

8. Zásady programování ve strojovém jazyku, základní řídicí konstrukce



Zásady programování ve strojovém jazyku

- podrobně analyzovat zadaný problém
- naznačit řešení v notaci vyššího programovacího jazyka
 - stačí náznaky algoritmu, i symbolicky (vývojový diagram...)
- strukturovat program
 - modularizace – rozdělit program na více jednodušších celků
- používat funkce
 - využít existující knihovny
 - znovu využívat vlastní kód
- používat makra
- podrobně komentovat každý úsek programu
 - komentář vkládat na místa, která nejsou zřejmě pochopitelná
 - není potřeba komentář na každý řádek – raději komentovat blok
 - funkce potřebuje popis (vstup/výstup, změny registrů, činnost)

Základní řídicí konstrukce a jazyk C – připomenutí

- **Větvení:** skok, podmíněné příkazy, přepínač `switch`(, funkce)
- **Cykly:** počítané, s podmínkou na začátku nebo na konci
- **cond** = výraz vyjadřující podmínku – zatím jen jednu jednoduchou
 - musíme převést do assembleru na *test* a podmínku *cc*
 - např.: `cond = EAX > 5` ---> `test = CMP EAX,5` a podmínka `cc = G`
- **statement** = výraz nebo výrazy seskupené do bloku {...}

Neúplný podmíněný příkaz:

```
if (cond) statement;
```

Skok:

```
goto label
```

Úplný podmíněný příkaz:

```
if (cond) statement1; else statement2;
```

Výběr varianty:

```
switch (statement) {
    case val1: ...
}
```

Cykly:

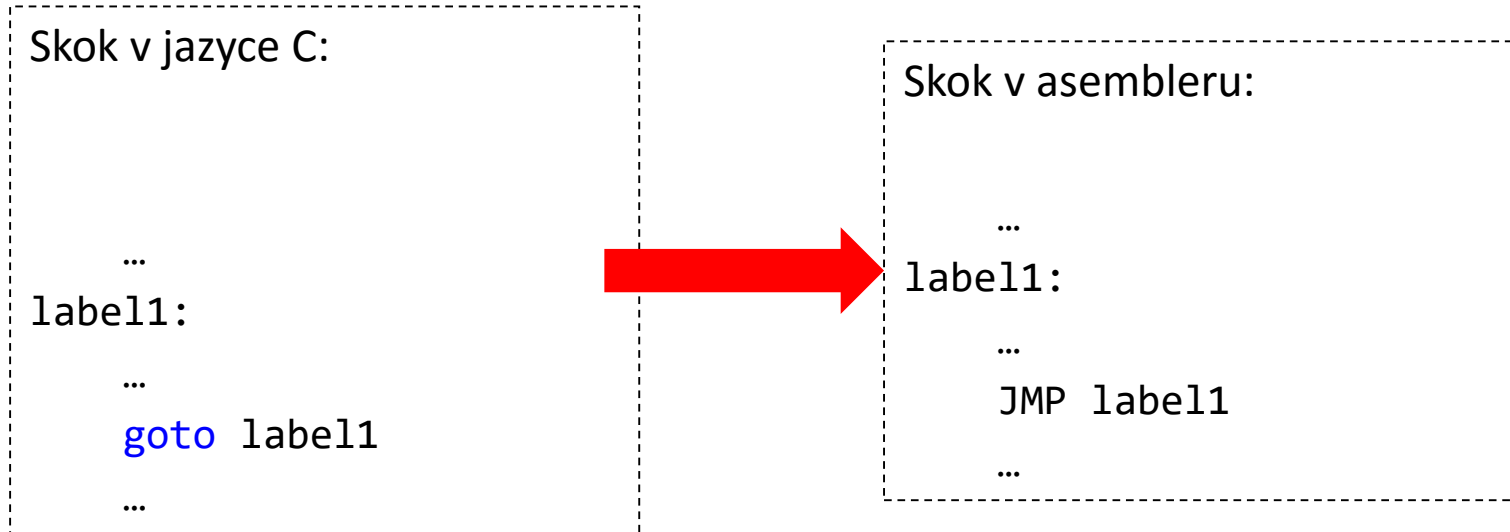
```
for (sInit; cond; sUpdate) statement;
while (cond) statement;
do statement; while (cond);
```

