UC Berkeley CS61C Fall 2018 Final Exam Answers

A mystery, byte-addressed cache has Tag:Index:Offset (T:I:O) = 9:4:3. For the computer,

a) What is the *virtual address space*? (select ONE) ◯8-bit ◯16-bit ◯32-bit ◯64-bit ●Not enough info!
Thanks to Virtual Memory (assumed standard in today's computers, wasn't always true), the virtual address space is not connected to the cache size or TIO width. When people casually say "it's a 32-bit machine", that means the virtual address space.

b) What is the *physical address space*? (select ONE) ◯8-bit ●16-bit ◯32-bit ◯64-bit ◯Not enough info!
Yep, T+I+0 = physical address size, here 16 bits.

Different caches can have the same T:I:O! Let's explore that in parts (b) and (c) below.

c) How could we ***maximize*** cache size, while preserving T:I:O=9:4:3? (select ONE per row)

Associativity	◯ 2-way set	◯ Direct Mapped	● 8-way set	◯ 4-way set
Block Size	◯ 4 bytes	● 8 bytes	◯ 12 bytes	◯ 3 bytes
# of Blocks	◯ 8 blocks	◯ 16 blocks	◯ 4 blocks	● 128 blocks

We start with block size, which is a function of the offset (3 bits), so that's 8 bytes regardless of cache associativity configuration. Since we have 4 index bits, we use the 2^4 = 16 indices to index into a set of things, here the *most* we can have (if we want to *maximize* cache size), which is 8-way set associative.
So the number of total blocks is 2^4 sets/cache * 2^3 blocks/set = 2^7 blocks/cache = 128 blocks in this 1 cache.

d) How could we ***minimize*** cache size, while preserving T:I:O=9:4:3? (select ONE per row)

Associativity	◯ 2-way set	● Direct Mapped	◯ 8-way set	◯ 4-way set
Block Size	◯ 4 bytes	● 8 bytes	◯ 12 bytes	◯ 3 bytes
# of Blocks	◯ 8 blocks	● 16 blocks	◯ 32 blocks	◯ 64 blocks

Similarly, we start with block size, which is a function of the offset (3 bits), so that's 8 bytes regardless of cache associativity configuration. Since we have 4 index bits, we use the 2^4 = 16 indices to index into a set of things, here the *least* we can have (to *minimize* cache size), which is 1-way set associative, or direct mapped.
So the number of total blocks is 2^4 sets/cache * 2^0 blocks/set = 2^4 blocks/cache = 16 blocks in this 1 cache.

e) Now we're working with a write-back, 1024B direct-mapped cache that has 8B blocks. We're interested in seeing if we can lower our AMAT if our memory access pattern is iterating through an array with a fixed stride. The majority of our misses are conflict misses, and we have an inconsequential amount of compulsory and capacity misses. For each of the following modifications, mark how it changes each component of AMAT (Hit Time, Miss Penalty, Miss Rate) and the overall Hardware Complexity.

Modification	Hit Time	Miss Penalty	Miss Rate	Hardware Complexity
Change block size from 8B to 16B, *but keep the cache size the same*	◯ Increase ◯ Decrease ● No effect	● Increase ◯ Decrease ◯ No effect	◯ Increase ● Decrease ◯ No effect	◯ Increase ◯ Decrease ● No effect

Change to 2-way Associativity (same cache & block size)	● Increase ○ Decrease ● No effect	○ Increase ○ Decrease ● No effect	○ Increase ● Decrease ○ No effect	● Increase ○ Decrease ○ No effect

A block size change simply involves changing the aspect ratio of the cache (so height [rows, indices, sets] * columns [blocks size controlled by offset] = constant) -- doubling the width is halving the height. So you're giving a bit from I to O. Doesn't affect hit time (the time to a cache hit … i.e., the time to determine if a cache hits, and the time to get the data back to the registers), but does increase miss penalty (how much data you have to bring from the lower level of the cache) and does reduce the miss rate because of the benefits of spatial locality (but not forever, you'll recall when we get TOO few blocks the miss rate actually goes up because of ping pong and other effects). Doesn't affect the hardware complexity in terms of needing a new comparator or logic block or anything else since the cache is still doing the same work.

An associativity change (1-way to 2-way) might (incrementally) increase the hit time since the hardware now has to parallel-compare which of the two blocks in a set match and then route it accordingly; it's still going to be on the order of 1 cycle for a hit, so we accepted "increase" or "no effect". Since we're not changing block size it doesn't affect the miss penalty, but hopefully we now don't have as many conflict misses so our miss rate will decrease, and we certainly need more hardware to do the tag comparisons.

