The Assignment

Recall that an assembler translates code written in mnemonic form in assembly language into machine code.

You will implement an assembler that supports a subset of the MIPS32 assembly language (specified below). The assembler will be implemented in C and executed on Linux. Your assembler will take a file written in that subset of the MIPS32 assembly language, and write an output file containing the corresponding MIPS32 machine code (in text format).

Supported MIPS32 Assembly Language

The subset of the MIPS assembly language that you need to implement is defined below. The following conventions are observed in this section:

- The notation `(m:n)` refers to a range of bits in the value to which it is applied. For example, the expression

  `(PC+4)(31:28)`

  refers to the high 4 bits of the address `PC+4`.
- `imm16` refers to a 16-bit immediate value; no assembly instruction will use a longer immediate
- `offset` refers to a literal applied to an address in a register; e.g., `offset(rs)`; offsets will be signed 16-bit values
- `label` fields map to addresses, which will always be 16-bit values if you follow the instructions
- `sa` refers to the shift amount field of an R-format instruction; shift amounts will be nonnegative 5-bit values
- `target` refers to a 26-bit word address for an instruction; usually a symbolic notation
- Sign-extension of immediates is assumed where necessary and not indicated in the notation.
- C-like notation is used in the comments for arithmetic and logical bit-shifts: $>>_a$, $<<_l$ and $>>_l$
- The C ternary operator is used for selection: `condition ? if-true : if-false`
- Concatenation of bit-sequences is denoted by `||`.
- A few of the specified instructions are actually pseudo-instructions. That means your assembler must replace each of them with a sequence of one or more other instructions; see the comments for details.

You will find the *MIPS32 Architecture Volume 2: The MIPS32 Instruction Set* to be a useful reference for machine instruction formats and opcodes, and even information about the execution of the instructions. See the Resources page on the course website.

MIPS32 assembly .data section:

`.word` This is used to hold one or more 32 bit quantities, initialized with given values. For example:

```
var1: .word 15       # creates one 32 bit integer called var1 and initializes
                     # it to 15.

array1: .word  2, 3, 4, 5, 6, 7  # creates an array of 6 32-bit integers,
                                 # initialized as indicated (in order).

array2: .word 2:10   # creates an array of 10 32 bit integers and initializes
                     # element of the array to the value 2.
```

`.asciiz` This is used to hold a NULL (0) terminated ASCII string. For example:

```
hello_w: .asciiz "hello world"   # this declaration creates a NULL
                                 # terminated string with 12 characters
                                 # including the terminating 0 byte
```

MIPS32 assembly .text section:

Your assembler must support translation of all of the following MIPS32 assembly instructions. You should consult the *MIPS32 Instruction Set* reference for opcodes and machine instruction format information. That said, we will evaluate your output by comparing it to the output produced by MARS 4.4; if there are any disagreements between the MIPS32 manual and MARS, follow the approach taken by MARS.

Your assembler must support the following basic load and store instructions:

```
lw    rt, offset(rs)          # Transfers a word from memory to a register
                              # GPR[rt] <-- Mem[GPR[rs] + offset]

sw    rt, offset(rs)          # Transfers a word from a register to memory
                              # Mem[GPR[rs] + offset] <-- GPR[rt]

lui   rt, imm16               # Loads a constant into the upper half of a register
                              # GPR[rt] <-- imm16 || 0x0000
```

Note that the constant offset in the load and store instructions (`lw` and `sw`) may be positive or negative.

