COSC-520, final exam
2012, spring

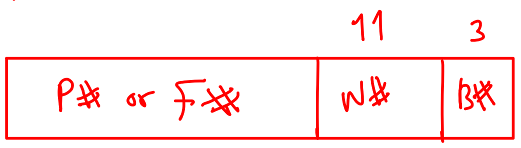
NAME: ____

Open book, open notes (laptops are ok). Partial credit is important; so, explain what you are doing in each question. Do not answer any question with only a number or similar unexplained answer. That will get you very little credit. Show your work, use the back of the pages as needed, but indicate which problem.

I. The maximum physically-addressable memory of machine M is 1024-GB (1-TB). M has 64-bit words, 64-bit virtual addresses, and memory is byte-addressable. Pages are 16 kB.

Q.1. How many bits wide is a TLB entry for M? Only the minimum number of bits needed to do address translation should be included. Also include 4 bytes of other page table information (such as PID, permissions, and so forth). Because TLB entries are read from memory, round up to an integer number of words.

Words are 8 B, so we need 3 bits for byte offset, B#. Page is 16 kB, or $2^4 2^{10}$ B, so offsets are 14 bits and the word offset within a page is 11 bits. Thus, a page number, P#, is $64-14=50$ bits.

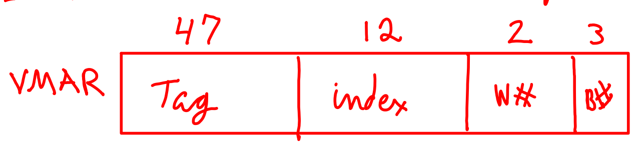


Physical memory is $2^{10}2^{30}$B, or 40 physical address bits, and a frame number, F#, is $40-14=26$ bits. A TLB line has $(\overset{P\#}{50}+\overset{F\#}{26}) = 76$ bits for address translation, which is 10 B after rounding. Total size is $(10+4)$B, which rounded to whole words $(8\text{B}/\text{word})$ is 16 B (or 2 words).

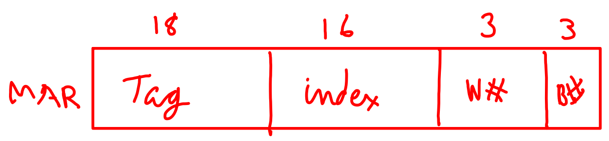
I (continued). M's L1 cache is virtually indexed, virtually tagged, 4-way set-associative, with 4-word blocks. Its total data size is 512 kB. L2 has 32 MB of data, is 8-way associative, physically indexed and tagged, with 8-word blocks.

Q.2. How many bits wide are L1 and L2 cache entries? Both have 1 valid, 1 dirty, 2 LRU, and 16 PID bits.

L1 has 4-word blocks $\Rightarrow$ 2 bits for W#. $2^9 2^{10}$ B of data in 4 ways $\rightarrow 2^{19}/4 = 2^{17}$ B per way. At $(32\text{ B}/\text{block})$, we have $\frac{2^{17}}{2^5} = 2^{12}$ entries per way, ie., 12 index bits: $64-(12+2+3) = 47$ bits for virtual tags. Cache entry is $(47\text{b} + 20\text{b} + 32\text{B}) = (67\text{b} + 256\text{b}) = 323\text{ b}$ per entry.



For L2, we have $(2^5 2^{20}\text{ B}/\text{cache})\left(\frac{\text{cache}}{8\text{ ways}}\right) = 2^{22}\text{ B}/\text{way}$. Blocks are 8 words $\left(\frac{8\text{ B}}{\text{word}}\right) = 64$ B of data per block.



$\#\text{entries} = 2^{22}\text{B}\left(\frac{\text{entry}}{64\text{ B}}\right) = 2^{16}$ entries: 16 b index. Tag is $40-(16+3+3) = 18$ bits. Total is $(18+20+512)\text{b} = (550\text{ b}$ per entry).

