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

Spring 2003 John Kubiatowicz

Midterm I

March 12, 2003
CS152 Computer Architecture and Engineering

Your Name:	
SID Number:	
Discussion Section:	

Problem	**Possible**	**Score**
1	25	
2	20	
3	25	
4	30	
Total		

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494 4

Problem 1: Short Answer

Problem 1a [3 pts]:
What is Amdhal's law? Give a formula and define the terms. How is this useful?

Problem 1b [3 pts]:
What are **setup** and **hold-time** and how can they be violated in a synchronous circuit? Can you still use a chip that is experiencing hold-time violations? How about **setup** violations?

Problem 1c [2 pts]:
Is the multi-cycle data path always faster than the single-cycle data path? Explain.

Problem 1d [3 pts]:
What are precise interrupts and why is it easy to provide them for our multi-cycle data path?

Problem 1e [3 pts]:
Suppose that you have analyzed a benchmark that runs on your company's processor. This processor runs at 500MHz and has the following characteristics:

Instruction Type	Frequency (%)	Cycles
Arithmetic and logical	40	1
Load and Store	30	2
Branches	20	1
Floating Point	10	12

What is the CPI and MIPS rating of this processor running this benchmark?

Problem 1f [2 pts]:
What is the technique used for a carry-select adder and how can it be used in general to speed up hardware (*hint: this is a very general technique to make a time⇔space tradeoff*)

Problem 1g [3pts]: What does it mean for a branch to have a delay slot, and why does it complicate the servicing of interrupts?

Problem 1h[3pts]: The Clark paper on testing talked about using randomness in at least two different ways during testing of the VAX. What were they?

Problem 1i [3 pts]:
The 1-bit Booth algorithm recodes one of the operands of a multiplier from binary into trinary logic with symbols: $\overline{1}$, 0, and 1. The transformation occurs one bit at a time, as given in class:

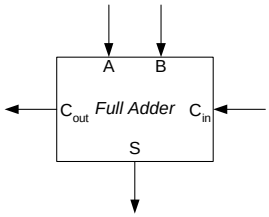
Cur	Prev	Out
0	0	0
0	1	1
1	0	$\overline{1}$
1	1	0

Suppose we encode 3 bits at a time. Finish filling out the following transformation table:

Cur	Prev	Out
000	0	0
000	1	1
001	0	1
001	1	2
010	0	
010	1	
011	0	
011	1	
100	0	
100	1	
101	0	
101	1	
110	0	
110	1	
111	0	
111	1	

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 [5pts]:
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: 100fF,
Propagation delay: TPlh=0.4ns, TPhl=0.4ns,
Load-Dependent delay: TPlhf=.0020ns, TPhlf=.0021ns

OR: Input load: 100fF
Propagation delay: TPlh=0.2ns, TPhl=0.6ns
Load-Dependent delay: TPlhf=.0020ns, TPhlf=.0021ns

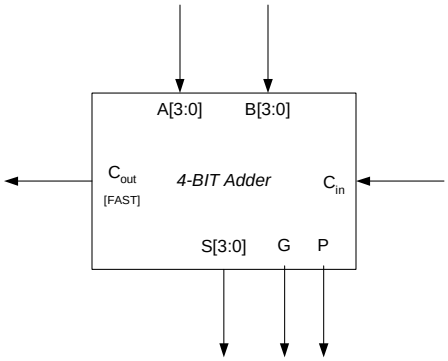
XOR: Input load: 200fF,
Propagation delay: TPlh=.8ns, TPhl=.8ns
Load-Dependent delay: TPlhf=.0040ns,TPhlf=.0042ns

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

Problem 2c [4pts]:
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):

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

Problem 2e [6pts]:
Suppose we wish to build a fast adder. One component might be a 4-bit adder with propagate and generate signals (such as might be used for carry-lookahead logic). Construct this adder as a 4-bit ripple adder internally (you can use your full-adders) with separate propagate and generate outputs. Let the output carry be a function of the propagate and generate signals. Draw a circuit for this component and compute the propagation delay for the slowest signal.



Problem 3: Non-Restoring Division

Here is the pseudo-code for an *unsigned* division algorithm. It is the non-restoring version of the last divider that we developed. Assume that **quotient** and **remainder** are 32-bit global values. As far as inputs are concerned **dividend** is 32 bits wide, while **divisor** is no more than 31 bits.

