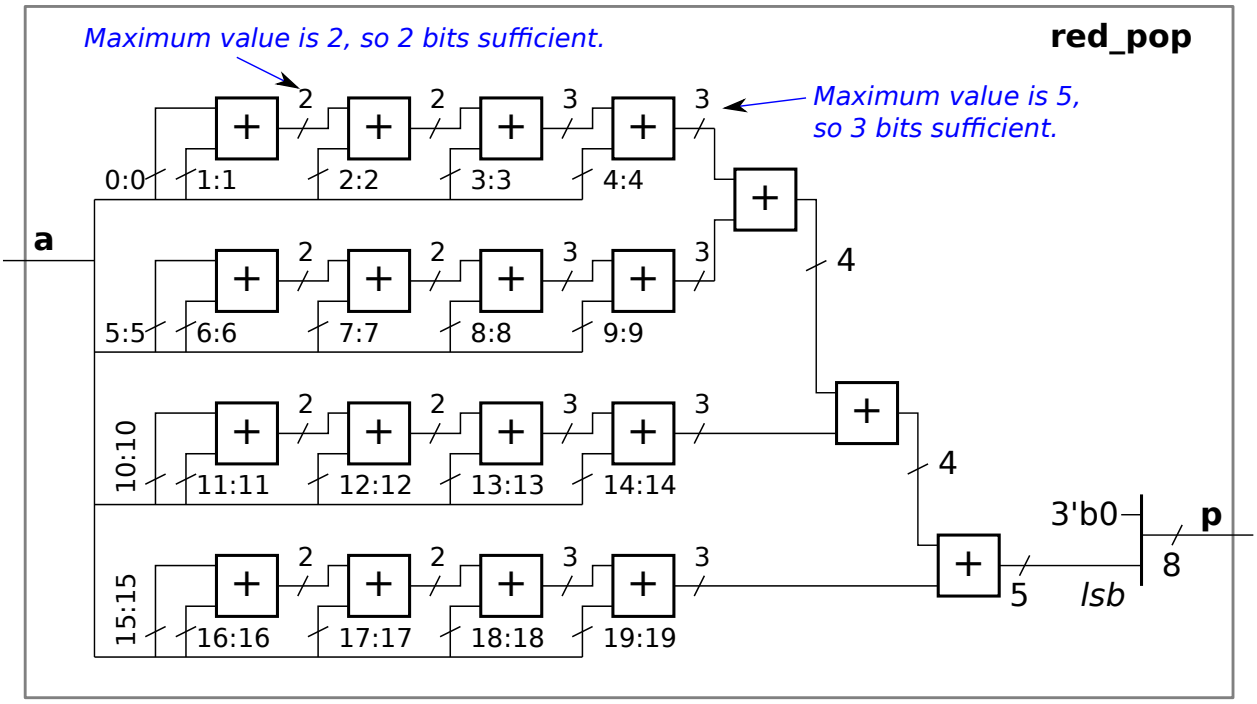
☑ Show synthesized logic without optimization.

The synthesized logic is shown below twice. In the first diagram the relationship between the logic and the behavioral code is emphasized. In the second diagram the size of each ripple adder is shown (as a bit with on the adder output).

Synthesized Logic Showing Relationship to Behavioral Code



Synthesized Logic Showing Ripple Adder Sizes



Each + in the diagrams is a ripple adder, the number at the output of each ripple adder indicates its size. For example, the ripple adder in the upper-left is a 2-bit ripple adder, meaning that it is constructed from 2 binary full adders.

The ripple adder size is based on the maximum possible value at the output. For example, the upper-left ripple adder, since it only sees the first two bits, has a maximum output of 2, which requires 2 bits. The next ripple adder to the right has a maximum value of 3, which still takes two bits. The next one has a maximum of 4, requiring 4 bits, and so on.

Other than reducing the size of the ripple adders, no optimization was required by the problem.

Many other optimizations could have been applied. For example, the ripple adders could have been built from binary half adders, and each ripple adder could have had two inputs from `a` (one input being an operand and the other the carry in). In fact, an optimized version would not use ripple adders organized as shown above. As was mentioned, these optimizations were not part of the problem.

Problem 3, continued: The module below is the same as the one on the previous page.

```
module red_pop(p,a);
   parameter N = 20;
   parameter M = 4;   // Number of parts.
   parameter N_PER_PART = N/M;
   input wire [N-1:0] a;        output reg [8:0] p;
   reg [8:0]    pa;             integer      i, j;

   always @( a ) begin
      for ( j=0; j<M; j=j+1 ) begin
         pa = a[ j*N_PER_PART ];
         for ( i=1; i<N_PER_PART; i=i+1 ) pa = pa + a[ j*N_PER_PART + i ];
         if ( j == 0 ) p = pa; else p = p + pa;
      end
   end
endmodule
```

(*b*) Compute the cost of the logic from the previous part assuming that all adders are ripple adders and that the cost of a BFA is 7 units. The size of the adders has been made as small as possible. Assume that the gates outside of the BFAs have a cost of one unit each.

