Project Tiny emu

Learning Outcomes

- Ability to decipher Tiny emu ISA from assembly language mnemonics to the bit layout of the machine instructions.
- Ability to implement a large C program that consists of multiple .c and .h files and that emulates Tiny emu ISA.
- Ability to integrate your code into an existing code base.
- Ability to use a makefile to build your C program.
- Ability to perform bit manipulations using C bitwise operators.
- Ability to solve problems and write programs in Tiny emu ISA assembly and machine code.

Introduction

For this project, you write a computer emulator similar to `emu`; however, the ISA is simplified.

Keep a log of when you work on the project. The following are sample log entries.
10/15 9AM to 10AM - studied the Project Tiny emu specification
10/18 10PM to midnight - built the provided Project Tiny emu code
10/19 3PM to 6 - began designing implementing cpu.c

Tiny emu General Registers

Tiny emu has 16 general purpose registers, numbers `r0` through `r15`. Each register is 32-bits. Several registers have special meanings.

- `r13` - Stack Pointer (`SP`) - The `SP` is used for stacks on function calls.
- `r14` - Link Register (`LR`) - The `LR` is used by the `bl` instruction.
- `r15` - Program Counter (`PC`). - The `PC` is used to implement the fetch, decode, execute cycle.

Tiny emu CPSR

Tiny emu has one 32-bit `CPSR`, which has 4 flags.

- `N` - bit 31 - Negative flag which is set when the operation is negative; otherwise cleared
- `Z` - bit 30 - Zero flag, which is set when the operation is zero; otherwise cleared
- `C` - bit 29 - Carry flag, which is set when the result of an unsigned operation creates an overflow; otherwise cleared. For example, when adding two 32-bit quantities creates an

answer that does not fit into 32 bits, the `C` flag is set. This bit can be used to implement 64-bit unsigned arithmetic with 32-bit registers. For Tiny emu, the `C` flag will always be 0.

- `V` - bit 28 - Overflow flag, which is set when signed arithmetic causes an overflow; otherwise cleared. If addition and a Positive+Positive yields Negative or Negative+Negative yields a Positive, the `V` flag is set. If subtraction and a Negative-Positive yields a Positive or Positive-Negative yields a Negative, the `V` flag is set. For Tiny emu, the `V` flag will always be 0.

Tiny emu ISA

The Tiny emu Instruction Set Architecture (ISA) consists of load/store instructions, move instructions, compare instructions, and branch instructions. Instructions are 32-bits and manipulate 32-bit quantities. Load and store instructions move 32-bits from/to memory. Arithmetic instructions manipulate 32-bit registers. Compare instructions compare 32-bit quantities.

Load and Store Instructions

- `ldr rd, address` - loads `rd` with four bytes of memory from `address+0`, `address+1`, `address+2`, and `address+3`.
- `ldx rd, rn, #offset` - loads `rd` with four bytes of memory. The starting `address` is the value in `rn` plus `offset`. If `rn` has `400` and offset is `10`, contents memory locations `410`, `411`, `412`, and `413` are loaded into `rd`
- `str rd, address` - stores content of `rd` in `address+0`, `address+1`, `address+2`, and `address+3`.
- `stx rd, rn, #offset` - stores content of `rd` into four bytes of memory. The starting `address` is the value in `rn` plus `offset`. If `rn` has `400` and offset is `10`, contents of `rd` are stored in memory locations `410`, `411`, `412`, and `413`.

Move Instructions

- `mov rd, rn` - moves the contents of `rn` to `rd`
- `mov rd, #num` - moves `num` to `rd`

Arithmetic and Logic Instructions

Arithmetic Instructions of `add`, `sub`, `mul`, `div`, `and`, `orr`, and `eor`. The `and`, `orr`, and `eor` instructions are bitwise instructions - equivalent to the C operators `&`, `|`, and `^`. Each instruction has the same format.

