Lecture 5

Y86-64 Instruction Set Architecture

Having explored how real programs are represented in x64-64 instructions, in this lecture we will begin to explore how the actual instruction set architecture is implemented in hardware. To do so, we study a subset of the full x86-64 instruction set known as Y86-64. This will allow us to explore the actual representation of instructions and how the hardware control logic actually implements a processor design.

Y86-64 Programmer Visible State

The Y86-64 programmer visible state includes 15 general purpose 64-bit registers, named `%rax`, `%rcx`, `%rdx`, `%rbx`, `%rsp`, `%rbp`, `%rsi`, `%rdi`, `%r8`, through to `%r14`. `%rsp` is the stack pointer used by `push`, `pop`, `call` and `ret` instructions. An additional program counter, `PC`, keeps track of the address of the current instruction in memory.

There are three single-bit condition codes, `ZF`, `SF`, and `OF`, which are updated by arithmetic instructions, and indicate that the most recent operation resulted in a zero value, a negative value, and/or a two's complement overflow.

The memory where both programs and data are stored is represented as an array of bytes, referenced by a virtual address (the hardware and operating system turn this into a real physical address).

Y86-64 Instructions and Encoding

The Y86-64 instruction set is a subset of x86-64. It includes only 64-bit integer operations, and operates on 64-bit data which are referred to as words. The assembly code is similar to that of the x86-64 instruction set, with separate instructions replacing some of the complex operand options of the x86-64 instructions.

- `halt` stops instruction execution.
- `nop` is 'no operation' and does nothing.
- `rrmovq`, `irmovq`, `rmmovq`, `mrmovq` are equivalent to the x86-64 `movq` instruction, each taking either a register, immediate value, and memory location as a source, and a register or memory location as a destination.
- `OPq` represents the arithmetical and logical operators `addq`, `subq`, `andq` and `xorq`.
- `JXX` represents a family of jump instructions which use the condition codes to provide `jmp`, `jle`, `jl`, `je`, `jne`, `jge` and `jg` instructions, all of which take a 64-bit absolute destination.
- `cmovXX` represents a family of conditional move instructions, again using the condition codes to provide `cmovle`, `cmovl`, `cmove`, `cmovne`, `cmovge` and `cmovg` instructions. Each indicates a register as the source, and a register as the destination.

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

Table 1: Y86-64 instruction set encoding.

- `call` and `ret` interact with the `%rsp` pointer in the same way as the x86-64 instruction do, with `call` using a 64-bit absolute destination.

Instructions have encoded lengths with varies from a single byte to ten bytes, as shown in Table 1. All addresses and immediate values are 8 bytes long and are encoded as little endian values. Note that many of these instructions are parameterised; using the least significant four bits of the first byte to indicate their operation (see Table 2). Also, many instructions have a *register specifier byte* indicating which registers the instruction operates on. When only one register is required, the unused one is replaced with `0xF`. The coding of the register specifier byte is shown in Table 3.

Y86-64 Exceptions

The state of operation of the processor implementing the Y86-64 instructions is indicated by a status code which can take the values shown in Table 4.

			jmp	7	0	rrmovq	2	0
			jle	7	1	cmovle	2	1
			jl	7	2	cmovl	2	2
addq	6	0	je	7	3	cmove	2	3
subq	6	1	jne	7	4	cmovne	2	4
andq	6	2	jge	7	5	cmovge	2	5
xorq	6	3	jg	7	6	cmovg	2	6

Table 2: Function codes for `OPq`, `jXX` and `cmovXX` instructions.

Table 3: Y86-64 registers.

Number	Register Name	Number	Register Name
0	`%rax`	8	`%r8`
1	`%rcx`	9	`%r9`
2	`%rdx`	A	`%r10`
3	`%rbx`	B	`%r11`
4	`%rsp`	C	`%r12`
5	`%rbp`	D	`%r13`
6	`%rsi`	E	`%r14`
7	`%rdi`	F	No register

Table 4: Y86-64 status codes.

