UNIVERSITY OF CALIFORNIA, LOS ANGELES UCLA

BERKELEY • DAVIS • IRVINE • LOS ANGELES • RIVERSIDE • SAN DIEGO • SAN FRANCISCO SANTA BARBARA • SANTA CRUZ

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 – do not make us enforce the rules. Keep your eyes on your own exam!

NAME: SOLUTION

ID: _____

Do not write anything in the area below on this page:

Problem 1: _____ (16)

Problem 2: _____ (30)

Problem 3: _____ (25)

Problem 4: _____ (20)

Problem 5: _____ (9)

Total: _____ (out of 100)

1. ***I Amdahl-ighted with Tradeoffs (16 points):*** For the following design decisions, list a benefit and a drawback of the decision. Be brief, 1-2 sentences per benefit or drawback – but be specific in your answers. The first one is done for you as an example.

> **Example:**
>
> ***Design Decision***: Choosing to use 32 registers instead of 64 registers in an ISA.
> *Benefit*: Fewer bits required to specify a register address and simpler register file implementation
> *Drawback*: More register spilling may be required (i.e. more loads and possibly stores)

(+2) a. ***Design Decision***: Using variable length instructions instead of fixed length instructions.
Benefit: programs take up less memory

(+2) *Drawback*: more complex decoding in the CPU

b. ***Design Decision***: Using a stack architecture instead of a load-store machine.
(+2) *Benefit*: simpler instructions; simpler decoding

(+2) *Drawback*: more memory accesses; higher IC

c. ***Design Decision***: Using a larger immediate field for I-type instructions.
(+2) *Benefit*: larger values can be loaded
(may avoid some lui or lw)

(+2) *Drawback*: opcode or register fields shrink → fewer opcodes or fewer registers

d. ***Design Decision***: Switching to a 64-bit datapath instead of 32 bits (i.e. data registers hold 64 bits, ALUs use 64 bits, memory addresses require 64 bits to specify – but instructions are still 32 bits and the number of registers stays the same).
(+2) *Benefit*: can compute with larger numbers;
huge increase in memory space

(+2) *Drawback*:
increase in wiring, area, latency, etc.

2. ***Just Killing Time (30 points)***: The following question assumes the use of the small subset of MIPS we covered in class. We will examine the performance of a processor on a particular application, which has the following breakdown: 15% beq instructions, 25% loads, 5% stores, 5% jumps, and 50% R-types. The application executes 1 billion instructions. We will use the single cycle datapath for our implementation – and further assume the following latencies for the major components:

Component	Latency
Instruction Memory	2 ns
32-bit ALU or adder	2 ns
Register File (read)	2 ns
Data Memory (read)	3 ns

For the purposes of this problem, assume that the latency of all other components (i.e. muxes, wires) of the processor are negligible compared to these major components and can be ignored. Further assume that the time to write memory and the register file will not impact the cycle time as they will be triggered by the clock edge.

(+10) a. Find the execution time of our test application on this processor.

ET: 9 s

Worst case is "lw": I-mem, Regfile, ALU, D-mem

CT = 2 + 2 + 2 + 3 = 9 ns

CPI = 1

IC = 10^9

T = (9 ns)(1)(10^9) = 9 s

b. We are going to evaluate removing base+displacement addressing for loads and stores in our ISA. The benefit is that loads and stores will no longer need to use the ALU – so the latency of the ALU will not be on the critical path for load/store instructions. This means that no operation will use both the ALU and the memory, and assume that the datapath will be changed to reflect this. Loads and stores will still use the I-type format, but the immediate field will simply be ignored in the datapath for loads and stores. Note that other I-type instructions, like *addi* should still make use of the immediate field and the ALU. Instead, any loads or stores that would actually have a nonzero displacement (i.e. the immediate field of the instruction would not be zero), will have to make use of an *addi* instruction first. So:

lw $s0, 8 ($s1)

would become

addi $s1, $s1, 8
lw $s0, 0 ($s1)

but

lw $t0, 0 ($t1)

would not need an *addi* instruction since the immediate is already 0. 50% of loads and 40% of stores have nonzero displacements. There are some other possible drawbacks to this optimization – for now, assume that the only possible drawbacks are the ones discussed above. Considering this change, indicate how the three components of execution time will be impacted by circling one of the three choices for each component:

(+3)	CPI will:	increase	decrease	stay the same
(+3)	Cycle time will:	increase	decrease	stay the same
(+3)	Instruction count will:	increase	decrease	stay the same

c. Assume the same optimization done in part b. As mentioned in part b, there may be other drawbacks to this optimization. Consider the example cited earlier:

(+11)

lw $s0, 8 ($s1)

would become

addi $s1, $s1, 8
lw $s0, 0 ($s1)

This example assumes that the original value of *$s1* is not required after the load instruction. If the value of *$s1* was needed (since it is a base address that may be used by many load instructions), then we would need to use a new register – like this:

addi $t0, $s1, 8
lw $s0, 0 ($t0)

This can lead to an increase in register pressure – meaning that we need more loads and stores to bring values to/from the register file from/to memory. Suppose that we now see 25% more loads and 10% more stores from this register pressure, but assume that these new loads and stores always use 0 displacement, so they do not need to be preceded by *addi* instructions, and do not themselves further contribute to register pressure. Find the new execution time for this optimization.

$CT_{New} = 2 + 2 + 3 = CT_{old} - 2 = 7\,ns$

ET_{new}: ~ 8.5 s

New "addi": $(50\%)(25\%)10^9 + (40\%)(5\%)10^9 = 145 \cdot 10^6$

New "lw"/"sw": $(25\%)(25\%)10^9 + (10\%)(5\%)10^9 = 67.5 \cdot 10^6$

$IC_{New} = 10^9 + 145 \cdot 10^6 + 67.5 \cdot 10^6 = 1.2125 \cdot 10^9$

$T_{New} = (7\,ns)(1)(1.2125 \cdot 10^9) = 8.4875\,s$

3. ***Single Cycle Psychosis (25 points)***: Consider the single-cycle processor implementation. Your task will be to augment this datapath with a new instruction: the *cai* instruction (conditional assign immediate). This instruction will be an I-type instruction, and will have the following effect:

```
if (M[R[rs]] == R[rt])
        R[rt]=SE(I)
else
        R[rt] = M[R[rs]] – R[rt]
```

Implement *cai* on the **single cycle datapath**. Use the I-type instruction format.

(+5)

