CS 61C Summer 2020 — Great Ideas in Computer Architecture

Final

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

○ You must choose either this option

○ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

Please complete and submit these questions before the exam starts.

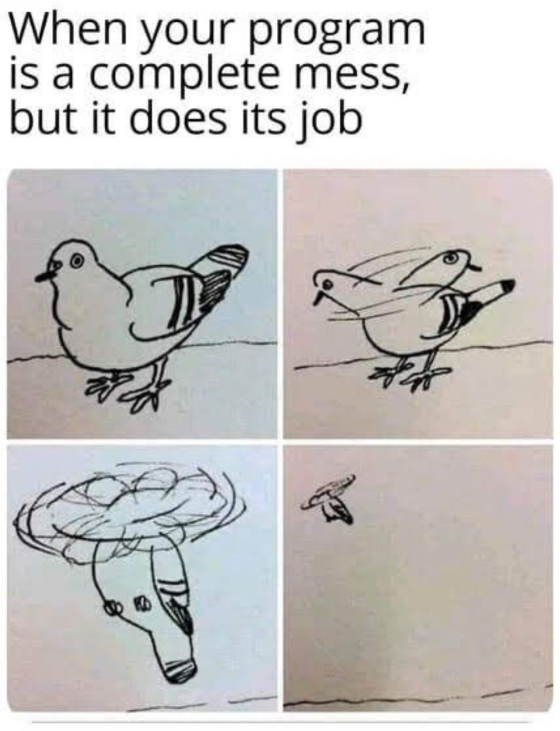
(a) What is your full name?

(b) What is your student ID number?

(c) If an answer requires hex input, make sure you only use capitalized letters! For example, 0xDEADBEEF 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!

Some of the questions may use images to describe a problem. If the image is too small, you can click and drag the image to a new tab to see the full image. You can also right click the image and download it or copy its address to view it better. You can use the image below to try this. You can also click the star by the question if you would like to go back to it (it will show up on the side bar). In addition, you are able see a check mark for questions you have fully entered in the sidebar. Questions will auto submit about 5 seconds after you click off of them, though we still recommend you click the save button.



Good luck!

1. (a) **Number Rep**

i. Translate the following numbers to their specified bases and representations. Do not include leading 0s, and remember to include the appropriate prefix for hex and binary, but no other base.

133_5

A. (1.5 pt) Decimal

43

$3 + 15 + 25 = 43$

B. (1.5 pt) Base 3 unsigned

1121

$(1 * 3^3) + (1 * 3^2) + (2 * 3^1) + (1 * 3^0) = 27 + 9 + 6 + 1 \rightarrow 1121$

ii. We want to use a new floating point format with base 3. Consider an 8 digit "minifloat" S EEE MMMM (1 sign trit, 3 exponent trits, 4 mantissa trits). All other properties of IEEE754 apply (bias, denormalized numbers,∞, NaNs, etc), which includes normalized numbers having an implicit leading 1 and denormalized numbers having an implicit leading 0. The sign digit only takes values of 0 and 1.

Normalized: $(-1)^{sign} * 3^{exponent+bias} * 1.mantissa$

Denormalized: $(-1)^{sign} * 3^{exponent+bias+1} * 0.mantissa$

Assume we have a bias of -10.

A. (2.5 pt) Represent $33.\overline{3}$ with our new floating point format.

01110201

$33.\overline{3} = 1 * 3^3 + 2 * 3 + \frac{1}{3} \rightarrow 1020.1 = 1.0201 * 3^3$ Sign : 0

Exp digits: $3 - (-10) = 1 \rightarrow 1 * 3^2 + 1 * 3^1 + 1 * 3^0 = 111$

Mantissa: 0201

Together: 0 111 0201

B. (2.5 pt) What is the decimal value of the largest positive normalized float? Express your answer in terms of powers of 3 from largest to smallest. Ex: `2*3^8+1*3^2+2*3^0`. Leave out the zero bits and do NOT add spaces. Do not add parenthses for powers!

`1*3^15+2*3^14+2*3^13+2*3^12`

Sign : 0

exp digits: $221 = 2 * 3^2 + 2 * 3 + 1 = 25 \rightarrow 25 - 10 = 15$

Mantissa: 2222

$3^{15} * (1.2222_3) = 1 * 3^{15} + 2 * 3^{14} + 2 * 3^{13} + 2 * 3^{12} = 31355019$

(b) Number Rep

i. Translate the following numbers to their specified bases and representations. Do not include leading 0s, and remember to include the appropriate prefix for hex and binary, but no other base.

114_5

A. (1.5 pt) Decimal

34

$1 * 25 + 1 * 5 + 4 * 1 = 34$

B. (1.5 pt) Base 3 unsigned

22122

$113 = (1 * 81) + (1 * 27) + (0 * 9) + (1 * 3) + (2 * 1) \rightarrow 11012$

ii. We want to use a new floating point format with base 3. Consider an 8 digit "minifloat" S EEE MMMM (1 sign trit, 3 exponent trits, 4 mantissa trits). All other properties of IEEE754 apply (bias, denormalized numbers,∞, NaNs, etc), which includes normalized numbers having an implicit leading 1 and denormalized numbers having an implicit leading 0. The sign digit only takes values of 0 and 1.

Normalized: $(-1)^{sign} * 3^{exponent+bias} * 1.mantissa$

Denormalized: $(-1)^{sign} * 3^{exponent+bias+1} * 0.mantissa$

Assume we have a bias of -10.

A. **(2.5 pt)** Represent $33.\overline{3}$ with our new floating point format.

01110201

$33.\overline{3} = 1 * 3^3 + 2 * 3 + \frac{1}{3} \rightarrow 1020.1 = 1.0201 * 3^3$

Sign : 0

Exp digits: $3 - (-10) = 1 \rightarrow 1 * 3^2 + 1 * 3^1 + 1 * 3^0 = 111$

Mantissa: 0201

Together: 0 111 0201

B. **(2.5 pt)** What is the decimal value of the largest positive denormalized float? Express your answer in terms of powers of 3 from largest to smallest. Ex: `2*3^8+1*3^2+2*3^0`. Leave out the zero bits and do NOT add spaces. Do not add parenthses for powers!

`2*3^-10+2*3^-11+2*3^-12+2*3^-13`

Sign: 0

Exp: $000 \rightarrow 0 + (-10 + 1) = -9$

Mantissa: 2222

$3^{-9} * 0.2222_3 = 2 * 3^{-10} + 2 * 3^{-11} + 2 * 3^{-12} + 2 * 3^{-13}$

(c) Number Rep

i. Translate the following numbers to their specified bases and representations. Do not include leading 0s, and remember to include the appropriate prefix for hex and binary, but no other base.

2121_3

A. (1.5 pt) Decimal

70

$54 + 9 + 6 + 1 = 70$

B. (1.5 pt) Base 5 bias (added bias of -100)

1140

$170 = 1 * 5^3 + 1 * 5^2 + 4 * 5^1 \rightarrow 1140$

ii. We want to use a new floating point format with base 3. Consider an 8 digit “minifloat” S EEEE MMM (1 sign trits, 4 exponent trits, 3 mantissa trits).

All other properties of IEEE754 apply (bias, denormalized numbers,∞, NaNs, etc), which includes normalized numbers having a most significant digit of 1 and denormalized numbers having an implicit leading 0. The sign digit only takes values of 0 and 1.

Normalized: $(-1)^{sign} * 3^{exponent+bias} * 1.mantissa$

Denormalized: $(-1)^{sign} * 3^{exponent+bias+1} * 0.mantissa$

Assume we have a bias of -33.

A. (2.5 pt) Represent $5.\overline{2}$ with our new floating point format.

01021202

Sign: 0

Exp digits: $1 - (-33) = 34 \rightarrow 1 * 3^3 + 2 * 3^1 + 1 * 3^0 \rightarrow 1021$

Mantissa: 202

$5.\overline{2} = 1 * 3^1 + 2 * 3^0 + 2 * \frac{1}{9} \rightarrow 12.02 = 1.202 * 3^1$

Answer: 0 1021 202

B. (2.5 pt) What is the decimal value of the largest positive normalized float? Express your answer in terms of powers of 3 from largest to smallest. Ex: `2*3^8+1*3^2+2*3^0`. Leave out the zero bits and do NOT add spaces. Do not add parenthses for powers!

```
1*3^46+2*3^45+2*3^44+2*3^43
```

Sign : 0

Exp digits: $2221 = 2 * 3^3 + 2 * 3^2 + 2 * 3 + 1 = 79 \rightarrow 79 - 33 = 46$

Mantissa: 222

$3^{46} * (1.222_3) = 1 * 3^{46} + 2 * 3^{45} + 2 * 3^{44} + 2 * 3^{43}$

(d) Number Rep

i. Translate the following numbers to their specified bases and representations. Do not include leading 0s, and remember to include the appropriate prefix for hex and binary, but no other base.

114_5

A. (1.5 pt) Base 3 bias (added bias of -79)

11012

$113 = (1 * 81) + (1 * 27) + (0 * 9) + (1 * 3) + (2 * 1) \rightarrow 11012$

B. (1.5 pt) Binary 2's Unsigned

0b100010

$34 = 1 * 2^5 + 1 * 2^1 = 0b100010$

ii. We want to use a new floating point format with base 3. Consider an 8 digit “minifloat” S EEEE MMM (1 sign trit, 4 exponent trits, 3 mantissa trits). All other properties of IEEE754 apply (bias, denormalized numbers,∞, NaNs, etc), which includes normalized numbers having an implicit leading 1 and denormalized numbers having an implicit leading 0. The sign digit only takes values of 0 and 1.

Normalized: $(-1)^{sign} * 3^{exponent+bias} * 1.mantissa$

Denormalized: $(-1)^{sign} * 3^{exponent+bias+1} * 0.mantissa$

Assume we have a bias of -33.

A. (2.5 pt) Represent $5.\overline{2}$ with our new floating point format.

01021202

$5.\overline{2} = 1$ * 3^1 + 2*3^0 + 2*1$\overline{\text{9}\rightarrow\text{12.02=1.202}*\text{3}^1\$}$

Sign: 0

Exp digits: $1 - (-33) = 34 \rightarrow 1 * 3^3 + 2 * 3^1 + 1 * 3^0 \rightarrow 1021$

Mantissa: 202

Answer: 0 1021 202

B. (2.5 pt) What is the decimal value of the largest positive denormalized float? Express your answer in terms of powers of 3 from largest to smallest. Ex: `2*3^8+1*3^2+2*3^0`. Leave out the zero bits and do NOT add spaces. Do not add parenthses for powers!

