University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 1999 John Kubiatowicz

Midterm I

SOLUTIONS

March 3, 1999

CS152 Computer Architecture and Engineering

Your Name:	
SID Number:	
Discussion Section:	

Problem	**Possible**	**Score**
1	15	
2	15	
3	20	
4	20	
5	30	
Total		

Problem 1: Performance

Problem 1a:
Name the three principle components of runtime that we discussed in class. How do they combine to yield runtime?

Three components: *Instruction Count, CPI, and Clock Period (or Rate)*

$$\textbf{Runtime} = \textbf{Inst Count} \times \textbf{CPI} \times \textbf{Clock period}$$
$$= \frac{\textbf{Inst Count} \times \textbf{CPI}}{\textbf{Clock Rate}}$$

Now, you have analyzed a benchmark that runs on your company's processor. This processor runs at 300MHz and has the following characteristics:

Instruction Type	Frequency (%)	Cycles
Arithmetic and logical	40	1
Load and Store	30	2
Branches	20	3
Floating Point	10	5

Your company is considering a cheaper, lower-performance version of the processor. Their plan is to remove some of the floating-point hardware to reduce the die size.

The wafer on which the chip is produced has a diameter of 10cm, a cost of $2000, and a defect rate of 1 / (cm^2). The manufacturing process has an 80% wafer yield and a value of 2 for α. Here are some equations that you may find useful:

$$\text{dies/wafer} = \frac{\pi \times (\text{wafer diameter}/2)^2}{\text{die area}} - \frac{\pi \times \text{wafer diameter}}{\sqrt{2 \times \text{die area}}}$$

$$\text{die yield} = \text{wafer yield} \times \left(1 + \frac{\text{defects per unit area} \times \text{die area}}{\alpha}\right)^{-\alpha}$$

The current procesor has a die size of 12mm × 12mm. The new chip has a die size of 10mm ×10mm, and floating point instructions will take 12 cycles to execute.

Problem 1b:
What is the CPI and MIPS rating of the original processor?

CPI = .4×1+ .3×2 + .2×2 + .1×5 = 2.1 cycles/inst

MIPS= $\frac{\textbf{300}\boldsymbol{MHz}}{\textbf{2.1}}$ *= 143 MIPS*

Problem 1c:
What is the CPI and MIPS rating of the new processor?

CPI = .4×1+.3×2+.2×3+.4×12= 2.8 cycles/inst

$$MIPS = \frac{\mathbf{300}\boldsymbol{MHz}}{\mathbf{2.8}} = 107\ MIPS$$

Problem 1d:
What is the original cost per (working) processor?

$$\textbf{dice/wafer} = \frac{\pi \times (\textbf{100mm/2})^{\mathbf{2}}}{\textbf{144mm}^{\mathbf{2}}} - \frac{\pi \times \textbf{100mm}}{\sqrt{\mathbf{2} \times \textbf{144mm}^{\mathbf{2}}}} = \mathbf{36.03} \Rightarrow \textbf{36 dice/wafer}$$

$$\textbf{die yield} = \mathbf{.8} \times \left(\mathbf{1} + \frac{\mathbf{1/cm}^{\mathbf{2}} \times \mathbf{1.44cm}^{\mathbf{2}}}{\mathbf{2}} \right)^{-\mathbf{2}} = \mathbf{0.27}$$

$$\textbf{good dice/wafer} = \mathbf{36} \times \mathbf{0.27} = \mathbf{9.7} \Rightarrow \textbf{9 good dice/wafer}$$

$$\textbf{cost/die} = \frac{\mathbf{\$2000}}{\mathbf{9}} = \mathbf{\$222.22/die}$$

Problem 1e:
What is the new cost per (working) processor?

$$\textbf{dice/wafer} = \frac{\pi \times (\textbf{100mm/2})^{\mathbf{2}}}{\textbf{100mm}^{\mathbf{2}}} - \frac{\pi \times \textbf{100mm}}{\sqrt{\mathbf{2} \times \textbf{100mm}^{\mathbf{2}}}} = \mathbf{56.33} \Rightarrow \textbf{56 dice/wafer}$$

$$\textbf{die yield} = \mathbf{.8} \times \left(\mathbf{1} + \frac{\mathbf{1/cm}^{\mathbf{2}} \times \mathbf{1cm}^{\mathbf{2}}}{2} \right)^{-2} = 0.355$$

$$\textbf{good dice/wafer} = 5\mathbf{6} \times \mathbf{0.355} = 19.9 \Rightarrow 19\ \textbf{good dice/wafer}$$

$$\textbf{cost/die} = \frac{\mathbf{\$2000}}{\mathbf{19}} = \$105.3/\textbf{die}$$

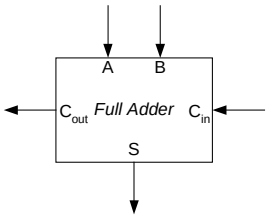
Problem 1f:
What is the improvement (if any) in price per performance?

$$\%\ \textbf{improvement} = \frac{(\textbf{price/perform})_{\boldsymbol{old}} - (\textbf{price/perform})_{\boldsymbol{new}}}{(\textbf{price/perform})_{\boldsymbol{old}}} = \frac{1.55 - 0.98}{1.55} = .37$$

