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

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.

We strictly follow UCLA's policy on academic dishonesty – keep your eyes on your own paper.

NAME: Solutions

ID: ______________________________

Problem 1 (15 points): __________

Problem 2 (15 points): __________

Problem 3 (20 points): __________

Problem 4 (10 points): __________

Problem 5 (20 points): __________

Problem 6 (20 points): __________

Total: ______________ (out of 100 points)

Problem 7 (OPTIONAL): __________

1. ***Drawing a Blank? (15 points):*** Given the following statements, fill in the blanks with the terms below the statement. You may reuse terms in case the statement has multiple blanks. Rather than writing the entire term, just use the ***letter*** preceding the term to fill in the blank.

   a. Ideally, the CPI for a single-issue (i.e. scalar) 5-stage pipeline would approach ___A___.

      A. 1.0
      B. 0.5
      C. 5.0
      D. 0.25
      E. 2.5

   b. A 5-stage pipeline that is initially empty would reach steady state in the ___E___ cycle, assuming it does not encounter any hazards.

      A. 1st
      B. 2nd
      C. 3rd
      D. 4th
      E. 5th

   c. For a particular application, you try two different processors – Alpha and Beta. These processors Alpha and Beta are identical except that Alpha's L1 data cache has double the block size of Beta's L1 data cache, but half the associativity. Their total size is the same though. You would think that Alpha would do better on applications with lots of ___B___ and Beta would do better on applications with lots of ___E___.

      A. Temporal locality
      B. Spatial locality
      C. Cold misses
      D. Capacity misses
      E. Conflict misses
      F. Instruction-level parallelism

   d. If your application mainly has stores that write an address once and have poor spatial and temporal locality, then you should use a ___B___ cache.

      A. Write through, write allocate
      B. Write through, write around
      C. Writeback, write allocate
      D. Writeback, write around

e. Virtual memory provides A, B. *Choose all answers that apply.*

- ***A.*** Protection between applications
- ***B.*** A larger memory space
- ***C.*** Lower load latency
- ***D.*** Fewer register spills

f. Your 5-stage pipelined architecture (with a 3 cycle branch penalty) currently uses hardware-based stalling to deal with control hazards. If you were to use dynamic branch prediction (i.e. a table of 2-bit predictors) you would expect that CPI would likely B, cycle time could possibly A or C, and the instruction count would C.

- ***A.*** increase
- ***B.*** decrease
- ***C.*** stay the same

g. Your 5-stage pipelined architecture currently resolves data hazards with hardware-based stalling – no forwarding logic. If you were to use NOP insertion instead, you would expect that CPI would likely B, cycle time could possibly B or C, and instruction count would likely A.

- ***A.*** increase
- ***B.*** decrease
- ***C.*** stay the same

h. Simultaneous multithreading (also known as hyperthreading) may help improve B but can reduce C.

- ***A.*** ALU latency
- ***B.*** Processor utilization (i.e. throughput)
- ***C.*** Single thread performance (when run with other threads)
- ***D.*** L1 cache latency

i. Direct Memory Access (DMA) means C.

- ***A.*** that the processor can directly access memory instead of going through the TLB or page table
- ***B.*** that the processor makes a translation from virtual to physical address, and then directly accesses memory without going through the cache hierarchy
- ***C.*** that I/O devices can directly read/write memory without processor intervention
- ***D.*** that cache can directly access DRAM without address translation

2. ***Hazard Countin' (15 points):*** Assume you are using the 5-stage pipelined MIPS processor, with a two-cycle branch penalty. In the following code:

```
loop:   lw $t0, 0($s0)
        beq $t0, $s1, here     50 times taken
        lw $t2, 8($s0)
        addi $t0, $t0, 4       2
        add $t0, $t2, $t0      2
        add $s0, $s0, $t0      2
here:   lw $t0, 4($s0)
        add $t1, $t0, $t1
        addi $s0, $s0, 16
        bne $s0, $s2, loop     100 times taken
        sw $t1, 0($s3)
```

We will measure the number of cycles to execute this code starting with the first *lw* instruction entering the pipeline (assume it is not stalled by any hazards) and continuing until the *sw* at the bottom leaves the pipeline. The *bne* at the bottom is taken 100 times. The *beq* is taken 50 times. Assume that all memory is ideal – i.e. we have no cache misses to worry about. Further assume that these 11 instructions are stored contiguously in memory.

a. 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. We handle branch hazards with a branch predictor that always guesses branch not taken. But this is ***only for conditional branches*** – jumps are handled differently and do not cause branch hazards. Calculate the number of cycles that this sequence of instructions would take, and show your work.

