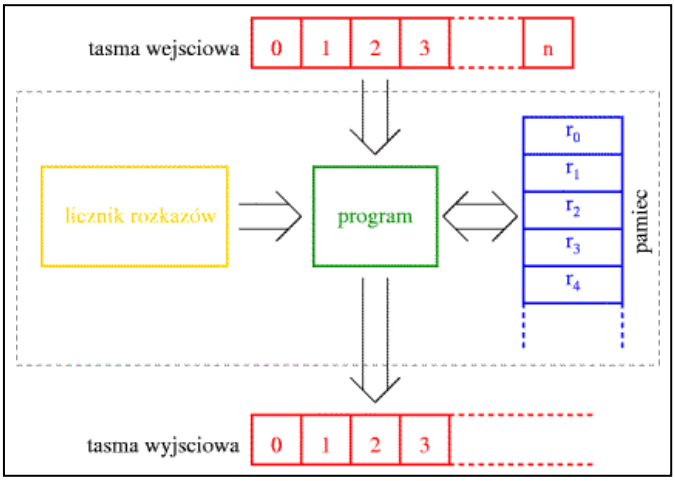
Podstawy działania i programowania procesorów

Opracowanie: Andrzej Bożek,
Politechnika Rzeszowska,
Katedra Informatyki i Automatyki,
Rzeszów, 2010.

Celem ćwiczenia jest nabycie podstawowej wiedzy na temat funkcjonowania procesorów i zasad ich programowania. W ćwiczeniu wykorzystuje się na emulator maszyny RAM [1], stanowiącej jeden z teoretycznych modeli obliczeń, którą można jednocześnie uznać za prosty, uogólniony model procesora. Emulator maszyny RAM programowany jest przy użyciu zbioru instrukcji asemblerowych o postaci zbliżonej do instrukcji stosowanych w rzeczywistych procesorach.

1. Architektura emulatora maszyny RAM

Struktura emulatora maszyny RAM została przedstawiona na rysunku 1.



Rys. 1. Struktura emulatora maszyny RAM [1]

W strukturze wyróżnić można pięć elementów:

1. **Program.** Jest to blok reprezentujący kod programu realizowanego przez maszynę, który stanowi zbiór rozkazów do wykonania. Maszyna RAM nie może modyfikować, ani w sposób bezpośrednio odczytywać jego zawartości.
2. **Licznik rozkazów.** Wskazuje instrukcję, która ma zostać wykonana w następnym cyklu maszynowym.
3. **Pamięć (zespół rejestrów).** Uporządkowany zbiór nieskończonej liczby rejestrów, mogących przechowywać dowolnie duże liczby całkowite. Pierwszy z rejestrów (**r0**), zwany **akumulatorem** lub **sumatorem** jest wyróżniony pod względem funkcjonalnym i stanowi domyślny argument dla większości instrukcji.
4. **Taśma wejściowa.** Jest źródłem danych wejściowych. Posiada nieskończoną długość. Może być tylko odczytywana. Pozwala na dostęp sekwencyjny, to znaczy dane są odczytywane kolejno według porządku umieszczenia ich na taśmie, bez możliwości przewijania.
5. **Taśma wyjściowa.** Jest miejscem zapisu danych wyjściowych. Posiada nieskończoną długość. Może być tylko zapisywana. Pozwala na dostęp sekwencyjny, to znaczy dane są zapisywane na taśmie kolejno, bez możliwości przewijania.

Maszyna RAM jest modelem obliczeń i jej emulator można uznać za symulator prostego procesora. Z uwagi na to, że kod programu wykonywanego przez maszynę RAM o strukturze przedstawionej na rysunku 1 jest całkowicie odseparowany od pamięci danych (zespołu rejestrów), rozważany emulator maszyny RAM odpowiada procesorowi o architekturze **harwardzkiej**.

Podstawowe różnice pomiędzy maszyną RAM i rzeczywistym procesorem są następujące:

1. W strukturze maszyny RAM, ze względu na jej abstrakcyjny charakter, nie uwzględnia się elementów sprzętowych, związanych z fizyczną realizacją obliczeń, takich jak budowa jednostki arytmetyczno-logicznej (ALU), szerokość szyn komunikacyjnych, częstotliwość taktowania itp. W rzeczywistym procesorze za każdą funkcjonalność odpowiadają określone składniki sprzętowe.
2. W rzeczywistym procesorze wszystkie zasoby, w szczególności pamięć, są ograniczone. Nie istnieje procesor o nieskończonej liczbie rejestrów lub nieograniczonej przestrzeni danych wejściowych i wyjściowych.
3. W rzeczywistych procesorach istnieją co najmniej dwa wyróżnione obszary pamięci, używane do przechowywania danych tymczasowych w toku obliczeń: obszar rejestrów oraz obszar pamięci operacyjnej. W strukturze maszyny RAM odpowiada im jeden element w postaci zespołu rejestrów.

Jeżeli nie rozważa się konkretnego modelu procesora, a jedynie ogólną zasadę jego działania i programowania, wymienione wyżej różnice nie maja istotnego znaczenia.

2. Instrukcje i składnia kodu

Emulator maszyny RAM rozpoznaje 12 rodzajów instrukcji. Pojedyncza instrukcja ma postać

mnemonik [*operand*]

i składa się z dwóch elementów. Pierwszy element, ***mnemonik***, jest kilkuliterowym skrótem, określającym wykonywany rozkaz, np. **add** (dodaj), **sub** (*substract*, odejmij), **mult** (*multiply*, pomnóż) itd. Drugi element, ***operand***, jest symbolem argumentu, którego dotyczy instrukcja. Operand jest składnikiem opcjonalnym instrukcji, gdyż nie towarzyszy wszystkim mnemonikom. W praktyce może być pominięty jednak tylko w przypadku instrukcji **halt**.

W pamięci programu mnemoniki zastępowane są odpowiednimi wartościami liczbowymi, zwanymi kodami rozkazów. Język programowania reprezentujący poszczególne instrukcje procesora za pomocą skrótów wyrazowych (mnemoników) nazywany jest **asemblerem**. Asembler umożliwia tworzenie kodu na najniższym poziomie (na poziomie rozkazów procesora), które jest efektywnie równoważne programowaniu w języku maszynowym (będącym ciągiem liczb stanowiących kod programu), ale jest bardziej przystępne i czytelne dzięki użyciu symboli rozkazów (mnemoników) i argumentów (operandów).

Zestaw instrukcji emulatora maszyny RAM został przedstawiony w tabeli 1.

Tab. 1. Zestaw instrukcji emulatora maszyny RAM

Nr	*mnem.* [*op.*]	Działanie	Opis
1	load =i	r0 := i	Wpisuje liczbę *i* do sumatora
	load p	r0 := rp	Wpisuje wartość z rejestru *rp* do sumatora
	load *p	r0 := r[rp]	Wpisuje wartość z rejestru o numerze zawartym w rejestrze *rp* do sumatora
2	store p	rp := r0	Wpisuje wartość z sumatora do rejestru *rp*
	store *p	r[rp] := r0	Wpisuje wartość z sumatora do rejestru o numerze zawartym w rejestrze *rp*
3	add =i	r0 := r0 + i	Dodaje do sumatora wartość *i*
	add p	r0 := r0 + rp	Dodaje do sumatora wartość z rejestru *rp*
	add *p	r0 := r0 + r[rp]	Dodaje do sumatora wartość z rejestru o numerze zawartym w rejestrze *rp*
4	sub =i	r0 := r0 – i	Odejmuje od sumatora wartość *i*
	sub p	r0 := r0 – rp	Odejmuje od sumatora wartość z rejestru *rp*
	sub *p	r0 := r0 – r[rp]	Odejmuje od sumatora wartość z rejestru o numerze zawartym w rejestrze *rp*
5	mult =i	r0 := r0 * i	Mnoży zawartość sumatora przez wartość *i*
	mult p	r0 := r0 * rp	Mnoży zawartość sumatora przez wartość z rejestru *rp*
	mult *p	r0 := r0 * r[rp]	Mnoży zawartość sumatora przez wartość z rejestru o numerze zawartym w rejestrze *rp*
6	div =i	r0 := floor(r0 / i)	Dzieli całkowitoliczbowo zawartość sumatora przez wartość *i*
	div p	r0 := floor(r0 / rp)	Dzieli całkowitoliczbowo zawartość sumatora przez wartość z rejestru *rp*
	div *p	r0 := floor(r0 / r[rp])	Dzieli całkowitoliczbowo zawartość sumatora przez wartość z rejestru o numerze zawartym w rejestrze *rp*
7	read p	rp := TWE	Wczytuje kolejną wartość z taśmy wejściowej do rejestru *rp*
	read *p	r[rp] := TWE	Wczytuje kolejną wartość z taśmy wejściowej do rejestru o numerze zawartym w rejestrze *rp*
8	write =i	TWY ← i	Zapisuje na taśmie wyjściowej wartość *i*
	write p	TWY ← rp	Zapisuje na taśmie wyjściowej wartość z rejestru *rp*
	write *p	TWY ← r[rp]	Zapisuje na taśmie wyjściowej wartość z rejestru o numerze zawartym w rejestrze *rp*
9	jump etykieta	skocz do etykiety	Skacze bezwarunkowo do miejsca w programie wskazanego etykietą
10	jzero etykieta	skocz do etykiety gdy r0 = 0	Skacze warunkowo do miejsca w programie wskazanego etykietą, gdy wartość sumatora jest zerowa
11	jgtz etykieta	skocz do etykiety gdy r0 > 0	Skacze warunkowo do miejsca w programie wskazanego etykietą, gdy wartość sumatora jest dodatnia
12	halt	zatrzymaj program	

