W tabelach będę posługiwał się wspomnianymi oznaczeniami r_{dst}, r_{src}, oraz $imm_{8/16/32}$. Zacznijmy od zaproponowania instrukcji z pierwszej grupy, czyli od kilku prostych operacji służących do wczytywania wartości do rejestrów, kopiowania wartości między rejestrami oraz operowania na pamięci.

Tabela 1. Instrukcje kopiujące dane

Opkod (hex)	Mnemonik oraz parametry	Opis
00	`VMOV` r_{dst}, r_{src}	***move*** – przenieś Kopiuje wartość rejestru r_{src} do r_{dst} – odpowiednik wysokopoziomowego $r_{dst} = r_{src}$. Przykład skopiowania wartości R5 do R2: `VMOV R2, R5` Kod maszynowy[29]: `00 02 05`
01	`VSET` r_{dst}, imm_{32}	***set*** – ustaw Ustawia wartość rejestru r_{dst} na podaną stałą. Odpowiednik wysokopoziomowego $r_{dst} = imm_{32}$. Przykład ustawienia wartości rejestru R4 na 0x00001234: `VSET R4, 0x1234` Kod maszynowy: `01 04 34 12 00 00` Jak wspomniałem wcześniej, stała jest zapisana przy użyciu metody *Little Endian*, a więc najmniej znaczące bajty mają pierwszeństwo – stąd bajt 0x34 znajduje się na początku.
02	`VLD` r_{dst}, r_{src}	***load*** – wczytaj/załaduj

Kopiuje 32 bity danych z pamięci operacyjnej spod adresu wskazanego w rejestrze r_{src} do rejestru wskazanego w r_{dst}.

W języku C operację tę można by zapisać jako:

```
r_dst = *(uint32_t*)r_src;.
```

Przykład wczytania 32 bitów danych z pamięci operacyjnej spod adresu 0x1234 do rejestru R1:

```
VSET R2, 0x1234
VLD R1, R2
```

Kod maszynowy:

```
01 02 34 12 00 00
02 01 02
```

Należy zwrócić uwagę, że wczytanie 32 bitów (4 bajtów) spod adresu 0x1234 należy rozumieć jako wczytanie czterech kolejnych bajtów z adresów (kolejno): 0x1234, 0x1235, 0x1236 oraz 0x1237.

---

03 `VST r_dst, r_src`

***st**ore* – zapisz/zachowaj

Kopiuje 32 bity danych z rejestru r_{src} do pamięci operacyjnej pod adres wskazany w rejestrze r_{dst}.

W języku C operację tę można by zapisać jako:

```
*(uint32_t*)r_dst = r_src;
```

Przykład: (zapisanie wartości 0x12345678 pod adres 0x1234)

```
VSET R9, 0x1234
VSET R5, 0x12345678
VST R9, R5
```

Kod maszynowy:

```
01 09 34 12 00 00
01 05 78 56 34 12
03 09 05
```

---

04 `VLDB` r_{dst}, r_{src}

load byte – wczytaj/załaduj bajt

Kopiuje 8 bitów danych z pamięci operacyjnej spod adresu wskazanego w rejestrze r_{src}, do rejestru wskazanego w r_{dst}.

W języku C operację tę można by zapisać jako:

```
r_dst = *(uint8_t*)r_src;
```

Przykład wczytania 8 bitów z pamięci operacyjnej spod adresu 0x1234 do rejestru R1:

```
VSET R3, 0x1234
VLDB R1, R3
```

Kod maszynowy:

```
01 03 34 12 00 00
04 01 03
```

---

05 `VSTB` r_{dst}, r_{src}

store byte – zapisz/zachowaj bajt

Kopiuje dolnych 8 bitów danych z rejestru r_{src} do pamięci operacyjnej pod adres wskazany w rejestrze r_{dst}.

W języku C operację tę można by zapisać jako:

```
*(uint8_t*)r_dst = r_src;
```

Przykład zapisania bajtu 0x41 pod adres 0x1234:

```
VSET R1, 0x41
VSET R2, 0x1234
VSTB R2, R1
```

Kod maszynowy:

```
01 01 41 00 00 00
01 02 34 12 00 00
05 02 01
```

Drugą grupą są instrukcje arytmetyczno-logiczne. Tutaj w zasadzie wystarczą nam najprostsze operacje arytmetyczne, czyli dodawanie, odejmowanie, mnożenie, dzielenie z resztą oraz podstawowe operacje logiczne, czyli OR (alternatywa), AND (koniunkcja), XOR (alternatywa wykluczająca) i NOT (negacja). Dodam, że opkody tej grupy zacznę od numeru 0x10, tak by logicznie oddzielić od siebie grupy instrukcji za pomocą czterech najbardziej znaczących bitów.

