EECS 370 Course Notes

by danlliu

This is a compiled set of notes that discusses topics from EECS 370 (Introduction to Computer Organization) at the University of Michigan.

Who am I?
I'm danlliu, and I was an IA for EECS 370 for three semesters (Winter 2021, Spring 2021, and Fall 2021). While I've left 370, I wanted to create these notes for future students to have at their disposal, similar to the course notes in other course such as EECS 280, 281, and 376. I hope you find these useful, and thank you for taking the time to read through these notes :)

n.b. For many of the examples in these notes, there are many ways to think about and approach the problem. I've tried to include a couple different approaches in most problems, but I'll generally follow the approach that I'm most familiar with. You may see different approaches to these problems from different instructors, but the final answer should be the same.

n.b. 2 This is **not** an official set of notes! Many 370 staff and I have gone through these notes to the best of our ability to check for errors, but there may be some small typos that remain. If you find any errors, please raise an issue on the GitHub repository! In case these notes and official course materials differ, please use the official course materials for reference.

Sources and a few notes:
Figures for NOT, AND, OR, NAND, NOR, and XOR gates were sourced from Wikipedia. (Almost) all other figures are hand-drawn. All examples in the notes were designed independently of official course material, and are designed to provide a conceptual understanding of the material covered in these notes. They are **not** guaranteed to be similar to prior or current exams. The specification for LC2K is sourced from lecture and discussion notes, as well as the Project 1 specification. The designs for the single-cycle, multi-cycle and pipelined processors are reproduced from lecture and discussion notes. Memes were made on memegenerator :)

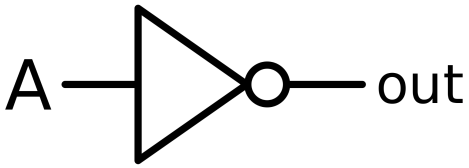
Contents:

Chapter 1: Wires, Transistors, and Gates

At the most fundamental level, computers are made of wires and transistors. Wires are made of **conductors**, which can transmit electricity freely. Transistors are a bit more complicated; we'll ignore those for the purposes of 370. The use of transistors are usually to construct **logic gates**. Logic gates take in one or more inputs and have one output. We'll consider the most common gates: NOT, AND, OR, NAND, NOR, and XOR.

The NOT Gate

The NOT gate is typically represented by the symbol shown below.



The NOT gate outputs a high signal when the input A is low, and a low signal when the input A is high. What does this all mean? To understand logic gates, we'll quickly discuss how electrical signals are handled by computers. In a computer, wires carry voltage, which is a measure of the "strength" of electricity. A high voltage will symbolize a "high" signal, which we'll represent with 1. A low voltage will symbolize a "low" signal, which we'll represent with 0. We won't go into detail about the voltage cutoffs or how transistors differentiate between low and high signals.

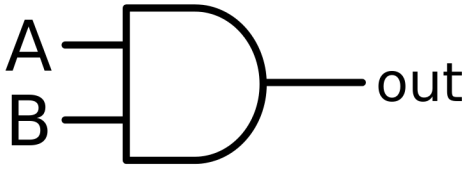
What this means for us is that when the NOT gate receives a high signal/1, it will output a low signal/0; and vice versa. In essence, the NOT gate "flips" the signal. We can represent the output of a gate with a **truth table**, which is a concise depiction of the output of a logic gate given the inputs. The truth table of the NOT gate is as follows:

A	NOT A
0	1
1	0

If we want to find the value of NOT A given that A is 0, we can check the corresponding row of the truth table where A = 0. We find that NOT A is 1. Similarly, when A = 1, we can see that NOT A = 0.

The AND Gate

The AND gate is typically represented by the symbol shown below.



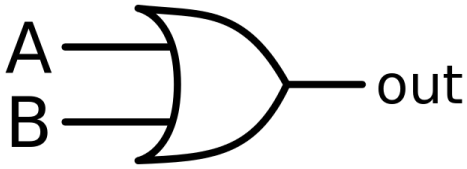
The AND gate outputs a high signal only when both inputs A **and** B are high. In all other cases, it will output a low signal. The truth table of the AND gate is shown below.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Another way to think about the AND gate is that it "passes" A when B is high, and is low when B is low. In other words, if B is high, then the output will be whatever A is. If B is low, the output will be low, guaranteed. Since there is no inherent ordering of the inputs to the AND gate, we can use this the other way around: "passing" B when A is high.

The OR Gate

The OR gate is typically represented by the symbol shown below.



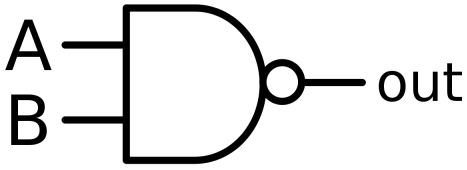
The OR gate outputs a high signal if **either** A **or** B are high. It will output a low signal when both A and B are low. The truth table of the OR gate is shown below.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

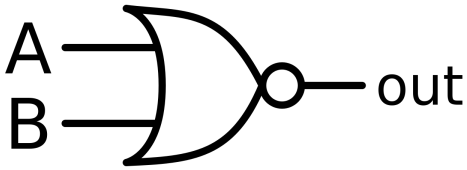
Similar to the AND gate, we can also think about the OR gate as "passing" A when B is **low**, and high when B is high. Again, we can make this analysis for "passing" B when A is low.

The NAND and NOR Gates

The NAND gate is typically represented by the symbol shown below.



The NOR gate is typically represented by the symbol shown below.



The observant reader will notice that the symbols for the NAND and NOR gate are the symbols for the AND and OR gate with an added circle. Additionally, this circle is also present in the NOT gate's symbol, on the right of the triangle. Along with the name, we get a clue to what NAND and NOR will do. NAND is equivalent to performing AND followed by NOT; while NOR is equivalent to performing OR followed by NOT. Thus, NAND is **low** when both A and B are high, and **high** otherwise; NOR is **low** when either A or B are high, and **high** if both A and B are low. The truth tables are shown below. Note that we have combined the truth tables in this case; to find the value of A NOR B or A NAND B, look in the corresponding column.

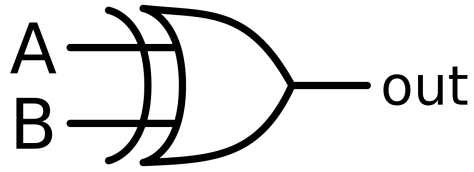
A	B	A NOR B	A NAND B
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	0

NOR and NAND have a special property: they are **functionally complete**. In other words, we can construct any logical operation out of solely NOR or NAND gates. For example, we can compute NOT A by computing A NOR A, or A NAND A. If A is 0, then A NOR A is 1, and A NAND A is also 1. If A is 1, then A NOR A is 0, and A NAND A is also 0. With the NOT gate constructed, OR follows from NOR (and AND from NAND) relatively simply, by chaining the NOR (or NAND) gate with a NOT gate.

From here, we can construct the remainder of the gates, by utilizing the NOT gate combined with De Morgan's Laws. For example, we can construct an AND gate from NOR gates by performing (A NOR A) NOR (B NOR B). This is logically equivalent to performing (NOT A) NOR (NOT B), which is logically equivalent to performining A AND B. This property will be extremely useful later on, especially when we discuss assembly.

The XOR Gate

The XOR gate is typically represented by the symbol shown below.



The "X" in XOR stands for **exclusive**: the XOR gate outputs high when **one of** A or B is high, but outputs low if A and B are both high or both low. We won't see the XOR gate as much in processors, but it is useful in a few cases. The truth table of the XOR gate is shown below.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Chapter 2: Numbers

Computers work with numbers a lot. Whether it's Wolfram Alpha doing your math homework for you or just your calculator, computers have to be able to store numbers in memory. Let's first start off with how humans store numbers in their memory, which we typically call "brains". Humans typically work with numbers in what we call base 10, or decimal. For example, the number "370" represents the value $3 * 10^2 + 7 * 10^1 + 0 * 10^0$. Let's take this same idea to computers! We'll represent each digit of our number in one wire. Immediately, we have a problem: how do we represent ten digits on one wire? We've mentioned before that usually wires carry either a low or a high voltage. We could specify that certain voltages mean certain digits, but there are many challenges associated with that[1]. With the decimal representation system, we would have to encode 10 possible values into one wire, when we can only encode 2 values on one wire. The only solution to this is to change our representation of numbers: we will use **binary**.

Binary

Binary revolves around the number 2. We choose 2 because we can represent two possible values on one wire: 0 or 1. Remember that in base 10, each digit represents a power of 10: the "3" in "370" represents $3 * 10^2$, while the "7" represents $7 * 10^1$, and the "0" represents $0 * 10^0$. In binary, each binary digit, or **bit**, represents a power of 2. Let's consider the binary number "10110_2". The subscript "2" at the end signifies that this is a binary number. We'll often denote binary numbers with a prefix "0b" as well. Thus, another way to represent "10110_2" would be "0b10110". We can find the decimal (base 10) version of 0b10110 by considering the value of each bit:

$$0b10110 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 16 + 4 + 2 = 22$$

Before we move on, I highly recommend that you either memorize or have on hand the powers of two. Recalling them quickly will help you convert binary into more familiar decimal numbers on the fly. The first few powers of two are:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072

Generally, you won't need anything past 1024 or 2048 in most scenarios.

Hexadecimal (Hex)

Hexadecimal is another common format we'll use for representing numbers. Just like decimal revolves around 10 and binary revolves around 2, hexadecimal revolves around **16**. Why 16? 2 is often too "small" once we get to larger numbers. For example, we represent decimal 255 as

[1] The main problem with this scheme is typically measurement accuracy, but it also is much harder to "add" voltages if we try to treat them as base 10 digits.

0b11111111. In hexadecimal, we can represent 255 as 0xFF ("0x" denoting hexadecimal representation). Hexadecimal allows us to represent numbers more concisely than binary and decimal.

In base 10, we have 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In binary, we have 2: 0 and 1. In hexadecimal, we now have 16 digits: 0 through 9, as well as A, B, C, D, E, and F. Why do we use A through F? Let's consider a simple example: let's count to 16 in hexadecimal.

1, 2, 3, 4, 5, 6, 7, 8, 9, ???

After this, we need something for the value "10". We can't say 0x10, since that would put the "1" into the 16^1 or 16's place. Thus, we need a single character that can represent the decimal value 10. We choose A = 10, B = 11, C = 12, D = 13, E = 14, and F = 15. Now, we can represent any number in hexadecimal. Let's consider the hexadecimal number 0xA6E. We can find its corresponding decimal value by calculating

$$10 * 16^2 + 6 * 16^1 + 14 * 16^0 = 10 * 256 + 6 * 16 + 14 * 1 = 2560 + 96 + 14 = 2670$$

Converting decimal to binary and hexadecimal

We've shown you how to convert binary and hexadecimal numbers to decimal. Let's start by converting a decimal number to binary and hexadecimal. Let's take 280 as an example. To convert 280 to binary, we need to consider the largest power of 2 that is less than 280. This is 256. We now can write 280 = 1 * 256 + 24. Now, we need to convert 24 into the sum of powers of 2. We can write 24 = 1 * 16 + 8, and finally 8 = 1 * 8. Thus, we know that

$$280 = 1 * 256 + 1 * 16 + 1 * 8.$$

From here, we can write it as a sum of powers of two:

$$280 = 1 * 2^8 + 1 * 2^4 + 1 * 2^3$$

and add in the "missing" terms:

$$280 = 1 * 2^8 + 0 * 2^7 + 0 * 2^6 + 0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$$
$$280 = 0b100011000$$

We can take a similar approach for hexadecimal. However, we need to now consider the largest power of 16, which is 256. We can write that 280 = 1 * 256 + 24. We can write 24 = 1 * 16 + 8. Finally, we write 8 = 8 * 1. Writing out the sum of powers of 16 gives us

$$280 = 1 * 256 + 1 * 16 + 8 * 1$$
$$280 = 1 * 16^2 + 1 * 16^1 + 8 * 16^0$$

Thus, 280 = 0x118.

Converting between binary and hexadecimal

The final conversion is between binary and hexadecimal. We'll be using these conversions a lot, especially once we start talking about caching and virtual memory. To convert binary to hexadecimal, or vice versa, we can convert the numbers to decimal, then to our desired form. However, there's a simple trick that we can use to make this conversion a lot easier.

Remember that hexadecimal is based around powers of 16, while binary is based around powers of 2. $16 = 2^4$. We'll keep this in mind as we go through this section. Let's consider decimal 482, which is equivalent to 0x1E2 and 0b111100010. Let's consider what these mean in their decimal representations:

$$0x1E2 = 1 * 16^2 + 14 * 16^1 + 2 * 16^0$$
$$0b111100010 = 1 * 2^8 + 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

Let's group up the terms in the binary representation in groups of four:

$$0b111100010 = (1 * 2^8) + (1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 0 * 2^4) + (0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0)$$

Now, we can factor out 2^8 and 2^4 from two of the terms:

$$0b111100010 = 2^8 * (1 * 2^0) + 2^4 * (1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0) + (0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0)$$

Now, remember that $2^8 = 2^4 * 2^4 = (2^4)^2$:

$$0b111100010 = \mathbf{(2^4)^2} * (1 * 2^0) + \mathbf{(2^4)^1} * (1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0) + \mathbf{(2^4)^0} * (0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0)$$

Expanding out the bolded powers of two gives:

$$0b111100010 = \mathbf{(16)^2} * (1 * 2^0) + \mathbf{(16)^1} * (1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0) + \mathbf{(16)^0} * (0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0)$$

We see that each group of 4 bits turns into the coefficient of a power of 16. This is again because $2^4 = 16$! Thus, to convert from binary to hexadecimal, we take groups of 4 bits, and convert them to a single hex digit. For example,

$$0b1001101101101 = 0b(1)\ (0011)\ (0110)\ (1101) = 0x(1_{10})\ (3_{10})\ (6_{10})\ (13_{10}) = 0x136D$$

We can convert hex to binary similarly:

$$0xEEC5 = 0x(14_{10})\ (14_{10})\ (12_{10})\ (5_{10}) = 0b(1110)\ (1110)\ (1100)\ (0101) = 0b1110111011000101$$

The ability to easily convert between binary and hex is why hexadecimal is so often used. To examine a certain bit, we can find which hex digit the chosen bit is found in, and then considering one of the four bits that the hex digit corresponds to.

Bitwise Operations

Now that we know how to handle numbers in binary, we'll discuss bitwise operations. Standard mathematical operations perform computations based on the numeric value. For example, calculating `5 + 3` looks at the value of 5. Regardless of how we represent it (5, 0b101, 0x5), it gets treated the same way. In contrast, bitwise operations work with the binary representation of a value.

One example of a bitwise operation is `5 & 3`, which performs a bitwise AND. Let's see how we would calculate `5 & 3`:

```
  0b0101
& 0b0011
  ------
= 0b0001
```

Recall the AND gate from Chapter 1. It outputs 1 if and only if both inputs are 1. Here, we're performing the same operation, just on each bit. Each pair of corresponding bits is considered separately in bitwise operations, and changing the value of a bit in the input only affects the corresponding output bit. This is an important property we'll revisit once we consider the uses of bitwise operations. First, let's go through the bitwise operators that are supported in C: bitwise AND, bitwise OR, bitwise NOT, and logical shifts. C also supports bitwise XOR (`a ^ b`), but we do not use it in 370 often.

Bitwise AND: `a & b`; performs the AND operation on each pair of corresponding bits[2].
Bitwise OR: `a | b`; performs the OR operation on each pair of corresponding bits.
Bitwise NOT: `~ a`; performs the NOT operation on each bit of the input.
Left Bit Shift: `a << x`; shifts the bits in a left by x places.
Right Bit Shift: `a >> x`; shifts the bits in a right by x places.

[2] In C, we also have **logical operators**, which are logical AND (&&), logical OR (||) and logical NOT (!). C handles logical operators differently than bitwise operators: it considers values in terms of only 0 or 1. For example, `5 || 3` would evaluate to 1, since 5 and 3 are nonzero and thus evaluated as 1. Mixing up logical and bitwise operators will cost you a **lot** of time in debugging.

Bitwise AND, OR, and NOT are relatively simple; they use the same logic as the AND, OR, and NOT gates described in Chapter 1, except applied bit by bit. For example, we can consider `3 | 5`:

```
  0b0000 00**11**
| 0b0000 0**1**0**1**
  -----------
= 0b0000 0111
```

Similarly, we can consider `3 & 5`:

```
  0b0000 001**1**
& 0b0000 0**1**0**1**
  -----------
= 0b0000 0001
```

We will use bitwise AND very often to perform **bit masking**. When we bit mask a number, we take the bitwise AND with another number to select which bits we want to "keep" of the original number. For example, we can consider the following bitwise AND:

```
  0b1011 0**110**
& 0b0000 0111      bit mask
  -----------
= 0b0000 0**110**
```

Notice that in our result, we select the three last bits, since those positions are 1 in the second number. The rest of the bits are 0. This takes advantage of the property that AND allows a value to "pass" if it is ANDed with 1, and always outputs 0 if a value is ANDed with 0.

Similarly, we can use bitwise OR to set specific bits. For example, we can consider the following bitwise OR:

```
  0b0000 0010
| 0b00**1**0 0000
  -----------
= 0b00**1**0 0010
```

Here, we see that we have kept the original 1 in the 2^1 place of the first number, but additionally we have set the 2^5 place to 1, which came from the second number. A special property of the bitwise OR is that we can set a bit regardless of its original state. While addition can be used in the example shown above, we can consider an example such as

```
  0b0010 0010
| 0b0010 0000
  -----------
= 0b0010 0000
```

where addition would lead to the result 0b0100 0010, which wouldn't be the intended result. We won't see bitwise OR as much as AND, but it is often useful in code to set bits, or to "join" together values by setting multiple bits at once.
Let's take a closer look at bit shifts. We'll start with the number 183, or 0b1011 0111. First, we will perform a **left shift** by 3 bits. In C, this is denoted by `183 << 3`.

183 << 0 = 0b **1011 0111**
183 << 1 = 0b**1 0110 1110**
183 << 2 = 0b**10 1101 1**100
183 << 3 = 0b**101 1011 1**000

We see that as we shift the number to the left, the original bits (**bold**) are moved to the left, and we add in 0 bits on the right hand side. Now, let's consider a **right shift** by 3 bits. In C, this is denoted by `183 >> 3`.

183 >> 0 = 0b**1011 0**111
183 >> 1 = 0b**101 10**11
183 >> 2 = 0b**10 110**1
183 >> 3 = 0b**1 0110**

The right shift moves the bits to the right. Note that as the bits go "off the end" of the number, they are cut off.

One useful property of left and right shifts is that they are equivalent to multiplication and division by 2, respectively. If we consider 183 << 1, we notice that its value is 183 * 2 = 366. Similarly, 183 >> 1 is 183 / 2 = 91 (rounded down).

Addition in Binary

We'll only cover how addition is done for now; we will discuss multiplication in a later section. Remember that in base 10, we typically do addition by adding together digits in the same position, and keeping track of the "carry". We'll do the same thing in binary. For example, let's add 0b1110 and 0b111 (decimal 14 and 7).

```
    11
  0b1110
+ 0b 111
  ------
   10101
```

We start by adding 0 and 1 in the 2^0 place (all the way on the right). We get 1, with no carry. Next, we add 1 and 1 in the 2^1 place. We get decimal 2, or 0b10. Thus, our result is 0, with carry of 1. We add this 1 with the two 1s in the 2^2 place, which gives us decimal 3, or 0b11. Our result is 1, with a carry of 1. Finally, we add 1 with 1 in the 2^3 place, giving us a result of 0 with a carry of 1. The carry of 1 adds with two implicit 0s to give us a 1 in the 2^4 place.

Sizes of Numbers

Computers will try to allocate fixed amounts of memory towards integers. For example, a standard integer in C will take up 32 bits. Here, we'll introduce a new unit: the **byte**. 1 byte is equivalent to 8 bits (binary digits), and we will use these two interchangeably. Here, a standard integer would take up 4 bytes. We'll see why we use bytes once we talk about the implementation of memory in the processor.

Two's Complement

So far, we've covered how to represent arbitrary positive numbers in binary and hexadecimal. If we want to represent bigger numbers, we use more digits. However, we don't know how to represent negative numbers yet. When we're writing out binary, we can just write –0b1010, and the reader will understand that this signifies decimal –10. We could take this approach for computers storing negative numbers: have a **sign bit**. For example, we could specify that for a 32-bit integer, the 31st bit (the 2^{31} place) will be a "1" if the number is negative, and "0" if the number is positive. This will work, but there's a problem in terms of performance: we need lots of special logic to handle adding positive and negative numbers together. What if we could have a representation that allowed us to directly add numbers together, regardless of their sign? This is what **two's complement** allows us to do.

Let's consider an example of two's complement. First of all, we need to remember that two's complement depends on having a specific size fixed for the number. This size can be arbitrary, but it must be set before we use two's complement. For a small example, let's consider integers being stored in 1 byte, or 8 bits (`int8_t` in C). Let's represent the decimal number –5 in two's complement. We'll start by writing the 8 bit representation of **positive** 5:

5 = 0b0000 0101

We've put a divider between each group of four bits to follow convention. (Remember that each group of 4 bits corresponds to one hex digit) From here, our first step will be to **invert the bits**. We will turn 1 into 0, and 0 into 1, leaving us with

0b1111 1010

Finally, we will add 1 to this value, treating it as a positive number. This results in

0b1111 1011

which is the two's complement representation of –5. Two's complement has a few useful properties. First, if the most significant (leftmost) bit is a 1, the number will be negative. We'll refer to this bit as the **sign bit** of two's complement numbers. Second, we can perform addition regularly, using standard addition as described previously. There's one catch: if we have a carry in the final bit, we discard that bit. Let's consider an example: 8 + (–5) in 8-bit two's complement. What is 8 in two's complement? While we introduced two's complement as a way to represent negative numbers, two's complement is able to represent both positive and negative numbers. We'll just treat the positive numbers normally, without going through the process of flipping bits and adding 1. Thus, 8 in 8-bit two's complement is 0b0000 1000. We know that –5 is 0b1111 1011. Now, we can set up the addition:

```
  11111
 0b0000 1000
+0b1111 1011
 ------------
   0000 0011
```

Here, we see that we have an extra "1" left after the addition, which would normally go into the 2^8 place. However, we will cut it off when adding two's complement numbers. We see that the result is 0b0000 0011, or 3 in decimal. 8 + (–5) is 3! We just added a positive and a negative number together without having to do anything special at all for the negative sign.

With two's complement, we also have access to a new shift operation: **arithmetic** right shift. We'll rename the previous bit shift to **logical** left/right shift. The key difference between arithmetic and logical shifts revolves around the sign bit of two's complement numbers Let's take a look at the difference between these, using –5:

```
5 = 0b1111 1011

    Arithmetic Shift Right 2 bits:
    0b11111 1110 = −2₁₀

    Logical Shift Right 2 bits:
    0b0011 1110 = 62₁₀
```

With arithmetic shifts, we "fill in" the empty spots with the sign bit, while with logical shifts, we "fill in" the empty spots with 0s. An arithmetic shift will preserve the sign of a two's complement number, while a logical shift will result in a positive number.

Two's Complement to Decimal

To convert a negative two's complement number to base 10, we can reverse the process, subtracting 1 and then inverting the bits to find the positive version of the number. There is a simpler conversion that we can use, however. Let's consider -5 in 8-bit two's complement: 0b1111 1011. When converting two's complement to base 10, we can treat the most significant bit as if it had a negative value than a positive value. Here, our most significant bit is the 2^7 place. If we were considering this as a non-two's complement number, we would calculate the decimal value as

$$1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$$

To treat the number as two's complement, we will invert the sign of the 2^7 term, and only the 2^7 term:

$$1 * (-2^7) + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = -5$$

IEEE-754 Floating Point

Now, we know how to deal with positive and negative integers. What happens when the number we want to represent isn't an integer? Let's start with a simple example: ¾, or 0.75 in decimal. How do we deal with decimals in base 10? Remember that 0.75 actually represents the following:

$$0.75 = 0 * 10^0 + 7 * 10^{-1} + 5 * 10^{-2}$$

We can consider the binary analogue:

$$0.75 = 0 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2}$$

From here, we can write decimal 0.75 as 0b0.11 in binary. Let's consider a hypothetical encoding where we reserve the first 16 bits for bits that come before the decimal point, and the last 16 bits for bits that come after the decimal point. The range of numbers that can be represented here spans from 0 to 2^{16} - 2^{-16}. What about negative numbers? We could treat the entire number as two's complement, making the range -2^{15} to 2^{15} - 2^{-15}. What happens for numbers outside this range? We're completely unable to represent anything larger than 2^{16}. Additionally, we only have limited precision, with 16 bits after the decimal point.

The IEEE-754 floating point representation tries to solve these problems. IEEE-754 numbers are stored in 32 bits. The bits are divided into three groups: the **sign bit**, the **exponent bits**, and the **mantissa bits**. Let's go into detail about what each one of these signifies:

Sign bit: The sign bit tracks the sign of the floating point number. If the sign bit is 1, then the number is negative. Else, the number is positive. Unlike two's complement, the sign bit does not affect the value of the floating point number besides its sign. IEEE-754 specifies that there is one sign bit.

Exponent bits: The exponent bits dictate the magnitude of the floating point number. To allow for small and large numbers, the exponent bits are offset by 127. We'll explain what this means once we get to an example. IEEE-754 specifies that there are 8 exponent bits.

Mantissa bits: The mantissa is a fancy way to say "the bits after the decimal point". To maximize the amount of information we can store in 32 bits, we insert a leading 1. We'll see how this works in an example. IEEE-754 specifies that there are 23 exponent bits.

This sounds very technical and confusing, so let's convert 0.75 to IEEE-754. We'll start with the sign bit: the number is positive, so the sign bit is 0. To find the exponent and mantissa bits, we'll start from the canonical representation of 0.75: 0b0.11 in binary. We'll start by writing this as

$$2^0 * 0b0.11$$

To convert this into IEEE-754 floating point, we want to place a **leading 1** in the 2^0 place. We can shift the binary to the left by one bit here, while accounting for the shift in the exponent.

$$2^{-1} * 0b1.1$$

From here, we can extract the exponent and mantissa. Let's start with the exponent. The exponent here is -1. Now, we need to take into account the offset of 127. We perform this by adding 127 to the exponent value, giving us -1 + 127 = 126. Converting 126 to binary gives us 0b0111 1110, which are the exponent bits. Finally, we can consider the mantissa. Remember that any number has an infinite number of leading and trailing zeros. Thus, 0b1.1 is equivalent to 0b1.1000 0000 0000 0000 0000 000. Thus, our mantissa is the 23 bits after the decimal point, or 0b100 0000 0000 0000 0000 0000. Putting it all together, we get

```
0bS  EEEE EEEE  MMM MMMM MMMM MMMM MMMM MMMM
0b0  0111 1110  100 0000 0000 0000 0000 0000
 0b 0011 1111 0100 0000 0000 0000 0000 0000
```

where S, E, and M denote the sign, exponent, and mantissa bits respectively. Converting this to hex, we get 0x3F400000.

Let's consider an example going the other way. We can consider the floating point number 0xC1B00000. First, we start by converting it to binary:

0xC1B00000 = 0b1100 0001 1011 0000 0000 0000 0000 0000

Next, we can split the bits into sign, exponent, and offset bits:

0b1 1000 0011 0110 0000 0000 0000 0000 000

The sign bit is **1**, meaning that the number is negative. The exponent bits are **0b1000 0011**, corresponding to 131 in decimal. Subtracting the fixed offset of 127 gives 4. Finally, the mantissa bits are **0b0110 0000 0000 0000 0000 000**. Thus, we can put together the decimal value as follows:

$$- 2^4 * 1.0110\ 0000\ 0000\ 0000\ 0000\ 000$$

From here, we can replace the multiplication by 2^4 with a bit shift, moving our decimal point four places to the right. This gives us 0b10110.0000 0000 0000 0000 000, or 0b10110. Note that IEEE-754 floating point numbers do not have to have bits after the decimal point; we are able to represent integers using IEEE-754 as well.

Addition in IEEE-754

Unfortunately, adding IEEE-754 numbers cannot be done using traditional add-and-carry. This is because the mantissa bits have a leading 1, and also may represent numbers of different magnitudes. To add two IEEE-754 floating point numbers, we need to follow four steps: sign comparison, normalization, addition/subtraction, and renormalization. We'll work through an example of adding two floating point numbers: 0xC05A0000 and 0x3FE40000.

Step 1: Sign Comparison

We start by extracting the sign bit of each number.

0xC05A0000 = 0b**1**100 0000 0101 1010 0000 0000 0000 0000
0x3FE40000 = 0b**0**011 1111 1110 0100 0000 0000 0000 0000

Here, we see that 0xC05A0000 is negative, while 0x3FE40000 is positive. Thus, we will have to perform subtraction.

Step 2: Normalization

Next, we need to extract the exponent and mantissa bits, and write out the value of each floating point number. Let's start with 0xC05A0000:

$$\text{Exponent bits} = 0b1000\ 0000 = \text{decimal } 128 - 127 = 2^1$$
$$\text{Mantissa bits} = 0b1011\ 0100\ 0000\ 0000\ 0000\ 000$$

Now, let's do the same for 0x3FE40000:

$$\text{Exponent bits} = 0b0111\ 1111 = \text{decimal } 127 - 127 = 2^0$$
$$\text{Mantissa bits} = 0b1100\ 1000\ 0000\ 0000\ 0000\ 000$$

Since we extracted the sign bits previously, we know whether the numbers are negative or positive. To make it simple to perform addition, we will write out the numbers in the format $2^x * 1.(\text{mantissa})_2$. Remember that the subscript "2" indicates that the number is in binary.

Thus, we know that the value of 0xC05A0000 is:

$$-2^1 * 1.1011\ 0100\ 0000\ 0000\ 0000\ 000_2$$

and the value of 0x3F640000 is:

$$+2^0 * 1.1100\ 1000\ 0000\ 0000\ 0000\ 000_2$$

To perform normalization, we need to make the exponents equal. Let's shift the smaller magnitude number to the right, while adjusting the exponent to keep the value the same:

$$+2^0 * 1.1100\ 1000\ 0000\ 0000\ 0000\ 000_2$$
$$= +2^1 * 0.1110\ 0100\ 0000\ 0000\ 0000\ 0000_2$$

Step 3: Subtraction

Now, we can perform our subtraction:

```
  0b0.1110 0100 0000
- 0b1.1011 0100 0000
  ------------------
```

There's a slight problem here though: we're subtracting a larger number from a smaller number! Recall that (A – B) is equivalent to –(B – A). Let's use this fact to help us perform this

subtraction. Remember that in IEEE-754, we can easily flip the sign of a number by changing solely the sign bit. Thus, we can calculate the result of

```
  0b1.1011 0100 0000
- 0b0.1110 0100 0000
  ------------------
  0b0.1101 0000 0000
```

Then, we just have to remember to flip the sign to become negative. If the number with a larger magnitude is already positive, the resulting sign will be positive.

Step 4: Renormalization

Finally, we need to renormalize. Similar to what we did when we converted 0.75 to IEEE-754, we need to ensure that we have a leading 1 in the 2^0 place. Remember that we calculated the previous result assuming an exponent of 2^1. The exponent is 2^1 because this is what we used as the common exponent in the normalization step. Thus, we currently have:

$$2^1 * 0.1101\ 0000\ 0000\ 0000\ 0000\ 000_2$$

To find the IEEE-754 mantissa, we will shift the number left by 1, adjusting the exponent to account for this change.

$$2^0 * 1.1010\ 0000\ 0000\ 0000\ 0000\ 000_2$$

Now, we can flip the sign to account for the reversed subtraction order:

$$-\ 2^0 * 1.1010\ 0000\ 0000\ 0000\ 0000\ 000_2$$

Finally, this is in a format we can easily convert to IEEE-754. The sign bit is 1, to indicate the number being negative. The exponent bits correspond to decimal 127 + 0 = 127, or 0b0111 1111. The mantissa bits are 0b1010 0000 0000 0000 0000 000. Thus, we can put this together to get 0b1011 1111 1101 0000 0000 0000 0000 0000, or 0xBFD00000. To confirm this answer, we can convert the two floating point numbers to decimal, and then add them and compare with 0xBFD00000 in decimal:

$$0xC05A0000 = -2^1 * 1.1011\ 0100\ 0000\ 0000\ 0000\ 000_2 = -0b11.01101$$
$$= -(2 + 1 + 2^{-2} + 2^{-3} + 2^{-5}) = -3.40625$$

$$0x3F640000 + 2^0 * 1.1100\ 1000\ 0000\ 0000\ 0000\ 000_2 = 0b1.11001$$
$$= (1 + 2^{-1} + 2^{-2} + 2^{-5}) = 1.78125$$

Adding these values gives us -1.625. We can consider the value of 0xBFD00000:

$$0xBFD00000 = -2^0 * 1.1010\ 0000\ 0000\ 0000\ 0000\ 000 = -0b1.101$$
$$= -(1 + 2^{-1} + 2^{-3}) = -1.625$$

which matches our expected result. In general, when adding IEEE-754 floating point numbers, you will have to perform the sign comparison and addition/subtraction steps. Normalization will often have to be performed, unless the two numbers start with the same exponent bits. Renormalization may have to be performed, depending on the result of addition/subtraction. In some cases, renormalization may not be needed. In other cases, renormalization will require a right shift rather than a left shift.

Chapter 3: Digital Logic

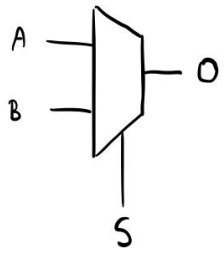
Before we start talking about how computers work, let's talk about the circuits that make computers possible. When designing a computer, we try to abstract away details of how the hardware is set up. Previously, we've seen how we can abstract away groups of transistors into logic gates such as AND and OR. Now, we'll combine logic gates into groups we call **components**.

Components

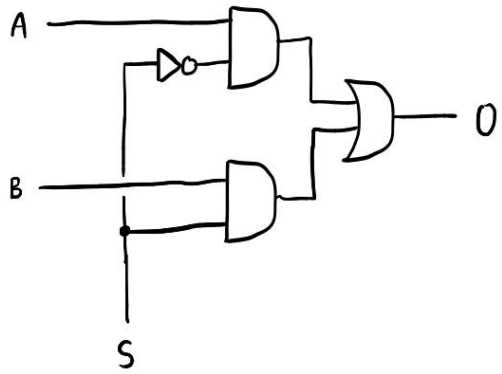
There are a wide variety of components that make computers possible. For the purposes of 370, we will talk about the MUX, the half adder, the full adder, the RS NOR latch, the D latch, and the D flip flop.

MUX

We'll start with the simplest component: a **multiplexer**, or **MUX**. A MUX is a component that allows selection between two inputs. MUXes are typically represented by the symbol shown below:



Here, we have two input bits A and B, a **select bit** S, and an output bit O. The basic principle behind a MUX is that if S is 0, then O = A. Else (when S is 1), then O = B. Let's take a look at how this is implemented with logic gates:



Let's recall a property of the AND and OR gates that we mentioned in Chapter 1: we can consider them as gates that "pass" a signal when the other input is either 0 or 1. Let's first

consider the two AND gates. The top AND gate takes in A and NOT S. Thus, we can treat this as "passing" A when NOT S is 1, and outputting 0 if NOT S is 0. Similarly, the lower AND gate "passes" B when S is 1, and outputs 0 if S is 0. Now, let's consider the relationship between S and NOT S. Remember that when S is 1, NOT S is 0, and vice versa. Thus, we have two cases:

Case 1: S = 0; NOT S = 1. Thus, the top AND gate will output the value of A, while the bottom AND gate will output 0. The OR gate receives a 0, meaning that it will output the value of the other input, which is A.

Case 2: S = 1; NOT S = 0. Thus, the bottom AND gate will output the value of A, while the top AND gate will output 0. The OR gate receives a 0, meaning that it will output the value of the other input, which is B.

Thus, we see that when S = 0, then regardless of the value of A or B, the MUX will always output the value of A. When S = 1, the MUX will always output the value of B. This ability to select between two inputs will be used extensively when we discuss processor datapaths.

Half Adder

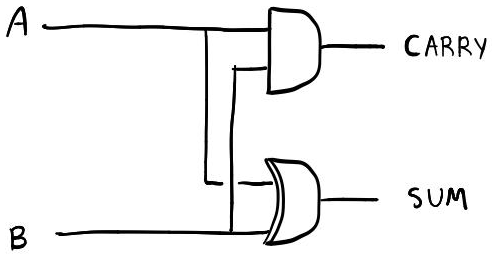
As we'll see later on, processors have to do a lot of addition. Let's create a circuit to add two bits together. Our desired output can be summarized in the following table:

A	B	Sum
0	0	0
0	1	1
1	0	1
1	1	10

There's a slight problem: we can't represent "10" in a single bit. Remember how we deal with this case in manual addition: we carry over a "1" into the next bit. Thus, we can split our output into two bits: a sum and a carry bit. Our desired behavior is now

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

This might look daunting to implement, so let's break it down. First, we have the sum bit. We notice that the sum bit is 1 when either A or B is 1, but not when both A and B are 1. Recall that the XOR gate has this exact functionality. The second part is the carry bit. Here, the carry bit is 1 only when both A and B are 1, meaning we have to use an AND gate. Thus, we can implement the half adder with an XOR gate and an AND gate, as shown below.



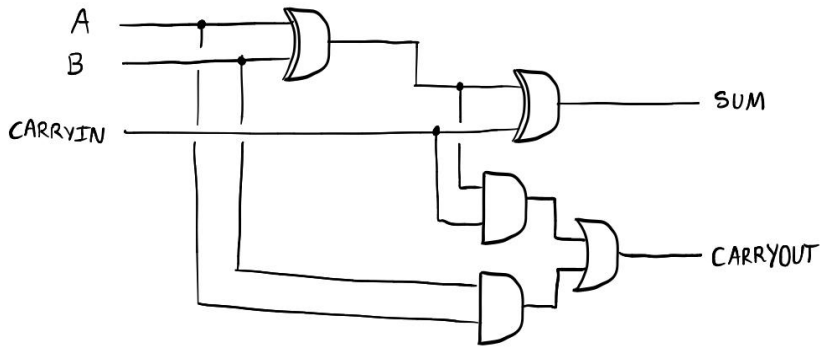
There's a slight catch: how do we handle further addition? We need to design a component that takes in A, B, and the carry bit from the previous bit, and outputs a carry and a sum. That's where the full adder comes in.

Full Adder

The full adder is similar to the half adder, but also takes into account the carry bit of the previous bit. The desired behavior is

A	B	CarryIn	Sum	CarryOut
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The circuitry for the full adder is a bit more complicated due to the 8 possibilities. To implement the full adder, we can take some inspiration from the half adder. We'll start by taking the XOR of A and B, followed by an XOR with the CarryIn. To determine the carry bit, we can check if either of the pairs of inputs to the XOR gates were a pair of 1s. If so, then we need to carry a 1. The final design is as follows:

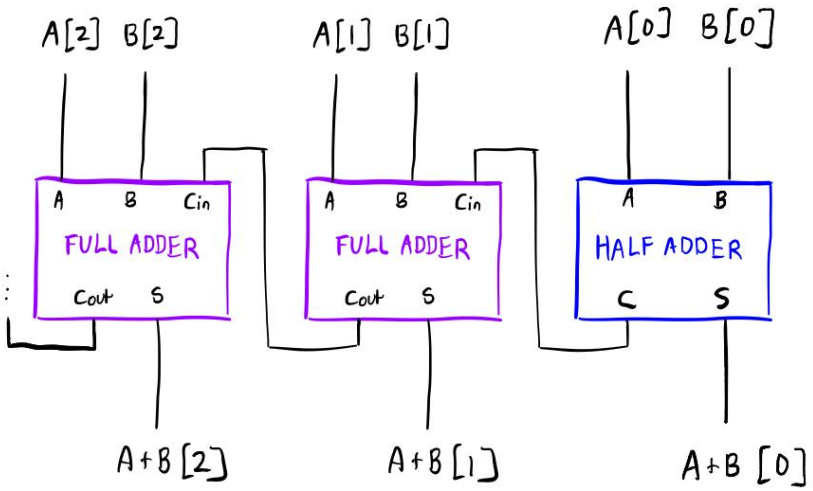


Numbers in Hardware

So far, our components work with individual bits. However, we eventually want to handle larger numbers beyond 0 and 1. How do we handle these? We can keep the bits "grouped" together in hardware by considering them as wires running alongside each other. Typically, we will refer to the bits as bit 0 for the 2^0 place, bit 1 for the 2^1 place, and so on. To denote multiple bits, we will denote a wire with "X[0:31]", for example. This indicates that we have an input X that consists of 32 bits (bits 0 through 31).

Ripple Adders

Now that we have the half and full adders, we can combine them to add up larger numbers. Let's consider adding 4 bit numbers. We will start by adding the 2^0 places together using a half adder. This gives us a sum, which will be the 2^0 place of our result. We also get a carry bit, which we will send to the 2^1 place. There, we use a full adder to give us a sum and a carry. The sum goes into the result, and the carry is sent to the 2^2 place. We can repeat this process as many times as needed to add all the bits. The process is shown below:

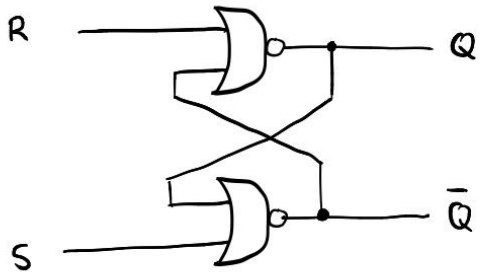


We call this setup a **ripple adder**, since the carry bit "ripples" through each adder. As mentioned before, this setup allows us to add numbers of arbitrary size.

RS NOR Latch

So far, all the components we've mentioned are **stateless**: they do not remember any information about the inputs they get, but rather determine their outputs based on the current state of their inputs. However, computers need to be able to remember values! We need to create a circuit that both stores a value for an arbitrary amount of time, as well as allows updates to the stored value. At first, this seems like a daunting task, since all the logic gates we've mentioned are stateless. How can we construct a memory cell out of stateless components?

The answer here is to utilize **feedback**. We can set up a feedback loop by using the output of a circuit as one of its inputs. Here, we'll set up a feedback loop between two NOR gates, as shown below:



At first, this doesn't seem like it does anything useful. We're just sending the output back into the inputs, and it could have any sort of behavior. However, let's consider what the potential **steady states** are.

First, let's start with the case where R = 1 and S = 0. We know that R = 1, meaning that the top NOR must be 0 regardless. Thus, Q, which is the output of the top NOR, is 0. We feed in 0 to the bottom NOR, and S is 0. Thus, the output of the bottom NOR is 1.

Next, let's consider the case where R = 0 and S = 1. We know that the bottom NOR must be 0 using similar logic to the first case. Thus, we feed 0 and 0 into the top NOR, making the value of Q 1.

Finally, let's consider the case where R = 0 and S = 0. We can't make any deductions about the outputs of the NOR gates yet, so let's split this into two subcases. In the first subcase, we'll assume Q is 1[3]. Then, we know that the output of the bottom NOR gate must be 0, since Q is 1. Thus, the output of the top NOR gate will become 1, maintaining the current state. In the second subcase, we'll assume Q is 0. Then, the output of the bottom NOR gate will be 1, meaning that the output of the top NOR gate is 0. This continues to maintain the state Q = 0.

After going through these cases, we can put together a pattern of behavior for the RS-NOR latch:

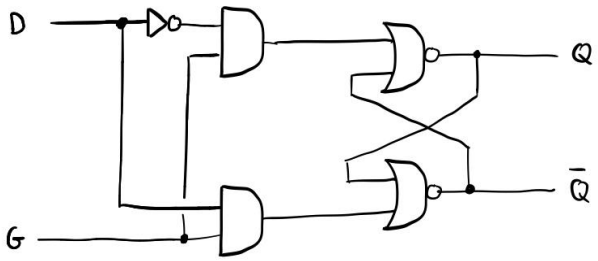
- If R = 1: Q = 0
- If S = 1: Q = 1
- If R = S = 0: keep Q the same

[3] You may be wondering here: why can we just "assume" Q is 0 or 1, since it would depend on the state of R and S? We'll talk about propagation delay later, but the basic concept is that the gates take some nonzero amount of time to update their output. Thus, we can consider the old value of Q as the input to the bottom NOR gate, and the same for the top NOR gate.

Congratulations, we've designed a memory cell! You may have noticed that we didn't include the case where R = S = 1. In this case, both NOR gates output 0, and this puts us into an inconsistent state. We design the RS-NOR latch such that in any valid state, the two output wires must be opposites of each other. This means that if both are 0, the output is undefined, which is not good[4]! However, there's nothing that actually stops us from setting R = S = 1. Let's fix that with the D Latch.

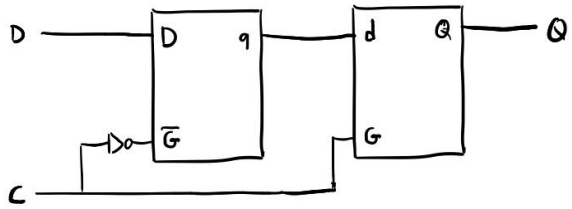
D Latch

The D Latch, or gated D Latch, is designed to avoid inconsistent state in the RS-NOR latch. By negating one of the inputs of the RS NOR latch, we ensure that the inputs can never be both 1 at the same time. We now have a data input, D, which corresponds to data we want to save. Additionally, we introduce a gating signal G. If G is 0, then the D latch maintains its old state. However, if G is 1, then the D latch will update its state with the value of D.



D Flip Flop

Finally, we have the D Flip Flop. Remember that the D Latch "follows" the input signal when G is 1. What if we want to only store a single value at a single point in time? We can use the D Flip Flop. To construct the D Flip Flop, we will chain together two D Latches as follows:



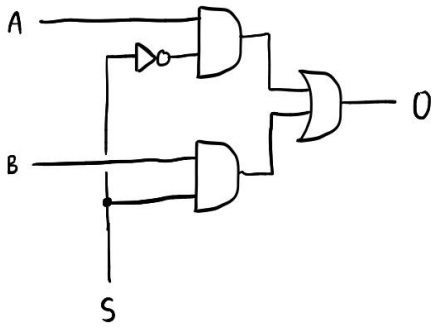
This specific setup is a **rising edge D flip flop**, where Q is updated to match D every time the signal C goes from 0 to 1 (a **rising edge**). We refer to C as the **clock signal**. All processors have a clock circuit which produces an output alternating between 0 and 1 at a regular frequency. On a rising clock edge, all flip flops in the processor will update their value,

[4] The behavior in this case would lead to a random choice between 0 or 1 depending on environmental factors, assuming both R and S are set to 0 immediately after.

