UCLA Samueli School of Engineering

CS M151B / EE M116C

MIDTERM EXAM

All work and answers should be written directly on these pages, use the backs of pages if needed.

This is an open book, open notes quiz – but you cannot share books or notes.

We will follow the departmental guidelines on reporting incidents of academic dishonesty.

Keep your eyes on your own exam!

NAME: _____

ID: _____

Problem 1: _____ (18)

Problem 2: _____ (14)

Problem 3: _____ (18)

Problem 4: _____ (14)

Problem 5: _____ (16)

Problem 6: _____ (10)

Problem 7: _____ (10)

Total: _____ (out of 100)

1. ***This ISA Question About Tradeoffs (18 points)***: For each of the following design decisions, circle the impact on the three components of execution time.

   A. We increased the size of the register file in the MIPS architecture for our single cycle datapath, which reduced register spilling. The increase in the number of bits in the register specifier field meant that we had to reduce other instruction fields, but assume there was no performance impact from that reduction in other instruction fields.

      a. Instruction Count: could increase **could decrease** must stay the same

      b. CPI: could increase could decrease **must stay the same**

      c. Cycle Time: **could increase** could decrease must stay the same

   B. We have a MIPS processor design (not a single cycle design) with a compiler. We make one change – we install a new compiler that employs a new strength reduction optimization: the compiler avoids using a single complex, multi-cycle instruction and instead uses multiple simpler instructions that each execute in fewer cycles.

      a. Instruction Count: **could increase** could decrease must stay the same

      b. CPI: could increase **could decrease** must stay the same

      c. Cycle Time: could increase could decrease **must stay the same**

   C. We manufacture our existing single cycle MIPS architecture datapath using new transistors that have lower latency. We run the same applications as before, without the need to recompile anything.

      a. Instruction Count: could increase could decrease **must stay the same**

      b. CPI: could increase could decrease **must stay the same**

      c. Cycle Time: could increase **could decrease** must stay the same

2. ***LWA: Straight Outta Computation (14 points)***: Consider the single cycle datapath we covered in class. The processor has a 3.5 GHz clock. We are running an application with 100 billion instructions. The application has the following instruction mix.

Instruction	% of Instructions
LW	40%
SW	10%
Other arithmetic R-Type or I-Type (e.g. ADD, ADDI, AND, OR, SUB)	30%
BEQ/BNE	15%
J	5%

What is the Execution Time for this application running on this processor? Show your work below.

CT:_____2.86e-10_____

CPI:_____1.0_____

IC:_____1e11_____

ET: ___28.57_____

CT = 1/3.5GHz = 2.86e-10 s
CPI = 1.0
IC = 1e11 instructions

ET = CT x CPI x IC = 2.86e-10 x 1 x 1e11 = 28.57

Now we are going to modify this architecture and the application. 25% of all LW instructions are immediately followed by an ADD instruction that uses the result of the LW. For example:

```
LW $t0, 4($s0)
ADD $s1, $s1, $t0
```

In these cases, the result written by the LW (e.g. register $t0) is only used once, by the following ADD instruction, and does not need to be stored for future instructions (e.g. the value in register $t0 is live up to the point of the ADD instruction, and the value is dead after that). So in all of these cases, we are going to replace these two instructions with a single **new** instruction we are adding to the ISA – the LWA instruction. The LWA instruction is an I type instruction with the following pseudocode:

R[RT] += M[R[RS] + SE(I)]

So the example code:

```
LW $t0, 4($s0)
ADD $s1, $s1, $t0
```

can be replaced with a single LWA instruction:

```
LWA $s1, 4($s0)
```

Now let's assume that the implementation you made adds 300 picoseconds of latency to the critical path of the single cycle datapath. Find the new Execution Time for this modified architecture and application. Show your work below.

CT:_____5.86e-10_____

CPI:_____1_____

IC:_____9e10_____

ET: _____52.71 s_____

CT = CTold + 300ps

25% of LW (and equivalent # of R-type) are replaced with a single instruction – so 10% of instructions are removed. IC = .9*ICold