Your assembler must support the following basic arithmetic/logical instructions:

```
add   rd, rs, rt              # signed addition of integers
                              # GPR[rd] <-- GPR[rs] + GPR[rt]

addi  rt, rs, imm16           # signed addition with 16-bit immediate
                              # GPR[rt] <-- GPR[rs] + imm16

addiu rt, rs, imm16           # unsigned addition with 16-bit immediate
                              # GPR[rt] <-- GPR[rs] + imm16

and   rd, rs, rt              # bitwise logical AND
                              # GPR[rd] <-- GPR[rs] AND GPR[rt]

andi  rt, rs, imm16           # bitwise logical AND with 16-bit immediate
                              # GPR[rd] <-- GPR[rs] AND imm16

mul   rd, rs, rt              # signed multiplication of integers
                              # GPR[rd] <-- GPR[rs] * GPR[rt]

nop                           # no operation
                              # executed as:  sll  $zero, $zero, 0

nor   rd, rs, rt              # bitwise logical NOR
                              # GPR[rd] <-- !(GPR[rs] OR GPR[rt])

or    rd, rs, rt              # bitwise logical OR
                              # GPR[rd] <-- GPR[rs] OR GPR[rt]

ori   rt, rs, imm16           # bitwise logical OR with 16-bit immediate
                              # GPR[rd] <-- GPR[rs] OR imm16

sll   rd, rt, sa              # logical shift left a fixed number of bits
                              # GPR[rd] <-- GPR[rs] <<l sa

slt   rd, rs, rt              # set register to result of comparison
                              # GPR[rd] <-- (GPR[rs] < GPR[rt] ? 0 : 1)
```

```
slti  rs, rt, imm16         # set register to result of comparison
                            # GPR[rd] <-- (GPR[rs] < imm16 ? 0 : 1)

sra   rd, rt, sa            # arithmetic shift right a fixed number of bits
                            # GPR[rd] <-- GPR[rs] >>a sa

sub   rd, rs, rt            # signed subtraction of integers
                            # GPR[rd] <-- GPR[rs] - GPR[rt]
```

Your assembler must support the following basic control-of-flow instructions:

```
beq   rs, rt, offset        # conditional branch if rs == rt
                            # PC <-- (rs == rt ? PC + 4 + offset <<l 2)
                            #                  : PC + 4)

blez  rs, offset            # conditional branch if rs <= 0
                            # PC <-- (rs <= 0 ? PC + 4 + offset <<l 2)
                            #                 : PC + 4)

bltz  rs, offset            # conditional branch if rs < 0
                            # PC <-- (rs < 0 ? PC + 4 + offset <<l 2)
                            #                : PC + 4)

bne   rs, rt, offset        # conditional branch if rs != rt
                            # PC <-- (rs != rt ? PC + 4 + offset <<l 2)
                            #                  : PC + 4)

j     target                # unconditional branch
                            # PC <-- ( (PC+4)(31:28) || (target <<l 2))

syscall                     # invoke exception handler, which examines $v0
                            # to determine appropriate action; if it returns,
                            # returns to the succeeding instruction; see the
                            # MIPS32 Instruction Reference for format
```

Your assembler must support the following pseudo-instructions:

```
ble   rs, rt, offset        # conditional branch if rs <= rt
                            # PC <-- (rs <= rt ? PC + 4 + offset <<l 2)
                            #                  : PC + 4)
                            # pseudo-translation:
                            #     slt   at, rt, rs
                            #     beq   at, zero, offset

blt   rs, rt, offset        # conditional branch if rs < rt
                            # PC <-- (rs < rt ? PC + 4 + offset <<l 2)
                            #                 : PC + 4)
                            # pseudo-translation:
                            #     slt   at, rs, rt
                            #     bne   at, zero, offset

la    rt, label             # load address label to register
                            # GPR[rd] <-- label
                            # pseudo-translation for 16-bit label:
                            #     addi  rt, $zero, label
```

```
li     rt, imm16              # load 16-bit immediate to register
                              # GPR[rd] <-- imm
                              # pseudo-translation:
                              #     addiu  rt, $zero, imm16

lw     rt, label              # load word at address label to register
                              # GPR[rd] <-- Mem[label]
                              # pseudo-translation:
                              #     lw   rt, label[15:0]($zero)
```

MIPS32 assembly format constraints:

The assembly programs will satisfy the following constraints:

- Labels will begin in the first column of a line, and will be no more than 32 characters long. Labels are restricted to alphanumeric characters and underscores, and are always followed immediately by a colon character (`':'`).
- Labels in the `.text` segment will always be on a line by themselves.
- Labels in the `.data` segment will always occur on the same line as the specification of the variable being defined.
- Labels are case-sensitive; that actually makes your task a bit simpler.
- MIPS instructions do not begin in a fixed column; they are preceded by an arbitrary amount of whitespace (possibly none).
- Blank lines may occur anywhere; a blank line will always contain only a newline character.
- Whitespace will consist of spaces, tab characters, or a mixture of the two. Your parsing logic must handle that.
- Registers will be referred to by symbolic names (`$zero`, `$t5`) rather than by register number.
- Instruction mnemonics and register names will use lower-case characters.
- Assembly source files will always be in UNIX format.