- `opcode rd, rm, rn` - applies `opcode` to `rm` and `rn` placing the result in `rd`. `rm` `opcode` `rn` is placed in `rd`. For example the instruction `add r3, r4, r5` results in the contents of `r4` minus the contents of `r5` being placed into `r3`.

Compare Instructions

Compare instructions set flags in the `CPSR`.

- `cmp rd, rn` - subtracts `rn` from `rd` (discarding the result) setting condition codes in `CPSR` as described in section Tiny emu CPSR.
- `cmp rd, #num` - subtracts `num` from `rd` (discarding the result) setting condition codes in `CPSR` as described in section Tiny emu CPSR.

Branch Instructions

The unconditional branch instruction (`b`) sets the `PC` to the target address. Conditional branch instructions examine the flags in the `CPSR`.

- `b address` - set `PC` to `address`
- `beq address` - set `PC` to `address` is `Z == 1`
- `bne address` - set `PC` to `address` if `Z == 0`
- `ble address` - set `PC` to `address` if `(Z == 1) or (N != V)`
- `blt address` - set `PC` to `address` if `N != V`
- `bge address` - set `PC` to `address` if `N == V`
- `bgt address` - set `PC` to `address` if `(Z == 0) and (N == V)`

Tiny emu Instruction Opcodes

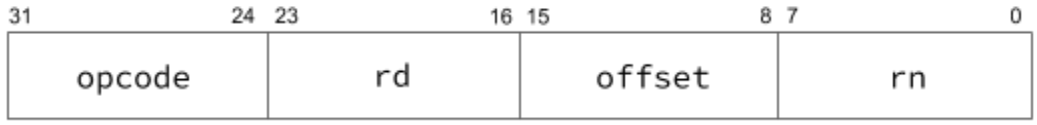
- `ldr - 0x11`
- `str - 0x12`
- `ldx - 0x13`
- `stx - 0x14`
- `mov - 0x21`
- `add - 0x31`
- `sub - 0x32`
- `mul - 0x33`
- `div - 0x34`
- `and - 0x35`
- `orr - 0x36`
- `eor - 0x37`
- `cmp - 0x41`
- `b - 0x51`

Tiny emu Instruction Format

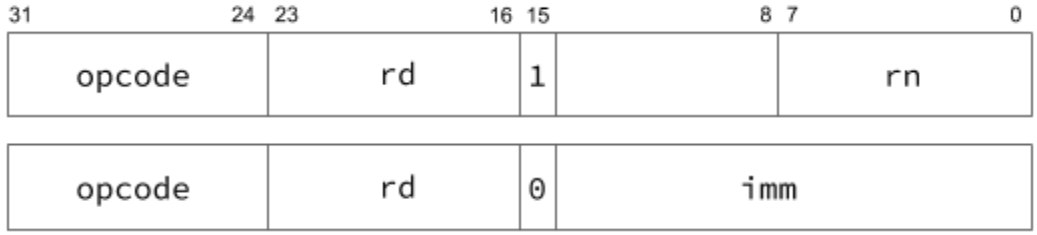
Each instruction is 32-bits. All register fields (`rd`, `rn`, `rm`) must be less than `16`. All `address` fields must be less than `1024`.

The `ldr` and `str` instructions have the following format, where the opcode is `0x11` for `ldr` and `0x12` for `str`.

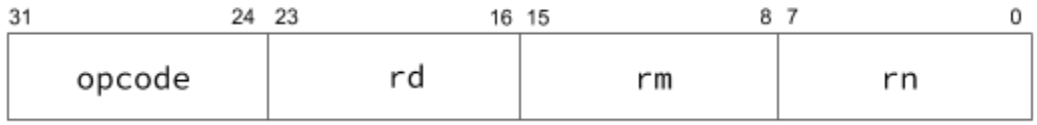
The `ldx` and `stx` instruction has the following format, where the opcode is `0x13` for `ldx` and `0x14` for `stx`.



