Binary Representations and Assembly Basics

University of Waterloo

January 15, 2020
• What does the 8-bit binary sequence 10100011 represent?
• Binary data has no inherent meaning: we need to be told, come up with, or assume an interpretation for it.
• **Exercise:** Give five different possible ways we could interpret the 4-bit binary sequence 1000.

**Solution:** Here are some “natural” ways of interpreting this sequence:

• As an unsigned 4-bit number, this is 8.
• As a signed (two’s complement) 4-bit, this is $-8$.
• An array of four booleans, where the first is “true” and the next three are “false”.
• The “backspace” character (ASCII code 8).
• The address of the third word in the MIPS machine’s memory, where the first word is at address zero.

However, bits can mean anything you want. You could write a program that interprets this bit sequence as a picture of a cat, a piece of music, or the decimal number one thousand.
What does the 8-bit binary sequence 10100011 represent?

Binary data has no inherent meaning: we need to be told, come up with, or assume an interpretation for it.

Exercise: Give five different possible ways we could interpret the 4-bit binary sequence 1000.

Solution: Here are some “natural” ways of interpreting this sequence:

- As an unsigned 4-bit number, this is 8.
- As a signed (two’s complement) 4-bit, this is $-8$.
- An array of four booleans, where the first is “true” and the next three are “false”.
- The “backspace” character (ASCII code 8).
- The address of the third word in the MIPS machine’s memory, where the first word is at address zero.
What does the 8-bit binary sequence 10100011 represent?

Binary data has no inherent meaning: we need to be told, come up with, or assume an interpretation for it.

**Exercise:** Give five different possible ways we could interpret the 4-bit binary sequence 1000.

**Solution:** Here are some “natural” ways of interpreting this sequence:

- As an unsigned 4-bit number, this is 8.
- As a signed (two’s complement) 4-bit, this is $-8$.
- An array of four booleans, where the first is “true” and the next three are “false”.
- The “backspace” character (ASCII code 8).
- The address of the third word in the MIPS machine’s memory, where the first word is at address zero.

However, bits can mean anything you want. You could write a program that interprets this bit sequence as a picture of a cat, a piece of music, or the decimal number one thousand.
We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

\[11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0)\]

\[= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26.\]
Converting Between Binary and Decimal

- We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

\[
11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0)
\]

\[
= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26.
\]

- We can convert positive decimal numbers to binary by repeated division by 2. For example, to convert 23:

\[
\begin{align*}
23 & \div 2 = 11 \text{ remainder } 1 \\
11 & \div 2 = 5 \text{ remainder } 1 \\
5 & \div 2 = 2 \text{ remainder } 1 \\
2 & \div 2 = 1 \text{ remainder } 0 \\
1 & \div 2 = 0 \text{ remainder } 1 \\
\end{align*}
\]

Reading the remainders from bottom to top, we get 10111 for the binary representation of 23.

\[
\text{Let's verify: } 10111 = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 23.
\]

- To write as an 8-bit binary number, add leading zeroes: 00010111.
Converting Between Binary and Decimal

- We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

\[ 11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0) \]

\[ = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26. \]

- We can convert positive decimal numbers to binary by repeated division by 2. For example, to convert 23:
  - \[ 23/2 = 11 \text{ remainder } 1 \]
We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

$$11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0)$$

$$= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26.$$

We can convert positive decimal numbers to binary by repeated division by 2. For example, to convert 23:

- $23/2 = 11$ remainder 1
- $11/2 = 5$ remainder 1
We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

\[
11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0)
\]

\[
= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26.
\]

We can convert positive decimal numbers to binary by repeated division by 2. For example, to convert 23:

- \(23/2 = 11\) remainder 1
- \(11/2 = 5\) remainder 1
- \(5/2 = 2\) remainder 1
- \(5/2 = 2\) remainder 1

Reading the remainders from bottom to top, we get 10111 for the binary representation of 23.

Let's verify: 10111 = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 23.

To write as an 8-bit binary number, add leading zeroes: 00010111.
Converting Between Binary and Decimal

- We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

\[ 11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0) \]
\[ = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26. \]

- We can convert positive decimal numbers to binary by repeated division by 2. For example, to convert 23:
  - \[ 23/2 = 11 \text{ remainder } 1 \]
  - \[ 11/2 = 5 \text{ remainder } 1 \]
  - \[ 5/2 = 2 \text{ remainder } 1 \]
  - \[ 2/2 = 1 \text{ remainder } 0 \]

  Reading the remainders from bottom to top, we get 10111 for the binary representation of 23.

  Let’s verify: 10111 = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 23.

  To write as an 8-bit binary number, add leading zeroes: 00010111.
Converting Between Binary and Decimal

- We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

  \[11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0)\]

  \[= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26.\]

- We can convert positive decimal numbers to binary by repeated division by 2. For example, to convert 23:
  - \(23/2 = 11\) remainder 1
  - \(11/2 = 5\) remainder 1
  - \(5/2 = 2\) remainder 1
  - \(2/2 = 1\) remainder 0
  - \(1/2 = 0\) remainder 1

  Reading the remainders from bottom to top, we get 10111 for the binary representation of 23.
  Let’s verify: 10111 = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 23.

- To write as an 8-bit binary number, add leading zeroes: 00010111.
Converting Between Binary and Decimal

- We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

\[
11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0) \\
= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26.
\]

- We can convert positive decimal numbers to binary by repeated division by 2. For example, to convert 23:
  - \(23/2 = 11\) remainder 1
  - \(11/2 = 5\) remainder 1
  - \(5/2 = 2\) remainder 1
  - \(2/2 = 1\) remainder 0
  - \(1/2 = 0\) remainder 1

- Reading the remainders from bottom to top, we get 10111 for the binary representation of 23.
Converting Between Binary and Decimal

• We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

\[
11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0)
\]

\[
= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26.
\]

• We can convert positive decimal numbers to binary by repeated division by 2. For example, to convert 23:

- \[23 / 2 = 11 \text{ remainder } 1\]
- \[11 / 2 = 5 \text{ remainder } 1\]
- \[5 / 2 = 2 \text{ remainder } 1\]
- \[2 / 2 = 1 \text{ remainder } 0\]
- \[1 / 2 = 0 \text{ remainder } 1\]

• Reading the remainders from bottom to top, we get 10111 for the binary representation of 23.

