The x86prime instruction set

By Finn Schiermer Andersen and Michael Kirkedal Thomsen

For students to learn about assembly programming is always hard. It is a significantly different model from what they often know from normal imperative programming (e.g. C, C++, C#). Understanding and working with all the details of X86_64, many of which are there as a reminiscent from old architecture designs, makes this even harder. To simplify this we have designed x86prime.

The instruction set x86prime has been designed to be an interesting subset of x86_64, but for parts that are unnecessarily complex (especially conditional jumps) have been inspired mainly RISC5.

In the following we will describe the instruction set of x86prime and its difference to x86. However, the following text is *not* as stand-alone text. It will not describe the detailed semantics of the instructions and we refer to other texts for this. But with a basic knowledge about instructions, you will be safe.

Register names

The resister names are the expected from x86_64 and the special purpose use are kept, with one noticeable change. we advise not to use `%r11` as it is used for procedure calls (see later).

`%rax`, `%rbc`, `%rcx`, `%rdx`, `%rbp`, `%rsi`, `%rdi`, `%rsp`, `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`.

Arithmetic and logical instructions

Arithmetic and logical instructions are close following the instructions from x86, that is a binary format where the destination register `d` is updated (by an operation) with the source register `s`. That is

```
<op> s,d
```

You can also use a constant (`i`) instead of the source register

```
<op> $i,d
```

Remember the leading `$` before the constant.

In the above `op` can be one of the following operations:

- `addq`: addition

- `subq` : subtraction
- `andq` : bitwise and
- `orq` : bitwise or
- `xorq` : bitwise xor
- `mulq` : signed multiplication
- `imulq` : unsigned multiplication
- `sarq` : shift arithmetic right (preserve topmost bit)
- `salq` : shift arithmetic left (zero into lsb, do not preserve topmost bit)
- `shrq` : shift (logical) right (zero into topmost bit)

Load effective address

Load effective address (lea) is a special x86 instruction. It is we was, as the name says, historically meant as an easy way of calculating addresses without reading from memory; hence do pointer arithmetic as in know from C. However, today it is also widely used by compilers do perform standard arithmetic. We have therefore also opted to include it in x86prime.

leaq can be used in the following ways:

Instruction	Semantics	Description
`leaq (s),d`	`s -> d`	Copy value from register `s` to register `d`
`leaq (,z,v),d`	`z * v -> d`	Scale register `z` by `v` before copy to `d`
`leaq (s,z,v),d`	`s + z * v -> d`	Scale and sum
`leaq i,d`	`i -> d`	Get a constant
`leaq i(s),d`	`i + s -> d`	Add a constant
`leaq i(,z,v),d`	`i + z * v -> d`	Add a constant to a scaled value
`leaq i(s,z,v),d`	`i + s + z * v -> d`	Do everything

Note `v` and `i` are constants while `s`, `d` and `z` are resister names. The scale factors `v` can only be the constants 1, 2, 4, and 8

Data transfer instructions

Like with x86_64 `movq` are the instruction for loading and storing data. However, x86prime has limited the different ways that is can be used.

Instruction	Description
`movq s,d`	reg->reg copy
`movq (s),d`	load (memory -> reg copy)
`movq d,(s)`	store (reg -> memory copy)
`movq $i,d`	constant -> register
`movq i(s),d`	load (memory -> reg copy)
`movq d,i(s)`	store (reg -> memory copy)

Conditional and unconditional jumps

Program flow control is where x86prime diverges significantly from x86_64. We have simply scrapped the concept of branch flags and instead uses conditional branches that includes simple branch expressions as seen in MIPS and RISC5.

The most simple of the instructions are the unconditional jump, which follows x86. In the following `p` is the label at which the execution should continue.

```
jmp p
```

More interesting are the conditional jumps (also called branches). To avoid the branch flags of x86, we will instead use a compare-and-branch instruction. This means that an instruction `<cb>` takes two values and compares these using a specified two-argument Boolean operator. If this evaluates to true execution will jump to a label `p`; otherwise the next instruction will be executed. It can thus be seen a performing a compare followed by a branch instruction in x86.

There are two way to give values to a compare-and-branch instruction. Either by comparing the values of two registers `s` and `d`

```
<cb> s,d,p
```

or comparing the values of a register `d` to an intimidate `i`.

```
<cb> $i,d,p
```

Though the compare-and-branch instruction is not part of the original x86 instruction set, we ensure that the specific conditions in x86prime carry the same meaning as for x86. Thus

- `cbe`: Equal
- `cbne`: Not equal
- `cbl`: less (signed)
- `cble`: less or equal (signed)
- `cbg`: greater (signed)
- `cbge`: greater or equal (signed)
- `cba`: above (unsigned)
- `cbae`: above or equal (unsigned)
- `cbb`: below (unsigned)
- `cbbe`: below or equal (unsigned)

Note that signed and unsigned comparisons are different.

For example, `cble %rdi,%rbp,target` evaluates if `%rdi` <= `%rbp` (signed) then jump to `target`.

Procedural instructions

The final instructions of x86prime are the instructions procedure calls. There is one significant difference between these and the similar x86 instructions. Instead of implicitly pushing and popping the return address to/from the stack, the instructions are give a register name in which the return address is handled. Thus

```
call p,d
```

makes a function call to label `p` and stores the current program counter in register `d`, while

```
ret s
```

returns from function call, by using the value in register `s` as the return program counter.

As an example a call/return to `procedure` will have to make sure to store the return address. By convention `primify` that converts an x86_64 program into an x86prime program uses `%r11` for the return address, but a programmer may choose any other register. Thus, when you make procedure

calls in x86prime, you will often see the following code. For the call `%r11` is added to store the return address:

```
call procedure,%r11
```

When entering a procedure `%r11` is pushed to the call stack and again popped from the stack before the return. This is done to ensure that any usage of `%r11` in the procedure body will not overwrite the return address.

```
procedure:
    subq $8, %rsp
    movq %r11, (%rsp)

    <procedure body>

    movq (%rsp), %r11
    addq $8, %rsp
    ret %r11
```

Note that x86prime does not have instructions for pushing and popping the stack, so this has to be done by two instructions.

Also, if `%r11` is not used inside the procedure body, these four instructions can be removed; however this is not done by the current translation.

Halting the machine

There is a special purpose instruction that halts execution of a x86prime program. This is

```
stop
```

It is not expected to do more than this.