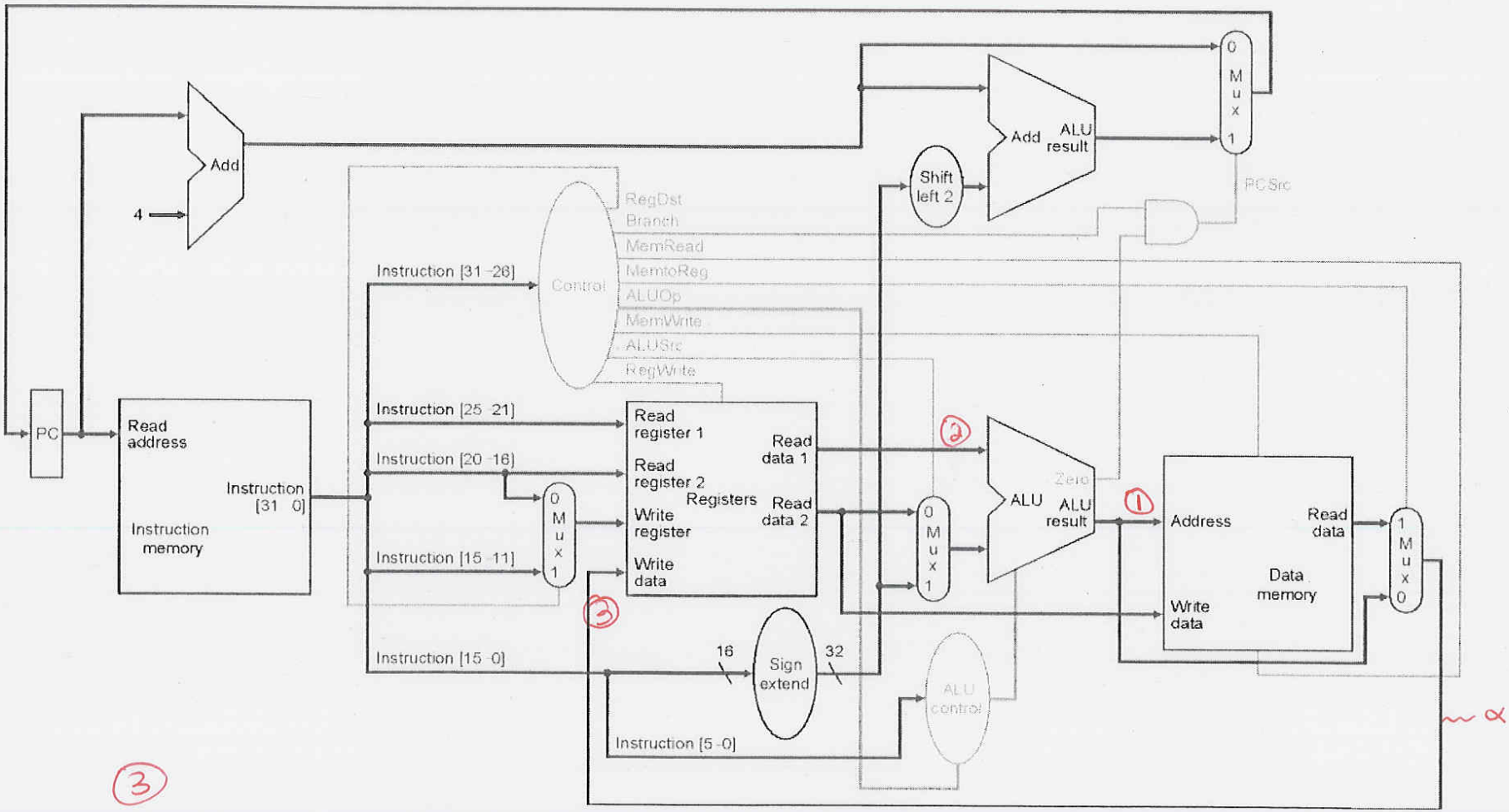
Implement your solution on the following two pages. All other instructions must still work correctly after your modifications. You should not add any new ALUs, register file ports, or ports to memory.

(+5)

① ALU output → 0, Regfile read data1 → 1, 0 → D-Mem Address, DoCAI

② Regfile read data1 → 0, D-Mem Read data → 1, 0 → Top ALU input, DoCAI

(+5)



(+5)

③ Write data Mux output (α) → 0, SignExt output → 1, 0 → Regfile Write data, Zero & DoCAI → AND → 1

(+5)

Main Controller

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

any unused opcode

subtract

ALU Controller

Opcode	**ALUOp**	**instruction**	**function**	**ALU Action**	**ALUCtrl**
Lw	00	load word	XXXXXX	add	010
Sw	00	store word	XXXXXX	add	010
Beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	AND	000
R-type	10	OR	100101	OR	001
R-type	10	SLT	101010	SLT	111