• Let’s verify: \[10111 = 2^4 + 2^2 + 2^1 + 2^0 = 16 + 4 + 2 + 1 = 23\].
We represent positive integers in binary using a place-value system, similar to decimal, but there are only two digits (0 and 1) and each digit corresponds to a power of 2. For example:

\[11010 = (1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0)\]

\[= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26.\]

We can convert positive decimal numbers to binary by repeated division by 2. For example, to convert 23:

- 23/2 = 11 remainder 1
- 11/2 = 5 remainder 1
- 5/2 = 2 remainder 1
- 2/2 = 1 remainder 0
- 1/2 = 0 remainder 1

Reading the remainders from bottom to top, we get 10111 for the binary representation of 23.

Let's verify: \[10111 = 2^4 + 2^2 + 2^1 + 2^0 = 16 + 4 + 2 + 1 = 23.\]

To write as an 8-bit binary number, add leading zeroes: 00010111.
Exercises:

- Convert the 8-bit binary number 01101001 into decimal.

**Solution:**

\[
\begin{align*}
2^7 &+ 2^5 + 2^3 + 2^0 = 128 + 32 + 8 + 1 = 169.
\end{align*}
\]

- Convert 35 into 8-bit binary.

**Solution:**

35 breaks down into 32 + 2 + 1 = 2^5 + 2^1 + 2^0, which gives 00100011. You can also obtain this by repeated division.

- Convert 216 into 8-bit binary.

**Solution:**

Using the division method:

\[
\begin{align*}
216 &/ 2 = 108 \text{ remainder } 0 \\
108 &/ 2 = 54 \text{ remainder } 0 \\
54 &/ 2 = 27 \text{ remainder } 0 \\
27 &/ 2 = 13 \text{ remainder } 1 \\
13 &/ 2 = 6 \text{ remainder } 1 \\
6 &/ 2 = 3 \text{ remainder } 0 \\
3 &/ 2 = 1 \text{ remainder } 1 \\
1 &/ 2 = 0 \text{ remainder } 1
\end{align*}
\]

This gives 11011000. Working backwards, we verify that 2^7 + 2^6 + 2^4 + 2^3 = 128 + 64 + 16 + 8 = 216.
Exercises:

- Convert the 8-bit binary number 01101001 into decimal.

  **Solution:** \(2^6 + 2^5 + 2^3 + 2^0 = 64 + 32 + 8 + 1 = 105\).
Exercises:

- Convert the 8-bit binary number 01101001 into decimal.
  **Solution:** \(2^6 + 2^5 + 2^3 + 2^0 = 64 + 32 + 8 + 1 = 105\).

- Convert 35 into 8-bit binary.
  **Solution:**
  35 breaks down into 32 + 2 + 1 = 2^5 + 2^1 + 2^0, which gives 00100011. You can also obtain this by repeated division.

- Convert 216 into 8-bit binary.
  **Solution:**
  Using the division method:
  - \(216 / 2 = 108\) remainder 0
  - \(108 / 2 = 54\) remainder 0
  - \(54 / 2 = 27\) remainder 0
  - \(27 / 2 = 13\) remainder 1
  - \(13 / 2 = 6\) remainder 1
  - \(6 / 2 = 3\) remainder 0
  - \(3 / 2 = 1\) remainder 1
  - \(1 / 2 = 0\) remainder 1
  This gives 11011000. Working backwards, we verify that \(2^7 + 2^6 + 2^4 + 2^3 = 128 + 64 + 16 + 8 = 216\).
Exercises:

- Convert the 8-bit binary number 01101001 into decimal.
  **Solution:** $2^6 + 2^5 + 2^3 + 2^0 = 64 + 32 + 8 + 1 = 105$.

- Convert 35 into 8-bit binary.
  **Solution:** 35 breaks down into $32 + 2 + 1 = 2^5 + 2^1 + 2^0$, which gives 00100011. You can also obtain this by repeated division.
Exercises:

- Convert the 8-bit binary number 01101001 into decimal.
  **Solution:** $2^6 + 2^5 + 2^3 + 2^0 = 64 + 32 + 8 + 1 = 105$.

- Convert 35 into 8-bit binary.
  **Solution:** 35 breaks down into $32 + 2 + 1 = 2^5 + 2^1 + 2^0$, which gives 00100011. You can also obtain this by repeated division.

- Convert 216 into 8-bit binary.
  Using the division method:
  - $216 / 2 = 108$ remainder 0
  - $108 / 2 = 54$ remainder 0
  - $54 / 2 = 27$ remainder 0
  - $27 / 2 = 13$ remainder 1
  - $13 / 2 = 6$ remainder 1
  - $6 / 2 = 3$ remainder 0
  - $3 / 2 = 1$ remainder 1
  - $1 / 2 = 0$ remainder 1

  This gives 11011000. Working backwards, we verify that $2^7 + 2^6 + 2^4 + 2^3 = 128 + 64 + 16 + 8 = 216$. 
Converting Between Binary and Decimal

Exercises:

- Convert the 8-bit binary number 01101001 into decimal.
  **Solution:** \(2^6 + 2^5 + 2^3 + 2^0 = 64 + 32 + 8 + 1 = 105\).

- Convert 35 into 8-bit binary.
  **Solution:** 35 breaks down into 32 + 2 + 1 = \(2^5 + 2^1 + 2^0\), which gives 00100011. You can also obtain this by repeated division.

- Convert 216 into 8-bit binary.
  **Solution:** Using the division method:
  - 216/2 = 108 remainder 0
  - 108/2 = 54 remainder 0
  - 54/2 = 27 remainder 0
  - 27/2 = 13 remainder 1
  - 13/2 = 6 remainder 1
  - 6/2 = 3 remainder 0
  - 3/2 = 1 remainder 1
  - 1/2 = 0 remainder 1

  This gives 11011000. Working backwards, we verify that \(2^7 + 2^6 + 2^4 + 2^3 = 128 + 64 + 16 + 8 = 216\).
To represent negative numbers as well as positive ones, we use an encoding called “two’s complement”.

