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

Spring 2001 John Kubiatowicz

Midterm I
SOLUTIONS
March 1, 2001
CS152 Computer Architecture and Engineering

Your Name:	
SID Number:	
Discussion Section:	

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

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494 4

Problem 1: Performance

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

Instruction count, Cycles per instruction (CPI), and clock period (or frequency)

$$Runtime = InstCount \times CPI \times clockperiod = InstCount \times CPI \div clock\ frequency$$

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	35	1
Load and Store	25	2
Branches	25	3
Floating Point	15	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 13 cycles to execute.

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

CPI = (0.35×1) + (0.25×2) + (0.25×3) + (0.15×5) = 2.35
MIPS = 300MhZ÷ CPI = 300MhZ/2.35 = 127.66 MIPS

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

CPI = (0.35×1) + (0.25×2) + (0.25×3) + (0.15×13) = 3.55
MIPS = 300MhZ ÷ 3.55 = 84.51

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

$$\textbf{die / wafer} = \left\lfloor \frac{\pi\left(\frac{100}{2}\right)^2}{12^2} - \frac{\pi(100)}{\sqrt{2 \cdot 12^2}} \right\rfloor = 36 \qquad \textbf{dieYield} = (0.80)\left(1 + \frac{1 \cdot 1.2^2}{2}\right)^{-2} = 0.27$$

$$\textbf{dieCost} = \frac{\textbf{waferCost}}{(\textbf{die / wafer}) \cdot (\textbf{dieYield})} = \frac{2000}{36 \cdot 0.27} = 205.76$$

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

$$\textbf{die / wafer} = \left\lfloor \frac{\pi\left(\frac{100}{2}\right)^2}{10^2} - \frac{\pi(100)}{\sqrt{2 \cdot 10^2}} \right\rfloor = 56 \qquad \textbf{dieYield} = (0.80)\left(1 + \frac{1 \cdot 1.0^2}{2}\right)^{-2} = 0.36$$

$$\textbf{dieCost} = \frac{\textbf{waferCost}}{(\textbf{die / wafer}) \cdot (\textbf{dieYield})} = \frac{2000}{56 \cdot 0.36} = 99.21$$

Problem 1f:
Assume that we are considering the other direction of improving the original processor by increasing the speed of floating point. What is the best possible speedup that we could get, and what would the CPI and MIPS rating be of the new processor?

The easiest thing to do is use Amdahl's law: $\textbf{speedup} = \frac{1}{(1-f) + \frac{f}{n}} \rightarrow \frac{1}{(1-f)}$ *as* $n \rightarrow \infty$.

(i.e. speeding up floating-point really well). In this case, f is the fraction of time normally devoted to floating point (in time!). So, f=CPI_{float}*/CPI=(0.15×5)÷2.35 = 0.319*

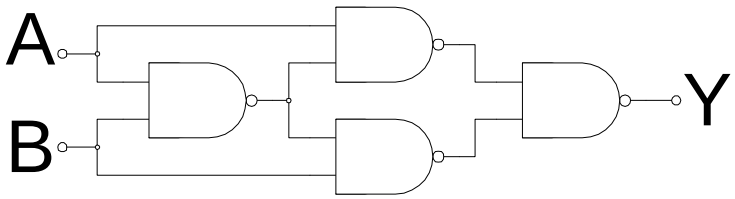
Max speedup = $(1-0.319)^{-1} = 1.47$
CPI computed with "zerocycle" floating-point instructions: 2.35-(0.15×5) = 1.6
MIPS = 300/1.6 = 187.5

Problem 2: Parallel Prefix

Assume the following characteristics for NAND gates:

Input load: 120fF,
Internal delay: TPlh=0.3ns, TPhl=0.6ns,
Load-Dependent delay: TPlhf=.0020ns, TPhlf=.0021ns

Problem 2a:
Suppose that we construct an XOR, as follows:



Compute the standard parameters for the linear delay models for this complex gate, assuming the parameters given above for the NAND gate. Assume that a wire doubles the input capacitance of the gate that it is attached to:

A Input Capacitance: ***240fF***
B Input Capacitance: ***240fF***

Load-dependent Delays:
TPAYlhf: ***0.0020 ns/fF***
TPAYhlf: ***0.0021 ns/fF***
TPBYlhf: ***0.0020 ns/fF***
TPBYhlf: ***0.0021 ns/fF***

Maximum Internal delays for A⇒Y:
TPAYlh:

Critical path goes through 3 gates. Low-to-high on output implies high-to-low on inputs to last gate, which implies low-to-high on input A. Note that the two internal nodes are driven, so we multiply capacitance by 2:

TPAYlh = 0.3ns+(2)(240fF)(0.0020ns/fF) + 0.6ns + (2)(120fF)(0.0021ns/fF) + 0.3ns = 2.664ns

TPAYhl:

High-to-low on output implies low-to-high on inputs to last gate, which implies high-to-low on input A.

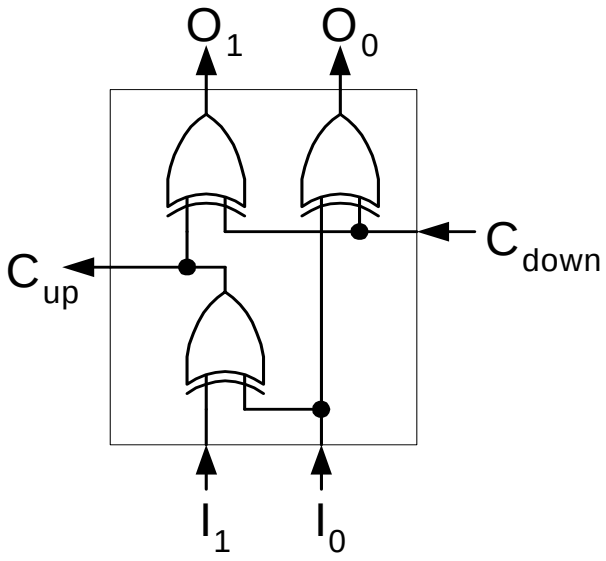
TPAYhl = 0.6ns + (2)(240fF)(0.0021ns/fF) + 0.3ns + (2)(120fF)(0.0020ns/fF) + 0.6ns = 2.988

An important operation that shows up in many different contexts is the parallel prefix circuit using XOR as the combining operation. This circuit takes as input a sequence of bits, such as: [I_0, I_1, I_2, I_3,] then outputs a new sequence, [O_0, O_1, O_2, O_3,…] which is the same length. The output bits are related to the input bits in the following fashion:

[$O_0=I_0$, $O_1=(I_0 \oplus I_1)$, $O_2=(I_0 \oplus I_1 \oplus I_2)$, $O_3=(I_0 \oplus I_1 \oplus I_2 \oplus I_3)$, …]

Each successive output bit is the XOR of the new input bit and the previous output bit.