`2*3^-33+2*3^-34+2*3^-35`

Sign: 0

Exp: $0000 \rightarrow 0 + (-33 + 1) = -32$

Mantissa: 222

$3^{-32} * 0.222_3 = 2 * 3^{-33} + 2 * 3^{-34} + 2 * 3^{-35}$

2. (a) I/O

We wish to communicate with an I/O device using Memory Mapped I/O. To do so, we have set aside a portion of our address space to communicate with this device, beginning at address `0xA0000000`. Below is a table describing all the special addresses (control/data registers) and the purpose of each value that lives there. Assume that our device has 16 pins for I/O which can each hold 32 bits of data, `sizeof(uint16_t) == 2`, `sizeof(uint32_t) == 4`, and `sizeof(uint64_t) == 8`:

Address	Field Name	Purpose
0xA0000100	IN_READY	The i-th bit indicates whether or not the device has a value at pin i that should be read by the computer via a 1 or 0, respectively
0xA0000104	OUT_READY	The i-th bit indicates whether or not the computer has a value for pin i that should be read by the device via a 1 or 0, respectively
0xA0000200	IN_DATA	The input data from the pin indicated by IN_READY
0xA0000204	OUT_DATA	The output data to the pin indicated by OUT_READY

i. Fill in the following C code to complete the implementation of a struct that will "cover" these addresses and allow us to manage this device without hard-coding all the addresses. For example, we should be able to access IN_READY by using IO_device->IN_READY. Assume that memory will be word-aligned, but not padded. You should be using all provided lines and can only have one semicolon per line:

```
typedef struct {
    uint16_t IN_READY;
    uint16_t padding1[<**CODE INPUT 1**>];
    <**CODE INPUT 2**>;
    uint16_t padding2[<**CODE INPUT 3**>];
    <**CODE INPUT 4**>
    uint32_t OUT_DATA;
} IO_device;
```

A. (1.0 pt) `<**CODE INPUT 1**>`

1

0x104 - 0x100 = 4 bytes, take away 2 for IN_READY, so we are left with 4 - 2 = 2 = 1 uint16_t of padding

B. (0.5 pt) `<**CODE INPUT 2**>`

uint16_t OUT_READY;

C. (1.0 pt) `<**CODE INPUT 3**>`

125

0x200 - 0x100 = 256 bytes, take away 2 for IN_READY, 2 for padding1, 2 for OUT_READY, so we are left with 256 - 6 = 250 / 2 = 125 uint16_t of padding

D. (0.5 pt) `<**CODE INPUT 4**>`

uint32_t IN_DATA;

ii. Now that you have this struct at your disposal, use it to complete the following functions that will allow you to communicate with your device. Assume that memory has been initialized to random data.

```
# define base_io_addr 0xA0000000
IO_device* IO_device_ptr = <**CODE INPUT 1**>;

uint32_t read_from_pin(int pin) {
    # Check if pin has something to be read, and if so, read this value. Else, return 0
    <**CODE INPUT 2**>
}

void write_to_pin(int pin, uint32_t data) {
    # Notify the device that we have something to write, and then write it.
    # Note that OUT_READY can only have one bit active at a time,
    # but our device handles resetting this value every time it reads.
    <**CODE INPUT 3**>
}
```

A. (1.0 pt) `<**CODE INPUT 1**>`

$base\_io\_addr + 0x100$

B. (3.0 pt) `<**CODE INPUT 2**>`

```
if (IO_device_ptr->IN_READY >> pin) & 1) {
   return IO_device_ptr->IN_DATA;
}
return 0;
```

C. (5.0 pt) `<**CODE INPUT 3**>`

```
IO_device_ptr->OUT_READY = IO_device_ptr->OUT_READY | (1 << pin);
IO_device_ptr->OUT_DATA = data;
```

(b) I/O

We wish to communicate with an I/O device using Memory Mapped I/O. To do so, we have set aside a portion of our address space to communicate with this device, beginning at address `0xA0000000`. Below is a table describing all the special addresses (control/data registers) and the purpose of each value that lives there. Assume that our device has 32 pins for I/O which can each hold 16 bits of data, `sizeof(uint16_t) == 2`, `sizeof(uint32_t) == 4`, and `sizeof(uint64_t) == 8`:

Address	Field Name	Purpose
0xA0000100	READY_IN	The i-th bit indicates whether or not the device has a value at pin i that should be read by the computer via a 1 or 0, respectively
0xA0000108	READY_OUT	The i-th bit indicates whether or not the computer has a value for pin i that should be read by the device via a 1 or 0, respectively
0xA0000200	DATA_IN	The input data from the pin indicated by READY_IN
0xA0000202	DATA_OUT	The output data to the pin indicated by READY_OUT

i. Fill in the following C code to complete the implementation of a struct that will "cover" these addresses and allow us to manage this device without hard-coding all the addresses. For example, we should be able to access READY_IN by using IO_device->READY_IN. Assume that memory will be word-aligned, but not padded. You should be using all provided lines and can only have one semicolon per line:

```
typedef struct {
    uint32_t READY_IN;
    uint32_t padding1[<**CODE INPUT 1**>];
    <**CODE INPUT 2**>;
    uint32_t padding2[<**CODE INPUT 3**>];
    <**CODE INPUT 4**>
    uint16_t DATA_OUT;
} IO_device;
```

A. (1.0 pt) `<**CODE INPUT 1**>`

1

0x108 - 0x100 = 8 bytes, take away 4 for IN_READY, so we are left with 8 - 4 = 4 = 1 uint32_t of padding

B. (0.5 pt) `<**CODE INPUT 2**>`

uint32_t READY_OUT;

C. (1.0 pt) `<**CODE INPUT 3**>`

61

0x200 - 0x100 = 256 bytes, take away 4 for IN_READY, 4 for padding1, 4 for OUT_READY, so we are left with 256 - 12 = 244 / 4 = 61 uint32_t of padding

D. (0.5 pt) `<**CODE INPUT 4**>`

uint16_t DATA_IN;

ii. Now that you have this struct at your disposal, use it to complete the following functions that will allow you to communicate with your device. Assume that memory has been initialized to random data.

```
# define base_io_addr 0xA0000000
IO_device* IO_device_ptr = <**CODE INPUT 1**>;

uint16_t read_from_pin(int pin) {
    # Check if pin has something to be read, and if so, read this value. Else, return 0
    <**CODE INPUT 2**>
}

void write_to_pin(int pin, uint16_t data) {
    # Notify the device that we have something to write, and then write it.
    # Note that READY_OUT can only have one bit active at a time,
    # but our device handles resetting this value every time it reads.
    <**CODE INPUT 3**>
}
```

A. **(1.0 pt)** `<**CODE INPUT 1**>`

base_io_addr + 0x100

B. **(3.0 pt)** `<**CODE INPUT 2**>`

```
if (IO_device_ptr->READY_IN >> pin) & 1) {
   return IO_device_ptr->DATA_IN;
}
return 0;
```

C. **(5.0 pt)** `<**CODE INPUT 3**>`

```
IO_device_ptr->READY_OUT = IO_device_ptr->READY_OUT | (1 << pin);
IO_device_ptr->DATA_OUT = data;
```

(c) I/O

We wish to communicate with an I/O device using Memory Mapped I/O. To do so, we have set aside a portion of our address space to communicate with this device, beginning at address `0xA0000000`. Below is a table describing all the special addresses (control/data registers) and the purpose of each value that lives there. Assume that our device has 16 pins for I/O which can each hold 64 bits of data, `sizeof(uint16_t) == 2`, `sizeof(uint32_t) == 4`, and `sizeof(uint64_t) == 8`:

Address	Field Name	Purpose
0xA0000100	IN_ready	The i-th bit indicates whether or not the device has a value at pin i that should be read by the computer via a 1 or 0, respectively
0xA0000108	OUT_ready	The i-th bit indicates whether or not the computer has a value for pin i that should be read by the device via a 1 or 0, respectively
0xA0000200	IN_data	The input data from the pin indicated by IN_ready
0xA0000208	OUT_data	The output data to the pin indicated by OUT_ready

i. Fill in the following C code to complete the implementation of a struct that will "cover" these addresses and allow us to manage this device without hard-coding all the addresses. For example, we should be able to access IN_ready by using IO_device->IN_ready. Assume that memory will be word-aligned, but not padded. You should be using all provided lines and can only have one semicolon per line:

```
typedef struct {
    uint16_t IN_ready;
    uint16_t padding1[<**CODE INPUT 1**>];
    <**CODE INPUT 2**>;
    uint16_t padding2[<**CODE INPUT 3**>];
    <**CODE INPUT 4**>
    uint64_t OUT_data;
} IO_device;
```

A. (1.0 pt) `<**CODE INPUT 1**>`

3

0x108 - 0x100 = 8 bytes, take away 2 for IN_READY, so we are left with 8 - 2 = 6 = 3 uint16_t of padding

B. (0.5 pt) `<**CODE INPUT 2**>`

uint16_t OUT_ready;

C. (1.0 pt) `<**CODE INPUT 3**>`

123

0x200 - 0x100 = 256 bytes, take away 2 for IN_READY, 3 * 2 = 6 for padding1, 2 for OUT_READY, so we are left with 256 - 10 = 246 / 2 = 123 uint32_t of padding

D. (0.5 pt) `<**CODE INPUT 4**>`

uint64_t IN_data;

ii. Now that you have this struct at your disposal, use it to complete the following functions that will allow you to communicate with your device. Assume that memory has been initialized to random data.

```
# define base_io_addr 0xA0000000
IO_device* IO_device_ptr = <**CODE INPUT 1**>;

uint64_t read_from_pin(int pin) {
    # Check if pin has something to be read, and if so, read this value. Else, return 0
    <**CODE INPUT 2**>
}

void write_to_pin(int pin, uint64_t data) {
    # Notify the device that we have something to write, and then write it.
    # Note that OUT_ready can only have one bit active at a time,
    # but our device handles resetting this value every time it reads.
    <**CODE INPUT 3**>
}
```

A. (1.0 pt) `<**CODE INPUT 1**>`

$base\_io\_addr + 0x100$

B. (3.0 pt) `<**CODE INPUT 2**>`

```
if (IO_device_ptr->IN_ready >> pin) & 1) {
   return IO_device_ptr->IN_data;
}
return 0;
```

