Name Solution

Computer Organization
EE 3755
Midterm Examination

Wednesday, 24 October 2012, 9:30–10:20 CDT

Problem 1		(15 pts)
Problem 2		(14 pts)
Problem 3		(14 pts)
Problem 4		(14 pts)
Problem 5		(18 pts)
Problem 6		(11 pts)
Problem 7		(14 pts)
Exam Total		(100 pts)

Alias A Century of Turing

Good Luck!

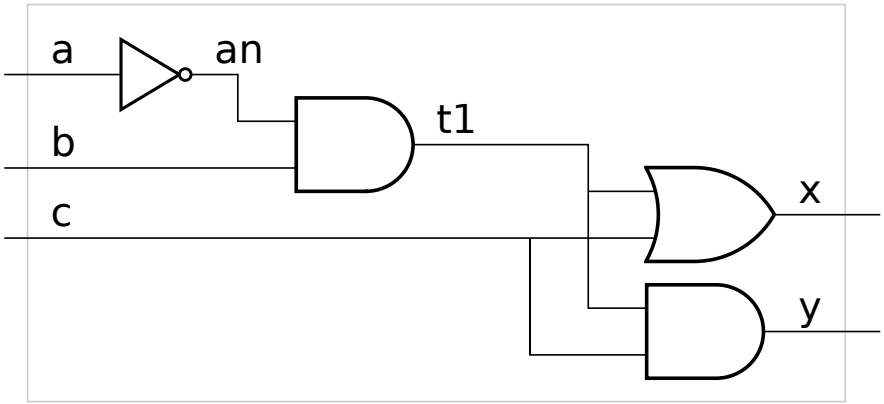
Problem 1: [15 pts] Consider the module below:

```
module module1(x,y,a,b,c);
   input a, b, c;    output x, y;
   wire   t1, an;

   not n1(an, a);
   and a1(t1, an, b);
   or  o1(x, t1, c);
   and a2(y, t1, c);
endmodule
```

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

☑ Logic diagram of `module1`.

Solution appears below.



(*b*) Complete the module below so that it performs the same operation as `module1` but is in implicit structural form.

☑ Implicit structural form of `module1`.

```
module module1(x,y,a,b,c);
   input a, b, c;    output x, y;

 // SOLUTION

 assign x = !a && b || c;

 assign y = !a && b && c;



endmodule
```

Problem 2: [14 pts] Appearing below is a Verilog description of a two-level adder used in Homework 2.

```
module adder_r4_c3(sum,a,b);
   input [11:0]  a, b;                    output [12:0] sum;
   wire [2:0]    P, G, carry, CO;

   ripple_4_block ad0(sum[3:0],  CO[0], a[3:0],  b[3:0],  carry[0]);
   ripple_4_block ad1(sum[7:4],  CO[1], a[7:4],  b[7:4],  carry[1]);
   ripple_4_block ad2(sum[11:8], CO[2], a[11:8], b[11:8], carry[2]);

   gen_prop_4 gp0(G[0], P[0], a[3:0],  b[3:0]);
   gen_prop_4 gp1(G[1], P[1], a[7:4],  b[7:4]);
   gen_prop_4 gp2(G[2], P[2], a[11:8], b[11:8]);

   assign      carry[0] = 1'b0;
   assign      carry[1] = G[0];
   assign      carry[2] = G[0] & P[1] | G[1];
   assign      sum[12]  = G[0] & P[1] & P[2] | G[1] & P[2] | G[2];
endmodule
```

(*a*) Suppose that the input to the adder were `a=0x123` and `b=0xabc`. Show the values on the wires indicated below: *Hint: If you don't remember the formulas for* `P` *and* `G`*, try to remember what they are supposed to do.*

☑ Wire `a[3:0]` Solution: $\boxed{= 3_{16} = 0011_2}$

☑ Wire `b[7:4]` Solution: $\boxed{= b_{16} = 1011_2}$

☑ Wire `P[2]`, `P[1]`, and `P[0]` Solution: `P[2]=0`, `P[1]=0`, and `P[0]=1`.

☑ Wire `G[2]`, `G[1]`, and `G[0]` Solution: `G[2]=0`, `G[1]=0`, and `G[0]=0`.

For the first two parts above, pay attention to the bit positions.

The upper-case `P` propagate signal is `1` if a carry-in to a block, such as **`ripple_4_block ad1`**, will propagate all the way though the block and emerge as a carry out. That will happen if all the lower-case `p` propagate signals are `1`. Recall that `p[i] = a[i] or b[i]`. Therefore `P[0]=1` and the others are zero.