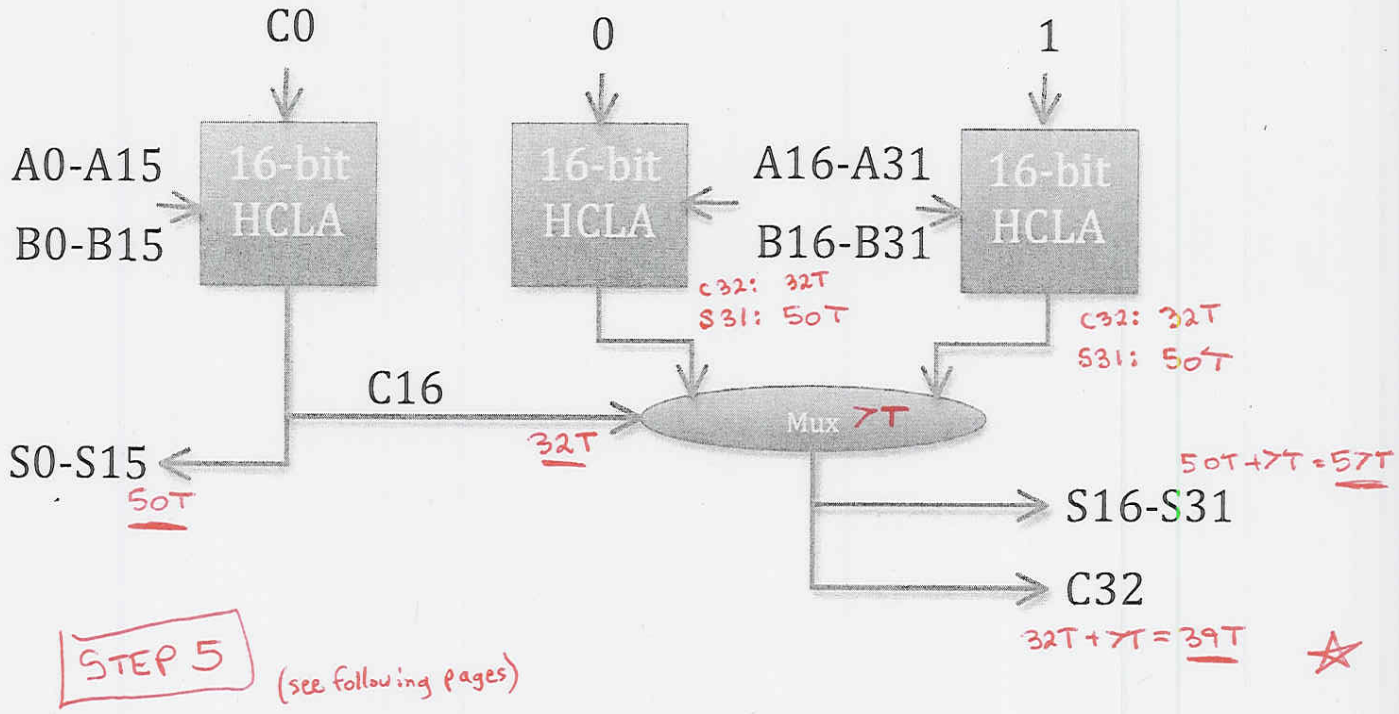
4. ***Getting Carried Away (20 points)***: Assume for the rest of this problem that logic gates (with fan-in k) have the following delays:

AND: kT
OR: $2kT$
XOR: $3kT$

So a 2-input AND gate would have delay 2T and a 4-input OR gate would have delay 8T.

For simplicity, assume that mux's have delay 7T regardless of fan-in.

We will create a 32-bit adder out of the full adders we covered in class. We will use the 4-bit CLA that we covered in class as the basic building block of this design, and we will use it (as we did in class) to make 16-bit hierarchical CLAs (HCLA). And again, just as we did in class, we will connect three of these 16-bit HCLAs together using a carry-select approach to make a 32-bit adder. The design will look as follows:



In this diagram, A0-A15 indicates the lower 16 bits of one of the input operands (A), and A16-A31 indicates the upper 16 bits of the same input operand. Note that the upper 16 bits of the two input operands are both used as input to the two 16-bit HCLAs in the right portion of the figure. S0-S31 are the sum bits, and C32 is the carry out of the 32-bit adder. Note that the output of the HCLAs is actually 17 bits – as labeled it branches off to 16 sum bits and 1 carry out bit.