M2) *Floating down the C...* [this is a 2-page question] (8 points = 1,1,2,1,1,1,1, 20 minutes)

Consider an 8-bit "minifloat" S**E**EEMMMM (1 sign bit, 3 exponent bits, 4 mantissa bits). All other properties of IEEE754 apply (bias, denormalized numbers, ∞, NaNs, etc). The bias is -3.

a) How many minifloats are there in the range [1, 4)? (i.e., 1 ≤ **f** < 4) **32**

Bias of -3 means the exponent can go from -3 to 4 → to 2^3 so we are in range. 1 and 4 are powers of 2, so that's two "ranges", and with MMMM = 16 mantissa values, that's **32** mantissa values.

b) What is the number line distance between 1 and the smallest minifloat bigger than 1? **1/16**

1 is a special number since the exponent is 0 (after the bias is applied), thus it's
2^0 * 1.MMMM → 1.MMMM (the binary shift left/right by $2^{EXP-Bias}$ goes away) → the least M is counting by **1/16**
so the next number after 1.0000_2 is 1.0001_2 which is 1+1/16.

c) Write **`times2`** in one line using integer operations. Assume the leftmost "E" bit of **`f`** (bolded above) is 0.

```
minifloat times2(minifloat f) { return f * 2.0; }
```

```
times2: addi a0, a0, 0b00010000 = 0x10 ## Assume f is in the lowest byte of the register

        Ret
```

We have to add one to the exponent to make it work,**c**ool!

Consider the following code (and truncated ASCII table; as an example of how to read it, "G" is `0b0100 0111`):

```
uint8_t mystery (uint8_t *A) {
   return *A ? (*A & mystery(A+1)) : 0xFF;
}
```

$b_7 b_6 b_5$ → / Bits	0 1 1	1 0 0	1 0 1
b_4 b_3 b_2 b_1 / Column → / Row ↓	3	4	5
0 0 0 0	0	@	P
0 0 0 1	1	A	Q
0 0 1 0	2	B	R
0 0 1 1	3	C	S
0 1 0 0	4	D	T
0 1 0 1	5	E	U
0 1 1 0	6	F	V
0 1 1 1	7	G	W
1 0 0 0	8	H	X
1 0 0 1	9	I	Y
1 0 1 0	10	:	J	Z
1 0 1 1	11	;	K	[
1 1 0 0	12	<	L	\
1 1 0 1	13	=	M	]
1 1 1 0	14	>	N	^
1 1 1 1	15	?	O	_

d) Where is **A** stored? (*not* what it points to, ***A**)

◯code ◯static ◯heap ◯stack

stack (since it's a local variable / parameter / immediate)

e) What is **(char)mystery("GROW")**? _____

The code does an AND of all the characters bits, the upper bits are 100 & 101 → 100, and the lower bits are 1111 & 0111 & 0010 → 0010, so it's 100 0010 → **B**

e) [alternate exam]

What is **(char)mystery("FLOW")**? _____

The code does an AND of all the characters bits, the upper bits are 100 & 101 → 100, and the lower bits are 1111 & 0111 & 0110 & 1100 → 0100, so it's 100 0010 → **D**

f) A two-character string is passed into **mystery** that makes it return the **uint8_t** value **0** (not the character "**0**"). The first character is "**M**" ["**K**" alternate exam], the second character is a number from 1-9. Which?

◯1 ◯2 ◯3
◯4 ◯5 ◯6
◯7 ◯8 ◯9

What number has no bits in common with M's bits=100 1101 → all numbers have the high nibble with no bits in common so it's only the bits that only have 1 in the 2s column, thus 0010 or 0000 (but 0 is not part of it), so it must be 0010 → **2**. (note the ASCII low nibble of a 0-9 number is the number itself)

[Alternate exam] What number has no bits in common with K's bits=100 1011 → all numbers have the high nibble with no bits in common so it's only the bits that only have 1 in the 4s column, thus 0100 or 0000 (but 0 is not part of it), so it must be 0100 → **4**. (note the ASCII low nibble of a 0-9 number is the number itself)

d) Incrementing or decrementing a variable is common: e.g., **sum += amount**, so we decide to make a new RISC-V instruction (based on I-format instructions), and reuse the unused bits from **rs1** to give to the immediate (**rd** will be read *and* written, don't touch `funct3` or `opcode`). Using *only* that operation, what's the most **amount** that **sum** could now increase by (approx)? (select ONE for each column)

