CS241 – Tutorial 3

Writing an Assembler

University of Waterloo

January 29, 2020
• Assembler is divided into two phases – “Analysis” phase and “Synthesis” phase.
• Goal of the Analysis phase is to read and interpret input, goal of the Synthesis phase is to produce output.
• We do two passes over the code, which don’t directly correspond to the phases.
• First pass is all Analysis, second pass is “finish Analysis and perform Synthesis”.
• Check correctness as much as possible in the first pass.
  • Do instructions have the correct number of operands?
  • Are all the commas, dollar signs, parentheses, etc. in the right places?
  • Do register numbers and integer literals fall in the correct ranges?
  • Label errors can’t be handled until the second pass.
• If a line seems correct, store the important data from the line in a data structure for future processing in the Synthesis phase.
The Symbol Table

- As you read the code in the first pass, construct a *symbol table*, which stores the values of all the labels defined in the code.

- In the Synthesis phase, this table is used to substitute labels with their values, or compute branch offsets using the label values.

- We must do two passes over the code because labels can be used before they are defined:
  
  ```
  beq $0, $0, end ; use of label end
  end: jr $31 ; definition of label end
  ```

- When constructing the symbol table, *remember to check for duplicate label definitions!*

  ```
  label:
  add $1, $2, $3
  label: ; duplicate definition, not allowed
  ```
During the second pass, before producing output we must finish checking correctness:

- Check for uses of undefined labels.
- Check that label operands, when converted to addresses or offsets, fall in the correct ranges.

These checks require a complete symbol table, so they could not be done in the first pass.

It may seem convenient to do integer range checks and label range checks at the same time, but it’s best to split it across the two passes to catch errors as soon as possible.

If these final checks for an instruction pass, write the binary form of the instructions into a buffer.

When you’re sure the whole program is correct, write the buffer to standard output!
Construct the symbol table for the following MIPS assembly program.

begin:
  label: beq $0, $0, after
  jr $4

after:
  sw $31, 16($0)
  lis $4
  abc0: abc1: .word after

loadStore:
  lw $20, 4($0)
  sw $20, 28($0)

end:
Mark the lines containing instructions (including `.word`).

```
begin:
X label: beq $0, $0, after
X jr $4

after:
X sw $31, 16($0)
X lis $4
X abc0: abc1: .word after

loadStore:
X lw $20, 4($0)
X sw $20, 28($0)

end:
```
Symbol Table Example

Make a list of labels in the program.

begin:
X label: beq $0, $0, after
X jr $4

after:
X sw $31, 16($0)
X lis $4
X abc0: abc1: .word after

loadStore:
X lw $20, 4($0)
X sw $20, 28($0)

end:
Count the marked lines preceding each label, and multiply by 4.

begin:
X label: beq $0, $0, after
X jr $4

after:
X sw $31, 16($0)
X lis $4
X abc0: abc1: .word after

loadStore:
X lw $20, 4($0)
X sw $20, 28($0)

end:

begin 0
label 0
after 8
abc0 16
abc1 16
loadStore 20
end 28

The table above is what you should output to standard error in A3P3.
Symbol Table Implementation Hints

- In C++, you can use the `std::unordered_map` class to store your symbol table. The underlying implementation is a hash table, which provides fast lookups on average.
- The `std::map` data structure will also work (lookups are slower on average but faster in the worst case).
- Even though label values represent memory addresses, which are non-negative, you should store them as signed integers in C++.
  - Why? To compute a branch offset, you need to do the calculation $(\text{labelValue} - \text{instructionLocation} - 4) / 4$.
  - If `labelValue` is an unsigned type, the other values in the subtraction will be converted to unsigned, and integer underflow will occur.
  - The variable `instructionLocation` should also be of signed type even though it represents a memory address.
- Keep separate counts of “number of lines read” and “number of instructions read”.
- Line numbers are useful for informative error messages, but label values should be based on the number of instructions read.
Representing Instructions and Operands

- Create an Instruction class that stores instruction information.
- In the Analysis phase, convert input lines to instances of the Instruction class, for easy processing in the Synthesis phase.
- Some ideas for what this class could contain:
  - The memory address where the instruction will appear in the output.
  - The line number on which the instruction appears in the input (optional, useful for informative error messages).
  - The opcode of the instruction (add, sub, etc.)
  - The operands of the instruction.
  - An `assemble` function that returns the corresponding binary data.
- Note that operands can be numbers or labels. What variable type should we use?
- If they’re integer type, we can’t fill in the correct integer for label operands until the second pass.
- Simple solution: just store the tokens returned by the scanner as operands. Tokens can represent numbers or labels.
• What type should the opcode be in our Instruction class?