Warto zwrócić uwagę, że na wielu architekturach występuje jeszcze jedna operacja – zmiana znaku, najczęściej pod postacią instrukcji NEG (*negate* – negacja). W naszym przykładzie ją pominąłem, ponieważ przedstawiony wcześniej zestaw instrukcji nie obsługuje liczb ujemnych (operujemy wyłącznie na liczbach naturalnych z zerem). Oczywiście i tak możliwe jest wykonanie dodawania i odejmowania na liczbach ujemnych w systemie U2, a więc zmianę znaku można uzyskać, korzystając z już dostępnych operacji (np. odejmując liczbę od zera lub wykonując negację bitową i dodając 1).

Tabela 2. Instrukcje arytmetyczno-logiczne

Opkod (hex)	Mnemonik oraz parametry	Opis

10	`VADD` r_{dst}, r_{src}	***add*** – dodaj
		
		Sumuje wartości rejestrów r_{src} z r_{dst} i zapisuje wynik w r_{dst}. Odpowiednik wysokopoziomowego r_{dst} += r_{src}.
		
		Przykład dodania 5 do 8:
		`VSET R1, 5`
		`VSET R2, 8`
		`VADD R2, R1`
		
		Kod maszynowy:
		`01 01 05`
		`01 02 08`
		`10 02 01`

11	`VSUB` r_{dst}, r_{src}	***sub****tract* – odejmij

Odejmuje wartość rejestru r_{src} od r_{dst} i zapisuje wynik w r_{dst}. Odpowiednik wysokopoziomowego `r_dst -= r_src`.

Przykład wyzerowania rejestru R1:
```
VSUB R1, R1
```

Kod maszynowy:
```
10 01 01
```

---

12 `VMUL r_dst, r_src`

***mul**tiply* – pomnóż

Mnoży wartość rejestru r_{src} przez wartość rejestru r_{dst} i zapisuje wynik w tym ostatnim. Odpowiednik wysokopoziomowego `r_dst *= r_src`.

Przykład podniesienia liczby w rejestrze R1 do kwadratu:
```
VMUL R1, R1
```

Kod maszynowy:
```
12 01 01
```

---

13 `VDIV r_dst, r_src`

***div**ide* – podziel

Dzieli wartość rejestru r_{dst} przez wartość rejestru r_{src} i zapisuje wynik w tym pierwszym. W przypadku, gdy w rejestrze r_{src} znajduje się liczba 0, generowane jest przerwanie 1 (`INT_DIVISION_ERROR`).
Odpowiednik wysokopoziomowego `r_dst /= r_src`.

Przykład dzielenia liczby w R1 przez 10:
```
VSET R2, 10
VDIV R1, R2
```

Kod maszynowy:
```
01 02 0A 00 00 00
13 01 02
```

14 `VMOD` r_{dst}, r_{src}

***mod**ulo* – reszta z dzielenia

Dzieli wartość rejestru r_{dst} przez wartość rejestru r_{src} i zapisuje resztę z dzielenia w tym pierwszym. W przypadku, gdy w rejestrze r_{src} znajduje się wartość 0, generowane jest przerwanie 1 (`INT_DIVISION_ERROR`). Odpowiednik wysokopoziomowego r_{dst} /= r_{src}.

Przykład uzyskania reszty z dzielenia liczby w R1 przez 10:

```
VSET R2, 10
VMOD R1, R2
```

Kod maszynowy:

```
01 02 0A 00 00 00
14 01 02
```

---

15 `VOR` r_{dst}, r_{src}

or – lub, alternatywa

Do rejestru r_{dst} zapisuje wynik alternatywy przeprowadzany na każdym bicie rejestrów z osobna (tzw. *bitwise OR*). Odpowiednik wysokopoziomowego r_{dst} |= r_{src}.

Przykład wyliczenia alternatywy wartości rejestrów R1 i R2:

```
VOR R1, R2
```

Kod maszynowy:

```
15 01 01
```

---

16 `VAND` r_{dst}, r_{src}

and – i, koniunkcja

Do rejestru r_{dst} zapisuje wynik koniunkcji przeprowadzonej na każdym bicie rejestrów z osobna (tzw. *bitwise AND*).

Odpowiednik wysokopoziomowego r_{dst} &= r_{src}.

Przykład nałożenia na rejestr R1 maski bitowej 0x0F:

```
VSET R2, 0x0F
VAND R1, R2
```

Kod maszynowy:

```
01 02 0F 00 00 00
16 01 02
```

---

17 `VXOR` r_{dst}, r_{src}

ex**clusive **or – alternatywa wykluczająca

Do rejestru r_{dst} zapisuje wynik alternatywy wykluczającej przeprowadzonej na każdym bicie rejestrów z osobna (tzw. *bitwise XOR*). Odpowiednik wysokopoziomowego r_{dst} ^= r_{src}.

Przykład wyzerowania rejestru R1:

```
VXOR R1, R1
```

Kod maszynowy:

```
17 01 01
```

---

18 `VNOT` r_{dst}

not – zaprzeczenie, dopełnienie bitowe

Zmienia stan wszystkich bitów wskazanego rejestru na przeciwny (tzw. *bitwise NOT*). Odpowiednik wysokopoziomowego r_{dst} = ~r_{dst}.