◯1 ◯2 ◯4 ◯8 ◯16 ◯32 ◯64 ◯128 ◯256 ◯512	◯<blank> ◯kilo ◯mega ◯giga ◯tera ◯peta ◯exa ◯kibi ◯mebi ◯gibi ◯tebi ◯pebi ◯exbi

12 bits of immediate + 5 more from register = 17, but it's signed so $-2^{16} \rightarrow 2^{16} - 1$, approx 2^{16} which is **64 kibi.**

M3) *Just one more thing…RISC-V self-modifying code!* (8 points, 20 minutes)

(this is meant to be a fairly hard problem, we recommend saving it until the end of the exam…)

Your Phase I date was too late, so you can't get into the course you want. You need to hack CalCentral's server to enroll yourself! You find the following program running on the CalCentral server:

```
.data ### Starts at 0x100, strings are packed tight (not word-aligned)
    benign: .asciiz "\dev/null"
    evil:   .asciiz "/bin/sh"

.text ### Starts at 0x0 )The alternate exam swapped t2,t0 for t1,t2, but otherwise it was the same
      addi t0 x0 0x100        ### Load the address of the string “\dev/null”
      addi t2 x0 '/'          ### Load the correct character. The ASCII of ’/’ is 47_10.
      jal ra, change_reg
      sb t2 0(t0)             ### Fix the backslash “\dev/null” → “/dev/null”
      addi a0 x0 0x100
      jal ra, os
```

The subroutine `change_reg` allows a user to arbitrarily set the value of any registers they choose when the function is executed (similar to the debugger on Venus). **`os(char *a0)`** runs the command at **`a0`**. *Select as few registers as necessary, set to particular values* to **MAKE THE RISC-V CODE MODIFY ITSELF** so the **`os`** function runs “**`/bin/sh`**” to hack into the CalCentral database. **Please note: even though `change_reg` can arbitrarily change any register it STILL follows the RISC-V calling convention. You CANNOT assume that the registers are initialized to zero on the launch of the program. Also, the assembler is NOT optimized.** Hint: Think about *where* the change needs to happen, then *what* it should be.

Reg	Value to set it to (in HEX without leading zeros)
☐ a0	0x
☐ a1	0x
☐ a2	0x
☐ s0	0x
☐ s1	0x
☐ s2	0x
■ t0	**0x12**
☐ t1	0x
■ t2	**0xA0**
☐	Not Possible

We have to change “**`addi a0 x0 0x100`**” to “**`addi a0 x0 0x10A`**” since the next string starts right after the first, which has 9 characters and a trailing 0, so that’s bytes 0-9, meaning byte 10, or 0x10A is the location of the string you need to pass to os in a0.

```
  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0← old
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
|<---------- IMM12 ---------------->|<-----rs1---->|< func3>|<-----rd ---->|<---- opcode ------>|
|<----------BYTE3------>|<----------BYTE2------>|<----------BYTE1------>|<----------BYTE0------>|
  0  0  0  1  0  0  0  0  1  0  1  0  0  0  0  0  0← new
```

We need to store it in byte 18 (4 words = 16 bytes to skip over and 2 bytes within the word to skip), and write `A0` into the 18th = 0x12 byte to clobber the lower nibble of the immediate with A and keep rs1 to be 0, to make “**`addi a0 x0 0x100`**” become “**`addi a0 x0 0x10A`**”

`[what]t2 =` **`0xA0`**`, [where]t0 =` **`0x12`**

The alternate exam swapped t2,t0 for t1,t2, but otherwise it was the same.

F1) *VM...* (20 points = 12+4+4, 30 minutes)

a) What are the steps that occur for a memory read (when a **page fault** is encountered)? You may assume there's room in memory for a new page, and we're using LRU replacement. Assume there's no data cache. Mark the order of the required actions, there's at most one choice per #, and e*very row/col should have a #.*

