UNIVERSITY OF CALIFORNIA, LOS ANGELES UCLA

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

CS M151B / EE M116C

Final Exam

Before you start, make sure you have all 13 pages attached to this cover sheet.

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 final – but you cannot share books, notes, or calculators.

NAME: ____________________

ID: ____________________

Problem 1 (20 points): __________

Problem 2 (20 points): __________

Problem 3 (12 points): __________

Problem 4 (20 points): __________

Problem 5 (40 points): __________

Problem 6 (20 points): __________

Total: ______________ (out of 132 points)

ANSWERS!

1. ***Hazardous Material (20 points):*** Assume you are using the 5-stage pipelined MIPS processor, with a single-cycle branch penalty. Further assume that we always use predict not taken, and that the branch penalty is only a single cycle. We achieve this single cycle branch prediction by doing the branch comparison in the ID stage. Consider the following instruction sequence, created by a fairly bad compiler, where the loop is taken twice before the program exits:

```
        ① addi $s0, $s8, 12
Loop :  ② lw $t0, 0($s0)
        ③ addi $t7, $t0, 6
        ④ lw $t1, 4($s0)
        ⑤ addi $t6, $t1, 5
        ⑥ add $t4, $t6, $t7
        ⑦ sw $t4, 0($s0)
        ⑧ addi $s0, $s0, 1
        ⑨ bne $s0, $s1, Loop
```

a. Assuming that the pipeline is empty before the first instruction:

i. Suppose we do not have any data forwarding hardware – we stall on data hazards. The register file is still written in the first half of a cycle and read in the second half of a cycle, so there is no hazard from WB to ID. Calculate the number of cycles that this sequence of instructions would take:

44 cycles.

dependencies.

①→②
②→③
④→⑤
⑤→⑥
⑥→⑦
⑧→⑨

- each data hazard dependency causes a 2 cycle stall.
- because of 5 stage pipeline, first instruction takes 5 cycles to complete
- ⑨ mis predicts "not-taken" exactly once if the body is run twice

cycles = 4 + 17 + (2×6) + (2×5) + 1 = 44cycles.

four extra cycles for first instr.

# of instructions executed

stalls in first iteration

stalls in second iteration.

branch misprediction.

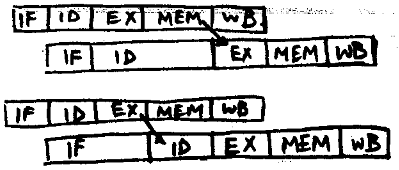
ii. How many cycles would this sequence of instructions take with data forwarding hardware: (HINT – consider how data forwarding and the changes we made for a single cycle branch penalty would interact together)

28 cycles.

- with forwarding most dependencies are resolved except for two cases:

lw → rtype

r-type → branch.



1 cycle stall

1 cycle stall.

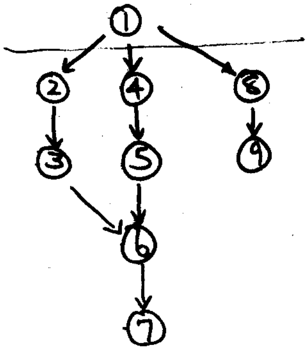
∴ remaining dependencies are.

② → ③
④ → ⑤
⑧ → ⑨

$$\text{cycles} = \underset{\text{four extra cycles for first instr.}}{4} + \underset{\text{\# instr. executed}}{17} + \underset{\text{stalls in first iteration}}{3} + \underset{\text{stalls in second iteration}}{3} + \underset{\text{branch misprediction.}}{1} = 28$$

✓ b. Reorder the code to eliminate all data hazards on a processor with data forwarding – obviously the code must still work as intended.

```
      addi  $s0, $s8, 12
Loop: lw    $t0, 0($s0)
      lw    $t1, 4($s0)
      addi  $t7, $t0, 6
      addi  $t6, $t1, 5
      add   $t4, $t6, $t7
      addi  $s0, $s0, 1
      sw    $t4, -1($s0)
      bne   $s0, $s1, Loop
```



- to eliminate data hazards we must make sure that ②→③, ④→⑤, and ⑧→⑨ are not scheduled next to each other.

