Architektura komputerów

Wykład 11
Teoretyczny model komputera – RISC

Wojciech Kordecki

Collegium Witelona
Wydział Nauk Technicznych i Ekonomicznych
Zakład Informatyki

Semestr letni 2023/24

Podziękowanie

Wykład został przygotowany na podstawie i z wykorzystaniem materiałów udostępnionych przez mgr. inż. Tomasza Wierzbickiego z Instytutu Informatyki Uniwersytetu Wrocławskiego.

Przy przygotowania slajdów skorzystałem z udostępnionego przez autora, TEX-owego tekstu źródłowego, z cytowaniem obszernych fragmentów opracowania i wykorzystaniem rysunków włącznie.

https://www.ii.uni.wroc.pl/~prz/200506/2006lato/cpp/zadania/sextium.pdf

Procesory CISC

Complex instruction set computer. Jeden cykl rozkazowy trwa kilkanaście lub kilkadziesiąt taktów zegara. Rozbudowana lista rozkazów może pomóc zoptymalizować kod wynikowy, jeśli programista pisze program w asemblerze, natomiast zbudowanie kompilatora języka wysokiego poziomu, który wykorzystywałby wiele rozkazów jest kłopotliwe.

W praktyce kompilatory używają tylko najprostszych rozkazów procesora CISC i dodatkowe możliwości takiego procesora związane z bardzo bogatą listą instrukcji nie są wykorzystywane (optymalizacja kodu jest trudna).

Procesory RISC

Reduced instruction set computer, procesor o ograniczonym zbiorze instrukcji: procesor w którym lista rozkazów jest maksymalnie skrócona.

Procesory RISC (posiadające kilkanaście do kilkudziesięciu rozkazów) są przy tej samej częstotliwości zegara kilkakrotnie bardziej efektywne niż procesory CISC (*complex instruction set computer*). Wynika to z faktu, że jeden cykl rozkazowy w takim procesorze trwa tylko jeden lub najwyżej kilka taktów zegara a nie kilkanaście czy kilkadziesiąt jak w procesorach typu CISC.

Rodziny x86 i ARM – porównanie

Istotną przewagą procesorów opartych na architekturze ARM nad procesorami Intel i innymi procesorami x86 jest zużycie energii. Okazuje się, że podejście RISC wraz ze specyficzną innowacją projektu ARM sprawia, że procesory są niezwykle oszczędne. Właśnie dlatego ARM zdominował rynek smartfonów i tabletów. Wydajność procesorów ARM rośnie wykładniczo z każdą generacją. Smartfony średniej klasy przekroczyły już próg „wystarczająco dobrego" pod względem mocy obliczeniowej i są wystarczająco wydajne, aby zaspokoić codzienne potrzeby użytkowników.

