Registrene er spesielle heltallsvariabler som er ekstra tett koblet til regneenheten. Vi skal bruke disse registrene:

%EAX %ECX %EDX %EBP %ESP

```
func:     movl    $0,%eax    # Initier til 0.
```
Navnelapp — Instruksjon — Parametre — Kommentar

	`.globl` *xxx*	Navnet *xxx* skal være kjent utenfor filen
`movl`	⟨v_1⟩,⟨v_2⟩	Flytt ⟨v_1⟩ til ⟨v_2⟩.
`cdq`		Omform 32-bits %EAX til 64-bits %EDX:%EAX.
`leal`	⟨v_1⟩,⟨v_2⟩	Flytt ⟨v_1⟩s *adresse* til ⟨v_2⟩.
`pushl`	⟨v⟩	Legg ⟨v⟩ på stakken.
`popl`	⟨v⟩	Fjern toppen av stakken og legg verdien i ⟨v⟩.
`negl`	⟨v⟩	Skift fortegn på ⟨v⟩.
`addl`	⟨v_1⟩,⟨v_2⟩	Adder ⟨v_1⟩ til ⟨v_2⟩.
`subl`	⟨v_1⟩,⟨v_2⟩	Subtraher ⟨v_1⟩ fra ⟨v_2⟩.
`imull`	⟨v_1⟩,⟨v_2⟩	Multipliser ⟨v_1⟩ med ⟨v_2⟩.
`idivl`	⟨v⟩	Del %EDX:%EAX med ⟨v⟩; svar i %EAX; rest i %EDX.
`andl`	⟨v_1⟩,⟨v_2⟩	Logisk AND.
`orl`	⟨v_1⟩,⟨v_2⟩	Logisk OR.
`xorl`	⟨v_1⟩,⟨v_2⟩	Logisk XOR.
`call`	⟨lab⟩	Kall funksjon/prosedyre i ⟨lab⟩.
`enter`	$⟨$n_1$⟩,$⟨n_2⟩	Start funksjon/prosedyre på blokknivå ⟨n_2⟩ med ⟨n_1⟩ byte lokale variabler.
`leave`		Rydd opp når funksjonen/prosedyren er ferdig.
`ret`		Returner fra funksjon/prosedyre.
`cmpl`	⟨v_1⟩,⟨v_2⟩	Sammenligning ⟨v_1⟩ og ⟨v_2⟩.
`jmp`	⟨lab⟩	Hopp til ⟨lab⟩.
`je`	⟨lab⟩	Hopp til ⟨lab⟩ hvis =.
`sete`	⟨v⟩	Sett ⟨v⟩=1 om =, ellers ⟨v⟩=0.
`setne`	⟨v⟩	Sett ⟨v⟩=1 om ≠, ellers ⟨v⟩=0.
`setl`	⟨v⟩	Sett ⟨v⟩=1 om <, ellers ⟨v⟩=0.
`setle`	⟨v⟩	Sett ⟨v⟩=1 om ≤, ellers ⟨v⟩=0.
`setg`	⟨v⟩	Sett ⟨v⟩=1 om >, ellers ⟨v⟩=0.
`setge`	⟨v⟩	Sett ⟨v⟩=1 om ≥, ellers ⟨v⟩=0.

4.5.2 Registre

Vi vil bruke disse registrene:

%EAX er det viktigste arbeidsregisteret. Alle uttrykk eller deluttrykk skal produsere et resultat i %EAX.

%ECX er et hjelperegister som brukes ved aritmetiske eller sammenligningsoperatorer eller til indeks ved oppslag i arrayer.

%EDX brukes til arrayadresser og som hjelperegister ved tilordning og divisjon.

%ESP peker på toppen av kjørestakken.

%EBP peker på den aktuelle funksjonens parametre og lokale variabler.

