Laboratorium programowania niskopoziomowego

LAB 3 – Instrukcje warunkowe, instrukcja wyboru, pętle.

Instrukcje warunkowe

Instrukcje warunkowe wykonują zadany kod w zależności od tego, czy sprawdzany warunek jest prawdziwy, czy fałszywy. Złożone wyrażenia warunkowe mogą być łączone z wykorzystaniem instrukcji logicznych **AND**, **OR**, **XOR** i **NOT**. Porównanie wartości liczbowych wykonuje się instrukcją **CMP**, natomiast badanie wartości logicznych i bitów realizuje się instrukcją **TEST**.

Należy zwrócić uwagę na fakt, że wiele instrukcji ustawia flagi w zależności od wyniku ich działania; flagi te można wykorzystać do wykonania skoku warunkowego, bez potrzeby użycia instrukcji *cmp* lub *test*.

W poniższych tabelach zostały przedstawione instrukcje skoków warunkowych.

Tabela 1. Instrukcje skoków warunkowych opartych o pojedyncze flagi:

JZ	skok jeżeli zero	JNZ	skok jeżeli nie zero
JC	skok jeżeli przeniesienie	JNC	skok jeżeli nie ma przeniesienia
JP, JPE	skok jeżeli parzystość	JNP, JPO	skok jeżeli nie parzystość
JO	skok jeżeli przepełnienie	JNO	skok jeżeli nie ma przepełnienia
JS	skok jeżeli znak (ujemny)	JNS	skok jeżeli nie ma znaku (dodatni)

Tabela 2. Instrukcje skoków warunkowych stosowane przy porównywaniu liczb:

Liczby ze znakiem		**Liczby bez znaku**	
JE	skok jeżeli vleft = vright	JE	skok jeżeli vleft = vright
JNE	skok jeżeli vleft != vright	JNE	skok jeżeli vleft != vright
JL, JNGE	skok jeżeli vleft < vright	JB, JNAE	skok jeżeli vleft < vright
JLE, JNG	skok jeżeli vleft <= vright	JBE, JNA	skok jeżeli vleft <= vright
JG, JNLE	skok jeżeli vleft > vright	JA, JNBE	skok jeżeli vleft > vright
JGE, JNL	skok jeżeli vleft >= vright	JAE, JNB	skok jeżeli vleft >= vright

Przykładowo poniższa instrukcja warunkowa sprawdza, czy wartość w rejestrze EAX/RAX jest większa lub równa 3. Jeżeli warunek jest fałszywy sterowanie zostanie przeniesione do instrukcji po etykiecie **skokJezeliFalsz,** w przypadku warunku prawdziwego wykona się przypisanie 1 do rejestru EBX/RBX.

nr	X86 **IF (eax >= 3)** *{ebx = 1}*	X64 **IF (rax >= 3)** *{rbx = 1}*
1 *2* *3* *4*	**cmp eax, 3;** **jl skokJezeliFalsz;** *lub jnge* *mov ebx, 1;* **skokJezeliFalsz:**	**cmp rax, 3;** **jl skokJezeliFalsz;** *lub jnge* *mov rbx, 1;* **skokJezeliFalsz:**

Następna instrukcja warunkowa sprawdza, czy wartość w rejestrze EAX/RAX jest większa lub równa 3. Jeżeli warunek jest prawdziwy sterowanie zostanie przeniesione do instrukcji po etykiecie **skokJezeliPrawda,** w przeciwnym przypadku zostanie wykonany blok instrukcji poniżej i zakończy się skokiem bezwarunkowym do etykiety **pomin** na końcu instrukcji warunkowej.

nr	X86 **IF (eax >= 3)** *{ebx = 1}* **ELSE** *{ebx = 2}*	X64 **IF (rax >= 3)** *{rbx = 1}* **ELSE** *{rbx = 2}*
1	**cmp eax, 3;**	**cmp rax, 3;**
2	**jge skokJezeliPrawda;**	**jge skokJezeliPrawda;**
3	*mov ebx, 2;*	*mov rbx, 2;*
4	**jmp pomin;**	**jmp pomin;**
5	**skokJezeliPrawda:**	**skokJezeliPrawda:**
6	*mov ebx, 1;*	*mov rbx, 1;*
7	**pomin:**	**pomin:**

W zależności od potrzeby można odwracać warunki i stosować inne instrukcje skoku, jak pokazano poniżej.