Wszystkie linie programu asemblerowego dla emulatora maszyny RAM mają postać:

```
numer_linii etykieta: mnemonik [operand] # komentarz
```

Linia programu składa się z czterech kolejno po sobie następujących elementów:

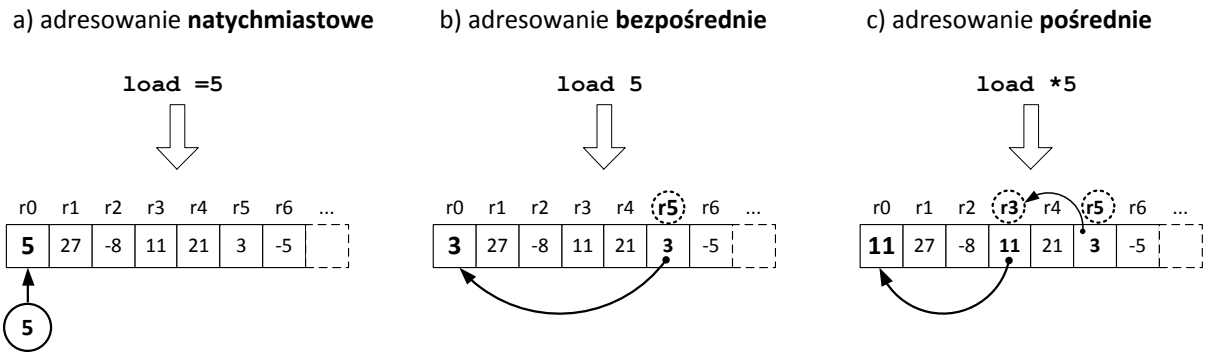
1. **Numer linii** – liczba całkowita dodatnia. Numer linii pełni rolę pomocniczą dla programisty i jest ignorowany przez kompilator.
2. **Etykieta** – dowolny ciąg znaków złożony z małych i dużych liter alfabetu angielskiego oraz z cyfr, którego pierwszym znakiem nie może być cyfra, a zakończony jest znakiem dwukropka. Nazwy etykiet nie mogą się powtarzać. Zadaniem etykiet jest wskazywanie miejsc, do których wykonany zostanie skok programu w przypadku użycia instrukcji *jump*, *jzero* lub *jgtz*.
3. **Instrukcja** – posiada omówioną wcześniej składnię *mnemonik* [*operand*]. Jeśli jest obecna w danej linii programu, stanowi jej zasadniczy składnik, definiujący rozkaz do wykonania.
4. **Komentarz** – rozpoczyna się symbolem # i rozciąga się do końca linii. Jest dowolnym ciągiem znaków, który podobnie jak numer linii pełni rolę pomocniczą dla programisty i jest ignorowany przez kompilator.

Uwagi dotyczące składni programu:

1. Zgodnie z powyżej podanymi definicjami, tylko dwa składniki w każdej linii kodu mają znaczenie dla kompilatora i wpływają na działanie programu, tj. etykiety oraz instrukcje. Numery linii oraz komentarze są przez kompilator pomijane, służą jako elementy pomocnicze dla programisty, dzięki którym kod może stać się bardziej czytelny.
2. Każdy z czterech składników linii programu (numer/etykieta/instrukcja/komentarz) jest opcjonalny i może zostać pominięty niezależnie od pozostałych składników, pod warunkiem, że kolejność pozostałych składników zostanie zachowana. Jeżeli w linii programu nieobecna jest ani etykieta, ani instrukcja, linia traktowana jest jako pusta i ignorowana podczas kompilacji. Jeżeli w linii programu obecna jest etykieta, ale brak instrukcji, linia traktowana jest jako instrukcja pusta (NOP, *No OPeration*), której wykonanie nie wprowadza żadnych zmian w stanie programu.
3. W zapisie mnemoników (tab. 1) wielkość liter jest ignorowana. Zatem np. zapisy Load 3, LOAD 3. lOaD 3 itd., są równoważne.

3. Tryby adresowania

Tryb adresowania oznacza sposób przekazywania argumentu (wartości liczbowej) do obliczeń w ramach danej instrukcji. Im więcej trybów adresowania zapewnia procesor, tym bardziej elastyczne i efektywne może być jego użycie. Nowoczesne procesory udostępniają bardzo liczne tryby adresowania. Podstawowy zbiór trybów adresowania, praktycznie nieodzowny w każdym, nawet najprostszym procesorze, składa się z trzech trybów: **natychmiastowego**, **bezpośredniego** i **pośredniego** (rys. 1).



Rys. 1. Tryby adresowania: a) natychmiastowy, b) bezpośredni, c) pośredni

Wymienione trzy tryby obsługuje także symulator maszyny RAM:

1. **Tryb natychmiastowy.** W trybie natychmiastowym (rys. 1a) wartość podana jako operand instrukcji pełni rolę wartości docelowej dla rozkazu. Stosownie do typu rozkazu, wartość ta bierze udział w obliczeniach, jest zapisywana w określonej lokalizacji, wchodzi w skład sprawdzanego warunku itp. Natychmiastowy tryb adresowania pozwala na wprowadzanie do kodu programu asemblerowego wartości stałych, znanych na etapie tworzenia programu. Zastosowanie trybu adresowania natychmiastowego w asemblerze jest odpowiednikiem użycia stałej lub przypisania stałej wartości początkowej do zmiennej w języku wysokiego poziomu.
2. **Tryb bezpośredni.** W tym trybie (rys. 1b) operand instrukcji reprezentuje numer rejestru, którego zawartość zostanie wykorzystana jako wartość docelowa dla rozkazu. Dzięki temu, liczne instrukcje programu mogą wielokrotnie odwoływać się do wybranego rejestru, zapisując lub odczytując jego wartość i w ten sposób wymieniając informacje między sobą. Tryb ten jest odpowiednikiem użycia zmiennych w języku wysokiego poziomu.
3. **Tryb pośredni.** W trybie pośrednim (rys. 1c) operand instrukcji reprezentuje numer rejestru, którego zawartość interpretowana jest jako numer rejestru docelowego, zawierającego wartość dla rozkazu. W ten sposób możliwy jest dostęp do rejestrów, których lokalizacja (numery) wyznaczone zostaną dopiero w trakcie wykonywania programu. Operacje realizowane w asemblerze z wykorzystaniem adresowania pośredniego odpowiadają w językach wysokiego poziomu zastosowaniu wskaźników, referencji i tablic.

4. Wyjątki procesora

Praktycznie niemożliwe jest takie zaprojektowanie zbioru instrukcji procesora, aby dowolne sekwencje rozkazów z dowolnymi wartościami danych zawsze mogły być uznane za poprawne i jednoznaczne w kontekście operacji wykonywanych przez procesor. Typowym przykładem jest operacja dzielenia przez zero. Z matematycznego punku widzenia działanie takie jest „zabronione”, tzn. nie można mu przypisać żadnego wyniku liczbowego. Inne niedozwolone działania mogą dotyczyć próby zaadresowania niedostępnego obszaru pamięci, skoku programu pod nieistniejący adres itp.