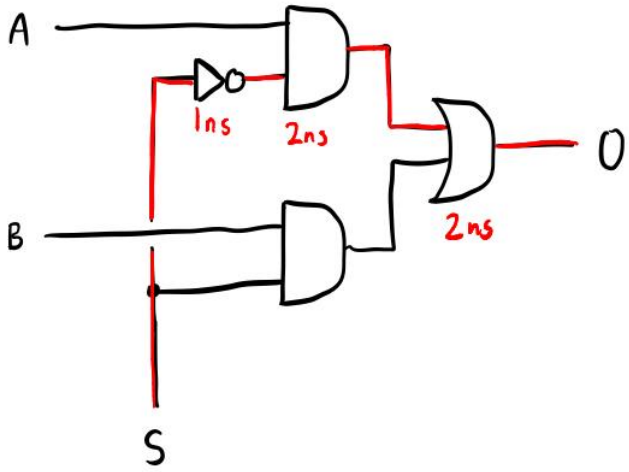
allowing us to send arbitrary intermediate values to the flip flops without worrying about changing the values unintentionally.

Propagation Delay

We've talked a lot about how to put together gates to perform more complex operations, but so far, we've assumed everything updates instantaneously. While electricity is fast, it is not instantaneous. Each gate that we have in our circuit takes a small amount of time to update its output. Let's consider a simple example: a MUX. Recall that the MUX consists of four gates: one NOT gate, two AND gates, and one OR gate, as shown below.

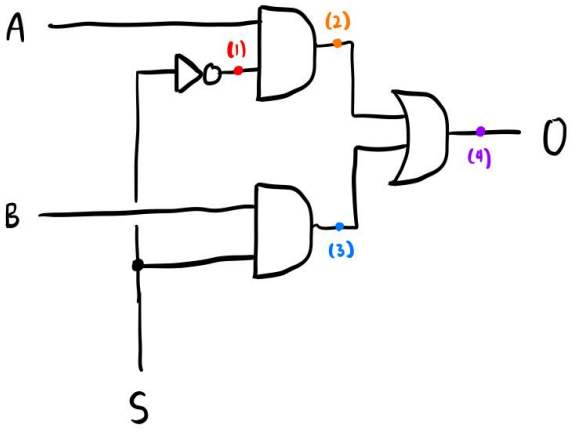


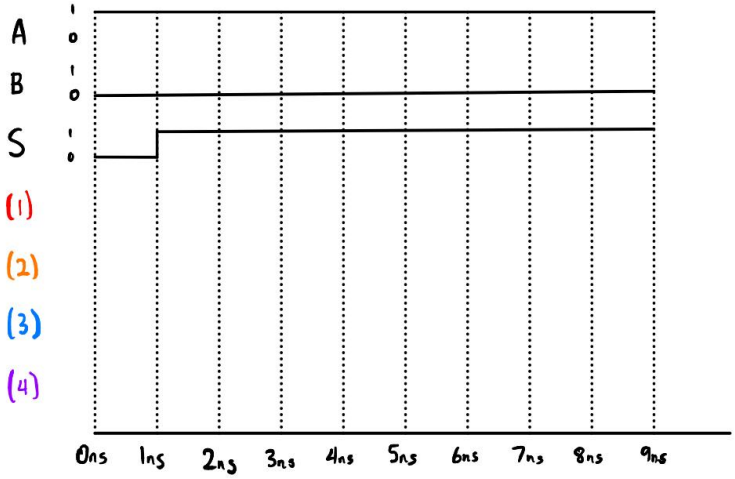
Let's assume that the output of AND and OR gates takes 2 nanoseconds (ns) to stabilize, while the output of a NOT gate takes 1 nanosecond to stabilize. Stabilization refers to the fact that the output may "bounce" between 0 and 1. We only want to consider the output of the gate once it has passed this phase. Let's find the propagation delay of the MUX as a whole. To find this, we need to find the path with the longest total delay. This path is highlighted in red in the figure below, and has a total delay of 5 ns.



If we consider a change in S from 0 to 1 at time 0 ns, we would see the output of the NOT stabilize at time 1 ns. After that, the top AND gate would stabilize 2 ns later, at 3 ns. Finally, the OR gate stabilizes at 5 ns. We can consider all other paths, but no path will take longer than 5 ns to stabilize. Thus, the propagation delay of this MUX is 5 ns.

Besides propagation delay, we can also look at the specific values of the output over time. Let's consider the same example as above. We'll set A = 1 and B = 0. S initially starts at 0, but changes to 1 at time 1 ns. Assume that A, B, and S have been in this state for a long time. We can represent this in a graph of A, B, and S over time.

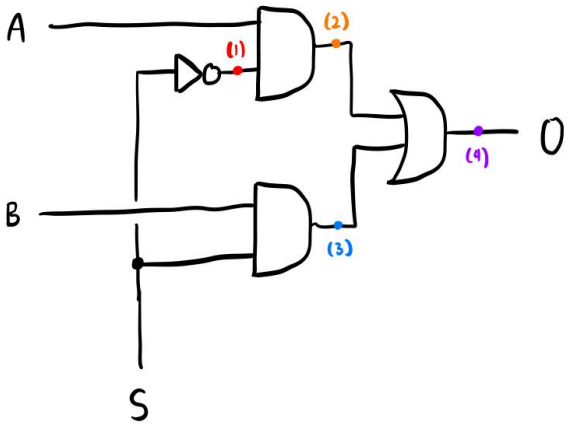


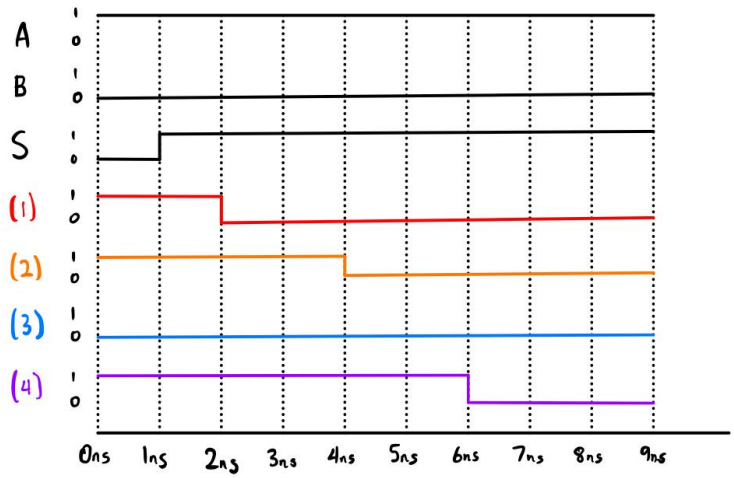


Next, let's consider point (1). We know that there is a propagation delay of 1 ns from S, meaning that at time 2 ns, the output of the NOT gate will stabilize at 0. From there, the output of the top AND gate, at (2), will take a further 2 ns to stabilize, stabilizing at a value of 0 at time 4 ns. We can plot the ouput of (1) and (2) on our graphs as well.

Let's now consider point (3). After S changes to 1 at time 1 ns, it takes 2 ns for the bottom AND gate to update. Thus, the value at (3) will stabilize at 3 ns with the new value of 0. However, the old value was already 0, so we see no change[5].

Finally, we can consider point (4). It originally starts off as 1. At time 4 ns, the value of (2) changes from 1 to 0. Since (3) is 0, we know that the value at (4) must become 0 after a 2 ns stabilization time. Thus, the value of (4) will stabilize to 0 at time 6 ns.
The final graph of each signal over time is shown below.





Notice that the time between the original change of S at 1 ns and the final stabilization of the output (4) at 6 ns is 5 ns, which is the propagation delay we calculated previously.

[5] We may see a quick fluctuation to 1 during the 2 ns of stabilization time, due to changes inside the AND gate. However, we'll ignore that for the purposes of 370.

Chapter 4: Assembly

It's finally time to write some programs! You may have experience writing code in C (or similar languages), for example:

```
int x = 5;
int y = 3;
y = x + y;
```

What really happens when we run this code? When we want to run a C program, we need to compile it, for example

```
$ gcc program.c
```

This creates a file called `a.out`, which is an executable file. If we try to open this executable file, it looks completely unreadable. The contents of executable files are in a format called **machine code**. Machine code is the lowest level of programming languages[6]: it is binary that the CPU can directly understand. It is possible to write machine code directly, but remembering the sequences of 0s and 1s that make up a certain operation is very difficult! Thus, we will use **assembly language**, which is a human-readable form of machine code. We can see assembly code by running a disassembler (such as **objdump**) on the executable file. We might get something along the lines of:

```
mov     DWORD PTR [rbp-4], 5
mov     DWORD PTR [rbp-8], 3
mov     eax, DWORD PTR [rbp-4]
add     DWORD PTR [rbp-8], eax
```

What does this all mean? And how does this even correspond to the previous code we showed? Over the next few chapters, we'll discuss the workings of assembly, various assembly languages, and how to program in assembly. Let's get started!

The LC2K ISA

Let's consider a simple assembly language, which we'll call LC2K (Little Computer 2000). The specifications of LC2K are as follows:

LC2K is a word-addressable ISA with 65,536 words of memory and 8 registers. The LC2K ISA uses 32-bit words, and contains 8 opcodes:

[6] When we say a language is low-level, we don't mean that it is less powerful than high-level languages; a low-level language is "closer" to machine code. High level languages typically have to be compiled into lower level language, and eventually to machine code, before being run.

- **`add regA regB destReg`**: adds the contents of **regA** and **regB** and stores the sum into **destReg**.
- **`nor regA regB destReg`**: computes the bitwise NOR of **regA** and **regB** and stores the result into **destReg**.
- **`lw regA regB offset`**: loads the value in memory at address **regA + offset** into register **regB**.
- **`sw regA regB offset`**: stores the value of **regB** into memory at address **regA + offset**.
- **`beq regA regB offset`**: if the values in **regA** and **regB** are equal, branch to PC = PC + 1 + **offset**. If the values in **regA** and **regB** are not equal, then we do not branch.
- **`jalr regA regB`**: store PC + 1 into **regB**, and then branch to the value stored in **regA**, in that order.
- **`halt`**: stop the machine.
- **`noop`**: no-operation. There is no change to the state of registers or memory.

What does this all mean? Let's start with the description: a word-addressable ISA wih 65,536 words of memory and 8 registers. The term ISA stands for **instruction set architecture**. An ISA is a specification for a processor: how much memory is available, how many registers are available, addressability, the instructions that can be used, and so on. The ISA **doesn't** specify how the CPU will be implemented, or factors such as the clock speed. We'll see that there are many ways to implement an ISA later on.

Word-addressability refers to the "fundamental unit" of memory in the processor. To access memory, we need to provide a unique **address**. This address acts similar to an array index: specifying a specific location in memory to read or write from. We will see two general types of addressability: byte-addressability and word-addressability. A byte-addressable ISA uses 1 byte as the "fundamental unit" of memory. For example, ARMv8 (which we'll discuss later in this chapter) is a byte-addressable ISA. Thus, accessing address 0x5 would give us a single byte (8 bits). An ISA that is word-addressable will have more than 1 byte corresponding to each address. For example, LC2K is word-addressable with 4-byte (32-bit) words. Thus, when we access address 0x5, we will get 4 bytes from memory at once. When we access address 0x6, we will get a completely different set of 4 bytes.

Finally, let's consider memory and registers. Memory refers to **main memory**, which is typically found close to the CPU. LC2K contains 65,536 words of memory. Since LC2K is a word-addressable architecture, this means that we can access 65,536 unique locations in memory. Registers are memory circuits which are placed inside the processor and designed for efficiency, allowing extremely quick access. We use register values in our instructions because this allows the instructions to run much quicker than working with memory values. Due to the limited space in the processor, we won't have the ability to store hundreds of registers. In LC2K, we have 8 registers, while ARMv8 has 32 registers. When we mention the value of a register in an ISA specification, we work with the number stored in the register. For example, in LC2K, each register holds a 32-bit value, while in ARMv8, each register holds a 64-bit value.

The register value itself does not have a specific format, such as two's complement or IEEE-754, but ISA instructions may assume a specific format when performing their operation.

Now that we know a bit more about how LC2K is set up, let's take a look at the instructions. We classify the instructions into four types: R type, I type, J type, and O type. This will be useful when we are converting LC2K assembly into machine code. We'll discuss converting LC2K assembly into machine code in the next section of this chapter.

`add regA regB destReg`: R type instruction
The **add** instruction takes the values in registers regA and regB and adds them to each other. The result is stored into the destReg register. For example, if register 1 contains the value 5 and register 2 contains the value 3, then **add 1 2 3** will take the sum 5 + 3 and store it into register 3. Thus, the value of register 3 will be updated to 8. **destReg** does not have to be different from **regA** or **regB**, a command such as **add 1 1 1** is completely valid, and will double the value of register 1 (add it to itself).

nor regA regB destReg: R type instruction
The **nor** instruction takes the values in registers regA and regB and takes the bitwise NOR. The result is stored into the destReg register. Similar to **add**, there are no restrictions on the registers used.

lw regA regB offset: I type instruction
The **lw** (load word) instruction loads a single word from memory into **regB**. The address is specified by the value in **regA**, plus the value of **offset**. Here, **offset** is an **immediate value**, which is a constant value specified in the machine code, as opposed to a value stored in a register. Thus, **lw 0 1 2** would calculate the address to load from by taking the value stored in **register 0**, and then adding **2**. The address loaded here does **not** depend on the value in register 2.

sw regA regB offset: I type instruction
The **sw** (store word) instruction stores the value from **regB** into memory. Similar to **lw**, the address is specified by the value in **regA** plus the immediate value **offset**.

beq regA regB offset: I type instruction
The **beq** (branch if equal) instruction is a **conditional branch**. To understand conditional branches, we need to first talk about the **program counter**, or **PC**. The PC keeps track of "where" in our program we are at. In 370, we will mainly consider **Von Neumann architectures**. In a Von Neumann architecture, there is no distinction between instructions and data in memory; both are stored in the same memory. The PC keeps track of the memory address of the next instruction. We'll see how the PC is used when we go through processor design.

In the **beq** instruction, we first compare the values of registers **regA** and **regB**. If the two values compare equal, we update our PC to equal the **current PC plus 1 plus the immediate**

value offset. Note that the branch target is **PC-relative**, meaning that with a constant offset, the branch target will vary depending on where the **beq** instruction is found in memory. If the values do not compare equal, there is no change to program flow. We'll see how **beq** is used when we look at converting C to LC2K.

jalr regA regB: J type instruction
The **jalr** (jump and link register) instruction performs an **unconditional branch** to a value stored in a register. The operations performed are:

1. Store current PC + 1 into **regB**
2. PC = value of **regA**

This allows us to branch to an address specified by the value of a register. While this does not seem very useful right now (why not just use **beq** if we know where to go?), we'll see how **jalr** is used when we discuss function calls.

halt: O type instruction
The **halt** instruction is used to terminate program execution.

noop: O type instruction
The **noop** (no-operation) instruction is an instruction that, by definition, does nothing. Why do we have an instruction that does nothing? When we discuss pipelining, we'll see how **noop** comes in handy when we want the processor to "wait" for certain operations to finish.

Running Assembly Code

Running assembly code on a processor takes three steps: assembling, linking, and loading. For non-assembly languages, we will add in a fourth step, compiling, before the assembling step. The assembling step is the first step, and involves converting assembly code into a format known as an **object file**. Object files are used by the linking step to create an **executable**. Executable files consist of machine code, and provide a machine-readable format for the computer to run. Finally, the loading step places an executable into memory, allowing the processor to run the program. We'll see each of these steps in action with LC2K.

Converting LC2K to Machine Code

For now, we'll consider assembly programs with only a single file. This allows us to skip the linking step, converting assembly directly into machine code. Let's consider a simple assembly program:

```
            lw          0     1     four
            add         1     2     3
            halt
four        .fill       4
```

You'll notice two things here that don't look standard: the usage of "four", as well as the ".fill". Let's start with ".fill". .fill is an **assembler directive**, which signifies to the assembler that we want to place a constant value in memory. The assembler is a program that covnerts assembly code, such as the code shown above, into machine code. This section gives an overview of how the LC2K assembler works.

Next, we notice "four" in the assembly code. Here, we use "four" as a **symbolic label**, which assigns a name to a specific line. In LC2K, we define symbolic labels by specifying their name on the **left** side of the opcode or directive. For example, "four" is defined on the fourth line of this program. We use symbolic labels to make it easier for programmers to modify and debug their code. While using a symbolic label is equivalent to putting in a fixed immediate value, when we modify the program, the values of the offset would change. Let's consider a version of the program above that performs the exact same operations, and will translate to the same machine code, but doesn't use symbolic labels:

```
            lw          0     1     3
            add         1     2     3
            halt
            .fill       4
```

We're running our program, and then we realize we forgot to store our result to memory! We need to add a **sw** instruction after the **add** to store the result to memory. We'll also add a new **.fill** directive to indicate a location to store our result into:

```
            lw          0     1     3
            add         1     2     3
            sw          0     1     5
            halt
            .fill       4
            .fill       0
```

If we run this program, we'll get the completely wrong result! This is because we forgot to update the offset field for the **lw** instruction, meaning that we load the wrong value into memory. If we used symbolic labels, the update program would be:

```
            lw          0     1     four
            add         1     2     3
            sw          0     1     result
            halt
four        .fill       4
result      .fill       0
```

Here, we don't have to worry about updating any of our offsets, because the assembler will compute the correct value of **offset** at assemble time.

Now that we understand how assembler directives and symbolic labels work, let's convert our LC2K assembly code into machine code! First, we will need to know how each instruction is encoded into memory. This is where the aforementioned R, I, J, and O type designations come into play. When encoding assembly instructions in machine code, we will use individual bits of a word to encode information about the instruction. Let's start with the simplest case: encoding an **add** instruction into machine code. Let's consider **add 1 2 3**. We know that **regA = 1**, **regB = 2**, and **destReg = 3**. To encode a R-type instruction in LC2K, we use the following scheme:

- Bits 24-22 = opcode
- Bits 21-19 = regA
- Bits 18-16 = regB
- Bits 2-0 = destReg
- Bits 31-25, 15-3 are unused and are set to 0

You may notice a term we haven't mentioned before: **opcode**. When encoding machine code, we try to make everything as compact as possible. Thus, each instruction (**add**, **nor**, etc.) is given a unique number, which we call an opcode. Often, we will use opcode to refer to the instruction name in assembly, since the two are equivalent to each other. We'll refer to the values that are used by an instruction, such as register numbers and immediate values, as **operands**. In LC2K, the opcodes are assigned as follows:

add = 0
nor = 1
lw = 2
sw = 3
beq = 4
jalr = 5
halt = 6
noop = 7

When we convert an instruction into machine code, we need to consider the binary representation of various fields in our assembly instruction. Thus, we will encode the opcode, as well as the registers used into binary. The opcode for **add** (0) becomes 0b000, while the three register operands are 0b001, 0b010, and 0b011 respectively. Now, we can encode the instruction into machine code. We will follow the scheme shown above, which will result in a 32-bit number.

unused	opcode	regA	regB	unused	destReg
31-25	24-22	21-19	18-16	15 - 3	2-0
0b0000000	000	001	010	0000000000000	011
assembly:	**add**	**1**	**2**		**3**

Notice how we can encode each of the different parts of the instruction into machine code. Putting this together gives us a value of 0b0000 0000 0000 1010 0000 0000 0000 0011, or 0x000A0003.

Let's take a look at a more complicated example. We're going to consider the LC2K assembly program shown below. Don't worry for now about what the program does. To assemble this program into machine code, we need to handle LC2K instructions, symbolic labels, and `.fill` directives.

```
            lw          0     1     N
            lw          1     2     3
start       beq         0     1     2
            add         1     2     1
            sw          1     1     Arr
            beq         0     0     start
end         noop
            halt
N           .fill       6
negOne      .fill       -1
Arr         .fill       0
```

First, we'll start by handling all symbolic labels. Let's make a table of where each label is defined. We'll use 0-indexed line numbers here to stay consistent with memory addresses.

start: defined on line 2, used in line 5
end: defined on line 6, not used
N: defined on line 8, used in line 0
negOne: defined on line 9, not used
Arr: defined on line 10, used in line 4

Next, we can go through and **resolve** each symbolic label: replacing each symbolic label with its corresponding immediate value. For example, we resolve **lw 0 1 N** by substituting the line number of **N**, which is 8, for **N**. There's a small catch: for **beq**, we need to update the offset field to set the branch target to the symbolic label that is referenced. For example. **beq 0 0 start** will be updated such that if the branch is taken, then the next instruction executed will be at the line corresponding to the symbolic label "start". The PC of **beq 0 0 start** is 5 (since it is on line 5), and the target PC is 2 (since "start" is defined on line 2). Thus, we can calculate

what the offset needs to be by solving the equation PC + 1 + offset = target PC. Substituting our values gives us 5 + 1 + offset = 2. Thus, our offset needs to be -4.

Doing the above for each instruction gives us the following assembly:

```
            lw          0     1     8
            lw          1     2     3
start       beq         0     1     2
            add         1     2     1
            sw          1     1     10
            beq         0     0     -4
end         noop
            halt
N           .fill       6
negOne      .fill       -1
Arr         .fill       0
```

Next, we can go through and convert each instruction to machine code. Let's first take a look at the encoding schemes for all four types of instructions:

R-Type Instructions: **opcode regA regB destReg**

- Bits 31-25: unused, set to 0
- Bits 24-22: opcode
- Bits 21-19: **regA**
- Bits 18-16: **regB**
- Bits 15-3: unused, set to 0
- Bits 2-0: **destReg**

I-Type Instructions: **opcode regA regB offset**

- Bits 31-25: unused, set to 0
- Bits 24-22: opcode
- Bits 21-19: **regA**
- Bits 18-16: **regB**
- Bits 15-0: **offset**, in 16-bit two's complement

J-Type Instructions: **opcode regA regB**

- Bits 31-25: unused, set to 0
- Bits 24-22: opcode
- Bits 21-19: **regA**
- Bits 18-16: **regB**
- Bits 15-0: unused, set to 0

O-Type Instructions: **opcode**

- Bits 31-25: unused, set to 0
- Bits 24-22: opcode
- Bits 21-19: unused, set to 0

We see that bits 31-25 are always 0, while bits 24-22 always contain the opcode. This will be useful when we want to create a processor to run LC2K machine code. Let's start with the first instruction: **lw 0 1 8**. We know that **lw** is an I-Type instruction, with an opcode of 2 (0b010). Thus, we encode the instruction as follows:

```
0b0000000 (unused) 010 (opcode) 000 (regA) 001 (regB)
                   0000 0000 0000 1000 (offset)
= 0b0000 0000 1000 0001 0000 0000 0000 1000 = 0x00810008
```

Similarly, we can encode **lw 1 2 3** with the same process:

```
0b0000000 (unused) 010 (opcode) 001 (regA) 010 (regB)
                   0000 0000 0000 0011 (offset)
= 0b0000 0000 1000 1010 0000 0000 0000 0011 = 0x008A0003
```

beq 0 1 2 is also an I-type instruction, this time with an opcode of 4 (0b100). Thus, we encode the instruction as follows:

```
0b0000000 (unused) 100 (opcode) 000 (regA) 001 (regB)
                   0000 0000 0000 0010 (offset)
= 0b0000 0001 0000 0001 0000 0000 0000 0010 = 0x01010002
```

Next, we have **add 1 2 1**. **add** is an R-type instruction with an opcode of 0 (0b000). Thus, we encode **add 1 2 1** as follows:

```
0b0000000 (unused) 000 (opcode) 001 (regA) 010 (regB)
                   0000000000000 (unused) 001 (dest)
= 0b0000 0000 0000 1010 0000 0000 0000 0001 = 0x000A0001
```

Next is **sw 1 1 10**. **sw** is an I-type instruction with an opcode of 3 (0b011). We can encode it similar to our previous I-type instructions:

```
0b0000000 (unused) 011 (opcode) 001 (regA) 001 (regA)
                   0000 0000 0000 1010 (offset)
= 0b0000 0000 1100 1001 0000 0000 0000 1010 = 0x00C9000A
```

Next, we have **beq 0 0 -4**. We first have to conver the negative offset into 16-bit two's complement. To do this, we start with the 16-bit representation of positive 4 (0b0000 0000 0000 0100), invert the bits (0b1111 1111 1111 1011), and then add 1, to get 0b1111 1111 1111 1100. This becomes our offset field:

```
0b0000000 (unused) 100 (opcode) 000 (regA) 000 (regB)
                   1111 1111 1111 1100 (offset)
= 0b0000 0001 0000 0000 1111 1111 1111 1100 = 0x0100FFFC
```

Finally, we have **noop** and **halt**. These two are O-type instructions, with opcodes of 7 (0b111) for **noop** and 6 (0b110) for **halt**. For O-type instructions, the only field that needs to be filled in is the opcode, since we have no operands. Thus, we encode **noop** as:

```
0b0000000 (unused) 111 (opcode) 00 0000 0000 0000 0000 0000
(unused)
= 0b0000 0001 1100 0000 0000 0000 0000 0000 = 0x01C00000
```

Similarly, **halt** is encoded as:

```
0b0000000 (unused) 110 (opcode) 00 0000 0000 0000 0000 0000
(unused)
= 0b0000 0001 1000 0000 0000 0000 0000 0000 = 0x01800000
```

Finally, we have to handle the **.fill** directives. Recall that **.fill** allows us to place an arbitrary value into the machine code, and consequently memory. Thus, **.fill 6** will place the value 6 into our machine code. We will use 32-bit two's complement for these values.

```
0b0000 0000 0000 0000 0000 0000 0000 0110
= 0x00000006
```

Next, we have **.fill -1**. We will encode -1 into 32-bit two's complement giving us:

```
+1 =         0b0000 0000 0000 0000 0000 0000 0000 0001
flip bits:   0b1111 1111 1111 1111 1111 1111 1111 1110
add 1:       0b1111 1111 1111 1111 1111 1111 1111 1111
           = 0xFFFFFFFF
```

Finally, we have **.fill 0**, which is encoded into 32-bit two's complement:

```
0b0000 0000 0000 0000 0000 0000 0000 0000
= 0x00000000
```

Thus, our final machine code is:

```
0x00810008
0x008A0003
0x01010002
0x000A0001
0x00C9000A
0x0100FFFC
0x01C00000
0x01800000
0x00000006
0xFFFFFFFF
0x00000000
```

This is our assembled program! We've converted our LC2K assembly to LC2K machine code, handling symbolic labels and **.fill** directives.

C to LC2K

Let's consider converting C code to LC2K. In C, we can declare variables, and then perform operations on them. In LC2K, we perform our operations primarily in registers. When we compile C code into any assembly language, variables in C will correspond to registers in assembly. Thus, if we consider a C expression such as

```
int x = 5;
```

the corresponding LC2K code could be

```
            lw          0     1     five
five        .fill       5
```

We can assign x to an arbitrary register; here we choose to assign it to register 1. Since we can't assign registers arbitrary values directly, we have to do so by loading values from memory. Now, let's consider assigning variable values to other variables. For example, we can consider the C code

```
int x = 5; // register 1
int y = x; // register 2
```

We'll assume that the code to set x to 5, which we showed above, has already run. We don't have an explicit way to copy the value of a register into another register, but we can use the

property that a + 0 = a. You might be wondering, though, how do we get 0? We can load 0 from memory each time, but that quickly becomes inefficient. In LC2K, we will use a convention that register 0 will always hold the value 0. This is not enforced by the processor, but rather by the programmer. Thus, we will always have quick access to the value 0. Now, we can represent `y = x` with the following LC2K statement:

```
            add         0     1     2
```

Now that we know how to set up variables, let's work with their values! We'll start with the simplest operation: addition.

```
int x = ...; // register 1
int y = ...; // register 2
int z = x + y; // register 3
```

Here, we want to add the values of register 1 and register 2 together, and store the result into register 3. We can use the **add** opcode to do this:

```
            add         1     2     3
```

Now, let's consider subtraction:

```
int x = ...; // register 1
int y = ...; // register 2
int z = x - y; // register 3
```

LC2K doesn't provide a subtraction opcode, so we'll have to use another method. Remember that we can flip the sign of a two's complement number by inverting the bits and adding 1. Since we can still add two's complement using the same addition logic as normal binary numbers, we can compute −y, and then compute `x + −y`. To invert the bits of register 2 (which corresponds to y), we can use the property that **nor** is functionally complete. Thus, we can take `y NOR y` to find ~y. From here, we can add the constant value 1, and then add x. The LC2K code is thus

```
            nor         2     2     3           calculate ~y
            lw          0     4     one
            add         3     4     3           add 1 to ~y to get −y
            add         1     3     3           add −y to x, store in r3
one         .fill       1
```

Here, we've also added comments to the LC2K code. In LC2K assembly, we can place comments after the operands of an instruction. Comments can contain any character, and are

ignored during assembling. We'll often write comments to guide anyone reading our code through the operations that are taking place, and to make it easier to see the "big picture" of the code.

Next, let's consider bitwise operations. We already know how to do **nor**, which uses the **nor** opcode. From here, we are able to perform all desired bitwise operations. This especially comes in useful when we want to calculate a number modulo a power of two. Recall that the modulo operator (% in C) calculates the remainder after dividing the two arguments. For example, `5 % 3 = 2`, since when we divide 5 by 3, we get 1 with a remainder of 2. Let's consider a simple example:

```
int x = ...; // register 1
int y = x % 4; // register 2
```

To calculate `x % 4` in LC2K, we could implement a loop that continues subtracting until the number becomes negative. However, we can implement modulus by a power of 2 much more simply, by taking advantage of some properties of binary numbers. First, let's consider a specific scenario: x = 15, or 0b1111. We can write out the decimal representation of x:

$$1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$$

Next, let's take `x % 4`. We can take the modulus of each term, and then add them up. Thus, we get

$$(1 * 2^3 \% 4) + (1 * 2^2 \% 4) + (1 * 2^1 \% 4) + (1 * 2^0 \% 4)$$

Now, let's consider which terms we know must be 0. Remember that if **a** is divisible by **b**, then **a** % **b** = 0. Thus, we know that the terms containing 2^3 and 2^2 must evaluate to 0, since they are divisible by $4 = 2^2$. However, the terms containing 2^1 and 2^0 must evaluate to a nonzero value. Thus, we can simplify the previous expression to

$$0 + 0 + (1 * 2^1) + (1 * 2^0)$$

since the value of the last two terms cannot go above 4. Notice that we've now removed all the bits except for the 2^1 and 2^0 bits. Thus, we can calculate `x % 4` by bit masking the last two bits! To perform this, we will use the bit mask 0b11, or 3. Thus, to compute `x % 4`, we can use the following LC2K code:

```
            nor         1     1     3           calculate ~x
            lw          0     2     mask
            nor         2     2     2           calculate ~3
            nor         2     3     2           ~x NOR ~3 = x & 3 = x % 4
mask        .fill       3
```

Conditional Statements

We now know how to do some basic operations in LC2K. However, a program that only has a single path of execution is pretty boring, and can't do too much. Let's start off by writing conditional statements (if/else statements). For example, consider the following C code:

```
int x = ...; // register 1
int y = ...; // register 2
if (x != 0) {
    y = x;
}
```

To implement conditionals in LC2K, we need to use the **beq** instruction. Recall that **beq** stands for "branch if equal", and compares the values in the two provided registers. If the values are equal, **beq** transfers control to a different location in the program, which we can specify using either an offset or a symbolic label. Let's consider the following very simple program:

```
            beq         0     1     label       resolves to offset = 1
            noop
label       noop
```

If we consider the control flow of this program, the first **noop** is conditionally executed, while the second **noop** is always executed regardless of the **beq** being taken or not taken. When will the first **noop** be executed? If the value in register 0 equals the value in register 1, then the first **noop** will be **skipped**. If the value in register 0 is not equal to the value in register 1, then the first **noop** will be **executed**. Thus, we can consider the equivalent C code as follows:

```
if (r0 != r1) {
    // first noop
}
// second noop
```

We've just written a very simple **if** statement in LC2K assembly! Let's use this pattern to implement the code from previously:

```
int x = ...; // register 1
int y = ...; // register 2
if (x != 0) {
    y = x;
}
```

Remember that by convention, register 0 is 0. Thus, we need to compare the values of register 1 (x) and register 0 (0). If the two values are equal, we want to skip over the statement `y = x`. In general, the branch condition in assembly will be reversed from the branch condition in C. We'll see this more when we cover C to ARMv8. The corresponding LC2K code for the C code above is therefore:

```
            beq         0     1     skip
            add         0     1     2
skip        halt
```

In general, we can replace the **add 0 1 2** with any statement or statements that should be executed conditionally.

Next, let's consider an **if-else** statement. We'll start by considering a case where the **if** clause is taken when two values are not equal:

```
int x = ...; // register 1
int y = ...; // register 2
int z = ...; // register 3
if (x != 0) {
    y = x;
} else {
    z = x;
}
```

We'll set up the **if** clause similarly to before:

```
            beq         0     1     else
            add         0     1     2
```

However, if we put the line **else add 0 1 3** after this, then we would always run **add 0 1 3** (which corresponds to `z = x`). However, we don't want to always run `z = x`! We only want to run `z = x` if x is equal to 0. Thus, we will add an **unconditional branch**. In LC2K, we can set up an unconditional branch by giving **beq** the same registers to compare. For example, we can use

```
            beq         0     0     end
```

to unconditionally transfer control to the code at the symbolic label "end". If we consider the code so far, we know that if register 1 is not zero, then **add 0 1 2** will execute, and then we will branch to a label "end". Now, let's put in our **else** clause:

```
else        add         0     1     3
end         halt
```

Observe that we branch to the "else" label if `x == 0`, and then continue on to the "end" label implicitly. If `x != 0`, then the first **beq** is not taken, and the unconditional branch will skip the **else** clause and continue to the "end" label. Thus, we know that if `x != 0`, then **add 0 1 2** will run, while if `x == 0`, then **add 0 1 3** will run. This is the exact behavior that we want from the **if-else** statement.

Notice that so far, we've only considered the conditional statement being `!=`. Let's consider a case where we want to check for equality, such as:

```
int x = ...; // register 1
int y = ...; // register 2
if (x == 0) {
    y = x;
}
```

Recall that we need to invert the condition: we will skip the statement `y = x` when `x != 0`. Thus, we would have to branch if two values are **not equal**. However, we don't have an operation for that in LC2K. Let's think about how we could rewrite this C code to use an inequality check:

```
int x = ...; // register 1
int y = ...; // register 2
if (x != 0) {
    // do nothing
} else {
    y = x;
}
```

We know how to write an if-else statement with an inequality condition! Let's adapt our previous LC2K assembly for the if-else statement:

```
            beq         0     1     else
            beq         0     0     end
else        add         0     1     2           y = x
end         halt
```

We can check that this works by considering the cases where x is 0, and where x is not 0. If x is 0, then the first **beq** will be taken, and thus we set y = x. If x is not 0, then the first **beq** will not be taken. The second **beq**, which is an unconditional branch, will run, skipping the **add** instruction. Therefore, y = x will not run.

Loops

Finally, let's consider loops. In C, we can consider two types of loops: **while** loops and **for** loops. Let's start with a **while** loop:

```
int x = 3; // register 1
while (x != 0) {
    x -= 1;
}
```

Now that we're creating more complicated code patterns, it is good practice to create **register assignments**. For example, we can assign registers for the above code as follows:

register 0 = constant value 0
register 1 = value of x
register 2 = constant value -1

Now, we have a consistent way to reference values we'll use in our program. Let's start by setting up the constant values:

```
            lw          0     1     three
            lw          0     2     negOne
```

For now, we won't put in **five .fill 5** or **negOne .fill -1**. Why do we not put these here? Remember that since we are working in a Von Neumann architecture, there is no difference between instructions and memory. Thus, the **.fill** directive will place a value into memory that is treated as an instruction! For example, let's consider the following code:

```
            lw          0     1     five
five        .fill       5
            halt
```

When this is assembled, it produces the machine code

```
0x810001
0x3
0x1800000
```

We'll see how the processor executes instructions later, but the key observation is that the value 0x5 is not treated as data, but rather as the instruction **add 0 0 5**! Thus, we will add the value of register 0 to itself, giving us 0, which then puts the value 0 into register 5! To avoid accidentally performing undesired operations, we typically will split up our code into a **text** and a **data** section. We'll see this again when we cover linking. The text section contains instructions such as **lw** and **beq**. The data section contains only **.fill** directives, and will come after the entire text section. If you take a look through the previous examples, you'll notice that we've placed the **.fill** directives after all instructions to follow this convention.

Returning to the **while** loop, we need to consider how we can set up a loop structure in assembly. Let's think about how a **while** loop is executed in C. First, we reach the condition of the while loop, and then we check the condition. If the condition is true, we execute the loop body. If the condition is false, we skip to after the loop body. After executing the loop body, we return back to the condition of the loop, and check the condition again. Let's think about how we can do this in assembly:

```
loop        beq         ?     ?     end
            (loop body)
            beq         0     0     loop
end         ...
```

The first **beq** acts as our condition check: it checks if we should break out of the loop. We may have to have multiple statements in some cases for the condition check, especially if we need to compute more complex conditions such as `(x % 2) != 0`. Here, we branch to the "end" label when the **beq** is taken, meaning that the condition check should be the opposite of the loop condition. Intuitively, we need the opposite checks for "stay in loop" and "end

loop", so the condition is reversed when writing assembly[7]. Thus, we can write the C **while** loop from before with the statements

```
            lw          0     1     three
            lw          0     2     negOne
loop        beq         0     1     end
            add         1     2     1
            beq         0     0     loop
end         halt
three       .fill       3
negOne      .fill       -1
```

Finally, let's consider a **for** loop. For example, let's convert the following C code into LC2K:

```
int i = 0; // register 1
int x = 0; // register 2
for (i = 0; i != 3; ++i) {
    x += i;
}
```

Remember that **for** loops can be written equivalently as **while** loops. Thus, we can write this loop in **while** loop form as:

```
int i = 0; // register 1
int x = 0; // register 2
i = 0;
while (i != 3) {
    x += i;
    ++i;
}
```

[7] A second assembly pattern for **while** loops is the following:

```
start       beq         0     0     cond
loop        (loop body)
cond        beq         x     y     loop
end         ...
```

We don't use this pattern in 370 as much, but it is often used by compilers. The advantage here is that the condition for the second **beq** matches the condition to stay in the loop, allowing us to implement a loop such as `while (x == 0) { ... }` more concisely.

Again, let's create a register assignment:

register 0 = constant value 0
register 1 = value of i
register 2 = value of x
register 3 = constant value 3
register 4 = constant value 1

Let's break up this code into four segments: loading constant values, initialization of i, the loop condition and body, and the update. First, we'll start with loading constant values. We'll perform the same process as before:

```
            lw          0     3     three
            lw          0     4     one
```

We'll have a corresponding data section of:

```
three       .fill       3
one         .fill       1
```

Next, let's consider the loop condition and body. We want to stay in the loop if `i != 3`, therefore our exit condition should be `i == 3`. We can compare registers 1 (i) and 3 (3) to check if we should exit the loop. Inside the loop, we add the value of i to x, and store the result back into x.

```
loop        beq         1     3     end
            add         1     2     2
```

Next, we have the update statement: `++i`. We can add the values of register 1 (i) and 4 (1) together, and store the result back into i.

```
            add         1     4     1
```

Finally, we need to complete the loop by branching back to the "loop" label. Thus, our final LC2K assembly is:

```
            lw          0     3     three       r3 = 3
            lw          0     4     one         r4 = 1
loop        beq         1     3     end         if i == 3, end loop
            add         1     2     2           x += i
            add         1     4     1           ++i
            beq         0     0     loop        continue loop
end         halt
```

With these patterns for variables, calculations involving variables, and control flow statements, you can now convert arbitrary C code into LC2K!

The ARMv8 ISA

LC2K is a great ISA when learning assembly, but it is tedious to write code in. For example, let's consider turning the following C code into LC2K:

```
int x = ...; // register 1
if (x < 5) {
    x += 1;
}
```

At first glance, this seems like a simple **if** statement: we'll invert the condition and then skip the increment if x is greater than or equal to 5. However, we don't have any conditional branches to check either < or >=. We'll have to use a property of two's complement here: a number is negative if it's most significant bit is 1. This doesn't seem very helpful: we're comparing x to 5, why do we care if it is negative? Let's take a look at the condition: x < 5. We can rewrite this to an equivalent condition: x – 5 < 0. Now, we just have to check if x – 5 is negative. We'll start by calculating x – 5:

```
            lw          0     2     neg5
            add         1     2     2           r2 = x – 5

(data section)
neg5        .fill       -5
```

Next, we'll check the MSB, which is in the 2^{31} place. Thus, we will create a bit mask to extract the 31st bit of r2:

```
            lw          0     3     mask
            nor         3     3     3
            nor         2     2     2
            nor         2     3     2          r2 & mask

data section:
mask        .fill       2147483648           2^31
```

Finally, we can check if the masked value is nonzero. If the masked value is nonzero, then the value of x – 5 was negative, and therefore we want to enter the **if** statement. If the masked value is zero, then the value of x – 5 was nonnegative, and therefore we want to skip the **if** statement.

```
            beq         0     2     skip
            lw          0     2     one
            add         1     2     1
skip        halt

data section:
one         .fill       1
```

This seems very tedious and a lot of unnecessary work to check if a number is less than another. Let's consider the same C code compiled into ARMv8:

```
            CMPI        X1,   #5
            B.GE        skip
            ADDI        X1,   X1,   #1
skip:
```

We'll give a quick overview of what the code above does. First, we run **CMPI X1, #5**. **CMPI** stands for "compare immediate", and compares the value in register 1 (X1) to the constant value 5. In ARMv8, all immediate values are marked with a "#" symbol, while registers are marked with an "X" in front of the register number. The next instruction is **B.GE**, which stands for "branch if greater than or equal to". This is a conditional branch, similar to **beq** in LC2K. You'll notice that in contrast to **beq**, which takes two registers and a target, we only provide a target to **B.GE**. This is because **B.GE** will use the result of the **CMPI** instruction to decide whether a branch is taken or not. We'll go more into this when we discuss the program state register (PSR). Finally, we have **ADDI X1, X1, #1**, which is an "add immediate" instruction. This allows us to add a constant value to a register, and corresponds to ++x in C.

ARMv8 is a 64-bit, byte addressable system. ARMv8 provides 31 general-purpose 64-bit registers (X0 through X30), and one special register, named XZR. XZR stands for "zero register", and will always hold a constant value of 0. We can never overwrite the value of XZR; if an instruction attempts to write a value to XZR, the value is discarded. ARMv8 also provides three other aliases: SP for X28, FP for X29, and LR for X30. We'll see these being used later when we discuss function calls.

In 370, we use a subset of ARMv8 called LEGv8. LEGv8 provides a "bare-bones" version of ARMv8 that is relatively straightforward to remember, but includes a wide range of functionality. Let's go through the different types of operations provided.

LEGv8 Arithmetic Operations

First, we will consider arithmetic operations. In LEGv8, there are eight such operations: **ADD**, **ADDI**, **ADDS**, **ADDIS**, **SUB**, **SUBI**, **SUBS**, and **SUBIS**. Let's start with **ADD**, **ADDI**, **ADDS**, and **ADDIS**. All four of these operations perform addition, and have the following semantics:

```
ADD         Xd,   Xn,   Xm          ; Xd = Xn + Xm
ADDS        Xd,   Xn,   Xm          ; Xd = Xn + Xm, set flags

ADDI        Xd,   Xn,   #uimm12     ; Xd = Xn + #uimm12
ADDIS       Xd,   Xn,   #uimm12     ; Xd = Xn + #uimm12, set flags
```

Let's start with **ADD**. The **ADD** opcode takes in three operands: a destination register (Xd) and two source registers (Xn and Xm). The values of Xn and Xm are added, and the result stored into Xd. Note that while LC2K's **add** opcode uses the ordering **regA regB destReg**, ARMv8 uses the ordering **Xd, Xn, Xm**, with the destination register coming first.

Next, we have **ADDI**. **ADDI** again takes three operands: a destination register (Xd), a source register (Xn), and an immediate value (#uimm12). The value of Xn is added to the immediate value, and the result stored into Xd. Again, the destination register is the first operand, followed by the source register, and then the immediate.

To specify immediate values, we will first use "#" to indicate an immediate, followed either "uimm" or "simm" to specify an unsigned or signed immediate value, and then the size in bits. Thus, the immediate value for ADDI, which is listed as #uimm12, is an unsigned 12-bit immediate value. When we specify an immediate as unsigned, this means that we do not treat it as a two's complement number. For example, let's consider the immediate value 0b1000 0000 0000. If we treat it as a #uimm12, or an unsigned 12-bit immediate value, then we will treat it as the value 2^{11}. However, if we treat it as a #simm12, or a signed 12-bit immediate value, then we will treat it as the value -2^{11}.
Next, we have **ADDS** and **ADDIS**. These instructions have the exact same semantics and effects as **ADD** and **ADDI**, but additionally set **flags** in the CPU. In ARMv8, we will use four

flags: N, V, Z, and C. The values of these flags are stored in a special register called the **program status register**, or **PSR** for short. The flags each represent a property about a computation:

N: **n**egative. The N flag is set if the MSB of the result is 1, meaning that the result is negative when represented in two's complement.

V: o**v**erflow. The V flag is set to the carry from the MSB XOR the carry from the second most significant bit. For example, with 64-bit registers, the MSB is the 2^{63} place, while the second most significant bit is the 2^{62} place. If the V flag is set, then this indicates that an overflow occurred when performing an operation on two two's complement numbers.
Z: **z**ero. The Z flag is set if the result is equal to 0.

C: **c**arry. The C flag is set if the carry of the MSB is 1.

While these do not seem very useful at first, we will see how we can use them to compare register values when we discuss conditional branches.

The "S" in **ADDS** and **ADDIS**, indicates that the N, V, Z, and C flags should be set based on the addition taking place. We'll see this again in **SUBS** and **SUBIS**.

Additionally, ARMv8 provides subtraction opcodes: **SUB**, **SUBS**, **SUBI**, and **SUBIS**. Their semantics are as follows:

```
SUB         Xd,   Xn,   Xm          ; Xd = Xn - Xm
SUBS        Xd,   Xn,   Xm          ; Xd = Xn - Xm, set flags

SUBI        Xd,   Xn,   #uimm12     ; Xd = Xn - #uimm12
SUBIS       Xd,   Xn,   #uimm12     ; Xd = Xn - #uimm12, set flags
```

Similar to **ADD**, we have a destination register (Xd) and two source registers (Xn and Xm) for **SUB**. We compute the difference Xn – Xm, and store the result into Xd. **SUBS** performs the same computation, but sets the N, V, Z, and C flags.

SUBI performs a subtraction with an immediate value. You may have been wondering: why doesn't **ADDI** allow addition with a signed immediate? The reason is because we have **SUBI**. By using an unsigned immediate, we allow addition and subtraction with a constant value in the range of 0 to 2^{12} - 1 = 4095, inclusive. However, if we used a signed immediate value, then the range of potential immediate values would become -2^{11} = –2048 to 2^{11} - 1 = 2047, inclusive. **SUBI** would also become obsolete, and we would only have half the range of immediate values that could be utilized.

Finally, we have **SUBIS**, which performs the same operation as **SUBI**, but sets the N, V, Z, and C flags.

