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

Fall 1999 John Kubiatowicz

Midterm I
SOLUTIONS

October 6, 1999
CS152 Computer Architecture and Engineering

Your Name:	
SID Number:	
Discussion Section:	

Problem	**Possible**	**Score**
1	20	
2	15	
3	35	
4	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)*

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

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

Problem 1b:
What is Amdahl's law for speedup? State as a formula which includes a factor for clock rate.

$$\textbf{Speedup} = \frac{\textbf{Time}_{\textbf{old}}}{\textbf{Time}_{\textbf{new}}} = \left(\frac{1}{(1-f) + \frac{f}{n}} \right) \times \left(\frac{\textbf{Freq}_{\textbf{new}}}{\textbf{Freq}_{\textbf{old}}} \right)$$

Where *f* = fraction of cycles sped up by optimization, n is the speedup.

Let us suppose that you have been running an important program on your company's 300MHz Acme II processor. By running a detailed simulator, you were able to collect the following instruction mix and breakdown of costs for ezach instruction type:

Instruction Class	Frequency (%)	Cycles
Integer arithmetic and logical	40	1
Load	20	1
Store	10	2
Branches	20	3
Floating Point	10	5

Problem 1c:
What is the CPI and MIPS rating of the Acme II for this program?

CPI = .4(1)+.2(1)+.1(2)+.2(3)+.1(5) = 1.9
MIPS = 300/1.9 = 157.9

Problem 1d:
Suppose that you turn on the optimizer and it eliminates 30% of the arithmetic/logic instructions (i.e. 12% of the *total* instructions), 30% of load instructions, and 20% of the floating-point instructions. None of the other instructions are effected. What is the speedup of the optimized program? (Be sure to state the formula that you are using for speedup and show your work)

The easiest way to compute this is to imagine that the original program had 100 instructions in it. Then, we can compute the number of cycles for the optimized version of this program vs the original:

$$\textbf{\textit{Speedup}} = \frac{\textbf{Time}_{\textbf{old}}}{\textbf{Time}_{\textbf{new}}} = \frac{\textbf{100} \times \textbf{1.9}}{\textbf{100} \times [\textbf{1.9} - \textbf{.12(1)} + \textbf{.06(1)} - \textbf{.02(5)}]} = \frac{\textbf{1.9}}{\textbf{1.62}} = \textbf{1.17}$$

Problem 1e:
What is the CPI and MIPS rating with the optimized version of the program? Compare your result to that of (1c) and explain the difference:

Our optimizer removed .12+.06+.02 = .2 of our instructions. That means that we need to rescale by 1/(1-0.2) = 1/0.8. You could imagine that the new table is:

Instruction Class	Frequency (%)	Cycles
Integer arithmetic and logical	$\frac{40(1-0.3)}{0.8} = 35$	1
Load	$\frac{20(1-0.3)}{0.8} = 17.5$	1
Store	$\frac{10}{0.8} = 12.5$	2
Branches	$\frac{20}{0.8} = 25$	3
Floating Point	$\frac{10(1-0.2)}{0.8} = 10$	5

Practically speaking, we have already done most of the required computation in the last problem: CPI = 1.62/0.8= 2.025, MIPS= 300/2.025 = 148.1

These numbers *seem* to reflect worse performance (higher CPI / lower MIPS) since the optimizer removed more fast instructions than slow ones. However, the program will run faster, since it has only 80% of the instructions of the original.

Problem 1f:
Now, suppose that the Acme III has just been introduced with a faster clock rate (450 MHz). However, in order to make the clock rate faster, the Acme engineers had to increase the CPI for arithmetic, logical, and load instructions to 2 cycles and floating point instructions to 6 cycles. What is the speedup of the Acme III over the Acme II on the *unoptimized* program? Show work

$\mathbf{CPI_{new}=0.6(2)+0.1(2)+0.2(3)+0.1(6) = 2.6}$

$$\textit{speedup} = \frac{\text{Time}_{\text{old}}}{\text{Time}_{\text{new}}} = \frac{\text{CPI}_{\text{old}}}{\text{CPI}_{\text{new}}} \times \frac{\text{Clock}_{\text{new}}}{\text{Clock}_{\text{old}}} = \frac{1.9}{2.6} \times \frac{450}{300} = 1.096$$

Problem 1g:
The engineers for Acme Inc are currently working on the Acme IV. Instead of increasing the clock rate again, they are working on reducing the time for the floating-point instructions. Use Amdahl's law to show the *maximum* speedup that you could expect between the Acme III and Acme IV on the u*noptimized* program (if the clock rates are both 450 MHz)?

