(http://www.ualberta.ca/)

CMPUT 229 - Computer Organization and Architecture I

Lab 6: RISC-V to WASM

Introduction

This lab introduces WebAssembly, an assembly language meant for the web. In addition to getting familiar with WebAssembly, students will be visiting and further developing topics they have seen throughout CMPUT 229 such as parsing the binary representation of instructions, managing a data structure in memory, and combining multiple non-trivial tasks into a larger project.

Background

Most of the applications found in the Web are implemented in Javascript. Javascript is widely available and has become a de-facto standard for web applications, and has more recently been used as a target for compilation from other languages. Thus, while many developers may write web applications in a language other than Javascript, the applications have to be compiled to one of the limited options for executable code on a browser (asm.js, PNaCl, etc.) or the code may have to be executed on the browser using a plugin (for example, a Java plugin). WebAssembly (WASM) was designed as an alternative to these previous attempts to develop a common representation of code for the Web. WASM is primarily intended to be a compilation target for other languages (C, C++, etc.) and arose not to replace Javascript but to complement it by offering a more suitable compilation target for other languages. WebAssembly is supported by almost all the major browsers and mobile browsers.

Task

The main task in this assignment is to translate the binary representation of a RISC-V assembly program into the binary representation of a WebAssembly program. The solution will parse each instruction in the RISC-V program and map it to a provided appropriate set of WebAssembly instruction(s). Some special consideration must be given to branch instructions, the immediate values in any I-type or shift RISC-V instruction, and using the `zero` register.

WASM vs. RISC-V

There are many differences between WASM and the RISC-V instruction set. In the RISC-V instruction set, every instruction is 4 bytes long. This simplifies parsing the binary representation of a RISC-V program as the parser can walk a pointer through the representation, incrementing by four after parsing each instruction. However, WASM does not have this guarantee as it can have variable-length instructions. WASM is a bytecode, which means that every instruction has an opcode that is one byte long. Thus, each instruction is at least one byte long. An instruction in WASM is composed of its opcode and its immediates, if present. Immediates may consist of other instructions. As a result, there are no guarantees on the maximum size of a WASM instruction. It is useful to think of WASM 'instructions' as expressions. An expression may be arbitrarily long because it may contain expressions within itself. WASM expressions appear at the bytecode level in postfix notation (https://en.wikipedia.org/wiki/Reverse_Polish_notation). Therefore, the bytes representing the operator **always** come after the bytes representing the operands.

Another way that WASM differs from RISC-V is in the encoding used for literal values. When an application is transferred over the internet, the size of the file must be taken into consideration. WASM was designed to minimize the number of bytes necessary to represent the program. All of the literal values in the program are encoded in LEB128 format, discussed in the LEB128 Literal Format section below. In LEB128, a 32-bit constant may occupy from 1 to 4 bytes. This representation is important when generating WASM instructions and returning the number of bytes generated by the translation program.

WAST and Bytecode Representations

While instructions are binary at the lowest level, it is convenient to also represent them in a text format so that human eyes may read the code. The textual representation format of WASM is called WAST and uses S-expressions. This representation is fairly different from the text representation of RISC-V (more can be read on S-expressions here (https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format)). In this lab, WebAssembly is presented both in text and binary format. However, the solution for this lab must generate the binary representation. Online examples can be found where the WAST S-expressions are presented in either postfix or prefix (https://en.wikipedia.org/wiki/Polish_notation) notation. For example, this prefix ordered expression:

```
(i32.add (get_local $0) (get_local $1))
```

Is equivalent to this postfix ordered expression:

```
(get_local $0) (get_local $1) (i32.add)
```

Although the textual representation of WebAssembly may be presented in either prefix or postfix notation, the bytecode in a WASM module **must** be in postfix order. Consider the following postfix ordered S-expression:

```
(i32.const 1)(set_local $0)
```

In the bytecode representation of this instruction, each component needs to be translated to a corresponding sequence of bytes. The WASM bytecode corresponding to this statement would exist in a WASM module as such:

```
0000023: 41                                        ; i32.const
0000024: 01                                        ; i32 literal
0000025: 21                                        ; set_local
0000026: 00                                        ; local index
```

The bytecode for `i32.const` is `0x41`, the bytecode for the `1` integer literal happens to also be `0x01`, and the bytecode for `set_local` is `0x21`. Finally, the variable `$0` is translated to `0x00`, see Variables for the specifics on how a submission is expected to handle translating variables.

In RISC-V the textual assembly representation of a RISC-V instruction and its binary representation contain the same information. The textual assembly is easier for the programmer to read and write while the binary representation can be interpreted by the CPU. In WebAssembly, WAST is for programmers and the bytecode representation is for interpreters. The main distinction between the binary representation of a RISC-V instruction and the bytecode representation of a WebAssembly instruction is that WASM makes each part of the instruction a discreet number of bytes while RISC-V compresses every part of the instruction into only 4 bytes.