Speciální klíčová slova:

```
break, continue
```

Skok

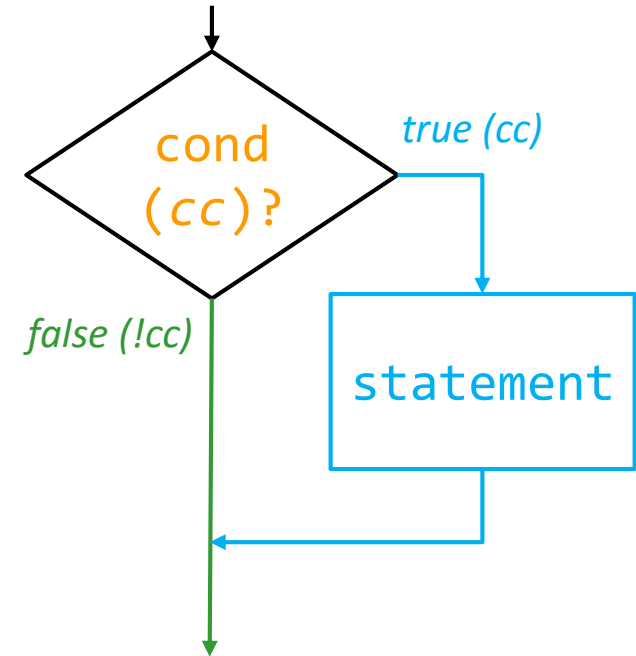
- klíčové slovo `goto`
- v jazyce C nepoužívat, pokud není jiná, procedurální cesta
- v assembleru nelze bez skoků programovat
- speciální klíčová slova: `break` a `continue` (ukážeme později)



Neúplný podmíněný příkaz

`if (cond) statement;`

- jednoduché podmíněné provedení kódu *statement* za předpokladu, že je splněna podmínka *cond*
- podmínka může být značně komplexní
- pro jednoduchost budeme brát v úvahu zatím jen **jednu jednoduchou podmínku** *cc* (viz podmíněné skoky Jcc) – například *Z, C, O*, apod.
- podmíněnému skoku předchází nastavení příznaků instrukcí **TEST** nebo **CMP**
- musíme použít negaci podmínky pro přeskočení kódu *statement* (větev *false*)



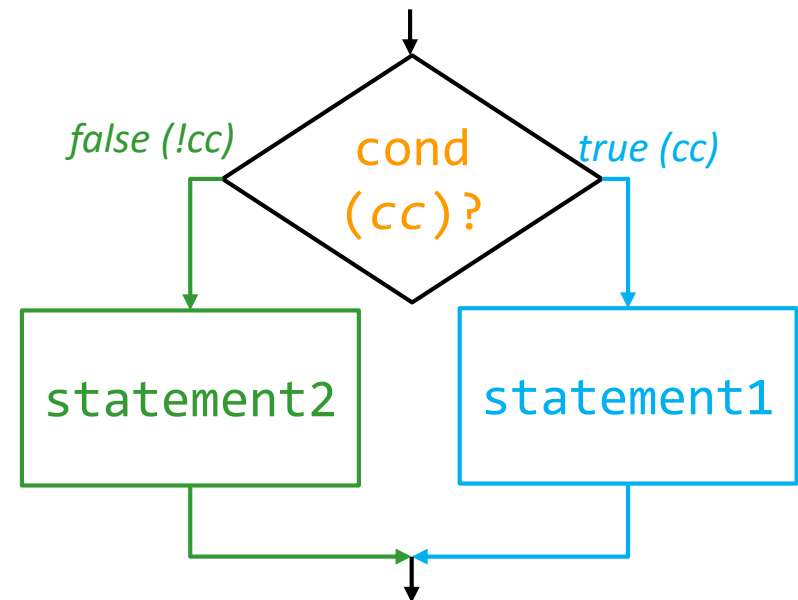
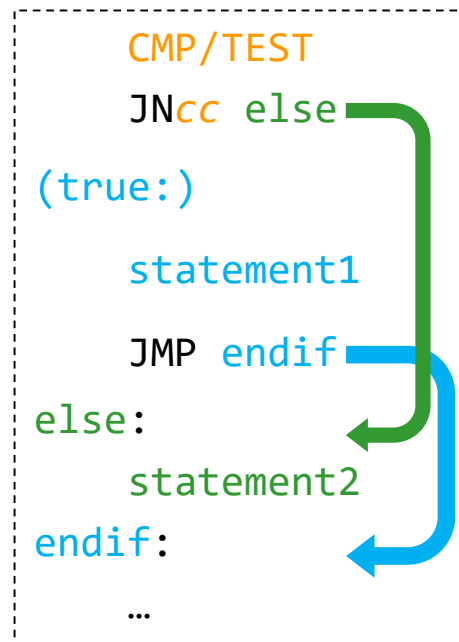
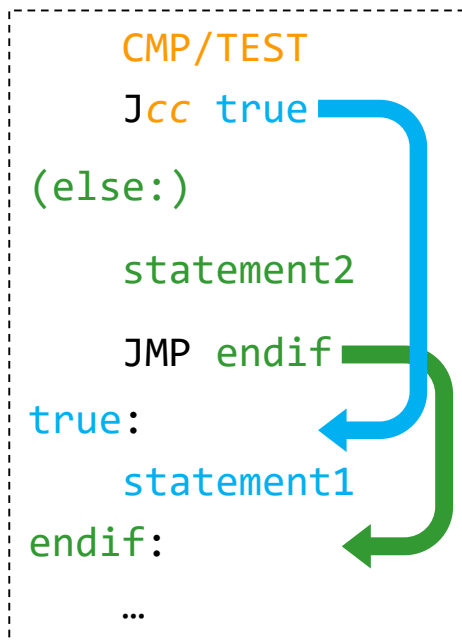
```

...
CMP/TEST
JNcc false
(true:)
statement
false:
...
  