Fraction of time devoted to float: $f = \frac{0.6}{2.6} = 0.231$

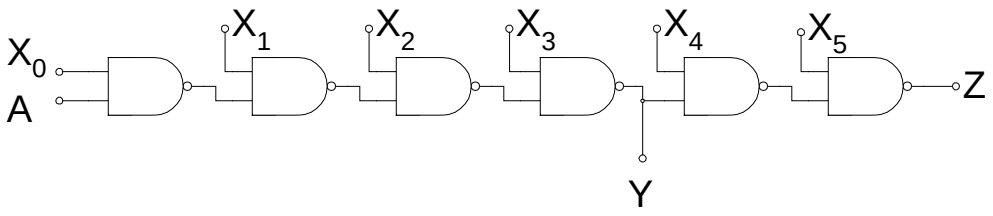
Max speedup = $\frac{1}{(1-f)} = \frac{2.6}{0.6} = 1.3$

Problem 2: Propagation Delay

Problem 2a:
Assume the following characteristics for NAND gates:

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

For the circuit below, assume that inputs $X_0 - X_5$ are all set to 1. What are the propagation delays from A to Y (for rising and falling-edges of Y)?



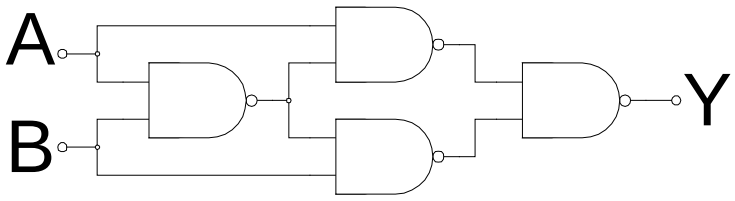
Because there are an even number of gates between A and Y, and because they are equally loaded, both transitions have equal propagation delay:

$$\mathbf{TAY\uparrow = TAY\downarrow = 2 \times \left[(0.002 \times 150 + 0.2) + (0.0021 \times 150 + 0.5) \right] = 2.63\ ns}$$

Or, including a wire-delay estimate:

$$\mathbf{TAY\uparrow = TAY\downarrow = 2 \times \left[(0.002 \times 300 + 0.2) + (0.0021 \times 300 + 0.5) \right] = 3.86\ ns}$$

Problem 2b:
Suppose that we construct a new gate, 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:

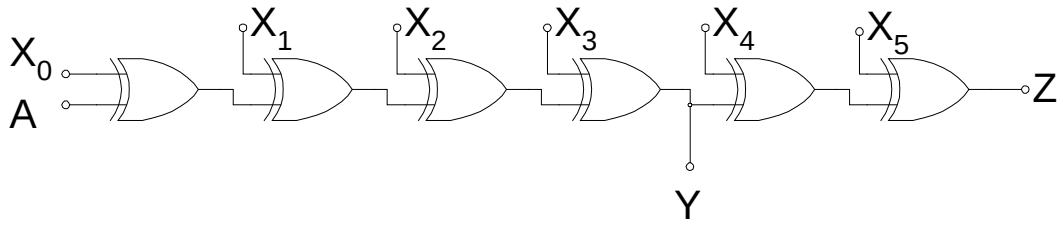
A Input Capacitance: **150+150=*300 fF***
B Input Capacitance: ***300fF***

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

Internal delays for A⇒Y, assuming that B is set to 1 (worst case delays):
TPAYlh: **0.2 + 0.002x300 + 0.5 + 0.0021x150 + 0.2 = 1.815 ns**
TPAYhl: **0.5 + 0.0021x300 + 0.2 + 0.002x150 + 0.5 = 2.13ns**

With estimated wire delay, these numbers would be: 2.73 and 3.06 respectively.

