CS 61C Summer 2020 — Number Representation

Discussion 1: June 22th, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Depending on the context, the same sets of bits may represent different things.

1.2 It is possible to get an overflow error in Two's Complement when adding numbers of opposite signs.

1.3 If you interpret a N bit Two's complement number as an unsigned number, negative numbers would be smaller than positive numbers.

1.4 If you interpret an N bit Bias notation number as an unsigned number (assume there are negative numbers for the given bias), negative numbers would be smaller than positive numbers.

2 Unsigned Integers

2.1 If we have an n-digit unsigned numeral $d_{n-1}d_{n-2}\dots d_0$ in *radix* (or *base*) r, then the value of that numeral is $\sum_{i=0}^{n-1} r^i d_i$, which is just fancy notation to say that instead of a 10's or 100's place we have an r's or r^2's place. For the three radices binary, decimal, and hex, we just let r be 2, 10, and 16, respectively.

Let's try this by hand. Recall that our preferred tool for writing large numbers is the IEC prefixing system:

Ki (Kibi) = 2^{10}	Gi (Gibi) = 2^{30}	Pi (Pebi) = 2^{50}	Zi (Zebi) = 2^{70}
Mi (Mebi) = 2^{20}	Ti (Tebi) = 2^{40}	Ei (Exbi) = 2^{60}	Yi (Yobi) = 2^{80}

(a) Convert the following numbers from their initial radix into the other two common radices:

1. 0b10010011
2. 63
3. 0b00100100
4. 0
5. 39
6. 437
7. 0x0123

(b) Convert the following numbers from hex to binary:

1. 0xD3AD
2. 0xB33F
3. 0x7EC4

(c) Write the following numbers using IEC prefixes:

- 2^{16}
- 2^{27}
- 2^{43}
- 2^{36}
- 2^{34}
- 2^{61}
- 2^{47}
- 2^{59}

(d) Write the following numbers as powers of 2:

- 2 Ki
- 512 Ki
- 16 Mi
- 256 Pi
- 64 Gi
- 128 Ei

3 Signed Integers

3.1 Unsigned binary numbers work for natural numbers, but many calculations use negative numbers as well. To deal with this, a number of different schemes have been used to represent signed numbers, but we will focus on two's complement, as it is the standard solution for representing signed integers.

- Most significant bit has a negative value, all others are positive. So the value of an n-digit two's complement number can be written as $\sum_{i=0}^{n-2} 2^i d_i - 2^{n-1} d_{n-1}$.
- Otherwise exactly the same as unsigned integers.
- A neat trick for flipping the sign of a two's complement number: flip all the bits and add 1.

- Addition is exactly the same as with an unsigned number.
- Only one 0, and it's located at 0b0.

For questions (a) through (c), assume an 8-bit integer and answer each one for the case of an unsigned number, biased number with a bias of -127, and two's complement number. Indicate if it cannot be answered with a specific representation.

(a) What is the largest integer? What is the result of adding one to that number?

1. Unsigned?
2. Biased?
3. Two's Complement?

(b) How would you represent the numbers 0, 1, and -1?

1. Unsigned?
2. Biased?
3. Two's Complement?

(c) How would you represent 17 and -17?

1. Unsigned?
2. Biased?
3. Two's Complement?

(d) What is the largest integer that can be represented by *any* encoding scheme that only uses 8 bits?

(e) Prove that the two's complement inversion trick is valid (i.e. that x and $\overline{x} + 1$ sum to 0).

(f) Explain where each of the three radices shines and why it is preferred over other bases in a given context.

4 Arithmetic and Counting

4.1 Addition and subtraction of binary/hex numbers can be done in a similar fashion as with decimal digits by working right to left and carrying over extra digits to the next place. However, sometimes this may result in an overflow if the number of bits can no longer represent the true sum. Overflow occurs if and only if two numbers with the same sign are added and the result has the opposite sign.

(a) Compute the decimal result of the following arithmetic expressions involving 6-bit Two's Complement numbers as they would be calculated on a computer. Do any of these result in an overflow? Are all these operations possible?

1. 0b011001 − 0b000111
2. 0b100011 + 0b111010
3. 0x3B + 0x06
4. 0xFF − 0xAA

(b) What is the least number of bits needed to represent the following ranges using any number representation scheme.

1. 0 to 256
2. -7 to 56
3. 64 to 127 and -64 to -127
4. Address every byte of a 12 TiB chunk of memory

CS 61C Summer 2020

C Basics

Discussion 2: June 24, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 True or False: C is a pass-by-value language.

1.2 What is a pointer? What does it have in common to an array variable?

1.3 If you try to dereference a variable that is not a pointer, what will happen? What about when you free one?

1.4 When should you use the heap over the stack? Do they grow?

2 C

C is syntactically similar to Java, but there are a few key differences:

1. C is function-oriented, not object-oriented; there are no objects.
2. C does not automatically handle memory for you.
   - Stack memory, or *things that are not manually allocated*: data is garbage immediately after the *function in which it was defined* returns.
   - Heap memory, or *things allocated with* `malloc`, `calloc`, *or* `realloc`: data is freed only when the programmer explicitly frees it!
   - There are two other sections of memory that we learn about in this course, *static* and *code*, but we'll get to those later.
   - In any case, allocated memory always holds garbage until it is initialized!
3. C uses pointers explicitly. If `p` is a pointer, then `*p` tells us to use the value that `p` points to, rather than the value of `p`, and `&x` gives the address of `x` rather than the value of `x`.

   On the left is the memory represented as a box-and-pointer diagram.

   On the right, we see how the memory is really represented in the computer.

Let's assume that `int* p` is located at `0xF9320904` and `int x` is located at `0xF93209B0`. As we can observe:

- `*p` evaluates to `0x2A` (42_{10}).
- `p` evaluates to `0xF93209AC`.
- `x` evaluates to `0x61C`.
- `&x` evaluates to `0xF93209B0`.

Let's say we have an `int **pp` that is located at `0xF9320900`.

2.1 What does `pp` evaluate to? How about `*pp`? What about `**pp`?

2.2 The following functions are syntactically-correct C, but written in an incomprehensible style. Describe the behavior of each function in plain English.

(a) Recall that the ternary operator evaluates the condition before the ? and returns the value before the colon (:) if true, or the value after it if false.

```
1  int foo(int *arr, size_t n) {
2      return n ? arr[0] + foo(arr + 1, n - 1) : 0;
3  }
```

(b) Recall that the negation operator, !, returns 0 if the value is non-zero, and 1 if the value is 0. The ~ operator performs a *bitwise not* (NOT) operation.

```
1  int bar(int *arr, size_t n) {
2      int sum = 0, i;
3      for (i = n; i > 0; i--)
4          sum += !arr[i - 1];
5      return ~sum + 1;
6  }
```

(c) Recall that ^ is the *bitwise exclusive-or* (XOR) operator.

```
1  void baz(int x, int y) {
2      x = x ^ y;
```

```
3       y = x ^ y;
4       x = x ^ y;
5   }
```

(d) (Bonus: How do you write the *bitwise exclusive-nor* (XNOR) operator in C?)

3 Programming with Pointers

3.1 Implement the following functions so that they work as described.

(a) Swap the value of two `int`s. *Remain swapped after returning from this function.*

```
void swap(
```

(b) Return the number of bytes in a string. *Do not use* `strlen`.

```
int mystrlen(
```

3.2 The following functions may contain logic or syntax errors. Find and correct them.

(a) Returns the sum of all the elements in `summands`.

```
1   int sum(int* summands) {
2       int sum = 0;
3       for (int i = 0; i < sizeof(summands); i++)
4           sum += *(summands + i);
5       return sum;
6   }
```

(b) Increments all of the letters in the `string` which is stored at the front of an array of arbitrary length, `n >= strlen(string)`. Does not modify any other parts of the array's memory.

```
1   void increment(char* string, int n) {
2       for (int i = 0; i < n; i++)
```

```
        *(string + i)++;
}
```

(c) Copies the string `src` to `dst`.

```
void copy(char* src, char* dst) {
    while (*dst++ = *src++);
}
```

(d) Overwrites an input string `src` with "61C is awesome!" if there's room. Does nothing if there is not. Assume that `length` correctly represents the length of `src`.

```
void cs61c(char* src, size_t length) {
    char *srcptr, replaceptr;
    char replacement[16] = "61C is awesome!";
    srcptr = src;
    replaceptr = replacement;
    if (length >= 16) {
        for (int i = 0; i < 16; i++)
            *srcptr++ = *replaceptr++;
    }
}
```

4 Memory Management

4.1 For each part, choose one or more of the following memory segments where the data could be located: **code, static, heap, stack**.

(a) Static variables

(b) Local variables

(c) Global variables

(d) Constants

(e) Machine Instructions

(f) Result of `malloc`

(g) String Literals

4.2 Write the code necessary to allocate memory on the heap in the following scenarios

(a) An array `arr` of k integers

(b) A string `str` containing p characters

(c) An $n \times m$ matrix `mat` of integers initialized to zero.

4.3 What's the main issue with the code snippet seen here? (Hint: `gets()` is a function that reads in user input and stores it in the array given in the argument.)

```
1  char* foo() {
2      char* buffer[64];
3      gets(buffer);
4
5      char* important_stuff = (char*) malloc(11 * sizeof(char));
6
7      int i;
8      for (i = 0; i < 10; i++) important_stuff[i] = buffer[i];
9      important_stuff[i] = "\0";
10     return important_stuff;
11 }
```

Suppose we've defined a linked list **struct** as follows. Assume `*lst` points to the first element of the list, or is `NULL` if the list is empty.

```
struct ll_node {
    int first;
    struct ll_node* rest;
}
```

4.4 Implement `prepend`, which adds one new `value` to the front of the linked list. Hint: why use ll_node $**lst$ instead of ll_node$*lst$?

```
void prepend(struct ll_node** lst, int value)
```

4.5 Implement `free_ll`, which frees all the memory consumed by the linked list.

```
void free_ll(struct ll_node** lst)
```

CS 61C Summer 2020

Floating Point

Discussion 3: June 29, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 True or False. The goals of floating point are to have a large range of values, a low amount of precision, and real arithmetic results

1.2 True or False. The distance between floating point numbers increase as the absolute value of the numbers increase.

1.3 True or False. Floating Point addition is associative.

2 Floating Point

The IEEE 754 standard defines a binary representation for floating point values using three fields.