ET = CT x CPI x IC = 52.71 s

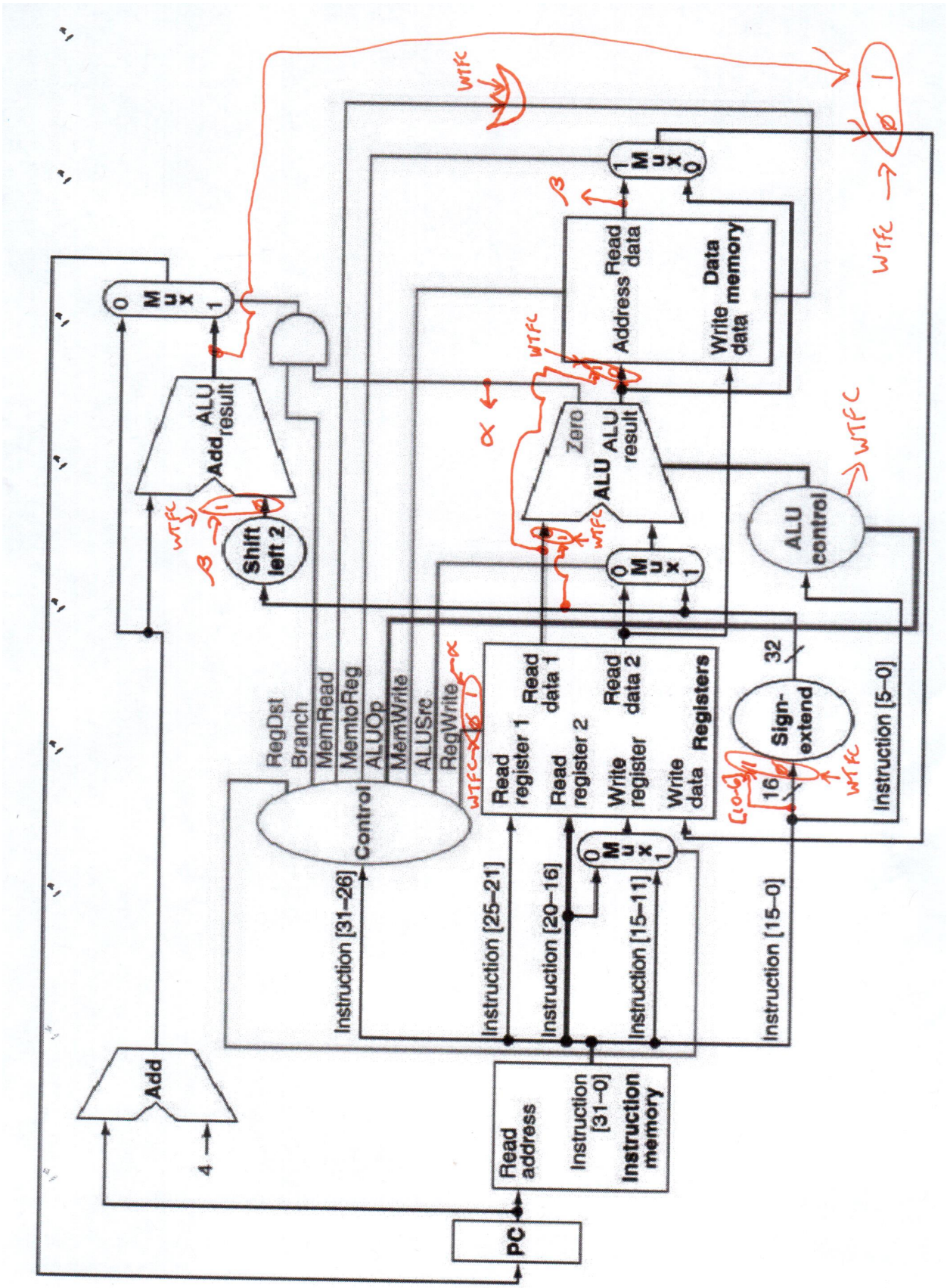
3. ***A Perfectly Reasonable and Useful Instruction (18 points)***: Extend the single cycle datapath and control we covered in class with the ***WTF*** instruction:

   WTF
   - R-Type Instruction (opcode is the same as all other R-Type Instructions)
   - Function field is unique.
   - Pseudocode:

   If (R[RT]==SE(SA))
     R[RD]=PC+4+M[R[RS]]

   This is an instruction that checks if the value in register RT is equal to the sign extended value in the shift amount field of the R-type format. Note that this instruction uses the shift amount field in a way not originally intended - and you may assume that sign extension hardware will extend a 5-bit value into a 32-bit value. If the value in RT is equal to the sign extended shift amount field of the instruction, then we set register RD to the sum of the PC+4 and the contents of memory at the address contained in register RS.

   Implement your solution on the following two pages. All other instructions supported in the datapath/control must still work correctly after your modifications. You should not add any additional ALUs, register file ports, or ports to memory.