Two’s complement encoding is based on modular arithmetic: instead of interpreting binary sequences as positive integers, we interpret them as integers modulo $2^b$ (where $b$ is the number of bits).

This representation is nice because addition, subtraction and multiplication work the same way for ordinary integers as they do for integers modulo $2^b$, so arithmetic is straightforward.

However, there are infinitely many integers which are congruent to a given integer modulo $2^b$. We use a simple rule to choose a single integer corresponding to each bit sequence:

- If the leftmost bit is 0, just treat it as an unsigned binary number.
- If the leftmost bit is 1, interpret it as an unsigned binary number and then subtract $2^b$. This gives the smallest (in absolute value) negative number that is congruent to the unsigned binary number.
To represent negative numbers as well as positive ones, we use an encoding called “two’s complement”.

Two’s complement encoding is based on modular arithmetic: instead of interpreting binary sequences as positive integers, we interpret them as integers modulo $2^b$ (where $b$ is the number of bits).
Two’s Complement Representation

- To represent negative numbers as well as positive ones, we use an encoding called “two’s complement”.
- Two’s complement encoding is based on modular arithmetic: instead of interpreting binary sequences as positive integers, we interpret them as integers modulo $2^b$ (where $b$ is the number of bits).
- This representation is nice because addition, subtraction and multiplication work the same way for ordinary integers as they do for integers modulo $2^b$, so arithmetic is straightforward.
Two’s Complement Representation

- To represent negative numbers as well as positive ones, we use an encoding called “two’s complement”.
- Two’s complement encoding is based on modular arithmetic: instead of interpreting binary sequences as positive integers, we interpret them as integers modulo $2^b$ (where $b$ is the number of bits).
- This representation is nice because addition, subtraction and multiplication work the same way for ordinary integers as they do for integers modulo $2^b$, so arithmetic is straightforward.
- However, there are infinitely many integers which are congruent to a given integer modulo $2^b$. 
To represent negative numbers as well as positive ones, we use an encoding called “two’s complement”.

Two’s complement encoding is based on modular arithmetic: instead of interpreting binary sequences as positive integers, we interpret them as integers modulo $2^b$ (where $b$ is the number of bits).

This representation is nice because addition, subtraction and multiplication work the same way for ordinary integers as they do for integers modulo $2^b$, so arithmetic is straightforward.

However, there are infinitely many integers which are congruent to a given integer modulo $2^b$.

We use a simple rule to choose a single integer corresponding to each bit sequence:
Two’s Complement Representation

- To represent negative numbers as well as positive ones, we use an encoding called “two’s complement”.
- Two’s complement encoding is based on modular arithmetic: instead of interpreting binary sequences as positive integers, we interpret them as integers modulo $2^b$ (where $b$ is the number of bits).
- This representation is nice because addition, subtraction and multiplication work the same way for ordinary integers as they do for integers modulo $2^b$, so arithmetic is straightforward.
- However, there are *infinitely many* integers which are congruent to a given integer modulo $2^b$.
- We use a simple rule to choose a single integer corresponding to each bit sequence:
  - If the leftmost bit is 0, just treat it as an unsigned binary number.
To represent negative numbers as well as positive ones, we use an encoding called “two’s complement”.

Two’s complement encoding is based on modular arithmetic: instead of interpreting binary sequences as positive integers, we interpret them as integers modulo $2^b$ (where $b$ is the number of bits).

This representation is nice because addition, subtraction and multiplication work the same way for ordinary integers as they do for integers modulo $2^b$, so arithmetic is straightforward.

However, there are infinitely many integers which are congruent to a given integer modulo $2^b$.

We use a simple rule to choose a single integer corresponding to each bit sequence:

- If the leftmost bit is 0, just treat it as an unsigned binary number.
- If the leftmost bit is 1, interpret it as an unsigned binary number and then subtract $2^b$. This gives the smallest (in absolute value) negative number that is congruent to the unsigned binary number.
Negating a Two’s Complement Number

- Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.

  - Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  - Flip all the bits to the left of the rightmost 1.

- To convert a negative decimal number $-x$ to binary, just convert the positive number $x$ and then use one of the techniques to negate it.

- To convert a two’s complement binary sequence to decimal:
  - If the leftmost bit is 0 just treat it as an unsigned number.
  - If the leftmost bit is 1, first negate the number, treat it as unsigned and convert it, and negate it again.

- Example: How would we convert the 4-bit two’s complement number 1010 to decimal?
  - Using the first technique, we flip the bits to get 0101, then add 1 to get 0110, which is 6. So 1010 is $-6$ in decimal.
  - The second technique gives the same result: flipping the bits to the left of the rightmost 1 gives 0110.
Negating a Two’s Complement Number

- Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.
- Here are two techniques for negating a two’s complement number:

  1. Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  2. Flip all the bits to the left of the rightmost 1.

To convert a negative decimal number $-x$ to binary, just convert the positive number $x$ and then use one of the techniques to negate it.

To convert a two’s complement binary sequence to decimal:

- If the leftmost bit is 0 just treat it as an unsigned number.
- If the leftmost bit is 1, first negate the number, treat it as unsigned and convert it, and negate it again.

Example:

How would we convert the 4-bit two’s complement number 1010 to decimal?

Using the first technique, we flip the bits to get 0101, then add 1 to get 0110, which is 6. So 1010 is $-6$ in decimal.

The second technique gives the same result: flipping the bits to the left of the rightmost 1 gives 0110.
often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.

here are two techniques for negating a two’s complement number:

- flip the bits (change 0s to 1s and 1s to 0s) and add 1.

example: how would we convert the 4-bit two’s complement number 1010 to decimal?

- using the first technique, we flip the bits to get 0101, then add 1 to get 0110, which is 6. so 1010 is \(-6\) in decimal.

the second technique gives the same result: flipping the bits to the left of the rightmost 1 gives 0110.
Negating a Two’s Complement Number

- Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.
- Here are two techniques for negating a two's complement number:
  - Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  - Flip all the bits to the left of the rightmost 1.