https://br.atsit.in/pl/?p=16798

Sextium

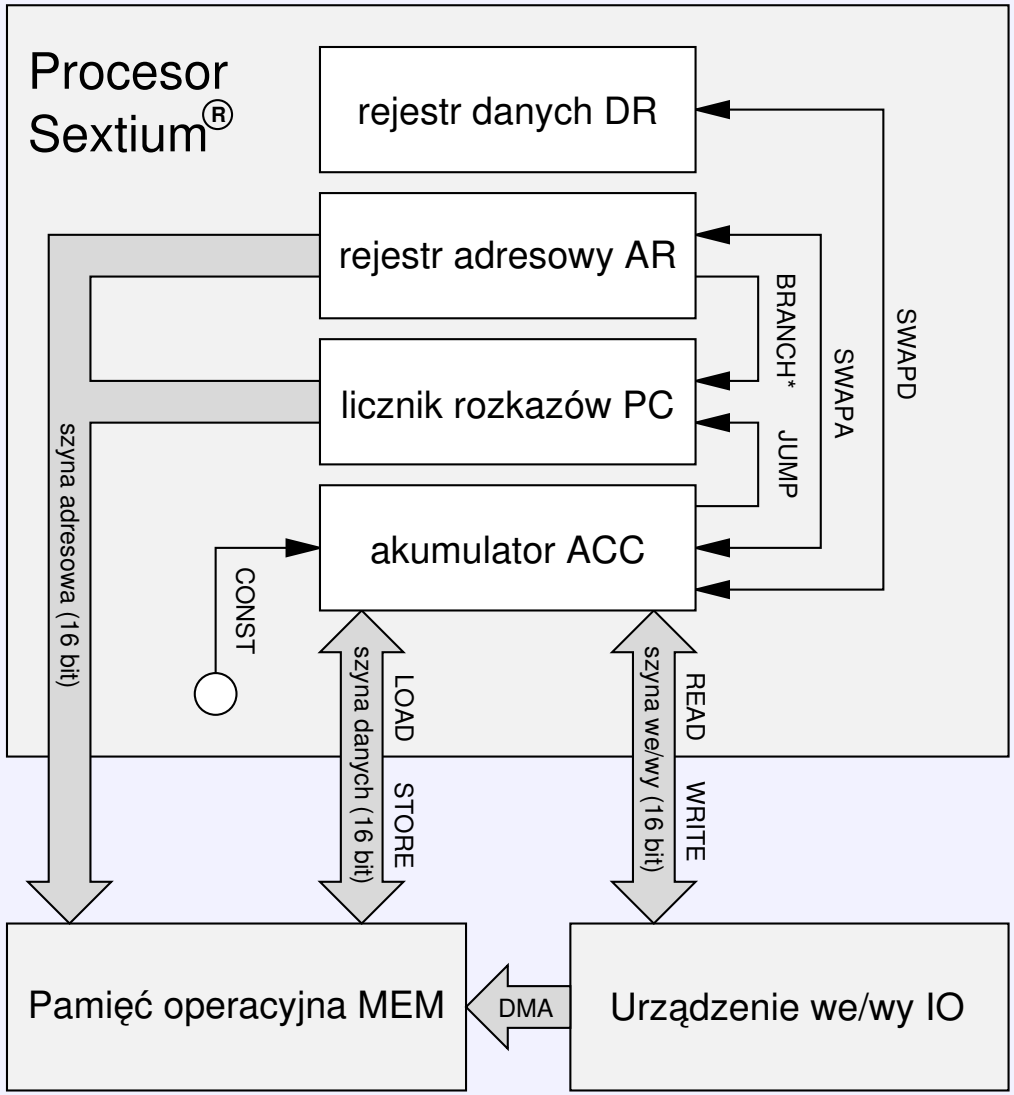
Procesor *Sextium* nie istnieje fizycznie. Jest uproszczonym modelem istniejących realnie procesorów typu RISC. Posiada jednak wszystkie podstawowe własności takich procesorów. Jego prostota pozwala na łatwiejsze zrozumienie idei procesorów RISC, niż analiza „prawdziwych" procesorów.

Architektura procesora Sextium II – opis

Poprzez szynę danych i szynę adresową do procesora jest podłączona pamięć (MEM) złożona z 64k słów 16-bitowych, a poprzez szynę wejścia/wyjścia – urządzenie wejścia/wyjścia (IO). Szyny: danych, adresowa i we/wy są 16-bitowe, podobnie jak rejestry:

- akumulator (ACC),
- rejestr danych (DR),
- rejestr adresowy (AR)
- licznik rozkazów (PC).

Architektura procesora Sextium II – schemat



Rejestry Sextium II

ACC, *accumulator*, akumulator: rejestr procesora, na którym są wykonywane operacje arytmetyczne i przesyłu danych.

DR, *data register*, rejestr danych: pomocniczy rejestr przechowujący drugi argument operacji arytmetycznych.

AR, *address register*, rejestr adresowy: rejestr, którego zawartość służy do adresowania pamięci w celu pobrania (zapisu) danych z (do) pamięci.

