

# Computer Arkitektur

## Maskin-niveau programmering II

### Kontrol and Arrays

Forelæsning 5  
Brian Nielsen

*Credits to  
Randy Bryant & Dave O'Hallaron (CMU)*

# Idag

- Hvordan laves højniveau kontrol-strukturer i assembler ?
  - If-then-else
  - while(){}
- Hvordan håndterer maskinen betingelser?
  - if(a<b) ..
- Hvordan repræsenterer maskinen sammensatte datatyper?
  - Arrays i 1D og 2D'
  - Structs

# Kursusgang 4-6: x86-64 Assembler

## Intro til x86 Assembler: Adressering

- X86 Historik
- Assembly og objekt kode
  - gcc,as,gdb,objdump
- Instruktionsæt arkitektur
- Words (Q,L,W, B)
- Registres
- Addressering
  - Immediate,
  - Registres
  - Mem
- Aritmetiske operationer
- Logiske operationer

## X86 Assembler: Kontrol- og data-strukturer

- Sammenligner
- Betingelsesflag
- Selektion
  - if-then-else
  - Betinget tildeling
- Iteration
  - While
  - Do-while
  - For
- Data-strukturer
  - Layout af Arrays i 1D og 2D
  - Indeksering
  - Structs
  - Alignment

## X86 Assembler: Procedurekald

- Køretidsstak
- Push/Pop
- Kald og retur
- Parameteroverførsel
- Kalder/kaldte gemte registre
- Lokale variable
- Stack-frame
- Rekursion
- Buffer-overløb
  - Sikkerhedshuller og angreb
- Kanarier
- Addresserums randomisering
- Non-executable stak beskyttelse

- Hvordan ser maskinens grænseflade ud overfor programmøren?
- Hvad er en Instruktionsæt Arkitektur?
- Hvordan kodes høj-niveau (C) kontrol strukturer op?
- Hvordan opstår og forebygges sårbarheder ved bufferoverløb?

Betingelser

# Processor tilstand (x86-64)

- Information om det kørende program program
  - Temporære data ( `%rax, ...` )
  - Placering af køretidsstak top ( `%rsp` )
  - Nuværende kontrol punkt ( `%rip, ...` )
  - Status på nyligt udførte tests ( **CF, ZF, SF, OF** )

Betingelsesflag  
Condition codes

## Registre

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

stak top →

`%rip`

Program tæller  
Instruction pointer  
(program counter)

CF	ZF	SF	OF
----	----	----	----



# Condition Codes (Implicit tildeling)

- Enkelt-bit registre
  - **CF** Carry Flag (for unsigned)
  - **SF** Sign Flag (for signed)
  - **ZF** Nul (Zero) Flag
  - **OF** Overløb (Overflow) Flag (for signed)

Mente-flag  
Fortegns-flag  
Nul-flag  
Overløbs-flag

- Implicit sat af **aritmetiske operationer**
  - Tænk på dem som en sideeffekt
- Eksempel: **`addq Src, Dest`  $\leftrightarrow$  `t = a+b`**

**CF sat til 1** hvis der er mente fra mest betydende bit (unsigned overløb)

**ZF sat til 1** hvis `t == 0`

**SF sat til 1** hvis `t < 0` (fortolket som signed)

**OF sat til 1** hvis two's-complement (signed) overløb, dvs. hvis

`(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- NB: Påvirkes ikke af `leaq` instruktionen



# Condition Codes (Explicit tildeling: Compare)

- Explicit sat af **Compare** Instruktion
  - `cmpq Src2, Src1`
  - **cmpq b, a** beregner **a-b** (som `subq b, a`), uden at ændre destination!
- **CF sat** hvis mente fra mest betydende bit (anvendt som sammenligning af unsigned)
- **ZF sat** hvis `a == b`
- **SF sat** hvis `(a-b) < 0` (som signed)
- **OF sat** hvis two's-complement (signed) overløb. Dvs. når

`(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

```
cmp b,a =a-b
burde give positivt tal, når a>0&&b<0
cmpq -4, 5
5- (-4) =9
```

```
cmp b,a =a-b
burde give negativt tal, når a<0&&b>0
cmpq 5, -4
-4 - (5) = -9
```