☑ Cost of circuit. Show work for partial credit.

☑ Account for size of adders, the sizes have been minimized.

In the diagram on the previous page the size of each adder is shown. The size is in bits, and for each bit we can assume a BFA is needed. (If additional optimization were performed the circuit would be re-arranged and there would be a mix of binary full adders and binary half adders.) The total number of BFAs needed is $4(2+2+3+3)+4+4+5 = 53$, and their cost is $53 \times 7 = 371$ units.

(*c*) Suppose the delay of each ripple adder is just 1 time unit. (Yes, just 1 time unit.) Find the delay of the circuit in terms of `N` and `M`. That is, don't assume `N=20` and `M=4`.

☑ Delay in terms of `N` and `M` assuming 1-time-unit ripple adders.

The critical path is from `a[0]`. It flows through the $N/M-1$ adders inferred from the `i` loop, then through $M-1$ adders inferred from the `if ( j == 0 )` statement. Since the delay of each ripple adder is 1 the total delay is $N/M+M-2$.

(*d*) In the module above `M` was set to 4 and `N` to 20. Consider the case where `N` is some arbitrary value, and we want to compute a value for `M` that minimizes delay for this `N`. Using the delay model from the previous part, find an expression for `M` in terms of `N` that will minimize delay.

☑ Best value of `M` in terms of `N`.

To find the value of M that minimizes delay we differentiate the delay expression with respect to M:

$$\frac{d}{dM}\left(\frac{N}{M}+M-2\right) = -\frac{N}{M^2}+1.$$

Setting this expression to zero and solving for M will give us the M that minimizes or maximizes delay (we need to check). Solving gives $M=\sqrt{N}$, for a delay of $2\sqrt{N}-2$. To verify that this isn't a maximum try setting M to 1. We only need to show $2\sqrt{N}-2 < N+1-2$, which is true for $N \geq 4$.

Problem 4: [11 pts] Answer the computer arithmetic questions below.

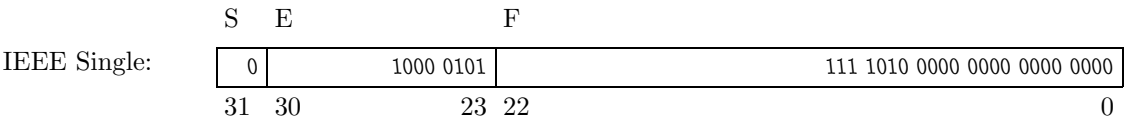
(*a*) Show the longhand steps needed to multiply $1234_{16} \times 1203_{16}$ using a radix-16 (four bit) multiplication algorithm, but do not add together the partial products. Show the work in binary or hexadecimal. This is **without** Booth recoding. (The product is $147de9c_{16}$, but remember there is no need to add the partial products.)

☑ Longhand steps for radix-16 (4 bit) $1234_{16} \times 1203_{16}$

The solution appears below. *Grading Note: Many students solved this in binary. Hexadecimal is much easier for radix 16.*

```
SOLUTION:
        1234
      × 1203
        ----
        369c   = 3 * 0x1234 = 0x369c.  Shifted by 0 bits.
           0   = 0 * 0x1234 = 0        Shifted by 4 bits.
     24 6800   = 2 * 0x1234 = 0x2468.  Shifted by 8 bits.
    123 4000   = 1 * 0x1234 = 0x1234.  Shifted by 12 bits.
    --------
    147 de9c   Note: The material above was sufficient for full credit,
               there was no need to add up the partial products.
```

(*b*) Show the value of IEEE 754 single-precision floating point number `0x42fa0000`. For your convenience the layout of an IEEE 754 single is shown below.



☑ Value of number (in decimal or as a formula to compute value) is:

First, split `0x42fa0000` into sign, exponent, significand: `0x42fa0000` ⟶ `0 1000 0101 11110100000000000000000`. The diagram above shows these in their proper place.

Next, compute the exponent. The biased exponent is $1000\,0101_2 = 133$. Subtracting the bias for an IEEE single number, 127, gives the exponent we need $133 - 127 = 6$.

The value in the significand field is used for the digits to the right of the binary point. To the left of a binary point there is a 1 (except for a special case). We get $1.1111\,01_2$ (with the trailing 0's omitted).

