H

hw1-doc

| Name | Last update |
| --- | --- |
| README.md (/cse320/hw1-doc/blob/master/README.md) | 2 days ago |

**README.md** (/cse320/hw1-doc/blob/master/README.md)

# Homework 1 - CSE 320 - Spring 2018

**Professor Eugene Stark**

**Due Date: Sunday 02/11/2018 @ 11:59pm**

**Read the entire doc before you start**

## Introduction

In this assignment, you will write a command line utility to translate MIPS machine code between binary and human-readable mnemonic form. The goal of this homework is to familiarize yourself with C programming, with a focus on input/output, strings in C, and the use of pointers.

You **MUST** write your helper functions in a file separate from `main.c`. The `main.c` file **MUST ONLY** contain `#include`s, local `#define`s and the `main` function. This is the only requirement for project structure. Beyond this, you may have as many or as few additional `.c` files in the `src` directory as you wish. Also, you may declare as many or as few headers as you wish. In this document, we use `hw1.c` as our example file containing helper functions.

# Getting Started

Fetch base code for `hw1` as described in `hw0`. You can find it at this link:
https://gitlab02.cs.stonybrook.edu/cse320/hw1 (https://gitlab02.cs.stonybrook.edu/cse320/hw1).

Both repos will probably have a file named `.gitlab-ci.yml` with different contents. Simply merging these files will cause a merge conflict. To avoid this, we will merge the repos using a flag so that the `.gitlab-ci.yml` found in the `hw1` repo will be the file that is preserved. To merge, use this command:

```
git merge -m "Merging HW1_CODE" HW1_CODE/master --strategy-option theirs
```

Here is the structure of the base code:

```
hw1
├── include
│   ├── const.h
│   ├── debug.h
│   ├── hw1.h
│   └── instruction.h
```

```
├── Makefile
├── rsrc
│   ├── bcond.asm
│   ├── bcond.bin
│   ├── examples.asm
│   ├── examples.bin
│   ├── jump.asm
│   ├── jump.bin
│   ├── matmult.asm
│   ├── matmult.bin
│   ├── typei.asm
│   ├── typei.bin
│   ├── typer.asm
│   └── typer.bin
├── src
│   ├── hw1.c
│   ├── instr_table.c
│   └── main.c
└── tests
    └── hw1_tests.c
```

> 🤓 Reference for pointers: http://beej.us/guide/bgc/output/html/multipage/pointers.html (http://beej.us/guide/bgc/output/html/multipage/pointers.html%5D).
>
> 🤓 Reference for command line arguments: http://beej.us/guide/bgc/output/html/multipage/morestuff.html#clargs (http://beej.us/guide/bgc/output/html/multipage/morestuff.html#clargs).

**Note**: All commands from here on are assumed to be run from the `hw1` directory.

## A Note about Program Output

What a program does and does not print is VERY important. In the UNIX world stringing together programs with piping and scripting is commonplace. Although combining programs in this way is extremely powerful, it means that each program must not print extraneous output. For example, you would expect `ls` to output a list of files in a directory and nothing else. Similarly, your program must follow the specifications for normal operation. One part of our grading of this assignment will be to check whether your program produces EXACTLY the specified output. If your program produces output that deviates from the specifications, even in a minor way, or if it produces extraneous output that was not part of the specifications, it will adversely impact your grade in a significant way, so pay close attention.

**Use the debug macro `debug` (described in the 320 reference document in the Piazza resources section) for any other program output or messages you many need while coding (e.g. debugging output).**

# Part 1: Program Operation and Argument Validation

In this part, you will write a function to validate the arguments passed to your program via the command line. Your program will support the following flags:

- If no flags are provided, you will display the usage and return with an `EXIT_FAILURE` return code
- If the `-h` flag is provided, you will display the usage for the program and exit with an `EXIT_SUCCESS` return code
- If the `-a` flag is provided, you will perform text-to-binary conversion (i.e. "assembly"), reading text from `stdin` and writing binary to `stdout`.
- If the `-d` flag is provided, you will perform binary-to-text conversion (i.e. "disassembly"), reading binary from `stdin` and writing text to `stdout`.