LEGv8 Logical Operations

LEGv8 provides three bitwise operators (bitwise AND, OR, and XOR), and logical left and right shifts. The opcodes are **AND**/**ANDI**, **ORR**/**ORRI**, **EOR**/**EORI**, **LSL**, and **LSR**. The semantics of these instructions are as follows:

```
AND         Xd,   Xn,   Xm          ; Xd = Xn & Xm
ANDI        Xd,   Xn,   #uimm12     ; Xd = Xn & #uimm12
ORR         Xd,   Xn,   Xm          ; Xd = Xn | Xm
ORRI        Xd,   Xn,   #uimm12     ; Xd = Xn | #uimm12
EOR         Xd,   Xn,   Xm          ; Xd = Xn ^ Xm
EOR         Xd,   Xn,   #uimm12     ; Xd = Xn ^ #uimm12

LSL         Xd,   Xn,   #uimm6      ; Xd = Xn << #uimm6
LSR         Xd,   Xn,   #uimm6      ; Xd = Xn >> #uimm6
```

LEGv8 Pseudoinstructions

LEGv8 provides three pseudoinstructions: **CMP**, **MOV**, and **MOVI**. The semantics are as follows:

```
CMP         Xn,   Xm                ; SUBS XZR, Xn , Xm
MOV         Xd,   Xn                ; ORR  Xd , XZR, Xn
MOVI        Xd,   #uimm12           ; ORRI Xd , XZR, #uimm12
```

These are called **pseudoinstructions** because they do not have a unique machine code translation. Rather, they are substituted with instructions such as **SUBS** or **ORR**, as shown above. Why do we use pseudoinstructions rather than directly performing the substitution? We use pseudoinstructions **semantically**, to indicate that we are performing a certain special case. For example, we use the **CMP** pseudoinstruction to indicate that we are comparing two values. While **SUBS XZR, X2, X1** would accomplish the exact same result as **CMP X2, X1**, it is easier to look at **CMP X2, X1** at a glance and recognize that a comparison is taking place. In contrast, when we see a **SUBS** opcode, we first think that a subtraction is being performed. Similarly, the **MOV** and **MOVI** opcodes **move** a value from one register (or an immediate value) to another.

LEGv8 Data Transfer Instructions

Recall that LEGv8 is a byte-addressable system with 64-bit registers. Thus, when we consider loading from memory, we can load up to 8 addresses at once from memory into a register. LEGv8 provides eight operations for loading from and storing to memory, allowing transfers of 1, 2, 4, and 8 bytes. The opcodes are **LDURB**, **LDURH**, **LDURSW**, **LDUR**, **STURB**, **STURH**, **STURW**, and **STUR**. Their semantics are as follows:

```
LDURB      Xt,   [Xn,  #simm9]    ; load 1B from M[Xn + #simm9]
                                  ; into the lower 1B of Xt
LDURH      Xt,   [Xn,  #simm9]    ; load 2B from M[Xn + #simm9]
                                  ; into the lower 2B of Xt
LDURSW     Xt,   [Xn,  #simm9]    ; load 4B from M[Xn + #simm9]
                                  ; into the lower 4B of Xt and
                                  ; perform sign extension
LDUR       Xt,   [Xn,  #simm9]    ; load 8B from M[Xn + #simm9]
                                  ; into Xt

STURB      Xt,   [Xn,  #simm9]    ; store the lower 1B of Xt into
                                  ; M[Xn + #simm9]
STURH      Xt,   [Xn,  #simm9]    ; store the lower 2B of Xt into
                                  ; M[Xn + #simm9]
STURW      Xt,   [Xn,  #simm9]    ; store the lower 4B of Xt into
                                  ; M[Xn + #simm9]
STUR       Xt,   [Xn,  #simm9]    ; store the value of Xt into
                                  ; M[Xn + #simm9]
```

What do we mean here by the **lower** 1/2/4/8 bytes of a register? We will often refer to the lower or upper X bytes of a register. These simply refer to one "end" of the register value. For example, let's consider a register containing the value 0x0123456789ABCDEF:

```
0b00000001 00100011 01000101 01100111 10001001 10101011 11001101 11101111
  upper 1B                                                        lower 1B
  <-- upper  2B -->                                     <-- lower  2B -->
  <----------- upper  4B -----------> <----------- lower  4B ----------->
```

Thus, when we perform **LDURB**, we will load a single byte from memory into the lower 1B of the target (Xt) register. Notice that all of the load and store instructions use the **base + offset** addressing method: we take a variable base (Xn) and add a constant offset (#simm9). Now, let's consider **LDURH** and **LDUR**. **LDURH** stands for "load unsigned **h**alf word", and loads 2 bytes into a register. In ARMv8, one word is 4 bytes, so a half word is 2 bytes. When we

perform **LDURH**, we will load 2 bytes from memory into a register. For example, let's consider the instruction **LDURH X1, [X0, #0]**, and assume that the value in register 0 is 0x1000. We can calculate the address to load from by adding the value in register 0 with the offset of 0. We get 0x1000. Since we are loading a half word, we will load the two bytes starting at address 0x1000, thus loading in 0x1000 and 0x1001. Similarly, **LDUR** will load in bytes 0x1000 through 0x1007. Finally, let's discuss **LDURSW**, which loads a **signed word**. If we use the previous example, we will load bytes 0x1000 through 0x1003. Next, let's assume that we've loaded the value 0x80000000 into the lower 4 bytes of register 1. Then, we need to treat this as a signed (two's complement) number. Since the register is 64 bits, we need to perform a **sign extension**. A sign extension involves "extending" the most significant bit to keep the value of a two's complement number the same while increasing the number of bits that it takes up. For example, we can consider sign extending a 4-bit two's complement number to 8-bit two's complement. Let's consider 0b1101. We will take the MSB, which is 1, and "fill in" bits 7-4 with 1. Thus, we get 0b1111 1101. If we consider the value of 0b1101 in 4-bit two's complement, we get -3. If we consider the value of 0b1111 1101 in 8-bit two's complement, we also get -3. Thus, by sign extending, we can increase the size of a number without changing its value. If we consider a positive number, we will sign extend with 0s, meaning that there is no change to the value as well. When we execute **LDURSW**, we load in 4 bytes from memory, and then perform a sign extension to extend the 32-bit two's complement number to the full 64 bits of the register. Thus, loading in 0x80000000 from memory would result in the value 0xFFFFFFFF80000000 being stored into the register.

Similarly, we have the store instructions: **STURB**, **STURH**, **STURW**, and **STUR**. Notice that we don't have a **STURSW** instruction. When we store from a register into memory, we're storing from a 64-bit register into either 8, 16, 32, or 64 bits of memory. Thus, sign extension is never necessary, since the region we're storing to is at most the size of the register. Just like the load instructions, we will store the lower 1, 2, 4, or 8 bytes (the lower 8 bytes being the entire register) into the corresponding region starting at memory address Xn + #simm9.

One small caveat to storing and reading memory is the order in which bytes are read. Let's consider the following contents of memory:

address	contents
0x1000	0x12
0x1001	0x34
0x1002	0x56
0x1003	0x78

If we perform **LDURSW X1, [XZR, #0x1000]**, do we load in the value 0x12345678, or do we load in the value 0x78563412? It may seem like an obvious choice: 0x12345678; however, the truth is that either value may be loaded into X1, depending on a property called *endianness*. We'll discuss endianness in the next chapter.

LEGv8 Register Data Transfer

LEGv8 provides two instructions to manipulate register values directly: **MOVZ** and **MOVK**. The semantics of these instructions are as follows:

```
MOVZ       Xd,   #uimm16,   [LSL N]    ; zero out Xd, and then
                                       ; place #uimm16 into a 16b
                                       ; slot of Xd, based on N
MOVK       Xd,   #uimm16,   [LSL N]    ; place #uimm16 into a 16b
                                       ; slot of Xd, based on N
```

Let's consider an example of both of these. Assume that register X1 starts with the value 0x0123456789ABCDEF. We can break up this value into four 16b slots:

0x0123 4567 89AB CDEF

Let's start with **MOVK**. If we run the instruction **MOVK X1, #0xEEC5, LSL 16**, then we will be modifying the second from last "slot" of the register value. The "K" in **MOVK** stands for "keep", and this means that we will only modify the specified "slot". Thus, the resulting value will be

0x0123 4567 **EEC5** CDEF

If we omit the "LSL N" portion, then the instruction defaults to LSL 0. Thus, if we run the instruction **MOVK X1, #0x0370**, then we will change the last "slot". The resulting value is thus

0x0123 4567 EEC5 **0370**

Now, let's consider **MOVZ**. The "Z" in **MOVZ** stands for "zero", meaning that we will set the value of X1 to 0 before updating the "slot". Let's consider **MOVZ X1, #0x0123 LSL 48**. This instruction will update bits 63-48, and the remaining bits (47-0) will be set to 0. Thus, the final value of X1 will be

0x**0123** 0000 0000 0000

LEGv8 Branches

LEGv8 provides two general classes of branches: unconditional and conditional branches. As we saw in LC2K, an unconditional branch is a branch that is always guaranteed to be taken, while a conditional branch is taken only when a certain condition is satisfied.

Unconditional Branches

LEGv8 provides three unconditional branches (also called **jumps**): **B**, **BR**, and **BL**. Their semantics are as follows:

```
B           #simm26          ; PC = PC + #simm26
BR          Xt               ; PC = value of Xt
BL          #simm26          ; X30 = PC + 4; PC = PC + #simm26
```

Let's start with **B**. **B** is an unconditional branch to a relative location. We can think of **B** as similar to LC2K's **beq 0 0 (target)**, where **target** is not directly the address we want to branch to, but rather a relative location. Using #simm26 allows us to perform both forward and backwards jumps.

Next, we have **BR**, which stands for "branch to register". Here, we update the PC to take on the value in Xt. This allows us to branch to an arbitrary location specified by a register value.

Finally, we have **BL**, which stands for "branch and link". Here, we perform an unconditional branch similar to **B**, but we also store PC + 4, which is the address of the next instruction (remember words are 4 bytes) into X30 (also known as **LR**/**link register**). We'll see why we want to store PC + 4 when we discuss function calls and the stack.

Conditional Branches

LEGv8 provides three general conditional branches: **CBZ**, **CBNZ**, and **B.COND**. **B.COND** is split up into multiple subcases depending on the specific condition code. The semantics are as follows:

```
CBZ         Xt,   #simm19    ; if (Xt == 0) PC = PC + #simm19
CBNZ        Xt,   #simm19    ; if (Xt != 0) PC = PC + #simm19
B.COND      #simm19          ; if (condition) PC + PC + #simm19
```

Let's start with **CBZ** and **CBNZ**. **CBZ** stands for "conditional branch if zero", and branches to PC + #simm19 if the value in Xt is 0. Similarly, **CBNZ**, or "conditional branch if not zero" branches to PC + #simm19 if the value in Xt is not zero.

Next, we have **B.COND**. **B.COND** encompasses the opcodes **B.EQ**, **B.NE**, **B.LT**, **B.LO**, **B.LE**, **B.LS**, **B.GT**, **B.HI**, **B.GE**, **B.HS**, **B.MI**, **B.PL**, **B.VS**, and **B.VC**. Let's go through what each of these opcodes means:

```
B.EQ       ; branch if equal (Z = 1)
B.NE       ; branch if not equal (Z = 0)
B.LT       ; branch if less than (N != V)
B.LO       ; branch if lower than (C = 0)
B.LE       ; branch if less than or equal (Z != 0 OR N != V)
B.LS       ; branch if lower than or same (Z != 0 OR N != V)
B.GT       ; branch if greater than (Z = 0 AND N = V)
B.HI       ; branch if higher (Z = 0 AND C = 1)
B.GE       ; branch if greater than or equal (N = V)
B.HS       ; branch if higher than or same (C = 1)
B.MI       ; branch if minus (N = 1)
B.PL       ; branch if plus (N = 0)
B.VS       ; branch if overflow set (V = 1)
B.VC       ; branch if overflow clear (V = 0)
```

Generally, don't worry about remembering which flags need to be set/not set/equal to another flag for each condition. Rather, remember the condition they are trying to test. Let's consider a simple example of these conditions:

```
            MOVI       X1,   #5
            CMP        XZR,  X1
            B.LT       end
            MOVI       X2,   #1
end:
```

Notice that **B.COND** does not take in any register values, but rather only an offset. **B.COND** uses the preexisting state of flags to determine if the branch is taken. Thus, we need an instruction such as **CMP** before **B.COND** to set the processor flags. For example, let's consider the example above. We start by setting X1 = 5, and then running **CMP XZR, X1**. Recall that **CMP** is a pseudoinstruction, so this is substituted with **SUBS XZR, XZR, X1**. Thus, we perform the subtraction XZR – X1, or 0 – 5. This gives us a result of -5. The results of the processor flags are as follows:

N = 1
V = 0
Z = 0
C = 0

Thus, when we run **B.LT**, we see that `N != V`, and therefore we take the branch and skip the second **MOVI**. In general, if we have a **CMP** followed by a **B.COND**, then the branch will be taken if:

```
     CMPI       X1,   X2
followed by:
     B.EQ       label      ; taken if X1 == X2
     B.NE       label      ; taken if X1 != X2
     B.LT       label      ; taken if X1 < X2
     B.LO       label      ; taken if X1 < X2
     B.LE       label      ; taken if X1 <= X2
     B.LS       label      ; taken if X1 <= X2
     B.GT       label      ; taken if X1 > X2
     B.HI       label      ; taken if X1 > X2
     B.GE       label      ; taken if X1 >= X2
     B.HS       label      ; taken if X1 >= X2
```

Notice that we have two condition codes for `<`, `<=`, `>`, and `>=`. We use **B.LT**, **B.LE**, **B.GT**, and **B.GE** for **signed** numbers, while we use **B.LO**, **B.LS**, **B.HI**, and **B.HS** for **unsigned** numbers. We'll see these in our C to ARMv8 chapter.

This concludes the LEGv8 ISA specification! There are quite a few instructions to remember, so don't worry about committing them to heart right away. The best way to start remembering these instructions is to see them in practice.

Chapter 5: Coding in Assembly

Before we can convert C code into assembly, we need to understand some of the lower-level concepts of C code. First, we will discuss endianness and struct alignment. From here, we will have the tools needed to convert any C program into LEGv8 assembly.

Endianness

When we discussed the load and store instructions in LEGv8, we alluded to deciding which order bytes are loaded into memory. For example, let's consider the following example.

address	value
0x1000	0x81
0x1001	0x2F
0x1002	0x35
0x1003	0x1D
0x1004	0xEC
0x1005	0x50
0x1006	0x00
0x1007	0x1F

Assume that register X0 is initialized to 0x1000. Let's consider the instruction **LDUR X1, [X0, #0]**. We load 8 bytes from addresses 0x1000 through 0x1007. There are two orderings we could read these bytes in: 0x81 2F 35 1D EC 50 00 1F or 0x1F 00 50 EC 1D 35 2F 81. These two orderings are called **big endian** and **little endian** respectively. In big endian, the most significant byte corresponds to the lowest address, and the least significant byte corresponds to the highest address. In little endian, the most significant byte corresponds to the highest address, and the least significant byte corresponds to the lowest address. Another way to remember which ordering is which is to remember that big endian is "reading order". If we look at the memory contents above, reading them top to bottom gives us the big endian result.

Let's consider a more complicated example. Let's start with the same memory contents as shown above, and we run the following instructions:

```
MOVZ      X0,  #0x1000
LDURSW    X1,  [X0, #0]
STURH     X1,  [X0, #4]
LDURB     X2,  [X0, #3]
STURW     X2,  [X0, #0]
```

Assume that all registers start as 0. After the **MOVZ**, X0 = 0x1000.

First, let's consider a big endian system. The **LDURSW** loads bytes 0x1000 through 0x1003. Since the system is big endian, 0x1000 becomes the most significant byte, while 0x1003 becomes the least significant byte. Thus, we will read in the number 0x812F351D. Next, we sign extend this to find a final value in X1 of 0xFFFFFFFF812F351D.

Next, we run **STURH X1, [X0, #4]**. We store 2B, in addresses 0x1004 through 0x1005. We take the lower 2B of X1, which are 0x351D. The most significant byte, which is 0x35, is stored into memory address 0x1004, while the least significant byte, which is 0x1D, is stored into memory address 0x1005. Thus, we get the following contents of memory:

address	value
0x1000	0x81
0x1001	0x2F
0x1002	0x35
0x1003	0x1D
0x1004	**0x35**
0x1005	**0x1D**
0x1006	0x00
0x1007	0x1F

After this, we run **LDURB X2, [X0, #3]**. We load a single byte from 0x1003, and thus we get the value 0x1D in register X2. Finally, we run **STURW X2, [X0, #0]**. We store the lower 4 bytes of X2, which are 0x0000001D, into memory addresses 0x1000 through 0x1003. Since the system is big endian, the most significant byte, which is 0x00, ends up in address 0x1000, while the least significant byte, which is 0x1D, ends up in address 0x1003. The intermediate bytes are placed in order. Thus, the final contents of memory are as follows:

address	value
0x1000	**0x00**
0x1001	**0x00**
0x1002	**0x00**
0x1003	**0x1D**
0x1004	0x35
0x1005	0x1D
0x1006	0x00
0x1007	0x1F

Now, let's consider the same example in a little endian system. We run **LDURSW X1, [X0, #0]**, which loads from addresses 0x1000 through 0x1003. Address 0x1000 becomes the **least significant** byte, while address 0x1003 becomes the **most significant byte**. Thus, we load the value 0x1D352F81, which is sign extended with 0s. The value of X1 after this load is thus 0x1D352F81. Next, we run **STURH X1, [X0, #4]**. We store to addresses 0x1004 through 0x1005, placing the most significant byte (0x2F) in **0x1005**, while the least significant byte (0x81) is placed into **0x1004**. Thus, the new state of memory is as follows:

address	value
0x1000	0x81
0x1001	0x2F
0x1002	0x35
0x1003	0x1D
0x1004	**0x81**
0x1005	**0x2F**
0x1006	0x00
0x1007	0x1F

Next, we run **LDURB X2, [X0, #3]**. Since we only load a single byte, the result is the same: 0x1D. Finally, **STURW X2, [X0, #0]** runs, storing 0x0000001D into addresses 0x1000 through 0x1003. The most significant byte is placed in address 0x1003, while the least significant byte is placed in address 0x1000. Thus, the final contents of memory are as follows:

address	value
0x1000	**0x1D**
0x1001	**0x00**
0x1002	**0x00**
0x1003	**0x00**
0x1004	0x81
0x1005	0x2F
0x1006	0x00
0x1007	0x1F

Struct Alignment

We've treated variables just as registers in the processor so far. However, the 64-bit limitation makes it difficult, or even impossible, to store larger data types. For example, let's consider the following C struct that keeps track of a player character:

```
struct {
    char name[9] = "Trippie";
    struct {
        int height[2] = {3, 6}; // 3'6"
        float weight = 92.0; // lb
        short alive = 1;
    } vitals;
    char color[10] = "red";
    int isImpostor = 1; // o.o
} trippie;
```

We definitely can't fit all of this into 64 bits; just the array of characters would take 16 bytes, or double the size of a register! Since we don't have enough space in our registers, we are forced to store the struct in memory. In general, we'll have a tradeoff between capacity and speed. Registers have a very low latency, but there is only a small amount of space available in registers. Memory has a higher latency, but we can store much more data in memory than we can in registers.

Now that we have enough memory to store all the elements of our struct, let's first look at how large each **primitive data type** needs to be. Primitive data types are the most fundamental data types; they are not composed of multiple smaller data types, but rather are types such as **int** and **float** which represent a single value. Primitive data types can be grouped into the following sizes:

1 B: **char**, **int8_t**, **uint8_t**
2 B: **short**, **int16_t**, **uint16_t**
4 B: **int**, **float**, **long** (32-bit system), pointers (32-bit system), **int32_t**, **uint32_t**
8 B: **double**, **long** (64-bit system), pointers (64-bit system), **int64_t**, **uint64_t**

Now, let's assign memory to our struct elements! Let's say that the struct starts at address 0. Then, we could make an assignment as such:

```
char name[9]:   addresses 0 - 8
int height[2]:  addresses 9 - 12, 13 - 16
float weight:   addresses 17 - 20
short alive:    addresses 21 - 22
char color[10]: addresses 23 - 32
int isImpostor: addresses 33 - 36
```

The total size of the struct would be 37 bytes. However, we run into a problem. Due to modern CPU design, accessing memory is the quickest when the starting address of a block we want to access is a multiple of the size of the block. For example, reading 8 B starting at address 0x0 would be quicker than reading 8 B starting at address 0x1, even though both are 8 bytes. Thus, let's consider an integer, for example **int isImpostor** from the above example. While it is most memory efficient to store **isImpostor** at addresses 33 through 36, it would be more time efficient to store it at, for example, addresses 36 through 39. To account for this, we will devise a set of rules for **struct alignment**. The rules are as follows:

1. All primitive types are aligned to their respective sizes.
2. Structs must align to the size of their largest primitive type.
3. Structs must have a total size which is a multiple of their largest primitive type.

To align struct members, the compiler adds **padding** between elements. For example, let's consider the previous example. **char name[9]** will still take up addresses 0 through 8.

However, the next element, **struct vitals**, must start at an address that is a multiple of 4, since its largest primitive type is 4 B (**int** and **float**). Note that arrays align based on the size of a single element, not of the entire array. The next multiple of 4 is 12. Thus, we will add 3 bytes of padding (addresses 9 through 11) after **char name[9]**, allowing **struct vitals** and consequently **int height[2]** to start at address 12. The array will continue until address 19. Next, we have **float weight**, which is 4 B. Since the next available address (20) is already a multiple of 4, we do not need padding. Thus, **float weight** takes up addresses 20 through 23. Finally, **short alive** will take up addresses 24 through 25, since 24 is a multiple of 2. At the end of the struct, we need to add additional padding. This is to handle arrays of structs, where consecutive structs must be able to align to the correct size. Thus, if we consider the **vitals** struct, its largest primitive type is 4 bytes. Thus, the overall **vitals** struct must have a size that is a multiple of 4 bytes. Thus, we add 2 bytes of padding (26 – 27) which allows the total size of the struct to be 16 bytes, which is a multiple of 4. Thus, **char color[10]** must start at address 28, and extends to address 37. Finally, we have **int isImpostor**, which must align to 4 B. Thus, we need to find the next address that is a multiple of 4, which is 40. Thus, we place padding at addresses 38 through 39. We place **isImpostor** at address 40, and this extends to address 43. Finally, we check the overall struct alignment. We see that the total struct takes up 44 bytes, and its largest element is 4 bytes. Thus, we do not need any additional padding at the end of the struct.

char name[9]: 0 through 8
padding: 9 through 11
int height[2]: 12 through 19
float weight: 20 through 23
short alive: 24 through 25
end of struct padding: 26 through 27
char color[10]: 28 through 37
padding: 38 through 39
int isImpostor: 40 through 43
Finally, we can look at the structs themselves:

```
struct {
    char name[9] = "Trippie";
    struct {
        int height[2] = {3, 6}; // 3'6"
        float weight = 92.0; // lb
        short alive = 1;
    } vitals;
    char color[10] = "red";
    int isImpostor = 1; // o.o
} trippie;
```

Thus, `struct vitals` takes up addresses 12 through 27 (size 16 bytes), while `struct trippie` takes up addresses 0 through 43 (size 44 bytes). Notice that the addresses taken up by structs include the padding at the end of the struct.

Function Calls

Let's consider a simple program:

```
int addOne(int x) {
    return x + 1;
}

int main() {
    int a = 0;
    while (a != 3) {
        a = addOne(a);
    }
}
```

Remember that in C, execution starts in the **main** function. When we call **addOne**, we transfer control to the **addOne** function, and we set the parameter **x** with the value of **a**. We run the contents of **addOne**, and then we return the value **x + 1**. This return value is sent to **main**, where we resume control immediately after the function call takes place. We assign this result to **a**, and we continue the loop.

Before we consider how to convert this to ARMv8 code, let's consider how we could convert this code to LC2K. Let's start with the simplest approach: **inlining** the function call. Inlining means that instead of actually transferring control to another area of code, we copy over the code from the function **addOne** into **main**. For example, we could compile the LC2K as such. Note that we've assigned registers 1 and 2 for **main**, and 3 and 4 for **addOne**.

```
            lw          0     1     zero        ; a = 0
            lw          0     2     three       ; const 3
loop        beq         1     2     end
            add         0     1     3           ; x = a
            lw          0     4     one         ; BEGIN addOne CODE
            add         3     4     3           ; x + 1
            add         0     3     1           ; a = return value
            beq         0     0     loop        ; END addOne CODE
end         halt
zero        .fill       0
one         .fill       1
three       .fill       3
```

However, we see that this quickly becomes unwieldy, especially if we have large functions. Additionally, this code is very hard to maintain; even with the BEGIN and END comments, if we need to change the behavior of the function, then we would have to go through and find every occurrence of the function. Let's consider a slightly more complex approach, which allows us to separate the function definition from **main**. We've assigned registers 1 and 2 for **main**, and 4 and 5 for **addOne**. Register 3 is used to transfer the return value of **addOne** back to **main**.

```
            lw          0     1     zero        ; a = 0
            lw          0     2     three       ; const 3
loop        beq         1     2     end
            add         0     1     4           ; x = a
            beq         0     0     addOne
ret         add         0     3     1           ; a = addOne(a)
            beq         0     0     loop
end         halt
            noop                                ; DEFINITION OF addOne
addOne      lw          0     5     one         ; const 1
            add         4     5     3           ; return value
            beq         0     0     ret
zero        .fill       0
three       .fill       3
one         .fill       1
```

Now, whenever we want to call **addOne**, we can branch to the **addOne** label. There's one last problem: in this example, we only have one call to the function. Let's consider another example:

```
int addOne(int x) {
    return x + 1;
}

int main() {
    int a = 0;
    addOne(a);
    addOne(a);
}
```

Let's consider the corresponding LC2K code (we've left out **.fill** directives for clarity):

```
            lw          0     1     zero        ; a = 0
            add         0     1     4           ; x = a
            beq         0     0     addOne      ; addOne
ret1        add         0     1     4           ; x = a
            beq         0     0     addOne      ; addOne
ret2        halt
addOne      lw          0     5     one         ; const 1
            add         4     5     3           ; return value
            beq         0     0     ???
```

At the end of **addOne**, where do we branch to? If we branch to **ret1**, the second call takes us into an infinite loop. If we branch to **ret2**, the second call is skipped. To fix this problem, we introduce the concept of a **return address**. The return address contains the PC that we will return to after the function call finishes. In LC2K, we will use the **jalr** instruction to save PC + 1 into register 7, and branch to a function. For example, we can run the following code:

```
            lw          0     1     zero        ; a = 0
            add         0     1     4           ; x = a
            lw          0     6     addAdr      ; load address of addOne
            jalr        6     7
            halt
            noop                                ; BEGIN addOne DEFINITION
addOne      lw          0     5     one         ; const 1
            add         4     5     3           ; return value
            jalr        7     6                 ; return
addAdr      .fill       addOne                  ; address of addOne
```

If we trace the flow of this program, we load in the address of addOne into register 6 on the third line. Coincidentally, this turns out to be 6. On the next instruction, we run **jalr 6 7**. First, we store PC + 1, which is 4, into register 7. Next, we branch to the value in register 6, which is PC 6. Thus, we have transfered control to the **addOne** function. We run the body of **addOne**, and then we run **jalr 7 6**. Here, we store PC + 1, which is 9, into register 6. Then, we branch to the value in register 7, which is 4. Thus, our new PC is 4, which corresponds to the **halt** instruction. To check the generalizability of this approach, let's consider the prior example, where we called **addOne** twice:

(code on the next page)

```
            lw          0     1     zero
            lw          0     6     addAddr
            add         0     1     4
            jalr        6     7                 ; first call
            add         0     1     4
            jalr        6     7                 ; second call
            halt
            noop                                ; BEGIN addOne DEFINITION
addOne      lw          0     5     one         ; const 1
            add         4     5     3           ; return value
            jalr        7     6                 ; return
addAdr      .fill       addOne                  ; address of addOne
```

The first call to **jalr 6 7** saves PC + 1 = 4 into register 7, and the **jalr 7 6** returns to PC 4. The second call to **jalr 6 7** saves PC + 1 = 6 into register 7, and the **jalr 7 6** returns to PC 6. Thus, we successfully execute both function calls while avoiding code duplication. Let's consider the corresponding code in ARMv8. Instead of **jalr**, we use **BL**, and we can directly specify a label. To return, we use **BR** and pass in the link register **LR** as the operand. Thus, we get the following code (we use X0 to store **a**, X1 to store **x**, and X2 for the return value):

```
            MOV         X0,   #0
            MOV         X1,   X0
            BL          addOne
            BL          addOne
            HLT                               ; stop the program
addOne:     ADDI        X2,   X1,   #1
            BR          LR
```

Register Assignments and Saving

So far, we've been assigning registers to variables arbitrarily. However, we're going to need a more structure approach to assigning registers when we start translating more complex programs. The protocol used for register assignment in an ISA is termed the **Application Binary Interface**, or **ABI**. Let's try building an ABI for LC2K. We'll start with a very basic ABI:

register 0 = constant value 0
registers 1 through 6 = scope variables
register 7 = return address

Let's try translating the following C code using the ABI above:

```
int main() {
    int x = 5;
    int y = 3;
    x = x + y;
}
```

Here, we can allocate two "scope variable" registers for x and y; let's use registers 1 and 2. We can thus assign the registers as follows:

register 0 = constant value 0
register 1 = value of x
register 2 = value of y
registers 3 through 6 = unused
register 7 = return address

Notice that due to the limited number of registers we have, if we have more than 6 variables in our function scope, we will "run out" of registers. We'll talk about how to handle this problem later on.

Let's consider another simple C program:

```
void foo() {
    int a = 1;
    int b = 2;
    a = a + b;
}

int main() {
    int x = 5;
    int y = 3;
    x = x + y;
}
```

We have four variables here: x, y, a, and b. Let's assign x and y as before:

register 0 = constant value 0
register 1 = value of x
register 2 = value of y

Since we still have registers 3 through 6 available, let's assign a and b to registers 3 and 4. If we assigned a and b to registers 1 and 2, then we would potentially overwrite the values of x and y.

register 3 = value of a
register 4 = value of b
registers 5, 6 = unused
register 7 = return address

We see here that our approach is to assign registers sequentially as functions are called. However, there's a fatal flaw in this approach: a function can get called from anywhere, with any number of registers being used. When we compile our code into assembly, we have to prespecify our register numbers, meaning that we can't dynamically assign registers to variables.

Let's try a new approach: every function is able to use registers 1 through 6. At first, you might think that this is a terrible idea: we have multiple functions modifying the same registers at the same time! However, let's think about how function scopes work in C. Let's consider the following functions:

```
void foo() {
    int x = 5;
    x += 1;
}

void bar() {
    int x = 3;
    x += 2;
}
```

Remember that we can only be in one function scope at a time, and that scopes are independent. Thus **foo** cannot change any of **bar**'s variables, and **foo** will not be running any code while **bar** is running[8]. Thus, even if we assign both **x**s to register 1, only one of them will use register 1. Thus, we can propose a new register assignment:

register 0 = constant value 0
register 1 = value of x in **foo**, value of x in **bar**
registers 2 through 6 = unused
register 7 = return address

There's one problem left: when we make a function call, nothing stops the called function from overwriting our register values! To solve this problem, we need **calling conventions**.

[8] Multithreading introduces additional logic to ensure register sharing does not take place; multiprocessing typically runs processes on different cores ensuring they have different register values.

Caller and Callee Save

Remember that variables inside of functions are treated independently of other variables in other functions, regardless of which register they're assigned to. Let's consider a simple example:

```
void foo() {
    int a = 1;
    int b = 2;
    a = a + b;
}

int main() {
    int x = 3;
    int y = 5;
    foo();
    x = x + y;
}
```

Let's compile this into LC2K code, using the following register assignment:

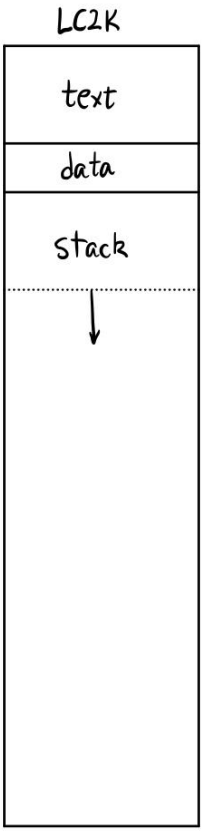
register 0 = constant value 0
register 1 = value of x in **main**, value of a in **foo**
register 2 = value of y in **main**, value of b in **foo**
registers 3 through 6 = unused
register 7 = return address

The LC2K code is as follows:

```
main        lw          0     1     three       x = 3
            lw          0     2     five        y = 5
            lw          0     6     fooAdr
            jalr        6     7                 foo()
            add         1     2     1
            halt
foo         lw          0     1     one         a = 1
            lw          0     2     two         b = 2
            add         1     2     1           a = a + b
            jalr        7     6
fooAdr      .fill       foo
one         .fill       1
two         .fill       2
three       .fill       3
five        .fill       5
```

If we look at what instructions get executed, we realize that when we call **foo**, we load 1 and 2 into registers 1 and 2 respectively. However, we never fix this when we return! Thus, when we return to **main**'s scope, the values of register 1 and 2 will be 1 and 2, not 3 and 5! We need to find a way to store these variables somewhere else.

Recall that in processors, we often have a tradeoff between capacity and speed. Registers are very fast, which is why we use registers for most computations in the processor. However, we only have a limited number of registers, and as we see in the example above, these conflicts cause us to have to find an alternative solution: storing values in memory. For this, we will use the **stack**. The stack is a special area of memory that is reserved for storing information about function calls. Let's take a look at how memory is laid out in LC2K:



As shown in the figure above, LC2K's memory is divided into three segments: the **text** section, the **data** section, and the **stack**. Recall that the text and data sections store machine code: the text section contains instructions, and the data section contains values from **.fill** directives. The stack starts directly after the data section, and is allowed to grow arbitrarily.

From previous programming courses, you may have learned about the function call stack. Recall that the function call stack stores **activation records**, which contain information about a function such as its variables. In assembly, we get to create these activation records. In general, a function's activation record, often referred to as a **stack frame**, contains the

function's return address and the values of the function's variables. Let's try designing stack frames for each of the functions in the program below:

```
void bar() {
    int c = 3;
    c++;
}

void foo() {
    int a = 1;
    int b = 2;
    bar();
    a = b + b;
}

int main() {
    int x = 4;
    foo();
    bar();
    x = x + 1;
}
```

First, we'll start with **bar**. **bar**'s stack frame needs to remember its return address as well as the value of c. Thus, we need to push two values onto the stack for **bar**. Next, we have **foo**. We need to push **foo**'s return address, the value of a, and the value of b to the stack, giving a total of three values. Finally, we have **main**, which will have a return address and the value of x. Thus, **main** will push two values onto the stack.

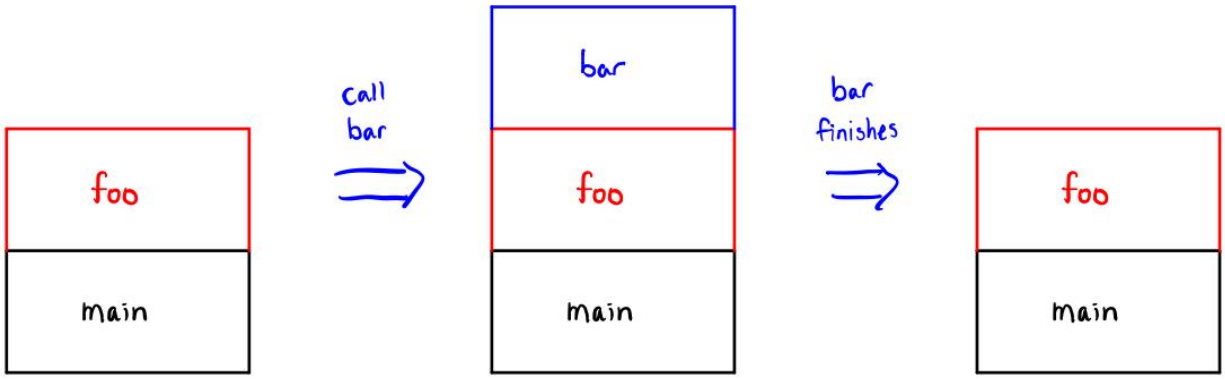
We can actually optimize the stack frames of **main** and **bar** slightly. Let's start with **main**. **main** is where program execution starts, and therefore it isn't "called" by any function. Thus, we don't need to store a return address for **main**. Next, let's consider **bar**. Since **bar** doesn't call any functions, we don't need to store its variables on the stack, because there are no called functions that could potentially overwrite the value of c. Let's take a closer look at which variables we don't have to store in stack frames. Consider the following functions:

(code on next page)

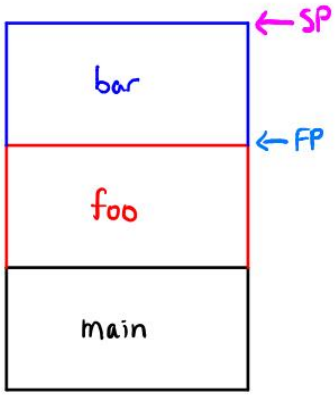
```
void funcA() {
    int c = 3;
    c++;
    foo()
}

void funcB() {
    int c = 3;
    foo();
    c = 2;
}
```

Let's start with **funcA**. While **funcA** does call **foo**, we notice that the value of c doesn't matter after the call to **foo**. In **funcA**, c is unused after **foo** is called. We see a similar situation in **funcB**, where c is immediately overwritten with a new value. This analysis is called **liveness analysis**, and allows us to determine which variables need to be stored onto the stack.

Where do we store variables on the stack? We aren't able to come up with a specific address, because the stack is variable. One key feature of the stack is that we only ever have to reference the topmost activation frame at any moment. The topmost activation frame represents the active function, and therefore we only need to consider that frame. When a function call takes place, a new activation record is placed onto the stack. When a function finishes, the topmost activation record is removed from the stack, and the activation record below it becomes active. This is illustrated in the figure below.



Here, we show the stack growing "upwards" towards smaller addresses. This mimics the behavior of the stack in architectures such as ARM and x86. Since we only ever need to reference the top of the stack, we can consider a pointer to the top of the stack, which we will refer to as the **stack pointer**. The stack pointer allows us to dynamically reference the active stack frame. In modern assembly, we will also see a **frame pointer**, which is a pointer to the beginning of the active stack frame. We won't use the frame pointer in 370, but it's good to know that it exists.



Let's return to caller and callee save. Caller and callee save are two types of **calling conventions**, which specify how the values of registers are saved across function calls. As mentioned before, we can do this using a function's activation record to store all values related to that function. To understand the difference between caller and callee save, let's take a look at an example in LC2K.

n.b. In LC2K, we will use a slightly different stack frame layout than is commonly used by ARM or x86. Rather than allocating space on the stack immediately, we will only push values onto the stack when necessary. Additionally, the stack begins directly after the data section, and grows in the direction of increasing addresses.

Consider the following C code:

```
void foo() {
    int a = 1;
    int b = 2;
    a = b + b;
}

int main() {
    int x = 4;
    foo();
    x = x + 1;
}
```

Let's assign all of our variables to be **caller saved**. When we use the term "caller saved", we indicate that it is the **caller's** responsibility to save the value of the register **before** the function call occurs. Thus, when **main** calls **foo**, it is responsible for saving the value of x before the call to foo, as shown in the LC2K code below[9]:

[9] Notice that we have a new label here: Stack. We'll see how Stack is treated when we discuss linking, but note that Stack will automatically be defined as the address immediately after the data section.

```
main        lw          0     1     four        x = 4
            lw          0     6     one
            sw          5     1     Stack       push x to stack
            add         5     6     5           increment stack pointer
            lw          0     6     fooAdr
            jalr        6     7                 foo()
            lw          0     6     negOne
            add         5     6     5           decrement stack pointer
            lw          5     1     Stack       pop x from stack
            lw          0     6     one
            add         1     6     1           x = x + 1
```

In LC2K, the stack pointer points to the **next unoccupied address** in the stack. Thus, we can directly push a value onto the stack (**sw 5 1 Stack**), but when we pop it from the stack, we have to decrement our stack pointer before loading the value from the stack. Here, we use register 5 as our stack pointer. Register 6 is being used as a temporary register, taking on a variety of values throughout the program.

When using caller save, the values of variables are saved before control is transferred to the called function. Thus, we do not need to worry about what registers are being changed by the called function, since the caller will load back the "correct" value once control is transferred back to the caller. While this approach is extremely flexible, it wastes a significant amount of time. Let's consider the following function:

```
int foo() {
    int x = 6; // r1
}

int bar() {
    int a = 1; // r1
    int b = 2; // r2
    int c = 3; // r3
    int d = 4; // r4
    foo();
    c = a + b;
}
```

Caller save would save all four variables, even though the values of c and d are never used after the function call, and only a is overwritten by the call to **foo**! Loading and storing from memory takes significantly longer than accessing registers, and when writing assembly, we want to minimize the number of unnecessary operations that are performed. We can solve the problem of unused values (c and d) by performing liveness analysis. We notice that after the call to **foo**, c is immediately overwritten, while d is never used. Thus, we do not need to push c or d to the stack, saving us two pairs of loads and stores. However, it is much trickier to

determine what registers are overwritten by a function. To handle this, we can consider using **callee save**.

In **callee save**, it is the responsibility of the called function to save register values. Let's consider the code from before:

```
void foo() {
    int a = 1;
    int b = 2;
    a = b + b;
}

int main() {
    int x = 4;
    foo();
    x = x + 1;
}
```

If we treat all variables as callee saved, we will have the following code in **foo**:

```
foo         lw          0     6     one
            sw          5     1     Stack       save r1 to stack
            add         5     6     5
            sw          5     2     Stack       save r2 to stack
            add         5     6     5
            lw          0     1     one         a = 1
            lw          0     2     two         b = 2
            add         2     2     1           a = b + b
            lw          0     6     negOne
            add         5     6     5
            lw          5     2     Stack       restore r2
            add         5     6     5
            lw          5     1     Stack       restore r1
            jalr        7     6                 return
```

Note that we only save registers 1 and 2 to the stack. This is because we only modify registers 1 and 2. While we often think of caller and callee save in terms of who is saving registers, we can also think about caller and callee save in terms of what "expectations" the caller and callee have:

Caller Save:

- Caller function: Must account for arbitrary register values after completion of callee.
- Callee function: May modify registers freely.

Callee Save:

- Caller function: May assume that all registers will be restored to original values after completion of callee.
- Callee function: Must ensure all registers are restored to their original values before returning control to the caller.

One of the advantages of callee save is that it allows us to only save which registers the callee will modify. However, we're not able to take advantage of optimizations such as liveness, since we are forced to restore all registers. There are advantages to both caller and callee save, so which one do we choose? In modern ABIs, we provide **both** caller and callee registers, and we allow the programmer (or compiler) to choose which calling convention to use for each variable. Let's take a look at the ARMv8 ABI[10]:

X0 through X7: function parameters and return values
X9 through X15: temporary (caller saved) registers
X19 through X28: callee saved registers
X30: link register (return address)
SP: stack pointer

When we write a program in ARMv8, we have the choice to specify if we want to use a caller or a callee saved register. Let's take a look at the following C code:

```
void bar() {
    int c = 3;
    c++;
}

void foo() {
    int a = 1;
    int b = 2;
    bar();
    a = a + b;
    bar();
    a = a + b;
}

int main() {
    int x = 3;
    int y = 4;
    foo();
    x = y + 1;
}
```

[10] https://developer.arm.com/documentation/ihi0055/d

Let's compile this code to LC2K. To handle the function calls, we will specify our own ABI for LC2K, giving us access to two caller save and two callee save registers:

register 0 = constant value 0
register 1 = return value
register 2 = temporary register (not saved)
register 3 = caller saved
register 4 = callee saved
register 5 = stack pointer
register 6 = temporary register (not saved)
register 7 = return address

To optimize the number of loads and stores that need to be performed, we can calculate the number of load/store pairs executed for each variable assuming caller save, and then assuming callee save. These values are independent of the effects of other variables, meaning that we can choose the calling convention that minimizes the number of loads for all variables. Let's first consider caller save, noting loads and stores performed:

```
void bar() {
    int c = 3;
    c++;
}

void foo() {
    int a = 1;
    int b = 2;
    STORE a, STORE b
    bar();
    LOAD a, LOAD b
    a = a + b;
    STORE a, STORE b
    bar();
    LOAD a, LOAD b
    a = a + b;
}

int main() {
    int x = 3;
    int y = 4;
    STORE y
    foo();
    LOAD y
    x = y + 1;
}
```

Now, we can go through and add up the number of times that each variable is stored and loaded from memory. We get the following totals:

Function	Variable	Caller Save	Callee Save
main	x	0	
	y	1	
foo	a	2	
	b	2	
bar	c	0	

Now, let's consider callee save. We can take a similar approach as before, noting where loads and stores have to be performed. Note that when we write "**STORE a**", this technically refers to storing the *previous* value of the register **a** corresponds to. Depending on how registers are assigned, this could be x, y, or even an unused register.

```
void bar() {
    STORE c
    int c = 3;
    c++;
    LOAD c
}

void foo() {
    STORE a, STORE b
    int a = 1;
    int b = 2;
    bar();
    a = a + b;
    bar();
    a = a + b;
    LOAD a, LOAD b
}

int main() {
    int x = 3;
    int y = 4;
    foo();
    x = y + 1;
}
```

Notice that no callee saves occur in **main**. **main** is a special function in that it is not "called" by any function. Thus, we do not need to save register values across **main**. We can fill in the remainder of our table:

Function	Variable	Caller Save	Callee Save
main	x	0	0
	y	1	0
foo	a	2	1
	b	2	1
bar	c	0	2

Note that c is saved **twice**, since **bar** is called twice in **foo** (which itself is called once from **main**). Now that we have the number of load/store pairs in each calling convention, we can begin assigning registers. Remember that each function has its "own" set of registers, meaning that while we are limited to one caller and one callee saved register in this example, this limit only applies within the scope of a function. Thus, we would not be able to use caller save for both x and y, since they both are in the same function scope (**main**). However, we would be able to assign x and a to caller saved registers, since a is used in **foo** and therefore does not contribute towards **main**'s limit.

To assign registers to caller or callee save, we will first start by choosing the calling convention that minimizes the number of loads and stores:

- x (main): no preference
- y (main): **callee save**
- a (foo): **callee save**
- b (foo): **callee save**
- c (bar): **caller save**

After this, we have two tasks left: x is unassigned, and **foo** is over the limit for the number of callee saved registers that are used.

First, let's resolve **main**'s "no preference". Since y minimizes the number of loads and stores performed by using callee save, we will assign y to the callee saved register, and assign x to the caller saved register.