C. (5.0 pt) `<**CODE INPUT 3**>`

```
IO_device_ptr->OUT_ready = IO_device_ptr->OUT_ready | (1 << pin);
IO_device_ptr->OUT_data = data;
```

(d) I/O

We wish to communicate with an I/O device using Memory Mapped I/O. To do so, we have set aside a portion of our address space to communicate with this device, beginning at address `0xA0000000`. Below is a table describing all the special addresses (control/data registers) and the purpose of each value that lives there. Assume that our device has 32 pins for I/O which can each hold 64 bits of data, `sizeof(uint16_t) == 2`, `sizeof(uint32_t) == 4`, and `sizeof(uint64_t) == 8`:

Address	Field Name	Purpose
0xA0000100	IN_READY	The i-th bit indicates whether or not the device has a value at pin i that should be read by the computer via a 1 or 0, respectively
0xA0000108	OUT_READY	The i-th bit indicates whether or not the computer has a value for pin i that should be read by the device via a 1 or 0, respectively
0xA0000200	IN_DATA	The input data from the pin indicated by IN_READY
0xA0000208	OUT_DATA	The output data to the pin indicated by OUT_READY

i. Fill in the following C code to complete the implementation of a struct that will "cover" these addresses and allow us to manage this device without hard-coding all the addresses. For example, we should be able to access IN_READY by using IO_device->IN_READY. Assume that memory will be word-aligned, but not padded. You should be using all provided lines and can only have one semicolon per line:

```
typedef struct {
    uint32_t IN_READY;
    uint32_t padding1[<**CODE INPUT 1**>];
    <**CODE INPUT 2**>;
    uint32_t padding2[<**CODE INPUT 3**>];
    <**CODE INPUT 4**>
    uint64_t OUT_DATA;
} IO_DEVICE;
```

A. (1.0 pt) `<**CODE INPUT 1**>`

1

0x108 - 0x100 = 8 bytes, take away 4 for IN_READY, so we are left with 8 - 4 = 4 = 1 uint32_t of padding

B. (0.5 pt) `<**CODE INPUT 2**>`

uint32_t OUT_READY;

C. (1.0 pt) `<**CODE INPUT 3**>`

61

0x200 - 0x100 = 256 bytes, take away 4 for IN_READY, 4 for padding1, 4 for OUT_READY, so we are left with 256 - 12 = 244 / 4 = 61 uint32_t of padding

D. (0.5 pt) `<**CODE INPUT 4**>`

uint64_t IN_DATA;

ii. Now that you have this struct at your disposal, use it to complete the following functions that will allow you to communicate with your device. Assume that memory has been initialized to random data.

```
# define base_io_addr 0xA0000000
IO_DEVICE* IO_DEVICE_ptr = <**CODE INPUT 1**>;

uint64_t read_from_pin(int pin) {
    # Check if pin has something to be read, and if so, read this value. Else, return 0
    <**CODE INPUT 2**>
}

void write_to_pin(int pin, uint64_t data) {
    # Notify the device that we have something to write, and then write it.
    # Note that OUT_READY can only have one bit active at a time,
    # but our device handles resetting this value every time it reads.
    <**CODE INPUT 3**>
}
```

A. (1.0 pt) `<**CODE INPUT 1**>`

base_io_addr + 0x100

B. (3.0 pt) `<**CODE INPUT 2**>`

```
if (IO_DEVICE_ptr->IN_READY >> pin) & 1) {
   return IO_DEVICE_ptr->IN_DATA;
}
return 0;
```

C. (5.0 pt) `<**CODE INPUT 3**>`

```
IO_DEVICE_ptr->OUT_READY = IO_DEVICE_ptr->OUT_READY | (1 << pin);
IO_DEVICE_ptr->OUT_DATA = data;
```

3. RISC-V Coding

We wish to implement a function, `reverse_str`, that will take in a pointer to a string, its length, and reverse it. Assume that the argument registers, `a0`, `a1`, hold the pointer to and length of the string, respectively. Complete the following code skeleton to implement this function. You **must** use commas to separate arguments in your code, e.g. `add x0, x0, x0`.

```
Reverse_str:

    # This part saves all the required registers you will use.
    # HIDDEN CODE

    mv s0, a0 # memory address
    mv s1, a1 # strlen
    addi t0, x0, 0 # iteration
Loop:

    # YOUR CODE HERE
    # retrieve left and right letters

    # switch chars

    # iterate if necessary

    # END YOUR CODE HERE

    # This part restores all of the registers which were used.
    # HIDDEN CODE
    ret
```

(a) (10.0 pt)

```
# retrieve left and right letters
add t1, s0, t0 # t1 is moving pointer from left (base + offset/iteration)
lb t2 0(t1) # t2 contains char from left
sub t3, s1, t0 # imm needs to be s1 - t0
addi t3, t3, -1 # since strlen indexes out of string
add t4, s0, t3 # t4 is moving pointer from right (base + strlen -
offset/iteration - 1)
lb t5 0(t4) # t5 contains char from right
# switch chars
sb t2, 0(t4)
sb t5, 0(t1)
# iterate if necessary
addi t0, t0, 1 # update iter
srli s8, s1, 1
bne t0, s8, Loop
mv a0, s0 # not necessary
```

4. RISC-V Instruction Format

You are working on a new chip for an embedded application, and want to create a new ISA. Fed up with the different RISC-V instruction types, you decide to include only one, universal type - the X-type instruction.

Say we wish to include the following instructions:

```
0.  add rd1, rs1, rs2
1.  and rd1, rs1, rs2
2.  lw rd1, offset1 (rs1)
3.  sw rs2, offset1 (rs1)
4.  addi rd1, rs1, imm1
5.  beq rs1, rs2, offset1
6.  lui rd1, offset1
7.  jal rd1, imm
8.  stw rs3, offset1, offset2 (rs1)
```

The new stw instruction stores the contents of `rs3` into both `rs1 + offset1` and `rs1 + offset2`. The RTL is:

`Mem(R[rs1] + offset1)`$\leftarrow$`R[rs3] AND Mem(R[rs1] + offset2)`$\leftarrow$`R[rs3]`

(a) **(2.0 pt)** You want to do away with the funct3 and funct7 fields and only use an opcode. If we only wish to support the instructions listed above, what is the minimum number of bits the opcode field can be?

4

4 bits

We have 9 instructions, so we need 4 bits to represent them.

(b) **(3.0 pt)** We want to be able to jump up to 64 KiB in either direction with a single instruction. How many bits are necessary to encode an immediate that would allow us to do this? Assume that, just like RV32, the least significant bit is an implicit 0 and is not stored in the instruction.

16

16 bits

64 KiB = 2^{16} B. Jal is the only jump instruction given, so it will determine the size of the immediate field. Recall that, in RISC-V, the immediates encoded in instructions work in units of half-instructions, so jumping up to 64 KiB in either direction is the same as saying we want to jump up to 2ˆ15 half instructions in both directions. Since immediates are signed, we need 16 bits to represent this range of values.

(c) **(2.0 pt)** Regardless of your previous answers, you finally decide on the instruction format below. You've added some fields to account for new instructions you might want to include later on. **The opcode for each instruction is the same as the list index given at the beginning of this problem (e.g. sw has opcode 3).**

imm2	imm1	rs3	rs2	rs1	rd2	rd1	opcode

This instruction format is quite long, so we decide to work on a 64-bit machine. Each immediate field is 11 bits, and the opcode is 7 bits. What is the maximum number of registers we can have?

128

128 registers

2*11-bit immediates + 7-bit opcode = 29 bits. 64 - 29 = 35, so we have 35 bits left over for the registers.

There are 5 registers, so we can use 7 bits per register. $2^7 = 128$ registers.

(d) **(3.0 pt)** Realizing supplies have run low due to COVID-19, you switch to a 32-bit machine, and finalize your instruction format to have 4 bits for each of the immediate fields, 4 bits for each register, and 4 bits for the opcode.

Convert the instruction `stw x8, 0, 4 (x5)` into machine code. Leave your answer in binary (don't forget the prefix!). If a field is not used, fill in the field with 'x's.

010000001000xxxx0101xxxxxxxx1000

Rs3 is x8 = 0b1000. Rs1 is x5 = 0b0101. Imm1 = 0b0000. Imm2 = 0b0100. So, the final instruction is 0100 | 0000 | 1000 | xxxx | 0101 | xxxx | xxxx | 1000.

5. CALL

Suppose we have compiled some C code using the Hilfinger-Approved(TM) CS61Compiler, which will compile, assemble, and link the files `max.c` and `jie.c`,among others, to create a wonderful executable. After the code has been assembled to RISC-V we have the following labels across all files: `sean`, `jenny`, `stephan`, `philspel`, `poggers`, `crossroads`, and `segfault`. Assume no two files define the same label, though each file interacts with every label, either via reference or definition.

Note: **segment** refers to a directive in any assembly file, e.g. `.data` or `.text`

The CS 61Compiler begins to fill out the relocation table on the first pass of assembling `max.s`, which defines or references all of the labels above. This is its relocation table after the first pass:

label	address
`sean`	????
`stephan`	????
`jenny`	????
`segfault`	????
`philspel`	????

(a) **(2.0 pt)** `sean`, `stephan`, `jenny`, `segfault`, and `philspel` all show up in the relocation table after the first pass through. Which of the following must be true? Select all that apply.

- [✓] They are referenced before they are defined.
- [ ] They belong in the `.text` segment.
- [ ] They are external references.
- [ ] They are referenced before `poggers` and `crossroads`.
- [ ] None of the other options

(b) **(2.0 pt)** After the first pass through, `poggers` and `crossroads` don't show up in the relocation table. What does this imply about the two function labels? Select all that apply.

- [✓] After the assembler is finished, they are in the same segment.
- [ ] They are both referenced before they are defined.
- [ ] They are `.globals`.
- [ ] None of the other options

(c) **(2.0 pt)** After the second pass by the assembler, we see that `philspel` is no longer in the relocation table. Which of the following is true about `philspel`? Select all that apply.