WASM Module

A WebAssembly program contained within a file is called a module. Each module contains sections and each section has a header that contains some information about that section. All sections in a module are optional. Each section's header contains fields but a header only contains fields that are necessary for a particular module. A valuable aspect of a WASM module is that it is very small. The smallest possible valid WASM module is eight bytes: `0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00`. The first four bytes specify that this is a WASM module and the next four bytes specify the WASM version number. Currently, WASM minimum viable product (MVP) development is settled on version 1. In this assignment, these bytes are provided by the `common.s` file. The following sections that are required to run the code produced by the solution are also included in the `common.s` file:

- **The code section** should contain the translated instructions generated by this assignment solution.
- **The function section** defines the WASM function that contains the generated code.
- **The type section** declares the function and its parameter and result type.
- **The export section** allows the resulting module to run.

More information about the layout of a WASM module can be found here (https://webassembly.github.io/spec/core/binary/modules.html).
A solution to this lab should not include section headers in the WASM binary representation.
The solution should only translate the RISC-V code, the rest will be handled in `common.s` to produce the final WASM module.

WASM Translations

The translation from RISC-V to WASM must preserve program semantics. To enable automated marking this assigment defines a clear mapping from categories of RISC-V instructions to WAST expressions. The mapping between RISC-V instructions and WAST expressions that is expected in the generated WASM code is shown below:

	RISC-V Instruction	WAST Expression Format (Prefix)	WAST Expression Format (Postfix)
Instructions Containing Immediates:	`operator var0, var1, immediate`	`(set_local var0 (operator (get_local var1) (i32.const immediate*)))`	`(get_local var1) (i32.const immediate*) (operator) (set_local var0)`
Instructions Containing Variables:	`operator var0, var1, var2`	`(set_local var0 (operator (get_local var1) (get_local var2)))`	`(get_local var1) (get_local var2) (operator) (set_local var0)`
Control Flow Operators Format:			
Forward Branches:	`...` `beq var0, var1, label` `...` `label:` `...`	`(block` `(i32.eq (var0) (var1))` `br_if 0` `(instructions between branch and label)` `end)` `(instructions between label and end of program)`	`(block` `(var0) (var1) (i32.eq)` `br_if 0` `(instructions between branch and label)` `end)` `(instructions between label and end of program)`
Backward Branches:	`...` `label:` `...` `bge var0, var1, label` `...`	`(loop` `(instructions between branch and label)` `(i32.ge_s (var0) (var1))` `br_if 0` `end)`	`(loop` `(instructions between branch and label)` `(var0) (var1) (i32.ge_s)` `br_if 0` `end)`

While the translated code is shown here, branch instructions are discussed in further detail under Branches. Furthermore, WASM requires structured control flow (https://en.wikipedia.org/wiki/Structured_program_theorem) whereas RISC-V allows unstructured control flow. A program that has unstructured control flow can be much more difficult to analyze, by both compilers and humans, than a program with structured control flow. The structured control flow requirement in WASM implies that no branch can branch into the body of a (different) `loop` or `block`. Loops and blocks may be nested. The lab solution is expected to handle nested loops and nested blocks.

All RISC-V programs provided as input in this lab will have structured control flow. Any test case created for this lab should also have structured control flow. An intuitive way of thinking is to avoid inserting a branch in the middle of a loop that specifies a target that is in the middle of another loop. See the class presentation slides in the Resources section for examples of valid/invalid test cases.

The order the variables matters in the translation. For example, given the instruction:

```
add rd, rs1, rs2
```

the following two translations preserve the semantics of the instruction:

```
(get_local rs1) (get_local rs2) (i32.add) (set_local rd)
(get_local rs2) (get_local rs1) (i32.add) (set_local rd)
```

However the WASM modules produced are different. To allow for automated grading, the lab solution must produce the first translation shown above. The general rule for variable order is that the translation must follow the variable order in the postfix section of the above table.

Branches

To adhere to the structured control flow of a WASM module some bytes must be placed into the WASM binary representation when the branch targets are encountered in the RISC-V program representation. This means that the solution needs to calculate the RISC-V branch targets. The address of a branch target is the address of the instruction that is executed immediately after the branch when the branch condition is true --- the branch is taken. Once they are calculated, these bytes can then be inserted appropriately as the translator scans the instructions of the program. The translator must also identify the direction of the branch: forward or backward branches.