• It may seem like it should obviously be a string, but note that you cannot use strings with switch statements in C++:

```cpp
// This is not allowed in C++
switch(stringVariable) {
    case "Hello": /* do something */ break;
    default: /* do something else */ break;
}
```

• If you want the convenience of being able to use switch statements, consider storing the opcode as an enum.

• You will need to do the initial conversion from string to enum with a series of if/else statements, but then you can use switch statements to check which instruction you are dealing with.
Range Checking

- Range checking can surprisingly difficult to get correct in this assignment if you don’t approach it correctly. We will look at the three types of range checks you must do, from easiest to hardest.

  - Register range checking:

  - Just check if the register number is between 0 and 31 inclusive.

  - .word range checking:

  - The operand for .word can be a decimal integer, a hexadecimal integer, or a label.

  - If it is a decimal integer, it must lie in the range $-2^{31}$ to $2^{32} - 1$ ($-2147483648$ to $4294967295$).

  - If it is a hexadecimal integer, it cannot exceed $0xffffffff$.

  - For label values, use the same range as decimal.

  - Question: can we use the same range check for decimal and hexadecimal?
Range Checking

- .word range checking (continued):
- The `Token::toLong` function provided in the C++ starter code behaves a bit strangely with regards to hexadecimal integers:
  - When it reads the hexadecimal integer, it will interpret it as unsigned.
  - However, it will then try to store it in a signed 64-bit integer variable.
  - If the unsigned value of the hexadecimal integer is too large, it will be capped at the maximum signed 64-bit value ($2^{63} - 1$).
  - Thus hexadecimal integers returned by `Token::toLong` are always non-negative.
  - This means decimal integers, hexadecimal integers and label values can all be handled by a single range check: $-2^{31}$ to $2^{32} - 1$ inclusive.
  - This advice applies if you are using the provided `Token::toLong` method and storing the value in a 64-bit integer type – if you are extracting the number in some other way or using a smaller integer type you might have problems!
Range Checking

- Immediate range checking:
- For `lw` and `sw`, the immediate value can be a decimal integer or hexadecimal integer.
- For `beq` and `bne`, labels are also allowed, and are converted to immediates via `(labelValue - instructionLocation - 4) / 4`.
- If the immediate is a decimal integer, it must be in the range $-2^{15}$ to $2^{15} - 1$ (−32768 to 32767).
- If the immediate is a hexadecimal integer, it cannot exceed `0xffffffff`.
- Label offsets use the decimal range.
- Since `Token::toLong` always treats hexadecimal values as unsigned, `0xffffffff` will be interpreted as $2^{16} - 1 = 65535$, which is outside of the decimal range.
- This means two separate range checks are required: one for decimal integers and labels, and one for hexadecimal integers.
- The decimal integer and label range check is $-2^{15}$ to $2^{15} - 1$ inclusive.
- The hexadecimal integer range check is 0 to $2^{16} - 1$ inclusive.
Your assembler must output raw binary data just like cs241.binasm.

To output a 32-bit (4-byte) binary word in C++, we use bitwise operations to store each byte of the word in a variable of char type.

When outputting a char, C++ will output the actual binary data stored in the variable.

```cpp
void output_word(int word) {
    char c;
    c = (word >> 24) & 0xff;
    std::cout << c;
    c = (word >> 16) & 0xff;
    std::cout << c;
    c = (word >> 8) & 0xff;
    std::cout << c;
    c = word & 0xff;
    std::cout << c;
}
```
• Modify the `asm.cc` starter file to print out a count of the number of
  lines which contain a syntactically valid `.word` directive.

• Such a line consists of zero or more label definitions, followed by a
  `.word` directive with exactly one operand (decimal integer,
  hexadecimal integer, or label), and nothing after the operand (except
  possibly comments).

• To make things simpler, since this is just a practice problem:
  • You do not need to check the range of the operands.
  • You do not need to check whether the label operands have a
    corresponding definition.
  • You may assume all input lines can be tokenized by the given scanner.
    That is, a ScanningFailure exception will never be thrown.