I (continued). Register \$3 = 0x000000000041FFB8 just before the following instructions are executed:

```
LW   $11, 0x0( $3 )
SUBi $3, $3, 0x100000
LW   $12, 0x0( $3 )
SUBi $3, $3, 0x100000
LW   $13, 0x0( $3 )
SUBi $3, $3, 0x100000
LW   $14, 0x0( $3 )
SUBi $3, $3, 0x100000
LF16 $F2, 0x0( $3 )
```

LF16 loads 16B from the memory location specified by 0x0(\$3) into the double-precision floating-point register, \$F2. SUBi subtracts the immediate value from \$3.

Q.3. What is the address of the low-order 64-bit word accessed by the LF16 instruction (show in hex)?

The starting address in \$3 is 0x0···041FFB8 and four subtractions of $\underline{(-0x0...0100000)\times 4}$ yields the low-order word's address: 0x 0... 001FFB8

Q.4. What is the high-order 8B-word's address? Word boundaries have low 3 bits of address = 0. We add 1000 (binary) to the address, or 0x8 to get

0x0 ... 01FFC0

Q.5. What is the low-order word's L1 cache index? What is its L1 tag? Show in binary and hex. Use "0...0" for repeated hex or binary zeroes.

	47	12	2	3
VMAR	Tag	index	W#	B#

L1 has 12-bit index, (4 word/block), and 8B words. The low-order 64b word is at address

0 ... 0 0 1 F F B 8 (hex)

$\underbrace{0 \ldots 0\ 0000\ 000}_{\text{Tag}}\underbrace{1\ 1111\ 1111\ 101}_{\text{index}}\underbrace{1\ 1}_{\text{W\#}}\underbrace{000}_{\text{B\#}}$ (binary)

low-order word has index = 1111 1111 1101 or 0xFFD. It's tag is 0...0 in hex or binary.

Q.6. What is the high-order word's L1 cache index? What is its L1 tag?

Address is 0 ... 0 1 F F C 0 (hex)

$\underbrace{0 \ldots 0\ 000}_{\text{Tag}}\underbrace{1\ 1111\ 1111\ 110}_{\text{index}}\overset{\uparrow}{\underset{\text{W\#}}{0}}\ \overset{\uparrow}{\underset{\text{B\#}}{0000}}$ (binary)

index is 1111 1111 1110 (binary) or F F E (hex)   Tag is also 0.

Q.7. Suppose some block in L1 with index xFFE and tag 0 is valid. Does the LF16 instruction's data access hit or miss in L1?

The sequence of addresses accessed is

x0...041ff B8
x0...031ff B8
x0...021ff B8
x0...011ff B8
x0...001ff B8

The digit 4 is in the tag bits, so each address has a different tag. They share the same 12-bit L1 index of xffD. Because L1 is 4-way, the first four accesses fill all 4 ways of L1 at that index. So, the 5th access misses. The last address accessed is x0...01FFC0 w/ index xFFE. L1 hits for that access.

Q.8. What is the page number for the LF16 instruction's data access?

50	11	3
P#	W#	B#

The low 14 bits are page offset. So, the upper 50 bits are the page #. There are 11 leading hex zeroes; so, the first 44 bits of the page number are zeroes. The remaining bits are x1FFB8. or $\underbrace{\ldots 0001\ 11}_{\text{page\#}}\underbrace{11\ 1111\ 1011\ 1000}_{\text{page offset}}$. In binary, the page# is

0...0111 (50 bits), in hex x0...07 (13 digits)

I (continued). The page table has this entry at index x0...07:

[F# = x1234567 (26 bits) , valid = 1, PID = 2367, dirty = 1, LRU = 3, RWX = 100]

Q.9. What two L2 indices does the LF16 instruction's data access have? Give binary and hex. What is the L2 tag?

For the low word:
The physical address is x1234567 concated w/ the low 14 bits of the virtual address (FFB8): $01\ 0010\ 0011\ 0100\ 0101\ \underbrace{0110\ 0111\ 11\ 1111\ 10}\underbrace{11\ 1}_{W\#}\underbrace{000}_{B\#}$

8 word blocks → 3-bit W#, 16-bit index = 0110 0111 1111 1110 or 67FE in hex.