Przykład dopełnienia bitowego na rejestrze R5:

```
VNOT R5
```

Kod maszynowy:

```
18 05
```

19

***sh**ift **l**eft* – przesuń bity w lewo

	VSHL r_{dst}, r_{src}	Dokonuje przesunięcia w lewo bitów znajdujących się w rejestrze r_{dst} o liczbę pozycji podaną w rejestrze r_{src}. Odpowiednik wysokopoziomowego `` $r_{dst}$ <<= $r_{src}$ ``.
		
		Przykład pomnożenia wartości w rejestrze R1 przez 8 (a więc przesunięcia bitowego w lewo o 3 pozycje[30]:
		`VSET R2, 3`
		`VSHL R1, R2`
		
		Kod maszynowy:
		`01 02 03 00 00 00`
		`19 01 02`

1A	VSHR r_{dst}, r_{src}	***sh**ift **r**ight* – przesuń bity w prawo
		
		Dokonuje przesunięcia w prawo bitów znajdujących się w rejestrze r_{dst} o liczbę pozycji podaną w rejestrze r_{src}. Odpowiednik wysokopoziomowego `` $r_{dst}$ >>= $r_{src}$ ``.
		
		Przykład podzielenia wartości w rejestrze R1 przez 16 (a więc przesunięcia bitowego w prawo o 4 pozycje):
		`VSET R2, 4`
		`VSHR R1, R2`
		
		Kod maszynowy:
		`01 02 04 00 00 00`
		`1A 01 02`

Kolejna grupa to instrukcje porównania oraz skoki warunkowe (patrz tab. 3). Pozwalają one zaimplementować funkcjonalność identyczną z blokami i pętlami

warunkowymi w językach wysokopoziomowych (`if`, `while`, `for` itp.). Zazwyczaj wymagana jest przynajmniej para instrukcji porównania i skoku, choć często tych instrukcji jest dużo więcej – tak jest w przypadku bardziej skomplikowanych warunków lub występowania bloku `else`. Warto dodać, że znane z języków wysokopoziomowych operatory logiczne `&&` (AND logiczny) oraz `||` (OR logiczny) często są implementowane właśnie jako seria skoków warunkowych (patrz ramki „Skoki warunkowe a operatory logiczne [VERBOSE]" oraz „Weryfikacja [BEYOND]").

Skoki warunkowe a operatory logiczne [VERBOSE]

Rozważmy następujący przykładowy kod w języku C:

```
if (r0 == r1 && r2 == r3) {
  // Kod.
} else {
  // Inny kod.
}
```

Kod jest bardzo prosty, ale należy wskazać jedną bardzo istotną kwestię: w większości języków programowania warunki są testowane jedynie, jeśli jest to konieczne (chodzi o tzw. leniwe sprawdzanie warunków). Oznacza to, że w przypadku przedstawionego powyżej kodu porównanie `r2 == r3` nie będzie wykonane, jeśli pierwsze porównanie `r0 == r1` zwróci wartość `false`. Biorąc ten fakt pod uwagę, możemy przetłumaczyć powyższy kod na równoważny, korzystając z instrukcji `goto`, która jest w zasadzie tożsama z niskopoziomową instrukcją skoku, w tym z instrukcją VJMP zdefiniowaną w tabeli 5:

```
if (r0 != r1) {
  goto else_branch;
}

if (r2 != r3) {
  goto else_branch;
}
```

```
  // Kod.

  goto end_of_if;

else_branch:
  // Inny kod.

end_of_if:
```

Powyższą wersję można bez problemu przetłumaczyć na kod naszej maszyny wirtualnej (patrz instrukcje w tab. 3 oraz 5):

```
  VCMP r0, r1
  VJNE else_branch
  VCMP r2, r3
  VJNE else_branch
  ; Kod.
  VJMP end_of_if
else_branch:
  ; Inny kod.
end_of_if:
```

Podsumowując, z uwagi na zasadę działania operatorów logicznych instrukcje warunkowe są w praktyce tłumaczone na zestaw porównań i skoków warunkowych.

Weryfikacja [BEYOND]

Opisany w ramce „Skoki warunkowe a operatory logiczne [VERBOSE]" mechanizm można bardzo prosto zweryfikować, tworząc warunki zawierające wywołania funkcji (np. wypisujące wiadomości na standardowe wyjście), następnie skompilować i uruchomić kod, sprawdzając, które z funkcji zostały w praktyce wykonane. Alternatywna metoda weryfikacji, która nie wymaga uruchomienia kodu, polega na podejrzeniu wygenerowanego przez kompilator niskopoziomowego kodu, co demonstrują dwa zaprezentowane poniżej przykłady.