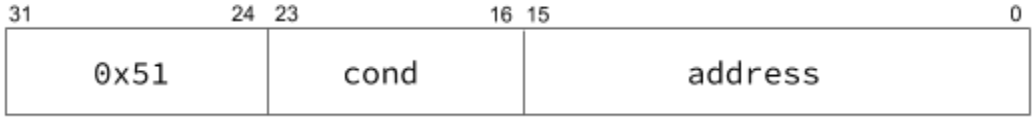
The `mov` and `cmp` instructions have the following format, where the opcode is `0x21` for `mov` and `0x41` for `cmp`.



The `add`, `sub`, `mul`, `div`, `and`, `orr`, and `eor` instructions have the following format, where the opcode is `0x31` for `add`, `0x32` for `sub`, `0x33` for `mul`, `0x34` for `div`, `0x35` for `and`, `0x36` for `orr`, and `0x37` for `eor`.



The `b`, `beq`, `bne`, `ble`, `blt`, `bge`, `bgt`, and `bl` instructions have the following format.



The `cond` field in the branch instruction determines if it is unconditional, conditional, or `bl`. It is one of the following.

- `al` - `0` - This is an unconditional branch - branch always, `b address`
- `eq` - `1` - This is `beq address`

- `ne - 2` - This is `bne address`
- `le - 3` - This is `ble address`
- `lt - 4` - This is `blt address`
- `ge - 5` - This is `bge address`
- `gt - 6` - This is `bgt address`
- `bl - 0x80` - This is `bl address`

Tiny emu Sample Code

Figure 1 shows Tiny emu sample code in three columns - left, middle, and right. The left column shows assembly code that can be assembled by the Tiny emu assembler. The right side column shows hex values corresponding to the assembly code. The middle column shows the addresses the code resides in memory.

```
# data address 100
.data 100
0xff                    0x100   0x000000ff
.label x
10                      0x104   0x00000010
.label feed
feedabee                0x108   0xfeedabee
# code following data
.text
mov r5, r7              0x10c   0x21058007
ldr r0  x               0x110   0x11000104
mov r1, #10             0x114   0x21010010
cmp r0, r1              0x118   0x41008001
bne else                0x11c   0x51020134
mov r0, #64             0x120   0x21000064
mov R3, #3              0x124   0x21030003
add r0, r0, r3          0x128   0x31000003
str r0, x               0x12c   0x12000104
b end                   0x130   0x5100013c
.label else
ldr r5, feed            0x134   0x11050108
str r5, x               0x138   0x12050104
.label end
b end # loop on self    0x13c   0x5100013c
```

Figure 1. Tiny emu Sample Code

Tiny emu Emulator Structure

The Tiny emu Emulator consists of several C modules. Each module consists of a `.h` and a `.c` file. You are provided code for the bolded modules.

- **`bit_functions`** - functions and macros useful for manipulating bits of int variables. For example to set bit 31 and to test bit 31. The entire `bit_functions` module is provided.
    - `int bit_find(int value)` - returns the bit position of the first bit set in `value`. The search begins with bit 0.
    - `int bit_test(int value, int bit_pos)` - returns 1 if `bit_pos` in `value` is 1 else 0.
    - `void bit_set(int *value, int bit_pos)` - sets `bit_pos` in `value` to 1.
    - `void bit_clear(int *value, int bit_pos)` - sets `bit_pos` in `value` to 0.
- **`memory`** - implements a low-level memory chip. Simulates a matrix of 32 by 32 bytes, which results in a 1024 byte memory - addresses 0 through 1023. This is a 10-bit address space with addresses `0x000` through `0x3ff`. The `memory` module owns the memory that is used by the program. The entire `memory` module is provided
- **`memory_system`** - implements a memory system that the cpu accesses with several functions. `memory_system` uses the low-level `memory` module. `memory_system` has a primary function `system_bus` and two utility functions - `memory_dump` and `memory_load`. The entire `memory_system` module is provided.
    - `system_bus(int address, int *value, int control)` - used to `READ` and `WRITE` 32-bit `value` from/to memory `address`. For example to write the value `0x31010203` to address `0x100`.
      ```
      int val = 0x31010203;
      system_bus(0x100, &val, WRITE);
      ```
    - `void dump_memory(int start_address, int num_bytes)` - displays memory bytes to the terminal. For example, `dump_memory(0x100, 16)` displays the following.
      ```
      0x0100 (0d0256) 0x00 0x00 0x00 0xff 0x00 0x00 0x00 0x10
      0x0108 (0d0264) 0xfe 0xed 0xab 0xee 0x21 0x05 0x80 0x07
      ```
    - `void load_memory(char *filename)` - loads a file in memory, where `filename` is the name of the file. The section Tiny emu Load File Format describes the format of the file.