PC, *program counter*, licznik rozkazów: rejestr, którego zawartość służy do adresowania pamięci w celu pobrania kolejnych rozkazów do wykonania.

Pamięć i urządzenia wejścia/wyjścia Sextium II

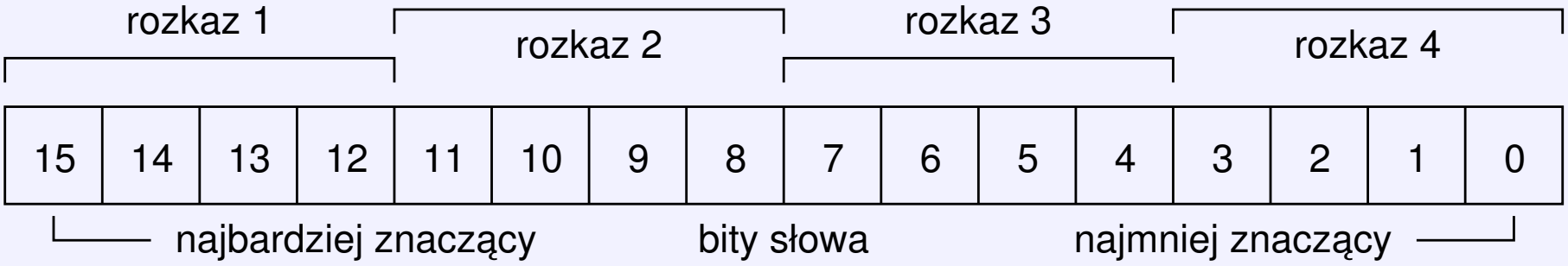
MEM, *memory*, pamięć operacyjna.

IO, *input/output*, urządzenie wejścia/wyjścia.

DMA, *direct memory access*, możliwość bezpośredniego zapisu do pamięci wprost z urządzenia wejścia/wyjścia bez udziału procesora.

Rozkazy

Rozkazy (jest ich 16) zajmują po 4 bity i są pakowane po 4 w słowie maszyny.



Zawartość pojedynczego słowa będzie opisana ciągiem czterech cyfr szesnastkowych. Dla wygody rozkazy mają swoje nazwy mnemoniczne. Zatem np. zamiast pisać „rozkaz `E`", piszemy „rozkaz `DIV`".

Dla procesora rozkaz, to cztery bity.

Cykl rozkazowy (1)

Cykl rozkazowy procesora polega na pobieraniu, dekodowaniu i wykonywaniu rozkazów. Wczytane słowo zawierające cztery rozkazy jest przechowywane w specjalnym *buforze rozkazów*. W celu pobrania rozkazów procesor odwołuje się do pamięci przeważnie raz na cztery cykle rozkazowe, chyba że wykonuje instrukcję skoku. Wówczas bufor rozkazów jest opróżniany.

Cykl rozkazowy (2)

Można zatem skoczyć jedynie do rozkazu, który zajmuje cztery najbardziej znaczące bity słowa (jeśli sprawia to jakieś problemy, można poprzednie słowo wypełnić rozkazami `NOP` w celu wyrównania wskazanego rozkazu do granicy słowa). Rozkaz `CONST` służy do załadowania stałej do akumulatora. Stała jest umieszczana w słowie następującym bezpośrednio po słowie zawierającym ten rozkaz (lub w następnych słowach, jeżeli w jednym słowie znajduje się więcej niż jeden rozkaz `CONST`).

Przykład. Następująca para słów: `6AB0 0001` zawiera ciąg rozkazów `SWAPD`, `CONST`, `ADD`, `NOP`, po którym następuje liczba 1. Wykonanie tych rozkazów powoduje zwiększenie zawartości akumulatora o jeden.