The generate signal is 1 if a block will have a carry out whether or not there is a carry in. The easy way to determine that for a block is to add their inputs and see if the result is strictly greater than 15. For example, for the block **`ad0`** the inputs are $a = 3_{16}$ and $b = c_{16}$. Their sum is $3_{16} + c_{16} = f_{16}$, and so their is no overflow. Neither is their overflow for blocks **`ad1`** and **`ad2`**, and so all of the generate bits are zero.

To see an example where a generate bit is 1, consider inputs $a = 572_{16}$ and $b = abc_{16}$ then the propagate bits would be 110_2 and the generate bits would be 010_2.

Problem 3: [14 pts] Appearing below is Verilog code for a carry lookahead adder which computes the carry signals two ways. First is the conventional Old way, the second computes the same values for the carry signals but the Verilog expression is much shorter. (Code for computing `carry0` to `carry3` is not shown.) An actual module would use one of these ways, not both.

```
// Old Way
assign carry4 = g0 & p1 & p2 & p3  |  g1 & p2 & p3  |  g2 & p3  |  g3;
assign carry5 = g0 & p1 & p2 & p3 & p4  |  g1 & p2 & p3 & p4
                | g2 & p3 & p4  |  g3 & p4  |  g4;

// "New" Way:
assign carry4 = carry3 & p3 | g3;
assign carry5 = carry4 & p4 | g4;
```

(*a*) Compute the amount of time it will take to compute `carry5` using the Old and "New" ways. The `p` and `g` signals are available at $t = 0$. All gates have a delay of one time unit, regardless of the number of inputs. Don't forget to account for `carry0`, `carry1`, ..., `carry3` when necessary, they **are not** available at $t = 0$. Show work or explain your answers.

☑ Time for `carry5` using Old way: Solution: 2 time units.

In the old way there is a five-input OR gate fed by four AND gates of various sizes. The inputs to these gates are all `p` and `g` signals, which are available at $t = 0$. The longest path a signal has to take is through two gates and so the `carry5` signal will be available after 2 time units.

☑ Time for `carry5` using "New" way:

Once `carry4` is available, `carry5` can be computed in two time units, one for the AND gate and one for the OR gate. The `carry4` signal is available 2 units after `carry3`, and so on down to `carry1`. The `carry0` signal would be the carry-in input to the circuit so its reasonable to assume that it's available at $t = 0$. Therefore, using a two-time-unit delay each for `carry1` through `carry5`, the `carry5` signal will not be available until 10 time units.

(*b*) There is an advantage to "New" way (and it's not performance). However, *"New" way* is not the best name.

☑ Explain the advantage of "New" way.

In the "New" way each carry signal is computed using only two gates, fewer than the number of gates needed for the Old way. Therefore the advantage is lower cost.

Grading Note: Many only answered that the advantage is the Verilog is shorter and more readable. That's true, however there is also a significant difference in the underlying logic, and that had to be mentioned for full credit.

☑ Calling it the *"New"* way of building a carry lookahead adder is misleading because...

A carry lookahead adder is supposed to compute the carry signals quickly by looking ahead at all of the inputs at once, rather than waiting for a carry signal to arrive after passing, or rippling, through the logic for the less significant bits, as happens in a ripple adder.

Because in the "New" way carry i must wait for carry $i-1$ it operates just like a ripple adder, sharing the performance disadvantage (because the carry signals take longer to arrive) and the cost advantage. Therefore, rather than being a new kind of carry lookahead adder, the "New" way is just a minor variation on the ripple adder.

Problem 4: [14 pts] Appearing below is the streamlined multiplier used in Verilog note set 7. Suppose that the multiplier is used with `multiplicand=0x8a` and `multiplier=0x85`. In the table below show the values in registers `bit` and `product` when execution reaches the indicated places in the code. The table is already filled in for the `INIT` block row, complete the four `PP` rows. Note that it takes 16 iterations to compute the product, so don't expect `product` to be $8a_{16} \times 85_{16} = 47b2_{16}$ in the fourth `PP` row.

```
module streamlined_mult(product,ready,multiplicand,multiplier,start,clk);
   input [15:0]  multiplicand, multiplier;
   input         start, clk;
   output        product, ready;
   reg [31:0]    product;            reg [4:0]     bit;
   wire          ready = !bit;
   initial bit = 0;

   always @( posedge clk )
     if ( ready && start ) begin:INIT   //   <-  THE INIT BLOCK
        bit     = 16;
        product = { 16'd0, multiplier };
        //  VALUES SHOWN WHEN EXECUTION REACHES HERE, IN THE INIT BLOCK
     end else if ( bit ) begin:PP      //   <-  THE PP BLOCK
        reg lsb;
        lsb     = product[0];
        product = product >> 1;
        bit     = bit - 1;
        if ( lsb ) product[31:15] = product[30:15] + multiplicand;
        //  SHOW VALUES WHEN EXECUTION REACHES HERE, IN THE PP BLOCK
     end
endmodule
```

☑ Show values for `bit` and `product` in the four `PP` rows table below.

location	bit	product
INIT	16	0x 0000 0085
		SOLUTION
PP	15	0x 0045 0042
PP	14	0x 0022 8021
PP	13	0x 0056 4010
PP	12	0x 002b 2008

Problem 5: [18 pts] Answer the computer arithmetic questions below.