- [✓] `philspel` is in the `.text` segment of `max.s`
- [ ] `philspel` is in the `.text` segment of `jie.s`
- [✓] The address for `philspel` was resolved.
- [ ] `philspel` is an external reference.
- [ ] None of the other options

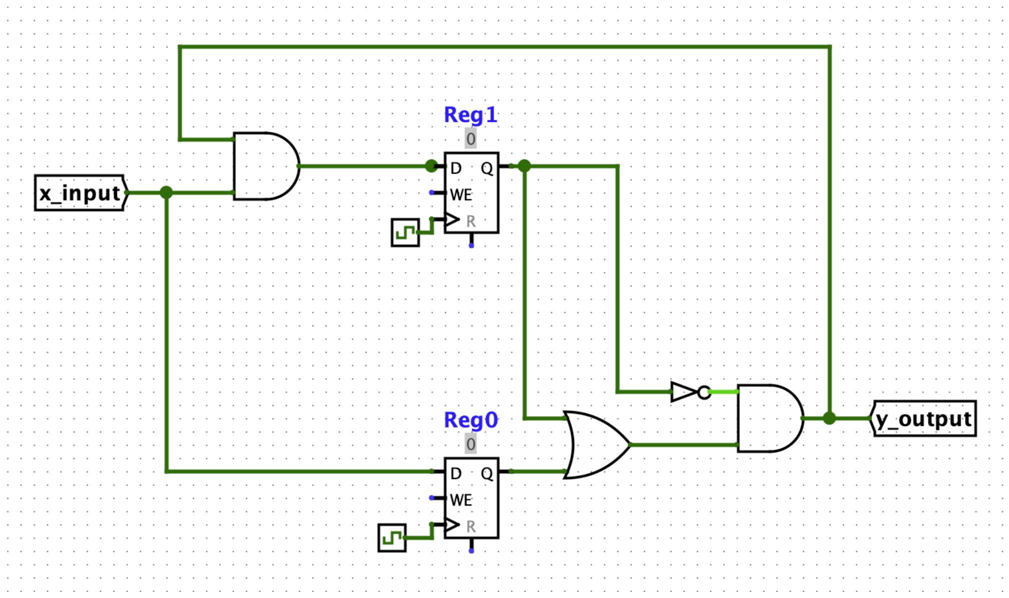
(d) **(2.0 pt)** After assembling `jie.s` to `jie.o` we have the following symbol table for `jie.o`. In linking `max.o` and `jie.o` we get `dan.out`. Which of the following could be true about 'sean' and 'jenny' after linking? Select all that apply.

label	address
`sean`	0x061c
`jenny`	0x1620

- [✓] `sean` and `jenny` are in different sections of `jie.s`.
- [✓] `sean` and `jenny` will have the same byte difference after linking as it did in `jie.o`.
- ☐ They are in different files.
- [✓] They are in the same segment.
- ☐ None of the other options

6. SDS, Logic

We will be analyzing the following circuit:



Circuit

Given the following information:

- **AND** gates have a propagation delay of 9ns
- **OR** gates have a propagation delay of 14ns
- **NOT** gates have a propagation delay of 5ns
- **x_input** switches value(i.e. 1 to 0, 0 to 1) 30 ns after the rising edge of the clk
- **y_output** is directly attached to a register
- **Setup** time is 3ns
- **Clk-to-q** delay time: 4ns

(a) (2.0 pt) What is the max hold time in ns?

18

Shortest CL: NOT -> AND = 5 + 9 = 14ns clk-to-q + shortest CL = 4ns+14ns = 18ns

(b) (2.0 pt) What is the minimum clock period in ns?

42

Critical path = clk-to-q + longest CL + setup = 30ns for x_input to change (includes clk-to-q) + 9 AND + 3 setup = 42 ns

(c) (3.0 pt) Regardless of your previous answers, assume the clock period is 50ns, the first rising edge of the clock is at 25 ns and x_input is initialized to 0 at 0ns. At what time in ns will y_output become 1?

102

25+50(clock period)+4(clk-to-q)+14(OR)+9(AND) = 102 ns

(d) **(3.0 pt)** How long will y_output remain equal to 1 before switching to 0?

41

If y_output changes to 1 at 102, the next rising edge is at 125. From there, it takes Clk-Q (4) + OR (14) + AND (9) = 27 ns to update y_output to 0 again, so at 125 + 18 = 152 ns, 152 - 102 = 50 ns

To see what is happened during each clock period:

During 0th clock period (starts before question): x_input stabilizes at 0, inputs to Reg0 and Reg1 stabilize at 0

During 1st clock period (starts at 25ns): x_input stabilizes at 1, input to Reg0 stabilizes at 1, y_output stabilizes at 0 –> input to Reg1 stabilizes at 0

During 2nd clock period (starts at 75ns): x_input stabilizes at 0, input to Reg0 stabilizes at 0, y_output stabilizes at 1, input to Reg1 (AND of current x_input and current y_output) stabilizes at 0

During 3rd clock period (starts at 125ns): x_input stabilizes at 1, input to Reg0 stabilizes at 1, y_output stabilizes at 0 –> input to Reg1 stabilizes at 0

7. Single Cycle Datapath

(a) Which of the following components are not utilized by the given instruction? As in, the output(s) of the component are not useful to the overall execution of the instruction. Select all that apply.

i. A. (2.0 pt) `lui s2, 0xC561C`

- ■ Branch comparator
- ☐ IMEM
- ☐ Immediate generator
- ☐ Register File
- ☐ All components are utilized by this instruction

B. (2.0 pt) `jal ra, label`

- ☐ ALU
- ■ DMEM
- ☐ PC register
- ☐ Control Logic Unit
- ☐ All components are utilized by this instruction

ii. A. (2.0 pt) `auipc s1, 0xC561C`

- ☐ ALU
- ☐ Register File
- ☐ PC register
- ■ DMEM
- ☐ All components are utilized by this instruction

B. (2.0 pt) `sw t0, 0(t1)`

- ☐ IMEM
- ☐ Control Logic Unit
- ☐ Immediate generator
- ■ Branch comparator
- ☐ All components are utilized by this instruction

iii. A. (2.0 pt) `lui s4, 0xC561C`

- ☐ Control Logic Unit
- ☐ ALU
- [✓ DMEM]
- ☐ PC register
- ☐ All components are utilized by this instruction

B. (2.0 pt) `lw t0, 0(t1)`

- ☐ Register File
- ☐ IMEM
- [✓ Branch Comparator]
- ☐ Immediate Generator
- ☐ All components are utilized by this instruction

iv. A. (2.0 pt) `auipc s3, 0xC561C`

- ☐ IMEM
- ☐ Register file
- [✓ Branch Comparator]
- ☐ Immediate generator
- ☐ All components are utilized by this instruction

B. (2.0 pt) `beq t6, t5, label`

- ☐ PC register
- ☐ ALU
- ☐ Control Logic Unit
- [✓ DMEM]
- ☐ All components are utilized by this instruction

(b) You've been running multiple recursive programs on your RISC-V CPU lately, and noticed that one of the main causes of slowdown is that you always have to save ra onto the stack before you do the recursive call. You decide to modify your current single-cycle RISC-V datapath to implement an instruction that can save to the stack and jump at the same time.

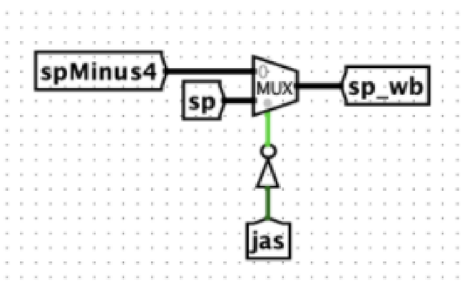
Jump-and-save

`jas label`

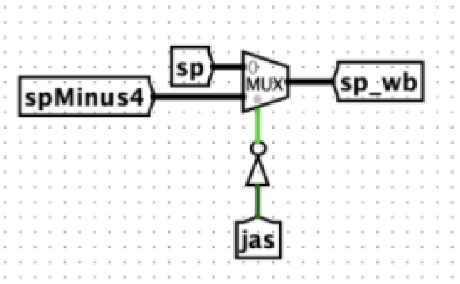
`R[ra] = PC + 4, R[sp] = sp - 4, PC = PC + offset, Mem[sp - 4] = PC + 4`

If the `jas` instruction is ran, the correspondingly named signal is set to 1.

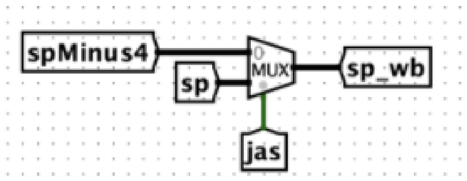
i. (2.0 pt) Which of the following will correctly implement `sp_wb`, the value that will get written back to `sp`? `spMinus4` is a pre-computed value equal to `sp - 4`.



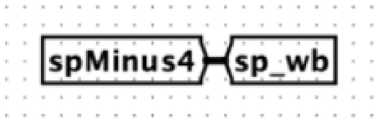
A



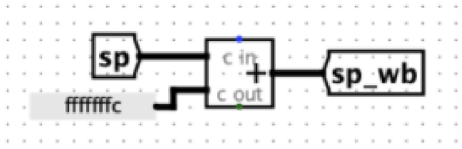
B



C



D



E

sp_wb Choices

[✓ A]