W pierwszym przypadku spójrzmy na kod napisany w języku Python 2.7, który definiuje funkcję `func`, a następnie ją disasembluje (tj. tłumaczy kod maszynowy, lub w tym wypadku bajtowy, na zapis mnemoniczny):

```
import dis

def func(a, b):
  if a == 5 and b == 10:
    print "Kod."
  else:
    print "Inny kod."

dis.dis(func)
```

Po uruchomieniu otrzymujemy następujący listing:

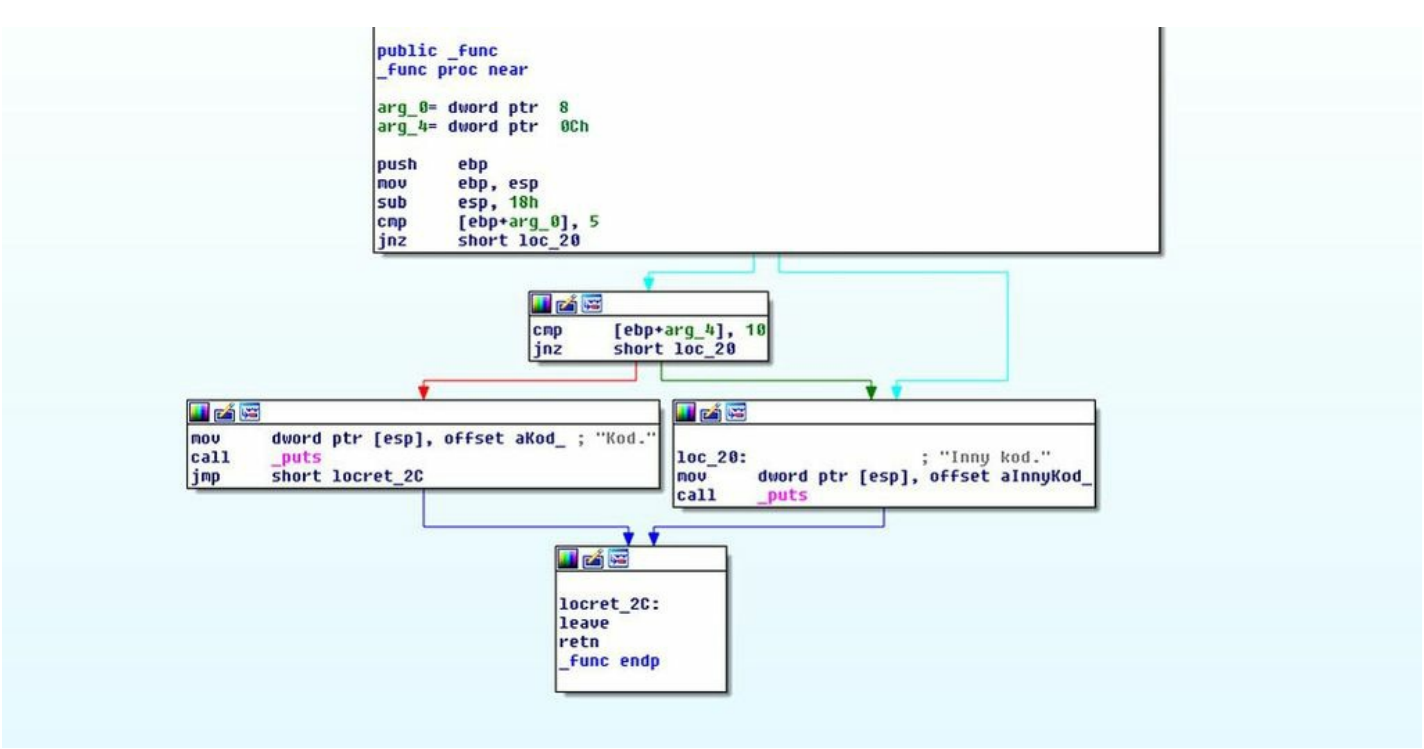
```
> test.py
  4           0 LOAD_FAST                0 (a)
              3 LOAD_CONST               1 (5)
              6 COMPARE_OP               2 (==)
              9 POP_JUMP_IF_FALSE       32
             12 LOAD_FAST                1 (b)
             15 LOAD_CONST               2 (10)
             18 COMPARE_OP               2 (==)
             21 POP_JUMP_IF_FALSE       32

  5          24 LOAD_CONST               3 ('Kod.')
             27 PRINT_ITEM
             28 PRINT_NEWLINE
             29 JUMP_FORWARD             5 (to 37)

  7     >>   32 LOAD_CONST               4 ('Inny kod.')
             35 PRINT_ITEM
             36 PRINT_NEWLINE
        >>   37 LOAD_CONST               0 (None)
             40 RETURN_VALUE
```

Analizując powyższy fragment (a w zasadzie jedynie tłumacząc kod z języka angielskiego na polski), można zaobserwować, że mamy do czynienia z dwoma skokami warunkowymi, które wykonają się w momencie niespełnienia jednego z warunków. Zgodnie z naszymi przewidywaniami operator logicznej koniunkcji w tym wypadku okazał się nie być „prawdziwym" operatorem, lecz zrealizowanym za pomocą skoków warunkowych.