- The *sign* determines the sign of the number (0 for positive, 1 for negative).
- The *exponent* is in **biased notation**. For instance, the bias is -127 which comes from -$(2^{8-1}-1)$ for single-precision floating point numbers.
- The *significand* or *mantissa* is akin to unsigned integers, but used to store a fraction instead of an integer.

The below table shows the bit breakdown for the single precision (32-bit) representation. The leftmost bit is the MSB and the rightmost bit is the LSB.

1	8	23
Sign	Exponent	Mantissa/Significand/Fraction

For normalized floats:

Value $= (-1)^{Sign} * 2^{Exp+Bias} * 1.\textbf{significand}_2$

For denormalized floats:

Value $= (-1)^{Sign} * 2^{Exp+Bias+1} * 0.\textbf{significand}_2$

Exponent	Significand	Meaning
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

Note that in the above table, our exponent has values from 0 to 255. When translating between binary and decimal floating point values, we must remember that there is a bias for the exponent.

2.1 How many zeroes can be represented using a float?

2.2 What is the largest finite positive value that can be stored using a single precision float?

2.3 What is the smallest positive value that can be stored using a single precision float?

2.4 What is the smallest positive normalized value that can be stored using a single precision float?

2.5 Cover the following single-precision floating point numbers from binary to decimal or from decimal to binary. You may leave your answer as an expression.

- 0x00000000
- 8.25
- 0x00000F00
- 39.5625
- 0xFF94BEEF
- $-\infty$

3 More Floating Point Representation

Not every number can be represented perfectly using floating point. For example, $\frac{1}{3}$ can only be approximated and thus must be rounded in any attempt to represent it. For this question, we will only look at positive numbers.

3.1 What is the next smallest number larger than 2 that can be represented completely?

3.2 What is the next smallest number larger than 4 that can be represented completely?

3.3 Define stepsize to be the distance between some value x and the smallest value larger than x that can be completely represented. What is the step size for 2? 4?

3.4 Now let's see if we can generalize the stepsize for normalized numbers (we can do so for denorms as well, but we won't in this question). If we are given a normalized number that is not the largest representable normalized number with exponent value x and with significand value y, what is the stepsize at that value? Hint: There are 23 significand bits.

3.5 Now let's apply this technique. What is the largest odd number that we can represent? Part 4 should be very useful in finding this answer.

CS 61C Summer 2020 — RISC-V Intro & Control Flow

Discussion 4: July 1, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 After calling a function and having that function return, the `t` registers may have been changed during the execution of the function, while `a` registers cannot.

1.2 Let `a0` point to the start of an array `x`. `lw s0, 4(a0)` will always load `x[1]` into `s0`.

1.3 Assuming no compiler or operating system protections, it is possible to have the code jump to data stored at `0(a0)` and execute instructions from there.

1.4 Adding the character `'d'` to the address of an integer array would get you the element at index 25 of that array (assuming the array is large enough).

1.5 Calling `jalr` is a shorthanded expression for `jal` that jumps to the specified label and does not store a return address anywhere.

1.6 Calling `j label` does the exact same thing as calling `jal label`.

2 RISC-V: A Rundown

RISC-V is an assembly language, which is comprised of simple instructions that each do a single task such as addition or storing a chunk of data to memory.

For example, on the left is a line of C code and on the right is a chunk of RISC-V code that accomplishes the same thing.

```
                                    // x -> s0, &y -> s1
int x = 5, y[2];                    addi s0, x0, 5
y[0] = x;                           sw   s0, 0(s1)
y[1] = x * x;                       mul  t0, s0, s0
                                    sw   t0, 4(s1)
```

2.1 Can you figure out what each line in the RISC-V code is doing?

3 Registers

In RISC-V, we have two methods of storing data: main memory and registers. Registers are much faster than using main memory, but are very limited in space (32 bits each). Note that you should ALWAYS use the named registers (e.g. `s0` rather than `x8`).

Register(s)	Alt.	Description
x0	zero	The zero register, always zero
x1	ra	The return address register, stores where functions should return
x2	sp	The stack pointer, where the stack ends
x5-x7, x28-x31	t0-t6	The temporary registers
x8-x9, x18-x27	s0-s11	The saved registers
x10-x17	a0-a7	The argument registers, a0-a1 are also return value

3.1 Can you convert each instruction's registers to the other form?

```
add s0, zero, a1     -->
or  x18, x1, x30     -->
```

4 Basic Instructions

For your reference, here are some of the basic instructions for arithmetic operations and dealing with memory (Note: ARG1 is argument register 1, ARG2 is argument register 2, and DR is destination register):

[inst]	[destination register] [argument register 1] [argument register 2]
add	Adds the two argument registers and stores in destination register
xor	Exclusive or's the two argument registers and stores in destination register

mul	Multiplies the two argument registers and stores in destination register
sll	Logical left shifts ARG1 by ARG2 and stores in DR
srl	Logical right shifts ARG1 by ARG2 and stores in DR
sra	Arithmetic right shifts ARG1 by ARG2 and stores in DR
slt/u	If ARG1 < ARG2, stores 1 in DR, otherwise stores 0, u does unsigned comparison
[inst]	[register] [offset]([register containing base address])
sw	Stores the contents of the register to the address+offset in memory
lw	Takes the contents of address+offset in memory and stores in the register
[inst]	[argument register 1] [argument register 2] [label]
beq	If ARG1 == ARG2, moves to label
bne	If ARG1 != ARG2, moves to label
[inst]	[destination register] [label]
jal	Stores the next instruction's address into DR and moves to label

You may also see that there is an "i" at the end of certain instructions, such as addi, slli, etc. This means that ARG2 becomes an "immediate" or an integer instead of using a register. There are also immediates in some other instructions such as **sw** and **lw**. NOTE: The size of an immediate in any given instruction depends on what type of instruction it is (more on this soon!).

4.1 Assume we have an array in memory that contains `int* arr = {1,2,3,4,5,6,0}`. Let register `s0` hold the address of the element at index 0 in arr. You may assume integers are four-bytes and our values are word-aligned. What do the snippets of RISC-V code do? Assume that all the instructions are run one after the other in the same context.

```
a) lw   t0, 12(s0)        -->

b) sw   t0, 16(s0)        -->

b) slli t1, t0, 2
   add  t2, s0, t1
   lw   t3, 0(t2)         -->
   addi t3, t3, 1
   sw   t3, 0(t2)

c) lw   t0, 0(s0)
   xori t0, t0, 0xFFF     -->
   addi t0, t0, 1
```

5 C to RISC-V

5.1 Translate between the C and RISC-V verbatim.

C	RISC-V

`// s0 -> a, s1 -> b`
`// s2 -> c, s3 -> z`
`int a = 4, b = 5, c = 6, z;`
`z = a + b + c + 10;`	

`// s0 -> int * p = intArr;`
`// s1 -> a;`
`*p = 0;`
`int a = 2;`
`p[1] = p[a] = a;`	

`// s0 -> a, s1 -> b`
`int a = 5, b = 10;`
`if(a + a == b) {`
`  a = 0;`
`} else {`
`  b = a - 1;`
`}`	

	`  addi s0, x0, 0`
	`  addi s1, x0, 1`
	`  addi t0, x0, 30`
	`loop:`
	`  beq  s0, t0, exit`
	`  add  s1, s1, s1`
	`  addi s0, s0, 1`
	`  jal  x0, loop`
	`exit:`

`// s0 -> n, s1 -> sum`
`// assume n > 0 to start`
`for(int sum = 0; n > 0; n--) {`
`  sum += n;`
`}`	

6 RISC-V with Arrays and Lists

Comment what each code block does. Each block runs in isolation. Assume that there is an array, `int arr[6] = {3, 1, 4, 1, 5, 9}`, which starts at memory address `0xBFFFFF00`, and a linked list struct (as defined below), `struct ll* lst`, whose first element is located at address `0xABCD0000`. Let `s0` contain `arr`'s address `0xBFFFFF00`, and let `s1` contain `lst`'s address `0xABCD0000`. You may assume integers and pointers are 4 bytes and that structs are tightly packed. Assume that `lst`'s last node's `next` is a NULL pointer to memory address `0x00000000`.

```
struct ll {
    int val;
    struct ll* next;
}
```

6.1
```
lw  t0, 0(s0)
lw  t1, 8(s0)
add t2, t0, t1
sw  t2, 4(s0)
```

6.2
```
loop: beq  s1, x0, end
      lw   t0, 0(s1)
      addi t0, t0, 1
      sw   t0, 0(s1)
      lw   s1, 4(s1)
      jal  x0, loop
 end:
```

6.3
```
       add  t0, x0, x0
loop:  slti t1, t0, 6
       beq  t1, x0, end
       slli t2, t0, 2
       add  t3, s0, t2
       lw   t4, 0(t3)
       sub  t4, x0, t4
       sw   t4, 0(t3)
       addi t0, t0, 1
       jal  x0, loop
 end:
```

7 RISC-V Calling Conventions

7.1 How do we pass arguments into functions?

7.2 How are values returned by functions?

7.3 What is `sp` and how should it be used in the context of RISC-V functions?

7.4 Which values need to saved by the caller, before jumping to a function using `jal`?

7.5 Which values need to be restored by the callee, before returning from a function?

7.6 In a bug-free program, which registers are guaranteed to be the same after a function call? Which registers aren't guaranteed to be the same?

8 Writing RISC-V Functions

8.1 Write a function `sumSquare` in RISC-V that, when given an integer `n`, returns the summation below. If `n` is not positive, then the function returns 0.

$$n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$$

For this problem, you are given a RISC-V function called `square` that takes in a single integer and returns its square.

First, let's implement the meat of the function: the squaring and summing. We will be abiding by the caller/callee convention, so in what register can we expect the parameter `n`? What registers should hold `square`'s parameter and return value? In what register should we place the return value of `sumSquare`?

8.2 Since `sumSquare` is the callee, we need to ensure that it is not overriding any registers that the caller may use. Given your implementation above, write a prologue and epilogue to account for the registers you used.

9 More Translating between C and RISC-V

9.1 Translate between the RISC-V code to C. What is this RISC-V function computing? Assume no stack or memory-related issues, and assume no negative inputs.

C	RISC-V

`// a0 -> x, a1 -> y,`
`// t0 -> result`
	`Func: addi t0 x0 1`
	`Loop: beq a1 x0 Done`
	`mul t0 t0 a0`
	`addi a1 a1 -1`
	`jal x0 Loop`
	`Done: add a0 t0 x0`
	`jr ra`

CS 61C Summer 2020 — CALL, RISC-V Procedures

Discussion 5: July 6, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 The compiler may output pseudoinstructions.

