CS 61C
Machine Structures

Final

Note (August 10, 2022): These are extremely rough drafts of rewritten solutions. They definitely contain errors and unfinished sections, but might have some useful parts for studying.

PRINT your name: ______________________, ______________________
(first) (last)

PRINT your student ID: ______________________

Read the following honor code and sign your name.

I understand that I may not collaborate with anyone else on this exam, or cheat in any way. I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in, at minimum, negative points on the exam and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: ______________________

You have 180 minutes. There are 10 questions of varying credit, for a total of 90 points.
For questions with **circular bubbles**, you may select only one choice.

- ○ Unselected option
- ● Only one selected option

For questions with **square checkboxes**, you may select one or more choices.

- ■ You can select
- ■ multiple squares

Anything you write that you ~~cross out~~ will not be graded. Anything you write outside the answer boxes will not be graded.

If an answer requires hex input, make sure you only use capitalized letters! For example, `0xDEAD BEEF` instead of `0xdeadbeef`. You will be graded incorrectly otherwise! Please always add the hex (`0x`) and binary (`0b`) prefix to your answers or you will receive 0 points. For all other bases, do not add the suffix or prefixes.

Do not add units unless the problem explicitly tells you to!

Question 1 ***Number Rep*** **(2 points)**

Q1.1 (2 points) Convert 52_8 to base 10. Leave the answer as a plain integer. DO NOT add the subscript indicating the base.

Solution: 42

Question 2 *Floating Point* (3 points)

Consider a 20-bit floating point number with the following components (1 sign bit, 6 exponent bits, 13 mantissa bits); i.e.,

SEEEEEEMMMMMMMMMMMMM

All other properties of IEEE754 apply (bias, denormalized numbers, ∞, NaNs, etc). The bias is the usual $-(2^{E-1}-1)$, which here would be $-(2^5-1)=-31$.

Q2.1 (3 points) What is the bit representation (in hex) of the floating-point number -8.25? Your answer must be in hex, must be prepended with `0x`, and all letters must be capitalized. Do not include any extra leading zeros.

Solution: `0xC4100`

Question 3 ***Potpourri*** **(16 points)**

Q3.1 (2 points) You have a program that spends some percentage of its time waiting for requests and the rest of the time performing calculations. Suppose you have 8 threads which you can use to parallelize calculations, with no overhead or non-parallelizable calculations. What is the maximum fraction of time that your sequential program can spend on waiting for requests if we would like to achieve at least 4 times speedup? Leave your answer as a single simplified fraction.

> **Solution:** $\frac{1}{7}$
>
> Amdahl's law says:
>
> Speedup $= \frac{1}{(1-F)+\frac{F}{S}}$
>
> where F is fraction of the code that can be sped up, and S is maximum possible speedup.
>
> F is what we want to solve for here (fraction of code that can be sequential = 1 - fraction of code that is parallelizable).
>
> S is 8 (we have 8 threads, so we can get 8x speedup).
>
> The desired total speedup is 4.
>
> Plugging all this in:
>
> $4 = \frac{1}{(1-F)+\frac{F}{8}}$
>
> Solving gives $F = \frac{6}{7}$. If 6/7 of the code can be parallelized, then 1/7 of the code is sequential.

The company you work at has a datacenter. Answer the following questions:

Q3.2 (2 points) The Mean Time Between Failures (MTBF) for this particular data center is 4000 hours. The Mean Time To Repair (MTTR) is 3 hours. What is the availability for this datacenter? Express your answer as a single simplified fraction.

> **Solution:** $\frac{3997}{4000}$
>
> MTBF only tells you how much time passes between failures; it doesn't account for repair time.
>
> Every 4000 hours, there is 1 failure. That failure takes 3 hours to repair. Therefore, the system is up for 3997 hours, for an availability of 3997/4000.

Q3.3 (2 points) Your company would like to restrict the annualized failure rate to be 1% for the individual machines in a large cluster. What does the Mean Time To Failure (MTTF) have to be to satisfy this annualized failure rate? Assume that the MTTF in this question is unrelated to that of Q3.2. Write down your answer in years.

> **Solution:** 100
>
> Intuitively, 1% of machines can fail every year, so any given machine should fail once every 100 years.

For each of the following functionalities, select the stage(s) in CALL for which the statement is true.

Q3.4 (2 points) Replaces pseudo instructions.

- ☐ Compiler
- ■ **Assembler**
- ☐ Linker
- ☐ Loader

Solution: TODO

Q3.5 (2 points) Calculate all absolute addresses.

- ☐ Compiler
- ☐ Assembler
- ■ **Linker**
- ☐ Loader

Solution: TODO

Q3.6 (2 points) Generates parse trees.

- ■ **Compiler**
- ☐ Assembler
- ☐ Linker
- ☐ Loader

Solution: TODO

Q3.7 (2 points) Constructs symbol table.

- ☐ Compiler
- ■ **Assembler**
- ☐ Linker
- ☐ Loader

Solution: TODO

```
A:
for (int i = 0; i < 5000 - 3; i += 3) {
    a[i+2] = a[i] + a[i+1];
}

B:
for (int i = 0; i < 5000 - 2; i += 2) {
    a[i+2] *= a[i];
}

C:
for (int i = 0; i < 5000 - 3; i++) {
    a[i+2] = a[i] + a[i+1];
}

D:
for (int i = 0; i < 5000; i++) {
    a[i] = 100;
}
```

Q3.8 (2 points) Which of the above code blocks would see a performance improvement if we placed a `#pragma omp for` over the outer for loop? Select all that apply.

■ **A**

☐ B

☐ C

■ **D**

☐ E. None of the above

Solution: Note: We think this question had a typo and meant to say `#pragma omp parallel for` instead of `#pragma omp for`.

In A and D, each iteration of the loop reads/writes to unrelated parts of memory. In A, the `i=0` iteration accesses `a[0]`, `a[1]`, and `a[2]`. Then the next iteration has `i=3` and accesses `a[3]`, `a[4]`, and `a[5]`. In D, each iteration has a different `i` and only writes to `a[i]`. Thus, multithreading the loop will cause performance improvement.

In B and C, multiple iterations of the loop need to access the same parts of memory. In B, the `i=0` iteration accesses `a[0]` and `a[2]`. Then the next iteration has `i=2` and accesses `a[2]` (also used by the previous iteration) and `a[4]`.

In C, the `i=0` iteration accesses `a[0]`, `a[1]`, and `a[2]`. Then the next iteration has `i=1` and accesses `a[1]` (also used by the previous iteration), `a[2]` (also used by the previous iteration), and `a[3]`.

Multithreading B and C will likely introduce data sharing issues (e.g. cache coherency, false sharing) that may actually slow the program down, or result in minimal or no performance improvement.