For the high word: Low 14 bits are (FFC0)

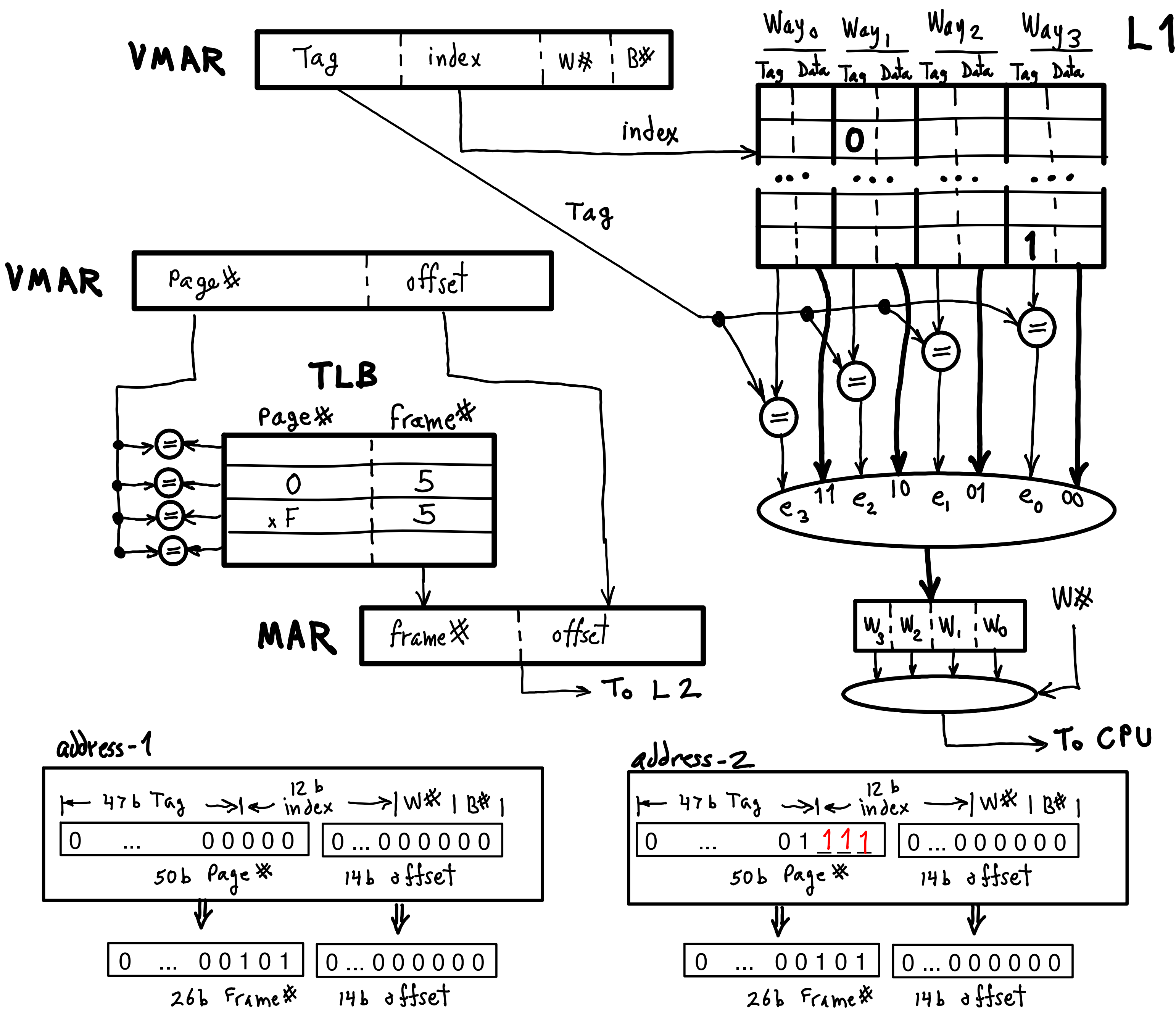
$\underbrace{01\ 0010\ 0011\ 0100\ 0101}_{\text{Tag} = x12345}\ \underbrace{0110\ 0111\ 11\ 1111\ 11}\underbrace{00\ 0}_{W\#}\underbrace{000}_{B\#}$

Index is 67FF.

