- (przesunięcie 6) 17 to `VXOR`, który jako argumenty przyjmuje dwa rejestry – r_{dst} oraz r_{src}: $r_{dst} = 00$, a więc rejestrem docelowym jest R0. $r_{src} = 00$, a więc rejestrem źródłowym również jest R0 (czyli instrukcja ma na celu wyzerowanie rejestru R0). Dla porównania drugą instrukcją w programie był `vxor r0, r0`, co jest zgodne z odczytanym przez nas `VXOR R0, R0`.

Można więc przyjąć założenie, że stworzone przez nas makra są poprawne (przynajmniej jeśli chodzi o przetestowane instrukcje).

3.11. Emulator

Jak pisałem w poprzednim podrozdziale, pełne kody „kompilatora" oraz emulatora nie zostały zamieszczone w książce, ale są dostępne wraz z dodatkowym komentarzem pod adresem:

http://gynvael.coldwind.pl/book/vm/

Najprostsza maszyna wirtualna składa się zazwyczaj z czterech bazowych elementów:

- zestawu rejestrów;
- modułu pamięci;
- implementacji instrukcji;
- głównej pętli wykonującej.

W przypadku naszego wirtualnego komputera dodatkowymi elementami są zewnętrzne urządzenia, czyli PIT oraz konsola, z którymi komunikacja odbywa się za pośrednictwem systemu portów i przerwań, jednak analizę implementacji tych systemów pozostawiam czytelnikowi w ramach ćwiczenia.

Zaczynając od rejestrów, w przykładowej implementacji (stworzonej w języku Python) pojedynczy rejestr jest reprezentowany za pomocą klasy `VMGeneralPurposeRegister` (plik *vm_regs.py*) zdefiniowanej w następujący, bardzo prosty sposób:

```
class VMGeneralPurposeRegister(object):
  def __init__(self):
    self.v = 0
```

W głównej klasie maszyny wirtualnej – `VMInstance` (*vm.py*) – tworzona jest tablica rejestrów ogólnego przeznaczenia, a także aliasy (alternatywne zmienne z referencjami do odpowiednich rejestrów) na oba rejestry o podwójnych nazwach (SP oraz PC):

```
def __init__(self):
    self.r = [VMGeneralPurposeRegister() for _ in xrange(16)]
    self.sp = self.r[14]
    self.pc = self.r[15]
    ...
```

Alternatywnie rejestry mogłyby być zwykłymi zmiennymi, w tym wypadku jednak należałoby poradzić sobie ze stworzeniem aliasów w inny sposób.

Moduł pamięci został zaimplementowany w klasie `VMMemory`, która, oprócz zarządzania samym buforem pamięci, udostępnia również zestaw metod odczytujących bądź zapisujących dane, a przy okazji wykrywających ewentualne błędy (np. próbę odwołania do nieprawidłowego adresu itp.). Podstawowa część implementacji tej klasy wygląda następująco:

```
class VMMemory(object):
  def __init__(self):
    self._mem = bytearray(64 * 1024)

  def fetch_byte(self, addr):
    if addr < 0 or addr >= len(self._mem):
      return None

    return self._mem[addr]

  def store_byte(self, addr, value):
    if addr < 0 or addr >= len(self._mem):
      return False
    self._mem[addr] = value
    return True

  ...
```

Podobnie jak w przypadku rejestrów, obiekt tej klasy jest tworzony w konstruktorze głównej klasy maszyny wirtualnej:

```
 def __init__(self):
    ...
    self.mem = VMMemory()
```

Implementacja instrukcji znajduje się w pliku *vm_instr.py*; każda z nich zaimplementowana jest w oddzielnej funkcji, z których wszystkie przyjmują obiekt maszyny wirtualnej w pierwszym argumencie oraz tablicę z parametrami (w formie surowych, nieprzetworzonych bajtów) w drugim. Kod obsługi kilku przykładowych instrukcji przestawia się następująco:

```
def VMOV(vm, args):
  vm.reg(args[0]).v = vm.reg(args[1]).v

def VSET(vm, args):
  vm.reg(args[0]).v = to_dd(args[1:1 + 4])
```