Question 4 *RISCV* **(15 points)**

Note: For the following question, you may NOT use Venus; indeed, the questions have been written so that using Venus is counterproductive.

Part 1: Mini coding questions

Implement the following functions in RISC-V 32b integer assembly. You may NOT use any extensions (eg: you cannot use `mul` and you can't use floating point or vector extensions). You may only write one line of RISC-V code per blank. Do NOT include commas.

`_mm32add_epu8`**:**

Inputs: `a0` and `a1` are vectors (NOT a pointer to the vector) of four 8-bit unsigned integers **less than 100**.

Output: `a0` returns a vector of four 8-bit unsigned numbers such that `result[i] = a0[i] + a1[i]` for `0<=i<4`.

Example: If the inputs `a0` and `a1` were `[15,2,3,99]` and `[5,6,7,99]` (encoded as `0x0F020363` and `0x05060763` respectively), the expected output would be `[20, 8, 10, 198]`.

```
_mm32add_epu8:
___________ # Code line 1
ret
```

Q4.1 (3 points) Code line 1:

Solution: `add a0 a0 a1`

`floatlessthan`**:**

Inputs: `a0` and `a1` are **positive** non-NaN IEEE-754 32-bit floating point numbers.

Output: `a0` returns `1` if `a0 < a1` and `0` otherwise.

Example: If the inputs `a0` and `a1` were 1.5 and 1.75, the expected output would be `1`. If `a0` and `a1` were 1.5 and 1.5 respectively, the expected output would be `0`.

```
floatlessthan:
___________ # Code line 2
ret
```

Q4.2 (3 points) Code line 2:

Solution: `slt a0 a0 a1` or `sltu a0 a0 a1`

`skipline`:

Inputs: None

Output: None

Effect: Skips the next assembly instruction in the caller function. We are only using the 32b RISC-V ISA (no 16 bit extension). You may assume that the next line of code exists, and is not a pseudoinstruction.

Exmple: Assume that the following code was run:

```
addi t0 x0 5
jal skipline
addi t0 t0 300
addi t0 t0 10
```

Then at the conclusion of this code, `t0` would equal to 15.

```
skipline:
___________ # Code line 3
ret
```

Q4.3 (3 points) Code line 3:

> **Solution:** `addi ra ra 4`
>
> Remember that `ra` stores the address of the instruction we should execute after the function returns. Adding 4 to `ra` causes us to return to the next instruction, skipping one instruction.

`endofstring`:

Inputs: `a0` is a pointer to a **nonempty** string.

Output: `a0` returns a pointer immediately after the end of the string.

Example: Let `a0` be the pointer `0x10000000` to string `s`, which is the string "`Hello`". Then the expected output would be `0x10000006`.

```
endofstring:
___t0______ # Code line 4.
            # You must use t0 as the first register of this instruction.
___________ # Code line 5
___________ # Code line 6
ret
```

Q4.4 (1 point) Code line 4:

Solution: `lb t0 0(a0)`

High-level idea: Iterate through the characters of the string until you encounter a null byte (which denotes the end of the string). Return the address of the null byte, plus 1 (since we want the address immediately after the end of the string).

The string is stored in memory (since we have a pointer to the string). For each character in the string, we need to load the character from memory into a register, where we can check if that character is the null terminator or not.

Note that a character is 1 byte long, so we use load-byte here, not load-word.

Q4.5 (1 point) Code line 5:

Solution: `addi a0 a0 1`

Here, we add 1 to the address of the string to move to the next character in the string. Note that we need to add 1 to the address before checking if the character is null, so that we can get the address immediately after the null byte.

Q4.6 (1 point) Code line 6:

Solution: `bneq t0 x0 endofstring` OR `bneq x0 t0 endofstring` OR `bnez t0 endofstring`

If the character loaded was not the null terminator (zero), then repeat all the steps above. Otherwise, we already have the address of the null terminator plus 1 in `a0` (the register where the return value should be), so we can return.

Part 2: RISC-V translation

As a reminder, you many NOT use Venus for this question. As a reminder, hexadecimal strings should be written with the "`0x`" prefix, with CAPITALIZED hex digits (ex. `0xDEADBEEF`).

Q4.7 (3 points) Translate the following instruction to hexadecimal: `srai t0 s3 16`. Remember to include the "`0x`" at the beginning!

Solution: `0x4109D293`

Opcode: `0b001 0011`

Funct3: `0b101`

Funct7: `0b010 0000`

Instruction type: I* (Note that I* is a 61C-specific instruction type. In official RISC-V documentation, `srai` is listed as an I-type instruction, but we use I* to denote the bit-shift I instructions, which use the top 7 bits of the immediate for the funct7 and only have a 5-bit immmediate.)

rs1: `s3 = x19 = 0b10011`

rd: `t0 = x5 = 0b00101`

Immediate: 16. I* type instructions use a 5-bit unsigned immediate, where 16 is `0b10000`.

Assembling together the I* instruction format: `funct7 imm rs1 funct3 rd opcode`

`0b010 0000 10000 10011 101 00101 001 0011`

Regrouping bits:

`0b0100 0001 0000 1001 1101 0010 1001 0011`

Converting to hex:

`0x4109D293`

Question 5 *Boolean Algebra* (2 points)

Q5.1 (2 points) Select all of the following which are equivalent to $\overline{A} + \overline{B}C$.

■ $A\overline{A} + \overline{(ABC + \overline{C}A)}$

■ $\overline{A(\overline{\overline{B}C + CB\overline{B}}) + \overline{C}A}$

□ $A\overline{B} + \overline{C}$

□ $\overline{C\overline{\overline{B}A} + C(\overline{CA} + BC)}$

Solution: Since you're given options to compare, and the truth tables only have 8 rows in them, one way to solve this question may be to write out 5 truth tables (one per expression) and compare them.

Another way to do this is to perform Boolean algebra and simplify the terms as much as possible to see the equality:

1) $A\overline{A} + \overline{(ABC + \overline{C}A)}$

$0 + \overline{(A(BC + \overline{C}))}$

$\overline{A(B + \overline{C})}$

$\overline{A} + \overline{B}C$

2) $\overline{A(\overline{\overline{B}C + CB\overline{B}}) + \overline{C}A}$

$\overline{A(\overline{\overline{B}C + 0}) + \overline{C}A}$

$\overline{A(B + \overline{C}) + \overline{C}A}$

$\overline{AB + A\overline{C} + \overline{C}A}$

$\overline{AB + A\overline{C}}$

$\overline{A(B + \overline{C})}$

$\overline{A} + \overline{B}C$

3) $A\overline{B} + \overline{C} \neq \overline{A} + \overline{B}C$

4) $\overline{C\overline{\overline{B}A} + C(\overline{CA} + BC)}$

$\overline{C(B + \overline{A}) + C(\overline{C} + \overline{A}) + BC)}$

$\overline{CB + \overline{A}C + C\overline{C} + C\overline{A} + BC}$

$\overline{CB + \overline{A}C + 0 + \overline{A}C + CB}$

$\overline{CB + \overline{A}C}$

$\overline{C(\overline{A} + B)}$

$\overline{C} + A\overline{B} \neq \overline{A} + \overline{B}C$

Question 6 ***C Structures*** **(13 points)**

Part 1: The Structure of Structures

Assume a 32-bit architecture with RISC-V alignment rules:

Consider the following structure definition and code:

```
struct foo {
    char a;
    uint16_t b;
    char *c;
    struct foo *d;
}
```

Q6.1 (2 points) What is `sizeof(struct foo)` (Answer as an integer, with no units)?