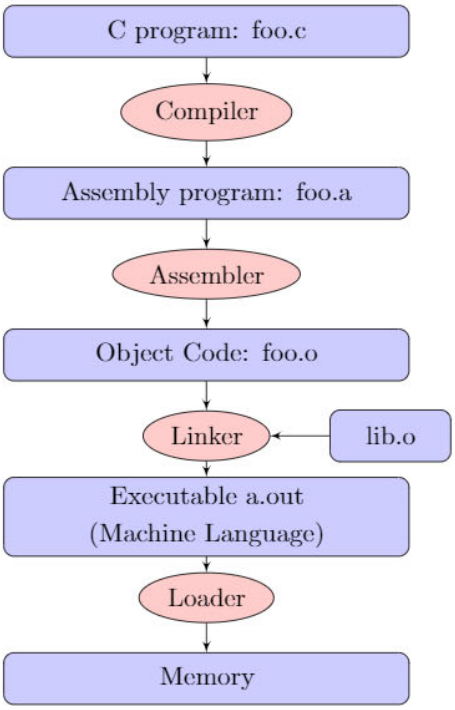
1.2 The main job of the assembler is to generate optimized machine code.

1.3 The object files produced by the assembler are only moved, not edited, by the linker.

1.4 The destination of all jump instructions is completely determined after linking.

2 CALL

The following is a diagram of the CALL stack detailing how C programs are built and executed by machines:



2.1 What is the Stored Program concept and what does it enable us to do?

2.2 How many passes through the code does the Assembler have to make? Why?

2.3 Describe the six main parts of the object files outputted by the Assembler (Header, Text, Data, Relocation Table, Symbol Table, Debugging Information).

2.4 Which step in CALL resolves relative addressing? Absolute addressing?

3 Assembling RISC-V

Let's say that we have a C program that has a single function `sum` that computes the sum of an array. We've compiled it to RISC-V, but we haven't assembled the RISC-V code yet.

```
1 .import print.s            # print.s is a different file
2 .data
3 array: .word 1 2 3 4 5
4 .text
5 sum:    la t0, array
6         li t1, 4
```

```
7           mv t2, x0
8   loop:   blt t1, x0, end
9           slli t3, t1, 2
10          add t3, t0, t3
11          lw t3, 0(t3)
12          add t2, t2, t3
13          addi t1, t1, -1
14          j loop
15  end:    mv a0, t2
16          jal ra, print_int   # Defined in print.s
```

3.1 Which lines contain pseudoinstructions that need to be converted to regular RISC-V instructions?

3.2 For the branch/jump instructions, which labels will be resolved in the first pass of the assembler? The second?

Let's assume that the code for this program starts at address `0x00061C00`. The code below is labelled with its address in memory (think: why is there a jump of 8 between the first and second lines?).

```
1   0x00061C00: sum:    la t0, array
2   0x00061C08:         li t1, 4
3   0x00061C0C:         mv t2, x0
4   0x00061C10: loop:   blt t1, x0, end
5   0x00061C14:         slli t3, t1, 2
6   0x00061C18:         add t3, t0, t3
7   0x00061C1C:         lw t3, 0(t3)
8   0x00061C20:         add t2, t2, t3
9   0x00061C24:         addi t1, t1, -1
10  0x00061C28:         j loop
11  0x00061C2C: end:    mv a0, t2
12  0x00061C30:         jal ra, print_int
```

3.3 What is in the symbol table after the assembler makes its passes?

3.4 What's contained in the relocation table?

4 RISC-V Addressing

We have several *addressing modes* to access memory (immediate not listed):

1. Base displacement addressing adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb).
2. PC-relative addressing uses the PC and adds the immediate value of the instruction (multiplied by 2) to create an address (used by branch and jump instructions).
3. Register Addressing uses the value in a register as a memory address. For instance, `jalr`, `jr`, and `ret`, where `jr` and `ret` are just pseudoinstructions that get converted to `jalr`.

4.1 What is the range of 32-bit instructions that can be reached from the current PC using a branch instruction?

4.2 What is the maximum range of 32-bit instructions that can be reached from the current PC using a jump instruction?

4.3 Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V green card!).

```
0x002cff00: loop: add t1, t2, t0          |_______|_______|_______|_______|_______|__0x33__|
0x002cff04:       jal ra, foo             |_____________________________|_____________|__0x6F__|
0x002cff08:       bne t1, zero, loop      |_______|_______|_______|_______|_______|__0x63__|
...
0x002cff2c: foo:  jr ra                 ra = ____________________________________
```

CS 61C Summer 2020

Logic, SDS, FSM

Discussion 6: July 8, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Simplifying boolean logic expressions has no effect on the performance of the hardware implementation.

1.2 The fewer gates the faster the circuit (assuming they all have the same delay).

1.3 It is allowed for clock-to-q plus the setup time to be greater than one clock cycle.

2 Boolean Logic

In digital electronics, it is often important to get certain outputs based on your inputs, as laid out by a truth table. Truth tables map directly to Boolean expressions, and Boolean expressions map directly to logic gates. However, in order to minimize the number of logic gates needed to implement a circuit, it is often useful to simplify long Boolean expressions.

We can simplify expressions using the nine key laws of Boolean algebra:

Name	AND Form	OR form
Commutative	$AB = BA$	$A + B = B + A$
Associative	$AB(C) = A(BC)$	$A + (B + C) = (A + B) + C$
Identity	$1A = A$	$0 + A = A$
Null	$0A = 0$	$1 + A = 1$
Absorption	$A(A + B) = A$	$A + AB = A$
Distributive	$(A + B)(A + C) = A + BC$	$A(B + C) = AB + AC$
Idempotent	$A(A) = A$	$A + A = A$
Inverse	$A(\overline{A}) = 0$	$A + \overline{A} = 1$
De Morgan's	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}(\overline{B})$

2 *Logic, SDS, FSM*

2.1 Use multiple iterations of De Morgan's laws to prove the identity $\bar{A} + AB = \bar{A} + B$.

2.2 Simplify the following Boolean expressions:

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

(b) $\bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + AB\bar{C} + A\bar{B}\bar{C} + ABC + A\bar{B}C$

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

(d) $\overline{A}(A + B) + (B + AA)(A + \overline{B})$

3 Logic Gates

3.1 Label the following logic gates:

3.2 Convert the following to boolean expressions on input signals A and B:

(a) NAND

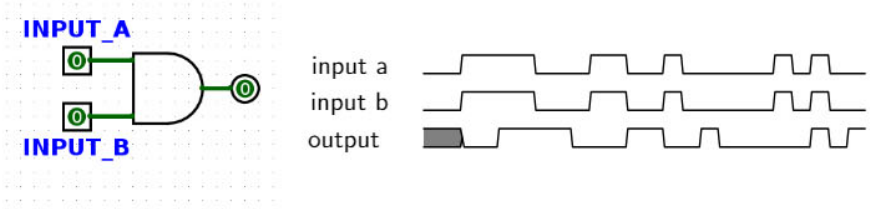
(b) XOR

(c) XNOR

3.3 Create an AND gate using only NAND gates.

4 State Intro

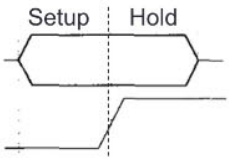
There are two basic types of circuits: combinational logic circuits and state elements. **Combinational logic** circuits simply change based on their inputs after whatever propagation delay is associated with them. For example, if an AND gate (pictured below) has an associated propagation delay of 2ps, its output will change based on its input as follows:



You should notice that the output of this AND gate always changes 2ps after its inputs change.

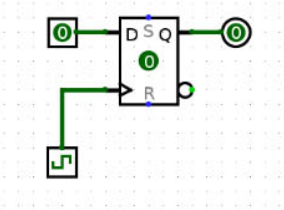
State elements, on the other hand, can *remember* their inputs even after the inputs change. State elements change value based on a clock signal. A rising edge-triggered register, for example, samples its input at the rising edge of the clock (when the clock signal goes from 0 to 1).

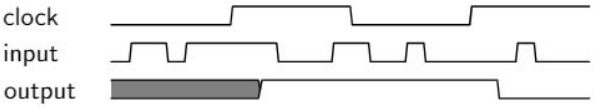
Like logic gates, registers also have a delay associated with them before their output will reflect the input that was sampled. This is called the **clk-to-q** delay. ("Q" often indicates output). This is the time between the rising edge of the clock signal and the time the register's output reflects the input change.



The input to the register samples has to be stable for a certain amount of time around the rising edge of the clock for the input to be sampled accurately. The amount of time before the rising edge the input must be stable is called the **setup** time, and the time after the rising edge the input must be stable is called the **hold** time. Hold time is generally included in clk-to-q delay, so clk-to-q time will usually be greater than or equal to hold time. Logically, the fact that clk-to-q $\geq$ hold time makes sense since it only takes clk-to-q seconds to copy the value over, so there's no need to have the value fed into the register for any longer.

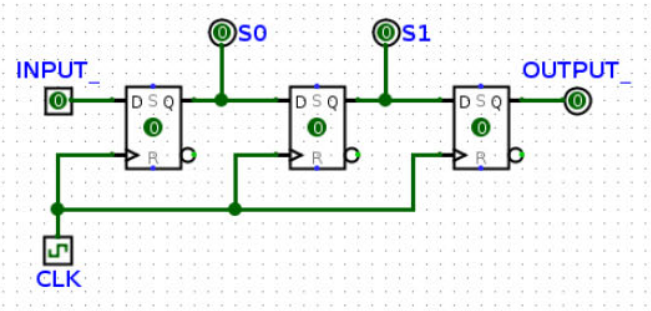
For the following register circuit, assume **setup** of 2.5ps, **hold** time of 1.5ps, and a **clk-to-q** time of 1.5ps. The clock signal has a period of 13ps.

