Writing an Assembler: Part 1
MIPS Assembly Language

• Instead of writing machine language...

00000000010000010001100000100010
00000000000000000010000000010100
00000000000000000000000000000100
00000000011001000001100000100000
10001100100100000000000000000000
00000011111100000000000000001000

• It's easier to write (and read) assembly language!

sub $3, $2, $1  Subtract ($3 = $2 - $1)
lis $4  Load Immediate & Skip ($4 = MEM[PC]; PC += 4)
.word 4  Decimal Integer 4 (encoded as unsigned 32-bit word)
add $3, $3, $4  Add ($3 = $3 + $4)
lw $5, 0($0)  Load Word ($5 = MEM[$0 + 0])
jr $31  Jump Register (PC = $31)
MIPS Assembly Language

• We've learned six machine language instructions so far.
• Eleven remain to be discussed (in our dialect of MIPS).
• The syntax is similar to the ones we've already seen. For example:
  • mult $s, $t (multiplication)
  • div $s, $t (division)
  • slt $d, $s, $t (less-than comparison)
  • beq $s, $t, i (conditional branching)

• Before we get back into MIPS programming, let's learn how to actually translate this convenient syntax into machine language.
• When discussing machine language, we learned there are two basic formats for instruction encoding:
  
  • **Register Format**: 000000 sssss ttttt dddddd 00000 ffffff
    
    - The s, t and d bit sequences encode **register numbers** (0 to 31).
    - The first 6 bits, the **opcode bits**, are always zero in Register Format instructions.
    - When the opcode bits are zero, the MIPS machine looks at the **function bits**, the last 6 bits, to determine which instruction to execute.

  • **Immediate Format**: oooooo sssss ttttt iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
    
    - The s and t bit sequences encode **register numbers** (0 to 31).
    - The i bits encode a 16-bit **immediate (constant) value**.
    - The first 6 bits, the **opcode bits**, are non-zero and indicate which instruction to perform.
MIPS Syntax in Detail

- **Register Format**: 000000 sssss ttttt ddddd 00000 fffffff
  - 13 of our MIPS instructions use this format. The assembly syntax is:

```plaintext
add   $d, $s, $t
sub   $d, $s, $t
slt   $d, $s, $t
sltu  $d, $s, $t
mult  $s, $t
multu $s, $t
div   $s, $t
divu  $s, $t
lis   $d
mflo  $d
mfhi  $d
jr    $s
jalr  $s
```

- Three Operands
- Two Operands
- One Operand ($d)
- One Operand ($s)
MIPS Syntax in Detail

- **Immediate Format**: `000000 sssss ttttt iiiiiiiiiiiiiiiiiiiiiiiiiii

  - Four of our instructions use this format:
    - `beq $s, $t, i`
    - `bne $s, $t, i`
    - `lw $t, i($s)`
    - `sw $t, i($s)`

  - What if we want to specify something that's not an instruction, like a constant to load with Load Immediate & Skip?

    - `.word i` is used to encode the decimal or hexadecimal value `i` as a 32-bit word.
      - If `i` is a negative decimal number, two's complement encoding is used.
      - If `i` is a non-negative decimal number, unsigned encoding is used.
      - If `i` is hexadecimal, each hex digit is translated directly into a 4-bit chunk (nibble).
Assemblers

• An **assembler** is a program that translates assembly language into machine language.

• To translate a line of code like "add $1, $2, $3", the process might be:
  • Break down the line into meaningful chunks (with a scanner!)
  • Check that the syntax of the line makes sense.
  • Look up the function bits for "add" in a hardcoded lookup table.
  • Extract the numbers 1, 2 and 3 from the register tokens.
  • Use this data to "fill in" the register format encoding and output it.

• **Register Format**: 000000 sssss ttttt dddddd 00000 ffffffff
Structure of a MIPS Assembly Program

• A MIPS program is divided into lines.
• A line can have at most one instruction (or .word directive).
• Lines can also be blank, or consist only of a comment.
  • A semicolon ; starts a comment, which runs to the end of the line.
• There's also a feature we haven't talked about yet called labels.
  • Labels let you assign a name to a particular location in the program, so that you can refer to it later.
  • A line can start with any number of labels, but labels must appear before the instruction on a line.
  • You can also have label-only lines...
A Simplifying Assumption

• We'll ignore blank lines, comments, and labels and assume for now: **Every line contains exactly one instruction or .word directive.**

• This means that every line of MIPS assembly language translates into exactly one 32-bit binary word.