```

Úplný podmíněný příkaz

`if (cond) statement1; else statement2;`

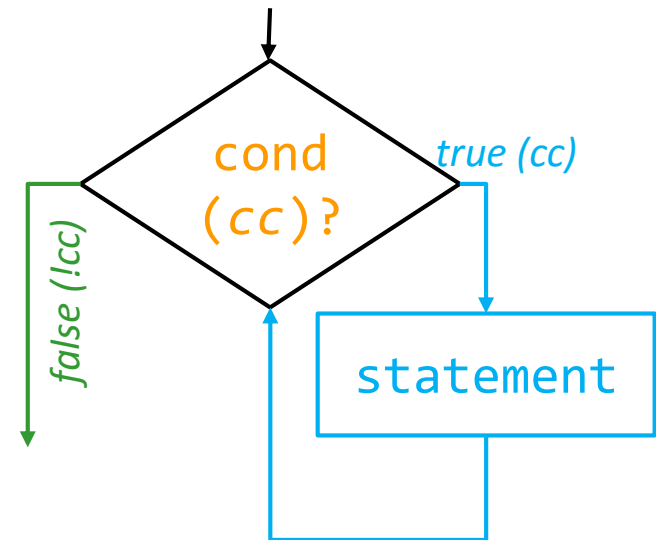
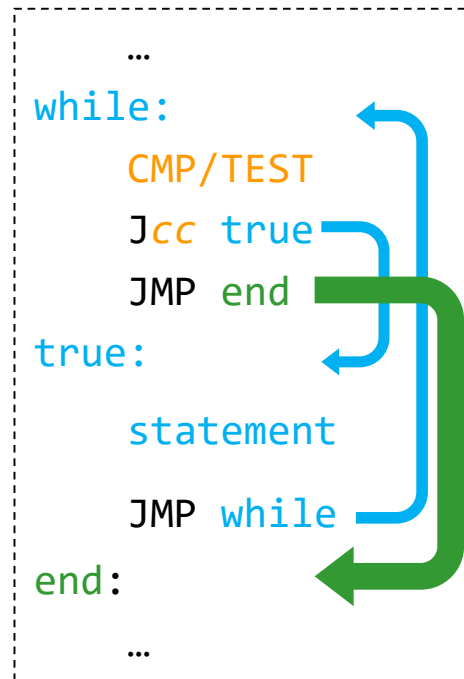
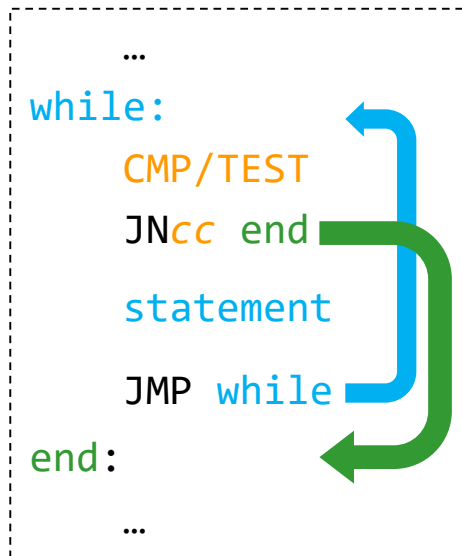
- úplné podmíněné provedení kódu přidává větev *else* a *statement2*
- jsou dvě varianty, jak převést do assembleru:
 - `Jcc true` vs. `JNcc else`
- u obou variant je potřeba po provedení větve přeskočit kód druhé větve (`JMP endif`)



Cyklus s podmínkou na začátku (*while*)

while (*cond*) *statement*;