The smallest parallel-prefix circuit has 2 inputs and two outputs. If this is intended to be part of a larger parallel prefix circuit, then we need "carry in" and "carry out" terminals such as shown to the right:



Problem 2b:
Using your answers from problem (2a), compute:

Input capacitance:

I_0: ***480 fF***
I_1: ***240 fF***
C_{down}: ***480 fF***

Load Dependent Delays for both outputs:
(as many parameters as appropriate):

Let X be with I_0 or I_1 and Y be O_0 or O_1:
TPXYlh: 0.0020 ns/fF
TPXYhl: 0.0021 ns/fF

Internal delays for the critical path (identify the critical path and compute delays):

.

Critical path is from inputs to O_1. The slow transition is high-to-low on internal node, so set C_{down} to make this always go in that direction

$TPI_0O_1hl = TPhl_{xor}+(TPhlf_{xor})(2)(C_{Axor}) + TPhl_{xor} = 2.988 + (0.0021)(2)(240) + 2.988=6.984ns$
$TPI_0O_1lh = TPhl_{xor}+(TPhlf_{xor})(2)(C_{Axor}) + TPlh_{xor} = 2.988 + (0.0021)(2)(240) + 2.664=6.660ns$

Problem 2c:

Now, put these 2-input blocks together to produce a 4-input block that takes I_0, I_1, I_2, and I_3, and C_{down} and produces: $O_0 = I_0 \oplus C_{down}$

$O_1 = I_1 \oplus I_0 \oplus C_{down}$

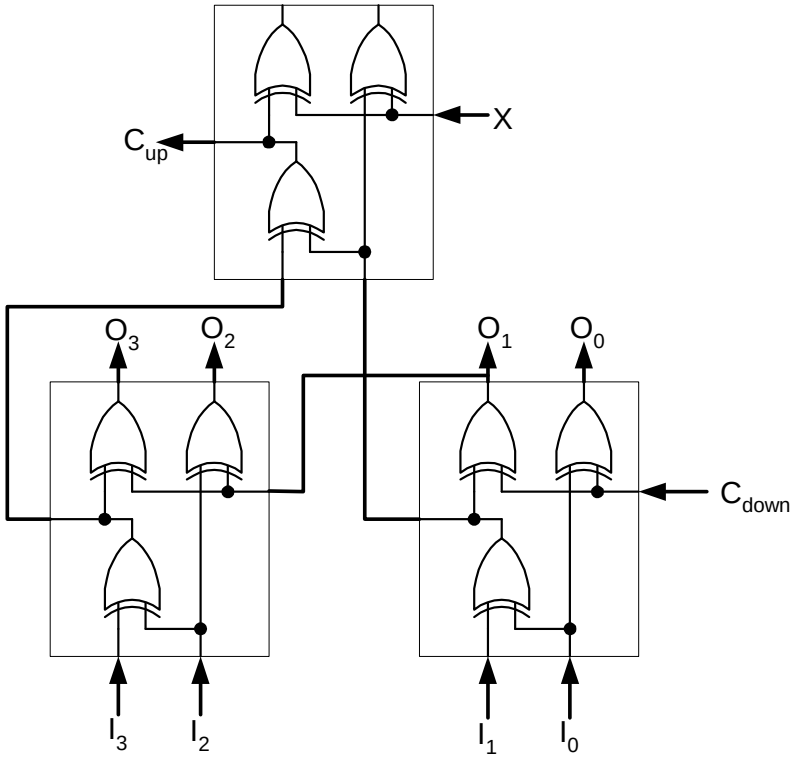
$O_2 = I_2 \oplus I_1 \oplus I_0 \oplus C_{down}$

$O_3 = I_3 \oplus I_2 \oplus I_1 \oplus I_0 \oplus C_{down}$

$C_{up} = I_3 \oplus I_2 \oplus I_1 \oplus I_0$

Your goal is to minimize the output delay of each block.

Using only blocks from part 2b:



Compute the input capacitance for each input:

I_0: 480, I_1: 240, I_2: 480, I_3: 240, C_{down}: 480

Identify the critical path of your circuit and compute the unloaded delay for this path.

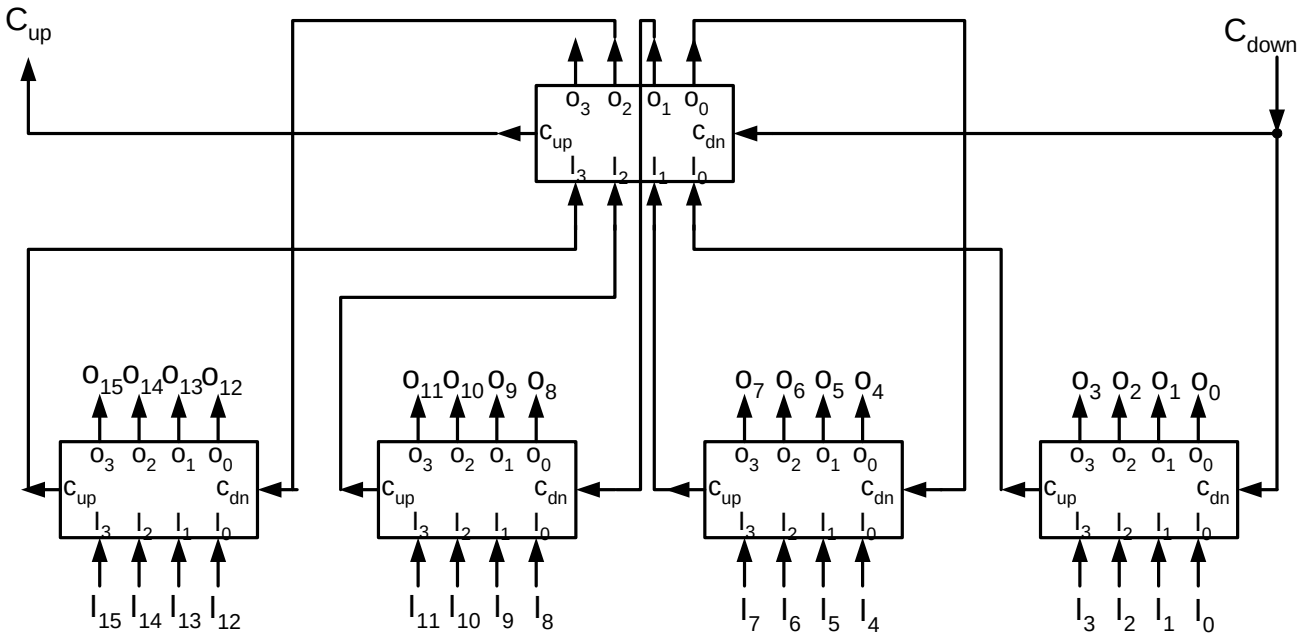
Critical path from I_0 to O_3. Arrange so that two internal nodes go from high-to-low:

$TPI_0O_3hl = 3 \times TPhl_{xor} + 2 \times [TPhlf_{xor} \times (2) \times (2) \times (240)] = 12.996$ ns

$TPI_0O_3lh = 2 \times TPhl_{xor} + 2 \times [TPhlf_{xor} \times (2) \times (2) \times (240)] + TPlh_{xor} = 12.672$ ns

Problem 2d:
Finally, show how the 4 input prefix circuit can be used as a building block to produce a 16-element prefix circuit that minimizes gate reuse and which has minimal delay. What is the critical path and how many XOR gates are in it?

Hint: this is very similar to a carry-lookahead adder.



The critical path is from I_0 up through the central logic and back through the C_{down} of the last stage to O_{14} or O_{15}.

Adding this up, we get: 2 + 3 + 2 = 7 XOR gates

Problem 2e:

How many XOR gates are in the critical path of a 64-bit parallel-prefix circuit?

This adds one more level of blocks. Tracing the first input to last output, we note that we have 2 for each level up, 3 for the top level, and 2 for each level down: 2 + 2 + 3 + 2 + 2 = 11 xor gates.

Problem 3: PI

This problem is not as bad as it looks. 3a and 3b can be done without understanding the math.

The book "A History of π" by Petr Beckmann is an amusing look at the history and politics behind the number PI. Among other things, this book shows several series that produce PI. One in particular is:

$$\frac{\pi}{4} = 4 \times \arctan\frac{1}{5} - \arctan\frac{1}{239}$$

In this problem, we will compute part of this series:

$$\arctan\frac{1}{x} = \frac{1}{x} - \frac{1}{3 \cdot x^3} + \frac{1}{5 \cdot x^5} - \frac{1}{7 \cdot x^7} + \ldots$$

Fortunately for us, each term of the series is smaller that the previous one by at least $\frac{1}{x^2}$. So, this means that each term of $\arctan\left(\frac{1}{5}\right)$ is smaller by at least $\left(\frac{1}{5}\right)^2 = 0.04$ and each term of $\arctan\left(\frac{1}{239}\right)$ is smaller by $\left(\frac{1}{239}\right)^2 < 1.8 \times 10^{-5}$. Thus, the series converges really quickly.

The secret to making this work is to note that each term in the series for PI is of the form 1/big number. Further, a lot of these numbers are related to each other. Consider:

$$A_0 = \frac{1}{x} \qquad \text{`} B_0 = \frac{1}{x} = \frac{A_0}{1}$$

$$A_1 = \frac{1}{x^3} = \frac{A_0}{x^2} \qquad B_1 = \frac{1}{3 \cdot x^3} = \frac{A_1}{3}$$

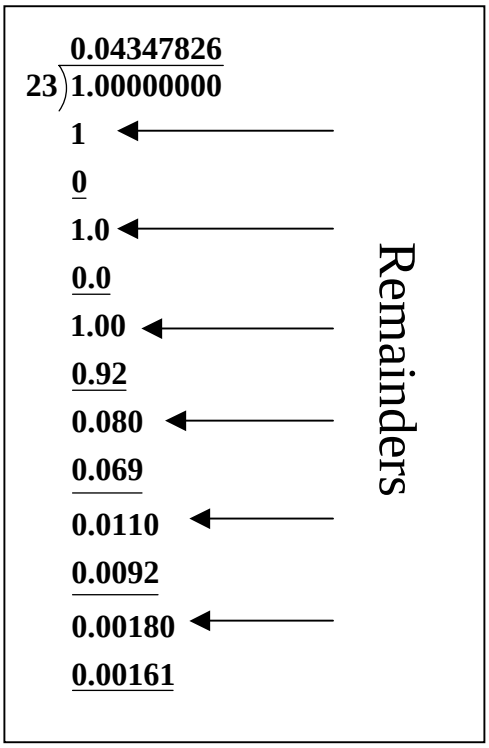
$$A_2 = \frac{1}{x^5} = \frac{A_1}{x^2} \qquad B_2 = \frac{1}{5 \cdot x^5} = \frac{A_2}{5}$$

So, $\arctan\frac{1}{x} = B_0 - B_1 + B_2 - \ldots$

Thus, all we need to do is figure out how to divide one number by another number for an arbitrary number of decimal places.

Suppose that we have a procedure that produces an infinite "stream" of digits for the series A_0. Then, we can input that stream as an input to the divide algorithm that produces A_1 (since it is A_0 divided by some integer like 25 or $(239)^2$. Further, we can send the stream of digits for A_1 to produce A_2 and B_1. Etc. That is our trick.

Recall how divide (in base 10) works The following shows a division of 1 by 23:



Suppose we had a procedure that produced each of the digits (zeros) in the dividend, one at a time. Consider the remainders as integers from the current decimal point. So, for instance, we have the remainders 1, 10, 100, 80, 110, 180, etc. At each stage, we multiply by ten, add the incoming digit (zero in the example), then

This could be combined with the current remainder but multiplying the remainder by 10, adding the new digit (which is zero in this case), then seeing how much the result divides the answer.