Finally, we finish assigning **foo**. a and b both minimize the number of loads and stores performed by using callee save. Let's consider both assignments:

- a = callee, b = caller: 3 load/store pairs
- a = caller, b = callee: 3 load/store pairs

Notice that each assignment results in the same number of load/store pairs. Thus, there is no difference between assigning the caller saved register to a versus b, and we can choose either combination arbitrarily. Thus, a valid assignment would be:

- x (main): caller save (r3)
- y (main): callee save (r4)
- a (foo): caller save (r3)
- b (foo): callee save (r4)
- c (bar): caller save (r3)

Finally, we can compile the C code to LC2K, using the register assignments above:

```
main        lw          0     3     three       x = 3
            lw          0     4     four        y = 4
            lw          0     6     one
            sw          5     3     Stack       SAVE x
            add         5     6     5
            lw          0     6     FooAdr
            jalr        6     7                 foo()
            lw          0     6     negOne
            add         5     6     5
            lw          5     3     Stack       LOAD x
            lw          0     6     one
            add         4     6     3           x = y + 1
            halt                                (end main)
foo         lw          0     6     one
            sw          5     7     Stack       SAVE return address
            add         5     6     5
            sw          5     4     Stack       SAVE r4
            add         5     6     5
            lw          0     3     one         a = 1
            lw          0     4     two         b = 2
            sw          5     3     Stack       SAVE a
            add         5     6     5
            lw          0     6     BarAdr
            jalr        6     7                 bar()
            lw          0     6     negOne
            add         5     6     5
            lw          5     3     Stack       LOAD a
            add         3     4     3           a = a + b
            lw          0     6     one
```

```
            sw          5     3     Stack       SAVE a
            add         5     6     5
            lw          0     6     BarAdr
            jalr        6     7                 bar()
            lw          0     6     negOne
            add         5     6     5
            lw          5     3     Stack       LOAD a
            add         3     4     3           a = a + b
            add         5     6     5
            lw          5     4     Stack       LOAD r4
            add         5     6     5
            lw          5     7     Stack       LOAD return address
            jalr        7     6                 (return)
bar         lw          0     3     three       c = 3
            lw          0     6     one
            add         3     6     3           c++
            jalr        7     6                 (return)
FooAdr      .fill       foo
BarAdr      .fill       bar
negOne      .fill       -1
one         .fill       1
two         .fill       2
three       .fill       3
four        .fill       4
```

A few points to note from the above code:

- Notice that the return address is saved in **foo**. This is because we overwrite the return address when we make a function call to **bar**, meaning that we have to restore the return address before we return control to **main**. If we don't store the original return address, we'll end up in an infinite loop since we continue to branch to the instruction after the second **bar()** call.
- We don't need to save the return address in **bar** because it doesn't get overwritten. The return address is very similar to a callee saved register in this sense.
- We don't need to save the value of **r4** in **bar** as well, since **bar** does not use **r4**.

Notice how long LC2K code can get when multiple functions are involved. This code can be made much cleaner by splitting it up into multiple files. We'll talk about how to handle multiple files in the next section.

Linking

So far, we've only dealt with single file LC2K programs. We can assemble a LC2K file into a machine code file, which can then be run on a processor. As we saw in the previous section, however, LC2K files can become extremely large, and we would like to improve readability by splitting up the code into multiple files. When we split up code into different files, we give ourselves the ability to group by functionality. For example, we can use C header files and multiple C code files to split up code into multiple files, allowing us to group variables and functions by what they do. Let's consider the LC2K program from the previous section. We would like to split this up into three files, one for each function. Let's start with what we'd like to have. Notice how we've split up **main**, **foo**, and **bar** into their individual files. However, we haven't split up their text and data sections. Additionally, notice how we have the label "one" present in each file; this allows us to modify the values independently in each file. For example, if we change "c++" to "c+=2" inside **bar**, we only have to modify a few lines in **bar.lc2k**, and the other files remain the same.

file main.lc2k

```
main        lw          0     3     three       x = 3
            lw          0     4     four        y = 4
            lw          0     6     one
            sw          5     3     Stack       SAVE x
            add         5     6     5
            lw          0     6     FooAdr
            jalr        6     7                 foo()
            lw          0     6     negOne
            add         5     6     5
            lw          5     3     Stack       LOAD x
            lw          0     6     one
            add         4     6     3           x = y + 1
            halt                                (end main)
negOne      .fill       -1
one         .fill       1
three       .fill       3
four        .fill       4
```

file foo.lc2k

```
foo         lw          0     6     one
            sw          5     7     Stack       SAVE return address
            add         5     6     5
            sw          5     4     Stack       SAVE r4
            add         5     6     5
            lw          0     3     one         a = 1
            lw          0     4     two         b = 2
            sw          5     3     Stack       SAVE a
```

```
            add         5     6     5
            lw          0     6     BarAdr
            jalr        6     7                 bar()
            lw          0     6     negOne
            add         5     6     5
            lw          5     3     Stack       LOAD a
            add         3     4     3           a = a + b
            lw          0     6     one
            sw          5     3     Stack       SAVE a
            add         5     6     5
            lw          0     6     BarAdr
            jalr        6     7                 bar()
            lw          0     6     negOne
            add         5     6     5
            lw          5     3     Stack       LOAD a
            add         3     4     3           a = a + b
            add         5     6     5
            lw          5     4     Stack       LOAD r4
            add         5     6     5
            lw          5     7     Stack       LOAD return address
            jalr        7     6                 (return)
negOne      .fill       -1
one         .fill       1
two         .fill       2
FooAdr      .fill       foo

file bar.lc2k
bar         lw          0     3     three       c = 3
            lw          0     6     one
            add         3     6     3           c++
            jalr        7     6                 (return)
one         .fill       1
three       .fill       3
BarAdr      .fill       bar
```

Notice that each of these files can *almost* be assembled into machine code by themselves. However, we have a few missing labels, such as "FooAdr" in **main.lc2k** and "BarAdr" in **foo.lc2k**. Notice that these labels, while undefined in the files where they are used, are defined in other files. We will treat these labels as **global labels**, which can be referenced by any file. To resolve these global labels, we need to add a step which combines our assembly files into a machine code file. We call this step **linking**.

Before we start linking, let's think about efficiency. Consider an extremely large program where we want to make a small change to one function (perhaps a bugfix!). If the linking process took in assembly files as input, we would need to reassemble every single instruction, which would take a lot of time! Let's think about how we can optimize this. Rather than having the linker convert assembly into machine code, we should have an assembler perform that for us. However, the assembler doesn't know how to resolve undefined global labels! Let's consider the file **main.lc2k** from before:

```
main        lw          0     3     three       x = 3
            lw          0     4     four        y = 4
            lw          0     6     one
            sw          5     3     Stack       SAVE x
            add         5     6     5
            lw          0     6     FooAdr
            jalr        6     7                 foo()
            lw          0     6     negOne
            add         5     6     5
            lw          5     3     Stack       LOAD x
            lw          0     6     one
            add         4     6     3           x = y + 1
            halt                                (end main)
negOne      .fill       -1
one         .fill       1
three       .fill       3
four        .fill       4
```

When we give this file to the assembler, it will complain about not being able to find "Stack" or "FooAdr". Let's tell the assembler instead to give us a "checklist" of what needs to be fixed. This "checklist" might look something like the following

- ☐ Find Stack
- ☐ Find FooAdr
- ☐ Fix **sw 5 3 Stack**
- ☐ Fix **lw 0 6 FooAdr**
- ☐ Fix **lw 5 3 Stack**

The remainder of the assembly can be translated to machine code as usual. To make it easier for the linker, LC2K resolves undefined global labels as if the label was defined at address 0. Thus, **sw 5 3 Stack** would be "assembled" as **sw 5 3 0**. In our object file, we'll have five sections: a header, the text and data sections corresponding to the assembly code, and two new sections: the **symbol table** and the **relocation table**. The symbol table lists all **global labels** that are used, either defined or undefined. The relocation table lists all instructions

that must be updated during linking. Let's take a look at the symbol and relocation tables for **main.lc2k**:

```
symbol table:
Stack       U     0
FooAdr      U     0

relocation table:
0     lw    three
1     lw    four
2     lw    one
3     sw    Stack
5     lw    FooAdr
7     lw    negOne
9     lw    Stack
10    lw    one
```

Notice how this correpsonds to the "checklist" we had before. The symbol table contains two entries, one for "Stack" and one for "FooAdr". Both of these are listed as "U", meaning that they are undefined. The relocation table contains eight entries, each corresponding to a line that needs to be updated during linking. First, we see the lines **3 sw Stack**, **5 lw FooAdr**, and **9 lw Stack**. These correspond to undefined global labels that need to be resolved by the linker. We have five other lines which correspond to local labels. Let's look at the symbol and relocation tables for the other files:

```
foo.lc2k symbol table:
Stack       U     0
BarAdr      U     0
FooAdr      D     3

foo.lc2k relocation table:
0     lw    one
1     sw    Stack
3     sw    Stack
5     lw    one
6     lw    two
7     sw    Stack
9     lw    BarAdr
11    lw    negOne
13    lw    Stack
15    lw    one
16    sw    Stack
18    lw    BarAdr
```

```
20    lw    negOne
22    lw    Stack
25    lw    Stack
27    lw    Stack
3     .fill foo
```

Again, the symbol table contains all unresolved global labels[11] ("Stack" and "BarAdr"). We also see a new line: "**FooAdr D 3**". This line indicates that the global label "FooAdr" is defined in the data section. The "3" indicates an offset into the data section, in this case, index 3 (the fourth line). This indexing scheme will be used in both the symbol and the relocation table. For example, the line "**0 lw one**" in the relocation table indicates that the instruction at index 0 in the text section uses the label "one" and needs to be updated during linking. The line "**3 .fill foo**" indicates that index 3 in the **data** section uses the label "foo" and needs to be updated during linking.

Finally, we can look at the symbol and relocation table entries for **bar.lc2k**:

bar.lc2k symbol table:
```
BarAdr      D     2
```

bar.lc2k relocation table:
```
0     lw    three
1     lw    one
2     .fill bar
```

Now that we have these, how do we go about linking the files? First, the linker combines each of the files' text and data into a combined text section and a combined data section. Next, the linker uses the symbol and relocation table entries to determine the new address (based on the offsets) of a specific label. These will be used to update the machine code accordingly[12].

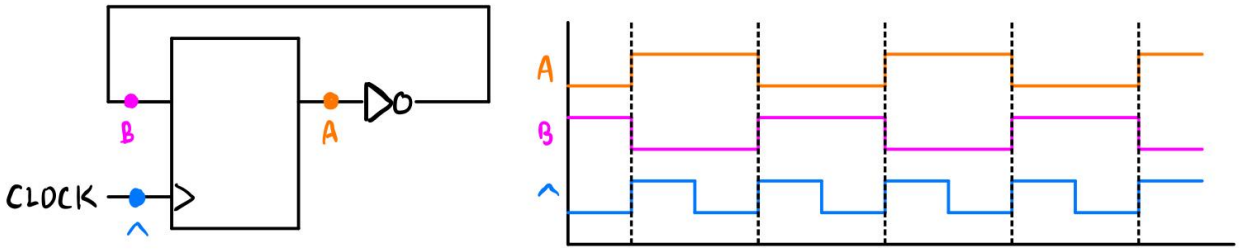
[11] It may seem a bit arbitrary that we can just call a label "global" if it's not defined. In LC2K, our convention is to have global labels start with a capital letter (A-Z), and local labels start with a lowercase letter (a-z).
[12] I wish I could go more in detail about the algorithm, but that's up to you in Project 2L!

Chapter 6: Processors

Now that we know how to write assembly code, and how to convert it to machine code, let's look at how machine code gets run on a processor. To run machine code on a processor, we need to use the components we designed in Chapter 3.

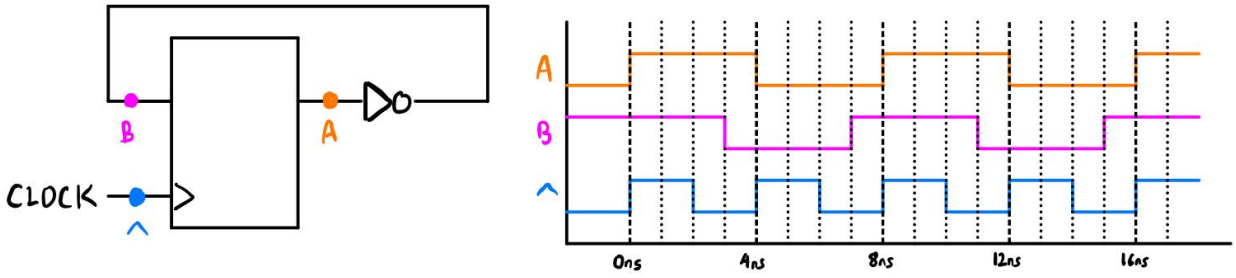
State in Processors

The most important component in a processor is the D flip flop. The D flip flop gives us a notion of "state", allowing us to store values over time. To understand how the D flip flop creates "state" in our systems, let's consider a very very simple example. For simplicity, we'll ignore propagation delays.

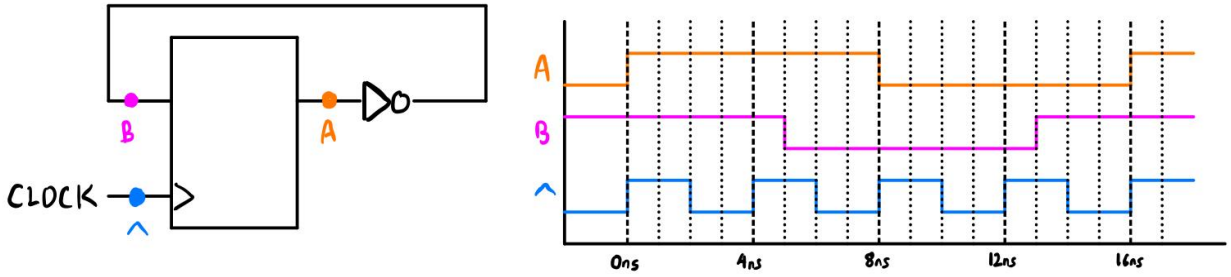


At first, this system might look a bit odd; we're sending the output of a NOT gate back into itself. However, remember a property of the D flip flop: we can give it any arbitrary input, but the output (A) only updates upon a rising clock edge (dashed lines). Even though the value of B changes to the opposite of A, the update of the D flip flop's output only occurs one clock cycle later.

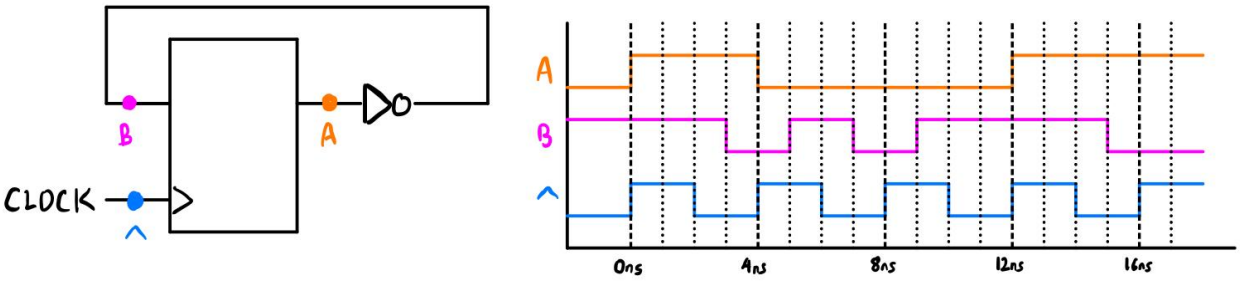
Next, let's take a look at the value of A during each clock cycle. To do this, we look at the value between each dashed line. We see that on each clock cycle, A alternates between low and high. In general, A takes on the value of B from the previous clock cycle upon a rising edge. During the time until the next rising edge, A retains this value, while B updates to its new value. Let's take a look at the same system, but assuming propagation delay in the NOT gate. Again for simplicity, assume that the D flip flop's output updates instantaneously. If the flip flop has delay, we can consider this delay as "part of" the NOT gate's delay. Let's set the clock period to 4 ns. First, we'll consider an example where the NOT gate has a propagation delay of 3 ns:



Notice that while it takes 3 seconds for B to properly update, the behavior of A remains the same! This is since the time needed for B to update is less than the clock period. Thus, B has fully updated by the time the next rising clock edge occurs. Let's consider another example, where the NOT gate now has a propagation delay of 5 ns:



At first glance, this doesn't look too bad. We've just doubled the clock period. However, since the output of the NOT gate is "undefined" during the 5 ns after, we could have **any** behavior here! Thus, the following output is possible:

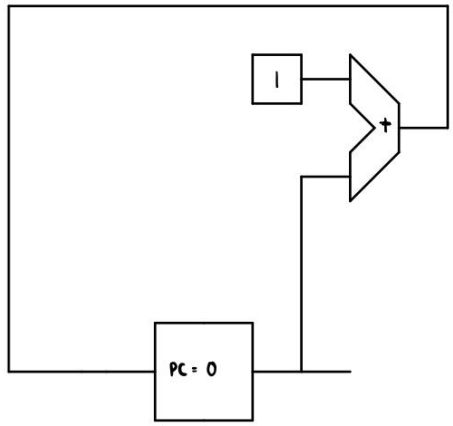


Suddenly our behavior becomes undefined due to the fluctuations in the NOT gate's output! If we replace the NOT gate with more complex circuits, there's no guarantee at all that we'll have the intended behavior.

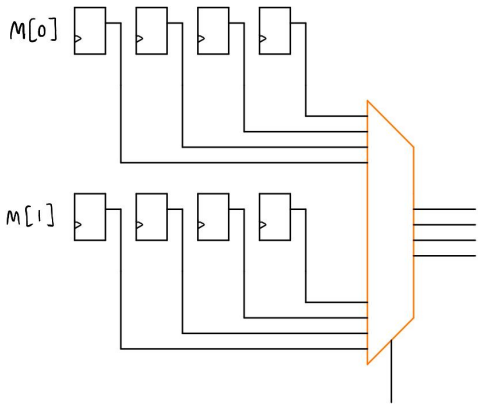
Building a Processor

Now that we understand the concept of state, let's build a processor for LC2K! This might feel like a massive jump, but we'll take it step by step.

First, we'll start by keeping track of our program counter (PC). Recall that the PC keeps track of the memory address of the next instruction. Let's start with a simplified version, where we increment the PC by one every clock cycle. This can be implemented again using D flip flops for state. Here, we've grouped 32 D flip flops together into a *register*[13], which represents a single 32-bit number. In this case, the register keeps track of the current PC. We use an adder to add the constant value 1 to the PC, and on the next clock cycle, PC will update to PC + 1.

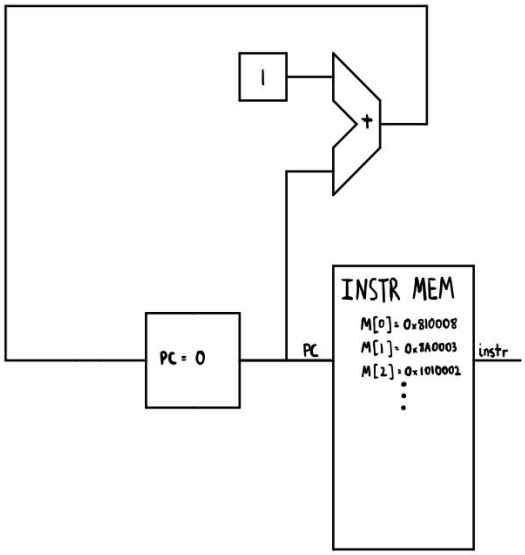


Now that we have our PC updating on each cycle, let's read from memory. We can think of memory as an array of D flip flops joined by a large MUX at the end. We can select an address to read from, and the value at the corresponding memory address will be output, similar to the diagram shown below (a 2-word, 4-bit memory circuit). This allows us to find the instruction that we want to execute.



[13] Yes, it's called a register again. It's easy to get processor registers confused with assembly registers. One way to remember this is that assembly registers are often implemented as processor registers. However, not all processor registers are assembly registers (for example, PC).

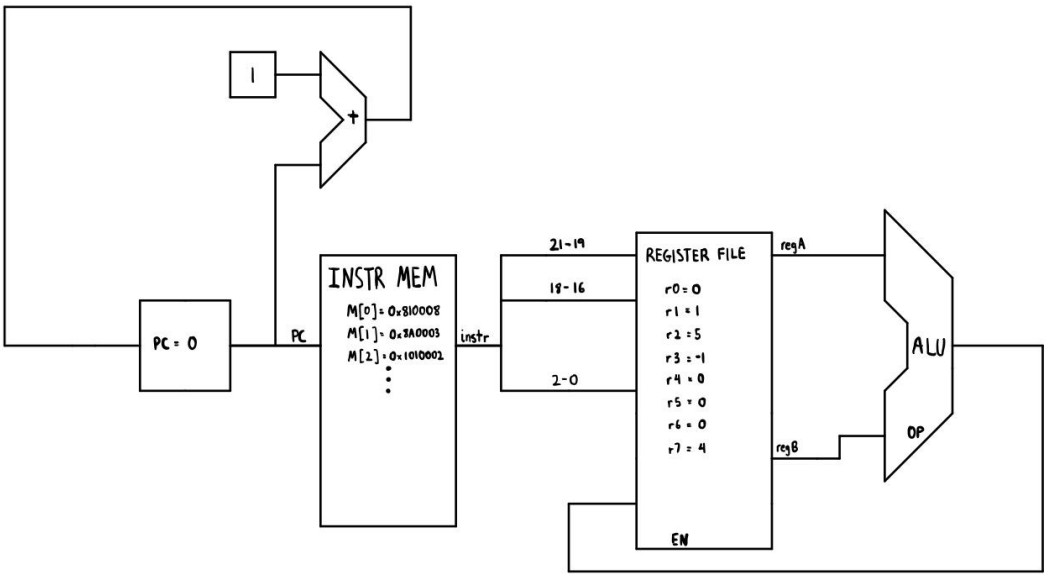
To read the corresponding instruction from memory, we use the PC as an input to instruction memory:



Now, our "processor" can fetch instructions one by one from memory, and output them. We'll refer to the process of reading instructions from memory as **instruction fetch**. Next, we have to determine what the instruction is doing, which we will call **instruction decode**. To determine this, we will split the instruction into all potential fields that we may need. Recall that since we cannot dynamically select bits in a circuit, we need to get all the information we may potentially need, and then use MUXes to select what we want. The fields we will extract are:

- Bits 24-22: **opcode**
- Bits 21-19: **regA** (add, nor, lw, sw, beq)
- Bits 18-16: **regB** (add, nor, lw, sw, beq)
- Bits 15-0: **offset** (lw, sw, beq)
- Bits 2-0: **destReg** (add, nor)

First, let's build up the functionality we need. Again, when designing processors, we need to get all potentially important values first, and then select what we need. Let's start with **add** and **nor**. These instructions will use the same datapath:



We've introduced two new components here: the **register file** and the **ALU**. The register file is very similar to instruction memory: it stores the values of all 8 registers, and allows the processor to access register values. Notice that the register file is **dual ported**, meaning that we can access the values in **regA** and **regB** simultaneously. We'll come back to the bottom two inputs on the left side of the register file after we discuss the ALU.

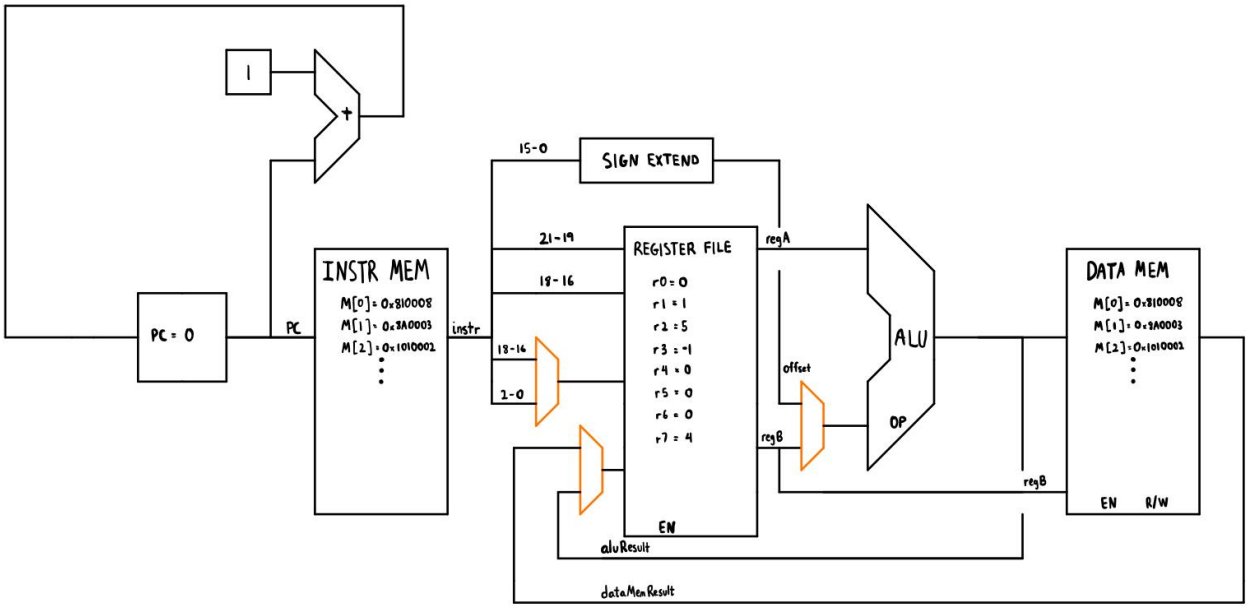
The ALU, or **arithmetic logic unit**, allows the processor to perform more complex operations. In LC2K, the ALU will only have to perform two operations: addition and bitwise NOR. Real-world ALUs may perform more complex operations, such as subtraction, multiplication, and other bitwise operations. The ALU is typically implemented by calculating each potential result, and then using a MUX to select the desired operation.

To determine the operation of the ALU, we will use the opcode. We can create a small combinational logic circuit that helps us select the ALU operation. We'll ignore this for now and return to selecting the operation when we discuss the control ROM.

Finally, let's consider the result of the ALU, which we see being sent to the register file. The bottom two inputs to the register file (on the left) allow us to write values to the register file. First, we specify which register we want to write to (bits 2-0, or **destReg**). Next, we provide a value to write to **destReg**, which is our ALU result. Now, we are able to write the result of **regA + regB** (or **regA NOR regB**) to **destReg**. Our processor is now able to run **add** and **nor** instructions!

Next, let's modify the datapath to allow **lw** and **sw** to run. Keep in mind that these modifications **must** ensure that **add** and **nor** are still able to run! Let's take a look at what needs to be modified. The first critical difference is that **lw** and **sw** work with memory. Since instruction memory isn't dual ported, we cannot access a second value, and thus must add a second memory component (**data memory**). Next, let's consider what address we need to

access in memory: **regA + offset**. Currently, we don't utilize the offset at all in the processor. Finally, we consider the register we need to write to with **sw**: **regB**. We'll need to add logic to handle this. Additionally, we write the value of **M[regA + offset]** to **regB**, so we will have to add logic to write the result of accessing data memory. Let's make these changes:



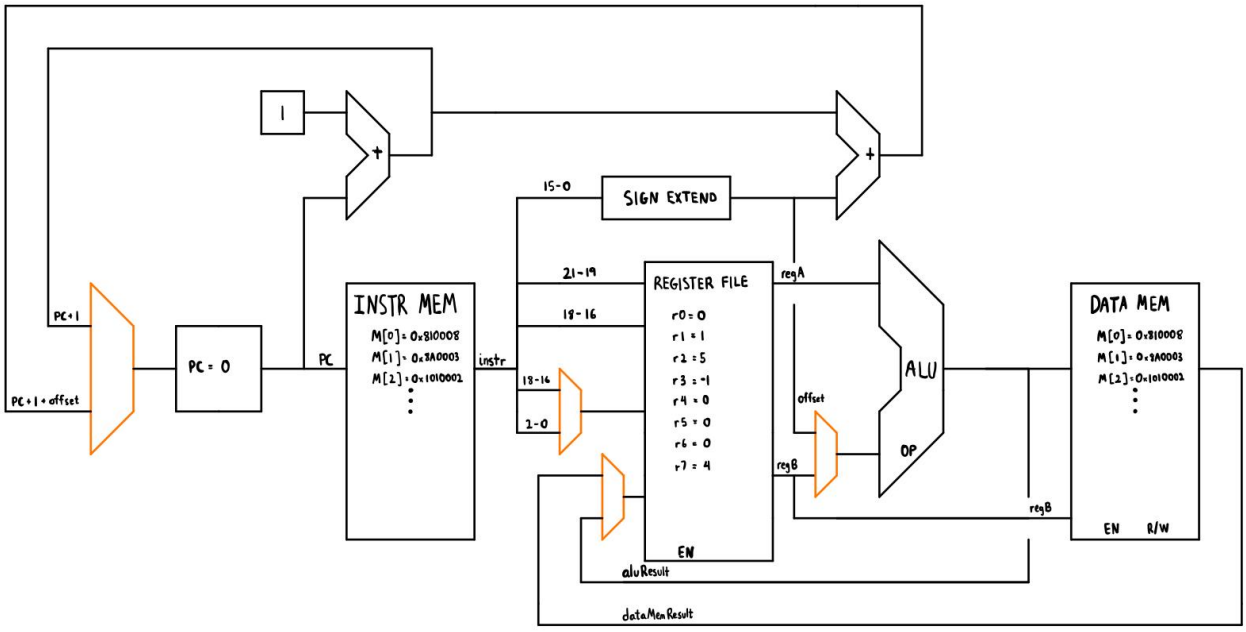
The first key difference is the addition of data memory. Data memory functions very similar to instruction memory, except that we allow writing to data memory. We add a MUX to the bottom input of the ALU allowing us to select between **regA + regB** (or **regA NOR regB**) and **regA + offset**. We won't ever use **regA NOR offset** in LC2K. Once we have calculated the desired address, we can send this address to data memory, and either read or write the value. If we are reading a value (**lw**), we will send this resulting value to the register file. We add a MUX to select between the ALU result and the value in data memory when writing to the register file. Additionally, we add a MUX to select between **regB** (18-16) and **destReg** (2-0) depending on the opcode of the current instruction. If we are storing a value, we will write the value of regB, which has been sent to the bottom input, to the desired address (from the ALU). No updates are needed to the register file.

Finally, let's add functionality for **beq**. When designing these processors, we won't implement **jalr** due to its relative complexity. Again, let's go through what we need:

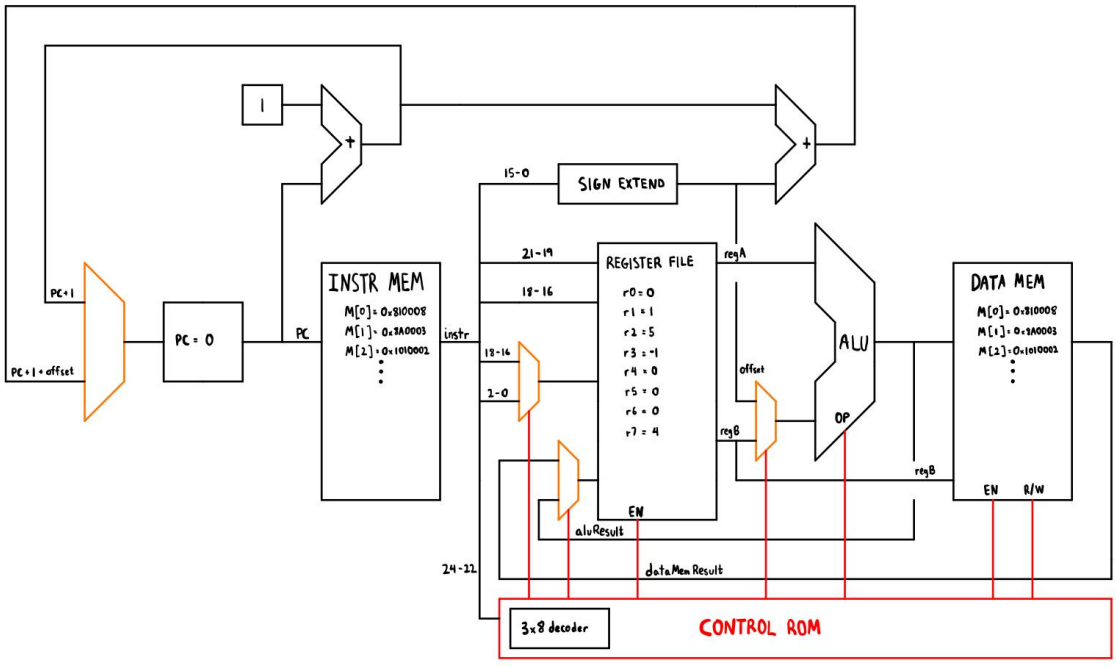
- Functionality to update PC = PC + 1 + offset
- Logic to compare **regA** and **regB** for equality

To update PC = PC + 1 + offset conditionally, we can add a new MUX, which inputs to the PC. This MUX will select between PC + 1 or PC + 1 + offset, depending on the opcode and the result of the equality check. We'll delegate the equality check to the ALU. We don't show the

logic used to determine the select bit of the newly added MUX, but it will check that the opcode is 0b100 (**beq**) and that the equality bit from the ALU is 1.

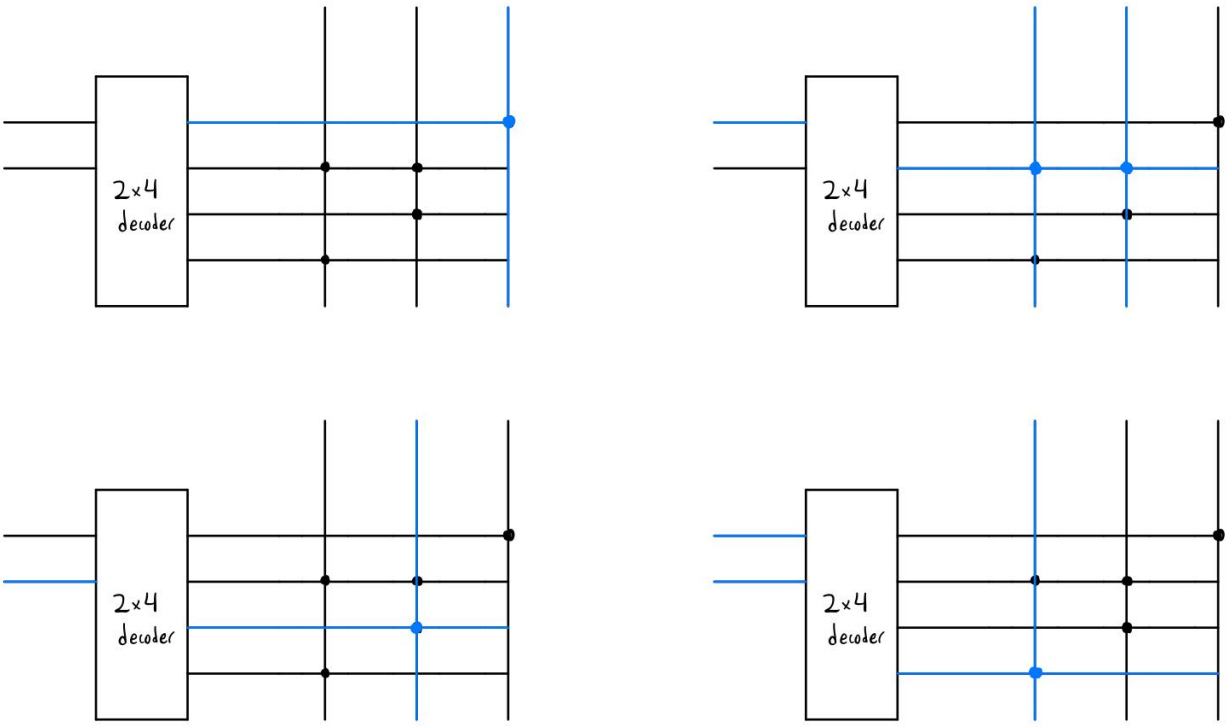


Now, our processor is able to run **add**, **nor**, **lw**, **sw**, and **beq**! Notice that we can easily run **noop** by not doing anything, and **halt** can be performed by stopping the clock (we won't go into how this is done). Thus, we have all the logic we need! Finally, we need to add a **control ROM**, which will handle the signals being sent to the MUXes, the register file, the ALU, and data memory:



Control ROMs

Let's take a closer look at how the control ROM from above works. First, we pass the opcode into a **3x8 decoder**. A decoder is a component that takes in a binary number with **n** bits, and outputs $\mathbf{2^n}$ bits using **one-hot encoding**. One-hot encoding means that only one out of the 2^n bits is high, while all the other bits are low. Let's take a look at a simple control ROM, which takes in 2 bits and outputs 3 bits:



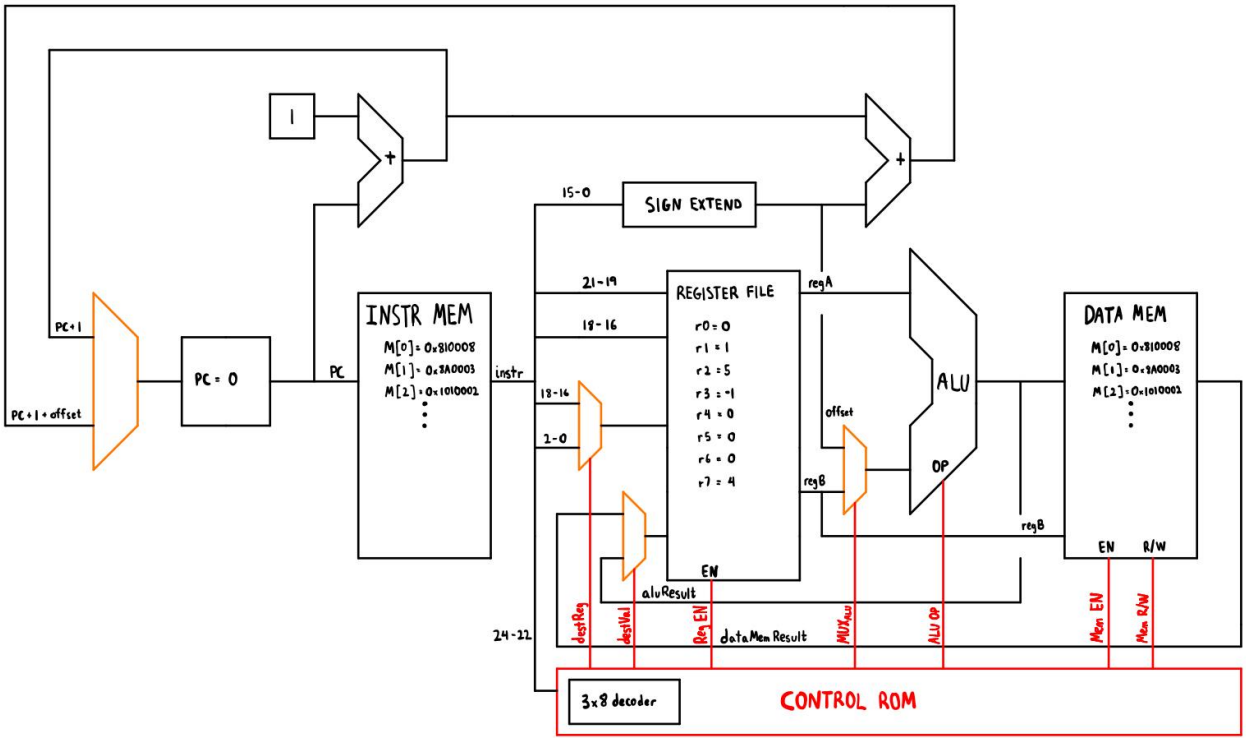
Here, each dot represents a connection between the horizontal and vertical "layers" of the control ROM. To avoid signals flowing back through these connections, we use a diode, which allows current to pass in one direction only. Notice how the control ROM allows us to encode one "set" of output signals for each potential combination of input signals. We can use this to our advantage in our processor, by setting the **control signals** for each datapath component. The control ROM for our processor will set the values for seven control signals. From left to right:

- $\text{MUX}_{\text{destReg}}$ (select between **regB** and **destReg**)
- $\text{MUX}_{\text{destVal}}$ (select between ALU result and memory value)
- Reg EN (allow/do not allow writing to register)
- MUX_{ALU} (select between **regB** and **offset**)
- ALU OP (select between addition and bitwise NOR)
- Mem EN (allow/do not allow reads/writes to memory)
- Mem R/W (read from memory or write from memory)

Let's write out the control signals for each opcode. Recall that in a MUX, **1** selects the bottom input, and **0** selects the top input. The select bits for the other components are as follows:

- ALU: 0 = addition, 1 = bitwise NOR
- Register EN: 0 = disallow writing to register, 1 = allow writing to register
- Memory EN: 0 = disallow reading/writing from memory, 1 = allow reading/writing from memory
- Memory R/W: 0 = read from memory, 1 = write to memory

We indicate any values that we don't care about with an X. The datapath with the seven control signals labeled is shown below.



Let's start with **add** and **nor**. We will perform either addition (ALU OP = 0) or bitwise NOR (ALU OP = 1) on **regA** and **regB** (MUX_{ALU} = 1), and write the value to **destReg** (Reg EN = 1, $MUX_{destVal}$ = 1, $MUX_{destReg}$ = 1). We do not modify memory (Mem EN = 0), and thus we do not care about Mem R/W (Mem R/W = X).

Opcode	$MUX_{destReg}$	$MUX_{destVal}$	Reg EN	MUX_{ALU}	ALU OP	Mem EN	Mem R/W
add	1	1	1	1	0	0	X
nor	1	1	1	1	1	0	X

Next, let's add in the control signals for **lw**. We will perform addition (ALU OP = 0) between **regA** and **offset** (MUX_{ALU} = 0), and use this as the address to read from memory (Mem EN = 1, Mem R/W = 0). We will then write this value to **regB** (Reg EN = 1, $MUX_{destVal}$ = 0, $MUX_{destReg}$ = 0).

Opcode	$MUX_{destReg}$	$MUX_{destVal}$	Reg EN	MUX_{ALU}	ALU OP	Mem EN	Mem R/W
add	1	1	1	1	0	0	X
nor	1	1	1	1	1	0	X
lw	0	0	1	0	0	1	0

We can analyze **sw** similarly. We will perform addition (ALU OP = 0) between **regA** and **offset** (MUX_{ALU} = 0), and use this as the address to store **regB** to memory (Mem EN = 1, Mem R/W = 1). We do not need to modify the register file (Reg EN = 0, $MUX_{destVal}$ = X, $MUX_{destReg}$ = X).

Opcode	$MUX_{destReg}$	$MUX_{destVal}$	Reg EN	MUX_{ALU}	ALU OP	Mem EN	Mem R/W
add	1	1	1	1	0	0	X
nor	1	1	1	1	1	0	X
lw	0	0	1	0	0	1	0
sw	X	X	0	0	0	1	1

Next is **beq**: we perform an equality check (ALU OP = X) between **regA** and **regB** (MUX_{ALU} = 1). We do not use memory (Mem EN = 0, Mem R/W = X), and we do not write to registers (Reg EN = 0, $MUX_{destVal}$ = X, $MUX_{destReg}$ = X).

Opcode	$MUX_{destReg}$	$MUX_{destVal}$	Reg EN	MUX_{ALU}	ALU OP	Mem EN	Mem R/W
add	1	1	1	1	0	0	X
nor	1	1	1	1	1	0	X
lw	0	0	1	0	0	1	0
sw	X	X	0	0	0	1	1
beq	X	X	0	1	X	0	X

Finally, we have **noop** and **halt**. Here, we do not need to perform any operations. The only thing we need to ensure is that all enable (EN) signals are low, to prevent unwanted modifications to registers or memory.

Opcode	$MUX_{destReg}$	$MUX_{destVal}$	Reg EN	MUX_{ALU}	ALU OP	Mem EN	Mem R/W
add	1	1	1	1	0	0	X
nor	1	1	1	1	1	0	X
lw	0	0	1	0	0	1	0
sw	X	X	0	0	0	1	1
beq	X	X	0	1	X	0	X
halt	X	X	0	X	X	0	X
noop	X	X	0	X	X	0	X

With this, we've finished designing the **single-cycle processor**! We call this a single-cycle processor because each instruction requires exactly one clock cycle to complete. Next, we'll look at the performance of the single-cycle processor.

Single Cycle Performance

Remember that each component in the single cycle processor has its corresopnding propagation delay. To find out the fastest clock speed we can run our processor at, we need to determine the critical path. Recall that to effectively maintain state, we need to have all operations complete before the next rising edge. To find the critical path in a single cycle processor, we need to find the **instruction** that has the longest runtime. Let's take a look at an example:

Consider a single cycle processor with the following component latencies:

- instruction memory read: 10 ns
- register file read: 5 ns
- register file write: 8 ns
- ALU: 15 ns
- data memory read: 25 ns
- data memory write: 35 ns

Treat all components not listed above as having 0 ns latencies.

First, let's take a look at which components listed above are used by each instruction:

- **add**: instruction memory read, register file read, ALU, register file write
- **nor**: instruction memory read, register file read, ALU, register file write
- **lw**: instruction memory read, register file read, ALU, data memory read, register file write
- **sw**: instruction memory read, register file read, ALU, data memory write
- **beq**: instruction memory read, register file read, ALU
- **noop/halt**: instruction memory read

Now, we can add up the component delays for each instruction:

- **add**: 10 ns + 5 ns + 15 ns + 8 ns = 38 ns
- **nor**: 10 ns + 5 ns + 15 ns + 8 ns = 38 ns
- **lw**: 10 ns + 5 ns + 15 ns + 25 ns + 8 ns = 63 ns
- **sw**: 10 ns + 5 ns + 15 ns + 35 ns = 65 ns
- **beq**: 10 ns + 5 ns + 15 ns = 30 ns
- **noop/halt**: 10 ns = 10 ns

We see that the longest instruction is **sw** with a total latency of 65 ns. Thus, the clock period of the processor is 65 ns. To calculate the performance of a program, we can use the **iron law**: runtime = # instructions * CPI * clock period. CPI refers to a value called **cycles per instruction**, and measures the number of cycles it takes on average to run a single instruction. For our single cycle processor, our CPI is always 1, since every instruction takes one cycle to execute. Let's consider the following **benchmark program**. A benchmark program is a program that we can use to compare the performance of different processors.

- 15 **add** instructions
- 15 **nor** instructions
- 30 **lw** instructions
- 20 **sw** instructions
- 10 **beq** instructions
- 10 **noop or halt** instructions

Here, we have a total of 100 instructions. On our single cycle processor, we again have a CPI of 1, and a clock period of 65 ns. Thus, the execution time of this benchmark will be 100 * 1 * 65 = 6500 ns.

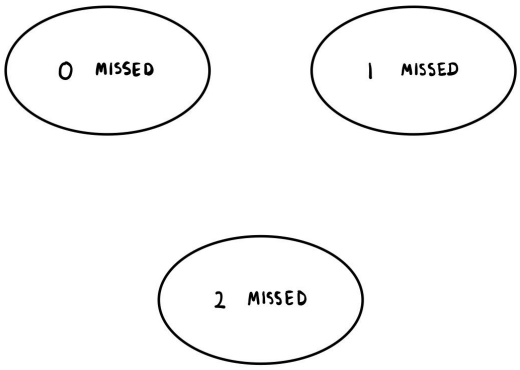
Notice that all instructions take the same amount of time to execute, meaning that there is significant downtime on shorter instructions such as **noop** or **add**. If we compare the latencies of **noop** and **add** to that of **sw**, almost 1/3 of the clock cycle is unused when running **add**, and over 5/6 of the clock cycle is unused when running **noop**. To fix this problem, we can split up each instruction into multiple cycles, and only run as many cycles as necessary. This is the concept of the **multi-cycle processor**. To design a multi-cycle processor, we will need to have a means to store more complex states over multiple clock cycles.