Podobny eksperyment wykonajmy jeszcze dla języka C:

```
#include <stdio.h>
void func(int a, int b) {
  if (a == 5 && b == 10) {
    puts("Kod.");
  } else {
    puts("Inny kod.");
  }
}
```

Powyższy kod został skompilowany kompilatorem z rodziny MinGW-w64 GCC 4.6.2 do pliku obiektowego poleceniem `gcc -c test.c`, a następnie zdisasemblowany za pomocą programu IDA Pro. Funkcja `func` została przedstawiona w postaci grafu na rysunku 3. Analizując sam graf, można zaobserwować dwa rozgałęzienia związane ze skokami warunkowymi: pierwsze w okolicy porównania (instrukcja `cmp`) ze stałą 5, a drugie przy porównaniu ze stałą 10. Nasze założenie okazało się ponownie prawdziwe.



Rysunek 3. Funkcja func jako graf przedstawiony w programie IDA Pro

Tabela 3. Instrukcje porównania oraz skoki warunkowe

Opkod (hex)	Mnemonik oraz parametry	Opis
20	VCMP r_{dst}, r_{src}	***compare*** – porównaj Porównuje wartości rejestrów r_{dst} z r_{src} i zapisuje wynik porównania do rejestru FR (technicznie VCMP wykonuje odejmowanie bez zapisywania wyniku i ustawia flagę ZF, jeśli wynik jest równy zero, oraz CF, jeśli wynik jest mniejszy od zera; w obu przypadkach, jeśli warunek nie został spełniony, dana flaga jest zerowa). W parze ze skokiem warunkowym jest to odpowiednik wysokopoziomowej konstrukcji: if (r_{dst} *warunek* r_{src}) goto *cel*

Zarówno faktyczny warunek, jak i cel zależą od użytego skoku warunkowego.

Przykłady porównania wartości w rejestrach R1 i R2:

```
VCMP R1, R2
```

Kod maszynowy:

```
20 01 02
```

Patrz również przykłady zamieszczone przy opisie skoków warunkowych w dalszej części tabeli.

---

21 `VJZ imm16` / `VJE imm16`

***j**ump if **z**ero* – skocz, jeśli zero
***j**ump if **e**qual* – skocz, jeśli równe

Sprawdza, czy flaga ZF jest ustawiona – jeśli tak, rejestr PC jest zwiększany o imm16 (modulo 2^{16}). W innym przypadku skok nie jest wykonany i instrukcja nie przynosi żadnych efektów.
O ile parametrem w zapisie mnemonicznym jest adres docelowy, o tyle na poziomie kodu maszynowego imm16 musi być zapisany jako różnica pomiędzy adresem docelowym a adresem instrukcji bezpośrednio po skoku warunkowym. Przeliczeń pomiędzy parametrem relatywnego skoku a adresem docelowym wykonuje się za pomocą poniższych dwóch wzorów:

```
adres_docelowy = (adres_instrukcji_skoku +
3 + imm16) mod 216
imm16 = (adres_docelowy -
(adres_instrukcji_skoku + 3)) mod 216
```

Przykład skoku pod adres 0x30, jeśli wartości w rejestrach R1 i R2 są równe (zakładam, że adres instrukcji VJZ to 0x13):

```
VCMP R1, R2
VJZ 0x30
```

Kod maszynowy:
```
20 01 02
21 1A 00
```

---

22 `VJNZ imm16` / `VJNE imm16`

***j**ump if **n**ot **z**ero* – skocz, jeśli nie zero
***j**ump if **n**ot **e**qual* – skocz, jeśli nie równe

Sprawdza, czy flaga ZF jest wyzerowana – jeśli tak, wykonuje relatywny skok do wskazanego miejsca (wg schematu opisanego przy instrukcji VJZ).

---

23 `VJC imm16` / `VJB imm16`

***j**ump if **c**arry* – skocz, jeśli nastąpiło przeniesienie
***j**ump if **b**elow* – skocz, jeśli mniejsze

Sprawdza, czy flaga CF jest ustawiona – jeśli tak, wykonuje relatywny skok do wskazanego miejsca (wg schematu opisanego przy instrukcji VJZ).

---

24 `VJNC imm16` / `VJAE imm16`

***j**ump if **n**ot **c**arry* – skocz, jeśli nie nastąpiło przeniesienie
***j**ump if **a**bove or **e**qual* – skocz, jeśli większe lub równe

Sprawdza, czy flaga CF jest wyzerowana – jeśli tak, wykonuje relatywny skok do wskazanego miejsca (wg schematu opisanego przy instrukcji VJZ).

---

25 `VJBE imm16`

***j**ump if **b**elow or **e**qual* – skocz, jeśli mniejsze lub równe

Sprawdza, czy ustawiona jest flaga CF lub ZF (lub obie) – jeśli tak, wykonuje relatywny skok do wskazanego miejsca (wg schematu opisanego przy instrukcji VJZ).