• This assumption is convenient for the assembler writer but not for the assembly programmer!

• We'll remove this assumption in the second version of our assembler.
  • Blank lines and comments are easy – just have the scanner ignore them.
  • Labels are tricky and will require significant refactoring.
• The term *compiler* usually refers to a tool that translates a "high-level" language to a "low-level" language, but more generally, it can mean any tool that translates between two programming languages.

• An assembler can be viewed as a "low-level to lower-level" compiler.
The Phases of Assembly (and Compilation!)

- The compilation process can be broadly divided into two phases:
  - **Analysis**: Understand the meaning of the source program.
  - **Synthesis**: Output an equivalent program in the target language.
The Phases of Assembly (and Compilation!)

- **Analysis** is usually divided further into three steps:
  - **Scanning**: Analysis of lexemes (grouping characters into meaningful strings).
  - **Parsing**: Analysis of syntax (grouping strings into larger structures).
  - **Semantic Analysis**: Analysis of semantics (meaning of structures in context).
• Parsing in an assembler involves looking at the tokens and figuring out which groups of tokens constitute an instruction or directive, while rejecting groups of tokens that are nonsensical (syntax errors).
The Phases of Assembly (and Compilation!)

- In high-level languages, which usually have complex nesting as opposed to the simple line-based style of assembly, the parsing phase groups tokens together into a tree representing the structure and components of the program.
• There is little to be done for semantic analysis in an assembler. Each instruction can be translated without considering its meaning in the wider context of the program... at least until labels are introduced. (More on this later)
In high-level languages, semantic analysis often deals with issues involving **identifier names** (variable, functions, etc.) and **types**. The C statement "int x = y;" is syntactically valid but the *meaning in context* might not make sense (maybe y is undefined, or y is a pointer).
The Phases of Assembly (and Compilation!)

- Once the structure of the source program is understood, the **Synthesis** phase produces an equivalent program in the target language. For an assembler, this means producing a machine language program equivalent to the assembly language program.
We've learned how to scan using maximal munch or simplified maximal munch. We just need a scanning DFA.

Let's review the syntax again to figure out what tokens we need:

- Instructions with register operands:
  - `jr $31`
  - `add $1, $2, $3`
- The `.word` directive (accepts signed decimal and unsigned hex):
  - `.word 241`
  - `.word -37`
  - `.word 0xCAFEBEEF`
- Instructions with immediate operands (signed decimal or unsigned hex):
  - `beq $1, $2, -1`
  - `lw $31, -4($30)`
  - `sw $1, 0x14($0)`
We've learned how to scan using maximal munch or simplified maximal munch. We just need a scanning DFA.

Let's review the syntax again to figure out what tokens we need:

- Instructions with register operands:
  - `jr $31`  `add $1, $2, $3`
- The `.word` directive (accepts signed decimal and unsigned hex):
  - `.word 241`  `.word -37`  `.word 0xCAFEBEEF`
- Instructions with immediate operands (signed decimal or unsigned hex):
  - `beq $1, $2, -1`  `lw $31, -4($30)`  `sw $1, 0x14($0)`

Instruction identifiers
We've learned how to scan using maximal munch or simplified maximal munch. We just need a scanning DFA.

Let's review the syntax again to figure out what tokens we need:

- Instructions with register operands:
  
  jr  $31
  add $1, $2, $3

- The .word directive (accepts signed decimal and unsigned hex):
  
  .word 241
  .word -37
  .word 0xCAFEBEEF

- Instructions with immediate operands (signed decimal or unsigned hex):
  
  beq $1, $2, -1
  lw $31, -4($30)
  sw $1, 0x14($0)

- Directive identifiers (like instructions, but start with a dot)
Scanning

• We've learned how to scan using maximal munch or simplified maximal munch. We just need a scanning DFA.