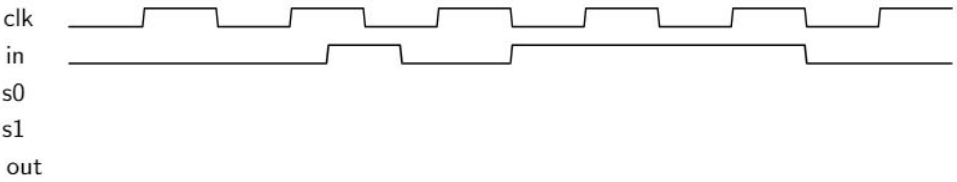


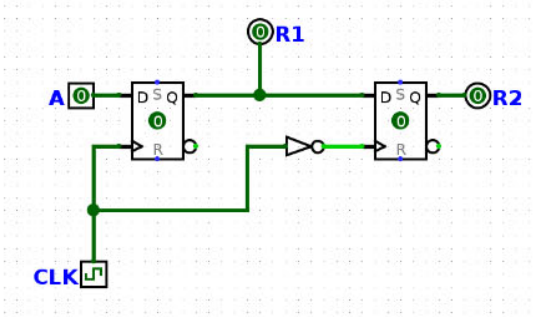


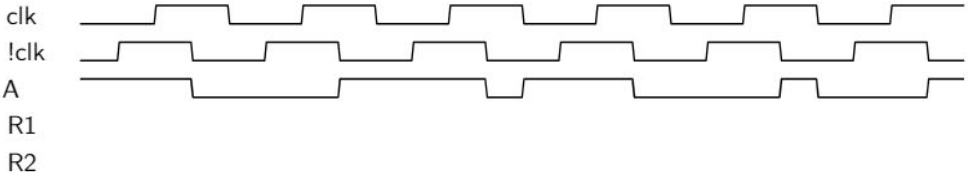
You'll notice that the value of the output in the diagram above doesn't change immediately after the rising edge of the clock. Clock cycle time must be small enough that inputs to registers don't change within the hold time and large enough to account for clk-to-q times, setup times, and combinational logic delays.

4.1 For the following 2 circuits, fill out the timing diagram. The clock period (rising edge to rising edge) is 8ps. For every register, clk-to-q delay is 2ps, setup time is 4ps, and hold time is 2ps. NOT gates have a 2ps propagation delay

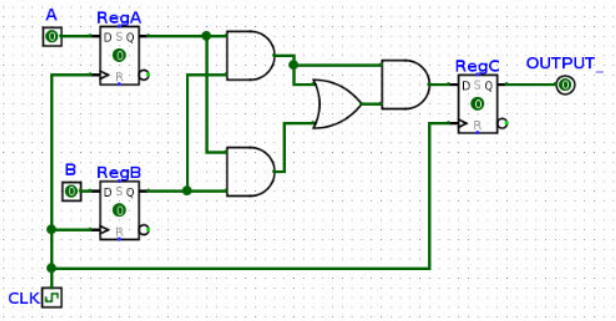








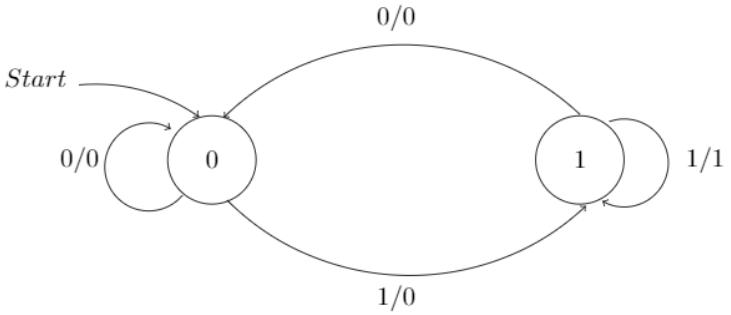
4.2 In the circuit below, RegA and RegB have setup, hold, and clk-to-q times of 4ns, all logic gates have a delay of 5ns, and RegC has a setup time of 6ns. What is the maximum allowable hold time for RegC? What is the minimum acceptable clock cycle time for this circuit, and clock frequency does it correspond to?



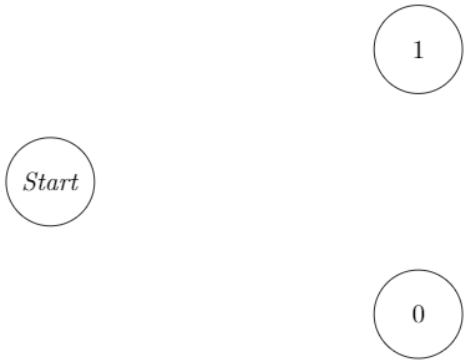
5 Finite State Machines

Automatons are machines that receive input and use various states to produce output. A finite state machine is a type of simple automaton where the next state and output depend only on the current state and input. Each state is represented by a circle, and every proper finite state machine has a starting state, signified either with the label "Start" or a single arrow leading into it. Each transition between states is labeled [input]/[output].

5.1 What pattern in a bitstring does the FSM below detect? What would it output for the input bitstring "011001001110"?



5.2 Fill in the following FSM for outputting a 1 whenever we have two repeating bits as the most recent bits, and a 0 otherwise. You may not need all states.



5.3 Write an FSM that will output a 1 if it recognizes the regex pattern {10+1}.

CS 61C Summer 2020 — RISC-V Single Cycle Datapath

Discussion 7: July 13, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 The single cycle datapath makes use of all hardware units for each instruction.

1.2 It is possible to execute the stages of the single cycle datapath in parallel to speed up execution of a single instruction.

1.3 Combinational logic is only used in the instruction decode stage.

2 Single-Cycle CPU

2.1 For this worksheet, we will be working with the single-cycle CPU datapath on the last page.

(a) On the datapath, fill in each **round** box with the name of the datapath component, and each **square** box with the name of the control signal.

(b) Explain what happens in each datapath stage.

IF Instruction Fetch

ID Instruction Decode

EX Execute

MEM Memory

WB Writeback

2.2 Fill out the following table with the control signals for each instruction based on the datapath on the previous page. Wherever possible, use * to indicate that what this signal is does not matter (as in, letting the value be whatever it wants won't affect the execution of the instruction). If the value of the signal does matter for correct execution, but can vary, list all of the values (for example, for a signal that matters with possible values of 0 and 1, write 0/1).

	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add											
ori											
lw											
sw											
beq											
jal											
bltu											

2.3 Clocking Methodology

- A **state element** is an element connected to the clock (denoted by a triangle at the bottom). The **input signal** to each state element must stabilize before each **rising edge**.
- The **critical path** is the longest delay path between state elements in the circuit. The circuit cannot be clocked faster than this, since anything faster would mean that the correct value is not guaranteed to reach the state element in the alloted time. If we place registers in the critical path, we can shorten the period by **reducing the amount of logic between registers**.

For this exercise, assume the delay for each stage in the datapath is as follows:

IF: 200 ps ID: 100 ps EX: 200 ps MEM: 200 ps WB: 100 ps

(a) Mark the stages of the datapath that the following instructions use and calculate the total time needed to execute the instruction.

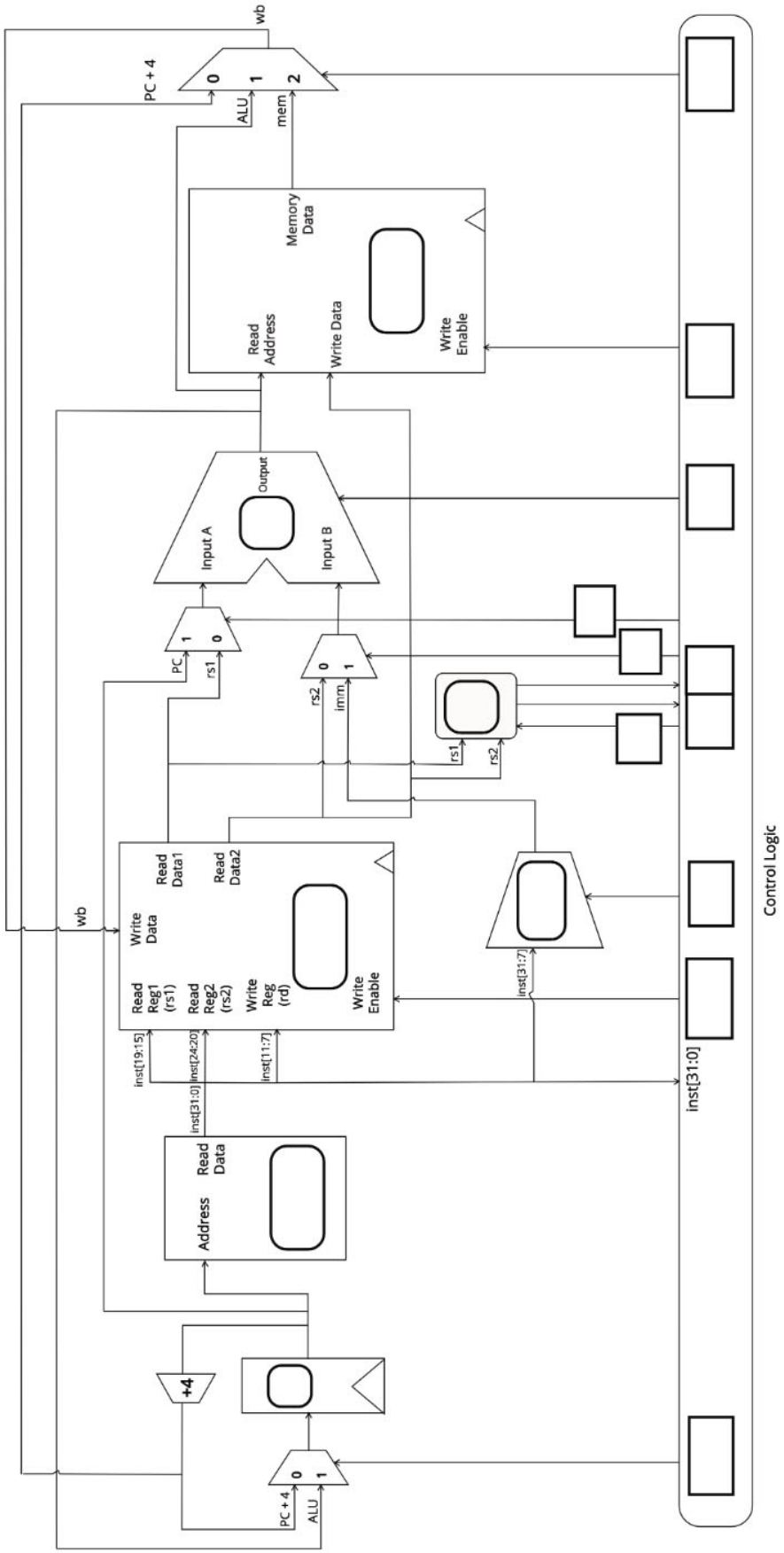
	IF	ID	EX	MEM	WB	Total Time
add						
ori						
lw						
sw						
beq						
jal						
bltu						

(b) Which instruction(s) exercise the critical path?

(c) What is the fastest you could clock this single cycle datapath?

(d) Why is the single cycle datapath inefficient?