Problem 2c:
Now, suppose we use our new XOR gate in the circuit below. Let $X_0 - X_5$ be set to 1.
Compute the propagation delays from A ⇒ Y (both rising and falling edges):



This has the same symmetry as in part (a). So:

$$\mathbf{TAY\uparrow = TAY\downarrow = 2 \times [1.815 + (.002 \times 300) + 2.13 + (.0021x300)] = 10.35}\boldsymbol{ns}$$

or, with wire delay:

$$\mathbf{TAY\uparrow = TAY\downarrow = 2 \times [2.73 + (.002 \times 600) + 3.06 + (.0021x600)] = 16.5}\boldsymbol{ns}$$

Problem 3: 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, $\sqrt{M}$. 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 S_i is less than the desired value S, i.e. $S_i \le S$. Next, assume that we add a small increment to this estimate to make a better estimate, S_{i+1}. Call this increment N_{i+1}:

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

Now, the remainder after the first estimate is: $R_i = M - S_i^2$,

while after the second estimate is:

$$\begin{aligned} R_{i+1} &= M - S_{i+1}^2 \\ &= M - (S_i + N_{i+1})^2 \\ &= M - S_i^2 - N_{i+1} \times (2 \times S + N_{i+1}) \end{aligned}$$

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

$$R_i - R_{i+1} = \mathbf{N_{i+1} \times (2 \times S_i + N_{i+1})}$$

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 = 0001     -    10101   ⇐ (2×S2+0001)×0001
                   Result < 0    ⇒ S4=S3 = 1010
                   R4= 00010010 (unchanged)
```

Final result: $\sqrt{\mathbf{01110110_2}} = 1010_2$ with 10010_2 remainder

or: $\sqrt{\mathbf{118}} = 10$ with 18 remainder!

Problem 3a:
The above example showed unsigned M. Is it easy extend the algorithm for a signed M?

Yes: Since it doesn't make sense to take the square root of a negative number, we simply need to cause a "bad operand" fault if the sign bit of M is set.

Problem 3b:
From this point on, assume M is unsigned. 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? Explain without using a calculator. (*hint: Start by finding the smallest integer that is bigger than S_{max}).*

First, note that $2^{32} \times 2^{32} = 2^{64} = M_{max} + 1$. So, $S_{max} < 2^{32}$. Further, we know that:

$$(2^{32} - 1)^2 < (2^{32})^2 = M_{max} + 1 \Rightarrow (2^{32} - 1)^2 \leq M_{max}$$

Thus, we can conclude: $S_{max} = (2^{32} - 1)$

This takes 32 bits to represent.

Problem 3c:
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 without using a calculator. (*Use the same hint as above).*

First, note that the spaces between successive squares keeps increasing:

1, 4, 9, 16, 25, 36, 49, ...

This means that the maximum remainder would be between S_{max}^2 and $(S_{max}+1)^2 = M_{max}+1$

$$R_{max} = (M_{max} - S_{max}^2) = (S_{max}+1)^2 - 1 - S_{max}^2 = S_{max}^2 + 2S_{max} + 1 - 1 - S_{max}^2 = 2S_{max}$$

Thus, $R_{max} = 2(2^{32} - 1)$

This would take 33 bits to represent.

Here is pseudo-code for a square root algorithm. *Assume that the input value of M has been restricted so that S_{max} is no more than 31 bits in size and R_{max} is no more than 32 bits in size.* Let **`Result`** and **`Remain`** be 32-bit global values which will store the square root and remainder respectively. Inputs $\mathbf{M_{hi}}$ and $\mathbf{M_{low}}$ are 32-bit arguments that give the upper and lower 32-bits of the input. **This code is modeled after version 3 of the divider from class:**

```
isqrt(Mlow,Mhi) ⇒ (Result, Remainder)
{       /* All temporaries are 32-bit values */
        int nextbit, temp, topbit, lowerbits;

        /* missing initialization instructions */

        while (nextbit > 0) {
            ROL96(topbits,Remainder,lowerbits);