• Let's review the syntax again to figure out what tokens we need:
  • Instructions with register operands:
    jr  $31  add $1, $2, $3
  • The .word directive (accepts signed decimal and unsigned hex):
    .word 241  .word -37  .word 0xCAFEBEEF
  • Instructions with immediate operands (signed decimal or unsigned hex):
    beq $1, $2, -1  lw $31, -4($30)  sw $1, 0x14($0)

• Registers
**Scanning**

- We've learned how to scan using maximal munch or simplified maximal munch. We just need a scanning DFA.
- Let's review the syntax again to figure out what tokens we need:
  - Instructions with register operands:
    ```
    jr $31    add $1, $2, $3
    ```
  - The `.word` directive (accepts signed decimal and unsigned hex):
    ```
    .word 241    .word -37    .word 0xCAFEBEEF
    ```
  - Instructions with immediate operands (signed decimal or unsigned hex):
    ```
    beq $1, $2, -1    lw $31, -4($30)    sw $1, 0x14($0)
    ```
- Decimal immediates
We've learned how to scan using maximal munch or simplified maximal munch. We just need a scanning DFA.

Let's review the syntax again to figure out what tokens we need:

- Instructions with register operands:
  - `jr $31`  `add $1, $2, $3`

- The `.word` directive (accepts signed decimal and unsigned hex):
  - `.word 241`
  - `.word -37`
  - `.word 0xCAFEBEEF`

- Instructions with immediate operands (signed decimal or unsigned hex):
  - `beq $1, $2, -1`
  - `lw $31, -4($30)`
  - `sw $1, 0x14($0)`

- Hexadecimal immediates
Scanning

• We've learned how to scan using maximal munch or simplified maximal munch. We just need a scanning DFA.

• Let's review the syntax again to figure out what tokens we need:
  
  • Instructions with register operands:
    
    jr $31  
    add $1, $2, $3

  • The .word directive (accepts signed decimal and unsigned hex):

    .word 241  
    .word -37  
    .word 0xCAFEBEEF

  • Instructions with immediate operands (signed decimal or unsigned hex):

    beq $1, $2, -1  
    lw $31, -4($30)  
    sw $1, 0x14($0)

• Punctuation (commas, parentheses)
Scanning

• We've learned how to scan using maximal munch or simplified maximal munch. We just need a scanning DFA.

• Let's review the syntax again to figure out what tokens we need:
  • Instructions with register operands:
    jr $31          add $1, $2, $3
  • The .word directive (accepts signed decimal and unsigned hex):
    .word 241       .word -37         .word 0xCAFEBEEF
  • Instructions with immediate operands (signed decimal or unsigned hex):
    beq $1, $2, -1   lw $31, -4($30)   sw $1, 0x14($0)

• We also need to deal with whitespace and newlines in the program...
Constructing the Scanning DFA

• We'll build DFAs for each of the following token kinds:
  • Instruction identifiers
  • Directive identifiers
  • Registers
  • Decimal immediates
  • Hexadecimal immediates
  • Punctuation
• We'll also discuss how to handle whitespace and newlines.
• Then we'll figure out how to combine everything into one large DFA!
Instruction Identifiers

• We could have separate token kinds for each of the 17 instructions.

• But designing our DFA to recognize these specific 17 strings would be annoying. We will take a simpler approach:

  • Any sequence of lowercase letters is an **Identifier** token.

  • We'll expand the definition of an **Identifier** when we introduce **labels** later on. For now, this is good enough.
Directive Identifiers

- Right now, the only directive we have is `.word`, so we could just hardcode our DFA to recognize .word.
- We'll be more general and define a **Dot-Identifier** to be an **Identifier** with a dot in front.
- We'll learn about two more directives later, but you won't be asked to implement them on assignments.
Registers

• Recognizing a $ sign followed by a sequence of digits is straightforward, but how strict do we want to be?
  • Should we accept things like $007 with useless leading zeroes?
  • Should we accept invalid register numbers like $32 or $241?

• Leading zeroes are probably okay but we want to treat invalid register numbers as an error.

• However, doing numeric range checking in a DFA can be awkward.

• Let's keep our DFA simple and do range checking outside the DFA.

• After producing a token, extract the number and check its range.
• This simple DFA treats things like $2147483648 as a valid Register token, but that's okay.

