2005Fa CS61C Final Exam Answers
[not to leave 385 Soda]

M1:Numbers

a) Overall bit patterns? $2^{32} = 4{,}294{,}967{,}296$ (the exact # is not required; roughly a bit more than 4,000,000,000)
How many encode a valid BCD? 8 decimal digits, so $10^8 = 100{,}000{,}000$
Ratio is $2^{32}/10^8 = 42.94967296 \approx 40$ (to one significant figure).

b) Each pixel is independent, and there are $4\text{x}8=32=2^5$ of them, so it's $4^{32} = (2^2)^{32} = 2^{64}$ = 16 exbi images.

c) Comparing `floats` using signed `int` compare, huh? The relative ordering of all positive numbers is the same (increasing from 0 to max_positive) for both encodings, so comparing two positive `floats` with signed compare works. Also, for both encodings the bit patterns for negative numbers all start with a leading 1 (`0x80000000` through `0xFFFFFFFF`) so comparing a negative `float` with a positive `float` using signed int compare will also yield the correct answer. However, when comparing two negative `floats`, the sign-magnitude nature of `floats` means that as we increase the bit patterns from (`0x80000000` through `0xFFFFFFFF`) `floats` move from 0 toward $-\infty$, but signed `ints` move the other way from $-\infty$ (-2^{31}, really) toward 0. Thus, we will get an incorrect answer when comparing two different negative numbers.

d) Put the corresponding letters for each 32-bit value in order from least to greatest:

A. `0xF0000000` (IEEE float) = - huge
B. `0xF0000000` (2's complement) = $-2^{31} + 2^{30} + 2^{29} + 2^{28}$
C. `0xF0000000` (sign-magnitude) = $-(2^{31} - 2^{28}) = -2^{31} + 2^{28}$
D. `0xFFFFFFFF` (2's complement) = -1
E. `0xFFFFFFFF` (1's complement) = -0
F. `0xF1000000` (IEEE float) = - huger
G. `0x70000000` (IEEE float) = + huge
H. `0x7FFFFFFF` (2's complement) = $2^{31} - 1$
I. `0x80000010` (IEEE float) = - small denorm (value doesn't matter)

f, a, c, b, d, i, e, h, g

Name: ______________________________ Login: cs61c-____

M2) "Those are some big numbers you got there…" (10 pts, 20 min)

A bignum is a data structure designed to represent large integers. It does so by abstractly considering all of the bits in the `num` array as belonging to one very large integer. This code is run on a standard 32-bit MIPS machine, where a `word` (defined below) is 32 bits wide and `halfword` is 16 bits wide.

```
typedef unsigned int   word;
typedef unsigned short halfword;
typedef struct bignum_struct {
   int length;     // number of words
   word *num;      // the actual data
} bignum;
```

This function shows how bignums are used:

```
void print_bignum(bignum *b) {
   printf("0x"); // Print hex prefix
   for (int i = b->length-1; i>=0; i--)
      printf("%08x", b->num[i]);
}
```

a) Is the ordering of `word`s in the `num` array BIG or [✓ LITTLE] endian? (circle one)

b) How many bytes would be used in the *static*, *stack* and *heap* areas as the result of lines 1, 3 and 4 below? **Treat each line independently!** E.g., For line 3, don't count the space allocated in line 1.

```
1 bignum biggie;
2 int main(int argc, char *argv[]) {
3   bignum bigTriple[3], *bigArray[4];
4   bigArray[1] = (bignum *) malloc (sizeof(bignum) * 2);
```

	static	*stack*	*heap*
Line 1	8	0	0
Line 3	0	3*8 + 4*4 = 40	0
Line 4	0	0	2*8 = 16

b) Complete the `add` function for two bignums, which you may assume **are the same** `length`. Our C compiler translates `z = x + y` (where `x,y,z` are `words`) to `add` (not `addu`, as is customary) and thus could generate a hardware (HW) overflow we don't want, as we're running on untrusted HW. Your code should be written so that `word`s never overflow in HW (so we do all adding in the `halfword`).

```
void add(bignum *a, bignum *b, bignum *sum, word carry_in, word *carry_out) {

   // reserve space for num array. Remember a and b are the SAME length...

   sum->num =  (word *) malloc (a->length * sizeof(word))

   for (int i=0; i < a->length; i++) {  // word-by-word do addition of lo, hi halfwords

      // add lo halfwords of a,b

      word lo     = (a->num[i]&0xffff) + (b->num[i]&0xffff) + carry_in;

      // add hi halfwords of a,b (but in the safe, low halfword area so no HW overflow)

      word hi     = (a->num[i] >> 16 ) + (b->num[i] >> 16 ) + (lo >> 16);

      // combine low and hi halfwords (put back in their places), like a lui-ori

      sum->num[i] = (hi << 16) | (halfword) lo;

      // what's the carry_in for the next word?

      carry_in    = hi >> 16;
   }
   sum->length = a->length;
   *carry_out  = carry_in;
}
```