Finite State Machines

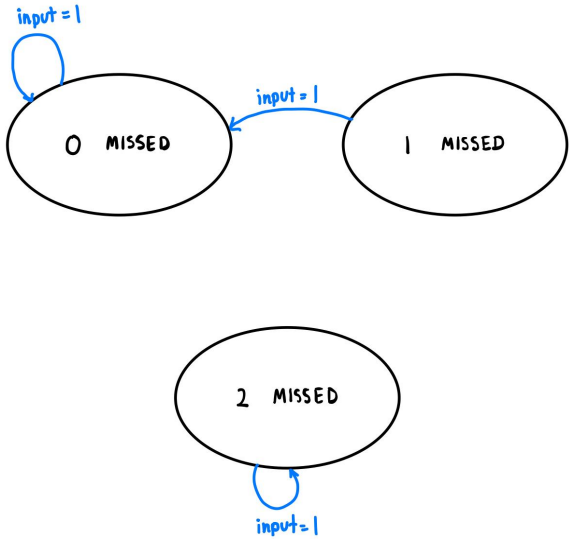
Finite state machines allow us to represent the state of some system, as well as handle transitions between these states. There are two general types of FSMs: Moore machines and Mealy machines. The key difference between Moore and Mealy machines lies in how the output is determined. In a Moore machine, the output is solely determined by the current state. In a Mealy machine, the output is determined by the current state, as well as the value of the provided inputs. Let's illustrate this with an example.

"Trippie" is continuing her in-game adventure[14], and has paired up with the mysterious "agt cooper" to finish some tasks. "agt cooper" has a transmitter that sends regular pings to a FSM on Trippie's computer. If a ping is received at the expected time, the FSM receives a 1. If a ping is missed, the FSM receives a 0. Design an FSM that outputs 1 if at least two pings have been missed in a row. Once two pings have been missed, the FSM should output 1 regardless of if future pings are received.

To design a FSM for this example, we can start by identifying the potential states that the system can be in. There are three states for this system: no pings missed in a row, one ping missed in a row, and two pings missed in a row.

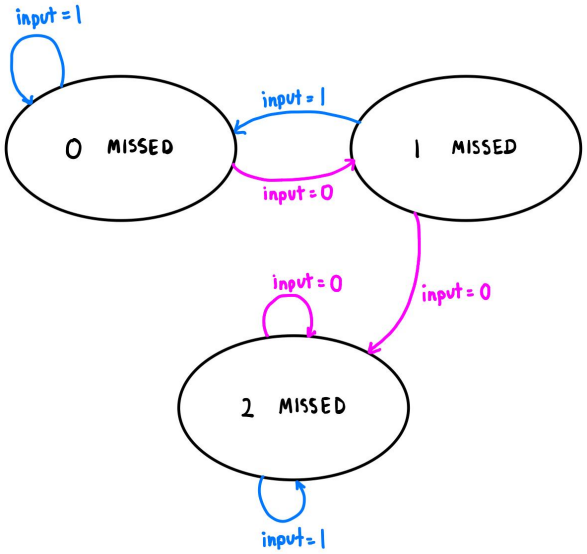


We can represent these states as shown above. Next, let's determine the transitions we need to include. If we receive a 1, then we no longer have any pings missed in a row. Thus, the "0 missed" and "1 missed" states will transition to "0 missed" upon receiving a 1. The "2 missed" state will **not** transition to "0 missed" since we ignore all further pings once 2 pings are missed. Instead, it will transition back to "2 missed".

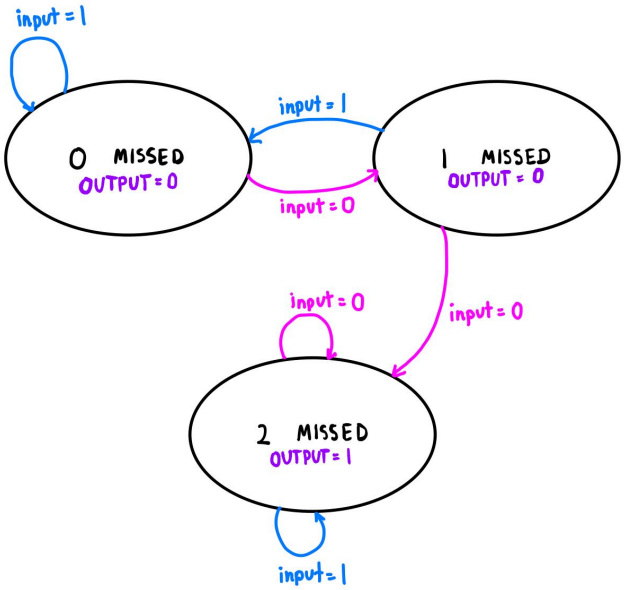


[14] From "Struct Alignment" in Chapter 5

Next, let's add transitions for when the input is 0 (a ping is missed). When a ping is missed, we will increment the number of pings missed in a row: 0 becomes 1, 1 becomes 2. Thus, our FSM is as shown below.

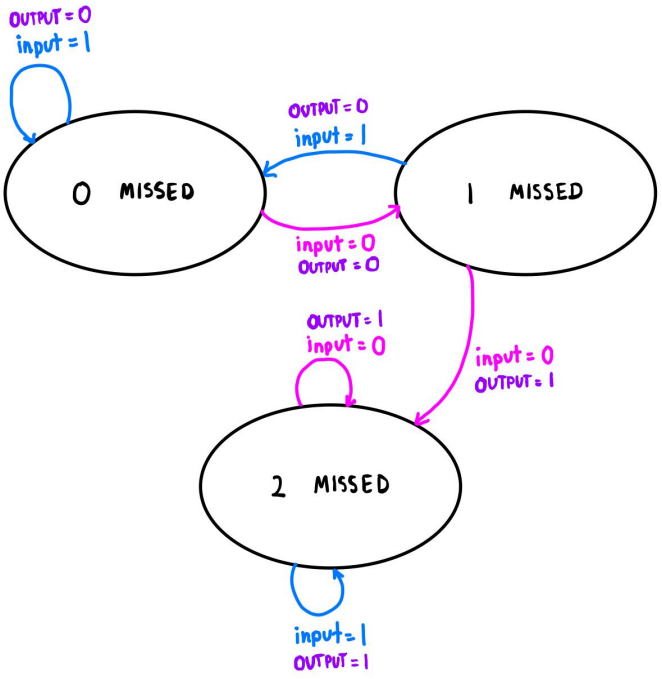


Finally, we need to determine the output of the FSM. First, let's consider a Moore machine. The output of a Moore machine is solely dependent on the state. In the "0 missed" and "1 missed" states, the output is 0. In the "2 missed" state, the output is 1. Thus, the FSM diagram would be drawn as follows:



In a Mealy machine, the output is dependent both on the state and the input. Thus, the output for a state can be different depending on if the input is 0 or 1. Let's think about how we can modify the Moore machine above to make it a Mealy machine. We need the output to

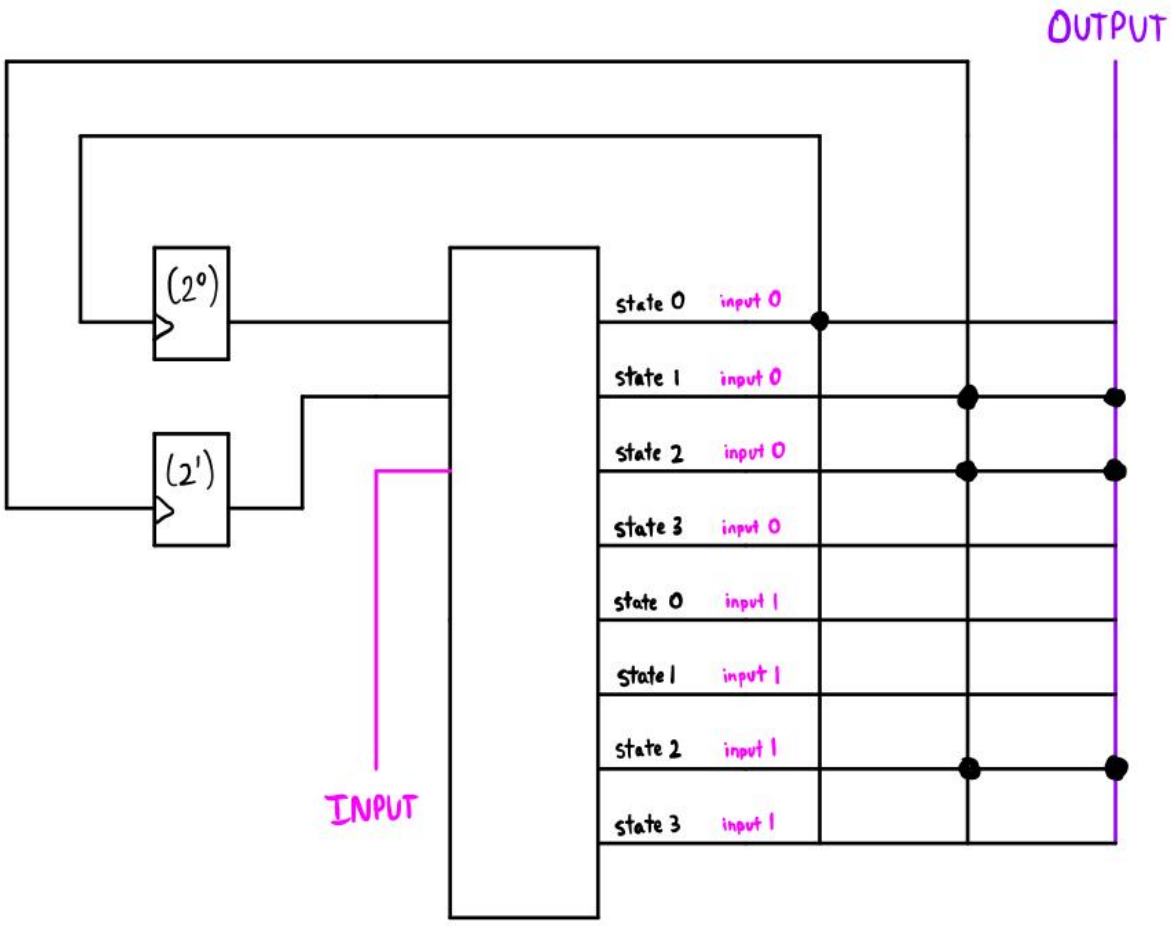
be what the resulting state's output (in the Moore machine) would be. Thus, our outputs are as follows:



FSMs in Hardware

To implement a FSM in hardware, we will again utilize a control ROM. Let's first consider implementing the Mealy machine from the previous example. First, we need to store the current state. We will assign each state an identification number in binary, and store the current state with 2 D flip flops (which allow us to address up to $2^2 = 4$ states). Next, we can use a control ROM to store our state transitions, as well as the outputs. The full control ROM is shown below. Notice that the lines labeled "state 3" are unused, since there is no state 3. States 0, 1, and 2 correspond to "0 missed", "1 missed", and "2 missed" respectively.

For a Moore machine, we can use a very similar setup, except that the connections in the "output" column of the control ROM are changed. Regardless of the input bits, the output will always be the same (for the same state).



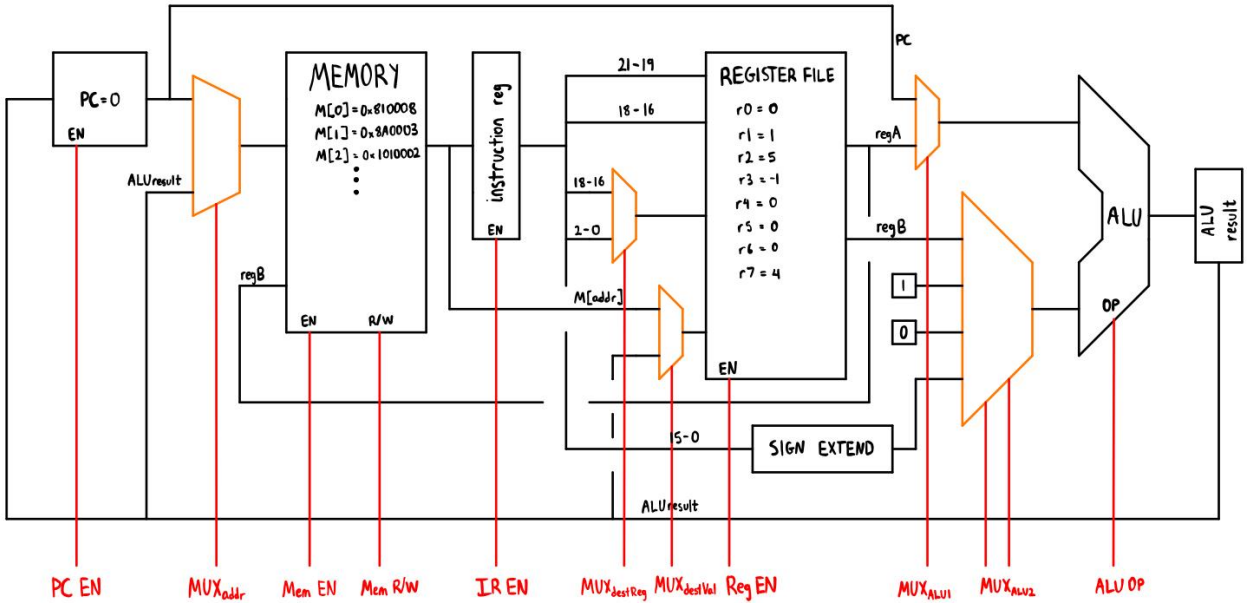
Now that we understand how to design and implement FSMs, let's design the multi-cycle processor.

The Multi-Cycle Processor

The main concept of the multi-cycle processor is to split up the operations of one instruction into multiple cycles. Let's break down what needs to be done for **add**:

- *Instruction Fetch*: Fetch instruction = M[PC], calculate PC + 1
- *Instruction Decode*: Update PC = PC + 1, fetch register values **regA**, **regB**
- *Execution*: Compute **regA + regB** in the ALU
- *Writeback*: write ALU result to **destReg**

Each of these cycles may utilize multiple components, but will have each component running in parallel. For example, fetching M[PC] doesn't depend on calculating PC + 1.



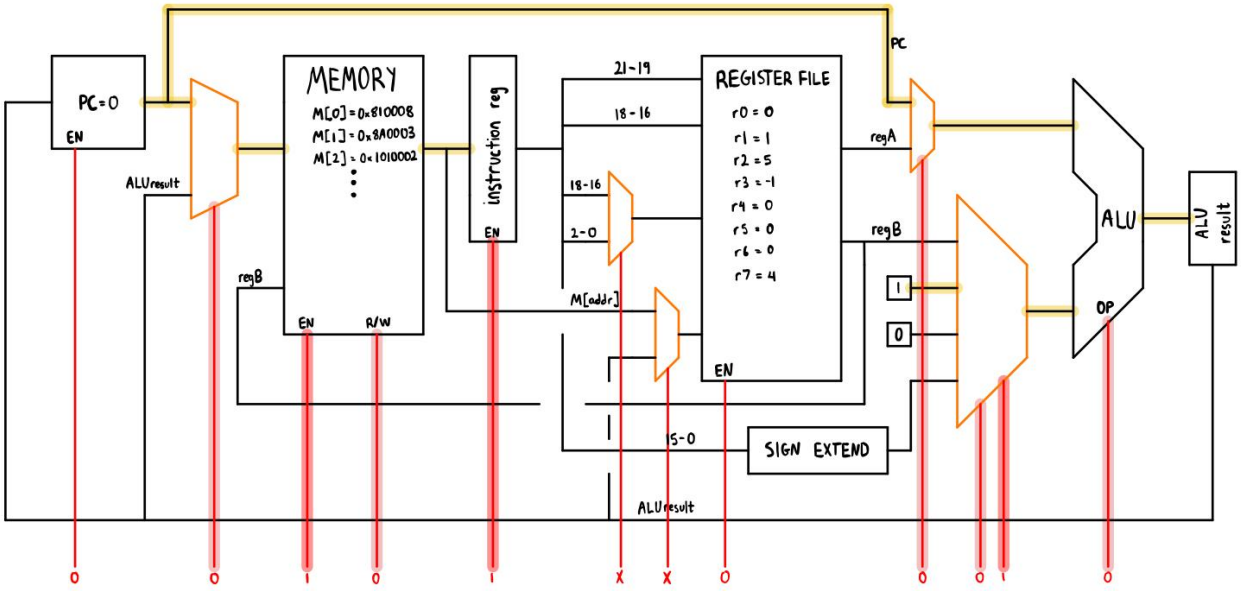
The full multi-cycle datapath is shown above. Notice that we no longer have a control ROM. Instead, the processor is controlled by a FSM, which dictates the control signals. We won't show the full ROM for the multi-cycle processor's FSM. Rather, let's take a look at how different instructions are executed on the datapath.

Instruction Fetch and Decode (Cycles 1 and 2)
All instructions begin with the same two cycles: instruction fetch and instruction decode. In the fetch cycle, we fetch the value in memory at address "PC". Additionally, we will also prepare to increment the PC by calculating PC + 1 in the ALU. If we didn't perform this computation in cycle 1 to avoid using the ALU in later cycles. Since a cycle cannot perform multiple computations using one component, we try to finish computations as soon as we have the values to perform them to avoid utilizing the ALU later on.

We'll denote the operations taking place in Cycle 1 as follows:

- IR = M[PC] (set instruction register to value in memory at address PC)
- ALUresult = PC + 1

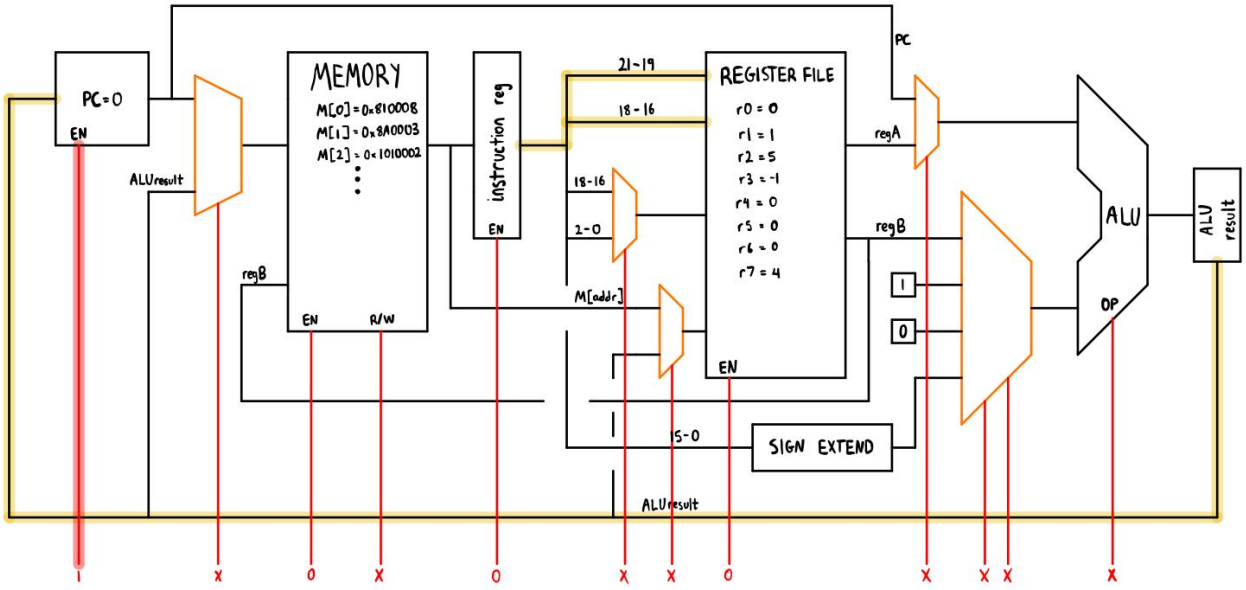
The control signals and relevant datapath components are shown below:



Next, in our second cycle, we will update our PC to the calculated PC + 1. We can't update this yet because the calculation of PC + 1 finishes at the end of the previous cycle, and the ALU result register would not be updated until the next clock cycle. The operations in the second cycle are as follows:

- PC = ALUresult
- valA = R[regA] (read the value of **regA**)
- valB = R[regB] (read the value of **regB**)

But wait! We don't have registers called **valA** or **valB**! In the cycle descriptions, we'll use **valA** and **valB** to denote the outputs from the register file. As long as we don't update the instruction register (which will only ever happen on Cycle 1), the outputs will stay consistent, and we can think of these outputs as a "pseudo-register". The operations in Cycle 2 are shown below:

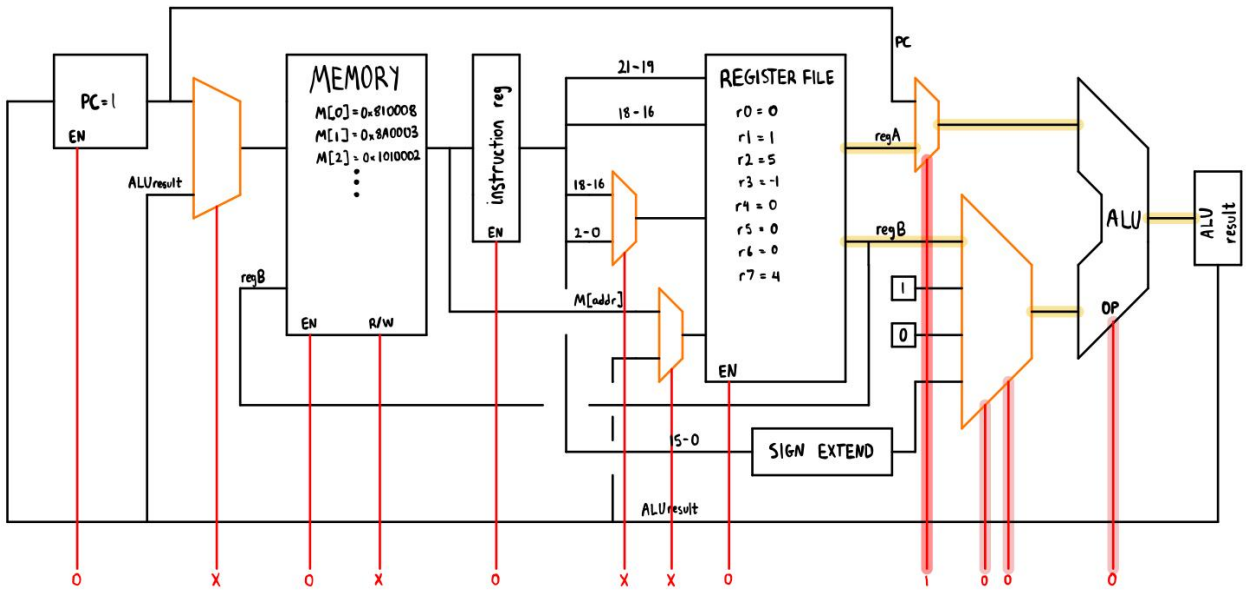


During Cycle 2, we will also extract the opcode from the instruction. From here, we will proceed forward depending on the opcode.

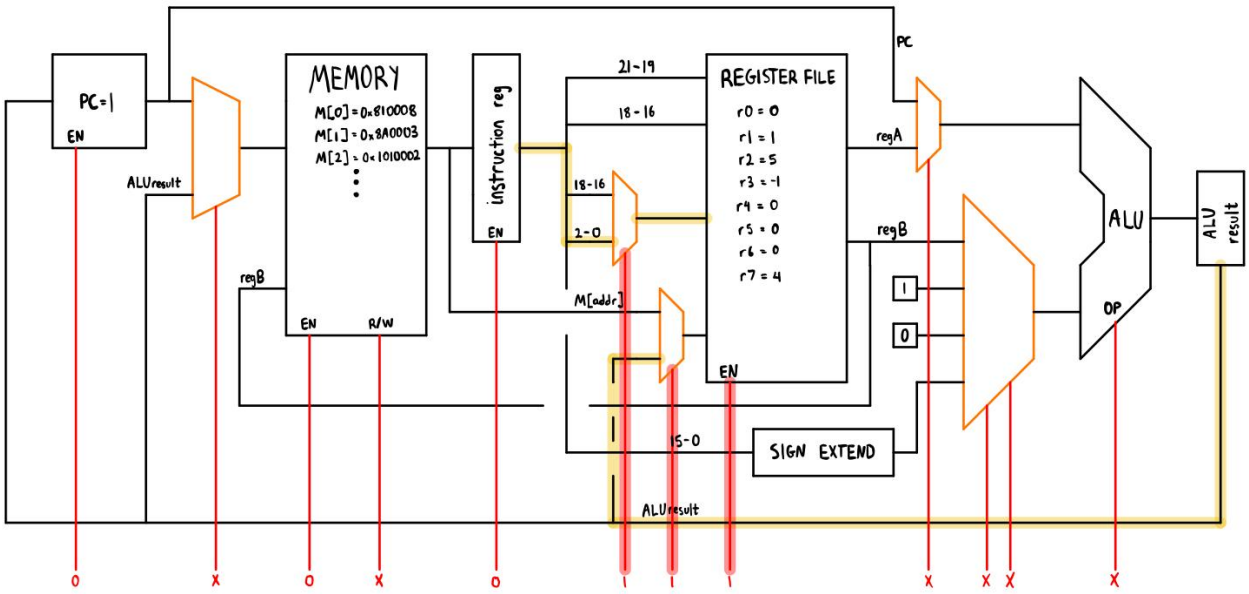
- **noop**: we've finished. Continue onto the next instruction (return to Cycle 1)
- **halt**: we've finished. Stop the processor
- **add**, **nor**, **sw**, **beq**: will run two more cycles
- **lw**: will run **three** more cycles

Instruction-Specific Cycles (3 - 5)

Let's start with **add**'s Cycle 3 and 4. In Cycle 3, we will calculate the value of **regA + regB**, and store this into ALUresult (ALUresult = valA + valB)



Next, in Cycle 4, we will write this value into **destReg** (R[destReg] = ALUresult)



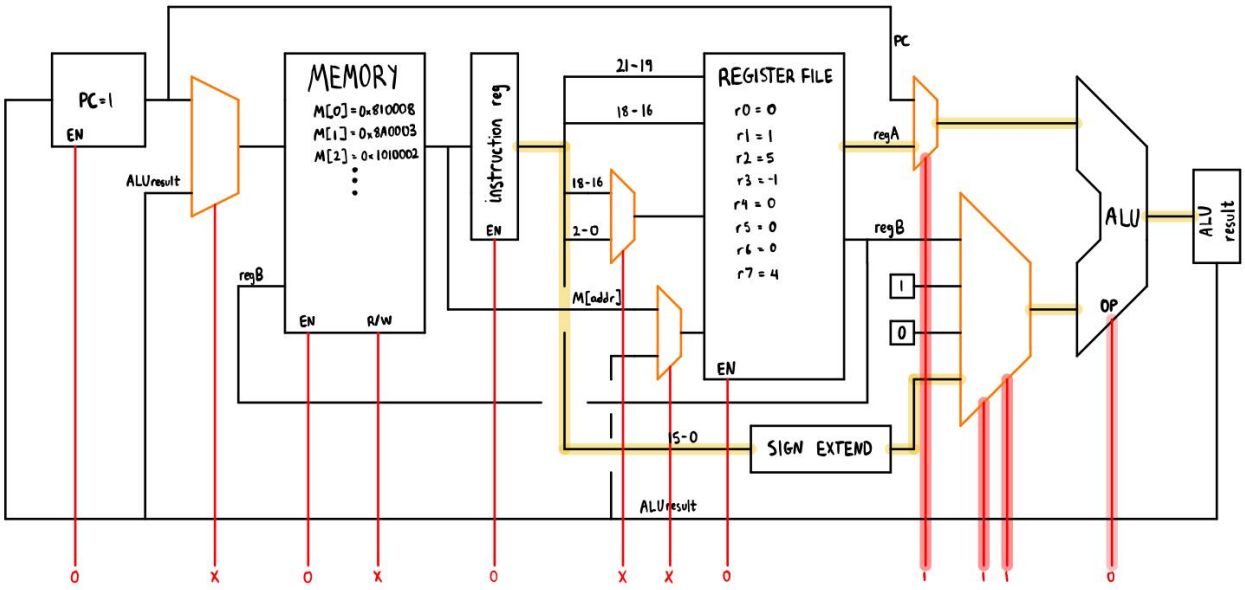
For **nor**, we will perform a very similar Cycle 3 (changing the ALU OP to 1 for bitwise NOR), and exactly the same Cycle 4. After Cycle 4, we return to Cycle 1, and we fetch the updated PC.

Next, let's look at the control signals for **lw**. **lw** requires three more cycles, which will perform the following operations:

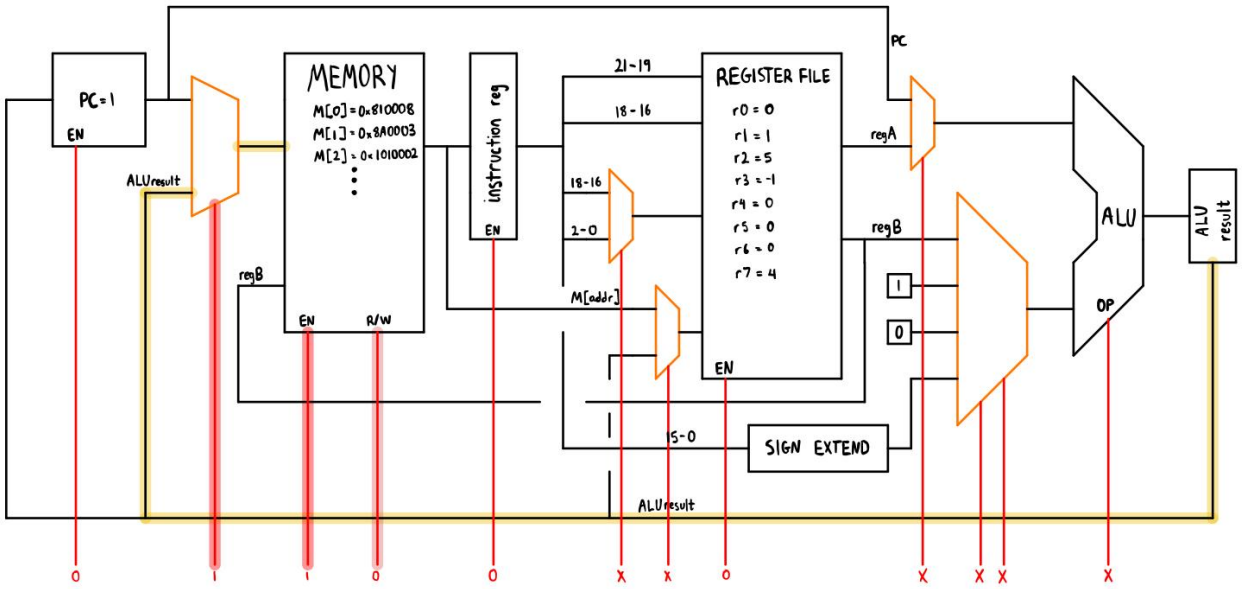
- Cycle 3:
  - ALUresult = regA + sign extended offset
- Cycle 4:
  - Read M[ALUresult]
- Cycle 5:
  - R[regB] = M[ALUresult]

Notice that we can't read M[ALUresult] and write to R[regB] in the same cycle, since this would require memory and the register file to run sequentially. We'll store the value we read from memory in a special register inside the memory file. We don't show this here, but this allows us to save the previously read value so that we can use it on the next cycle.

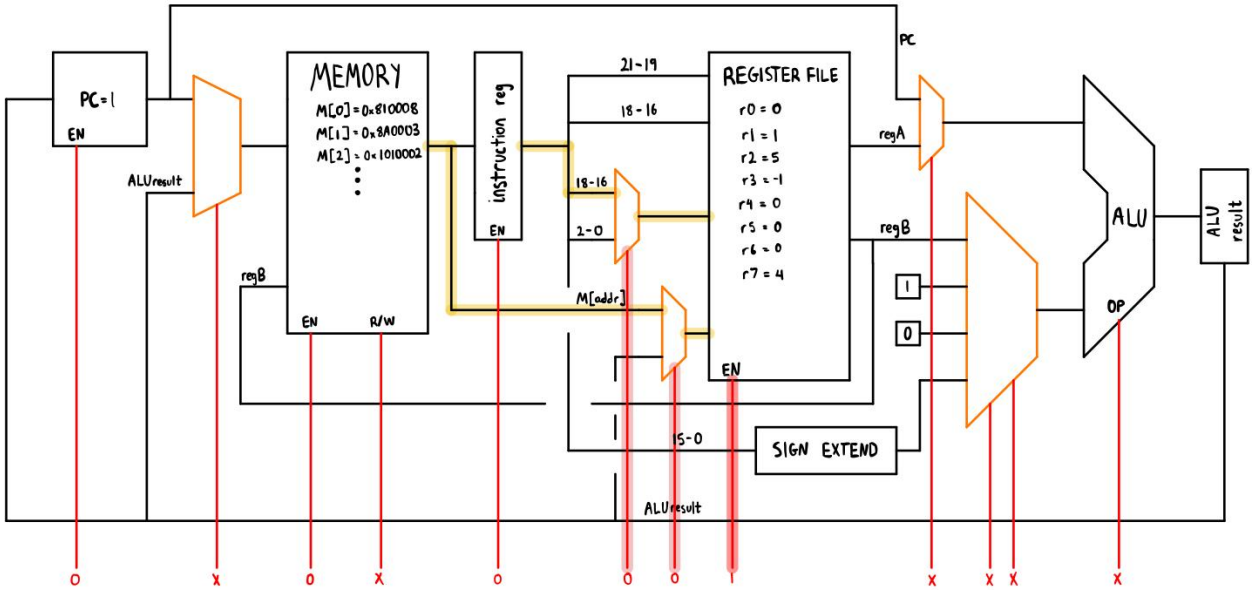
Let's start with Cycle 3. We'll perform an addition (ALU OP = 0) between regA and offset (MUX_{ALU1} = 1, MUX_{ALU2} = 11). The remaining bits are set to 0 for EN bits, or X for all other bits. Thus, the control signals are set as shown below.



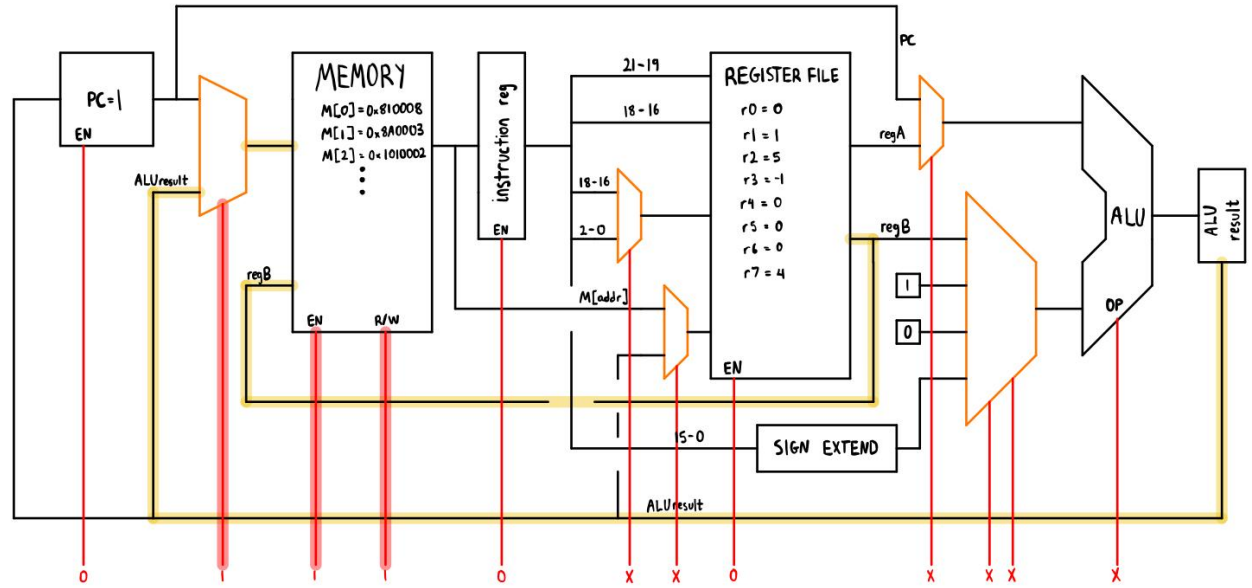
On Cycle 4, we need to read M[**regA + offset**]. We computed **regA + offset** on the previous cycle, so we will read from address ALUresult.



Finally, on Cycle 5, we will write back the read value into **regB** (bits 18-16). We set Reg EN to write to the register file, and set the destination register and value to write.

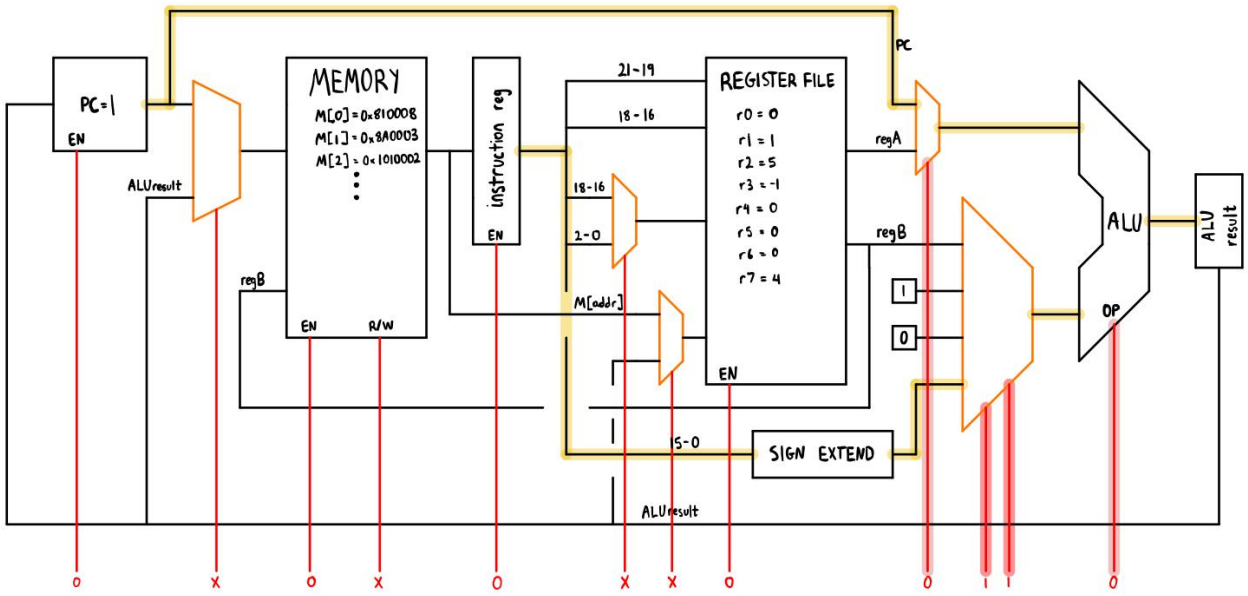


Next is **sw**. **sw** will use the same Cycle 3 as **lw**, since both instructions need to calculate **regA + offset**. After this, **sw** finishes within one cycle, since the only changes necessary are to memory. In Cycle 4, **sw** runs the operation M[ALUresult] = valB.

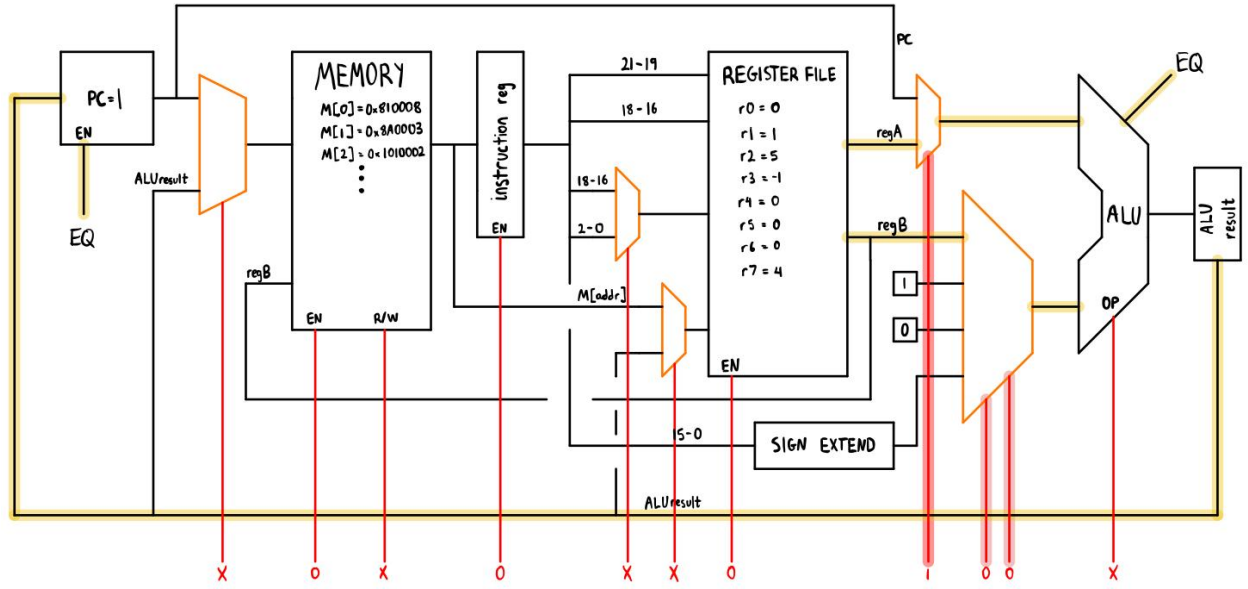


Finally, we have **beq**. **beq** will calculate the branch target on Cycle 3 (new PC + offset), and check equality and update the PC if necessary on Cycle 4.

On Cycle 3, we perform the operation ALUresult = PC + offset. We can use PC because we've already updated PC = PC + 1 in Cycle 2.



Finally, on Cycle 4, we conditionally update the PC based on the equality check in the ALU.



In a full processor, we would add conditional logic to update the PC only if the opcode was **beq** or if PC EN was set. We'll leave out this logic for clarity.

With this, we are able to execute all instructions on the multicycle pipeline! This may seem like a lot to remember; the best way to remember the different cycles is to understand how different components of each instruction are split up across cycles. From there, the control signals can be derived from the operations being performed.

Multi-Cycle Performance

It wouldn't make sense to design a new processor if it would run slower than our previous designs. Let's consider the same specifications as before, and calculate the performance of the multi-cycle processor.

Consider a multi-cycle processor with the following component latencies:

- register file read: 5 ns
- register file write: 8 ns
- ALU: 15 ns
- memory read: 25 ns
- memory write: 35 ns

Treat all components not listed above as having 0 ns latencies.

We again will run our program on the following benchmark:

- 15 **add** instructions
- 15 **nor** instructions
- 30 **lw** instructions
- 20 **sw** instructions
- 10 **beq** instructions
- 10 **noop or halt** instructions

To find the clock period, recall that in the multi-cycle processor, a cycle cannot ever have two components running sequentially. Thus, the clock period is the longest latency of any **single** component. In this case, it will be 35 ns. Next, let's look at the next two components of the Iron Law: CPI and # instructions. For a multi-cycle processor, it is typically easier to first calculate the total number of cycles executed. Recall that the number of cycles for each instruction is:

- **add**: 4 cycles
- **nor**: 4 cycles
- **lw**: 5 cycles
- **sw**: 4 cycles
- **beq**: 4 cycles
- **noop** / **halt**: 2 cycles

Thus, we can calculate the total number of cycles by adding up the number of cycles from each instruction:

4 * 15 **add** + 4 * 15 **nor** + 5 * 30 **lw** + 4 * 20 **sw** + 4 * 10 **beq** + 2 * 10 **noop / halt** = 410 cycles

Next, we multiply this by our clock period to determine the total runtime: 410 cycles * 35 ns / cycle = 14,350 ns.

Let's compare this with a single cycle processor, using the same latencies. The single cycle processor, with the same component latencies and benchmark, would take 7500 ns to run. Let's take a look at why the single cycle processor wins here. Let's compare the runtime of each instruction between the two processors:

- **add**: 75 ns on single cycle, 140 ns on multi-cycle
- **nor**: 75 ns on single cycle, 140 ns on multi-cycle
- **lw**: 75 ns on single cycle, 175 ns on multi-cycle
- **sw**: 75 ns on single cycle, 140 ns on multi-cycle
- **beq**: 75 ns on single cycle, 140 ns on multi-cycle
- **noop / halt**: 75 ns on singe cycle, 70 ns on multi-cycle

Only **noop** and **halt** are faster on the multi-cycle datapath, and only marginally so! However, let's consider a new set of latencies:

- register file read: 10 ns
- register file write: 15 ns
- ALU: 25 ns
- memory read: 25 ns
- memory write: 25 ns

The longest instruction here is **lw**, with a total latency of 25 + 10 + 25 + 25 + 15 = 100 ns. The new multi-cycle clock period is 25 ns. Thus, our instruction times are as follows:

- **add**: 100 ns on single cycle, 100 ns on multi-cycle
- **nor**: 100 ns on single cycle, 100 ns on multi-cycle
- **lw**: 100 ns on single cycle, 125 ns on multi-cycle
- **sw**: 100 ns on single cycle, 100 ns on multi-cycle
- **beq**: 100 ns on single cycle, 100 ns on multi-cycle
- **noop / halt**: 100 ns on singe cycle, 50 ns on multi-cycle

Let's consider the following benchmark:

- 15 **add**
- 15 **nor**
- 15 **lw**
- 15 **sw**
- 15 **beq**
- 25 **noop** or **halt**

The runtime for the single cycle processor is 10,000 ns, since we run 100 instructions at a cycle time of 100 ns each. On the multi-cycle processor, we run 60 * 4 + 15 * 5 + 25 * 2 = 365 cycles, at 25 ns per cycle. Thus, our total runtime is 9,125 ns on the multi-cycle processor! The key takeaway from these examples is that the multi-cycle processor can run faster or

slower than the single cycle processor, depending on the specific component latencies and benchmark used to test the processor.