Cykl rozkazowy (3)

Podobnie ciąg sześciu słów:
`A6AD 0004 0005 6A56 FFFA 2000`
który symbolicznie zapiszemy jako:

```
CONST SWAPD CONST MUL
0004
0005
SWAPD CONST SWAPA SWAPD
FFFA
STORE NOP NOP NOP
```

powoduje załadowanie do rejestru danych liczby 4, do akumulatora liczby 5, przemnożenie zawartości akumulatora przez zawartość rejestru danych i zapisanie wyniku (liczby 20) do komórki pamięci o adresie `FFFA`.

Pobieranie i wykonywanie rozkazu

- Program dla procesora jest wpisywany do pamięci wprost z urządzenia wejścia/wyjścia (poprzez szynę DMA). W tym czasie procesor nie pracuje.
- Urządzenie wejścia/wyjścia wysyła sygnał do procesora, który zeruje zawartość wszystkich swoich rejestrów i opróżnia bufor rozkazów, a następnie rozpoczyna wykonywanie cyklu rozkazowego.
- Procesor przekazuje sterowanie do urządzenia wejścia/wyjścia po wykonaniu rozkazu `HALT` lub w razie próby dzielenia przez zero.
- Pracę procesora może również przerwać urządzenie wejścia/wyjścia.
- Po uruchomieniu licznik rozkazów ma wartość 0000.

Zestaw rozkazów procesora Sextium II (1)

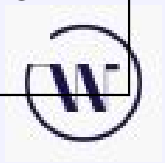
rozkaz	symb.	opis	uwagi
0	`NOP`	nic nie rób	przejdź do wykonania następnego rozkazu
1	`LOAD`	MEM[AR] $\rightarrow$ ACC	załaduj słowo o adresie znajdującym się w rejestrze adresowym do akumulatora
2	`STORE`	ACC $\rightarrow$ MEM[AR]	zapisz zawartość akumulatora do słowa o adresie znajdującym się w rejestrze adresowym
3	`READ`	IO $\rightarrow$ ACC	wczytaj słowo z urządzenia wejściowego do akumulatora
4	`WRITE`	ACC $\rightarrow$ IO	prześlij zawartość akumulatora do urządzenia wyjściowego

Zestaw rozkazów procesora Sextium II (2)

rozkaz	symb.	opis	uwagi
5	`SWAPA`	ACC $\leftrightarrow$ AR	zamień miejscami zawartość rejestru adresowego i akumulatora
6	`SWAPD`	ACC $\leftrightarrow$ DR	zamień miejscami zawartość rejestru danych i akumulatora
7	`JUMP`	ACC $\rightarrow$ PC	wpisz do licznika instrukcji zawartość akumulatora (skocz do instrukcji o adresie znajdującym się w akumulatorze)

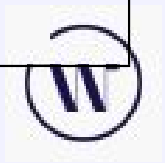
Zestaw rozkazów procesora Sextium II (3)

rozkaz	symb.	opis	uwagi
8	`BRANCHZ`	if ACC $= 0$ then AR $\rightarrow$ PC	jeżeli akumulator zawiera liczbę 0, zapisz do licznika instrukcji zawartość rejestru adresowego (skocz do instrukcji o adresie znajdującym się w rejestrze adresowym)
9	`BRANCHP`	if ACC > 0 then AR $\rightarrow$ PC	jeżeli akumulator zawiera liczbę dodatnią, zapisz do licznika instrukcji zawartość rejestru adresowego (skocz do instrukcji o adresie znajdującym się w rejestrze adresowym)



Zestaw rozkazów procesora Sextium II (4)

rozkaz	symb.	opis	uwagi
A	CONST	MEM[PC++] $\rightarrow$ ACC	zapisz słowo znajdujące się w komórce pamięci o adresie wskazywanym przez bieżącą zawartość licznika rozkazów do akumulatora
B	ADD	ACC + DR $\rightarrow$ ACC	dodaj do akumulatora zawartość rejestru danych
C	SUB	ACC − DR $\rightarrow$ ACC	odejmij od akumulatora zawartość rejestru danych
D	MUL	ACC $\times$ DR $\rightarrow$ ACC	pomnóż zawartość akumulatora przez zawartość rejestru danych