Total cycles:____________________

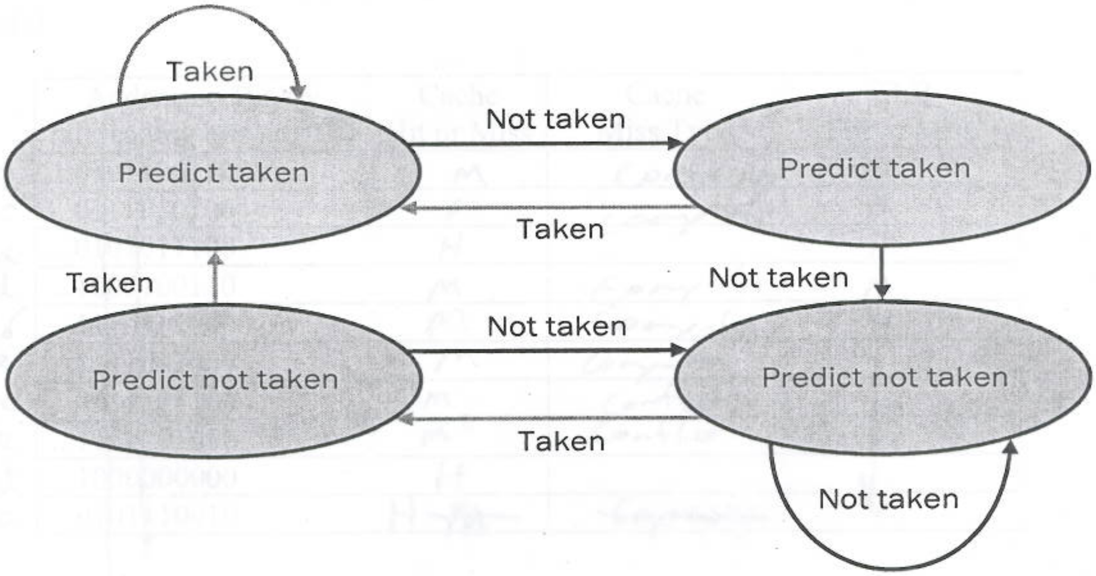
4 + 6×101 + 4×50 + 1 + 2×(1+1+1)×101 + (2+2+2)×50 + 2×(100+50)

4 + 606 + 204 + 1 + 606 + 306 + 300

~~2067~~

2027

b. Now assume we have full data forwarding as covered in class, and we have an 8 entry, 2-bit branch predictor with FSM as shown below. Assume that we use the simplified branch indexing formula: index = (PC>>2) % predictor_size. The symbol % represents the modulo operator. Further assume that all entries in the branch predictor start in the state at the upper left of the FSM. But this predictor is only used for conditional branches – jumps are handled differently and do not cause branch hazards. Calculate the number of cycles that this sequence of instructions would take, and show your work.



Total cycles:____________________

$4 + 6\times101 + 4\times51 + 1 + (1+1)\times101 + 2\times(1+51)$

$4+606 + 204 + 1 + 202 + 104$

~~1117~~

1121

$4 + 6\times101 + 4\times51 + 1 + (1+1)\times101 + 2\times(1+51)$

4	6×101	4×51	1	(1+1)×101	2×(1+51)
pipe fill	instr always exec	instr after 1st beq up to here	sw	data stalls from always exec path	branch stalls on bne and beq

3. ***Cache Me If You Can (20 points):*** Find the cache hit or miss stats for a given set of addresses. The data cache we will evaluate is a 256B, direct mapped cache with 64-byte blocks. You are using a 48-bit virtual address space, a 32-bit physical address space, and use 512B pages in your virtual memory. Your TLB has 32 entries (i.e. locations to store translations) and is 4-way associative. Each entry in the page table has one extra bit beyond the physical page number.

   a. Find the hit/miss behavior of the data cache and TLB for the given byte address stream, and label cache misses as compulsory, capacity, or conflict misses. All blocks in the cache and TLB are initially invalid.

	Address in Binary (all leading bits are 0's)	Cache Hit or Miss	Cache Miss Type	TLB Hit or Miss
a	...0101100110	M	compulso	M
c	...0111010100	M	compulso	H
a	...0101011100	H		H
d	...1000100110	M	Compulso	M
b	...1110111000	M	Compulso	H
e	...1111011000	M	compulso	↓
c	...0111101100	M	Conflict	
e	...1111101010	M	Conflict	
d	...1000000000	H		
a	...0101110010	H ~~M~~	~~Capacity~~	

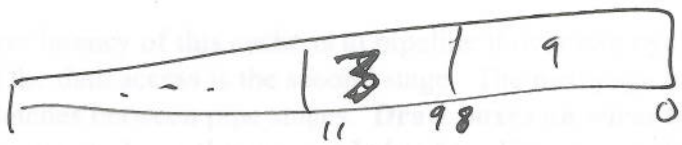
   b. What percent of the pages in virtual memory can fit in physical memory at one time? $\frac{2^{23}}{2^{39}} = \frac{100}{2^{16}}\%$

   c. How large (in bytes) is the page table? $2^{37} * 3B$

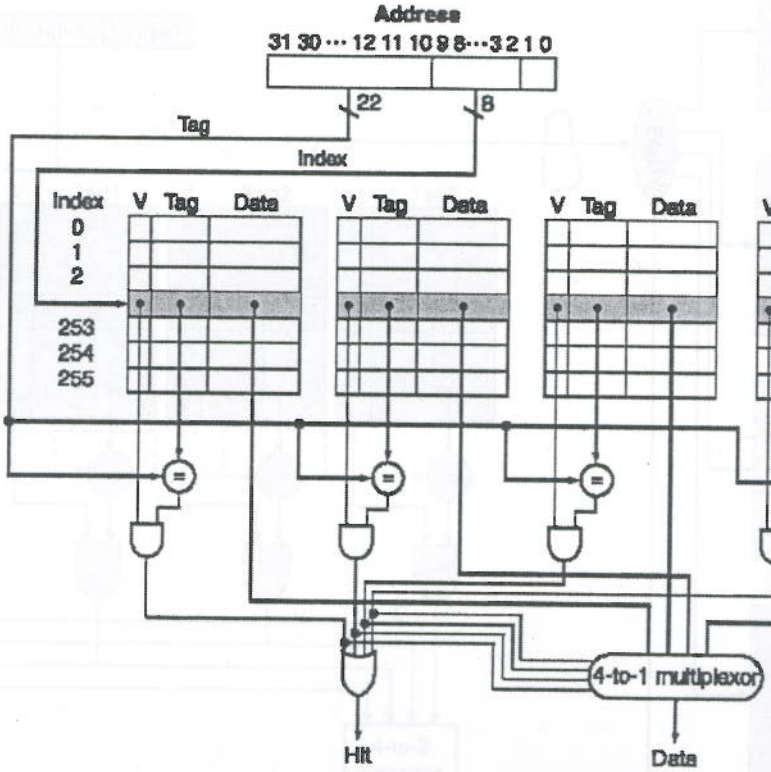
   d. What fraction of the total number of page translations can fit in the TLB? $\frac{2^{5}}{2^{39}} = \frac{1}{2^{34}}$