You must be sure to test your implementation with all the posted test files; that way you should avoid any unfortunate surprises when we test your implementation.

Input

The input files will be MIPS assembly programs in ASCII text. The assembly programs will be syntactically correct, compatible with the MARS MIPS simulator, and restricted to the subset of the MIPS32 instruction set defined above. Example programs will be available from the course website.

Each line in the input assembly file will either contain an assembly instruction, a section header directive (such as `.data`) or a label (a jump or branch target). The maximum length of a line is 256 bytes.

Your input file may also contain comments. Any text after a `'#'` symbol is a comment and should be discarded by your assembler. Section header directives, such as `.data` and `.text` will be in a line by themselves. Similarly, labels (such as `loop:`) will be on a line by themselves. The input assembly file will contain one data section, followed by one text section.

Your assembler can be invoked in either of the following ways:

```
assemble <input file> <output file>
assemble -symbols <input file> <output file>
```

The specified input file must already exist; if not, your program should exit gracefully with an error message to the console window. The specified output file may or may not already exist; if it does exist, the contents should be overwritten.

Output

Output when invoked as: `assemble <input file> <output file>`

Your assembler will resolve all references to branch targets in the .text section and variables in the `.data` section and convert the instructions in the `.text` section into machine code.

To convert an instruction into machine code follow the instruction format rules specified in the class textbook. For each format (R-format, I-format or J-format), you should determine the opcode that corresponds to instruction, the values for the register fields and any optional fields such as the function code and shift amount fields for arithmetic instructions (R-format) and immediate values for I-format instructions.

The output machine code should be saved to the output file specified in the command line. The output file should contain the machine code corresponding to instructions from the `.text` section followed by a blank line followed by variables from the `.data` section in human readable binary format (0s and 1s). For example to represent the decimal number `40` in 16-bit binary you would write `0000000000101000`, and to represent the decimal number -40 in 16-bit binary you would write `1111111111011000`.

The output file is a text file, not a binary file; that's a concession to the need to evaluate your results.

Your output file should match the machine file generated by the MARS simulator version 4.4 (see following section). A sample showing the assembler's translation of the `adder.asm` program is given at the end of this specification.

Output when invoked as: `assemble -symbols <input file> <output file>`

Your assembler will write (to the specified output file) a well-formatted table, listing every symbolic name used in the MIPS32 assembly code and the address that corresponds to that label. Addresses will be written in hex.

Note that MARS makes the following assumptions about addresses:

- The base address for the text segment is 0x00000000, so that's the address of the first machine instruction.
- The base address of the data segment is 0x00002000, so that's the address of the first thing declared in the data segment.

The second fact above implies that the text segment cannot be longer than 8 KiB or 2048 machine instructions. You don't need to do anything special about that fact.

How can I verify my output or test my code?

In order to test your assembler you should use the MARS simulator, version 4.5. A link to the MARS site is posted on the Resources page of the course website. You will need to do the following steps in the MARS simulator:

1. First, ensure that MARS is configured to start the text segment at address 0x00000000. If you do not do this, MARS will use default values for the start addresses for the text and data segments, and that will result in 32-bit addresses for various labels.

   i) Open the MARS simulator and modify the memory configuration settings through **Settings->Memory Configuration** and select **Compact, Text at Address 0** and DO NOT modify any of the remaining addresses on the right. Click **Apply and Close** to exit memory configuration settings.

2. Now, in order to generate the actual machine file for the assembly program, you will need to dump the **binary text format** for the text and data sections. Load your program into MARS and select **Run->Assemble**.

   i) Generate machine code for .text section of your assembly program.

   Open the assembly program and select **File->Dump memory**. Select the **.text (0x00000000 - 0x00000044)** in the **Memory Segment** and select **Binary Text** for dump format and click on **dump to a file** button and specify an output file (say *text_segment_of_add_asm.txt*) to dump the machine code of the assembly program. Now you have the machine instructions for your text segment of your assembly program.

   Note that the actual end address of `.text` segment will vary depending on your assembly program. For the sample input file `add.asm`, attached with this project, you will find that the text segment starts at `0x00000000` and extends up to `0x00000044`.

   ii) Generate machine code for your `.data` segment of your assembly program.

   a) Open the assembly program and select **File->Dump memory** from the MARS IDE.

   b) Select the **.data (0x00002000 - 0x00002ffc)** in the **Memory Segment** and select **Binary Text** for dump format and click on **dump to a file** button. Give an output file (say *data_segment_of_add_asm.txt*) to dump the machine code of the assembly program. Now you have the machine instructions for your data segment of your assembly program.