W przypadku niektórych procesorów o prostych architekturach, wykonanie zabronionego działania powoduje ściśle zdefiniowany efekt (np. wynik dzielenia przez 0 wyniesie 0) nie zakłócając sekwencji kolejnych rozkazów lub przeciwnie, wykonanie niedozwolonego działania skutkuje nieprzewidywalnym rezultatem (np. skok pod niedozwolony adres powoduje przyjęcie przypadkowej wartości przez licznik rozkazów). W przypadku architektów bardziej zaawansowanych, w tym procesorów przeznaczonych dla

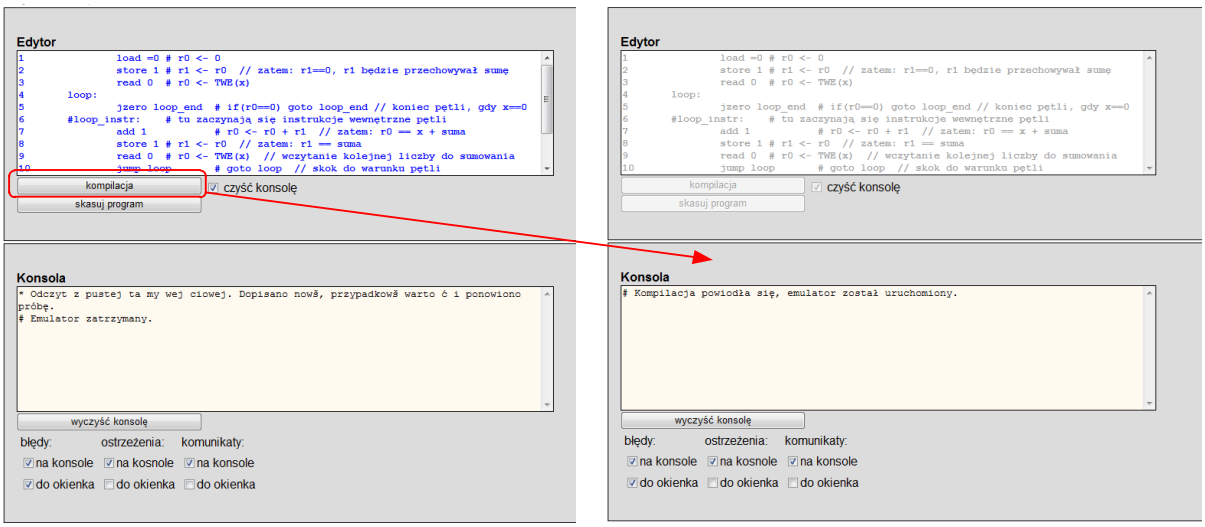
komputerów PC, istnieje możliwość dokładnego zaprogramowania reakcji procesora na zdarzenia nieprzewidziane dla toku normalnego wykonywania programu. Zdarzenia te nazywane są w takim wypadku **wyjątkami procesora** (*exceptions*). W chwili wystąpienia wyjątku przerywany jest normalny tok wykonywania instrukcji i następuje skok do specjalnie przygotowanego podprogramu, zwanego procedurą obsługi wyjątku. Procedura ta może podjąć próby wyprowadzenia procesora ze stanu niedozwolonego, a jeżeli jest to niemożliwe, zakończyć/zrestartować wykonywanie programu lub systemu operacyjnego.

W emulatorze maszyny RAM także działa mechanizm detekcji operacji niedozwolonych. Nie ma jednak możliwości programowania obsługi sytuacji wyjątkowych. W wyniku próby wykonania niedozwolonej operacji następuje zatrzymanie pracy emulatora i wyświetlenie informacji o błędzie.

5. Obsługa emulatora

Oryginalna dokumentacja [1] szczegółowo przedstawia zasady obsługi emulatora. Poniżej zostaną omówione w punktach czynności, które należy wykonać, aby uruchomić analizowane w ćwiczeniu programy i prześledzić przebieg ich działania instrukcja po instrukcji.

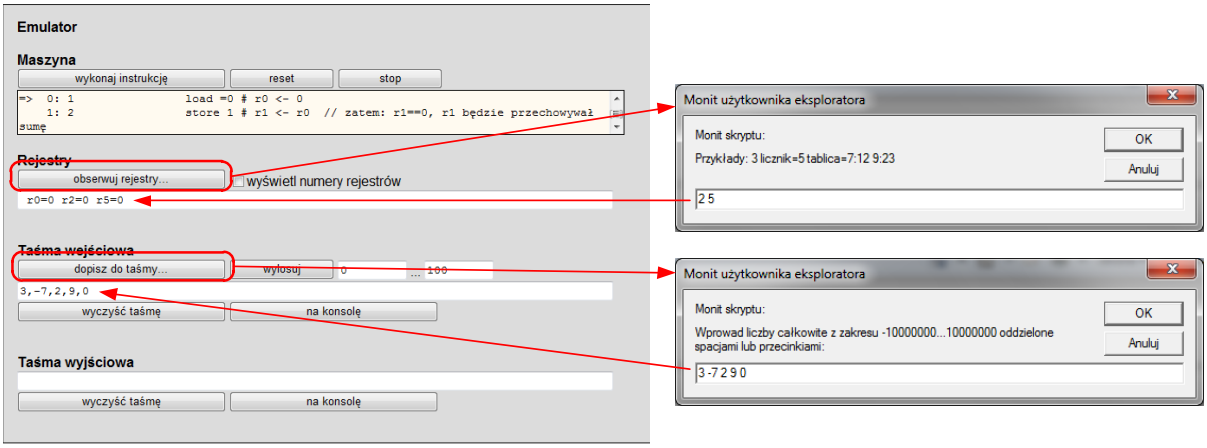
1. Kod programu wprowadzić do okna edytora (tekst można edytować bezpośrednio oraz kopiować z innych lokalizacji). Po wprowadzeniu kodu nacisnąć przycisk *kompilacja*. Jeżeli kompilacja przebiegnie poprawnie (rys. 2), odpowiedni komunikat zostanie wyświetlony w oknie konsoli, elementy wejściowe okna edytora zostaną zablokowane, natomiast odblokowane zostaną przyciski okna emulatora – program będzie gotowy do symulacji. Jeśli kompilacja się nie powiedzie, wyświetlone zostanie wyskakujące okno, informujące o błędzie. Po zamknięciu tego okna, edytor będzie nadal aktywny, umożliwiając usunięcie błędów programu.



Rys. 2. Wprowadzanie i kompilacja kodu programu

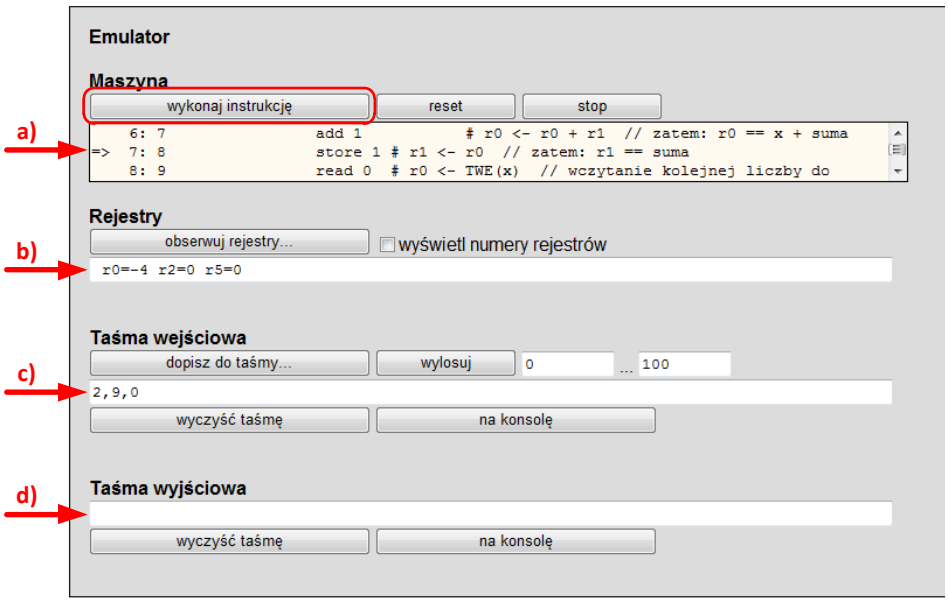
2. W oknie emulatora wcisnąć przycisk *obserwuj rejestry…* Otwarte zostanie okno dialogowe, pozwalające wprowadzić numery rejestrów, których wartości będą wyświetlane w celu obserwacji. Wprowadzane numery należy oddzielać przecinkami lub spacjami, nie trzeba wpisywać numeru 0, ponieważ zawartość sumatora jest wyświetlana zawsze (rys. 3).

3. W oknie emulatora, w sekcji *Taśma wejściowa*, wcisnąć przycisk *dopisz do taśmy…* Otwarte zostanie okno dialogowe, pozwalające edytować zawartość taśmy wejściowej. Wprowadzić zawartość taśmy wejściowej właściwą w danym przypadku, kolejne liczby należy oddzielać przecinkami lub spacjami (rys. 3).