Zestaw rozkazów procesora Sextium II (5)

rozkaz	symb.	opis	uwagi
E	DIV	ACC / DR $\rightarrow$ ACC	podziel zawartość akumulatora przez zawartość rejestru danych
F	HALT	zatrzymaj	przekaż sterowanie do urządzenia we/wy

Algorytm pracy procesora Sextium II

wyzeruj rejestry ACC, PC, AR i DR i opróżnij bufor rozkazów
loop
  if bufor rozkazów jest pusty **then**
    pobierz słowo z pamięci o adresie zawartym w PC
    PC++
  end if
  $R \leftarrow$ odkoduj następny rozkaz
  if $R =$ `LOAD`, `STORE`, `READ`, `WRITE`, `SWAPA`, `SWAPD`, `ADD`, `SUB`, `MUL`, `DIV` **then**
    wykonaj tę operację zgodnie z opisem w tablicach
  else if $R =$ `CONST` **then**
    załaduj słowo z pamięci o adresie zawartym w PC do ACC
    PC++
  else if $R =$ `JUMP` **then**
    zapisz do PC zawartość ACC
    opróżnij bufor rozkazów
  else if ($R =$ `BRANCHZ` i `ACC` $= 0$) lub ($R =$ `BRANCHP` i `ACC` > 0) **then**
    zapisz do PC zawartość AR
    opróżnij bufor rozkazów
  else if $R =$ `HALT` **then**
    zatrzymaj pracę
  end if
end loop

Liczby

Słowa w operacjach arytmetycznych reprezentują liczby całkowite ze znakiem z przedziału $-2^{15} \div (2^{15} - 1)$ w zapisie uzupełnieniowym do dwóch, tj. zmiana znaku liczby polega na zanegowaniu wszystkich bitów liczby i dodaniu jedynki. Słowa od 0000 do 7FFF reprezentują liczby nieujemne 0 do 32767, słowa 8000 do FFFF zaś liczby -32768 do -1.

Jednostki leksykalne asemblera

Jednostkami leksykalnymi asemblera procesora Sextium II są *literały całkowitoliczbowe*, tj. niepuste ciągi cyfr dziesiętnych, poprzedzone opcjonalnie znakiem minus, np. 73,

Jednostki leksykalne asemblera

Jednostkami leksykalnymi asemblera procesora Sextium II są *literały całkowitoliczbowe*, tj. niepuste ciągi cyfr dziesiętnych, poprzedzone opcjonalnie znakiem minus, np. 73,

literały szesnastkowe, tj. ciągi dokładnie czterech cyfr szesnastkowych poprzedzone znakami `0x` lub `0X`, np. `0xFAFA`,

Jednostki leksykalne asemblera

Jednostkami leksykalnymi asemblera procesora Sextium II są *literały całkowitoliczbowe*, tj. niepuste ciągi cyfr dziesiętnych, poprzedzone opcjonalnie znakiem minus, np. 73,

literały szesnastkowe, tj. ciągi dokładnie czterech cyfr szesnastkowych poprzedzone znakami `0x` lub `0X`, np. `0xFAFA`,

słowa kluczowe:
`ADD BRANCHP BRANCHZ CONST DATA DIV HALT JUMP LOAD MUL READ STORE SUB SWAPA SWAPD WRITE`

Jednostki leksykalne asemblera

Jednostkami leksykalnymi asemblera procesora Sextium II są *literały całkowitoliczbowe*, tj. niepuste ciągi cyfr dziesiętnych, poprzedzone opcjonalnie znakiem minus, np. 73,

literały szesnastkowe, tj. ciągi dokładnie czterech cyfr szesnastkowych poprzedzone znakami `0x` lub `0X`, np. `0xFAFA`,

słowa kluczowe:
`ADD BRANCHP BRANCHZ CONST DATA DIV HALT JUMP LOAD MUL READ STORE SUB SWAPA SWAPD WRITE`

identyfikatory, tj. ciągi liter i cyfr zaczynające się literą i różne od słów kluczowych,