---

26 `VJA imm16`

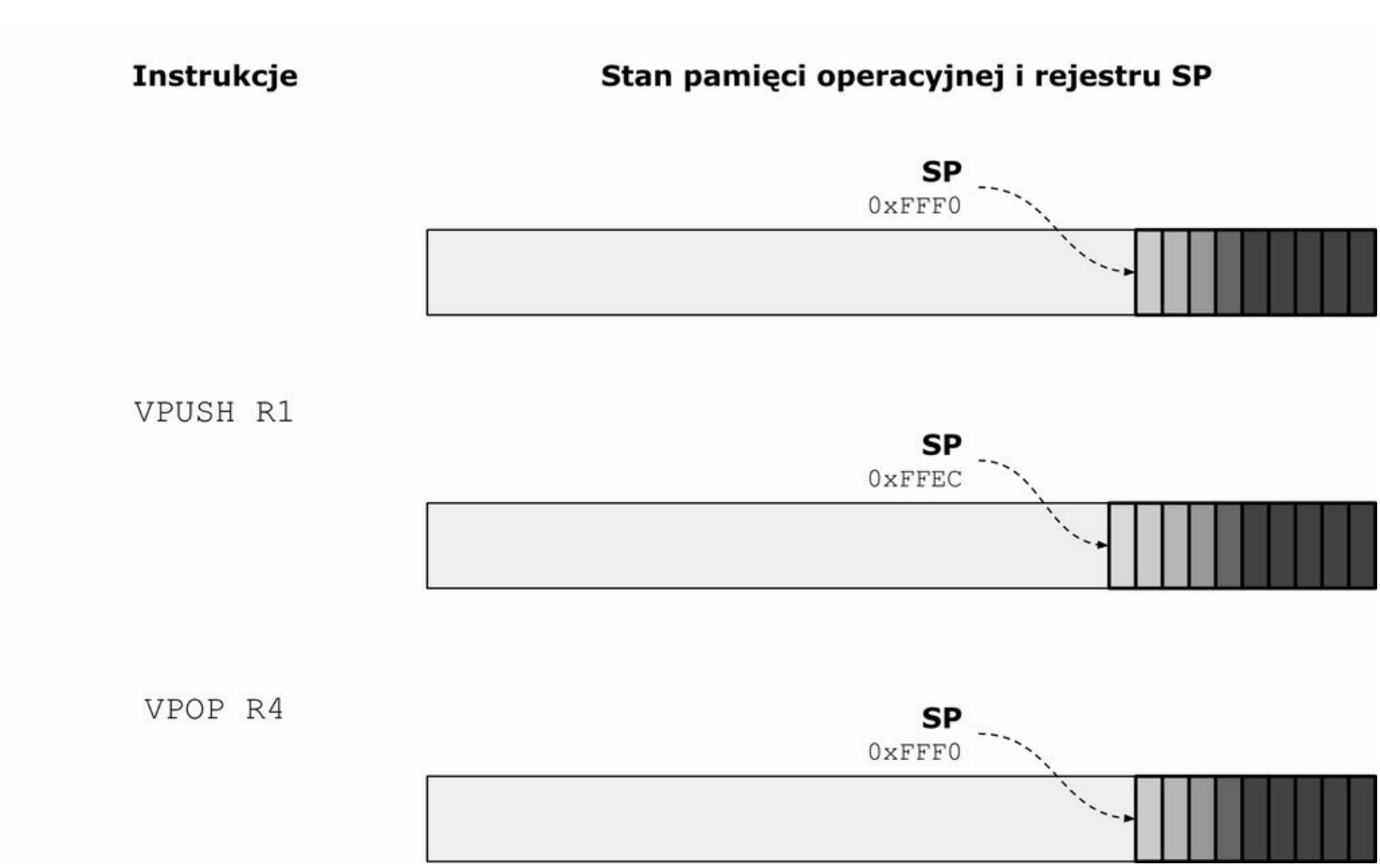
***j**ump if **a**bove* – skocz, jeśli większe

	Sprawdza, czy ustawiona jest flaga CF i czy jednocześnie flaga ZF jest wyzerowana – jeśli tak, wykonuje relatywny skok do wskazanego miejsca (wg schematu opisanego przy instrukcji VJZ).

Idąc dalej, kolejną grupą będą instrukcje operujące na stosie (patrz tab. 4). O ile w przypadku wirtualnych maszyn stosowych stos jest zazwyczaj dedykowaną, oddzielną strukturą danych, o tyle w przypadku większości prawdziwych (niewirtualnych) architektur stos jest po prostu fragmentem pamięci operacyjnej, który nie różni się niczym od innych regionów. Tak jest w przypadku architektur x86[31] czy ARM i tak też będzie w przypadku naszej architektury.

Zasada działania takiego stosu jest bardzo prosta: wszystkie elementy są tej samej wielkości, zazwyczaj powiązanej bezpośrednio z bitowością procesora. W naszym przypadku procesor jest 32-bitowy, więc każdy element na stosie będzie zajmował tyleż bitów (4 bajty). Od strony naszego CPU jedynym wyznacznikiem tego, gdzie stos faktycznie się znajduje, jest rejestr SP, w którym przechowywany jest adres wierzchołka stosu. Na rejestrze SP operują m.in. instrukcje `VPUSH` oraz `VPOP`, które, kolejno, umieszczają element na stosie i ustawiają SP, by na niego wskazywał, oraz zdejmują element ze stosu i ustawiają SP, by wskazywał na poprzedni element (patrz również rys. 4).