Solution: 12

This question (and solution) uses the alignment rule where an n-byte struct field must start at an n-byte-aligned boundary. For example, `char *c` is a 4-byte field, so it must start at a 4-byte-aligned boundary (the offset of `c` relative to the start of the struct must be a multiple of 4).

`char a` takes up 1 byte (byte 0).

The next byte (byte 1) is padding so that `b` can start at a 2-byte-aligned boundary.

`uint16_t b` takes up 2 bytes (bytes 2-3).

`char *c` is a pointer, so it takes up 4 bytes in a 32-bit system (bytes 4-7).

`struct foo *d` is a pointer, so it takes up 4 bytes in a 32-bit system (bytes 8-11).

In total: $1 + 1 + 2 + 4 + 4 = 12$.

Q6.2 (2 points) If `b` and `c` are swapped, this increases the size of the structure:

● **True** ○ False

Solution: `char a` takes up 1 byte (byte 0).

The next 3 bytes (byte 1-3) are padding so that `c` can start at a 4-byte-aligned boundary.

`char *c` is a pointer, so it takes up 4 bytes in a 32-bit system (bytes 4-7).

`uint16_t b` takes up 2 bytes (bytes 8-9).

The next 2 bytes (bytes 10-11) are padding so that `d` can start at a 4-byte-aligned boundary.

`struct foo *d` is a pointer, so it takes up 4 bytes in a 32-bit system (bytes 12-15).

In total: $1 + 3 + 4 + 2 + 2 + 4 = 16$.

Part 2: Skipping Around

Consider the following code:

```
struct SLN{
   void *data;
   struct SLN **next;
}

/* Only the following in the code actually matters:
    cmp(find, sl->next[level]->data))
    But if you are curious, look up "Skiplist"
    */

int SL_find(void *find, int level, SkipListNode *sl,
                (int)(*cmp)(void *, void *){
    if(cmp(find, sl->data) == 0) return 1;
    if(level == 0 && sl->next[level] == null) return 0;
    if(sl->next[level] == null) {
        return SL_find(find, level-1, sl, cmp);
    }
    if(cmp(find, sl->next[level]->data)) > 0)){
        return SL_find(find, level-1, sl, cmp);
    }
    return SL_find(find, level, sl->next[level], cmp);
}
```

In translating the code `cmp(find, sl->next[level]->data))` into RISC-V we need to store arguments on the stack. So we will have `find` at `sp(0)`, `level` at `sp(4)`, `sl` at `sp(8)` and `cmp` at `sp(12)`. We're going to break up the translation into pieces.

Q6.3 (1 point) Load `find` into `a0`

> **Solution:** `lw a0 0(sp)`
> The question states that `find` is at `sp(0)`. In other words, the register `sp` holds an address (the address of bottom of the stack). This address, plus 0, is the address of `find` in memory. Note that `void *find` is a pointer, so its size is 1 word. We can load it into memory with the load-word instruction.

Q6.4 (1 point) The next thing we need to do is get `sl->next[level]->data` into `a1`. The RISC-V code for that would be (fill in based on comments):

Load `sl` into `t0`

> **Solution:** `lw t0 8(sp)`
> The question states that `sl` is at `sp(8)`. In other words, the register `sp` holds an address, and when we add 8 to this address, we get the address of `sl` in memory. Note that `SkipListNode *sl` is a pointer, so its size is 1 word. We can load it into memory with the load-word instruction.

Q6.5 (1 point) Load `level` into `t1`

Solution: `lw t1 4(sp)`

Similar logic as the previous parts. The question states that `level` is at `sp(4)`, and `int level` is a 4-byte (1 word = 4 bytes) integer.

Q6.6 (1 point) Load `sl->next` into `t0`

Solution: `lw t0 4(t0)`

`SkipListNode *sl` is currently in `t0`. Note that `sl->next` does two things; it dereferences the `sl` pointer to get the `SkipListNode` struct, then accesses the `next` field in the struct.

Looking at the struct definition, we note that `void *data` takes up the first 4 bytes of the struct in memory, and `struct SLN **next` takes up the next 4 bytes of the struct in memory. Since we want the `next` field, we want bytes 4-7 of the struct in memory; in other words, we want to load 4 bytes (1 word) of memory, starting at the address of the struct (`sl`, which is in `t0`), with an offset of 4 (to get the `next` field).

Q6.7 (2 points) Load `sl->next[level]` into `t0`

Solution:

```
slli t1 t1 2
add t0 t0 t1
lw t0 0(t0)
```

`s1->next` is a variable of type `struct SLN **`. In other words, it's a pointer to an array of `struct SLN *` (an array of pointers). Its value is in `t0` from the previous subpart.

We want to access `s1->next[level]`, the `level`th element in the array.

`slli t1 t1 2`: Each element in the array is a pointer, which is 4 bytes long. To find the address of the `level`th element, we need to first multiply `level` by 4 to find out the byte offset of the `level`th element. (In other words, this number tells us how many bytes after the start of the array we can expect to find the `level`th element.)

`add t0 t0 t1`: We add the byte offset we found in the previous instruction (`t1`) to the address of the start of the array (`t0`), putting the result in `t0`. Now `t0` contains the address of the `level`th element of the array.

`lw t0 0(t0)`: Finally, we don't want the address of the `level`th element, we want the actual element (pointer) in the array. We can get the value in the array in memory with a load-word instruction. (Note that in C, the bracket syntax `s1->next[level]` is actually shorthand for `*(s1->next + level)`, so this last load-word step is performing the dereference operation.

Q6.8 (1 point) Load `data` into `a0`

Solution: `lw a0 0(t0)`

From the previous subpart, `t0` now has `s1->next[level]`, which is a `struct SLN *` (a pointer). We want to dereference this pointer struct and access its `data` field.

The idea here is similar to when we were given `sl`, which was also a `struct SLN *`, and wanted to find `sl->next`. In this case, we want the `data` field, which is the first 4 bytes in the struct, so we should dereference the pointer to the struct and load a word with an offset of 0.