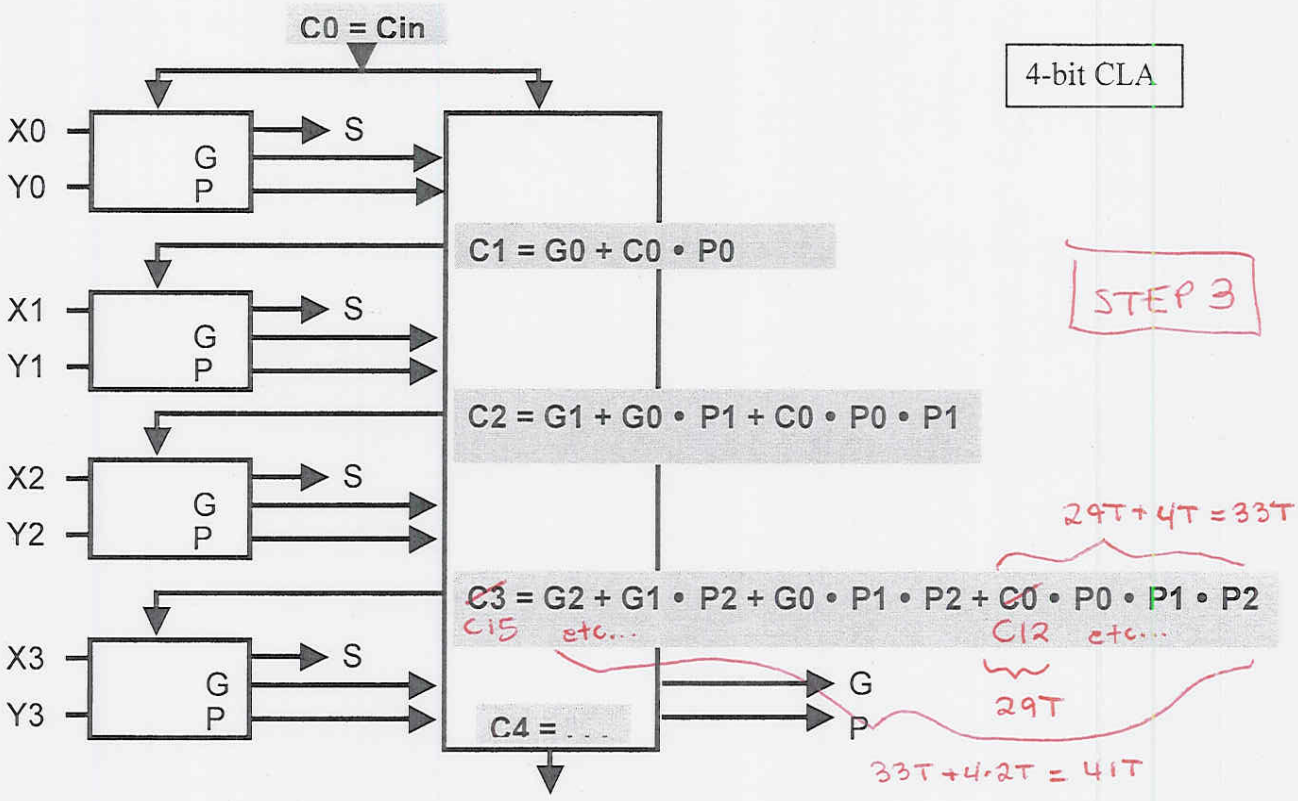
☆ In reality, a multi-bit Mux is just several 1-bit Muxes, so the C32 output is independent of the S16-S31 outputs. However, if you wrote 57T, that's OK too.

Your task is to find the maximal delay of this design – i.e. the maximal delay of bits S0-S31 and C32 will be the maximal delay of the design. Fill in the values in the table on the following page to receive full credit (and to help with possible partial credit).

Output	Delay	
G0	2T	*(1 point)*
P0	6T	*(1 point)*
G16	2T	*(1 point)*
Gα	18T	*(2 points)*
Pα	10T	*(2 points)*
C12	29T	*(2 points)*
C16	32T	*(2 points)*
S15	50T	*(2 points)*
☆ C32 (after mux)	39T	*(2 points)*
S31 (after mux)	57T	*(2 points)*

Find the maximum delay **in terms of T** of the 32-bit adder – take the maximum of all output bits – including the sum bits (S_0-S_{31}) and the final carry out (C_{32}). Show your work clearly in the table above. The two figures below are taken from the class notes, if you need to refer to them.