   Note that the actual end address of `.data` section will vary depending on your assembly program. For the `add.asm` example above, you will find that data segment starts at `0x00002000` and extends up to `0x00002ffc`. Similar to the text segment, the actual size of the data segment depends on your input file.

   iii) Copy the contents of your "text_segment_of_adder_asm.txt" and your "data_segment_of_adder_asm.txt" with a blank line in between them to distinguish the two segments into another file (say `add_asm.txt`). Now you have your output machine file ready. Your assembler program should produce an output identical to this machine file, except that your representation of the `.data` segment should end with the last byte of the final variable declared within it.

The sample machine file posted with this project on the course website was generated using the above approach. Note that the text segment should always start at 0x00000000 and the data segment should start at 0x00002000.

MARS generates a fixed-size data segment, and will pad with 0's to achieve that. Your assembler should <u>not</u> do this.

It is also possible to run MARS from the command-line and generate the dumps of the text and data segments that way; be aware that MARS generates somewhat different results when run in that manner.

Sample Assembler Input

```
# adder.asm
#
# The following program initializes an array of 10 elements and computes
# a running sum of all elements in the array. The program prints the sum
# of all the entries in the array.

.data
message:    .asciiz "The sum of numbers in array is: "
array:      .word   0:10          # array of 10 words
array_size: .word   10            # size of array

.text
main:
        la   $a0, array           # load address of array
        la   $a1, array_size      # load address of array_size
        lw   $a1, 0($a1)          # load value of array_size variable
loop:
        sll  $t1, $t0, 2          # t1 = (i * 4)
        add  $t2, $a0, $t1        # t2 contains address of array[i]
        sw   $t0, 0($t2)          # array[i] = i
        addi $t0, $t0, 1          # i = i+1
        add  $t4, $t4, $t0        # sum($t4) = ($t4 + array[i])
        slt  $t3, $t0, $a1        # $t3 = ( i < array_size)
        bne  $t3, $zero, loop     # if ( i < array_size ) then loop

        li   $v0, 4               # system call to print string
        la   $a0, message         # load address of message into arg register
        syscall                   # make call

        li   $v0, 1               # system call to print an integer
        or   $a0, $t4, $zero      # load value to print into arg register
        syscall                   # make call

        li   $v0, 10              # system call to terminate program
        syscall                   # make call
```

Sample Assembler Output

Here's my assembler's output when invoked as `assemble adder.asm adder.asm`:

```
00100000000000100001000000100100   la
00100000000001010010000001001100   la
10001100101001010000000000000000   lw
00000000000100001001000100000000   sll
00000000100010010101000000100000   add
10101101010010000000000000000000   sw
00100001000010000000000000000001   addi
00000001100010000110000000100000   add
00000001000001010101100000101010   slt
00010101011000001111111111111001   bne
00100100000000100000000000000100   li
00100000000001000010000000000000   la
00000000000000000000000000001100   syscall
00100100000000100000000000000001   li
00000001100000000010000000100101   or
00000000000000000000000000001100   syscall
00100100000000100000000000001010   li
00000000000000000000000000001100   syscall

00100000011001010110100001010100
00100000011011010111010101110011
01101110001000000110011001101111
01100101011000100110110101110101
01101001001000000111001101110010   message
01110010011000010010000001101110
00100000011110010110000101110010
00100000001110100111001101101001
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000   array
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000001010   array_size
```

Text segment of `adder.asm`

blank line

Data segment of `adder.asm`

Data segment notes:

The characters making up `message` are written in a rather puzzling manner. If you examine the bytes of the first line of the data segment:

`00100000 01100101 01101000 01010100` represents the characters `" ehT"`

So, the characters are displayed in 4-byte chunks, in something like big-endian order. The last character stored for the message is `01101001` or `'i'`. That is followed by a zero-byte to terminate the string. That is followed by three more zero-bytes to pad to the next value; that's because the next value is a 32-bit integer and MIPS requires 32-bit values to be aligned on word boundaries.

Next, `array` is stored as a sequence of 32-bit values, all zeros.

