Wstęp do informatyki

Architektura i działanie komputera

Piotr Fulmański

19 grudnia 2023

Spis treści

1. **Abstrakcyjny model maszyny przetwarzającej**
2. **Bramki logiczne**
3. **Bramki logiczne**
4. **Współczesny komputer**
   - Maszyna analityczna
   - Architektura von Neumanna
   - Najpowszechniejszy współcześnie model architektury
   - Procesor

Maszyna Turinga

- Sekwencyjność operacji – intuicja prowadząca do maszyny.

Maszyna Turinga

- Sekwencyjność operacji – intuicja prowadząca do maszyny.

Abstrakcyjny model

Maszyna Turinga składa się z:

- **taśmy**;
- **głowicy**;
- **stanów**;
- **funkcji przejścia** (lub **tablicy przejść**).

Abstrakcyjny model – taśma

- **Taśma** podzielona jest na przylegające do siebie **komórki**.
- Każda komórka zawiera jeden symobl (obiekt) z pewnego **skończonego alfabetu**. Alfabet zawiera zawsze co najmniej **symbol pusty** i **symbol końcowy** oraz opcjonalnie inne symbole.
- Taśma jest nieskończona. Oznacza to, że maszyna Turinga zawsze ma dostępnej tyle pamięci ile potrzebuje.

Abstrakcyjny model – głowica

Głowica odczytuje i zapisuje symbole (obiekty) z taśmy (za jednym razem tylko z jednej komórki), oraz powoduje przemieszczanie taśmy w lewą stronę, prawą lub pozostawia ją nieprzemieszczoną. W niektórych elementem ruchomym jest głowica (to ona przesuwa się nad nieruchmą taśmą.

Abstrakcyjny model – stany

Stany opisują stany w jakich może znaleźć sie maszyna Turinga – każdy stan utożsamiać można z wynikiem pewnej operacji, np. wypisanie czegoś na ekran, reakcja na naciśnięcie przycisku itp. Ilość wszystkich stanów jest skończona i istnieje jeden specjalny stan – **stan początkowy** – od którego rozpoczynamy analizę działania maszyny.

Abstrakcyjny model – funkcja przejścia

Funkcja przejścia, określa, że będąc w pewnym stanie i odczytując pewien symbol powinniśmy:

- zapisać (albo nie) jakiś symbol w aktualnie odczytywanej komórce,
- przesunąć głowicę (w lewo – 'L', w prawo –'R', pozostawić na miejscu '_'),
- wskazać stan w jakim maszyna powinna się znaleźć (może to być ten sam stan).

W niektórych modelach w przypadku natrafienia na nieistniejącą kombinację `(stan,odczytany_symbol)` maszyna zatrzymuje się, w innych wymaga się określenia wszystkich możliwych kombinacji.

Abstrakcyjny model – jak to działa?

Zauważmy, że wszystkie elementy maszyny – jej stany, akcje (np. drukowanie, kasowanie, przesuwanie taśmy) – są skończone, dyskretne i jednoznacznie rozróżnialne. Poza tym dysponujemy teoretycznie nieskończoną ilości taśmy (pamięci).

Idea maszyny Turinga bazuje na sekwencyjności elementarnych operacji polegających na zmianie według precyzyjnie i dobrze określonych reguł zawartości nieskończonej taśmy papieru „komórka po komórce”. Opis postępowania jest bardzo prosty i sprowadza się do formuł postaci „*Jeśli jesteś w stanie X i odczytanym symbolem jest Y to zapisz symbol Z, przesuń się o jedną komórkę w prawo i przejdź do stanu (przyjmij, że jesteś w stanie) A.*"

Turing machine – formalna definicja

Formalna definicja:

$$M = (Q, \Sigma, \delta, s),$$

gdzie:

- Q – skończony niepusty zbiór **stanów**;
- Σ – skończony niepusty zbiór dopuszczalnych **symboli taśmowych**, zwany również **alfabetem**; Zbiór ten musi zawierać co najmniej dwa specjalne symbole: $\sqcup$ (**symbol pusty**) i $\triangleright$ (**symbol końcowy**);
- δ – **funkcja następnego ruchu** (nazywana także **diagramem** lub **tablicą przejść**), która jest odwzorowaniem ze zbioru $\delta : Q \times \Sigma$ w zbiór $(Q \cup \{k, t, n\} \times \Sigma) \times \{\leftarrow, \rightarrow, -\}$,[a] gdzie: k – **stan końcowy**, t – **stan akceptujący**, n – **stan odrzucający**, zaś $\leftarrow, \rightarrow, -$ oznaczają ruch głowicy odpowiednio w lewo, w prawo oraz pozostanie w miejscu; symbole $\leftarrow, \rightarrow, -$ nie należą do zbioru Q.
- s – **stan początkowy** (wyróżniony stan należący do zbioru Q).