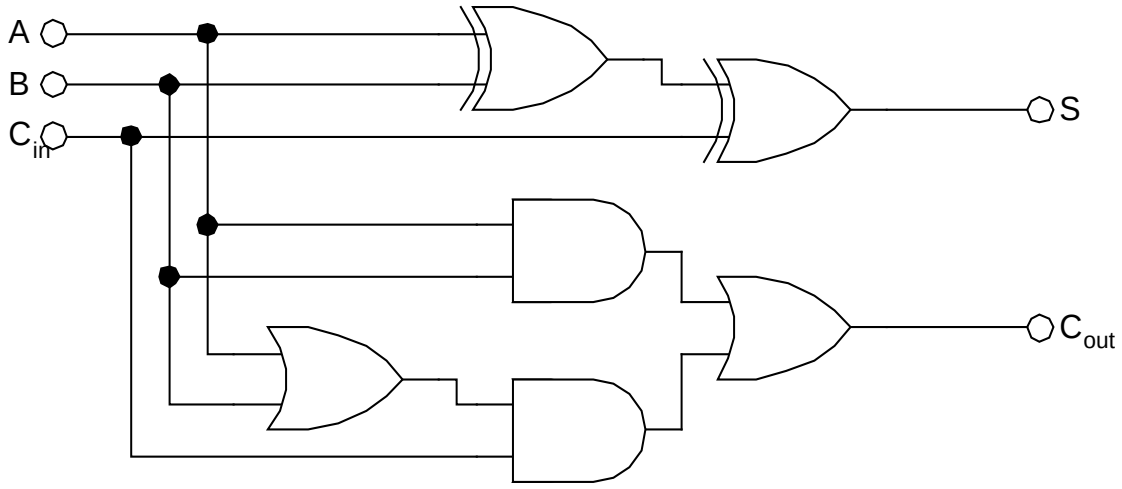
So, 39% improvement. Note that we took most reasonable solutions that took ratios of the old and new price-performance ratios.

Problem 2: Delay For a Full Adder

A key component of an ALU is a full adder. A symbol for a full adder is:



'Problem 2a:
Implement a full adder using as few 2-input AND, OR, and XOR gates as possible. Keep in mind that the Carry In signal may arrive much later than the A or B inputs. Thus, optimize your design (if possible) to have as few gates between Carry In and the two outputs as possible:



Assume the following characteristics for the gates:

AND: Input load: 150fF,
Propagation delay: TPlh=0.2ns, TPhl=0.5ns,
Load-Dependent delay: TPlhf=.0020ns, TPhlf=.0021ns

OR: Input load: 100fF
Propagation delay: TPlh=0.5ns, TPhl=0.2ns
Load-Dependent delay: TPlhf=.0020ns, TPhlf=.0021ns

XOR: Input load: 200fF,
Propagation delay: TPlh=.8ns, TPhl=.8ns
Load-Dependent delay: TPlhf=.0040ns,TPhlf=.0042ns

Problem 2b:
Compute the input load for each of the 3 inputs to your full adder:

Input Load$_A$ = (150 + 100 + 200) fF = 450 fF
Input Load$_B$ = (150 + 100 + 200) fF = 450 fF
Input Load$_{Cin}$ = (150 + 200) fF = 350 fF

Problem 2c:
Identify two critical paths from the inputs to the Sum and the Carry Out signal. Compute the propagation delays for these critical paths based on the information given above. (You will have 2 numbers for each of these two paths):

Critical path to Sum is from either A or B to Sum. Since the High⇒Low transition is slowest for the XOR gate, we will choose the value of the C_{in} signal so that the XOR gate tying A and B together goes from high to low:

TPlh = 0.8 + 0.0042 ×200 + 0.8 = 2.44 ns
TPhl = 0.8 + 0.0042 ×200 + 0.8 = 2.44 ns

Critical path for Carry Out signal is also from A or B:

TPlh = 0.5ns + 0.0020 ×150 + 0.2ns + .0020×100 + .5ns= 1.7 ns
TPhl = 0.2 ns +0.0021 ×150 + 0.5ns + .0021×100 + .2ns= 1.425 ns

Problem 2d:
Compute the Load Dependent delay for your two outputs.

This is easy: it is just equal to the LDD of the output gate:

TPlhf$_{SUM}$ = 0.0040. TPhlf$_{SUM}$ =0.0042
TPlhf$_{Cout}$ = 0.0020. TPhlf$_{SUM}$ = 0.0021

Problem 3: Division

Here is pseudo-code for an ***unsigned*** division algorithm. It is essentially the last divider (#3) that we developed in class. Assume that **quotient** and **remainder** are 32-bit global values, and the inputs **divisor** and **dividend** are also 32 bits.

```
divide(dividend, divisor)
{  int count;

   /* Missing initialization instructions. */

   ROL64(remainder,quotient,0);
   while (count > 0) {
      count--;
      if (remainder ≥ divisor) {
         remainder = remainder – divisor;
         temp = 1;
      } else {
         temp = 0;
      }
      ROL64(remainder,quotient,temp);
   }
   /* Something missing here */
}
```

The `ROL64(hi,lo,inbit)` pseudo-instruction treats `hi` and `lo` together as a 64-bit register. Assume that `inbit` contains *only* 0 or 1. ROL64 shifts the combined register left by one position, filling in the single bit from `inbit` into the lowest bit of lo.