• Once simplified maximal munch finds a token, we can examine the token's lexeme and do a range check in our high-level code.
Decimal Immediates

• We are not strict here, and we allow things like useless leading zeroes and the strange sequence -0 in Decimal tokens.
Hexadecimal Immediates

- Recognizing **Hexadecimal** tokens is straightforward on its own, but it will be a little tricky to combine this with the previous DFA.
- "0" could be a decimal number OR the start of "0x..."
Punctuation

• We treat all three punctuation characters as separate token kinds because they are used in different syntactic contexts.
Handling Non-Newline Whitespace

• We want to filter out non-newline whitespace because it has no meaning in a MIPS program.
  • On the other hand, MIPS is a line-based language, so newlines are important.

• Two approaches:
  • Create a token kind for **Whitespace**, but modify our scanner so that when it detects a **Whitespace** token, it simply discards it instead of adding it to the tokenization.
  • Augment (simplified) maximal munch with logic that skips past whitespace after producing a token before reading the next token. (No change to DFA)

• We'll use the second approach in these slides so our DFA is less complicated, but we'll use the first approach in the scanning project.
Handling Newlines

- Newlines can't just be discarded because they have actual meaning in a MIPS program (they separate instructions).
- We can approach this in two ways.
  - Modify the DFA to recognize newlines and output them as their own token.
  - Leave the DFA as is. Then, when writing the assembler, have it run the scanner on each line individually instead of on the entire program.
- Again, for these slides, we'll use the approach that doesn't involve making our DFA more complicated.
- For the project we will ask you to output Newline tokens.
The Complete Scanning DFA
Decimal and Hexadecimal

• The only tricky part of combining the DFAs is how we handled the **Decimal** and **Hexadecimal** tokens.
Decimal and Hexadecimal

• We created a separate state for the decimal number **Zero**. Why?
Decimal and Hexadecimal

• After reading "0", if we see "xFF", we should end up in the Hexadecimal state.
Decimal and Hexadecimal

• But after reading "1", if we see "xFF", we get "1xFF" which is not a valid token at all.
Decimal and Hexadecimal

• Therefore "0" and "1" cannot go to the same state in the full DFA. We need a separate state for "0".
Decimal and Hexadecimal

• Note that our scanner should still output a **Decimal** token for zero.
Decimal and Hexadecimal

• There is not always a perfect correspondence between token kinds and DFA states.
The result after scanning is that each line has been converted to a sequence of tokens.

Parsing involves syntax checking (making sure the right tokens appear in the right order) and extracting information from each line.

For example, suppose you see this token line:

```
[Identifier "add"] [Register "$1"] [Comma ","] [Register "$2"] [Comma ","] [Register "$3"]
```

- Look up the identifier "add" to determine the expected syntax (in this case, three registers separated by commas).
- Read the following tokens to verify the syntax rules are followed.
- Extract the information needed for Synthesis (instruction name and registers).
Parsing

- The result after scanning is that each line has been converted to a sequence of tokens.
- Parsing involves syntax checking (making sure the right tokens appear in the right order) and extracting information from each line.
- Some instructions are a little trickier than "add", e.g., "lw".
  - Both of these sequences of token kinds are valid for lw:
    Identifier Register Comma Decimal L-Paren Register R-Paren
    Identifier Register Comma Hexadecimal L-Paren Register R-Paren
  - A Decimal/Hexadecimal token can be up to 32 bits in general, but must be in the 16 bit two's complement range for lw (extra range check needed).
  - A detailed syntax reference will be posted with the Assembler project.
Synthesis

• In the first version of our assembler, there's nothing to do for Semantic Analysis, so let's move on to Synthesis.

• Let's assume we have extracted the key information from each tokenized line, and we want to generate machine code.
  
  \[ \text{add} \, \$3, \, \$2, \, \$1 \rightarrow \{ \text{instr: "add"}, \, s: \, 2, \, t: \, 1, \, d: \, 3 \} \]

• Encoding for add $d, $s, $t:
  
  000000 sssss ttttt ddddd 00000 100000

• Use a lookup table to figure out the last 6 bits (function bits).