   e. What bits of a virtual address will be used for the index to the TLB? Specify this as a range of bits – i.e. bits 4 to 28 will be used as the index. The least significant bit is labeled 0 and the most significant bit is labeled 31.

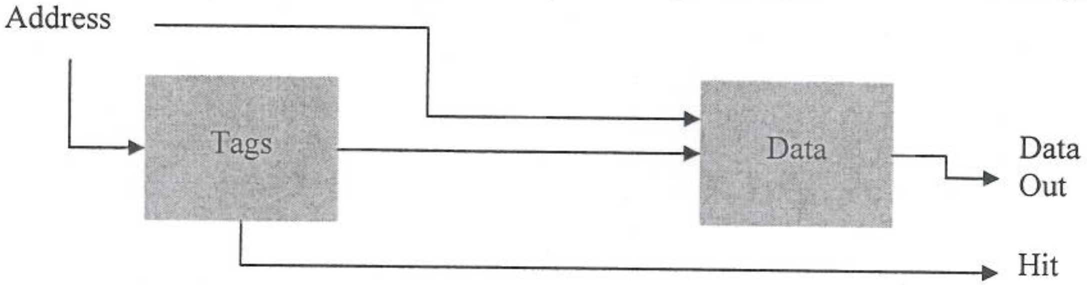
      11–9



4. ***Serial (Access) Killer (10 points)***: Consider the 4-way set associative cache we explored in class:

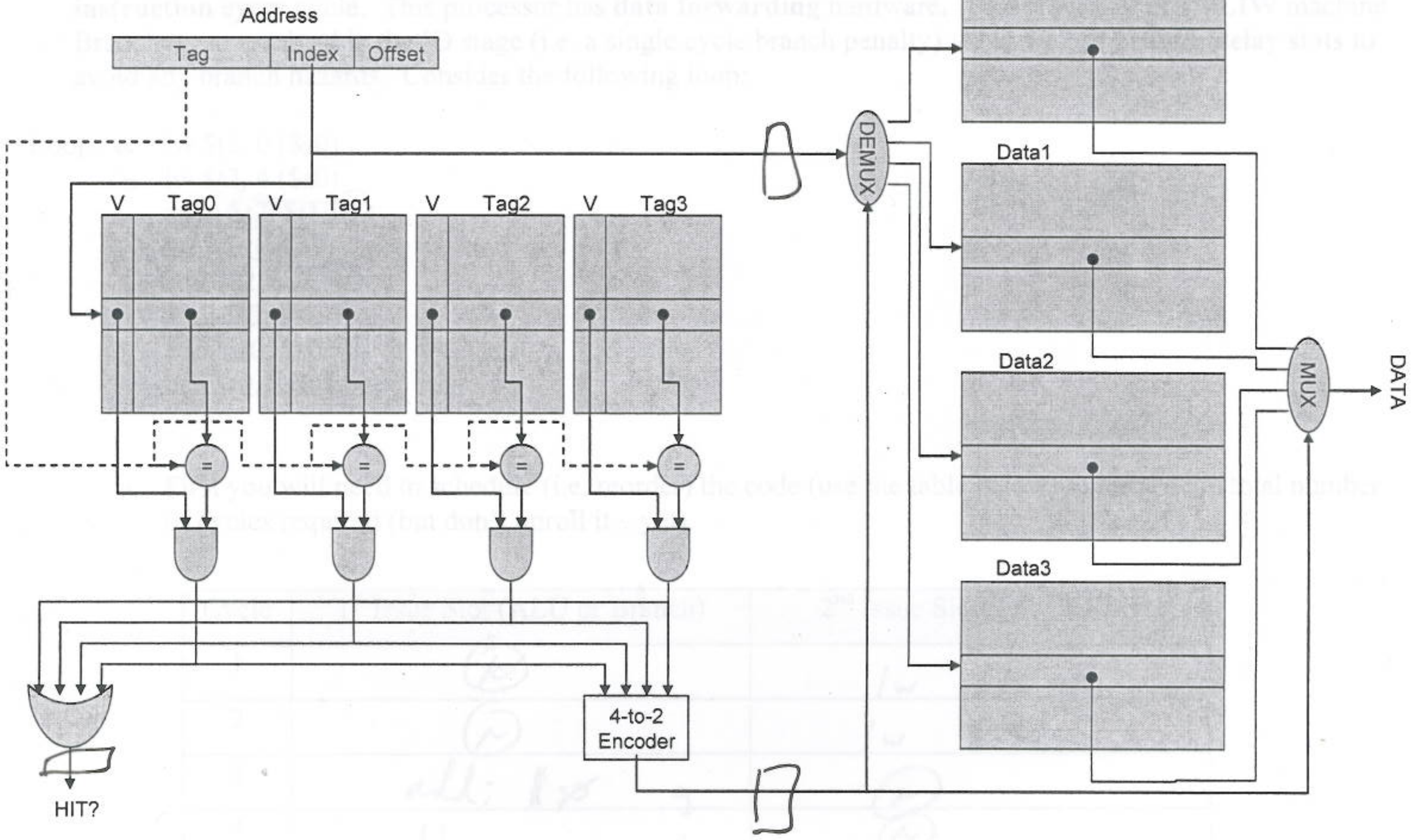


This is known as a ***parallel access*** cache because the Tag and Data components are accessed in parallel. Another way to design a cache is to put the components in series – a ***serial access*** cache. This type of cache can be more power efficient, but has longer latency. At a high level, a serial access cache just looks like this:



The address is used to index the tag and data components of the cache, but only one way of the data component is actually driven out (instead of all ways in the parallel access cache). The two outputs are still the same (Hit and the Data Out).

One way to deal with the longer latency of this cache is to pipeline it over two cycles. The tag access is the first stage of the pipeline, and the data access is the second stage. The next page has the serial access design in more detail, but it is missing latches between pipe stages. **Draw boxes on wires to indicate where values should be saved in between cycles to have the correct behavior.** Keep in mind that we want to have two accesses to the cache in the pipeline at the same time (don't worry about hazards). For timing purposes, we will keep the 4-to-2 encoder, AND, and OR gates in the first stage of the pipeline, and the demultiplexor (DEMUX) in the second stage of the pipeline. Don't worry about the address offset.


Address
Tag
Index
Offset
V
Tag0
V
Tag1
V
Tag2
V
Tag3
=
=
=
=
4-to-2
Encoder
HIT?
DEMUX
Data0
Data1
Data2
Data3
MUX
DATA

5. ***Starting Some Static (Scheduling) (20 points):*** Consider the 2-way superscalar processor we covered in class – a five stage pipeline where we can issue **one ALU or branch instruction** along with **one load or store instruction** every cycle. This processor has **data forwarding** hardware. This processor is a **VLIW** machine. Branches are resolved in the ID stage (i.e. a single cycle branch penalty) – and we use **branch delay slots** to avoid any branch hazards. Consider the following loop:

```
Loop: a  lw $t1, 0 ($s0)
      b  lw $t2, 4 ($s0)
      c  addu $t2, $t1, $t2
      d  lw $t3, 0 ($t2)
      e  add $s2, $s2, $t3
      f  addi $t0, $t0, 1
      g  addi $s0, $s0, -8
      h  bne $t0, $s1, Loop
```

a. First you will need to schedule (i.e. reorder) the code (use the table below) to reduce the total number of cycles required (but don't unroll it…yet).

Cycle	1st Issue Slot (ALU or Branch)	2nd Issue Slot (LW or SW)
1	ⓝ	lw $t1 a
2	ⓝ	lw $t2 b
3	addi $s0 g	ⓝ
4	addu c	ⓝ
5	addi $t0 f	lw $t3 d
6	bne	ⓝ
7	add e	ⓝ
8		
9		
10		
11		
12		
13		