Q6.9 (2 points) Finally, how do we call `cmp` (using `t0` as a temporary)

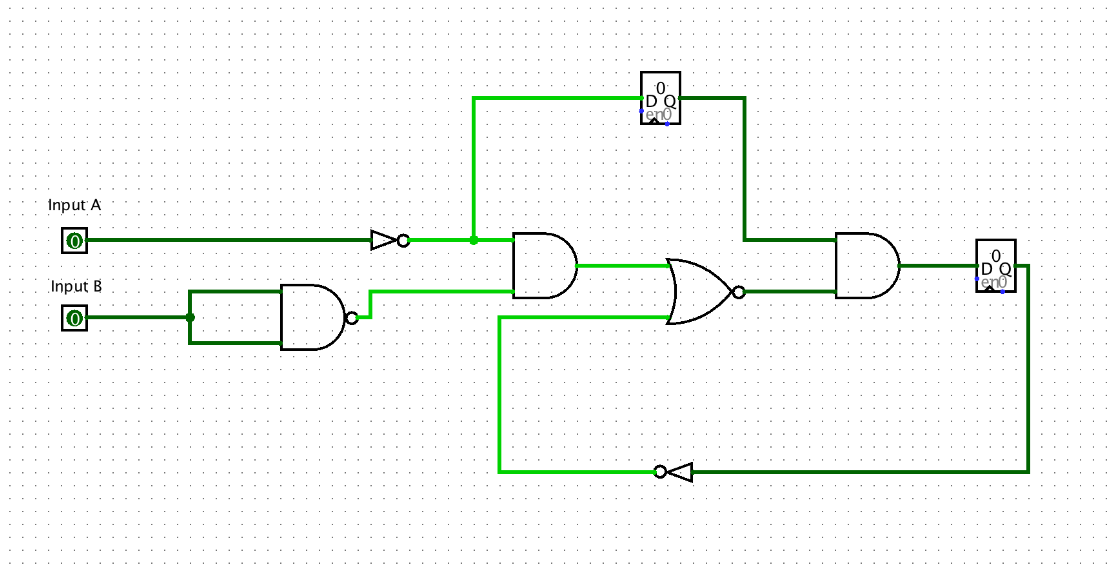
Solution:

```
lw t0 12(sp)
jalr ra t0
```

The question states that `cmp` is at `sp(12)`. `cmp` is a function pointer (the address of some instructions in the code section of memory), so we can put this address in `t0`. Then, we can use the `jalr` instruction, which jumps to the address in a register. In this case, we're jumping to the address in `t0` and remembering the return address in `ra`.

Note that we're using `jalr` because the address we want to jump to is in a register. We would use `jal` if the address we want to jump to was a line of code with a label.

Question 7 ***SDS*** **(6 points)**



State Machine

Assume input A and input B come from registers. Please include `ns` in your answer.

Assume all 2-input logical gates have a 10 ns propagation delay. The NOT gate has a 5 ns delay. All registers have a clk-to-q of 15 ns and setup time of 20 ns.

Q7.1 (2 points) Find the minimum clock period to ensure the validity of the circuit.

> **Solution:** 75 ns
>
> We have the following paths:
>
> - Input A (clock-to-q) → NOT → Register (setup) = 15 ns + 5 ns + 20 ns = 40 ns
> - Input A (clock-to-q) → NOT → AND → NOR → AND → Register (setup) = 15 ns + 5 ns + 10 ns + 10 ns + 10 ns + 20 ns = 70 ns
> - Input B (clock-to-q) → NAND → AND → NOR → AND → Register (setup) = 15 ns + 10 ns + 10 ns + 10 ns + 10 ns + 20 ns = 75 ns
> - Register (clock-to-q) → NOT → NOR → AND → Register (setup) = 15 ns + 5 ns + 10 ns + 10 ns + 20 ns = 60 ns
>
> So we need the max of them which would be 75 ns.

Q7.2 (2 points) Find the maximum hold time such that there are no hold time violations.

Solution: 20 ns

For the maximum hold time, we need to look at the same paths to see what would be the shortest path to get to the register:

- Input A (clock-to-q) → NOT → Register (NO setup) = 15 ns + 5 ns = 20 ns
- Input A (clock-to-q) → NOT → AND → NOR → AND → Register (NO setup) = 15 ns + 5 ns + 10 ns + 10 ns + 10 ns = 50 ns
- Input B (clock-to-q) → NAND → AND → NOR → AND → Register (NO setup) = 15 ns + 10 ns + 10 ns + 10 ns + 10 ns = 55 ns
- Register (clock-to-q) → NOT → NOR → AND → Register (NO setup) = 15 ns + 5 ns + 10 ns + 10 ns = 40 ns

For this one, when we get to the register, we do NOT want to include the setup time as we want to see what is the shortest time to get to a register. This means we take a min of the above paths (which does NOT include the setup) which would be 20 ns.

Q7.3 (2 points) Select the different ways you can decrease the critical path (ignore wire delay)?

- ☐ Move components closer together
- ☐ Use transistors instead of AND, OR, and NOT gates
- ☐ Use AND and OR gates instead of NAND and NOR gates
- ■ **Add pipeline registers**
- ■ **Simplify circuit (boolean) logic**
- ☐ Make the clock speed faster

Solution: Move components closer together: False. Since we're ignoring wire delay, the distance between components does not affect the critical path delay.

Use transistors instead of AND, OR, and NOT gates: False. Logic gates are built out of transistors, so it doesn't make sense to replace logic gates with transistors.

Use AND and OR gates instead of NAND and NOR gates: False. In order to preserve the same logic, we would have to replace the NAND gate with two gates (AND gate, plus a NOT gate), which would result in additional delay.

Add pipeline registers: True. Splitting the computation into multiple stages means that the critical path is now the longest path through one stage (between pipeline registers), instead of through the entire circuit. Another way to think of this one is that adding additional registers shortens the longest path between any two registers.

Simplify boolean logic: True. Simplifying the circuit logic can result in fewer gates being used, which would reduce the critical path length.