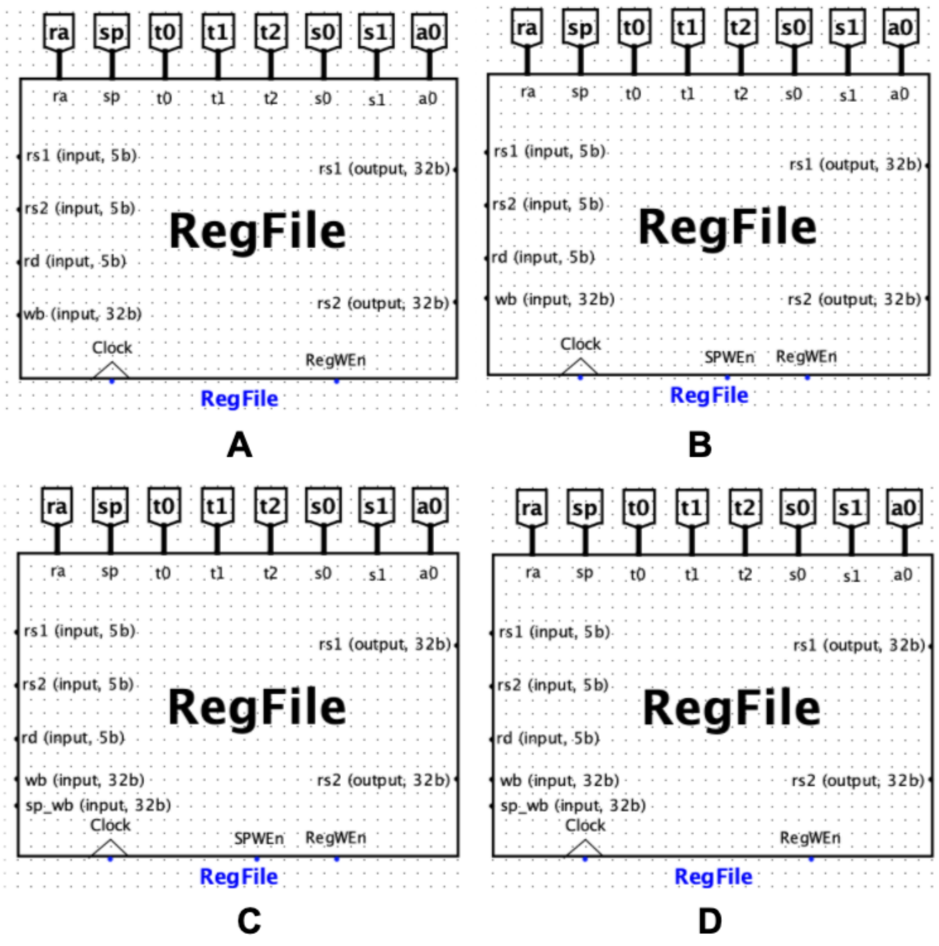
☐ B

☐ C

☐ D

☐ E

After the solutions were released, some students pointed out that the move to make sp_wb and SPWEn the exclusive method of writing to sp caused more logical complexity than is shown, such as changing the RegFile to ignore RegWEn when dealing with sp. Due to the lack of clarity in this regard, there was no penalty for putting D.

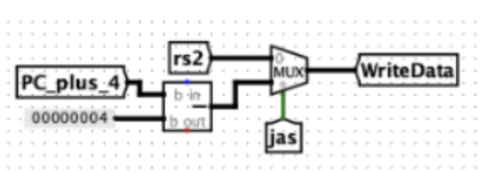


RegFile Choices

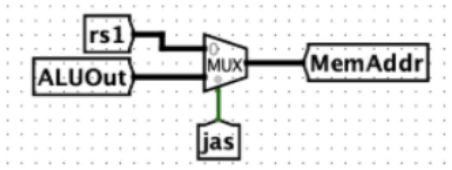
ii. (2.0 pt) Now that we have `sp_wb`, which of the following will correctly write it back to the RegFile? `SPWEn` is a signal analogous to `RegWEn`: it is 1 when we wish to write to `sp` and 0 otherwise. Furthermore, if `SPEn` is false, `SP` will not be updated, even if `RegWEn` is true.

- ☐ A
- ☐ B
- [✓ C]
- ☐ D

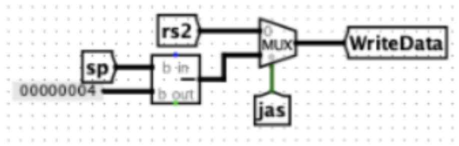
iii. (2.0 pt) Which combination of the following circuits will correctly implement the "save to the stack" operation?



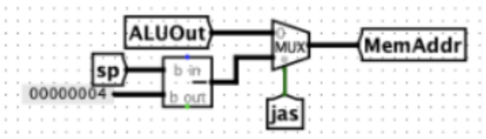
A



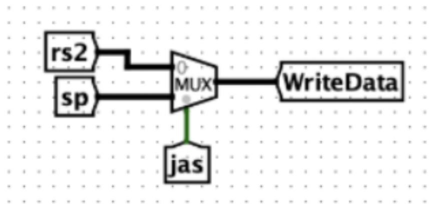
B



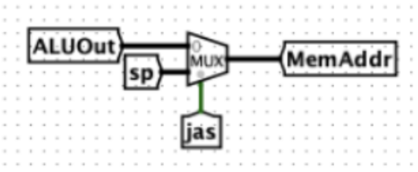
C



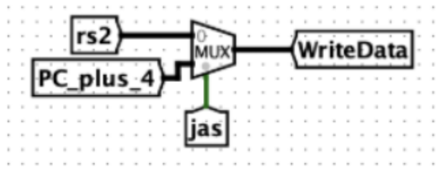
D



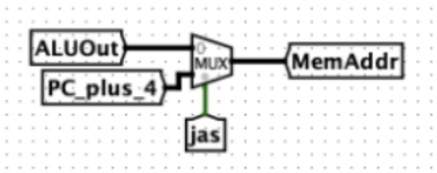
E



F



G



H

Memory Choices

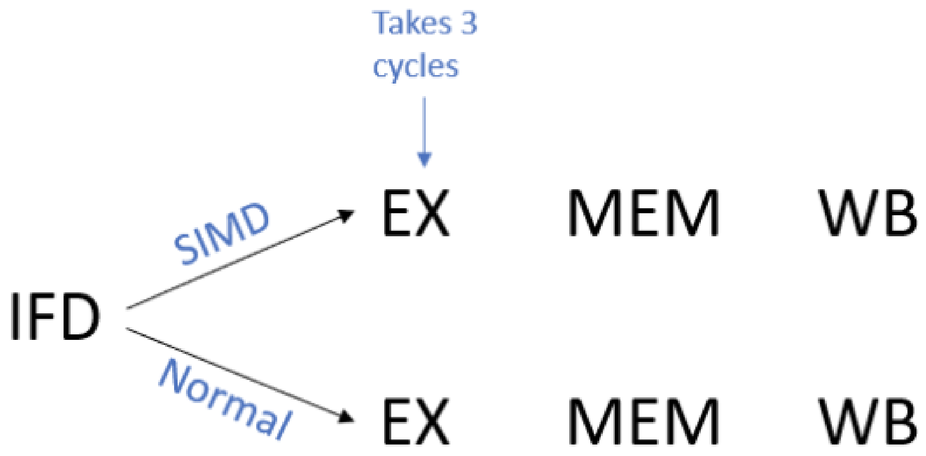
- ☐ A
- ☐ B
- ☐ C
- [✓ D]
- ☐ E
- ☐ F
- [✓ G]
- ☐ H

8. Pipeline

We wish to implement SIMD instructions in our pipelined RISC-V datapath. In order to do so, we will take 2 steps:

Combine the IF and ID stages into an IFD stage Create 2 paths for the datapath to take after IF: one path for normal RISC-V instructions and one for SIMD RISC-V containing specialized hardware.

The only problem is that the **SIMD EX stage takes 3 cycles** to complete instead of 1, and **no other SIMD instruction is allowed to enter the SIMD EX stage while another SIMD instruction is there.**



Pipeline

(a) **(3.0 pt)** This "delay" from the SIMD EX stage necessitates the use of **stalls** to ensure proper functionality. Which of the following implementations correctly generates the `stall` signal? You may ignore any kinds of stalls caused by hazards; we are only concerned with this special case in our new pipeline. However, we still want to maintain a good instruction throughput. To do this, we should allow normal instructions to continue through the CPU, as they are not blocked from doing so by the SIMD path.

The signal `SIMD_Inst` is an indicator that the current instruction fetched from IFD is a SIMD instruction, while the signal `SIMD_count` refers to the number of the cycle the SIMD instruction is completing in the EX stage, i.e. when it is in the first cycle of the EX stage, `SIMD_count` $= 1$. If there is no instruction in the SIMD EX stage, this value is 0. The comparators are unsigned. Select all that apply.

☐ A

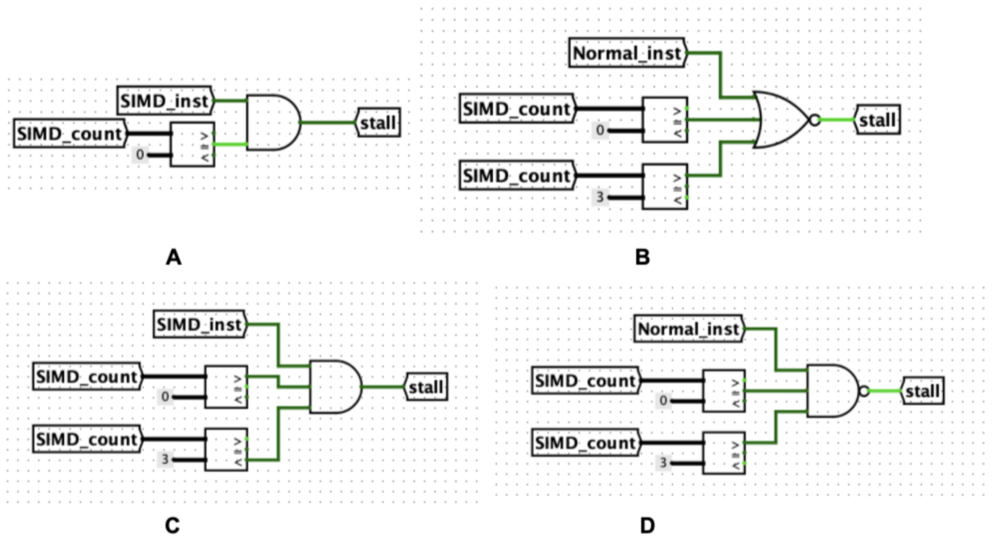
☐ B

■ C

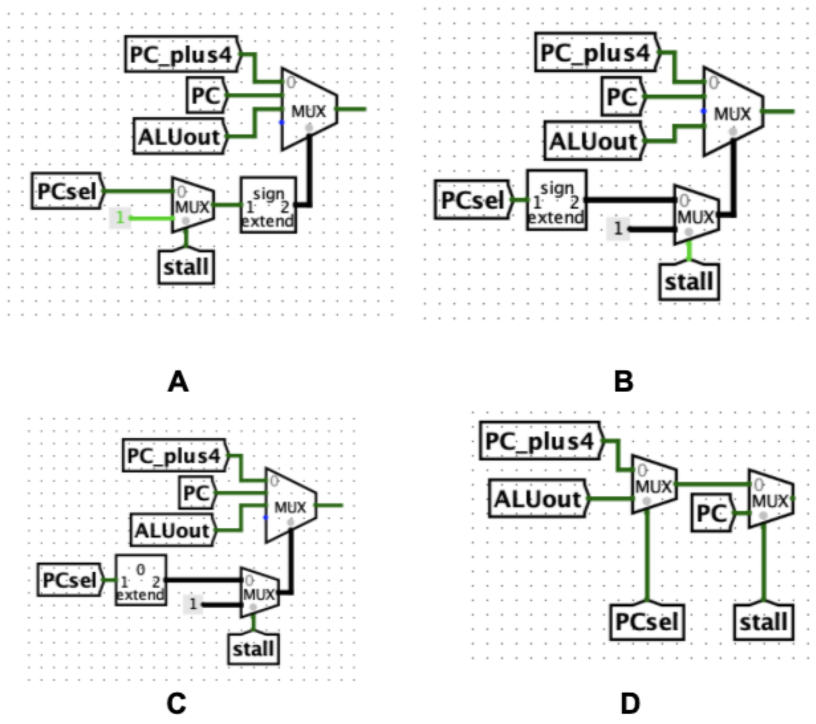
☐ D

☐ None of the other options

C. We need to stall when we fetch a SIMD instruction, but one is already in the EX stage and will remain there for the next cycle (i.e. simd_count $>=$ 1, but $<=$ # of cycles for EX - 1 $=$ 2). Otherwise, if a normal instruction is fetched, it can just be pushed to the Normal path, and no stall is needed.



