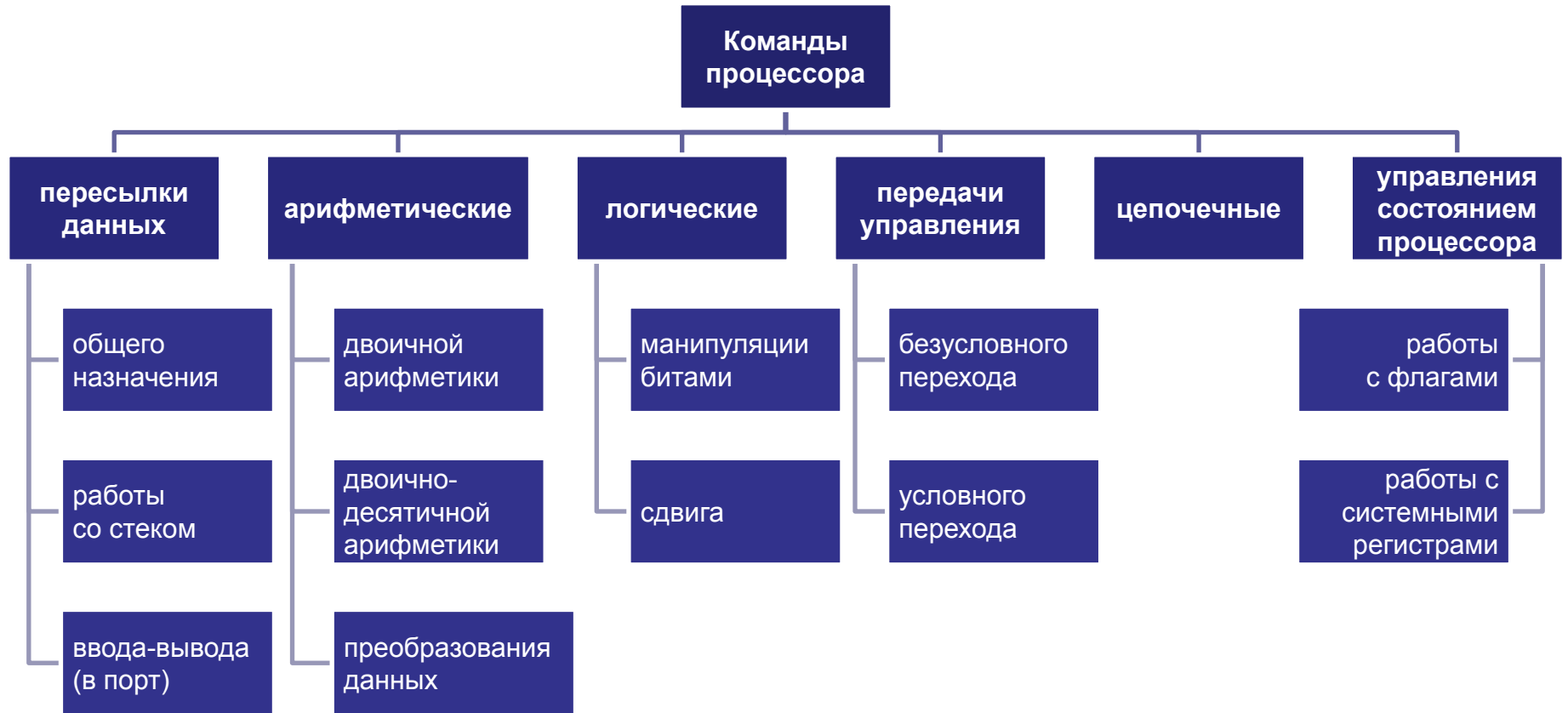


Система команд микропроцессора Intel 80x86

Система команд микропроцессора



Команды пересылки данных

Команда MOV – пересылка данных

Формат команды

mov <Приемник>, <Источник>

Действие команды

В операнд **Приемник** заносится значение операнда **Источник**

Запись на языке высокого уровня

Приемник = Источник;

Команда MOV – пересылка данных

Пример 1. Обмен значениями регистров (EAX и EBX)

```
mov ECX, EAX ; ECX = EAX
```

```
mov EAX, EBX ; EAX = EBX
```

```
mov EBX, ECX ; EBX = ECX
```

Команда MOV – пересылка данных

Пример 2. Реализация команды A=B

```
mov EAX, A
```

```
mov B, EAX
```

Арифметические команды

Команда ADD – сложение

Формат команды

add <Приемник>, <Источник>

Действие команды

В операнд **Приемник** заносится сумма операнда **Приемник** и операнда **Источник**

Запись на языке высокого уровня

Приемник += Источник;

Команда ADD – сложение

Пример 1. Сложение двух регистров ($ECX = EAX + EBX$)

```
mov ECX, EAX ; ECX = EAX
```

```
add ECX, EBX ; ECX += EBX
```

Команда ADD – сложение

Пример 2. Реализация команды $C=A+B$

```
mov EAX, A
```

```
add EAX, B
```

```
mov C, EAX
```

Команда ADC – сложение с учетом переноса

Формат команды

adc <Приемник>, <Источник>

Действие команды

В операнд **Приемник** заносится сумма операнда **Приемник**, операнда **Источник** и бита **CF** (переноса от предыдущего арифметического действия)

Запись на языке высокого уровня

Приемник += Источник + CF;

Команда ADC – сложение с учетом переноса

Пример 1. Сложение двух 64-разрядных чисел
(EBX; EAX) += (EDX; ECX)

```
add EAX, ECX
```