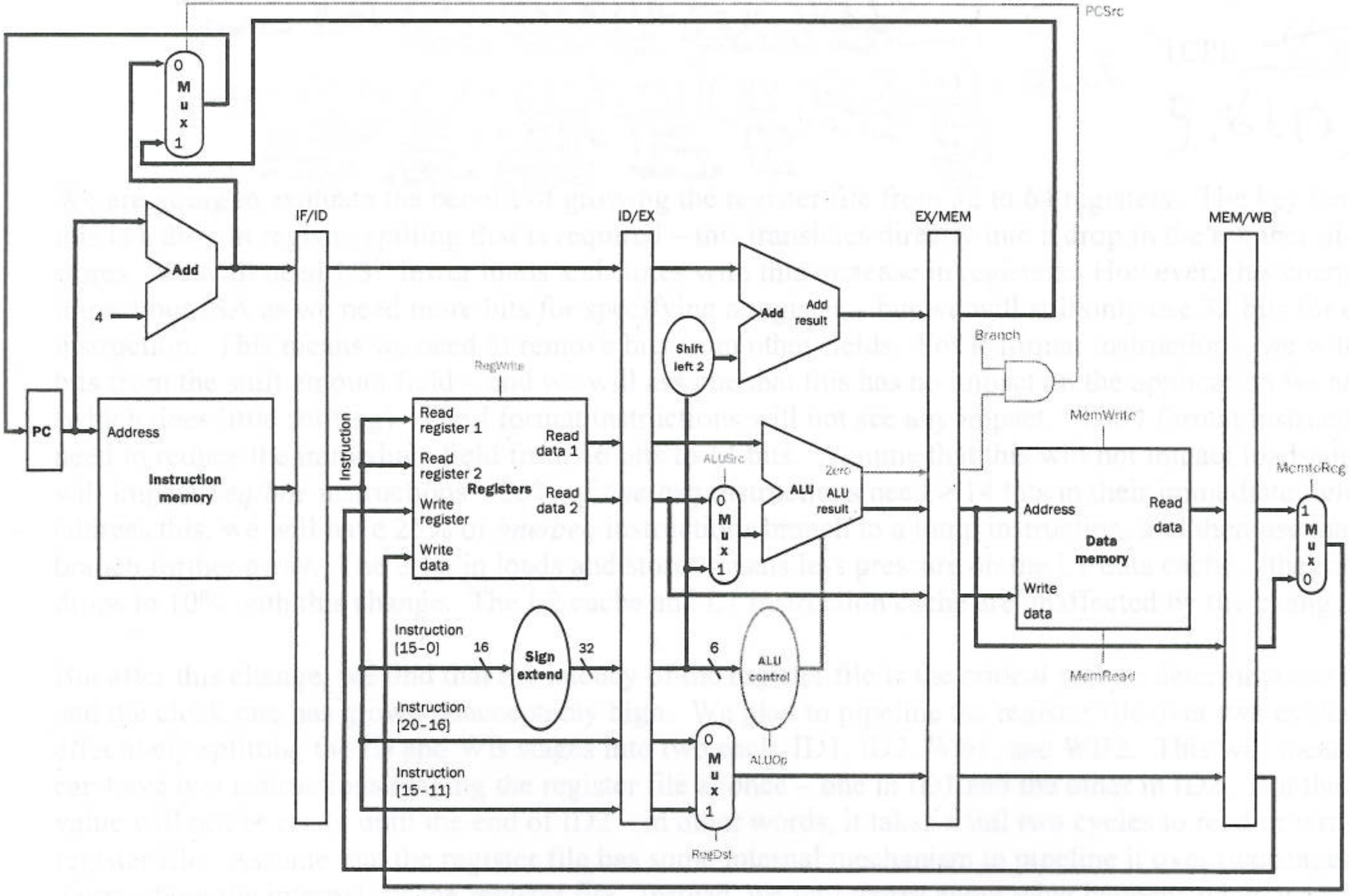
(this is the same loop as the page before – just replicated for your convenience)

```
Loop:  a1 lw $t1, 0 ($s0)
       b1 lw $t2, 4 ($s0)
       c1 addu $t2, $t1, $t2
       d1 lw $t3, 0 ($t2)
       e1 add $s2, $s2, $t3
       f1 addi $t0, $t0, 1
       g1 addi $s0, $s0, -8
       h1 bne $t0, $s1, Loop
```

b. Second, unroll the loop once (i.e. so the loop body contains two iterations) and schedule (i.e. reorder) the code (use the table below) to reduce the total number of cycles required.

Cycle	1st Issue Slot (ALU or Branch)	2nd Issue Slot (LW or SW)
1	N	a1 lw $t1, 0
2	N	b1 lw $t2, 4
3	f1 addi $t0	a2 lw $t4, -8
4	c1 addu $t2	b2 lw $t5, -4
5	g1 addi $s0	d1 lw $t3
6	e2 addu $t5	
7	e1 add	d2 lw $t6
8	bne	N
9	e2 add	N
10		
11		
12		
13		

6. ***You'll be looking FORWARD to these HAZARDS just being a distant MEMORY … (20 points)***: Consider the scalar pipeline we have explored in class, shown below.



For the rest of this question, this architecture will serve as the baseline that we will compare against. It has a 2 GHz clock, and the application that we are testing on this architecture executes 10 billion instructions. Branches are resolved in the MEM stage as shown in the figure above. To handle branch hazards, we will always predict that conditional branches will not be taken. Assume that we have some way of dealing with jumps so that they never cause branch hazards. Conditional branches (*beq* and *bne*) are taken 30% of the time with this application. To handle data hazards, we use data forwarding, just as we covered in class, stalling for one cycle when a load is followed by a dependent instruction. 25% of loads are immediately followed by a dependent instruction (i.e. the next instruction is dependent on the load). 10% of loads are followed by a dependent instruction exactly two instructions later (i.e. the next instruction is independent of the load, but the instruction after that is dependent on the load). And 5% of loads are followed by a dependent instruction exactly three instructions later. 25% of instructions are R-type, 30% are loads, 20% are *bne* or *beq*, 10% are jumps, and 15% are stores. We are using a 16KB direct mapped instruction cache with 32 byte blocks that has a miss rate of 10% for this application. Our data cache is a 32KB 4-way set associative cache with 64 byte blocks and a miss rate of 15%. And we have a unified L2 cache that is a 1MB 8-way set associative cache with 128 byte blocks and a miss rate of 5%. The L1 caches can be accessed in a single cycle, and the miss penalty to go to the L2 is 20 cycles from both L1 caches. The L2 has a 150 cycle miss penalty. Assume that stores ***may*** stall the pipeline – and ***should*** be considered in MCPI.

a. Find the TCPI for this architecture and application. Show your work below:

$$BCPI = \overbrace{1.0}^{\text{peak CPI}} + \overbrace{\underbrace{(.3)}_{\text{30\% loads}} \times \underbrace{(.25)}_{\text{25\% dep.}} \underbrace{(1)}_{\text{1 cycle stall}}}^{\text{pipeline stalls per instruction}} + \overbrace{\underbrace{(.2)}_{\text{20\% branches}} \underbrace{(.3)}_{\text{30\% hazards}} \underbrace{(3)}_{\text{3 cycle stalls}}}^{\text{branch hazard stalls per instruction}} = 1.255$$

$$MCPI = \left(\underbrace{(.10)}_{\text{10\% miss rate for 16KB direct mapped}} \underbrace{(1)}_{\text{L1 cache access *Instr. cache*}} + \underbrace{(.15)}_{\text{15\% miss rate for data cache (32 KB)}} \underbrace{(.45)}_{\text{(\% loads + \% stores) *data cache*}} \right) \left(\underbrace{20}_{\text{penalty to go to L2}} + \underbrace{(.05)}_{\text{L2 miss rate}} \underbrace{(150)}_{\text{penalty for missing L2}} \right) = 4.606$$