(e) How can you improve its performance? What is the purpose of pipelining?


wb
PC + 4
0
1
2
ALU
mem
Memory Data
Read Address
Write Data
Write Enable
Output
Input A
Input B
PC
rs1
rs2
imm
rs1
rs2
Read Data1
Read Data2
Write Data
Read Reg1 (rs1)
Read Reg2 (rs2)
Write Reg (rd)
Write Enable
inst[19:15]
inst[24:20]
inst[11:7]
inst[31:7]
inst[31:0]
inst[31:0]
Read Data
Address
+4
PC + 4
ALU
Control Logic

CS 61C Summer 2020 — RISC-V Pipelining and Hazards

Discussion 8: July 15, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Pipelining the CPU datapath results in instructions being executed with higher latency and throughput.

1.2 Without forwarding, data hazards will usually result in 3 stalls.

1.3 All data hazards can be resolved with forwarding.

1.4 What are some techniques that can be used to resolve control hazards?

2 Pipelining Registers

In order to pipeline, we add registers between the five datapath stages. Label each of the five stages (IF, ID, EX, MEM, and WB) on the diagram attached at the end of the worksheet.

2.1 What is the purpose of the new registers?

2.2 Why do we add +4 to the PC again in the memory stage?

2.3 Why do we need to save the instruction in a register multiple times?

3 Performance Analysis

Register clk-to-q 30 ps	**Branch comp.** 75 ps	**Memory write** 200 ps
Register setup 20 ps	**ALU** 200 ps	**RegFile read** 150 ps
Mux 25 ps	**Memory read** 250 ps	**RegFile setup** 20 ps

3.1 With the delays provided above for each of the datapath components, what would be the fastest possible clock time for a single cycle datapath?

3.2 What is the fastest possible clock time for a pipelined datapath?

3.3 What is the speedup from the single cycle datapath to the pipelined datapath? Why is the speedup less than 5?

4 Hazards

One of the costs of pipelining is that it introduces three types of pipeline hazards: structural hazards, data hazards, and control hazards.

Structural Hazards

Structural hazards occur when more than one instruction needs to use the same datapath resource at the same time. There are two main causes of structural hazards:

Register File The register file is accessed both during ID, when it is read, and during WB, when it is written to. We can solve this by having separate read and write ports. To account for reads and writes to the same register, processors usually write to the register during the first half of the clock cycle, and read from it during in the second half. This is also known as double pumping.

Memory Memory is accessed for both instructions and data. Having a separate instruction memory (abbreviated IMEM) and data memory (abbreviated DMEM) solves this hazard.

Something to remember about structural hazards is that they can always be resolved by adding more hardware.

Data Hazards

Data hazards are caused by data dependencies between instructions. In CS 61C, where we will always assume that instructions are always going through the processor in order, we see data hazards when an instruction **reads** a register before a previous instruction has finished **writing** to that register.

Forwarding

Most data hazards can be resolved by forwarding, which is when the result of the EX or MEM stage is sent to the EX stage for a following instruction to use.

4.1 Look for data hazards in the code below, and figure out how forwarding could be used to solve them.

Instruction	C1	C2	C3	C4	C5	C6	C7
1. `addi t0, a0, -1`	IF	ID	EX	MEM	WB		
2. `and s2, t0, a0`		IF	ID	EX	MEM	WB	
3. `sltiu a0, t0, 5`			IF	ID	EX	MEM	WB

4.2 Imagine you are a hardware designer working on a CPU's forwarding control logic. How many instructions after the addi instruction could be affected by data hazards created by this addi instruction?

Stalls

4.3 Look for data hazards in the code below. One of them cannot be solved with forwarding—why? What can we do to solve this hazard?

Instruction	C1	C2	C3	C4	C5	C6	C7	C8
1. `addi s0, s0, 1`	IF	ID	EX	MEM	WB			
2. `addi t0, t0, 4`		IF	ID	EX	MEM	WB		
3. `lw t1, 0(t0)`			IF	ID	EX	MEM	WB	
4. `add t2, t1, x0`				IF	ID	EX	MEM	WB

4.4 Say you are the compiler and can re-order instructions to minimize data hazards while guaranteeing the same output. How can you fix the code above?

Detecting Data Hazards

Say we have the $rs1$, $rs2$, $RegWEn$, and rd signals for two instructions (instruction n and instruction $n+1$) and we wish to determine if a data hazard exists across the instructions. We can simply check to see if the rd for instruction n matches either $rs1$ or $rs2$ of instruction $n+1$, indicating that such a hazard exists (think, why does this make sense?).

We could then use our hazard detection to determine which forwarding paths/number of stalls (if any) are necessary to take to ensure proper instruction execution. In pseudo-code, this could look something like the following:

```
if (rs1(n + 1) == rd(n) || rs2(n + 1) == rd(n) && RegWen(n) == 1) {
    forward ALU output of instruction n
}
```

Control Hazards

Control hazards are caused by **jump and branch instructions**, because for all jumps and some branches, the next PC is not PC + 4, but the result of the computation completed in the EX stage. We could stall the pipeline for control hazards, but this decreases performance.

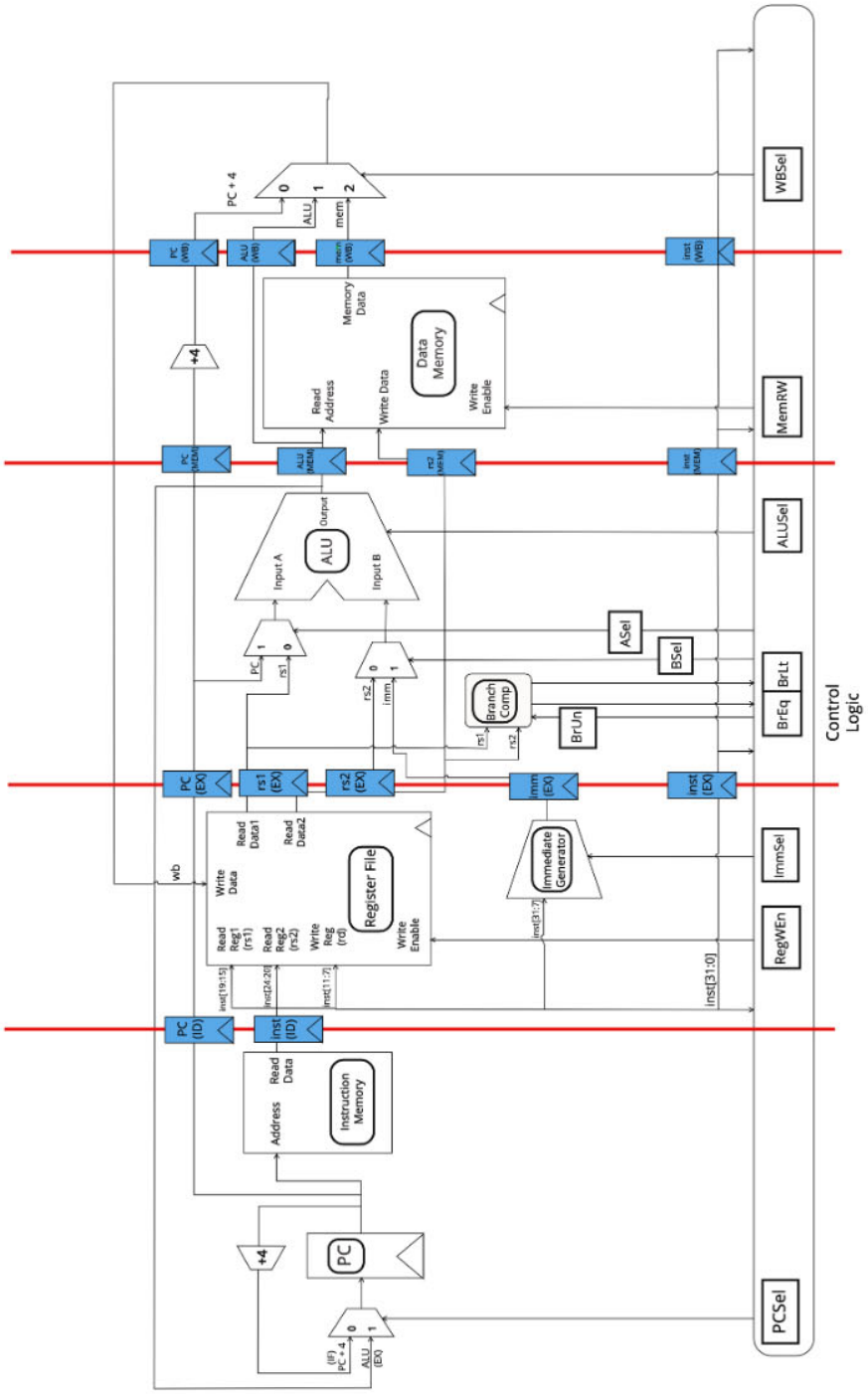
4.5 Besides stalling, what can we do to resolve control hazards?

Extra for Experience

4.6 Given the RISC-V code above and a pipelined CPU with no forwarding, how many hazards would there be? What types are each hazard? Consider all possible hazards from all pairs of instructions.

How many stalls would there need to be in order to fix the data hazard(s)? What about the control hazard(s)?

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9
1. `sub t1, s0, s1`	IF	ID	EX	MEM	WB				
2. `or s0, t0, t1`		IF	ID	EX	MEM	WB			
3. `sw s1, 100(s0)`			IF	ID	EX	MEM	WB		
4. `bgeu s0, s2, l`				IF	ID	EX	MEM	WB	
5. `add t2, x0, x0`					IF	ID	EX	MEM	WB


WBSel
MemRW
ALUSel
ASel
BSel
BrLt
BrEq
BrUn
ImmSel
RegWEn
PCSel
Control Logic
PC + 4
ALU
mem
Memory Data
Data Memory
Read Address
Write Data
Write Enable
Output
Input A
Input B
Branch Comp
rs1
rs2
imm
Read Data1
Read Data2
Write Data
Register File
Read Reg1 (rs1)
Read Reg2 (rs2)
Write Reg (rd)
Write Enable
Immediate Generator
inst[31:7]
inst[31:0]
inst[19:15]
inst[24:20]
inst[11:7]
Read Data
Address
Instruction Memory
PC
(IF) PC + 4
(EX) ALU
PC (WB)
ALU (WB)
mem (WB)
inst (WB)
PC (MEM)
ALU (MEM)
rs2 (MEM)
inst (MEM)
PC (EX)
rs1 (EX)
rs2 (EX)
imm (EX)
inst (EX)
wb
PC (ID)
inst (ID)

CS 61C Summer 2020

Caches

Discussion 9: July 20, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 For the same cache size and block size, a 4-way set associative cache will have fewer index bits than a direct-mapped cache.