nr	X86 **IF (eax >= 3)** *{ebx = 1}* **ELSE** *{ebx = 2}* **IF (eax < 3)** *{ebx = 2}* **ELSE** *{ebx = 1}*	X64 **IF (rax >= 3)** *{rbx = 1}* **ELSE** *{rbx = 2}* **IF (rax < 3)** *{rbx = 2}* **ELSE** *{rbx = 1}*
1	**cmp eax, 3;**	**cmp rax, 3;**
2	**jl skokJezeliPrawda;**	**jl skokJezeliPrawda;**
3	*mov ebx, 1;*	*mov rbx, 1;*
4	**jmp pomin;**	**jmp pomin;**
5	**skokJezeliPrawda:**	**skokJezeliPrawda:**
6	*mov ebx, 2;*	*mov rbx, 2;*
7	**pomin:**	**pomin:**

nr	X86 **IF (eax >= 3)** *{ebx = 1}* **ELSE** *{ebx = 2}* **IF** (!**(eax >= 3))** *{ebx = 2}* **ELSE** *{ebx =1}*	X64 **IF (rax >= 3)** *{rbx = 1}* **ELSE** *{rbx = 2}* **IF** (!**(rax >= 3))** *{rbx = 2}* **ELSE** *{rbx =1}*
1	**cmp eax, 3;**	**cmp rax, 3;**
2	**jnge jezeliPrawda;**	**jnge jezeliPrawda;**
3	*mov ebx, 1;*	*mov rbx, 1;*
4	**jmp pomin;**	**jmp pomin;**
5	**jeżeliPrawda:**	**jeżeliPrawda:**
6	*mov ebx, 2;*	*mov rbx, 2;*
7	**pomin:**	**pomin:**

Różnice w implementacji w różnych architekturach (x86 i x64) sprowadzają się do nazw rejestrów.

Należy tutaj zaznaczyć, że w systemie 64-bitowym można używać rejestrów 32-bitowych (młodszej połowy rejestrów 64-bitowych). Tak samo w programach 32- i 64-bitowych można używać również rejestrów 16-bitowych i 8-bitowych. Jednakże należy pamiętać, że zmniejsza się wówczas zakres danych liczbowych tych rejestrów, co może skutkować błędami przekroczenia zakresu.

Zadanie 1:
Proszę napisać funkcję w asemblerze odpowiadającą poniższemu kodowi w C/C++:

```
int if_1(int a, int b)
{
     int x=0;
     if( a < b )
          x = b - a;
     return x;
}
```

Zadanie 2:
Proszę napisać funkcję w asemblerze odpowiadającą poniższemu kodowi w C/C++:

```
int if_2(int a, int b)
{
     int x=0;
     if( a < b )
          x = b - a;
     else
          x = a - b;
     return x;
}
```

Zadanie 3:
Proszę napisać funkcję w asemblerze odpowiadającą poniższemu kodowi w C/C++:

```
int if_3(int a, int b)
{
     int x=0;
     if( a < b )
          x = b*b - a;
     return x;
}
```

Zadanie 4:
Proszę napisać funkcję w asemblerze odpowiadającą poniższemu kodowi w C/C++:

```
int if_4(int a, int b)
{
     int x=0;
     if( a < b )
          x = a*a + b;
     else
          x = a*a - b;
     return x;
}
```

Po rozwiązaniu zadań proszę spróbować zastosować instrukcję CMOV powyższych zadań.

Instrukcja wyboru

Konstrukcja SWITCH, tak samo jak konstrukcja IF, wykorzystuje skoki warunkowe - głównie skoki JE. Na początku umieszczona jest sekcja sprawdzająca wszystkie warunki (linie 1..10). Dalsza część, to ciała kolejnych bloków warunkowych z kodem wpisanym przez programistę (linie 11..24). Kończą się one instrukcją skoku bezwarunkowego, która odpowiada instrukcji „break” i pomija pozostałe warunki.