            /* Above restrictions on M ensure temp only 32 bits. */
            temp = (2 * Result) | nextbit;
            if (topbits > 0 || Remainder ≥ temp) {
                Result = Result | nextbit;
                SUBcarry(topbits, Remainder, temp);
            }
            nextbit = nextbit >> 1;
        }
}
```

The `ROL96(hi,low,extra)` pseudo-instruction takes three 32-bit registers and treats them as a combined 96-bit register. It shifts the combined value left by one position, inserting a zero at the far right (of the `extra` register).

The `SUBcarry(hi,low,subvalue)` pseudo-instruction takes three 32-bit registers. It treats the first two as a combined 64-bit register. It subtracts the 32-bit `subvalue` from this 64-bit register.

Problem 3d:
The pseudo-code is missing some initialization instructions. What should be there? (*hint: look at the example square root again and try to figure out what the various arguments to ROL96 must be. Also, make sure that every variable has an initial value!):*

`nextbit = 2`^{31}

`topbits = 0`
`Remainder =` $\mathbf{M_{hi}}$
`lowerbits =` $\mathbf{M_{low}}$

Initialization for the 96 bit shift register

`Result = 0`

Problem 3e:
Assume that you have a MIPS processor that is missing the `isqrt()` instruction. Implement `isqrt()`as a procedure. Assume that M_{low} and M_{high} are in the $a0 and $a1 registers respectively, and that the `Result` and `Remain` values are returned in registers $v0 and $v1 respectively. You can use ROL96 and SUBcarry pseudo-instructions, but don't use any other pseudo-instructions. Make sure to adhere to all MIPS calling conventions!

In the following solution, we assume that there is no branch delay slot.

```
isqrt:      lui     $t0, 0x8000      ; Initialize nextbit
            ori     $t1, $r0, 0      ; Initialize topbits
            ori     $v1, $a1, 0      ; Initialize Remaider
            ori     $t2, $a0, 0      ; Initialize lowerbits
            ori     $v0, $r0, 0      ; Initialize Result
loop:       ROL96   $t1,$v1,$t2
            add     $t3, $v0, $v0
            or      $t3, $t3, $t0
            bne     $t1, $r0, doit   ; topbits ≠ 0, go for it
            sltu    $t4, $v1, $t3    ; Check: Remainder < temp?
            bne     $t4, $r0, endloop  ; Yup, skip
doit:       ori     $v0, $v0, $t1    ; Add in bit to Result
            SUBcarry$t1, $v1, $t3    ; Calculate new remainder
endloop:    srl     $t3, $t3, 1      ; New value of nextbit
            bne     $t3, $r0, loop
            jr      $ra              ; Done, return with result
```

Problem 3f:
Implement the ROL96($t0,$t1,$t2) pseudo-instruction in 7 MIPS instructions. Assume that $t0, $t1, and $t2 are the three input registers (with $t0 the most significant).
(hint: what happens if you use signed slt on unsigned numbers?)

Soln: We are going to use the assembler register $at as a temp to hold the MSBs

```
slt       $at, $t1, $r0        ; Get top bit of $t1
sll       $t0, $t0, 1
or        $t0, $at, $t0
slt       $at, $t2, $r0        ; Get top bit of $t2
sll       $t1, $t1, 1
or        $t0, $at, $t1
sll       $t2, $t2, 1
```

Problem 3g:
Implement the SUBcarry($t0,$t1,$t2) pseudo-instruction in 3 MIPS instructions.

Soln: Compute the carry out and put it in $at. Be careful to use `sltu`***!***