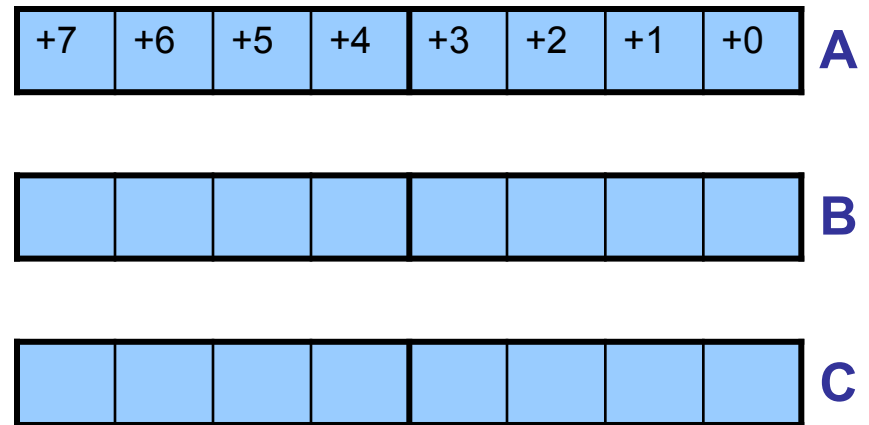
```
adc EBX, EDX
```



Команда ADC – сложение с учетом переноса

Пример 2. Сложение двух 64-разрядных чисел
($C = A + B$)

```
mov EAX, A  
add EAX, B  
mov C, EAX  
mov EAX, A + 4  
adc EAX, B + 4  
mov C + 4, EAX
```



Команда INC – увеличение на единицу

Формат команды

inc <Операнд>

Действие команды

Операнд увеличивается на 1

Запись на языке высокого уровня

Операнд++;

Команда SUB – вычитание

Формат команды

sub <Приемник>, <Источник>

Действие команды

В операнд **Приемник** заносится разность операнда **Приемник** и операнда **Источник**

Запись на языке высокого уровня

Приемник -= Источник;

Команда SUB – вычитание

Пример 1. Вычитание двух регистров ($ECX = EAX - EBX$)

```
mov ECX, EAX ; ECX = EAX
```

```
sub ECX, EBX ; ECX -= EBX
```


Команда SUB – вычитание

Пример 2. Реализация команды $C = A - B$

```
mov EAX, A
```

```
sub EAX, B
```

```
mov C, EAX
```

Команда SBB – вычитание с учетом переноса

Формат команды

sbb <Приемник>, <Источник>

Действие команды

В операнд **Приемник** заносится разность операнда **Приемник** и суммы операнда **Источник** и бита **CF** (переноса от предыдущего арифметического действия)

Запись на языке высокого уровня

Приемник -= Источник + CF;

Команда SBB – вычитание с учетом переноса

Пример 1. Вычитание двух 64-разрядных чисел
(EBX; EAX) -= (EDX; ECX)

```
sub EAX, ECX
```

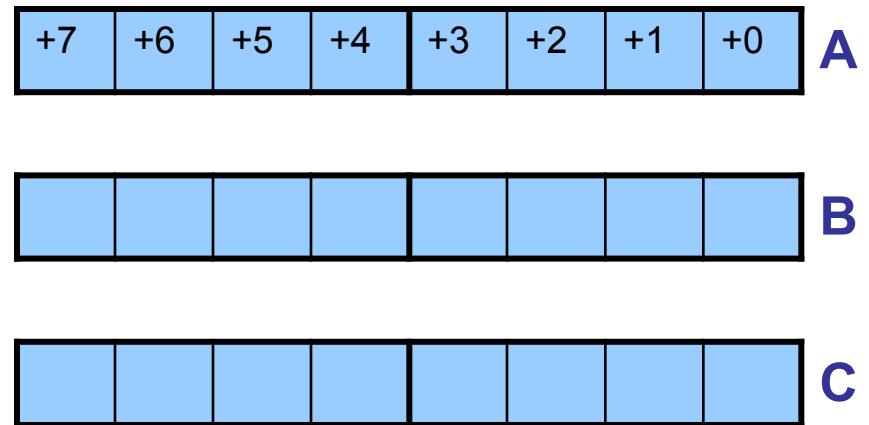
```
sbb EBX, EDX
```



Команда SBB – вычитание с учетом переноса

Пример 2. Вычитание двух 64-разрядных чисел
($C = A - B$)

```
mov EAX, A
sub EAX, B
mov C, EAX
mov EAX, A + 4
sbb EAX, B + 4
mov C + 4, EAX
```



Команда DEC – уменьшение на единицу

Формат команды

dec <Операнд>

Действие команды

Операнд уменьшается на 1

Запись на языке высокого уровня

Операнд--;

Команда MUL – умножение беззнаковых чисел

Формат команды

mul <Источник>

Действие команды

В зависимости от размера операнда **Источник**:

1 байт: $AX = AL * \text{Источник};$

2 байта: $(DX; AX) = AX * \text{Источник};$

4 байта: $(EDX; EAX) = EAX * \text{Источник};$

Команда MUL – умножение беззнаковых чисел

Особенности команды

Размер *произведения* всегда **в два раза** больше размера *множителей*

Пример. Реализация команды $C = A * B$

```
mov EAX, A
```

```
mul B
```

```
mov C, EAX ; возможна потеря  
; разрядов !!!
```

Команда DIV – деление беззнаковых чисел

Формат команды

div <Источник>

Действие команды

В зависимости от размера операнда **Источник**:

1 байт: $AL = AX / \text{Источник};$

$AH = AX \% \text{Источник};$

2 байта: $AX = (DX; AX) / \text{Источник};$

$DX = (DX; AX) \% \text{Источник};$

4 байта: $EAX = (EDX; EAX) / \text{Источник};$

$EDX = (EDX; EAX) \% \text{Источник};$

Команда DIV – деление беззнаковых чисел

Особенности команды

Размер неполного *частного* и *остатка* всегда в два раза меньше размера *делимого*.

Пример. Реализация команды $C = A / B$

```
mov EAX, A
```

```
mov EDX, 0
```

```
div B
```

```
mov C, EAX
```

Команда IMUL – умножение знаковых чисел

Формат команды

imul <Источник>

imul <Приемник>, <Источник>

imul <Приемник>, <Источник1>,
<Источник2>

Команда IMUL – умножение знаковых чисел

Действие команды, случай **первый**

соответствует команде MUL,
но учитывается знаковый бит

Пример.