M3:MIPS->C

a)
```
char *foo (char *src, size_t size) {
    // forgetting sizeof(char) below is ok
    char *dest, *d, *end;
    dest = (char *) malloc ((size+1)*sizeof(char));

    for (d=dest,end=src+size; d != end; d++, src++) {
          *d = *src | 0x20;
    }

    *d = 0;
    return dest;
}
```

b) `strnlowercasecpy` (make lowercase)
We'll also accept a name that doesn't reference the size, like `strlowercasecpy`

c) Two possibilities, each equally valid

- Memory leak! (You call `malloc` but never free the space…).
- We don't check whether `malloc` will fail! (which ties into the previous reason; if you leak memory and call `printf("..",foo())` lots of times, eventually this error will come up. It comes up quicker if `size` is big!

d) Here are the things it could do

- Segmentation Fault (you run off the end of the string into an unallocated area)
- Prints the output of `foo` correctly
- Prints the output of `foo` followed by some garbage

F1:Datapath

`srjr $ra, $sp, 16`

a) R[rt] = R[rt] + (ZeroExt(Imm) << 2); PC = R[rs]

b) 256 kibi (16 unsigned 0xFFFF bits of words = 18 unsigned bytes)

c)

i. Add mux so `Ra` input is sometimes `Rs`, sometimes `Rt`, call the control signal `RegSrc`
ii. Modify Extender so that it can do a "`ZeroShiftExtend`", widen `ExtOp` control line

d)

- `RedDst=rt (0)`
- `RegWr=1`
- `nPC_sel=Jump`
- `ExtOp=ZeroShiftExtend`
- `ALUSrc=Extender (1)`
- `ALUctr=ADD`
- `MemWr=0`
- `MemtoReg=ALU (0)`
- `[NEW]RegSrc=Rt`

F4:SDS

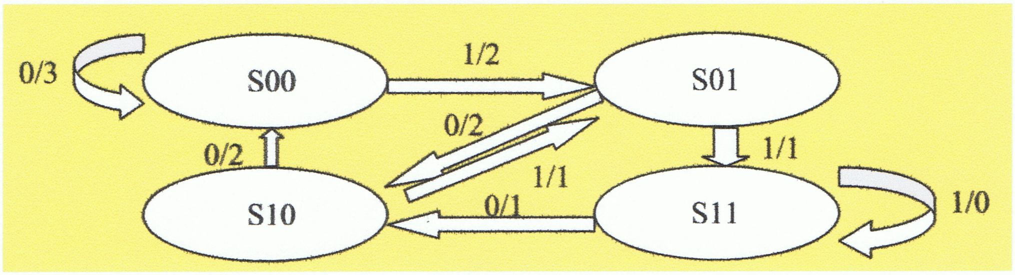
F4a) From `S00` we have two transition possibilities, `I=0` and `I=1`. I've felt it useful to think about the past values `I(t-2)`, `I(t-1)` and `I(t)` to figure out where to go. This is a simple box (a shift register) that keeps the last two values in the state variables `Sx` and `Sy`. Every step we output `~Sx` + `~Sy` + `~I` = `~P1` + `~P0` + `~I`. Also every step `N1=P0`, `N0=I`. We don't even need a truth table to know this – it's part of the definition of `Sxy`.

```
 PP I     OO  NN (Input/Output label for edge)   [#ZI(ABC) = NumberOfZerosIn(P1,P0,I)]
 10       10  10
---------------
S00 0 ➔ 11 S00 (0/3) # Had two 0s, another one means we stay here and output #ZI(000)=3
S00 1 ➔ 10 S01 (1/2) # This is our first 1 in a while, register we've seen a 1 by
                     # setting I(t-1) to 1 (i.e., S01) and output #ZI(001)=2
S01 0 ➔ 10 S10 (0/2) # Saw a 01 before but this 0 means we goto S10 and output #ZI(010)=2
S01 1 ➔ 01 S11 (1/1) # This is the 2nd 1 in a row, go to S11 and output #ZI(011)=1

S10 0 ➔ 10 S00 (0/2) # Saw a 1 2 timesteps ago, nothing since. Goto S00,output #ZI(100)=2
S10 1 ➔ 01 S01 (1/1) # Saw a 1 2 timesteps ago, a 1 now. Goto 01, output #ZI(101)=1

S11 0 ➔ 01 S10 (0/1) # Saw 2 straight 1s, now a 0. Goto S10, output #ZI(110)=1
S11 1 ➔ 00 S11 (1/0) # Everything is coming up 1s! Stay here (in S11), output #ZI(111)=0
```

F4b)



Fully reduced expressions for `O1`,`O0` and `N1`,`N0`, huh? Well, some are easier than others. We'll do the easier ones first. Looking at the truth table (not doing the mindless sum-of-products calculation), we see:

```
N0=I
N1=P0
```

Which we already knew from part (a)! There are no names for these circuits. Let's now look at `O1` and `O0`. If we're extremely clever, we remember the two bit patterns for an adder's two output bits: `O1` is a *minority circuit* and `O0` is *a 3-input xnor*. Let's see if we can figure that out even if we don't remember these facts. Let's study the truth table and look at the negative spaces (the times when the output is zero). We see when `P1` is 0 `O0` looks like `xnor(P0,I)` = `~(P0 ⊕ I)`. When `P1` is `1` `O0` looks `xor(P0,I)` = `(P0 ⊕ I)`. That is, `P0⊕I` is being *conditionally inverted* by `P1`, which is what an `xor` does! From this, we see that

`O0 = ~[P1⊕(P0⊕I)]`, i.e. the post-negation of two cascaded `xor`s, which is the same as a 3-input `xnor`!

`O1` is a little harder. We can still study the table and see some patterns. That is, when `P1` = `0`, `O1` looks like `nand(P0,I)` = `~(P0*I)`. When `P1=1`, `O1` is like a `nor(P0,I)` = `~(P0+I)`. This yields

$$\mathtt{O1} = \overline{\mathtt{P1}}*(\overline{\mathtt{P0*I}}) + \mathtt{P1}*(\overline{\mathtt{P0+I}})$$

$$= \overline{\mathtt{P1}}*(\overline{\mathtt{P0}}+\overline{\mathtt{I}}) + \mathtt{P1}*(\overline{\mathtt{P0}}*\overline{\mathtt{I}}) \quad \text{\# DeMorgan's law}$$

$$= \overline{\mathtt{P1}}\ \overline{\mathtt{P0}} + \overline{\mathtt{P1}}\ \overline{\mathtt{I}} + \mathtt{P1}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}} \quad \text{\# distribution}$$

Now it might look like this is minimal, but we can check two ways that it's not. First, there's symmetry to the bit patterns (the expression is true whenever at least two of the three components `P1`,`P0` or `I` are false) BUT there's *not* symmetry to the expression. Also, we can see that `~P0~I` yields a `1` in `O1` independent of `P1` from the truth table. We can also do some funky Boolean algebra…
Recall the following *distributive+law-of-1s+identity* simplification?

`A+AB = A(1+B) = A(1) = A`

Well, we can run it backwards. That is, we can start with **`A`** and generate **`A+AB`**.
We do that here with `~PI~P0`:

$$\overline{\mathtt{P1}}\ \overline{\mathtt{P0}} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}(1) = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}(1+\overline{\mathtt{I}}) = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}} + \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}}$$

So that means our *three* terms for `O1` are now *four*:

$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}} + \overline{\mathtt{P1}}\ \overline{\mathtt{I}} + \mathtt{P1}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}} \quad \text{\# \textit{from above}}$$

$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}} + \overline{\mathtt{P1}}\ \overline{\mathtt{I}} + \mathtt{P1}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}} + \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}} \quad \text{\# \textit{distributive+law-of-1s+identity}}$$

$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}} + \overline{\mathtt{P1}}\ \overline{\mathtt{I}} + (\mathtt{P1}+\overline{\mathtt{P1}})\overline{\mathtt{P0}}\ \overline{\mathtt{I}} \quad \text{\# \textit{distribution}}$$

$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}} + \overline{\mathtt{P1}}\ \overline{\mathtt{I}} + (\ \ 1\ \ )\overline{\mathtt{P0}}\ \overline{\mathtt{I}} \quad \text{\# \textit{complementarity}}$$

$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}} + \overline{\mathtt{P1}}\ \overline{\mathtt{I}} + \overline{\mathtt{P0}}\ \overline{\mathtt{I}} \quad \text{\# \textit{identity}}$$

$$\mathtt{O1} = \overline{(\mathtt{P1P0} + \mathtt{P1I} + \mathtt{P0I})} \quad \text{\# \textit{lots more Boolean algebra!}}$$

…a *NotMajority, or AntiMajority, or Minority circuit*!

We could also do this the standard plug-and-chug SoP (sum-of-products) way:

$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}} + \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}\ \mathtt{I} + \overline{\mathtt{P1}}\ \mathtt{P0}\ \overline{\mathtt{I}} + \mathtt{P1}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}} \quad \text{\# \textit{sum-of-products}}$$

$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}} + \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}\ \mathtt{I} + \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}} + \overline{\mathtt{P1}}\ \mathtt{P0}\ \overline{\mathtt{I}} + \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}} + \mathtt{P1}\ \overline{\mathtt{P0}}\ \overline{\mathtt{I}}$$
# rev idempotent, commutativity

$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}(\overline{\mathtt{I}}+\mathtt{I}) + \overline{\mathtt{P1}}\ \overline{\mathtt{I}}(\overline{\mathtt{P0}}+\mathtt{P0}) + \overline{\mathtt{P0}}\ \overline{\mathtt{I}}(\overline{\mathtt{P1}}+\mathtt{P1}) \quad \text{\# \textit{commutativity, rev distrib}}$$

$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}}(\ 1\ ) + \overline{\mathtt{P1}}\ \overline{\mathtt{I}}(\ \ 1\ \ ) + \overline{\mathtt{P0}}\ \overline{\mathtt{I}}(\ \ 1\ \ ) \quad \text{\# \textit{complementarity}}$$

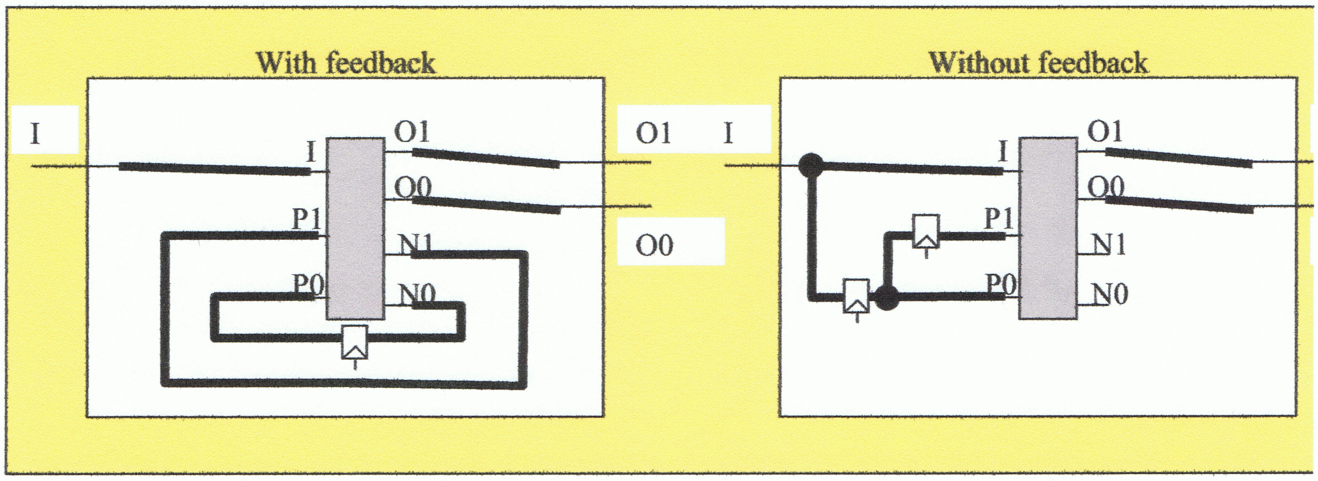
$$\mathtt{O1} = \overline{\mathtt{P1}}\ \overline{\mathtt{P0}} + \overline{\mathtt{P1}}\ \overline{\mathtt{I}} + \overline{\mathtt{P0}}\ \overline{\mathtt{I}} \quad \text{\# \textit{identity}}$$

$$\mathtt{O1} = \overline{(\mathtt{P1P0} + \mathtt{P1I} + \mathtt{P0I})} \quad \text{\# \textit{lots more Boolean algebra!}}$$
…a *NotMajority, or AntiMajority, or Minority circuit*!

F4c)

The feedback circuit is the standard synchronous digital systems model we've seen several times, where the output is passed through flip-flops and sent back to the input.

The non-feedback circuit we haven't seen before. However, from the problem description we know that `Sx` and `Sy` (i.e., `P1` and `P0`) are really just time-delayed versions of the inputs. I.e., `P0=I(t-1)` and `P1=I(t-2)`, we have the answer on the right.



With feedback
I
I
O1
O0
P1
N1
P0
N0
O1
O0
I
Without feedback
I
O1
O0
P1
N1
P0
N0