Jednostki leksykalne asemblera

Jednostkami leksykalnymi asemblera procesora Sextium II są *literały całkowitoliczbowe*, tj. niepuste ciągi cyfr dziesiętnych, poprzedzone opcjonalnie znakiem minus, np. 73,

literały szesnastkowe, tj. ciągi dokładnie czterech cyfr szesnastkowych poprzedzone znakami `0x` lub `0X`, np. `0xFAFA`,

słowa kluczowe:
`ADD BRANCHP BRANCHZ CONST DATA DIV HALT JUMP LOAD MUL READ STORE SUB SWAPA SWAPD WRITE`

identyfikatory, tj. ciągi liter i cyfr zaczynające się literą i różne od słów kluczowych,

symbol pomocniczy „:”.

Identyfikatory i komentarze

Identyfikatory są używane jako *etykiety*.

Wielkie i małe litery są rozróżniane w identyfikatorach i słowach kluczowych, natomiast cyfry szesnastkowe `A`–`F` można pisać zarówno wielką, jak i małą literą.

Znakiem początku komentarza jest #. Komentarz rozciąga się do końca wiersza w którym występuje (do znaku nowego wiersza).

Struktura programu (1)

Program w asemblerze jest ciągiem *instrukcji asemblera*, nie więcej niż jednej w wierszu. Instrukcja może się składać z pojedynczego słowa kluczowego

`ADD BRANCHP BRANCHZ DATA DIV HALT JUMP LOAD MUL READ STORE SUB SWAPA SWAPD WRITE`

lub może być postaci

`CONST` *parametr*

gdzie *parametr* to albo literał dziesiętny lub szesnastkowy, albo identyfikator (etykieta). Każda instrukcja może być opcjonalnie poprzedzona napisem postaci

etykieta :

Struktura programu (2)

Rozkaz procesora `NOP` *nie jest* instrukcją asemblera, natomiast instrukcja `DATA` nie ma odpowiednika wśród rozkazów procesora. Identyfikatory (etykiety) są symbolicznymi reprezentacjami adresów pamięci. Każda etykieta powinna pojawić się dokładnie raz w programie przed dwukropkiem i może pojawiać się wielokrotnie w instrukcji `CONST`.

Zadaniem asemblera jest przetłumaczenie symbolicznych nazw instrukcji na odpowiadające im rozkazy, spakowanie rozkazów po cztery w 16-bitowe słowa (wypełniając puste miejsca rozkazami `NOP`), wyznaczenie faktycznych adresów instrukcji opatrzonych etykietami i wygenerowanie kodu binarnego.

Działanie asemblera (1)

Asembler generuje rozkazy w takiej kolejności, w jakiej odpowiednie instrukcje znajdują się w tekście programu.
Instrukcja `CONST` powoduje wygenerowanie rozkazu ładującego stałą opisaną podanym literałem lub etykietą do akumulatora.
Instrukcja `DATA` powoduje zarezerwowanie słowa w pamięci w miejscu, w którym występuje (dlatego zwykle umieszcza się ją na końcu programu).
Do tego słowa w pamięci wpisywana jest liczba `0x0000` (tj. pamięć dla zmiennych jest inicjowana podczas uruchamiania programu).
Do adresu tego słowa można się odwoływać poprzez etykietę tej instrukcji (zatem użycie rozkazu `DATA` bez etykiety ma niewiele sensu).

Działanie asemblera (2)

Program tłumaczący program w asemblerze na kod wynikowy jest dwuprzebiegowy: w pierwszym przebiegu tworzy się kod wynikowy ”‘na próbę”, nie wstawiając do niego stałych opisanych przez etykiety (nie są one bowiem jeszcze znane).
W tym przebiegu ustala się ich wartości.
W drugim przebiegu generuje się ostatecznie kod wynikowy, ponieważ wartości etykiet są już ustalone.

