Project – Spring 2018

The group project will be a partial x86 emulator.

Some key limitations:

* + - Only standard user-mode instructions are included
      * + The software interrupt instruction is included, but used as a way to access O.S. routines, and are not implemented beyond that point
    - Except for MOV, only register destinations are supported
    - Only signed 32-bit operands are supported
    - Registers supported: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP
      * EFlags will only contain the Carry
    - Only the following addressing modes are supported
      * Register
      * Register Indirect: [ECX]
      * Indexed: 99[ECX]
        + 8 bit offset
        + 32 bit offset
      * Absolute addressing: XYZ
      * 32-bit Immediate value
    - Conditional jumps will only jump to locations within a 8-bit offset
      * Unconditional jumps and subroutine calls do not have this limitation
    - We will only use register parameters for subroutines
    - Shifts will only be by constant values (i.e., SHL EAX,5)

General Structure

* + - Most of the instructions (add, sub, xor, etc.) have the format:
      * First byte: Opcode (add, sub, xor, etc.)
      * ModR/M Byte
        + This will specify the destination register, as well as the source location
        + Register encodings

|  |  |
| --- | --- |
| EAX | 000 |
| ECX | 001 |
| EDX | 010 |
| EBX | 011 |
| ESP | 100 |
| EBP | 101 |
| ESI | 110 |
| EDI | 111 |

* + - * + Format: mm ddd sss
        + mm – Addressing mode to use

00 – Register indirect (the sss bits will indicate which register to use)

Exception: 00 101 (otherwise [EBP]) is absolute addressing, and the Mod R/M byte will be followed by the 32-bit address of the memory operand

01 – Indexed

The sss bits indicate the register

The ModR/M byte will be followed by a 8-bit offset

10 – Indexed

The sss bits indicate the register

The ModR/M byte will be followed by a 32-bit offset

11 - Register

The sss bits indicate the register

The ddd bits indicate the destination register

For instructions that have a constant as the second operand, the ddd bits will be used to supply additional opcode information, while the rest of the ModR/M byte specifies the register (we won’t be using memory locations in this instance)

Instructions we will support:

* + “reg/mem32” is used to refer to either a register or memory location specified by the Mod R/M byte, and “(offset)” is used to indicate a offset or memory address if appropriate.
  + “rel32” is a signed 32-bit IP-relative offset
  + “rel8” is a 8-bit signed IP-relative offset

Note: If you find any discrepancies between what is here and the hex values in listing (.LST) file,

the .LST file is correct.

|  |  |  |
| --- | --- | --- |
| ADD reg,reg/mem32 | 00000011 ModR/M (offset) | Add to register |
| ADD reg,const | 10000011 11000ddd const | Add constant to register |
| AND reg,reg/mem32 | 00100011 ModR/M (offset) | Logical and |
| AND reg,const | 10000011 11111ddd const | Logical and with constant |
| CALL addr | 11101000 rel32 | Call Subroutine |
| CMP reg,reg/mem32 | 00111011 ModR/M (offset) | Compare |
| CMP reg,const | 10000001 11111ddd const | Compare to constant |
| INT xx | 11001101 xxxxxxxx | Software Interrupt |
| Warning: There are also forms of the conditional jumps that accept a 32-bit IP-relative offset. You do not have to implement those instructions as part of your project, but the assembler will quietly use one if your jump is too far away. You can look at the .LST file to check if this has happened. | | |
| JE loc | 01110100 rel8 | Jump Equal (also JZ) – Jump if the Zero flag is set |
| JNE loc | 01110101 rel8 | Jump if not equal (also JNZ) – Jump if the Zero flag is clear |
| JG loc | 01111111 rel8 | Jump if Greater Than – Jump if Zero is clear and Sign == Overflow |
| JGE loc | 01111101 rel8 | Jump if Greater or Equal – Jump if Sign == Overflow |
| JL loc | 01111100 rel8 | Jump if Less Than – Jump if Sign != Overflow |
| JLE loc | 01111110 rel8 | Jump if Less or Equal – Jump if Zero is set or Sign != Overflow |
| JMP loc | 11101001 rel8 | Unconditional jump, short form |
| JMP loc | 11101001 rel32 | Unconditional jump, long form |
| MOV reg,reg/mem32 | 10001011 ModR/M (offset) | Move |
| MOV reg,const | 10111rrr const32 | Move a constant to a register (the register is part of the opcode byte) |
| MOV EAX,addr | 10100001 addr32 | Move memory to EAX |
| MOV reg/mem32,reg | 10001001 ModR/M (offset) | Move register to memory – Note that in the ModR/M byte, the ddd bits indicate the register, while the mm/sss bits still indicate the memory location |
| MOV addr,EAX | 10100011 addr32 | Move EAX to memory |
| OR reg,reg/mem32 | 00001011 ModR/M (offset) | Logical or |
| OR reg,const | 10000011 11001ddd const | Logical or with constant |
| POP reg | 01011rrr | Pop register |
| PUSH const | 01101010 const8 | Push 8-bit constant (sign extended) |
| PUSH const | 01101000 const32 | Push 32-bit constant |
| PUSH reg | 01010rrr | Push register |
| RET | 11000011 | Return from subroutine |
| SAL reg,const | 11000001 11100rrr const8 | Shift arithmetic left by constant |
| SAR reg,const | 11000001 11111rrr const8 | Shift arithmetic left by constant |
| SUB reg,reg/mem32 | 00101011 ModR/M (offset) | Subtract register |
| SUB reg,const | 10000011 11101ddd const | Subtract constant from register |
| XOR reg,reg/mem32 | 00110011 ModR/M (offset) | Exclusive-Or |
| XOR reg,const | 10000011 11110ddd const | Exclusive-Or constant |

Creating test programs: I have included a few test programs, but if you want to create a test program of your own, follow the following steps:

1. Create your assembler program “MyProg.asm”. You can use Visual Studio, Notepad, or your favorite editor.
   1. Note: Instead of “INCLUDE IRVINE32.INC”, you should use “INCLUDE PROJECT.INC”
2. Open up a command line prompt, and run “mm MyProg.asm”. This will create the executable (.EXE), a listing file (.LST) and some other files we will ignore
   1. Note: The file “mm.bat” is designed to run in the lab. If you have Visual Studio at home, you may need to adjust the path the “ML.EXE” and “LINK.EXE”.
   2. Note: Because the proper way to access Windows routines is too messy for this project, the resulting executable files will only run in the emulator
3. Copy (if necessary) the “MyProg.asm”, “MyProg.exe”, and “MyProg.lst” to your working location

O.S. Methods and the INT instruction: The way I have chosen to handle user input/output is via the software INT instruction. There are six routines corresponding to the routines CrLF, ReadInt, WriteInt, WriteString, and ExitProcess. These routines take the same values as their Irvine32 counterparts. We will not do a proper emulation of the INT instruction, but simply emulate the corresponding operation and continue with the next instruction.

Running my solution to the project:

Type “java –jar SimX86.jar” from the command line. The program will then ask what program you want to load. Respond with either “MyProg.exe” or just “MyProg”. You will then get a prompt “Command:”, with the following options:

* Q – Quit (exit the emulator)
* E *xxx* – Evaluate the expression where all numbers are assumed to be hex and only + and – are allowed, and display the result
* S – Step: Execute one instruction, display the new value of the registers, and stop
* G – Go: Run the program at full speed
* M *xxx* – Display memory starting at address *xxx*
* U *xxx* – Unassemble – Display the machine code and assembler instructions starting at the given address (if no address is provided, the current value of EIP will be used)
  + This relies on information in the .LST file
* L *prog* – Load – Reset the CPU and load the specified program. If no program is specified, the existing program will be reloaded

I will note that my version of the emulator does include a few items (including IMUL and IDIV) which I have chosen to remove from the project. You do not have to implement anything that is not specified here.