4.5.2.1 Navn

Hovedprogrammet, funksjoner og prosedyrer beholder sitt Pascal2016-navn men med en endelse så vi unngår dobbeltdeklarasjoner: `proc$`*`name_nnn`*.

Eksterne navn benyttes ved kall på biblioteksprosedyrer; navnet på startpunktet, dvs`main`, er også et eksternt navn. Slike navn skrives som de er i Linux, mens de trenger en understreking («_») foran navnet i Windows and Mac OS X.

Parametre trenger ikke navn i assemblerkoden siden de er gitt utfra posisjonen i parameterlisten: Første parameter har tillegg («offset») 8, andre parameter 12, tredje parameter 16 etc.

Variabler trenger heller ikke navn siden de også ligger på stakken. Nøyaktig hvor de ligger på stakken må kompilatoren vår regne seg frem til; dette avhenger av de andre lokale variablene i samme funksjon eller prosedyre.

Ekstra navn har vi behov for når assemblerkoden skal hoppe i løkker og annet. De får navn **`.L0001`**, `.L0002`, osv.

4.5.3 Oversettelse av uttrykk

Hovedregelen når vi skal lage kode for å beregne uttrykk, er at resultatet av alle uttrykk og deluttrykk skal ende opp i %EAX. Dette gjør kodegenereringen svært mye enklere, men vi får ikke alltid den mest optimale koden.

gcd.pas

```
1  program GCD;
2  /* A program to compute the {greatest common} of two numbers,
3     i.e., the biggest number by which the two original
4     numbers can be divided without a remainder. */
5
6  const v1 = 1071; v2 = 462;
7
8  var res: integer;
9
10 function GCD (m: integer; n: integer): integer;
11 begin
12    if n = 0 then
13       GCD := m
14    else
15       GCD := GCD(n, m mod n)
16 end; { GCD }
17
18 begin
19    res := GCD(v1,v2);
20    write('G', 'C', 'D', '(', v1, ',', v2, ')', '=', res, eol);
21 end.
```

Figur 4.15: Et litt større Pascal2016-program `gcd.pas`

```
1  # Code file created by Pascal2016 compiler 2016-07-29 11:47:15
2          .globl  main
3  main:
4          call    prog$gcd_1              # Start program
5          movl    $0,%eax                 # Set status 0 and
6          ret                             # terminate the program
7  func$gcd_2:
8          enter   $32,$2                  # Start of gcd
9                                          # Start if-statement
10         movl    -8(%ebp),%edx
11         movl    12(%edx),%eax           #   n
12         pushl   %eax
13         movl    $0,%eax                 #   0
14         popl    %ecx
15         cmpl    %eax,%ecx
16         movl    $0,%eax
17         sete    %al                     # Test =
18         cmpl    $0,%eax
19         je      .L0003
20         movl    -8(%ebp),%edx
21         movl    8(%edx),%eax            #   m
22         movl    -8(%ebp),%edx
23         movl    %eax,-32(%edx)          # gcd :=
24         jmp     .L0004
```

```
.L0003:
        movl    -8(%ebp),%edx
        movl    8(%edx),%eax            #   m
        pushl   %eax
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   n
        movl    %eax,%ecx
        popl    %eax
        cdq
        idivl   %ecx
        movl    %edx,%eax               #   mod
        pushl   %eax                    # Push param #2
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   n
        pushl   %eax                    # Push param #1
        call    func$gcd_2
        addl    $8,%esp                 # Pop parameters
        movl    -8(%ebp),%edx
        movl    %eax,-32(%edx)          # gcd :=
.L0004:
                                        # End if-statement
        movl    -32(%ebp),%eax          # Fetch return value
        leave                           # End of gcd
        ret
prog$gcd_1:
        enter   $36,$1                  # Start of gcd
        movl    $462,%eax               #   462
        pushl   %eax                    # Push param #2
        movl    $1071,%eax              #   1071
        pushl   %eax                    # Push param #1
        call    func$gcd_2
        addl    $8,%esp                 # Pop parameters
        movl    -4(%ebp),%edx
        movl    %eax,-36(%edx)          # res :=
        movl    $71,%eax                #   'G'
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        movl    $67,%eax                #   'C'
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        movl    $68,%eax                #   'D'
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        movl    $40,%eax                #   '('
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        movl    $1071,%eax              #   1071
        pushl   %eax                    # Push next param.
        call    write_int
        addl    $4,%esp                 # Pop param.
        movl    $44,%eax                #   ','
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        movl    $462,%eax               #   462
        pushl   %eax                    # Push next param.
        call    write_int
        addl    $4,%esp                 # Pop param.
        movl    $41,%eax                #   ')'
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        movl    $61,%eax                #   '='
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        movl    -4(%ebp),%edx
        movl    -36(%edx),%eax          #   res
        pushl   %eax                    # Push next param.
        call    write_int
        addl    $4,%esp                 # Pop param.
        movl    $10,%eax                #   10
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        leave                           # End of gcd
        ret
```