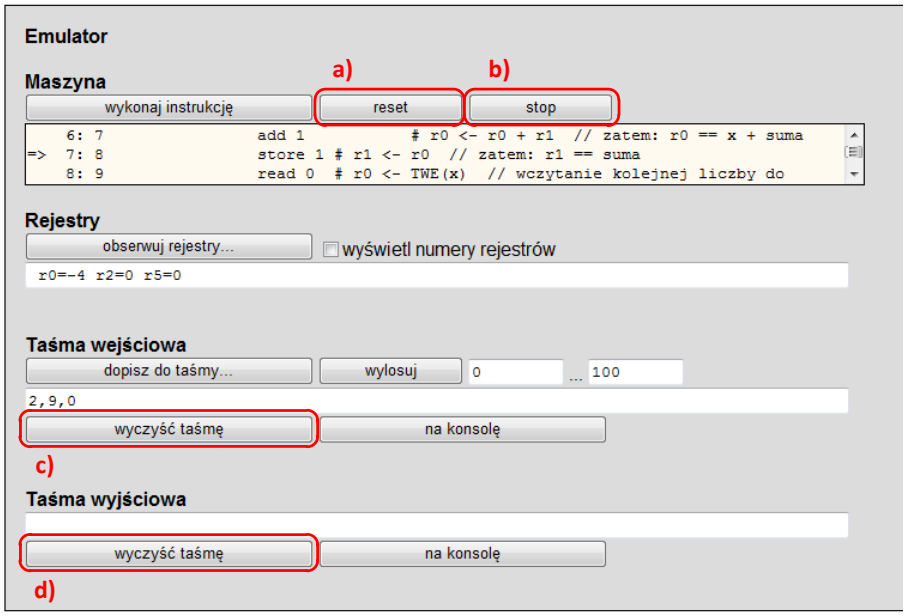
Rys. 3. Konfiguracja emulatora

4. Program wykonywać w trybie krokowym. Każde wciśnięcie przycisku *wykonaj instrukcję* (rys. 4) wykonuje jedną instrukcję kodu. Instrukcja przeznaczona do wykonania w najbliższym kroku wskazywana jest przez symbol strzałki => w oknie poniżej przycisku. Każdorazowe wykonanie instrukcji powoduje (rys. 4):
   a) Aktualizację instrukcji wybranej do wykonania w kolejnym kroku.
   b) Ewentualne zmiany zawartości rejestrów.
   c) Ewentualne pobranie wartości z taśmy wejściowej.
   d) Ewentualny zapis wartości na taśmę wyjściową.
5. Należy obserwować, czy wykonywanie poszczególnych instrukcji w oczekiwany sposób wpływa na stan programu oraz, czy po zakończeniu wykonywania programu (osiągnięcie rozkazu *halt*) zawartość taśmy wyjściowej będzie poprawna.



Rys. 4. Krokowe wykonywanie programu

6. Na każdym etapie krokowej symulacji programu można wykonać następujące operacje (rys. 5):
   a) Zresetować wykonywanie program (przycisk *reset*), co powoduje powrót licznika rozkazów do wartości początkowej, tj. rozpoczęcie symulacji programu od nowa.
   b) Zatrzymać wykonywanie programu (przycisk *stop*), co powoduje dezaktywację okna symulatora i uaktywnienie okna edytora, pozwalając na modyfikację programu lub wprowadzenie nowego kodu.
   c) Wyczyścić taśmę wejściową.
   d) Wyczyścić taśmę wyjściową.



Rys. 5. Działania wykonywane w oknie emulatora

6. Przykładowe programy

Asembler jest językiem programowania imperatywnego. Program asemblerowy określa algorytm pracy procesora za pomocą ciągu instrukcji, definiujących krok po kroku wykonywane działania. W ogólnym ujęciu program asemblerowy jest tworzony na podobnych zasadach jak program języka C lub Pascal – określa się sekwencję działań do wykonania, z możliwością rozgałęziania i zapętlania przebiegu instrukcji. Instrukcje asemblera, odpowiadające rozkazom procesora, reprezentują bardziej elementarne działania, niż języki programowania wysokiego poziomu. Z tego powodu, liczba instrukcji asemblerowych w programie realizującym dany algorytm jest najczęściej kilkakrotnie większa od liczby instrukcji programu o tym samym działaniu, zapisanego w języku C lub Pascal.

Dalej przytoczone zostały przykładowe programy asemblerowe dla emulatora maszyny RAM. Programy demonstrują realizację podstawowych operacji arytmetycznych, użycie różnych trybów adresowania, obsługę warunkowego przepływu sterowania oraz pętli programowych.

Przyjęto następujące konwencje:

1. Komentarze w kodzie asemblerowym, zapisane bezpośrednio po znaku #, dotyczą działań wykonywanych na poziomie maszyny RAM (np. przesłanie wartości między

rejestrami, odczyt z taśmy, skok itd.). Jeżeli potrzebny jest także komentarz dodatkowy, uszczegóławiający działanie instrukcji lub przedstawiający je na wyższym poziomie, następuje on w dalszej kolejności, po znaku podwójnego ukośnika //.

2. Taśma wejściowa oznaczana jest skrótem TWE, a taśma wyjściowa skrótem TWY. Jeżeli w danej instrukcji wartość wczytywana z taśmy lub na nią zapisywana posiada w kontekście realizowanego programu określony symbol (*a*, *b*, *x* itp.), dla lepszego sprecyzowania działania instrukcji symbol ten uwzględniany jest w komentarzu, poprzez zapis postaci TWE(a), TWY(x) itd.
3. Wszystkie programy przykładowe posiadają numerowane linie kodu. W opisach działania programów odwołania do numerów linii realizowane są przy użyciu nawiasów ostrych, np. <7> - pojedyncza linia, <2,5,11> - zbiór linii programu, <3-7> - przedział (blok) linii programu.

6.1. Suma dwóch liczb

Program dodaje dwie liczby, będące dwoma kolejnymi argumentami z taśmy wejściowej. Wynik zapisywany jest na taśmie wyjściowej.

```
1   read 0   # r0 <- TWE(a)
2   read 1   # r1 <- TWE(b)
3   add 1    # r0 <- r0 + r1
4   write 0  # TWY <- r0
5   halt
```

Przykład wywołania: TWE(-13, 37) → TWY(24)

Program wykorzystuje jedynie tryb adresowania bezpośredniego, gdyż wszystkie dane pobierane są taśmy wejściowej i zapisywane do rejestrów, pełniących rolę zmiennych.

6.2. Pole powierzchni koła

W kolejnym programie zastosowano adresowanie natychmiastowe. Jest ono wykorzystane w celu umieszczenia w kodzie programu wartości stałej, którą jest liczba π, aby możliwe było wykonanie obliczenia według wzoru $P = \pi r^2$. Program pobiera jedną liczbę, którą jest promień koła, z taśmy wejściowej i wartość obliczonego pola zapisuje na taśmie wyjściowej.

```
1   read 0   # r0 <- TWE(r)
2   mult 0   # r0 <- r0 * r0
3   mult =314      # r0 <- r0 * 314
4   div =100 # r0 <- r0 div 100
5   write 0  # TWY(P) <- r0
6   halt
```

Przykład wywołania: TWE(20) → TWY(1256)

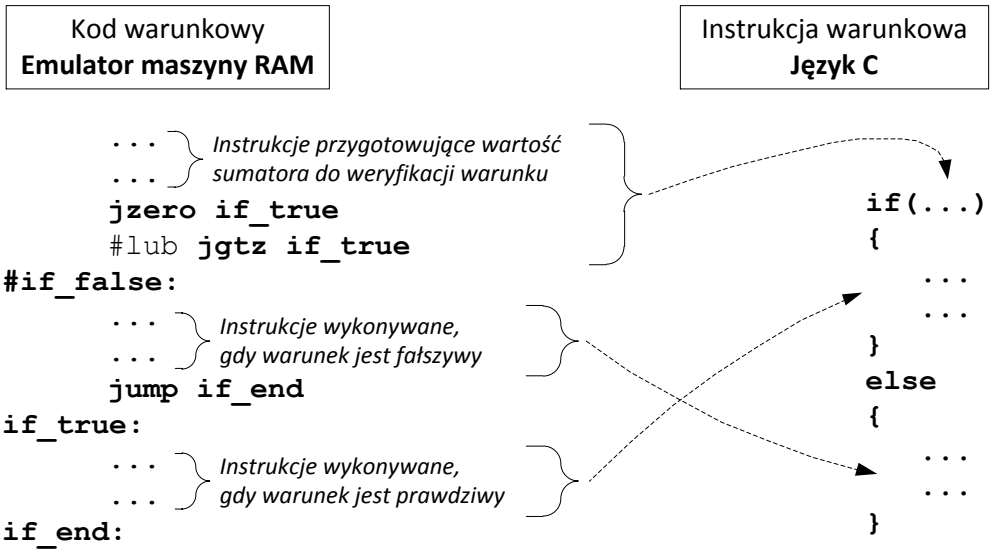
Ponieważ emulator może pracować jedynie na liczbach całkowitych, przemnożenie kwadratu promienia przez przybliżenie liczby π zrealizowano najpierw mnożąc ten kwadrat przez wartość 314, a następnie dzieląc przez 100. Kolejność tych operacji nie jest dowolna. Aby uzyskać maksymalnie dokładny rezultat przybliżony wartością całkowitą (zaokrągloną w dół), najpierw musi być wykonane mnożenie. Na przykład, przemnożenie liczby 3 przez ułamek ¾ należy wykonać w poprawny sposób poprzez działania (3 * 3) *div* 4 = 2, natomiast fałszywy wynik da obliczenie (3 *div* 4) * 3 = 0.

