OFC-T34 Documentation

Author: Nathan Ducasse

The T34 is a 24-bit word-addressable accumulator architecture. It supports up to 64 instructions, up to four addressing modes, and 212 (4096) words of memory.

Compilation/System Requirements:

Requires Python3.6

Running the Program:

Python3 prog2.py <objectfile.o> [-memdump] [-parse]

Program Arguments:

<objectfile.o>: Name of the object file that the T34 Emulator will run. This argument is required for program execution.

-parse: Running the T34 Emulator with the -parse flag runs the Parse function at the end of the program, once the object file has been processed.

-memdump: Running the T34 Emulator with the -memdump flag runs the Memory Dump function at the end of the program, after the object file has been processed and the parse function has finished executing, if specified.

Runtime Notes:

Parse and Memory Dump functions are executed at the end of the program if their flags are specified, and are executed in that order. Parse requires input from the user.

Input for the parse function is accepted in two formats: ‘0xfff i’ and ‘fff i’
Example: “(‘0xfff i’ or ‘fff i’): 0x0c5 1” and “(‘0xfff i’ or ‘fff i’): 0c5 1” will both print the contents of memory at address 0c5, whereas “(‘0xfff i’ or ‘fff i’): 0x0c5 2” will print the contents at 0c5 followed by 0c6.

Implementation Notes:

One global constant is used for convenience, the CONST_OPCODES tuple. The CONST_OPCODES constant is used for checking the intended operation for each given opcode in the do_instructions function.

In terms of functionality, the program for the most part does not directly emulate the T34 hardware and datapath, but rather functionally emulates a portion of it. For example, the alu_op function is not emulating an actual ALU but rather returns whatever the ALU would have calculated for the given arguments, where it is called. Instructions that do not utilize the ALU do not call alu_op, and alu_op would not handle them properly if they did.

The program will start at the given pc and continue until a halt signal is reached, either through an error code or a HALT operation. The program has a check for buffer overflow and will exit with the error “Error” if an overflow occurs.

The X0-X4 registers are implemented but currently unused. The MAR and MDR registers are not used in this implementation. The ABUS and DBUS are also unimplemented. The Emulator does not support indexing or indirect address modes yet, and all instructions using registers are also currently unimplemented.

Notes on Exceptions:

The operation exceptions are evaluated in the following order:

-Halt Operation (end of program)
-Undefined Opcode
-Unimplemented Opcode
-Illegal Address Mode
-Unimplemented Address Mode
-Overflow Error (output as "Error")

Function Notes:

Main: The Main function is where the mem and pc variables are declared locally, rather than globally. Main is where the passed object file is parsed and read into mem, and then the memory and PC get passed to do_instructions for execution. Memdump and Parse are called after do_instructions finishes execution if their respective flags are specified.

Do_instructions: do_instructions is the function which executes the provided object file. It uses a while loop that executes the next command until either a HALT instruction is executed, or an exception occurs. The function first creates all necessary registers and flags, and then proceeds to the loop. Within the loop, the current operation is loaded into ir from memory. The instruction is then split into its address, opcode, and address mode. Once the opcode is checked for validity, if the operation does not care about address mode it executes. If the operation cares about address mode (does not ignore address mode), then the address mode is checked for validity. If valid, it performs the operation on that address mode if it is implemented. Then the PC is updated and the loop executes again. At each step of the way, the program outputs the desired information available at that step to the screen, in the form:

```
address:  instruction  mnemonic  EA(hex)/"   "/IMM  AC[accumulator] X0-4[value]
```

Is_valid_op: The is_valid_op function takes the name of an operation and specifies if it is a valid operation.

Alu_op: alu_op takes two values and an instruction, and returns the values after applying the instruction. If the given instruction was invalid, it returns the second value by default.

Memdump: The Memdump function takes in the mem list and prints all of the memory locations that are not empty (contain nonzero values), from smallest to largest memory location.

Parse: The parse function is passed the mem list and takes input from the user. It prints the contents of mem at the address specified by the user and subsequent addresses up to the number after the initial address that was also specified by the user.

Testing:

The main program functionality was tested with the provided .o files:

Object File 1:

```
50 1 000000
c4 5 050404 200800 300800 102840 050c00
101 2 300 9
200 1 30
300 1 10
c4
```

Output:

```
0c4:  050404  LD    IMM  AC[000050]  X0[000]  X1[000]  X2[000]  X3[000]
0c5:  200800  ADD   200  AC[000080]  X0[000]  X1[000]  X2[000]  X3[000]
0c6:  300800  ADD   300  AC[000090]  X0[000]  X1[000]  X2[000]  X3[000]
0c7:  102840  SUB   102  AC[000087]  X0[000]  X1[000]  X2[000]  X3[000]
0c8:  050c00  J     050  AC[000087]  X0[000]  X1[000]  X2[000]  X3[000]
050:  000000  HALT       AC[000087]  X0[000]  X1[000]  X2[000]  X3[000]
Machine Halted - HALT instruction executed
```

Object File 2:

```
50 1 000000
100 5 200400 201840 202800 005844 050c84
200 3 30 31 10
100
```

Output:

```
100:  200400  LD    200  AC[000030]  X0[000]  X1[000]  X2[000]  X3[000]
101:  201840  SUB   201  AC[ffffff]  X0[000]  X1[000]  X2[000]  X3[000]
102:  202800  ADD   202  AC[00000f]  X0[000]  X1[000]  X2[000]  X3[000]
103:  005844  SUB   IMM  AC[00000a]  X0[000]  X1[000]  X2[000]  X3[000]
104:  050c84  JN    ???  AC[00000a]  X0[000]  X1[000]  X2[000]  X3[000]
Machine Halted - illegal addressing mode
```

Object File 3:

```
50 1 000000
75 3 30 20 10
ff 7 075400 040804 077440 005884 076400 077800 0309c4
ff
```

Output:

```
0ff:  075400  LD    075  AC[000030]  X0[000]  X1[000]  X2[000]  X3[000]
100:  040804  ADD   IMM  AC[000070]  X0[000]  X1[000]  X2[000]  X3[000]
101:  077440  ST    077  AC[000070]  X0[000]  X1[000]  X2[000]  X3[000]
102:  005884  CLR   IMM  AC[000000]  X0[000]  X1[000]  X2[000]  X3[000]
103:  076400  LD    076  AC[000020]  X0[000]  X1[000]  X2[000]  X3[000]
104:  077800  ADD   077  AC[000090]  X0[000]  X1[000]  X2[000]  X3[000]
105:  0309c4  ????  IMM  AC[000090]  X0[000]  X1[000]  X2[000]  X3[000]
Machine Halted - undefined opcode
```

The memdump and parse functions were tested by making variations to the given object file and ensuring that the functions had the desired output.

Object File 1:

```
50 1 000000
4c1 20 050404 200800 300800 102840 050c00 001000 2fc121 9888ff fffcfe 432a22 \
000000 abcdee 012345 789abc 0caab2 ab3bab 9c9c9c 0f0f0f 888888 123123
101 2 300 9
200 1 30
300 1 10
c4
```

Output:

```
Parsing: Enter address and interval to dump:
('0xfff i' or 'fff i'):4c1 20
ADDR
OP AM
4c1: 000001010000 010000 000100
4c2: 001000000000 100000 000000
4c3: 001100000000 100000 000000
4c4: 000100000010 100001 000000
4c5: 000001010000 110000 000000
4c6: 000000000001 000000 000000
4c7: 001011111100 000100 100001
4c8: 100110001000 100011 111111
4c9: 111111111111 110011 111110
4ca: 010000110010 101000 100010
4cb: 000000000000 000000 000000
4cc: 101010111100 110111 101110
4cd: 000000010010 001101 000101
4ce: 011110001001 101010 111100
4cf: 000011001010 101010 110010
4d0: 101010110011 101110 101011
4d1: 100111001001 110010 011100
4d2: 000011110000 111100 001111
4d3: 100010001000 100010 001000
4d4: 000100100011 000100 100011
Memory Dump:
101: 000300
102: 000009
200: 000030
300: 000010
4c1: 050404
4c2: 200800
4c3: 300800
4c4: 102840
4c5: 050c00
4c6: 001000
4c7: 2fc1214c8: 9888ff
4c9: fffcfe
```

4ca: 432a22
4cc: abcdee
4cd: 012345
4ce: 789abc
4cf: 0caab2
4d0: ab3bab
4d1: 9c9c9c
4d2: 0f0f0f
4d3: 888888
4d4: 123123