• How do we handle the s, t and d values?
Synthesis

• One approach is to convert s, t and d to 5-character strings of 0s and 1s and create a string representation of the encoding.
  
  ```cpp
  std::string instruction = "00000000010000010001100000100000";
  ```

  • Then, use your solution to Question 2 to output the machine code.
  • This *works*, but it's inefficient and wasteful because it involves a lot of unnecessary string operations.
  • Instead of creating a 32-byte string, it's much more efficient to store the encoding in a single *32-bit value*.
  • We'll construct the encoding as an integer!
Encoding Instructions as Integers

• add $3, $2, $1 \rightarrow \{ \text{instr: "add", s: 2, t: 1, d: 3} \}

• Encoding for add $3, $2, $1:
  \begin{align*}
  &000000 \ 00010 \ 00001 \ 00011 \ 00000 \ 100000 \\
\end{align*}

• As a binary number, this is: $2^{22} + 2^{16} + 2^{12} + 2^{11} + 2^{5}$

• Let's do some math...
  \begin{align*}
  2^{22} + 2^{16} + 2^{12} + 2^{11} + 2^{5} &= (2 \cdot 2^{21}) + (1 \cdot 2^{16}) + (3 \cdot 2^{11}) + 2^{5} \\
  &= (s \cdot 2^{21}) + (t \cdot 2^{16}) + (d \cdot 2^{11}) + 2^{5}
\end{align*}

• We managed to express this value in terms of our register numbers!

• We could proceed like this, but there's a more idiomatic way.
Bitwise Operations

• Most programming languages offer these operations, which let you manipulate the bits in a binary value.

• **Bitwise OR:** $x \mid y$ applies logical OR to each pair of bits in $x$ and $y$.
  
  ```
  int x = 10; int y = 12;
  printf("%d\n", x | y); // prints 14
  ```

• **Bitwise AND:** $x \& y$ applies logical AND to each pair of bits in $x$ and $y$.

• There is also **Bitwise XOR** (exclusive OR), denoted $x \wedge y$.

• **Bitwise NOT:** $\sim x$ applies logical NOT to each bit in $x$.
  
  • For example, $\sim x + 1$ takes the two’s complement of $x$. 

Bitwise Operations: Shifting

• The last two bitwise operators are not based on logical operators.
• They are the left bit shift (<<) and right bit shift (>>) operators.
  • In C++, << and >> are also used for stream I/O. This has nothing to do with their use as bitwise operators.
• Left bit shift examples (8 bits):
  
  \[
  \begin{align*}
  \text{00001011} & \quad << \quad 3 \\
  & = \quad \text{01011000} \\
  \text{00011011} & \quad << \quad 6 \\
  & = \quad \text{11000000}
  \end{align*}
  \]
  • The left operand is “the sequence to shift”.
  • The right operand is “how far to shift” in bits.
  • The number is padded with zeroes on the right.
  • Overflowing bits typically get discarded.
Bit Shifting and Signedness

- With left bit shifting, the number is always padded with zeroes.
- For right bit shifting, there are two distinct kinds of shifts:
  - An arithmetic right shift pads with whatever the leftmost bit is. This will preserve the sign of a two’s complement number.
  - A logical right shift pads with zeroes, which may change the sign.
- Examples using 8-bit two’s complement values:

<table>
<thead>
<tr>
<th>10110001 &gt;&gt; 3</th>
<th>-79 &gt;&gt; 3</th>
<th>10110001 &gt;&gt; 3</th>
<th>-79 &gt;&gt; 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110110</td>
<td>-10</td>
<td>00010110</td>
<td>22</td>
</tr>
</tbody>
</table>

Arithmetic right shift | Logical right shift
Encoding "add" with Bitwise Operations

• You will need four pieces of information to encode add $d, $s, $t:
  • The values d, s and t as numbers.
  • The function bits for add. (Use a lookup table)

• The encoding of add $d, $s, $t has the form:

```
000000  sssss  ttttt  ddddd  00000  100000
```

```
↑    ↑     ↑     ↑     ↑     ↑      ↑
```

```
Bit  31   26    21    16    11    6      0
```