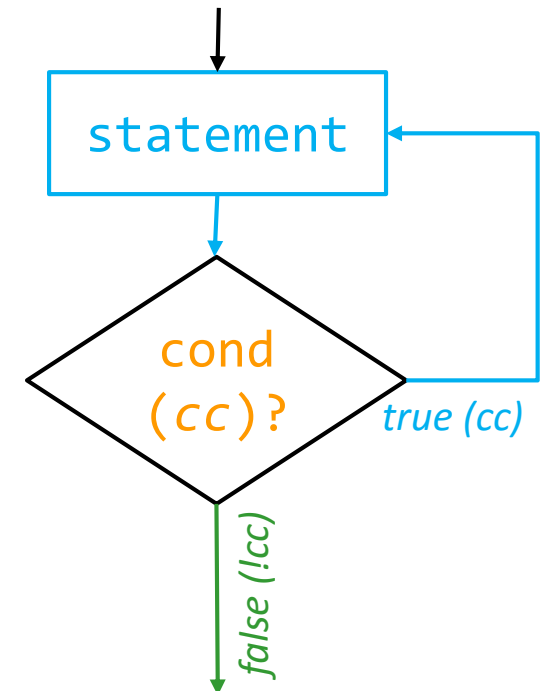
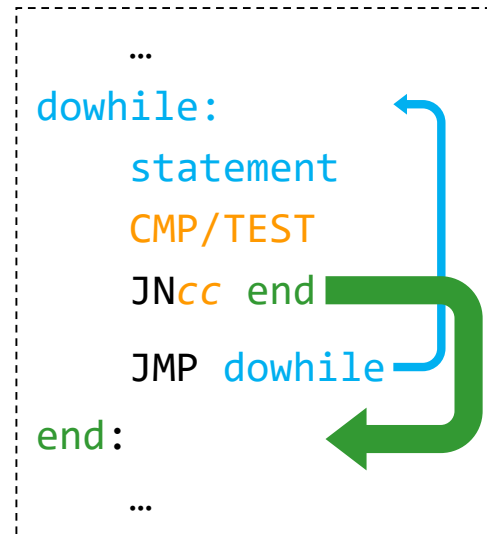
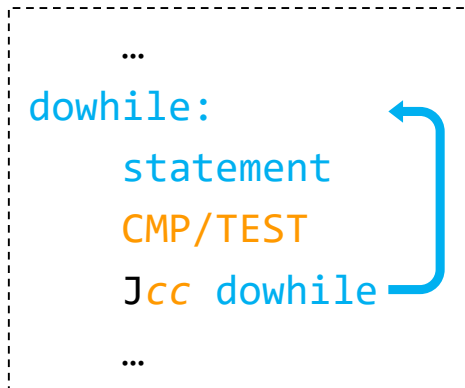
- provedení kódu *statement* se opakuje, dokud je splněna podmínka *cond*



Cyklus s podmínkou na konci (`do...while`)

`do statement while (cond);`

- provedení kódu *statement* se opakuje, dokud je splněna podmínka *cond* => stejné jako `while`, ale je zaručeno jedno provedení cyklu = jedno spuštění příkazu *statement* (`while` může skončit ještě před jeho provedením)



Cyklus se známým (?) počtem iterací (**for**)

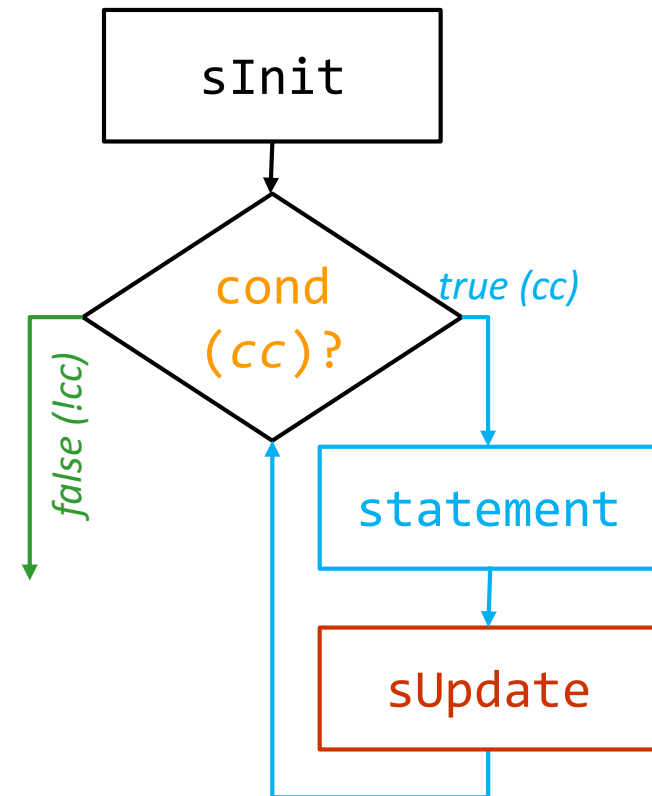
for (sInit; cond; sUpdate) statement;

- jazyk C umožňuje obecně posunout tento typ cyklus na zcela jinou úroveň (sInit, cond i sUpdate lze vynechat, podmínka může být jakákoliv, výraz aktualizace může být funkce...), spíše odpovídá **while**:

```
...
sInit;
while (cond) {
    statement;
    sUpdate;
}
...
```

cyklus **for** jazyka C
obecně ~ **while**

```
...
sInit
for:
    CMP/TEST
    JNcc end
    statement
    sUpdate
    JMP for
end:
...
```