The value of `array_size` is 10, which is expressed in hex as `0x0000000A`. In the data segment display above, the value is displayed in big-endian byte order:

```
          00000000 00000000 00000000 00001010
      Low address                    High address
```

You must be sure that you write the bytes of integers in big-endian order as well.

Here's my assembler's output when invoked as `assemble -symbols adder.asm adder.sym`:

```
Address       Symbol
-------------------------------
0x00002000    message
0x00002024    array
0x0000204C    array_size
0x00000000    main
0x0000000C    loop
```

The order in which you display the symbols is up to you. My implementation writes them in the order they occur in the source file, but that is not a requirement.

Advice

The following observations are purely advisory, but are based on my experience, including that of implementing a solution to this assignment. These are advice, not requirements.

First, and most basic, analyze what your assembler must do and design a sensible, logical framework for making those things happen. There are fundamental decisions you must make early in the process of development. For example, you could represent the machine instructions in a number of ways as you build them. They can be represented as arrays of individual bits (which could be integers or characters), or they can be represented in binary format, which would be the expected format for a "real" assembler's final output. I am not convinced that either of those approaches is inherently better, or that there are not reasonable alternatives. But, this decision has ramifications that will propagate throughout your implementation.

It helps to consider how you would carry out the translation from assembly code to machine instructions by hand. If you do not understand that, you are trying to write a program that will do something you do not understand, and your chances of success are reduced to sheer dumb luck.

Second, and also basic, practice incremental development! This is a sizeable program, especially so if it's done properly. My solution, including comments, runs something over 1500 lines of code. It takes quite a bit of work before you have enough working code to test on full input files, but unit testing is extremely valuable.

Record your design decisions in some way; a simple text file is often useful for tracking your deliberations, the alternatives you considered, and the conclusions you reached. That information is invaluable as your implementation becomes more complete, and hence more complex, and you are attempting to extend it to incorporate additional features.

Write useful comments in your code, as you go. Leave notes to yourself about things that still need to be done, or that you are currently handling in a clumsy manner.

A preprocessing phase is helpful; for example, it gives you a chance to filter out comments, trim whitespace, and gather various pieces of information. Do not try to do everything in one pass. Compilers and assemblers frequently produce a number of intermediate files and/or in-memory structures, recording the results of different phases of execution.

Consider how you would carry out the translation of a MIPS32 assembly program to machine code if you were doing it manually. If you don't understand how to do it by hand, you cannot write a program to do it!

Take advantage of tools. You should already have a working knowledge of gdb. Use it! The debugger is invaluable when pinning down the location of segfaults; but it is also useful for tracking down lesser issues if you make good use of breakpoints and watchpoints. Some memory-related errors yield mysterious behavior, and confusing runtime error reports. That's especially true when you have written past the end of a dynamically-allocated array and corrupted the heap. This sort of error can often be diagnosed by using valgrind.

Enumerated types are extremely useful for representing various kinds of information, especially about type attributes of structured variables. For example:

```
enum Breed {BEAGLE, CHOW, SHIH_TSU, GOLDEN_RETRIEVER, MIXED};

struct _Dog {
   char*        Name;
   enum Breed   Type;
   unsigned int Weight;
};
typedef struct _Dog Dog;
```

Think carefully about what information would be useful when analyzing and translating the assembly code. Much of this is actually not part of the source code, but rather part of the specification of the assembly and machine languages. Consider using static tables of structures to organize language information; by static, I mean a table that's directly initialized when it's declared, has static storage duration, and is private to the file in which it’s created. For example:

```
#define TABLE_SZ  20

static Dog Table[TABLE_SZ] = {

      {"Amber", GOLDEN_RETRIEVER, 110},  // initializes first struct element,
      {"Dammit", BEAGLE, 27},            //  etc.
      {"Sassy", MIXED, 30},
      . . .
};
```

Obviously, the sample code shown above does not play a role in my solution. On the other hand, I used this approach to organize quite a bit of information about instruction formats and encodings. It's useful to consider the difference between the inherent attributes of an instruction, like its opcode, and situational attributes that apply to a particular occurrence of an instruction, like the particular registers it uses. Inherent attributes are good things to keep track of in a table. Situational attributes must be dealt with on a case-by-case basis.