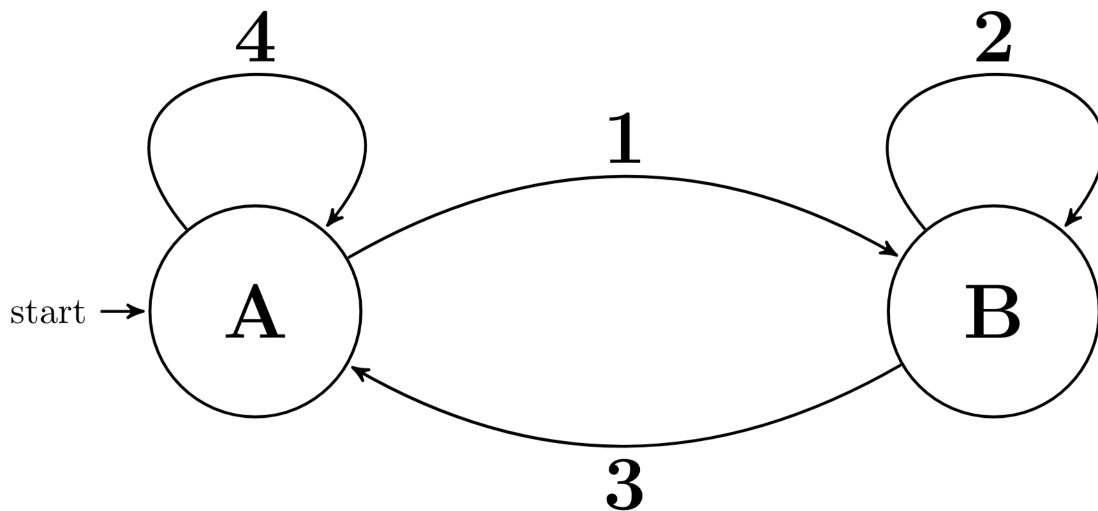
Make the clock speed faster: False. Shortening the time between rising edges of the clocks does not affect the length of the longest path. (In fact, shortening the clock period too much may actually cause design violations like setup time or hold time violations.)

Question 8 ***FSM*** **(8 points)**

We want to construct a finite-state machine that determines if adding together two binary unsigned numbers causes an overflow. The machine consumes two bit strings of equal length, starting from their least significant bits. After consuming each pair of bits from the two inputs, the machine outputs `1` if addition of the two strings (as seen so far) would cause an overflow, or a `0` otherwise.

For example, if the machine consumes the two bit strings `0111` and `0100`, the sequence of output values will be `0`, `0`, `1`, `0`.

A diagram for this finite-state machine is shown below. We have given the states generic placeholder names. Transition labels use the notation `x,y/o`, where `x` is a bit read from the first input string, `y` is a bit read from the second input string, and `o` is the output. To simplify, you are allowed to assign multiple labels to the same arrow in the diagram—each label corresponds to a different transition with the same start and end states.



Q8.1 (2 points) Which of the following labels (each corresponding to one transition) should be assigned to **Arrow 1** in the diagram?

- ☐ `0,0/0`
- ☐ `0,0/1`
- ☐ `0,1/0`
- ☐ `0,1/1`
- ☐ `1,0/0`
- ☐ `1,0/1`
- ☐ `1,1/0`
- ■ `1,1/1`

Solution: Intuitively, one of these states corresponds to the carry-in bit currently being 1, and the other state corresponds to the carry-in bit currently being 0. We should start with the carry-in bit being 0, so A is the state where the carry-in bit is 0, and B is the state where the carry-in bit is 1.

Moving from A to B should happen when we trigger an overflow and make the carry-in bit 1. If the current carry-in bit is 0, this can only happen if our two inputs are 1, giving us $1 + 1 + 1$. In this case, we output 1 because we caused an overflow.

Q8.2 (2 points) Which of the following labels (each corresponding to one transition) should be assigned to **Arrow 2** in the diagram?

- ☐ `0,0/0`
- ☐ `0,0/1`
- ☐ `0,1/0`
- ■ `0,1/1`
- ☐ `1,0/0`
- ■ `1,0/1`
- ☐ `1,1/0`
- ■ `1,1/1`

Solution: Moving from B to B should happen when we trigger an overflow and make the carry-in bit 1. The carry-in bit is already 1, so overflow will happen if our input is `0,1` (creating $1 + 0 + 1$) or `1,0` (creating $1 + 0 + 1$) or `1,1` (creating $1 + 1 + 1$). In this case, we output 1 because we caused an overflow.

Q8.3 (2 points) Which of the following labels (each corresponding to one transition) should be assigned to **Arrow 3** in the diagram?

- ■ `0,0/0`
- ☐ `0,0/1`
- ☐ `0,1/0`
- ☐ `0,1/1`
- ☐ `1,0/0`
- ☐ `1,0/1`
- ☐ `1,1/0`
- ☐ `1,1/1`

Solution: Moving from B to A should happen when we do not trigger an overflow and make the carry-in bit 0. The carry-in bit is already 1, so overflow does not occur only if our input is `0,0` (creating $0+0+1$). In this case, we output 0 since we did not create an overflow.

Q8.4 (2 points) Which of the following labels (each corresponding to one transition) should be assigned to **Arrow 4** in the diagram?

- ■ `0,0/0`
- ☐ `0,0/1`
- ■ `0,1/0`
- ☐ `0,1/1`
- ■ `1,0/0`
- ☐ `1,0/1`
- ☐ `1,1/0`
- ☐ `1,1/1`

Solution: Moving from A to A should happen when we do not trigger an overflow and the carry-in bit stays 0. The carry-in bit is currently 0, so overflow does not occur if our input is `0,0` (creating $0 + 0 + 0$) or `0,1` (creating $0 + 1 + 0$) or `1,0` (creating $1 + 0 + 0$). In this case, we output 0 since we did not create an overflow.

Question 9 *Datapath* (11 points)

Part 1: New Instruction

Given the standard RISC-V datapath (Figure 1), determine if the following is implementable or not without any additional functional units? Assume the instruction is not a pseudoinstruction encoding.

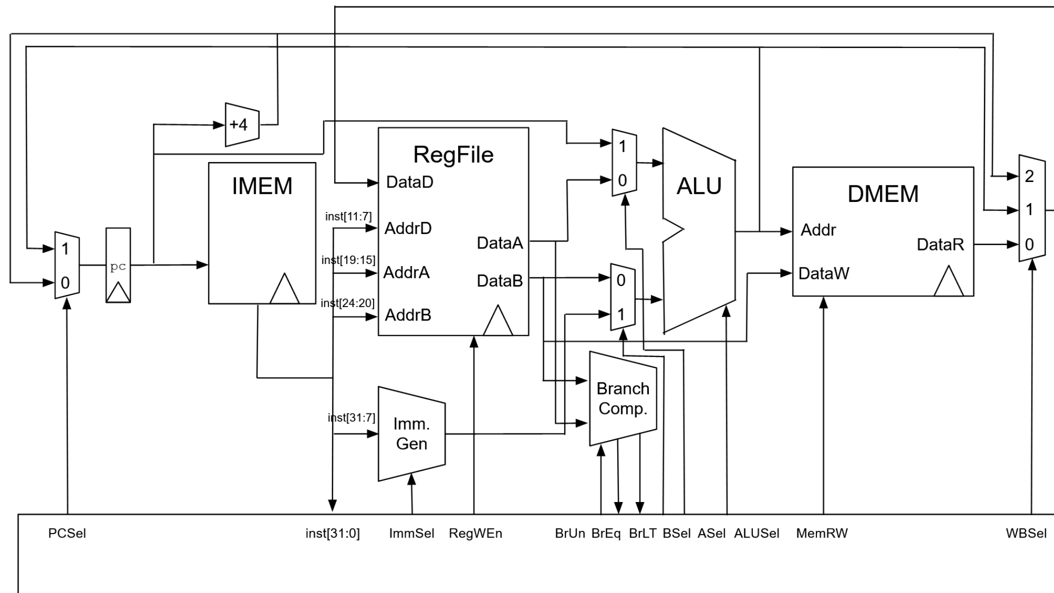


Figure 1

Q9.1 (1 point) `is_null rd, rs1`: checks if an input given through `rs1` is considered `NULL` or not by C standard. The result is returned through `rd` as a bit.