> The `-a` and `-d` flags are not allowed to be used in combination with each other
>
> 🤓 `EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>` which represent success and failure return codes respectively.
>
> 🤓 `stdin`, `stdout`, and `stderr` are special files that are opened upon execution for all programs and do not need to be reopened.

Some of these operations will also need other command line arguments which are described in each part of the assignment. The two usages for this program are:

```
usage: ./hw1 -h [any other number or type of arguments]
usage: bin/hw1 [-h] -a|-d [-b BASEADDR] [-e ENDIANNESS]
    -a       Assemble: convert mnemonics to binary code
    -d       Disassemble: convert binary code to mnemonics
             Additional parameters: [-b BASEADDR] [-e ENDIANNESS]
                -b            BASEADDR is the starting memory address for the code
                              It must be a hexadecimal number of 8 digits or less
                -e            ENDIANNESS specifies the byte order of the binary code
                              It must be a single character:
                                 b for big-endian, or
                                 l for little-endian
    -h       Display this help menu.
```

A valid invocation of the program implies that the following hold about the command-line arguments:

- All positional arguments ( `-a|-d` ) come before any optional arguments ( `-b and -e` ). The optional arguments may come in any order after the positional ones.
- If the `-h` flag is provided, it is the first positional argument after the program executable.
- If an option requires a parameter, the corresponding parameter must be provided (e.g. `-e` must always be followed by an ENDIANNESS specification).
- If `-b` is given, the BASEADDR argument will be given as a hexadecimal number in which in addition to the digits ('0'-'9) either upper-case letters ('A'-'F') or lower-case letters ('a'-'f') may be used, in any combination.
- If `-e` is given, then the ENDIANNESS argument will be a single word (i.e. will have no whitespace).

> 😱 You may only use `argc` and `argv` for argument parsing and validation. Using any libraries that parse command line arguments (e.g. `getopt` ) is prohibited.
>
> 😱 Any libraries that help you parse strings are prohibited as well ( `string.h`, `ctype.h`, etc). *This is intentional and will help you practice parsing strings and manipulate pointers.*
>
> 😱 You **MAY NOT** use dynamic memory allocation in this assignment (i.e. `malloc`, `realloc`, `calloc`, `mmap`, etc)

For example, the following are a subset of the possible valid argument combinations:

- `$ bin/hw1 -h ...`
- `$ bin/hw1 -a`
- `$ bin/hw1 -a -e b`
- `$ bin/hw1 -d -b D000d000 -e l`

Some examples of invalid orderings would be:

- `$ bin/hw1 -e b -d`
- `$ bin/hw1 -b D000d000 -a -e b`

> 😱 The `...` means that all arguments, if any, are to be ignored; e.g. the usage `bin/hw1 -h -a -b D00D000 -e b` is equivalent to `bin/hw1 -h`

**NOTE:** The makefile compiles the `hw1` executable into the `bin` folder. Assume all commands in this doc are run from from the `hw1` directory of your repo.

### Required Validate Arguments Function

In `const.h`, you will find the following function prototype (function declaration) already declared for you. You **MUST** implement this function as part of the assignment.

```
/**
 * @brief Validates command line arguments passed to the program.
 * @details This function will validate all the arguments passed to the
 * program, returning 1 if validation succeeds and 0 if validation fails.
 * Upon successful return, the selected program options will be set in the
 * global variable "global_options", where they will be accessible
 * elsewhere in the program.
 *
 * @param argc The number of arguments passed to the program from the CLI.
 * @param argv The argument strings passed to the program from the CLI.
 * @return 1 if validation succeeds and 0 if validation fails.
 * Refer to the homework document for the effects of this function on
 * global variables.
 * @modifies global variable "global_options" to contain a bitmap representing
 * the selected options.
 */
int validargs(int argc, char **argv);
```

> 😱 This function must be implemented as specified as it will be tested and graded independently. **It should always return -- the USAGE macro should never be called from validargs.**