Below, ◯⇒Select ONE, ☐⇒Select ALL that apply	1	2	3	4	5	6	7	8	9	10
Request data using the (◯**physical** ●**virtual**) address	●	◯	◯	◯	◯	◯	◯	◯	◯	◯
Access Page Table with ◯**PPN** ●**VPN** ◯**Offset**	◯	◯	◯	●	◯	◯	◯	◯	◯	◯
Access TLB with ◯**PPN** ●**VPN** ◯**Offset**	◯	◯	●	◯	◯	◯	◯	◯	◯	◯
Adjust LRU bits in ■**TLB** ☐**Page Table** ☐**Memory**	◯	◯	◯	◯	◯	◯	◯	◯	●	◯
Split physical address into (●**PPN+Offset** ◯**VPN+Offset** ◯**PPN+VPN**) fields	◯	◯	◯	◯	◯	●	◯	◯	◯	◯
Split virtual address into (◯**PPN+Offset** ●**VPN+Offset** ◯**PPN+VPN**) fields	◯	●	◯	◯	◯	◯	◯	◯	◯	◯
Request new page from OS/Memory manager	◯	◯	◯	◯	●	◯	◯	◯	◯	◯
Update Page Table with new ■**PPN** ☐**VPN** ☐**Offset**	◯	◯	◯	◯	◯	◯	●	◯	◯	◯
Update TLB with new ■**PPN** ■**VPN** ☐**Offset**	◯	◯	◯	◯	◯	◯	◯	●	◯	◯
Return the data *(this is the last thing we do)*	◯	◯	◯	◯	◯	◯	◯	◯	◯	●

_1_ Request data using the **virtual** address
_2_ Split virtual address into **VPN, Offset** fields
_3_ Access TLB with **VPN**
_4_ Access Page Table with **VPN**
_5_ Request new page from OS/Memory manager
_6_ Split physical address into **PPN, Offset** fields
_7_ Update page table with new **PPN**
_8_ Update TLB with new **PPN,VPN**
_9_ Adjust LRU bits on **TLB**
_10_ Return the data

b) Mark the following questions as either **True** or **False**:

●**True** ◯**False**	If we have a TLB which contains a number of entries equal to `MEMORY_SIZE` / `PAGE_SIZE`, every TLB miss will also be a page fault.
◯**True** ●**False**	If we change our TLB to direct-mapped, we're likely to see fewer TLB misses.
◯**True** ●**False**	Every TLB miss is equally expensive in terms of the amount of time it takes for us to resolve our virtual address to a physical address.
◯**True** ●**False**	A virtual address will always resolve to the same physical address.

__**True**__ If we have a TLB which contains a number of entries equal to MEMORY_SIZE / PAGE_SIZE, every TLB miss will also be a page fault.
__**False**__ If we change our TLB to direct-mapped, we're likely to see fewer TLB misses.

__**False**__ Every TLB miss is equally expensive in terms of the amount of time it takes for us to resolve our virtual address to a physical address. (Page faults are much more expensive than Page-Table hits. Both are possible outcomes of a TLB miss.)
__**False**__ A virtual address will always resolve to the same physical address.

c) Consider a VM system on a RISC-V 32-bit machine with 2^{20} page table rows, no TLB, and limited physical RAM, choose ONE of the following code snippets that **would always have the most page faults per memory access** by touching elements of a page-aligned `uint8_t` array `A`, and choose ONE value of `STRIDE` (choose the minimum possible value that accomplishes it). Both `A_SIZE` and `STRIDE` are powers of 2, and `A_SIZE > STRIDE`. `random(N)` returns a random integer from 0 to N.

```
○ for (i = 0;      i < STRIDE; i++)       { A[random(A_SIZE)] = random(255); }
○ for (i = 0;      i < A_SIZE; i++)       { A[random(STRIDE)] = random(255); }
○ for (i = 0;      i < A_SIZE; i++)       { A[i] = random(STRIDE); }
○ for (i = 0;      i < STRIDE; i++)       { A[i] = random(255); }
● for (i = 0;      i < A_SIZE; i+=STRIDE) { A[i] = random(255); }
○ for (i = STRIDE; i < A_SIZE; i++)       { A[i] = A[i-STRIDE];  A[i-STRIDE] = A[i]; }
```

STRIDE: ○2^0 ○2^1 ○2^2 ○2^3 ○2^4 ○2^5 ○2^6 ○2^7 ○2^8 ○2^9 ○2^{10} ○2^{11} ●2^{12}
○2^{13} ○2^{14} ○2^{15} ○2^{16} ○2^{17} ○2^{18} ○2^{19} ○2^{20} ○2^{21} ●2^{22} ○2^{23} ○2^{24}
○2^{25} ○2^{26} ○2^{27} ○2^{28} ○2^{29} ○2^{30} ○2^{31} ○2^{32}

To pound on the memory system the most, you'd request a different page with every access. (the first two can't guarantee that). Stride should be page size, and since 32 bits virtual address = VPN (20 bits since 2^{20} page table rows) + offset, then offset = 12. Therefore stride should be page size = 2^{offset} = 2^{12}. The alternate exam had 2^{10} page table rows, so using the same reasoning, it'd be stride = page size = 2^{22}.