10000000	10000000
<u>mul 00000010</u>	<u>imul 00000010</u>
0000000100000000	1111111100000000
(128 * 2 = 256)	(-128 * 2 = -256)

Команда IMUL – умножение знаковых чисел

Действие команды, случаи **второй** и **третий**

- **операнд-приемник** должен быть регистром;
- **операнд-источник²** должен быть непосредственным значением из диапазона $[-128; +127]$;
- результат умножения усекается до размера **операнда-приемника** (возможна потеря разрядов)

Команда `IMUL` – умножение знаковых чисел

Пример. Реализация команды $C = A * B$

```
mov EAX, A
```

```
imul EAX, B ; возможна потеря !!!
```

```
mov C, EAX
```

Команда IDIV – деление знаковых чисел

Формат команды

idiv <Источник>

Действие команды

Соответствует команде DIV,
но учитывается знаковый бит

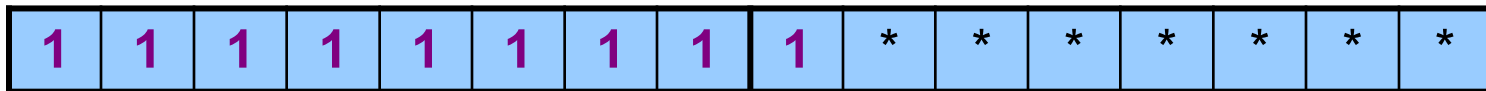
Команда CBW – преобразование байта в слово

Формат команды

cbw

Действие команды

Заполняет регистр **AX** значением старшего бита регистра **AL**, т.е. расширяет **AL** → **AX**



Команда CBW – преобразование байта в слово

Пример 1. Вычисление $C = A + B$
(Слово = Байт + Слово)

```
mov AL, A  
cbw  
add AX, B  
mov C, AX
```


Команда CBW – преобразование байта в слово

Пример 2. Вычисление $C = A / B$ (Байт = Байт / Байт)

```
mov AL, A
```

```
cbw
```

```
idiv B
```

```
mov C, AL
```

Команда CWD – преобразование слова в двойное слово

Формат команды

cwd

Действие команды

Заполняет регистр **DX** значением старшего бита регистра **AX**, т.е. расширяет **AX** → (**DX**; **AX**)

Команда **CWDE** – преобразование слова в двойное слово

Формат команды

cwde

Действие команды

Заполняет старшую часть регистра **EAX** значением старшего бита регистра **AX**, т.е. расширяет **AX** → **EAX**

Команда CDQ – преобразование двойного слова в учетверенное слово

Формат команды

cdq

Действие команды

Заполняет регистр **EDX** значением старшего бита регистра **EAX**, т.е. расширяет **EAX** → (**EDX**; **EAX**)

Команда CDQ – преобразование двойного слова в учетверенное слово

Пример. Вычисление $C = A / B$ (знаковые операнды)

```
mov EAX, A
```

```
cdq
```

```
idiv B
```

```
mov C, EAX
```

Арифметические команды

Для преобразования типа *беззнаковых* операндов достаточно заполнить соответствующий регистр (часть регистра) нулевыми битами, например, с помощью команды MOV

Пример. Вычисление $C = A / B$ (беззнаковые операнды)

```
mov EAX, A
```

```
mov EDX, 0 ; xor EDX
```

```
div B
```

```
mov C, EAX
```

Команды перехода

Команды перехода предназначены для изменения линейной последовательности выполнения программы.

Принцип работы всех команд перехода заключается в модифицировании значения регистра **EIP** (указателя инструкций).

Все команды перехода имеют одинаковый **формат**:

j* <адрес команды>**

Адрес команды может указываться непосредственно, но чаще всего он задается с помощью *символьной метки*:

<метка>: <команда>

.....

j* <метка>**

Все команды перехода делятся на команды *безусловного* и *условного* перехода.

При выполнении **команды безусловного перехода** переход осуществляется всегда.

Команда JMP – безусловный переход

Формат команды

jmp <адрес команды>

Действие команды

заносят в регистр **EIP** указанное значение
(**EIP = <адрес команды>**)

При выполнении **команды условного перехода** переход осуществляется, если выполняется некоторое *условие* перехода.

Условием перехода может являться значение некоторого *флага* или комбинация значений нескольких *флагов*.

Команды перехода

Команда условного перехода	Условие перехода
jo (Jump if O verflow)	OF == 1
jno (Jump if N o O verflow)	OF == 0
js (Jump if S ign)	SF == 1
jns (Jump if N o S ign)	SF == 0
jz (Jump if Z ero)	ZF == 1
jnz (Jump if N o Z ero)	ZF == 0
jp (Jump if P arity)	PF == 1
jnp (Jump if N o P arity)	PF == 0
jc (Jump if C arry)	CF == 1
jnc (Jump if N o C arry)	CF == 0

Обычно команды условного перехода размещают в программе после *арифметических команд*.

(Напомним, что биты регистра флагов *EFlags* изменяются в зависимости от результата арифметической операции).

Таким образом, команды условного перехода позволяют проанализировать *результат арифметической операции*: отрицательный или положительный, равен нулю или не равен нулю и т.п.

Часто в программе возникает необходимость *сравнить* значения двух чисел. Для этих целей перед командами условного перехода используется команда **СМР**.

Формат команды

сmp <операнд 1>, <операнд 2>

Действие команды

От <операнда 1> отнимает <операнд 2>.

Результат вычитания нигде **не сохраняется**, но в соответствии с его значением изменяются **флаги**.