The `validargs` function should return 0 if there is any form of failure. This includes, but is not limited to:

- Invalid number of arguments (too few or too many)
- Invalid ordering of arguments
- A missing parameter to an option that requires one (e.g. `-e` with no ENDIANNESS specification).
- Invalid base address (if one is specified). A base address is invalid if it contains characters other than the digits ('0'-'9), upper-case letters ('A'-'F'), and lower-case letters ('a'-'f'), if it is more than 8 digits in length, or if it is not a multiple of 4096 (i.e. the twelve least-significant bits of its value are not all zero).
- Invalid endianness (if one is specified). An endiannness is invalid if either it does not consist of a single character or that single character is not either 'b' or 'l'.

The `global_options` variable of type `unsigned int` is used to record the mode of operation (i.e. assemble/disassemble) of the program, as well as any selected flags and base address. This is done as follows:

- If the `-h` flag is specified, the least significant bit is 1
- The second least significant bit is 0 if `-a` is passed (i.e. the user wants assembly mode) and 1 if `-d` is passed (i.e. the user wants disassembly mode)
- The third least signficant bit is 1 if `-e b` is passed (i.e. the user wants big-endian byte ordering) and 0 otherwise.
- If the `-b` option was specified, then the base address is given by taking the value of `global_options` and clearing the 12 least significant bits. If the `-b` option was not specified, then the 20 most significant bits of `global_options` should all be 0 (i.e. the default base address is 0).

If `validargs` returns 0 indicating failure, your program must print `USAGE(program_name, return_code)` and return `EXIT_FAILURE`. **Once again, `validargs` must always return, and therefore it must not call the `USAGE(program_name, return_code)` macro itself. That should be done in `main`.**

If `validargs` sets the least significant bit of `global_options` to 1 (i.e. the `-h` flag was passed), your program must print `USAGE(program_name, return_code)` and return `EXIT_SUCCESS`.

> 🤓 The `USAGE(program_name, return_code)` macro is already defined for you in `const.h`.

If validargs returns 1 and the least significant bit of `global_options` is 0, your program must perform assembly or disassembly accordingly and return `EXIT_SUCCESS` upon successful completion, or `EXIT_FAILURE` in case of an error.

If `-b` is provided, you must check to confirm that the specified base address is valid.

If `-e` is provided, you must check that the specified endianness is either the single character `b` or the single character `l`.

> 🤓 Remember `EXIT_SUCCESS` and `EXIT_FAILURE` are defined in `<stdlib.h>`. Also note, `EXIT_SUCCESS` is 0 and `EXIT_FAILURE` is 1.
>
> 🤓 We suggest that you create functions for each of the operations defined in this document. Writing modular code will help you isolate and fix problems.

### Sample validargs Execution

The following are examples of `global_options` settings for given inputs. Each input is a bash command that can be used to run the program. In the examples, all don't care bits (bits 3-11, where the least significant bit is numbered 0 and the most significant bit is numbered 31) have been set to 0.

- Input: `bin/hw1 -h`. Setting: 0x1 ( `help` bit is set. All other bits are don't cares.)
- Input: `bin/hw1 -d`. Setting: 0x2 ( `disassemble` bit is set).
- Input: `bin/hw1 -d -e b`. Setting: 0x6 ( `disassemble` and `big endian` bits are set).
- Input: `bin/hw1 -d -e b -b BaB000`. Setting: 0xBAB006 ( `disassemble` and `big endian` bits are set, base address is 0xBAB000).
- Input: `bin/hw1 -e b -d -b BaB000`. Setting: 0x0. This is an error case because the argument ordering is invalid ( `-e` is before `-d` ). In this case `validargs` returns 0, leaving `global_options` unset.

# Part 2: MIPS Instruction Format

Presumably you learned something about the MIPS process and its instruction set in CSE 220. If you need to, review the materials used for that course. You might also find useful information via this link (https://en.wikipedia.org/wiki/MIPS_architecture#Instruction_formats). Below we summarize the information about the MIPS instruction format that will be needed to do the assignment.