To convert a negative decimal number $-x$ to binary, just convert the positive number $x$ and then use one of the techniques to negate it.

To convert a two's complement binary sequence to decimal:
- If the leftmost bit is 0 just treat it as an unsigned number.
- If the leftmost bit is 1, first negate the number, treat it as unsigned and convert it, and negate it again.

Example:
How would we convert the 4-bit two’s complement number 1010 to decimal?

- Using the first technique, we flip the bits to get 0101, then add 1 to get 0110, which is 6. So 1010 is $-6$ in decimal.
- The second technique gives the same result: flipping the bits to the left of the rightmost 1 gives 0110.
Negating a Two’s Complement Number

- Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.
- Here are two techniques for negating a two’s complement number:
  - Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  - Flip all the bits to the left of the rightmost 1.
- To convert a negative decimal number \(-x\) to binary, just convert the positive number \(x\) and then use one of the techniques to negate it.
Negating a Two’s Complement Number

- Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.
- Here are two techniques for negating a two’s complement number:
  - Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  - Flip all the bits to the left of the rightmost 1.
- To convert a negative decimal number $-x$ to binary, just convert the positive number $x$ and then use one of the techniques to negate it.
- To convert a two’s complement binary sequence to decimal:
Negating a Two’s Complement Number

- Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.
- Here are two techniques for negating a two’s complement number:
  - Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  - Flip all the bits to the left of the rightmost 1.
- To convert a negative decimal number \(-x\) to binary, just convert the positive number \(x\) and then use one of the techniques to negate it.
- To convert a two’s complement binary sequence to decimal:
  - If the leftmost bit is 0 just treat it as an unsigned number.
Negating a Two’s Complement Number

- Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.
- Here are two techniques for negating a two’s complement number:
  - Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  - Flip all the bits to the left of the rightmost 1.
- To convert a negative decimal number \(-x\) to binary, just convert the positive number \(x\) and then use one of the techniques to negate it.
- To convert a two’s complement binary sequence to decimal:
  - If the leftmost bit is 0 just treat it as an unsigned number.
  - If the leftmost bit is 1, first negate the number, treat it as unsigned and convert it, and negate it again.

Example:
How would we convert the 4-bit two’s complement number 1010 to decimal?

Using the first technique, we flip the bits to get 0101, then add 1 to get 0110, which is 6. So 1010 is \(-6\) in decimal.

The second technique gives the same result: flipping the bits to the left of the rightmost 1 gives 0110.
Negating a Two’s Complement Number

- Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.
- Here are two techniques for negating a two’s complement number:
  - Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  - Flip all the bits to the left of the rightmost 1.
- To convert a negative decimal number \(-x\) to binary, just convert the positive number \(x\) and then use one of the techniques to negate it.
- To convert a two’s complement binary sequence to decimal:
  - If the leftmost bit is 0 just treat it as an unsigned number.
  - If the leftmost bit is 1, first negate the number, treat it as unsigned and convert it, and negate it again.

**Example:** How would we convert the 4-bit two’s complement number 1010 to decimal?

- Using the first technique, we flip the bits to get 0101, then add 1 to get 0110, which is 6. So 1010 is \(-6\) in decimal.
- The second technique gives the same result: flipping the bits to the left of the rightmost 1 gives 0110.
Negating a Two’s Complement Number

• Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.

• Here are two techniques for negating a two’s complement number:
  • Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  • Flip all the bits to the left of the rightmost 1.

• To convert a negative decimal number $-x$ to binary, just convert the positive number $x$ and then use one of the techniques to negate it.

• To convert a two’s complement binary sequence to decimal:
  • If the leftmost bit is 0 just treat it as an unsigned number.
  • If the leftmost bit is 1, first negate the number, treat it as unsigned and convert it, and negate it again.

• **Example:** How would we convert the 4-bit two’s complement number 1010 to decimal?

• Using the first technique, we flip the bits to get 0101, then add 1 to get 0110, which is 6. So 1010 is $-6$ in decimal.
Negating a Two’s Complement Number

• Often the easiest way to compute the binary representation of a negative number is to take the positive version and negate it.
• Here are two techniques for negating a two’s complement number:
  • Flip the bits (change 0s to 1s and 1s to 0s) and add 1.
  • Flip all the bits to the left of the rightmost 1.
• To convert a negative decimal number \(-x\) to binary, just convert the positive number \(x\) and then use one of the techniques to negate it.
• To convert a two’s complement binary sequence to decimal:
  • If the leftmost bit is 0 just treat it as an unsigned number.
  • If the leftmost bit is 1, first negate the number, treat it as unsigned and convert it, and negate it again.
• Example: How would we convert the 4-bit two’s complement number 1010 to decimal?
• Using the first technique, we flip the bits to get 0101, then add 1 to get 0110, which is 6. So 1010 is \(-6\) in decimal.
• The second technique gives the same result: flipping the bits to the left of the rightmost 1 gives 0110.
Exercises: Use 8-bit two’s complement representation for the following.

- Convert $-12$ to binary.

Solution:

12 in unsigned binary is 00001100. Flipping the bits gives 11110011. Adding 1 gives 11110100. Alternatively, flipping the bits to the left of the rightmost 1 in 00001100, which also gives 11110100.

In $b$-bit binary, the two’s complement representation of $-x$ is equal to the unsigned representation of $2^b - x$. (Why?)

Convert $-123$ to binary using this fact.

Solution:

We have $2^8 - 123 = 133$. We see that $133 = 128 + 4 + 1 = 2^7 + 2^2 + 2^0$ and so the binary representation of $-123$ is 10000101.
Exercises: Use 8-bit two’s complement representation for the following.

- Convert $-12$ to binary.
  
  **Solution:** 12 in unsigned binary is 00001100. Flipping the bits gives 11110011. Adding 1 gives 11110100. Alternatively, flipping the bits to the left of the rightmost 1 in 00001100, which also gives 11110100.
Exercises: Use 8-bit two’s complement representation for the following.

- Convert $-12$ to binary.
  