Сравнение *беззнаковых* чисел

cmp <операнд 1>, <операнд 2>
j** <адрес>

Команда условного перехода	Условие перехода
je (Jump if E qual)	равно
jne (Jump if N o E qual)	не равно
ja (Jump if A bove)	больше
jae (Jump if A bove or E qual)	больше или равно
jb (Jump if B elow)	меньше
jbe (Jump if B elow or E qual)	меньше или равно

Команды перехода

Для удобства восприятия программы можно использовать команды-синонимы:

ja ↔ **jnbe**

jae ↔ **jnb**

jb ↔ **jnae**

jbe ↔ **jna**

Сравнение знаковых чисел

cmp <операнд 1>, <операнд 2>
j** <адрес>

Команда условного перехода	Условие перехода
je (Jump if E qual)	равно
jne (Jump if N o E qual)	не равно
jg (Jump if G reater)	больше
jge (Jump if G reater or E qual)	больше или равно
jl (Jump if L ess)	меньше
jle (Jump if L ess or E qual)	меньше или равно

Команды перехода

Для удобства восприятия программы можно использовать команды-синонимы:

жg ↔ жnle

жge ↔ жnl

жl ↔ жnge

жle ↔ жng

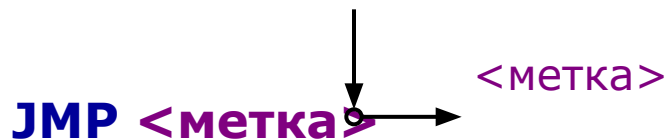
Реализация алгоритмических структур

Реализация алгоритмических структур

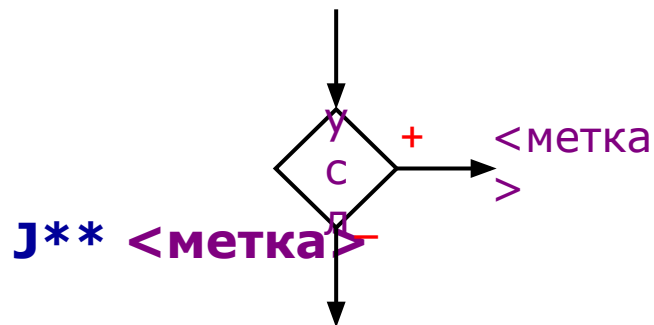
Как было сказано ранее, использование команд перехода позволяет реализовать последовательность выполнения команд, отличную от линейной.

Блок-схемы отдельных команд перехода можно изобразить так:

Команда безусловного перехода

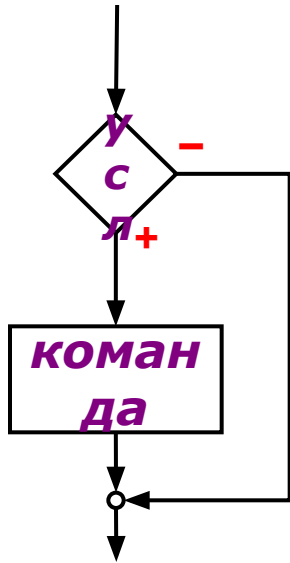


Команды условного перехода



1. Неполное ветвление

if (усл) { команда; }

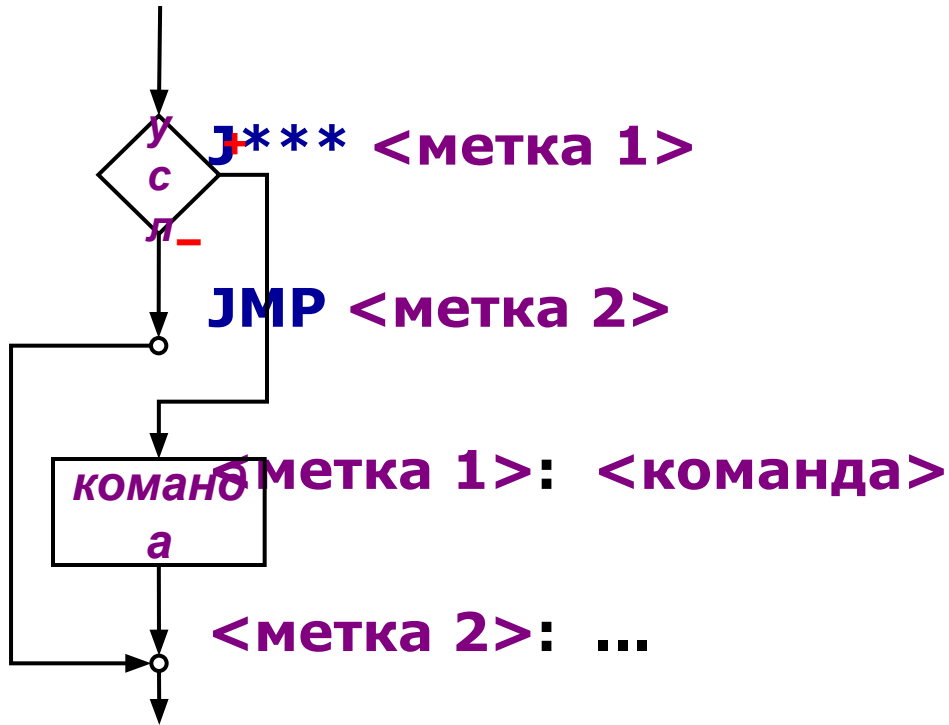


Непосредственная реализация затруднительна, поскольку блок-схема отдельных элементов конструкции не соответствует блок-схемам имеющихся операторов перехода.

Необходимо заменить блок-схему на другую.