• To compute the encoding:

```c
int encoding = (s << 21) | (t << 16) | (d << 11) | fnBits["add"]; // where fnBits["add"] is 32 in decimal
```
int encoding = (s << 21) | (t << 16) | (d << 11) | fnBits["add"];  // where fnBits["add"] is 32 in decimal

s = 2                                      000000 00000 00000 00000 00000 00000 00010

\[2 \times 2^{21} = 2^{22} = 00010\]

t = 1                                      000000 00000 00000 00000 00000 00000 00001
\[1 \times 2^{16} = 2^{17} = 00001\]

d = 3                                      000000 00000 00000 00000 00000 00000 00011
\[3 \times 2^{11} = 2^{12} = 00011\]

fnBits = 32                                000000 00000 00000 00000 00000 00000 100000
\[32 \times 2^{11} = 2^{23} = 100000\]

(2 << 21)                                  000000 00010 00000 00000 00000 00000 00000
\[2^{22} \times 2^{16} = 2^{38} = 00010\]

(1 << 16)                                  000000 00000 00001 00000 00000 00000 00000
\[2^{17} \times 2^{11} = 2^{28} = 00001\]

(3 << 11)                                  000000 00000 00000 00011 00000 00000 00000
\[2^{22} \times 2^{11} = 2^{33} = 00011\]

32                                         000000 00000 00000 00000 00000 00000 100000
\[2^{23} = 100000\]

encoding                                    000000 00010 00001 00011 00000 100000
\[2^{22} \times 2^{11} = 2^{33} = 100000\]

• Note the difference in order between the assembly (add $3, $2, $1) and machine code (s = 2, t = 1, d = 3) !!
Encoding an "sw" Instruction

• Let's now try to encode "sw $31, -4($30)".
• Encoding format for sw $t, i($s):

  \[
  \begin{array}{cccccccc}
  & 1 & 0 & 1 & 0 & 1 & 1 & sssss \\
  & ttttt & iiiiiiiiiiiiiii
  \end{array}
  \]

  \[
  \begin{array}{ccccccc}
  \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow
  \end{array}
  \]

  Bit  31  26  21  16  0

• Note the difference in operand order again.
• The value i is encoded as a 16-bit two’s complement integer.
• We have opcode bits (first six bits) instead of function bits.
• First attempt (WRONG):

  \[
  \text{int encoding} = (\text{opBits["sw"] } \ll 26) | (s \ll 21) | (t \ll 16) | i;
  \]
Encoding an "sw" Instruction

```c
int encoding = (opBits["sw"] << 26) | (s << 21) | (t << 16) | i;
```

• What goes wrong when encoding "sw $31, -4($30)"?

```
opBits = 43 000000 0000 0000 000000000101011
s = 30 000000 0000 0000 0000000000011110
i = -4 111111 11111 11111 11111111111100
```
Encoding an "sw" Instruction

```c
int encoding = (opBits["sw"] << 26) | (s << 21) | (t << 16) | i;
```

- What goes wrong when encoding "sw $31, -4($30)"?

<table>
<thead>
<tr>
<th>opBits</th>
<th>s</th>
<th>t</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>30</td>
<td>31</td>
<td>-4</td>
</tr>
<tr>
<td>000000</td>
<td>000000</td>
<td>000000</td>
<td>0000000000101011</td>
</tr>
<tr>
<td>101011</td>
<td>11110</td>
<td>11110</td>
<td>11111</td>
</tr>
<tr>
<td>000000</td>
<td>000000</td>
<td>000000</td>
<td>000000000011111</td>
</tr>
<tr>
<td>1111111111111100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Encoding an "sw" Instruction

```c
int encoding = (opBits["sw"] << 26) | (s << 21) | (t << 16) | i;
```

- What goes wrong when encoding "sw $31, -4($30)"?

<table>
<thead>
<tr>
<th>opBits</th>
<th>s</th>
<th>t</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>101011</td>
<td>1110</td>
<td>1111</td>
<td>1111111111111100 (No!)</td>
</tr>
</tbody>
</table>
Encoding an "sw" Instruction

int encoding = (opBits["sw"] << 26) | (s << 21) | (t << 16) | i;

• What goes wrong when encoding "sw $31, -4($30")?"