  **Solution:** $12$ in unsigned binary is $00001100$. Flipping the bits gives $11110011$. Adding $1$ gives $11110100$. Alternatively, flipping the bits to the left of the rightmost $1$ in $00001100$, which also gives $11110100$.

- In $b$-bit binary, the two’s complement representation of $-x$ is equal to the unsigned representation of $2^b - x$. (Why?) Convert $-123$ to binary using this fact.
Exercises: Use 8-bit two’s complement representation for the following.

- Convert $-12$ to binary.

  **Solution:** 12 in unsigned binary is 00001100. Flipping the bits gives 11110011. Adding 1 gives 11110100. Alternatively, flipping the bits to the left of the rightmost 1 in 00001100, which also gives 11110100.

- In $b$-bit binary, the two’s complement representation of $-x$ is equal to the unsigned representation of $2^b - x$. (Why?)

  Convert $-123$ to binary using this fact.

  **Solution:** We have $256 - 123 = 133$. We see that

  $$133 = 128 + 4 + 1 = 2^7 + 2^2 + 2^0$$

  and so the binary representation of $-123$ is 10000101.
Two’s Complement Conversions

Exercises: (continued)

• Negate the two’s complement representation of $-128$. What happens? Explain why this result makes sense.
Exercises: (continued)

• Negate the two’s complement representation of $-128$. What happens? Explain why this result makes sense.

Solution: The two’s complement representation of $-128$ is $10000000$. If we flip the bits we get $01111111$. Adding 1 gives $10000000$ again! The “negation” of $-128$ is just $-128$ again.
Exercises: (continued)

- Negate the two’s complement representation of $-128$. What happens? Explain why this result makes sense.

  **Solution:** The two’s complement representation of $-128$ is $10000000$. If we flip the bits we get $01111111$. Adding $1$ gives $10000000$ again! The “negation” of $-128$ is just $-128$ again.

  - This is because $128$ is congruent to $-128$ modulo $2^8 = 256$. The range of 8-bit two’s complement binary is $-128$ to $127$. Since $128$ cannot be represented, we take the unique number within the range that is congruent to $128$ modulo $2^8$, which is $-128$. 


Exercises: (continued)

- Negate the two’s complement representation of $-128$. What happens? Explain why this result makes sense.

**Solution:** The two’s complement representation of $-128$ is $10000000$. If we flip the bits we get $01111111$. Adding 1 gives $10000000$ again! The “negation” of $-128$ is just $-128$ again.

- This is because $128$ is congruent to $-128$ modulo $2^8 = 256$. The range of 8-bit two’s complement binary is $-128$ to $127$. Since $128$ cannot be represented, we take the unique number within the range that is congruent to $128$ modulo $2^8$, which is $-128$.

- In general, the range of values for unsigned $b$-bit binary representation ranges from $0$ to $2^b - 1$, while the range of values for signed two’s complement $b$-bit binary representation ranges from $-2^{b-1}$ to $2^{b-1} - 1$. 
Exercises: (continued)

- Negate the two’s complement representation of $-128$. What happens? Explain why this result makes sense.
  **Solution:** The two’s complement representation of $-128$ is 10000000. If we flip the bits we get 01111111. Adding 1 gives 10000000 again! The “negation” of $-128$ is just $-128$ again.

  - This is because 128 is congruent to $-128$ modulo $2^8 = 256$. The range of 8-bit two’s complement binary is $-128$ to 127. Since 128 cannot be represented, we take the unique number within the range that is congruent to 128 modulo $2^8$, which is $-128$.
  - In general, the range of values for unsigned $b$-bit binary representation ranges from 0 to $2^b - 1$, while the range of values for signed two’s complement $b$-bit binary representation ranges from $-2^{b-1}$ to $2^{b-1} - 1$.
  - For example, for 8-bit binary, the unsigned range is 0 to 255 while the signed range is $-128$ to 127.
Assembly languages are simple programming languages which let us perform simple arithmetic and logic operations and transfer values between registers and memory.

Example of an assembly language instruction:
```
add $3, $1, $2
```
means "add together the values in registers 1 and 2 and place the result in register 3."
• Assembly languages are simple programming languages which let us perform simple arithmetic and logic operations and transfer values between registers and memory.

• While high-level programming languages usually can run on many computers, assembly languages are typically designed for a specific processor architecture.
Assembly Language

- Assembly languages are simple programming languages which let us perform simple arithmetic and logic operations and transfer values between registers and memory.
- While high-level programming languages usually can run on many computers, assembly languages are typically designed for a specific processor architecture.
- We will use a simplified version of the “MIPS” (Microprocessor without Interlocked Pipelined Stages) assembly language.
• Assembly languages are simple programming languages which let us perform simple arithmetic and logic operations and transfer values between registers and memory.

• While high-level programming languages usually can run on many computers, assembly languages are typically designed for a specific processor architecture.

• We will use a simplified version of the “MIPS” (Microprocessor without Interlocked Pipelined Stages) assembly language.

• Example of an assembly language instruction: `add $3, $1, $2` means “add together the values in registers 1 and 2 and place the result in register 3.”
Assembly languages are simple programming languages which let us perform simple arithmetic and logic operations and transfer values between registers and memory.

While high-level programming languages usually can run on many computers, assembly languages are typically designed for a specific processor architecture.

We will use a simplified version of the “MIPS” (Microprocessor without Interlocked Pipelined Stages) assembly language.

Example of an assembly language instruction: add $3, $1, $2 means “add together the values in registers 1 and 2 and place the result in register 3.”

Note that the destination comes first, just like assignment statements in most programming languages, e.g., r3 = r1 + r2.
More About Registers

• Registers in our simplified MIPS architecture each hold 32 bits of information, and can be thought of as being similar to variables: they store data and allow the data to be accessed and manipulated.

• Data can also be stored in memory (RAM). Accessing registers is much faster than accessing memory, but memory has much more space for data storage.

• The MIPS machine has 32 numbered registers denoted $0$ to $31$, and some special registers used internally by the processor (e.g., PC, IR, HI, LO). Only the numbered registers can be used in instructions.