Pipeline 1



Pipeline 2

(b) **(3.0 pt)** Because we wish to actually stall and not flush, how should the PC and PC mux be updated to allow for this? Assume `stall` is a signal that is 1 when we should stall, and therefore not fetch a new instruction, or 0 otherwise. Select all that apply.

- ☐ A
- ■ B
- ☐ C
- ■ D
- ☐ None of the other options

B, D

When stall = 1, we want the new PC to be the old PC, otherwise the behavior should be unchanged. Option A is incorrect since the selector bits of 1 or 2 will never be chosen; 0 will be sign extended to 00 (PC+4), while 1 will be sign extended to 11 (undefined). Option B is correct. Option C is close, but incorrect as ALUout is at index 2 of the mux, which is unreachable as the normal PCSel = 1 that would choose the ALUout becomes 11 after sign extension. Option D is correct.

(c) **(2.0 pt)** How many stalls caused by the SIMD EX stage are needed for the following piece of code?

```
1. addi t0, x0, 1
2. simd_add x1, x2, x3
3. simd_sub x2, x1, x3
4. addi t1, t0, 2
5. simd_mul x2, x2, x2
6. sub t0, t0, t1
7. mul t0, t0, t0
8. simd_div x2, x2, x1
```

3

Line 2 adds 2 stalls.

Line 5 adds 1 stall left over from simd_sub.

9. Cache and MOESI

Consider a computer which has 2 processors, each with their own cache. Both have the same design: A 128 B cache size, 2-way set associative, 4 ints per block, write-back, and write-allocate with LRU replacement. Each cache takes in 20-bit addresses. Assume that ints are 4 bytes, and we are using the MOESI cache-coherence protocol.

(a) **(0.25 pt)** The 20-bit addresses are Virtual Addresses

○ True

[✓ False]

(b) i. **(0.25 pt)** How many Offset bits?

4

4 bits

ii. **(0.25 pt)** How many Index bits?

2

128 / 16 = 8 blocks in the cache / 2 ways = 4 sets, so 2 bits for index

iii. **(0.25 pt)** How many Tag bits?

14

20 - 2 - 4 = 14 bits

(c) We decide to parallelize a for loop across these 2 processors, but instead of using OpenMP, we have each thread do a strided memory access, where processor 0 handles even indices, while processor 1 handles odd indices. However, **the memory accesses are perfectly interleaved, i.e. the order of array accesses are still A[0], A[1], A[2], A[3]...**

```
# define ARR_LEN 32
// A is located at address 0xA0000
int A[ARR_LEN];

// Processor 0's loop
for (int i = 0; i < ARR_LEN; i += 2) {
    A[i] +=  i
}

// Processor 1's loop
for (int j = 1; j < ARR_LEN; j += 2) {
    A[j] += j
}
```

For each memory access below,

i. Classify it as a Hit or Miss. Snooping another cache for data is considered a coherency Miss.
ii. Since we are working in a multiprocessor system, classify the state of the block that the data accessed resides in **from the specified processors perspective**.

i. `A[0]` Read

A. (0.25 pt)

○ Hit

[✓ Miss]

B. (0.25 pt) State from proc 0's point of view.

○ M

○ O

[✓ E]

○ S

○ I

D. (0.25 pt) State from proc 1's point of view.

○ M

○ O

○ E

○ S

[✓ I]

ii. `A[0]` Write

A. (0.25 pt)

[✓ Hit]

Miss

C. (0.25 pt) State from proc 0's point of view.

[✓ M]

O

E

S

I

D. (0.25 pt) State from proc 1's point of view.

M

O

E

S

[✓ I]

iii. `A[1]` Read

A. (0.25 pt)

- ○ Hit
- [✓ Miss]

B. (0.25 pt) State from proc 0's point of view.

- ○ M
- [✓ O]
- ○ E
- ○ S
- ○ I

D. (0.25 pt) State from proc 1's point of view.

- ○ M
- ○ O
- ○ E
- [✓ S]
- ○ I

iv. `A[1]` Write

A. (0.25 pt)

- [✓ Hit]
- ○ Miss

B. (0.25 pt) State from proc 0's point of view.

- ○ M
- ○ O
- ○ E
- ○ S
- [✓ I]

D. (0.25 pt) State from proc 1's point of view.

- [✓ M]
- ○ O
- ○ E
- ○ S
- ○ I

v. `A[2]` Read

A. (0.25 pt)

- ○ Hit
- [✓ Miss]

B. (0.25 pt) State from proc 0's point of view.

- ○ M
- ○ O
- ○ E
- [✓ S]
- ○ I

D. (0.25 pt) State from proc 1's point of view.

- ○ M
- [✓ O]
- ○ E
- ○ S
- ○ I

vi. `A[2]` Write

A. (0.25 pt)

- [✓ Hit]
- Miss

B. (0.25 pt) State from proc 0's point of view.

- [✓ M]
- O
- E
- S
- I

D. (0.25 pt) State from proc 1's point of view.

- M
- O
- E
- S
- [✓ I]

vii. `A[3]` Read

A. (0.25 pt)

- ○ Hit
- [✓ Miss]

B. (0.25 pt) State from proc 0's point of view.

- ○ M
- [✓ O]
- ○ E
- ○ S
- ○ I

D. (0.25 pt) State from proc 1's point of view.

- ○ M
- ○ O
- ○ E
- [✓ S]
- ○ I

viii. `A[3]` Write

A. (0.25 pt)

- [✓ Hit]
- Miss

B. (0.25 pt) State from proc 0's point of view.

- M
- O
- E
- S
- [✓ I]

D. (0.25 pt) State from proc 1's point of view.

- [✓ M]
- O
- E
- S
- I

(d) **(2.0 pt)** What is the overall hit rate? Leave your answer as a fully simplified fraction.

1/2

The pattern above continues and repeats for all 8 blocks, giving us a 50% HR.

(e) **(2.0 pt)** What fraction of misses are coherency misses? Leave your answer as a fully simplified fraction.

3/4

Out of the 4 misses in each "access pattern block", 1 is compulsory, while the other 3 are coherency misses, so 75% of the overall misses.

(f) **(1.0 pt)** In total, how many times did we need to go to main memory to write-back?

0

As the array fits perfectly into the cache, we never need to evict an block and write-back, so 0.

(g) **(2.0 pt)** We want to avoid all the coherency misses, so we look to see if we can rewrite our code to optimize for cache performance. Which of the following methods will lead to a higher HR than that from the interleaved accesses?

- [✓] Letting processor 0 start and finish, then processor 1 starts and finishes
- [✓] Letting processor 1 start and finish, then processor 0 starts and finishes
- ☐ None of the other options

Both of these approaches would be better, since then there would be no coherency misses during the first processor's execution (load in the block, then hit on the other 3 WRW, so 75% HR). Then, the second processor would begin, but instead of compulsory missing, just coherency miss, but get the same HR pattern of MHHH for each block.

10. Virtual Memory

(a) We are working with a system with a 4 GiB physical memory, and 16 MiB virtual memory, and a page size of 4 KiB. For each PTE, we choose to store 12 bits of metadata (dirty bit, permissions).

i. For this part, assume we are working with a single level page table.

A. (0.5 pt) How many bits are in the page offset?

12

$\log_2(\text{size of page}) = \log_2(2^{12}) = 12$

B. (0.5 pt) How many bits are in the PPN?

20

$\log_2(\text{size of PM}) - \text{ pg offset bits} = \log_2(2^{32}) - 12 = 20$

C. (0.5 pt) How many bits are in the VPN?

12

$\log_2(\text{size of VM}) - \text{ pg offset bits} = \log_2(2^{24}) - 12 = 12$

D. (0.5 pt) How many bits are in a PTE?

32

$\text{\# PPN bits} + \text{\# metadata bits} = 20 + 12 = 32$

(b) For the rest of the problem, we will be working with a 2-level, hierarchical page table with no TLBs. Assume the VPN bits are split evenly between levels, so every PT at every level has the same number of PTEs.

i. For each page table level, calculate the number of PTEs in total, across all possible page tables in that level.

A. **(0.5 pt)** `L1` Number of PTEs

$\boxed{\textbf{64}}$

# L1 PTs * # of PTEs in L1 PT = 1 * 2^6 = 64

B. **(0.5 pt)** `L2` Number of PTEs

$\boxed{\textbf{4096}}$

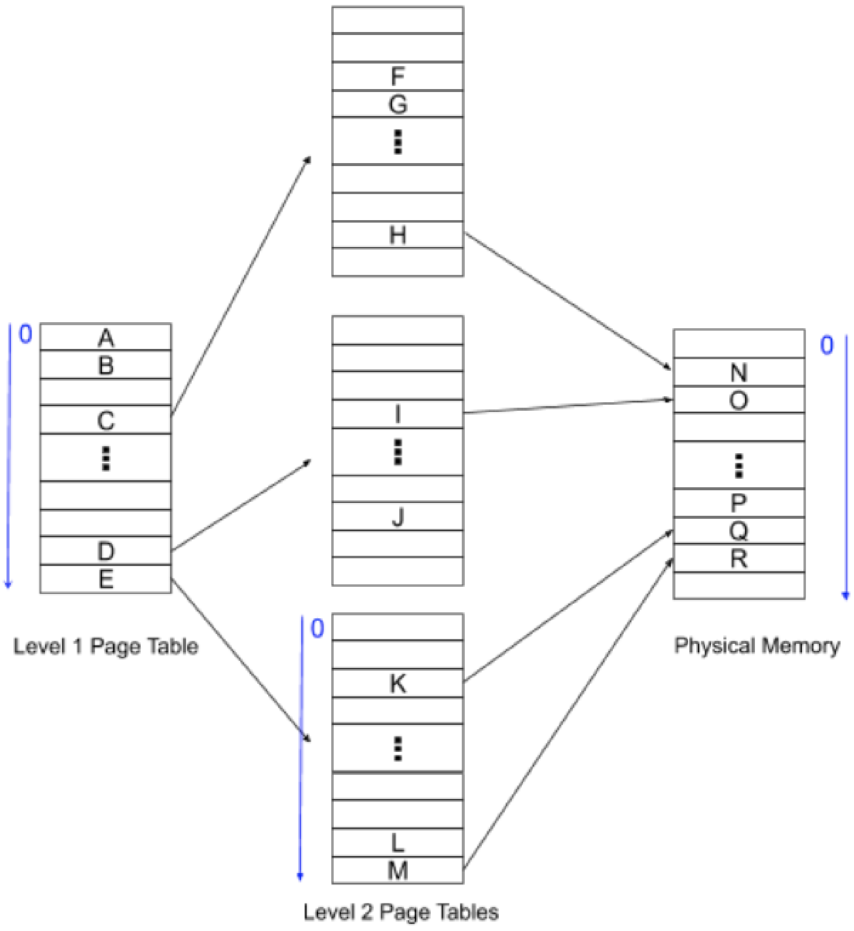
# L2 PTs * size of L2 PT = 2^6 * 2^6 = 2^12 = 4096

ii. **(2.0 pt)** Let's say the computer just started up, meaning that the page table has yet to allocate any pages in the physical memory. We then store 8 contiguous bytes to memory. In the worst case, how many page tables will we use?

3

The 8 contiguous bytes could span 2 pages in the worst case. This means we would need to allocate space for two L2 pages tables in addition to one L1 page table = 3 total PTs.

iii. Consider the following hierarchical page table. Regardless of your previous answers, assume that there are 64 PTEs in each page table. Arrows from one level to another represent a valid PTE for the page tables, or page for physical memory. The indices of the PTEs/pages are ordered from the top-down, i.e. the top-most refers to index 0. Only consider slots with a letter inside of them.



Multi-Level Page Table

Given the following PTEs accessed at each level, reconstruct the virtual and physical addresses in **hex**. If the data provided creates an invalid address, your answers should be N/A. For all memory accesses, we are attempting to access the 0th byte of the page.

A. `L1 PTE: E`
`L2 PTE: K`

B. **(2.0 pt)** Virtual Address

0xFC2000

Address breakdown: 6 bits for L1 VPN, 6 bits for L2 VPN, 12 bits for offset.

E is located at the bottom-most index = 63 (since by part (a)(i) the L1 PT has 64 entries, or by recomputing this, if we have 6 bits for the L1 VPN, then we have 64 PTEs). So the L1 VPN is 63, which points to the "bottom" L2 PT. There, we use PTE K, which is the 2nd from the top, so the L2 VPN = 2. Lastly, we need the offset. Since it is stated that we want the 0th byte of the page, our offset is 0.

Therefore, our VA is L1 VPN | L2 VPN | Offset = 111111 | 000010 | 00000000000 = 0xFC2000

C. **(2.0 pt)** Physical Address

0xFFFFD000

Address breakdown: 20 bits for PPN, 12 bits for offset

L2 PTE K has an arrow to page Q in physical memory, which is indexed 2rd from the bottom. To get this exact value, we need to know how many pages there are in physical memory: 2^32 / 2^12 = 2^20 pages. PPN = 0xFFFFF - 0x2 = 0xFFFFD. PA = PPN | Offset = 0xFFFFD000

D. `L1 PTE: C`
`L2 PTE: H`

E. (2.0 pt) Virtual Address

0x0FE000

Address breakdown: 6 bits for L1 VPN, 6 bits for L2 VPN, 12 bits for offset.

E is located at the bottom-most index = 63 (since by part (a)(i) the L1 PT has 64 entries, or by recomputing this, if we have 6 bits for the L1 VPN, then we have 64 PTEs). So the L1 VPN is 63, which points to the "bottom" L2 PT. There, we use PTE K, which is the 2nd from the top, so the L2 VPN = 2. Lastly, we need the offset. Since it is stated that we want the 0th byte of the page, our offset is 0.

Therefore, our VA is L1 VPN | L2 VPN | Offset = 000011 | 111110 | 000000000000 = 0x0FE000

F. (2.0 pt) Physical Address

0x00001000

Address breakdown: 20 bits for PPN, 12 bits for offset

L2 PTE K has an arrow to page Q in physical memory, which is indexed 2rd from the bottom. To get this exact value, we need to know how many pages there are in physical memory: 2^32 / 2^12 = 2^20 pages. PPN = 0xFFFFF - 0x2 = 0xFFFFD. PA = PPN | Offset = 0x00001000

G. `L1 PTE: E`
`L2 PTE: M`

H. **(2.0 pt)** Virtual Address

0xFFF000

Address breakdown: 6 bits for L1 VPN, 6 bits for L2 VPN, 12 bits for offset.

E is located at the bottom-most index = 63 (since by part (a)(i) the L1 PT has 64 entries, or by recomputing this, if we have 6 bits for the L1 VPN, then we have 64 PTEs). So the L1 VPN is 63, which points to the "bottom" L2 PT. There, we use PTE K, which is the 2nd from the top, so the L2 VPN = 2. Lastly, we need the offset. Since it is stated that we want the 0th byte of the page, our offset is 0.

Therefore, our VA is L1 VPN | L2 VPN | Offset = 111111 | 111111 | 000000000000 = 0xFFF000

I. **(2.0 pt)** Physical Address

0xFFFFE000

Address breakdown: 20 bits for PPN, 12 bits for offset

L2 PTE K has an arrow to page Q in physical memory, which is indexed 2rd from the bottom. To get this exact value, we need to know how many pages there are in physical memory: 2^32 / 2^12 = 2^20 pages. PPN = 0xFFFFF - 0x2 = 0xFFFFD. PA = PPN | Offset = 0xFFFFE000

iv. Given the following virtual addresses, first identify whether it is a page hit or page fault. If it is a hit, write out the sequence of "letters" that make up the path. If it page faults, then you must play the role of the OS, create the appropriate mapping given the available PTEs/physical pages, and leave your answer as the new path that will now be taken. **Format your answer without spaces between the letters e.g. ABC.**

A. VA = 0xF83000

B. (0.5 pt)

[✓ Page Hit]

○ Page Fault

C. (1.5 pt) `Path`

DIO

The VPNs are in the first 12 bits: 111110 | 000011. L1 VPN is 62 so we look at PTE D, which points to the "middle" L2 PT. L2 VPN is 3, so we look at PTE I. This finally points to page O.

D. VA = 0x0C3000

E. (0.5 pt)

○ Page Hit

[✓ Page Fault]

F. (1.5 pt) `Path`

CGP

The VPNs are in the first 12 bits: 000011 | 000011. L1 VPN is 3, so we look at PTE C, which points to the "top" L2 PT. L2 VPN is 3 as well, so we look at PTE G, where there is no arrow, so we page fault and assign it a valid mapping to page P.

11. TLP

In signal processing, the technique of **cross-correlation** (or *sliding dot-product*) is often used to determine the delay of a signal. In this problem, we will implement a cross-correlation function in C, parallelized of course! (You don't need to know anything about EE to ace this problem!) :0

The following function, `sliding_dot`, takes two arrays. `original` contains an array of length n and `other` contains `n + k` elements. We will shift `other` `k` times, and for each shift, compute its dot product with `original`. We then store these values in `result`.

We want to parallelize `sliding_dot` with OpenMP. Examine our attempts below and choose the behavior(s) you expect from each version. Assume the processor has four threads, 32B cache blocks, and `sizeof(int) = 4`.You may also assume that all calls to calloc() succeed.

Here is the template of the code where we will replace the `/* OPTIMIZED CODE */` with the code on each question.

```
int * sliding_dot(int * other, int * original, int n, int k) {
    int * result = (int *) calloc(k * sizeof(int))
    // shift the array
    for (int shift = 0; shift <= k; shift++) {
        /* OPTIMIZED CODE */
    }
    return result;
}
```

(a) (2.0 pt)

```
int * shifted = other + shift;
int dot_product = 0;
#pragma omp parallel for private(dot_product)
for (int i = 0; i < n; i++) {
        #pragma omp critical
    dot_product += shifted[i] * original[i];
}
result[shift] = dot_product;
```

How will this code behave?

☐ Always Correct, faster than serial

☐ Always Correct, slower than serial

[✓ Sometimes Incorrect]

(b) (2.0 pt)

```
int * shifted = other + shift;
int dot_product = 0;
#pragma omp parallel for reduction(+:dot_product)
for (int i = 0; i < n; i++) {
        #pragma omp critical
    dot_product += shifted[i] * original[i];
}
result[shift] = dot_product;
```

How will this code behave?

☐ Always Correct, faster than serial

[✓ Always Correct, slower than serial]

☐ Sometimes Incorrect

(c) (2.0 pt)

```
int * shifted = other + shift;
int dps[4] = {0,0,0,0};
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < n; i++) {
        int id = omp_get_thread_num();
        dps[id] += shifted[i] * original[i];
    }
}
result[shift] = dps[0] + dps[1] + dps[2] + dps[3];
```

How will this code behave?

[✓ Always Correct, faster than serial]

[✓ Always Correct, slower than serial]

☐ Sometimes Incorrect

The intended answer was that the code ends up being slower. Note that the due to false sharing and MOESI, elements of dps in the same cache block will have to continuously kick each other out, thus slowing down the code to slower than serial. Note that we specify the size of a cache block as 32 B.

However, this applies only when the array is block-aligned. In reality, we can expect that this array is randomly placed in a word-aligned location. Some of these will indeed be such that dps takes up two cache blocks, thus creating up to a 2x speedup. It is thus dependent on the exact location of dps, whether the parallelized version runs faster or slower than serial.

(d) (2.0 pt)

```
int * shifted = other + shift;
int dot_product = 0
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    dot_product += shifted[i] * original[i];
}
result[shift] = dot_product;
```

How will this code behave?