```
sltu   $at, $t1, $t2
sub    $t1, $t1, $t2
sub    $t0, $t0, $t3
```

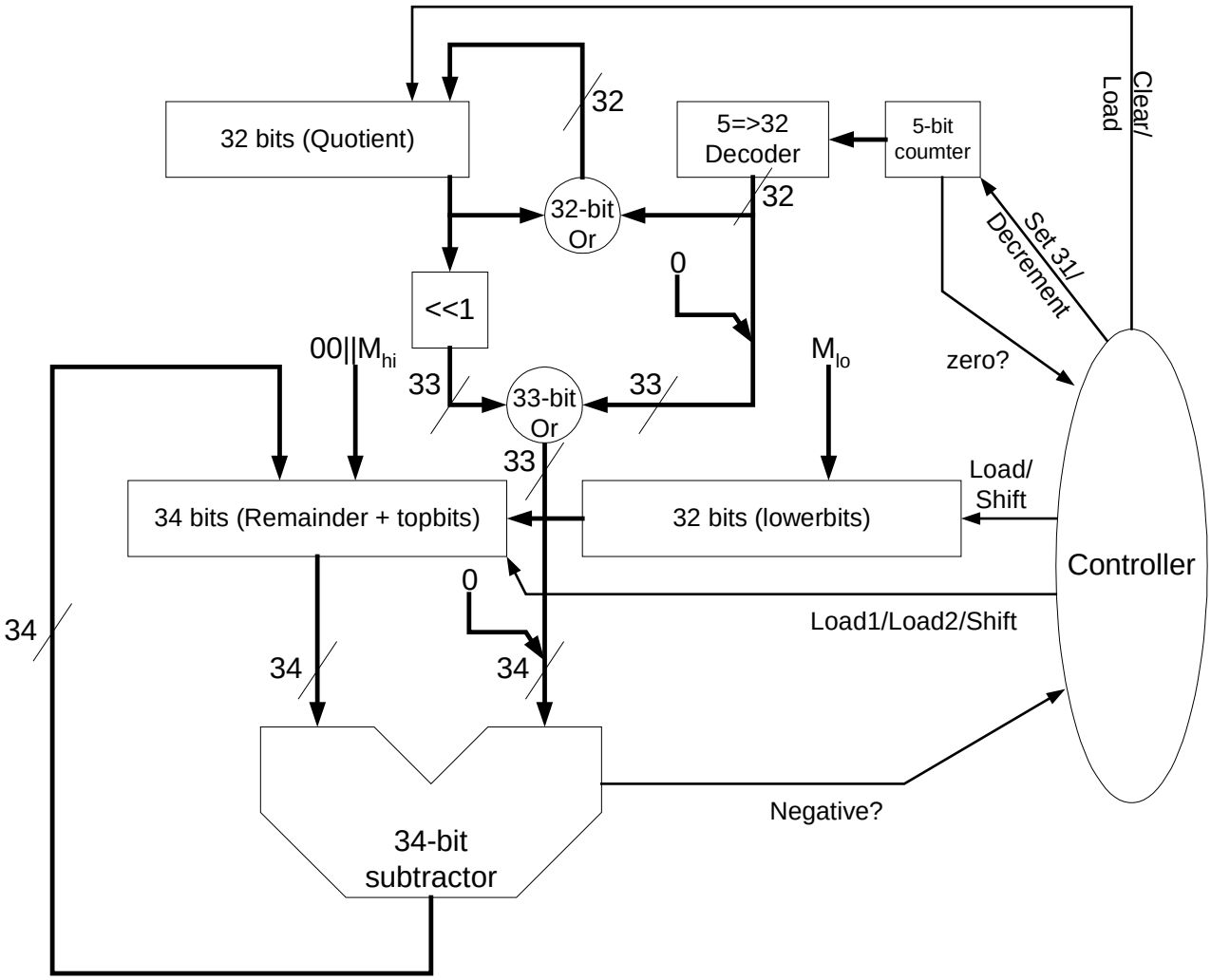
Problem 3h:
What is the maximum "CPI" of your `isqrt()` procedure? (i.e. what is the total number of cycles to perform an isqrt)? Assume that each real MIPS instruction takes 1 cycle, and pseudo-instructions ROL96 and SUBcarry take 7 and 3 cycles respectively:

Number of cycles in inner loop = 18
Start up = 5 cycles
Ending = 1 cycle

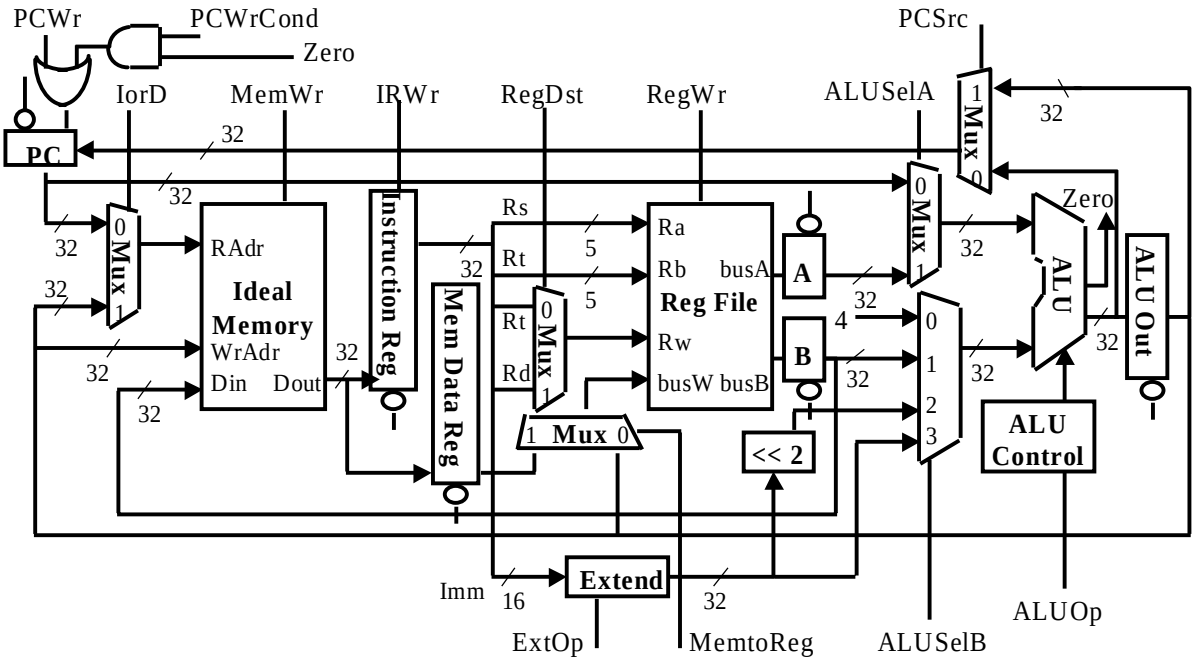
CPI for this "instruction" = 5 + 18x32 + 1 = 582
`

EXTRA CREDIT [5pts => Save until last!]:
Draw the data path for a hardware square-root engine that does 64-bit square-roots. Explain what you are doing and how this will be controlled.

The following data path will do the trick. Notice that we have essentially duplicated the algorithm in hardware. A little thinking will verify that you never need more than 34-bits inthe remainder register to handle the maximum temporary results.



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 three new instructions to this data path:

```
lui     $rd, <const>          ⇒  R[rd] ← Imm16 || 0000000000000000
multacc $rd, $rs, $rt         ⇒  R[rd] ← (R[rs]×R[rt]) + R[rd]
bltual  $rs, $rt <offset>     ⇒  if (R[rs] < R[rt]) then
                                     PC ← PC + 4 + sign_ext(Imm16) || 00
                                     R[31] ← PC + 4
                                 else
                                     PC ← PC + 4
```

1. The `lui` instruction is familiar to you from the normal MIPS instruction set. It places the 16 bit immediate field into the upper 16 bits of R[rd], filling the lower 16 bits of R[rd] with zeros. *Important note: the encoding for the `lui` instruction has a zero in the rs field.*

2. The `multacc` instruction (multiply-accumulate) uses register R[rd] as both a source and a destination register. It multiplies the values R[rs] and R[rt], adds the result to register R[rd], then places the result back into register R[rd]. Assume that this instruction does not overflow.

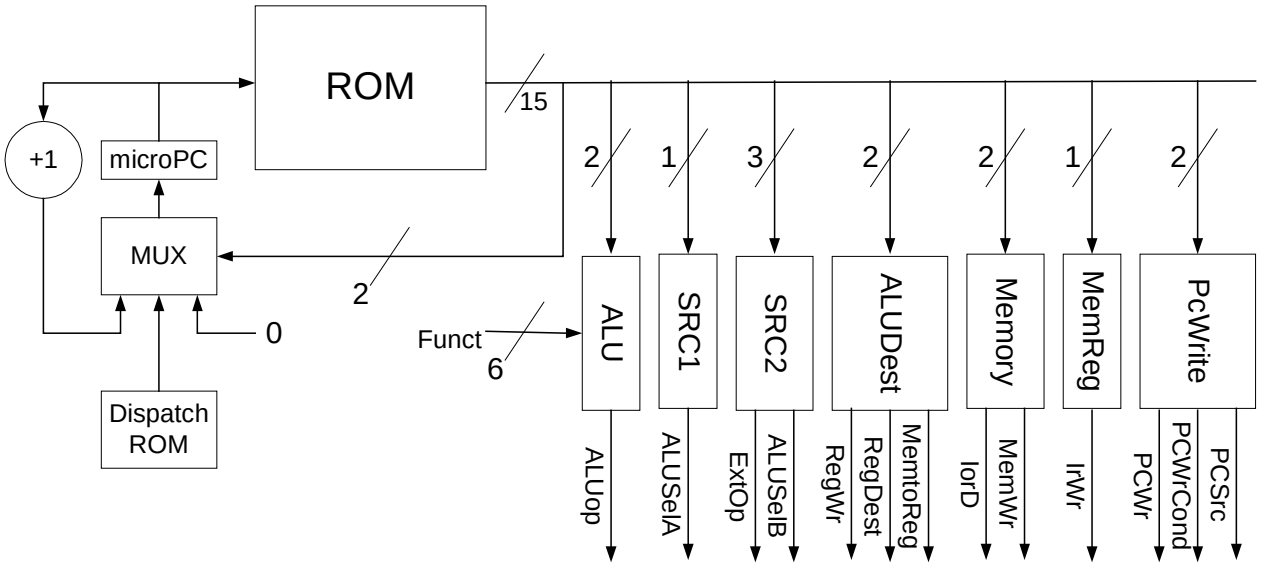
3. The `bltual` instruction (branch on less than unsigned and link) checks to see if R[rs] is less than R[rt]. If it is, it will save the PC in $ra (like `jal`), then branch to the offset.

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

15 = 2+1+3+2+2+1+2+2

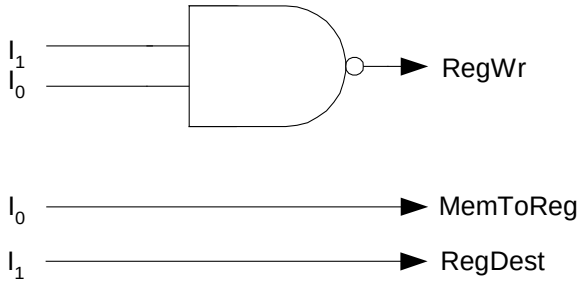
The trickiest part of this computation is the PC Write field. We have to remember to represent the "do nothing" option, which means that there are actually three different values for the PC Write field.

Problem 4b:
Draw a block diagram of a microcontroller for the unmodified datapath. 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).*

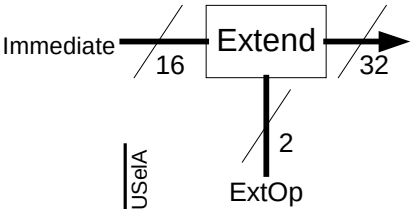


Problem 4c:
Come up with a binary encoding for the `ALUDest` field of the microinstruction (`rd-ALU`, `rt-ALU`, `rt-Mem`, or `blank`). Construct logic which maps this binary field to the appropriate control signals from problem 4b.

Name	*Code*
rt-ALU	*00*
rt-mem	*01*
rd-ALU	*10*
blank	*11*



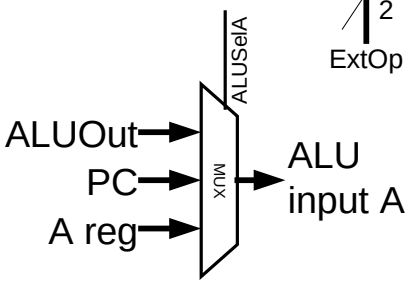
Problem 4d:
Describe/sketch the modifications needed to the datapath for the new instructions (`lui`, `multacc`, and `bltual`). Assume that the original datapath had only enough functionality to implement the original 6 instructions. Try to add as little additional hardware as possible. Make sure that you are very clear about your changes.



`lui`: Enhance the extend box to support a "Shift left by 16" mode.

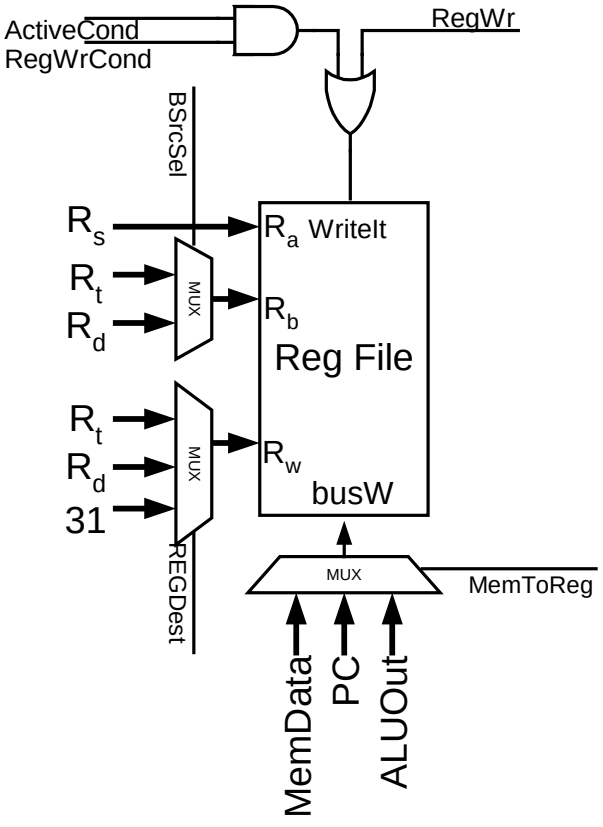