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

     /* Missing initialization instructions */

     MOBIUS64(remainder,quotient);
     while (count > 0) {
          count--;
          /* Low bit of quotient is inverted sign of remainder*/
          if (quotient & 0x1)
               remainder = remainder - divisor;
          else
               remainder = remainder + divisor;

          MOBIUS64(remainder,quotient);
     }

     /* Missing trailing instructions */
}
```

The `MOBIUS64(hi,lo)` instruction rotates the 64-bit value <hi,lo> around to the left 1 bit, wrapping the top bit of hi back to the bottom of lo and inverting it in the process. So, for instance: MOBIUS64(0xFFFFFFFF, 0x000000FF) ⇒ 0xFFFFFFFE, 0x000001FE
And: MOBIUS64(0x0FFFFFFF, 0x000000FF) ⇒ 0x1FFFFFFE, 0x000001FF

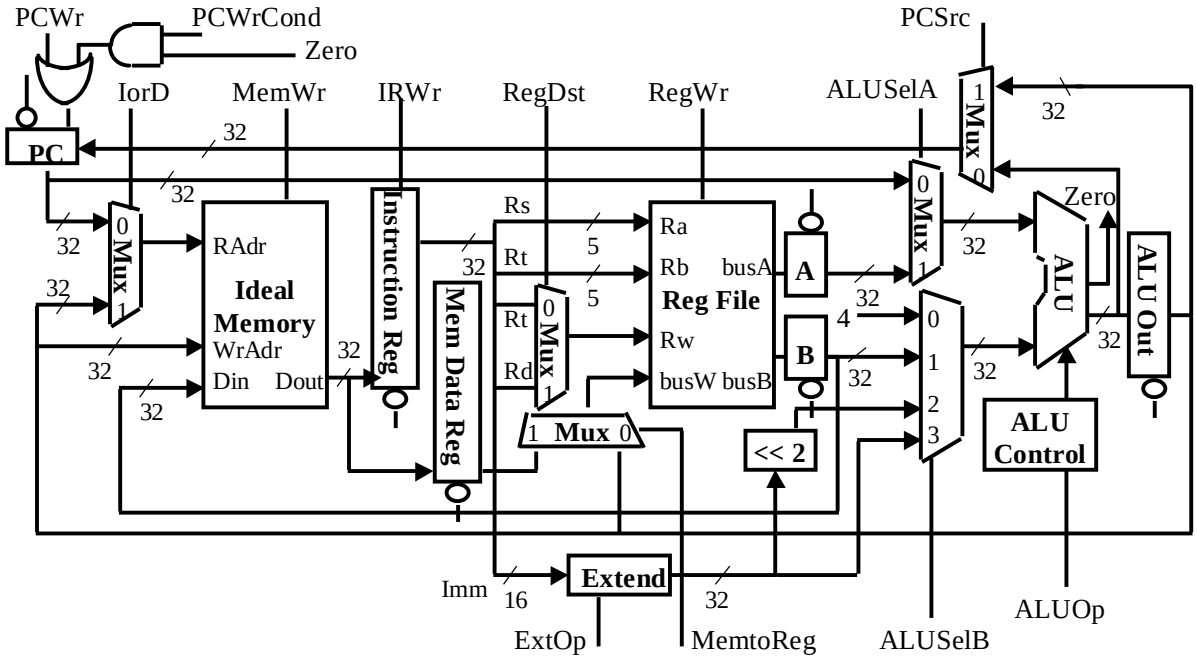
Problem 3a [5pts]:
Implement MOBIUS64($t1, $t0) as 6 MIPS instructions. Assume $t2 contains the constant 0x80000000. *Hint: what can different flavors of* `slt/sltu` *do to help?*

Problem 3b [5pts]:
The divide algorithm is incomplete. It is missing some initialization and some final code. What is missing? (Careful – what if the remainder is negative?)

Problem 3c [11pts]:
Assume that you have a MIPS processor that is missing the divide instruction. Assume that **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 MOBIUS64 as a pseudo-instruction that takes 3 registers (don't forget the constant in $t2). Make sure to adher to MIPS register conventions, and optimize the loop as much as possible.

Problem 3d [4pts]:
How will this algorithm have to change if the **divisor** is allowed to be 32 bits in size?

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 five new instructions to this data path:

`jal`	`<const>`	⇒	PC ← zero_ext(Instr[25:0]) ||00 R[31] ← PC + 4
`addiu`	`$rt, $rs, <const>`	⇒	R[rt] ← R[rs] + sign_ext(Imm16)
`multu`	`$rs, $rt`	⇒	<**`hi,lo`**> ← R[rs] × R[rt]
`mfhi`	`$rs`	⇒	`R[rs] ← <hi>`
`mflo`	`$rs`	⇒	`R[rs] ← <lo>`

1. The `jal` instruction is familiar to you from the normal MIPS instruction set.
2. The `addiu` instruction is also a normal MIPS instruction that has an immediate value
3. The multu instruction is an unsigned multiply, and the mfhi/mflo instructions are for getting the result.

