Instruction Set Architecture

CPSC 235 - Computer Organization

References

- Slides adapted from CMU

Instruction Set Architecture (ISA)

- Assembly Language View
  - Processor state (registers, memory, etc.)
  - Instructions
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use a variety of methods to make it run fast
    - For example, execute multiple instructions simultaneously

Y86-64 ISA

- The Y86-64 processor is a simple Instruction Set Architecture based on x86-64
  - Fewer data types, instructions, and addressing modes
  - Simple byte-level encoding
  - ISA sufficiently complete to write programs that manipulate integer data
- Good example for processor design; just complicated enough to show the challenges involved in implementation

Y86-64 Processor State

- Program registers
  - 15 64-bit registers (omit `%r15`)
- Condition codes
  - Single bit flags set by arithmetic or logical instructions
  - Zero (ZF), Negative (SF), Overflow (OF)
- Program Counter
  - Indicates address of next instruction
- Program status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

Y86-64 Instruction set

Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0:0									
`nop`	1:0									
`cmovXX rA, rB`	2:n	rA:rB								
`irmovq V, rB`	3:0	F:rB	V	V	V	V	V	V	V	V
`rmmovq rA, D(rB)`	4:0	rA:rB	D	D	D	D	D	D	D	D
`mrmovq D(rB), rA`	5:0	rA:rB	D	D	D	D	D	D	D	D
`OPq rA, rB`	6:n	rA:rB								
`jXX L`	7:n	L	L	L	L	L	L	L	L	
`call L`	8:0	L	L	L	L	L	L	L	L	
`ret`	9:0									
`pushq rA`	A:0	A:F								
`popq rA`	B:0	A:F								

Y86-64 Instructions

- Format
    - 1-10 bytes of information read from memory
        - Can determine length from first byte
        - Not as many instruction types, and simpler encoding than with x86-64
    - Each accesses and modifies some part(s) of the program state

`cmovXX` Instructions

Byte	0	1	2	3	4	5	6	7	8	9
`rrmovq rA, rB`	2:0	rA:rB								
`cmovle rA, rB`	2:1	rA:rB								
`cmovl rA, rB`	2:2	rA:rB								
`cmove rA, rB`	2:3	rA:rB								
`cmovne rA, rB`	2:4	rA:rB								
`cmovge rA, rB`	2:5	rA:rB								
`cmovg rA, rB`	2:6	rA:rB								

OPq Instructions

Byte	0	1	2	3	4	5	6	7	8	9
`addq rA, rB`	6:0	rA:rB								
`subq rA, rB`	6:1	rA:rB								
`andq rA, rB`	6:2	rA:rB								
`xorq rA, rB`	6:3	rA:rB								

`jXX` Instructions

Byte	0	1	2	3	4	5	6	7	8	9
`jmp L`	7:0	L	L	L	L	L	L	L	L	
`jle L`	7:1	L	L	L	L	L	L	L	L	
`jl L`	7:2	L	L	L	L	L	L	L	L	
`je L`	7:3	L	L	L	L	L	L	L	L	
`jne L`	7:4	L	L	L	L	L	L	L	L	
`jge L`	7:5	L	L	L	L	L	L	L	L	
`jg L`	7:6	L	L	L	L	L	L	L	L	

Encoding Registers

- Each register has 4-bit ID

`%rax`	0	`%r8`	8
`%rcx`	1	`%r9`	9
`%rdx`	2	`%r10`	A
`%rbx`	3	`%r11`	B
`%rsp`	4	`%r12`	C
`%rbp`	5	`%r13`	D
`%rsi`	6	`%r14`	E
`%rdi`	7		F

- Register ID 15 (0xF) indicates “no register”
  - This will be used in the hardware design

Instruction Example

- Addition instruction
  - generic form: `addq rA, rB`
  - encoded representation: `60 rA rB`
- Add value in register rA to that in register rB
  - store result in register rB
  - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- Example: `addq %rax, %rsi`, encoding: `60 06`
- Two-byte encoding
  - first indicates instruction type
  - second gives source and destination registers

Arithmetic and Logical Operations

- Referred to generically as “OPq”
- Encodings differ only by “function code”
  - Low-order 4 bytes in first instruction word
- Set condition codes as side effect