- **`isa`** - Defines a data type struct `decoded` and implements several functions that manipulate decoded. The `isa` module manipulates the Tiny emu ISA. A skeleton `isa` module is provided. The `disassemble` function is complete. You have to implement `decode`.
    - `decoded *decode(unsigned int inst)` - decodes the Tiny emu instruction in inst and returns a `decoded struct` with values. `decode` `malloc`s memory for the `decoded struct`.
    - **`char *disassemble(decoded *d)`** - disassembles the Tiny emu instruction in `d` and returns a `char *` containing the assembly corresponding to `d`.

      The statement `printf("%s\n", disassemble(decode(0x31010203)));`

prints `add r1, r2, r3`

- `cpu` - performs the fetch, decode, execute cycle for Tiny emu ISA. `cpu` uses the `system_bus` function to access memory for fetching instructions, loading registers with `ldr` and `ldx`, and storing registers with `str` and `stx`. A skeleton cpu module is provided. You have to implement the functions.
    - `void set_reg(int reg, int value)` - sets reg to value.
    - `int get_reg(int reg)` - returns the value of reg.
    - `int get_cpsr()` - returns the value of the CPSR.
    - `void show_regs()` - displays all registers to the terminal.
    - `void step()` - perform one fetch, decode, execute cycle of the instruction pointed to by the PC. The PC is updated. If the instruction executed is not a branch that branches, the PC is incremented by 4. If the instruction executed is a branch instruction, the PC is updated with the destination address on the branch.
    - `void step_n(int n)` - calls `step` `n` times.
    - `void step_show_reg()` - like `step`, but it shows any registers changed by the instruction.
    - `void step_show_reg_mem()` - like `step`, but it shows any registers and any memory locations changed by the instruction.

Tiny emu Load File Format

The `load_memory(char *filename)` - described in the previous section in the bulleted item for memory_system module loads a file in memory, where `filename` is the name of the file. Figure 2 shows the format of the input file. The file in Figure 2 corresponds to the Tiny emu Sample code shown in Figure 1. Each line of the file is a hex number. The numbers can include or exclude the `0x` prefix. Regardless of the prefix the number is hex. The first line of the file contains the start address. The remaining lines are 32-bit values that are loaded into memory. In Figure 2, `0xff` is loaded to memory address `0x100`, `10` to address `0x104`, feedabee to address `0x108`, `0x21058007` to address `0x10c`, and so on.

```
100
0xff
10
feedabee
0x21058007
0x11000104
0x21010010
0x41008001
0x51020134
0x21000064
0x21030003
0x31000003
0x12000104
0x5100013c
```

```
0x11050108
0x12050104
0x5100013c
```

Figure 2. load_memory Formatted File

cpu Module Guidance

The following are attributes that I used in my implementation.