Here is complete pseudo code for computing one of the streams (*Note: we have fixed a couple of the typos)*:

```
Stream(digitnum,incoming,oddnum,sign,xsquared,termID,maxtermID) {
   ARemainder = A_REMARRAY[termID];
   ARemainder = ARemainder × 10 + incoming;

   ; This is a quotient/remainder operation
   (ADigit, ARemainder) = ARemainder / xsquared;
   A_REMARRAY[termID] = ARemainder;

   BRemainder = B_REMARRAY[termID];
   BRemainder = BRemainder × 10 + Adigit;
   (BDigit, BRemainder) = BRemainder / oddnum;
   B_REMARRAY[termID] = BRemainder;

   AddInDigit(BDigit, digitnum, sign);

   If ((termID = maxtermID ) && (ADigit != 0)) {
      A_REMARRAY[termID+1] = 0;
      B_REMARRAY[termID+1] = 0; /* This was missing originally */
      maxtermID++;
   }

   If (termID < maxtermID) {
      MaxtermID =
          Stream(digitnum, ADigit,(oddnum+2),-sign,
                 xsquared, (termID+1), maxtermID);
   }
   return maxtermID;  /* This was missing originally */
}
```

Problem 3a:
Write MIPS assembly for this pseudo code. Make sure to adhere to MIPS conventions. Assume that A_REMARRAY[] and B_REMARRAY[] are word arrays that are addressed via constants (assume that you can use the **la** pseudo instruction to load their addresses into registers. Also, assume that there are 7 **argument registers ($a0 - $a6) for the sake of this problem.** Note that **AddInDigit** is a procedure call.

```
Stream:     subiu$sp, $sp, 36           ; 7 args, 1 ret addr, 1 temp (ADigit)
            sw   $ra, 36($sp)           ; Save return address
            sw   $a0, 32($sp)           ; Save $a0
              <... etc ...>             ; Save $a1 - $a5
            sw   $a6, 8($sp)            ; Save $a6

            sll  $t0, $a5, 2            ; Convert termID to word index
            la   $t1, A_REMARRAY
            addu $t1, $t1, $t0          ; address of ARemainder
            lw   $t2, 0($t1)            ; Get ARemainder
            mul  $t2, $t2, 10           ; x 10 (pseudo instruction)
            addu $t2, $t2, $a1
            divu $t2, $a4
            mfhi $t2                    ; New remainder
            sw   $t2, 0($t1)            ; Save it into array
            mflo $t3
            sw   $t3, 4($sp)            ; Save ADigit for later

            la   $t1, B_REMARRAY        ;
            addu $t1, $t1, $t0          ; address of BRemainder
            lw   $t2, 0($t1)            ; Get BRemainder
            mul  $t2, $t2, 10           ; x10 (pseudo-instruction)
            addu $t2, $t2, $t3          ; Add in ADigit
            divu $t2, $a2
            mfhi $t2                    ; New BRemainder
            sw   $t2, 0($t1)            ; Save back into array

            move $a2, $a3               ; sign (third arg)
            move $a1, $a0               ; digitnum (second arg)
            mflo $a0                    ; Get BDigit
            jal  AddInDigit
            lw   $a0, 32($sp)           ; Restore digitnum (arg 1)
            lw   $a1, 4($sp)            ; Restore ADigit to $a1
            lw   $a2, 24($sp)           ; restore oddnum
            lw   $a3, 20($sp)           ; restore sign
            lw   $a4, 16($sp)           ; restore xsquared
            lw   $a5, 12($sp)           ; restore termID
            lw   $v0, 8($sp)            ; restore maxTermID (will return)

            bne  $a5, $v0, finalcheck   ; termId != maxTermID
            beq  $t3, $r0, finalcheck   ; ADigit == 0
            sll  $t1, $a5, 2
            la   $t1, A_REMARRAY
            addu $t1, $t1, $t0          ; address of A_REMARRAY[termID]
            sw   $r0, 4($t1)            ; store zero at A_REMARRAY[termID+1]
            la   $t1, B_REMARRAY
            addu $t1, $t1, $t0          ; address of B_REMARRAY[termID]
            sw   $r0, 4($t1)            ; store zero at B_REMARRAY[termID+1]
            addiu$v0, $v0, 1            ; maxterm++

finalcheck: blt  $a5, $v0, return       ; Check termID < maxtermID (pseudo-op)
            addiu$a2, $a2, 2            ; oddnum+2
            subu $a3, $r0, $a3          ; sign = -sign
            addiu$a5, $a5, 1            ; termID+1
            jal  stream

return:     lw   $ra, 36($sp)
            addiu$sp, $sp, 36           ; restore stack
            jr   $ra                    ; return
```

Problem 3b:
The procedure **AddInDigit** takes 3 arguments. A digit (a number from 0 to 9), a digit position (digitnum), and a sign. Assume that we have an infinite precision *decimal* number in memory, one digit per **byte**, starting at address **FINALVALUE.** Assume that "digitnum" specifies a byte offset from this address at which we need to add (sign =1) or subtract (sign=-1) the incoming digit. Write this procedure. *Assume that the result must be still in decimal. Thus, if you add the digit at* **FINALVALUE[digitnum]** *and it overflows (is bigger than 9), then you must carry to the next most significant digit (at* **digitnum-1***). Same is true of subtract (when sign = -1).*

Again assuming no delay slots:

```
AddInDigit: la   $t0, FINALVALUE        ; Get address of array
            addu $t0, $t0, $a1          ; Address of current digit

loopit:     lb   $t1, 0($t0)            ; get digit
            slt  $t2, $a2, $r0          ; Sign negative?
            bne  $t2, $r0, handleneg    ; Yup.  Go deal with it

            add  $t1, $t1, $a0          ; Add in new digit
            slti $t2, $t1, 10           ; Carry needed?
            bne  $t2, lastupdate        ; Nope.  Store digit and exit
            subi $t1, $t1, 10           ; subtract extra 10 from digit
            sw   $t1, 0($t0)            ; Store updated value
            subi $t0, $t0, 1            ; Go to next most significant digit
            addi $a0, $r0, 1            ; Next digit: 1
            j    loopit                 ; go handle carry

handleneg:  subu $t1, $t1, $a0          ; Subtract digit
            slti $t2, $t1, 0            ; result less than 0?
            beq  $t2, lastupdate        ; Nope.  Store digit and exit
            addi $t1, $t1, 10           ; Correct digit
            sw   $t1, 0($t0)            ; Store updated value
            subi $t0, $t0, 1            ; Go to next most significant digit
            addi $a0, $r0, 1            ; Next digit: 1
            j    loopit                 ; go handle borrow

lastupdate: sw   $t1, 0($t0)            ; write last digit
            jr   $ra                    ; return
```

‘

Problem 3c:
Explain the initialization of the A_REMVALUE[] and B_REMVALUE[] arrays if we were going to compute $\left(\mathbf{4}\cdot\mathbf{arctan}\frac{\mathbf{1}}{\mathbf{5}}\right)$. What is the purpose of the **termID** and **maxtermID** parameters?

We are just going to fold the 4 into our calculations. If we let the 4 be part of the A_0 computation, then every other term will be multiplied by 4 automatically (since A_1 depends on A_0, etc). Thus, we simply have an outer loop that produces the digits of $\frac{\mathbf{4}}{\mathbf{5}}$ one at a time and feed them to "stream". So, we will use A_REMVALUE[] and B_REMVALUE[] for all terms beyond the first one. Since each new remainder gets zeroed as it is needed, we merely have to set the first element of each array to zero. Thus, let A_REMVALUE[0] = 0 and B_REMVALUE[0]=0.