Problem 3a [3 pts]:
Implement ROL64 as 5 MIPS instructions. Assume that $t0, $t1, and $t2 are the arguments.
Hint: what happens if you use signed slt on unsigned numbers?

```
ROL64:   slt    $t3, $t1, $zero     ; Put t1[31] => t3
         sll    $t0, $t0, 1         ; Shift MSW
         or     $t0, $t0, $t3 ; Or in lowest bit
         sll    $t1, $t1, 1         ; Shift LSW
         or     $t1, $t1, $t2 ; Or in lowest bit
```

Problem 3b [4 pts]:
This divide algorithm is incomplete. It is missing some initialization and some final code. What is missing?

```
Init: Count = 32; Quotient = Dividend; Remainder = 0;
Final: Remainder >>= 1;

/* The following is needed to catch case in which high bit of remainder
   is set.  This is only possible if high-bit of divisor is set &
   quotient is zero.  Very special case: We didn't require this! */

if (quotient == 0) Remainder = dividend;
```

Problem 3c:
Assume that you have a MIPS processor that is missing the divide instruction. Implement the above divide operation as a procedure. Assume **dividend** and **divisor** are in $a0 and $a1 respectively, and that **remainder** and **quotient** are returned in registers $v0 and $v1 respectively. You can use ROL64 as a pseudo-instruction that takes 3 registers. Don't use any other pseudo-instructions, however. Make sure to adher to MIPS register conventions, and optimize the loop as much as possible.

Solution:

```
1)  divide:     ori   $t4, $zero, 32        ;Initialize count
2)              ori   $v0, $zero, 0         ;Rem = 0
3)              add   $v1, $a0, $zero       ;Quotient = Dividend
4)              ROL64 $v0, $v1, $zero
5)  divloop:    addi  $t4, $t4, -1          ;Decrement count
6)              sltu  $t5, $v0, $a1         ;Check: Rem < Divisor ?
7)              bne   $t5, $zero, nosub     ;Yes.  No subtract
8)              subu  $v0, $v0, $a0         ;Rem=Rem-divisor
9)              ori   $t6, $zero, 1         ;Bit to shift (temp)
10)             j     dorol
11) nosub:      ori   $t6, $zero, 0         ;Bit to shift (temp)
12) dorol:      ROL64 $v0,$v1,$t6           ;Do the ROL operation
13)             bne   $t4,$zero,divloop     ;Loop is count nonzero
14)             srl   $v0,$v0,1             ;Final shift of remainder
15)             bne   $v1,$zero,exit        ;Check quotient=0
16)             add   $v0,$a0,$zero         ;Ah.  Let rem=dividend
17) exit:       jr    $ra
```

Notes on solution (by line number):

- This inner loop is 12 cycles, which is the average good solution (accepted as answer)
- We did require that you be as minimal as possible on the inner loop.
- Note that we have moved loop check to end of loop (line 13), saving a cycle in loop.
- Note that signed ops (such as slt) at line 6 doesn't work for 32-bit *unsigned* values. Similiarly, subtracting and checking the "sign" of the result doesn't work either. The simplest way to understand this is to consider the case when the quotient has its high bit set. Then , even though remainder = 0, we will think we should subtract!
- It is important to have a logical shift in line 14, since you don't want to simply copy the high bit. Again, that is because this is an unsigned multiplication.
- Further note on closing lines 15 and 16 (which we didn't require):
  This is a special case required to get the correct answer on those cases for which the remainder has its high bit set. Since our last action (line 14) is to logically shift left, there is no way to get such a remainder. Thus, something is clearly broken up to line 14. Fortunately, we know that remainder < divisor. So, if the remainder has its high bit set, so does the divisor. However, since:

  $$\text{dividend} = \text{quotient} \times \text{divisor} + \text{remainder}$$

  we see that we can't get a 32-bit result on the left if *both* the quotient×divisor and remainder terms have their high-bits set. This means that:

  $$\text{remainder has high-bit set} \Rightarrow \text{quotient} = 0.$$

  Thus, the easy fix is to check for quotient = 0, and always set remainder=dividend (always works).

Here is a faster inner loop (I believe this is the fastest, but I could certainly be wrong). The trick is to use the result bits from the comparison in line 6 directly to form the quotient. Note that they are inverted from what we want. So, we just use them and invert all the bits of the quotient after we are done. This saves 32 x 2 – 2 = 62 cycles

```
1)   divide:     ori   $t4, $zero, 32        ;Initialize count
2)               ori   $v0, $zero, 0         ;Rem = 0
3)               add   $v1, $a0, $zero       ;Quotient = Dividend
4)               ROL64 $v0, $v1, $zero
5)   divloop:    addi  $t4, $t4, -1          ;Decrement count
6)               sltu  $t5, $v0, $a1         ;Check: Rem < Divisor ?
7)               bne   $t5, $zero, dorol     ;Yes.  No subtract
8)               subu  $v0, $v0, $a0         ;Rem=Rem-divisor
9)   dorol:      ROL64 $v0,$v1,$t5           ;Note: t5 is inverted!
10)              bne   $t4,$zero,divloop     ;Loop is count nonzero
11)              addi  $t6,$zero,-1          ;All 1s in t6
12)              xor   $v1,$v1,$t6           ;Invert bits in quotient
13)              srl   $v0,$v0,1             ;Final shift of remainder
14)              bne   $v1,$zero,exit        ;Check quotient=0
15)              add   $v0,$a0,$zero         ;Ah.  Let rem=divident
16)  exit:       jr    $ra
```

Problem 3d:
What is the "CPI" of your divide procedure (i.e. what is the total number of cycles to perform a divide)? Assume that each MIPS instruction takes 1 cycle.