Q. If L1 misses for both words for the LF16 instruction, and the tag x12345 is not in a valid L2 cache line, how many words of memory must be read from main memory (there is no L3)? How many words of data must be communicated in total (both memory-to-cache and L2-to-L1 reads)? Ignore writing dirty blocks for evicted cache lines and communicating addresses.

Because the access is across an L2 block boundary, and both miss, we must read 2 L2 blocks from memory @ 8 words. We also need to send 2 L1 blocks from L2 @ 4 words. Total is 24 words.

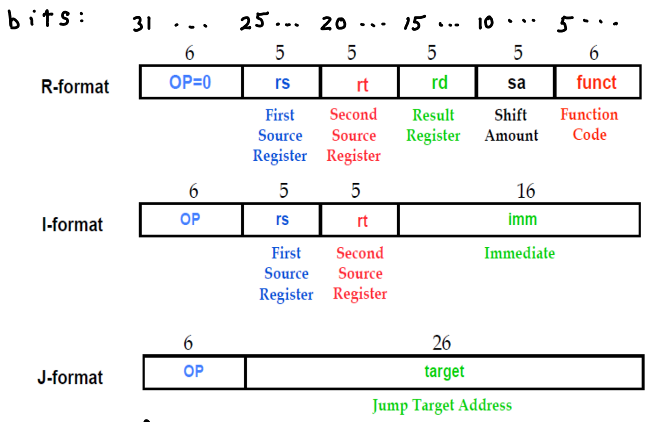
I (conitinued). Shown below are two virtual memory address registers (VMAR). They are the same register with two separate sets of wires tapping its outputs in parallel: one goes directly to L1, the other to the TLB for address translation and then on to L2. The TLB shows that pages 0 and xF map to physical frame 5. L1 shows a valid block with virtual tag 0, and a valid block with virtual tag 1. (Their positions do not indicate their L1 indices.) This is the synonym problem: two virtual addresses refer to the same physical address whose block appears twice in L1. Below is shown an address whose virtual tag is 0 that maps to the same physical word as another address whose tag is 1.



Q. Fill in the missing bits for the second address. Suppose we increased the page size to 128 kB, would it be possible for synonyms to have different L1 indices? Why or why not?

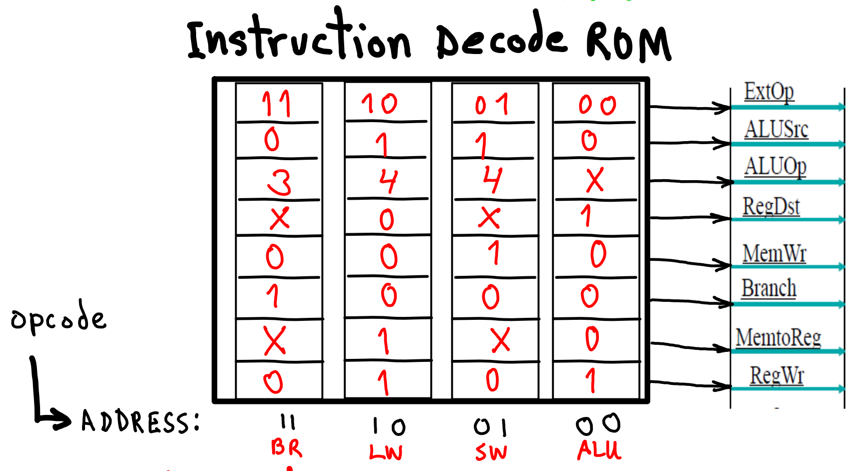
Both addresses are shown as mapping to frame #5 (0...00101). Address-1 has page# = 0, address-2 has page# > 0. The only mapping shown is from page xF (0...01111); so, 3 ones go into missing bits. 128 kB page ⇒ $10^7 \cdot 10^{10}$ B/page ⇒ 17-bit page offset. Because there would be no overlap w/ page# (now 47-bit), all synonyms share identical low 17-bits, and 12-bit indices would always match. So, synonyms would always have same index.

II. Below is shown a 5-stage pipeline architecture (Fetch, Decode, Execute, Memory, and Write-back). Stage registers are the narrow vertical rectangles. At right are instruction formats (high-order bit at the left). Control signals are decoded and then passed through a MUX and written to a pipeline stage register. The control signal destinations are shown as select inputs to datapath MUXs, write-enable signals, and ALU operation control. Branches are taken when the ALU result is 0 (ALU.Zero == 1).