1.2 Any cache miss that occurs when the cache is full is a capacity miss.

1.3 Increasing cache size by adding more blocks always improves (increases) hit rate.

2 Understanding T/I/O

When working with caches, we have to be able to break down the memory addresses we work with to understand where they fit into our caches. There are three fields:

Tag - Used to distinguish different blocks that use the same index. Number of bits: (# of bits in memory address) - Index Bits - Offset Bits
Index - The set that this piece of memory will be placed in. Number of bits: $\log_2$(# of indices)
Offset - The location of the byte in the block. Number of bits: $\log_2$(size of block)

Given these definitions, the following is true:

$$\log_2(\text{memory size}) = \text{address bit-width} = \#\text{ tag bits} + \#\text{ index bits} + \#\text{ offset bits}$$

Another useful equality to remember is:

$$\text{cache size} = \text{block size} * \text{num blocks}$$

2.1 Assume we have a direct-mapped byte-addressed cache with capacity 32B and block size of 8B. Of the 32 bits in each address, which bits do we use to find the index of the cache to use?

2.2 Which bits are our tag bits? What about our offset?

2.3 Classify each of the following byte memory accesses as a cache hit (H), cache miss (M), or cache miss with replacement (R). Tip: Drawing out the cache can help you see the replacements more clearly.

Address	T/I/O	Hit, Miss, Replace
`0x00000004`		
`0x00000005`		
`0x00000068`		
`0x000000C8`		
`0x00000068`		
`0x000000DD`		
`0x00000045`		
`0x00000004`		
`0x000000C8`		

3 Cache Associativity

In the previous problem, we had a Direct-Mapped cache, in which blocks map to specifically one slot in our cache. This is good for quick replacement and finding out block, but not good for efficiency of space!

This is where we bring associativity into the matter. We define associativity as the number of slots a block can potentially map to in our cache. Thus, a Fully-Associative cache has the most associativity, meaning every block can go anywhere in the cache.

For an N-way associative cache, the following is true:

$$\text{N} * \# \text{ sets} = \# \text{ blocks}$$

3.1 Here's some practice involving a 2-way set associative cache. This time we have an 8-bit address space, 8 B blocks, and a cache size of 32 B. Classify each of the following accesses as a cache hit (H), cache miss (M) or cache miss with replacement (R). For any misses, list out which type of miss it is. Assume that we have an LRU replacement policy (in general, this is not the case).

Address	T/I/O	Hit, Miss, Replace
`0b0000 0100`		
`0b0000 0101`		
`0b0110 1000`		
`0b1100 1000`		
`0b0110 1000`		
`0b1101 1101`		
`0b0100 0101`		
`0b0000 0100`		
`0b1100 1000`		

3.2 What is the hit rate of our above accesses?

4 The 3 C's of Cache Misses

4.1 Go back to questions 2 and 3 and classify each M and R as one of the 3 types of misses described below:

1. Compulsory: First time you ask the cache for a certain block. A miss that must occur when you first bring in a block. Reduce compulsory misses by having longer cache lines (bigger blocks), which bring in the surrounding addresses along with our requested data. Can also pre-fetch blocks beforehand using a hardware prefetcher (a special circuit that tries to guess the next few blocks that you will want).
2. Conflict: Occurs if, hypothetically, you went through the ENTIRE string of accesses with a fully associative cache (with an LRU replacement policy) and wouldn't have missed for that specific access. Increasing the associativity or improving the replacement policy would remove the miss.
3. Capacity: Capacity misses are independent of the associativity of your cache. If you hypothetically ran the ENTIRE string of memory accesses with a fully associative cache (with an LRU replacement policy) of the same size as your cache, and it was a miss for that specific access, then this miss is a capacity miss. The only way to remove the miss is to increase the cache capacity.

Note: The test you can use to see if a miss is a conflict miss is the same as the test you can use to see if a miss is a capacity miss.

Note: There are many different ways of fixing misses. The name of the miss doesn't necessarily tell us the best way to reduce the number of misses.

5 Code Analysis

Given the follow chunk of code, analyze the hit rate given that we have a byte-addressed computer with a total memory of **1 MiB**. It also features a **16 KiB** Direct-Mapped cache with **1 KiB** blocks. Assume that your cache begins cold.

```
#define NUM_INTS 8192   // 2^13
int A[NUM_INTS];        // A lives at 0x10000
int i, total = 0;
for (i = 0; i < NUM_INTS; i += 128) {
    A[i] = i;           // Line 1
}
for (i = 0; i < NUM_INTS; i += 128) {
    total += A[i];      // Line 2
}
```

5.1 How many bits make up a memory address on this computer?

5.2 What is the T:I:O breakdown?

5.3 Calculate the cache hit rate for the line marked Line 1:

5.4 Calculate the cache hit rate for the line marked Line 2:

CS 61C OS & I/O

Summer 2020 Discussion 10: July 22, 2019

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Polling and interrupts are only relevant concepts for low level programming.

1.2 Memory-mapped IO only works with polling.

1.3 Responsibilities of the OS include loading programs, handling services, multiplexing resources, and combining programs together for efficiency.

1.4 The purpose of supervisor mode is to isolate certain instructions and routines from user programs.

1.5 User programs call into OS routines using system calls.

2 Polling & Interrupts

2.1 Fill out this table that compares polling and interrupts.

Operation	Definition	Pro/Good for	Con
Polling			
Interrupts			

3 Memory Mapped I/O

3.1 For this question, the following addresses correspond to registers in some I/O devices and not regular user memory.

- `0xFFFF0000`—Receiver Control: LSB is the ready bit (in the context of polling), there may be other bits set that we don't need right now.
- `0xFFFF0004`—Receiver Data: Received data stored at lowest byte.
- `0xFFFF0008`—Transmitter Control: LSB is the ready bit (in the context of polling), there may be other bit set that we don't need right now.
- `0xFFFF000C`—Transmitter Data: Transmitted data stored at lowest byte.

Recall that receiver will only have data for us when the corresponding ready bit is 1, and that we can only write data to the transmitter when its ready bit is 1. Write RISC-V code that reads byte from the receiver (busy-waiting if necessary) and writes that byte to the transmitter (busy-waiting if necessary).

4 Forking

One of the many responsibilities of the OS is to load new programs, and in order to do this it creates a new process and loads in the program to execute. In Linux, the system call to create a new process is fork(). fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process. In the parent process, fork() returns the process ID of the child or -1 if the fork has failed. In the child process, it returns 0.

Use this information to complete the code block below, which creates a child process to change the value of y while the parent process changes the value of x.

```
int x = 10;
int y = 0;
int pid = __________;
if(__________){
    y++
}
else{
    x--;
}
```

CS 61C Summer 2020

Virtual Memory

Discussion 11: July 27, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 If a page table entry can not be found in the TLB, then a page fault has occurred.

1.2 The virtual and physical page number must be the same size.

2 Addressing

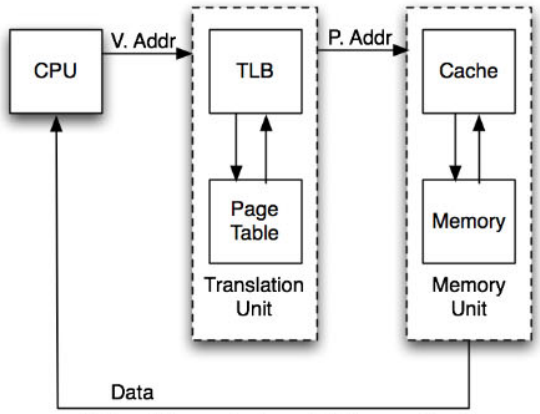
Virtual Address (VA) What your program uses

Virtual Page Number (VPN)	Page Offset

Physical Address (PA) What actually determines where in memory to go

Physical Page Number (PPN)	Page Offset

For example, with 4 KiB pages and byte addresses, there are 12 page offset bits since 4 KiB $= 2^{12}$ B $= 4096$ B.



Pages

A chunk of memory or disk with a set size. Addresses in the same virtual page map to addresses in the same physical page. The page table determines the mapping.

Valid	Dirty	Permission Bits	PPN
— Page entry (VPN: 0) —			
— Page entry (VPN: 1) —			

Each stored row of the page table is called a **page table entry**. There are $2^{\text{VPN bits}}$ such entries in a page table. Say you have a VPN of 5 and you want to use the page table to find what physical page it maps to; you'll check the 5th (0-indexed) page table entry. If the valid bit is 1, then that means that the entry is valid (in other words, the physical page corresponding to that virtual page is in main memory as opposed to being only on disk) and therefore you can get the PPN from the entry and access that physical page in main memory. The page table is stored in memory: the OS sets a register (the Page Table Base Register) telling the hardware the address of the first entry of the page table. If you write to a page in memory, the processor updates the "dirty" bit in the page table entry corresponding to that page, which lets the OS know that updating that page on disk is necessary (remember: main memory contains a subset of what's on disk). This is a similar concept as having a dirty bit for each cache block in a write-back cache, which we covered in lecture and in Lab 9. Each process gets its own illusion of full memory to work with, and therefore its own page table.

Protection Fault The page table entry for a virtual page has permission bits that prohibit the requested operation. This is how a segmentation fault occurs.

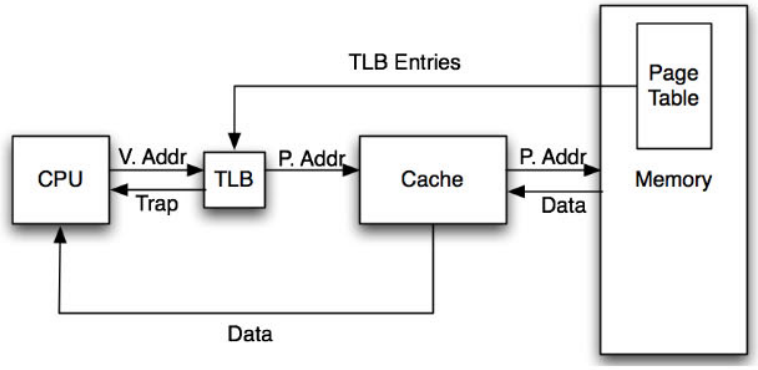
Page Fault The page table entry for a virtual page has its valid bit set to false.

This means that the entry is not in memory, so we pull it from disk, add the page to memory (evicting another page if necessary), and add the mapping to the page table *and the TLB*.

Translation Lookaside Buffer

A cache for the page table. Each block is a single page table entry. If an entry is not in the TLB, it's a TLB miss. Assuming fully associative:

TLB Valid	Tag (VPN)	Page Table Entry		
		Page Dirty	Permission Bits	PPN
— *TLB entry* —				
— *TLB entry* —				



To access some memory location, we get the virtual page number (VPN) from the virtual address (VA) and first try to translate the VPN to a physical page number (PPN) using the translation lookaside buffer (TLB). If the TLB doesn't contain the desired VPN, we check if the page table contains it (remember: the TLB is a subset of the page table!). If the page table doesn't contain an entry for the VPN, then this is a page fault; memory doesn't contain the corresponding physical page! This means we need to fetch the physical page from disk and put it into memory, update the page table entry, and load the entry into the TLB, Then, we use the physical page and the offset of the physical address in the page to access memory as the program intended.

2.1 What are three specific benefits of using virtual memory?

2.2 What should happen to the TLB when a new value is loaded into the page table address register?

3 VM Access Patterns

3.1 A processor has 16-bit addresses, 256 byte pages, and an 8-entry fully associative TLB with LRU replacement (the LRU field is 3 bits and encodes the order in which pages were accessed, 0 being the most recent). At some time instant, the TLB for the current process is the initial state given in the table below. Assume that all current page table entries are in the initial TLB. Assume also that all pages can be read from and written to. Fill in the final state of the TLB according to the access pattern below.

Free Physical Pages `0x17`, `0x18`, `0x19`

Access Pattern

1. `0x11f0` (**Read**)
2. `0x1301` (**Write**)
3. `0x20ae` (**Write**)
4. `0x2332` (**Write**)
5. `0x20ff` (**Read**)
6. `0x3415` (**Write**)

Initial TLB

VPN	PPN	Valid	Dirty	LRU
`0x01`	`0x11`	1	1	0
`0x00`	`0x00`	0	0	7
`0x10`	`0x13`	1	1	1
`0x20`	`0x12`	1	0	5
`0x00`	`0x00`	0	0	7
`0x11`	`0x14`	1	0	4
`0xac`	`0x15`	1	1	2
`0xff`	`0xff`	1	0	3

Final TLB

VPN	PPN	Valid	Dirty	LRU

CS 61C Summer 2020

AMAT, DLP

Discussion 12: August 3, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 If a page table entry can not be found in the TLB, then a page fault has occurred.

1.2 The local miss rate of one level of a cache is always greater than or equal to the global miss rate of that cache.

1.3 SIMD is a form of instruction-level parallelism.

2 AMAT

Recall that AMAT stands for Average Memory Access Time. The main formula for it is:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

In a multi-level cache, there are two types of miss rates that we consider for each level.

Global: Calculated as the number of accesses that missed at that level divided by the total number of accesses *to the cache system*.

Local: Calculated as the number of accesses that missed at that level divided by the total number of accesses *to that cache level*.

2.1 • An L2$, out of 100 total accesses to the cache system, missed 20 times. What is the global miss rate of L2$?

2.2 If L1$ had a miss rate of 50%, what is the local miss rate of L2$?

Suppose your system consists of:

1. An L1$ that has a hit time of 2 cycles and has a local miss rate of 20%
2. An L2$ that has a hit time of 15 cycles and has a global miss rate of 5%
3. Main memory where accesses take 100 cycles

2.3 What is the local miss rate of L2$?

2.4 What is the AMAT of the system?

2.5 Suppose we want to reduce the AMAT of the system to 8 cycles or lower by adding in a L3$. If the L3$ has a local miss rate of 30%, what is the largest hit time that the L3$ can have?

3 Flynn's Taxonomy

3.1 Explain SISD and give an example if available.

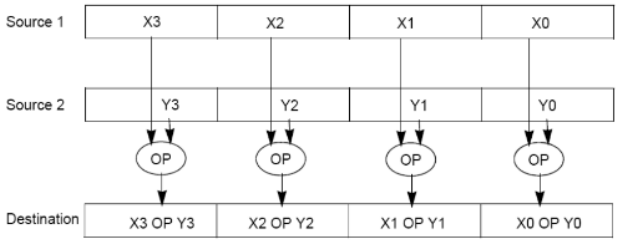
3.2 Explain SIMD and give an example if available.

3.3 Explain MISD and give an example if available.

3.4 Explain MIMD and give an example if available.

4 Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.



Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to use these wider registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type __m128i is used when these registers hold 4 ints, 8 shorts or 16 chars; __m128d is used for 2 double precision floats, and __m128 is used for 4 single precision floats. Where you see "epiXX", epi stands for extended packed integer, and XX is the number of bits in the integer. "epi32" for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

- `__m128i _mm_set1_epi32(int i)`:
  Set the four signed 32-bit integers within the vector to `i`.
- `__m128i _mm_loadu_si128( __m128i *p)`:
  Load the 4 successive ints pointed to by `p` into a 128-bit vector.
- `__m128i _mm_mullo_epi32(__m128i a, __m128i b)`:
  Return vector $(a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)$.
- `__m128i _mm_add_epi32(__m128i a, __m128i b)`:
  Return vector $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- `void _mm_storeu_si128( __m128i *p, __m128i a)`:
  Store 128-bit vector `a` at pointer `p`.
- `__m128i _mm_and_si128(__m128i a, __m128i b)`:
  Perform a bitwise AND of 128 bits in a and b, and return the result.
- `__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)`:
  The ith element of the return vector will be set to 0xFFFFFFFF if the ith elements of a and b are equal, otherwise it'll be set to 0.

Notice: On this worksheet, we are using the *unaligned* versions of the commands that interface with memory (i.e. `storeu/loadu` vs. `store/load`). This is because the `store/load` commands require that the address we are loading at is aligned at some byte boundary (and not necessarily just word-aligned), whereas the unaligned versions have no such requirements. For instance, `_mm_store_si128` needs the address to be aligned on a 16-byte boundary (i.e. is a multiple of 16). There is extra work that needs to be done to achieve these alignment requirements, so for this class, we just use the unaligned variants.

4 *AMAT, DLP*

4.1 You have an array of 32-bit integers and a 128-bit vector as follows:

```
int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
__m128i vector = _mm_loadu_si128((__m128i *) arr);
```

For each of the following tasks, fill in the correct arguments for each SIMD instruction, and where necessary, fill in the appropriate SIMD function. Assume they happen independently, i.e. the results of Part (a) do not at all affect Part (b).

(a) Multiply `vector` by itself, and set `vector` to the result.

```
vector = ____________________(____________________, ____________________);
```

(b) Add 1 to each of the first 4 elements of the `arr`, resulting in `arr = {2, 3, 4, 5, 5, 6, 7, 8}`

```
__m128i vector_ones = _mm_set1_epi32(____________________);
__m128i result = _mm_add_epi32(____________________, ____________________);
_mm_storeu_si128(____________________, ____________________);
```

(c) Add the second half of the array to the first half of the array, resulting in `arr = {1 + 5, 2 + 6, 3 + 7, 4 + 8, 5, 6, 7, 8} = {6, 8, 10, 12, 5, 6, 7, 8}`

```
__m128i result = _mm_add_epi32(_mm_loadu_si128(____________________), ____________________);
_mm_storeu_si128(____________________, ____________________);
```

(d) Set every element of the array that is not equal to 5 to 0, resulting in `arr = {0, 0, 0, 0, 5, 0, 0, 0}`. Remember that the first half of the array has already been loaded into vector.

```
__m128i fives = ____________________(____________________);
__m128i mask = ____________________(____________________, ____________________);
__m128i result = ____________________(____________________, ____________________);
_mm_storeu_si128(____________________, ____________________);
vector = _mm_loadu_si128(____________________);
mask = ____________________(____________________, ____________________);
result = ____________________(____________________, ____________________);
_mm_storeu_si128(____________________, ____________________);
```

4.2 SIMD-ize the following function, which returns the product of all of the elements in an array. Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn't a multiple of 4? Can we always SIMD-ize an entire array? What can we do to handle this tail case?

```
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}


static int product_vectorized(int n, int *a) {
    int result[4];
    __m128i prod_v = ________________________________________;
    for (int i = 0; i < _____; i += ___) { // Vectorized loop
        prod_v = ____________________________________________________________________________;
    }
    __mm_storeu_si128(_____________________________, _____________________________);
    for (int i = _____________; i < ____________; i++) { // Handle tail case
        result[0] *= _________________________;
    }
    return ____________________________________________________;
}
```

CS 61C Summer 2020 — Coherency and Atomics, TLP

Discussion 13: August 5, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Each hardware thread in the CPU uses a shared cache.

1.2 Atomicity can only be guaranteed within a single RISC-V instruction.

1.3 The amount of speedup is directly proportional to the increase in number of threads.

2 Coherency and Atomics

The benefits of multi-threading programming come only after you understand concurrency. Here are two of the most common concurrency issues:

1. **Cache-incoherence**: each hardware thread has its own cache, hence data modified in one thread may not be immediately reflected in the other. This can often be solved by bypassing the cache and writing directly to memory, i.e. using volatile keywords in many languages.
2. **Read-modify-write**: Read-modify-write is a very common pattern in programming. In the context of multi-threading programming, the **interleaving** of R, M, W stages often produces a lot of issues.

In order to solve the problems created by Read-modify-write, we have to rely on the idea of **uninterrupted execution**, also known as atomic execution.

In RISC-V, we have two categories of atomic instructions:

1. **Load-reserve, store-conditional**: allows us to have uninterrupted execution across multiple instructions
2. **Amo.swap**: allows for uninterrupted memory operations within a single instruction

Both of these can be used to achieve atomic primitives. Here are examples for each:

Test-and-set

```
Start:  addi         t0 x0 1 # Locked = 1
        amoswap.w.aq t1 t0 (a0)
        bne          t1 x0 Start
# If the lock is not free, retry

        ... # Critical section

        amoswap.w.rl x0 x0 (a0) # Release lock
```

Compare-and-swap

```
# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
# a0 holds return value, 0 if successful, !0
    otherwise
cas:
    lr.w t0, (a0) # Load original value.
    bne t0, a1, fail # Doesnt match, so fail.
    sc.w a0, a2, (a0) # Try to update.
    jr ra # Return.
fail:
    li a0, 1 # Set return to failure.
    jr ra # Return.
```

Instruction definitions:

1. **L**oad-reserve: Loads the four bytes at `M[R[rs1]]`, writes them to `R[rd]`, sign-extending the result and registers a reservation on that word in memory.
2. **S**tore-conditional **rd, rs2, (rs1)**: Stores the four bytes in register `R[rs2]` to `M[R[rs1]]`, provided there exists a load reservation on that memory address. Writes 0 to `R[rd]` if the store succeeded, or a nonzero error code otherwise.
3. **Amoswap rd, rs2, (rs1)**: Atomically, puts the sign-extended word located at `M[R[rs1]]` into `R[rd]` and puts `R[rs2]` into `M[R[rs1]]`.

