Midterm I

October 10th, 2001
John Kubiatowicz

CS152 Computer Architecture and Engineering

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 [2 pts]:
How can clock skew cause incorrect behavior in a synchronous circuit?

Problem 1c [3 pts]:
Is the multi-cycle data path always faster than the single-cycle data page? Explain.

Problem 1d [2 pts]:
What is the difference between horizontal and vertical microcode?

Problem 1d [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	3
Floating Point	10	7

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

Problem 1e [4 pts]:
The micro-sequencer shown in class had three options: "zero (or fetch)", "dispatch", and "increment". Assume an 8-bit µPC if some external condition is true, or advance to the next microinstruction (increment the µPC), if the condition is false. Draw a block-diagram of this micro-sequencer, complete with 2-bit control input, 8-bit µPC output, 1-bit condition input, and 4-bit signed branch offset input.

Problem 1f [3 pts]:

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

Cur	Prev	Out
0	0	0
0	1	1
1	0	$\underline{1}$
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: General Base Conversions

In this problem, construct a subroutine that can convert a string of *any* base (from 2 to 16) to an integer. The specifications are as follows:

- The first argument (a0) contains a pointer to a null-terminated string
- The second argument (a1) contains the base and will be such that $2 \leq a1 \leq 16$ (you can assume that this is true)
- On exit, v0 will contain the result and v1 will contain an error code
- This procedure (call it "convert") must adhere to all MIPS conventions

If any character in the string (before the final null character) is not a legal character for the specified base, then the result is an error. Also, a zero-length string (no characters at all) is an error. Finally, an overflow (result doesn't fit in 32-bits) is an error. In the case of an error, the value of v0 is undefined.

Examples:

If the input string is "B6" and the base is 16, the result register (v0) should contain 182
If the input string is "B6" and the base is 12, the result register, should contain 138
If the input string is "B6" and the base is 10, this is a "bad character" error
If the input string is "" (i.e. first character is null), this is a "null string" error
If the input string is "10110" and the base is 2, the result register should contain 22
If the input string is "FFFFFFFFF" and the base is 16, this is an "overflow" error

Error codes (for v1):

Success: 0
Null String: 1
Bad character: 2
Overflow: 3

The important portion of the ASCII character table is as follows (Note that values are in HEX notation):

Character	ASCII Value
'0'	0x30
'1'	0x31
'2'	0x32
'3'	0x33
'4'	0x34
'5'	0x35
'6'	0x36
'7'	0x37
'8'	0x38
'9'	0x39

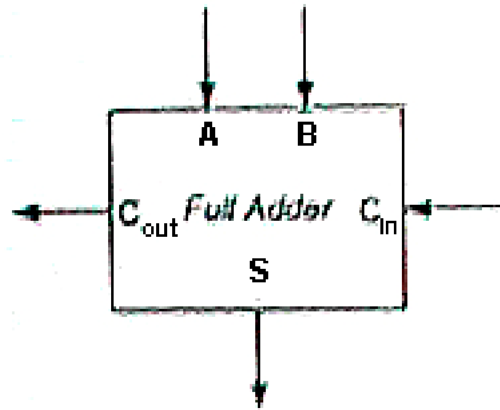
Character	ASCII Value
'A'	0x41
'B'	0x42
'C'	0x43
'D'	0x44
'E'	0x45
'F'	0x46

Problem 2a [13 pts]:
Write this routine in as few instructions as possible. Assume a virtual MIPS machine (no branch delay slots). Call this routine convert(). *Hint: process characters one at a time. Each new character will multiply the current result by the base.*

Problem 2b [2 pts]:
How would the arguments and return values to convert() have to change in order to allow it to operate on arbitrarily long input strings (i.e. to remove the possibility of an "overflow" error)?

Problem 3: Delay For a Full Adder

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



Problem 3a [4 pts]:
Implement a full adder using as few 2-input NAND, NOR, 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:

NAND: Input load: 60fF,
Propagation delay: TPlh=0.5ns, TPhl=0.2ns
Load-Dependent delay: TPlhf=.0021ns, TPhlf=.0020ns

NOR: Input load: 60fF
Propagation delay: TPlh=0.8ns, TPhl=0.1ns
Load-Dependent delay: TPlhf=.0042ns, TPhlf=.0010ns

XOR: Input load: 200fF
Propagation delay: TPlh=0.9ns, TPhl=0.9ns
Load-Dependent delay: TPlhf=.0032ns, TPhlf=.0030ns

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

Problem 3c [2pts]:
Compute the Load-Dependent delay for each of the two outputs:

Problem 3d [6 pts]:
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). Account for wire delay.