Each MIPS instruction consists of one 32-bit word. We will number the bits from 0 (least significant bit) to 31 (most significant bit) and we will think of bit 31 as being "leftmost". To indicate a particular bit field from the instruction word we will use a notation like 31:26, which indicates bits 31 down to 26; that is, the 6 "leftmost", or most significant bits.

In every MIPS instruction, bit field 31:26 is used as a 6-bit opcode. Most instructions are directly identified by one of the 64 possible values of this field, but as we will see there are some special cases. There are three types of MIPS instructions: R, I, and J. Instructions of type R take up to three registers as arguments. Instructions of type I take up to two registers and a 16-bit immediate value (obtained from the 16 least significant bits of the instruction word). Instructions of type J take a jump target from the 26 least signficant bits of the instruction word. The MIPS processor has 32 registers, which means that it takes 5 bits to specify a register. The registers are specified by the contents of bit fields 25:21 (called `RS` ), 20:16 (called `RT` ), and 15:11 (called `RD` ), or, in some cases, bit field 10:6.

In the files `instruction.h` and `instr_table.c` you have been provided with a set of tables that can be used to decode MIPS binary instruction words. Rather than going through full details of the MIPS instruction format, we will just go through the procedure for decoding an instruction using the tables. The type `Opcode` is an enumerated type that assigns to integer values in the range 0 to 63 the names of MIPS instructions, and in addition defines names for three additional values `SPECIAL` (64), `BCOND` (65), and `ILLEGL` (66). `Opcode` values in the range 0 to 63 serve as indices into the instruction table `instrTable`. Each entry in this table uniquely identifies a particular type of MIPS instruction and provides further information about it. Our first objective in decoding an instruction is to determine the proper `Opcode` value (in the discussion below we refer to this as "the Opcode"), thereby obtaining access to the proper entry from the instruction table.

The starting point for obtaining the Opcode is the value in bits 31:26 of the instruction word. This value is used as an index into `opcodeTable` and the value (of type `Opcode`) at that index in the table is retrieved.

- If the value obtained from `opcodeTable` is neither `SPECIAL` nor `BCOND`, then it is the Opcode.
- If the value obtained from `opcodeTable` is `SPECIAL` (this occurs when the value of bits 31:26 is 000000), then the value in bits 5:0 of the instruction word is used as an index into the table `specialTable` to obtain the Opcode.
- If the value obtained from `opcodeTable` is `BCOND`, then the value in bits 20:16 is examined. If the value is 00000, 00001, 10000, or 10001, then the Opcode is `OP_BLTZ`, `OP_BGEZ`, `OP_BLTZAL`, or `OP_BGEZAL`, respectively, otherwise it is an error.

Having determined the Opcode, it is then used as an index into `instrTable` and the corresponding `Instr_info` structure is retrieved. What happens next depends on the value of the `type` field. This value can be `NTYP` (which occurs in a few entries of the table that do not correspond to actual instructions), `RTYP`, which indicates an instruction of type R, `ITYP`, which indicates an instruction of type I, and `JTYP`, which indicates an instruction of type J.

The next task is to determine the sources of the instruction arguments. For this, the information in the `srcs` field of the `Instr_info` structure is used. This field consists of an array of three values of type `Source`. The first entry in this array specifies the source of the first instruction argument, the second entry specifies the source of the second instruction argument, and the third entry specifies the source of the third argument. There are five possible source values: `RS`, `RT`, `RD`, `EXTRA`, and `NSRC`. The value `RS` indicates that the argument source is the register specified by the RS field of the instruction word. Similarly, the values `RT` and `RD` the argument source is the register specified by the RT or RD field of the instruction word, respectively. The value `EXTRA` indicates that the argument value has to be decoded from the instruction word in a way that depends on the particular type of instruction. The value `NSRC` is used as a place-holder value for instructions that take fewer than three arguments.