2. ***Caching In on the TLB (20 points)***: Consider the data cache for a processor that ~~uses byte~~ addressed memory. The cache is a 4KB 2-way set associative cache with an 8 byte block size that uses LRU replacement within a set. Stores on our processor use write around and write through. Our cache is vir~~tually in~~dexed and physically tagged. We use an 8-way set associative 1KB TLB, and our virtual memory uses 16KB pages. We have 2^{48} B of virtual memory and 2^{30} B of physical memory. There are ***no*** extra bits required in each page table entry aside from the bits needed for the physical page number.

Calculate the hit rate of the cache and TLB on the given stream of virtual byte addresses. Each address is shown in binary and in decimal. The type of instruction that accessed the data cache is also shown – load or store. Note that there are 6 unique byte addresses here – and that the sequence of six addresses is repeated to make 12 total addresses below. Mark whether the cache and TLB has a "hit" or "miss" for each address – i.e. whether or not the desired memory address is found in the cache and whether or not the desired translation is in the TLB. For the addresses – assume that "..." means all leading 0's. Assume that the cache and TLB are completely empty (all entries invalid) at the start of the stream. Classify each cache miss as capacity, compulsory, or conflict (i.e. the type of miss). Only consider hits and misses – ignore the latency of cache or TLB misses.

	Instruction Type	Address in Binary	Address Label	Cache Hit or Miss	Cache Miss Type	TLB Hit or Miss
1	Load	...1111101100001000	64264	miss	compulsory	Miss.
2	Store	...1010001100001000	41736	miss	compulsory	miss
3	Load	...1010001100001100	41740	miss	compulsory	hit
4	Load	...1010001100000100	41732	miss	compulsory	hit
5	Store	...1011001100000010	45826	miss	compulsory	hit
6	Load	...1110001100001101	58125	miss	compulsory	hit
7	Load	...1111101100001000	64264	miss	conflict	hit
8	Load	...1010001100001000	41736	miss	conflict	hit
9	Load	...1010001100001100	41740	hit		hit
10	Load	...1010001100000100	41732	hit		hit
11	Store	...1011001100000010	45826	miss	compulsory	hit
12	Load	...1110001100001101	58125	miss	conflict	hit

TLB index ← | ← | index cache → | offset cache

4KB cache: 2 way assoc.
8 byte block
2^8 sets: 256. ← since we are using only 6 address we should have No capacity misses.

37	8	3

→ write around cache means that a miss on a STORE command does not load that cache block into the cache

1KB TLB: 8way assoc
16KB pages.
∴ 512 entries

9 TLB index

34	14

↓

16	14

2 bytes / TLB entry

→ from the table only two pages are used in this program; they have the TLB index values (...11) and (...10)

→ they cause a TLB miss the first time accessed.

cache contents (only two sets active)

01100001	~~64264~~ 1 ~~58125~~ 6 41736 8	~~41740~~ 3 ~~64264~~ 7 58125 12
0 11 00000	41732 4	

3. ***The Good, the Bad, and the Ugly (12 points)***: For the following implementation choices, list ONE benefit and ONE drawback to the given choice.

use	***instead of***	***comments***
EXAMPLE		
adds and shifts	**multiplies**	
benefit: lower CPI		
drawback: higher instruction count		
pipelined datapath	**multicycle datapath**	
benefit: lower CPI		
drawback: Complicated control for hazard resolution.		
direct mapped cache	**set associative cache**	**(with same number of entries)**
benefit: Fast, easy to implement		
drawback: possibly higher number of conflict misses.		
Booth's algorithm	**1st version of multiply algorithm**	
benefit: signed multiplication		
drawback: none! (difficult to understand?)		
virtually indexed cache	**physically indexed cache**	**(both use physical tags)**
benefit: better performance for single process		
drawback: hard to share data between processes ; complex context switch logic		
writeback	**writethrough**	**(for a first level data cache)**
benefit: faster when a memory location is written to often in short time period.		
drawback: larger miss penalty		
branch prediction	**branch delay slots**	**(single cycle branch penalty)**
benefit: good performance if good predictor chosen without adding to size of SW.		
drawback: logic may add to area power of processor		