Inną kwestią, którą musimy rozważyć, jest pytanie: w którą stronę stos rośnie? Czy po umieszczeniu nowego elementu na stosie rejestr SP powinien być zwiększany o 4, czy też zmniejszany o 4? Odpowiedzią, która się nasuwa, jest „zwiększany o 4”, ale w praktyce, szczególnie w przypadku ograniczonej ilości pamięci, lepszym rozwiązaniem jest zmniejszanie wskaźnika stosu. W takim wypadku możemy powiedzieć, że stos ma swój początek (dno) na końcu pamięci, a więc będzie rósł w kierunku mniejszych adresów (początku pamięci). Dzięki temu trochę upraszczamy problem tego, gdzie umieścić stos w pamięci, tak by z jednej strony miał dużo miejsca, ale z drugiej zostawił go możliwie jak najwięcej dla reszty programu (np. implementacji sterty), a także minimalizujemy fragmentację pamięci[32].



Rysunek 4. Przykładowa zmiana rejestru SP podczas operowania na stosie

Tabela 4. Instrukcje operujące na stosie

Opkod (hex)	Mnemonik oraz parametry	Opis

30	`VPUSH` r_{src}	***push*** – odłóż (na stosie)
		
		Zmniejsza adres w rejestrze SP o 4, a następnie kopiuje 32 bity wartości z rejestru r_{src} pod wskazany przez SP adres pamięci.
		
		Przykład umieszczenia 8 bajtów zerowych na stosie:
		`VXOR R1, R1`
		`VPUSH R1`
		`VPUSH R1`
		
		Kod maszynowy:
		`17 01 01`

```
30 01
30 01
```

31	`VPOP` r_{dst}	***pop*** – zdejmij (ze stosu)
		
		Wczytuje wartość z pamięci operacyjnej spod adresu, na który wskazuje SP, do rejestru r_{dst}, a następnie zwiększa adres w rejestrze SP o 4. Przykład ściągnięcia wartości ze stosu do rejestru R5:
		`VPOP R5`
		
		Kod maszynowy:
		`31 05`

Przedostatnią grupą instrukcji są skoki bezwarunkowe. Choć mogłem opisać je wraz ze skokami warunkowymi, z uwagi na instrukcje z rodziny `VCALL` chciałem uczynić to dopiero po omówieniu stosu. Instrukcje `VCALL` oraz `VCALLR` służą do wywoływania funkcji – różnica pomiędzy nimi a zwykłymi skokami bezwarunkowymi (`VJMP`, `VJMPR`) polega na tym, że przy wywołaniu funkcji musi zostać zapamiętany adres powrotu. Istnieje kilka metod, by to osiągnąć, natomiast w naszym przypadku (podobnie jak ma to miejsce w architekturze x86) po prostu odłożymy wartość z rejestru PC na stos – czyli o instrukcji `VCALL` możemy myśleć jako o sekwencji pseudoinstrukcji `VPUSH PC+3` oraz `VJMP adres`. Dzięki temu, gdy będziemy chcieli wrócić do miejsca wywołania, będziemy mogli pobrać adres ze stosu i do niego skoczyć. W tym celu na liście umieścimy instrukcję `VRET`, która w zasadzie będzie odpowiednikiem sekwencji instrukcji `VPOP` r_{tmp} oraz `VJMPR` r_{tmp}.

Tabela 5. Skoki bezwarunkowe

Opkod (hex)	Mnemonik oraz parametry	Opis
40	`VJMP imm16`	***jump*** – skocz

Wykonuje relatywny skok do wskazanego miejsca (wg schematu opisanego przy instrukcji VJZ).
Odpowiednik `goto` z języków wyższego poziomu.

---

41 `VJMPR` r_{src}

***jump** to address from **r**egister* – skocz do adresu z rejestru

Wykonuje bezwzględny skok do wskazanego w rejestrze adresu. Technicznie kopiuje wartość z rejestru r_{src} (modulo 2^{16}) do rejestru PC.

Przykład skoku pod adres 0x1234:
```
VSET R1, 0x1234
VJMPR R1
```

Kod maszynowy:
```
01 01 34 12 00 00
41 01
```

---

42 `VCALL imm16`

call – wywołaj

Zapisuje adres kolejnej instrukcji (PC+3) na stosie, a następnie wykonuje relatywny skok do wskazanej lokalizacji (wg schematu opisanego przy instrukcji `VJZ`).

---

43 `VCALLR` r_{dst}

***call** an address from **r**egister* – wywołaj funkcję spod adresu z rejestru

Zapisuje adres kolejnej instrukcji (PC+2) na stosie, następnie wykonuje bezwzględny skok do wskazanego adresu (patrz również `VJMPR`).