Also, be careful about making assumptions about the instruction formats… Consult the manual *MIPS32 Architecture Volume 2*, linked from the Resources page. It has lots of details on machine language and assembly instruction formats. I found it invaluable, especially in some cases where an instruction doesn't quite fit the simple description of MIPS assembly conventions in the course notes (e.g., `sll` and `syscall`).

Feel free to make reasonable assumptions about limits on things like the number of variables, number of labels, number of assembly statements, etc. It's not good to guess too low about these things, but making sensible guesses let you avoid (some) dynamic allocations.

Write lots of "utility" functions because they simplify things tremendously; e.g., string trimmers, mappers, etc.

Data structures play a role because there's a substantial amount of information that must be collected, represented and organized. However, I used nothing fancier than arrays.

Data types, like the structure shown above, play a major role in a good solution. I wrote a significant number of them.

Explore `string.h` carefully. Useful functions include `strncpy()`, `strncmp()`, `memcpy()` and `strtok()`. There are lots of useful functions in the C Standard Library, not just in `string.h`. One key to becoming proficient and productive in C, as in most programming languages, is to take full advantage of the library that comes with that language.

When testing, you should create some small input files. That makes it easy to isolate the various things your assembler must deal with. Note that the assembler is not doing any validation of the logic of the assembly code, so you don't have to worry about producing assembly test code that will actually do anything sensible. For example, you might use a short sequence of R-type instructions:

```
.text
        add   $t0, $t1, $t2
        sub   $t3, $t1, $t0
        xor   $s7, $t4, $v0
```

Milestones

In order to assess your progress, there will be two milestones for the project. Each of these will require that you submit a partial solution that achieves specified functionality. Each milestone will be evaluated by using a scripted testing environment, which will be posted on the course website at least two weeks before the corresponding milestone is due.

Your score on each milestone will constitute 10% of your final score on the project.

Milestone 1

The first milestone will be due approximately three weeks before the final project deadline. Your submission must support translation of a MIPS assembly program that consists of a `.text` segment including the following instructions:

```
add   rd, rs, rt            # signed addition of integers
                            # GPR[rd] <-- GPR[rs] + GPR[rt]

sub   rd, rs, rt            # signed subtraction of integers
                            # GPR[rd] <-- GPR[rs] - GPR[rt]

addi  rt, rs, imm16         # signed addition with 16-bit immediate
                            # GPR[rt] <-- GPR[rs] + imm16

syscall                     # invoke exception handler, which examines $v0
                            # to determine appropriate action; if it returns,
                            # returns to the succeeding instruction; see the
                            # MIPS32 Instruction Reference for format
```

Your milestone submission must support references to the `$s*` and `$v0` registers. The test files for this milestone will not include a `.data` segment, and there be no symbolic labels in the `.text` segment.

Milestone 2

The second milestone will be due approximately one and a half to two weeks before the final submission. Your submission for this milestone must support translation of a MIPS assembly program that includes both a `.data` and a `.text` segment. The `.data` segment may include:

.word This is used to hold one or more 32 bit quantities, initialized with given values. For example:

```
var1:   .word 15     # creates one 32 bit integer called var1 and initializes
                     # it to 15.

array2: .word 2:10   # creates an array of 10 32 bit integers and initializes
                     # element of the array to the value 2.
```

The `.text` segment may include any of the instructions from the first milestone, and also:

```
lw    rt, offset(rs)        # Transfers a word from memory to a register.
                            # GPR[rt] <-- Mem[GPR[rs] + offset]

la    rd, label             # load address label to register
                            # GPR[rd] <-- label
                            # pseudo-translation for 16-bit label:
                            #     addi  rd, $zero, label

nor   rd, rs, rt            # bitwise logical NOR
                            # GPR[rd] <-- !(GPR[rs] OR GPR[rt])
```

```
beq   rs, rt, offset         # conditional branch if rs == rt
                             # PC <-- (rs == rt ? PC + 4 + offset <<1 2)
                             #                  : PC + 4)

bne   rs, rt, offset         # conditional branch if rs != rt
                             # PC <-- (rs != rt ? PC + 4 + offset <<1 2)
                             #                  : PC + 4)
```

Your milestone submission must support the `$s*`, `$v0` and `$zero` registers, and symbolic labels in both the `.text` and `.data` segments.

NO LATE SUBMISSIONS WILL BE ACCEPTED FOR EITHER MILESTONE!