F2) *SDS* (20 points = 8+5+7, 30 minutes)

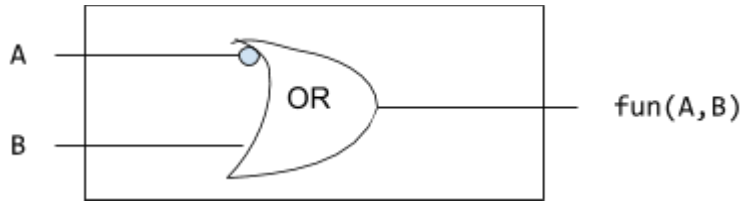
a) Transform the **`fun`** function below into the *fewest Boolean gates* that implement the same logic. You may use AND, OR, XOR and NOT gates. *Hint: start with the truth table.*
`bool fun(bool A, bool B) { return (A == B) ? true : B; }`

a) If you write out the truth table, it's

A	B	fun(A,B)
FALSE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

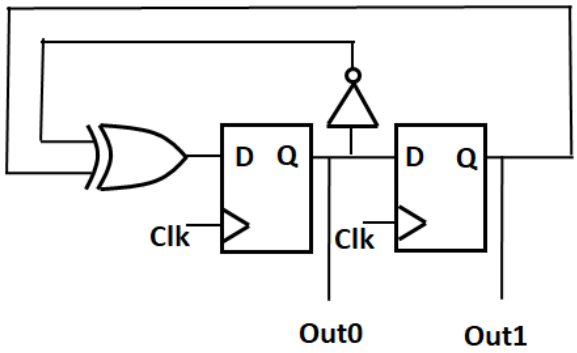
...and using sum of products is:
!A!B + !AB + AB
!A (!B + B) + AB distribution
!A + AB complimentarity, identity
!A + B [uniting theorem v.2: x + !xy = x + y (here x = !A, B = y)]

(the alternate exam swapped A and B. so they would have **A+!B** (the bubble would be on B, not A below).



---

b) The logic implementation of a state machine is shown in the figure below. How many states does this state machine have? (Assume that it always starts from Out0=0, Out1=0)



Out0 ← xor(!Out0, Out1)
Out1 ← Out0

Out0	**Out1**	Binary
FALSE	FALSE	0
TRUE	FALSE	2
FALSE	TRUE	1
FALSE	FALSE	0
etc	etc	etc

(Out0=TRUE, OUT1=TRUE) is never accessed.

Number of states = ○1 ○2 ●3 ○4 ○5 ○6 ○7 ○8 ○9

---

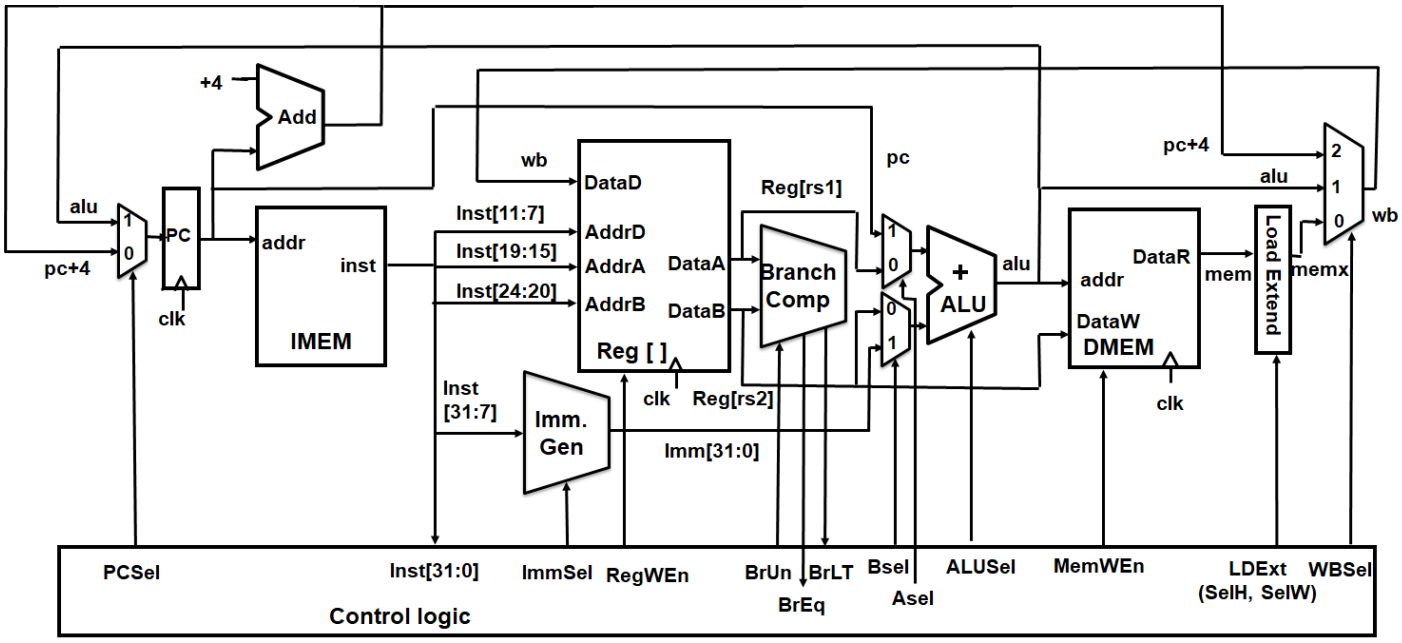
c) In the figure above, flip-flop clk-to-q delay is **40ps**, setup time is **30ps** and hold time is **30ps**. XOR delay is **20ps** and the inverter delay is **10ps**. What is the *maximum frequency* (F_{max}) of operation?