The variable **termID** *tracks which term of the series we are currently working on. Since the first term (the $\frac{\mathbf{1}}{\boldsymbol{x}}$ term) is a little special (It is not derived from other terms by dividing by x^2, we will let* **termID=0** *be the $\frac{\mathbf{1}}{\mathbf{3}\boldsymbol{x}^{\mathbf{3}}}$ term,* **termID=1** *be the $\frac{\mathbf{1}}{\mathbf{5}\boldsymbol{x}^{\mathbf{5}}}$ etc. The* **maxtermID** *is the maximum term that we have produced nonzero values for up to now. Note that in the stages of the design, almost all terms are zero, hence we start* **termID=maxtermID=0**

Problem 3d:
Explain the initialization of the **FINALVALUE** array:

Each digit of the FINALVALUE array must be initialized to zero before it is used. Since we are walking though the "answer" one digit at a time, we can choose to initialize this digit before we use it. (I.e. when we are working on the 10^{th}s place, we don't care what is in the 100^{th}s or 1000^{th}s place, since we know to ignore it.

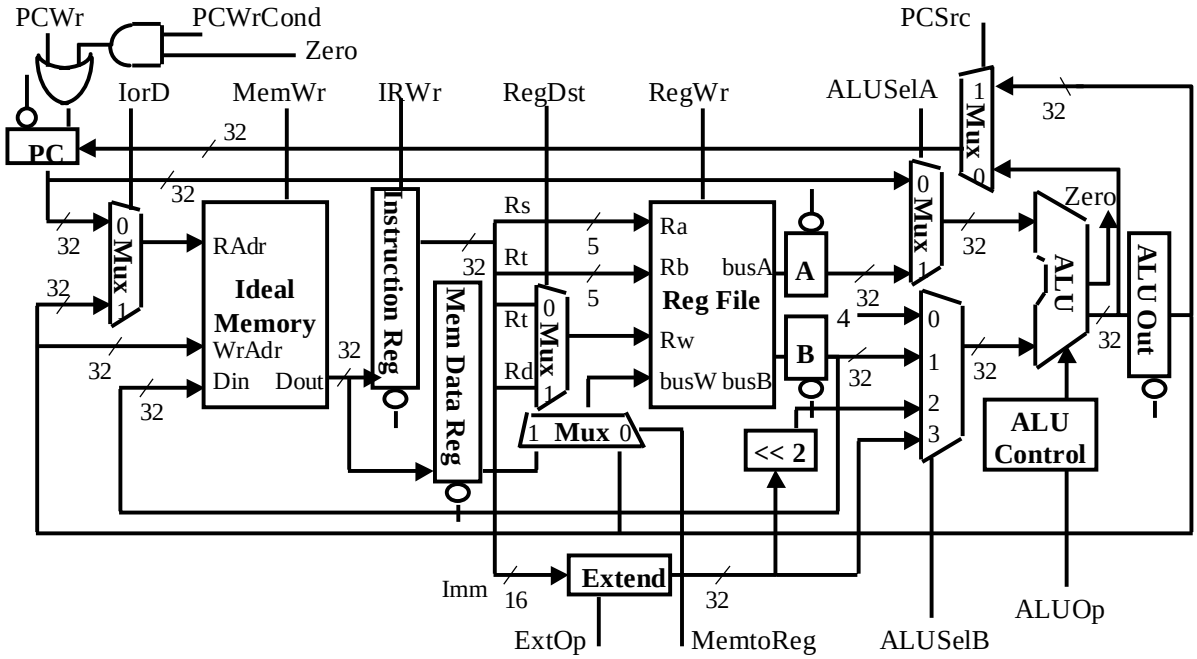
Problem 3e:
Write pseudo-code to compute $\left(\mathbf{4}\cdot\mathbf{arctan}\frac{\mathbf{1}}{\mathbf{5}}\right)$ using **stream**(). Assume that the initialization in (3c) and (3d) are accomplished..

```
FINALVALUE[0]=0     ; Set ones place to zero
FINALVALUE[1]=8     ; This is 4/5
A_REMVALUE[0]=B_REMVALUE[0] = 0    ; Start with 1 term

; Handle first digit (10th's place)
maxtermID = stream(1,8,3,-1,25,0,0)
for (digitnum=2; true; digitnum=digitnum+2) {
     FINALVALUE[digitnum] = 0;
     maxtermID=stream(digitnum,0,3,-1,25,0,maxtermID);
}
```

[This page intentionally left blank]

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



The Multi-Cycle datapath developed in class and the book is shown above. In class, we developed an assembly language for microcode. It is included here for reference:

Field Name	Values For Field	Function of Field
ALU	`Add`	ALU Adds
	`Sub`	ALU subtracts
	`Func`	ALU does function code (Inst[5:0])
	`Or`	ALU does logical OR
SRC1	`PC`	PC ⇒ 1st ALU input
	`rs`	R[rs] ⇒1st ALU input
SRC2	`4`	4 ⇒ 2nd ALU input
	`rt`	R[rt] ⇒ 2nd ALU input
	`Extend`	sign ext imm16 (Inst[15:0]) ⇒ 2nd ALU input
	`Extend0`	zero ext imm16 (Inst[15:0]) ⇒ 2nd ALU input
	`ExtShft`	2nd ALU input = sign extended imm16 << 2
ALU Dest	`rd-ALU`	ALUout ⇒ R[rd]
	`rt-ALU`	ALUout ⇒ R[rt]
	`rt-Mem`	Mem input ⇒ R[rt]
Memory	`Read-PC`	Read Memory using the PC for the address
	`Read-ALU`	Read Memory using the ALUout register for the address
	`Write-ALU`	Write Memory using the ALUout register for the address
MemReg	`IR`	Mem input ⇒ IR
PC Write	`ALU`	ALU value ⇒ PCibm
	`ALUoutCond`	If ALU Zero is true, then ALUout ⇒ PC
Sequence	`Seq`	Go to next sequential microinstruction
	`Fetch`	Go to the first microinstruction
	`Dispatch`	Dispatch using ROM

In class, we made our multicycle machine support the following six MIPS instructions:

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 + 4 + sign_ext(Imm16) || 00
	else	PC ← PC + 4

For your reference, here is the microcode for two of the 6 MIPS instructions:

Label	ALU	SRC1	SRC2	ALUDest	Memory	MemReg	PCWrite	Sequence
Fetch	Add	PC	4		ReadPC	IR	ALU	Seq
Dispatch	Add	PC	ExtShft					Dispatch
RType	Func	rs	rt					Seq
				rd-ALU				Fetch
BEQ	Sub	rs	rt				ALUoutCond	Fetch

In this problem, we are going to add four new instructions to this data path:

`jal` `<const>` ⇒ PC ← zero_ext(Instr[25:0]) || 00
R[31] ← PC + 4

`add` `$rd, $rs, $rt` ⇒ **if** (R[rs]+ R[rt] doesn't overflow) **then**
R[rd] ← R[rs] + R[rt]
PC←PC+4
Else
EPC←PC
Cause←12
PC←0x80000080

`mfc0` `$rd, $rt` **if** ($rt == 13) **then**
R[rd] ←Cause
Else if ($rt == 14) **then**
R[rd] ←EPC
PC←PC+4

`compmul` `$rd, $rs, $rt` ⇒ R[rd]=(R[rs]×R[rt]) – (R[rs+1]×R[rt+1])
R[rd+1]= (R[rs]×R[rt])+(R[rs+1]×R[rt+1])
PC←PC+4

This math was a typo. The real way to compute complex multiply is:

`compmul` `$rd, $rs, $rt` ⇒ R[rd]=(R[rs]×R[rt]) – (R[rs+1]×R[rt+1])
R[rd+1]= (R[rs]×R[rt+1])+(R[rs+1]×R[rt])
PC←PC+4

We will give the solution with the original spec (for fairness)

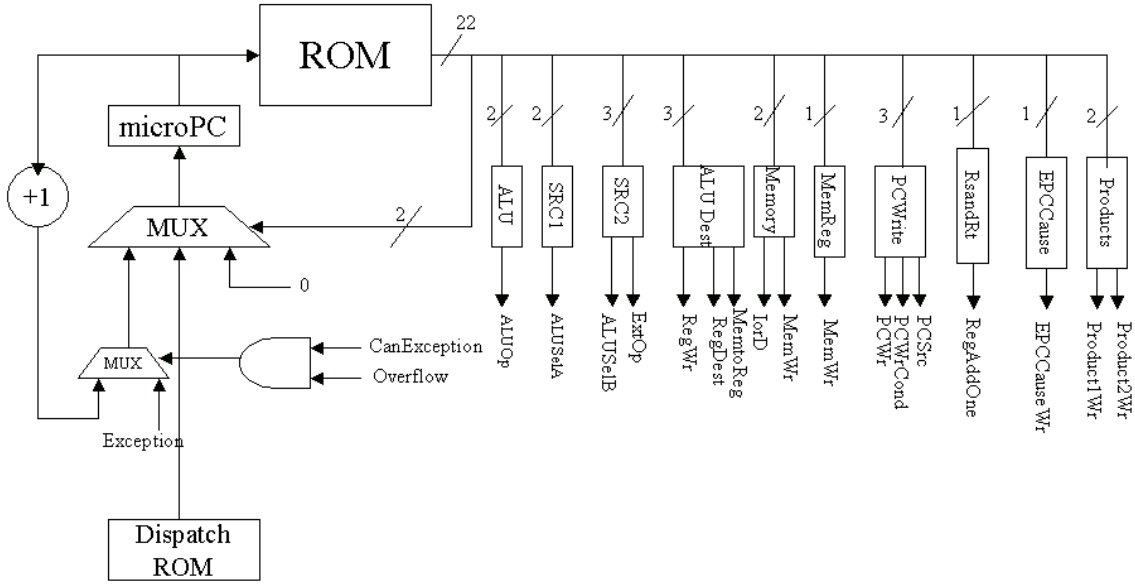
1. The `jal` instruction is familiar to you from the normal MIPS instruction set.
2. The `add` instruction is a normal add except that it causes an overflow exception if there is overflow. You need to implement the EPC (error PC) and Cause registers. Just assume that EPC gets the PC of the bad instruction and Cause gets the number 12.
3. The `mfc0` instruction is used to get the EPC and Cause values into normal registers
4. The `compmul` instruction does a complex multiply. It is assumed that the registers rd, rs, and rt are **even** registers and that the two source complex values are in R[rs], R[rs+1] (real, imaginary) and R[rt], R[rt+1] (real, imaginary), and that the results are put into R[rd] and R[rd+1] (real,imaginary).

Problem 4a: (2 pts)
How *wide* are microinstructions in the original datapath (answer in bits and show some work!)?

2 + 1 + 3 + 2 + 2 + 1 + 2 + 2 = 15 bits wide

The trickiest part of this computation is the PC Write field. We have to remember to represent the "do nothing" option, which means that there are actually three different values for the PC Write field.

Problem 4b: (4 points)
Draw a block diagram of a microcontroller that will support the new instructions (it will be slightly different than that required for the original instructions). Include sequencing hardware, the dispatch ROM, the microcode ROM, and decode blocks to turn the fields of the microcode into control signals. Make sure to show all of the control signals coming from somewhere. (*hint: The PCWr, PCWrCond, and PCSrc signals must come out of a block connected to thePCWrite field of the microinstruction).*



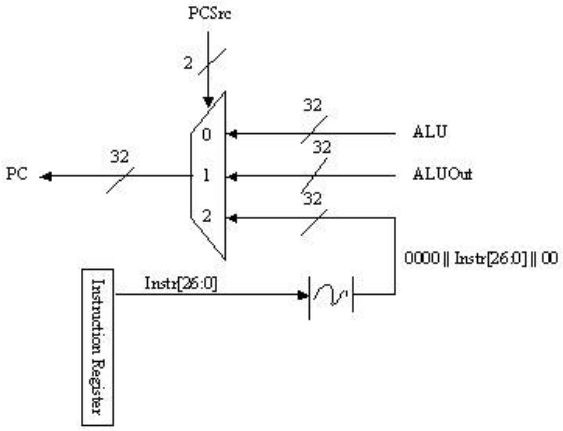
2 points were given for drawing a decent microcontroller for the old datapath. 1 point was given if the branching (exception) mechanism was implemented with a mux. Another point was given for showing some new control signals (EPCWrite is the most notable).

Problem 4c: (15 points)
Describe/sketch the modifications needed to the datapath for the new instructions (`jal`, `add`, `mfc0`, and `compmul`). Asume that the original datapath had only enough functionality to implement the original 6 instructions. Try to add as little additional hardware as possible. Make sure that you are very clear about your changes.