Chapter 7: Pipelining

In our single cycle and multi-cycle processors, we only ever had one active instruction at a time. In the single cycle processor, we incurred lots of downtime on shorter instructions, since these instructions did not use a lot of the processor's components. In the multi-cycle processor, we solved the problem of extra downtime by splitting up each instruction across multiple cycles, but now each cycle only used one or two components at a time, leaving the rest idle. Let's take a look at what components we use for the **add** instruction on each cycle:

- Cycle 1: memory, ALU (for incrementing)
- Cycle 2: register file (read)
- Cycle 3: ALU
- Cycle 4: register file (write)

We don't use memory at all after cycle 1, so why should the next instruction wait to load from memory until the previous instruction has finished? We can make the same argument for the register file and the ALU. Let's think about how we could potentially run multiple instructions in parallel. Let's consider the following, very simple program.

```
            lw          0     1     five
            nor         2     3     4
            halt
five        .fill       5
```

On the single cycle processor, this program would execute as follows:

- Cycle 1: fetch **lw 0 1 five** from memory, read registers 0 and 1, calculate **regA + offset**, read memory at address 3, store **5** to register 1
- Cycle 2: fetch **nor 2 3 4** from memory, read registers 2 and 3, calculate **regA NOR regB**, store **-1** to register 4
- Cycle 3: fetch **halt** from memory, stop the processor

On the multi-cycle processor, this program would execute as follows:

- Cycle 1: fetch **lw 0 1 five** from memory
- Cycle 2: read registers 0 and 1
- Cycle 3: calculate **regA + offset**
- Cycle 4: read memory at address 3
- Cycle 5: store **5** to register 1
- Cycle 6: fetch **nor 2 3 4** from memory
- Cycle 7: read registers 2 and 3
- Cycle 8: calculate **regA NOR regB**
- Cycle 9: store **-1** to register 4

- Cycle 10: fetch **halt** from memory
- Cycle 11: stop the processor

If we tried to run the instructions as follows:

- Cycle 1: fetch **lw 0 1 five** from memory; fetch **nor 2 3 4** from memory; fetch **halt** from memory
- Cycle 2: read registers 0 and 1; read registers 2 and 3; (stop the processor?)
- Cycle 3: calculate **regA + offset**; calculate **regA NOR regB**
- Cycle 4: read memory at address 3
- Cycle 5: store **5** to register 1, store **-1** to register 4

we would have quite a few problems. First, we only have a single ALU, so we can't run all these calculations in parallel. SImilarly, we can't read an arbitrary number of instructions from memory during one cycle; memory is single ported! However, if we slightly stagger the execution of each instruction, then we end up with the following:

- Cycle 1: fetch **lw 0 1 five** from memory
- Cycle 2: read registers 0 and 1; fetch **nor 2 3 4** from memory
- Cycle 3: calculate **regA + offset**; read registers 2 and 3; fetch **halt**
- Cycle 4: read memory at address 3; calculate **regA NOR regB**; do nothing
- Cycle 5: store **5** to register 1; do nothing
- Cycle 6: store **-1** to register 4; stop the processor

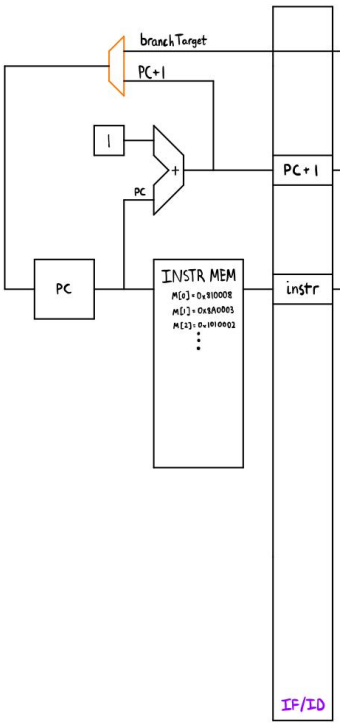
These cycles closely mirror the multi-cycle processor's cycles, but we're able to run them in parallel. Each instruction is broken up into five stages:

- Instruction fetch: fetch instruction from memory
- Instruction decode: read registers associated with the instruction
- Execution: use ALU to calculate a value, such as **regA + offset** or **regA NOR regB**
- Memory: read memory at address **regA + offset** (or don't do anything if not **lw** / **sw**)
- Writeback: store a value into **destReg** or **regB**

We'll represent these stages as IF, ID, EX, MEM, and WB. If we go through the cycles again, we can look at which stage is running at each time:

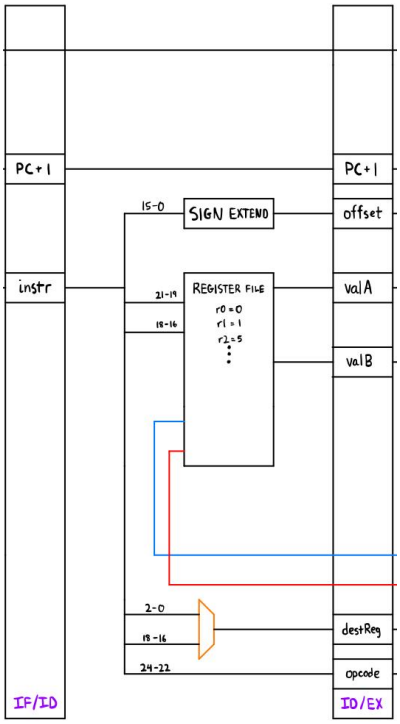
- Cycle 1: **lw 0 1 5** IF
- Cycle 2: **lw 0 1 5** ID, **nor 2 3 4** IF
- Cycle 3: **lw 0 1 5** EX, **nor 2 3 4** ID, **halt** IF
- Cycle 4: **lw 0 1 5** MEM, **nor 2 3 4** EX, **halt** ID
- Cycle 5: **lw 0 1 5** WB, **nor 2 3 4** MEM, **halt** EX
- Cycle 6: **nor 2 3 4** WB, **halt** MEM (stop processor)

Each stage corresponds to one component in the datapath: IF uses instruction memory, ID uses the register file, EX uses the ALU, and MEM uses data memory. Thus, while one instruction is in MEM, a second instruction can be in EX, a third can be in ID, and a fourth can be in IF without any instructions sharing the same components! To pass data between each of these stages, we will use **pipeline registers**. We will have one pipeline register between each stage, giving us four total pipeline registers. We'll call these IF/ID, ID/EX, EX/MEM, and MEM/WB. Let's start with the IF stage. Here, we will reuse the logic from the single cycle processor, computing PC + 1 and fetching the instruction from address PC.

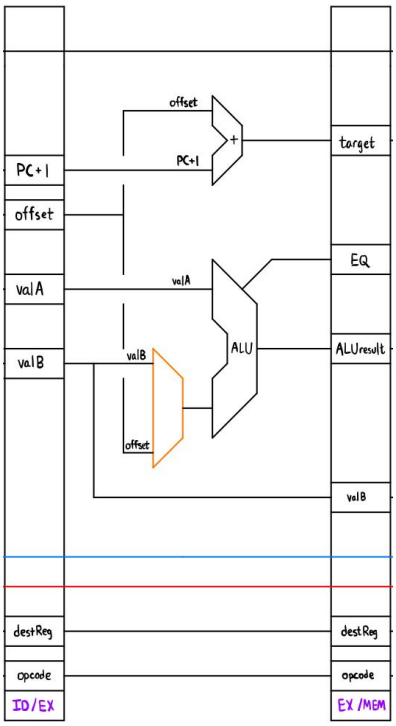


One of the key ideas behind pipelining is that each stage will be processing a different instruction at any given time. Thus, we need to store *all* information about an instruction that we will potentially need in later stages. For example, we will need PC + 1 for **beq** in the EX stage, so we have to pass it forward through the IF/ID and ID/EX registers to ensure that the EX stage can use the PC + 1 value corresponding to the **beq** instruction. If we simply connect a wire between the IF and EX stages to fetch PC + 1, this will get us the PC + 1 value for a later instruction.

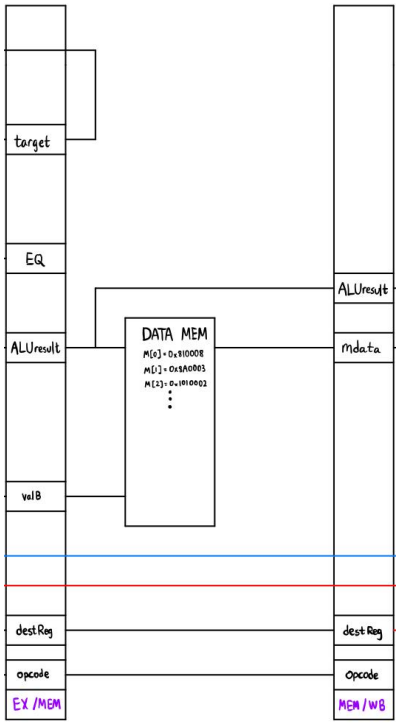
Next, in the decode stage, we will split up the instruction into its corresponding bits and read the register values associated with the instruction. Additionally, we will sign extend the offset, and determine which register we are writing to.



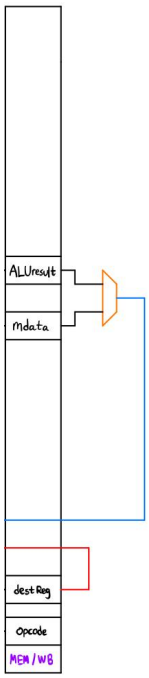
Next is the execute stage. Here, we will perform computations with the ALU: calculating the result for **add** and **nor**, calculating the desired address for **lw** and **sw**, or checking equality for **beq**. We'll also have a second adder to calculate the branch target for **beq**.



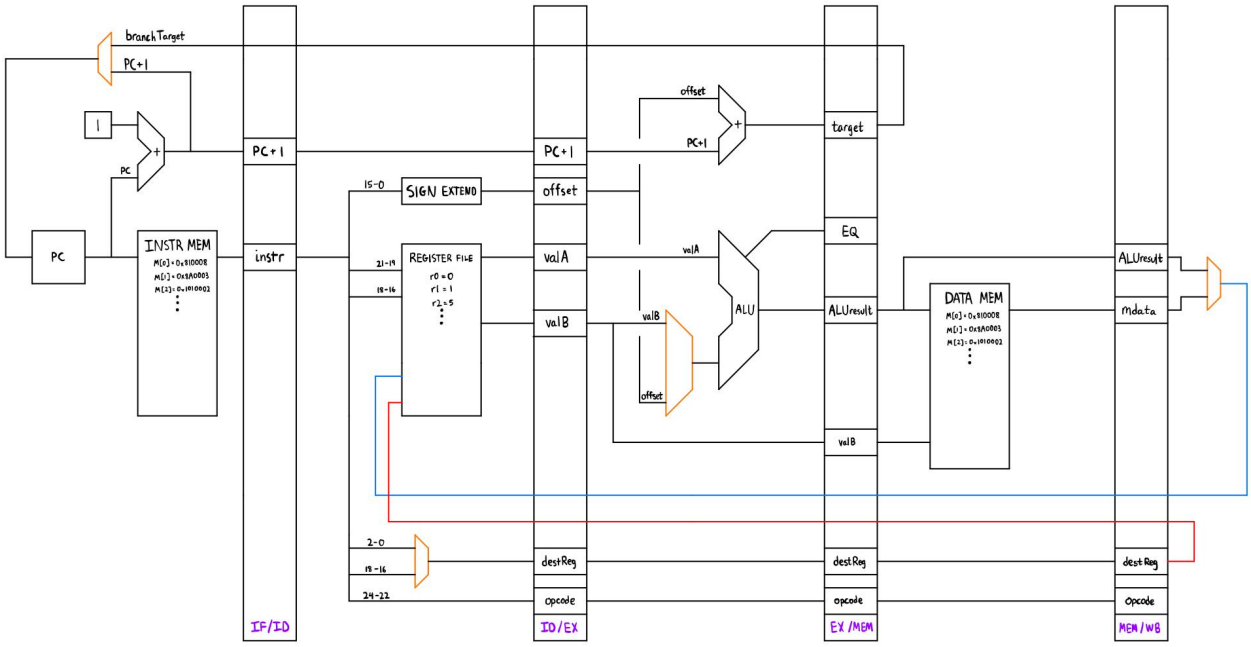
After this, we have the memory stage, where we access the contents of memory for **lw** and **sw**. We also keep track of the result from the ALU, passing it to the writeback stage.



Finally, we have the writeback stage. Here, we select the result that we want to write to the register file. This will either be the ALU result or the result of reading data memory.



Putting this all together, we get the full pipeline datapath:

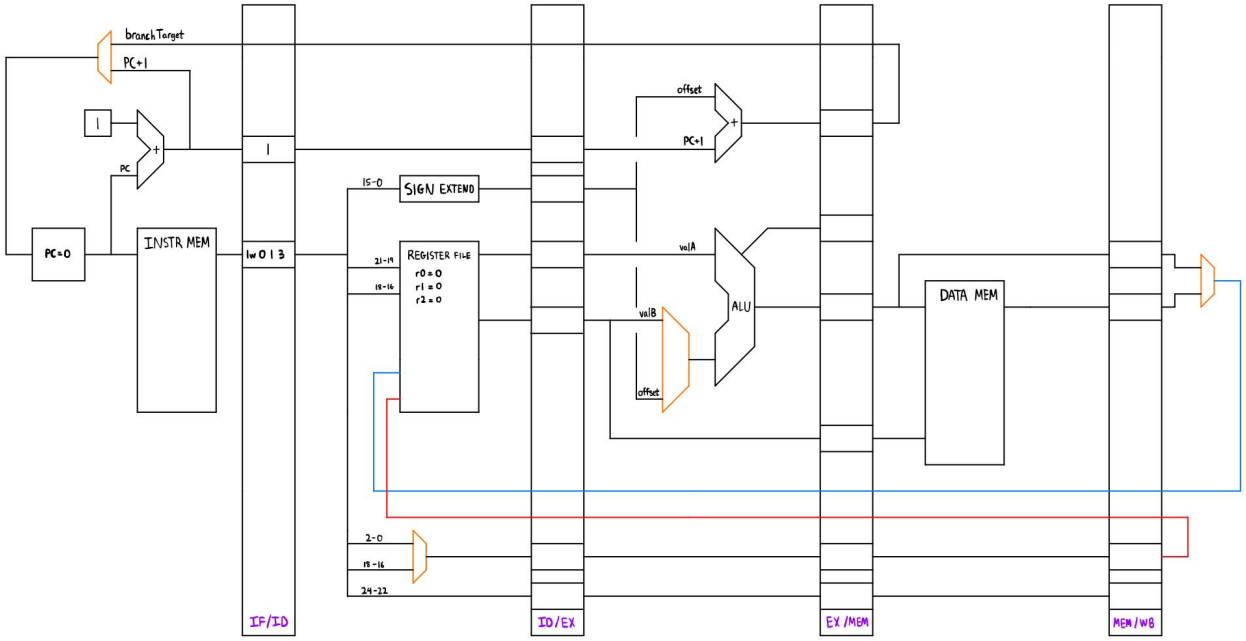


Data Hazards

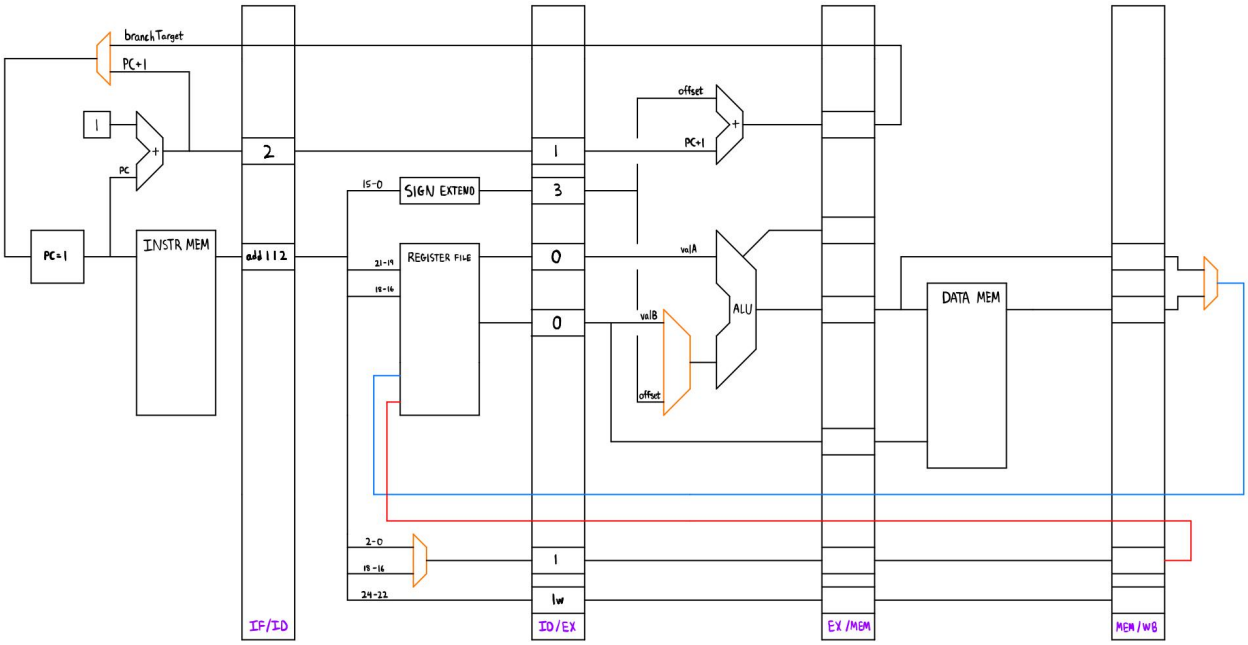
Let's take a look at a simple example of a program running on the pipeline datapath.

```
            lw          0     1     five
            add         1     1     2
            halt
five        .fill       5
```

This program loads the value 5 into register 1, then doubles the value of register 1 and stores it into register 2. Thus, we should expect that the value of register 2 is 10 at the end of the program.

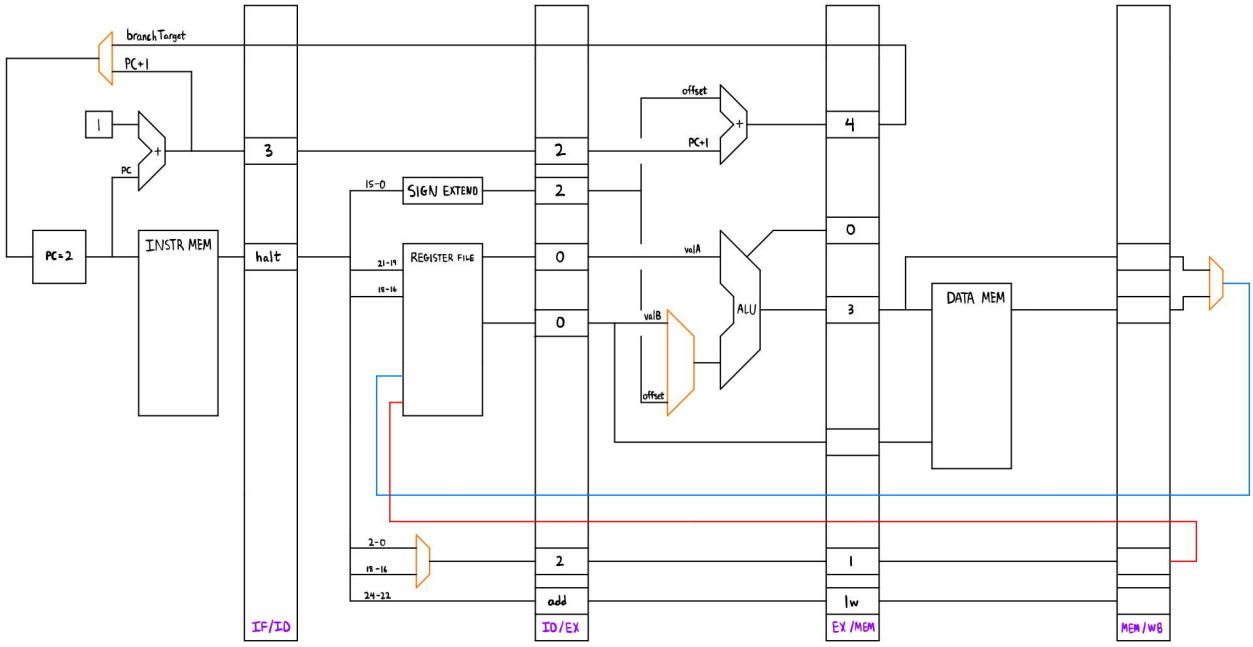


On the first cycle (shown above), we load in **lw 0 1 five** into the IF/ID pipeline register, along with its associated PC + 1 value. The remainder of the pipeline registers are initialized with **noop** instructions, ensuring that the initial state of the pipeline registers does not affect the behavior of the program. These pipeline registers are left blank for clarity.



On the second cycle, we read registers 0 and 1 from memory in the ID stage, and store these values into the ID/EX pipeline register. We extract the offset and sign extend it (3), and pass

along the value of PC + 1. In the IF stage, we fetch **add 1 1 2** from memory, and store it in the IF/ID register.



On the third cycle, we calculate **regA + offset** in the EX stage, and store this value into the EX/MEM register. In the ID stage, we read the value of register 1, which is... 0. Since we're running instructions in parallel, **lw** hasn't reached the writeback stage when **add** is reading register values, and therefore the values in the register file haven't been updated. This is what we call a **data hazard**. Specifically, we will term this type of hazard a **read after write**, or **RAW** hazard. A RAW hazard occurs when an instruction writes to a register, and a later instruction reads from the same register before the update to the register is complete. How can we fix this?

Resolving Data Hazards

To resolve data hazards in the pipeline, we have one of three methods: avoidance, detect and stall, or detect and forward. Let's start with the simplest method: avoidance. In avoidance, we leave it to the *programmer* to avoid data hazards. We can do so by inserting **noop** instructions between pairs of instructions that cause RAW hazards. Let's consider the following program:

```
            lw          0     1     five
            noop
            noop
            add         1     1     2
            halt
five        .fill       5
```

There's no change to the behavior of this program, except for the addition of two **noop** instructions, which will increase the total runtime. To analyze a pipeline processor, we will often use **pipe traces**, which allow us to track the state of the processor across multiple cycles in a tabular form. For example, here's the pipe trace of the previous program:

Cycle #	1	2	3	4	5	6	7	8
lw 0 1 five	IF	ID	EX	MEM	WB			
noop		IF	ID	EX	MEM	WB		
noop			IF	ID	EX	MEM	WB	
add 1 1 2				IF	ID	EX	MEM	WB
halt					IF	ID	EX	MEM

This pipe trace indicates to us that in cycle 1, the IF stage is fetching **lw 0 1 five** from memory. In cycle 2, the ID stage reads registers 0 and 1 (for **lw 0 1 five**), while the IF stage fetches the first **noop** instruction. Here, we're able to clearly visualize what happens in the pipeline without having to draw out each stage. Notice that the ID stage of **add 1 1 2** takes place during cycle 5, which is when the WB stage of **lw 0 1 five** takes place. Thus, the updated value is being stored into the register file during cycle 5, and we're able to use this updated value in the ID stage. This takes place through **internal forwarding**, where reading a register that is being written to will give us the updated value[15].

There's a few drawbacks to avoidance. For example, if we modify our pipeline design to add or remove hazards, we'll have to change our code. This means that we might have different version of code for different processors, and new updates have to be created whenever a new processor comes out. Instead of leaving it to the programmer to avoid data hazards, let's have the processor avoid data hazards for us.

We'll modify our pipeline datapath to detect RAW hazards. Upon reaching a RAW hazard, the pipeline will insert **noop** instructions at the ID stage, stalling the upcoming instruction in the IF/ID register until the previous instruction's WB stage has finished. Let's think about when we need to stall the pipeline. Let's consider the original program with a RAW hazard (without **noop**s inserted)

```
            lw          0     1     five
            add         1     1     2
            halt
five        .fill       5
```

[15] If we don't have internal forwarding in the pipeline, we need to insert one *additional* **noop** instruction, bringing the total up to 3 **noop** instructions between dependent instructions.

If we draw a pipe trace of this program, we get the following:

Cycle #	1	2	3	4	5	6	7	8
lw 0 1 five	IF	ID	EX	MEM	**WB**			
add 1 1 2		IF	**ID**	EX	MEM	WB		
halt			IF	ID	EX	MEM		

Notice that the WB stage of **lw 0 1 five** takes place *after* the ID stage of **add 1 1 2**, meaning that the register file wont be updated at this point. To ensure that **add 1 1 2** can use the correct register values, we need to stall by 2 cycles:

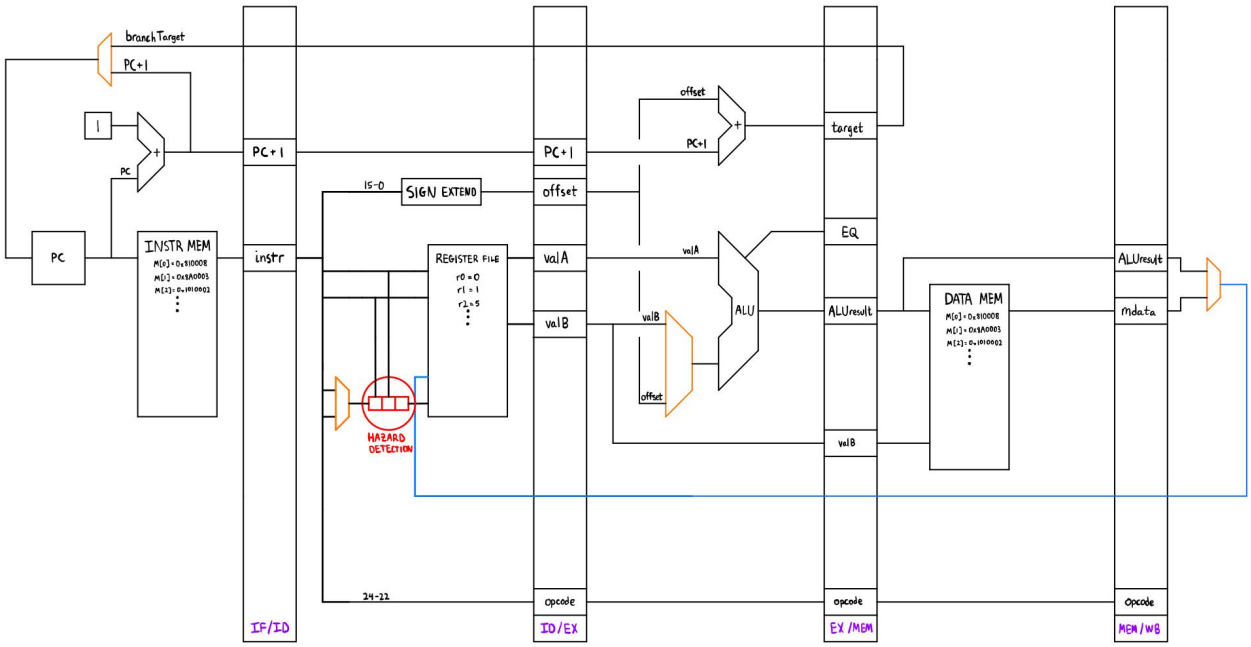
Cycle #	1	2	3	4	5	6	7	8
lw 0 1 five	IF	ID	EX	MEM	WB			
add 1 1 2		IF	ID*	ID*	ID	EX	MEM	WB
halt			IF*	IF*	IF	ID	EX	MEM

Here, we've used ID* to denote that the **add** instruction is stalled in the ID stage, and cannot proceed due to **noop**s being inserted in the ID stage. IF* indicates that the **halt** instruction is stalled in IF, due to **add** being stalled in ID. If we consider the inserted **noop** instructions in the pipe trace, we see the following:

Cycle #	1	2	3	4	5	6	7	8
lw 0 1 five	IF	ID	EX	MEM	WB			
(noop)			ID	EX	MEM	WB		
(noop)				ID	EX	MEM	WB	
add 1 1 2		IF	ID*	ID*	ID	EX	MEM	WB
halt			IF*	IF*	IF	ID	EX	MEM

Notice that we don't have an IF stage for the inserted noops since they were inserted directly into the ID stage by our detect-and-stall logic. These pipe traces give us a key piece of information we need for designing our updated datapath: we need to continue stalling as long as a dependent instruction is in the **EX** or **MEM** stages. To detect a dependency, we can compare the register being written to (either **regB** or **destReg**) to the registers being used by the instruction (**regA** and potentially **regB**). We'll keep track of the register being written to using a **shift register**. A shift register can be thought of similar to a queue, where on each

clock cycle, a new value is stored, the oldest value is discarded, and all intermediate values move "forward" one position. The shift register will keep track of the register being written to by the instructions in the EX, MEM, and WB stages. This gives us the additional benefit of not having to store the destReg field throughout the pipeline, instead using the value stored in the shift register. The updated pipeline datapath is shown below.



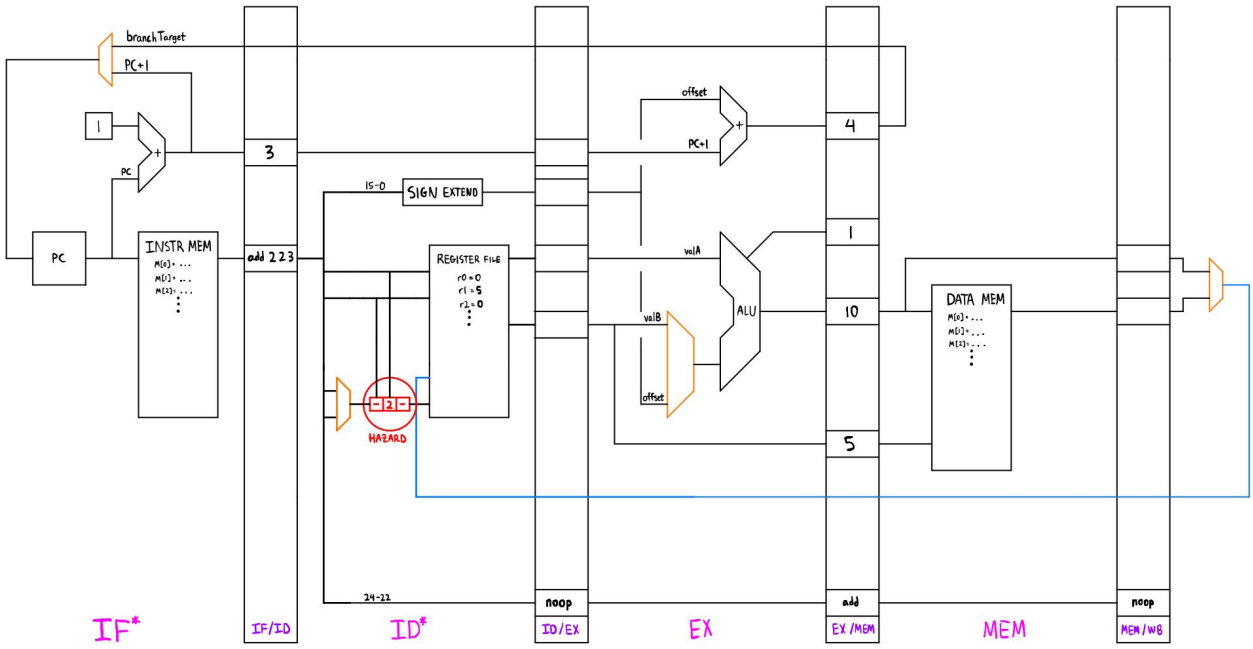
Can we optimize this further? Let's take a look at another example program:

```
            lw          0       1       five
            add         1       1       2
            add         2       2       3
            halt
five        .fill       5
```

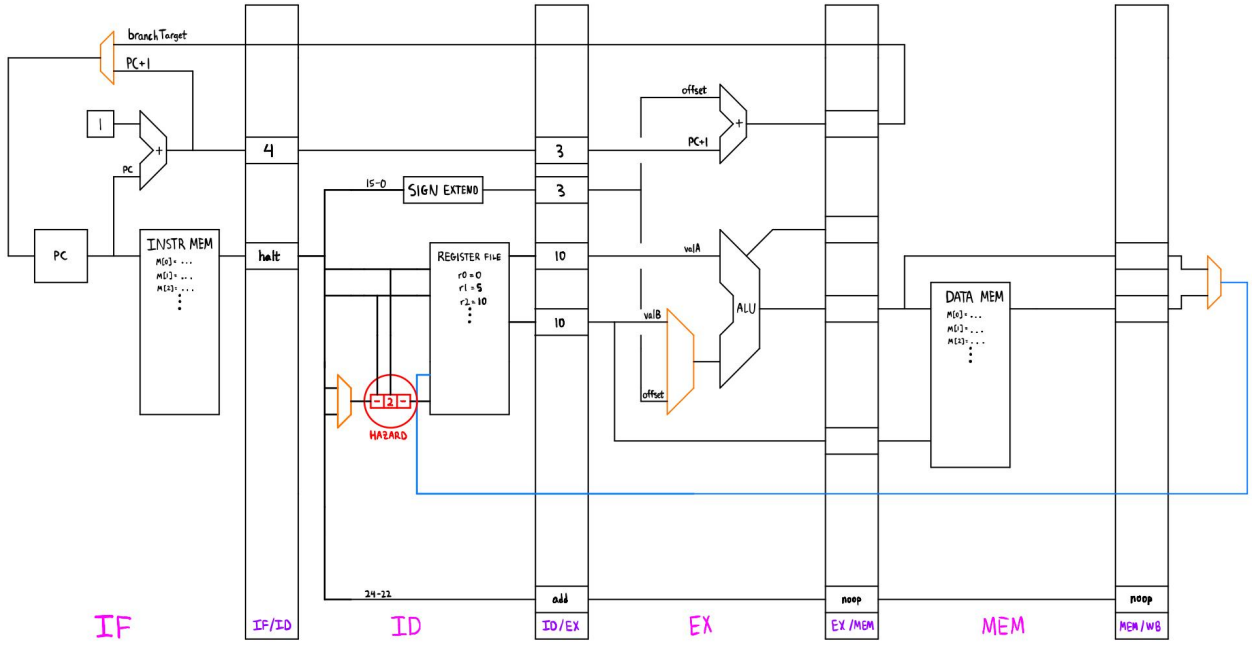
The first ten cycles of the pipe trace when using detect-and-stall is shown below:

Cycle	1	2	3	4	5	6	7	8	9	10
lw 0 1 five	IF	ID	EX	MEM	WB					
add 1 1 2		IF	ID*	ID*	ID	EX	MEM	WB		
add 2 2 3			IF*	IF*	IF	ID*	ID*	ID	EX	MEM
halt						IF*	IF*	IF	ID	EX

Let's take a look at the state of the pipeline before cycle 7 starts. **add 1 1 2** is in the EX/MEM register, and about to proceed to the memory stage. **add 2 2 3** is in the IF/ID register, but is stalled due to inserted **noop** instructions. Fetching **halt** is stalled due to **add 2 2 3** being stalled.



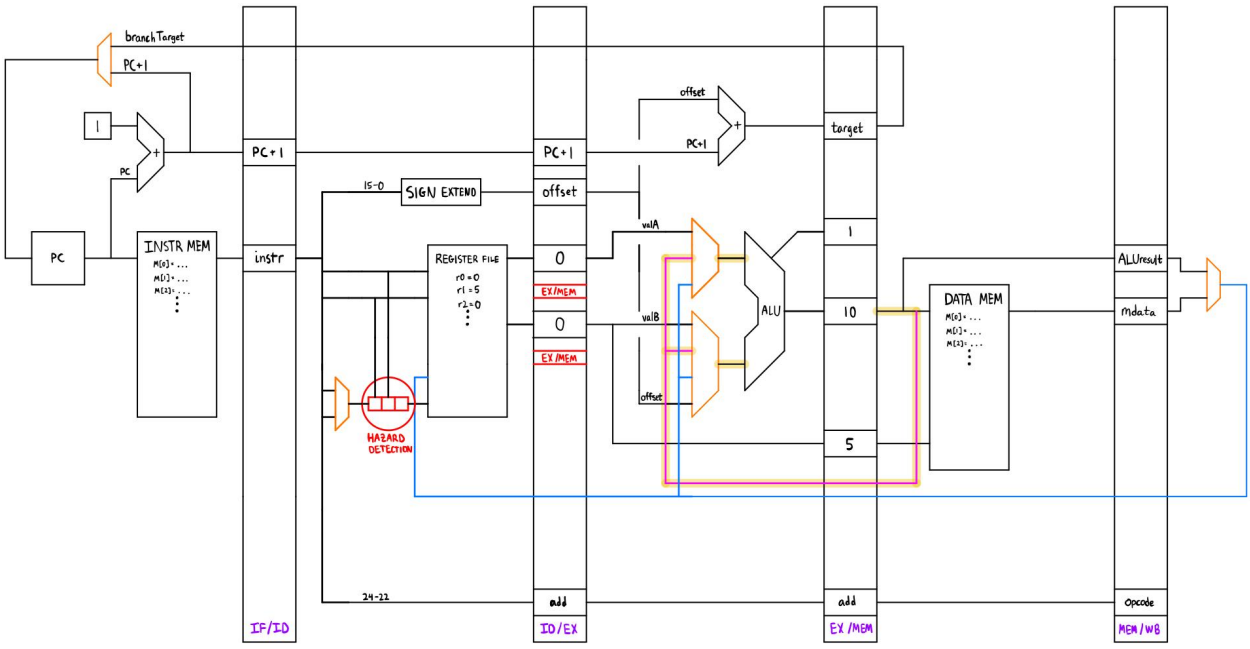
Notice that we already have computed the value that **add 2 2 3** is waiting on: 10! Let's think about where this value will be used. Let's take a look at the state of the pipeline before cycle 9 starts.



Notice that while we wait for 10 to be written into the register file, we don't do anything meaningful with it until the EX stage. What if we put **add 2 2 3** in the EX stage during cycle 7, and we gave it the "correct" value of register 2? This is the concept behind **detect and forward**. To perform forwarding, we'll add wires from the MEM stage and the WB stage to **forward** the correct values of registers to the EX stage. We'll keep our hazard detection logic in the ID stage, but rather than inserting **noop** instructions, we'll use this to "notify" the EX stage that it should forward a register value. Let's take a look at the following program:

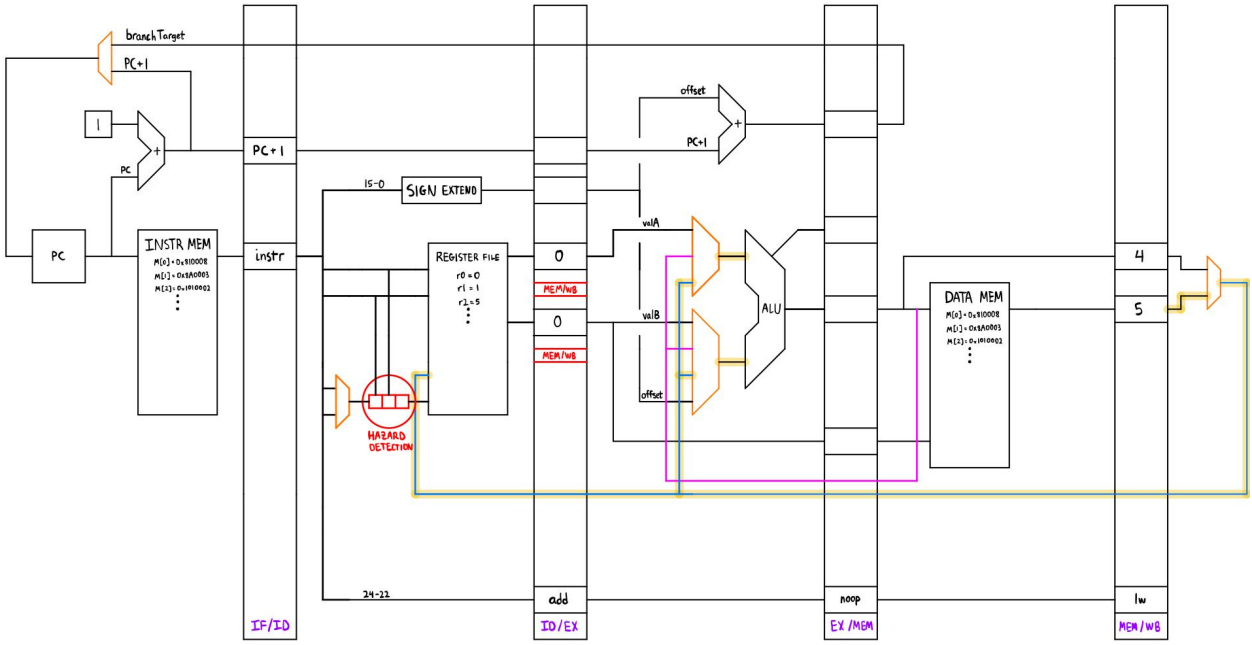
```
            lw          0     1     five
            add         1     1     2
            add         2     2     3
            halt
five        .fill       5
```

When the first **add** reaches the EX stage, it will compute 5 + 5 = 10. This value will be stored into the EX/MEM register. At the same time, the second **add** will reach the ID stage, and it will fetch the "incorrect" value of 0 for register 2. However, the ID stage will recognize the RAW hazard between **add 1 1 2** and **add 2 2 3**, and flag both **regA** and **regB** for forwarding from the EX/MEM register. The state of the pipeline is thus as follows (only the two **adds** are shown for clarity)



On the next cycle, the EX stage will execute **add 2 2 3**. Rather than using the value of 0, we have added in MUXes that allow us to select the forwarded value of 10 from the EX/MEM register. Thus, we can get the correct behavior without **any stalls** between the two **add** instructions! Next, let's consider the dependency between the **lw** and the first **add**. The first point we will have the "correct" value of register 1 is after the MEM stage of **lw**. Thus, **add 1 1**

2 will be in the ID/EX register when **lw 0 1 five** is in the MEM/WB register. Due to this, we must still stall between a **lw** and an immediate dependency. Once **lw** finishes the MEM stage, we can forward the value from MEM/WB directly to the EX stage.



The pipe trace for this program would therefore be as follows:

Cycle	1	2	3	4	5	6	7	8
lw 0 1 five	IF	ID	EX	MEM	WB			
add 1 1 2		IF	ID*	ID	EX	MEM	WB	
add 2 2 3			IF*	IF	ID	EX	MEM	WB
halt					IF	ID	EX	MEM

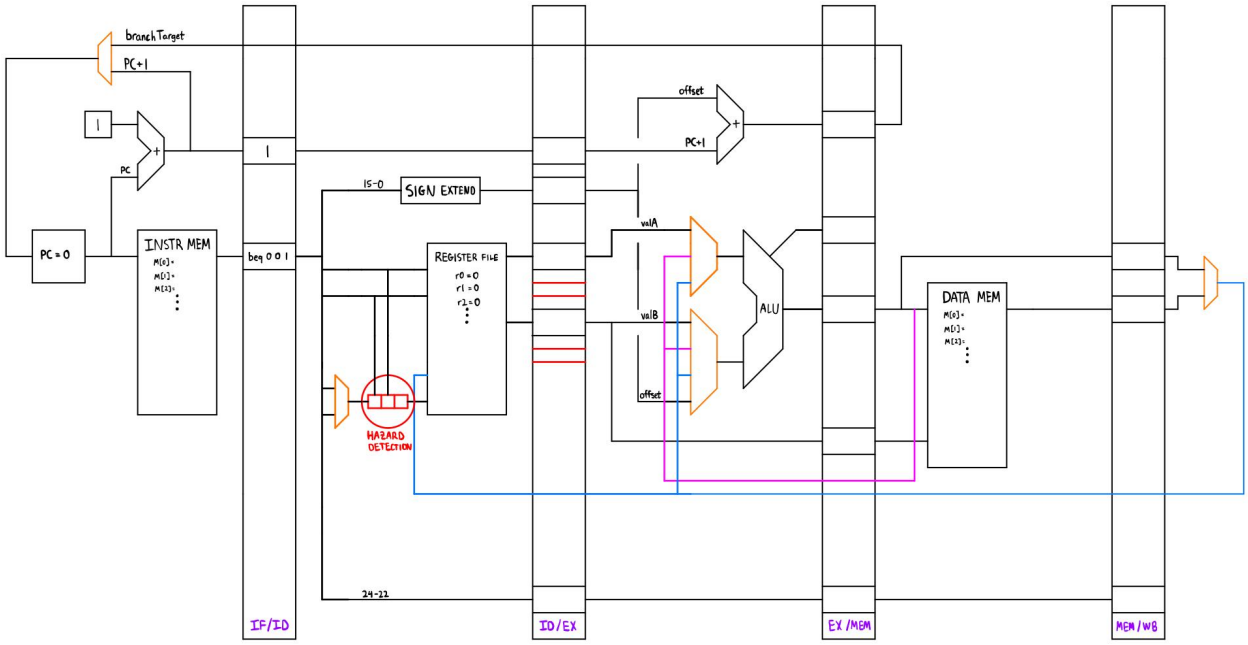
We no longer have to stall for most instructions, since we can take advantage of forwarding from MEM/WB and EX/MEM.

Control Hazards

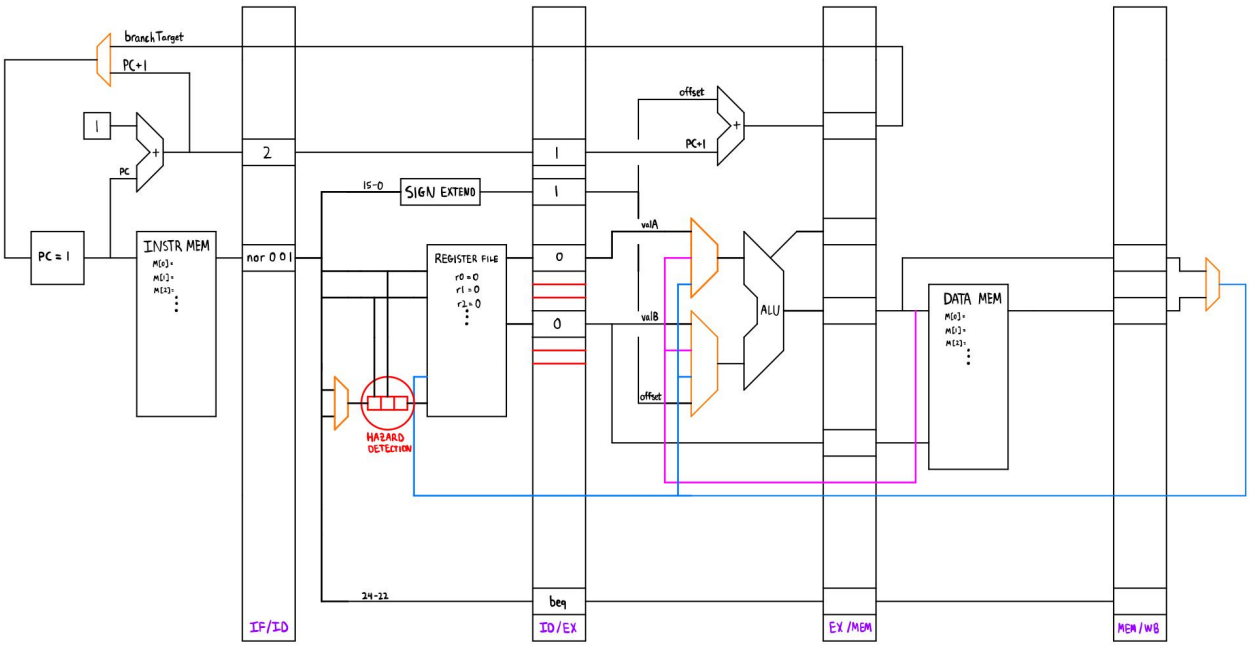
Now that we know how to resolve data hazards, let's handle one more hazard: control hazards. While data hazards arise from instructions being run before the register values have a chance to update, control hazards arise from instructions being loaded into the pipeline before we are sure they will run. This scenario arises when we have a branch, such as **beq**. Let's take a look at the following program:

```
            beq         0     0     label
            nor         0     0     1
label       add         1     1     1
            halt
```

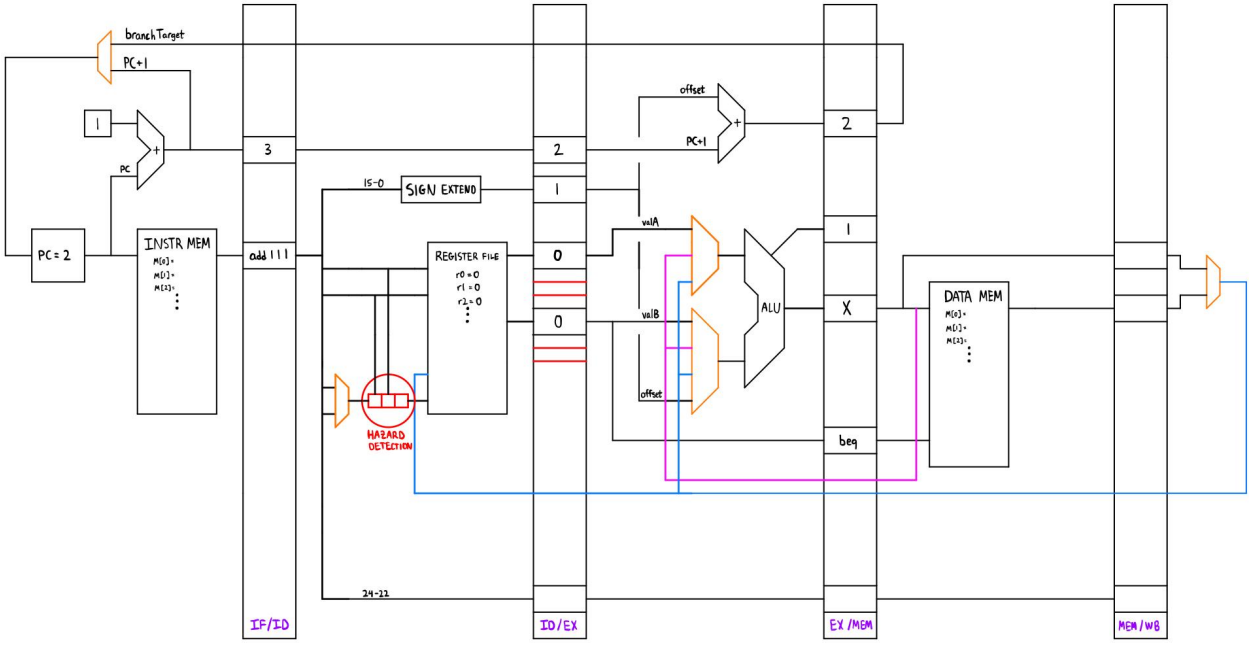
Let's take a look at how our current pipeline would handle this program. On cycle 1, we load in **beq 0 0 label** in the IF stage.