- I have two variables defined outside of the functions
  - `static int registers[16]` - This array represents the 16 registers.
  - `static int cpsr` - This variable represents the current program status register. You can use `bit_functions` to manipulate the flags.
- The `step()` function performs one iteration of the **fetch-execute-decode** cycle. `step()` has the following attributes.
  - `step()` **fetches** the instruction stored at the address pointed to by `registers[PC]`.
  - `step()` **decodes** the instruction by calling `decode`, which you must implement.
  - `step()` **executes** the instruction. Use the values returned from `decode`. The following is a small snippet of my code.

```
switch (d->opcode) {
 case LDR:
  system_bus(d->address, &registers[d->rd], READ);
```

main Module (`main.c`)

I provide you with a `main.c` to get you started. You update `main.c` as part of the project. The following shows a sample session of my running my version of main.c, which has cpu module complete. Bolded text are commands entered. The program loaded, `proga.tisa`, corresponds to the code shown in Figure 1.

```
% ./tinyemu
Enter cmd: load
Enter file name: proga.tisa
Enter cmd: dump
Enter address: 100
Enter num bytes: 100
start_boundary: 256 0x0100
0x0100 (0d0256) 0x00 0x00 0x00 0xff 0x00 0x00 0x00 0x10
0x0108 (0d0264) 0xfe 0xed 0xab 0xee 0x21 0x05 0x80 0x07
0x0110 (0d0272) 0x11 0x00 0x01 0x04 0x21 0x01 0x00 0x10
0x0118 (0d0280) 0x41 0x00 0x80 0x01 0x51 0x02 0x01 0x34
0x0120 (0d0288) 0x21 0x00 0x00 0x64 0x21 0x03 0x00 0x03
```

```
0x0128 (0d0296) 0x31 0x00 0x00 0x03 0x12 0x00 0x01 0x04
0x0130 (0d0304) 0x51 0x00 0x01 0x3c 0x11 0x05 0x01 0x08
0x0138 (0d0312) 0x12 0x05 0x01 0x04 0x51 0x00 0x01 0x3c
0x0140 (0d0320) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0148 (0d0328) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0150 (0d0336) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0158 (0d0344) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0160 (0d0352) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
Enter cmd: set_reg
Enter reg: 15
Ender reg value: 268
Enter cmd: regs
R0:  0x00000000, R1:  0x00000000, R2:  0x00000000, R3:  0x00000000
R4:  0x00000000, R5:  0x00000000, R6:  0x00000000, R7:  0x00000000
R8:  0x00000000, R9:  0x00000000, R10: 0x00000000, R11: 0x00000000
R12: 0x00000000, R13: 0x00000000, R14: 0x00000000, R15: 0x0000010c
Enter cmd: step
PC: 0x0000010c, inst: 0x21058007, mov r5, r7
CPSR: 0x00
Enter cmd: step_s
PC: 0x00000110, inst: 0x11000104, ldr r0, 0x104
CPSR: 0x00
reg[0]: before: 0x00000000, after: 0x00000010
reg[15]: before: 0x00000110, after: 0x00000114
Enter cmd: step_n
Enter num of steps: 3
PC: 0x00000114, inst: 0x21010010, mov r1, #0x10
CPSR: 0x00
PC: 0x00000118, inst: 0x41008001, cmp r0, r1
CPSR: 0x40000000
PC: 0x0000011c, inst: 0x51020134, bne 0x134
CPSR: 0x40000000
Enter cmd: regs
R0:  0x00000010, R1:  0x00000010, R2:  0x00000000, R3:  0x00000000
R4:  0x00000000, R5:  0x00000000, R6:  0x00000000, R7:  0x00000000
R8:  0x00000000, R9:  0x00000000, R10: 0x00000000, R11: 0x00000000
R12: 0x00000000, R13: 0x00000000, R14: 0x00000000, R15: 0x00000120
Enter cmd: translate
Enter inst: 0x51020134
num: 51020134
bne 0x134
Enter cmd: step_s
PC: 0x00000120, inst: 0x21000064, mov r0, #0x64
CPSR: 0x40000000
reg[0]: before: 0x00000010, after: 0x00000064
reg[15]: before: 0x00000120, after: 0x00000124
Enter cmd: step_s
PC: 0x00000124, inst: 0x21030003, mov r3, #0x3
```

```
CPSR: 0x40000000
reg[3]: before: 0x00000000, after: 0x00000003
reg[15]: before: 0x00000124, after: 0x00000128
Enter cmd: step_s
PC: 0x00000128, inst: 0x31000003, add r0, r0, r3
CPSR: 0x40000000
reg[0]: before: 0x00000064, after: 0x00000067
reg[15]: before: 0x00000128, after: 0x0000012c
Enter cmd: step_m
PC: 0x0000012c, inst: 0x12000104, str r0, 0x104
CPSR: 0x40000000
reg[15]: before: 0x0000012c, after: 0x00000130
addr: 0x0104, before: 0x00000010, after: 0x00000067
Enter cmd: step
PC: 0x00000130, inst: 0x5100013c, b 0x13c
CPSR: 0x40000000
Enter cmd: step
PC: 0x0000013c, inst: 0x5100013c, b 0x13c
CPSR: 0x40000000
Enter cmd: step
PC: 0x0000013c, inst: 0x5100013c, b 0x13c
CPSR: 0x40000000
Enter cmd: dump
Enter address: 100
Enter num bytes: 100
start_boundary: 256 0x0100
0x0100 (0d0256) 0x00 0x00 0x00 0xff 0x00 0x00 0x00 0x67
0x0108 (0d0264) 0xfe 0xed 0xab 0xee 0x21 0x05 0x80 0x07
0x0110 (0d0272) 0x11 0x00 0x01 0x04 0x21 0x01 0x00 0x10
0x0118 (0d0280) 0x41 0x00 0x80 0x01 0x51 0x02 0x01 0x34
0x0120 (0d0288) 0x21 0x00 0x00 0x64 0x21 0x03 0x00 0x03
0x0128 (0d0296) 0x31 0x00 0x00 0x03 0x12 0x00 0x01 0x04
0x0130 (0d0304) 0x51 0x00 0x01 0x3c 0x11 0x05 0x01 0x08
0x0138 (0d0312) 0x12 0x05 0x01 0x04 0x51 0x00 0x01 0x3c
0x0140 (0d0320) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0148 (0d0328) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0150 (0d0336) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0158 (0d0344) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x0160 (0d0352) 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
Enter cmd: q
%
```