Forward Branches

For this assignment, a branch instruction whose target is ahead of the branch instruction in the program is a *forward branch*. In WASM there are no labels to mark the target of branches. Instead, WASM code is structured with blocks. A block has the following format: `(block .. WASM instructions .. end)`. A `br_if` instruction that appears inside a block is a conditional break that transfers the execution to the end of the block. A RISC-V forward branch is translated as the beginning of a block whose first instruction computes the branch condition. The second instruction in this block is a `br_if` instruction that is the conditional break that transfers execution to the end of the block if the condition is true. The syntax for the translation of a forward branch is as follows:

```
(block
    (comparison_operator (var0) (var1))
    br_if 0
...
```

Where `comparison_operator` is a WASM operator that computes the comparison needed for the type of branch that is being translated. For instance, the translation of a `beq` requires `i32.eq` operator. The `i32.ge_s` WASM operator can be used to translate RISC-V `bge`. Bytecode translations are specified in the WASM Instructions section. The `0` after the `br_if` specifies the break depth. In a WASM program `br_if` can break out of multiple blocks. The solution to this lab does not implement this feature. For the solution, break depth is always 0. Here is an example of a branch instruction and its translation:

RISC-V Code	WAST Translation

```
...
    beq t0, t1, label1
    ...
label1:
    ...
```
	```
	...
	(block
	    (i32.eq (get_local 22) (get_local 23))
	    br_if 0
	    ...
	end)
	...
	```

The use of `(get_local 22), (get_local 23)` to translate registers `t0, t1` is specified in the section WASM Variables.

Backward Branches

In general, to determine if a branch is forward or backward a flow analysis that computes a dominance relation is needed. These concepts are thought in advanced compiler courses. Therefore, for this assignment, a branch instruction whose target appears before the branch instruction in the code is classified as a *backward branch*. The translating of `br_if` and the comparison operator is identical to forward branches. The main difference is that instead of inserting a `(block ... end)` control structure, the translator inserts a `(loop ... end)` control structure. The byte code for `loop` must appear right before the target instruction and the `end` byte code appears after the condition and `br_if`. For example:

RISC-V Code	WAST Translation

```
...
label1:
    ...
    beq t0, t1, label1
    ...
```
	```
	...
	(loop
	    ...
	    (i32.eq (get_local 22) (get_local 23))
	    br_if 0
	end)
	...
	```

If an instruction `I` is both the target of a forward branch and the target of a backward branch, the translated WASM bytecode has the `end` of the forward branch block, followed by the `loop` bytecode, followed by `I`. This translation expresses the semantics of the RISC-V program in the structured control flow of WASM.

Behavior of the br_if Instruction

A potential snag when reasoning about how branches work in WASM is that the `br_if` instruction behaves differently in a `(block ... end)` structure versus a `(loop ... end)`. The main difference being that in a block, the `br_if` acts like a `break;` statement in C++, while in a loop it acts like a `continue;` statement. That is, when the branch condition is true, `br_if` branches to the end of a `block`, while in a loop it branches to the beginning, or the instruction right in front of the `(loop` bytecode.

Calculating Targets

The calculation of the offset of a branch instruction or of the target of a jump instruction requires the value contained in the `PC`. During program execution, the `PC` contains the memory address of the next instruction to execute. In this lab, the representation of the program is located in the user data segment of memory. The value of the address used to load the binary representation of an instruction into a register is the value of the `PC` for that instruction.

In RISC-V a branch instruction specifies the distance between the branch instruction and its target as an immediate value. To compute branch targets, the combination of the value of the PC and the value of the immediate must be used. The following algorithm is recommended to compute branch targets:

1. Extract the immediate 12-bits from the branch instruction and save it to the bits[12:1] of a register with bit[0] as 0. This value specifies the byte offset from the branch and the target --- it may be negative.
2. Sign extend the value.
3. Add it to the `PC` value (the address of the branch instruction).

Self-branches
An unintuitive consequence of how branch instructions are being translated in this assignment is that there is no clear/unambiguous way to translate branch instructions that have themself as their target. Thus, no test cases will ever have a self-branching instruction and students do not need to handle this situation in their solution. For example, the following code is not allowed in a test case:

```
self: beq t0, t1, self
```

Table for Branch Targets

The branch targets need to be calculated beforehand to insert `loop` and `end` bytecodes where appropriate. One solution is to precompute all of the branch target addresses and store them somewhere, then for each instruction to be translated, check if it is the target of a forward and/or backward branch. This lab requires the implementation of some table(s) of branch target counts. This(These) table(s) will, for every instruction in the given RISC-V program, store two counters indicating: how many forward branches and how many backward branches target that particular instruction. While translating the RISC-V program, the solution must access these table(s), retrieve the counters, and insert the appropriate number of `end` or `loop` bytecodes for every instruction.

This lab specifies an interface, constituted of three functions, to access the table(s):

- `generateTargetTable` : calculates the appropriate counts for each branch target in the program;
- `readTargetCount` : retrieves the value of the count for a given instruction from the table, for use when translating instructions;
- `incrTargetCount` : increments the count for a specific target address, helper function for `generateTargetTable` ;

Complete specifications of these functions are provided.