nr	X86	X64
1	**switch (a)**	**switch (a)**
2	**{**	**{**
3	**case 1: {/*…*/} break;**	**case 1: {/*…*/} break;**
4	**case 2: {/*…*/} break;**	**case 2: {/*…*/} break;**
5	**case 3: {/*…*/} break;**	**case 3: {/*…*/} break;**
6	**case 4: {/*…*/} break;**	**case 4: {/*…*/} break;**
7	**default: {/*…*/} break;**	**default: {/*…*/} break;**
8	**}**	**}**
1	**mov eax, a;**	**mov rax, a;**
2	**cmp eax, 1;**	**cmp rax, 1;**
3	**je case1;**	**je case1;**
4	**cmp eax, 2;**	**cmp rax, 2;**
5	**je case2;**	**je case2;**
6	**cmp eax, 3;**	**cmp rax, 3;**
7	**je case3;**	**je case3;**
8	**cmp eax, 4;**	**cmp rax, 4;**
9	**je case4;**	**je case4;**
10	**jmp default;**	**jmp default;**
11	**case1:**	**case1:**
12	*; …*	*; …*
13	**jmp exit;**	**jmp exit;**
14	**case2:**	**case2:**
15	*; …*	*; …*
16	**jmp exit;**	**jmp exit;**
17	**case3:**	**case3:**
18	*; …*	*; …*
19	**jmp exit;**	**jmp exit;**
20	**case4:**	**case4:**
21	*; …*	*; …*
22	**jmp exit;**	**jmp exit;**
23	**default:**	**default:**
24	*; …*	*; …*
25	**exit:**	**exit:**

Zadanie 5:
Proszę napisać funkcję w asemblerze odpowiadającą poniższemu kodowi w C/C++:

```
int case_1(int x, int op)
{
     int wynik;
     switch( op )
     {
     case 3:   wynik = x + 7;
               break;
     case 5:   wynik = x * (x – 7);
               break;
     case 7:   wynik = x * x;
               break;
     default:  wynik = 0;
     }
     return wynik;
}
```

Zadanie 6:
Proszę napisać funkcję w asemblerze odpowiadającą poniższemu kodowi w C/C++:

```
int case_2(int x, int op)
{
     int wynik;
     switch( op )
     {
     case 0:   wynik = x + 7;
               break;
     case 1:   wynik = x * (x + 3);
               break;
     case 2:   wynik = x * x + 2 * x;
               break;
     case 3:   wynik = x * x - 12;
               break;
     default:  wynik = 0;
     }
     return wynik;
}
```

Pętle

Pętle w assemblerze są ściśle połączone z instrukcjami warunkowymi i instrukcjami skoku. Poniżej został przedstawiony przykład implementacji pętli **FOR(){…}** zrealizowany na kilka sposobów (oczywiście nie są to wszystkie możliwości).

Poniżej przedstawiona jest pętla FOR z inkrementacją:

nr	X86	X64
1	**int N = 100;**	**__int64 N = 100;**
2	**for (int i=0; i<N; ++i)**	**for (__int64 i=0; i<N; ++i)**
3	**{**	**{**
4	**//…**	**//…**
5	**}**	**}**
1	**mov ecx, 0;**	**mov rcx, 0;**
2	**petlaFOR:**	**petlaFOR:**
3	**cmp ecx, N;**	**cmp rcx, N;**
4	**jge koniecFOR;**	**jge koniecFOR;**
5	*;…*	*;…*
6	**inc ecx;**	**inc rcx;**
7	**jmp petlaFOR**	**jmp petlaFOR**
8	**koniecFOR:**	**koniecFOR:**
	O P T Y M A L I Z A C J A	
1	**xor ecx, ecx;**	**xor rcx, rcx;**
2	**mov ebx, N;**	**mov rbx, N;**
3	**cmp ecx, ebx;**	**cmp rcx, rbx;**
4	**jge pominFOR;**	**jge pominFOR;**
5	**petlaFOR:**	**petlaFOR:**
6	*;…*	*;…*
7	**inc ecx;**	**inc rcx;**
8	**cmp ecx, ebx;**	**cmp rcx, rbx;**
9	**jl petlaFOR;**	**jl petlaFOR;**
10	**pominFOR:**	**pominFOR:**
1	**ebx = N; Pseudokod**	**rbx = N; Pseudokod**
2	**ecx = 0;**	**rcx = 0;**
3	**If (ecx < ebx)**	**If (rcx < rbx)**
4	**{**	**{**
5	**do{**	**do{**
6	**//…**	**//…**
7	**ecx++;**	**rcx++;**
8	**}while (ecx < ebx);**	**}while (rcx < rbx);**
9	**}**	**}**

