Budowa maszyn typu von Neumanna

Podstawowe składniki:

⇒ **Pamięć operacyjna**, podzielona na komórki, ponumerowane liczbami naturalnymi (adresami), Mem[0..MAXMEM], adres jest liczbą z przedziału 0.. MAXMEM.

Zawartość komórki jest ciągiem bitów i może być traktowana jako:

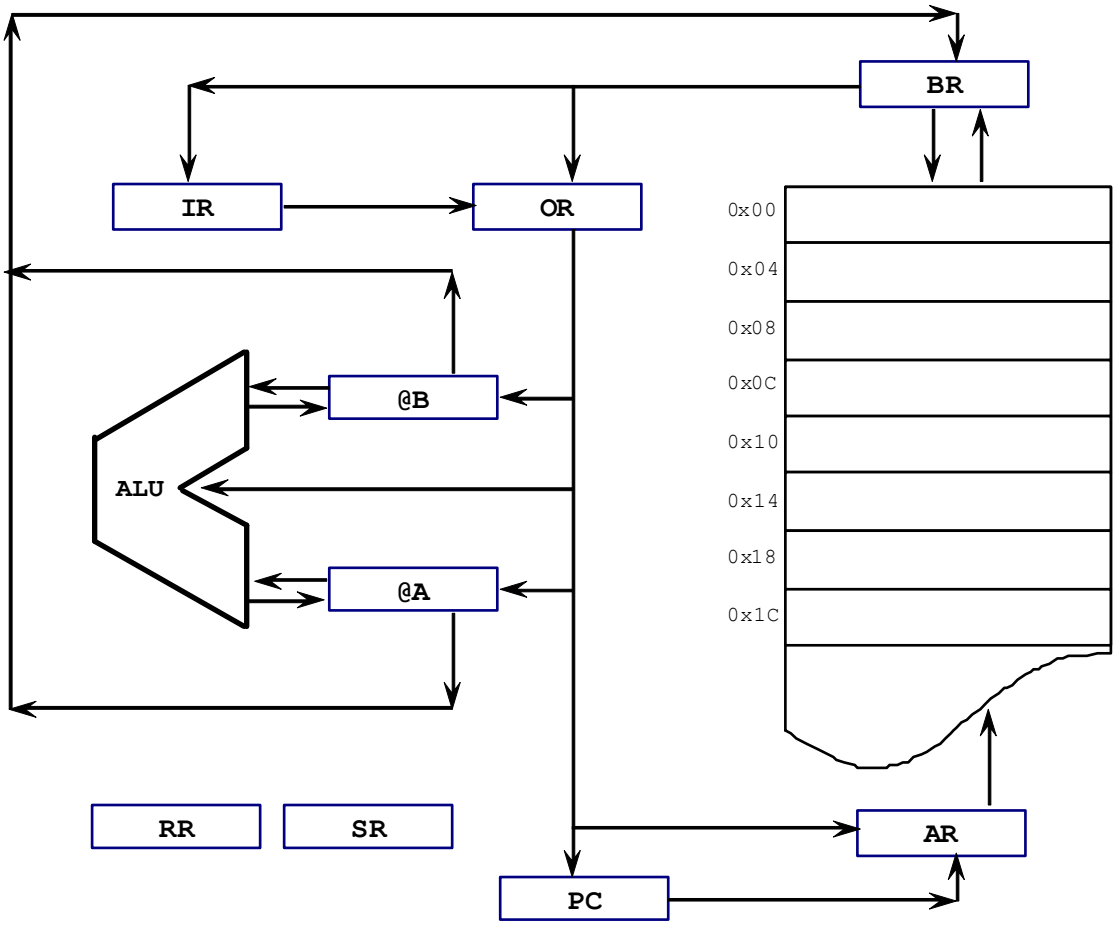
- liczba (dana przetwarzana przez program)
- instrukcja (rozkaz) programu;

Własność maszyny von Neumanna: **patrząc na zawartość komórki pamięci nie możemy stwierdzić czy jest tam zapamiętana jakaś dana czy instrukcja**

⇒ **Procesor,** zawierający m.in.

- jednostkę arytmetyczno-logiczną (**ALU** –ang. *Arithmetic & Logic Unit*, wykonuje operacje arytmetyczne i logiczne)
- rejestry uniwersalne (akumulatory – ozn. **@A**): do wykonywania operacji arytmetyczno-logicznych przez ALU (stąd pobierany jest jeden z operandów i tam zapisywany jest wynik działania), akumulator zazwyczaj zawiera jeden z operandów, drugi zwykle pobierany z pamięci (pośrednio poprzez rejestr operanda).
- rejestr licznika rozkazów – ozn. **PC**: adres komórki, której zawartość będzie potraktowana jako rozkaz w następnym cyklu rozkazowym; na początku działania programu (po jego załadowaniu do pamięci jest tam adres pierwszego rozkazu programu).
- rejestr adresowy –ozn. **AR**: rejestr zawierający adres PaO skąd będzie pobierana zawartość komórki lub adres komórki do której będzie zapis.
- rejestr buforowy pamięci – ozn. **BR**: rejestr poprzez który następuje odczyt/zapis z/do pamięci.
- rejestr instrukcji – ozn. **IR**: rejestr, do którego będzie załadowana wykonywana instrukcja – zawiera kod instrukcji oraz operand, który jest wykorzystywany podczas wykonywania cyklu rozkazowego do obliczenia operanda efektywnego.
- rejestr operanda – ozn. **OR**: zawiera operand z rejestru instrukcji lub obliczony operand efektywny.
- rejestr stanu procesora – ozn. **SR**: zawiera szereg bitów określających jaka wartość była wynikiem ostatniej operacji względem zera lub w wyniku operacji porównania np. bit większe od zera (`SR[pos]`) oraz równe zero (SR[`zero`]) (w rozszerzonym opisie mogą być dodatkowe bity).

  Bity tego rejestru są ustawiane po każdej modyfikacji jednego z akumulatorów i dotyczą wartości zmienionego akumulatora lub po wykonaniu operacji porównania.
- rejestr żądań – ozn. **RR**: rejestr zawiera bit RUN (ozn. `RR.run`), który jeżeli zostanie zgaszony to maszyna staje (bit jest ustawiany na 1 gdy startuje program po załadowaniu do pamięci i ustawieniu PC), bit wskazujący nadejście żądania obsługi urządzenia zewnętrznego np. we-wy. Itp.



Komórka interpretowana jako rozkaz zawiera :

- część operacyjną – kod operacji

  przykładowe operacje:
  - załaduj do akumulatora,
  - dodaj do akumulatora,
  - wstaw zawartość akumulatora do pamięci,
  - skocz do wskazanej komórki w programie,
- część adresową (skąd wziąć operand rozkazu lub operand natychmiastowy)

Praca maszyny czyli sposób wykonanie programu:

Maszyna pracuje zgodnie ze schematem zwanym **cyklem rozkazowym procesora** - niepodzielny, składający się z 4 faz:

- Pobranie rozkazu (z komórki o adresie podanym w PC) i zwiększenie PC,
- Obliczenie operanda efektywnego,
- Wykonanie rozkazu, i w przypadku modyfikacji akumulatora ustawienie bitów SR
- Sprawdzenie czy podczas wykonywania nie zaszło zdarzenie wymagające specjalnej obsługi (np. obsługa urządzenia we/wy) – sprawdzenie RR

Wykonywanie cyklu rozkazowego aż do:
- napotkania zgaszenie bitu RUN w rejestrze RR, lub

- wystąpienie błędu krytycznego uniemożliwiającego (np. z powodu pomyłki programisty niedozwolone odwołanie do pamięci poza programem, dzielenie przez zero)
- itp.

Przykładowa maszyna cyfrowa DC2 (Didactic Computer with 2 accumulators)

Organizacja pamięci: słowo maszynowe: 32-bitowe (4-bajtowe)
Wszystkie rejestry są 32 bitowe (4 bajtowe). Ponieważ rozkazy są 32 bitowe to przy przechodzeniu do następnego rozkazu licznik rozkazów zwiększany jest o 4.
Słowo traktowane jako dana: zawartość jest liczbą całkowitą w kodzie uzupełnieniowym
Słowo traktowane jako rozkaz dzieli się na następujące części:

- kod rozkazu (code) bity 0–7
- wybór akumulatora bit 9
- rodzaj adresacji (mod) bity 12-13
- część adresowa (adr) bity 16–31

Rodzaje adresacji (czyli sposób wyliczania operandu efektywnego)

Rodzaj adresacji to sposób, w jaki wylicza się ostateczną wartość operanda (tzw. operand efektywny), na podstawie części .adr.

PMC (Przykładowa Maszyna Cyfrowa) ma tylko 3 rodzaje adresacji:

- natychmiastowa: zawartość części adresowej rozkazu jest operandem efektywnym
  `OR := IR.adr`
- bezpośrednia: część adresowa rozkazu jest adresem pamięci zawierającej operand efektywny
  `OR := Mem[IR.adr]`
- pośrednia: część adresowa rozkazu jest adresem pamięci zawierającej adres pamięci zawierającej operand efektywny (stosowany najczęściej prze dostępie do elementów bloków pamięci np. tablic)
  `OR:=Mem[ Mem[IR.adr] ]`

Operand (argument) efektywny jest liczony przed wykonaniem rozkazu i umieszczany w rejestrze operanda OR. Jest on, w fazie wykonania rozkazu jednym z argumentów dla operacji dwuargumentowych lub adresem słowa pamięci przy rozkazach przesyłania lub skoku.

Podstawą do obliczenia argumentu efektywnego jest argument natychmiastowy umieszczony w rejestrze instrukcji na bitach od 16 do 31 (`IR.adr`) – dwa bajty pola adresowego. Wartość tego pola jest traktowana jako 16-bitowa liczba całkowita zapisana w kodzie uzupełnieniowym. Aby nie zmienić wartości tej liczby przy umieszczaniu jej w rejestrze operanda OR, należy uzupełnić ją z lewej strony (starsze dwa bajty w liczbie 32-bitowej) zerami, gdy jest dodatnia i jedynkami, gdy jest ujemna.

Lista rozkazów, z opisem symbolicznym (język asemblera):

Dla opisu rozkazów przyjęto oznaczenie: AC oznacza jeden z akumulatorów @A lub @B.

Rozkazy organizacyjne:

null nic nie rób
halt zatrzymaj maszynę (zakończ wykonywanie programu RR.run=0)

Rozkazy przesyłania:

load załadować operand efektywny do akumulatora
AC = OR np. load, @A, 5
store zapamiętać zawartość akumulatora w pamięci pod adresem zapisanym w OR
Mem[OR] = AC, lub Mem[Mem[OR]] = AC, np. store, @A, 4

Rozkazy sterujące:

W rozkazach tych AC oznacza rejestr ostatnio używany tzn ten na którym była wykonywana ostania operacja.

jump skok bezwarunkowy, np. `jump, label1`
`PC=OR`,
jzero skok jeżeli zero (`SR[zero]=0`), `np. jzero, label1`
`if( AC==0 ) PC= OR`
jnzero skok jeżeli nie zero (`SR[zero]!=0`),
`if( AC!=0 ) PC= OR`,
jpos skok jeżeli większe od zera, (`SR[pos]=1`)
`if( AC>0 ) PC= OR`
jneg skok jeżeli nie zero (`SR[neg]=1`),
`if( AC<0 ) PC= OR`

Rozkazy arytmetyczne:

add dodaj do akumulatora operand efektywny, np. `add, @a, (8)`
`AC <- AC + OR`
sub odejmij od akumulatora operand efektywny, np. `sub, @B, (4)`
`AC <- AC - OR`
mult pomnóż akumulator przez operand egektywny, np. `mult, @B, (4)`
`AC <- AC * OR`
div podziel akumulator prze operand efektywny, np. `div, @B, (4)`
`AC <- AC / OR` (dzielenie całkowite)

Struktura programu w języku assemblera maszyny DC2

Przyjęto założenie, że wszystkie identyfikatory złożone są ze znaków liter (dużych i małych), cyfr i znaku podkreślenia a zaczynające się od litery lub znaku podkreślenia.

- Początek programu: **`.UNIT, id_nazwy_programu`**
- Początek segmentu danych**: `.DATA`**
- Początek segmentu kodu: **`.CODE`**
- Koniec programu: **`.END`**

W segmencie danych deklaracja zmiennej (rezerwacja pamięci dla zapamiętywania danych) realizowana jest zgodnie z następującym formatem:

`id_zmiennej: .WORD, wartość_początkowa`

lub w przypadku deklarowania bloku pamięci (tablicy):

id_zmiennej: .WORD, wartość_początkowa$_1$,...,wartość_początkowa$_n$

Dodatkowo, przy inicjowaniu bloku tablicy można użyć naku # to wskazania ilości powtórzeń danej wartości jak np. `t : .WORD, 2,5,10#0`, co oznacza blok złożony z 12 słów z których pierwsze ma wartość 2 drugie wartość 5 a kolejne 10 ma wartość 0.

Deklaracja zmiennej oznacza zarezerwowanie pewnej ilości kolejnych słów pamięci, wypełnieniu ich podanymi wartościami i związaniu nazwy (identyfikatora zmiennej) z adresem pierwszego słowa zarezerwowanego obszaru. **Nazwa zmiennej od tej pory odpowiada nie zawartości pamięci (jak to jest w językach programowania wysokiego poziomu) lecz jest *symbolicznym oznaczeniem adresu zarezerwowanego obszaru pamięci***.

Przykłady deklaracji:

```
tab : .WORD, 25,-18, 4,0, 10#1, 3, 2
suma: .WORD, 0
adr_t: . WORD, tab
```

W segmencie kodu występują rozkazy dwu- lub jedno- lub zero-argumentowe. W każdym rozkazie identyfikator rozkazu zakończony jest przecinkiem i ewentualne agrumenty są rozdzielane przecinkami.

Przykłady rozkazów:

```
load, @A, (suma)
add, @A, ((adr_t))
jump, label_end
halt,
```

Etykiety do których realizowany jest skok zapisywane są jako **id_etykiety:** (identyfikator zakończony znakiem dwukropka).

Przykłady użycia etykiet:

```
label1: add, @B, 1
label2: null
label3:
     add, @B, 1
```

Komentarze zapisuje się po znaku średnika (;) tylko do końca linii, w której wystąpił.

Przykłady programów w języku assemblera maszyny DC2

Przykład dodawania dwóch liczb:

```
     .UNIT, dodawanie1
;------------------------------
; program liczy sumę dwóch liczb

     .DATA
     x   : .WORD, -125
     y   : .WORD, 5321
     suma:.WORD, 0
```

```
      .CODE
      load, @A, (x)       ; @A <- Mem[x]
      add,  @A, (y)       ; @A <- @A + Mem[y]
      store,@A, suma      ; Mem[suma] <- @A
      null,
      halt,

      .END
```

Ten sam program można zapisać też w inny sposób wykorzystując własność maszyny typu von Neumanna, iż dane i kod mogą się dowolnie przeplatać:

```
      .UNIT, dodawanie2
;-------------------------------
; program liczy sumę dwóch liczb

      .DATA
      x: .WORD, -125
      y: .WORD, 5321

      .CODE
      load, @A, (x)       ; @A <- Mem[x]
      add,  @A, (y)       ; @A <- @A + Mem[y]
      jump, lab1

      .DATA
      suma:.WORD, 0

      .CODE
lab1: store, @A, suma     ; Mem[suma] <- @A
       null,
       halt,
      .END
```

Przykład mnożenia przy pomocy dodawania:

```
      .UNIT, mnoz_dod
;----------------------
      .DATA
      x  : .WORD, 14
      y  : .WORD, 12
      res: .WORD, 0
      .CODE
     load,   @A, (y)
lab1:jzero, lab2
     load,  @A, (res)
      add,   @A, (x)
      store, @A, res      ; (res) <- (res) + (x)
      load,  @A, (y)
      sub,   @A, 1
      store, @A, y        ; (y) <- (y) - 1
      jump,  lab1
lab2:halt,
      .END
```

Powyższy program można też zapisać z użyciem dwóch akumulatorów redukując tym samym ilość rozkazów wykonywanych w pętli (w poniższym przykładzie zmienna y nie jest modyfikowana)

```
      .UNIT, mnoz_dod
;---------------------
      .DATA
      x: .WORD, 14
      y: .WORD, 12
      res:.WORD, 0
      .CODE
      load, @B, (y)
lab1: jzero, lab2
     load, @A, (res)
      add, @A, (x)
      store, @A, res
      sub, @B, 1
      jump, lab1
lab2: halt,
      .END
```

Przykład znajdowania ilości wystąpień danej liczby w tablicy:

```
      UNIT, example
;----------------
      .DATA
x  :.WORD, 4
tab:.WORD, 1, 2, 4, 3#5, 2, 0, 4, 4, 8, 2
n  :.WORD, 14           ; ilość elementów tablicy
adr:.WORD, tab
res:.WORD, 0            ; ilość wystąpień x w tablicy
i  :.WORD, 0            ; ilość przeglądniętych elem. tablicy

      .CODE
et1:  load,   @A, (i)
      sub,    @A, (n)
      jzero,  et_end
      load,   @A, ((adr)) ; porównanie elem. tab z x
      sub,    @A, (x)
      jnzero, et2

      load,  @A, (res)   ; element x znaleziony w tablicy
      add,   @A, 1
      store, @a, res

et2:  load,  @A, (adr)     ; przejdź do następnego elem. tablicy
      add,   @A, 4
      store, @a, adr
      load,  @A, (i)       ; kolejny elem. tablicy przeglądnięty
      add,   @A, 1
      store, @A, i
      jump, et1

et_end:
      halt,
      .END
```

Powyższy przykład może oczywiście zostać zoptymalizowany przez zastosowanie drugiego akumulatora lub przynajmniej przestawienie etykiety `et1` do następnej instrukcji (czyli do `sub, @A, (n)`, gdyż przed wykonaniem skoku w akumulatorze @A jest właśnie wartość zmiennej spod adresu oznaczonego symbolicznie ‘i’).

Przykładowe tematy do rozwiązania:

- Algorytm obliczenia reszty z dzielenia całkowitego
- Algorytm dzielenia z resztą dwóch liczb całkowitych (nieujemnej przez dodatnią) przy pomocy odejmowania
- Algorytm znajdowania NWP(x, y)
- Algorytm znajdowania NWW(x, y)
- Algorytm obliczający sumę, spośród liczb $a_1,..., a_n$, dla których liczba k jest podzielnikiem
- Algorytm znajdowania elementu minimalnego lub maksymalnego w ciągu $a_1,..., a_n$
- Algorytm obliczania n-tego wyrazu ciągu Fibonacciego ($F_0=0$, $F_1=1$, $Fn=F_{n-2}+F_{n-1}$)
- Algorytm odwracania kolejności elementów w ciągu $a_1,..., a_n$
- Iloczyn skalarny dwóch wektorów