The value of the entire FP number is

$$1.111101_2 \times 2^6 = \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{64}\right) \times 2^6 = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0 = 125$$

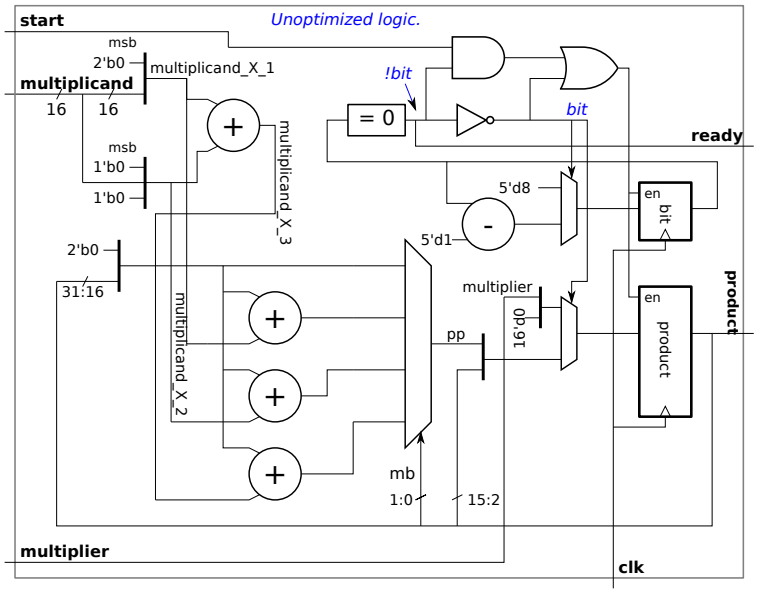
.

The same value can be obtained by treating the significand as an integer:

$$\left(1 + \frac{11110100000000000000000_2}{2^{23}}\right) \times 2^6 = \left(1 + \frac{111101_2}{2^6}\right) \times 2^6 = 64 + 61 = 125$$

.

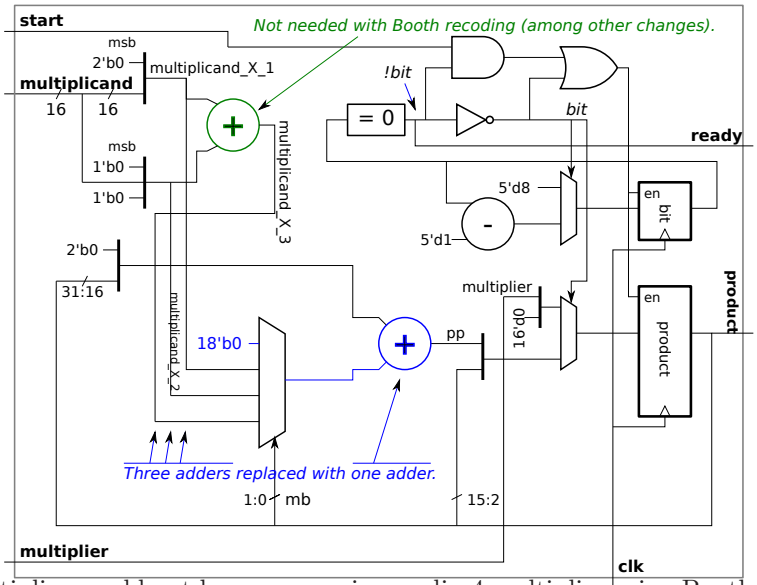
Problem 5: [11 pts] Appearing below is unoptimized logic for an ordinary radix-4 multiplier.



(*a*) As described in class, the logic around the four-input multiplexor can be reduced in cost. Show how.

☑ Sketch of lower-cost logic around the 4-input mux.

Solution appears to the right in blue. Notice that upper input to the three adders at the input the multiplexor all use the same value. So, the three adders at the mux input can be replaced by one adder at the mux output, with the mux being used to select the lower adder input.



(*b*) One of the adders in the illustrated multiplier would not be necessary in a radix-4 multiplier using Booth recoding. Show which adder would not be necessary, and explain what would be done instead.

☑ Indicate which adder not necessary.

☑ Explain what would be done instead.

The unneeded adder is shown above in green. A radix-4 multiplier using Booth recoding avoids the need to pre-compute $3\times$ the multiplicand by subtracting the multiplicand and then adding $4\times$ the multiplicand (in the next step).

Problem 6: [11 pts] Answer the following MIPS questions.

(*a*) Show the values in register `$s0` after the execution of each instruction below in the spaces indicated.

☑ Fill in the `s0 =` blanks below.