4. ***Slot, Delay, and Unroll (20 points):*** In this problem, we will schedule code to execute on a 2-way superscalar pipelined processor. For this processor, assume that **ANY** two independent instructions can be executed in each cycle – and that full data forwarding is provided. Assume that there is a single-cycle branch penalty, and that the processor uses branch delay slots to resolve this single cycle penalty. Consider the following MIPS fragment:

```
loop:  lw $t0, 0($s1)
       lw $t1, 4($s1)
       add $t2, $t0, $t1
       addi $s1, $s1, 4
       addi $s0, $s0, 4
       sw $t2, 0($s1)
       bne $s0, $t3, loop
```

Assume that \$s0, \$s1, and \$t3 are initialized before the loop is entered, and that the loop will always be taken a number of times that is a multiple of two. Unroll the loop once (i.e. to make **two** copies of the loop body) and schedule the instructions. Be sure to fill the branch delay slots after the *bne*, placing *nop*'s if needed.

	Cycle	1st Issue Slot (for ANY instruction)	2nd Issue Slot (for ANY instruction)
loop	1	lw $t0, 0($s1)	lw $t1, 4($s1)
	2	lw $t5, 8($s1)	addi $s0, $s0, 8
	3	add $t2, $t0, $t1	addi $s1, $s1, 8
	4	add $t6, $t2, $t5	bne $s0, $t3, loop
	5	sw $t2, -4($s1)	sw $t6, 0($s1)
	6		
	7		
	8		
	9		
	10		
	11		
	12		
	13		
	14		

THIS SPACE LEFT BLANK FOR UNROLLING
(THE EXAM CONTINUES ON THE NEXT PAGE)

① unroll and rename

```
loop:  lw $t0, 0($s1)
       lw $t1, 4($s1)
       add $t2, $t0, $t1
       addi $s1, $s1, 4 }
       addi $s0, $s0, 4 }
       sw $t2, 0($s1)
       lw $t4, 0($s1)
       lw $t5, 4($s1)
       add $t6, $t4, $t5
       addi $s1, $s1, 4 }
       addi $s0, $s0, 4 }
       sw $t6, 0($s1)
       bne $s0, $t3, loop
```

② re index

```
loop:  lw $t0, 0($s1)
       lw $t1, 4($s1)
       add $t2, $t0, $t1
       addi $s1, $s1, 8
       addi $s0, $s0, 8
       sw $t2, -4($s1)     }  ←  $t4 = t2.
       ~~lw $t4, -4($s1)~~ }
       lw $t5, 0($s1)
       add $t6, $t4, $t5
       sw $t6, 0($s1)
       bne $s0, $t3, loop
```

③ re schedule (lw at frount
sw at end)

```
loop:  lw $t0, 0($s1)
       lw $t1, 4($s1)
       lw $t5, 8($s1)
       add $t2, $t0, $t1
       add $t6, $t2, $t5
       addi $s1, $s1, 8
       addi $s0, $s0, 8
       sw $t2, -4($s1)
       sw $t6, 0($s1)
       bne $s0, $t3, loop
```

→ • now place into slots.

• careful:
lw → add
dependency still causes a stall!

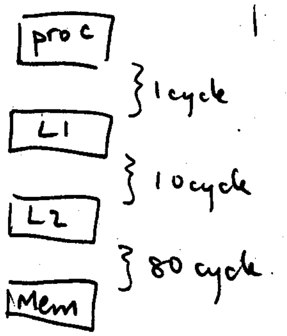
5. ***Why, oh why, must we do TCPI? (40 points)***: We are going to assess branch and cache performance on the pipelined datapath from class – we have full data forwarding. Our peak CPI is 1.0. Assume that 30% of instructions are branches, and that we have a single cycle branch hazard on this processor. Our branch predictor **always** guesses *not taken*. 50% of branches are not taken. Our processor has an instruction cache and data cache – both take a single cycle to access. The instruction cache miss rate is 10% and the data cache miss rate is 30%. The next level of the memory hierarchy is an L2 cache with a miss rate of 20% and an access time of 10 cycles, this is in addition to the L1 cache latency. Main memory has an access time of 80 cycles, this is in addition to the latency of the L1 and L2 caches. 20% of instructions are loads, and stores do not stall the processor on a cache miss. 3/5ths of loads have dependent instructions following them. Our target application executes 1,000,000 instructions. The processor clock runs at 2 GHz.