Maximal Delay: 57T *(3 points)*



STEP 1

$G_0 = \underbrace{X_0 \cdot Y_0}_{2T}$ G16 is the same.

$P_0 = \underbrace{X_0 \oplus Y_0}_{6T}$

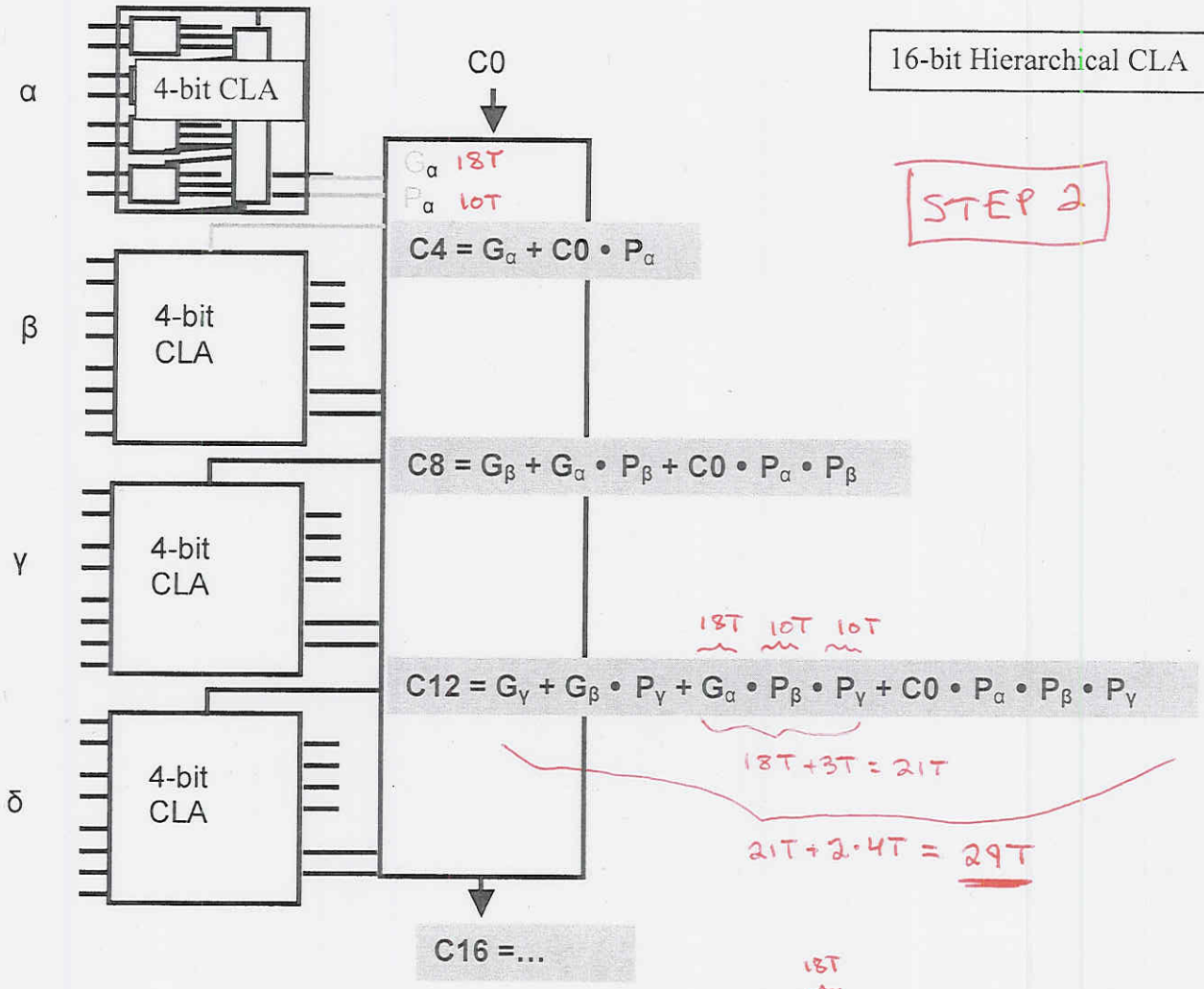
$G_\alpha = \underbrace{G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + \overbrace{G_0 \cdot P_1 \cdot P_2 \cdot P_3}^{6T+4T=10T}}_{10T + 4 \cdot 2T = 18T}$