Реализация алгоритмических структур

Можно преобразовать блок-схему так, чтобы она содержала только подходящие элементы



Пример. Фрагмент алгоритма поиска наибольшего элемента массива

```
if(max < A[i])
```

```
    JL Metka1
```

```
    JMP Metka2
```

```
{           Metka1:
```

```
  max = A[i];
```

```
}
```

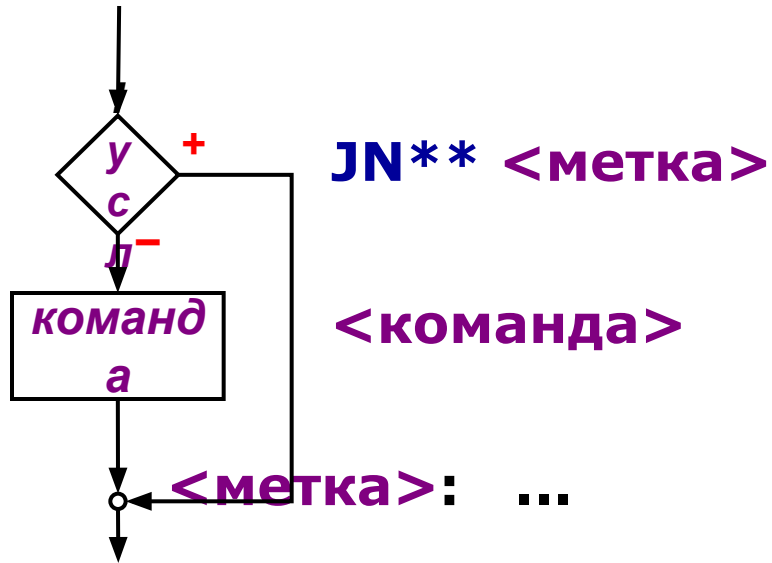
```
           Metka2: ...
```

```
CMP EAX, A[ESI]
```

```
MOV EAX, A[ESI]
```


Реализация алгоритмических структур

Эффективнее будет заменить условие на противоположное



Пример. Фрагмент алгоритма поиска наибольшего элемента массива

```
if(max < A[i])
```

```
{
```

```
  max = A[i];
```

```
}
```

Metka: ...