$F_{max} = 1/(t_{clk\text{-}q} + t_{inverter} + t_{xor} + t_{setup}) = 1/(100ps) = 10$ GHz .

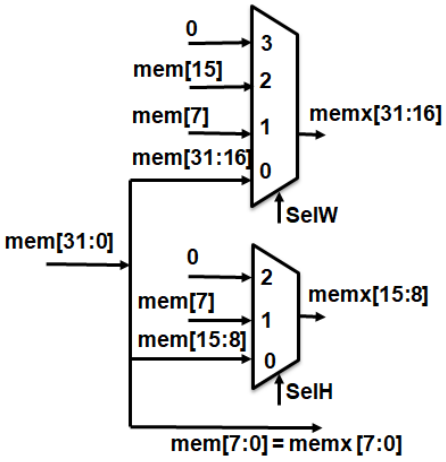
The alternate exam had double the delays and setup/hold times, so the Fmax would be 1/200ps = 5 GHz.

F3) *Datapathology* [this is a 2-page question] (20 points = 4+10+6, 30 minutes)

The datapath below implements the RV32I instruction set. We'd like to implement sign extension for loaded data, but our loaded data can come in different sizes (recall: **`lb`**, **`lh`**, **`lw`**) and different intended signs (**`lbu`** vs. **`lb`** and **`lhu`** vs **`lh`**). Each load instruction will retrieve the data from the memory and "right-aligns" the LSB of the byte or the half-word with the LSB of the word to form **`mem[31:0]`**.



a) To correctly load the data into the registers, we've created two control signals **SelH** and **SelW** that perform sign extension of **`mem[31:0]`** to **`memx[31:0]`** (see below). **SelH** controls the half-word sign extension, while **SelW** controls sign extension in the two most significant bytes. What are the Boolean logic expressions for the four (0, 1, 2, 3) **SelW** cases in terms of Inst[14:12] bits to handle these five instructions (**`lb`**, **`lh`**, **`lw`**, **`lbu`** and **`lhu`**)? **SelH** has been done for you. In writing your answers, use the shorthands "I14" for Inst[14], "I13" for Inst[13] and "I12" for Inst[12]. You don't have to reduce the Boolean expressions to simplest form. (Hint: green card!) Answers (and simplified form)



SelW=3	**I14 • ~I13 • ~I12 + I14 • ~I13 • I12 = I14**
SelW=2	**~I14 • ~I13 • I12 = ~I14 • I12**
SelW=1	**~I14 • ~I13 • ~I12**
SelW=0	**~I14 • I13 • ~I12 = I13**
SelH=2	I14 • ~I13 • ~I12
SelH=1	~I14 • ~I13 • ~I12
SelH=0	I13 + I12

(Single-bit values **`mem[7]`** and **`mem[15]`** are wired to 8 or 16 outputs)