Explanations for both methodologies:

1. **Test-and-set**: We have a lock stored at the address specified by `a0`. We utilize `amoswap` to put in 1 and get the old value. If the old value was a 1, we would not have changed the value of the lock and we will realize that someone currently has the lock. If the old value was a 0, we will have just "locked" the lock and can continue with the critical section. When we are done, we put a 0 back into the lock to "unlock" it.
2. **Compare-and-swap**: CAS tries to first reserve the memory and gets the value stored and compares it to the expected value. If the expected value and the value that was stored do not match, the entire process fails and we must restart to update based on the new information. Otherwise, we register a reservation on the memory and try to store the new value. If the exit code is nonzero, something went wrong with the store and we must retry the entire LR/SC process. Otherwise with a zero exit code, we continue into the critical section, then release the lock.

2.1 Why do we need special instructions for these operations? Why can't we use normal load and store for `lr` and `sc`? Why can't we expand `amoswap` to a normal load and store?

2.2 Now that we have atomic operations, let's try to experiment with them. Let us try to implement an algorithm that enforces ordered thread execution. This means that if we have four threads, thread 0 goes first, thread 1 goes next, etc. For this problem assume that `a1` holds the location of a piece of memory we have access to for the entire duration of our algorithm. Also, we can assume there exists a label `get_thread_num` that returns the thread's number in `a0`. Try to fill in the blanks below. Please use LR/SC for this problem:

```
        addi t0, x0, 0

        _____________________________ # Setup for the first (0-th) thread
        ...
        # Assume we now spawn 4 threads in this code
        ...

Check: jal ______________________________ # Get the current thread number
                                         # Get the ID of the next thread that should operate
        _____________________________ # (make sure this can't get interfered with)

        ____________________________
Done:  addi t0, t0, 1

        _____________________________ # Set which thread is next to run

        bne ________________________
        ...
        # Assume we now join the 4 threads in this code
        ...
        jr ra
```

3 Thread-Level Parallelism

As powerful as data level parallelization is, it can be quite inflexible, as not all applications have data that can be vectorized. Multithreading, or running a single piece of software on multiple hardware threads, is much more powerful and versatile.

OpenMP provides an easy interface for using multithreading within C programs. Some examples of OpenMP directives:

- The `parallel` directive indicates that each thread should run a copy of the code within the block. If a for loop is put within the block, **every** thread will run every iteration of the for loop.

```
#pragma omp parallel
{
    ...
}
NOTE: The opening curly brace needs to be on a newline or else there
    will be a compile-time error!
```

- The `parallel for` directive will split up iterations of a for loop over various threads. Every thread will run **different** iterations of the for loop. The following two code snippets are equivalent.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    ...
}
```

```
#pragma omp parallel
{
#pragma omp for
    for (int i =0; i < n; i++) { ... }
}
```

There are two functions you can call that may be useful to you:

- `int omp_get_thread_num()` will return the number of the thread executing the code
- `int omp_get_num_threads()` will return the number of total hardware threads executing the code

3.1 For each question below, state and justify whether the program is **sometimes incorrect**, **always incorrect**, **slower than serial**, **faster than serial**, or **none of the above**. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume `arr` is an `int[]` of length `n`.

(a)

```
// Set element i of arr to i
#pragma omp parallel
{
    for (int i = 0; i < n; i++)
        arr[i] = i;
}
```

(b)

```
// Set arr to be an array of Fibonacci numbers.
```

```
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
    arr[i] = arr[i-1] + arr[i - 2];
```

(c)

```
// Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    arr[i] = 0;
```

3.2 What potential issue can arise from this code?

```
// Decrements element i of arr. n is a multiple of omp_get_num_threads()
#pragma omp parallel
{
    int threadCount = omp_get_num_threads();
    int myThread = omp_get_thread_num();
    for (int i = 0; i < n; i++) {
        if (i % threadCount == myThread) arr[i] -= 1;
    }
}
```

3.3

```
// Assume n holds the length of arr
double fast_product(double *arr, int n) {
    double product = 1;
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        product *= arr[i];
    }
    return product;
}
```

(a) What is wrong with this code?

(b) Fix the code using `#pragma omp critical`

(c) Fix the code using `#pragma omp reduction(operation: var)`.

4 Amdahl's Law

In the programs we write, there are sections of code that are naturally able to be sped up. However, there are likely sections that just can't be optimized any further to maintain correctness. In the end, the overall program speedup is the number that matters, and we can determine this using Amdahl's Law:

$$\text{True Speedup} = \frac{1}{S + \frac{1-S}{P}}$$

where S is the non-sped-up part and P is the speedup factor (determined by the number of cores, threads, etc.).

4.1 You are going to run a convolutional network to classify a set of 100,000 images using a computer with 32 threads. You notice that 99% of the execution of your project code can be parallelized on these threads. What is the speedup?

4.2 You run a profiling program on a different program to find out what percent of this program each function takes. You get the following results:

Function	% Time
`f`	30%
`g`	10%
`h`	60%

(a) We don't know if these functions can actually be parallelized. However, assuming all of them can be, which one would benefit the most from parallelism?

(b) Let's assume that we verified that your chosen function can actually be parallelized. What speedup would you get if you parallelized just this function with 8 threads?

CS 61C Summer 2020 — MR, Spark, WSC, RAID, ECC

Discussion 14: August 10, 2020

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 MapReduce is more general than Spark since it is lower level.

1.2 The higher the PUE the more efficient the datacenter is.

1.3 Hamming codes can detect any type of data corruption.

1.4 All RAID levels improve reliability.

2 Hamming ECC

Recall the basic structure of a Hamming code. We start out with some bitstring, and then add parity bits at the indices that are powers of two (1, 2, 8, etc.). We don't assign values to these parity bits yet. **Note that the indexing convention used for Hamming ECC is different from what you are familiar with.** In particular, the 1 index represents the MSB, and we index from left-to-right. The ith parity bit $P\{i\}$ covers the bits in the new bitstring where the *index* of the bit under the aforementioned convention, j, has a 1 at the same position as i when represented as binary. For instance, 4 is `0b100` in binary. The integers j that have a 1 in the same position when represented in binary are 4, 5, 6, 7, 12, 13, etc. Therefore, $P4$ covers the bits at indices 4, 5, 6, 7, 12, 13, etc. A visual representation of this is:

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																✗	✗	✗	✗	✗	

Source: `https://en.wikipedia.org/wiki/Hamming_code`

2.1 How many bits do we need to add to 0011_2 to allow single error correction?

2.2 Which locations in 0011_2 would parity bits be included?

2.3 Which bits does each parity bit cover in 0011_2?

2.4 Write the completed coded representation for 0011_2 to enable single error correction. Assume that we set the parity bits so that the bits they cover have even parity.

2.5 How can we enable an additional double error detection on top of this?

2.6 Find the original bits given the following SEC Hamming Code: 0110111_2. Again, assume that the parity bits are set so that the bits they cover have even parity.

2.7 Find the original bits given the following SEC Hamming Code: 1001000_2

3 RAID

3.1 Fill out the following table:

	Configuration	**Pro/Good for**	**Con/Bad for**
RAID 0			
RAID 1			
RAID 2			
RAID 3			
RAID 4			

RAID 5			

4 MapReduce

For each problem below, write pseudocode to complete the implementations. Tips:

- The input to each MapReduce job is given by the signature of `map()`.
- `emit(key k, value v)` outputs the key-value pair `(k, v)`.
- `for var in list` can be used to iterate through `Iterables` or you can call the `hasNext()` and `next()` functions.
- Usable data types: **`int`**, **`float`**, `String`. You may also use lists and custom data types composed of the aforementioned types.
- `intersection(list1, list2)` returns a list of the common elements of `list1`, `list2`.

4.1 Given a set of coins and each coin's owner in the form of a list of CoinPairs, compute the number of coins of each denomination that a person has.

```
CoinPair:
    String person
    String coinType
```

```
1 map(CoinPair pair):
```

```
1 reduce(_____________, _____________):
```

4.2 Using the output of the first MapReduce, compute each person's amount of money. `valueOfCoin(String coinType)` returns a float corresponding to the dollar value of the coin.

```
1 map(tuple<CoinPair, int> output):
```

```
1 reduce(_____________, _____________):
```

5 Spark

Resilient Distributed Datasets (RDD) are the primary abstraction of a distributed collection of items

Transforms $RDD \rightarrow RDD$

`map`(f) Return a new transformed item formed by calling f on a source element.

`flatMap`(f) Similar to map, but each input item can be mapped to 0 or more output items (so f should return a sequence rather than a single item).

`reduceByKey`(f) When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function f, which must be of type $(V, V) \rightarrow V$.

Actions $RDD \rightarrow Value$

`reduce`(f) Aggregate the elements of the dataset *regardless of keys* using a function f.

Call `sc.parallelize(data)` to parallelize a Python collection, `data`.

5.1 Given a set of coins and each coin's owner, compute the number of coins of each denomination that a person has. Then, using the output of the first result, compute each person's amount of money. Assume `valueOfCoin(coinType)` is defined and returns the dollar value of the coin.

The type of `coinPairs` is a tuple of (person, coinType) pairs.

```
coinData = sc.parallelize(coinPairs)
```

5.2 Given a student's name and course taken, output their name and total GPA.

```
CourseData:
    int courseID
    float studentGrade // a number from 0-4
```

The type of `students` is a list of (studentName, courseData) pairs.

```
studentsData = sc.parallelize(students)
```

6 Warehouse-Scale Computing

Sources speculate Google has over 1 million servers. Assume each of the 1 million servers draw an average of 200W, the PUE is 1.5, and that Google pays an average of 6 cents per kilowatt-hour for datacenter electricity.

6.1 Estimate Google's annual power bill for its datacenters.

6.2 Google reduced the PUE of a 50,000-machine datacenter from 1.5 to 1.25 without decreasing the power supplied to the servers. What's the cost savings per year?