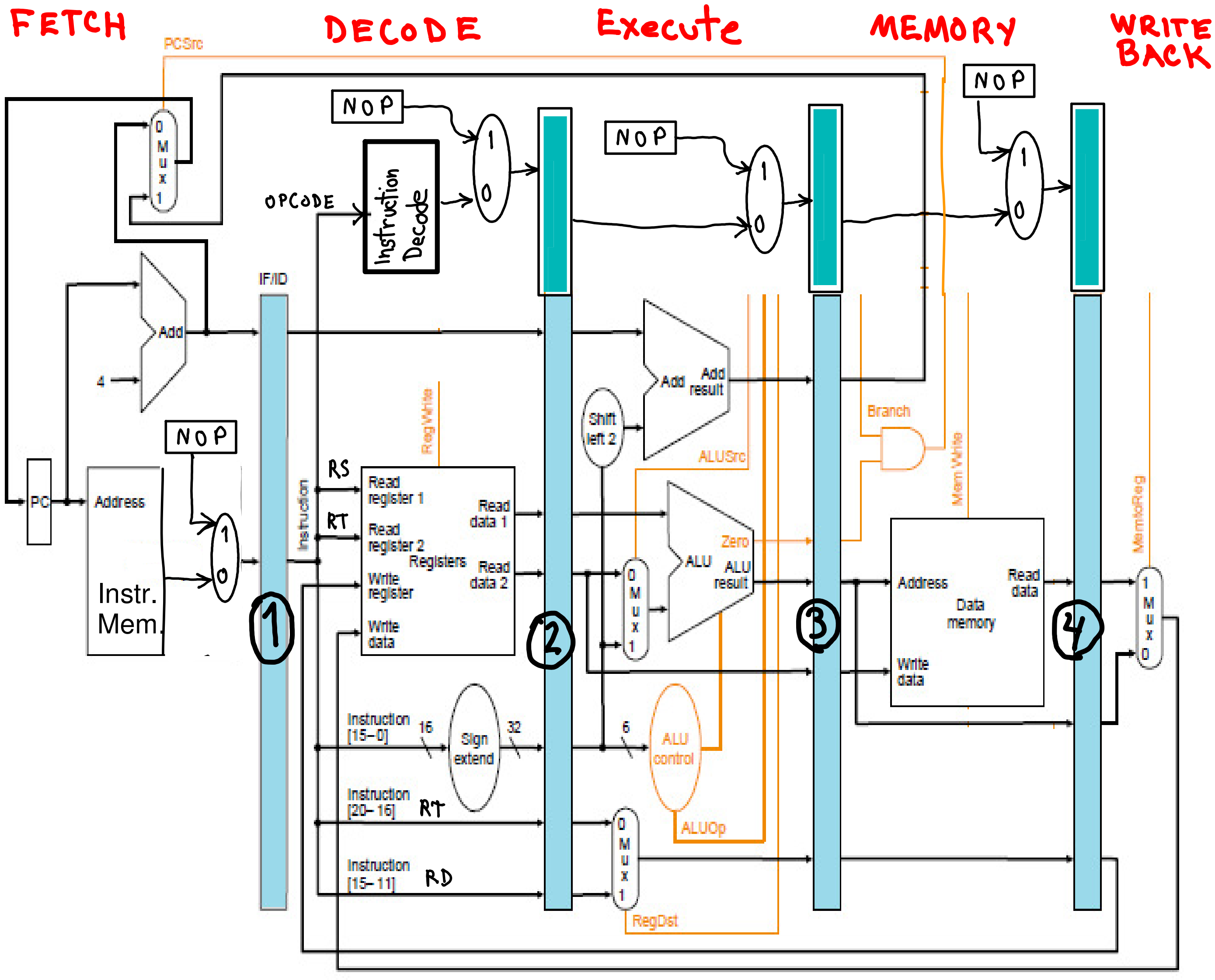
Instruction Decoder.
A ROM containing an 8-element word for each opcode.
For simplicity, assume 2-bit opcodes: BR==11, LW=10, SW=01, ALU operations=00.