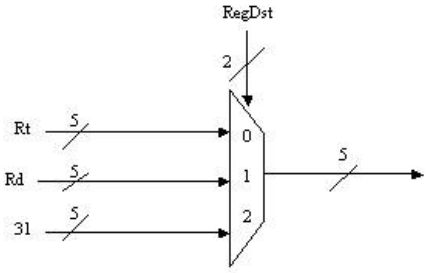
`jal`: 3 points

1) *Expand PCSrc mux to take in jump address.*

Alternatively, you could have modified the extender to take in 26 bits and have additional functionality. This solution requires more hardware though, and you would also need to either create a way for SRC1 to be zero or draw a wire from the output of the <<2 shifter to the PCSrc mux.

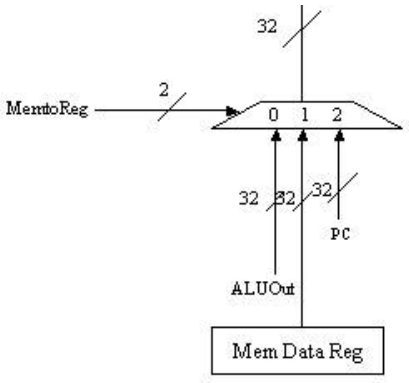


2) *Expand RegDst mux to take in constant 31.*



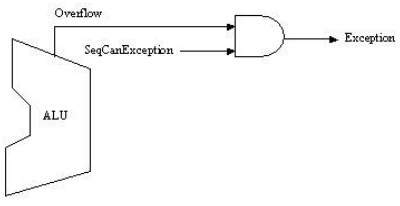
3) *Expand MemtoReg mux to take in PC.*

This was the most commonly omitted part. Part of the reason it looks okay to omit at first is that SRC1 can be the PC. However, if we used this, we would need a way to force SRC2 to be zero. Furthermore, `jal` would require 4 instructions instead of just 3.



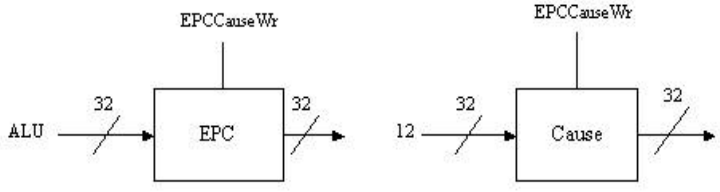
`add`: 4 points

1) *Give ALU an overflow output:.*

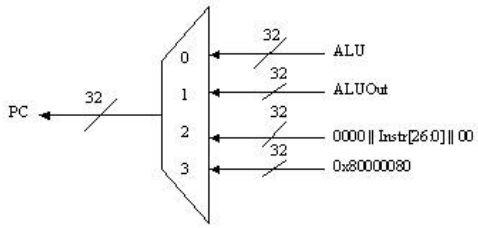


2) *Add EPC and Cause registers.*

The given spec doesn't let Cause take on values other than 12, so it was okay to just omit the cause register and use a hardcoded 12.

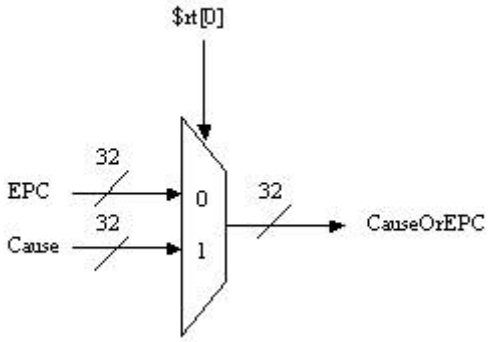


3) *Expand PCSrc mux to take in 0x80000080.*



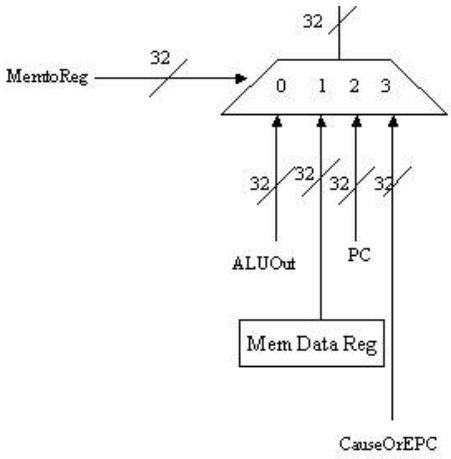
`mfc0`: *4 points*

1) *13 and 14 only differ by 1 bit, so just use a mux with the LSB of $rt as the selector to choose between Cause and EPC. Any other values of $rt are dontcares.*



2) *Expand MemtoReg mux to take in the CauseOrEPC.*

Alternatively, some students expanded SRC1 to be able to have the value of CauseOrEPC, but this has the disadvantage that you need to create a way for SRC2 to be forced to zero, and mfc0 would then require 4 instead of 3 microinstructions.



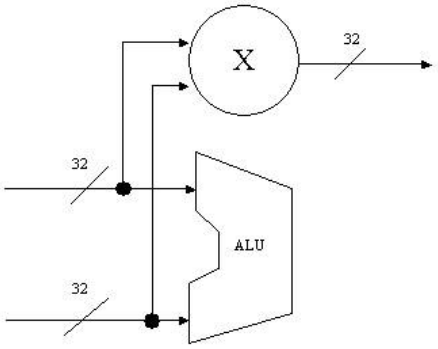
`compmul`: *4 points*

Correction: The math in the original test was wrong. The spec given on the exam was:

`compmul` `$rd, $rs, $rt` => R[rd] ← (R[rs]*R[rt]) – (R[rs+1]*R[rt+1])
R[rd+1] ← (R[rs]*R[rt]) + (R[rs+1]*R[rt+1])
PC ← PC + 4

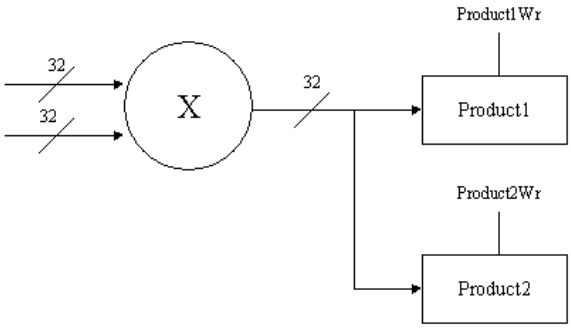
But anyways, this error makes the problem a bit simpler, because with the buggy problem we need to calculate only two products instead of four, so this solution will go with the original instructions.