I tabell 4.5 er vist hvilken kode som må genereres for å hente en verdi ⟨n⟩, en enkel variabel $\langle v^{(b)o} \rangle$ (der b er blokknivået og `o er variabelens «offset»`), et arrayelement $\langle a^{(b)o} \rangle$`[`⟨e⟩`]` eller et uttrykk i parenteser (<e>) inn i register %EAX.

⟨n⟩	⇒	`movl` $ ⟨n⟩`,%eax`
$\langle v^{(b)o} \rangle$	⇒	`movl` $-4b$`(%ebp),%edx` `movl` o`(%edx),%eax`

$\langle a^{(b)o} \rangle$`[`⟨e⟩`]`	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩
		`subl` $ *low*`,%eax` (Dropp om *low*=0)
		`movl` $-4b$`(%ebp),%edx`
		`leal` o`(%edx),%edx`
		`movl (%edx,%eax,4),%eax`

`(`⟨e⟩`)`	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩

Tabell 4.5: Kode for å hente en verdi inn i %EAX

`+` ⟨e⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩
`-` ⟨e⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩ `negl %eax`
`not` ⟨e⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩ `xorl $1,%eax`

Tabell 4.6: Kode generert av unære operatorer i uttrykk

$\langle e_1 \rangle$ `+` $\langle e_2 \rangle$	⇒	⟨Beregn $\langle e_1 \rangle$ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn $\langle e_2 \rangle$ med svar i %EAX⟩
		`movl %eax,%ecx`
		`popl %eax`
		`addl %ecx,%eax`

$\langle e_1 \rangle$ `div` $\langle e_2 \rangle$	⇒	⟨Beregn $\langle e_1 \rangle$ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn $\langle e_2 \rangle$ med svar i %EAX⟩
		`movl %eax,%ecx`
		`popl %eax`
		`cdq`
		`idivl %ecx`

$\langle e_1 \rangle$ `=` $\langle e_2 \rangle$	⇒	⟨Beregn $\langle e_1 \rangle$ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn $\langle e_2 \rangle$ med svar i %EAX⟩
		`popl %ecx`
		`cmpl %eax,%ecx`
		`movl $0,%eax`
		`sete %al`

Tabell 4.7: Kode generert av binære operatorer i uttrykk

	⇒	

Tabell 4.8: Kode generert av tom setning

`begin` ⟨S_1⟩`;` ⟨S_2⟩`;` `...` `end`	⇒	⟨S_1⟩ ⟨S_2⟩ ⋮

Tabell 4.9: Kode generert av sammensatt setning

⟨$v^{(b)o}$⟩ `:=` ⟨e⟩`;`	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩
		`movl` $-4b$`(%ebp),%edx`
		`movl` `%eax,`o`(%edx)`