On the next cycle, **beq** moves on to the ID stage, and **nor 0 0 1** is fetched.



On the following cycle, **beq** moves on to the EX stage, and **nor** moves on to the ID stage. **add 1 1 1** is fetched.



On the next cycle, we will update the PC to 2, meaning that we've performed the branch successfully! But wait, **nor 0 0 1** shouldn't be in the pipeline! We have no way to stop **nor** from continuing through the MEM and WB stages, and therefore writing back! This creates a **control hazard**.

Resolving Control Hazards

To resolve control hazards, we have three options: avoidance, detect and stall, or speculate and squash. Let's start with avoidance. Avoidance is very similar to avoidance in data hazards. We leave the responsibility to the programmer to insert **noop** instructions into their code to ensure that no harmful side effects occur. Consider the following program:

```
            beq         0     0     label
            nor         0     0     1
label       halt
```

With avoidance, we will modify the program and add **three noop** instructions after the **beq**:

```
            beq         0     0     label
            noop
            noop
            noop
            nor         0     0     1
label       halt
```

The first six cycles of the pipe trace are as follows[16]:

Cycle	1	2	3	4	5	6
beq 0 0 label	IF	ID	EX	MEM **(set PC)**	WB	
noop		IF	ID	EX	MEM	WB
noop			IF	ID	EX	MEM
noop				IF	ID	EX
nor						
halt					IF	ID

Notice that now, the **nor** is never loaded into the pipeline, meaning that we do not have to worry about its side effects.

Next, let's consider an example where the branch is not taken:

```
            nor         0     0     1
            beq         0     1     label
            nor         1     1     2
label       halt
```

When we modify this for avoidance, we **still** have to add three **noop** instructions! While we do know that this branch is not taken in this example, we can't make this generalization in all cases. Thus, we will still insert three **noop**s:

```
            nor         0     0     1
            beq         0     1     label
            noop
            noop
            noop
            nor         1     1     2
label       halt
```

If we consider the behavior of this program, we'll continue executing correct instructions, but we spend an extra three cycles executing **noop** instructions, lowering our efficiency.

[16] Pipe traces get a bit more complicated with branches, since the instructions can be laid out in a variety of ways on the left hand side. For these notes, the instructions will be listed on the left in the order that they appear in the assembly code. This will lead to empty rows for instructions that are skipped, and there may be multiple appearances of pipeline stages in a row which is loaded into the pipeline multiple times.

The first eight cycles of the pipe trace are as follows:

Cycle	1	2	3	4	5	6	7	8
nor 0 0 1	IF	ID	EX	MEM	WB			
beq 0 1 label		IF	ID	EX	MEM	WB		
noop			IF	ID	EX	MEM	WB	
noop				IF	ID	EX	MEM	WB
noop					IF	ID	EX	MEM
nor 1 1 2						IF	ID	EX
halt							IF	ID

Resolving control hazards using detect and stall is analogous to resolving data hazards with detect and stall: rather than leaving it up to the programmer to insert **noop** instructions, the processor will insert **noop**s into the pipeline. Here, the logic is somewhat simpler: while a **beq** is present in the ID/EX or EX/MEM stages, we insert a **noop** into the IF/ID register, blocking any instructions from being fetched until the **beq** is resolved. The first seven cycles of the pipe trace are as follows:

Cycle	1	2	3	4	5	6	7
beq 0 1 label	IF	ID	EX	MEM **(set PC)**	WB		
(noop)		IF	ID	EX	MEM	WB	
(noop)			IF	ID	EX	MEM	WB
(noop)				IF	ID	EX	MEM
nor 1 1 2		IF*	IF*	IF*	not fetched		
halt					IF	ID	EX

Now, **nor** is stalled in the IF stage until the **beq** resolves (in cycle 4), updating the PC to fetch **halt** on the next cycle.

One of the biggest inefficiencies here is that even if a branch is not taken, we still add three **noop** instructions. In a real program, we don't know whether a branch is taken or not, so we have to add **noop**s in both cases for detect and stall. Let's think about what we would do if we had perfect information:

- Branch taken: stall the pipeline for three cycles, then continue as normal.
- Branch not taken: do not stall the pipeline, continue execution as normal.

When **beq** is in the ID stage, however, we don't have enough information to determine how the branch will proceed. Instead, we will use **specuate and squash**, where we make a prediction about whether the branch will be taken or not taken, and then "squash", or roll back our changes, if we predicted incorrectly[17]. Let's take a look at a simple example of speculate and squash, where we predict that all branches will never be taken. Consider the following program:

```
            beq         0     0     label
            nor         0     0     1
label       halt
```

If we consider the pipe trace for this program, using predict not-taken, we will have the following (only the first 7 cycles shown):

Cycle	1	2	3	4	5	6	7
beq 0 0 label	IF	ID	EX	MEM	WB		
nor 0 0 1		IF?	ID?	EX?	squashed		
halt			IF?	ID?	IF	ID	EX

Here, speculative execution is marked with a "?" after the stage. Once the **beq** is resolved in cycle 4, the processor realizes that the branch was taken, and squashes the IF/ID, ID/EX, and EX/MEM registers (the **beq** has moved on to the MEM/WB register). Thus, in the next cycle, we fetch **halt**, which is the correct next instruction. Now, let's consider a branch which is not taken, and thus predicted correctly.

```
            nor         0     0     1
            beq         0     1     label
            nor         0     0     1
label       halt
```

Here, we will continue to load **nor** and **halt** into the processor, and we speculatively execute them. Once we resolve the **beq** in the MEM stage (cycle 5), we recognize that we predicted the branch correctly, and therefore no squashing is required.

[17] This can lead to some very interesting hardware vulnerabilities if we don't perform this "rollback" correctly. We'll see more about this when we discuss caching.

The pipe trace is now (first 6 cycles):

Cycle	1	2	3	4	5	6
nor 0 0 1	IF	ID	EX	MEM	WB	
beq 0 1 label		IF	ID	EX	MEM	WB
nor 0 0 1			IF?	ID?	EX?	MEM
halt				IF?	ID?	EX

Notice that we no longer have to insert **noop**s into the pipeline! Since our speculative execution was correct, we can continue fetching and executing instructions without having to wait for branch resolution (in cycle 5).

Branch Predictors

In our previous example, we showed how speculate-and-squash can help improve the performance of the pipeline when resolving control hazards. We predicted that all branches would be **not taken**. This seems like a bit of a silly choice, surely if we're putting branches into our pipeline, then we want to take them, right? To help improve our performance even further, we can design **branch predictors**, which predict the outcome of a branch at runtime. We've already seen one example of a branch predictor: the static predict not taken predictor. We can group branch predictors into two general classes: static predictors and dynamic predictors.

Static branch predictors are the simpler type of branch predictors: they always make the same prediction for each branch, regardless of information about whether the branch was taken or not taken before. Let's take a look at three static branch predictors: predict not taken, predict taken, and predict backwards taken, forward not taken. For each of the branch predictors, we will use the following program:

```
            lw          0     1     five        i = 5
            lw          0     2     negOne
            lw          0     3     mask1
            lw          0     4     mask2
loop        beq         0     1     done        // beq1
            nor         1     3     5
            nor         5     5     5
            beq         3     5     inc         // beq2
            nor         1     4     5
            nor         5     5     5
            beq         4     5     inc         // beq3
```

```
inc         add         1     2     1
            beq         0     0     loop        // beq4
five        .fill       5
mask1       .fill       -4                      0b1111...1100
mask2       .fill       -2                      0b1111...1110
```

This code corresponds to the following C code:

```
for (int i = 5; i != 0; --i) {
    if (i % 4 == 0) continue;
    if (i % 2 == 0) continue;
}
```

If we look at the **true** sequence of branches being taken/not taken, we have the following sequence of 20 branches being executed:

(i = 5)
beq1 not taken
beq2 not taken
beq3 not taken
(i = 4)
beq4 taken
beq1 not taken
beq2 taken
(i = 3)
beq4 taken
beq1 not taken
beq2 not taken
beq3 not taken
(i = 2)
beq4 taken
beq1 not taken
beq2 not taken
beq3 taken
(i = 1)
beq4 taken
beq1 not taken
beq2 not taken
beq3 not taken
(i = 0)
beq4 taken
beq1 taken

We'll use this sequence of branches being taken/not taken as an example for our branch predictors.

Predict Not Taken:
The predict not taken branch predictor (unsurprisingly) always predicts that branches are not taken. We can analyze the performance of branch predictors by writing out their predictions in a table:

	actual result	prediction
(i = 5)		
beq1	not taken	**not taken**
beq2	not taken	**not taken**
beq3	not taken	**not taken**
(i = 4)		
beq4	taken	**not taken**
beq1	not taken	**not taken**
beq2	taken	**not taken**
(i = 3)		
beq4	taken	**not taken**
beq1	not taken	**not taken**
beq2	not taken	**not taken**
beq3	not taken	**not taken**
(i = 2)		
beq4	taken	**not taken**
beq1	not taken	**not taken**
beq2	not taken	**not taken**
beq3	taken	**not taken**
(i = 1)		
beq4	taken	**not taken**
beq1	not taken	**not taken**
beq2	not taken	**not taken**
beq3	not taken	**not taken**
(i = 0)		
beq4	taken	**not taken**
beq1	taken	**not taken**

In this example, our static branch predictor predicts 12 out of 20 branches correctly.

Predict Always Taken:
Next, we have the **predict always taken** predictor. As the name suggests, it predicts that every branch will be taken. To implement this in hardware, we need to remember the PC of each branch's target. We will use the **branch target buffer**, or **BTB** to store the destination of all branches. The BTB can be initialized upon loading the program into memory, or it can be dynamically generated as the program executes. With the BTB, we can begin fetching

instructions from the predicted **new** PC. If we resolve incorrectly, we will squash the incorrectly fetched instructions, and fetch from the correct location (old PC + 1).

In our previous example, the always taken predictor has an accuracy of 8 / 20. However, it does perform better on **beq4**, which is always taken. Let's compare **beq1**, **beq2**, and **beq3**, which are more often not taken, with **beq4**, which is always taken. Notice that **beq1**, **beq2**, and **beq3** are forward branches, which branch to a *higher* PC than the **beq** instruction. On the other hand, **beq4** is a backwards branch, which branches to a *lower* PC than the **beq** instruction. If we think about our assembly patterns (Chapter 4), backwards branches typically correspond to unconditional branches that return us to the beginning of a loop. Forward branches are conditional statements, or the exit condition of loops. Thus, it makes sense that our backwards branches typically are always taken, while our forwards branches are more often not taken. Let's design a new predictor that tries to account for this:

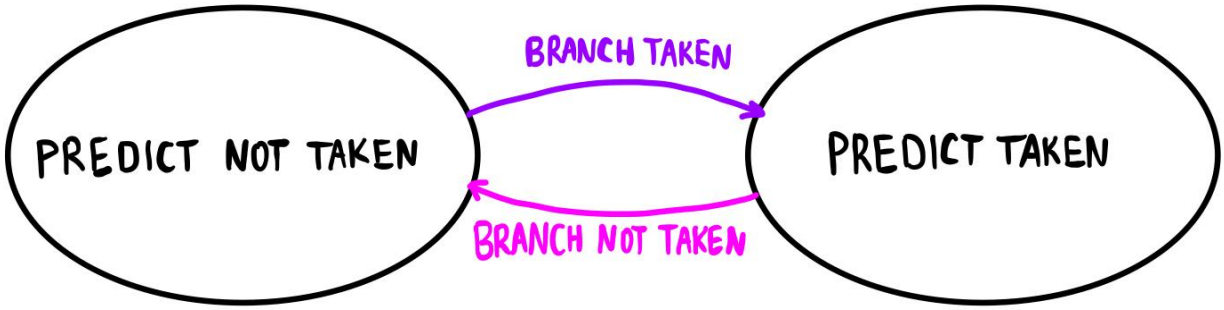
Predict Backwards Taken, Forward Not Taken:
The backwards taken, forward not taken predictor uses the direction of a branch to help predict if a branch will be taken. As detailed above, backwards branches are more likely to be taken in general due to the structure of conditionals and loops in assembly. Thus, predicting backwards branches as taken and forward branches as not taken will help improve our accuracy in more complex programs.

	actual result	prediction
beq1	not taken	**not taken**
beq2	not taken	**not taken**
beq3	not taken	**not taken**
beq4	taken	**taken**
beq1	not taken	**not taken**
beq2	taken	**not taken**
beq4	taken	**taken**
beq1	not taken	**not taken**
beq2	not taken	**not taken**
beq3	not taken	**not taken**
beq4	taken	**taken**
beq1	not taken	**not taken**
beq2	not taken	**not taken**
beq3	taken	**not taken**
beq4	taken	**taken**
beq1	not taken	**not taken**
beq2	not taken	**not taken**
beq3	not taken	**not taken**
beq4	taken	**taken**
beq1	taken	**not taken**

Now, we are able to achieve an accuracy of 17 out of 20! Quite impressive for a static branch predictor.

One of the downsides of predict backwards taken, forward not taken is that we begin to create "cases". While we could improve upon this even more, what if we could create a branch predictor that adapts to whatever situation it is placed in? This is where **dynamic branch predictors** come into play.

Let's start with a very simple dynamic branch predictor: the 1-bit saturating branch predictor. We can represent this branch predictor with an FSM. The FSM is shown below.



When we initialize the branch predictor, we will initialize it in either the "predict not taken" or the "predict taken" state. From here, we will predict the corresponding outcome, and update the FSM state accordingly based on the actual outcome. Let's take a look at an example:

	FSM state	prediction	actual result
beq1	NT	not taken	not taken
beq2	NT	not taken	not taken
beq3	NT	not taken	not taken
beq4	NT	not taken	taken

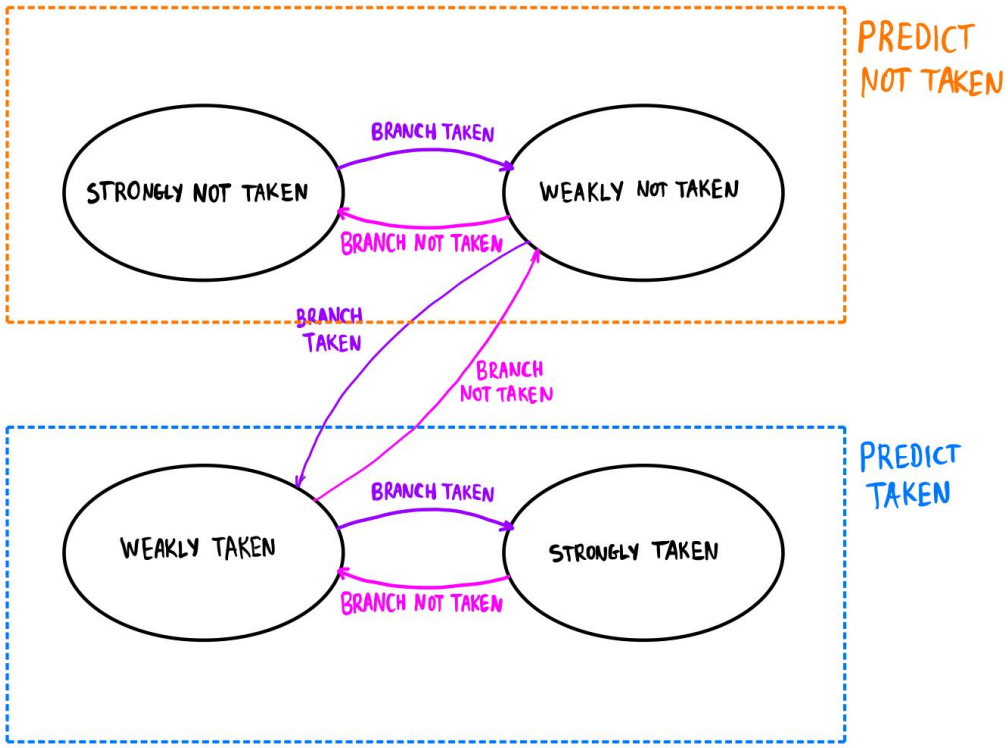
We initialize the branch predictor to the "not taken" (NT) state. Thus, the branch predictor will predict not taken for **beq1**. The true outcome is indeed not taken, and thus we remain in the "not taken" state. This continues until our first incorrect prediction, on **beq4**. We predict "not taken", since we were in the NT state. However, the branch is actually taken. Thus, we transition to the "taken" (T) state for our next prediction.

beq1	T	taken	not taken

On the next prediction, we predict taken, but the branch is actually not taken! Thus, we update our state back to "not taken" (NT). We can continue this for the remaining branch outcomes.

beq2	NT	not taken	taken
beq4	T	taken	taken
beq1	T	taken	not taken
beq2	NT	not taken	not taken
beq3	NT	not taken	not taken
beq4	NT	not taken	taken
beq1	T	taken	not taken
beq2	NT	not taken	not taken
beq3	NT	not taken	taken
beq4	T	taken	taken
beq1	T	taken	not taken
beq2	NT	not taken	not taken
beq3	NT	not taken	not taken
beq4	NT	not taken	taken
beq1	T	taken	taken

Here, we achieve an accuracy of 11 correctly predicted branches out of 20. Let's try to do better. If we look at the pattern of branches in our example above, we notice that the majority of the branches are not taken, with a few "taken"s breaking up runs of "not taken". Our 1 bit branch predictor is guaranteed to mispredict twice for a sequence such as "NT, NT, T, NT, NT". To improve our performance in these scenarios, we will introduce a **2-bit saturating branch predictor**. The 1-bit predictor uses an FSM with two states (2^1), while the 2-bit predictor will use an FSM with four states (2^2).



Notice that we now have four states, two of which are "taken" states and two of which are "not taken" states. We name the two "taken" states "weakly taken" and "strongly taken", and the two "not taken" states "weakly not taken" and "strongly not taken". When a branch resolves as taken, we move our FSM state towards "strongly taken", while when a branch resolves as not taken, we move our FSM state towards "strongly not taken". Let's take a look at an example.

	FSM state	prediction	actual result
beq1	WNT	not taken	not taken
beq2	SNT	not taken	not taken
beq3	SNT	not taken	not taken
beq4	SNT	not taken	taken

We initialize the branch predictor to the "weakly not taken" (WNT) state. Thus, the branch predictor will predict not taken for **beq1**. The true outcome is indeed not taken, and thus we move to the "strongly not taken" (SNT) state. From here, we will continue predicting "not taken", and since **beq2** and **beq3** indeed are not taken, we remain in the SNT state. When we reach **beq4**, we are in the SNT state, and we predict "not taken". However, the branch is actually taken, so we return to the WNT state.

beq1	WNT	not taken	not taken

On the next prediction, we are in the WNT state, so we predict not taken! This shows how the 2-bit branch predictor is better at handling occasional deviations from a long run of not takens (and consequently takens) than the 1-bit predictor. We can continue analyzing each branch:

beq2	SNT	not taken	taken
beq4	WNT	not taken	taken
beq1	WT	taken	not taken
beq2	WNT	not taken	not taken
beq3	SNT	not taken	not taken
beq4	SNT	not taken	taken
beq1	WNT	not taken	not taken
beq2	SNT	not taken	not taken
beq3	SNT	not taken	taken
beq4	WNT	not taken	taken
beq1	WT	taken	not taken
beq2	WNT	not taken	not taken
beq3	SNT	not taken	not taken
beq4	SNT	not taken	taken
beq1	WNT	not taken	taken

Here, our accuracy drops even further, to 10 out of 20 branches correct! What is happening here is that due to the order of branches being executed, the branch predictor is getting "mixed up" between the patterns of different branches. Currently, we are using a **global branch predictor**, which uses one branch predictor for all branches in the program. Let's switch to a **local branch predictor**, which creates separate branch predictors for each branch. We will initialize all our 2-bit branch predictors in the WNT state.

	FSM State	prediction	actual result	new state
beq1	WNT	not taken	not taken	SNT
beq2	WNT	not taken	not taken	SNT
beq3	WNT	not taken	not taken	SNT
beq4	WNT	not taken	taken	WT
beq1	SNT	not taken	not taken	SNT
beq2	SNT	not taken	taken	WNT
beq4	WT	taken	taken	ST
beq1	SNT	not taken	not taken	SNT
beq2	WNT	not taken	not taken	SNT
beq3	SNT	not taken	not taken	SNT
beq4	ST	taken	taken	ST
beq1	SNT	not taken	not taken	SNT
beq2	SNT	not taken	not taken	SNT
beq3	SNT	not taken	taken	WNT
beq4	ST	taken	taken	ST
beq1	SNT	not taken	not taken	SNT
beq2	SNT	not taken	not taken	SNT
beq3	WNT	not taken	not taken	SNT
beq4	ST	taken	taken	ST
beq1	SNT	not taken	taken	WNT

We achieve an accuracy here of 16 correct predictions out of 20 branches! This massive gain in accuracy stems from the fact that each branch has its own unique pattern (some are almost always not taken, some are always taken), and the ordering of these branches leads to a global predictor becoming "confused". On the other hand, a local predictor only sees the history of a specific branch, and therefore can understand these patterns better.

Pipeline Performance

Let's analyze the performance of the new pipeline processor. Let's use the same latencies and benchmark as our single cycle processor, so that we can compare the performance of the three processor designs.

Consider a pipelined processor with the following component latencies:

- instruction memory read: 10 ns
- register file read: 5 ns
- register file write: 8 ns
- ALU: 15 ns
- data memory read: 25 ns
- data memory write: 35 ns

Treat all components not listed above as having 0 ns latencies.

To find the clock period, we can take the longest latency from these components, giving us 35 ns. Next, we can look at our benchmark to find the number of cycles executed.

- 15 **add** instructions
- 15 **nor** instructions
- 30 **lw** instructions
- 20 **sw** instructions
- 10 **beq** instructions
- 10 **noop or halt** instructions

We have 100 instructions, and each instruction takes 5 cycles. However, we can execute 5 instructions in parallel at once, meaning that the program will take 104 cycles. Why 104? Remember that the pipeline doesn't start with five instructions loaded in, but has to start with the first instruction in IF. Thus, we have an extra four cycles that are required to fill in all stages of the pipeline. Thus, our performance is 104 cycles * 35 ns = 3,640 ns. However, this number is for the pipeline **without any hazards**. Let's take a look at how having data and control hazards impacts our performance. Let's start with data hazards.

Impact of Data Hazards on Pipeline Performance

Let's consider the two main approaches to resolving data hazards: detect and stall, or detect and forward. Let's consider the same benchmark as above, but with the following extra information:

- 40% of **lw** instructions are immediately followed by a dependent instruction
- 20% of **lw** instructions are followed by a non-dependent instruction, which is followed by a dependent instruction (on the **lw**). We'll call these dependencies 2-away dependencies for convenience.
- 30% of **add** or **nor** instructions are immediately followed by a dependent instruction
- 20% of **add** or **nor** instructions are followed by a 2-away dependency (on the **add**/**nor**)
- The processor supports internal forwarding.

First, let's consider detect and stall. We have to stall two cycles when we have an immediate dependency and one cycle when we have a 2-away dependency, regardless of the instruction causing a dependency. If we think about the pipe traces, with an immediate dependency, we

would have the following pipe trace. We need to align **add 1 2 3**'s ID stage with **lw 0 1 five**'s WB stage since we have internal forwarding.

Cycle	1	2	3	4	5
lw 0 1 five	IF	ID	EX	MEM	WB
add 1 2 3		IF	ID*	ID*	ID

With a 2-away dependency, we would have the following pipe trace. Again, we align **add 1 2 3**'s ID stage with **lw 0 1 five**'s WB stage.

Cycle	1	2	3	4	5
lw 0 1 five	IF	ID	EX	MEM	WB
noop		IF	ID	EX	MEM
add 1 2 3			IF	ID*	ID

One of the best ways to work out the number of stall cycles for a hazard is to draw the pipe trace of a small example with the specified hazard, as shown above. With these dependencies, we can calculate the total number of stall cycles in our benchmark:

- 30 **lw** * 40% of **lw** with immediate dependency * 2 stalls = 24 stall cycles
- 30 **lw** * 20% of **lw** with 2-away dependency * 1 stall = 6 stall cycles
- 30 **add**/**nor** * 30% of **add**/**nor** with immediate dependency * 2 stalls = 18 stall cycles
- 30 **add**/**nor** * 20% of **add**/**nor** with 2-away dependency * 1 stall = 6 stall cycles

Thus, the total number of stall cycles incurred is 24 + 6 + 18 + 6 = **54 stall cycles** using detect and stall.

Next, let's consider detect and forward. Here, recall that the only scenario where a stall is required is with a **lw** followed by an immediate dependency, since we have to wait until the MEM stage to know the updated register value. Thus, the total number of stall cycles is 30 **lw** * 40% of **lw** with immediate dependency * 1 stall = **12 stall cycles**. Observe that by using data forwarding, we can eliminate a significant number of stall cycles (compare this to the 54 stall cycles in detect and stall).

Impact of Control Hazards on Pipeline Performance

In general, we can treat control hazards independently from data hazards. This allows us to split up our cycle calculation into four parts:

instructions + # startup cycles + # data hazard stalls + # control hazard stalls

Let's take a look at the two methods for resolving control hazards: detect and stall and speculate and squash. We'll use the following additional information about the benchmark:

- 40% of branches are taken

First, let's consider detect and stall. We need to stall 3 cycles regardless of if a branch is taken or not; thus, we have 10 **beq** * 3 cycles = **30 stall cycles**.

Next, let's consider speculate and squash. We can use a variety of branch predictors, but let's assume we have a branch predictor which predicts correctly 70% of the time. Thus, we need to stall only for an incorrect prediction, which accounts for 30% of the **beq** instructions. From this, we can calculate the number of stalls incurred: 10 **beq** * 30% mispredicted * 3 cycles = **9 stall cycles**.

Putting this all together, we can calculate the performance of the pipeline on any combination of data hazard and control hazard resolution methods.

Data Hazards	Control Hazards	Total Cycles
Detect and stall: 54	Detect and stall: 30	100 + 4 + 54 + 30 = **188** cycles
Detect and stall: 54	Speculate and squash: 9	100 + 4 + 54 + 9 = **167** cycles
Detect and forward: 12	Detect and stall: 30	100 + 4 + 12 + 30 = **146** cycles
Detect and forward: 12	Speculate and squash: 9	100 + 4 + 12 + 9 = **125** cycles

Modified Pipelines

To optimize the pipeline, one of the biggest contributors towards performance is the number of pipeline stages that are present. With a deeper pipeline (more stages), we can reduce the clock period by splitting up logic between multiple stages, but we eventually reach a balance point, where the reduction in clock period is offset by the increasing number of control hazards present. Let's consider a pipeline where we split the memory stage into two stages: MEM1 and MEM2. Branches finish in MEM1, while **lw** and **sw** finish in MEM2. Let's consider the impact of this change on data and control hazards.

First, let's start with data hazards. If we are using detect and stall, we have to stall one extra cycle, since the extra MEM stage puts one more stage between ID and WB. The pipe trace would be as follows:

Cycle	1	2	3	4	5	6
add 1 1 2	IF	ID	EX	MEM1	MEM2	WB
add 2 2 3		IF	**ID***	**ID***	**ID***	ID

Next, let's consider detect and forward. Our two main cases will be an **add**/**nor** followed by an immediate dependency, and a **lw** followed by an immediate dependency. Let's start with **add**/**nor**. We know that the result of the **add**/**nor** is still computed in the EX stage, meaning that we can continue to forward from the **add**/**nor**'s EX stage to the next instruction's EX stage. Thus, we do not need to incur any stalls (see the pipe trace below)

Cycle	1	2	3	4	5	6
add 1 1 2	IF	ID	**EX**	MEM1	MEM2	WB
add 2 2 3		IF	ID	**EX**	MEM1	MEM2

Next, let's consider a **lw** followed by an immediate dependency. We know that **lw** finishes reading the value from memory in the MEM2 stage, meaning that we need to wait until MEM2 finishes to forward the value to the dependency's EX stage. Thus, we have to incur **two** stall cycles.

Cycle	1	2	3	4	5	6
lw 0 1 five	IF	ID	EX	MEM1	**MEM2**	WB
add 1 1 2		IF	ID*	ID*	ID	**EX**

Next, we have control hazards. Let's consider detect and stall for simplicity. Using speculate and squash will incur the same number of stall cycles, but only when a branch is mispredicted.

Cycle	1	2	3	4	5	6
beq 0 0 A	IF	ID	EX	MEM1 **(set PC)**	MEM2	WB
(noop)		IF	ID	EX		
(noop)			IF	ID		
(noop)				IF		
add 1 1 2					not fetched	

Here, we see that since branches are still resolved in MEM1, we again only have to stall for three cycles.

Chapter 8: Caching

We've spent the last two chapters working to design a faster processor by changing how instructions are executed and allowing our processor to parallelize multiple instructions with a pipeline. However, there's another way to design a faster processor: make the individual components faster. Let's consider one of the longest components in terms of latency: memory. Due to the large address space available to the program, memory is often placed further away from the processor. Additionally, fast memory circuits (SRAM) are much more expensive to produce, and using SRAM for a program's memory would be prohibitively expensive. Let's think about how we can optimize memory accesses. Consider the following program:

```
            lw          0     1     N
            lw          0     2     negOne
loop        beq         0     1     end
            add         1     2     1
            beq         0     0     loop
end         halt
N           .fill       1000
negOne      .fill       -1
```

This program runs a loop for 1000 iterations, and thus fetches **beq 0 1 end**, **add 1 2 1**, and **beq 0 0 loop** from memory 1000 times each. Each of these will have to wait on memory to fetch the instruction from DRAM. Since we reuse these instructions so much, what if we could store them in the processor, so that if they were ever fetched again, we could access them quicker? This is the main concept behind a **cache**. Caches allow us to store smaller amounts of data in SRAM, allowing us to access it quicker.

Temporal and Spatial Locality

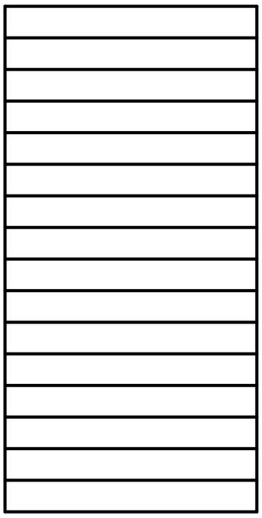
When we consider caches, we can consider two main areas for optimization: temporal and spatial locality. Temporal locality refers to the fact that when an address is accessed once, it is likely that the exact **same** address will be accessed again later during the program. For example, the previous program is a good example of spatial locality. Next, we have spatial locality, which refers to the fact that when an address is accessed once, **nearby** addresses are likely to be accessed again later during the program. The following program demonstrates spatial locality. Here, we access every element of a 16-word array, one after the other. When iterating through an array, we will access each element sequentially. Thus, we will access Array[0], then Array[1], and so on. Spatial locality arises from the fact that arrays are stored contiguously in memory.

```
            lw          0     2     ArrLen
            lw          0     3     one
loop        beq         1     2     end
            lw          1     4     Array
            beq         0     0     loop
end         halt
one         .fill       1
ArrLen      .fill       16
Array       .fill       0
```

We can design our cache to improve our performance by utilizing temporal and spatial locality.

Cache Design and Organization

Let's create a cache for our LC2K processor with 16 words of memory. We'll create 16 words of SRAM to store our data.

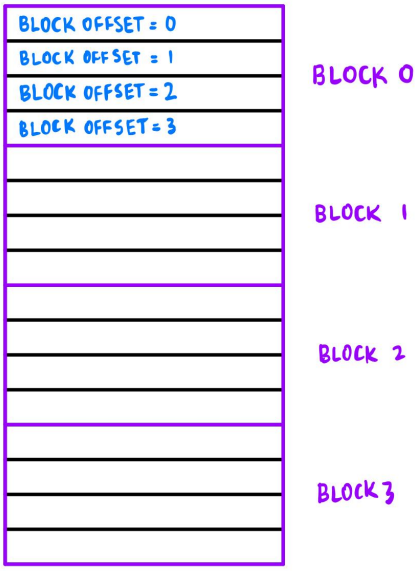


How do we organize the cache? To decide how we keep memory in the cache, we will use the address of the memory being accesed. If we consider an address such as 0x5:

```
0b    0000  0000  0000  0101
```

Changing the bits on the right (the least significant bits) changes the value of the address by a **small** amount, while changing the bits on the left (the most significant bits) changes the value of the address by a **large** amount. Thus, we want to group up addresses with the same most significant bits (meaning their values are very similar) to utilize spatial locality. To do this, we will load in **blocks**, which are contiguous sequences of addresses. For example, we could load in the block from address 0 to address 3 (inclusive). At the same time, we do have a tradeoff:

as we increase the **block size** (the number of contiguous addresses loaded in per block), we decrease the number of unique blocks that can be stored in the cache, meaning that the cache's utilization of temporal locality decreases. For this example, we will choose a block size of 4 words, meaning that we will have 4 blocks. Thus, the cache will be organized as show below:

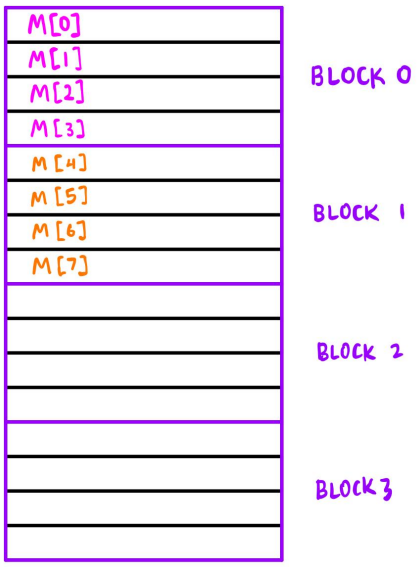


We've created 4 blocks, each with a size of 4 words. For each address, we will calculate its **block offset**, which tells us the "index" into the block. To calculate the block offset, we will use the binary value of the address. Let's consider 0x5 again (0b0000 0101). Since our block size is 4 words, we will need 2 bits to index the 4 indices into the block ($2^2 = 4$). Thus, we will take the 2 *least significant* bits from the address (0b01), and use these as our block offset. This allows us to take advantage of spatial locality: addresses with similar least significant bits are close together. We will refer to the remaining bits (0b0000 01) as the **tag**. To determine if an address is in the cache, we can perform a tag comparison between the desired address and the blocks in the cache; if a block with a matching tag is found, we can index into that block using the block offset.

Next, we have the question of how we will assign blocks from memory to blocks in the cache. We can think of memory as also being split into blocks of 4 words, and the cache allows us to access a few blocks quicker. We'll start with a simple method: we'll allow blocks to place themselves anywhere in the cache. Let's consider the following sequences of accesses:

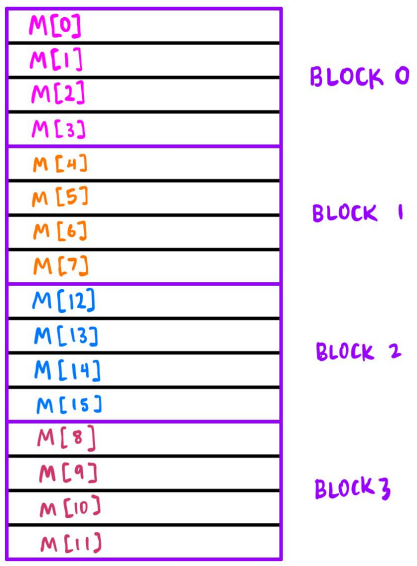
- 0x0
- 0x5
- 0x1
- 0xD
- 0x8
- 0x10

First, we'll access address 0x0, which is in the block of memory from address 0 through 3. We'll bring this into the cache and place it into block 0. Next, we access address 0x5, which is in the block of memory from address 4 through 7. We'll also bring this into the cache and place it into block 1. After these two accesses, the state of the cache is as follows:



Our next access is to address 0x1. We check the cache, and we find that we've already loaded in address 1 into the cache! This allows us to directly access the value in the cache, meaning that we do **not** have to access main memory!

Next, we'll access address 0xD (13 in decimal). We'll load in the block containing addresses 12 through 15 into the cache. After this, we access 0x8, which loads the block containing addresses 8 through 11 into the cache. Now, our cache contains the following blocks:



Our next access is to address 0x10, or 16 in decimal. 0x10 resides in the block of memory containing addresses 16 through 19. However, we don't have room in our cache anymore! Let's think about how to design an **eviction policy** for our cache. To determine which block should be removed, or **evicted**, from the cache, we choose to use a **least recently used**, or **LRU** policy. When a block needs to be evicted from the cache, we will choose the block that was least recently accessed. In our previous example, the block with addresses 4 through 7 is least recently used, since the other three blocks were accessed more recently. In the hardware of the cache, we will use 2 bits (2^2 = 4 blocks to order) to keep track of the ordering of block accesses. This means that we will incur additional **overhead bits**. The block combined with the overhead bits is referred to as a **cache line**. Overhead bits will consist of three main parts: a **valid bit**, which indicates if the cache line is being used or not; a **dirty bit**, which indicates if the cache line has been modified with respect to main memory (more on this later); and the **LRU bits**, which indicate the order of accesses.

The example we just showed above was an example of a **fully associative cache**. In a fully associative cache, blocks from memory can be placed in *any* of the blocks in the cache. If we consider the lookup process in the cache, we would have to check every single line in the cache. To reduce the number of comparisons we need to make, we can specify that each block from memory *must* go into a specific cache line. This is the concept behind a **direct mapped cache**.

In a direct mapped cache, we calculate a **line index** for each address, as well as a block offset. Again, we will use the bits of the address to calculate both the line index and the block offset. Programs typically access memory in a relatively small area of the large address space accessible to them. Thus, if we used the most significant bits to determine which cache line the block would be assigned to, we would always assign blocks to only one or two lines, which reduces our efficiency significantly. Instead, we will use the least significant bits to represent the line index. We will thus split up the address into three portions: the tag, the line index, and the block offset. If we again consider a 16 word cache with 4 word blocks, we have 4 blocks, meaning that our line index will be 2 bits (to address 2^2 lines). Thus, the address 0xD (decimal 13, binary 0b0000 1101) would have a tag of 0b0000, a line index of 0b11, and a block offset of 0b01. The line index indicates that the block containing 0xD would be placed into line 3 of the cache.

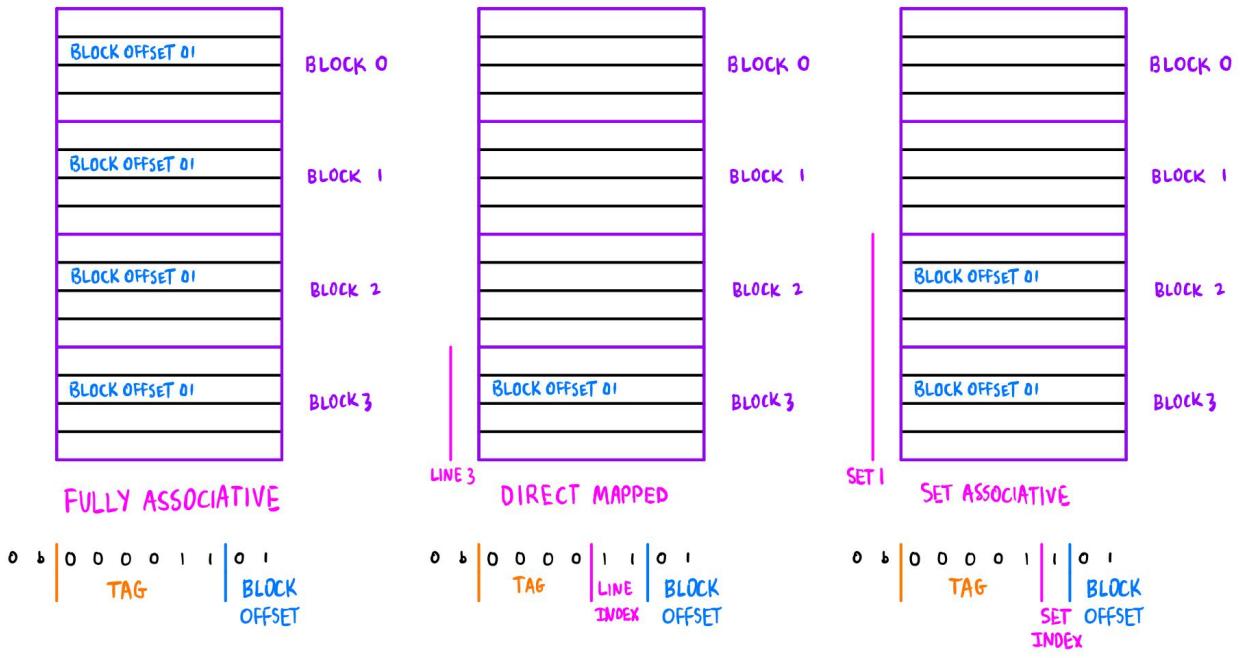
How does this help us when looking up addresses in the cache? A direct mapped cache makes a guarantee that there is only a single location that a specific block can be placed in the cache. Thus, we will only ever have to perform a single tag comparison for a specific address. One of the biggest downsides of the direct mapped cache, however, is that we have a very low **associativity**: we cannot store more than one block with the same line index in the cache at the same time. If we compare this to the fully associative cache, we are able to store any number (limited by the size of the cache) of blocks in the cache, regardless of their tag bits. The **set associative cache** is a compromise between these designs. Here, we will still use bits from the address to "group" blocks into different sets in the cache (analogous to how the direct mapped cache assigns a specific line to each block). However, each set will have

multiple lines that a block can be placed in (often called **ways**). Within each set, we will use a LRU eviction policy to evict blocks as new addresses are accessed.

A comparison of the three types of caches is shown below[18]. In the fully associative cache, addresses are split into a tag and block offset. The tag is compared with the tag of **every** cache line, and the block offset is used to index into the block with a matching tag.

In the direct mapped cache, addresses are split into a tag, a line index, and a block offset. The line index specifies the specific line that the desired block can be found at in the cache. The tag is compared with the tag of the **specific** line (designated by the line index). Finally, the block offset is used to index into the block.

Finally, in the set associative cache, addresses are split into a tag, a set index, and a block offset. The set index narrows down the number of possible lines that the desired block can be found at. The tag is compared with the tag of this **subset** of all cache lines. Finally, the block offset is used to index into the block with a matching tag. Below, we've shown a **2-way** set associative cache, meaning that each set contains two "ways", or lines.



[18] In essence, the fully associative and direct mapped caches are just the two extremes of a set associative cache. We can think of the fully associative cache as a set associative cache with only a single set, while the direct mapped cache is a set associative cache with an equal number of sets and blocks.

Writing to the Cache

Caches are great for reading data, but we want to extend their functionality to writing data as well. To handle writing to the cache, we have two potential options: implementing a **write-through** cache or a **write-back** cache. In a write-through cache, we immediately send write operations to main memory, and update the values stored in the cache correspondingly. This allows us to only write to the bytes that we need, but results in more writes in the long run. In a write-back cache, we keep track of the updated values of memory in the cache block, and only update memory when the cache block is evicted. While we have to keep a dirty bit for each block, we can often reduce the number of writes to memory, especially if a block is being written to multiple times over the course of a program.

Additionally, we can differentiate caches by their **write-allocation policy**. First, we have **allocate on write** caches, where a block is allocated for writes to addresses not originally in the cache. We also have **no allocate on write** caches, where blocks are not allocated unless a read operation is performed.

Let's take a look at a simple example. Consider a fully associative cache with a block size of 2 words and a total cache size of 4 words. We will make the following series of memory accesses:

- Read from address 0
- Write to address 5

Let's start with a **write-through**, **allocate on write** cache. On our first access, we will load addresses 0 and 1 into the cache. Thus, the state of our cache is as follows:

Line 0, Block Offset 0: **address 0**
Line 0, Block Offset 1: **address 1**
Line 1, Block Offset 0: empty
Line 1, Block Offset 1: empty

Next, we will write to address 5. Since this is an allocate on write cache, we will allocate line 1 for addresses 4 and 5. The write to memory takes place here, writing address 5 to main memory. Thus, our cache ends with addresses 0, 1, 4, and 5 in the cache.

Next, let's consider a **write-through**, **no allocate on write** cache. On our first access, we allocate line 0 for addresses 0 and 1 again. However, on our second access, since the cache is no allocate on write, addresses 4 and 5 are **not** loaded into the cache. Thus, the final state of the cache will only contain addresses 0 and 1.

If we switch our writeback policy to **write-back**, and use an **allocate on write** policy, then we will allocate line 1 for addresses 4 and 5. However, we do not write to main memory here. We will instead mark line 1 as dirty. Later, if we evict line 1, we will write back **both** addresses 4 and 5 to main memory. This is because the dirty bit applies to the entire block, rather than a specific line.

Finally, we can consider a **write-back**, **no allocate on write** policy. Here, we never load addresses 4 and 5 to main memory, but rather the write is sent directly to main memory. If we had a different scenario, where we loaded address 5 into the cache with a **read**, and then performed a write, we would mark the line as dirty.