Kod hex → asembler – drobny przykład

A532:

`CONST` – zapisz słowo znajdujące się w komórce pamięci o adresie wskazywanym przez bieżącą zawartość licznika rozkazów do akumulatora

`SWAPA` – zamień miejscami zawartość rejestru adresowego i akumulatora

`READ` – wczytaj słowo z urządzenia wejściowego do akumulatora

`STORE` – zapisz zawartość akumulatora do słowa o adresie znajdującym się w rejestrze adresowym

Algorytm Euklidesa – największy wspólny podzielnik

Szukamy NWD liczb x i y.

1. Oblicz z jako resztę z dzielenia x przez y.
2. Zastąp x liczbą y, następnie y liczbą z.
3. Jeżeli wartość y wynosi 0, to x jest szukaną wartością NWD, w przeciwnym wypadku przejdź do kroku 1.

Przykład.
Dane $x = 36$ i $y = 30$.
Reszta $36/30 = 1 \mod 6$. $z = 6$.
$x = 30$. $y = 6$.
$30/6 = 5 \mod 0$. $z = 0$.
$x = 6$

Algorytm Euklidesa – implementacja

wczytaj x
wczytaj y
while $y \neq 0$ **do**
  $z \leftarrow x \bmod y$
  $x \leftarrow y$
  $y \leftarrow z$
end while
wypisz x

Implementacje w różnych językach programowania:
https://pl.wikipedia.org/wiki/Algorytm_Euklidesa
https://pl.wikibooks.org/wiki/Kody_%C5%BAr%C3%B3d%C5%82owe/Algorytm_Euklidesa

C/C++

```
while (i != j)
{
    if (i > j)
        i -= j;
    else
        j -= i;
}
```

ARM

```
loop CMP    Ri, Rj    ; porównaj i z j, ustawiając
                      ; flagi warunkowe:
                      ;   - "GT" dla (i > j)
                      ;   - "LT" dla (i < j)
                      ;   - "NE" dla (i != j)
  SUBGT  Ri, Ri, Rj   ; jeśli "GT", wykonaj i := i - j;
  SUBLT  Rj, Rj, Ri   ; jeśli "LT", wykonaj j := j - i;
  BNE    loop         ; jeśli "NE",
                      ; wykonaj skok do etykiety 'loop'
```

NASM x86 (1)

```
section .text
global getgcd
getgcd:
  push ebp
  mov  ebp,esp
  mov  eax,[ebp+8]
  mov  ebx,[ebp+12]
petla:
  cmp  eax,ebx
  jg   wiekszy
  jl   mniejszy
  jmp  koniec
wiekszy:
  sub  eax,ebx
  jmp  petla
```

NASM x86 (2)

```
mniejszy:
   sub   ebx,eax
   jmp   petla
koniec:
   mov   esp,ebp
   pop   ebp
   ret
```

Sextium II

```
# Program gcd.asm
# Wczytuje dwie liczby
# i wypisuje
# ich największy
#wspólny dzielnik
        CONST x
        SWAPA
        READ
        STORE
        CONST y
        SWAPA
        READ
        STORE
dalej:  CONST y # y=0?
        SWAPA
        LOAD
        SWAPD
        CONST koniec
        SWAPA
        SWAPD
        BRANCHZ
        SWAPD   # z=x/y
        CONST x
        SWAPA
        LOAD
        DIV
        MUL     # z=z*y
        SWAPD   # z=x-z
        CONST x
        SWAPA
        LOAD
        SUB
        SWAPD
        CONST z
        SWAPA
        SWAPD
        STORE
        CONST y # x=y
        SWAPA
        LOAD
        SWAPD
        CONST x
        SWAPA
        SWAPD
        STORE
        CONST z # y=z
        SWAPA
        LOAD
        SWAPD
        CONST y
        SWAPA
        SWAPD
        STORE
        CONST dalej
        JUMP
koniec: CONST x # pisz x
        SWAPA
        LOAD
        WRITE
        HALT
     x: DATA # zmienne
     y: DATA
     z: DATA
```