☐ Always Correct, faster than serial

☐ Always Correct, slower than serial

[✓ Sometimes Incorrect]

12. (a) DLP

In many applications, we wish to not only find the maximum element of an array, but the **index** of the maximum element, or the **argmax**. To do this quickly, we decide to utilize Data Level Parallelism. The following function, `argmax`, takes in an array, `arr`, and its length, `n`, and returns the index of the maximum value. If there exist multiple indices which contain the same maximum value, the function returns the first of these indices.

Use the provided "pseudo" SIMD intrinsics to fill in the function so it behaves as expected. The SIMD intrinsics operate on `vec` structs which represent SIMD vectors that contain 4 packed integers (exactly like Intel's `__m128i` structs). You may not need all lines.

SIMD Instructions:

```
vec sum_epi32 (vec a, vec b)
    // returns a + b
vec and_epi32 (vec a, vec b)
    // returns a & b
vec set_epi32 (int a)
    // return SIMD vector with all entries set to a
vec load_epi32 (int *a)
    // return SIMD vector with entries a[0], a[1], a[2], and a[3] respectively
int reducemax_epi32 (vec a)
    // return the value of the maximum int in vector a
vec maskeq_epi32 (vec a, int b)
    // return mask vector with 0xFFFFFFFF for indices where a is equal to b and 0 otherwise
int vfirst_epi32 (vec a)
    // return index of first entry with lowest bit set to 1

int argmax(int *arr, int n) {
    int curr, index = 0, running_max = -2147483648; // -2^31
    vec temp;

    /* Your Code Here */

    return index;
}
```

i. **(10.0 pt)** `/* Your Code Here */`

```
for (int i = 0; i < n / 4 * 4; i += 4) {
   temp = load_epi32(arr + i);
   curr = reducemax_epi32(temp);
   if (curr > running_max):
      temp = maskeq_epi32(temp, curr);
      index = i + vfirst_epi32(temp);
      running_max = curr;
}
for (int i = n / 4 * 4; i < n; i += 1) {
   if (arr[i] > running_max) {
      running_max = arr[i];
      index = i;
   }
}
```

(b) DLP

In many applications, we wish to not only find the maximum element of an array, but the **index** of the maximum element, or the **argmax**. To do this quickly, we decide to utilize Data Level Parallelism. The following function, `argmax`, takes in an array, `arr`, and its length, `n`, and returns the index of the maximum value. If there exist multiple indices which contain the same maximum value, the function returns the first of these indices.

Use the provided "pseudo" SIMD intrinsics to fill in the function so it behaves as expected. The SIMD intrinsics operate on `vec` structs which represent SIMD vectors that contain 4 packed integers (exactly like Intel's `__m128i` structs). You may not need all lines.

SIMD Instructions:

```
vec sum_epi32 (vec a, vec b)
    // returns a + b
vec and_epi32 (vec a, vec b)
    // returns a & b
vec set_epi32 (int a)
    // return SIMD vector with all entries set to a
vec load_epi32 (int *a)
    // return SIMD vector with entries a[0], a[1], a[2], and a[3] respectively
int reducemax_epi32 (vec a)
    // return the value of the maximum int in vector a
vec maskeq_epi32 (vec a, int b)
    // return mask vector with 0xFFFFFFFF for indices where a is equal to b and 0 otherwise
int vfirst_epi32 (vec a)
    // return index of first entry with lowest bit set to 1

int argmax(int *arr, int n) {
    int curr, index = 0, running_max = -2147483648; // -2^31
    vec temp;

    /* Your Code Here */

    return index;
}
```

i. **(10.0 pt)** `/* Your Code Here */`

```
for (int i = 0; i < n / 4 * 4; i += 4) {
   temp = load_epi32(arr + i);
   curr = reducemax_epi32(temp);
   if (curr > temp_max):
      temp = maskeq_epi32(temp, curr);
      index = i + vfirst_epi32(temp);
      temp_max = curr;
}
for (int i = n / 4 * 4; i < n; i += 1) {
   if (arr[i] > temp_max) {
      temp_max = arr[i];
      index = i;
   }
}
```

(c) DLP

In many applications, we wish to not only find the maximum element of an array, but the **index** of the maximum element, or the **argmax**. To do this quickly, we decide to utilize Data Level Parallelism. The following function, `argmax`, takes in an array, `arr`, and its length, `n`, and returns the index of the maximum value. If there exist multiple indices which contain the same maximum value, the function returns the first of these indices.

Use the provided "pseudo" SIMD intrinsics to fill in the function so it behaves as expected. The SIMD intrinsics operate on `vec` structs which represent SIMD vectors that contain 4 packed integers (exactly like Intel's `__m128i` structs). You may not need all lines.

SIMD Instructions:

```
vec sum_epi32 (vec a, vec b)
    // returns a + b
vec and_epi32 (vec a, vec b)
    // returns a & b
vec set_epi32 (int a)
    // return SIMD vector with all entries set to a
vec load_epi32 (int *a)
    // return SIMD vector with entries a[0], a[1], a[2], and a[3] respectively
int reducemax_epi32 (vec a)
    // return the value of the maximum int in vector a
vec maskeq_epi32 (vec a, int b)
    // return mask vector with 0xFFFFFFFF for indices where a is equal to b and 0 otherwise
int firstv_epi32 (vec a)
    // return index of first entry with lowest bit set to 1

int argmax(int *arr, int n) {
    int curr, index = 0, running_max = -2147483648; // -2^31
    vec temp;

    /* Your Code Here */

    return index;
}
```

i. **(10.0 pt)** `/* Your Code Here */`

```
for (int i = 0; i < n / 4 * 4; i += 4) {
   temp = load_epi32(arr + i);
   curr = reducemax_epi32(temp);
   if (curr > running_max):
      temp = maskeq_epi32(temp, curr);
      index = i + firstv_epi32(temp);
      running_max = curr;
}
for (int i = n / 4 * 4; i < n; i += 1) {
   if (arr[i] > running_max) {
      running_max = arr[i];
      index = i;
   }
}
```

(d) DLP

In many applications, we wish to not only find the maximum element of an array, but the **index** of the maximum element, or the **argmax**. To do this quickly, we decide to utilize Data Level Parallelism. The following function, `argmax`, takes in an array, `arr`, and its length, `n`, and returns the index of the maximum value. If there exist multiple indices which contain the same maximum value, the function returns the first of these indices.

Use the provided "pseudo" SIMD intrinsics to fill in the function so it behaves as expected. The SIMD intrinsics operate on `vec` structs which represent SIMD vectors that contain 4 packed integers (exactly like Intel's `__m128i` structs). You may not need all lines.

SIMD Instructions:

```
vec sum_epi32 (vec a, vec b)
    // returns a + b
vec and_epi32 (vec a, vec b)
    // returns a & b
vec set_epi32 (int a)
    // return SIMD vector with all entries set to a
vec load_epi32 (int *a)
    // return SIMD vector with entries a[0], a[1], a[2], and a[3] respectively
int reducemax_epi32 (vec a)
    // return the value of the maximum int in vector a
vec mask_epi32 (vec a, int b)
    // return mask vector with 0xFFFFFFFF for indices where a is equal to b and 0 otherwise
int vfirst_epi32 (vec a)
    // return index of first entry with lowest bit set to 1

int argmax(int *arr, int n) {
    int curr, index = 0, running_max = -2147483648; // -2^31
    vec temp;

    /* Your Code Here */

    return index;
}
```

i. **(10.0 pt)** `/* Your Code Here */`

```
for (int i = 0; i < n / 4 * 4; i += 4) {
   temp = load_epi32(arr + i);
   curr = reducemax_epi32(temp);
   if (curr > running_max):
      temp = mask_epi32(temp, curr);
      index = i + vfirst_epi32(temp);
      running_max = curr;
}
for (int i = n / 4 * 4; i < n; i += 1) {
   if (arr[i] > running_max) {
      running_max = arr[i];
      index = i;
   }
}
```

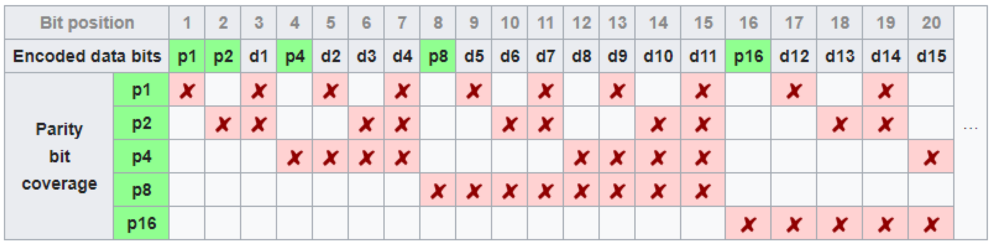
13. ECC, RAID

(a) **(0.5 pt)** If we want to tolerate 1 disk failure, which version(s) of RAID should we use?

- ☐ RAID 0
- [✓ RAID 1]
- [✓ RAID 4]
- [✓ RAID 5]
- ☐ None of the other options

(b) **(0.5 pt)** Which version of RAID is fastest for small random writes?

- [✓ RAID 0]
- ☐ RAID 1
- ☐ RAID 4
- ☐ RAID 5
- ☐ None of the other options



Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	✗		✗		✗		✗		✗		✗		✗		✗		✗		✗		
	p2		✗	✗			✗	✗			✗	✗			✗	✗			✗	✗		…
	p4				✗	✗	✗	✗					✗	✗	✗	✗					✗	
	p8								✗	✗	✗	✗	✗	✗	✗	✗						
	p16																✗	✗	✗	✗	✗	

Parity Bits

(c) i. **(2.0 pt)** What is the minimum Hamming distance necessary to allow for Single Error Detection and Single Error Correction?

3

ii. **(2.0 pt)** How many bits of data can we cover if we have 7 parity bits?

120

$2^7 - 7 - 1 = 120$

iii. Consider the following codeword we wish to send: `0b10110100`.

A. **(3.0 pt)** What is the Hamming ECC we should send over to ensure that we can detect and correct a 1-bit error?

0b001001110100

p1: 1, 0, 1, 0, -> 0 p2: 1, 1, 1, 1, 0 -> 0 p4: 0, 1, 1, 0 -> 0 p8: 0, 1, 0, 0, -> 1

ECC is 0b001001110100

B. (2.0 pt) We receive the word, but notice something is off. We are unable to see the contents of the bits, but we are told that only the parity check for p1 failed. Given this information, which bit position holds the error? (Remember that indices begin at 1 for ECC)

1

Since only p1 had an odd parity, the error must have been in the only bit who is covered solely by p1: bit 1.

No more questions.