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

Y86-64 Code Examples

We can simulate the execution of Y86-64 instructions in a number of different ways. In order to fully explore the hardware implementation of the Y86-64 later in the course we will use a simulation of the actual processor hardware. However, here we use an instruction level simulator that can compile Y86-64 instructions to an object file, and then execute the instructions in this file.

Below is an example of Y86-64 code for such a simulator, which implements the addition of the elements of an array (as we saw earlier in C code).

```
.pos 0
    irmovq stack, %rsp
    call main
    halt

.align 8
array:
    .quad 0x0000000000000001
    .quad 0x0000000000000002
    .quad 0x0000000000000003
    .quad 0x0000000000000004

main:
    irmovq array, %rdi
    irmovq $4, %rsi
    call sum
    ret

sum:
    irmovq $8, %r8
    irmovq $1, %r9
    xorq %rax, %rax
    andq %rsi, %rsi
    jmp test
loop:
    mrmovq (%rdi), %r10
    addq %r10, %rax
    addq %r8, %rdi
    subq %r9, %rsi
test:
    jne loop
    ret

.pos 0x200
stack:
```

This code results in the following compiled code when using the `yas` assembler to give as output.

```
0x000:                      | .pos 0
0x000: 30f40002000000000000 |      irmovq stack, %rsp
0x00a: 803800000000000000   |      call main
0x013: 00                   |      halt
                            |
0x018:                      | .align 8
0x018:                      | array:
0x018: 0100000000000000     |     .quad 0x0000000000000001
0x020: 0200000000000000     |     .quad 0x0000000000000002
0x028: 0300000000000000     |     .quad 0x0000000000000003
0x030: 0400000000000000     |     .quad 0x0000000000000004
                            |
0x038:                      | main:
0x038: 30f71800000000000000 |      irmovq array, %rdi
0x042: 30f60400000000000000 |      irmovq $4, %rsi
0x04c: 805600000000000000   |      call sum
0x055: 90                   |      ret
                            |
0x056:                      | sum:
0x056: 30f80800000000000000 |      irmovq $8, %r8
0x060: 30f90100000000000000 |      irmovq $1, %r9
0x06a: 6300                 |      xorq %rax, %rax
0x06c: 6266                 |      andq %rsi, %rsi
0x06e: 708700000000000000   |      jmp test
0x077:                      | loop:
0x077: 50a70000000000000000 |      mrmovq (%rdi), %r10
0x081: 60a0                 |      addq %r10, %rax
0x083: 6087                 |      addq %r8, %rdi
0x085: 6196                 |      subq %r9, %rsi
0x087:                      | test:
0x087: 747700000000000000   |      jne loop
0x090: 90                   |      ret
                            |
0x200:                      | .pos 0x200
0x200:                      | stack:
```

When run with the `yis` simulator, the following output is generated.

```
Stopped in 34 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x000000000000000a
%rsp:   0x0000000000000000      0x0000000000000200
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000008
%r9:    0x0000000000000000      0x0000000000000001
%r10:   0x0000000000000000      0x0000000000000004

Changes to memory:
0x01f0: 0x0000000000000000      0x0000000000000055
0x01f8: 0x0000000000000000      0x0000000000000013
```

The output summarises the changes to registers and memory contents. In this case, the changes to the memory correspond to the return addresses pushed to the stack.

Simulator

The `yas` assembler and `yis` simulator can be downloaded from the links provided with Practical 2.

Further Reading

There are many descriptions of Y86-64 available on the web, and a description is included in Chapter 4 of the book below:

> Computer Systems: A Programmer's Perspective
> Randal E. Bryant and David R. O'Hallaron, Pearson, 2010.

A browser-based Y86-64 simulator which I modified from an existing Y86-32 one can be found here:

> http://www.cs.ox.ac.uk/people/alex.rogers/Y86-64/

The unmodified version is available from here:

> http://xsznix.github.io/js-y86/