<table>
<thead>
<tr>
<th>opBits</th>
<th>s</th>
<th>t</th>
<th>i</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>30</td>
<td>31</td>
<td>-4</td>
<td>11111111111111111100</td>
</tr>
</tbody>
</table>

(43 << 26) 101011 00000 0000 000000000000000000000000101011
(30 << 21) 00000 00000 00000 0000000000000000000000001110
(31 << 16) 00000 00000 00000 0000000000000000000000001111
-4         11111 11111 11111 11111111111111111100 |

• The 32-bit two's complement encoding of -4 overwrites everything.
Solution: Bit Masking

- You might think to store the offset "i" in a 16-bit two’s complement type, e.g., int16_t.

- This probably won’t work because C++ will automatically "promote" the 16-bit type when it is used alongside 32-bit types.
  - Maybe you can find a way to make it work if you study the promotion rules carefully, but there's an easier method!

- Use bitwise AND for bit masking (selectively zero out parts of a value).

  -4 1111 1111 1111 1111 1111 1111 1111 1100
  0xFFFF (mask) 0000 0000 0000 0000 1111 1111 1111 1100
  (-4 & 0xFFFF) 0000 0000 0000 0000 1111 1111 1111 1100

- The 0 bits in the mask will zero out the corresponding bits in -4.
- The 1 bits in the mask will copy over the corresponding bits in -4.
Encoding an "sw" Instruction (with masking)

```c
int encoding = (opBits["sw"] << 26) | (s << 21) | (t << 16) | (i & 0xFFFF);
```

- Encoding "sw $31, -4($30)"

  opBits = 43  
  s = 30  
  t = 31  
  i = -4  

  (43 << 26)  
  (30 << 21)  
  (31 << 16)  
  (-4 & 0xFFFF)

  encoding

- The same trick is needed for lw and for branch instructions.
Producing Output

• Once we have the encoding in a variable, producing output is easy...

```cpp
int encoding = ...;
```

• The 32-bit (4-byte) encoding of "add $3, $2, $1" is:

```
00000000010000010001100000100000
```

which is 4266016 in decimal.

• `std::cout << encoding;` will print out the 7-byte string "4266016".

• This makes perfect sense. When you print an integer, you normally want an ASCII representation of the decimal value, and not the raw 4 bytes stored in the int variable.

• We need to find a way to **extract** each of the 4 bytes individually.
Extracting the Bytes

- We can use a combination of **right bit shifts** and **masking**.
- Example: Extract the leftmost byte of 0xC001BABE.
  
  In binary: 11000000 00000001 10111010 10111110

  - We can do: (0xC001BABE >> 24) & 0xFF

<table>
<thead>
<tr>
<th>0xC001BABE</th>
<th>11000000 00000001 10111010 10111110</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC001BABE &gt;&gt; 24</td>
<td>11111111 11111111 11111111 11000000</td>
</tr>
<tr>
<td>0xFF</td>
<td>00000000 00000000 00000000 11111111</td>
</tr>
<tr>
<td>result</td>
<td>00000000 00000000 00000000 11000000</td>
</tr>
</tbody>
</table>

- We are assuming an **arithmetic right shift** (pad with leftmost bit) rather than a **logical right shift** (pad with zeroes).
  
  - The masking makes it so there is no difference.
Printing the Bytes

• Hopefully, you learned how to do this in Question 2.
• In Racket you can use the "write-byte" function.
• The equivalent in C++ is "std::putchar".
• You can also use std::cout, if the byte is stored in a char variable.
  • If it's in an int, it will be formatted and printed in decimal.
• In Racket, bit-masking is necessary, because write-byte will complain if you give it a value outside the range 0 to 255.
• In C++, putchar will truncate values larger than one byte to the lowest 8 bits automatically, so you can skip bit-masking.
A Simple Assembler

• We know enough now to write a simple MIPS assembler.

• We made the simplifying assumption that every line contains an instruction or .word directive, so that there is a one-to-one correspondence between assembly lines and machine code words.

• We'll now get back into MIPS programming and write programs that actually have interesting structures like conditionals and loops!

• However, we are not done with our assembler. We'll soon see that these programming techniques are somewhat inconvenient to work with, prompting us to extend our assembler with new features.