T34 Emulator CSC317 Lucas Carpenter

Description

T34Emulator is a program that creates a memory array of 4096 (2^{12}) words where each word contains 24 bits. Memory can be read from and written to using the putWord and getWord functions, a memory dump can be executed using the memdump function and you can parse words of memory into their instructions using the parse function. The program also must read in an object file that contains memory addresses and their data to store, the last item in the object file will be the address the program counter will start at, this value is stored in a char* as a variable called 'pc'. To perform a memory dump, the function memdump(byte*) can be used, or to parse a string the function parse(byte*, std::string) can be used.

Once an object file has been loaded into memory an Emulator class will instantiate and pass control of the allocated memory array to the class and run on a fetch, decode, execute, output loop till the emulation is halted by an illegal/unimplemented addressing mode, an undefined/unimplemented opcode, a HALT instruction is executed, or the end of memory is reached by the loop. This emulation class runs off the T34 architecture outlined in the program write-up, using a data-path with the following registers:

MAR – Memory Address Register | A register containing the address of memory to access

MDR – Memory Data Register | A register containing the value of memory at MAR

ALU – Arithmetic Logic Unit | A unit that performs arithmetic/logical operations

IC - Instruction Counter | A register containing the current instruction to fetch

IR – Instruction Register | A register containing the values of the instruction to decode

AC – ACcumulator | A register containing the accumulated value

While the framework exists for more registers, these are the only ones currently in use by the emulator.

Compiling & Usage

To compile the program, use g++ as follows:

$ g++ -std=c++11 -o T34Emulator T34Emulator.cpp MemIO.cpp Emulation.cpp

To run the program, use the following line:

$ T34Emulator [objfile.obj]

Where [objfile.obj] is replaced with the file path for your .obj file

Function Definitions/Information

T34Emulator.cpp

main(int, char[])*

- **Description**: Function is the starting point of our program; the program begins and ends here.
- **Inputs**: int - the number of arguments from command line
  char*[] - a 2D array of chars returning input from command line
- **Outputs**: returns 0 on successful program finish, physical output to terminal if invalid number of arguments.
- **Testing Methods**: Nothing really to test, this is where all the testing for other functions working was done.

readObj(char, byte*)*

- **Description**: Function takes in an array of characters representing an object filename to be opened and an array of bytes representing memory. The function reads through the object file and uses a specific object file format to store values into continuous memory.
- **Inputs**: char* - an array of characters representing an obj filename
  byte* - an array of bytes representing memory
- **Outputs**: no data output, physical output only if file can't be opened
- **Testing Methods**: I tested the function would read from the char* filepath by checking to see if the file was opened and read and output to the screen the contents of the file, and by seeing if the function would exit if the file path didn't exist. I tested the byte* memarray by outputting the contents of the array to the screen in the main function to see if everything matched.

MemIO.cpp

putWord(byte, char*, char*, int=16)*

- **Description**: Function takes in an array of bytes acting as memory, an array of characters representing in hex a memory address, another array of characters representing a value in a base to store in the hex memory address, and an int defaulted to 16 to represent the base of the value being put into the hex address is needed to be converted from.
- **Inputs**: byte* - an array of bytes acting as memory
  char* - an array of chars representing a hex memory address
  char* - an array of chars representing a value in an unknown base
  int - an int, defaulted to 16, representing the previous char*'s base
- **Outputs**: no data output, no physical output

- **Testing Methods**: To test this function I would run this function in main and output the contents of memarray afterwards. I changed the base a few times to see if it would always convert properly and it appears to change bases correctly

getWord(byte, char*, char*, int=16)*

- **Description**: Function takes in an array of bytes acting as memory, an array of characters representing in hex a memory address and two ints representing input base and output base both defaulted to 16 and 10 respectively. The code converts the input base memory address to a decimal integer and navigates to the correct offset in the byte array. Then, the current byte and the next two bytes are OR'd together and shifted appropriately to create a 24bit word, this word is converted to the proper output base as a char* and returned as output.
- **Inputs**: byte* - an array of bytes acting as memory
  char* - an array of chars representing a hex memory address
  int - an input base to convert char* to decimal representation correctly
  int - an output base for to convert the output value to
- **Outputs**: char* - a character array in a specific base containing the 24bit value of a memory address
- **Testing Methods**: To test this function I would load in an obj file and pull the value from different memory addresses, if the values all matched what the input file had, then I assumed the function was working.

memdump(byte)*

- **Description**: Function takes in an array of bytes functioning as memory, the code loops through each memory address and outputs any addresses with a value not equal to 0.
- **Inputs**: byte* - an array of bytes acting as memory
- **Outputs**: no data output, physical output to terminal in form…
  [MEMADDR] - HEXVAL(6 hex digits)
- **Testing Methods**: To test this function I would populate memarray by reading in an obj file and then just call the function afterwards and see if the output matched what the obj file had

parse(byte, std::string)*

- **Description**: Function takes in an array of bytes functioning as static memory and a string of hex addresses separated by ','s or ' 's and splits the string into a list of strings, then each address is parsed and the output of each byte in memory is displayed as 3 bitstrings
- **Inputs**: byte* - an array of bytes acting as memory
  std::string - a string of memory addresses to parse

- **Outputs**: no data output, physical output to terminal in form…
  [MEMADDR]: ADDR(12bits) OP(6bits) AM(6bits)
- **Testing Methods**: To test this function I would populate the byte array by reading in an obj file then call the function using "", "0c5", "0c5 0c6", and "0c5,0c6 0c7", this let me see if my parsing of the string would work so you could chain multiple calls in one function call instead of in multiple separate calls and to see if the output matched the binary representation of the values in each memory address

putMemory(byte, std::bitset<12>, std::bitset<24>)*

- **Description**: This function takes in an array of bytes as a static memory, a 12 bit bitset to give a 24-bit word address to the byte memory, and a 24 bit bitset to store into memory. The 24-bit value is broken into 3 bytes and stored individually starting at the word offset, given by the 12 bit bitset, in an array of bytes.
- **Inputs:** byte* - an array of bytes to act as a static memory
  std::bitset<12> - a 12-bit bitset representing a memory address offset
  std::bitset<24> - a 24-bit bitset value to be stored in memory
- **Outputs**: none
- **Testing Methods:** To test, two bitsets were created, one 12-bit and one 24-bit and values were fed into them as address offsets and memory values respectively, a memdump was performed to verify that the memory had a different value after each function call.

putHalfMemory(byte, std::bitset<12>, std::bitset<24>)*

- **Description**: This function takes in an array of bytes as a static memory, a 12 bit bitset to give a 24-bit word address to the byte memory, and a 24 bit bitset to store into memory. The 24-bit value trims off the lower 12 bits and stores only the upper 12 bits into memory.
- **Inputs:** byte* - an array of bytes to act as a static memory
  std::bitset<12> - a 12-bit bitset representing a memory address offset
  std::bitset<24> - a 24-bit bitset value to be stored in memory
- **Outputs**: none
- **Testing Methods:** To test, various memory was created and the function was called, a memdump was performed to verify that the memory had been altered only in the upper 12 bits of each start of word.

getMemory(byte, std::bitset<12>, std::bitset<24>&)*

- **Description**: This function takes in an array of bytes as a static memory, a 12 bit offset to give a 24-bit word address to the byte memory, and a 24 bit bitset to put the contents of memory into. The 24-bit bitset is passed by reference so the contents will be updated. The byte array of static memory is given an offset defined by the 12-bit bitset and then that byte

as well as the next two bytes are extracted and merged into one 24-bit value and stored as the 24-bit bitset

- **Inputs**: byte* - an array of bytes to act as a static memory
  std::bitset<12> - a 12-bit bitset representing a memory address offset
  std::bitset<24> - a 24-bit bitset address to store a value pulled from memory
- **Outputs**: none
- **Testing Methods**: To test, memory was populated with an object file, then the function was called and a 24-bit bitset was stored as a dummy variable and outputted, I'd pull a random address that may or may not have contained a previously set value, then compare the memdump data to the value returned from this function.

Emulation.cpp

Registers::Registers()

- **Description**: A constructor for the Registers struct, instantiates a register object and sets default values for registers.
- **Inputs**: none
- **Outputs:** none
- **Testing Methods**: Nothing to test, if this doesn't work right then there's an issue beyond my control.

Registers::~Registers()

- **Description**: A destructor for the Registers struct, it destroys dynamic data and cleans up any variables from the structure in use.
- **Inputs**: none
- **Outputs**: none
- **Testing Methods**: This function ran when the Emulator class was cleaning up and has the sole job of freeing up the 4 bytes we declared when we the readObj function ran, as long as they exist and control was transferred properly.

Emulator::Emulator(char)*

- **Description:** A constructor for the Emulator class, instantiates a new class object and sets default values for all variables.
- **Inputs**: char* - A 4-character array containing the address to set the IC register to
- **Outputs**: none
- **Testing Methods**: This function was run and GDB was used to verify that the object's register struct contained the correct instruction counter register

Emulator::~Emulator()

- **Description**: A destructor for the Emulator class, it recursively destroys all data and variables that were created as part of the object.
- **Inputs**: none
- **Outputs**: none
- **Testing Methods**: This function was tested by making sure it was calling other destructors while it was running.

Emulator::setMemAddress(byte)*

- **Description:** This function takes in an array of bytes and sets the class memory pointer to the input pointer, if this function is not called before running the emulator then the emulator memory will be a nullptr and will segmentation fault.
- **Inputs:** byte* - a byte array pointer containing a 4,096 24-bit word memory
- **Outputs:** none
- **Testing Methods:** To test this, after the function ran, the memory pointer of the Emulator class was accessed to verify that all the values lined up.

Emulator::halt(int&, int&, std::string, std::string)

- **Description:** This function is used as a helper function to the decoder. If this function is called, the current instruction has been deemed illegal or undoable in some way and the two ints passed by reference will be set to 0 and 1 respectively to force the current instruction to perform a NOP instead, then the address info will be printed out if possible as if nothing wrong ever happened and an error message will be set to be displayed after the instruction finishes.
- **Inputs:** int& - a pointer to the instruction type being executed
  int& - a pointer to the instruction operation being executed
  std::string – addressing output to display after the mnemonic
  std::string – the error message to display after the current instruction executes
- **Outputs:** none
- **Testing Methods:** To test this function, illegal instructions were sent through the decoder and screen output was verified to match for different cases, as well as when the function returned, instruction type and operation were checked and verified to be changed to a NOP instruction

Emulator::IDandEXE()

- **Description**: This function handles the instruction decoding and execution for the emulator. The function begins by decoding the instruction into an instruction type, instruction operation, and addressing mode and then verifies the validity of the instruction by checking a pre-defined table of mnemonics and legal addressing modes. If any mnemonics return a "????" value, or have bit 9 set (indexed opcode), then an unknown/unimplemented operation is called and the halt function sets the instruction to a NOP and outputs the effective address as if nothing had ever happened. If the instruction's addressing mode value doesn't exist in the correct pre-defined list of legal addressing modes or the addressing mode is an indexed or indirect addressing mode then an illegal/unimplemented addressing mode is declared and the halt function sets the instruction to a NOP instruction and outputs a "???" for the effective address. If it makes it past the checks for a legal instruction then the instruction is decoded further to set up the EA (effective address) and load from memory if needed and specific ALU flags determining the conditions, ALU output, operation, and op1 type are set and the ALU is ran. For all instruction except the exchange memory instruction the ALU runs once and then exits the function, for the exchange memory instruction the ALU is set up as a store into memory, ran and then set up again as a load into AC instruction, where it is ran again and then the function is exited.
- **Inputs**: none
- **Outputs**: no value output, physical output to terminal giving a 4-character mnemonic and a 3-character effective address
- **Testing Methods**: This one was difficult to test, but it was broken up into sections. First the decoding portion was tested by verifying instructions being fed in matched the correct values, then to test the legal instruction checking different illegal instructions were fed in and the instructions were verified to have changed to NOPs after passing through, then to test addressing modes, different immediate values and addressing containing values were tested and EA was output to terminal each time to verify the data was correct. Then ALU flags were set up and tested by making sure the ALU output the incoming flag and verifying the flag information was correct for various instructions. After all this, the last test was that the ALU was executing the proper operations and that registers, and memory were being modified correctly. This was verified by the output of the accumulator, index registers, and a memdump at program end.

Emulator::ALUOp(int, int)

- **Description**: This function performs the execution of the ALU given two integer inputs containing flags for input, output, operation, and conditions to be met and an operand 2 input. Bits [0,2] contain the operation type including add, sub, clr, com, and, or, and xor; bits [3,5] contain the output, a 0 value did nothing, a 1 value output the result to an IDX register, a 2 value output to the memory data register, a 4 value output the result to the accumulator,

and a 6 value put the instruction into the instruction counter; bits [6, 7] defined what the operand 1 would be with 0 being a 0 input, 1 being the accumulator register and 2 being an IDX register input; bits [8-9] defined conditions to be met to run the ALU, if 0 the operation would run unconditionally, if 1 the operation would only run if the accumulator was 0, if 2 the operation would run if the accumulator was negative, and if 3 the operation would run if the accumulator was positive; and finally, bits [10-11] chose an index register Once the flags were set, operand 1 was set up and conditions were verified, then the operation would run and put the result in the ALU register, the ALU register would then route the result to the specified output and the function would exit.

- **Inputs**: int – a 12-bit flag containing information for the ALU to execute on data
  int – the operand 2 of the accumulator, also known as the effective address (EA)
- **Outputs**: none
- **Testing Methods**: To test this function, first flags were identified and output, once all the flags were working correctly and operand 1 was verified to be correct the conditionals were set up, currently these only operate on jump operations so to test different jump instructions were executed to verify they ran only under certain conditions. Then, the operations were tested and the ALU register was output to the screen, to verify operations were working correctly the ALU had to match with the operation given, add giving EA + AC or EA + 0 depending on flags, and so on, then the output was verified by checking the output registers IC, MDR, IDX[N], and AC after each execution cycle to verify that they were updated when necessary.

Emulator::getBits(std::bitset<24>, int, int);

- **Description:** This function is a helper function used in decoding instructions to extract specific bits from a 24-bit bitset and shift them appropriately. To achieve this the function would start at the starting index of the bitset and ORR the result to a temp variable shifted over by the number of times through the loop, then the resulting variable would be returned
- **Inputs:** std::bitset<24> - a 24-bit bitset containing the full undecoded instruction
  int – the starting index for extraction
  int – the ending index for extraction
- **Outputs:** int – the value extracted from the bitset between the starting and ending indexes
- **Testing Methods:** To test this method I would pass in a 24 bit bitset of varying values and extract different indexes from it, if the values returned matched up with the correct bits extracted then it was deemed correct.

Emulator::printAccumulator()

- **Description**: This function is a helper function that is called at the end of every instruction execution to output the resulting accumulator register and the four index registers. This is

achieved with a single printf statement containing the formatted output and passing it the appropriate register information.
- **Inputs**: none
- **Outputs**: no value output, physical output to the terminal containing accumulator register value and the four index register values formatted with 6 hexadecimal bits and 3 hexadecimal bits respectively.

Emulator::run()

- **Description:** This is the beating heart of the emulator, this is the main loop that controls the flow of everything. It runs a loop that only ends when the emulator indicates it's been halted or the instruction counter has hit the end of memory. The loop begins by fetching the instruction and outputting it as a 3 digit hex value onto the screen followed by a colon, then printing out the raw value of the instruction once fetched from memory as a 6 digit hexadecimal value. The instruction value is passed to the instruction register and the decode/execute function is called to decode the instruction and run ALU operations on the instruction. Once the function returns the values of the accumulator and four index registers are output to the terminal and the program checks for an error message, if one exists the program prints out "Machine Halted – " followed by an error message giving a reason for the halt and the loop ends, otherwise the loop continues with the next instruction.
- **Inputs:** none
- **Outputs:** no value output, physical output to terminal containing instruction address, value, and accumulator and four index register values along with a halt message if the emulation has halted.
- **Testing Methods:** This method was tested by verifying the IC was incrementing by one address each pass, then by verifying that the instruction register contained the value of the previously fetched instruction and was being passed to the decode and execute function. Once returned from that function the last test was with halting messages displaying when halt was executed, this was tested by trying different instructions and illegal/legal opcodes and addressing modes and verifying the error messages matched correctly.