`multacc:`

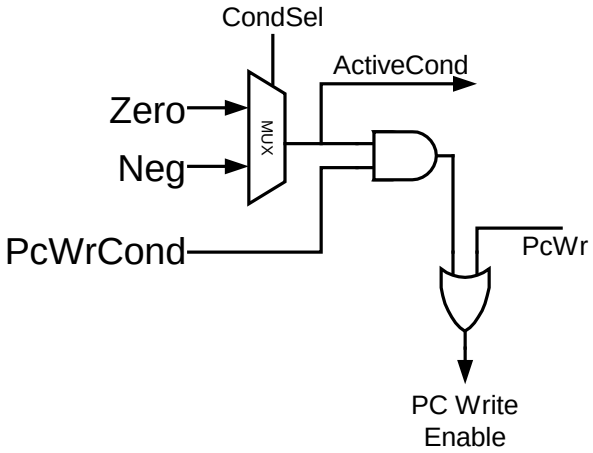
1. *Enhance the ALU to include multiplication.*
2. *Feed back ALUout register to ALUSelA mux*
3. *Place new mux at the input to the Rb input of the register file to select between the Rt and Rd fields of the instruction*



`bltual:`

1. *Enhance ALU to include a "NEG" output to indicate that output is negative (note that it is not enough to specify the sign bit of the result, since we are dealing with unsigned numbers).*
2. *Feed PC into MemToReg mux*
3. *Enhance PC control with new signal (CondSel), which selects either the "Zero" or "NEG" signals to make a decision.*
4. *Enhance RegDst MUX to include the explicit number "31" as a destination (see above)*
5. *Enhance the register write signal to include option of only writing if the condition specified by "CondSel" is true.*





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

New Field: BSRC. Possible values: GetRT, GetRD, blank. GetRT and blank are equivalent

Addition to ALU field:	*Mul*	*(Do a multiplication)*
Addition to SRC1 field:	*ALUOut*	*(Use ALUOut register value)*
Addition to SRC2 field:	*Shift16*	*(Use value of immediate shifted left by 16)*
Addition to ALUDest field:	*ra-PC-cond*	*(Write PC to $ra if PC condition is true)*
Addition to PCWrite:	*ALUNegBr*	*(Update PC if ALU NEG output is true)*

Need 1 new bit for BSRC, 1 additional bit for ALU field, 1 additional bit for SRC1, 0 additional bits for SRC2 field, 1 additional bit for ALUDest, 0 additional bits for PCWrite. So, total = 19

Problem 4f:
Write complete microcode for the three 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	BSRC	SRC1	SRC2	ALUDest	Memory	MemReg	PCWrite	Sequence
Fetch	Add		PC	4		ReadPC	IR	ALU	Seq
Dispatch	Add	GetRT	PC	ExtShft					Dispatch
lui	Add		rs	Shift16					Seq
					rd-ALU				Fetch
multacc	Mult	GetRD	rs	rt					Seq
	Add		ALUOut	rt					Seq
					rd-ALU				Fetch
bltual	Sub		rs	rt	ra-PC-cond			ALUNegBr	Fetch

Note that we assert GetRT during the dispatch stage, so that the first post-dispatch cycle of every instruction has the value of R[rt] in the "B" register (so to not mess up our other instructions). Since there are only two options for the BSRC field, you could imagine that "blank" produces this normal behavior. However, we have coded it explicitly in order to make our point.

Problem 4g:
What are the CPI values for each of the three new instructions?

lui: CPI = 4 (if you didn't go through an "Add", this could be as low as 3)
multacc: CPI = 5
bltual: CPI = 3