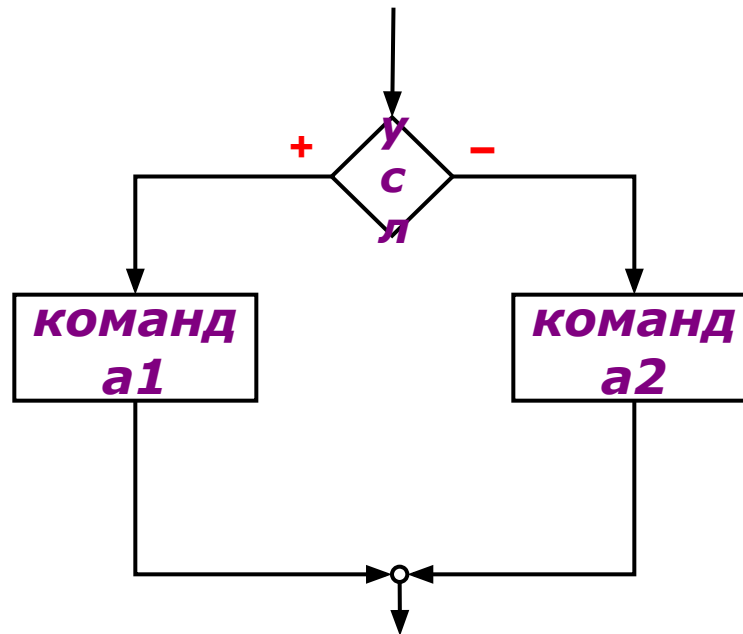
JNL Metka

```
CMP EAX, A[ESI]
```

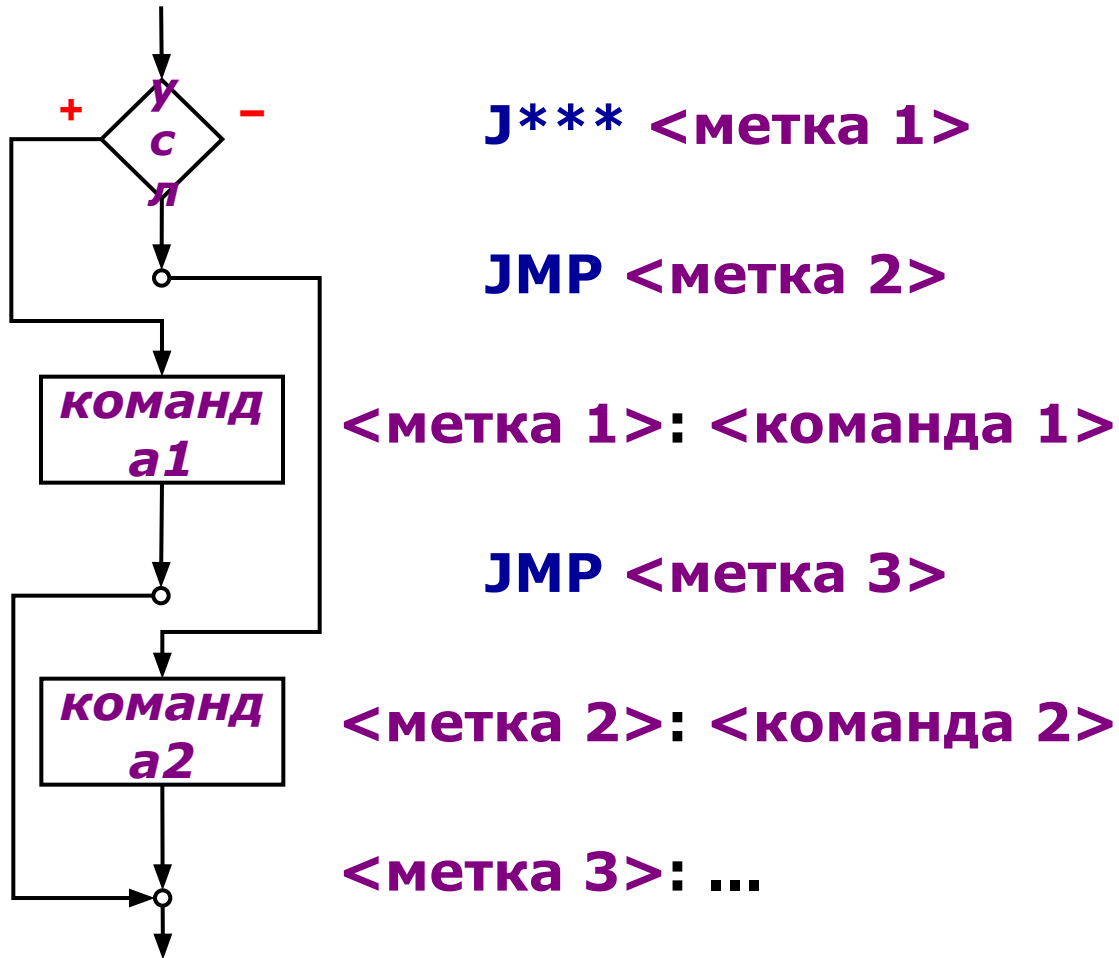
```
MOV EAX, A[ESI]
```

2. Полное ветвление

```
if (усл) { команда1; }  
else { команда2; }
```



Заменим блок-схему на более подходящую



Пример. Поиск наибольшего из двух чисел

```
if(A > B)
```

```
CMP EAX, EBX
```

```
JG Metka1
```

```
JMP Metka2
```

```
{ Metka1:
```

```
  C = A;
```

```
  MOV ECX, EAX
```

```
} JMP Metka3
```

```
else
```

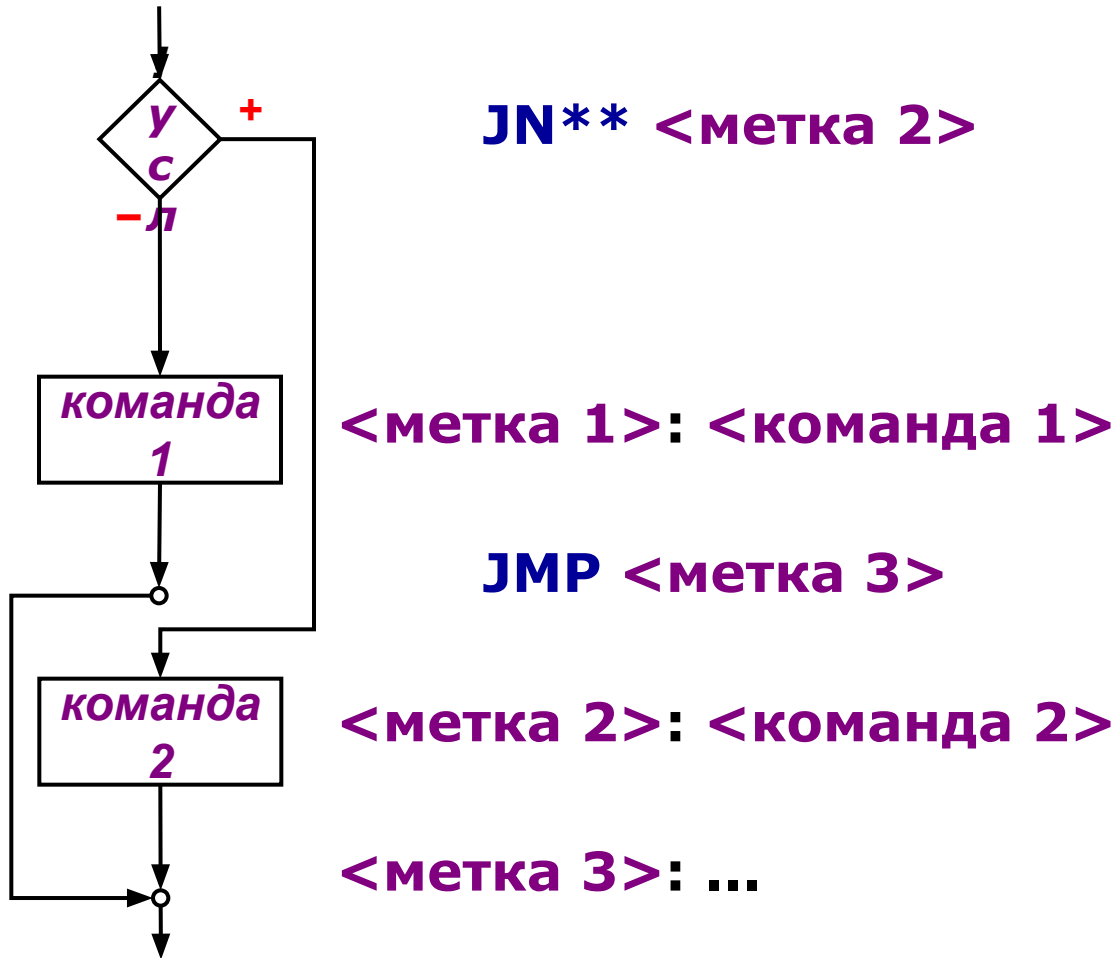
```
{ Metka2:
```

```
  C = B;
```

```
  MOV ECX, EBX
```

```
} Metka3: ...
```

Замена условия упрощает конструкцию:



Пример. Фрагмент алгоритма нахождения НОД

```
if(A > B)
```

```
{
```

```
  A -= B;
```

```
}
```

```
else
```

```
{
```

```
  B -= A;
```

```
}
```

```
  Metka: ...
```

```
  CMP EAX, EBX
```