Add
4
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
ALU
Zero
ALU result
Add
Shift left 2
Address
Read data
Write data
Data memory
ALU control
Sign-extend
16
32
M u x
0
1

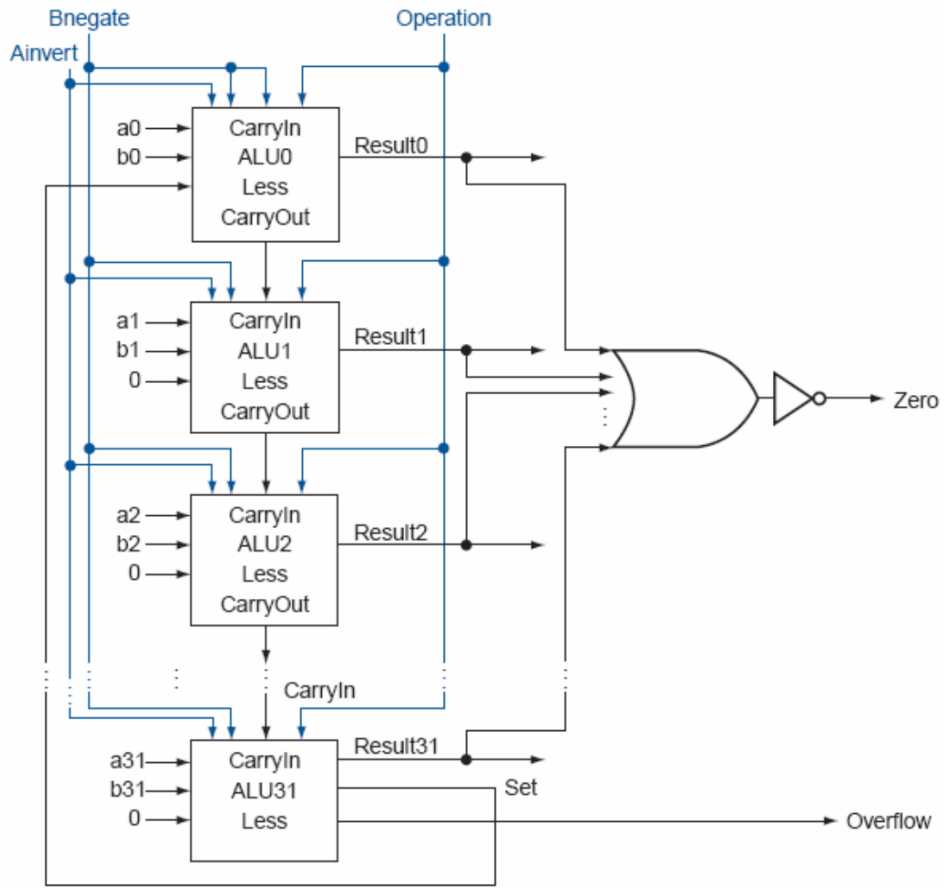
Main Controller

Input or Output	Signal Name	R-format	lw	sw	Beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

ALU Controller

opcode	ALUOp	Operation	funct	ALU function	ALU control	WTFC
lw	00	load word	XXXXXX	add	0010	Ø
sw	00	store word	XXXXXX	add	0010	Ø
beq	01	branch equal	XXXXXX	subtract	0110	Ø
R-type	10	add	100000	add	0010	Ø
		subtract	100010	subtract	0110	Ø
		AND	100100	AND	0000	Ø
		OR	100101	OR	0001	Ø
		set-on-less-than	101010	set-on-less-than	0111	Ø
		WTF	UNIQUE	SUBTRACT	Ø11Ø	1

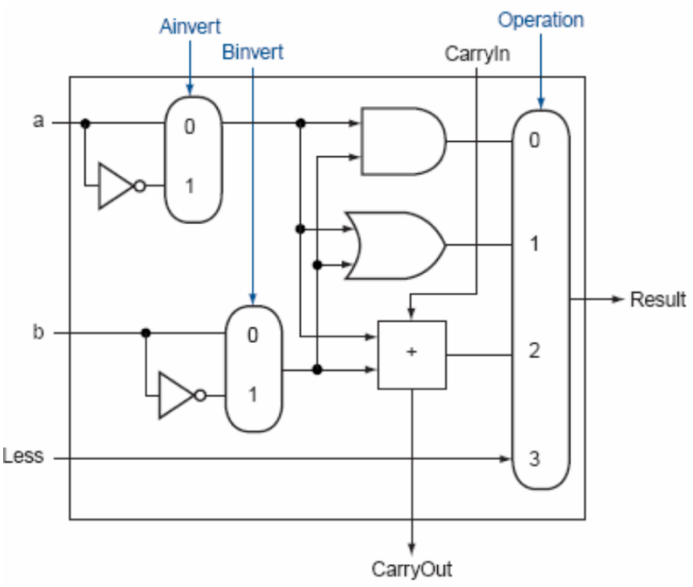
4. ***The Sum of all Fears (14 points)***: Here is the 6-function 32-bit ALU we covered in class:



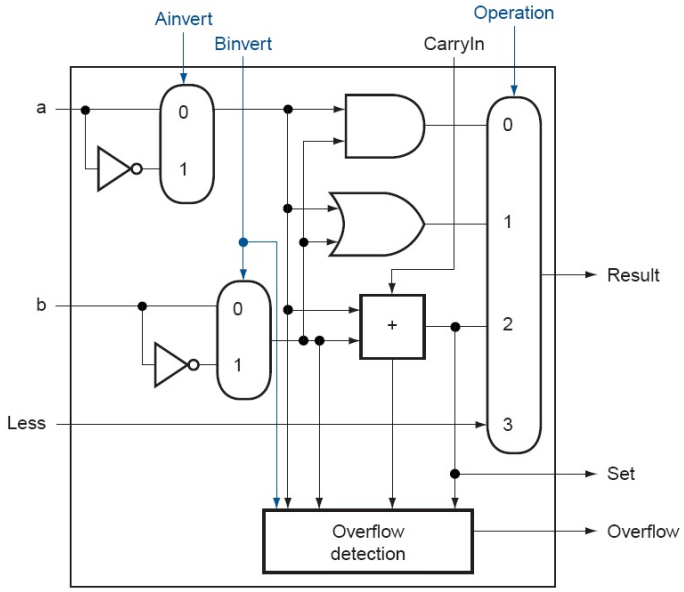
This 32-bit ALU has inputs A and B (broken down into individual bits a0, a1, a2, etc), control signals Ainvert, Bnegate, and Operation, and outputs Result (broken down into individual bits Result0, Result1, etc), Zero, and Overflow. Recall that this ALU includes an implementation of a set-less-than function (slt). The 6 functions and their control signals are shown below:

Function	Ainvert	Binvert	Operation
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
slt	0	1	11
nor	1	1	00

As we covered in class, each 1-bit ALU (except the ALU for the most significant bit) has the following design:

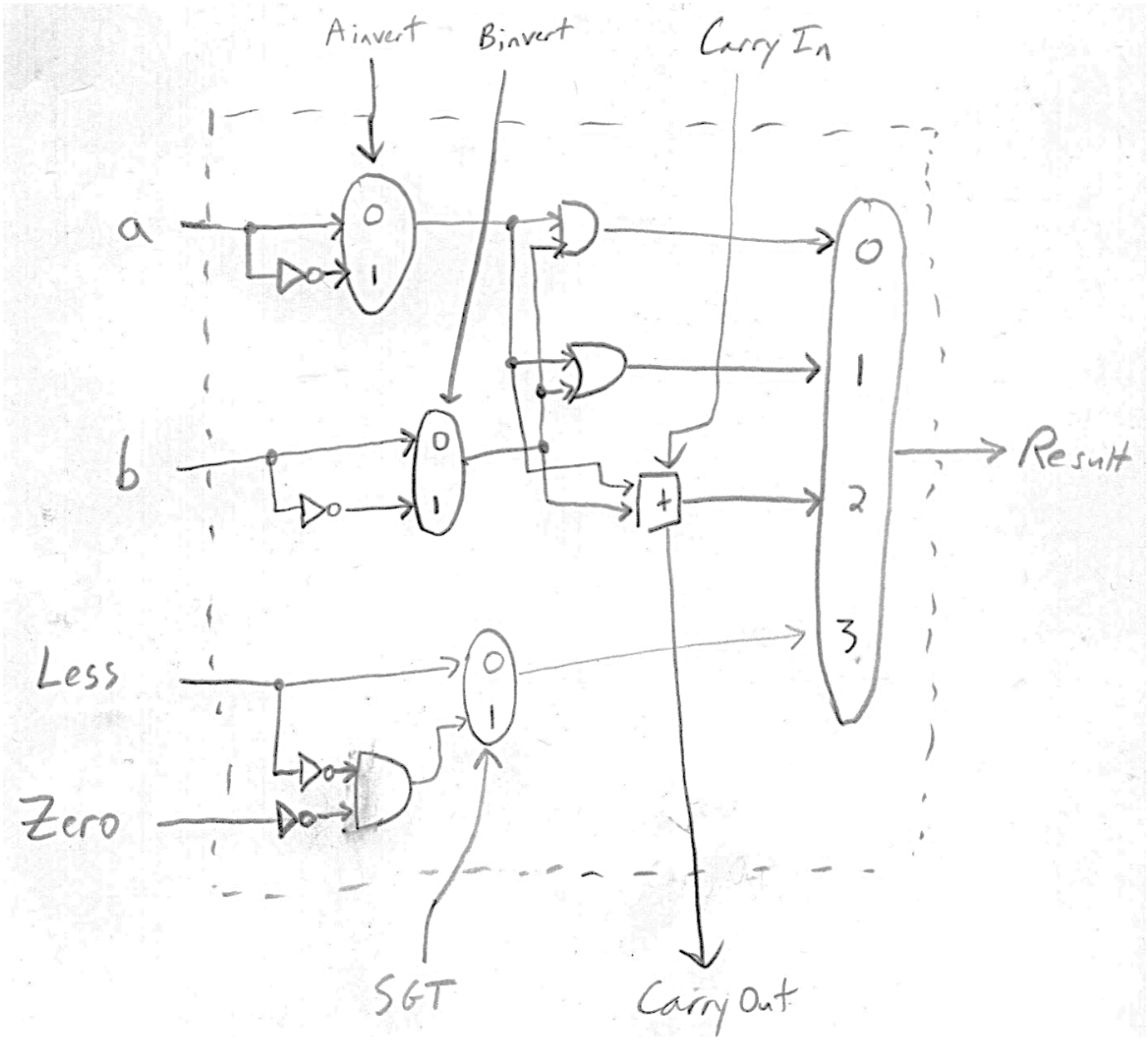


And the ALU for the most significant bit has the following design:



Given this design (again, the one we covered in class), you want to add support for a new function – so you are designing a 7-function 32-bit ALU. The new function will be set-greater-than (sgt) which will set the output to 1 if a > b and 0 otherwise.

Your friend suggests using the slt function as a starting place to building the sgt function. They suggest a change to just the ALU for the least significant bit in this professionally drawn diagram below:

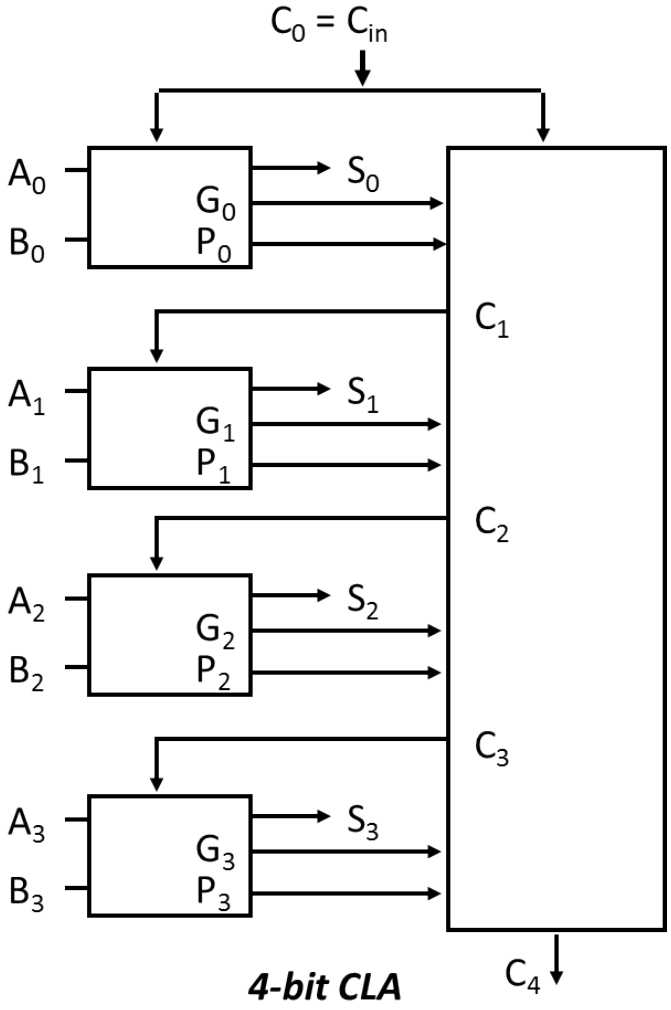


The change here is related to the Less input. There is a control signal (SGT) that selects between the regular less input and the AND of the negation of the Less input and the negation of the Zero output of the entire 7-function 32-bit ALU. Your friend argues that negation of the Less input will be 1 when a ≥ b (NOT slt) and that the negation of the Zero output will be 1 when a ≠ b (just like a BNE instruction). So if both of these are 1, it will be the case that a > b (hence the AND). The other ALUs (other than the LSb) are unmodified.