Move Operations

Instruction	Source	Destination
`rrmovq rA, rB`	Register	Register
`irmovq V, rB`	Immediate	Register
`rmmovq rA, D(rB)`	Register	Memory
`mrmovq D(rA), rB`	Memory	Register

- Similar to x86-64 `movq` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

x86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
`movq %rsp, %rbx%`	`rrmovq %rsp, %rbx`
`movq -12(%rbp), %rcx`	`mrmovq -12(%rbp), %rcx`
`movq %rsi, 0x41c(%rsp)`	`rmmovq %rsi, 0x41c(%rsp)`

Move Instruction Examples

Y86-64	Encoding
`irmovq $0xabcd, %rdx`	`30 82 cd ab 00 00 00 00 00 00`
`rrmovq %rsp, %rbx`	`20 43`
`mrmovq -12(%rbp), %rcx`	`50 15 f4 ff ff ff ff ff ff ff`
`rmmovq %rsi, 0x41c(%rsp)`	`40 64 1c 04 00 00 00 00 00 00`

Conditional Move Instructions

- Referred to generically as “cmovXX”
- Encodings differ only by “function code”
- Based on values of condition codes
- Variants of `rrmovq` instruction
  - conditionally copy value from source to destination

Jump Instructions

- Referred to generically as “jXX”
- Encodings differ only by “function code”
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
    - Unlike PC-relative addressing in x86-64

Y86-64 Program Stack

- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by `%rsp`
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement the stack pointer
  - After popping, increment stack pointer

Stack Operations

- `pushq rA`: `A 0 rA F`
  - Decrement `%rsp` by 8
  - Store word from rA to memory at %rsp
- `popq rA`: `B 0 rA F`
  - Read word from memory at `%rsp`
  - Save in rA
  - Increment `%rsp` by 8

Subroutine Call and Return

- `call Dest`
    - push address of next instruction on the stack
    - start executing instructions at Dest
- `ret`
    - Pop value from stack
    - Use as address for next instruction

Miscellaneous Instructions

- `nop`: `1 0`
  - Do nothing (no operation)
- `halt`: `0 0`
  - Stop executing instructions
  - x86-64 has a comparable instruction but cannot execute it in user mode
  - Encoding ensures that program hitting memory initialized to zero will halt

Status Conditions

Mnemonic	Code	Comment
`AOK`	1	Normal operation
`HLT`	2	Halt instruction encountered
`ADR`	3	Bad address (either instruction or data) encountered
`INS`	4	Invalid instruction encountered

- Desired behavior
    - If `AOK` keep going
    - Otherwise, stop program execution

Y86-64 Sample Program Structure

```
init:              # Initialization
    ...
    call Main
    halt

    .align 8       # Program data
array:
    ...
Main:              # Main function
    ...
    call len
len:
   ....

   .pos 0x100      # Placement of stack
Stack:
```

Y86-64 Sample Program Structure

- Program starts at address 0
- Must set up stack
  - Location
  - Pointer values
  - Make sure code is not overwritten
- Must initialize data

Y86-64 Sample Program Structure

```
init:
    # Set up stack pointer
    irmovq Stack, %rsp
    # Execute main program
    call Main
    # Terminate
    halt

# Array of 4 elements + terminating 0
    .align 8
Array:
    .quad 0x000d000d000d000d
    .quad 0x00c000c000c000c0
    .quad 0x0b000b000b000b00
    .quad 0xa000a000a000a000
    .quad 0
```

CISC Instruction Sets

- Complex Instruction Set Computer
- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- Arithmetic instructions can access memory
- Condition Codes
  - Set as side effect of arithmetic and logical instructions

RISC Instruction Sets

- Reduced Instruction Set Computer
- Fewer, simpler instructions
  - Might take more to get given task done
  - Can execute them with small and fast hardware
- Register-oriented instruction set
  - Many more (typically 32) registers
  - Used for arguments return pointer, temporaries
- Only load and store instructions can access memory
- No condition codes
  - Test instructions return 0/1 in register

Summary

- Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings
  - Somewhere between CISC and RISC
- How important is ISA Design?
  - Less now than before; with enough hardware can make almost anything fast