For arguments with source `EXTRA`, the actual argument is determined as follows:

- If the Opcode is `OP_BREAK`, then the argument consists of the 20-bit value in bits 21:6 of the instruction word.
- For instructions of type R, the argument consists of the 5-bit value in bits 10:6 of the instruction word.
- For instructions of type I, the argument is obtained by extracting the 16-bit value in bits 15:0, treating bit 15 as a sign bit, and performing sign-extension to a 32-bit signed integer. For non-branch instructions of type I (such as `ADDI`), this 32-bit signed integer is the immediate argument to the instruction.

For the conditional branch instructions `BEQ`, `BGEZ`, `BGEZ`, `BGEZAL`, `BGTZ`, `BLEZ`, `BLTZ`, `BLTZAL`, `BNE`, the 32-bit signed integer value is further processed by shifting it left by two bits (which amounts to multiplication by 4) and then treating it as a PC-relative branch offset. It is added to the current value of the PC register (this will be the memory address at which the instruction "lives", plus 4) to obtain an absolute address which is the branch target.

- For instructions of type J, the argument is obtained by extracting the 26-bit value in bits 25:0 of the instruction word and treating it as an unsigned integer. This value is shifted left by two bits and then added to the value obtained from the PC by zeroing the 28 least significant bits, to obtain an absolute address that is the jump target. (As above, the PC value is given by the memory address of the instruction, plus 4.)

### Example 1:

The instruction word is 0x00c72820, which when written in binary is:

```
0000 0000 1100 0111 0010 1000 0010 0000
OOOO OOSS SSST TTTT DDDD D      FF FFFF
```

The letters written underneath the bits indicate the various bit fields: `O` for the opcode field in bits 31:26, `S` for the RS field in bits 25:21, `T` for the RT field in bits 20:16, `D` for the RD field in bits 15:11, and `F` for the function code in bits 5:0. The value in bits 31:26 is 000000; i.e. 0. Using this as an index into `opcodeTable` yields `SPECIAL`, so it is then necessary to use the value in bits 5:0 as an index into `specialTable`. This index is 100000, or 32 in decimal, and the entry at that index is `OP_ADD`. The corresponding entry in `instrTable` indicates that the `ADD` instruction is of type R, and that the three arguments are given by RD, RS, and RT. The value of RD (in bits 15:11) is 00101 indicating that the first argument is register 5. The value of RS (in bits 25:21) is 00110 indicating that the second argument is register 6. The value of RT (in bits 20:16) is 00111 indicating that the third argument is register 7. So the mnemonic form of this instruction is `add $5,$6,$7`.

### Example 2:

The instruction word is 0x8cc50007, which when written in binary is:

```
1000 1100 1100 0101 0000 0000 0000 0111
OOOO OOSS SSST TTTT XXXX XXXX XXXX XXXX
```

The value in bits 31:26 is 100011, or 35. Using this as an index into `opcodeTable` yields OP_LW. The corresponding entry from instrTable indicates that the instruction is of type I, with first argument source RT, second argument source EXTRA, and the third argument source RS. RT is 00101 so the first argument is register 5. RS is 00110 so the third argument is register 6. The second argument is obtained from bits 15:0, which have the value 7. So the mnemonic form of this instruction is `lw $5,7($6)`.

### Example 3:

The instruction word is 0x10effc1f, which when written in binary is:

```
0001 0000 1110 1111 1111 1100 0001 1111
OOOO OOSS SSST TTTT XXXX XXXX XXXX XXXX
```

The value in bits 31:26 is 000100, or 4. Using this as an index into `opcodeTable` yields OP_BEQ. The corresponding entry from instrTable indicates that the instruction is of type I, with first argument source RS, second argument source RT, and the third argument source EXTRA. RS is 00111 so the first argument is register 7. RT is 01111 so the third argument is register 15. The third argument is obtained from bits 15:0, which is `fc1f` in hex. This 16-bit value is sign-extended to the 32-bit signed value `fffffc1f`, which is then shifted two bits to obtain `ffff f07c`, or -3972 in decimal. This is the PC-relative branch offset. This offset is added to the current value of the PC (i.e. the memory address of the instruction, plus 4) to obtain the final absolute address that is the branch target. Assuming the memory addess of this instruction is `1000` in hex, or 4096 in decimal, the branch target is 4096 + 4 - 3972, or 128 in decimal. So the mnemonic form of this instruction is `beq $7,$15,128`.