The following guarantees for the RISC-V programs to be translated simplify the solution for this lab:

- Every program will have at most 2000 instructions;
- No instruction can be the target of more than 255 branches;

Thus, memory space for the table(s) can be statically allocated in the `.data` section. The implementation may create two separate tables, one for forward and another for backward branch target counts, or it may create a single table with space to store the counters for forward and backward branches. A possible implementation for the described table(s) is to have an array of counts indexed by the position of the instruction in the program. For example, the 10th instruction will be associated with the 10th count in the table, and so on.

Types

Unlike RISC-V, WebAssembly is a typed language. Functions, values, and blocks have types. This lab works only with the 32-bit integer value type, denoted by the notation `i32` .

WASM Variables

A WASM module has a function section that contains all of the function declarations and specifies the variables used in the function. However, this lab is only translating the RISC-V code to a WASM code section and is not concerned with the function section. Therefore, the solution can assume that there is a one-to-one mapping between the registers used in the RISC-V program and the variables declared in the WASM function.

RISC-V Register	WASM Variable	RISC-V Register	WASM Variable
`a0 (x10)`	`0`	`s8 (x24)`	`14`
`a1 (x11)`	`1`	`s9 (x25)`	`15`
`a2 (x12)`	`2`	`s10 (x26)`	`16`
`a3 (x13)`	`3`	`s11 (x27)`	`17`
`a4 (x14)`	`4`	`t3 (x28)`	`18`
`a5 (x15)`	`5`	`t4 (x29)`	`19`
`a6 (x16)`	`6`	`t5 (x30)`	`20`
`a7 (x17)`	`7`	`t6 (x31)`	`21`
`s2 (x18)`	`8`	`t0 (x5)`	`22`
`s3 (x19)`	`9`	`t1 (x6)`	`23`
`s4 (x20)`	`10`	`t2 (x7)`	`24`
`s5 (x21)`	`11`	`s0 (x8)`	`25`
`s6 (x22)`	`12`	`s1 (x9)`	`26`
`s7 (x23)`	`13`	`zero (x0)`	`i32.const 0`

By default, the arguments declared in the WASM module are placed into the first *n* local variables, corresponding to *n* arguments. Thus, in this lab, registers `x10-x17` are mapped to variables `0-7` . In general, the lab variable mapping is not required by a WASM module, but it is a requirement for this assignment to allow automated marking.

The `common.s` file declares four arguments for each module to provide enough bytes for the module to be runnable. Any unused argument will contain 0 because all variables are initialized to 0 in WASM. In general, a WASM module can have an arbitrary number of arguments. In this assignment, modules are declared to have **only** four arguments.

WASM does not have a hardwired zero variable. Therefore any use of the register `zero (x0)` must be changed to a constant value of zero (`i32.const 0`) with corresponding bytes " `0x41 0x00` ". For example, the following table shows one such translation.

RISC-V Instruction	WAST	WASM Byte Codes
`or s1 s2 zero`	`(((get_local 8) (i32.const 0) i32.or) set_local 26)`	`0x20 0x08 0x41 0x00 0x72 0x21 0x1a`

LEB128 Literal Format

In RISC-V it is possible to find literal constants represented in various bit fields in instructions. For example, in an `addi` instruction, the literal immediate value is specified in the highest 12 bits of the instruction. The 32-bit integer value corresponding to an immediate value in RISC-V is obtained by sign extending (or zero extending depending on the instruction) the immediate field.

WASM modules specify literals in LEB128 format. This format aims to represent numbers using a minimal number of bytes. An immediate value in LEB128 can take up anywhere from a single byte to as many as needed, depending on its value. LEB128 uses little-endian byte order.

The binary representation of a number is converted to LEB128 with the following steps:

1. Sign extend the binary representation to the minimum multiple of 7 bits.*
2. Split this sign-extended number into groups of 7 bits.
3. Form a byte with each group of 7 bits by adding a most-significant bit (MSB). Set the MSB to 1 in all but the last group (MSB should be 0 in the last group).

*Note that this should include the sign-bit - that is, the zero following the leading 1-bit if it is positive, or the 1 following the leading 0-bit if it is negative. For example, `64` would need 8 bits: `01000000` and thus be sign-extended to 14 bits total, or 2 groups of 7.

In the LEB128 format, the most-significant bit on every byte serves to indicate if the following bytes should be interpreted as a part of the literal. In the same way that the null-terminator is a sentinel marking the end of a string, the MSB that is 0 is a sentinel marking the end of the number. Below is an example conversion for the decimal value `345566`. There are more examples in the class presentation in the Resources section.

To convert `345566` from decimal to LEB128, first convert it to binary:
`345566 => 1010100010111011110`
Now, break it into groups of 7 bits:
`1010100010111011110 => 0010101 0001011 1011110` (Padding with zeros to next multiple of 7).
Add sentinel bits:
`0010101 0001011 1011110 => 00010101 10001011 11011110 => 0x15 0x8b 0xde`

The encoded number is `0x15 0x8b 0xde`. WebAssembly uses little-endian byte order, therefore this number appears as `0xde 0x8b 0x15` in the WASM representation.