● **Implementable** ○ Not implementable

Solution: Regfile can support this instruction, since we're only reading from one register and writing to one register. (Recall that Regfile supports reading up to two registers and writing to one register.)

We can use the branch comparator to compare the value in `rs1` to 0.

Reading/writing to registers and comparing the values in the registers are the only operations this instruction needs to do, so we can implement this instruction without adding hardware.

Q9.2 (4 points) What changes would you need to make in order for the instruction to be able to execute correctly? Assume all modifications and additions are done on top of the existing single cycle datapath. Select all that apply.

- ☐ Modify ALU buses.
- ☐ Modify control logic for ALU/ALUSel.
- ■ **Modify the control logic for the Branch Comparator.**
- ■ **Modify the control logic for parsing `instr[31:0]`.**
- ■ **Modify the control logic for `WBSel`.**
- ☐ Modify Branch Comparator logic.
- ☐ Modify the control signals to the ALU.
- ☐ Add an additional comparator.
- ☐ Add additional control signals for the writeback mux.
- ☐ None of the above.

Solution: TODO double-check this one –Peyrin

Modify ALU buses: False. We don't need to change the wires being sent to ALU, because the first input (value of `rs1`) is something we can already send into the ALU, and we can use the immediate generator to send the second value (zero) into the ALU.

Modify control logic for ALU/ALUSel: False. We aren't using the ALU to compute anything (the comparison is happening in the branch comparator).

Modify the control logic for the Branch Comparator: True.

Modify the control logic for parsing `instr[31:0]`: True.

Modify the control logic for `WBSel`: True.

Modify Branch Comparator logic: False.

Modify the control signals to the ALU: False. We aren't using the ALU to compute anything.

Add an additional comparator: False. The branch comparator is enough to perform the comparison we need.

Add additional control signals for the writeback mux: False. `WBSel`

Q9.3 (2 points) `is_null rd, rs1` is not in a standard RISC-V instruction format; as we're attempting to reduce the number of hardware changes in our datapath. We instead choose to implement our instruction as a pseudoinstruction in the following format. Which of the following statements is true? Assume earlier changes propagate. Select all that apply.

Format: `R-type` Instruction

- ○ We need to wire `x0` as a comparator for all branch comparisons.
- ○ We need to provide a second argument `x0` when calling the instruction and modify the control signals.
- ● **We need to wire `x0` as `rs2` and modify the control signals.**
- ○ We need to provide a second argument `x0` as a comparator for all branch comparisons.
- ○ It is impossible to represent as an R-Type instruction.

Part 2: Cached Datapath - One Cache

We now choose to add caching to the single cycle datapath. We add a single single level cache that is shared between instruction and data memory and is always accessed prior to a direct memory access.

Q9.4 (0 points) As we observe our datapath, we see that having heavy caching in addition to our respective memory arrays does help our average access times in comparison to having no caching, as is the expected behaviour. However, in our worst-case scenarios, we see that the time taken is suboptimal. We instead propose an alternative method such that our memory arrays (IMEM and DMEM) are replaced by caches. We have one shared cache such that IMEM and DMEM accesses both go through Cache A. What modifications need to be made to the control logic signals? Give your answer in 10 words or fewer.

Note: this question was dropped.

Solution: Generally speaking, in order to handle caching replacing memory, there are a few major control logic parts to keep in mind. The write-to-cache signal has to be specified, as well as what happens when a cache line is dirty. The answer also should've specified some version of a TIO breakdown in order to index and locate data in the cache. We generally assume all memory we need is in IMEM/DMEM when that's part of our datapath, but with caches, we actively know that it's not the case so some mention of notifying control on a miss such that either a main memory or disk access is needed is the last major component. Adding caches to the datapath in reality is more complicated but these are the main components needed at a broad level.

Q9.5 (0 points) How does missing in the cache affect how our datapath operates? Frame your answer in terms of how it affects the control signals and hazards in 2 or fewer sentences.

Note: this question was dropped.

Solution: A few key points that should've been brought up were that the control logic will have to handle fetching data on a miss and subsequently updating the cache. In addition, data hazards due to a miss will take longer to resolve and cause more cycles to be taken. The control logic will also have to handle properly evicting both instructions and data from the cache. You could've also mentioned that hazards due to various types of misses could've occurred, as instruction and data memory share the cache and thus can interfere with accesses; think of new instruction blocks being accessed and kicking out a data block that would be reused later (it's similar to the concept of coherency misses).

Q9.6 (2 points) Which of the following is always true about our caching setup?

- ○ The AMAT for instruction accesses will be different from data accesses since they reside in different portions of main memory.
- ○ Assembly with few control-flow instructions will cause a high hit rate for instruction accesses.
- ● **Instuction accesses will affect data access hit rates.**
- ○ It is possible for the cache to be entirely filled with data blocks.
- ○ None of the above.

Solution: If we're using the same cache to cache both instructions and data, then accessing instructions may cause data to be kicked out of the cache.

The AMAT for instruction accesses will be different from data accesses since they reside in different portions of main memory: This is false because we usually don't consider access time for different sections of memory to take different amounts of time.

Assembly with few control-flow instructions will cause a high hit rate for instruction accesses: This is false, because fetching data may still cause instructions to be kicked out of the cache.

It is possible for the cache to be entirely filled with data blocks: This is false because there will always be at least one instruction cached (the most recent one fetched).

Part 3: Cached Datapath - Two Caches

We now choose to add caching to the single cycle datapath. We add 2 single level caches; Cache A is accessed only on instruction fetch and Cache B is accessed only on data memory access prior to direct memory accesses.

Q9.7 (0 points) As we observe our datapath, we see that having heavy caching in addition to our respective memory arrays does help our average access times in comparison to having no caching, as is the expected behaviour. However, in our worst-case scenarios, we see that the time taken is suboptimal. We instead propose an alternative method such that our memory arrays (IMEM and DMEM) are replaced by identical caches. Cache A replaces IMEM and Cache B replaces DMEM. What modifications need to be made to the control logic signals? Give your answer in 10 words or fewer.

Note: this question was dropped.