### Example 4:

The instruction word is 0x08000400, which when written in binary is:

```
0000 1000 0000 0000 0000 0100 0000 0000
OOOO OOXX XXXX XXXX XXXX XXXX XXXX XXXX
```

The value in bits 31:26 is 000010, or 2. Using this as an index into `opcodeTable` yields OP_J. The value in bits 25:0 is 0000400, which is shifted left two bits to obtain `00001000` in hex. Assuming that the memory address of the instruction is `40000000` in hex, the PC value at the time of execution would be `40000004`. Clearing the 28 least-significant bits yields `40000000`, and adding this to `00001000` yields `40001000` in hex. So the mnemonic form of this instruction is `j 0x40001000`.

> 😱 The MIPS instruction set does not support jumps to addresses whose four most-significant bits differ from those of the current PC value. Consequently, an attempt to assemble a jump instruction (i.e. `J` or `JAL`) whose target address differs in its four most-significant bits from the base address supplied with `-b` should be treated as an error.

# Part 3: Required `encode` and `decode` functions

In order to provide some additional structure for you, as well as to make it possible for us to perform additional unit tests on your program, you are required to implement the two functions below as part of your program. The prototypes for these functions are given in `const.h`. Once again, you **MUST** implement these functions as part of the assignment, as we will be testing them separately.

```
/**
 * @brief Computes the binary code for a MIPS machine instruction.
 * @details This function takes a pointer to an Instruction structure
 * that contains information defining a MIPS machine instruction and
 * computes the binary code for that instruction.  The code is returne
 * in the "value" field of the Instruction structure.
 *
 * @param ip The Instruction structure containing information about the
 * instruction, except for the "value" field.
 * @param addr Address at which the instruction is to appear in memory.
 * The address is used to compute the PC-relative offsets used in branch
 * instructions.
 * @return 1 if the instruction was successfully encoded, 0 otherwise.
 * @modifies the "value" field of the Instruction structure to contain the
 * binary code for the instruction.
 */
int encode(Instruction *ip, unsigned int addr);
```

```
/**
 * @brief Decodes the binary code for a MIPS machine instruction.
 * @details This function takes a pointer to an Instruction structure
 * whose "value" field has been initialized to the binary code for
 * MIPS machine instruction and it decodes the instruction to obtain
 * details about the type of instruction and its arguments.
 * The decoded information is returned by setting the other fields
 * of the Instruction structure.
 *
 * @param ip The Instruction structure containing the binary code for
 * a MIPS instruction in its "value" field.
 * @param addr Address at which the instruction appears in memory.
 * The address is used to compute absolute branch addresses from the
 * the PC-relative offsets that occur in the instruction.
 * @return 1 if the instruction was successfully decoded, 0 otherwise.
 * @modifies the fields other than the "value" field to contain the
 * decoded information about the instruction.
 */
int decode (Instruction *ip, unsigned int addr);
```

These functions each take as an argument a pointer to a structure of type `Instruction`, which is defined in `instruction.h`. The `decode` function assumes that the `value` field has been set to the binary code for a MIPS instruction, and it decodes this value to fill in the other fields. The `info` field should be set to a pointer to the appropriate `Instr_info` structure obtained from `instrTable`. The entries of the `regs` array should be set to the contents of the RS, RT, and RD fields of the instruction word. (**Note:** these should always be set even if the particular instruction does not use those fields.) The `extra` field should be set to the "extra" argument decoded from the instruction word. As this is done differently for each type of instruction, this does not have to be set unless the instruction uses `EXTRA` as an argument source. The entries of the `args` field should be set to the final values of the instruction arguments, as required in order print out the instruction in mnemonic form (i.e. `args[0]` corresponds to the first `%` in the `format` string, `args[1]` corresponds to the second `%`, and `args[2]` to the third). If the instruction takes fewer than three arguments, the unused entries (i.e. the ones with src `NSRC`) should be set to 0.