Cyklus se známým (!) počtem iterací (**for**)

for (sInit; cond; sUpdate) statement;

- pro naše účely je cyklus **for** = cyklus se známým počtem iterací, např.:

```
for(ECX=počet; ECX>0; ECX--) statement;
```

...

```
MOV ECX,počet
```

for:

```
statement
```

```
LOOP for
```



<- sUpdate je součástí instrukce LOOP (ECX--)

<- podmínka je součástí instrukce LOOP (ECX>0)

...

```
MOV ECX,počet <- inicializace cyklu (sInit) může být cokoliv
```

while:

```
CMP ECX,0
```

```
JNG end
```

```
statement
```

```
DEC ECX
```

```
JMP while
```

end:



<- podmínka (cond) může být libovolná

<- zde: ECX > 0, je-li splněna, neskočí se


<- aktualizace (sUpdate) může být cokoliv

Ukončení cyklu (**break**)

- skok na místo, které interně definuje překladač
- umožňuje ukončit cyklus mimo počáteční nebo koncové podmínky
- nutnost použití v cyklu = občas chyba návrhu, většinou lze jinak

```
while (cond1) {
    ...
    if (cond2) break;
    ...
}
```

```
while:
    CMP/TEST
    JNcc1 end
    ...
    CMP/TEST
    JNcc2 end
    ...
    JMP while
end:
```



```
EAX = 50; EDX = 100;
while (EAX > 0) {
    EAX = EAX - 1;
    if (EDX < 0) break;
    EDX = EDX - EAX;
}
```


```
MOV EAX, 50
MOV EDX, 100
while:
    CMP EAX,0
    JLE end
    SUB EAX,1
    CMP EDX,0
    JL end
    SUB EDX,EAX
    JMP while
end:
```

Přerušení iterace cyklu (`continue`)

- umožňuje ukončit iteraci cyklu = skok na test podmínky cyklu (`while` a `do ... while`) případně skok na kód `sUpdate` u `for`
- nutnost použití v cyklu = občas chyba návrhu, většinou lze jinak

```
while (cond1) {
    ...
    if (cond2) continue;
    ...
}
```

```
while:
    CMP/TEST
    JNcc1 end
    ...
    CMP/TEST
    Jcc2 while
    ...
    JMP while
end:
```



```
EAX = 50; EDX = 100;
while (EAX > 0) {
    EAX = EAX - 1;
    if (EDX < 0) continue;
    EDX = EDX - EAX;
}
```

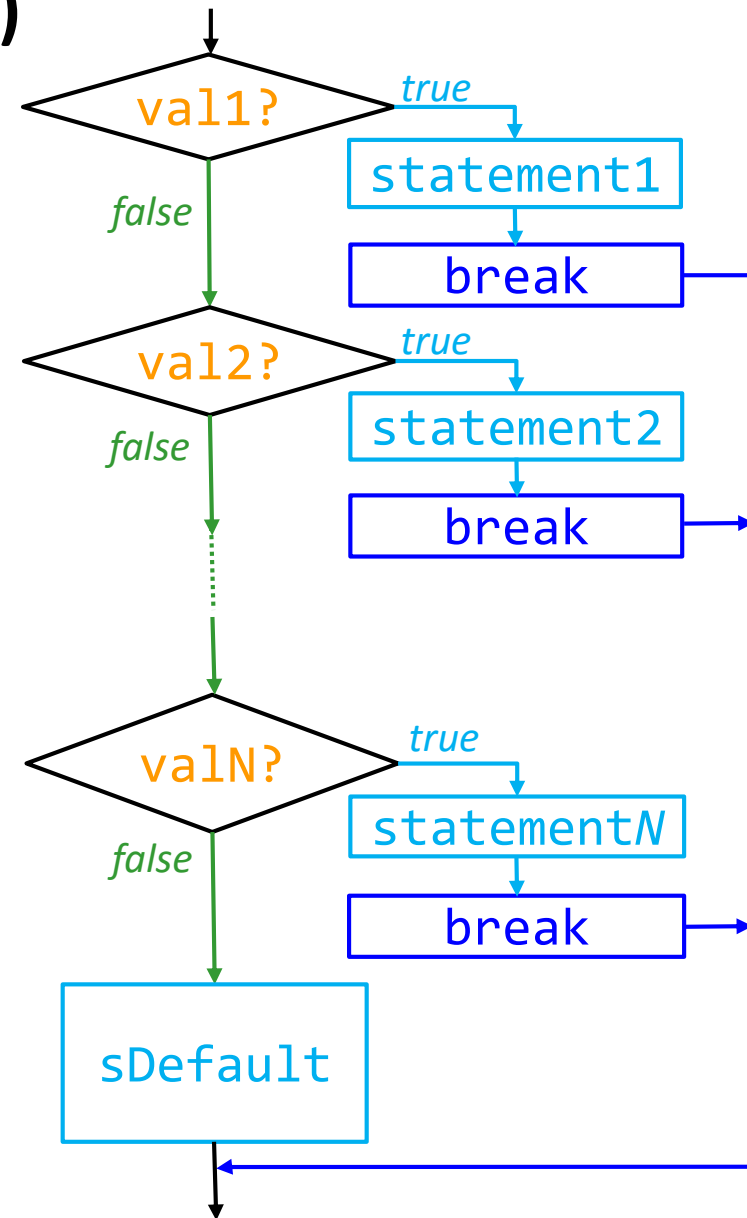
```
MOV EAX, 50
MOV EDX, 100
while:
    CMP EAX,0
    JNG end
    SUB EAX,1
    CMP EDX,0
    JL while
    SUB EDX,EAX
    JMP while
end:
```