Tiny emu Assembler (teasm) User Guide

The Tiny emu Assembler (`teasm`) translates Tiny emu Assembly Code into a text file of hex numbers, which can be loaded into a Tiny Computer simulator using the `load_memory` function

of the `memory_system` module. `easm` is a Python 3 script that has been tested on several programs. Thus it may not process incorrect Tiny emu assembly code, and it may have bugs. If easm does not work for you, just update the Python code. An alternative is to write Tiny emu Assembler in a text file with hex numbers.

Sample Use

The following shows a typical use of `teasm`. The first line of the Python code is a shebang, `#!/usr/bin/python3`, which allows it to be run like an executable provided you have set the execute permission of the file.

```
% python3 teasm.py prog1.txt
% ./teasm.py prog1.txt
```

The output file is `prog1.tisa`, where `tisa` is Tiny emu ISA. See Memory Simulation and Computer Simulation programming assignments for the format of a `.tisa` file.

Tiny assembly code consists of data and instructions. You can use upper and lowercase; however, `teasm` converts the input stream to lowercase before processing. This means labels `Gusty` and `gusty` are identical. Data are placed after a `.data` directive and instructions are placed after a `.text` directive. Labels are created with a `.label` directive. `teasm` trims leading and trailing whitespace, converts characters to lowercase, replaces commas with spaces, and splits an instruction with general whitespace (allows multiple spaces between opcodes and operands). All numbers in `teasm` are hex. The numbers can include or exclude the `0x` prefix. Regardless of the prefix the number is hex.