Need to consider (1) initialization code, (2) 32 x loop code and, (3) closing code. Remember that ROL is 5 cycles (5 instructions). Some people who neglected to move their branch condition to the end of the loop forgot that there would be one last execution of the branch when count = 0. Assume worst-case delay (the maximum time). To do this, we find the largest time that we might take in the inner loop.

For our first solution:

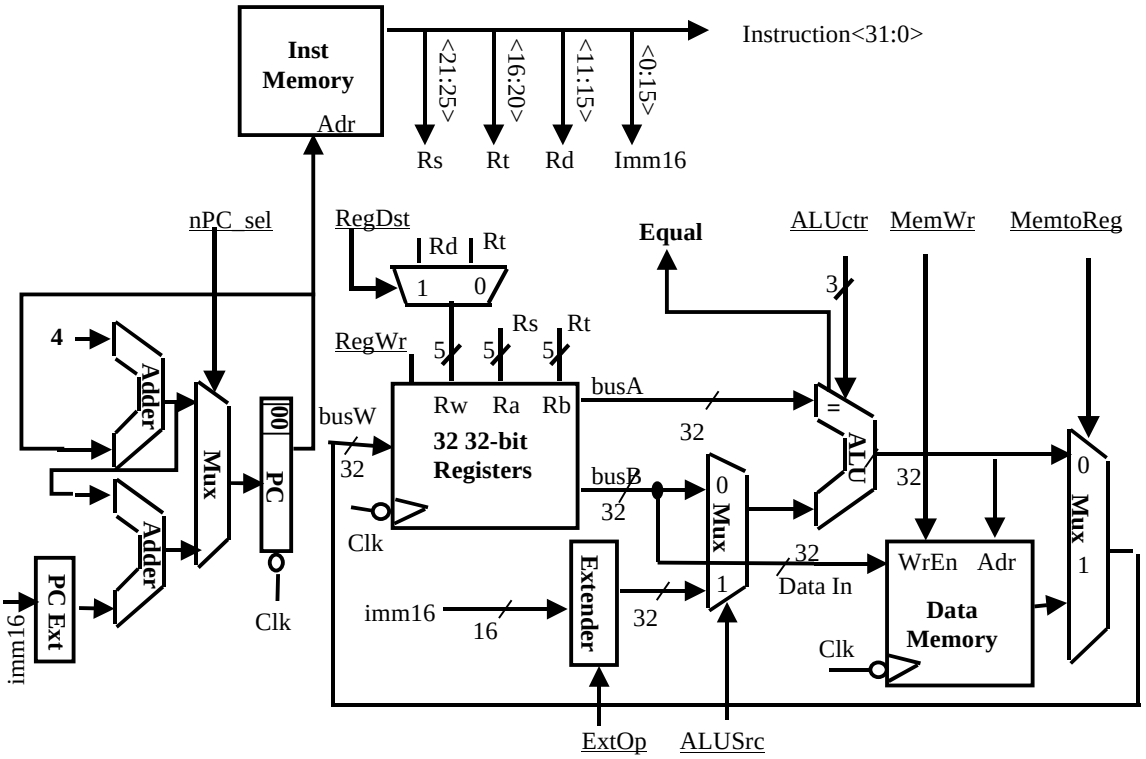
CPI = 8 + 32 x 12 + 4 = 396 cycles/divide

For the second:

CPI = 8 + 32 x 10 + 6 = 334 cycles/divide

Problem 4: New instructions for a single-cycle data path

The Single-Cycle datapath developed in class is shown below (similar to the one in the book):



It supports the following instructions. (Note that, as in the virtual machine, the branch is not delayed.)

op | rs | rt | rd | shamt | funct = MEM[PC]
op | rs | rt | Imm16 = MEM[PC]

INST	Register Transfers	
ADDU	R[rd] ← R[rs] + R[rt];	PC ← PC + 4
SUBU	R[rd] ← R[rs] - R[rt];	PC ← PC + 4
ORI	R[rt] ← R[rs] + zero_ext(Imm16);	PC ← PC + 4
LW	R[rt] ← MEM[R[rs] + sign_ext(Imm16)];	PC ← PC + 4
SW	MEM[R[rs] + sign_ext(Imm16)] ← R[rs];	PC ← PC + 4
BEQ	**if** (R[rs] == R[rt]) **then**	PC ← PC + sign_ext(Imm16)∪ 00
	else	PC ← PC + 4

Consider adding the following instructions: **ADDIU, XOR, JAL**, and **BGEZAL** (branch on greater than or equal to zero and link). This last instruction branches if the value of register **rs** ≥ 0; further, it saves the address of the next instruction in $ra (like **JAL).** Remember that the JAL format has a 6-bit opcode + 26 bits used as an offset...

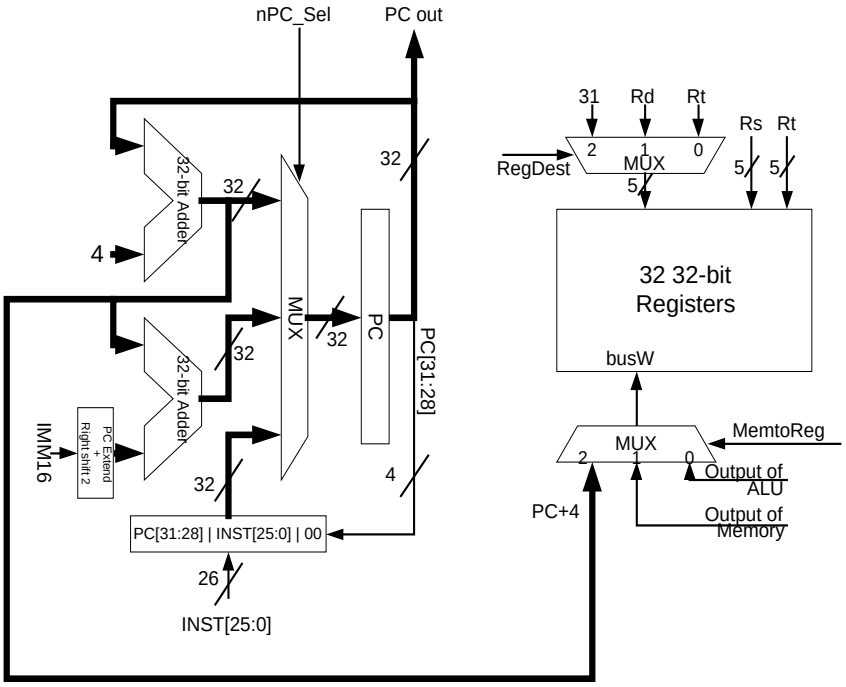
Problem 4a:
Describe/sketch the modifications needed to the datapath for each instruction. Try to add as little hardware as possible. Make sure that you are very clear about your changes. Also assume that the original datapath had only enough functionality to implement the original 5 instructions:

***ADDIU**: No change to data path*
***XOR**: Must enhance the ALU to include the XOR functionality.*

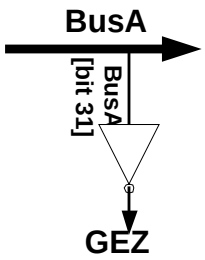
***JAL**: Must enhance data path to (1) permit proper update to PC and (2) write back PC+4 to register file (into the $ra register). There are many ways to dot this. We will show one. Note that the bottom 26 bits of the instruction must be combined with the top-4 bits of the current PC and 2 zero bits to form the new PC for a JAL:*

NewPC = OldPC[31:28] || Instruction[25:0] || 00.

Also, notice that we now have more possibilites for three of the control signals: nPC_SEL, RegDest, and MemToReg.



***BGEZAL**: Same modifications to register destination logic and MemToReg mux as above. Also, to check for the condition, it turns out that checking R[rs]≥0 is equivalent to checking the MSB of the RegA output bus from the register file:*



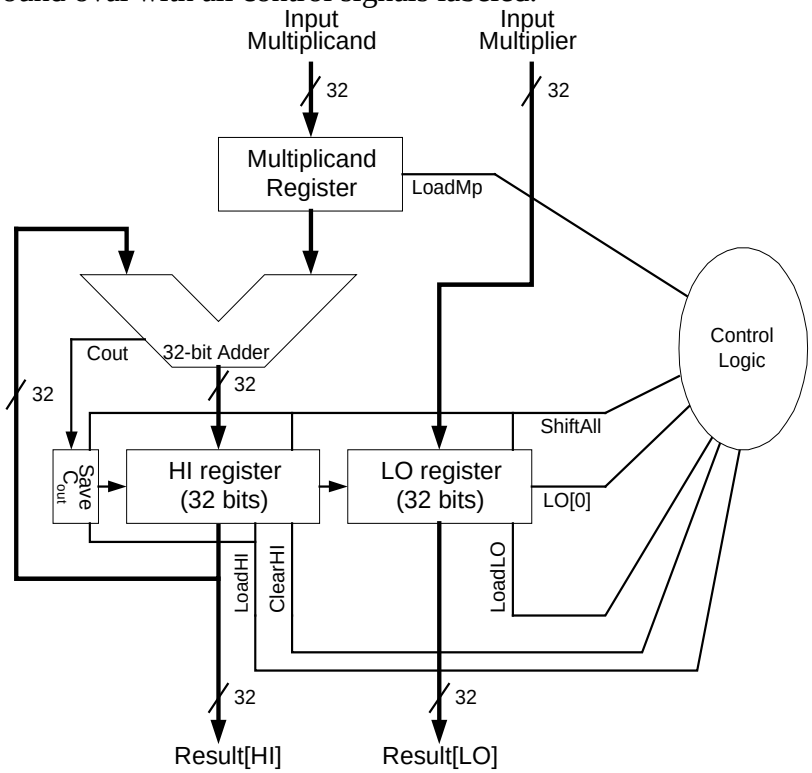
Problem 4b:
Specify control for each of the new instructions. Make sure to include values (possibly "x") for all of the control points.

*Note that we expected you to include all of the relevant control signals in your design. Further, note that the two lines for BGEZAL (taken/not taken) are **different**. So, if you included only one line there, you needed to make sure that it was clear how signals depended on GEZ.*

Instr	GEZ	nPC_ sel	Reg Wr	Reg-Dest	ALUctrl	ExtOp	ALU src	Mem Wr	MemTo Reg
ADDIU	X	0	1	0	Add	SignEx	1	0	0
XOR	X	0	1	1	XOR	X	0	0	0
JAL	X	2	1	2	X	X	X	0	2
BGEZAL	0	0	0	X	X	X	X	0	X
BGEZAL	1	1	1	2	X	X	X	0	2

Problem 5: Multiplication

Problem 5a:
Draw the datapath for a multi-cycle, 32-bit x 32-bit *unsigned* multiplier. Try to minimize hardware resources. Assume a single 32-bit adder. Further, to be consistent with MIPS, call the two 32-bit registers that contain the result "hi" and "lo". Draw the control for this multiplier as a round oval with all control signals labeled.



Note: People who simply copied the data path from the book without labeling all of the required control signals didn't get full credit for their datapath.

Problem 5b:
Describe the algorithm for performing a multiplication with this hardware. You can use pseudo-code. Make sure to include any initialization that might be required. Also, make sure that any loops are labeled with the number of iterations.

WE assume that the controller has an internal counter that it can use to count to 32. This is equivalent to assuming that it has a state machine with 32 states.

1. *Assert LoadLO/ClearHI to initialize; Set count = 32;*
2. *if (LO[0] == 1) assert LoadHI*
3. *Assert ShiftAll*
4. *decrement count and loop to 2 if count != 0*
5. *Result in HI/LO registers*

Note that this trick with the carry-out of the ALU is necessary to get full precision on the result. However, if you assumed only a zero input, this will work "most of the time" (and we didn't dock for this).

[Problem 5 continued]
Single-bit Booth encoding results from noticing that a sequence of ones can be represented by two non-zero values at the endpoints:

$$\mathbf{111111 = 1000000 - 1 = 10000\bar{1}}$$

The encoding uses three symbols, namely: $\mathbf{\bar{1}}$**, 0, and 1**. (The $\bar{1}$ stands for "-1"). A more complicated example of Booth encoding, used on a two's-compliment number is the following:

$$\mathbf{1111111100111011 = 1000000\bar{1}01001\bar{1}0\bar{1}}$$

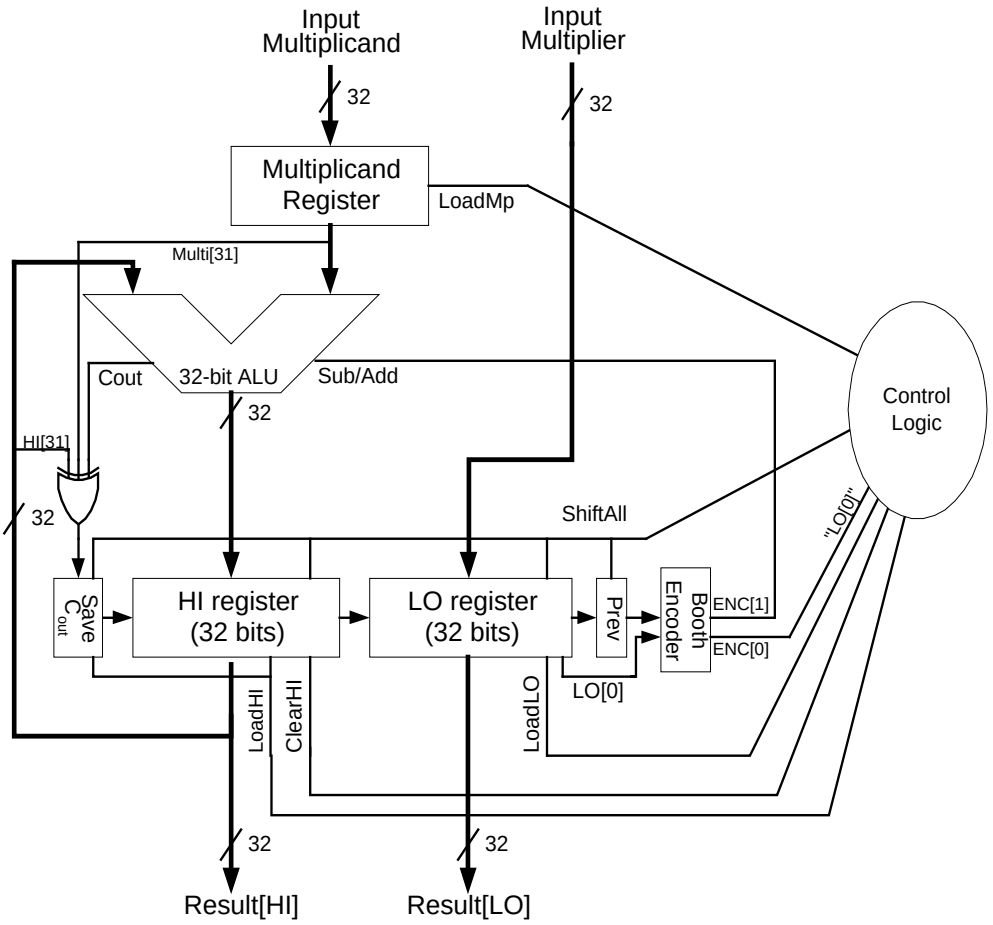
To perform Booth encoding, we build a circuit that is able to recognize the beginning, middle, and end of a string of ones. When encoding a string of bits, we start at the far right. For each new bit, we base our decision of how to encode it on both the current bit and the previous bit (to the right).

Problem 5c:
Write a table describing the this encoding. It should have 2 input bits (current and previous) and should output a 2 bit value which is the two's compliment value of the encoded digit (representing $\bar{1}, 0, \text{or } 1$):

Answer was given in class:

Cur	Prev	Out
0	0	00
0	1	01
1	0	11
1	1	00

Problem 5d:
Modify your datapath to do 32x32 bit *signed* multiplication by Booth-encoding the multiplier (the operand which is shifted during multiplication). Draw the Booth-encoder as a black-box in your design that takes two bits as input and produces a 2-bit, two's complement encoding on the output. Assume that you have a 32-bit ALU that can either add or subtract. *(Hint: Be careful about the sign-bit of the result during shifts. Also, be careful about the initial value of the "previous bit".)* Explain how how your algorithm is different from the previous multiplier.



The neat thing about this type of booth encoding, is that you don't have to change the algorithm (controller). Just use the "sign" bit of the booth encoded signal to control the ALU. Call the low bit of the booth output "LO[0]" (even though it isn't really), and POOF, the algorithm is identical.

*Note that you **DO** have to be careful about the sign bit. For fully correct operation, you have to assume that the adder will overflow and need to reconstruct the sign bit. To fix this, we conceptually sign extend the two 32-bit values to 33-bit values. Then, to build a 33-bit adder, we use our 32-bit adder in combination with an extra 3-input xor-gate (which does the "sum" portion of the top bit -- see problem 2). This way we know that we won't get an overflow, and will have the correct 33rd bit to shift in when we shift. Note that we accepted solutions which simply sign-extended by wrapping the high-bit of the HI register back during shifting. Neglecting sign-extention was not ok, however.*

Problem 5e: By encoding two bits at a time, we could potentially speed up multiplication a lot. To do this, you simply use your table from problem 5c on two successive bits at the same time. So, we are going to make a table that has *three* input bits (namely the 2 bits for encoding and one previous bit) and which has a single radix-2 symbol as an output. This output will be one of: $\bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3$.

To help in this, you will first list the two encoded bits that result from using the table in (5c) to encode bits In1/In0 and bits In0/Prev. Don't bother with two's complement, just use the symbols: $\bar{1}, 0,$ **and** 1 . Call these results Enc1 and Enc0. Then, treat these two encoded bits as representing a composite number, with the left digit being in the "2s" place (call this the "output-2" column). So, a sample line from your table will be:

Example:

In1	In0	Prev	Enc1	Enc0	Output
1	0	1	$\bar{1}$	1	$\bar{1}$ → (this is 2x(-1) + 1 = -1)

Write the complete 8-entry table (note again that you are leaving the output symbols as one of $\bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3$):

This answer should have been a simple matter of plugging in values from table 5d:

In1	In0	Prev	Enc1	Enc0	Output
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	$\bar{1}$	1
0	1	1	1	0	2
1	0	0	$\bar{1}$	0	$\bar{2}$
1	0	1	$\bar{1}$	1	$\bar{1}$
1	1	0	0	$\bar{1}$	$\bar{1}$
1	1	1	0	0	0

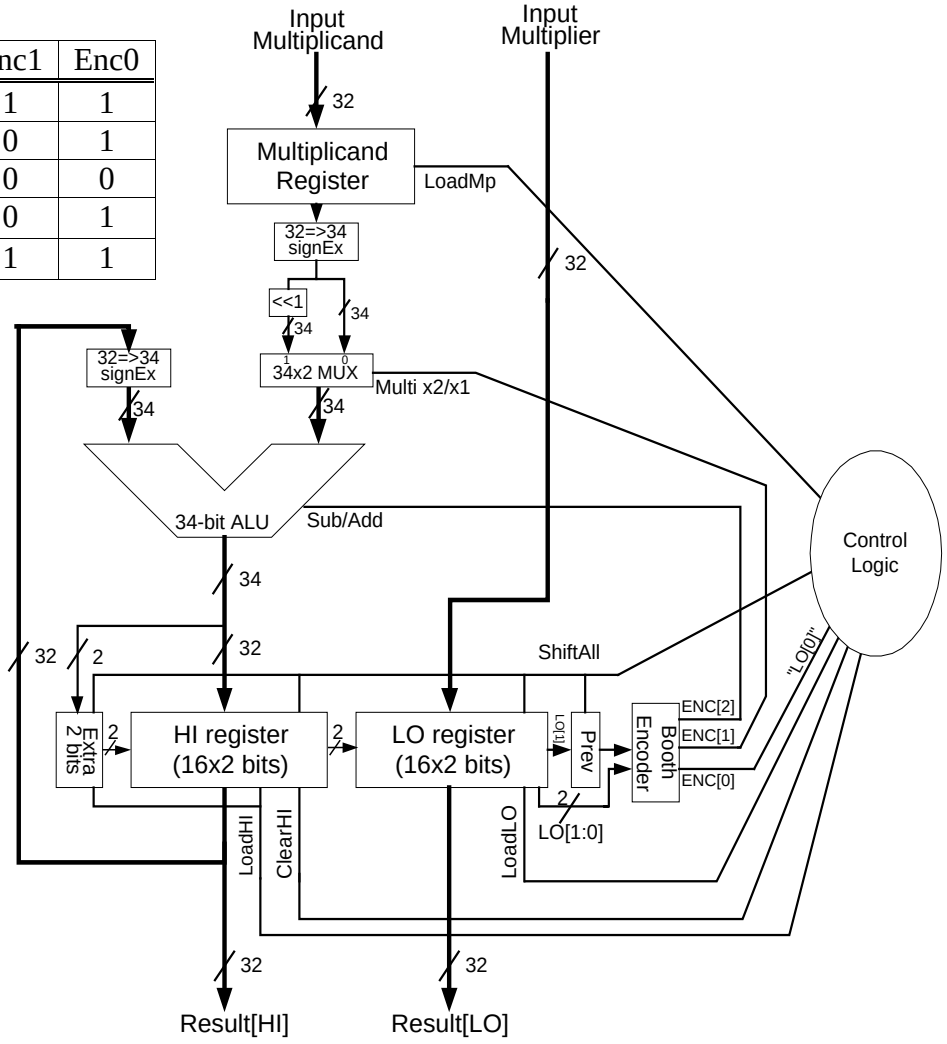
Problem 5f: Notice that 3 and $\bar{3}$ never show up in the output column. Explain why this is true. Also, explain why this is good for implementing radix-4 (two-bit at a time) multiplication.

Because 3 *and* $\bar{3}$ *represent two overlapping "end of string of ones" or "beginning of string of ones" respectively. It is not possible, for instance, for a string of ones to end on two consecutive bits. Alternatively, you could say that* 3 *derives from both Enc1=1 and Enc0=1, which would require In0 to be both zero and one.*

This is good for radix-4 multiplication, because it means that we only have to multiply the multiplicand by 0, 1, or 2. These are easy. Multiplying by 3 would be more challenging.

Problem 5g [Extra Credit]:
Draw a datapath that does signed multiplication, two bits at a time and include the multiplication algorithm. Draw the two-bit Booth encoder as a black box which implements output from the table in problem 5f . Make sure that you describe how the 5 possible output symbols (i.e. $\bar{2}, \bar{1}, 0, 1,$ and 2) are encoded *(hint: two's complement is not the best solution here).* As before, assume that you have a 32-bit ALU that can add andsubtract:

Code	Enc2	Enc1	Enc0
2	0	1	1
1	0	0	1
0	0	0	0
$\bar{1}$	1	0	1
$\bar{2}$	1	1	1



This solution is remarkably similiar to the other two solutions. Notice that we have chosen an encoding for the Booth symbols that is similiar to sign/magnitude. Thus, the sign bit is Enc2, which goes directly to select Add/Subtract. The next bit (Enc1) indicates whether or not we should multiply the multiplicand by 2. Finally, the last bit indicates "non-zero", and is once again equivalent to "LO[0]". If we had gone with complete sign-magnitude (also ok), we would "or" together Enc0/Enc1 to get our replacement for "LO[0]". Except for the fact that this solution shifts only 16 times, the control state machine is identical to that used in the previous two problems.

[continued on next page]

The key changes to the datapath are two-fold

1. *We are now multiplying the multiplicand by* $\bar{2}, \bar{1}, 0, 1,$ or 2. *To do this, we need to either shift by one or not. This is the mux just under the multiplicand register.*
2. *We have formatted our high and low registers so that we can shift pairs of bits at once. Notice that "ShiftAll" now causes two matched shift registers (one containing even bits, the other containing odd ones) to shift. This is like shifting by two in the previous solution.*
3. *Rather than dealing with a "hack" carry solution like the last one, we simply use a 34 bit adder (built with a 32-bit + full/adders if necessary). We sign-extend everything to 34 bits. We need 33 bits for normal operation, since a 32-bit value times 2 may actually be a 33 bit item. The 34th bit is used for exactly the same reason that we used the 33rd bit in the previous solution – to have the right thing to shift in.*

Notice that a number of people tried to handle multiplication by two by first shifting by one, adding, then shifting by one again. While this will work, it destroys the advantage in speed gained by radix-4 Booth encoding (i.e. taking half the number of cycles to multiply).

Finally, to make this really fast, you would want to combine the logic for "LoadHI" with the logic for "ShiftAll", so that you could do them at the same time. This would make the multiplier truly run in 16 cycles. We didn't expect you to come up with this.