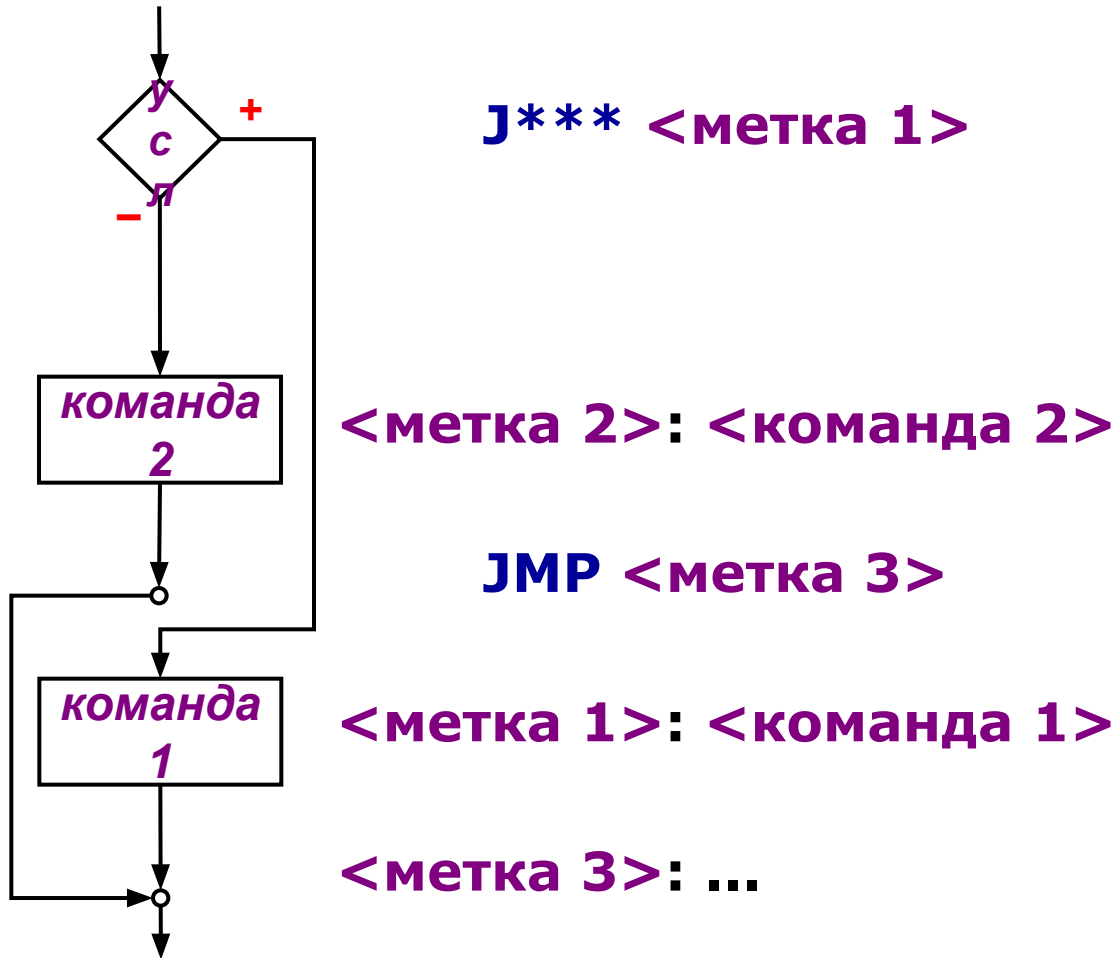
```
  JNG MetkaB
```

```
  MetkaA: SUB EAX, EBX
```

```
  JMP Metka
```

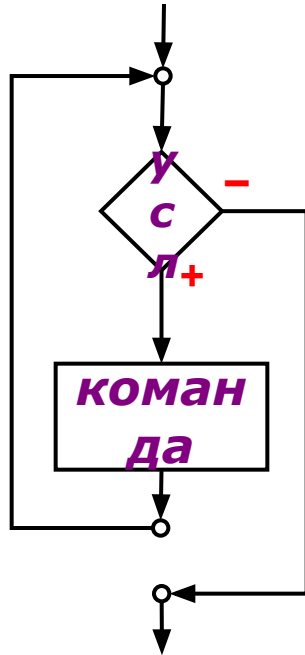
```
  MetkaB: SUB EBX, EAX
```

Можно переставить блоки местами:



3. Цикл с предусловием

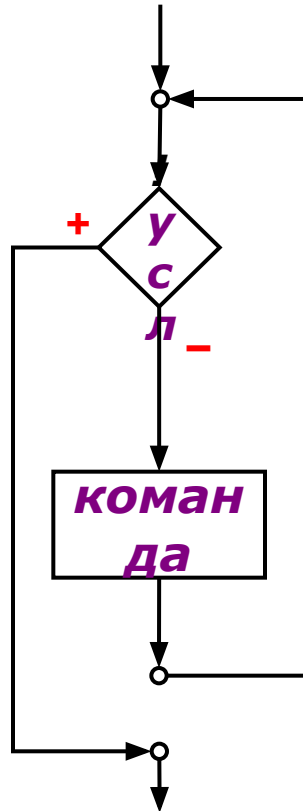
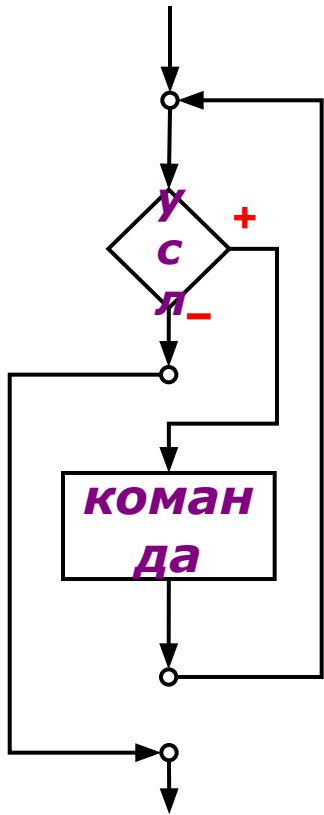
while (усл) { команда; }



Получается из неполного ветвления путем добавления команды перехода в начало конструкции (к проверке условия)

Реализация алгоритмических структур

Возможно несколько вариантов реализации, например:



Реализация алгоритмических структур

Возможно несколько вариантов реализации, например:

NachaloCikla:

...

J* TeloCikla**

JMP KonecCikla

TeloCikla:

...

JMP NachaloCikla

KonecCikla:

...

NachaloCikla:

...

JN KonecCikla**

TeloCikla:

...

JMP NachaloCikla

KonecCikla:

...