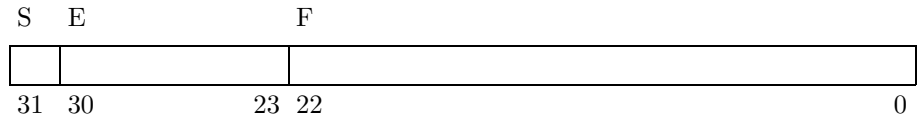
(*a*) Show the longhand steps needed to multiply $6c_{16} \times 39_{16}$ using a radix-4 (two bit) multiplication algorithm, but do not add together the partial products. Show the work in binary or hexadecimal. This is **without** Booth recoding. (The product is $180c_{16}$, but remember there is no need to add the partial products.)

☑ Longhand steps for radix-4 (2 bit) $6c_{16} \times 39_{16}$:

SOLUTION:

```
          0110 1100  = 0x6c
        * 0011 1001  = 0x39
          ---------
          0110 1100  =  1 * 1 * 0x6c
       011 0110 0000 =  4 * 2 * 0x6c
   1 0100 0100 0000  = 16 * 3 * 0x6c
   ----------------
   1 1000 0000 1100  = 0x180c        <- Not required for full credit.
```

(*b*) Show the value of IEEE 754 single-precision floating point number `0x 4181 0000`. For your convenience the layout of an IEEE 754 single is shown below. *Note: In the original version of this exam, the encoded number was* `0x 418 1000`*, which was much smaller than intended.*



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

Two solutions are shown, first the one to the number intended, the second to the one given on the original exam.

First, split `0x 4181 0000` into sign, exponent, significand: `0x 4181 0000` $\longrightarrow$ `0 10000011 00000010000000000000000`. The biased exponent is $1000\,0011_2 = 131$. Subtracting the bias for an IEEE single number, 127, gives an exponent of $131-127 = 4$. The fraction is obtained by prepending a `1.` to the significand, yielding 1.0000001_2 (with the trailing 0's omitted). The value is $1.0000001_2 \times 2^4 = (1 + 2^{-7}) \times 2^4 = 16.125_{10}$. The same value can be obtained by treating the significand as an integer: $\left(1 + \frac{00000010000000000000000_2}{2^{23}}\right) 2^4$.

The original question asked for a conversion of `0x 418 1000`. Those who mis-read it as `0x 4181 0000` were graded based on the answer above. Here is the solution for `0x 418 1000`.

`0x 418 1000` $\longrightarrow$ `0 00001000 00110000001000000000000`. The biased exponent is $1000_2 = 8$, subtracting the bias yields the exponent $8-127 = -119$. The fraction is $1.00110000001_2 \approx 1.18799_{10}$ or $1 + \frac{10000001000000000000_2}{2^{23}}$. Applying the exponent yields the value $1.00110000001_2 \times 2^{-119} \approx 1.78749 \times 10^{-36}$.

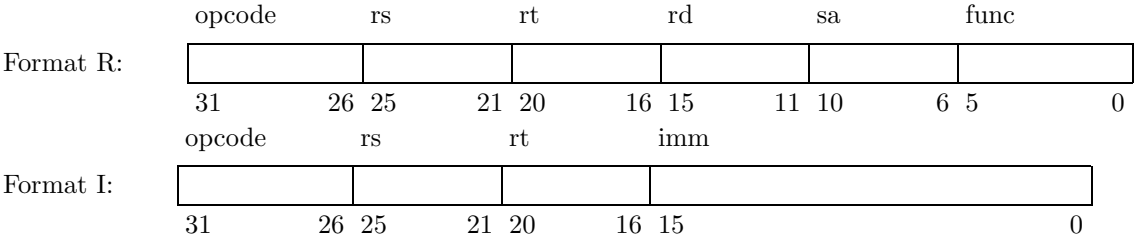
Problem 6: [11 pts] Show the values in register `$s0` after the execution of each instruction below in the spaces indicated.

☑ Fill in the `s0 =` blanks below.