Solution: Generally speaking, in order to handle caching replacing memory, there are a few major control logic parts to keep in mind. The write-to-cache signal has to be specified, as well as what happens when a cache line is dirty. The answer also should've specified some version of a TIO breakdown in order to index and locate data in the cache. We generally assume all memory we need is in IMEM/DMEM when that's part of our datapath, but with caches, we actively know that it's not the case so some mention of notifying control on a miss such that either a main memory or disk access is needed is the last major component. Adding caches to the datapath in reality is more complicated but these are the main components needed at a broad level. You could've also mentioned that individual logic would be needed to handle which control signals address which cache but that the control logic itself could be reused.

Q9.8 (0 points) How does missing in the cache affect how our datapath operates? Frame your answer in terms of how it affects the control signals and hazards in 2 or fewer sentences.

Note: this question was dropped.

Solution: A few key points that should've been brought up were that the control logic will have to handle fetching data on a miss and subsequently updating the cache. In addition, data hazards due to a miss will take longer to resolve and cause more cycles to be taken. The control logic will also have to handle properly evicting both instructions and data from the cache.

(Question 9 continued. . .)

Q9.9 (2 points) Which of the following is always true about our caching setup?

- ○ The AMAT for Cache A and Cache B will be the same since the caches are identical.
- ○ Intended IMEM accesses will cause cache incoherence with intended DMEM accesses if the memory addresses are close.
- ● **Assembly with few control-flow instructions will cause a high hit rate for Cache A.**
- ○ Instuction and data accesses will cause both Cache A and Cache B's states to change for every access.
- ○ None of the above.

Solution: Few control-flow instructions would mean the program has less branch/jump instructions which would limit the number of times the program will be accessing different points in the instruction memory. A program running on straight-line execution will only encounter compulsory misses at the start of the cache block, then hit for the rest of the instructions within the same block. Jumping around to different points in the instruction memory can lead to conflict misses which decreases hit rate.

AMAT cannot be guaranteed identical because we don't know the instruction and data access patterns of the program (different hit/miss rates will translate to different AMAT)

Cache incoherence will happen if both IMEM and DMEM caches hold the same data. If memory addresses are close, incoherence is not guaranteed to happen.

Cache A and Cache B states is not guaranteed to change states at every access, unless they are both storing data at the same memory address.

Question 10 ***Virtual Memory Caching*** **(14 points)**

Assume all systems are 32-bit.

We're given a system with an 4-way set associative cache of size 512 KiB with 128 B blocks. How many bits are allocated to the tag, index, and offset bits respectively?

Q10.1 (0.6 point) How many bits are allocated to the **tag**?

> **Solution:** 15. 32 - 10 - 7 = 15 bits

Q10.2 (0.6 point) How many bits are allocated to the **index**?

> **Solution:** 10. $2^{19}/(4 \cdot 128) = 10$ bits.

Q10.3 (0.8 point) How many bits are allocated to the **offset**?

> **Solution:** 7. Blocks are 2^7 bits large.

We run the following code with our caching setup unchanged from the previous question. What is the hit rate of the following lines of code?

```
#define base_arr_addr 0x12345678
#define ARR_SIZE 4096
#define I_BOUNDARY 2048
#define J_BOUNDARY 4096
#define I_STRIDE 128
#define J_STRIDE 64
int arr[ARR_SIZE];
uint32_t dummy_func(void) {
    //Loop 1
    for (int i = 0; i < I_BOUNDARY; i += I_STRIDE) {
        for (int j = 0; j < J_BOUNDARY; j += J_STRIDE) {
            arr[i] = arr[i] + arr[j];
        }
    }
    //Loop 3
    for (int j = 0; j < J_BOUNDARY; j += J_STRIDE) {
        for (int i = 0; i < I_BOUNDARY; i += I_STRIDE) {
            arr[j] = arr[j] * arr[i];
        }
    }
}
```

Q10.4 (2 points) `arr[i] = arr[i] + arr[j]`

Solution: $\frac{47}{48}$

Ther outer loop executes 16 times and the inner loop per round executes 64 times. Because the accesses only happen in the inner loop, we can tally the HR for that first and then see how the outer accesses affect the HR. For the first inner iteration, we see we have a miss on the `arr[0]` read, then a hit on `arr[1]` and on the `arr[0]` write. For the next iteration, we get a `arr[0]` read and write. The next `arr[j]` is where accesses become tricky; we're stepping by $64 \cdot 4B = 256B$; this is larger than one block which means every subsequent `arr[j]` access will be a miss for one outer iteration. Thus, for the first outer iteration, our overall HR is $\frac{2}{3}$. On the next outer iteration however, we notice that the entirety of the array can fit in the cache without conflicts. Because our `I_STRIDE` is larger than `J_STRIDE`, we know every future `arr[i]` access will have already had a compulsory miss by `arr[j]` in previous iterations and because nothing is evicted, we have a HR of 1 for all future access. This gives us a total of: $\text{HR} = \frac{2}{3} \cdot \frac{1}{16} + \frac{3}{3} \cdot \frac{15}{16} = \frac{47}{48}$. NOTE: the fact our array is 4-way does not affect us here because despite our 128B jumps in accesses, we have 2^{11} sets available to us which means we won't fill up the first way in each set before we fill the second way. Because we only have 2 nested loops with two access patterns, nothing will get kicked out in each set for another access pattern.

Q10.5 (2 points) `arr[j] = arr[j] * arr[i]`

Solution: 1

The entire array can fit in memory. Because the access pattern is effectively the same, all accesses are hits.

NOTE: for all parts, assume changes propagate unless otherwise stated.

As we're working on running the code snippet, we realise we want to run different instances of the same code. We choose to employ virtual memory on our memory space. We have 4 GiB of virtual memory and 16 MiB of physical memory mapped with a single level page table with a page size of 4 KiB. We choose to store 8 bits of metadata with each page table entry.

Q10.6 (2 points) After running one iteration of the inner loop for the code given in line, how many physical pages will our page table take up?

Solution: 1024

First, work out the length of the addresses, page numbers, and offsets. Each page is 4 KiB = 2^{12} bytes, so the offset is 12 bits. We have 4 GiB = 2^{32} bytes of virtual memory, so a virtual address is 32 bits. This leaves $32-12=20$ bits for the VPN. We have 16 MiB = 2^{24} bytes of physical memory, so a physical address is 24 bits. This leaves $24-12=12$ bits for the PPN.

Remember that each PTE needs to store the PPN, and optionally, some metadata. The PPN is 12 bits, and there are 8 bits of metadata for a total of 20 bits per entry. Assuming a word-aligned system, we round up to the nearest multiple of 4 bytes, which is 32 bits = 4 bytes per PTE.

The VPN has 20 bits, so there are 2^{20} virtual pages. The page table maps each virtual page to a physical page, so there are 2^{20} PTEs. Each PTE is 4 bytes, so the page table takes up $2^{20} \times 4 = 2^{22}$ bytes in total.