   a. Calculate the average memory access time (AMAT) of the ~~processor:~~ DL1 $

      AMAT: 8.8 cycles

$$\text{AMAT} = 1 + \overset{\text{L1 D\$ miss rate}}{0.3} \times \left(\text{L1 miss penalty}\right) = 8.8 \text{ cycles}$$

$$\text{L1 miss penalty}: 10 + \underset{\text{L2 \$ miss rate}}{\underline{0.2}}(80) = 26$$



   b. Calculate TCPI for our target application on our processor.

      TCPI: 5.43 Cycles.

$$\text{BCPI} = 1 + \underbrace{0.3 \times 0.5}_{\text{branch flush}} + \underbrace{0.6 \times 0.2}_{\text{lw stall}} = 1 + 0.15 + 0.12 = 1.27$$

$$\text{MCPI} = \underbrace{1\,(\overset{\text{I\$ miss.}}{0.1}) \times 26}_{\text{instr cache}} + \underbrace{\overset{\text{load instr.}}{0.2} \times \overset{\text{D\$ miss}}{0.3} \times 26}_{\text{data cache}} = 4.16$$

$$\text{TCPI} = 1.27 + 4.16 = 5.43$$

c. Suppose 1/6th of all branches are procedure calls. Each procedure call (i.e. a jal instruction) in our application also has a return (i.e. a jr instruction). These will all be mispredicted because we always guess not taken. One approach to reducing branch hazards in such a case is to *in-line* the procedure call. The compiler basically takes the instructions in the body of the procedure call and replaces **all** calls to that procedure with these instructions. This means that instead of the code:

```
        add $s0, $s0, $t1
        jal Target
        add $s0, $s0, $t2
        jal Target
        .....
        .....
        .....
        .....
Target: lw $t3, 0 ($s0)
        addi $t3, $t3, 200
        sw $t3, 0 ($s0)
        jr $ra
```

We would have the code:

```
add $s0, $s0, $t1
lw $t3, 0 ($s0)
addi $t3, $t3, 200
sw $t3, 0 ($s0)
add $s0, $s0, $t2
lw $t3, 0 ($s0)
addi $t3, $t3, 200
sw $t3, 0 ($s0)
.....
.....
.....
.....
```

The benefit in this simple example is that we avoid four branches (two jal's and two jr's), but the size of the instruction text segment in memory (i.e. the size of the actual program we are running) has increased. Now, instead of the lw, addi, and sw being in one place in the text segment, they are in two places. This can increase the miss rate of the instruction cache.

Suppose that we try in-lining on our processor. In order for performance to improve, the cost of increasing the instruction cache miss rate must not exceed the benefit of reducing branch hazards. Using TCPI as the CPI in the equation for Execution Time, provide an $\boxed{\text{upper bound}}$ on the miss rate of the instruction cache to improve performance when using in-lining. Assume that the L2 cache's miss rate does not change.

The instruction cache miss rate must be <= 12%

$ET_{old} = TCPI_{old} \times 10^6 = 5.43 \times 10^6$ cycles.

Number of instructions

	old	new	new %
branch	300 000	200 000	$22.\bar{2}$
load /store	200 000	200 000	$22.\bar{2}$
r-type	500 000	500 000	$55.\bar{5}$
total	1000000	900 000	100%

Since $\frac{1}{6}$ of branches are jal

then $\frac{1}{6}$ of branches are jr

$\therefore$ $\frac{2}{6}$ of branches are removed when inlining

new # branch = old # branch $\times \frac{4}{6}$ = 200 000

In old program 50% branches taken (150,000 instr.)