1675

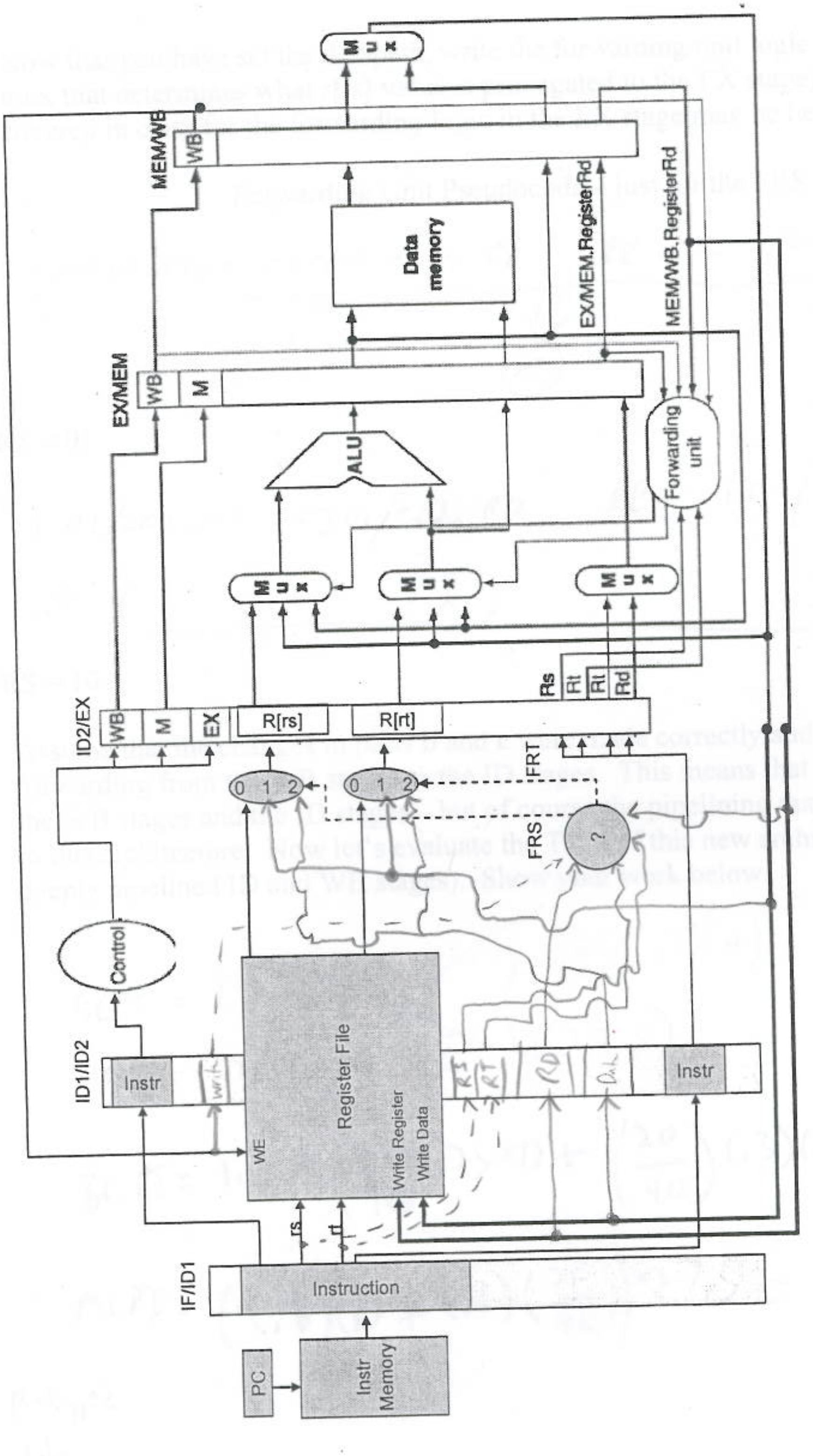
27.5

TCPI: ~~4.61~~ 5.86125

We are going to evaluate the benefit of growing the register file from 32 to 64 registers. The key benefit from this is a drop in register spilling that is required – this translates directly into a drop in the number of loads and stores. We will need 1/3rd fewer loads and stores with this increase in registers. However, this change will impact our ISA as we need more bits for specifying a register – but we will still only use 32 bits for each instruction. This means we need to remove bits from other fields. For R format instructions, we will take the bits from the shift amount field – and we will assume that this has no impact on the application we are using (which does little shifting). The J format instructions will not see any impact. The I format instructions will need to reduce the immediate field from 16 bits to 14 bits. Assume that this will not impact loads/stores, but it will impact *beq/bne* instructions – 25% of *bne/beq* instructions need > 14 bits in their immediate field. To address this, we will have 25% of *bne/beq* instructions branch to a jump instruction, and then use that jump to branch further away. The drop in loads and stores means less pressure on the L1 data cache – the miss rate drops to 10% with this change. The L2 cache and L1 instruction cache are unaffected by the change.