Tłumaczenie programu na kod procesora Sextium II (1)

	adres	słowo	opis zawartości słowa
	0000	A532	CONST SWAPA READ STORE
	0001	001C	adres reprezentowany przez zmienną x
	0002	A532	CONST SWAPA READ STORE
	0003	001D	adres reprezentowany przez zmienną y
dalej =	0004	9516	CONST SWAPA LOAD SWAPD
	0005	001D	adres reprezentowany przez zmienną y
	0006	A568	CONST SWAPA SWAPD BRANCHZ
	0007	0019	adres reprezentowany przez zmienną koniec
	0008	6A51	SWAPD CONST SWAPA LOAD
	0009	001C	adres reprezentowany przez zmienną x
	000A	ED6A	DIV MUL SWAPD CONST
	000B	001C	adres reprezentowany przez zmienną x
	000C	51C6	SWAPA LOAD SUB SWAPD
	000D	A562	CONST SWAPA SWAPD STORE
	000E	001E	adres reprezentowany przez zmienną z

Tłumaczenie programu na kod procesora Sextium II (2)

	adres	słowo	opis zawartości słowa
	000F	A516	CONST SWAPA LOAD SWAPD
	0010	001D	adres reprezentowany przez zmienną y
	0011	A562	CONST SWAPA SWAPD STORE
	0012	001C	adres reprezentowany przez zmienną x
	0013	A516	CONST SWAPA LOAD SWAPD
	0014	001E	adres reprezentowany przez zmienną z
	0015	A562	CONST SWAPA SWAPD STORE
	0016	001D	adres reprezentowany przez zmienną y
	0017	A700	CONST JUMP NOP NOP
	0018	0004	adres reprezentowany przez zmienną dalej
koniec =	0019	A514	CONST SWAPA LOAD WRITE
	001A	001C	adres reprezentowany przez zmienną x
	001B	F000	HALT NOP NOP NOP
x =	001C	0000	miejsce przechowywania zmiennej x
y =	001D	0000	miejsce przechowywania zmiennej y
z =	001E	0000	miejsce przechowywania zmiennej z

Kod hex

Czysty kod hex:

```
A532 001C A532 001D 9516 001D A568 0019
6A51 001C ED6A 001C 51C6 A562 001E A516
001D A562 001C A516 001E A562 001D A700
0004 001C F000 0000 0000 0000
```

Format hex Intela

Linie z wyjątkiem ostatniej:

- Pierwszy znak (:) oznacza początek linii danych.
- Następne dwa znaki oznaczają długość danych w linii (10h w tym przypadku).
- Następne cztery znaki oznaczają adres zapisywania danych (0080h w tym przypadku).
- Następne dwa znaki oznaczają typ danych.
- Dalej następują dane.
- Ostatnie dwa znaki są sumą kontrolną, tzn. suma wszystkich znaków (cyfr) w linii jest podzielna przez $256 = 100H$.

Typy danych:

- 00 – dane.
- 01 – koniec pliku,

Kod hex w formacie Intela

Czysty kod hex – 30 słów:

```
A532 001C A532 001D 9516 001D A568 0019
6A51 001C ED6A 001C 51C6 A562 001E A516
001D A562 001C A516 001E A562 001D A700
0004 001C F000 0000 0000 0000
```

Kod w formacie hex Intela – ładowany od adresu 0:

```
:10000000A532001CA532001D9516001DA5680019A8
:100001006A51001CED6A001C51C6A562001EA51651
:10000200001DA562001CA516001EA562001DA70073
:0C000400001CF0000000000000E0
:00000001FF
```