There is an error in your friend's overall design. In at most three sentences describe (1) what is wrong and (2) how to fix it:

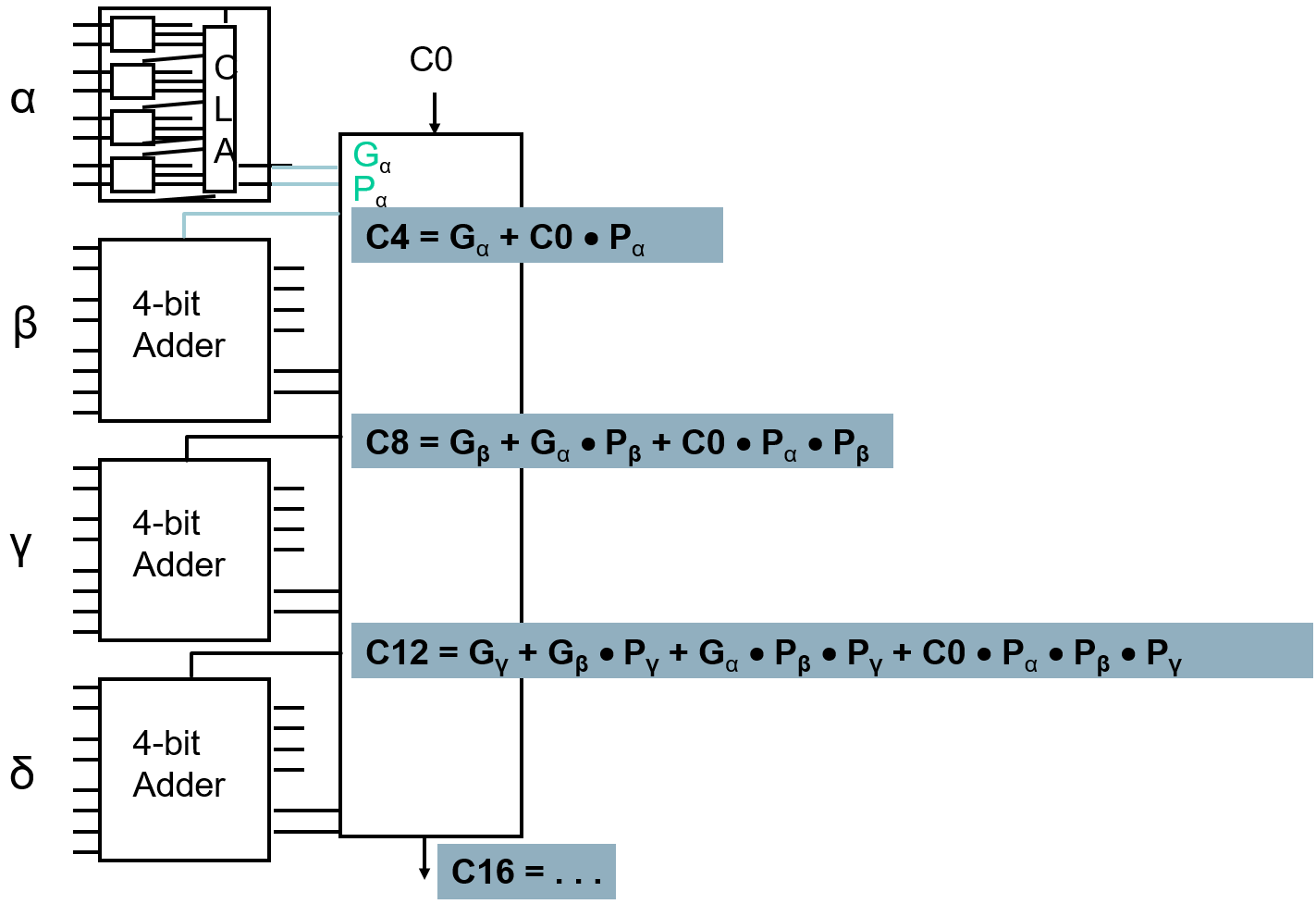
The zero output is based on the output of the ALU as a whole – if the operation selected is "3", then the zero output would NOT be the result of a-b (like in the BNE). To fix this, we could add a new NOR gate that is connected to the adder result inside the ALU – before the "2" input of the mux.

5. ***This Adder Bytes (16 points)***: In class, we discussed the design of a 4-bit CLA, like this:



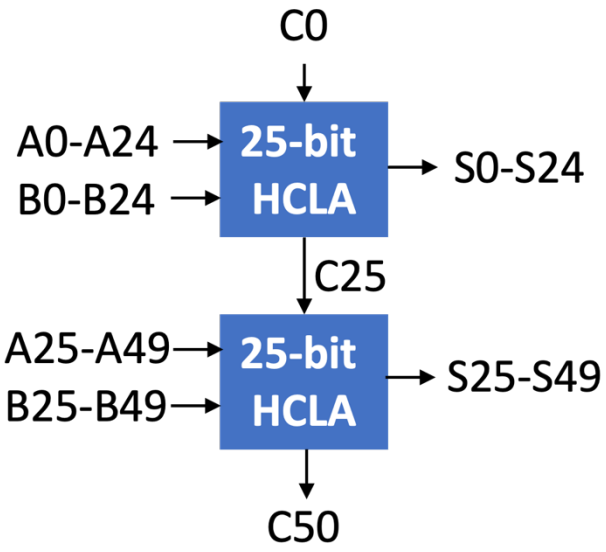
As part of this question, we are going to extend this design and make a ***5-bit CLA***.

In class, we also talked about making a 16-bit HCLA that was comprised of four 4-bit CLAs (α, β, γ, δ):



As part of this question, we are going to extend this design and make a ***25-bit HCLA*** comprised of 5-bit CLAs.

In particular, we want to make a specialized 50-bit adder for an application that requires 50 bits of precision. We are going to change the designs above to use 5-bit CLAs (comprised of five single bit adders) and 25-bit HCLAs (comprised of five 5-bit CLAs with labels alpha, beta, gamma, delta, and epsilon). The two 25-bit HCLAs are going to be connected together in ripple carry fashion (e.g. C25 from the first HCLA will directly connect to the Cin of the second HCLA):



Assume that the delay of any gate (e.g. OR, XOR, AND) is (k+1)T, where k is the fan-in of the gate, and T is some technology dependent variable. For example, a 2-input AND gate would have delay 3T. All of your answers should be in terms of T.

Determine the maximum delay of the adder: _____53T_____

Show your work by filling in the intermediate delays below:

	Delta	Delay
G0	3T	3T
P0	3T	3T
G25	3T	3T
P25	3T	3T
Galpha	10T	13T
Palpha	5T	8T
C25	13T	26T
C50	14T	40T
C45	12T	38T
C49	12T	50T
S49	3T	53T

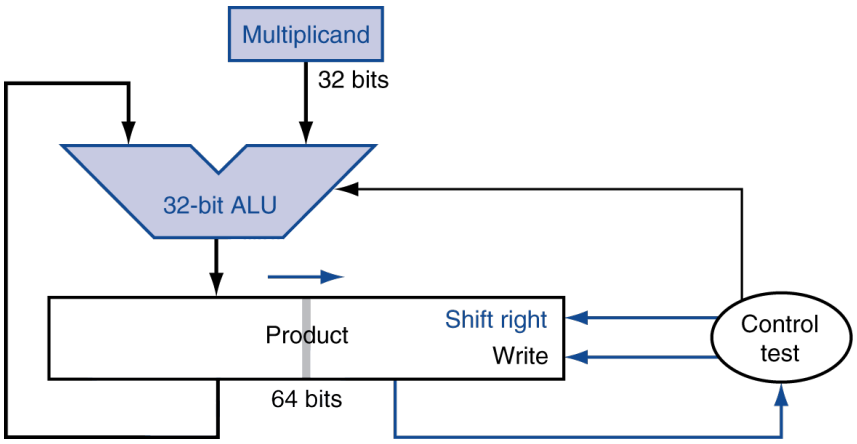
"Delta" was used to determine partial credit. "Delay" was the correct answer we were looking for.

6. ***Sea Pea Eye (10 points)***: We are designing a 4 GHz processor that is not the single-cycle datapath we covered in class. Instructions on this datapath take multiple cycles – but there is no pipelining – only one instruction may execute at a time. R-type instructions take 7 cycles. LW instructions take 9 cycles. SW instructions take 6 cycles. BEQ/BNE instructions take 5 cycles. For a particular application, 30% of instructions are LW, 20% are BEQ/BNE, 10% are SW, and the rest are R-type. What is the CPI of the application on this processor? Show your work.

CPI:_____

Weighted average: .4 x 7 + .3 x 9 + .2 x 5 + .1 x 6 = 7.1

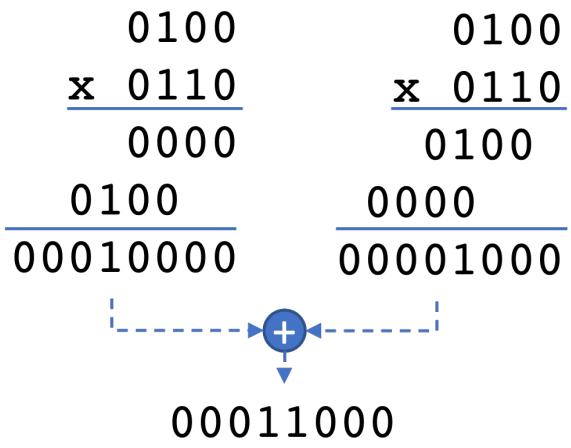
7. ***The Times They Are A-Changing (10 points)***: We covered a multiplication unit in class that looked like this:



Let's try and make a version that has more parallelism. Instead of one unit like the above, let's use two of them to make things faster. We'll leverage the associative property of addition to make this work. Consider the example 4-bit multiplication below of 4 x 6 to form an 8-bit product. This is the usual way we would work it out:

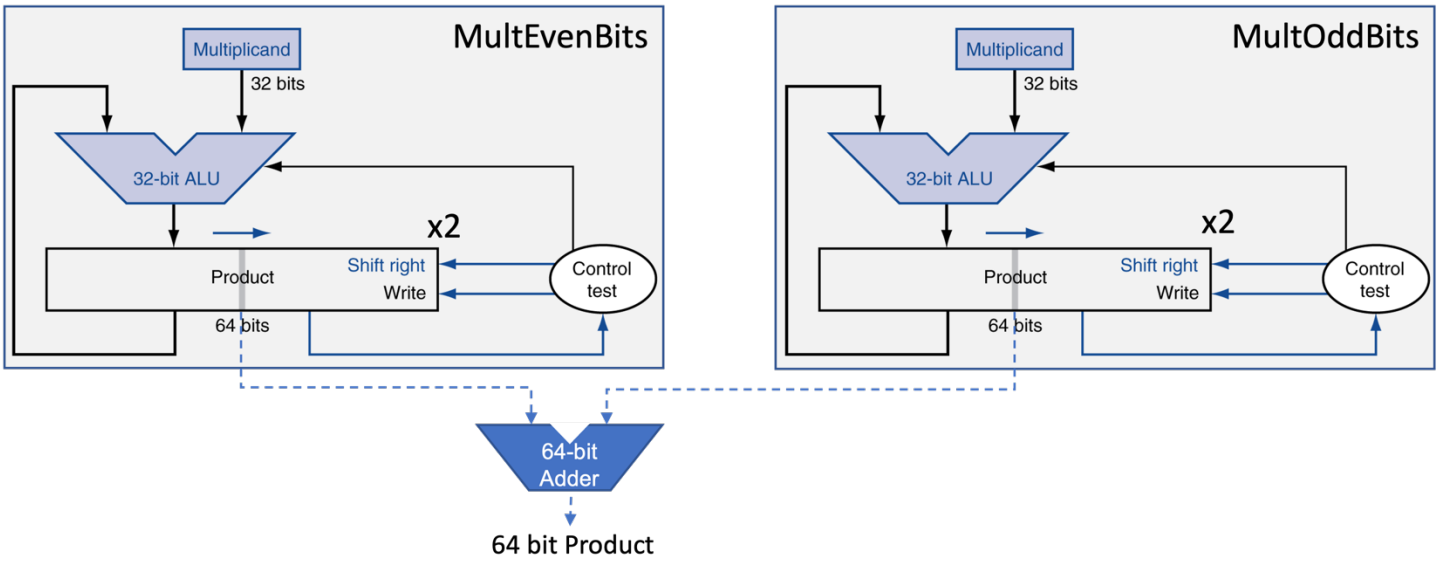
```
    0100
x   0110
    0000
   0100
  0100
 0000
00011000
```

It is the sum of four 4-bit integers – and these integers are formed by looking at each of the four bits of the multiplier (0110 in this case). But what if we did two parallel multiplications where we only looked at half of the four bits of the multiplier. Like this:



In this case, we handle the even bits of the multiplier at the left (a sum of two 4-bit integers) and the odd bits of the multiplier at the right (a sum of two 4-bit integers) – and then we add those two 8-bit partial sums together to form the final 8-bit integer.

Now let's do the same thing with the multiplication unit – we'll have two multiplication units, one to handle the odd bits of the multiplier (MultEvenBits) and one to handle the even bits of the multiplier (MultOddBits). And we will change our control unit to shift by two places instead of one (designated by the x2). We'll put the multiplier in the right half of the product register in the MultEvenBits multiplication unit, and we'll put the multiplier shifted right by one into the right half of the product register in the MultOddBits multiplication unit. Then, instead of running 32 iterations in the multiplication unit, since we are only doing half the bits (shifting by 2 and only doing either odd or even bits), we will only run for 16 iterations. Here's the design:



Given this design, answer the following questions.

a) if it takes 5 nanoseconds for each iteration of the multiplication unit, how long would it take the original multiplication unit to multiply two 32 bit integers?

5 x 32 = 160 ns

b) how long would it take our modified design (with MultOddBits and MultEvenBits) to multiply two 32 bit integers? Assume that shifting by 2 takes the same amount of time as shifting by 1. Further assume that a 64-bit adder takes 5 nanoseconds to compute a sum.

5 x 16 + 5 = 85 ns