COMP 300: Lab 0 - Assembler

Assembler

It is difficult and error-prone to manually write machine code. To address this problem, people usually use an assembler, which automatically generates a machine code from an assembly file. Moreover, many C compilers first generate assembly files and then simply feed it to an assembler to get the executable machine code. For these reasons, you should write an assembler for your ISA. The assembler reads a program written in an assembly language, then translates it into binary code and generates two output files containing executable machine code. You will use the generated output files for both the simulator and the actual hardware you will implement.

Assembler Framework

In order to save your time in developing a new assembler, you can extend the java-based assembler framework to generate the executable machine code. The core of our assembler framework is an abstract class "Assembler". To complete your assembler, you may create a new class that inherits the Assembler class and realize the abstract methods in the Assembler class. The detailed description of our assembler framework is located at:

http://home.sandiego.edu/~sat/comp300/assembler.html

The Assembler class is available on Blackboard.

Input and Output Requirements

- Your assembler should accept following command line input:
  - *[$name].s* : a single .s (assembly test bench) file
  - *[$prefix]* : the prefix of your assembler output.
- Your assembler have two outputs - *$prefix_i.coe* and *$prefix_d.coe*. *$prefix_i.coe* corresponds to a 17-bit instruction memory while *$prefix_d.coe* corresponds to a 34-bit data memory. They must have the following forms:
  - *$prefix_i.coe* - 17 bit word size

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
00000,
00001,
00002,
```

```
00003,
00004,
.....,
1DEAD
<EOF>
```

- *$prefix_d.coe* - 34 bit word size

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
000000000,
000000001,
000000002,
000000003,
000000004,
.........,
3DEADBEEF
<EOF>
```

- The first line in each output file specifies the numerical format used in the file. We use hexadecimal notation. Note that each entry of the 'MEMORY_INITIALIZATION_VECTOR' is data for each word, starting from address 0x0. Thus an entry of the instruction memory is 17-bit, while that of the data memory is 34-bit. (For historical reasons, the framework supports 17 bits instruction width. You are expected to work within the instruction width provided by the framework by padding zeros to the high order bits of your instructions to make it 17 bits. For example, if you choose a 14-bit ISA, you should pad 3 zeroes to the left of each instruction while implementing the assembler. That is, an instruction 0x3BAD becomes 0x03BAD. *This does not change the fact that your ISA has 14 bit wide instructions.*) These output files will be used as inputs to your simulator.
- *Aside:* This is also the file format that Xilinx design tools used to initialize memory ROM and RAM modules. If you were using Xilinx design tools to implement a processor that adheres to your ISA, you would use these .coe files directly. Since you are using Altera tools to realize processor design, you will not be able to use .coe files directly--Altera tools use files in Intel Hex format to initialize ROMs and RAMs. In order to use the COE files you'll be generating in this class to initialize ROMs and RAMs in the lab class, you can use hex_converter.java to convert COE (*.coe) files to HEX files (*.hex). Run the script with the -h option to see how to use it.
- Since a COE file always starts out at 0x0, you must fill out unused memory area from 0x0 to *-text_addr* or *-data_addr* with arbitrary values (typically 0x0). However, for other unused memory locations, you do not need to specify values; unspecified memory locations will have arbitrary values.

- Assembler example usage:
  - ***prompt> java asm garbage.s garbage***

    The assembler will generate two files, *garbage_i.coe* and *garbage_d.coe*, which contain the translated .text and .data components of *garbage.s* and will be loaded by the simulator at addresses 0x0 of the instruction memory and 0x0 of the data memory, respectively.