ExtOp:	(use the opcode for this field)	
ALUsrc select:	0 = ReadData2,	1 = immediate
ALUOp select:	3 = Subtract,	4 = Add
RegDst select:	0 = RT,	1 = RD
MemWr:	0 = Read,	1 = Write
Branch:	instruction is a branch	
MemtoReg select:	0 = ReadData,	1 = ALUresult
RegWr:	0 = No write,	1 = Write



Instruction Decode ROM
opcode
ADDRESS:

ADDRESS:	11 BR	10 LW	01 SW	00 ALU
ExtOp	11	10	01	00
ALUSrc	0	1	1	0
ALUOp	3	4	4	X
RegDst	X	0	X	1
MemWr	0	0	1	0
Branch	1	0	0	0
MemtoReg	X	1	X	0
RegWr	0	1	0	1



Q. Fill in the missing control signal values in the instruction decoder ROM. A branch is taken if two register values are equal. Branch is an I-format instruction. LW uses RT as its destination register field.

ALUsrc controls MUX to lower ALU input. LW/SW use immediate value from input 1. ALU and BR use two register values; so, use input 0. RegDst controls MUX into ③ 0 for RT field as destination and 1 for RD field selects destination. Only LW and ALU ops do reg. write, so others are X · 0 for LW (RT), 1 for ALU (RD).

Q. In a crude form, this machine does branch prediction. Does it predict taken or not-taken? What is the misprediction penalty?

By fetching in-order below a BR, this is effectively predict not-taken. If taken, target address is ready for PC when BR is in Execute ②, but ① and ② are about to get incorrectly predicted instructions $\Rightarrow$ 2 NOPs inserted into ①, ②.

Q. Suppose a data access (load or store) causes a page fault exception. How many NOPs are injected into the pipeline? Note that the mechanism for jumping to an execption handler is not shown.

The address of the exception handler is ready to be written to the PC when LW/SW is in MEM. Instructions after that must be nullified to have a precise exception: ①, ②, and ③ need NOPs written to them: 3 bubbles.

Q. Consider an instruction mix of 25% ALU, 30% loads, 20% stores, and 25% branch instructions. What would the average CPI be for this CPU? Assume load-use and branch-delay slots cannot be filled by the compiler; all loads incur a load-use bubble; all branches are taken. There is a two-level cache, and misses stall the pipeline. Instruction fetch and data accesses have an L1 miss rate of 5% and an L2 miss rate of 1%. An L1 miss incurs a 10 cycle L2 delay. An L2 miss incurs an extra 100 cycle delay.

For N instruction fetches, we need 1 cycle to access from L1, whether we hit or miss. If we miss L1, we need 10 cycles to access L2 whether we hit or miss L2. This happens for N(5%) which miss L1. For (1%) of L2 accesses (which are all L1 misses), we miss in L2 and pay another 100 cycles. This happens for N(5%)(1%) of instruction fetches. Fetch cycles $= N(1 + (5\%)10 + (5\%)(1\%)100)$. There are $N((30\%)_{LW} + (20\%)_{SW})$ data accesses w/ the same cache performance: data access cycles $= (0.5)N(1 + (5\%)10 + (5\%)(1\%)100)$.

BRs cause 2 bubbles each, so for each BR exiting 2 NOPs also exit. There are $N(1/4)$ BR instructions: BR cycles $= N(1/4)(1+2)$.

There is no mention made of forwarding. LW can make its read data availble for RegFile read when LW is in Write-back ④. Because RegFile is neg-edge triggered, an instruction needing data from LW can be in Decode, but ③ and ④ must be NOPs. There are N(20%) LW, LW exits w/ 2 NOPs $\Rightarrow$ LW cycles $= N(20\%)(1+2)$.

All other instructions exit w/o bubbles: (SW, ALU) cycles $= N(20\%)_{SW}(1) + N(30\%)_{ALU}(1)$.