The `encode` function does the inverse operation from `decode`: it assumes that all the fields other than `value` have been set, and it computes the binary code for the instruction and stores it in the `value` field.

> 😱 You should not define additional tables to help you map instruction mnemonics to `Opcode` values for implementing `encode`. Instead, to perform this mapping you should use a linear search of the existing `instrTable`. You should use the `sscanf` function to match a format string from the `instrTable` against a mnemonic instruction read from the input. The mapping defined by the `specialTable` should be inverted using a similar linear scan approach.

The implementation of `validargs`, `encode`, and `decode` constitutes most of the work involved in implementing the program. Once these have been written, finishing the program should be easy.

One requirement we have yet to consider is the endianness option. Recall that "endianness" refers to the order in which the bytes in a multi-byte quantity are stored in memory or written to a file. In "little-endian" byte order, the **least-significant** byte is stored at the lowest-numbered memory address or written first to a file. In "big-endian" byte order, the **most-significant** byte is stored or written first.

The default mode of operation of your program should be to use little-endian byte order for reading and writing binary MIPS code. However, if the `-e b` option is specified, then big-endian ordering should be used instead.

# Part 4: Running the Completed Program

In either assembly or disassembly mode, the program reads from `stdin` and writes to `stdout`. In assembly mode, since the input is text, it is possible to enter to enter assembly code directly from the terminal:

```
$ bin/hw1 -a
add $5,$6,$7
j 0x1000
```

**NOTE:** In the above example, the program encrypts one line at a time and stops encrypting after it reads `^d` (control-d) from `stdin`. Entering `^d` into a terminal in a UNIX system signals an `EOF` (end of file) to the program.

If you run the program in assembly mode this way, the binary output of the program will also be sent to the terminal. This binary data will appear as "garbage" in the output. To avoid this, the binary output should be redirected, either to a file or else via a pipe to a program that can produce a printable representation of it.

To redirect the output to a file `hw1.out`, you can use:

```
$ bin/hw1 -a > hw1.out
add $5,$6,$7
j 0x1000
$ echo $?
0
```

> 🤓 The `>` symbol tells the shell to perform "output redirection": the file `hw1.out` is created (or truncated if it already existed -- be careful!) and the output produced by the program is sent to that file instead of to the terminal.
>
> 🤓 `$?` is an environment variable in bash which holds the return code of the previous program run. In the above, the `echo` command is used to display the value of this variable.

The contents of `hw1.out` can then be viewed using the `od` ("octal dump") command:

```
$ od -X hw1.out
0000000 00c72820 08000400
0000010
```

🤓 The `-X` flag instructs `od` to interpret the file as a sequence of 32-bit words, which are printed as 8-digit hexadecimal values. In this case, the file contains two such words: `00c72820` and `08000400`. The values in the first column indicate the offsets from the beginning of the file, specified as 7-digit octal (base 8) numbers.

Alternatively, the output of the program could be redirected via a "pipe" to the `od` command, without using any file:

```
$ bin/hw1 -a | od -X
add $5,$6,$7
j 0x1000
0000000 00c72820 08000400
0000010
```

🤓 In this case, you won't see the output produced by `od` until `^d` has been typed, because when the output of a program is redirected to a pipe the system assumes that the program is being run non-interactively, so for efficiency it buffers a larger amount of the output rather than emitting it a line at a time.

In disassembly mode, it is not very useful to read the input from the terminal, since it would be very difficult to generate the necessary binary data using the keyboard. Instead, the input should be redirected *from* a file:

```
$ bin/hw1 -d < hw1.out
add $5,$6,$7
j 0x1000
```

Finally, a pipe can be used to assemble and disassemble in a single run. This is one way to test whether your program is working properly:

```
$ bin/hw1 -a | bin/hw1 -d
add $5,$6,$7
j 0x1000
add $5,$6,$7
j 0x1000
```

The output should be identical to the input if the program is working properly.