Since jal and jr are always taken, after in-lining:

150 000 - 100 000 = 50 000 branches are taken

200 000 - 50 000 = 150 000 branches are not taken

If we predict branch not taken, the branch miss rate $= \frac{50\,000}{200\,000} = 25\%$

$$BCPI_{new} = 1 + \underbrace{0.\bar{2} \times 0.25}_{\text{branch flush}} + \underbrace{0.6 \times 0.\bar{2}}_{\text{lw stall}} = 1.1\bar{8}$$

$$MCPI_{new} = 1 \cdot (\text{I\$ miss rate}) \times 26 + 0.\bar{2} \times 0.3 \times 26 = (\text{I\$ miss rate}) \times 26 + 1.7\bar{3}$$

$$TCPI_{new} = BCPI_{new} + MCPI_{new} = 1.1\bar{8} + (\text{I\$ miss rate}) \times 26 + 1.7\bar{3} = 2.9\bar{2} + (\text{I\$ miss rate}) \times 26$$

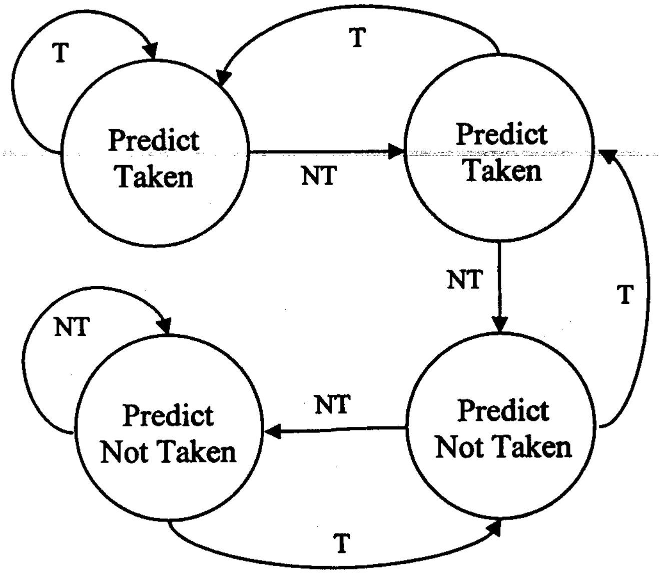
$$ET_{new} = TCPI_{new} \times (9 \times 10^5 \text{ instr})$$

$$= 2.63 \times 10^6 + (\text{I\$ miss rate}) \times 2.34 \times 10^7 \leq (ET_{old} = 5.43 \times 10^6)$$

$$\therefore \text{I\$ miss rate} \leq 0.1197$$

d. Rather than do in-lining, we will try using a branch predictor on our architecture. Now instead of always predicting not taken, we will use a variation on the 2-bit predictor that works as follows:

Two possible answers depending on assumptions.



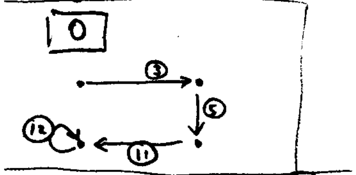
Our predictor has a 1024 entries, and each entry has a 2-bit counter implementing the above diagram. Each node represents one of the four states of the 2-bit counter. Each node is labeled with the prediction that will be made when the counter is in that state. Each edge is labeled with an NT or a T. NT means not taken, T means taken. When the predictor is in a given state, the edges represent the next state in response to the actual direction of the branch – i.e. if the branch is not taken, but we are in the predict taken state in the upper left corner of the figure, we will transition to the state in the upper right of the figure.

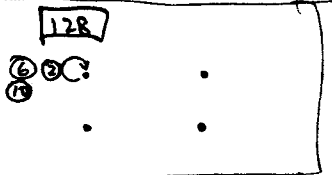
Assume that bits [14...5] of PC used as index to predictor table

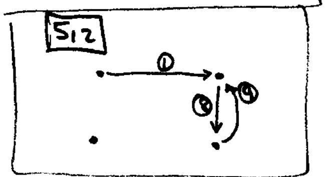
Given the following stream of branch PCs, fill in the following table. Assume that each 2-bit counter starts in the upper left state. The first one has been done for you.