• Some of the numbered registers are also special:
  • $0$ always contains the value 0 and the contents of $0$ cannot be modified in any way.
  • $31$ is reserved for the return address of the program (or later in the course, procedures).
  • Registers $3$, $29$, and $30$ are special by convention in this course. Try to avoid using these special registers for things like computations or temporary values. Their intended purpose will be explained later.
More About Registers

• Registers in our simplified MIPS architecture each hold 32 bits of information, and can be thought of as being similar to variables: they store data and allow the data to be accessed and manipulated.

• Data can also be stored in memory (RAM). Accessing registers is much faster than accessing memory, but memory has much more space for data storage.

The MIPS machine has 32 numbered registers denoted $0$ to $31$, and some special registers used internally by the processor (e.g., PC, IR, HI, LO). Only the numbered registers can be used in instructions.

• Some of the numbered registers are also special:
  - $0$ always contains the value 0 and the contents of $0$ cannot be modified in any way.
  - $31$ is reserved for the return address of the program (or later in the course, procedures).
  - Registers $3$, $29$, and $30$ are special by convention in this course.

Try to avoid using these special registers for things like computations or temporary values. Their intended purpose will be explained later.
Registors in our simplified MIPS architecture each hold 32 bits of information, and can be thought of as being similar to variables: they store data and allow the data to be accessed and manipulated.

Data can also be stored in memory (RAM). Accessing registers is much faster than accessing memory, but memory has much more space for data storage.

The MIPS machine has 32 numbered registers denoted $0$ to $31$, and some special registers used internally by the processor (e.g., PC, IR, HI, LO). Only the numbered registers can be used in instructions.
• Registers in our simplified MIPS architecture each hold 32 bits of information, and can be thought of as being similar to variables: they store data and allow the data to be accessed and manipulated.
• Data can also be stored in memory (RAM). Accessing registers is much faster than accessing memory, but memory has much more space for data storage.
• The MIPS machine has 32 numbered registers denoted $0$ to $31$, and some special registers used internally by the processor (e.g., PC, IR, HI, LO). Only the numbered registers can be used in instructions.
• Some of the numbered registers are also special:
More About Registers

- Registers in our simplified MIPS architecture each hold 32 bits of information, and can be thought of as being similar to variables: they store data and allow the data to be accessed and manipulated.
- Data can also be stored in memory (RAM). Accessing registers is much faster than accessing memory, but memory has much more space for data storage.
- The MIPS machine has 32 numbered registers denoted $0$ to $31$, and some special registers used internally by the processor (e.g., PC, IR, HI, LO). Only the numbered registers can be used in instructions.
- Some of the numbered registers are also special:
  - $0$ always contains the value 0 and the contents of $0$ cannot be modified in any way.
More About Registers

- Registers in our simplified MIPS architecture each hold 32 bits of information, and can be thought of as being similar to variables: they store data and allow the data to be accessed and manipulated.

- Data can also be stored in memory (RAM). Accessing registers is much faster than accessing memory, but memory has much more space for data storage.

- The MIPS machine has 32 numbered registers denoted $0$ to $31$, and some special registers used internally by the processor (e.g., PC, IR, HI, LO). Only the numbered registers can be used in instructions.

- Some of the numbered registers are also special:
  - $0$ always contains the value 0 and the contents of $0$ cannot be modified in any way.
  - $31$ is reserved for the return address of the program (or later in the course, procedures).
More About Registers

- Registers in our simplified MIPS architecture each hold 32 bits of information, and can be thought of as being similar to variables: they store data and allow the data to be accessed and manipulated.
- Data can also be stored in memory (RAM). Accessing registers is much faster than accessing memory, but memory has much more space for data storage.
- The MIPS machine has 32 numbered registers denoted $0$ to $31$, and some special registers used internally by the processor (e.g., PC, IR, HI, LO). Only the numbered registers can be used in instructions.
- Some of the numbered registers are also special:
  - $0$ always contains the value 0 and the contents of $0$ cannot be modified in any way.
  - $31$ is reserved for the return address of the program (or later in the course, procedures).
  - Registers $3$, $29$, and $30$ are special by convention in this course.
More About Registers

- Registers in our simplified MIPS architecture each hold 32 bits of information, and can be thought of as being similar to variables: they store data and allow the data to be accessed and manipulated.
- Data can also be stored in memory (RAM). Accessing registers is much faster than accessing memory, but memory has much more space for data storage.
- The MIPS machine has 32 numbered registers denoted $0$ to $31$, and some special registers used internally by the processor (e.g., PC, IR, HI, LO). Only the numbered registers can be used in instructions.
- Some of the numbered registers are also special:
  - $0$ always contains the value 0 and the contents of $0$ cannot be modified in any way.
  - $31$ is reserved for the return address of the program (or later in the course, procedures).
  - Registers $3$, $29$, and $30$ are special by convention in this course.
- Try to avoid using these special registers for things like computations or temporary values. Their intended purpose will be explained later.
Most MIPS instructions in this course deal with values stored in registers instead of constant values.

To load constant values into registers, we use the Load Immediate and Skip (lis) instruction together with the .word directive. For example, this snippet of assembly stores the value 7 into $5:

```
lis $5
.word 7
```

The "Load Immediate and Skip" name comes from the fact that the lis instruction skips over the constant value 7 after loading it into the register. If it wasn't for the skip, the MIPS machine would try to execute the binary representation of 7 as if it was an instruction.

Constant values are often called immediate values in the context of assembly language.
Using Constant Values

- Most MIPS instructions in this course deal with values stored in registers instead of constant values.
- While real MIPS machines have instructions that allow you to do things like add a constant to a value in a register, the simplified MIPS machine in this course lacks these instructions.

For example, this snippet of assembly stores the value 7 into $5.

```assembly
lis $5
.word 7
```

The "Load Immediate and Skip" name comes from the fact that the `lis` instruction skips over the constant value 7 after loading it into the register. If it wasn’t for the skip, the MIPS machine would try to execute the binary representation of 7 as if it was an instruction.

Constant values are often called immediate values in the context of assembly language.
Most MIPS instructions in this course deal with values stored in registers instead of constant values.

While real MIPS machines have instructions that allow you to do things like add a constant to a value in a register, the simplified MIPS machine in this course lacks these instructions.