Výběr možnosti – větvení (*switch*)

```
switch (x) {
  case val1: statement1; break;
  case val2: statement2; break;
  ...
  case valN: statementN; break;
  default: sDefault;
}
```

- výběr varianty lze řešit několika způsoby:

- série `if ... else if ... else if ...`
- použití tabulky skoků
 - pole hodnot a ukazatelů (dvojice)
 - pole ukazatelů, index = hodnota



Výběr možnosti (**switch**) – poslounost skoků

```
switch (x) {  
  case val1: statement1; break;  
  case val2: statement2; break;  
  ...  
  case valN: statementN; break;  
  default: sDefault;  
}
```

- nejméně efektivní, ale univerzální
- lze optimalizovat přesunem hodnot s nejčastějším výskytem na začátek
 - dříve nalezneme hodnotu s častým výskytem a neprocházíme zbytečně ostatní)

```
CMP x,val1  
JNE case_val2  
statement1  
JMP break  
case_val2:  
CMP x,val2  
JNE case_val3  
statement2  
JMP break  
...  
case_valN:  
CMP x,valN  
JNE case_default  
statementN  
JMP break  
case_default:  
sDefault  
break:
```

Výběr možnosti (**switch**) – pole hodnot a ukazatelů

```
switch (x) {
  case val1:
    statement1;
    break;
  ...
  case valN:
    statementN;
    break;
  default:
    sDefault;
}
```

```
    CMP x,val1
    JNE case_val2
    statement1
    JMP break
case_val2:
    CMP x,val2
    JNE case_val3
    statement2
    JMP break
  ...
case_valN:
    CMP x,valN
    JNE case_default
    statementN
    JMP break
case_default:
    sDefault
break:
```

```
segment .data
    cases DD val1, case_val1
           .....
           DD valN, case_valN
segment .text
    MOV ECX,N
next_case:
    CMP dword x,[cases+8*ECX-8]
    JE dword [cases+8*ECX-4]
    LOOP next_case
case_default:
    sDefault
    JMP break
case_val1:
    statement1
    JMP break
  ...
case_valN:
    statementN
break:
```

Výběr možnosti (*switch*) – pole ukazatelů

- nejefektivnější
- potřeba pokrýt všechny možné hodnoty $x \Rightarrow$ nehodí se pro „velké“ datové typy
- hodnoty bez „vlastního“ ukazatele nastavit na ukazatel *case_default*

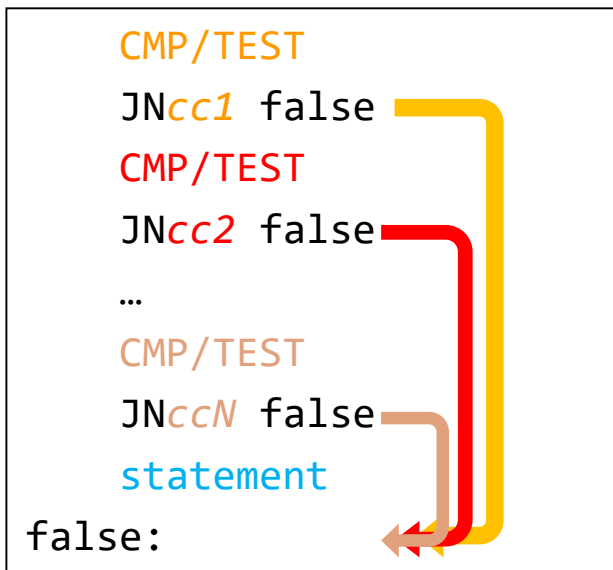
```
switch (x) {
  case val1: statement1; break;
  case val2: statement2; break;
  ...
  case valN: statementN; break;
  default: sDefault;
}
```

```
segment .data
  cases DD case_val1
        DD case_val2
        DD case_default
        .....
        DD case_valN
segment .text
  MOV EAX,x
  JMP dword [cases+4*EAX]
case_default:
  sDefault
  JMP break
case_val1:
  statement1
  JMP break
...
case_valN:
  statementN
break:
```


