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

Spring 2001 John Kubiatowicz

Midterm I

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]

3.141592653589793238462643383279502884197169399375105820974944

Problem 1: Performance

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

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 $1 / (\text{cm}^2)$. The manufacturing process has an 80% wafer yield and a value of 2 for α. Here are some equations that you may find useful:

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

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

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?

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

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

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

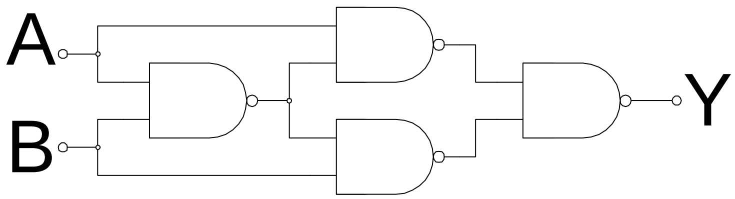
Problem 1f:
Assume that we are considering the other direction of improving the original processor by increasing the speed of floating point. What is the best possible speedup that we could get, and what would the CPI and MIPS rating be of the new processor?

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:
B Input Capacitance:

Load-dependent Delays:
TPAYlhf:
TPAYhlf:
TPBYlhf:
TPBYhlf:

Maximum Internal delays for A⇒Y:
TPAYlh:

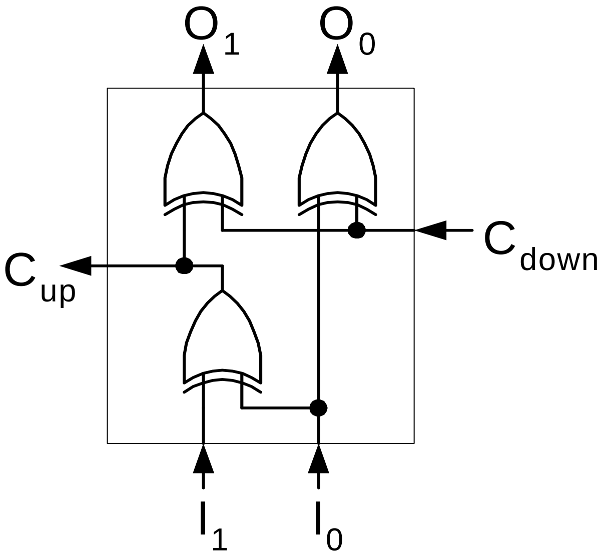
TPAYhl:

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:
I_1:
C_{down}:

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

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

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.

Compute the input capacitance for each input:

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

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.

Problem 2e:

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

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:

$$A_0 = \frac{1}{x} \qquad\qquad ‘\, B_0 = \frac{1}{x} = \frac{A_0}{1}$$

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

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

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

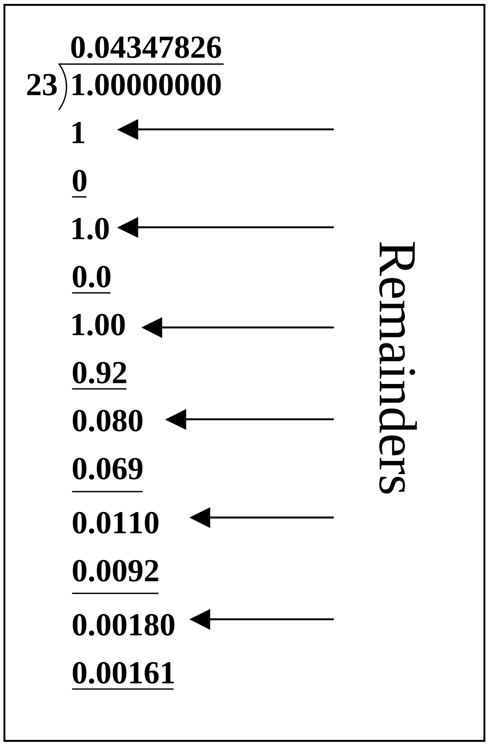
Thus, all we need to do is figure out how to divide one number by another number for an arbitrary number of decimal places.

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:

```
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;
      maxtermID++;
   }

   If (termID < maxtermID) {
      Stream(digitnum, ADigit,(oddnum+2),-sign, xsquared, (termID+1),
              maxtermID);
   }
}
```

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.

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).*

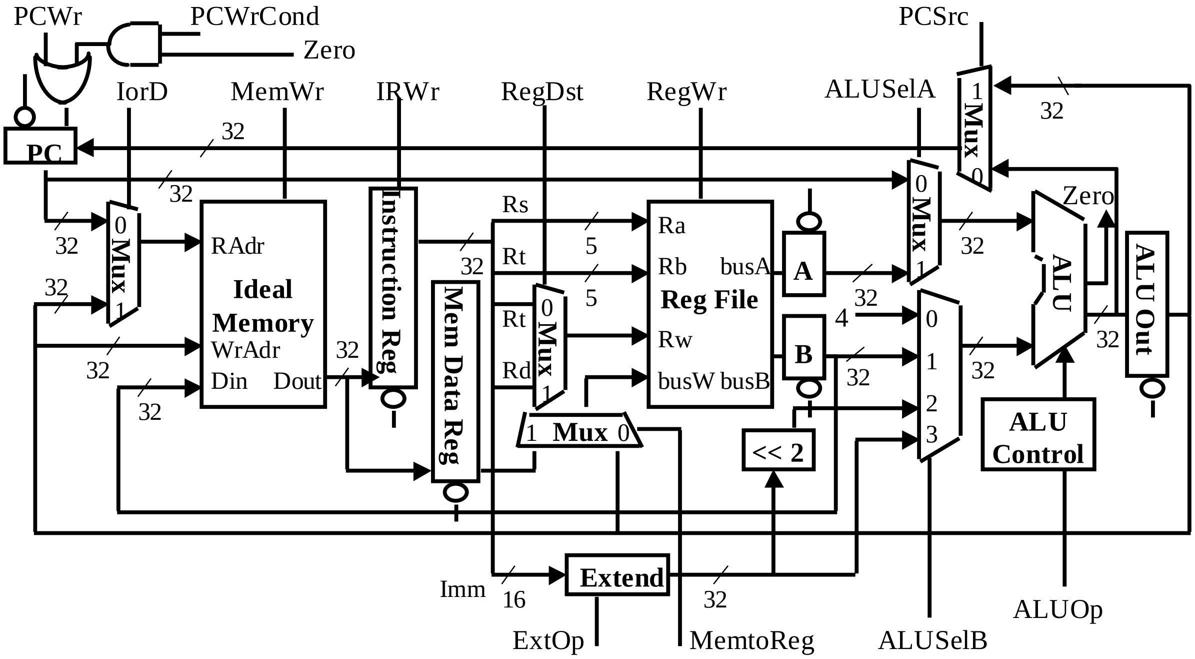
Problem 3c:
Explain the initialization of the A_REMVALUE[] and B_REMVALUE[] arrays if we were going to compute $\left(\mathbf{4 \cdot arctan \frac{1}{5}}\right)$. What is the purpose of the **termID** and **maxtermID** parameters?

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

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

[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
```

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

Problem 4b:
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).*

Problem 4c:

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.

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

Problem 4e:
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).*