The issue of producing the minimal number of bytes becomes more of a concern when working with negative numbers. For example, consider the number `-34`. The binary representation of `-34` in a signed 12-bit immediate is `111111011110`. If we just directly sign-extend this to the next multiple of 7 bits we get `1111111 1011110`, and we see that the second grouping of `1111111` is extraneous.

Lets do this for the value `-12`, coming from a 12-bit immediate. First, convert it to binary:
`-12 => 111111110100`
Clipping it to the first 1 past the most significant 0 bit:
`111111110100 => 10100`
Sign-extending to the next multiple of 7 bits:
`10100 => 1110100`
And continue as before.

A browser tool can be used to test/check LEB128 encodings, linked in the Testing section.

Some examples of what the expected output of the `encodeLEB128` function is:

Decimal value of the 12-bit immediate	Argument passed in to `a0`	Expected output in `a0`
388	`0x0000 0184`	`0x0000 0384`
-64	`0x0000 0fc0`	`0x0000 0040`
64*	`0x0000 0040`	`0x0000 00c0`
2047	`0x0000 07ff`	`0x0000 0fff`
0	`0x0000 0000`	`0x0000 0000`
-2048	`0x0000 0800`	`0x0000 7080`
-1	`0x0000 0fff`	`0x0000 007f`
1	`0x0000 0001`	`0x0000 0001`

*The zero byte following `0xc0` is still a part of the encoding for this example, be careful to write `0xc0 0x00` to the WASM module in this case.

WASM Instructions

The WASM instruction set has a wide variety of instructions. This assignment only uses the small subset listed below with opcodes in hexadecimal representation.

Instruction	**Opcode**	**Immediates**	**Description**
block	`0x02`	sig : block_type	begin a sequence of expressions, yielding 0 or 1 return values
loop	`0x03`	sig : block_type	begin a block which can also form control flow loops
end	`0x0b`	n/a	end a code section, block, loop, or if
br_if	`0x0d`	relative_depth : varuint32	conditional break that targets an outer nested block
return	`0x0f`	n/a	return zero or one value from this function
get_local	`0x20`	local_index : varuint32	read a local variable or parameter
set_local	`0x21`	local_index: varuint32	write a local variable or parameter
i32.add	`0x6a`	n/a	sign-agnostic addition

i32.sub	`0x6b`	n/a	sign-agnostic subtraction
i32.and	`0x71`	n/a	sign-agnostic bitwise and
i32.or	`0x72`	n/a	sign-agnostic bitwise inclusive or
i32.shl	`0x74`	n/a	sign-agnostic shift left
i32.shr_s	`0x75`	n/a	sign-replicating (arithmetic) shift right
i32.shr_u	`0x76`	n/a	zero-replicating (logical) shift right
void	`0x40`	n/a	signature type void, see below
i32.const	`0x41`	value : varint32	a constant value interpreted as i32
i32.eq	`0x46`	n/a	sign-agnostic compare equal
i32.ge_s	`0x4e`	n/a	signed greater than or equal

The sig immediate specified above for the control-flow instructions indicates a signature that describes the expected stack operations of this block. Blocks are similar to functions in that they interact with a value stack. The expected return type must be specified. If a block or function has no return value the expected return type must still be specified, similar to a void function declaration in C. For this assignment the signature used for all blocks and loops should be void.

RISC-V Instructions to Translate

The following are all of the RISC-V instructions that must be handled by this binary translator. In the encoding, `s` specifies a source register, `t` a target register, `d` a destination register, `i` an immediate value.

Instruction	Encoding	Type
`ANDI    d, s, imm`	`iiii iiii iiii ssss s111 dddd d001 0011`	I
`AND     d, s, t`	`0000 000t tttt ssss s111 dddd d011 0011`	R
`ORI     d, s, imm`	`iiii iiii iiii ssss s110 dddd d001 0011`	I
`OR      d, s, t`	`0000 000t tttt ssss s110 dddd d011 0011`	R
`ADDI    d, s, imm`	`iiii iiii iiii ssss s000 dddd d001 0011`	I
`ADD     d, s, t`	`0000 000t tttt ssss s000 dddd d011 0011`	R
`SUB     d, s, t`	`0100 000t tttt ssss s000 dddd d011 0011`	R
`SRAI    d, s, imm`	`0100 000i iiii ssss s101 dddd d001 0011`	I
`SRLI    d, s, imm`	`0000 000i iiii ssss s101 dddd d001 0011`	I
`SLLI    d, s, imm`	`0000 000i iiii ssss s001 dddd d001 0011`	I
`SRL     d, s, t`	`0000 000t tttt ssss s101 dddd d011 0011`	R
`SLL     d, s, t`	`0000 000t tttt ssss s001 dddd d011 0011`	R
`BEQ     s, t, offset`	`imm[12|10:5]t tttt ssss s000 imm[4:1|11]110 0011`	SB
`BGE     s, t, offset`	`imm[12|10:5]t tttt ssss s101 imm[4:1|11]110 0011`	SB

Remember to be careful when translating srai. The immediate isn't the full 12 bits as the other I-Type instructions and the extra 1 will give incorrect immediates otherwise.