[a] Funkcja δ może nie być określona dla wszystkich argumentów.

Transition function can be written as a table of the following form

Current state	Current symbol	Next state	Next symbol	Tape move
...	...	...	...	...

where we can list all possible combinations of states and symbols line by line.

Other option is to create matrix where columns corresponds to states and rows to tape (input) symbols

	Stete 1	State 2	...	State N
Symbol 1	...	Next state; Next symbol; Tape move	...	...
...	...	...	...	...
Symbol M	...	...	...	...

Uwaga

W informatyce dowodzi się równoważności wielu różnych wariantów maszyny Turinga. Np. dość łatwo jest pokazać, że maszyna Turinga z wieloma taśmami nie różni się istotnie od klasycznej maszyny jednotaśmowej. Również niedeterministyczne maszyny Turinga są równoważne deterministycznym.

	S	K
a	S; b; R	END
b	S; a; R	
$\sqcup$ | $\triangleright$	K; –; –	

```
State    Tape
S        ba[a]bbaa
```

State	Tape
S	ba[a]bbaa

We know a simple 2 steps procedure to convert integers expressed in U2 to opposite value

1. Negate all bits.
2. Add 1 to the previous result.

```
00010100 = +20 (U2)

11101011   negation
       1   add 1
========
11101100 = -20 (U2)
```

There is one more procedure which is more automatical than previous

1. Move to the most rhs (right hand side) bit.
2. Move from right to left bit by bit and until bit has value 0 copy it to the final result.
3. Copy also to the final result first bit which has value 1.
4. Copy to the final result all other bits one by one changing its values to opposite.

```
00010100 = +20 (U2)

      v                         v
00010100   step 1               100   step 3

    v                         v
     00   step 2              11101100   step  4

                              11101100 = -20 (U2)
```

Based on the presented idea we can create Turing machine. This machine acts not exactly the just presented way – this is left as an exercise. Below slightly modified approach is used.

	S	1	2	K
0	1; 1; R	1; 1; R	K; 1; L	
1	1; 0; R	1; 0; R	2; 0; L	END
⊔	S; –; R	2; –; L	K; –; –	

```
State    Tape
S        _[_]00010100__
```

```
State    Tape
S        _[_]00010100__
```

If we prefer not to left head on last symbol but move it to the initial position, we can use the following, slightly modified, machine.

	S	1	2	3	K
0	1; 1; R	1; 1; R	3; 1; L	3; –; L	END
1	1; 0; R	1; 0; R	2; 0; L	3; –; L	
⊔	S; –; R	2; –; L	K; –; –	K; –; –	

```
State    Tape
S        _[_]00010100__
```

Problem

Consider a following problem: turing machine stops in accept state iff reads at least one time every symbol from the set: $\{1, 2, 3\}$. Show your solution using a following tape

Tape
11021223012

Note 1: there is no precise information about tape symbols we can use, so let's assume that the set of all possible symbols consists of elements 0, 1, 2 and 3.
Note 2: with this turing machine we have a „tool" to accept (detect) a specific word (or sequence of chars). If we start from most lhs symbol a sequence detected by our machine would be 11021223, if we start from 5-th lhs a sequence would be 1223.

Turing machine

Example: useful example number 2 (accept a word)

	S	1	2	3	4	5	6	K
0	S; –; R	1; –; R	2; –; R	3; –; R	4; –; R	5; –; R	K; –; R	END
1	1; –; R	1; –; R	4; –; R	5; –; R	4; –; R	5; –; R	6; –; R	
2	2; –; R	4; –; R	2; –; R	6; –; R	4; –; R	K; –; R	6; –; R	
3	3; –; R	5; –; R	6; –; R	3; –; R	K; –; R	5; –; R	6; –; R	

```
State    Tape
S        [1]1021223012
```

```
State    Tape
S        [1]1021223012
```

Hipoteza Churcha-Turinga

„Każdy problem, dla którego przy nieograniczonej pamięci oraz zasobach istnieje efektywny algorytm jego rozwiązywania, da się rozwiązać na maszynie Turinga"[a]

---

[a] Jest to tylko hipoteza. Nie można sprawdzić jej na drodze matematycznych rozważań jako że łączy w sobie zarówno ścisłe, jak i nieprecyzyjne sformułowania, których interpretacja może zależeć od konkretnej osoby.