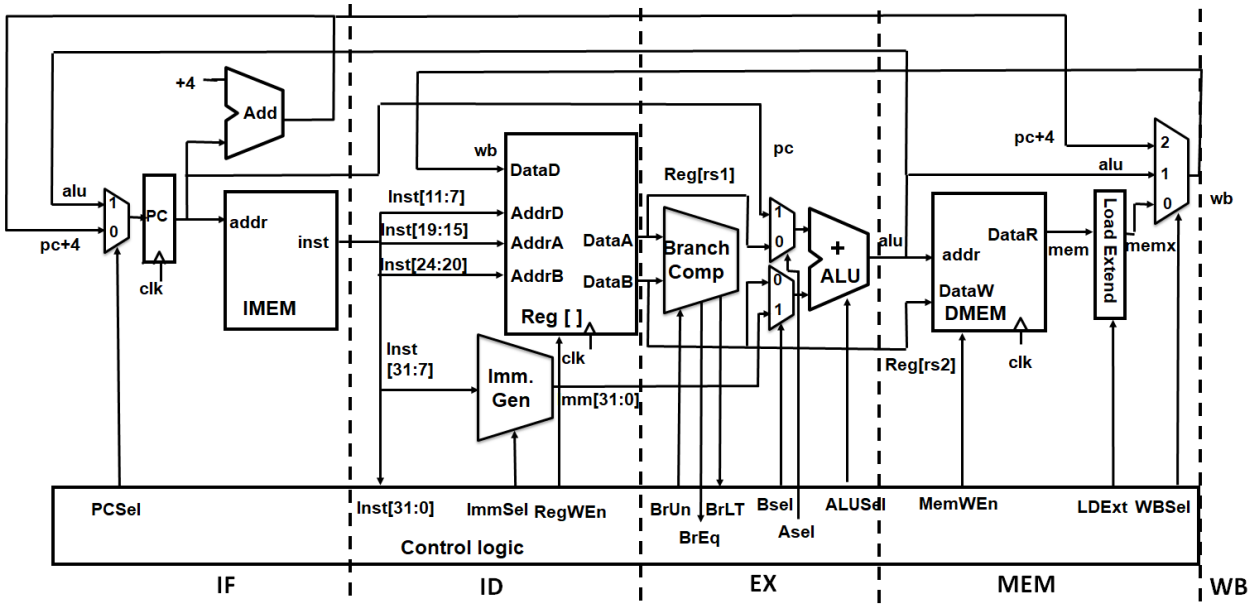
Here's how we figured it out -- we made a table:

	INST 111 432							
Inst	funct3	byte3+byte2			byte1		byte0	
lb	000	mem[7]	SelW=1	|	mem[7]	SelH=1	| mem[7:0]	
lh	001	mem[15]	SelW=2	|	mem[15:8]	SelH=0	| mem[7:0]	
lw	010	mem[31:16]	SelW=0	|	mem[15:8]	SelH=0	| mem[7:0]	
lbu	100	0	SelW=3	|	0	SelH=2	| mem[7:0]	
lhu	101	0	SelW=3	|	mem[15:8]	SelH=0	| mem[7:0]	

… and then reversing the table for the cases based on the INST funct3 bits above yields the values above.

F3) *Datapathology, continued* (20 points = 4+10+6, 30 minutes)

(this is the same diagram as on the previous page, with five stages of execution annotated)



b) In the RISC-V datapath above, mark what is used by a `jal` instruction. *(See green card for its effect...)*

Select one per row	**PCSel Mux:**	○ "pc + 4" branch	● "alu" branch	○ * (don't care)	
	ASel Mux:	● "pc" branch	○ Reg[rs1] branch	○ * (don't care)	
	BSel Mux:	● "imm" branch	○ Reg[rs2] branch	○ * (don't care)	
	WBSel Mux:	● "pc + 4" branch	○ "alu" branch	○ "mem" branch	○ * (don't care)
Select all that apply	**Datapath units:**	☐ Branch Comp	■ Imm. Gen	☐ Load Extend	
	RegFile:	☐ Read Reg[rs1]	☐ Read Reg[rs2]	■ Write Reg[rd]	

c) If we convert the above datapath to a 5-stage pipeline with **no forwarding**, what types of hazards (S=structural, D=data, C=control) exist **after** each line in the following code; how many `nops` must we add? (Assume a register can be written and read in the same cycle, and that the Branch Comp is in the EX stage.)

```
start: lw    t0, 0(a0)          Hazard (circle one): S D C None   # of nop _____2_____
```
`t0` being written to and read from the next op causes a data hazard and 2-cycle stall (or introduction of 2 NOPs so the W and R of the register file lines up -- thankfully we can read and write registers the same cycle)
```
       beq   t0, x0, end        Hazard (circle one): S D C None   # of nop _____2_____
```
We need to wait until after the EX stage to know whether `t0==x0` before we can load the correct next instruction, so that causes a control hazard and a 2-cycle stall (or introduction of 2 NOPs)
```
       addi  t0, t0, 2          Hazard (circle one): S D C None   # of nop _____2_____
```
`t0` being written to and read from the next op causes a data hazard and 2-cycle stall (or introduction of 2 NOPs so the W and R of the register file lines up -- thankfully we can read and write registers the same cycle)
```
       sw    t0, 0(a0)          Hazard (circle one): S D C None   # of nop _____________
end:
```

inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw t0 0(a0)	IF	ID a0	EX	MEM r	WB t0									
beq t0,x0,end		NOP	NOP	IF	ID t0	EX	MEM	WB						
addi t0, t0, 2					NOP	NOP	IF	ID t0	EX	MEM	WB t0			
sw t0, 0(a0)								NOP	NOP	IF	ID t0 a0	EX	MEM w	WB

F4) *What's that smell?! Oh, it's Potpourri...* (20 points=2 each, 30 minutes)

a) We build a small Internet-of-things device to measure dog body temperature and send it to a receiver. It will only send the following temperatures: {100.0, 100.1, 100.2, 100.4, 100.8, 101.6, 103.2, 106.4}, and any time the temperature is not those exact values, it'll send whatever value is the closest one. What encoding/decoding *scheme* would you use for these numbers and how many total *bits* would you need?
Scheme: ○Unsigned fixed point ○Bias fixed point ○2s complement fixed point ●Other
Bits: ●3 ○4 ○5 ○6 ○7 ○8 ○9 ○10 ○11 ○12 ○13 ○14 ○15 ○16

Other (just have a lookup table) using **3 bits** to choose from the 8 values

b) ○True ●False A `0/0` ALU operation would cause an *interrupt*, dealt with by the trap handler.

False, that's an exception

c) ○True ●False *DMA (Direct Memory Access)* is a form of Programmed I/O the CPU handles.

False, DMA obviates the need for Programmed I/O where the CPU does the work

d) ◯True ●False A *shared-based network* is another kind of parallelism; multiple nodes can talk to each other at the same time, "sharing" the network.

False, that's a switched network. A shared network is a "bus" that can only be driven by one source at a time

e) ◯First-fit ●Next-fit ◯Best-fit would make sense for *allocating blocks on a Flash (SSD) drive*.

Next-fit, since it needs to spreads the writes out to have even read wear

f) ◯Control ◯Datapath ●Memory ◯Input ◯Output causes the most headaches with multi-core.

Memory, with all the cache coherence issues we've seen

g) ◯True ●False Introducing *locks* in C (e.g., code below) cures the race condition bug with threads.

```
while (lock != 0) ; // spin-wait until the variable lock is released
// lock == 0 now (unlocked)
lock = 1; // set lock (locked)
// access shared resource ...
lock = 0; // release lock (unlocked)
```

False, it doesn't work in C, you need an assembly-level ATOMIC test-and-set mechanism

h) The code below was written to sum the numbers from **1** to **N** (always a positive number). Select ONE.

◯ It works, but it has to be run on a machine with 16 *physical* cores ◯ It works, but it has to be run on a machine with 16 *logical* cores ◯ It works, but only when **N** is bigger than the number of *physical* cores ◯ It works, but only when **N** is bigger than the number of *logical* cores ● It has a race condition bug ◯ It has a deadlock bug ◯ It always works	**int sumup(int N) {** **int THREADS = 16, TOTAL = 0, sum[THREADS];** **omp_set_THREADS(THREADS);** **for (int i=0;i<THREADS;i++) sum[i]=0;** **#pragma omp parallel** **{** **int id = omp_get_thread_num();** **for (int i=id;i<=N;i+=THREADS) sum[id] += i;** **TOTAL+=sum[id];** **}** **return TOTAL;}**

Race condition bug (**TOTAL** is read and incremented by multiple threads)

i) This: **main() { for(uint8_t i = 9; i >= 0; --i) printf("%u", i); }** causes… (select ONE)

◯ A compile error because the **printf** statement needs to be on a different line
◯ A compile error because the **printf** statement needs to be surrounded by curly brackets **{ }**
● An infinite loop
◯ Nothing to print out because there's no trailing **\n** (so the output doesn't get flushed)
◯ The numbers to print out like this: **987654321**
◯ The numbers to print out like this: **876543210**
◯ The numbers to print out like this: **9876543210**

Infinite loop (the unsigned number is never negative to break out of the loop)