⟨$a^{(b)o}$⟩`[`⟨e_1⟩`]` `:=` ⟨e_2⟩	⇒	⟨Beregn ⟨e_2⟩ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn ⟨e_1⟩ med svar i %EAX⟩
		`subl $`low`,%eax` (Dropp om low=0)
		`movl` $-4b$`(%ebp),%edx`
		`leal` o`(%edx),%edx`
		`popl %ecx`
		`movl %ecx,(%edx,%eax,4)`

⟨$f^{(b)}$⟩ `:=` ⟨e⟩`;`	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩
		`movl` $-4(b+1)$`(%ebp),%edx`
		`movl %eax,-32(%edx)`

Tabell 4.10: Kode generert av tilordning

f`(`⟨e_1⟩`,` ⟨e_2⟩`)`	⇒	⟨Beregn ⟨e_2⟩ med svar i %EAX⟩
		`pushl %eax`
		⟨Beregn ⟨e_1⟩ med svar i %EAX⟩
		`pushl %eax`
		
		`call proc$`f_n
		
		`addl $8,%esp`

Tabell 4.11: Kode generert av prosedyrekall

`write(`⟨int-e⟩`)`	⇒	⟨Beregn ⟨int-e⟩ med svar i %EAX⟩
		`pushl %eax`
		`call write_int`
		`addl $4,%esp`

`write(`⟨char-e⟩`)`	⇒	⟨Beregn ⟨char-e⟩ med svar i %EAX⟩
		`pushl %eax`
		`call write_char`
		`addl $4,%esp`

`write(`⟨bool-e⟩`)`	⇒	⟨Beregn ⟨bool-e⟩ med svar i %EAX⟩
		`pushl %eax`
		`call write_bool`
		`addl $4,%esp`

Tabell 4.12: Kode generert av kall på `write`

`if` ⟨e⟩ `then` ⟨S⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩ `cmpl $0,%eax` `je` ⟨lab⟩ ⟨S⟩ ⟨lab⟩:
`if` ⟨e⟩ `then` ⟨S_1⟩ `else` ⟨S_2⟩	⇒	⟨Beregn ⟨e⟩ med svar i %EAX⟩ `cmpl $0,%eax` `je` ⟨lab_1⟩ ⟨S_1⟩ `jmp` ⟨lab_2⟩ ⟨lab_1⟩: ⟨S_2⟩ ⟨lab_2⟩:

Tabell 4.13: Kode generert av if-setning

`while` ⟨e⟩ `do` ⟨S⟩	⇒	⟨lab_1⟩: ⟨Beregn ⟨e⟩ med svar i %EAX⟩ `cmpl $0,%eax` `je` ⟨lab_2⟩ ⟨S⟩ `jmp` ⟨lab_1⟩ ⟨lab_2⟩:

Tabell 4.14: Kode generert av while-setning

`function` ⟨f⟩ `(...):` ⟨type⟩;
⟨D⟩
`begin`
⟨S⟩
`end`
	⇒	`func$`⟨f⟩*_n*:
		`enter $`⟨32+ant byte i ⟨D⟩⟩`,$`⟨blokknivå⟩
		⟨S⟩
		`movl -32(%ebp),%eax`
		`leave`
		`ret`

`procedure` ⟨p⟩ `(...);`
⟨D⟩
`begin`
⟨S⟩
`end`
	⇒	`proc$`⟨p⟩*_n*:
		`enter $`⟨32+ant byte i ⟨D⟩⟩`,$`⟨blokknivå⟩
		⟨S⟩
		`leave`
		`ret`

Tabell 4.15: Kode generert av funksjons- og prosedyredeklarasjon

`program` *xx*;
⟨D⟩
`begin`
⟨S⟩
`end.`
	⇒	`.globl main`
		`main:`
		`call prog$`*xx_n*
		`movl $0,%eax`
		`ret`
		`prog$`*xx_n*:
		`enter $`⟨32+ant byte i ⟨D⟩⟩`,$1`
		⟨S⟩
		`leave`
		`ret`

Tabell 4.16: Kode generert av hovedprogrammet