Pętla FOR z dekrementacją:

nr	X86	X64
1	**int N = 100;**	**__int64 N = 100;**
2	**for (int i=N; i>=0; --i)**	**for (__int64 i=N; i>=0; --i)**
3	**{**	**{**
4	**//…**	**//…**
5	**}**	**}**
1	**mov ecx, N;**	**mov rcx, N;**
2	**petlaFOR:**	**petlaFOR:**
3	**cmp ecx, 0;**	**cmp rcx, 0;**
4	**jl koniecFOR;**	**jl koniecFOR;**
5	*;…*	*;…*
6	**dec ecx;**	**dec rcx;**
7	**jmp petlaFOR**	**jmp petlaFOR**
8	**koniecFOR:**	**koniecFOR:**
	O P T Y M A L I Z A C J A 1	
1	**mov ecx, N;**	**mov rcx, N;**
2	**cmp ecx, 0;**	**cmp rcx, 0;**
3	**jl pominFOR;**	**jl pominFOR;**
4	**petlaFOR:**	**petlaFOR:**
5	*;…*	*;…*
6	**dec ecx;**	**dec rcx;**
7	**cmp ecx, 0;**	**cmp rcx, 0;**
8	**jge petlaFOR;**	**jge petlaFOR;**
9	**pominFOR:**	**pominFOR:**
1	**ecx = N;**	**rcx = N;**
2	**If (ecx >= 0)**	**If (rcx >= 0)**
3	**{**	**{**
4	**do{**	**do{**
5	*//…*	*//…*
6	**ecx--;**	**rcx--;**
7	**}while (ecx >= 0);**	**}while (rcx >= 0);**
8	**}**	**}**
	O P T Y M A L I Z A C J A 2	
1	**mov ecx, N;**	**mov rcx, N;**
2	**cmp ecx, 0;**	**cmp rcx, 0;**
3	**jl pominFOR;**	**jl pominFOR;**
4	**inc ecx;**	**inc rcx;**
5	**petlaFOR:**	**petlaFOR:**
6	*;…*	*;…*
7	**loop petlaFOR;**	**loop petlaFOR;**
	pominFOR:	**pominFOR:**

	O P T Y M A L I Z A C J A 3	
1	**mov ecx, N;**	**mov rcx, N;**
2	**cmp ecx, 0;**	**cmp rcx, 0;**
3	**jl pominFOR;**	**jl pominFOR;**
4	**petlaFOR:**	**petlaFOR:**
5	*;…*	*;…*
6	**dec ecx;**	**dec rcx;**
7	**jns petlaFOR;**	**jns petlaFOR;**
8	**pominFOR:**	**pominFOR:**

Zadanie 7:
Proszę napisać funkcję w asemblerze odpowiadającą poniższemu kodowi w C/C++:

```
int for_1(int N)
{
     int wynik=0;
     for(int i=0;i<N;i++)
     {
          wynik+=2*i;
     }
     return wynik;
}
```

Zadanie 8:
Proszę napisać funkcję w asemblerze obliczającą silnię za pomocą pętli for.

Implementacja pętli WHILE(){…}

nr	X86	X64
1	**int N = 100;**	**__int64 N = 100;**
2	**int i = 0;**	**__int64 i=0;**
3	**while(i<N)**	**while(i<N)**
4	**{**	**{**
5	**//…**	**//…**
6	i++;	i++;
7	**}**	**}**
1	**mov ecx, 0;**	**mov rcx, 0;**
2	**petlaWHILE:**	**petlaWHILE:**
3	**cmp ecx, N;**	**cmp rcx, N;**
4	**jge koniecWHILE;**	**jge koniecWHILE;**
5	;…	;…
6	**inc ecx;**	**inc rcx;**
7	**jmp petlaWHILE**	**jmp petlaWHILE**
8	**koniecWHILE:**	**koniecWHILE:**
	O P T Y M A L I Z A C J A	
1	**xor ecx, ecx;**	**xor rcx, rcx;**
2	**mov ebx, N;**	**mov rbx, N;**
3	**cmp ecx, ebx;**	**cmp rcx, rbx;**
4	**jge pominWHILE;**	**jge pominWHILE;**
5	**petlaWHILE:**	**petlaWHILE:**
6	;…	;…
7	**inc ecx;**	**inc rcx;**
8	**cmp ecx, ebx;**	**cmp rcx, rbx;**
9	**jl petlaWHILE;**	**jl petlaWHILE;**
10	**pominWHILE:**	**pominWHILE:**
1	**ebx = N;**	**rbx = N;**
2	**ecx = 0;**	**rcx = 0;**
3	**If (ecx < ebx)**	**If (rcx < rbx)**
4	**{**	**{**
5	**do{**	**do{**
6	//…	//…
7	ecx++;	rcx++;
8	**}while (ecx < ebx);**	**}while (rcx < rbx);**
9	**}**	**}**