Extra Credit

For 10% extra credit, implement your assembler so that it will handle a MIPS32 assembly program with a single data segment and a single text segment, in either order. The feature must operate automatically, without any extra command-line switches or recompilation. This will be evaluated by performing a single test, and there will be no partial credit for the feature.

What should I turn in, and how?

For both the milestone and the final submission, create an uncompressed tar file containing:

- All the `.c` and `.h` files which are necessary in order to build your assembler.
- A GNU makefile named "`makefile`". The command "`make assembler`" should build an executable named "`assemble`". The makefile may include additional targets as you see fit.
- A `readme.txt` file if there's anything you want to tell us regarding your implementation. For example, if there are certain things that would cause your assembler to fail (e.g., it doesn't handle **`la`** instructions), telling us that may result in a more satisfactory evaluation of your assembler.
- A `pledge.txt` file containing the pledge statement from the course website.
- Nothing else. Do not include object files or an executable. We will compile your source code.

Submit this tar file to the Curator, by the deadline specified on the course website. Late submissions of the final project will be penalized at a rate of 10% per day until the final submission deadline.

Grading

The evaluation of your solution will be based entirely on its ability to correctly translate programs using the specified MIPS32 assembly subset to MIPS32 machine code. That is somewhat unfortunate, since there are many other issues we would like to consider, such as the quality of your design, your internal documentation, and so forth. However, we do not have sufficient staff to consider those things fairly, and therefore we will not consider them at all.

At least two weeks before the due date for each milestone, we will release a tar file containing a testing harness (test shell scripts and test cases). You can use this tar file to evaluate your milestone submission, in advance, in precisely the same way we will evaluate it. We are posting the test harness as an aid in your testing, but also so that you can verify that you are packaging your submission according to the requirements given above. Submissions that do not meet the requirements typically receive extremely low scores.

The testing of your final assembler submission will simply add test cases to cover the full range of specified MIPS32 instructions and data declarations, and to evaluate any extra-credit features that may be specified for this assignment. We will release an updated test harness for the final submissions at least two weeks before the final deadline.

Our testing of your assembler and milestone submissions will be performed using the test harness files we will make available to you. We expect you to use each test harness to validate your solution. We will not offer any accommodations for submissions that do not work properly with the corresponding supplied test harness.

Test Environment

Your assembler will be tested on the rlogin cluster, running 64-bit CentOS and gcc 4.4.7. There are many of you, and few of us. Therefore, we will not test your assembler on any other environment. So, be sure that you compile and test it there before you submit it. Be warned in particular, if you use OS-X, that the version of gcc available there has been modified for Apple-specific reasons, and that past students have encountered significant differences between that version and the one running on Linux systems.

Maximizing Your Results

Ideally you will produce a fully complete and correct solution. If not, there are some things you can do that are likely to improve your score:

- Make sure your assembler submission works properly with the posted test harness (described above). If it does not, we will almost certainly not be able to evaluate your submission and you are likely to receive a score of 0.
- Make sure your assembler does not crash on any valid input, even if it cannot produce the correct results. If you ensure that your assembler processes all the posted test files, it is extremely unlikely it will encounter anything in our test data that would cause it to crash. On the other hand, if your assembler does crash on any of the posted test files, it will certainly do so during our testing. We will not invest time or effort in diagnosing the cause of such a crash during our testing. It's your responsibility to make sure we don't encounter such crashes.
- If there is a MIPS32 instruction or data declaration that your solution cannot handle, document that in the `readme.txt` file you will include in your submission.
- If there is a MIPS32 instruction or data declaration that your solution cannot handle, make sure that it still produces the correct number of lines of output, since we will automate much of the checking we do. In particular, if your assembler encounters a MIPS32 instruction it cannot handle, write a sequence of 32 asterisk characters (`'*'`) in place of the correct machine representation (or multiple lines for some pseudo-instructions). Doing this will not give you credit for correctly translating that instruction, but this will make it more likely that we correctly evaluate the following parts of your translation.

Credits

The original formulation of this project was by Dr Srinidhi Vadarajan, who was then a member of the Dept of Computer Science at Virginia Tech. His sources of inspiration for this project are lost in the mists of time.

The current modification was produced by William D McQuain, as a member of the Dept of Computer Science at Virginia Tech. Any errors, ambiguities and omissions should be attributed to him.

Change Log

No changes at this time.