Problem 4: Hardware Square Root

Suppose that you have a 32-bit value, M, and you wish to find the closest integer, S, less than its square-root, sqr(M). Let's call S the "integer square root of M". Since you are forcing S to be an integer, you will end up with a remainder, $R = M - S^2$. In this problem, we will come up with an iterative mechanism to compute S one bit at a time.

Let us suppose that we have an estimate, S_i, for the square root of M. We will assume that Si is less than the desired value S, i.e. $S_i <= S$. Next, assume that we add a small increment to this estimate to make a better estimate, Si+1. Call this increment Ni+1:

$$S_{i+1} = (S_i + N_{i+1}) <= S$$

Now, the remainder after the first estimate is: $R_i = M - S_i^2$,
while after the second estimate is: $R_{i+1} = M - S_{i+1}^2 = M - S_i^2 - N_{i+1} * (2 * S + N_{i+1})$

So, each time we pass through the algorithm, we subtract the following from the remainder:

$$Q_{i+1} = R_i - R_{i+1} = N_{i+1} * (2 * S + N_{i+1})$$

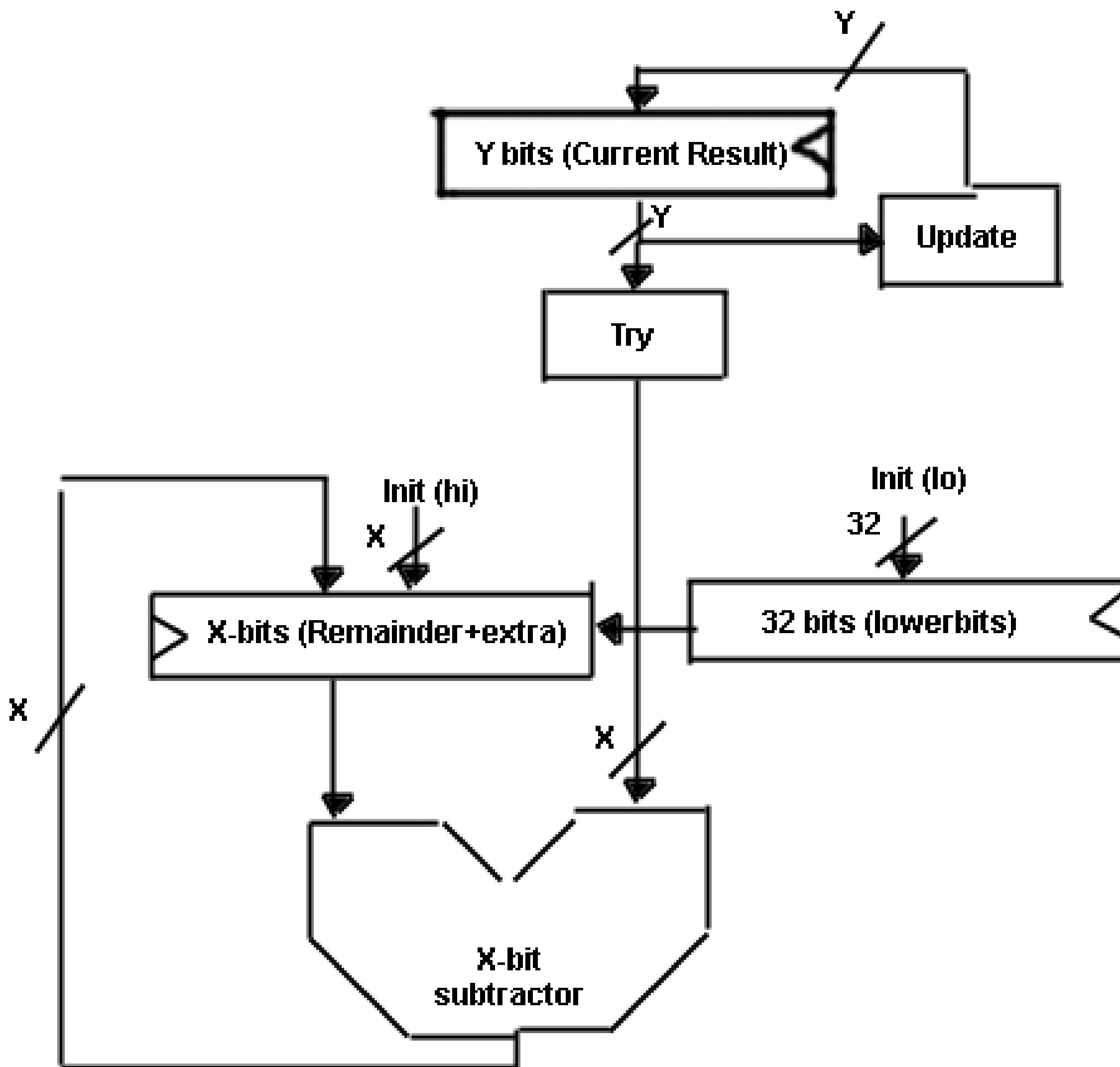
In binary, the increment values (the N's) are single bits. Thus, each iteration through the algorithm, we multiply the previous estimate by 2, add in the new bit (N_{i+1}), then shift by the number of zeros in N_{i+1} before subtracting from our remainder. This is very much like a divide in which the divisor keeps changing. For example, consider finding the 4-bit square root of 118:

```
     Starting:  M = R0 =  01110110  and  S0 = 0000
Try: N1 = 1000             -  1000    ←     (2*S0 + 1000) * 1000
                    R1 =  00110110   →     S1 = S0 + 1000 = 1000
Try: N2 = 0100             -  10100   ←     (2*S1 + 0100) * 0100
                    Result < 0       →     S2 = S1 = 1000
                    R2 =  00110110   (unchanged)
Try: N3 = 0010             -   10010  ←     (2*S2 + 0010) * 0010
                    R3 =  00010010   →     S3 = S2 + 0010 = 1010
Try: N4 = 1000             -    10101 ←     (2*S3 + 0001) * 0001
                    Result < 0       →     S4 = S3 = 1010
                    R4 =  00010010   (unchanged)
```

Final result: sqrt(01110110) = 1010 with 10010 remainder
or: sqrt(118) = 10 with 18 remainder!

Problem 4a [10 pts]:

Below is part of a circuit to extract square roots. This circuit is very similar to the circuit for divide. There are three things missing: (1) Internals for the "Update" and "Try" blocks, (2) logic to initialize the circuit, and (3) control logic. Draw these into the circuit. Draw the controller as a block with inputs and outputs (no internals). Draw all control signals! *Hint: part of the update of the remainder each cycle (Q_{i+1}) is accomplished by shifting the remainder to the left.*



Problem 4b [5 pts]:
Draw a state machine to control your circuit in Problem 4a. Include all of the control signals. Assume that there is a reset signal that starts the computation and that the controller produces a "Done" signal when finished.

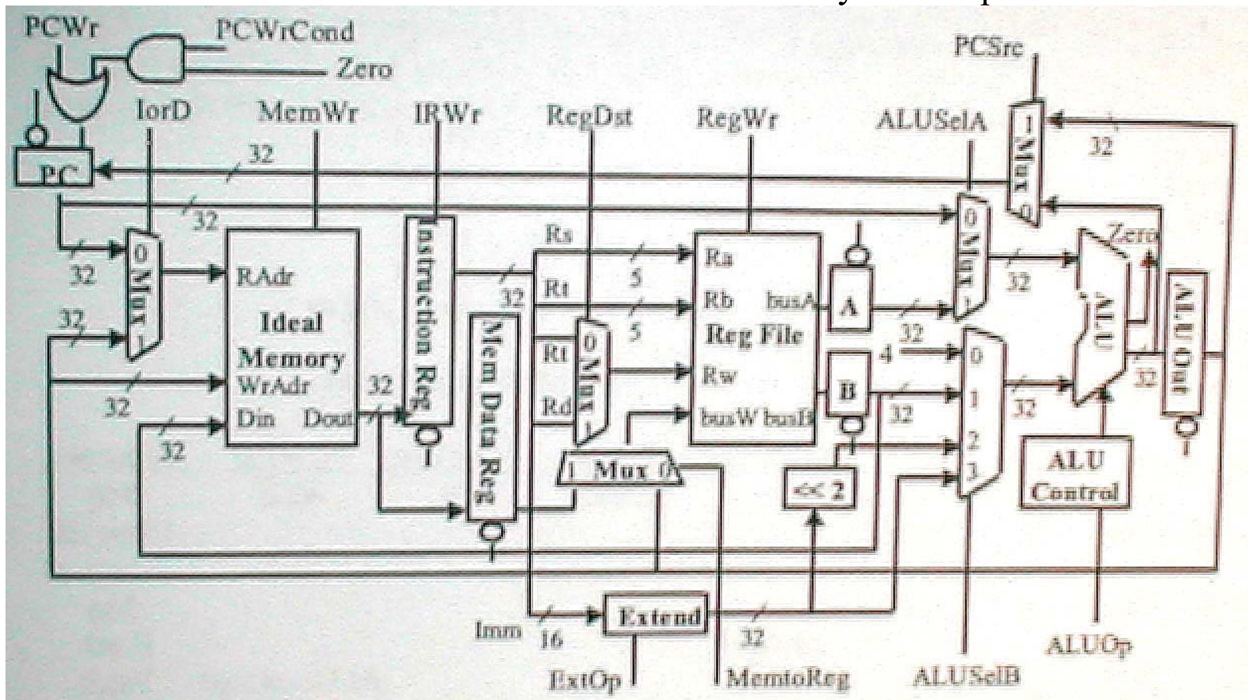
Problem 4c [2 pts]:
The circuit handles unsigned M. Is it easy to extend the algorithm for a signed M? Explain.

Problem 4d [2 pts]:
For a 64-bit, unsigned-value M, what is the largest possible integer square-root, S-max? How many bits would it take to represent (This is "Y" in the above circuit)? Explain. *(Hint: Start by finding the smallest integer that is larger than S-max)*

Problem 4e [3 pts]:
Also for a 64-bit unsigned-value M, what is the largest possible remainder, R-max? How many bits would it take to represent? Explain *(Use the same hint as above)*.

Problem 4f [3 pts]:
What is the maximum size of the ALF (in bits) that you would need to perform a 64-bit square-root? *(be careful here!)* (for both multiplication and division, the answer was 32-bits). Explain your answer (This is "X" in the above circuit).

Problem 5: 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])
SRC1	`Or`	ALU does logical OR
	`PC`	PC ➔ 1st ALU input
SRC2	`rs`	R[rs] ➔ 1st ALU input
	`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 ALF Zero is true, then ALUout ➔ PC
Sequence	`Seq`	Go to next sequential microinstruction
	`Fetch`	Go to first microinstruction
	`Dispatch`	Dispatch using ROM

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

op | rs | rt | rd | shamt | func = 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
LW	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>`	→	R[31] ← + 3 PC ← zero_ext(Instr[25:0]) || 00
`addiu`	`$rt, $rs, <const>`	→	R[rt] ← R[rs] + sign_ext(Imm16)
`bcp`	`$rd, $rs, $rt`	→	Copy block of *words* of length R[rt] from address R[rs] to address R[rd]

1. The `jal` and `addiu` instructions should be familiar to you from the normal MIPS instruction set.
2. Several notes about the block copy operation:
   a. The block copy operation has word addresses as input and a length in words. Keep this in mind (so as not to be off by a factor of 4 in an incorrect direction).
   b. Assume that the register file *cannot be altered* during this process, i.e. registers `$rs, $rt` and `$rd` must be the same before and after the execution of the instruction.
   c. Don't worry about the case in which source and destination blocks overlap.

Problem 5a [10 pts]:
Describe/sketch the modifications needed to the datapath for the new instructions (`jal`, `addiu`, and `bcp`). Assume that the original datapath had only enough functionality to implement the original 6 instructions. Try to add as little additional hardware as possible. *You are not required to redraw the whole datapath.* Make sure that you are very clear about your changes.

Problem 5b [5 pts]:
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 (*Hint: see problem [1e],* 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 the PCWrite field of the microinstruction)*

Problem 5c [4 pts]:
Describe changes to the microinstruction assembly language for these new instructions. How wide are the encoded microinstructions?

Problem 5d [6 pts]:
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).*