Zadanie 9:
Proszę napisać funkcję w asemblerze odpowiadającą poniższemu kodowi w C/C++:

```
int while_1(int a, int r, int max)
{
	int wynik;
	int w;
	wynik = a;
	w = a+r;
	while(w<max)
	{
		wynik += w;
		w += r;
	}
	return wynik;
}
```

Zadanie 10:
Proszę napisać funkcję w asemblerze obliczającą silnię za pomocą pętli while().

Implementacja pętli DO{…}WHILE() z inkrementacją

nr	X86	X64
1	**int N = 100;**	**__int64 N = 100;**
2	**int i = 0;**	**__int64 i=0;**
3	**do {**	**do {**
4	**//…**	**//…**
5	i++;	i++;
6	**} while(i<N)**	**} while(i<N)**
1	**mov ecx, 0;**	**mov rcx, 0;**
2	**petlaWHILE:**	**petlaWHILE:**
3	;…	;…
4	inc ecx;	inc rcx;
5	**cmp ecx, N;**	**cmp rcx, N;**
6	**jl petlaWHILE**	**jl petlaWHILE**
	O P T Y M A L I Z A C J A	
1	**xor ecx, ecx;**	**xor rcx, rcx;**
2	**mov ebx, N;**	**mov rbx, N;**
3	**petlaWHILE:**	**petlaWHILE:**
4	;…	;…
5	inc ecx;	inc rcx;
6	**cmp ecx, ebx;**	**cmp rcx, rbx;**
7	**jl petlaWHILE;**	**jl petlaWHILE;**

Implementacja pętli DO{…}WHILE() z dekrementacją

nr	X86		X64
1	**int i = 100;**		**__int64 i = 100;**
2	**i--**		**i--**
3	**do {**		**do {**
4	**//…**		**//…**
5	**i--;**		**i--;**
6	**} while(i>0)**		**} while(i>0)**
1	**mov ecx, N;**		**mov rcx, N;**
2	**dec ecx;**		**dec rcx;**
3	**petlaWHILE:**		**petlaWHILE:**
4	;…		;…
5	**dec ecx;**		**dec rcx;**
6	**cmp ecx, 0;**		**cmp rcx, 0;**
7	**jg petlaWHILE**		**jg petlaWHILE**
	O P T Y M A L I Z A C J A 1		
1	**mov ecx, N;**		**mov rcx, N;**
2	**dec ecx;**		**dec rcx;**
3	**petlaWHILE:**		**petlaWHILE:**
4	;…		;…
5	**loop petlaWHILE;**		**loop petlaWHILE;**
	O P T Y M A L I Z A C J A 2		
1	**mov ecx, N;**		**mov rcx, N;**
2	**dec ecx;**		**dec rcx;**
3	**petlaWHILE:**		**petlaWHILE:**
4	;…		;…
5	**dec ecx;**		**dec rcx;**
6	**jnz petlaWHILE;**		**jnz petlaWHILE;**

Zadania do samodzielnego wykonania:

Proszę napisać podprogramy:

1. jeżeli *a>=b* to do *w* przypisz 10;
2. jeżeli `a != b` to do `w` przypisz `a*b`, w przeciwnym razie `0`;
3. jeżeli `a != 0` to do `w` przypisz `b/a`, w przeciwnym razie sprawdź warunek `b!= 0` i jeżeli jest prawdą, to do `w` przypisz `a/b`;
4. instrukcja warunkowa `switch` ze zmienną `c` z 4 warunkami oraz warunkiem domyślnym. Zawartość ciała danego warunku jest dowolna;
5. dowolna pętla `for`;
6. pętla obliczająca wg wzoru a^2 => a*a (np.5^2 =>5*5 =>w= 5+5+5+5+5)
7. powyższe zadanie na dwa inne sposoby dla różnych pętli (np. For i Do While)
8. silnia z N;

Wykonaj zadania 1 i 2 w wersji dla 32 bitów, a pozostałe dla 64 bitów.