Složené podmínky – logický součin (AND, &&)

`if (cond1 && cond2 && ... && condN) statement;`

- logický součin – AND – && – na logické úrovni lze naprogramovat dvěma způsoby:
 - kombinace podmíněných skoků s negacemi podmínek – nesplnění podmínky => neprovedení kódu `statement` => skok pryč
 - kombinace výpočtu výsledku logického výrazu a následného podmíněného skoku (hodí se spíše u komplikovaných podmínek)



```

EAX = 50; EDX = -100;
if (EAX > 0 && EDX <= 0) {
    EDX = EDX - EAX;
}

```

```

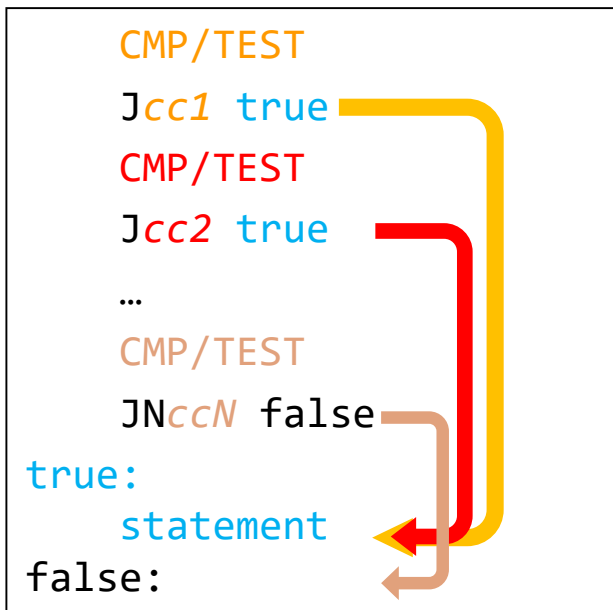
MOV EAX, 50
MOV EDX, 100
CMP EAX, 0
JNG false
CMP EDX, 0
JNLE false
SUB EDX, EAX
false:

```

Složené podmínky – logický součet (OR, ||)

```
if (cond1 || cond2 || ... || condN) statement;
```

- logický součet – OR – || – na logické úrovni lze naprogramovat podobně jako AND kombinací podmíněných skoků s podmínkami – splnění jediné podmínky => provedení kódu `statement` => skok na provedení kódu, pokud se žádný skok neprovede => *false* a skok pryč



```

EAX = 50; EDX = -100;
if (EAX > 0 || EDX <= 0) {
    EDX = EDX - EAX;
}

```

```

MOV EAX, 50
MOV EDX, -100
CMP EAX, 0
JG true
CMP EDX, 0
JLE true
JMP false
true:
    SUB EDX, EAX
false:

```

Složené podmínky – kombinace (&&, ||)

```
if (cond1 && cond2 || cond3 && cond4) statement;
```

- komplikovanější, je potřeba dávat pozor na priority operátorů (přednost má součin – AND)

```

CMP/TEST
JNcc1 c12_false
CMP/TEST
JNcc2 c12_false
JMP true
c12_false:
  CMP/TEST
  JNcc3 false
  CMP/TEST
  JNcc4 false
true:
  statement
false:
```

```

if (  EAX > 0 && EDX >= 0
    || EAX >= 0 && EDX <= 0) {
    EDX = EDX - EAX;
}
```

```

CMP EAX,0
JNG c12_false
CMP EDX,0
JNGE c12_false
JMP true
c12_false:
  CMP EAX,0
  JNGE false
  CMP EDX,0
  JNLE false
true:
  SUB EDX,EAX
false:
```

Složené podmínky – kombinace (||, &&)

```
if (cond1 || cond2 && cond3 || cond4) statement;
```

- záměna operátorů && a || vede na zcela jiný kód
- změna pořadí vyhodnocování podmínek může zkrátit kód

```

CMP/TEST
Jcc1 true
CMP/TEST
JNcc2 c23_false
CMP/TEST
JNcc3 c23_false
JMP true
c23_false:
  CMP/TEST
  JNcc4 false
true:
  statement
false:

```

==

```

CMP/TEST
Jcc1 true
CMP/TEST
Jcc4 true
CMP/TEST
JNcc2 false
CMP/TEST
JNcc3 false
true:
  statement
false:

```

Složené podmínky – výpočet pravdivosti podmínky

- využití například instrukce **SETcc dest**

```
if (cond1 ||1
    cond2 &&2 cond3 ||3
    cond4)
    statement;
```

```
CMP/TEST
SETcc1 AL
CMP/TEST
SETcc4 AH
1OR AL,AH
CMP/TEST
SETcc2 BL
CMP/TEST
SETcc3 BH
2AND BL,BH
3OR AL,BL
JZ false
    statement
false:
```

```
if (cond1 ||1
    (cond2 &&2 cond3 ||3 cond4) &&4 (!cond5))
    statement;
```

```
CMP/TEST
SETcc1 AL
CMP/TEST
SETcc2 BL
CMP/TEST
SETcc3 BH
2AND BL,BH
CMP/TEST
SETcc4 BH
3OR BL,BH
CMP/TEST
SETNcc4 BH
4AND BL,BH
1OR AL,BL
JZ false
    statement
false:
```

SETcc dest

Pokud je splněna podmínka cc, pak uloží do cíle 1, jinak do cíle uloží 0.