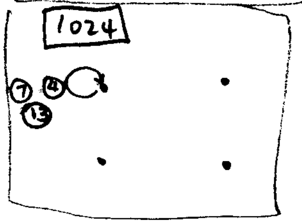
• Assume that bits [9...0] of PC used as index to predictor.

• this means that addresses 0 and 1024 share an entry

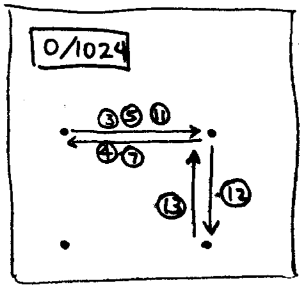




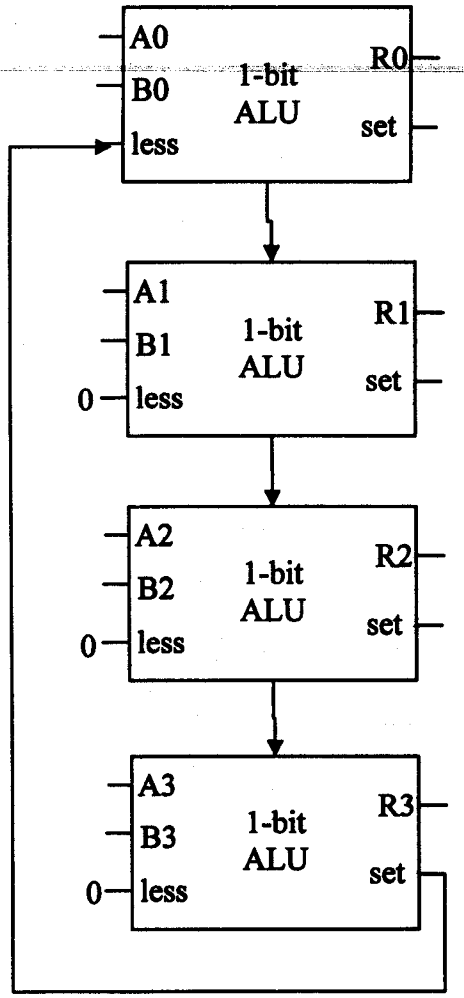




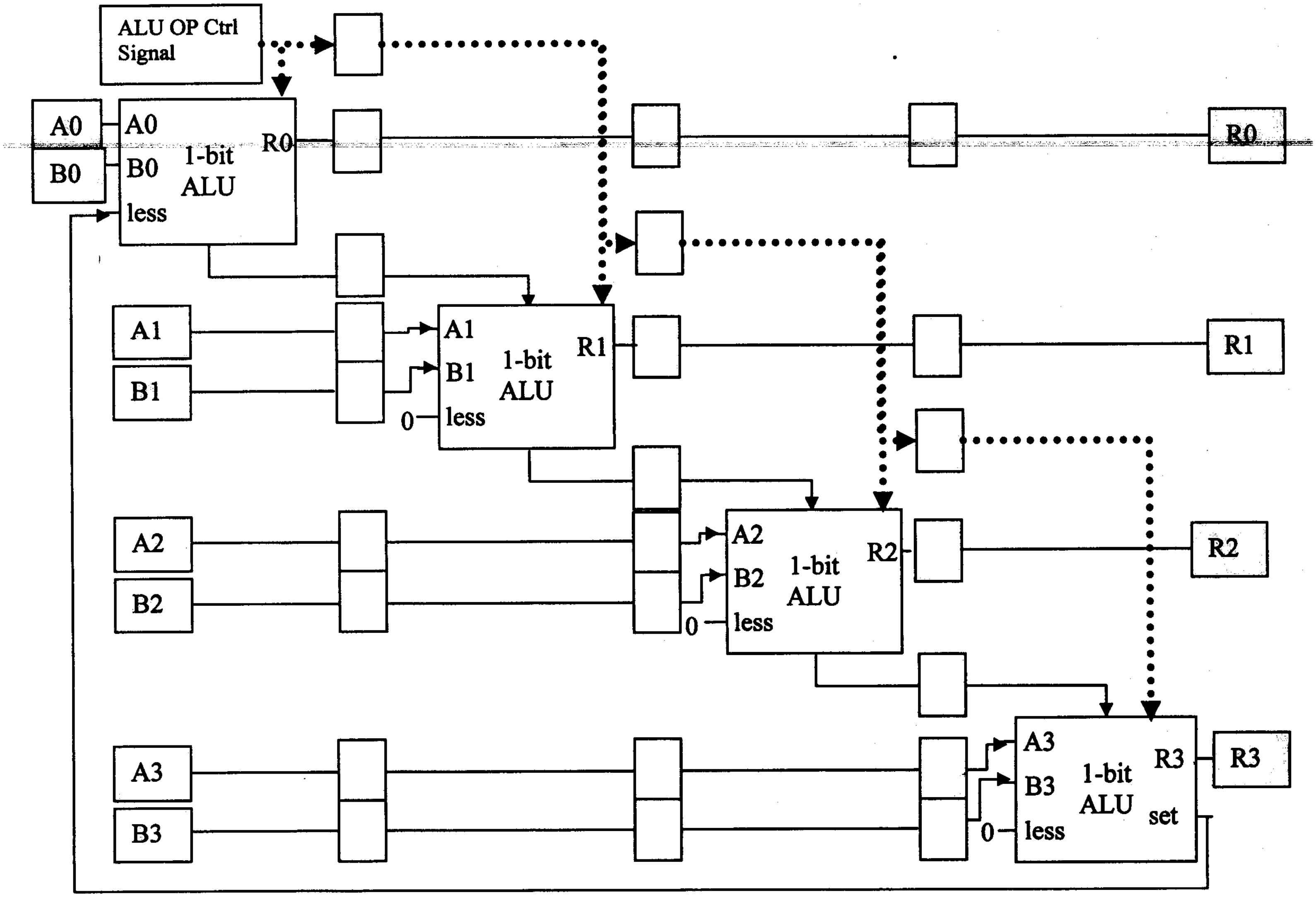
	PC	Actual Branch Direction	Predicted Correctly?	
①	512	Not Taken	No	
②	128	Taken	Yes	
③	0	Not Taken	No.	
④	1024	Taken	Yes	
⑤	0	Not Taken	No	
⑥	128	Taken	Yes	
⑦	1024	Taken	Yes	
⑧	512	Not Taken	No	
⑨	512	Taken	No	
⑩	128	Taken	Yes	
⑪	0	Not Taken	Yes	No
⑫	0	Not Taken	Yes	No
⑬	1024	Taken	Yes.	No



6. ***Got SLT ? (20 points)***: Consider the following 4-bit ALU that implements the SLT operation. Assume that the control lines like binvert and the ALU operation selection control are all implemented, but are not shown to make the drawing simpler.