To load constant values into registers, we use the Load Immediate and Skip (lis) instruction together with the .word directive.
Most MIPS instructions in this course deal with values stored in registers instead of constant values.

While real MIPS machines have instructions that allow you to do things like add a constant to a value in a register, the simplified MIPS machine in this course lacks these instructions.

To load constant values into registers, we use the Load Immediate and Skip (lis) instruction together with the .word directive.

For example, this snippet of assembly stores the value 7 into $5.

```assembly
lis $5
.word 7
```
Most MIPS instructions in this course deal with values stored in registers instead of constant values.

While real MIPS machines have instructions that allow you to do things like add a constant to a value in a register, the simplified MIPS machine in this course lacks these instructions.

To load constant values into registers, we use the Load Immediate and Skip (lis) instruction together with the .word directive.

For example, this snippet of assembly stores the value 7 into $5.

```
lis $5
.word 7
```

The “Load Immediate and Skip” name comes from the fact that the lis instruction skips over the constant value 7 after loading it into the register.
Using Constant Values

- Most MIPS instructions in this course deal with values stored in registers instead of constant values.
- While real MIPS machines have instructions that allow you to do things like add a constant to a value in a register, the simplified MIPS machine in this course lacks these instructions.
- To load constant values into registers, we use the Load Immediate and Skip (lis) instruction together with the .word directive.
- For example, this snippet of assembly stores the value 7 into $5.
  
  ```assembly
  lis $5
  .word 7
  ```
- The “Load Immediate and Skip” name comes from the fact that the lis instruction skips over the constant value 7 after loading it into the register.
- If it wasn’t for the skip, the MIPS machine would try to execute the binary representation of 7 as if it was an instruction.
Using Constant Values

• Most MIPS instructions in this course deal with values stored in registers instead of constant values.
• While real MIPS machines have instructions that allow you to do things like add a constant to a value in a register, the simplified MIPS machine in this course lacks these instructions.
• To load constant values into registers, we use the Load Immediate and Skip (lis) instruction together with the .word directive.
• For example, this snippet of assembly stores the value 7 into $5.
  
  ```assembly
  lis $5
  .word 7
  ```
• The “Load Immediate and Skip” name comes from the fact that the lis instruction skips over the constant value 7 after loading it into the register.
• If it wasn’t for the skip, the MIPS machine would try to execute the binary representation of 7 as if it was an instruction.
• Constant values are often called immediate values in the context of assembly language.
Assembling Instructions

- The process of converting assembly language into binary code the computer can process (machine code) is called **assembling**.
• The process of converting assembly language into binary code the computer can process (machine code) is called assembling.

• Later in the course we will automate this process by writing an assembler, but for now we’ll do it by hand.
Assembling Instructions

- The process of converting assembly language into binary code the computer can process (machine code) is called **assembling**.
- Later in the course we will automate this process by writing an **assembler**, but for now we’ll do it by hand.
- The MIPS assembly language reference sheet gives a template for how to assemble each 32-bit instruction.
Assembling Instructions

- The process of converting assembly language into binary code the computer can process (machine code) is called **assembling**.
- Later in the course we will automate this process by writing an **assembler**, but for now we’ll do it by hand.
- The MIPS assembly language reference sheet gives a template for how to assemble each 32-bit instruction.
- There are two instruction formats: register and immediate. Register format is for instructions that only take registers as arguments, while immediate format is for instructions which include immediate values.
Assembling Instructions

- The process of converting assembly language into binary code the computer can process (machine code) is called assembling.
- Later in the course we will automate this process by writing an assembler, but for now we’ll do it by hand.
- The MIPS assembly language reference sheet gives a template for how to assemble each 32-bit instruction.
- There are two instruction formats: register and immediate. Register format is for instructions that only take registers as arguments, while immediate format is for instructions which include immediate values.
- The format of the instruction is listed in the second-last column on the MIPS reference sheet: R for register and I for immediate.
Suppose we want to assemble the register-format instruction `mult $5, $4`.

The reference sheet gives the following form for the instruction:

```
0000 00ss ssst tttt 0000 0000 0001 1000
```

1. First, we need to convert 5 into 5-bit binary: 00101. This is `s`.
2. Next, we need to convert 4 into 5-bit binary: 00100. This is `t`.
3. Next, we simply need to replace the `s` and `t` values for `mult` with these binary sequences. Since `mult` has no `d` register, 00000 is simply substituted in that position (this is already done for you on the reference sheet). We end up with:

```
0000 0000 1010 0100 0000 0000 0001 1000
```

4. For the assignment, we need to rewrite this as hexadecimal since cs241.wordasm only accepts hexadecimal values and not binary values.
5. Converting each nibble (4-bit chunk) we end up with `0x00a40018`.

Assembling immediate-format instructions such as `beq $0, $1, -2` is very similar, except they include a 16-bit two's complement immediate value that you must convert from decimal.
Assembling Instructions

- Suppose we want to assemble the register-format instruction `mult $5, $4`.
  - The reference sheet gives the following form for the instruction:
    0000 00ss ssst tttt 0000 0000 0001 1000
  - First, we need to convert 5 into 5-bit binary: 00101. This is s.
  - Next, we need to convert 4 into 5-bit binary: 00100. This is t.
  - Next, we simply need to replace the s and t values for mult with these binary sequences. Since mult has no d register, 00000 is simply substituted in that position (this is already done for you on the reference sheet). We end up with:
    0000 0000 1010 0100 0000 0000 0001 1000
  - For the assignment, we need to rewrite this as hexadecimal since cs241.wordasm only accepts hexadecimal values and not binary values.
  - Converting each nibble (4-bit chunk) we end up with 0x00a40018.

- Assembling immediate-format instructions such as `beq $0, $1, -2` is very similar, except they include a 16-bit two's complement immediate value that you must convert from decimal.
Suppose we want to assemble the register-format instruction `mult $5, $4`.

- The reference sheet gives the following form for the instruction:
  
  0000 00ss ssst tttt 0000 0000 0001 1000

- First, we need to convert 5 into 5-bit binary: 00101. This is s.

- Next, we need to convert 4 into 5-bit binary: 00100. This is t.