The left column of Figure 1 is sample `teasm` assembly code. The right column of Figure 1 is sample output from `teasm`. The output file can be loaded into Tiny emu emulator..

teasm Directives

`teasm` directives begin with a dot `.` in position 0 of a line. There are three directives.

`.data <address>`

The directive `.data` begins a data section. The optional address specifies the address placed on the first line of the output file. A `.data` directive without an address continues counting addresses from the previous section. Data lines follow `.data` directives. Figure 3 starts a data section at address `0x100`. The two variables `x` and `y` are located at addresses `0x100` and `0x104`, and their initial values are `1` and `2`. Values on data lines are replicated in the output file. Since `load_memory` function reads lines as hex, you place hex numbers on data lines. A data line with `gusty` is replicated as `gusty` in the output file.

```
.data 100
.label x
1
.label y
2
```
Figure 3. Sample .data section

`.label label_name`

The directive `.label label_name` associates the `label_name` with the address of the next data or instruction. You can use labels as the operand on the following instructions on branch, `ldr`, and `str` instruction.

- Branch instructions can branch to a label or a hex number. Place a `.label` directive in front of an instruction, and use a branch instruction to branch to that label. You can also branch to a hex number. Figure 4 shows if-then-else with labels, along with its equivalent C code.

```
int a = 0x45, b = 0x60;
if (a == b)
   a += 3;
else
   a += 2;
.data 100
.label a
45
.label b
60
.text
ldr r0, a
ldr r5, b
cmp r0, r5
blt else
mov r2, #3
b endif
.label else
mov r2, #2
.label endif
add r0, r0, r2
str r0, b
```
Figure 4. .label Directive and Branches

- ldr and str instructions. ldr and str instructions can load/store a label or a hex number as the address from which to load or to which to store.
    - `ldr r0, gusty` - loads `r0` with the contents of the memory address with label `gusty`

- `ldr r0, 300` - loads `r0` with the contents of memory address 300 hex

`.text address`

A .text directive indicates that the following lines contain Tiny Assembly instructions. Each Tiny Assembly instruction is 32-bits. The instructions must conform to the syntax samples shown in Figure 5. Note the machine instruction for `mov r7, #100` is valid since the bit layout has a 15-bit immediate field; however, `teasm` does not translate this correctly. `teasm` requires the `#` number to be less than or equal to two digits.

```
ldr r10, 111                    str r10, 111
ldr r11, #111                   str r11, #111
ldr r12, x                      str r12, x
ldx r10, r5, #10                stx r10, r5, #10
ldx r11, r5, 10                 stx r11, r5, 10
mov r7, r8                      cmp r10, r11
mov r7 #10                      cmp r10, #10
mov r7, 10                      cmp r10, 10
b end                           bl end
beq end                         bne end
ble end                         blt end
bge end                         bgt end
add r10, r11, r12               sub r10, r11, r12
mul r10, r11, r12               div r10, r11, r12
and r10, r11, r12               orr r10, r11, r12
eor r10, r11, r12
```

Figure 5. teasm Assembly Instruction Format

Project Tiny emu Work That You Do

1. Build and run the code provided. The code runs but does not emulate Tine emu ISA.
2. In `isa.c`, design and implement the `decode` function. `decode` shall `malloc` memory for the returned value, examine the bits of an instruction, and fill in the values of the `decoded struct`.
3. In `cpu.c`, design and implement the `set_reg`, `get_reg`, `get_cpsr`, `show_regs`, `step`, `step_n`, `step_show_reg`, and `step_show_reg_mem`.
    a. `set_reg` shall set a register to a value.
    b. `get_reg` shall return the value of a register.
    c. `get_cpsr` shall return the value of the `CPSR`.
    d. `show_regs` shall display the registers on the terminal.
    e. `step` shall execute a fetch, decode, execute cycle for the instruction pointed to by register 15. `step` shall use the `decode` function in `isa`. `step` shall `exit(-1)` if it fetches an invalid instruction. An instruction is invalid if it has an invalid opcode, it

has a register greater than 15, or has an address that is greater than 1023. `step` shall display on the terminal the `PC`, the hex of the instruction, and the disassembled instruction each time it is called.

```
PC: 0x0000010c, inst: 0x21058007, mov r5, r7
CPSR: 0x0000000
```

f. `step_show_reg` shall display `step`'s display plus it shall display the values of registers changed by the instruction.

```
PC: 0x00000128, inst: 0x31000003, add r0, r0, r3
CPSR: 0x40000000
reg[0]: before: 0x00000064, after: 0x00000067
reg[15]: before: 0x00000128, after: 0x0000012c
```

g. `step_show_reg_mem` shall display `step_show_reg`'s display plus it shall display the values of memory locations changed by the instruction.

```
PC: 0x0000012c, inst: 0x12000104, str r0, 0x104
CPSR: 0x40000000
reg[15]: before: 0x0000012c, after: 0x00000130
addr: 0x0104, before: 0x00000010, after: 0x00000067
```

4. Write Tiny emu ISA assembly programs equivalent to the C programs shown in section Tiny emu ISA Assembly Programs. You can do this with the Tiny emu Assembler just write the code in hex.
5. Test you Tiny emu ISA assembly programs with your Tiny emu ISA emulator to demonstrate your implementation is correct.

Project Submission

6. Submit your log file of when you worked on Tiny emu, your source code that implement Tiny emu ISA emulator (all .c and .h, makefile), your Tiny emu ISA assembly code for Programs 1 through 5, and logs of you testing your Tiny emu ISA assembly code.

Tiny emu ISA Assembly Programs

Program 1 - if-else Statement

Be sure to change the values of a and b such that a < b, a == b, and a > b.

```
int a = 10, b = 20, c = 0;
if (a < b)
   c = a;
else
   c = b;
```

Program 2 - Summation from 1 to n

Be sure to change the value of n to test various summations.

```
int sum = 0, n = 10;
```

```
for (int i = 1; i <= n; i ++)
    sum += i;
```

Program 3 - switch Statement

Be sure to change the value of a to force execution of each case of the switch statement.

```
int a = 1; b = 2; c = 3;
switch (a) {
  case 1:
    a = b + c;
    break;
  case 2:
    a = b - c;
    break;
  case 3:
    a = b * c;
    break;
  case 4:
    a = b / c;
    break;
  case 5:
    a = b & c;
    break;
  case 6:
    a = b | c;
    break;
  case 7:
    a = b ^ c;
    break;
  default:
    a = 0;
}
```

Program 4 - Function Calls - passing arguments, returning value

```
int a[] = {1, 3, 50, 6, 3, 3, 6, 8, 9, 10, 0};
int x = count(a, 10, 3);
x = largest(a, 10);
x = size(a);
// counts occurrences of v in a
int count(int *a, int s, int v) {
  int c = 0;
  int i = 0;
  while (i < s) {
    if (*a == v)
      c++;
    a++;
    i++;
```

```
  }
  return c;
}
// finds largest element in a
int largest(int *a, int s) {
  int l = *a;
  for (int *p = a; p < a + s; p++)
    if (*p > l)
      l = *p;
  return l;
}

// treats a as a null-terminated array of int, returns size of a
int size(a) {
   for (int i = 0; a[i] != 0; i++);
   return i;
}
```

Program 5 - Invalid Instructions

First identify each of the following instructions. The 100 is a load address. The remaining hex values are almost valid instructions, but they have invalid registers or addresses. Create a load file with the following values. Load the file. Set the PC to 100, 104, 108, 10c, 110 and demonstrate that your program indicates these are invalid instructions.

```
100
0x120A2111
0x310A0B10
0x5101313c
0x21078012
0x4111800B
```