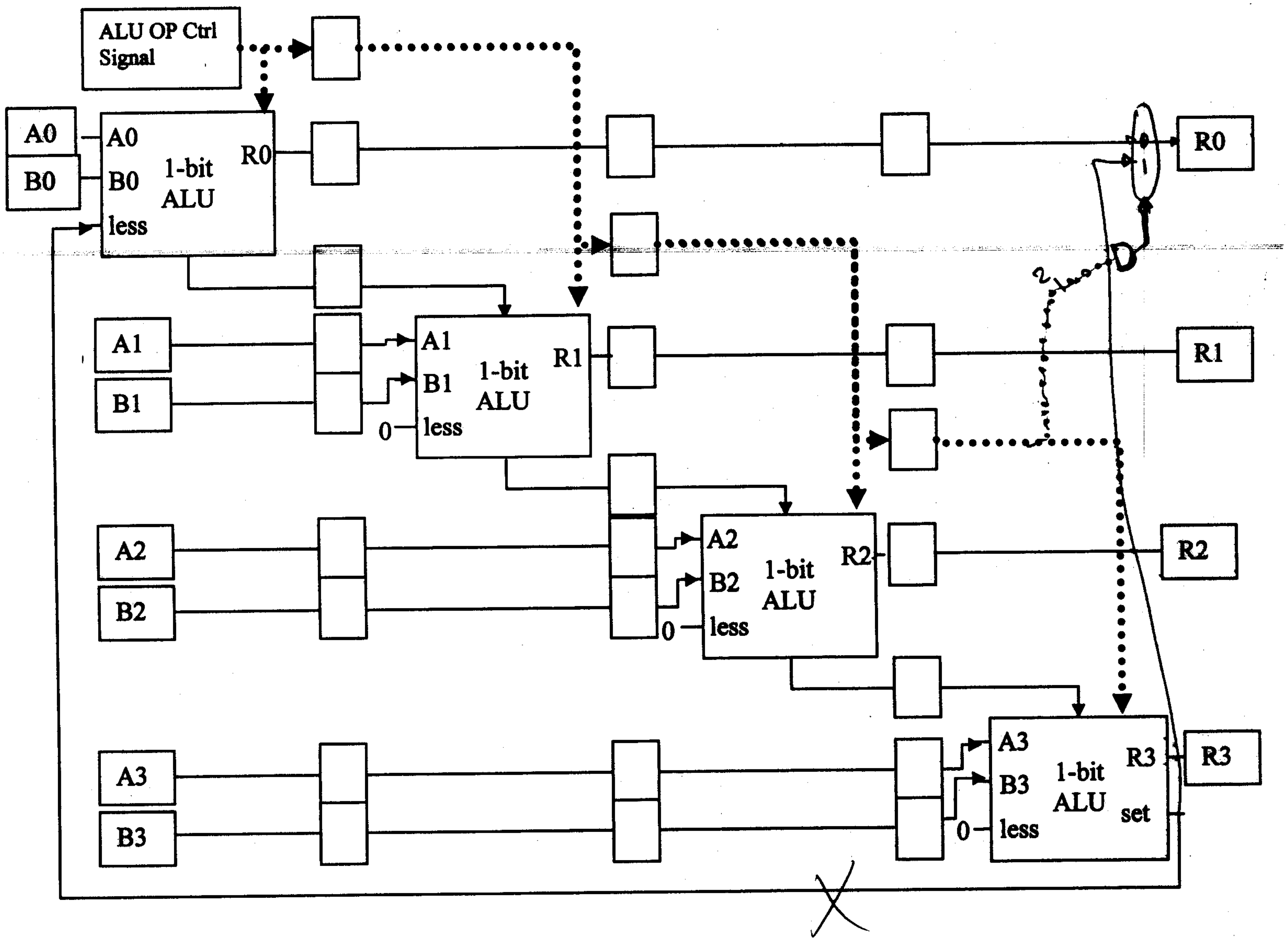


We want to make a pipelined version of this ALU – separating each stage into a different cycle. We will use registers to latch values between pipeline stages. Your friend proposes the following pipelined version of the ALU (where the shaded boxes are registers that latch a value after every clock cycle):



Assume that the input values (A0-A3, B0-B3) are all initially latched in registers, and that there are registers to latch the final output values (labeled R0-R3). Further assume that the control signals (the ALU operation and binvert) are also latched in registers and are correctly propagated to the 1-bit ALUs.

You suspect something is wrong with your friend's implementation. Fix it on the next page – you may add registers, wires, logic gates, and/or multiplexors, but you may not add any more 1-bit ALUs. If you remove registers or wires, clearly put an X on the register or wire. Currently, the ALU supports the following operations: add (00), sub (01), and (10), and slt (11). The number in ()'s is the ALU OP Ctrl signal for that operation. **NOTE**: The ALU must still be able to perform all 4 operations correctly after your changes.



Use this space to **briefly** explain what is wrong with the design and what your solution is:

The "set" output of the MSB ALU cannot be feedback to LSB ALU since then you will mix data from different pipeline stages. The solution is to break the feedback path and forward the answer directly to the last registers.