- Next, we simply need to replace the s and t values for `mult` with these binary sequences. Since `mult` has no d register, 00000 is simply substituted in that position (this is already done for you on the reference sheet). We end up with:

  0000 0000 1010 0100 0000 0000 0001 1000

- For the assignment, we need to rewrite this as hexadecimal since cs241.wordasm only accepts hexadecimal values and not binary values.

- Converting each nibble (4-bit chunk) we end up with 0x00a40018.

Assembling immediate-format instructions such as `beq $0, $1, -2` is very similar, except they include a 16-bit two’s complement immediate value that you must convert from decimal.
• Suppose we want to assemble the register-format instruction
  `mult $5, $4`.
  • The reference sheet gives the following form for the instruction:
    `0000 00ss ssst tttt 0000 0000 0001 1000`
  • First, we need to convert 5 into 5-bit binary: 00101. This is s.
  • Next, we need to convert 4 into 5-bit binary: 00100. This is t.
  • For the assignment, we need to rewrite this as hexadecimal since
    `cs241.wordasm` only accepts hexadecimal values and not binary values.
    Converting each nibble (4-bit chunk) we end up with
    `0x00a40018`. 
• Suppose we want to assemble the register-format instruction 
mult $5, $4.

  • The reference sheet gives the following form for the instruction:
    0000 00ss ssst tttt 0000 0000 0001 1000
  • First, we need to convert 5 into 5-bit binary: 00101. This is s.
  • Next, we need to convert 4 into 5-bit binary: 00100. This is t.
  • Next, we simply need to replace the s and t values for mult with these 
binary sequences. Since mult has no d register, 00000 is simply 
substituted in that position (this is already done for you on the 
reference sheet). We end up with:
    0000 0000 1010 0100 0000 0000 0001 1000
Suppose we want to assemble the register-format instruction `mult $5, $4`.

- The reference sheet gives the following form for the instruction:
  
  0000 00ss ssst tttt 0000 0000 0001 1000

- First, we need to convert 5 into 5-bit binary: 00101. This is s.
- Next, we need to convert 4 into 5-bit binary: 00100. This is t.
- Next, we simply need to replace the s and t values for `mult` with these binary sequences. Since `mult` has no d register, 00000 is simply substituted in that position (this is already done for you on the reference sheet). We end up with:
  
  0000 0000 1010 0100 0000 0000 0001 1000

- For the assignment, we need to rewrite this as hexadecimal since `cs241.wordasm` only accepts hexadecimal values and not binary values.
Suppose we want to assemble the register-format instruction
\texttt{mult \$5, \$4}.

The reference sheet gives the following form for the instruction:
\begin{quote}
0000 00ss ssst tttt 0000 0000 0001 1000
\end{quote}

First, we need to convert 5 into 5-bit binary: 00101. This is s.
Next, we need to convert 4 into 5-bit binary: 00100. This is t.
Next, we simply need to replace the s and t values for mult with these binary sequences. Since mult has no d register, 00000 is simply substituted in that position (this is already done for you on the reference sheet). We end up with:
\begin{quote}
0000 0000 1010 0100 0000 0000 0001 1000
\end{quote}

For the assignment, we need to rewrite this as hexadecimal since \texttt{cs241.wordasm} only accepts hexadecimal values and not binary values.
Converting each nibble (4-bit chunk) we end up with 0x00a40018.
Suppose we want to assemble the register-format instruction `mult $5, $4`.

- The reference sheet gives the following form for the instruction:
  
  0000 00ss ssst tttt 0000 0000 0001 1000

- First, we need to convert 5 into 5-bit binary: 00101. This is s.
- Next, we need to convert 4 into 5-bit binary: 00100. This is t.
- Next, we simply need to replace the s and t values for mult with these binary sequences. Since mult has no d register, 00000 is simply substituted in that position (this is already done for you on the reference sheet). We end up with:
  
  0000 0000 1010 0100 0000 0000 0001 1000

- For the assignment, we need to rewrite this as hexadecimal since `cs241.wordasm` only accepts hexadecimal values and not binary values.
  
  Converting each nibble (4-bit chunk) we end up with `0x00a40018`.

Assembling immediate-format instructions such as `beq $0, $1, -2` is very similar, except they include a 16-bit two’s complement immediate value that you must convert from decimal.
Exercise: Assemble the following program by first writing out its binary representation, then converting it to hexadecimal representation.

```
lis $5
.word 7
add $1, $1, $5
beq $0, $1, -2
jr $31
```

Solution: Here is the final result:

```
.word 0x00002814  # Binary representation of 7
.word 0x00000007
.word 0x00250820
.word 0x1001fffe
.word 0x03e00008
```

This program adds 7 to the value in $1, and then checks if $1 is zero. If so, it branches back and performs the addition again, repeating until $1 is non-zero. The result is that if $1 is -7, then it is increased by 14, and otherwise $1 is increased by 7.
Exercise: Assemble the following program by first writing out its binary representation, then converting it to hexadecimal representation.

```assembly
lis $5
.word 7
add $1, $1, $5
beq $0, $1, -2
jr $31
```

Solution: Here is the final result:

```assembly
.word 0x00002814
.word 0x00000007
.word 0x00250820
.word 0x1001ffe
.word 0x03e00008
```

This program adds 7 to the value in $1, and then checks if $1 is zero. If so, it branches back and performs the addition again, repeating until $1 is non-zero. The result is that if $1 is -7, then it is increased by 14, and otherwise $1 is increased by 7.
Assembling Instructions

**Exercise:** Assemble the following program by first writing out its binary representation, then converting it to hexadecimal representation.

```
lis $5
.word 7
add $1, $1, $5
beq $0, $1, -2
jr $31
```

**Solution:** Here is the final result:

```
.word 0x00002814
.word 0x00000007
.word 0x00250820
.word 0x1001fffe
.word 0x03e00008
```

This program adds 7 to the value in $1, and then checks if $1 is zero. If so, it branches back and performs the addition again, repeating until $1 is non-zero. The result is that if $1 is -7, then it is increased by 14, and otherwise $1 is increased by 7.