6.3. Test podzielności liczb

Program wczytuje z taśmy wejściowej dwie liczby (*a* i *b*) i sprawdza podzielność liczby *a* przez *b*. Podzielność weryfikowana jest na podstawie wartości reszty z dzielenia *a*/*b*. Gdy liczby są podzielne, na taśmę wyjściową zostaje zapisana wartość 1, w przeciwnym wypadku wartość 0.

Realizacja programu wymaga kodu warunkowego, różnicującego przepływ sterowania zależnie od spełnienia zadanego warunku. Zbiory rozkazów procesorów (a także emulatora maszyny RAM) nie zawierają wysokopoziomowych instrukcji sterujących, takich jak instrukcje warunkowe, instrukcje wielokrotnego wyboru czy instrukcje pętli. Wszystkie tego typu instrukcje muszą być w asemblerze zastępowane odpowiednim układem skoków warunkowych i bezwarunkowych.

Sposób realizacji kodu warunkowego, odpowiadającego instrukcji warunkowej *if* języka C, ukazany został na rysunku 6. Poprzez prawdziwość warunku rozumiana jest wartość sumatora powodująca skok w instrukcjach *jzero* oraz *jgtz*, a więc odpowiednio r0==0 w pierwszym przypadku i r0>0 w drugim przypadku. Etykieta *if_false*, rozpoczynająca ciąg instrukcji wykonywanych, gdy warunek nie będzie spełniony, posiada charakter komentarza, ponieważ w programie nie występuje żaden rzeczywisty skok do tej etykiety.



Rys. 6. Realizacja kodu warunkowego w asemblerze emulatora maszyny RAM

Inaczej niż w języku C, gdzie warunkiem może być dowolne wyrażenie języka, w emulatorze maszyny RAM zweryfikować można jedynie wartość zerową lub dodatnią sumatora. Z tego względu, rozkaz skoku warunkowego poprzedzony jest instrukcjami przygotowującymi wartość sumatora do sprawdzenia warunku. W rozważanym przykładzie, po wykonaniu ostatniej z instrukcji przygotowujących <6>, w sumatorze znajdzie się wartość równa –*a*%*b*.

Jeżeli będzie ona zerowa, co oznacza podzielność, instrukcja <12> zapisze na taśmie wyjściowej 1, w przeciwnym wypadku instrukcja <9> zapisze na taśmie wyjściowej 0.

```
1     read 1       # r1 <- TWE(a)
2     read 2       # r2 <- TWE(b)
3     load 1       # r0 <- r1
4     div 2        # r0 <- r0 div r2  // zatem: r0 == a div b
5     mult 2       # r0 <- r0 * r2
6     sub 1        # r0 <- r0 - r1  // zatem aktualnie: r0 == (a div b)*b-a
7     jzero if_true     # if(r0==0) goto if_true
8   #if_false:
9     write =0          # TWY <- 0
10    jump if_end       # goto if_end
11  if_true:
12    write =1          # TWY <- 1
13  if_end:
14    halt
```

Przykład wywołania: TWE(18, 6) → TWY(1)

6.4. Dzielenie całkowitoliczbowe

Poniższy program wykonuje dzielenie dwóch liczb, wczytanych z taśmy wejściowej i wypisuje wynik na taśmie wyjściowej. Nie różni się on niczym w porównaniu z programem z przykładu 6.1, za wyjątkiem zamiany rozkazu *add* na *div* <3>.

```
1     read 0       # r0 <- TWE(a)
2     read 1       # r1 <- TWE(b)
3     div 1        # r0 <- r0 div r1
4     write 0      # TWY <- r0
5     halt
```

Jednak w przypadku dzielenia, zerowa wartość dzielnika jest niedozwolona. Z tego powodu, wywołanie programu z zabronioną wartością dzielnika, np. TWE(13, 0) powoduje wystąpienie błędu (wyjątku) i zatrzymanie emulatora, wraz w wyświetleniem odpowiedniego komunikatu.

Aby umożliwić poprawne zakończenie programu w każdej sytuacji, niezależnie od danych wejściowych, można zastosować kod warunkowy, który sprawdzi wartość dzielnika. W przypadku próby dzielenia przez zero, na taśmę wyjściową zostanie wypisana wartość umownie oznaczająca ten fakt. Kod odpowiednio rozszerzonego programu przedstawiony jest poniżej. W przypadku zerowej wartości dzielnika, wypisywana jest wartość -1 <10,12>, w przeciwnym wypadku wypisywany jest wynik dzielenia <5-7,12>.

```
1     read 1       # r1 <- TWE(a)
2     read 0       # r0 <- TWE(b)
3     jzero if_true     # if(r0==0) goto if_true  // skok, gdy b==0
```

```
4   #if_false:
5      store 2      # r2 <- r0
6      load 1       # r0 <- r1
7      div 2        # r0 <- r0 div r2  // zatem: r0 == a div b
8      jump if_end # goto if_end
9   if_true:
10     load =-1     # r0 <- (-1)  // zablokowana próba dzielenia przez 0
11  if_end:
12     write 0      # TWY <- r0
13     halt
```

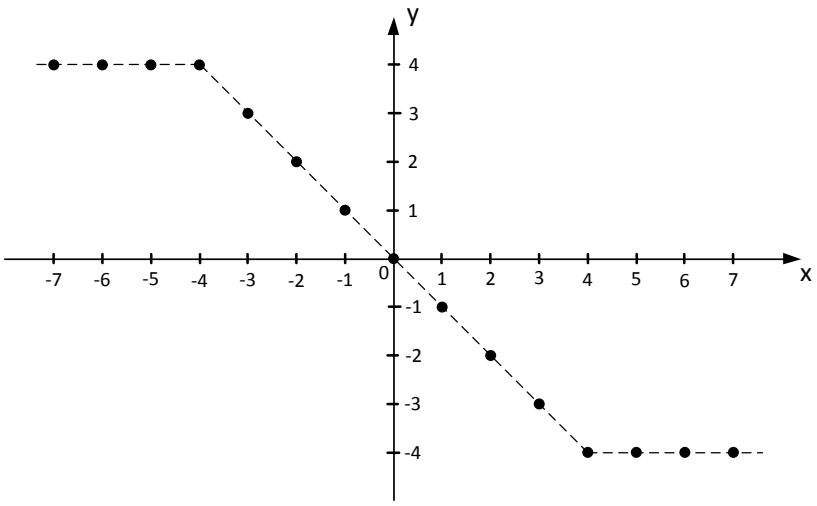
Przykłady wywołania: TWE(13, 3) → TWY(4); TWE(13, 0) → TWY(-1)

6.5. Obliczanie wartości funkcji

Program ma obliczać wartości funkcji, określonej dla liczb całkowitych i przyjmującej wartości całkowite

$$f\ x\ = \begin{matrix} 4\ dla\ x \in\ -\infty,-4 \\ -x\ dla\ x \in\ -4,4 \\ -4\ dla\ x \in\ 4,+\infty \end{matrix}$$

(wykres funkcji ukazuje rys. 7). Argument funkcji odczytywany jest z taśmy wejściowej, a obliczona wartość zapisywana jest na taśmie wyjściowej.



Rys. 7. Wykres funkcji, której wartości wyznacza program z przykładu 6.5

Ponieważ funkcję definiują trzy różne wyrażenia dla trzech przedziałów wartości argumentu, do ich rozróżnienia zastosowano zagnieżdżone bloki warunkowe. Warunek zewnętrzny <4> jest prawdziwy, gdy $x \geq 4$, wtedy wykonywana jest instrukcja <19>. W przeciwnym wypadku wykonywany jest wewnętrzny kod warunkowy <6-16>, na który składa się przygotowanie wartości sumatora <6-7>, sprawdzenie warunku wewnętrznego <8> i zwrócenie odpowiedniej wartości, zależnej od tego czy warunek jest prawdziwy <13-15>, czy fałszywy <10>.

```
1     read 1       # r1 <- TWE(x)  // argument x jest przechowywany w r1
2     load 1       # r0 <- r1
3     sub =3       # r0 <- r0 - 3
4     jgtz if_1_true    # if(r0 > 0) goto if_1_true  // skok, gdy x >= 4
5   #if_1_false:
6     load 1       # r0 <- r1
7     add =4       # r0 <- r0 + 4
8     jgtz if_2_true    # if(r0 > 0) goto if_2_true  // skok, gdy x > -4
9   #if_2_false:  # tu przejdzie sterowanie, gdy x <= -4
10    write =4     # TWY(y) <- 4
11    jump if_2_end     # goto if_2_end
12  if_2_true:     # tu przejdzie sterowanie, gdy -4 < x < 4
13    load 1       # r0 <- r1
14    mult =-1     # r0 <- r0 * (-1)
15    write 0      # TWY(y) <- r0  // zatem TWY(y) <- (-x)
16  if_2_end:      # koniec instrukcji if_2 (wewnętrznej)
17    jump if_1_end     # goto if_1_end
18  if_1_true:           # tu przejdzie sterowanie, gdy x > 3
19    write =-4    # TWY(y) <- (-4)
20  if_1_end:      # koniec instrukcji if_1 (zewnętrznej)
21    halt
```

Przykłady wywołania: TWE(11) → TWY(-4); TWE(-3) → TWY(3)

6.6. Kalkulator z operacjami +/-

Program wczytuje 3 argumenty z taśmy wejściowej: *a*, *b*, *op*. Parametr *op* oznacza rodzaj wykonywanego działania. Jeżeli *op*==1, na taśmę wyjściową zapisywany jest wynik działania *a*+*b*, jeżeli *op*==2, na taśmę wyjściową zapisywany jest wynik działania *a*-*b*. W przypadku jakiejkolwiek innej wartości *op*, po wykonaniu programu taśma wyjściowa pozostaje pusta.

Podobnie jak w programie z przykładu 6.5, zastosowano zagnieżdżone bloki warunkowe. Warunek bloku zewnętrznego <6> spełniony jest, gdy *op*==1, wtedy wykonywane są instrukcje <21-23>. W przeciwnym wypadku wykonywany jest zagnieżdżony kod warunkowy <8-18>, w którym sprawdzany jest warunek *op*==2 <8-10>. Spełnienie tego warunku skutkuje obliczeniem różnicy *a*-*b* <15-17>, jeśli warunek nie jest spełniony (*op* jest różne od 1 i 2) nie jest podejmowane żadne działanie <12>.

```
1     read 1       # r1 <- TWE(a)
2     read 2       # r2 <- TWE(b)
3     read 3       # r3 <- TWE(op)  // liczba określająca rodzaj działania
4     load 3       # r0 <- r3
5     sub =1       # r0 <- r0 - 1
```

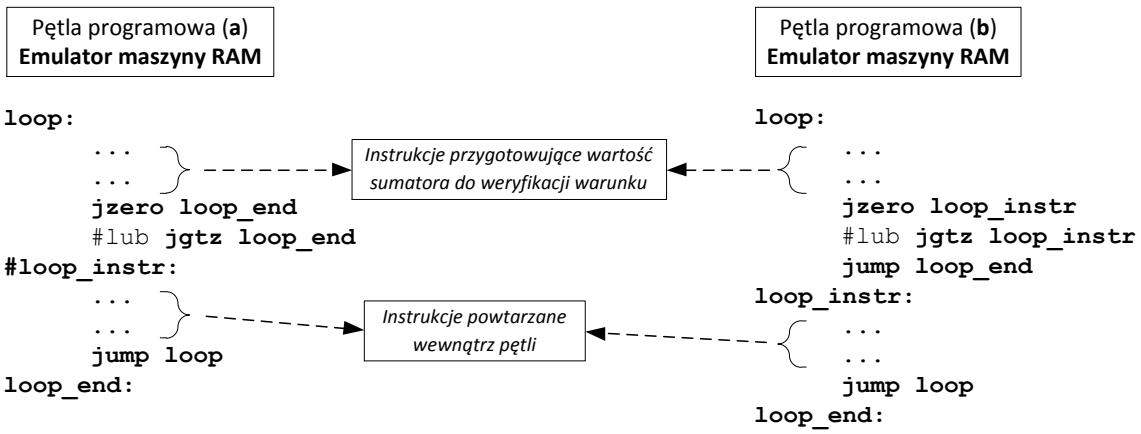
```
6     jzero if_1_true   # if(r0==0) goto if_1_true  // skok, gdy op == 1
7   #if_1_false:
8     load 3      # r0 <- r3
9     sub =2      # r0 <- r0 - 2
10    jzero if_2_true   # if(r0==0) goto if_1_true  // skok, gdy op == 2
11  #if_2_false:  # tu przejdzie sterowanie, gdy (op != 1) && (op != 2)
12    # NOP       # w takim przypadku program nic nie oblicza
13    jump if_2_end     # goto if_2_end
14  if_2_true:    # tu przejdzie sterowanie, gdy op == 2
15    load 1      # r0 <- r1  // zatem: r0 == a
16    sub 2       # r0 <- r0 - r2  // zatem: r0 == a - b
17    write 0     # TWY <- r0
18  if_2_end:     # koniec instrukcji if_2 (wewnętrznej)
19    jump if_1_end     # goto if_1_end
20  if_1_true:    # tu przejdzie sterowanie, gdy op == 1
21    load 1      # r0 <- r1  // zatem: r0 == a
22    add 2       # r0 <- r0 - r2  // zatem: r0 == a + b
23    write 0     # TWY <- r0
24  if_1_end:     # koniec instrukcji if_1 (zewnętrznej)
25    halt
```

Przykłady wywołania: TWE(4, 8, 2) → TWY(-4); TWE(17, 8, 5) → TWY()

6.7. Suma ciągu liczb

Program wczytuje kolejne liczby z taśmy wejściowej, aż do napotkania wartości 0. Na taśmie wyjściowej zapisywana jest suma wczytanych liczb.

Algorytm wymaga wielokrotnego powtórzenia pewnych operacji przez zastosowanie pętli programowej. Sposób realizacji pętli w asemblerze emulatora maszyny RAM został ukazany na rysunku 8. Wybrano pętlę, w której najpierw sprawdzany jest warunek, a następnie wykonywane są instrukcje wewnętrzne pętli, jest to zatem funkcjonalny odpowiednik pętli *while* z języka C.



Rys. 8. Realizacja pętli programowej w asemblerze emulatora maszyny RAM

Wyróżniono dwa warianty kodu: a), b), charakteryzujące się tym, że instrukcje wewnętrzne pętli wykonywane są dopóki warunek skoku w instrukcjach *jzero* lub *jgzt* **a) – nie jest / b) – jest** spełniony. Odpowiedni wariant należy wybrać indywidualnie dla każdej pętli w kodzie programu, analizując dla jakich wartościach sumatora odpowiednie jest powtórzenie, a dla jakich opuszczenie pętli.

Kod programu zaczyna się wyzerowaniem rejestru r1 <1-2>, który został przeznaczony do przechowywania sumy wprowadzonych liczb. Po wczytaniu pierwszej liczby <3>, rozpoczyna się wykonywanie pętli <4-11>. Wybrano wariant a) kodu pętli, ponieważ jej wykonywanie ma zostać przerwane przy zerowej zawartości sumatora. W ciągu instrukcji powtarzanych w pętli <7-9> do zawartości rejestru r0 dodawana jest zawartość sumatora <7-8> oraz z taśmy wejściowej odczytana zostaje kolejna liczba <9>. Po opuszczeniu pętli wypisywany jest wynik sumowania <12>.

```
1     load =0      # r0 <- 0
2     store 1      # r1 <- r0  // zatem: r1==0, r1 będzie przechowywał sumę
3     read 0       # r0 <- TWE(x)
4   loop:
5     jzero loop_end    # if(r0==0) goto loop_end // koniec pętli, gdy x==0
6   #loop_instr:  # tu zaczynają się instrukcje wewnętrzne pętli
7     add 1        # r0 <- r0 + r1  // zatem: r0 == x + suma
8     store 1      # r1 <- r0  // zatem: r1 == suma
9     read 0       # r0 <- TWE(x)  // wczytanie kolejnej liczby do sumowania
10    jump loop    # goto loop  // skok do warunku pętli
11  loop_end:      # koniec pętli
12    write 1      # TWY(suma) <- r1
13    halt
```

Przykład wywołania: TWE(4, -7, 2, 14, 0) → TWY(13)

6.8. Silnia

Program wczytuje jedną liczbę *n* z taśmy wejściowej, oblicza wartość jej silni i wynik zapisuje na taśmie wyjściowej. Na początku programu <1-2> zawartość rejestru r1 (w komentarzach oznaczanego przez *s*) ustawiana jest na 1. Rejestr ten zawiera bieżącą wartość iloczynu, który na końcu programu staje się równy *n*! Do rejestru r2 (w komentarzach oznaczanego przez *n*) wpisywana jest wartość wejściowa <3>. Pętla programowa <5-15> wykonuje się, dopóki $n > 0$. W każdym powtórzeniu pętli realizowane są działania $n \leftarrow n \cdot s$ <9-10> oraz $n \leftarrow n\text{-}1$ <11-13>.

```
1     load =1      # r0 <- 1
2     store 1      # r1 <- r0 // r1 - bieżąca wartość iloczynu, s==1
3     read 2       # r2 <- TWE(n)
4     load 2       # r0 <- r2
5   loop:
6     jgtz loop_instr   # if(r0 > 0) goto loop_instr // skok, gdy n > 0
```

```
7     jump loop_end     # skok opuszczający pętlę
8   loop_instr:         # tu zaczynają się instrukcje wewnętrzne pętli
9     mult 1      # r0 <- r0 * r1  // zatem: r0 == n * s
10    store 1     # r1 <- r0
11    load 2      # r0 <- r2  // zatem: r0 == n
12    sub =1      # r0 <- r0 - 1
13    store 2     # r2 <- r0  // zatem: n <- n-1
14    jump loop   # goto loop  // skok do warunku pętli
15  loop_end:     # koniec pętli
16    write 1     # TWY(s) <- r1
17    halt
```

Przykład wywołania: TWE(5) → TWY(120)

6.9. Implementacja tablicy

Program demonstruje przykładowy sposób obsługi tablicy w kodzie asemblerowym. Działanie programu dzieli się na dwa etapy. W pierwszym etapie z taśmy wejściowej wczytywane są kolejne wartości i zapisywane do kolejnych rejestrów, począwszy od ustalonego rejestru, stanowiącego początek tablicy (w przykładzie początkiem tablicy jest rejestr r10). Tablica nie posiada ograniczonego rozmiaru, można wczytać dowolnie dużo liczb. Wczytanie wartości 0 kończy pierwszy i rozpoczyna drugi etap działania programu. W drugim etapie kolejno odczytywane liczby interpretowane są jako indeksy poprzednio uzupełnionej tablicy i elementy tablicy zawarte pod tymi indeksami wypisywane są na taśmę wyjściową. Ponowne wczytanie wartości 0 kończy drugi etap i cały program. Ponieważ wartość 0 traktowana jest jako znacznik końca, indeksowanie tablicy zaczyna się od 1 (inaczej niż w języku C).

Pierwszy etap realizują instrukcje <2-14>. Do rejestru r1 wpisywana jest wartość 10 <2-3>. Rejestr ten pełni funkcję rejestru adresowego, wskazując bieżący numer rejestru, do którego wpisany ma być kolejny element wczytywanej tablicy. Wczytywanie elementów tablicy realizuje pętla *loop_1* <5-14>, w której powtarzane są instrukcje <8-12>, aż do wczytania wartości 0. Wewnątrz pętli odbywa się zapis wczytanej wartości pod adres wskazany przez rejestr adresowy <8> (tryb adresowania pośredniego), zwiększenie wartości rejestru adresowego o 1 <9-11> oraz wczytanie kolejnej liczby z taśmy wejściowej <12>.

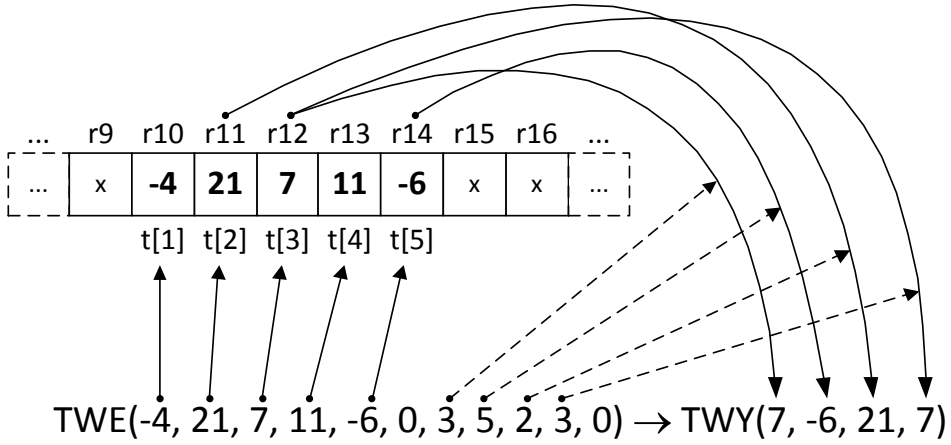
Drugi etap realizują instrukcje <16-24>. Pierwszy indeks tablicy odczytywany jest w instrukcji <16>. Kolejne operacje wykonywane są cyklicznie w pętli *loop_2*, aż do odczytania z taśmy wejściowej wartości 0. Wewnątrz pętli najpierw do wartości odczytanej z taśmy dodawana jest wartość *bazaTab*-1 <20>, co powoduje, że zawartość sumatora staje się numerem rejestru odpowiadającego elementowi tablicy o odczytanym indeksie. Wartość z rejestru o numerze wskazanym przez sumator wypisywana jest na taśmie wyjściowej <21> (tryb adresowania pośredniego), a następnie z taśmy wejściowej wczytywany jest kolejny indeks.

```
1   # Etap 1 - wczytywanie liczb do tablicy
2     load =10    # r0 <- 10
3     store 1     # r1 <- r0  // r1 == 10, r1 - rejestr adresowy tablicy
4     read 0      # r0 <- TWE(x)
```

```
5   loop_1:
6     jzero loop_1_end  # if(r0==0) goto loop_1_end  // skok, gdy x == 0
7   #loop_1_instr:# tu zaczynają się instrukcje wewnętrzne pętli loop_1
8     store *1    # r[r1] <- r0  // na koniec tablicy dołączane jest x
9     load 1      # r0 <- r1
10    add =1      # r0 <- r0 + 1
11    store 1     # r1 <- r0  // r1 <- r1+1, zwiększenie indeksu tablicy
12    read 0      # r0 <- TWE(x)  // wczytanie kolejnej liczby
13    jump loop_1 # goto loop_1  // skok do warunku pętli loop_1
14  loop_1_end:   # koniec pętli loop_1
15  # Etap 2 – wypisywanie liczb z tablicy
16    read 0      # r0 <- TWE(n)
17  loop_2:
18    jzero loop_2_end  # if(r0==0) goto loop_2_end  // skok, gdy n == 0
19  #loop_2_instr:# tu zaczynają się instrukcje wewnętrzne pętli loop_2
20    add =9      # r0 <- r0 + 9  // nrRejestr <- indeksTab + (bazaTab – 1)
21    write *0    # TWY <- r[r0]  // wypisanie wartości spod nrRejestru
22    read 0      # r0 <- TWE(n)  // wczytanie kolejnego indeksu
23    jump loop_2 # goto loop_2  // skok do warunku pętli loop_2
24  loop_2_end:   # koniec pętli loop_2
25    halt
```

Przykład wywołania: TWE(-4, 21, 7, 11, -6, 0, 3, 5, 2, 3, 0) → TWY(7, -6, 21, 7)

Rysunek 9 ilustruje zapis i odczyt wartości z rejestrów emulatora maszyny RAM dla podanej powyżej przykładowej zawartości taśmy wejściowej.



Rys. 9. Ilustracja działania programu przykładowego 6.9

6.10. Implementacja wskaźników

Program jest funkcjonalnie podobny do programu podanego w poprzednim przykładzie. Jego działanie także przebiega dwuetapowo i każdy etap kończy się wraz z odczytaniem z taśmy

wejściowej wartości 0. W tym przypadku jednak wartości nie są zapisywane do zwartego obszaru kolejno po sobie następujących rejestrów, rozumianego jako tablica, ale dla każdej operacji zapisu i odczytu można wskazać indywidualny adres (nr rejestru). Numery rejestrów, wprowadzane dla określenia lokalizacji zmiennych przeznaczonych do zapisu i odczytu, są asemblerową implementacją wskaźników.

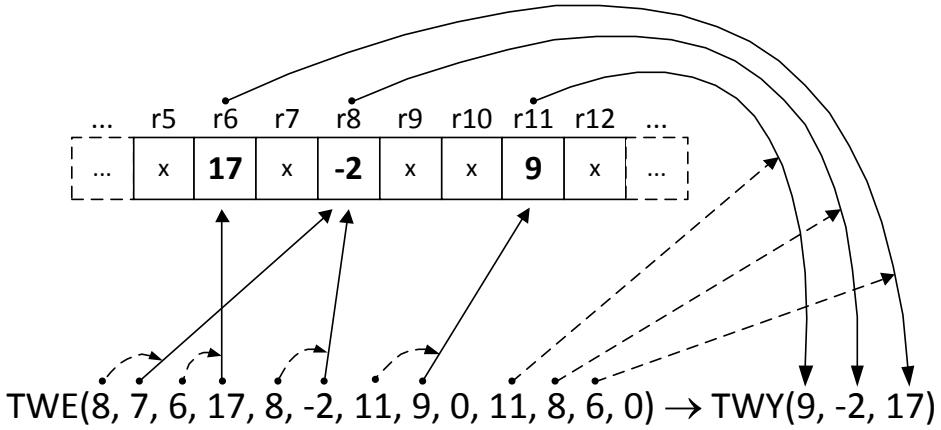
W pierwszym etapie działania programu dane są odczytywane z taśmy wejściowej parami. Pierwsza wartość pary określa numer rejestru docelowego (wskaźnik), a druga jest wartością do zapisania. Pierwsza wartość jest wczytywana z taśmy wejściowej do sumatora <2> (lub <9>), aby umożliwić weryfikację warunku pętli <4>, a następnie jest przenoszona do rejestru r1 <6>. Druga wartość pary wczytywana jest do sumatora <7> i z sumatora kopiowana do rejestru o numerze wskazanym przez rejestr r1 <8> (tryb adresowania pośredniego).

W drugim etapie liczby z taśmy wejściowej traktowane są jako numery rejestrów (wskaźniki), których zawartość jest odczytywana i wypisywana na taśmie wyjściowej <17> (tryb adresowania pośredniego).

```
1   # Etap 1 – zapisywanie wartości do wskazanych rejestrów
2     read 0       # r0 <- TWE(p)  // wczytanie adresu (wskaźnika)
3   loop_1:
4     jzero loop_1_end  # if(r0==0) goto loop_1_end  // skok, gdy p == 0
5   #loop_1_instr: # tu zaczynają się instrukcje wewnętrzne pętli loop_1
6     store 1      # r1 <- r0  // adres zostaje zachowany w r1
7     read 0       # r0 <- TWE(x)  // wczytanie wartości
8     store *1     # r[r1] <- r0  // zapis wartości x pod adres p
9     read 0       # r0 <- TWE(p)  // wczytanie kolejnego adresu
10    jump loop_1 # goto loop_1  // skok do warunku pętli loop_1
11  loop_1_end:    # koniec pętli loop_1
12  # Etap 2 – odczytywanie wartości ze wskazanych rejestrów
13    read 0       # r0 <- TWE(p)  // wczytanie adresu (wskaźnika)
14  loop_2:
15    jzero loop_2_end  # if(r0==0) goto loop_1_end  // skok, gdy p == 0
16  #loop_2_instr: # tu zaczynają się instrukcje wewnętrzne pętli loop_2
17    write *0     # TWY <- r[r0]  // wypisanie wartości spod adresu p
18    read 0       # r0 <- TWE(p)  // wczytanie kolejnego adresu (wskaźnika)
19    jump loop_2 # goto loop_2  // skok do warunku pętli loop_2
20  loop_2_end:    # koniec pętli loop_2
21    halt
```

Przykład wywołania: TWE(8, 7, 6, 17, 8, -2, 11, 9, 0, 11, 8, 6, 0) → TWY(9, -2, 17)

Rysunek 10 ilustruje zapis i odczyt wartości z rejestrów emulatora maszyny RAM dla podanej powyżej przykładowej zawartości taśmy wejściowej.



Rys. 10. Ilustracja działania programu przykładowego 6.10

7. Zadania

1. Przetestować działanie wszystkich programów podanych w rozdziale 6, uruchamiając je z przykładowymi danymi wejściowymi oraz z danymi wejściowymi przygotowanymi samodzielnie.
2. Wzorując się programem z przykładu 6.2, utworzyć i przetestować program, który będzie obliczał:
   a) Objętość kuli: $V = \frac{4}{3}\pi r^3$
   b) Objętość stożka: $V = \frac{1}{3}\pi r^2 h$
   c) Wartość wielomianu: $W\ x\ = 2x^2 - 5x + 3$
3. Opierając się na programie z przykładu 6.5, utworzyć i przetestować program, który będzie obliczał wartość funkcji: $f\ x\ = \begin{matrix} -3\ dla\ x \in\ -\infty,-4 \\ -2x\ dla\ x \in\ -4,4 \\ 3\ dla\ x \in\ 4,+\infty \end{matrix}$
4. Rozbudować program kalkulatora z przykładu 6.6 tak, aby poza operacjami 1-dodawanie, 2-odejmowanie, obsługiwał także operacje 3-mnożenie, 4-dzielenie.
5. Wzorując się na programach przykładowych 6.5 i 6.6 (instrukcje warunkowe) oraz 6.7 i 6.8 (pętle), utworzyć i przetestować program, który:
   a) Wczytuje dwie liczby a i b ($a \leq b$) z taśmy wejściowej, oblicza sumę wszystkich liczba całkowitych od a do b i zapisuje wynik na taśmie wyjściowej,
   b) Wczytuje ciąg kolejnych liczb całkowitych z taśmy wejściowej, aż do napotkania wartości 0, wyznacza największą z wprowadzonych wartości i zapisuje wynik na taśmie wyjściowej.
6. Rozważyć program przykładowy 6.9:
   a) Jakie będą konsekwencje, jeżeli w miejscach określających indeksy elementów odczytywanych z tablicy wstawione zostaną niepoprawne wartości? Na przykład, w zbiorze danych wejściowych TWE(4, -6, 2, 0, 3, -12, 5, 1, 0) dwie liczby: -12 i 5 są niepoprawne, gdyż dla wprowadzonej trójelementowej tablicy właściwymi wartościami indeksów są 1, 2 oraz 3. WSKAZÓWKA: Wyróżnić dwa przypadki

zachowania się programu w sytuacji niepoprawnych indeksów, zależnie od zakresu ich wartości.

b) Rozszerzyć program 6.9 tak, aby niepoprawne wartości indeksów tablicy były ignorowane i nie powstawały odpowiadające im wpisy na taśmie wyjściowej, na przykład: TWE(4, -6, 2, 0, 3, -12, 5, 1, 0) $\rightarrow$ TWY(2, 4).

7. Rozszerzyć program przykładowy 6.9 w taki sposób, aby tablica posiadała ograniczony rozmiar (np. 6). Po wprowadzeniu 6 elementów, dalsze wprowadzane liczby nie powinny być zapisywane do rejestrów.

8. Rozważyć program przykładowy 6.10:

   a) Czy odczyt i zapis za pomocą wskaźników (adresów rejestrów) możliwy jest dla dowolnie wybranych adresów wyrażonych liczbami całkowitymi? Jeżeli nie, jakie będą konsekwencje niewłaściwej wartości adresu. WSKAZÓWKA: Zapoznać się ze wskazówką dla zadania 6.6.

   b) Rozszerzyć program 6.10, uzupełniając go o prosty mechanizm ochrony pamięci. Ochrona pamięci ma polegać na ignorowaniu operacji zapisu i odczytu, które będę odwoływały się do rejestrów spoza zdefiniowanego zakresu, np. 10…100.

9. Utworzyć i przetestować program symulujący działanie kostki sześciościennej. Program powinien po każdorazowym wykonaniu zapisywać na taśmie wyjściowej jedną, losowo wybraną liczbę z zakresu 1…6, odpowiadającą liczbie wylosowanych oczek. Wykorzystać podane poniżej informacje pomocnicze:

   a) Odczyt z pustej taśmy wejściowej skutkuje odczytaniem losowo wybranej liczby całkowitej.

   b) Resztę z dzielenia całkowitoliczbowego *a* % *b* wyznacza program:

```
read 0  # r0 <- TWE(a)
read 1  # r1 <- TWE(b)
store 2 # r2 <- r0
div 1   # r0 <- r0 div r1
mult 1  # r0 <- r0 * r1
store 1 # r1 <- r0
load 2  # r0 <- r2
sub 1   # r0 <- r0 - r1
write 0 # TWY <- r0
halt
```

---

[1] Muła W., Emulator maszyny RAM, http://wm.ite.pl/stud/ram/index.html