Return Values

At the time that this lab was written, there was **no support** for a WASM module to return multiple values. The organization of a module requires the number of return values for a function to be specified earlier than the code section of the module. The module bytecode provided in the `common.s` specifies a single return value. Thus, the WASM code that the solution generates must always return a value. If the RISC-V code did not specify a return value, the translated WASM function will return 0. The designated return value variable for this lab is `0 (a0)`. It is fine to return this variable even if it was not assigned; all local variables in WASM are initialized to 0.

The return instruction must be in the following format: `(return (get_local 0))`
This should be in postfix notation as the rest of the instructions, so it should be written to the WASM representation as `(get_local 0)(return)`.

This return instruction is to be used as the **second last** instruction in the module.
The solution for this lab must write the `return` instruction followed by the `end` opcode at the end of the translated program before returning from `RISCVtoWASM`.

Assignment

The goal of this assignment is to translate the binary representation of a RISC-V assembly program into the binary representation of a WebAssembly program. This assignment implements a binary translator for a subset of RISC-V instructions. This subset is Turing-complete, and thus anything could theoretically be computed with this subset. Additionally, special consideration must be given to branch instructions, the encoding of immediates in I-type or shift RISC-V instructions, and using the `zero` register.

Specification

The solution for this assignment must implement the following functions:

- `RISCVtoWASM`
  This function translates the binary representation of a RISC-V program into the binary representation of a WASM program.
  Arguments:
  - `a0` : pointer to memory containing the binary representation of a RISC-V function. The end of the RISC-V program is marked by the sentinel word `0xFFFFFFFF` (The integer -1).
  - `a1` : pointer to pre-allocated memory where the generated WASM binary representation of a function must be stored.

  Return Values:
  - `a0` : The number of bytes of WASM code that were generated by the function.

  Side Effects:
  - The generated WASM binary representation of the RISC-V function is stored in the pre-allocated memory specified by `a1` . The end of the binary representation **MUST BE** the instruction `(return (get_local 0))` followed by the `end` opcode.

- `translateIType`
  This function translates an I-Type RISC-V instruction into WASM. It also writes the translated instruction to the binary representation of the WASM code. It is guaranteed that only valid I-Type instructions will be passed to this function when grading.
  Arguments:
  - `a0` : Address to write instruction representation translated into WASM.
  - `a1` : Address of the RISC-V instruction to parse.
  - `a2` : WASM opcode for the WASM instruction to build. **NOT** the RISC-V opcode for the instruction.

  Return Values:
  - `a0` : The number of bytes in the translated WASM instruction.

  Side Effects:
  - The translated instruction is written in the address specified by `a0` .

- `translateRType`
  This function translates an R-Type RISC-V instruction into WASM. It also writes the translated instruction to the binary representation of the WASM code. It is guaranteed that only valid R-Type instructions will be passed to this function when grading.
  Arguments:
  - `a0` : Address to write instruction representation translated into WASM.
  - `a1` : Address of the RISC-V instruction to parse.
  - `a2` : Opcode for the WASM instruction to build. **NOT** the RISC-V opcode for the instruction.

  Return Values:
  - `a0` : The number of bytes in the translated WASM instruction.

  Side Effects:
  - The translated instruction is written in the address specified by `a0` .

- `translateBranch`
  This function translates a branch RISC-V instruction into WASM, with special care being taking to account for differences in forward and backward branches. It also writes the translated instruction to the binary representation of the WASM code. It is guaranteed that only valid branch instructions will be passed to this function when grading.
  Arguments:
  - `a0` : Address to write instruction representation translated into WASM.
  - `a1` : Address of the RISC-V instruction to parse.
  - `a2` : Opcode for the WASM instruction to build. **NOT** the RISC-V opcode for the instruction.

  Return Values:
  - `a0` : The number of bytes in the translated WASM instruction.

  Side Effects:
  - The translated instruction is written in the address specified by `a0` .

- `encodeLEB128`
  This function converts a 12-bit two-complement value into LEB128 representation.
  Arguments:
  - `a0` : A 12-bit two-complement value, stored in the lower half of the word, to be converted into LEB128 representation.

  Return Values:
  - `a0` : The LEB128 representation of the input value. The least-significant byte of the LEB128 representation should be in the least-significant byte of `a0` . Examples of conversions are shown here
  - `a1` : The number of bytes that were needed to represent the resulting LEB128 formatted value.

- `generateTargetTable`
  This function initializes table(s) of counts and then computes all forward and backward branch target counts in the given RISC-V program. Depending on the implementation there can either be one or two tables, however, if there are two, both must be generated here. Calls `incrTargetCount` as a helper function.
  Arguments:
  - `a0` : Pointer to the binary representation of the RISC-V program.