- výhoda = jen jeden skok
- nevýhoda = nelze předčasně ukončit výpočet pravdivosti podmínky při jejím vyhodnocení
 - např. pokud cond1 == TRUE, pak lze ukončit výpočet, protože podmínka = TRUE

Příklad (1)

- Kreslení úsečky Bresenhamovým algoritmem z bodu $[x_1, y_1]$ do bodu $[x_2, y_2]$ s otestováním umístění bodů v definovaném okně šířky WIDTH a výšky HEIGHT, výměnou chybného pořadí bodů a platné pouze pro první oktant (sklon $0 - 45^\circ$)

```
int x1, y1, x2, y2, tmp, dx, dy2, D;
...
if (x1 >= 0 && x1 < WIDTH && y1 >= 0 && y1 < HEIGHT &&
    x2 >= 0 && x2 < WIDTH && y2 >= 0 && y2 < HEIGHT)
{
    if (x1 > x2) {
        tmp = x1; x1 = x2; x1 = tmp;
        tmp = y1; y1 = y2; y1 = tmp;
    }
    dx = x2 - x1; dy2 = 2*(y2 - y1); D = -dx;
    do {
        PutPixel(x1, y1);
        x1++; D += dy2;
        if (D > 0) { y1++; D -= 2*dx; }
    } while (x1 <= x2)
}
```

Příklad (2)

```

#include 'rw32-2015.inc'
segment .data
; int x1, y1, x2, y2, tmp, dx, dy2, D
    x1 DD 5
    y1 DD 10
    x2 DD 25
    y2 DD 33
    sPutPixel DB "Bod: (",0
; tmp nepotřebujeme, dx = EAX, dy2 = EBX, D = ECX
segment .text
main:
; if (x1 >= 0 && x1 < WIDTH && y1 >= 0 && y1 < HEIGHT &&
;     x2 >= 0 && x2 < WIDTH && y2 >= 0 && y2 < HEIGHT)
    CMP [x1], dword 0
    JL false
    CMP [x1], dword WIDTH ; %define WIDTH 200 / makra probereme později
    JGE false
    CMP [y1], dword 0
    JL false
    CMP [y1], dword HEIGHT ; %define HEIGHT 300
    JGE false
    CMP [x2], dword 0
    JL false
; ... totéž pro bod [x2, y2] - pouze && => jednoduché

```

Příklad (3)

```
    MOV ESI, [x1]
    MOV EDI, [y1]
; if (x1 > x2) {
    CMP ESI, [x2]
    JNG no_xchg
; tmp = x1; x1 = x2; x1 = tmp;
    XCHG ESI, [x2]
; tmp = y1; y1 = y2; y1 = tmp;
    XCHG EDI, [y2]
; }
no_xchg:
; dx = x2 - x1;
    MOV EAX, [x2]
    SUB EAX, ESI
; dy2 = 2*(y2 - y1);
    MOV EBX, [y2]
    SUB EBX, EDI
    SHL EBX, 1
; D = -dx;
    MOV ECX, [y2]
    NEG ECX
```

x1 = ESI
y1 = EDI

dx = EAX
dy2 = EBX
D = ECX

Příklad (4)

```

; do {
do_while:
; PutPixel(x, y);
    CALL PutPixel
; x1++;
    INC ESI
; D += dy2;
    ADD ECX,EBX
; if (D > 0) {
    CMP ECX,0
    JNG endif
; y1++;
    INC EDI
; D -= 2*dx;
    SUB ECX,EAX
    SUB ECX,EAX
; }
endif:
; } while (x1 <= x2)
    CMP ESI,[x2]
    JLE do_while
; }
false:
    RET

```

dx	=	EAX
dy2	=	EBX
D	=	ECX
x1	=	ESI
y1	=	EDI

```

PutPixel:
    PUSH ESI
    PUSH EAX
    PUSH ESI
    MOV ESI,sPutPixel
    CALL WriteString
    POP ESI
    MOV EAX,ESI
    CALL WriteInt25
    MOV AL,', '
    CALL WriteChar
    MOV EAX,EDI
    CALL WriteInt25
    MOV AL,')'
    CALL WriteChar
    CALL WriteNewLine
    POP EAX
    POP ESI
    RET

```