The results appear below. Note that the immediate in the `addi` instruction is sign-extended, and so its value is -1. In contrast, the `ori` instruction does not sign-extend its immediate which is why `s0` is written with `0xffff` rather than `0xffffffff`

```
 # Initial register values:  $s1 = 0x18,  $s2 = 0x30

or $s0, $s1, $s2
                    # $s0 = SOLUTION: 0x38

sll $s0, $s2, 2
                    # $s0 = SOLUTION: 0xc0

slt $s0, $s1, $s2
                    # $s0 = SOLUTION: 1

addi $s0, $s1, 0xffff
                    # $s0 = SOLUTION: 0x17

ori $s0, $s1, 0xffff
                    # $s0 = SOLUTION: 0xffff
```

(*b*) There is a problem with the instruction below. Describe what the problem is and show replacement instruction(s) that do what was intended.

```
addi $s0, $s0, 0x12345
```

☑ The problem with the instruction is:

The immediate is too large, more than 16 bits, and so there is no way to encode the instruction. An assembler would give an error for such an "instruction."

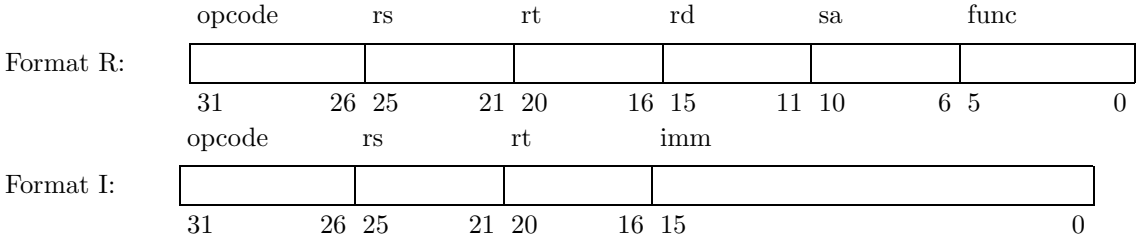
☑ Replacement that does the same thing:

The solution is to load the constant in two parts:

```
# BEST SOLUTION
 lui $s0, 0x1
 ori $s0, $s0, 0x2345

# OK SOLUTION
 addi $s0, $0, 0x1
 sll  $s0, $s0, 16
 ori  $s0, $s0, 0x2345

# CORRECT, just 2 instructions, but not generalizable.
 ori $s0, $0, 0xe345
 addi $s0, $s0, 0x4000
```

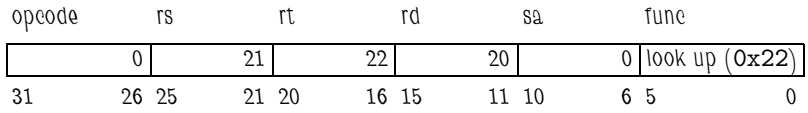
Problem 7: [6 pts] Answer each MIPS encoding question below.

(*a*) Show the encoding of each assembly language instruction below. Some field values would have to be looked up in a table, for those write "look up" instead of the value. For your convenience the layout of the MIPS R and I formats are shown, answers can be written in these or they can be copied.



☑ Encoding for `sub $20, $21, $22`

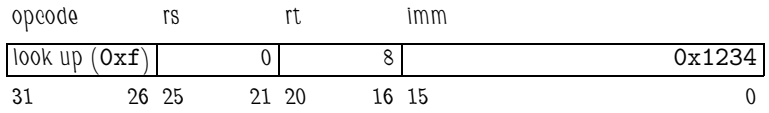
The encoding is:



We assume that the opcode is zero because it is a type-R instruction. The `sa` field (shift amount) is zero because the field is unused by the instruction and the MIPS ISA definition usually sets unused fields to zero (preserving them for future use). The `func` field is essentially a continuation of the `opcode` field for format R, and there is no way of knowing the value for `sub` short of memorization, so the correct answer is *look up*. For the record, the actual value is `0x22`.

☑ Encoding for `lui $8, 0x1234`

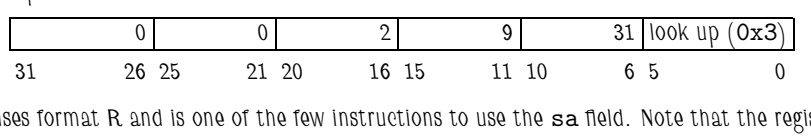
The encoding is:



Note that the destination (r8) is specified in the `rt` field. The `lui` has no source operand, and so one should guess that a 0 goes in the `rs` field, which is correct. The `opcode`, which was not required for full credit, is `0xf`.

☑ Encoding for `sra $9, $2, 31`

The encoding is:



The `sra` instruction uses format R and is one of the few instructions to use the `sa` field. Note that the register source operand (r2) is the `rt` field.