- `readTargetCount`
  This function takes in the address of an instruction and returns the count of branch targets that have it as a target. For use as a helper function when translating instructions.
  Arguments:
  - `a0` : Pointer to the binary representation of the RISC-V program.
  - `a1` : Instruction address to get the count for.
  - `a2` : Flag denoting whether the instruction is the target of a forward or backward branch; 1 = forward, 0 = backward.

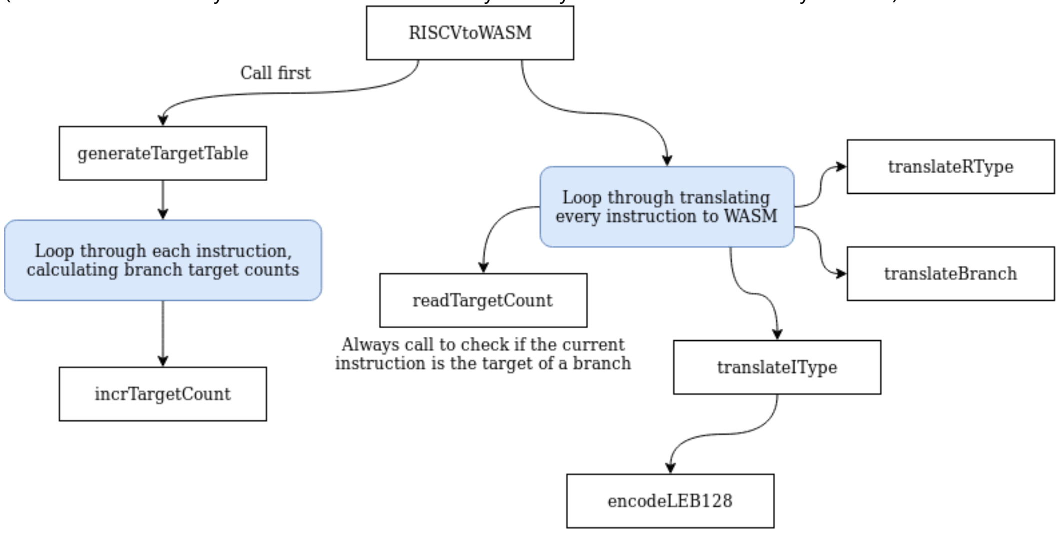
  Return Values:
  - `a0` : The associated count of how many branches have the given instruction as a target.

- `incrTargetCount`
  This function takes in the address of an instruction and increments the count of branch targets that have it as a target by 1. For use as a helper function in `generateTargetTable`.
  Arguments:
  - `a0` : Pointer to the binary representation of the RISC-V program.
  - `a1` : Address of the instruction to increment the count for.
  - `a2` : Flag denoting whether the instruction is the target of a forward or backward branch; 1 = forward, 0 = backward.

Notes and Tips

- Download everything in a zip file. (../Code.zip)
- There are multiple non-trivial parts to this lab so it is **highly** recommended that students get started early. Starting the night before the due date will not end well.
- Initially, the lab can feel overwhelming. However, it is built in such a way to help alleviate this by breaking it down into discrete functions. Each function is a much more digestible task. If you find yourself stuck or not even knowing where to begin, start with implementing and testing the functions by themselves before combining everything in `RISCVtoWASM`. If you still find yourself stuck/overwhelmed, reach out to a TA for help.
- Building on the previous point, write modular code! Break up the code into discrete blocks of instructions that complete a task or even an entire function. Test these logical blocks and functions as you go and the lab will come together much easier than it otherwise would.
- It is possible for there to be a label at the bottom of a RISC-V program. This is a valid edge case that can be easy to miss.
- Use the tools provided for testing as you write the functions! They are there for your benefit and they do help catch mistakes you wouldn't have otherwise seen.
- If you are running out of time or are unable to fully implement the lab, grab partial marks where you can. You will get marks for your LEB128 encoding whether or not you ever translate a single instruction.
- Here is a text document walking through one of the test cases byte-by-byte : link (example_wasm.txt). If you are struggling to picture what the final WASM code looks like, it is highly recommended to walk through this file.

The functions provided in the spec are designed in such a way as to help guide you through solving the overall task of the lab. Below is a diagram depicting the intended control flow of the program, at a function level
(You of course can add your own functions and modify where you make function calls as you see fit):


RISCVtoWASM
Call first
generateTargetTable
Loop through each instruction, calculating branch target counts
incrTargetCount
Loop through translating every instruction to WASM
readTargetCount
Always call to check if the current instruction is the target of a branch
translateRType
translateBranch
translateIType
encodeLEB128

Testing

To generate test cases, write short structured RISC-V programs using the subset of instructions provided, and convert them into binary files using the following command:

```
rars "YOUR_RISCV_FILE" a dump .text Binary "YOUR_DESIRED_BINARY_FILE"
```

The provided common.s (../Code/common.s) file loads RISC-V binary from a file and generates a `.wasm` file after calling the functions specified above. The program in `common.s` takes the name of the file containing the test to load as an argument. Thus, it can be run using `rars common.s pa RISCV_BINARY_FILE`. The submitted solution **must not** contain the `common.s` attached. It also **must not** contain a `main` function.