---

44 `VRET`

***ret**urn* – powróć

Pobiera ze stosu adres i do niego skacze. Odpowiednik pseudoinstrukcji `VPOP PC` lub sekwencji `VPOP` r_{tmp} i `VJMPR` r_{tmp}.

Przykład wywołania funkcji i powrotu (zakładam, że adres instrukcji `VCALL` to `0x10`, a adres etykiety `func` to `0x40`):

```
VCALL func
...
func:
  VSET R1, 0x1234
  VRET
```

Kod maszynowy:

```
0x10: 42 2D 00
...
0x40: 01 01 34 12 00 00
      44
```

Ostatnia grupa instrukcji jest odmienna od przedstawionych i służy do komunikacji z zewnętrznymi urządzeniami oraz do sterowania zachowaniem procesora w wyjątkowych sytuacjach. Obie te kwestie są omówione dokładniej w dalszej części niniejszego rozdziału.

Tabela 6. Dodatkowe instrukcje sterujące

Opkod (hex)	Mnemonik oraz parametry	Opis

F0	`VCRL` imm_{16}, r_{src}	***c**ontrol **r**egister **l**oad* – wczytaj rejestr sterowania
		
		Kopiuje wartość rejestru r_{src} do specjalnego rejestru sterującego o numerze imm_{16}. W przypadku, gdy rejestr specjalny nie istnieje, zostanie wygenerowany wyjątek 2 (`INT_GENERAL_ERROR`).
		
		Przykład ustawienia rejestru specjalnego 0x110 na wartość 1:
		`VSET R0, 1`
		`VCRL 0x110, R0`

Kod maszynowy:
```
01 00 01 00 00 00
F0 00 10 01
```

F1 — `VCRS imm16, rdst`

***c**ontrol **r**egister **s**tore* – zachowaj rejestr sterowania

Kopiuje wartość ze specjalnego rejestru sterującego o numerze imm_{16} do rejestru docelowego r_{dst}. W przypadku, gdy rejestr specjalny nie istnieje, zostanie wygenerowany wyjątek 2 (`INT_GENERAL_ERROR`).

F2 — `VOUTB imm8, rsrc`

***out**put **b**yte* – zapisz bajt na wyjście

Wysyła dolny bajt z rejestru r_{src} do urządzenia (portu) wskazanego przez imm_8.

Przykład wysłania litery "A" (kod 0x41) na konsolę:
```
VSET R0, 0x41
VOUTB 0x20, R0
```

Kod maszynowy:
```
01 00 41 00 00 00
F2 00 20
```

F3 — `VINB imm8, rdst`

***in**put **b**yte* – wczytaj bajt z wejścia

Odbiera dostępny bajt od wskazanego przez imm_8 urządzenia (portu) i zapisuje go do rejestru r_{dst}. W zależności od urządzenia praca procesora może zostać wstrzymana aż do pojawienia się bajtu danych.

Przykład odebrania bajtu z konsoli:
```
VINB 0x20, R0
```

Kod maszynowy:
```
F3 00 20
```

F4 — `VIRET`

***interrupt **ret**urn* – powróć z obsługi przerwania

		Przywraca stan zachowanych na stosie rejestrów, w tym rejestru PC, tym samym wracając do stanu i miejsca wykonania, w którym nastąpiło przerwanie.
FF	VOFF	*power **off*** – wyłącz maszynę Przerywa działanie maszyny wirtualnej.

Mając gotowy zestaw instrukcji, możemy omówić kolejne elementy systemu, a następnie przejść do implementacji samej maszyny wirtualnej.

3.5. Pamięć operacyjna

O pamięci operacyjnej w naszym wypadku można myśleć jako o tablicy bajtów (tj. wartości naturalnych z zakresu 0 do 255 włącznie) o rozmiarze 65536 elementów (indeksowanych od 0 do 65535). Indeksy tej tablicy są jednocześnie adresami w pamięci maszyny wirtualnej, co ułatwi implementację.

Zanim przejdziemy do następnego punktu, warto zwrócić uwagę na różnicę między pamięcią operacyjną a pamięcią fizyczną: pamięć operacyjna to pamięć, do której procesor może się bezpośrednio odwołać. W przypadku naszej maszyny mamy tylko jeden rodzaj pamięci – pamięć RAM o wielkości 64 kB (co czyni z niego naszą pamięć operacyjną), niemniej jednak prawdziwe komputery często posiadają kilka rodzajów pamięci, które mogą być bezpośrednio dostępne dla procesora. Idealnym przykładem jest VRAM (*Video Random Access Memory*, czyli pamięć karty graficznej), której fragment lub całość jest zazwyczaj podmapowana w przestrzeni adresowej procesora. Na przykład w starszych procesorach z rodziny x86 w momencie startu systemu operacyjnego pod adresami 0xB8000–0xBFFFF (włącznie) procesor „widział” pamięć karty graficznej, a konkretniej tzw. bufor tekstowy (a co za tym idzie, pamięć RAM pod tymi adresami była niedostępna). Za inny przykład może posłużyć pamięć podręczna procesora (*cache*), a także pamięć karty sieciowej itp.