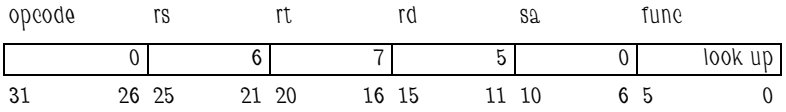
```
 # Initial register values:  $s1 = 10,  $s2 = 20

add $s0, $s1, $s2
                    # $s0 = SOLUTION: 30

and $s0, $s1, $s2
                    # $s0 = SOLUTION: 0

sll $s0, $s1, 2
                    # $s0 = SOLUTION: 40 = 101000 = 0x28

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

addi $s0, $s1, 30
                    # $s0 = SOLUTION: 40

lui $s0, 0x4321
                    # $s0 = SOLUTION: 0x 4321 0000
```

Problem 7: [14 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.

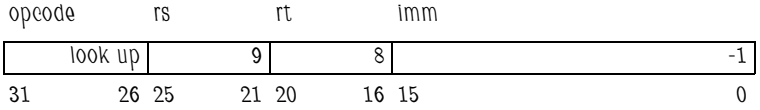


☑ Encoding for `add $5, $6, $7`



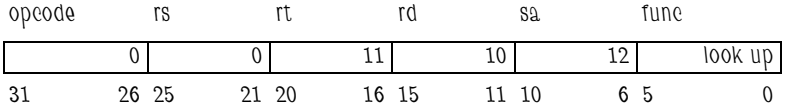
We assume that the opcode is zero because it is a type-R instruction. The `sa` field (shift amount) is zero because the field is unused by the instruction and the MIPS ISA definition usually sets unused fields to zero (preserving them for future use). The `func` field is essentially a continuation of the `opcode` field for format R, and there is no way of knowing the value for `add` short of memorization, so the correct answer is *look up*. For the record, the actual value is `0x20`.

☑ Encoding for `addi $8, $9, -1`



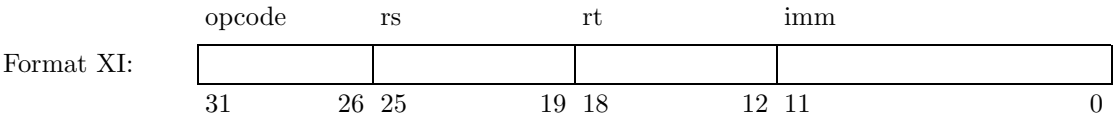
Note that the destination (r8) is specified in the `rt` field and the source (r9) is specified in the `rs` field. The `opcode`, which was not required for full credit, is `0x08`.

☑ Encoding for `sll $10, $11, 12`



The `sll` instruction uses format R and is one of the few instructions to use the `sa` field. Note that the register source operand (r11) is the `rt` field. Full credit was given for a `func` field value of *look up*, though some might have remembered that `sll 0,0, 0` is the "official" expansion of the `nop` pseudo instruction because it's encoding is all zeros, including the `func` field.

(*b*) Shown below is instruction format XI, a format for XIPS, an instruction set similar to MIPS, but with significant differences. Describe two differences of XIPS from MIPS in terms directly relevant to an assembly language programmer. That is, don't just answer that the `rs` and `rt` fields are two bits larger than in MIPS.



☑ Difference of XIPS relevant to a assembly language programmer:

In both XIPS and MIPS an instruction is 32 bits, but in XIPS the `imm` field is four bits shorter and the register fields are each two bits longer.

For an assembly language programmer this means that code using constants of size 13 to 16 bits can use immediate instructions on MIPS (such as `andi r1, r2, 0xfecb`) but on XIPS such code would have to use multiple instructions, perhaps two or three to place the constant in a register followed by the operation. See the example below. This difference is clearly a disadvantage of XIPS.

```
# MIPS Code
andi r1, r2, 0xfecb

# Equivalent XIPS Code, without ‘‘new’’ instructions.
ori r3, r0, 0xfec
sll r3, r3, 4
ori r3, r3, 0xb
andi r1, r2, r3

# Equivalent XIPS Code, with a reasonable new instruction: lmi.
lmi r3, 0xf   # Load Middle Immediate. Loads bits 23:12
ori r3, r3, 0xecb
andi r1, r2, r3
```

An advantage of XIPS for an assembly language programmer is that there are presumably four times as many registers (since the register fields are two bits larger). That's an advantage for code generating many intermediate values. Suppose some piece of code required 100 intermediate values. On XIPS all could be placed in registers. On MIPS one would have to use store (such as `sw`) and load (such as `lw`) instructions to *spill* and *fill* these values in and out of memory, adding to code size and execution time. See the example below.

```
# XIPS Code
add r100, r101, r102

# Equivalent MIPS Code
sw r10, 400(r29)   # Save value of r10 in stack to make room for ...
lw r10, 440(r29)   # ... value needed by the add instruction.
sw r15, 200(r29)   # Save r15 on the stack to make room for add's result.
add r15, r8, r10
```