You are given a browserWASM.html (../Code/browserTool/browserWASM.html) file to run your WASM modules. This implementation uses a WebAssembly Javascript API (https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running). The completion of this assignment **does not** require that the module be run on a browser. However, doing so would allow you to check that the implementation of the `encodeLEB128` function is correct and that the generated WASM preserves the semantics of the input RISC-V program.

This assignment also provides the program WASMDisassembler.s (../Code/WASMDisassembler.s) that prints WASM instructions in a WAST textual representation. The output of the disassembler is one instruction per line and all expressions are presented in *postfix* order. This disassembler prints only values that meet the above spec, producing question marks where no valid interpretation is possible. You are responsible for creating test cases to ensure compliance with the assignment specification. The program can be run using:

```
rars WASMDisassembler.s pa main.wasm
```

The program terminates on the WASM module `return` opcode. Therefore any output should, at the very least, contain a return statement in order to run the disassembler. The disassembler can be tested from the very start of this assignment using the provided `.wasm` files in the browserTool (../Code/browserTool) directory.

To view the bytecode contents of the generated `.wasm` files in a terminal, use the following command:

```
hexdump main.wasm
```

Another tool that may prove useful for testing your submission is the wat2wasm (https://webassembly.github.io/wabt/demo/wat2wasm/) tool, which takes the WAST representation of a program and converts it to the WASM byte code.

Here are some test cases that you can use to test your program.

RISC-V Program	**RISC-V Binary**	**WASM Binary**	**WAST**
BackwardBranch.s (testCases/BackwardBranch.s)	BackwardBranch.bin (testCases/BackwardBranch.bin)	BackwardBranch.wasm (testCases/BackwardBranch.wasm)	BackwardBranc (testCases/Bac
BranchToFirstInstruction.s (testCases/BranchToFirstInstruction.s)	BranchToFirstInstruction.bin (testCases/BranchToFirstInstruction.bin)	BranchToFirstInstruction.wasm (testCases/BranchToFirstInstruction.wasm)	BranchToFirstIr (testCases/Bra
BranchToLastInstruction.s (testCases/BranchToLastInstruction.s)	BranchToLastInstruction.bin (testCases/BranchToLastInstruction.bin)	BranchToLastInstruction.wasm (testCases/BranchToLastInstruction.wasm)	BranchToLastIr (testCases/Bra
ForwardBranch.s (testCases/ForwardBranch.s)	ForwardBranch.bin (testCases/ForwardBranch.bin)	ForwardBranch.wasm (testCases/ForwardBranch.wasm)	ForwardBranch (testCases/For
MultipleBranches.s (testCases/MultipleBranches.s)	MultipleBranches.bin (testCases/MultipleBranches.bin)	MultipleBranches.wasm (testCases/MultipleBranches.wasm)	MultipleBranch (testCases/Mul

Resources

Class slides used for the lab (.pptx (../Slides/Lab_RISCVtoWASM_Class_pres.pptx)) (.pdf (../Slides/Lab_RISCVtoWASM_Class_pres.pdf))

Class presentation for the lab (Google Drive (https://drive.google.com/file/d/1lA7gfn770WnFAVEJuRl-W5kvGfZjvH15/view?usp=sharing)) (YouTube (https://youtu.be/JLXq0_NG0uQ))

Some test cases as well as tools to aid in test case generation are available in the `Code` directory and instructions can be found in the README. (../Code/README)

You can use the online WebAssembly Studio (https://webassembly.studio) to see the translation of C/Rust/WAST code to WASM/WAST.

More information on WebAssembly can be found here (http://webassembly.org/).

Marking Guide

Assignments too short to be adequately judged for code quality will be given a zero. Register translation is vital for all instructions. Therefore it is difficult for a binary translator that does not do correct register translation to pass ANY of the grading test cases. Please, ensure proper register translation according to the table above.

- 10% For correct implementation of encodeLEB128
- 25% For correctly implementing the table(s) of counts and populating it with associated functions
- 15% For correctly translating I-type instructions
- 15% For correctly translating R-type instructions
- 20% For correctly translating branch instructions
- 15% For code cleanliness, readability, and comments

A copy of the marksheet to be used can be found here (Marksheet.txt). For the instruction translations, an incomplete set of translated instructions can still earn part marks.

Submission

Files to submit for this assignment:

- File `wasm.s` should contain the assembly program you wrote.
  - it should contain only the code for the functions specified above, and any functions used in their implementation that you added
  - **Do not** add a `main` label to this file
  - **Do not** modify the line `.include "common.s"`
  - Submit only your `wasm.s`