Accessing the Cache

Let's go through an example of utilizing the cache. Consider the following program:

```
            lw          0     1     one
            add         1     1     2
            sw          0     2     two
            halt
one         .fill       1
two         .fill       2
```

When we integrate a cache into our program, we can use an **instruction cache (I-cache)**, a **data cache (D-cache)**, or a **unified cache**. Most commonly, we will use either two caches (one I-cache, one D-cache) or a unified cache. Let's consider the example above using a unified fully-associative, allocate-on-write, write-through cache with a block size of 2 words and a total size of 4 words. To approach this, we will first write out the **access pattern**, in other words, which addresses are accessed.

load 0 (read **lw 0 1 one**)
load 4 (effect of **lw 0 1 one**)
load 1 (read **add 1 1 2**)
load 2 (read **sw 0 2 two**)
store 5 (effect of **sw 0 2 two**)
load 3 (read **halt**)

Next, we can simulate cache accesses. For each access, we will mark it as either a **cache hit** or a **cache miss**. A cache hit occurs when the memory we need to access is found in the cache. On the other hand, a cache miss occurs when the memory we need to access is *not* found in the cache.

First, let's start with loading address 0. Since our cache is empty, this is a **cache miss**. We load in the block containing addresses 0 and 1 into the cache. Thus, our cache is organized as show below:

Line 0, Block Offset 0: **address 0**
Line 0, Block Offset 1: **address 1**
Line 1, Block Offset 0: empty
Line 1, Block Offset 1: empty

On our next access, we load address 4. We can split address 4 (0b0100) into its tag and block offset bits (recall a fully associative cache does not use set bits). Since our block size is 2 words, we have 1 block offset bit. Thus, the block offset of address 4 is 0, with a tag of 0b010. If we check the tag of line 0, it is 0b000. Thus, we have a **cache miss**, since we cannot find the corresponding block in the cache. Thus, we will load the block containing addresses 4 and 5 into the cache. The new state of the cache is shown below:

Line 0, Block Offset 0: **address 0**
Line 0, Block Offset 1: **address 1**
Line 1, Block Offset 0: **address 4**
Line 1, Block Offset 1: **address 5**

Next, we load address 1. We can split address 1 (0b0001) into a tag of 0b000 and a block offset of 0b1. When we check line 0, we see that the tags match (both 0b000). Thus, we have our first **cache hit**, and we can directly return the value from the cache instead of accessing main memory.

On our next access, we load address 2. We split address 2 (0b0010) into a tag of 0b001 and a block offset of 0b0. Comparing the tag with the tags of line 0 and line 1 tells us that the desired block is not in the cache, and thus we have a **cache miss**. In this case, we will load the block containing addresses 2 and 3 into the cache. However, since the cache is full, we have to choose a block to evict. This will take place utilizing the **least recently used** eviction policy. The least recently used block is line 1 (since we accessed address 1 after address 4), and thus we will evict line 1 and replace it with our new block. The new state of the cache is now:

Line 0, Block Offset 0: **address 0**
Line 0, Block Offset 1: **address 1**
Line 1, Block Offset 0: **address 2**
Line 1, Block Offset 1: **address 3**

Next, we write to address 5. We split address 5 (0b0101) into a tag of 0b010 and a block offset of 0b1. We again will perform tag comparison with the blocks in the cache; this access is a **cache miss**. Since our cache is **allocate-on-write**, we will allocate a block for address 5. Since the cache is **write-through**, we will send this write to main memory. Again, we will evict the least recently used block, which is line 0 in this case. Thus, the new state of our cache is shown below:

Line 0, Block Offset 0: **address 4**
Line 0, Block Offset 1: **address 5**
Line 1, Block Offset 0: **address 2**
Line 1, Block Offset 1: **address 3**

Finally, we read address 3. We split address 3 (0b0011) into a tag of 0b001 and a block offset of 0b1. We find that this tag matches the tag of line 1, and therefore we have a **cache hit**. The final state of our cache is thus:

Line 0, Block Offset 0: **address 4**
Line 0, Block Offset 1: **address 5**
Line 1, Block Offset 0: **address 2**
Line 1, Block Offset 1: **address 3**

Classifying Cache Misses

To optimize the cache, we want to minimize the number of cache misses that take place. We can profile our cache by analyzing an access pattern and grouping the misses that occur into three general categories: **compulsory**, **capacity**, and **conflict** misses. Let's take a look at what each of these categories comprises:

- **Compulsory misses** are misses that, as the name suggests, cannot be avoided. These misses take place because the corresponding block in memory has never been accessed before, and thus even an infinitely sized cache would not result in a cache miss.
- **Capacity misses** are misses that arise due to the finite size of the cache. Capacity misses try to access an address that has been brought into the cache previously, but was evicted by other blocks before being accessed again.
- **Conflict misses** are misses that arise due to the finite size of sets. Conflict misses try to access an address that has been brought into the cache previously, but was evicted by other blocks before being accessed again. The difference between capacity and conflict misses lies in the number of accesses in between the previous access and the

cache miss: a capacity miss will result in a cache miss when the same access pattern is simulated on a fully associative cache of the same size, while a conflict miss will result in a cache hit when the same access pattern is simulate on a fully associative cache of the same size.

Let's take a look at the previous example:

```
load   0     cache miss
load   4     cache miss
load   1     cache hit
load   2     cache miss
store  5     cache miss
load   3     cache hit
```

Let's classify the above cache misses as compulsory, capacity, or conflict misses. To perform this classification, we will first simulate this access pattern on an infinite cache. We obtain the following results:

```
load   0     cache miss
load   4     cache miss
load   1     cache hit
load   2     cache miss
store  5     cache hit
load   3     cache hit
```

Notice that our store to address 5 now becomes a cache **hit**, since the block was brought in previously. The remainder of our original misses (load 0, load 4, load 2) are cache misses. Thus, we can classify load 0, load 4, and load 2 as **compulsory misses**.

Next, we simulate this access pattern on a fully associative cache. Since our cache is already fully associative, we will get the exact same results. Thus, the remaining cache miss (store 5) is a **capacity** miss.

How can we use the type of cache miss to optimize our cache? If we see a lot of conflict misses, we can increase our associativity. This allows more blocks to be stored in one set, meaning that they will evict each other less often. If we see a lot of capacity misses, we can increase our cache size. This allows more blocks to be stored in the cache, meaning that it takes longer to fill up the entire cache. If we see a lot of compulsory misses, we can increase our block size. This increases the number of addresses loaded into the cache at once, meaning that more addresses are already loaded into the cache.

Chapter 9: Virtual Memory

Over the last 8 chapters, we've built up a huge set of tools that allow us to design and run programs in assembly language. Our program can access the full memory space available on the machine. However, on modern computers, we often have hundreds of processes running at the same time, and they have to share the computer's memory. How can we allow processes to share memory? Let's start with a simple idea: each process is allocated a certain region of memory.

addresses 0x0000 through 0x3FFF: process A
addresses 0x4000 through 0x7FFF: process B
addresses 0x8000 through 0xBFFF: process C
addresses 0xC000 through 0xFFFF: process D

Let's consider the following (very simple) program:

```
            halt
one         .fill       1
oneAdr      .fill       one
```

If we convert this to machine code, we get the following:

```
            25165824
            1
            1
```

If we had to run this program in a different region of memory, we would want to update the value of **oneAdr**, but not the value of **one**! However, there's nothing different between the machine code for **one** and for **oneAdr** that would help us determine which one should be updated. We could propose a solution where each executable includes information about which lines are constants versus which lines are address references, but that incurs a lot of overhead and slows down loading significantly. We need a better strategy. At first, this seems a bit problematic. Changing the base address of a program is doable but tricky, but only a single process can occupy each memory address at the same time. Our solution, as counterintuitive as it sounds, is to give every process the *illusion* that it is the only process on the machine, and that it has the entire memory space available. The operating system helps create this illusion, through the use of **virtual memory**.

The intuition behind virtual memory is very simple: the process will attempt to access an address space starting at 0. The operating system intercepts each memory access, maps it to the "true" address, and returns the value at the "true" address. Thus, the process believes that

it is the only process running, and that it has the ability to access the entirety of the machine's memory. Let's see how this is accomplished in practice.

Units of Memory

When discussing virtual memory, we will often see much larger units of memory, such as the kilobyte (KB), megabyte (MB), and gigabyte (GB). In these notes, we will define these as 2^{10}, 2^{20}, and 2^{30} bytes, respectively[19]. Thus, when we convert a quantity such as 32 GB to bytes, we can consider this as 32 * 2^{30} B, or 2^5 * 2^{30} B. Thus, 32 GB = 2^{35} B. We can perform a similar conversion for quantities in units of KB or MB.

Designing Virtual Memory

When we work with virtual memory, we will have two address spaces: the **physical address space** and the **virtual address space**. The **physical address space** corresponds to the machine's memory, while the **virtual address space** corresponds to the illusion of an address space that the operating system provides to each process. When using virtual memory, processes access addresses in the **virtual** address space. The operating system captures these requests, and converts the **virtual address** the process requested to a **physical address** in memory. Then, the operating system looks up the physical address in the **physical** address space to get the correct memory value.

If we have a sufficiently large physical address space, we can map all virtual address spaces onto the physical address space uniquely. However, this typically isn't feasible. Let's take a look at an example. The M1 Max chip supports up to 64 GB of RAM. If we run 512 processes (a low estimate), each process is allocated 128 MB of memory. However, even if we assume our processes are 32-bit, each will have a 4 GB (2^{32}) virtual address space! We simply don't have the space to store everything we need in RAM. Thus, we have to allocate space on the **disk** to store all of the memory used by each process. Problem solved!

Except, we've created a new problem. Accessing the disk is very, very, very slow. A processor will stall millions of cycles for each disk access. In comparison, accesses to RAM take tens or hundreds of cycles. We want to find a way to utilize faster memory (RAM) as much as possible, while storing a larger amount of data in slower memory (disk). This should sound familiar: it's the same principle we built caching on. When designing virtual memory, we can take some concepts from caching. Cache blocks become **pages** in the physical and virtual address spaces. We assign each page a **physical page number (PPN)** or a **virtual page number (VPN)** depending on which address space it is in. How do we organize all these pages?

For every process, the operating system keeps a **page table**, which maps virtual pages to physical pages. The page table takes the form of an array, indexed using the VPN. **Page table**

[19] Another common definition for these units are 10^3, 10^6, and 10^9 bytes respectively. We choose to use the powers of two since it makes calcuating bits of an address field simpler.

entries contain information about the physical page mapped to a specific virtual page. When the process makes a memory access, the operating system will use the page table to determine where in physical memory the requested memory is located. To begin, we'll examine a **single level page table**. We'll see how multiple levels come into play later on.

In a single level page table, our page table is a large array of page table entries, each corresponding to a single virtual page number. Let's consider the following situation:

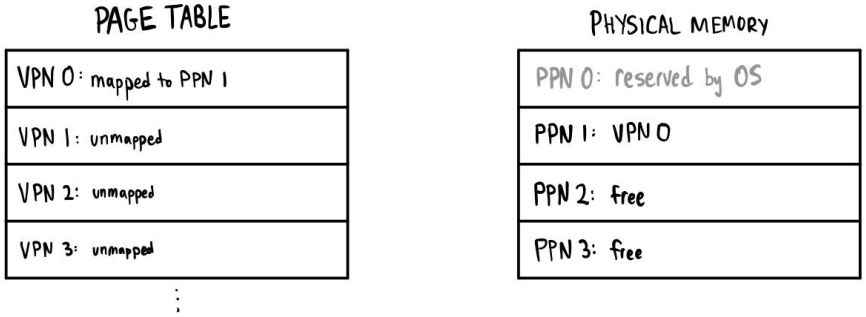
- Processes have access to a 256 B address space
- The machine has 64 B of RAM available
- Each page is 16 B in size

Let's first consider how many pages are in the virtual and physical address spaces. In the virtual address space, we have 256 B / 16 B = 16 virtual pages, which will be labeled VPN 0 through 15. Similarly, in the physical address space, we have 64 B / 16 B = 4 physical pages, which will be labeled PPN 0 through 3. Our page table thus will contain 16 page table entries, one for each virtual page.

The process now makes a memory access to address 0x00. Let's take a look at what happens behind the scenes here[20]:

First, the operating system receives the process's request for virtual address 0x00. The **memory management unit (MMU)** handles this request, and starts by splitting the virtual address into bits for the virtual page number and the **page offset**. Similar to how we split addresses into a tag and a page offset in caching, we can split address 0x00 into a VPN of 0b0000 and a page offset of 0b0000. Recall that since we have 16 B pages, our page offset will comprise of 4 bits, since $2^4 = 16$.

Next, the MMU looks up the mapping for the virtual page number in the page table. It finds that VPN 0 is not currently mapped to any physical pages, thus causing a **page fault**. Upon a page fault, the MMU will choose a physical page to assign VPN 0 to. In this case, let's assume that PPN 0 is reserved by the operating system; we will assign VPN 0 to PPN 1. The state of our page table and physical memory now is shown below.



[20] This process is covered in depth in EECS 482 (Operating Systems) as well; if you're interested in learning more about virtual memory, take 482!

From here, we can continue making additional memory accesses, and performing the same process. We continue to map virtual pages to physical pages, until we fill up all of physical memory. At this point, we don't have space left in physical memory to map all out virtual pages. We again will take inspiration from caching: we evict the least recently used physical page to the disk, and replace it with the new page we want to access[21]. Everything here is kept track of by page table entries in the page table. Page table entries contain information regarding the mapping from virtual pages to physical pages. First, we will have the physical page number, or a physical address (we'll see when we use one or the other). Next, we have a **valid bit**, which indicates if the virtual page is mapped to a physical page or not. Finally, we have additional overhead bits. These perform other bookkeeping functions for the MMU, which we won't go into too much detail about. However, we will incorporate these overhead bits into the calculation of page table entry sizes.

Page Table Entries

Let's consider a larger virtual memory system, described as follows:

- The virtual address space is a 48-bit address space
- The system has 4 GB of physical memory available
- The page size is 4 KB
- Page table entries consist of a physical page number, a valid bit, and 3 overhead bits. The page table size is rounded upwards to the nearest power of two.

To approach this problem, we will start by determining the number of bits in a physical page number and in a virtual page number. Pages are 4 KB, or 2^{12} B in size. Thus, our page offset comprises the 12 least significant bits of the address. First, let's consider the virtual address: 48 bits, 12 of which are used for the page offset. Thus, we have a 36 bit virtual page number. Next, we consider the physical address: 32 bits, 12 of which are used for the page offset. Thus, we have a 20 bit physical page number.

From here, we can calculate the size of the page table entry. We know that physical page numbers are 20 bits in size. Additionally, we have a valid bit (1 bit, total 21 bits), and 3 bits of overhead (total 24 bits). Thus, the page table size is 24 bits, or 3 bytes. We round this to the nearest power of 2, giving us 4 bytes.

From here, we can calculate the size of a single level page table. Remember that the single level page table is an array of page table entries, one for every virtual page number. We can find the number of virtual pages by dividing the virtual address space size by the page size. We take 2^{48} B / 2^{12} B to get 2^{36} virtual pages. Multiplying this by 4 bytes gives us **2^{38} bytes** in the single level page table.

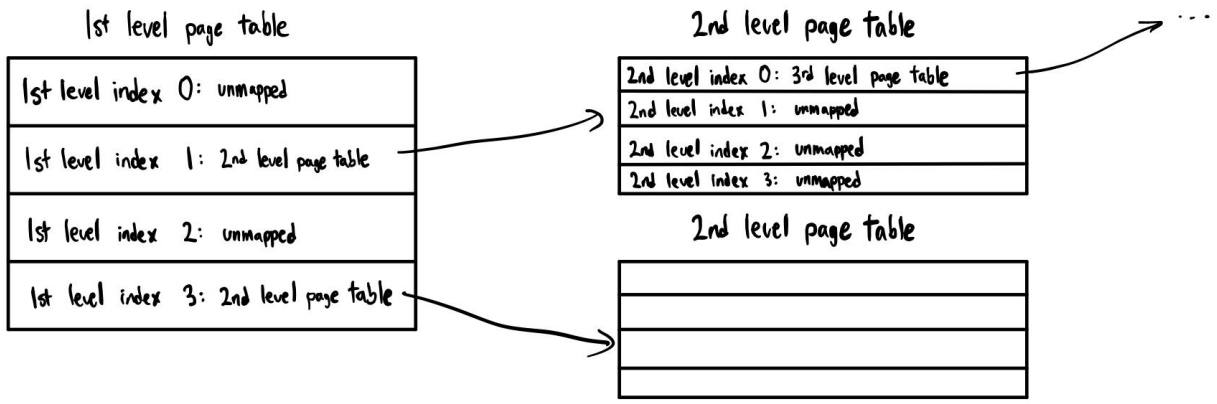
[21] Some more sneak peeks of EECS 482: keeping track of LRU bits is often extremely costly and finding the true LRU takes a lot of time, so many operating systems will utilize algorithms such as pseudo-LRU or clock queues. Check out https://en.wikipedia.org/wiki/Cache_replacement_policies if you're interested!

We don't even have enough physical memory to store the page table! Let's try redesigning the page table to utilize less memory. What can we optimize? To save memory, we'll utilize **multi-level page tables**.

Multi-Level Page Tables

"We can solve any problem by introducing an extra level of indirection"
- [Fundamental Theorem of Software Engineering]

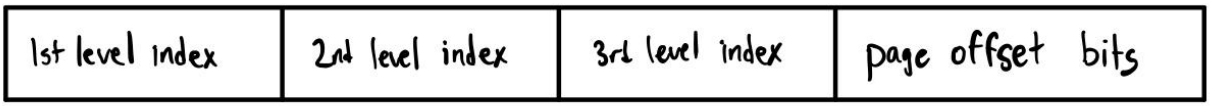
Let's try introducing a few more levels of indirection. In a single-level page table, the page table contains page table entries to every single page. Now, we will have various levels of page tables, each containing page table entries to the next level. The very last level of page tables will contain page table entries to all available pages. The overall design will look similar to the following:



How do we index into a multi-level page table? Let's think about how we indexed into a single-level page table:

index into page table = VPN bits

For a multi-level page table, we can "break up" the VPN bits into various subindices, which correspond to the index at each level:

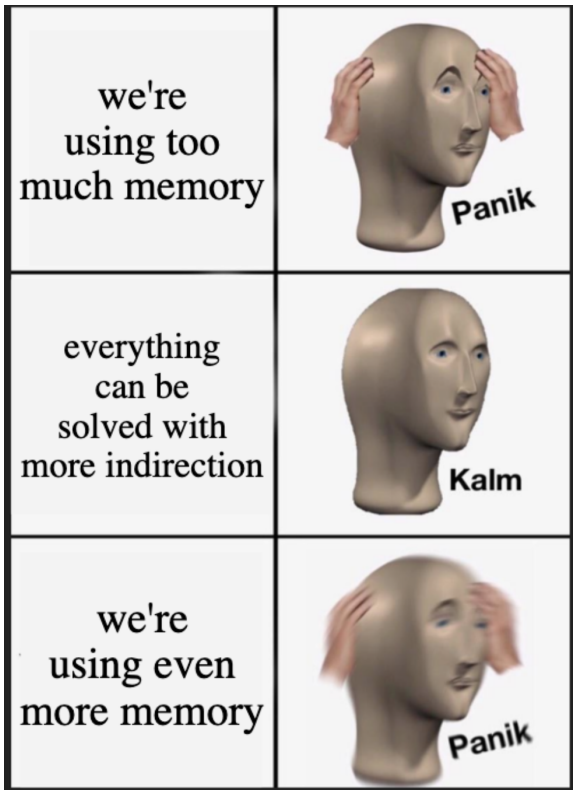


index into 1st level page table = 1st level index
index into 2nd level page table = 2nd level index
index into 3rd level page table = 3rd level index

Let's take a look at the prior example again, except we now utilize a three-level page table, where each page table contains 2^{12} page table entries. Let's take a look at the total number of page tables we'll have:

- 1st level page table: **1** with 2^{12} entries = **2^{14} B**
- 2nd level page tables: **2^{12}** with 2^{12} entries each = **2^{26} B**
- 3rd level page tables: **2^{24}** with 2^{12} entries each = **2^{38} B**

Our total size is therefore **2^{38} + 2^{26} + 2^{14} B**. Did we save memory? Our single-level page table took up... 2^{38} B. Hang on...



Wait, didn't you say multi-level page tables would save memory? They indeed do, in the common case[22]. With multi-level page tables, one optimization we can make is to allocate page tables when they are needed, rather than allocating all the space at once. Let's take a look at another example, where we only access a few addresses.

Again, we'll use the same configuration as before: a 3-level page table, with each page table containing 2^{12} page table entries. We'll access the following addresses (for the purposes of this question, we assume in the specified order, but it doesn't matter at the end)

- 0x100 000 000 000
- 0x100 000 000 370
- 0x100 001 000 000
- 0x200 000 000 000

We've conveniently written the addresses such that each group of 3 hex digits, or 12 bits, is separated by spaces. Let's take a look at the page tables allocated at each stage.

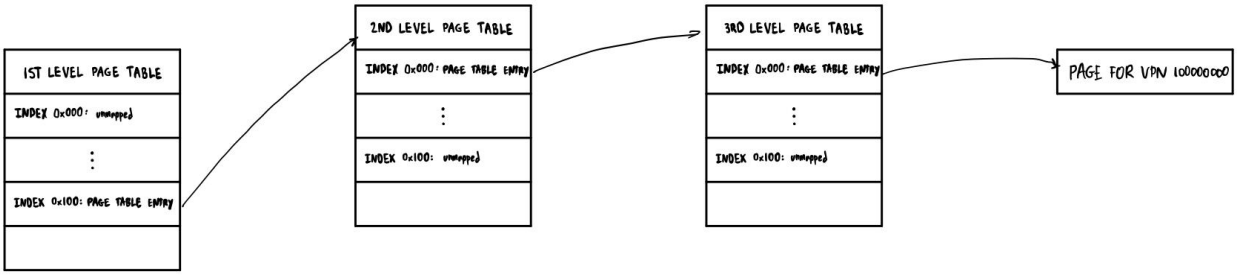
[22] Making the common case more efficient at the cost of additional resources every now and then is something we've seen over and over: storing values in registers vs. memory, caller/callee save, caching, and now page tables.

Before the first access, the only page table we have allocated is the root first level page table. This contains 2^{12} page table entries (2^{14} B), and thus our page table only takes up 2^{14} B of space.

Next, we access address 0x100 000 000 000. We check the first level page table: we do not have a page table entry corresponding to the first level index 0x100 (**page fault!**). Thus, we create this page table entry, as well as a **second level page table** that services all addresses with a first level index of 0x100. We allocate this new page table, which takes up another 2^{14} B. (Note the page table entry does not take up additional memory, it just takes up 2 B of memory that we already allocated in the first level page table)

But wait, we're not done! We look at the second level page table, and we check for a page table entry corresponding to the second level index 0x000. We just created the page table, so there is none. Thus, we do the same thing again, creating a new page table entry, as well as a **third level page table** that services all address with a first level index of 0x100 and a second level index of 0x000.

One more level to go! We look at the third level page table and check for a page table entry corresponding to the third level index 0x000. Again, we just created this page table, so there is none. We create a new page table entry. However, we've reached the last level of page tables, so we can directly allocate a **page** of memory corresponding to the virtual page number 0x100 000 000. The page offset is 0x000, and thus we return the corresponding location in physical memory.



At this point, we've allocated two new page tables, so our memory consumption increases to 3 * 2^{14} B. Still a lot lower than the 2^{38} B of a single level page table!

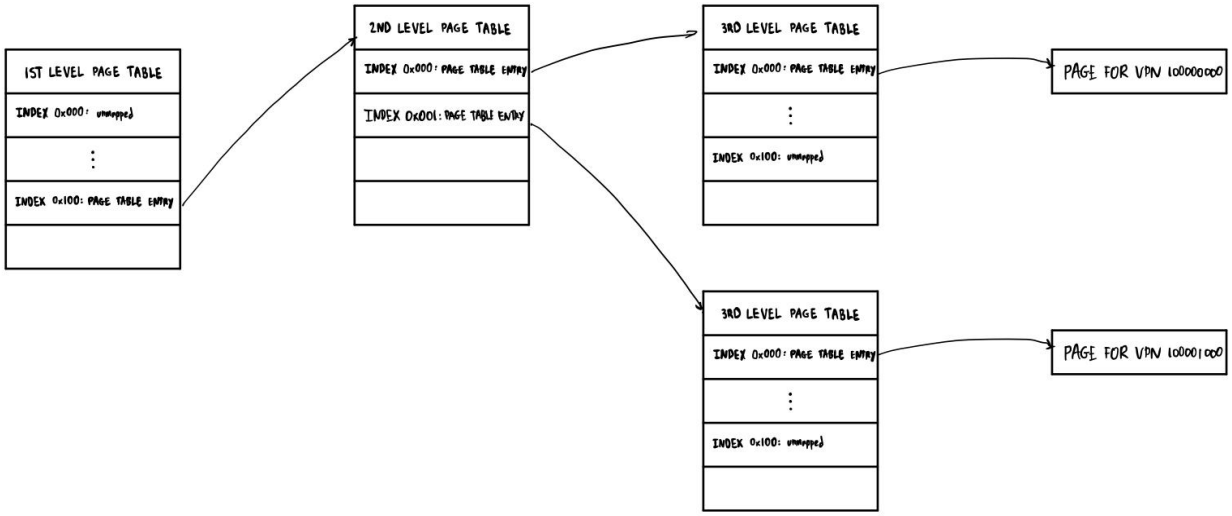
Our next access is to address 0x100 000 000 370. We check our first level page table for the first level index 0x100. At this offset, we find a page table entry (the one we just created previously!) which points us to the corresponding **second level page table**. Next, we check the second level page table for the second level index[23] 0x000. At this offset, we find a page table entry (again, we just created it) which points us to the corresponding **third level page**

[23] Yes, I'm going overboard on the emphasis. Page table walks often get confusing very quickly when different index levels get mixed up, so it's good to stay very methodical about keeping track of which level of page table you're on.

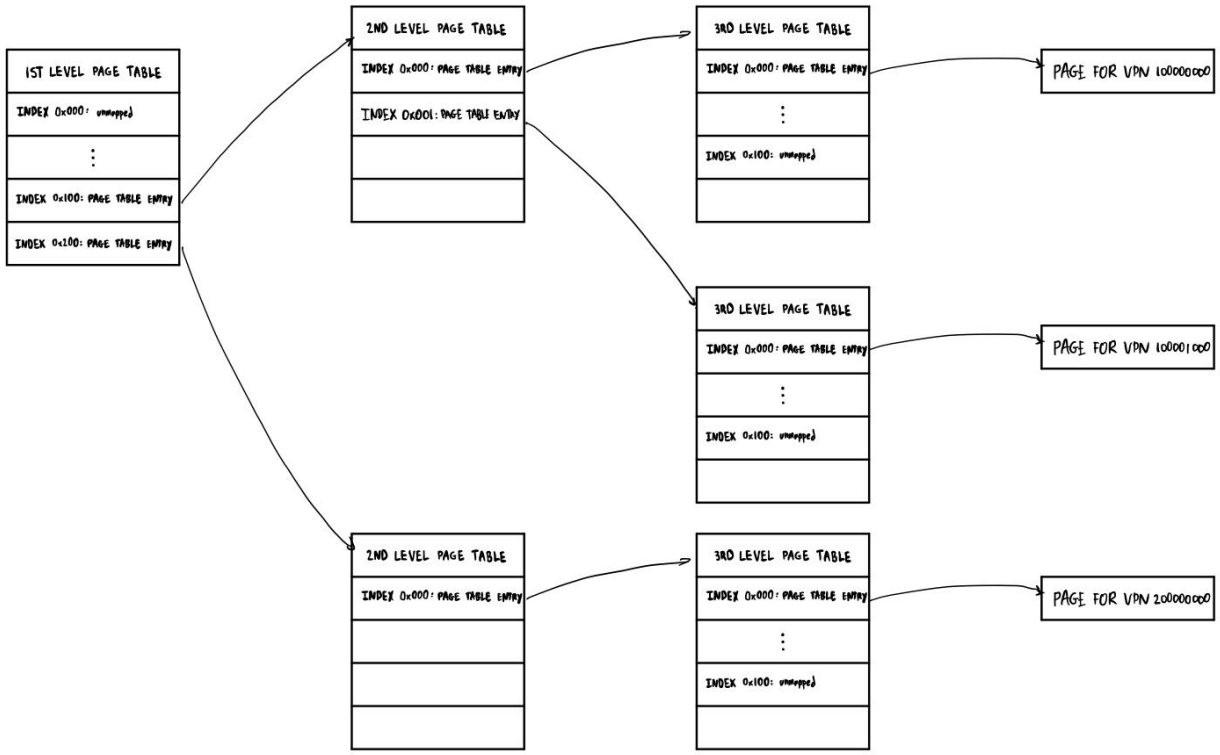
table. Finally, we check the third level page table for the third level index 0x000, and find a page table entry which points us to the page that we allocated in the previous access.

This process of accessing a page table, reading the page table entry, and following the page table entry to the next page table is called a **page table walk**. This process takes *n* * *m* time, where *n* is the number of levels in the page table hierarchy, and *m* is the time it takes to access memory. (We'll see why we noted this fact later)

Next, we access address 0x100 001 000 000. We look up the first level index 0x100 in the first level page table, which gives us a page table entry corresponding to the **second level page table** we created previously. Next, we look up the second level index 0x001 in the second level page table, which tells us that there is no page table entry corresponding to 0x001 (**page fault!**). Thus, we have to create a new page table entry in the second level page table, and allocate a new **third level page table**. Our page table walk continues, and we create a new third level page table entry as well as allocate our new page. So far, the memory taken up by the page table is at 4 * 2^{14} B!



Finally, we access address 0x200 000 000 000. The first level index 0x200 does not have a corresponding page table entry (**page fault!**), and thus we allocate new second **and** third level page tables. Even though the second and third level indices are the same as our existing second and third level page tables, we cannot reuse these since the existing page tables service a first level index of **0x100**, while the new tables have to service a first level index of **0x200**. Thus, we allocate two more page tables, increasing our total to 6 * 2^{14} B.



As we can see from this example, multi-level page tables provide significant space savings in the common case. Only when almost all virtual addresses are being used (which occurs very rarely in practice) does the multi-level page table perform worse than the single-level page table in terms of memory usage.

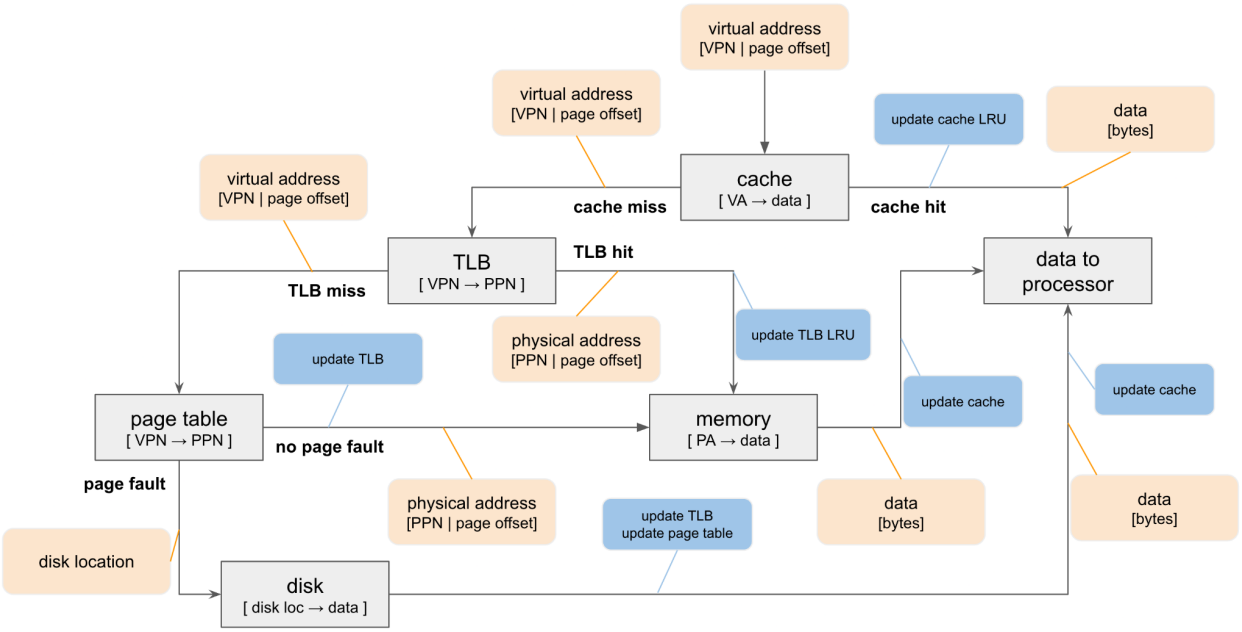
Caching with Virtual Memory

In the previous chapter, we discussed caching, where we can store a subset of the contents of memory in faster SRAM to improve our access time.

Now, we have virtual memory. Currently, to access any virtual address, we have to perform a page table walk. Let's speed up the common case again and introduce caching. To handle caching alongside virtual memory, we have two options, corresponding to the two different types of addresses:

- **Physically-addressed** caches use the **physical address** as the cache address. Physically-addressed caches require virtual addresses to be resolved to physical addresses *before* a cache lookup.
- **Virtually-addressed** caches use the **virtual address** as the cache address. Virtually-addressed caches can directly use the address a program requests (remember, programs operate in the virtual address space), but have to be cleared whenever the operating system switches between processes (we refer to this as a **context switch**).

Let's take a look at how these caches work, starting with the virtually addressed cache. The workings of a virtually addressed cache are shown in the diagram below[24].



Woah, what does this all mean? Let's break it down step by step.

Our overall goal is to convert a **virtual address** requested by a program into **data** bytes. Since our cache is virtually addressed, we can directly give the cache our virtual address, and see if we're able to get a cache hit (right branch). If we have a cache hit, we directly have our data, and we can send it to our processor. On the other hand, if we have a cache miss (left branch), we know we have to perform the page table walk.

Or do we? Following the left branch, we see a new component, called the **Translation Lookaside Buffer**, or **TLB**. The TLB acts as a second cache for page table lookups, by storing virtual page number to physical page number translations. Thus, we only have to perform a page table walk when the TLB cannot find our desired virtual page number.

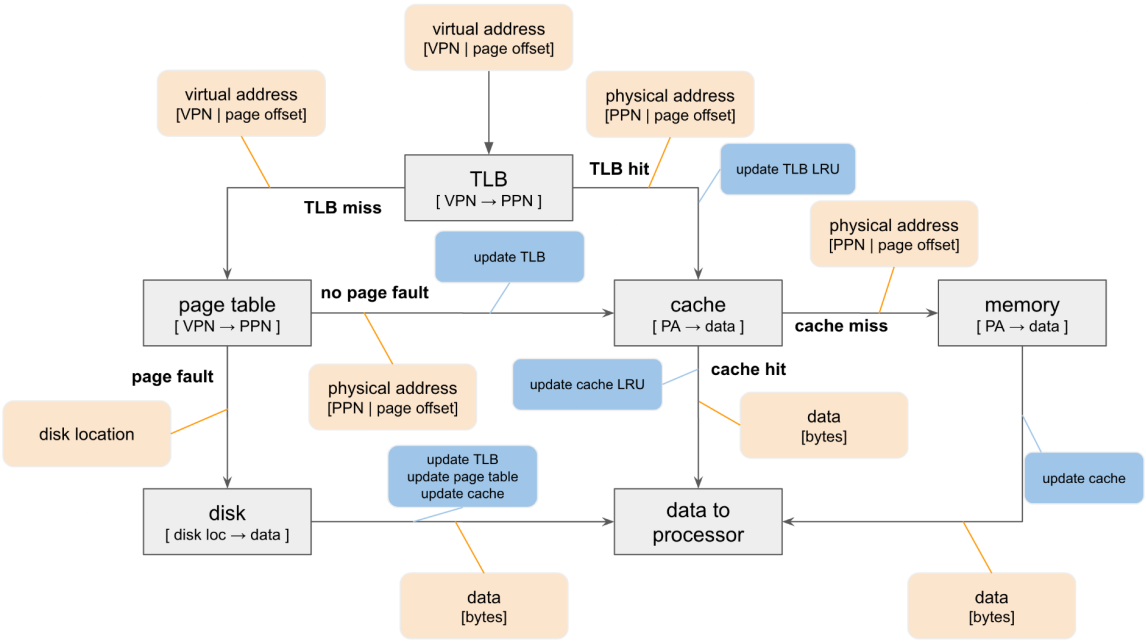
If we have a TLB miss (left branch), we proceed to a page table walk. If the page table walk results in a page fault, we access the disk to find our data[25]. Otherwise, we have our physical page number, which can be used to address physical memory to find the desired data. If we have to go to disk, we have to update the page table, as well as the TLB. Otherwise, we can simply update the TLB.

[24] Fun trivia fact: you may notice that these are some of the only diagrams here that aren't hand-drawn. I created these in Spring 2021, but they never made it to the discussion slides :(
[25] In 370, we assume that we can find everything in the disk. Take EECS 491 (Distributed Systems) to see some examples of when you'll have to find data on other computers.

If we have a TLB hit (right branch), we can skip the page table walk, and directly proceed to physical memory. In this case, we do not need to add the virtual page lookup to the TLB, but rather we update its LRU indexes (or equivalent).
Regardless of if the TLB hits or misses, we'll have to update the cache with the new data. This ensures that future accesses to this address will be resolved quicker.

Notice that on every path, we perform updates to the cache, TLB, and/or page table. These updates do take time, but we'll ignore them since they can be performed in parallel to executing the instruction. Additionally, many processors will perform the cache and TLB lookups in parallel. If a cache miss occurs, the TLB result is ready; if a cache hit occurs, the TLB result is simply discarded.

Next, we have the physically addressed cache.



Having walked through the virtually addressed cache, the physically addressed cache is very similar. In fact, we can just think of the cache being "moved" from the initial entry point to after a physical address has been resolved. Thus, we start with a TLB lookup, which either directs us to a page table walk or to the cache. From here, the page table either raises a page fault (leading us to disk and thus giving us the data directly) or gives us a physical address. The physical address is then given to the cache. If a cache hit occurs, we can send the data directly to the processor. Else, we resolve the physical address in memory, and return the data. Again, the required updates to cache, TLB, and/or page table are shown above.

Average Memory Access Time (AMAT)

Previously, we've evaulated caches by their hit/miss rate: a cache that a better hit rate will, all other things being equal, perform better than a cache with a worse hit rate. However, we now have two different sources of hit rates: the cache and the TLB. How can we compare, for example, a physically addressed cache with a 95% cache hit rate and a 90% TLB hit rate; versus a virtually addressed cache with a 90% cache hit rate and a 95% TLB hit rate? To perform this comparison, we turn to the **average memory access time**, which is the average time it takes for a virtual address lookup to return data.

How can we calculate the AMAT? Let's try it out with a physically addressed cache:

- The TLB takes 10 ns to access, and has a 90% hit rate.
- The cache takes 20 ns to access, and has a 95% hit rate.
- Main memory takes 100 ns to access.
- We have a single-level page table that raises a page fault on 1% of accesses.
- The disk takes 1,000,000 ns to access.

The key in these problems is to work "backwards" on the cache diagram, starting from Data to Processor and working slowly up towards the initial entry point. Let's start by finding the average time it takes the cache to resolve a physical address:

The lookup in the cache takes **20 ns**. In 95% of cases, we can directly proceed to Data to Processor, but in **5%** of cases, we have an extra **100 ns** of delay due to a main memory access. Thus, the average time to access the cache is **20 ns** + **5%** * **100 ns** = **25 ns**.

Next, we can determine the average time it takes the page table to resolve a virtual address:

The page table walk requires (1 level[26]) * (100 ns) = **100 ns**.[27] In 1% of cases, we will incur a page fault and access the disk (**+1,000,000 ns**). In the other 99% of cases, we proceed to the cache, which we determined has a **25 ns** average access time (notice that these averages can be computed independently of each other). Thus, we have a total average access time for the page table of **100 ns** + **1%** * **1,000,000 ns** + **99%** * **25 ns** = **10,124.75 ns**.

Finally, we can find the AMAT starting from the root of the tree: the TLB.

TLB lookup requires **10 ns**. In 90% of cases, we can proceed to the cache (**25 ns**), while in 10% of cases, we proceed to a page table walk (**10,124.75 ns**). Thus, our AMAT is:

10 ns + **90%** * **25 ns** + **10%** * **10,124.75 ns** = **1044.975 ns**.

---

[26] In more complex problems with multi-level page tables, the percentage of page faults at each level will typically be specified. A page fault at the first level invokes only a single memory access, while a page fault at the second level invokes two memory accesses, and so on.

[27] Remember when we noted down that the time for a page table walk was (*levels*) * (*memory access time*)? This is why!

How does this compare to the virtually addressed cache we mentioned previously? Let's take a look.

Again, we'll use the same specifications, with slight modifications:

- The TLB takes 10 ns to access, and has a **95%** hit rate.
- The cache takes 20 ns to access, and has a **90%** hit rate.
- Main memory takes 100 ns to access.
- We have a single-level page table that raises a page fault on 1% of accesses.
- The disk takes 1,000,000 ns to access.

Now, since the cache is our first access, we start by computing the AMAT of the page table. In this case, the page table walk takes **100 ns**, and in **1%** of cases, we require a **1,000,000 ns** disk access. Otherwise, we make another main memory access (**99%** of cases, **100 ns**). Thus, our AMAT is **100 ns** + **1%** * **1,000,000 ns** + **99%** * **100 ns** = **10,199 ns**.

Next, we can consider the TLB's AMAT. We have a **10 ns** access time, with a **95%** hit rate which will send us to main memory (**100 ns**). In the remaining **5%** of cases, we will perform a page table walk (**10,199 ns**). Thus, our AMAT is **10 ns** + **95%** * **100 ns** + **5%** * **10,199 ns** = **614.95 ns**.

Finally, we can compute the overall AMAT. The cache takes **20 ns** to access, and has a **90%** hit rate which will send data directly to the processor. In **10%** of cases, we will have to perform a TLB lookup (**614.95 ns**). Thus, our AMAT is **20 ns** + **10%** * **614.95 ns** = **81.495 ns**.

Woah, that's a huge difference. What's the reason behind this? The key advantage of the virtually-addressed cache in this case is that the cache is placed directly at the beginning, meaning that in the majority of cases, there is no need to ever perform a page table lookup or disk access. On the other hand, the physically addressed cache requires additional time to look up the physical page number translation, which often leads to additional time spent looking up in the page table or on disk.

Notice that this isn't actually the fastest we could make our virtually addressed cache; if we consider the aforementioned optimization of running the TLB and cache in parallel, we can shave another **1 ns** off of the AMAT. In general, virtually addressed caches will run faster because of these optimizations and since there is no need to perform a virtual to physical page number translation before accessing the cache. However, virtually addressed caches have to be invalidated upon context switches, which creates a tradeoff[28].

[28] Modern CPUs often utilize a Virtually Indexed, Physically Tagged cache, which is out of the scope of these notes. If you're curious, check out this article which provides a general overview of the VIPT cache: https://www.geeksforgeeks.org/virtually-indexed-physically-tagged-vipt-cache.

Recap, Tips and Tricks

Virtual memory problems are often very diverse in form, so it's not possible to go through every single possible format they can appear in. However, one way to master virtual memory is to understand the concepts behind page tables, and how to convert between different forms of information.

Some of the key formulas to remember:

- Virtual Address size (bits) = Virtual Page Number size (bits) + Page Offset size (bits)
- Physical Address size (bits) = Physical Page Number size (bits) + Page Offset size (bits)
- Virtual Address size (bits) = total size of Level Indexes (bits) + Page Offset size (bits)
    ex. if we have a 3-level page table, where each level has 2^{12} entries (12 bits), the virtual address size is 12 + 12 + 12 + # of page offset bits

- # entries in a **single level** page table = $2^{\text{Virtual Page Number size (bits)}}$
- # entries in a **multi level** page table = $2^{\text{Level Index size (bits)}}$
- Total # of virtual pages addressable = $2^{\text{Virtual Page Number size (bits)}}$
- Total # of physical pages available = $2^{\text{Physical Page Number size (bits)}}$

- Physical Page Number size (bits) = $\log_2$(Physical Memory Size / Page Size)
- Virtual Page Number size (bits) = $\log_2$(Virtual Memory Size / Page Size)
- Page Offset size (bits) = $\log_2$ (Page Size)

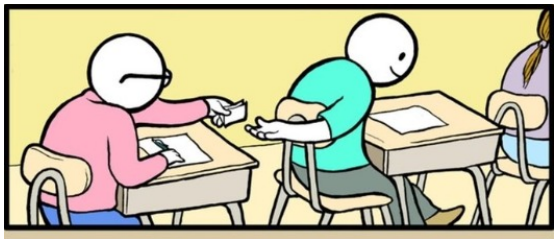
Finally, a list of abbreviations I personally use. It's not required to use these, but I find that they help shorten up the notation associated with virtual memory.

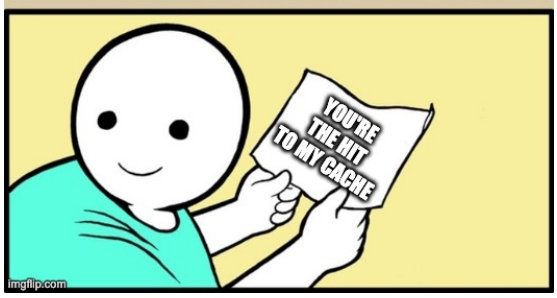
VPN = Virtual Page Number
PPN = Physical Page Number
VA = Virtual Address
PA = Physical Address

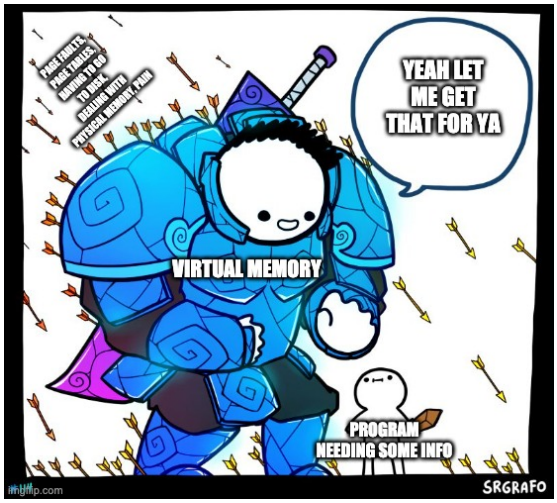
PTE = Page Table Entry
1LPT = First Level Page Table, 2LPT = Second Level Page Table, ...

Chapter 10: It's over!

That's it. There's no more content left to cover. If you're reading to this point studying for the final, I wish you the best of luck! Make sure to drink water, take adequate breaks, and don't pull too many all-nighters. Here's some memes from when I took the course (credit @aisaav)







And finally, a few closing credits:

A huge thank you to all of the EECS 370 staff who helped review this document. In no particular order:

- Alexandra Saavedra
- Christian Ghoubrial
- Mason Nelson
- Maximos Nolan
- Nathan Kubczak
- Trevor Mudge
- Yumeng Bai

Thank you to Andrew Zhou for giving me the motivation to finish the virtual memory portion of these notes, and get this to a releasable form. It truly is a win-win scenario, as you said :)

And finally, to you, the reader, thank you for taking the time and going through these notes. I don't know when you'll be reading this, but I hope you find them useful.