But after this change, we find that the latency of the register file is the critical path to determine our cycle time and the clock rate has grown unacceptably high. We plan to pipeline the register file over two cycles – effectively splitting the ID and WB stages into two each: ID1, ID2, WB1, and WB2. This will mean that we can have two instructions reading the register file at once – one in ID1 and the other in ID2. But the register value will not be ready until the end of ID2 – in other words, it takes a full two cycles to read or write the register file. Assume that the register file has some internal mechanism to pipeline it over two stages – don't worry about the internals of the register file. Instead, we will worry about what happens when we read and write the same register. Pipelining the register file over multiple cycles breaks our assumption that we can write to the register file at the first half of the cycle and read from the register file in the second half. Instead – we will have to add some additional forwarding hardware. At the ID2/EX latch, we will store one of three values for R[rs], and one of three values for R[rt]. The muxes in front of the latch show these three options – but two of these options for each mux are currently unconnected.

b. Your first task is to figure out where to connect these wires. This may require you to do more than just hook up the wires somewhere – you may need to add latches and extend existing wires. Make your modifications to the figure on the following page. There is no WB1/WB2 register bank – so you can use space in the ID1/ID2 register bank if you need to save any values between WB1 and WB2. Focus just on the flow of register values here – you may assume that other values like PC+4, the immediate, and instruction bits 25-21, 20-16, and 15-11 are correctly latched and propagated. Also – do not worry about forwarding from ID2/EX or EX/MEM – these stages will be handled by the normal forwarding logic that we already have in place at the EX stage. The circle with the ? inside represents the forwarding logic. We will fill in the internals of this logic in part c of this question – it will control the selection of the muxes using signals FRS and FRT. But be sure to add whatever other inputs are necessary for the correct operation of this logic – you may want to look at the forwarding logic we covered in class for hints on what to add.


PC
Instr Memory
Instruction
IF/ID1
rs
rt
WE
Register File
Write Register
Write Data
ID1/ID2
Instr
wrt
RS
RT
RD
Dat
Instr
Control
0 1 2
0 1 2
FRS
FRT
?
ID2/EX
WB
M
EX
R[rs]
R[rt]
Rs
Rt
Rt
Rd
M u x
M u x
M u x
ALU
EX/MEM
WB
M
Data memory
EX/MEM.RegisterRd
MEM/WB.RegisterRd
Forwarding unit
MEM/WB
WB
M u x

c. Now that you have set the datapath, write the forwarding unit logic for the FRS control signal (i.e. the mux that determines what r[rs] value is propagated to the EX stage). HINT – the pseudocode that we covered in class for the forwarding logic in the EX stage may be helpful to look at for this.

Forwarding Unit Pseudocode – just for the FRS control signal

```
if( ID1/ID2.RS== MEM/WB.Register Rd  &&  ID1/ID2.RS != Ø  &&  MEM/WB.RegWrite
                              (X)
)
 FRS = 01
if( ID1/ID2.RS ==ID1/ID2.RD  &&  ID1/ID2.RS != Ø  &&  ID1/ID2.RegWrite
    && !(            (X)             )
)
 FRS = 10
```

d. Assume that the changes in parts **b** and **c** were made correctly and we can now correctly handle forwarding from the WB stages to the ID stages. This means that there is no new data hazard between the WB stages and the ID stages – but of course the pipelining may have worsened the existing hazards in this architecture. Now let's evaluate the TCPI of this new architecture (64 registers, and more deeply pipelined ID and WB stages). Show your work below.

~~TCPI~~ CPI: 4.68

~~$BCPI = 1.0 + \left(\frac{30}{105}\right)(.25\times 2) + \left(\frac{20}{105}\right)(.3)(4)$~~

~~$MCPI = \left((.1)(1) + (.10)\left(\frac{45}{105}\right)\right)(27.5)$~~

$BCPI = 1.0 + \left(\frac{20}{90}\right)(.25\times 1) + \left(\frac{20}{90}\right)(.3)(4) = 1,322$

$MCPI = \left((.1)(1) + (.1)\left(\frac{20}{90}\right)\right)\times 27.5 = 3,361$

25 R-type

$20 = \frac{2}{3}\times 30$ lds

20 bne/beq

10 jmp

$10 = \frac{2}{3}\times 15$ st

$5 = .25\times 20$ additional jmp

90 insts

e. Assuming that the cycle time is 10% lower in this new architecture than the original baseline, find the speedup in **execution time** for the application we have been considering on the new architecture over the old architecture. Express this as a % increase. Show your work.