## Testing Your Program

In testing your program, it is useful to be able to compare two files to see if they have the same content. The `diff` command (use `man diff` to read the manual page) is useful for comparison of text files. On the other hand, the `cmp` command can be used to perform a byte-by-byte comparison of two files, regardless of their content:

```
$ cmp file1 file2
```

If the files have identical content, `cmp` exits silently. If one file is shorter than the other, but the content is otherwise identical, `cmp` will report that it has reached `EOF` on the shorter file. Finally, if the files disagree at some point, `cmp` will report the offset of the first byte at which the files disagree.

We can take this a step further and run an entire test without using any files:

```
$ cmp <(echo "j 0x1000") <(echo "j 0x1000" | bin/hw1 -a | bin/hw1 -d)
$ echo $?
0
```

🤓 `<(...)` is known as process substitution. It is allows the output of the program(s) inside the parentheses to appear as a file for the outer program.

Because both strings are identical, `cmp` outputs nothing.

Finally, we can test the program on entire files with a similar command:

```
$ cmp <(cat rsrc/bcond.asm) <(cat rsrc/bcond.asm | bin/hw1 -a -b 1000 | bin/hw1 -d -b 100
$ echo $?
0
```

🤓 `cat` is a command that outputs a file to `stdout`.

## Unit Testing

Unit testing is a part of the development process in which small testable sections of a program (units) are tested individually to ensure that they are all functioning properly. This is a very common practice in industry and is often a requested skill by companies hiring graduates.

🤓 Some developers consider testing to be so important that they use a work flow called test driven development. In TDD, requirements are turned into failing unit tests. The goal is then to write code to make these tests pass.

This semester, we will be using a C unit testing framework called Criterion (https://github.com/Snaipe/Criterion), which will give you some exposure to unit testing. We have provided a basic set of test cases for this assignment.

The provided tests are in the `tests/hw1_tests.c` file. These tests do the following:

- `validargs_help_test` ensures that `validargs` sets the help bit correctly when the `-h` flag is passed in.
- `validargs_disassem_test` ensures that `validargs` sets the Disassembly bit correctly when the `-d` flag is passed in.
- `help_system_test` uses the `system` syscall to execute your program through Bash and checks to see that your program returns with `EXIT_SUCCESS`.

### Compiling and Running Tests

When you compile your program with `make`, a `hw1_tests` executable will be created in your `bin` directory alongside the `hw1` executable. Running this executable from the `hw1` directory with the command `bin/hw1_tests` will run the unit tests described above and print the test outputs to `stdout`. To obtain more information about each test run, you can use the verbose print option: `bin/hw1_tests --verbose=0`.

The tests we have provided are very minimal and are meant as a starting point for you to learn about Criterion, not to fully test your homework. You may write your own additional tests in `tests/hw1_tests.c`. However, this is not required for this assignment. Criterion documentation for writing your own tests can be found here (http://criterion.readthedocs.io/en/master/).

# Hand-in instructions

**TEST YOUR PROGRAM VIGOROUSLY!**

Make sure your directory tree looks like this and that your homework compiles:

```
hw1
├── include
│   ├── const.h
```

```
│   ├── debug.h
│   ├── hw1.h
│   ├── instruction.h
│   └── ... Any additional .h files you defined
├── Makefile
├── rsrc
│   ├── bcond.asm
│   ├── bcond.bin
│   ├── examples.asm
│   ├── examples.bin
│   ├── jump.asm
│   ├── jump.bin
│   ├── matmult.asm
│   ├── matmult.bin
│   ├── typei.asm
│   ├── typei.bin
│   ├── typer.asm
│   ├── typer.bin
│   └── ... Any sample text files given or created (will not be used in graded)
├── src
│   ├── hw1.c
│   ├── instr_table.c
│   └── main.c
└── tests
    ├── hw1_tests.c
    ├── instr_table.c
    └── ... Any additional criterion test files you may have written
```

This homework's tag is: `hw1`

```
$ git submit hw1
```

> 🤓 When writing your program try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly!