Introduction to Y86

CSO Tutorial 5 - Feb 17th '20

Prepared by Kripa Anne

What have we learnt?

Instruction Set Architecture (x86) supported by a particular family of processors (eg: Intel IA32).

What is Y86?

Another instruction set, basically simpler x86. We plan to build a pipelined processor (with a sequential design) that can implement Y86.

What will learning Y86 teach us?

How processor hardware systems execute the instructions of a particular ISA, how processor design is done, etc.

Y86 Programmer Visible State

- 8 program registers
- 3 condition codes Zero, Sign, Overflow (info about arithmetic & logical instructions)
- PC - address of instr. being executed
- Memory holds program & data (virtual addressing)
- Status code Stat indicates if any or no Exception occurred (overall state of program execution)

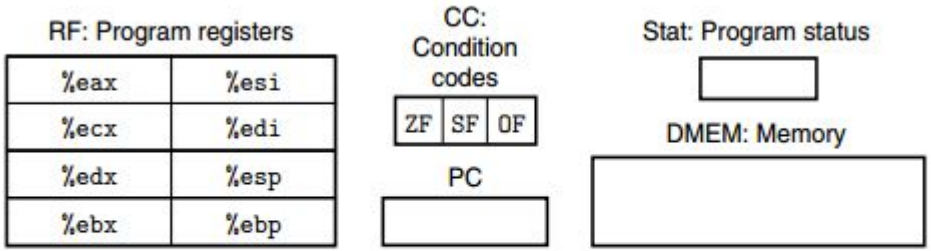


Figure 4.1 **Y86 programmer-visible state.** As with IA32, programs for Y86 access and modify the program registers, the condition code, the program counter (PC), and the memory. The status code indicates whether the program is running normally, or some special event has occurred.

Status Codes

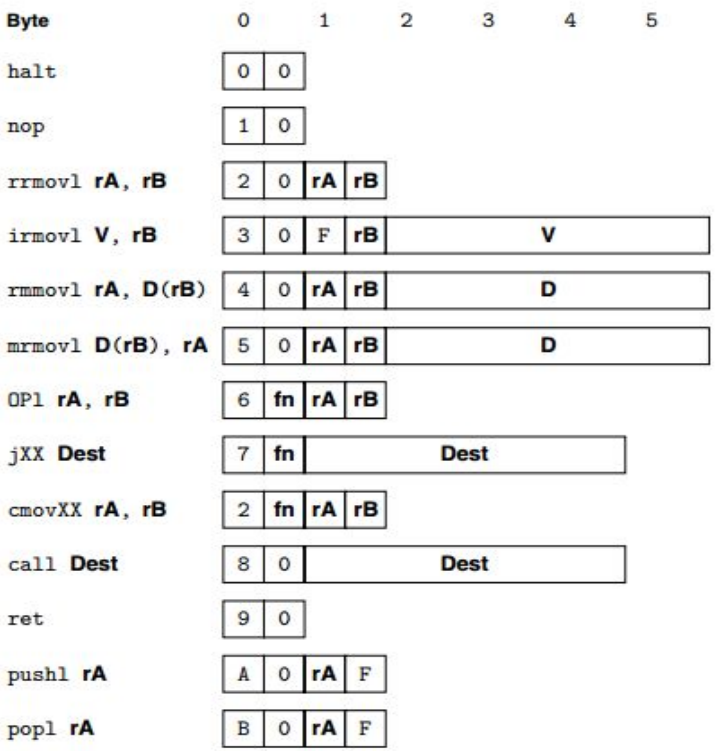
Value	Name	Meaning
1	AOK	Normal operation
2	HLT	halt instruction encountered
3	ADR	Invalid address encountered
4	INS	Invalid instruction encountered

Figure 4.5 **Y86 status codes.** In our design, the processor halts for any code other than AOK.

Instead of handling the exception, our processor will stop executing when **stat** changes from AOK to any of the other 3 statuses.

Y86 Instructions

Figure 4.2
Y86 instruction set. Instruction encodings range between 1 and 6 bytes. An instruction consists of a 1-byte instruction specifier, possibly a 1-byte register specifier, and possibly a 4-byte constant word. Field `fn` specifies a particular integer operation (`OPl`), data movement condition (`cmovXX`), or branch condition (`jXX`). All numeric values are shown in hexadecimal.