Each physical page is 2^{12} bytes, so the page table takes up $2^{22}/2^{12} = 2^{10} = 1024$ physical pages in memory.

NOTE: because we did not specify alignment for the system, if you solved for a byte-aligned system, you will get the points for 768 pages.

Alternate solution: Assuming a byte-aligned system, we round up to the nearest byte, which is 24 bits = 3 bytes per PTE.

As before, there are 2^{20} PTEs, but now, each PTE is 3 bytes, so the page table takes up 3×2^{20} bytes in total.

Each physical page is 2^{12} bytes, so the page table takes up $3 \times 2^{20}/2^{12} = 3 \times 2^{8} = 768$ physical pages in memory.

Q10.7 (2 points) If our caching system remains as seen in Q10.6, how many caches would be needed to fully fit our page table? Give your answer as a decimal to two decimal places.

Solution: 8

Alternate solution: 6 caches

The cache size is 512 KiB = 2^{19} bytes. The page table takes up 2^{22} bytes in total, which is $2^{22}/2^{19} = 2^{3} = 8$ caches.

Alternate solution: The page table takes up 3×2^{20} bytes in total, which is $3 \times 2^{20}/2^{19} = 3 \times 2 = 6$ caches.

Q10.8 (2 points) We now switch our system to having a 2-level page table instead. Assuming we restart our system and run one iteration of the inner loop, and that the VPN bits are split evenly between levels, how many pages will our active page tables span?

> **Solution:** 2
>
> No alternate solution.
>
> The L1 page table is active no matter what. However, not all L2 page tables need to be active in a 2-layer page table. (In other words, some of the entries in the L1 page table can be invalid; they don't need to all point to page with L2 page tables.)
>
> In one iteration of the inner loop, we need to access `arr[0]`. This is one address, so it forces us to make one L2 page table active, with one entry in it.
>
> Note that since we split the VPN bits evenly, we're using 10 bits to index into the L1 page table and 10 bits to index into the L2 page table. This means that all page tables (L1 and L2) have 2^{10} entries each. Each entry is either 3 or 4 bytes (depending on your alignment), but either way, the page table fits in one page. (3×2^{10} or 2^{12} bytes are both within 1 page = 4 KiB = 2^{12} bytes.)
>
> In total, this is one L1 page table and one L2 page table, for 2 pages.

Q10.9 (2 points) Assume now our system chooses to switch to another process. Which of the following events will occur. Select all that apply.

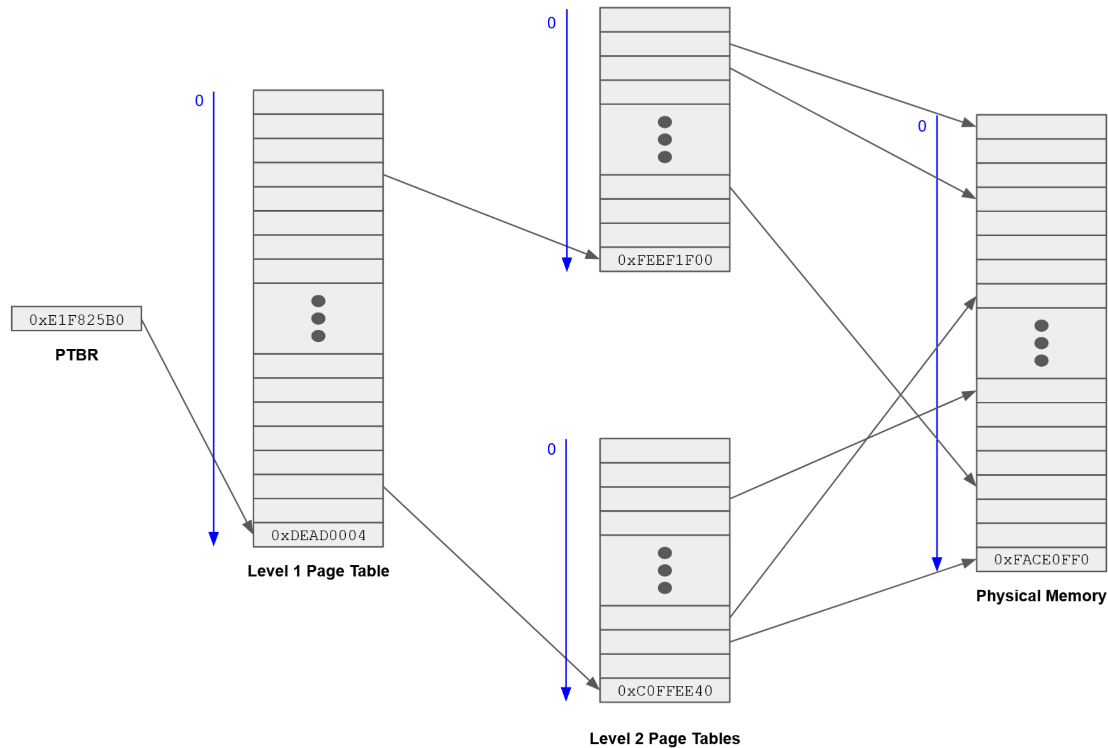
- ☐ Save the page table to the stack.
- ☐ The user triggers a timer interrupt.
- ■ **Invalidate the TLB.**
- ☐ None of the above.

> **Solution:** Save page table to stack: False. The stack is part of user-accessible memory, and the page table is in kernel-accessible (operating system) memory.
>
> The user triggers a timer interrupt: False. The OS triggers the interrupt.
>
> Invalidate the TLB: True. When switching to a different process, the mapping of VPNs to PPNs is no longer valid.

(Question 10 continued. . .)

NOTE: due to a staff error in versioning, everyone will receive full points for this question.

Given the two-level page table we have in Figure 2, translate the following virtual addresses to physical addresses. Assume all page tables shown are active. If an address does not point to an active page table at any point, write "Invalid Address".



Q10.10 (0 points) Address: `0xBFC90AEA`

Solution: This question is probably broken, but here's how you would try to solve it:

Write the address out in binary: `0b1011 1111 1100 1001 0000 1010 1110 1010`

Group the L1 VPN, L2 VPN, and offset bits: `0b10 1111 1111 00 1001 0000 1010 1110 1010`

Use the L1 VPN bits to index into the L1 page table. Usually this would be at an identifiable location like the first (L1 VPN = `0x000`) or last (L1 VPN = `0x3FF`) entry in the page table. Here it's not really clear which element of the L1 page table we want, but let's pretend it's the third from the bottom, with a protruding arrow. This tells us to look at the bottom L2 page table.

We use the L2 VPN bits to index into the bottom L2 page table. Again, this is usually at a more identifiable location. You should then be able to read off this page table entry to figure out the PPN. Then attach the offset after the PPN to get the physical address.