1) *Add 32-bit multiplication capability. Either add the multiply operation to the ALU or put down a multiplier that takes in the same inputs as the ALU.*

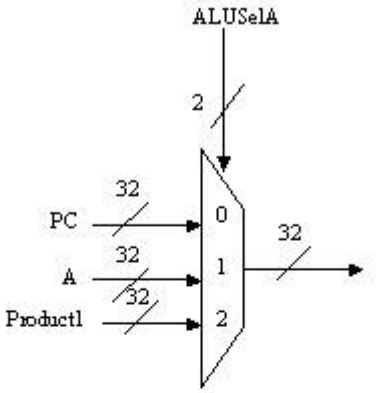


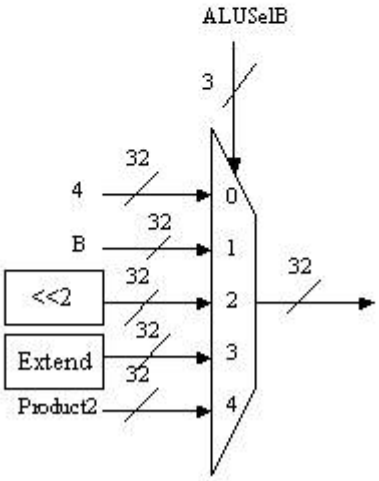
2) *Add registers to store products.*

You need at least two. Well, actually if a multiply-accumulate unit is used instead of a multiplier, you could go with just one, but that would make things complicated.

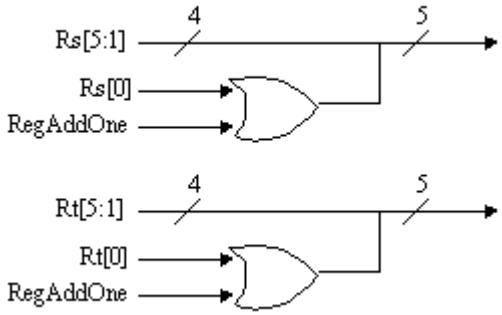


3) *Expand ALUSelA and ALUSelB muxes to take in these products.*



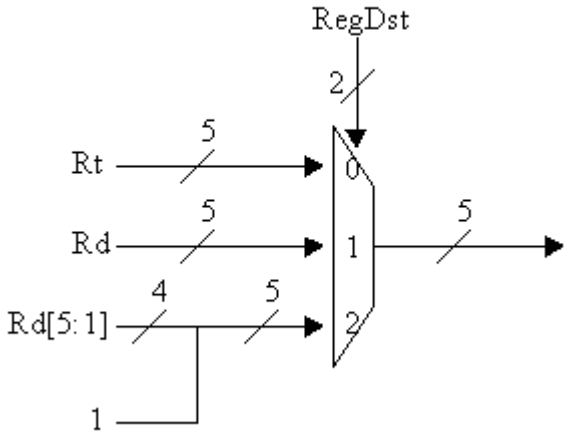


4) *Add capability to read rs+1 and rt+1.*



Some students did this with 5-bit adders and muxes. That's fine, but you don't need that much hardware because the registers are guaranteed to be even.

5) *Add capability to read rd+1.*



Problem 4d:
Describe changes to the microinstruction assembly language for these new instructions. How wide are your microinstructions now?

ALU:	*no changes*	*4 values → 4 values (0 new bits)*
SRC1:	*1 new value: Product1*	*2 values → 3 values (1 new bit)*
SRC2:	*1 new value: Product2*	*5 values → 6 values (0 new bits)*
ALU Dest:	*3 new values: 31-PC*	
	rd-CauseorEPC, rd+1-ALU	*4 values → 7 values (1 new bit)*
Memory:	*no changes*	*3 values → 3 values (0 new bits)*
MemReg:	*no changes*	*2 values → 2 values (0 new bits)*
PCWrite:	*2 new values: JumpAddr, Kernel*	*3 values → 5 values (1 new bit)*
Sequence:	*1 new value: SeqCanException*	*3 values → 4 values (0 new bits)*
**RsandRt:*	*2 new values: RegEven, RegOdd*	*0 values → 2 values (1 new bit)*
**EPCCause:*	*2 new values: EPCCauseWr, (do nothing)*	*0 values → 2 values (1 new bit)*
**Products:*	*3 new values: Product1, Product2,*	
	(do nothing)	*0 values → 3 values (2 new bits)*

15 + 0 + 1 + 0 + 1 + 0 + 0 + 1 + 0 + 1 + 1 + 2 = 22 bits wide

Answers may vary a lot, e.g. you may have:

Added a multiply value to the ALU field.
Altered the extender so that SRC2 would require another value, say Extend26.
Added a zero value to SRC1.
Added a zero value to SRC2.
Put SeqCanException in a field by itself.
Made separate fields for the Cause and EPC registers.

Or done even some other things differently that would still be correct if they matched your answer in 4b, 4d, and 4f.

4 points were given for having most of the proper microcode changes.

1 point was given for summing to some number for a new microinstruction width, and knowing that 1 new value does not equate to 1 new bit.

Problem 4e: *4 points*
Write complete microcode for the new instructions. Include the Fetch and Dispatch microinstructions. If any of the microcode for the original instructions must change, explain how (*Hint: since the original instructions did not use R[rd] as a register input, you must make sure that your changes do not mess up the original instructions).*

Label	ALU	SRC1	SRC2	ALU Dest	Memory	MemReg	PCWrite	Sequence	RsandRt	CauseEPC	Products
Fetch	Add	PC	4		ReadPC	IR	ALU	Seq			
Dispatch	Add	PC	ExtShift					Dispatch	RegEven		
Jal				31-PC			JumpAddr	Fetch			
Add	Add	rs	rt					SeqCanException			
				rd-ALU				Fetch			
Exception	Sub	PC	4				Kernel	Fetch		CauseEPCWr	
Mfc0				rd-CauseorEPC				Fetch			
Compmul		rs	rt					Seq	RegOdd		Product1
		rs	rt					Seq			Product2
	Sub	Product1	Product2					Seq			
	Add	Product1	Product2	rd-ALU				Seq			
				rd+1-ALU				Fetch			

1 point was given for each mostly correct instruction.

Many students neglected to copy down the Fetch and Dispatch microinstructions.

Some students did `jal` *in 4 microinstructions instead of 3. This should be okay if it matches your answer in part 4d.*

There should be a separate microinstruction for exception, rather than an add microinstruction that can do two different things.

The EPC is generated by subtracting the new PC – 4.

Conversely, `mfc0` *should not be done by two different microinstruction paths, because then you would need to lay down more branching hardware in the microcontroller.*

For `compmul`*, many students didn't specify where to store the products.*