Speedup of the new architecture over the baseline architecture: 55 %

$$\underline{(IC \times 1.05)(CT \times .9)(T_{CPI-new})}$$

$$IC \times CT \times 4.61$$

$$\frac{P_{new}}{P_{old}} = \frac{E_{Told}}{E_{Tnew}} = \frac{IC_{old} \times CT_{old} \times CPI_{old}}{\cancel{10e9 \times .25 \times 5.86 \times 1}\ IC_{new} \times CT_{new} \times CPI_{new}}$$

or $\frac{E_{Told} - E_{Tnew}}{ET_{new}} \times 100\%$

$$CT_{new} = .9CT_{old}$$

$$IC_{old} = 10e9$$

$$IC_{new} = 9e9$$

$IC_{new} = (0.9) IC_{old} = 9e9$

$$\frac{P_{new}}{P_{old}} = \frac{10e9 \times CT_{old} \times 5.86}{9e9 \times .9CT_{old} \times 4.68} = 1.55$$

7. ***Bonus Round (Optional).*** Let me be clear – this problem is ***optional***!! It is worth ***10 bonus points*** – these points will be added to your final exam, but will not impact the curve in any way – they are added after the curve. This is only 10 bonus points, so I'd recommend you save this until you are sure you've done your best on the rest of the exam. This problem will deal with dynamic scheduling – we will look at a relatively simple out-of-order processor that has five main structures we will be concerned with: a 128-entry reorder buffer (ROB), a 12-entry reservation station (RS), the register alias table (RAT), the architectural register file (ARF), and two functional units (FUs). We will assume that the ROB holds output values for instructions that have executed, but not committed. The ARF is the physical storage for the set of committed logical registers. The RAT points a logical register to either the ROB or ARF. The RS holds register specifiers, input operand values, ROB entry #'s, and operations for instructions that have entered the scheduling and instruction windows, but have not yet executed. Assume that the RS is just shared among both FUs – there is no dedicated mapping of resources, as the RS is just a single architectural unit. We will consider functional units to just be a combination of ALUs for execution and data caches for memory.

We will assume that instructions coming from the fetch unit go through three major phases:

a. Allocation – instructions are allocated space in the ROB and RS, and their registers are renamed in the RAT. This takes one cycle, and up to two instructions may allocate in a single cycle. We can continue to allocate up to two instructions every cycle as long as there is space in both the ROB and RS – otherwise we stall.
b. Issue – instructions leave the RS to be executed in FUs. Once all input operands are ready for an instruction in an RS, issue takes one cycle. We will assume that R-type instructions take just one cycle to issue and execute, and that loads take 2 cycles to issue and execute. At the end of the cycle, the instruction writes the value they computed into the ROB. We will assume that there are two FUs, so up to two instructions may issue in a single cycle.
c. Commit – instructions leave the ROB, write to the ARF, and may modify the RAT. This takes a single cycle – and up to two instructions may commit in a single cycle.

Allocation and Commit are done in original program order. Instructions can issue in any order. We will just ignore stores and branches here, as these both complicate out-of-order execution beyond our working example.

For the following sequence of code:

	Alloc	Issue - WB	Commit
lw $t0, 0($s0)	} i	i+1 → i+2	i+3
add $s1, $t0, $s1		i+3 → i+3	i+4
lw $t0, 4($s0)	} i+1	i+2 → i+3	i+4
add $s1, $t0, $s1		i+4 → i+4	
lw $t0, 8($s0)	} i+2	i+3 → i+4	
add $s1, $t0, $s1		i+5 → i+5	
lw $t0, 12($s0)	} i+3	i+4 → i+5	
add $s1, $t0, $s1			
lw $t0, 16($s0)	} i+4		
add $s1, $t0, $s1			
lw $t0, 20($s0)			
add $s1, $t0, $s1			

	Fu 1	Fu 2
i	—	—
i+1	lw	—
i+2	—	lw
i+3	add	lw
[illegible]	"	lw

Assume that the first two instructions went through allocation at cycle i. Prior to cycle i, the ROB and RS are completely empty, all logical registers in the RAT are mapped to logical registers in the ARF, and the ARF contains the following register values:

ARF
(only three entries shown)

Register	Value
$s0	4100
$s1	0
$t0	0

Memory
(only six entries shown)

Address	Value
4100	10
4104	20
4108	30
4112	40
4116	50
4120	60

In the following figures, fill in the state that will be stored in each structure at the end of cycle *i+4*. If an instruction commits from the ROB, just check the "Committed?" entry to indicate that it has left the pipeline.

ROB

Index	Instruction	Result	Committed?
0	i lw	10	✓
1	add	10	✓
2	i+1 lw	20	✓
3	add	30	
4	i+2 lw	30	
5	add	?	
6	i+3 lw	?	
7	add	?	
8	i+4 lw	?	
9	add	?	
10			
11			
12			
…			

~~I [illegible]~~ should be "?" if FU not pipelined. "30" if FU pipelined.

RS

Operation (i.e. Add or Load)	Register ID #1	Register Value	Register ID #2	Register Value	ROB Index
lw	$s0	4100	—		0
add	0	10	$s1	0	1
lw	$s0	4100	—		2
add	2	20	1	10	3
lw	$s0	4100	—		4
add	4	30	3	?	5
lw	$s0	4100	—		6
add	6	?	5	?	7
lw	$s0	4100	—		8
add	8	?	7	?	9

should be "?" if assume FU not pipelined
"30" if FU pipelined

RAT
(only three entries shown)

Register	ROB or ARF?	ROB Index or ARF Register
$s0	ARF	$s0
$s1	RoB	~~8~~ 9
$t0	RoB	~~9~~ 8

ARF
(only three entries shown)

Register	Value
$s0	4100
$s1	10
$t0	20

FU not pipelined

		i	i+1	i+2	i+3	i+4
0	lw	A	I1	I2	C	
1	add	A			I2	C
2	lw		A	I1	I2	C
3	add		A			I2
4	lw			A	x	I1 ←
5	add			A		
6	lw				A	
7	add				A	
8	lw					A
9	add					A

I2 if FU pipelined
I1 if FU pipelined