A Few Things To Note About The Y86 Instruction Set:

- Includes only 4 byte integer operations
- Prev. known `movl` instruction has been split into 4 different mov's:
  - `irmovl` (immediate source - register destination)
  - `rrmovl` (register source - register destination)
  - `mrmovl` (memory source - register destination)
  - `rmmovl` (register source - memory destination)
- Both memory movement instructions use base-displacement addressing.
- 4 integer operations: `addl, subl, andl, xorl`. Only on register data.
- 7 jump instructions
- 6 conditional move instructions - similar to `rrmovl`.
- `call & ret` jumps to and returns from destination address respectively.
- `pushl & popl` implement push and pop.
- `halt` stops instruction execution.

Instruction Encoding

Each instruction requires between 1 and 6 bytes. The first byte in every instruction specifies the *instruction type* and is split into two 4-bit parts - high-order/**code** part and low-order/**function** part.

eg: `addl` is [6, 0] while `subl` is [6, 1].

Note: `rrmovl` is treated as an unconditional move, so is grouped along with the other conditional moves and given function code 0. Similarly, `jmp` is unconditional jump (`jXX`).

Longer instructions will have one or two *register specifier bytes* or an immediate 4-byte constant word.

RISC and CISC architecture

Reduced Instruction Set Computers

- Fewer instructions.
- RISC instructions are simpler, so no single instruction has a long execution time.
- Fixed-length encoding
- Simple addressing formats (base + displacement)
- No condition codes
- Arithmetic and logical operations only use register operands. Memory referencing is allowed through a load/store architecture.

Complex Instruction Set Computers

- Large number of instructions.
- Since individual instructions are more complex, some instructions have long execution times.
- Variable length encoding
- Multiple addressing formats.
- Have condition codes
- Arithmetic and logical operations can be applied to both memory and register operands.
- Recent CISC machines use pipelining to translate CISC instructions into a sequence of RISC-like instructions.

Writing a Program in Y86 + using the Y86 Emulator

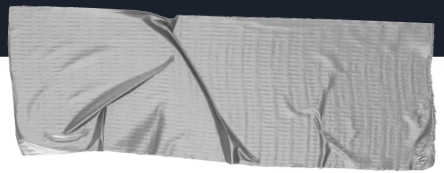
Program to compute sum of a 4-element array

```
# Execution begins at address 0
        .pos 0
init:   irmovl Stack, %esp      # Set up stack pointer
        irmovl Stack, %ebp      # Set up base pointer
        call Main               # Execute main program
        halt                    # Terminate program

# Array of 4 elements
        .align 4
array:  .long 0xd
        .long 0xc0
        .long 0xb00
        .long 0xa000

Main:   pushl %ebp
        rrmovl %esp,%ebp
        irmovl $4,%eax
        pushl %eax              # Push 4
        irmovl array,%edx
        pushl %edx              # Push array
        call Sum                # Sum(array, 4)
        rrmovl %ebp,%esp
        popl %ebp
        ret

        # int Sum(int *Start, int Count)
Sum:    pushl %ebp
        rrmovl %esp,%ebp
        mrmovl 8(%ebp),%ecx     # ecx = Start
        mrmovl 12(%ebp),%edx    # edx = Count
        xorl %eax,%eax          # sum = 0
        andl   %edx,%edx        # Set condition codes
        je     End
Loop:   mrmovl (%ecx),%esi      # get *Start
        addl %esi,%eax          # add to sum
        irmovl $4,%ebx          #
        addl %ebx,%ecx          # Start++
        irmovl $-1,%ebx         #
        addl %ebx,%edx          # Count--
        jne    Loop             # Stop when 0
End:    rrmovl %ebp,%esp
        popl %ebp
        ret
```



Class Task

1) Modify the sample Y86 program above to implement a function **AbsSum** that computes the sum of absolute values of an array. Use a conditional jump instruction within your inner loop.

2) Write a program in Y86 to compute the product of the maximum and minimum value stored in a 4-element array (and debug it) using the emulator.

For a more detailed explanation and further exploration of Y86, refer to Section 4.1 and onwards of Computer Systems: A Programmer's Perspective.