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 4x8=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 -∞, but signed `ints` move the other way from -∞ (-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

M2:C

a)

	static	stack	heap
1:	4+16+4+4=28 B	0	0
3:	0	28*2 + 2*4 = 64 B	0
4:	0	0	280 B

b) Two solutions…Longest (with the best style)

```
int Delete (slicenode_t *plan) {
  if(plan->type == RECTANGLE) { /* leaf */
    free(plan);
    return(1);
  }
  else {
    if plan->type == CUT) { /* inner node */
      slidenode_t *L, *R;
      L = plan->L; R = plan->R;
      free(plan);
      return(1+Delete(L)+Delete(R));
    } else {
      printf("Delete(): plan->type was %d, expected %d or %d", plan->type, RECT
CUT);
      exit(1);
    }
  }
}
```

… and longest!

```
int Delete (slicenode_t *plan) {
  if(plan->type == RECTANGLE) { /* leaf */
    free(plan);
    return(1);
  }
  else {
    slidenode_t *L, *R;
    L = plan->L;
    R = plan->R;
    free(plan);
    return(1+Delete(L)+Delete(R));
  }
}
```

… and shortest!

```
int Delete (slicenode_t *plan) {
  if(plan->type == RECTANGLE) {
    return(1+(0*free(plan)));
  }
  else { /* inner node */
    return(1+Delete(plan->L)+Delete(plan->R)+(0*free(plan)));
  }
}
```

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`

F2:Cache/VM

a) With 8-byte blocks (3 bits for offset) and a fully associative cache (0 bits for index), and a MIPS machine (32-bit addresses), we have `29:0:3`

b) The cache size, or "area" is the "height" (128 = 2^7 blocks) times the "width" (8 B/block = 2^3 B/block),
which is 2^{10} bytes i.e., 1 KibiByte.

c) To minimize cache misses, we should never stride so far that the initial `sum` loop couldn't fit entirely into the 128-entry cache. So the farthest we could stride is the *entire size of the cache*, or 1 KiB. Any stride smaller than that will also have the property that the `sum` loop has a miss for each block but the `product` loop has all hits.

d) Each block loaded by `sum` will be a miss, but `product`'s requests will be all hits. If the stride is 1 KibiB, that's 2^7 misses for each of the outer loop iterations, and there are 4 MiB / 1 KiB (= 4 Ki) = 2^{12} iterations. So that's 2^7 misses/iteration * 2^{12} iterations = 2^{19} misses = 512 KiMisses.

e) If we are not page-aligned, what will happen is that the last page request for `sum` will kick the first page out. As a result, the first page request for `product` won't be there, and we'll be charged a page miss! The problem propagates, unfortunately. That is, the first `product` page we just loaded will kick the second `sum` page out (since it's the last to be accessed) so when the second `product` page comes by it will *also* have a miss! So basically you have no cache savings at all! However, there's a small detail. When we shift our machinery down another stride and increment `i` (which we do a total of 2^{12} times), the last block loaded by the `product` loop will be the first one requested by the `sum` loop! (But that's a really small detail) So the answer is simple – *with* block alignment every sum is a miss and every product is a hit. Without block alignment every sum is a miss AND every product is a miss. Thus we double our misses.

f) Sure, Virtual and Physical address widths are independent. The VA width how much (virtual) memory our program thinks we have (here 32-bits of it) and PA controls how much resident memory we have, a completely independent quantity.

g) Sure, because we could have more resident pages (for the OS, other processes [either ours or another users]) and there'd be less thrashing.

h) The **`and`** instruction was the last instruction in the page and the **`or`** is the first instruction in the next page and we just experienced a page fault! Since pages are 4 KiB, the last instruction must have the LS bits be page size – 4B, so we know the last three nibbles (offset) are 0XFFC. (i.e., one more instruction and the offset becomes 0x000 – a new page!)

F3:Pipelining

a) Here is the chart

1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
1	F	D	E	M	W0														Start @ 1
2		F	D	D	D0	E	M	W1											Stall for $a0 to be writte before it can be read (nc forwarding)
3			F					D1	E	M	W1								Stall for $a1 to be writte before it can be read (nc forwarding)
4								F			D1	E	M	W					Stall for $a1 to be writte before it can be read
5											F	D	E	M	W1				Stall until the above inst finishes before I can finish (no out-of-order exec!)
6												F	D	E	M	W			No hazards; proceed as normal
7														F	D	E	M	W	Stall because we have non-delayed branches but we don't know which instr to take until after the 2nd stage

b) Here's the chart with the addition of forwarding and delayed branches

1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
1	F	D	E	M	W0														Start @ 1
2		F	D	E	M1	W													$a0 is forwarded from ALU->ALU
3			F		D1	E	M	W											Standard load delay stall because we need to wait for the data fror memory (to eventually be stored in $a1), which is forwarded to the ALU.
4					F	D1	E	M	W										The value of $a1 is forwarded to the memory's "data in" line
5						F	D	E	M	W									No hazards, proceed as normal
6							F	D	E	M	W								No hazards, proceed as normal
7								F	D	E	M	W							Branch-delay slot is filled, no need to stall

F4:SDS

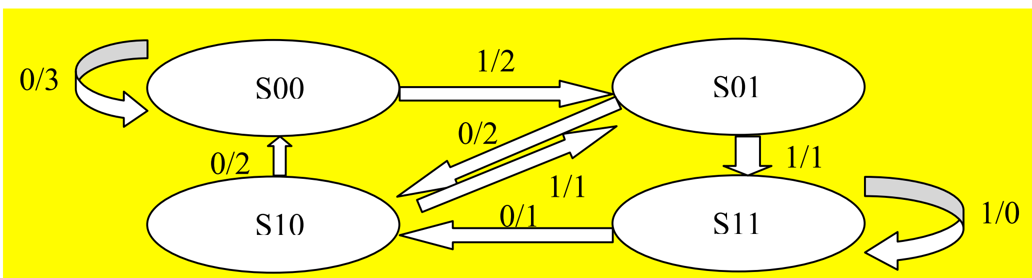
F4a) From `S00` we have two transition possibilities, `I=0` and `I=1`. I've felt it useful to think about the past values `I(t-2)`, `I(t-1)` and `I(t)` to figure out where to go. This is a simple box (a shift register) that keeps the last two values in the state variables `Sx` and `Sy`. Every step we 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

$$
\begin{aligned}
O1 &= \overline{P1}*(\overline{P0*I}) + P1*(\overline{P0+I}) \\
&= \overline{P1}*(\overline{P0}+\overline{I}) + P1*(\overline{P0}*\overline{I}) && \text{\# DeMorgan's law} \\
&= \overline{P1}\ \overline{P0} + \overline{P1}\ \overline{I} + P1\ \overline{P0}\ \overline{I} && \text{\# distribution}
\end{aligned}
$$

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`:

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

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

$$
\begin{aligned}
O1 &= \overline{P1}\ \overline{P0} + \overline{P1}\ \overline{I} + P1\ \overline{P0}\ \overline{I} && \textit{\# from above} \\
O1 &= \overline{P1}\ \overline{P0} + \overline{P1}\ \overline{I} + P1\ \overline{P0}\ \overline{I} + \overline{P1}\ \overline{P0}\ \overline{I} && \textit{\# distributive+law-of-1s+identity} \\
O1 &= \overline{P1}\ \overline{P0} + \overline{P1}\ \overline{I} + (P1+\overline{P1})\overline{P0}\ \overline{I} && \textit{\# distribution} \\
O1 &= \overline{P1}\ \overline{P0} + \overline{P1}\ \overline{I} + (\ 1\ )\overline{P0}\ \overline{I} && \textit{\# complementarity} \\
O1 &= \overline{P1}\ \overline{P0} + \overline{P1}\ \overline{I} + \overline{P0}\ \overline{I} && \textit{\# identity} \\
O1 &= \overline{(P1P0 + P1I + P0I)} && \textit{\# lots more Boolean algebra!}
\end{aligned}
$$

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

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

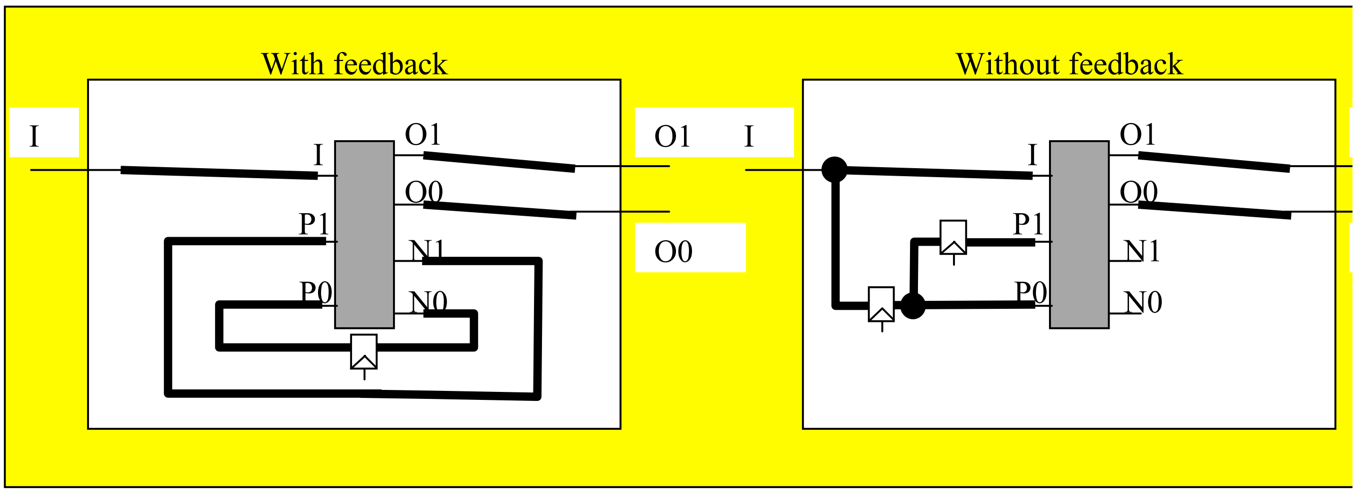
$$
\begin{aligned}
O1 &= \overline{P1}\ \overline{P0}\ \overline{I} + \overline{P1}\ \overline{P0}\ I + \overline{P1}\ P0\ \overline{I} + P1\ \overline{P0}\ \overline{I} && \textit{\# sum-of-products} \\
O1 &= \overline{P1}\ \overline{P0}\ \overline{I} + \overline{P1}\ \overline{P0}\ I + \overline{P1}\ \overline{P0}\ \overline{I} + \overline{P1}\ P0\ \overline{I} + \overline{P1}\ \overline{P0}\ \overline{I} + P1\ \overline{P0}\ \overline{I} \\
& && \textit{\# rev idempotent, commutativity} \\
O1 &= \overline{P1}\ \overline{P0}(\overline{I}+I) + \overline{P1}\ \overline{I}(\overline{P0}+P0) + \overline{P0}\ \overline{I}(\overline{P1}+P1) && \textit{\# commutativity, rev distrib} \\
O1 &= \overline{P1}\ \overline{P0}(\ 1\ ) + \overline{P1}\ \overline{I}(\ 1\ ) + \overline{P0}\ \overline{I}(\ 1\ ) && \textit{\# complementarity} \\
O1 &= \overline{P1}\ \overline{P0} + \overline{P1}\ \overline{I} + \overline{P0}\ \overline{I} && \textit{\# identity} \\
O1 &= \overline{(P1P0 + P1I + P0I)} && \textit{\# lots more Boolean algebra!}
\end{aligned}
$$

…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.



F5:Potpourri

a) One of 9. The lesson was [debug & test rigorously as if lives depend, expect the unexpected. Design with failure as a possibility. Add redundancy]
   1. Mariner I space probe
   2. Soviet gas pipeline
   3. Buffer overflow in Unix finger daemon
   4. Kerberos Random # generator
   5. AT&T network outage
   6. Intel Pentium floating pt
   7. Ping of death
   8. Ariane 5 Flight 501
   9. National Cancer Institute

b) SPUR: Security, Privacy, Usability, Reliability

c) What are the constraints on the timing?
To maintain t_{setup} time constraints (and starting from the rising edge of a clock), we have the usual equation that helps us determine how fast we can run the clock. This is that the signal, from when it leaves the FF, goes through all the gates, until it comes around again, has to arrive on the inputs earlier than t_{setup} *before* the next clock rising edge. Thus, we have:
$t_{clk\text{-}to\text{-}q}, + t_{inverter} + t_{or} + t_{setup} < t_{clock}$
... and to maintain t_{hold} time constraints, which state that the signal *cannot* get back around and change *before* (less time) the hold time t_{hold} has passed, yields the following constraint:
$t_{clk\text{-}to\text{-}q}, + t_{inverter} + t_{or} > t_{hold}$
So, isolating for $t_{inverter}$ in both of these inequalities yields the constraints:
$t_{hold} - (t_{clk\text{-}to\text{-}q}, + t_{or}) < t_{inverter} < t_{clock} - (t_{clk\text{-}to\text{-}q}, + t_{or} + t_{setup})$

d) How fast do branches for B need to be? Well, let's figure out the equations:

$$CPUtime_A = CPUtime_B \text{ [1]}$$

But we know the equation for CPUtime as:

$$CPUtime = InstructionCount * CPI * ClockTime \text{ [2]}$$

So substituting that into [1] gives us

$$InstructionCount_A * CPI_A * ClockTime_A = InstructionCount_B * CPI_B * ClockTime_B \text{ [3]}$$

But since it's the same program,

$$InstructionCount_A = InstructionCount_B \text{ [4]}$$

Equation [3] now simplifies to:

$$CPI_A * ClockTime_A = CPI_B * ClockTime_B \text{ [5]}$$

And substituting

$$ClockTime_i = 1/ClockFreq_i \text{ [6]}$$

into [5] gives

$$CPI_A / ClockFreq_A = CPI_B / ClockFreq_B \text{ [7]}$$

So solving for CPI_B:

$$CPI_B = ClockFreq_B / ClockFreq_A (CPI_A) \text{ [8]}$$

$$CPI_B = 4/2\ CPI_A = 2 * CPI_A$$

So now we only have to solve for CPI_A and CPI_B from the table:
$CPI_A = 2(2/10) + 2(3/10) + 2(5/10) = (4+6+10)/10 = 20/10 = 2$ cycles/instruction
$CPI_B = 1(2/10) + 1(3/10) + X(5/10) = (2+3+5X)/10 = (5+5X)/10 = 4$ cycles/instruction

Solving for X yields
$5+5X=40 \Rightarrow 5X=35 \Rightarrow X = 7$

e) $\lg(16 \text{ exbi}) = \lg(2^{64}) = 64$ bits total. $\lg(12_{10}\text{x}2^{10}) = \lg(2^{14}) = 14$ MSBs. $\lg(200{,}000{,}000) = \lg(2^{28}) = 28$ LSBs. Therefore we have 64-14-28=50-28=22 bits left, which can encode 4 mebithings.

f) How much could we store? Well, here is the standard equation:

Capacity(B) = Density(B/in^2) * Area/Surface (in^2/Surf) * SurfacesPerPlatter (Surf/Plat) * #Platters (Plat)

And we're given

Density = ? B/in^2
#Platters = 4 Plat
SurfacesPerPlatter = 2 Surf/Plat (if we want to maximize capacity, we use BOTH sides!)
Area/Surface = area of the disk = $\pi r^2_{outer} - \pi r^2_{inner} = \pi (30/\pi - 22/\pi) = 8$ in^2/Surf
Thus, ? Gibi (B/in^2) * 8 (in^2/Surf) * 2 (Surf/Plat) * 4 (Plat) = ? $2^3 2^1 2^2$ B = 2^{40} B = 1 TebiByte, so ? = 2^{24} B/in^2 = 16 GiB/in^2

g) 0: 32 TebiB, 1: 16 TebiB, 3: 31 TebiB, 5: 31 TebiB