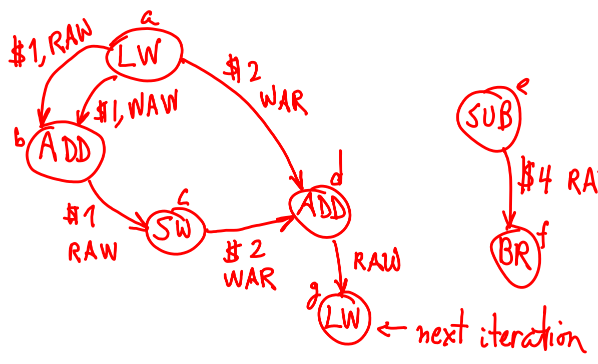
Total cycles is sum of above factors:

$$C(N) = N(1.5)(1 + (5\%)(10 + (1\%)100)) + N(1/4)(3) + N(1/5)(3) + N(1/2)$$

$$\overline{CPI} = \frac{C(N)}{N} = \left(\frac{3}{2}\right)\left(\frac{5}{100}\right)(11) + 3/4 + 3/5 + 1/2 = \frac{3}{2}\left(\frac{11}{20}\right) + \frac{15+12+10}{20} = \frac{33+74}{40} = \frac{110}{40} = 2\,3/4$$

II (continued). We have the following loop:

```
  LOOP:                           ;---- Repeat
a   LW   $1, 0( $2 )              ;----   $1    <== *PTR
b   ADD $1, $1, $3                ;----   $1    <== $1 + $3
c   SW   $1, 0( $2 )              ;----  *PTR <== $1
d   ADD  $2, $2, 4                ;----   PTR  <== PTR++
e   SUB  $4, $4, 1                ;----    CNT--
f   BRn  $4, $0, LOOP            ;---- Until( CNT == 0 )
g LW
```



BRn branches if ($4 - $0) is not zero, and $0 always contains 0. So, the branch runs until $4 reaches 0.

Q. Identify all dependencies (RAW, WAR, etc.) and which registers cause them in the above code. Which ones represent pipeline control or data hazards, and how many pipeline bubbles are caused by these hazards?

a-b is a load-use hazard and w/o forwarding this creates 2 bubbles. b-c is a data RAW hazard and also requires 2 bubbles. e-f is a BR data hazard, also 2 bubbles.

Total is 6 bubbles.

Q. Reschedule the loop to reduce the number of bubbles as much as possible. You may alter individual instructsions (adjusting offsets). How many bubbles result?

We can move LW from the next iteration. Now the LW-ADD hazard is eliminated. Moving SUB removes the SUB-BR hazard, and ADD $1 – SW $1 hazard is eliminated by intervening instructions. SW $1 is not affected by LW $1. ADD $2 dependency hazards are also gone.

BR $4 ADD $2,$2,4 SW $1,0($2) LW $1,4($2) SUB $4,$4,1 ADD $1,$1,$3

pipeline ⟶

Loop needs preamble code though:

nop nop LW $1,0($2)
⟶

Result code: LW $1,0($2)
NOP
NOP
→ LOOP: ADD $1
SUB $4
LW $1,4($2)
SW $1
ADD $2
BR $4

Q. Assuming all instruction fetches and data accesses hit in L1, what is the speedup of the rescheduled loop w.r.t. the original loop?

$$S = \frac{T_{old}}{T_{new}} = \frac{(\text{old loop cycles})(1/CR)}{(\text{new loop cycles})(1/CR)} = \frac{6 + 6\ \text{NOPs}}{6} = 2$$

Q. Given the previous question's assumptions, what fraction of the total program's execution would the loop have to represent to attain an overall speedup of 1.25 from this rescheduling?

$$S = \frac{1}{(1-f) + f/2} = 5/4 \Rightarrow (1-f) + f/2 = 4/5$$

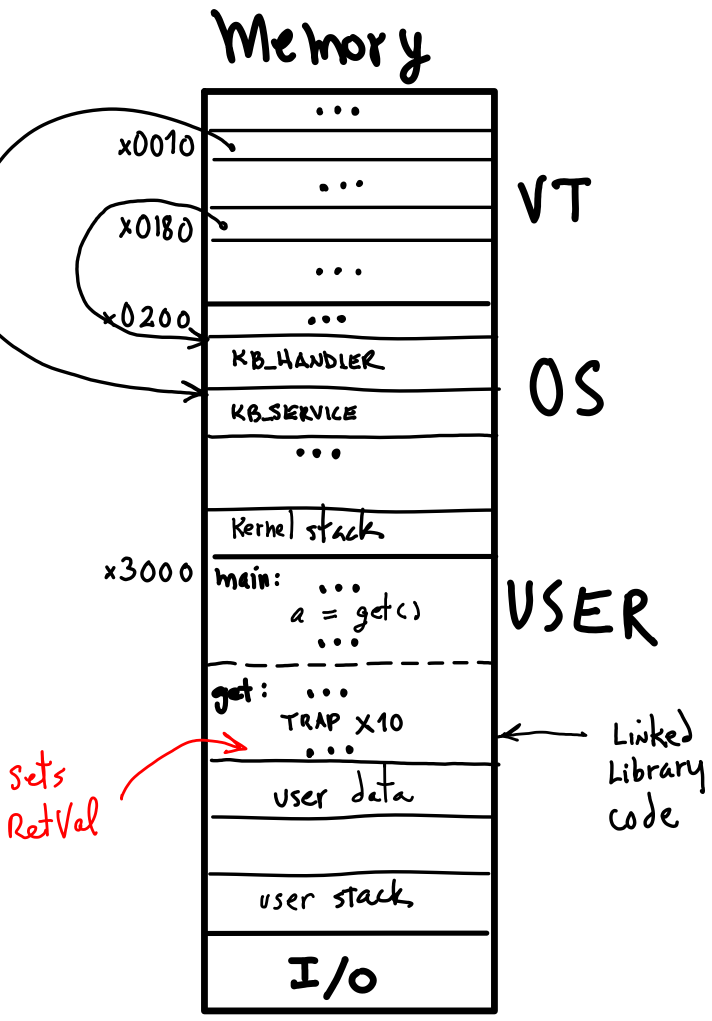
$$2 - 2f + f = 8/5$$

$$-f = 8/5 - 2 = -2/5 \Rightarrow$$ 40% of execution is loop body.

III. Joe is working on the OS for the LC3, and has written a device driver for the keyboard. He has also written a service routine that user programs can call to get keyboard input data. The service routine is envoked with "TRAP x10". The interrupt vector for the keyboard is x0180. The service routine is called by a library function "get()" that is assembled separately and linked with the user's C code. Joe's code is shown below (mostly just the comments): The runtime memory map is shown at right. Traps act like exceptions and run in kernel mode. OS space is protected.

```
KB_INIT
   ;--- write VT for KB_SERVICE
   ;--- write VT for KB_HANDLER
   ;--- enable keyboard interrupts
        RET
KB_HANDLER
   ;--- get keyboard data
   ;--- insert in buffer
   ;--- set Ready flag variable
        TRAP x12
   ;--- enable keyboard interrupts
        RTI
KB_SERVICE
   ;--- while (1)
   ;---   check Ready flag
   ;---   if Ready
   ;---      get data from buffer
   ;---      put data in R0 as return value
             RTI
   ;---   else
             TRAP x11
   ;--- end-while
```

*TRAP x11 jumps to a scheduler that starts another program running. TRAP x12 checks whether some program is waiting for keyboard data, and if so, schedules it to be reloaded and run as if it was returning from its TRAP x11 call.



Q. The C call to get() in user's main() returns a character value into the variable "a". The code generated for this by the compiler uses the C function-call stack protocol. From which stack, kernel or user, would this code get its return value? Where in the above map is the code that puts the return value onto the stack?

The C call is user code; so, return value comes from get()'s call frame on user's stack. Get() code must set up stack return value, just after TRAP x10 call.

Q. The call to sleep, "TRAP x11", in KB_SERVICE, switches contexts to another program, including swapping the entire user memory content. When the sleeping program is awakened, its context has been restored and it is exiting the sleep() trap via RTI. Before sleep() returns, what task must it complete so that RTI and the TRAP x10 call both return corrrectly? Explain.

The kernel stack must have the PC, PSR values that were pushed onto the stack by the TRAP x11 call. Also, the stack needs to be restored so that TRAP x10 call's pushed PC, PSR are below, and the return to user code will work.