$P_\alpha = \underbrace{P_0 \cdot P_1 \cdot P_2 \cdot P_3}_{6T+4T=10T}$

STEP 4

$S_{15} = \underbrace{X_{15} \oplus Y_{15} \oplus \overbrace{C_{15}}^{41T}}_{41T + 3 \cdot 3T = 50T}$

16-bit Hierarchical CLA



STEP 2

5. ***Silicon Salvaging Scenarios (9 points)***: This problem will present a number of scenarios where part of the single cycle processor is not functioning correctly. For each problem, determine whether there is a way to write code for the problem that will still make the processor perform correctly, briefly explain how, and briefly explain the cost in terms of the components of execution time. You may not change the hardware or ISA – just the programs that run on these. This kind of software workaround is useful in cases where there is a hardware fault or inefficiency, or even in patent infringement cases where you want to disable certain hardware paths. Consider each scenario independently (i.e. only consider that there is one fault – the faults do not accumulate). We will use the reduced set of MIPS instructions from class – so a basic set of R type functions (AND, OR, INV, ADD, SUB), the LW and SW instructions, BEQ, and J (jump). You are considering code that at some time or another will make use of all instructions in the ISA, so you must ensure that all would work properly. The first will be done for you to illustrate the problem.

   a. The ALUSrc control signal is stuck at the value zero (i.e. cannot be one).
      i Is there a software workaround (yes or no)?

      Yes

      ii If Yes, how would this be accomplished? If No, why not (be brief in either case)?

      R types and BEQ will work normally (ALUSrc is 0 anyway). J will work normally (no ALU required). LW and SW will be impacted – you cannot use base + displacement addressing for loads and stores, and so you may need additional add instructions to compute addresses – but it is worse than that. You would also need an add instruction before each LW and SW to decrement the rs register by the rt register, since there will be a forced add by the LW or SW instruction (i.e. R[rs] + R[rt]).

(+3)

   b. The Zero output of the ALU is stuck at the value 0 (i.e. cannot be one).
      i Is there a software workaround (yes or no)?

      ii If Yes, how would this be accomplished? If No, why not (be brief in either case)?

      BEQ does not work (the branch will never be taken).

      No other way to do conditional jumps.

(+3) c. The output of the shift left (in front of the ALU for PC-relative address computation) is stuck at the value 8.

i Is there a software workaround (yes or no)?

ii If Yes, how would this be accomplished? If No, why not (be brief in either case)?

Can only branch +3 instructions.

[beq rs,rt,label] → [beq rs,rt, dont_care
beq rs,rs, dont_care
<don't care>
j label]

(+3) d. The rd field from the instruction is stuck at register specifier *$t1* – this only affects R type instructions (assume the problem is with the mux that is used to select RD for the write register destination in the register file). The immediate fields for I and J type instructions are not impacted by this error, nor is there any problem writing to rt in I type instructions.

i Is there a software workaround (yes or no)?

ii If Yes, how would this be accomplished? If No, why not (be brief in either case)?

R-types are forced to put result in $t1.

We can still store to memory and load back into any register.

Example:

[add $t2, $t3, $t4] → [add $t1, $t3, $t4
sw $t1, -4($sp)
lw $t2, -4($sp)]

C.

Alternative:

[beq rs,rt, label] → [beq rs,rt, dont_care
j label2
<don't care>
label: /* taken path */
j label3
label2: /* fallthrough path */
label3:]