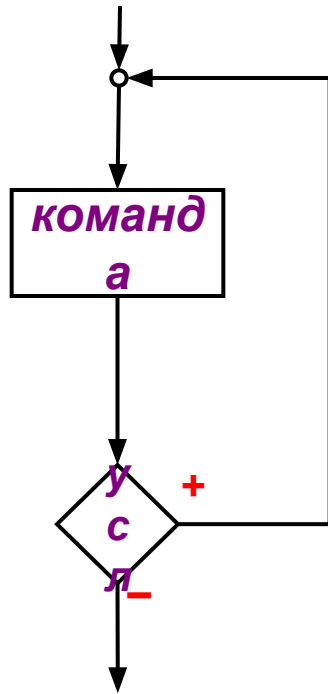
Пример. Алгоритм нахождения НОД

```
while(A != B)
{
  if(A > B)
  {
    A -= B;
  }
  else
  {
    B -= A;
  }
}
```

```
Nachalo:
  CMP EAX, EBX
  JE Konec
  JNG MetkaB
MetkaA:
  SUB EAX, EBX
  JMP Metka
MetkaB:
  SUB EBX, EAX
Metka:
  JMP Nachalo
Konec:
  ...
```

4. Цикл с постусловием

do { **команда**; } **while**(**усл**);



Nachalo:

... ; тело
... ; цикла

Proverka:

...
J* Nachalo**

...

Реализация *цикла с постусловием* на языке Ассемблера оказывается настолько простой, что часто её используют и для реализации *цикла с предусловием*:

JMP Proverka

Nachalo:

... ; тело

... ; цикла

Proverka:

...

J* Nachalo**

...

5. Цикл с параметром

```
for(i = A; i <= B; i++) { команда; }  
for(i = A; i >= B; i--) { команда; }
```

```
MOV ESI, A
```

```
Nachalo:
```

```
CMP ESI, B
```

```
JNLE Конеч
```

```
... ; тело
```

```
... ; цикла
```

```
INC ESI
```

```
JMP Nachalo
```

```
Конеч:
```

```
...
```

```
MOV ESI, A
```

```
Nachalo:
```

```
CMP ESI, B
```

```
JNGE Конеч
```

```
... ; тело
```

```
... ; цикла
```

```
DEC ESI
```

```
JMP Nachalo
```

```
Конеч:
```

```
...
```

Массивы

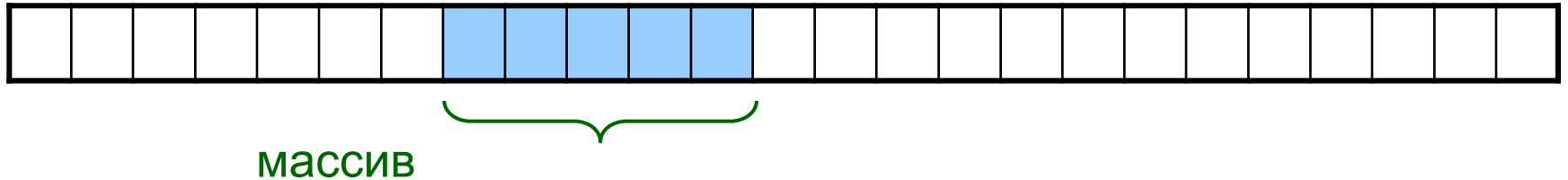
Одним из самых распространенных применений циклов является обработка *массивов* ^{*}.

^{*} *Массив – структурированный тип данных, состоящий из некоторого числа элементов одного типа.*

Массивы

При работе с массивами необходимо помнить, что все элементы массива располагаются в памяти *последовательно*.

Память



Архитектура процессора не накладывает никаких ограничений на смысл и правила использования элементов массивов, т.к. в процессоре *не имеется никаких средств, позволяющих как-то по особенному обрабатывать элементы массивов*, и, вообще, *процессор не отличает массивов от других видов данных*.

Только программист с помощью составленного им алгоритма обработки определяет, как нужно трактовать последовательность байт (слов, удвоенных слов и т.п.), составляющих массив.

Точно также понятие *индекса элемента массива* является условным, поскольку для процессора существуют лишь *адреса ячеек памяти*.

Поэтому задача программиста – *обеспечить верное вычисление адресов элементов массивов*.

В общем случае адрес элемента массива вычисляется по формуле:

$$\text{база} + \text{индекс} * \text{размер_элемента}$$

При работе с массивами используются *косвенные методы адресации*:

- косвенная базовая

INC [EBX]

- косвенная базовая со смещением

INC [EBX - 4]

- косвенная базовая индексная

INC [EBX + ESI * 4]

и т.д.

Схема последовательной обработки элементов массива:

MOV <базовый регистр>, <адрес массива>

<начало цикла>:

...

<обработка> [<базовый регистр>]

...

ADD <базовый регистр>, <размер элемента>

...

<конец цикла>:

Пример. Инициализация элементов массива

MOV EBX, offset Massiv ; адрес начала массива

MOV ESI, 0 ; индекс элемента массива

Nachalo:

CMP ESI, N ; дошли до конца?

JNL Konec

MOV dword ptr [EBX], 0 ; инициализация

INC ESI ; индекс следующего элемента

ADD EBX, 4 ; адрес следующего элемента

JMP Nachalo

Konec:

...

В том случае, когда размер элемента массива равен 1, 2, 4 или 8, при вычислении адреса можно использовать *масштабирование*:

MOV <базовый регистр>, <адрес массива>

MOV <индексный регистр>, 0

<начало цикла>:

...

<обработка> [**<базовый регистр>** +
 <индексный регистр> * **<масштаб>**]

...

INC <индексный регистр>

...

<конец цикла>:

Пример. Сумма элементов массива

MOV EBX, offset Massiv ; адрес начала массива

MOV ESI, 0 ; индекс элемента массива

MOV EAX, 0 ; здесь будет сумма

Nachalo:

CMP ESI, N ; дошли до конца?

JNL Konec

ADD EAX, [EBX + ESI * 4]

INC ESI ; индекс следующего элемента

JMP Nachalo

Konec:

...