Wnioski

- Hipoteza pozwala dokładniej sformułować pojęcie samego algorytmu, mówiąc jednocześnie, że komputery są w stanie go wykonać.
- Wszystkie komputery mają jednakowe możliwości, jeśli chodzi o zdolność do wykonywania algorytmów, a nie dostępne zasoby.
- Nie jest możliwe zbudowanie komputera o zdolności do rozwiązywania większej liczby problemów, niż najprostsza maszyna Turinga.

Wnioski

- Hipoteza pozwala dokładniej sformułować pojęcie samego algorytmu, mówiąc jednocześnie, że komputery są w stanie go wykonać.
- Wszystkie komputery mają jednakowe możliwości, jeśli chodzi o zdolność do wykonywania algorytmów, a nie dostępne zasoby.
- Nie jest możliwe zbudowanie komputera o zdolności do rozwiązywania większej liczby problemów, niż najprostsza maszyna Turinga.

Wnioski

- Hipoteza pozwala dokładniej sformułować pojęcie samego algorytmu, mówiąc jednocześnie, że komputery są w stanie go wykonać.
- Wszystkie komputery mają jednakowe możliwości, jeśli chodzi o zdolność do wykonywania algorytmów, a nie dostępne zasoby.
- Nie jest możliwe zbudowanie komputera o zdolności do rozwiązywania większej liczby problemów, niż najprostsza maszyna Turinga.

Wnioski

- Hipoteza pozwala dokładniej sformułować pojęcie samego algorytmu, mówiąc jednocześnie, że komputery są w stanie go wykonać.
- Wszystkie komputery mają jednakowe możliwości, jeśli chodzi o zdolność do wykonywania algorytmów, a nie dostępne zasoby.
- Nie jest możliwe zbudowanie komputera o zdolności do rozwiązywania większej liczby problemów, niż najprostsza maszyna Turinga.

Wnioski

- Dowolny problem, mający rozwiązanie na znanych nam obecnie popularnych komputerach, można rozwiązać, konstruując odpowiadającą mu maszynę Turinga.
- Twierdzenie przeciwne nie jest prawdziwe: Nie jest prawdą, że jeśli pewien algorytm może wykonać maszyna Turinga to może go też wykonać komputer. Żaden komputer nie jest tak potężny jak maszyna Turinga, gdyż nie mają nieskończenie dużo pamięci oraz operują na zmiennych o ograniczonej wielkości.
- Wszystko jedno jak by konstrukcja maszyny Turinga nie była złożona[a] a ona sama realizowała nie wiadomo jak skomplikowane zadania, to cały problem można sprowadzić do bardzo elementarnych działań – manipulowania symbolami z pewnego ograniczonego alfabetu według ściśle określonych reguł.

[a] Złożona w sensie ilości stanów i definicji funkcji przejścia, bo sama maszyna jest bardzo prosta.

Wnioski

- Dowolny problem, mający rozwiązanie na znanych nam obecnie popularnych komputerach, można rozwiązać, konstruując odpowiadającą mu maszynę Turinga.
- Twierdzenie przeciwne nie jest prawdziwe: Nie jest prawdą, że jeśli pewien algorytm może wykonać maszyna Turinga to może go też wykonać komputer. Żaden komputer nie jest tak potężny jak maszyna Turinga, gdyż nie mają nieskończenie dużo pamięci oraz operują na zmiennych o ograniczonej wielkości.
- Wszystko jedno jak by konstrukcja maszyny Turinga nie była złożona[a] a ona sama realizowała nie wiadomo jak skomplikowane zadania, to cały problem można sprowadzić do bardzo elementarnych działań – manipulowania symbolami z pewnego ograniczonego alfabetu według ściśle określonych reguł.

[a] Złożona w sensie ilości stanów i definicji funkcji przejścia, bo sama maszyna jest bardzo prosta.

Wnioski

- Dowolny problem, mający rozwiązanie na znanych nam obecnie popularnych komputerach, można rozwiązać, konstruując odpowiadającą mu maszynę Turinga.
- Twierdzenie przeciwne nie jest prawdziwe: Nie jest prawdą, że jeśli pewien algorytm może wykonać maszyna Turinga to może go też wykonać komputer. Żaden komputer nie jest tak potężny jak maszyna Turinga, gdyż nie mają nieskończenie dużo pamięci oraz operują na zmiennych o ograniczonej wielkości.
- Wszystko jedno jak by konstrukcja maszyny Turinga nie była złożona[a] a ona sama realizowała nie wiadomo jak skomplikowane zadania, to cały problem można sprowadzić do bardzo elementarnych działań – manipulowania symbolami z pewnego ograniczonego alfabetu według ściśle określonych reguł.

[a] Złożona w sensie ilości stanów i definicji funkcji przejścia, bo sama maszyna jest bardzo prosta.

Wnioski

- Dowolny problem, mający rozwiązanie na znanych nam obecnie popularnych komputerach, można rozwiązać, konstruując odpowiadającą mu maszynę Turinga.
- Twierdzenie przeciwne nie jest prawdziwe: Nie jest prawdą, że jeśli pewien algorytm może wykonać maszyna Turinga to może go też wykonać komputer. Żaden komputer nie jest tak potężny jak maszyna Turinga, gdyż nie mają nieskończenie dużo pamięci oraz operują na zmiennych o ograniczonej wielkości.
- Wszystko jedno jak by konstrukcja maszyny Turinga nie była złożona[a] a ona sama realizowała nie wiadomo jak skomplikowane zadania, to cały problem można sprowadzić do bardzo elementarnych działań – manipulowania symbolami z pewnego ograniczonego alfabetu według ściśle określonych reguł.

[a]Złożona w sensie ilości stanów i definicji funkcji przejścia, bo sama maszyna jest bardzo prosta.

Bramka

Bramka jest układem elektronicznym, którego sygnał wyjściowy jest wynikiem operacji Boole'a na sygnałach wejściowych.

- Symbole
- Analiza działania

Bramka

Bramka jest układem elektronicznym, którego sygnał wyjściowy jest wynikiem operacji Boole'a na sygnałach wejściowych.

- Symbole
- Analiza działania

Bramka

Bramka jest układem elektronicznym, którego sygnał wyjściowy jest wynikiem operacji Boole'a na sygnałach wejściowych.

- Symbole
- Analiza działania

System funkcjonalnie pełny

Zbiór funkcji boolowskich nazywa się *systemem funkcjonalnie pełnym* (bazą), jeśli dowolna funkcja boolowska może być przedstawiona za pomocą funkcji należących do tego zbioru i argumentów funkcji.

Funkcje sumy (OR), iloczynu (AND) i negacji (NOT) tworzą tzw. *podstawowy system funkcjonalnie pełny*. Nie jest to jednak system minimalny. Systemy funkcjonalnie pełne tworzą również:

- iloczyn i negacja;
- suma i negacja;
- funkcja Sheffera (NAND)
- funkcja Peirce'a (NOR)

System funkcjonalnie pełny

Zbiór funkcji boolowskich nazywa się *systemem funkcjonalnie pełnym* (bazą), jeśli dowolna funkcja boolowska może być przedstawiona za pomocą funkcji należących do tego zbioru i argumentów funkcji.

Funkcje sumy (OR), iloczynu (AND) i negacji (NOT) tworzą tzw. *podstawowy system funkcjonalnie pełny*. Nie jest to jednak system minimalny. Systemy funkcjonalnie pełne tworzą również:

- iloczyn i negacja;
- suma i negacja;
- funkcja Sheffera (NAND)
- funkcja Peirce'a (NOR)

System funkcjonalnie pełny

Zbiór funkcji boolowskich nazywa się *systemem funkcjonalnie pełnym* (bazą), jeśli dowolna funkcja boolowska może być przedstawiona za pomocą funkcji należących do tego zbioru i argumentów funkcji.

Funkcje sumy (OR), iloczynu (AND) i negacji (NOT) tworzą tzw. *podstawowy system funkcjonalnie pełny*. Nie jest to jednak system minimalny. Systemy funkcjonalnie pełne tworzą również:

- iloczyn i negacja;
- suma i negacja;
- funkcja Sheffera (NAND)
- funkcja Peirce'a (NOR)

System funkcjonalnie pełny

Zbiór funkcji boolowskich nazywa się *systemem funkcjonalnie pełnym* (bazą), jeśli dowolna funkcja boolowska może być przedstawiona za pomocą funkcji należących do tego zbioru i argumentów funkcji.

Funkcje sumy (OR), iloczynu (AND) i negacji (NOT) tworzą tzw. *podstawowy system funkcjonalnie pełny*. Nie jest to jednak system minimalny. Systemy funkcjonalnie pełne tworzą również:

- iloczyn i negacja;
- suma i negacja;
- funkcja Sheffera (NAND)
- funkcja Peirce'a (NOR)

System funkcjonalnie pełny

Zbiór funkcji boolowskich nazywa się *systemem funkcjonalnie pełnym* (bazą), jeśli dowolna funkcja boolowska może być przedstawiona za pomocą funkcji należących do tego zbioru i argumentów funkcji.

Funkcje sumy (OR), iloczynu (AND) i negacji (NOT) tworzą tzw. *podstawowy system funkcjonalnie pełny*. Nie jest to jednak system minimalny. Systemy funkcjonalnie pełne tworzą również:

- iloczyn i negacja;
- suma i negacja;
- funkcja Sheffera (NAND)
- funkcja Peirce'a (NOR)

System funkcjonalnie pełny

Zbiór funkcji boolowskich nazywa się *systemem funkcjonalnie pełnym* (bazą), jeśli dowolna funkcja boolowska może być przedstawiona za pomocą funkcji należących do tego zbioru i argumentów funkcji.

Funkcje sumy (OR), iloczynu (AND) i negacji (NOT) tworzą tzw. *podstawowy system funkcjonalnie pełny*. Nie jest to jednak system minimalny. Systemy funkcjonalnie pełne tworzą również:

- iloczyn i negacja;
- suma i negacja;
- funkcja Sheffera (NAND)
- funkcja Peirce'a (NOR)

Maszyna analityczna (Charles Babbage, ok. 1833)

Maszyna ta miała składać się z

- magazynu (dzisiejszy odpowiednik pamięci),
- młyna (jednostka licząca),
- mechanizmu sterującego (jednostka sterująca).

- **Pamięć** miała służyć do przechowywania danych oraz wyników z przeprowadzanych na nich operacji.
- **Młyn**, odpowiednik dzisiejszej jednostki arytmetyczno-logicznej, miał wykonywać proste działania arytmetyczne.
- **Mechanizm sterujący** miał kierować działaniem całego urządzenia i w założeniach, miał być programowany.

Maszyna analityczna – dodatkowe informacje

- Siłą napędową maszyny miał być **silnik parowy**.
- Maszyna miała mieć ok. **30 metrów długości** i **10 metrów szerokości**.
- Dane wejściowe (program oraz dane) wprowadzane miały być za pomocą **kart dziurkowanych**, powszechnie wykorzystywanych w tamtym czasie przez krosna mechaniczne.
- Do sygnalizowania i udostępniania danych wyjściowe maszyna pisiadała a printer, a curve plotter and a bell. Dodatkowo wyniki mogły być przedstawione na kartach dziurkowanych w celu dalszego ich wykorzystania.
- Maszyna wykorzystywała stało przecinkową arytmetykę oraz system dziesiętny.
- Magazyn mógł pomieścić 1000 liczb 50-cio cyfrowych.
- Młyn mógł wykonać wszystkie cztery operacje arytmetyczne, porównanie oraz obliczyć pierwiastek kwadratowy.

Maszyna analityczna

Maszyna analityczna nie doczekała się realizacji praktycznej i z uwagi na proponowaną technologię realizacji (napędzanie silnikiem lokomotywy parowej, czysto mechaniczna, wysokiej złożoności konstrukcja) istnieje wątpliwość, czy konstrukcja tego typu byłaby zdolna do sprawnej, bezawaryjnej pracy. Jednakże jej konstrukcja posłużyła późniejszym twórcom do opracowania dzisiejszych komputerów.

Komputery pre-von Neumannowskie

Komputery pre-von Neumannowskie

Mimo idei zapoczątkowanej przez Babbage'a pierwsze zautomatyzowane maszyny liczące posiadały sztywno zakodowany program. W celu jego zmiany należało zmienić sieć połączeń elektrycznych, strukturę maszyny lub nawet przeprojektować ją. W istocie pierwsze „komputery" były nie tyle programowalne co przebudowywalne (nie były „zaprogramowane" ale „zaprojektowane"). Programowanie, jeśli w ogóle możliwe, było bardzo fizycznym procesem, rozpoczynającym się od wstępnego szkicowania diagramów przepływu danych i wielu notatek, przez projektowanie inżynierskie na żmudnym procesie przebudowywania maszyny kończąc.

Komputery pre-von Neumannowskie

Wspomniane problemy z „programowalnością” doprowadziły zapewne do narodzenia się ideii programu przechowywanego w komputerze (w strukturach jakie dostarczał a nie w sposobie budowy tych struktu lub ich wzajemnego oddziaływania). Powstanie architektur operujących na zestawie pewnych, zdefiniowanych i ściśle określonych instrukcji, uczyniło ten proces znacznie bardziej przyjaznym, efektywnym a zarazem elastycznym. Traktowanie instrukcji jak danych (bo instrukcje też miały być przechowywane w pamięci) pozwalało nawet na samomodyfikacje działającego programu.

Architektura von Neumanna (1945)

Wyróżniamy w niej trzy podstawowe części:

- procesor (w ramach którego wydzielona bywa część sterująca oraz część arytmetyczno-logiczna)
- pamięć komputera (zawierająca dane i sam program)
- urządzenia wejścia/wyjścia

Architektura von Neumanna (1945)

System komputerowy zbudowany w oparciu o architekturę von Neumanna powinien:

- mieć skończoną i funkcjonalnie pełną listę rozkazów;
- mieć możliwość wprowadzenia programu do systemu komputerowego poprzez urządzenia zewnętrzne i jego przechowywanie w pamięci w sposób identyczny jak danych;
- dane i instrukcje w takim systemie powinny być jednakowo dostępne dla procesora;
- informacja jest przetwarzana dzięki sekwencyjnemu odczytywaniu instrukcji z pamięci komputera i wykonywaniu tych instrukcji przez procesor.

Architektura von Neumanna (1945)

- Podane warunki pozwalają przestawić system komputerowy z wykonania jednego zadania (programu) na inne bez fizycznej ingerencji w strukturę systemu, a tym samym gwarantują jego uniwersalność.
- System komputerowy von Neumanna nie posiada oddzielnych pamięci do przechowywania danych i instrukcji.
- Instrukcje jak i dane są zakodowane w postaci liczb.
- Bez analizy programu trudno jest określić czy dany obszar pamięci zawiera dane czy instrukcje.
- Wykonywany program może się sam modyfikować traktując obszar instrukcji jako dane, a po przetworzeniu tych instrukcji – danych – zacząć je wykonywać.
- Model komputera wykorzystującego architekturę von Neumanna jest często nazywany **przykładową maszyną cyfrową**.

Von Neumann?

Termin **architektura von Neumanna** pochodzi od nazwiska matematyka **John von Neumann's** autora raportu First Draft of a Report on the EDVAC[a]. Datowany na **30 czerwca 1945 roku**, był wczesną pisemną notatką dotyczącą komputera ogólnego przeznaczenia przechowującego program w pamięci (ang. *a general purpose stored-program computing machine*) jakim miał być EDVAC. Jakkolwiek prace von Neumanna należy zaliczyć do pionierskich to już sam termin „architektura von Neumannowska" wydaje się krzywdzić zarówno osoby współpracujące z von Neumannem jak i te na których wynikach oparł swoje doświadczenia.

[a] Electronic Discrete Variable Automatic Computer

Von Neumann?

- Idea opisana prze von Neumanna pojawiła się już w **1936 w opatentowanym rozwiązaniu Konrada Zuseego.**
- Pomysł na stworzenie komputera przechowującego program w pamięci pojawiła się w The Moore School of Electrical Engineering at the University of Pennsylvania zanim jeszcze von Neumann usłyszał cokolwiek o ENIAC. Nie jest znany autor tej ideii.

Von Neumann?

- John William Mauchly oraz J. Presper Eckert pisali o ideii przechowywania programu w pamięci w **grudniu 1943** podczas ich prac nad ENIAC. Ponadto Grist Brainerd, dyrektor (?) (ang. project administrator) projektu ENIAC, w grudniu 1943 w raporcie z postępów prac nad ENIAC nie wprost proponował takie rozwiąznie (jednocześnie odrzucając je jako możliwe do zastosowania w ENIAC (?)) stwierdzając, że *„w celu uproszczenia projektu i niekomplikowania spraw”* ENIAC zostanie skonstruowany bez żadnych *„automatycznych regulacji”* (?).
- 19 lutego 1946 Alan Turing przedstawił dokument zawierający kompletny projekt komputera przechowującego kod w pamięci (ang. stored-program computer), Pilot ACE (jednego z pierwszych komputerów zbudowanych w Wielkiej Brytanii w The National Physical Laboratory (NPL) we wczesnych latach 50-tych).

Elementy współczesnego modelu

Elementy współczesnego modelu

- procesor (CPU – ang. central processing unit)
- pamięć
- wejście/wyjście (ang. input/output (I/O))
- łącznik w postaci magistral

Procesor

Procesor (ang. processor) nazywany często CPU (ang. Central Processing Unit) – urządzenie cyfrowe sekwencyjne potrafiące pobierać dane z pamięci, interpretować je i wykonywać jako rozkazy. Wykonuje on bardzo szybko ciąg prostych operacji (rozkazów) wybranych ze zbioru operacji podstawowych określonych zazwyczaj przez producenta procesora jako lista rozkazów procesora.

Zestaw instrukcji

Zestaw instrukcji (rozkazów) jest listą wszystkich instrukcji (we wszystkich przewidzianych przez architekturę dopuszczalnych wariantach) jakie procesor jest zdolny wykonać.
W skład instrukcji wchodzą:

1. instrukcje arytmetyczne, np. **add** czy **subtract**
2. instrukcje logiczne, np. **and**, **or** czy **not**
3. instrukcje operujące na danych[a], np. **move**, **input**, **output**, **load** czy **store**
4. instrukcje sterujące przepływem danych, np. **goto**, **if ... goto**, **call** czy **return**.

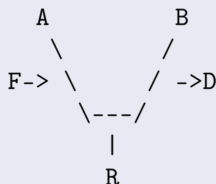
[a] W sensie traktowania ich jako sekwencji zer i jedynek bez wnikania w ich zanczenie

Procesor

W funkcjonalnej strukturze procesora można wyróżnić takie elementy, jak:

- zespół rejestrów do przechowywania danych i wyników (rejestry mogą być ogólnego przeznaczenia lub mają specjalne przeznaczenie),
- jednostkę arytmetyczną (arytmometr, ALU) do wykonywania operacji obliczeniowych na danych,
- jednostkę obsługi pamięci,
- jednostę obsługi instrukcji (pobieranie i dekodowanie).

A typical schematic symbol for an ALU



A,B - operandy (wejście)

R - wyjście

F - wejście z jednostki sterującej

D - wyjście statusu

Rejestry

Rejestry stanowią bardzo małą pamięć, do której procesor ma zdecydowanie najkrótszy czas dostępu. Większość współczesnych architektur przenoszeniu danych z pamięci operacyjnej do rejestrów, wykonaniu na nich operacji i przeniesieniu wyników do pamięci.

Wielkość rejestrów wyrażana jest w bitach, jakie mogą one pomieścić, np. rejestr 8-bitowy czy rejestr 32-bitowy.

Rodzaje rejestrów

Zazwyczaj procesor zawiera kilka rodzajów rejestrów, które można klasyfikować zależnie od ich zawartości lub rodzaju operacji jakie można na nich wykonać:

- Rejestry dostępne dla użytkownika, np. rejestry danych rejestry adresowe.
- Rejestry danych służą do operowania na danych, np. wartościach liczbowych będących liczbą całkowitą. Specjalny rejestr tego typu nazywany akumulatorem często wykorzystywany jest jako domyślny rejestr pewnych instrukcji.
- Rejestr adresowy służy do przechowywania adresów i jest wykorzystywany przez instrukcje posługujące się adresowaniem pośrednim.
- Rejestr flagowy - rodzaj rejestru warunkowego informującego o wykonaniu lub nie pewnej operacji lub o konieczności lub braku konieczności wykonania operacji.

Rodzaje rejestrów

- Rejestry ogólnego przeznaczenia są połączeniem rejestrów danych i adresowych.
- Rejestry zmiennoprzecinkowe przeznaczone do operowania na liczbach zmiennoprzecinkowych.
- Rejestry specjalnego przeznaczenia zazwyczaj opisują stan wykonania programu. Zazwyczaj są to
  - wspaźnik instrukcji (ang. *program counter, instruction pointer*),
  - wskaźnik stosu,
  - rejestr flagowy (ang. *status register, processor status word*).
- Rejestr instrukcji przechowuje obecnie wykonywaną i mające być niebawem wykonane instrukcje.
- Rejestry indeksowe modyfikujące adres operandów (argumentów) instrukcji.

Cykl pracy

Na cykl pracy składa się kilka cyklicznie powtarzanych operacji:

- pobranie (ang. *fetch*),
- dekodowanie (ang. *decode*),
- wykonanie (ang. *execute*),
- zapisanie wyników (ang. *writeback*).

Pobranie

The instruction that the CPU fetches from memory is used to determine what the CPU is to do.
The fetch step involves retrieving an instruction (which is represented by a number or sequence of numbers) from program memory. The location in program memory is determined by a program counter (PC), which stores a number that identifies the current position in the program. In other words, the program counter keeps track of the CPU's place in the current program. After an instruction is fetched, the PC is incremented by the length of the instruction word in terms of memory units. Often the instruction to be fetched must be retrieved from relatively slow memory, causing the CPU to stall while waiting for the instruction to be returned.

Decode

In the decode step, the instruction is broken up into parts that have significance to other portions of the CPU. The way in which the numerical instruction value is interpreted is defined by the CPU's instruction set architecture (ISA). Often, one group of numbers in the instruction, called the opcode, indicates which operation to perform. The remaining parts of the number usually provide information required for that instruction, such as operands for an addition operation. Such operands may be given as a constant value (called an immediate value), or as a place to locate a value: a register or a memory address, as determined by some addressing mode. In older designs the portions of the CPU responsible for instruction decoding were unchangeable hardware devices. However, in more abstract and complicated CPUs and ISAs, a microprogram is often used to assist in translating instructions into various configuration signals for the CPU. This microprogram is sometimes rewritable so that it can be modified to change the way the CPU decodes instructions even after it has been manufactured.

Execute

During the execute step, various portions of the CPU are connected so they can perform the desired operation. If, for instance, an addition operation was requested, an arithmetic logic unit (ALU) will be connected to a set of inputs and a set of outputs. The inputs provide the numbers to be added, and the outputs will contain the final sum. If the addition operation produces a result too large for the CPU to handle, an arithmetic overflow flag in a flags register may also be set.

Writeback

The final step, writeback, simply „writes back" the results of the execute step to some form of memory. Very often the results are written to some internal CPU register for quick access by subsequent instructions. In other cases results may be written to slower, but cheaper and larger, main memory. Some types of instructions manipulate the program counter rather than directly produce result data. These are generally called „jumps" and facilitate behavior like loops, conditional program execution (through the use of a conditional jump), and functions in programs. Many instructions will also change the state of digits in a „flags" register. These flags can be used to influence how a program behaves, since they often indicate the outcome of various operations. For example, one type of „compare" instruction considers two values and sets a number in the flags register according to which one is greater. This flag could then be used by a later jump instruction to determine program flow.

Adresowanie – analogia z życia wzięta

Odszukiwanie numeru telefonu: z naszej pamięci, z notatnika, od kolegi, z notatnika kolegi itd.

Adresowanie

Różne rodzaje adresowania

- natychmiastowe (opcode, argument),
- bezpośrednie (opcode, adres),
- pośrednie (opcode, pamięć/rejestr z adresem),
- pośrednie z użyciem rejestru bazyowego i offsetu (opcode, offset) (przy ustalonej bazie – rejestrze bazowym),
- pośrednie z użyciem rejestru bazowego i indeksowego (opcode) (przy ustalonej bazie – rejestrze bazowym i automatycznie zwiększanym rejestrze indeksowym).

Assembler

An assembly language is a low-level language for programming computers. It implements a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. This representation is usually defined by the hardware manufacturer, and is based on abbreviations (called mnemonics) that help the programmer remember individual instructions, registers, etc. An assembly language is thus specific to a certain physical or virtual computer architecture (as opposed to most high-level languages, which are portable).

Assembler

A program written in assembly language consists of a series of instructions mnemonics that correspond to a stream of executable instructions, when translated by an assembler, that can be loaded into memory and executed.

Assembler

Dla przykładu procesor architektury x86/IA-32 może wykonać następujący rozkaz zapisany w języku maszyny (ciąg zer i jedynek):

1. Binary: `10110000 01100001` (Hexadecimal: `0xb061`)

Odpowiednik tego rozkazu wyrażony w instrukcji asemblera jest znacznie łatwiejszy do zapamiętania (mnemoniczny[a])

1. `mov al, #061h`

Instrukcja ta oznacza:

1. Przenieś szesnastkową wartość 61 (97 dziesiętnie) do rejestru procesora oznaczonego jako *al*.

Mnemonik `mov` odpowiada kodowi operacji (ang. *opcode*) 1011, który powoduje przeniesienie wartości będącej drugim operandem do rejestru określonego przez pierwszy operand. Mnemonik wybrany został przez projektanta listy rozkazów jako skrót od *przenieś* (ang. *move*) co znacznie łatwiej zapamiętać. Lista argumentów rozdzielonych przecinkiem następuje za kodem instrukcji; jest to typowy zapis asemblera.

[a] Mnemonika – zespół sposobów ułatwiających zapamiętywanie nowego materiału.

Asembler

In practice many programmers drop the word mnemonic and, technically incorrectly, call „mov” an opcode. When they do this, they are referring to the underlying binary code which it represents. To put it another way, a mnemonic such as „mov” is not an opcode, but as it symbolizes an opcode, one might refer to „the opcode mov” for example when one intends to refer to the binary opcode it symbolizes rather than to the symbol–the mnemonic–itself.

Przykład kodu asemblera

```
mov ax, 0D625h   ;wprowadź do rejestru AX liczbę
                 ;szesnastkową D625 (54821 dziesiętnie)
mov es, ax       ;załaduj rejestr segmentowy ES wartością
                 ;znajdującą się w AX (D625)
mov al, 24       ;załaduj dolną (młodszą) połówkę rejestru
                 ;AX (AL) liczbą dziesiętną 24
mov ah, 0        ;wyzeruj górną (starszą) połówkę rejestru
                 ;AX (AH)
int 21h          ;wywołaj przerwanie 21H
```

Przykład programu w języku wysokiego poziomu...

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  printf("Witaj świecie\n");
  exit(0);
}
```

...i jego asemblerowa postać

```
.file   "main.c"
        .section        .rodata
.LC0:
        .string "Witaj \305\233wiecie"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $4, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, (%esp)
        call    exit
        .size   main, .-main
        .ident  "GCC: (GNU) 4.1.2 20060901 (prerelease) (Debian 4.1.1-1
        .section        .note.GNU-stack,"",@progbits
```