Problem 4a [10pts]:
Describe/sketch the modifications needed to the datapath for the new instructions. Assume that the original datapath had only enough functionality to implement the original 6 instructions. Try to add as little hardware as possible. In particular***, you must use the existing ALU for multiply.*** You can showing additions to the data path – you do not need to completely redraw it. Make sure that you are very clear about your changes. *Hint: you need to add* ***hi*** *and* ***lo*** *registers. They can be made from shift-registers. Your CPI for multiply should be no more than 40. So: how will you loop 32 times for multu???*

Problem 4b [6pts]:
Draw a block diagram of as 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. *Hint: you will need to provide a branching facility to your sequencer. Make sure to include this functionality!*

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

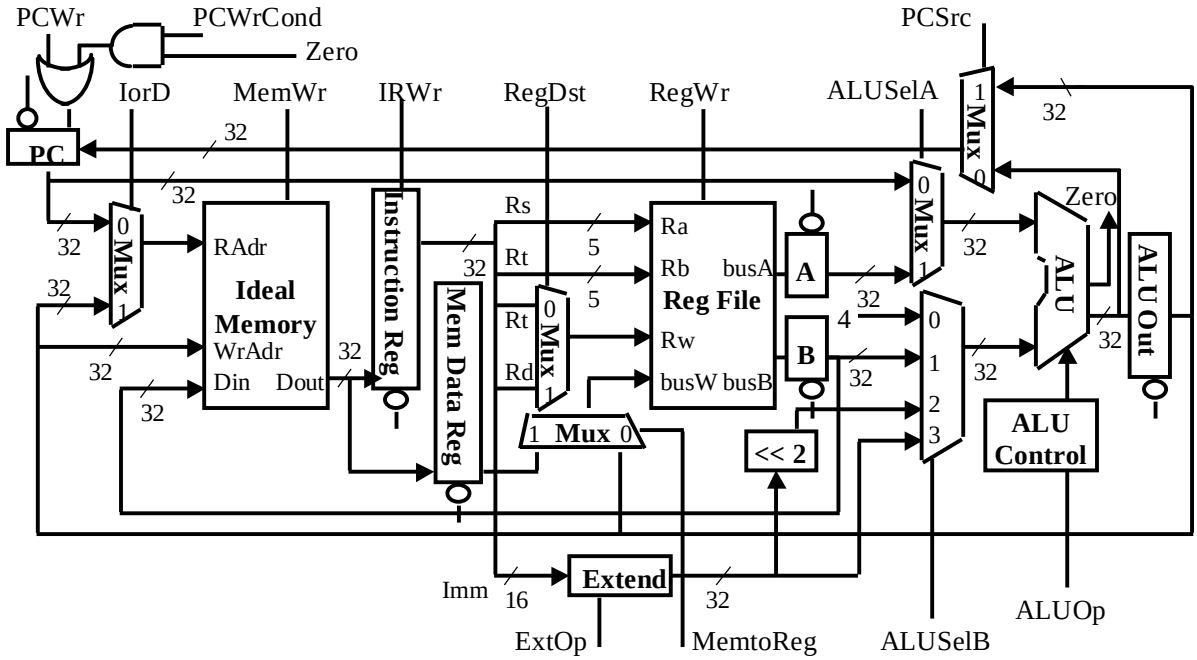
Problem 4d [10pts]:
Write complete microcode for the new instructions. Include the Fetch and Dispatch microinstructions. If any of the microcode for the original instructions must change, explain how.

Problem 4e [Extra Credit: 5pts]: Describe a (relatively) small change that will allow you to implement the signed `mult` instruction (think Booth!). Be complete with your answer (datapath/microcode, etc).

[This page left for scratch]

[This page left for scratch]

EXTRA SHEET FOR PROBLEM 4: FEEL FREE TO REMOVE



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

EXTRA SHEET FOR PROBLEM 4: FEEL FREE TO REMOVE

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 five new instructions to this data path:

`jal`	`<const>`	⇒	PC ← zero_ext(Instr[25:0]) ||00 R[31] ← PC + 4
`addiu`	`$rt, $rs, <const>`	⇒	R[rt] ← R[rs] + sign_ext(Imm16)
`multu`	`$rs, $rt`	⇒	<**`hi,lo`**> ← R[rs] × R[rt]
`mfhi`	`$rs`	⇒	`R[rs] ← <hi>`
`mflo`	`$rs`	⇒	`R[rs] ← <lo>`

4. The `jal` instruction is familiar to you from the normal MIPS instruction set.
5. The `addiu` instruction is also a normal MIPS instruction that has an immediate value
6. The multu instruction is an unsigned multiply, and the mfhi/mflo instructions are for getting the result.