# Fortolkning af Condition Codes v. CMP

Betingelse (efter cmp)	Beskrivelse
ZF	Equal / Zero
$\sim ZF$	Not Equal / Not Zero
SF	Negative Sign
$\sim SF$	Nonnegative Sign
$\sim (SF \wedge OF) \wedge \sim ZF$	Greater (Signed)
$\sim (SF \wedge OF)$	Greater or Equal (Signed)
$(SF \wedge OF)$	Less (Signed)
$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
$\sim CF \wedge \sim ZF$	Above (unsigned)
CF	Below (unsigned)

`cmpq b, a` #compare a:b  
#a-b  
ZF sand hvis  $a==b$

`cmpq b, a` #compare a:b  
#a-b  
SF xor OF sand hvis  $a < b$

OF	SF	a<b?	a-b tilfælde
0	0	0	a størst
0	1	1	b størst
1	0	1	$(a < 0 \ \&\& \ b > 0 \ \&\& \ (a-b) > 0)$
1	1	0	$(a > 0 \ \&\& \ b < 0 \ \&\& \ (a-b) < 0)$

NB: **TEST** instruktion analog (bruger AND)



# Aflæsning af Condition Codes

- **SetX Dst** instruktioner, X angiver variant
  - Sætter destination byte til 0 or 1 afhængigt af kombinationer af condition codes
  - Dst er et "byte" register (eller byte destination i hukommelsen)
  - Hvis resultat skal leveres som 64-bit: resterende (7) bytes nulstilles med `movzbl`

SetX	Betingelse (efter cmp)	Beskrivelse
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative Sign
setns	~SF	Nonnegative Sign
setg	~(SF^OF) & ~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)   ZF	Less or Equal (Signed)
seta	~CF & ~ZF	Above (unsigned)
setb	CF	Below (unsigned)

```
long gt (long x, long y)
{
    return x > y;
}
```

```
#%rdi=x, %rsi=y
#%rax=result

cmpq    %rsi, %rdi #Compare x:y (x-y)
setg    %al        #Set when >
movzbl  %al, %eax  #Zero rest of %rax
ret
```

Forgrening og kontrol strukturer



# Ubetinget Program forgrening: “Jumping”

- Ubetingede hop

- Direkte

- **Jmp** Adresse/label      `#jmp mylabel`

- Indirekte

- **Jmp** \*operand      `#jmp *%rax`  
                             `#jmp *(%rax)`

C-variant

```
long x=0;  
  
loop:  
    x++;  
goto loop;
```

ASM

```
movl $0, %eax  
  
loop:  
    incq %rax  
jmp loop
```

Assembler/Linker erstatter labels med adresser i objekt-koden  
(typisk indkodet som PC relative hop, jvf afsnit 3.6.4)



# Betinget Program forgrening: “Jumping”

- Betingede Hop til andet sted i koden eller ej, afhængigt af aktuelle værdier af “condition codes flag”

jX	Betingelse	Beskrivelse
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative Sign
jns	$\sim SF$	Nonnegative Sign
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

# Betinget forgrenig "if-statements"

## Generelt format

### C kode

```
If (cond)  
    then-blok  
Else  
    else-blok
```

- Kode-blok

```
{  
    Statement1;  
    Statement2;  
    ...  
    Statementn;  
}
```



# Betinget forgrening: Eksempel

- Oversættelse fra C til ASM `gcc -Og -S absdiff.c`

## Original

```
long absdiff
(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

## ASM

```
##%rdi=x, %rsi=y
##%rax=result
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:       # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

## C "goto" version

```
long absdiff_j
(long x, long y)
{
    long result;
    long ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

NB: Negering af test til at springe over "then-blok"

# Ingen ELSE blok

```
long absdiff
(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
    result = y-x;
    return result;
}
```

```
long absdiff_j
(long x, long y)
{
    long result;
    long ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    //goto Done
Else:
Done:
    return result;
}
```

## Fordelagtigt

- Når der ingen "Else" blok undgås "goto Done"
- Goto's/Jumps er skidt for pipeline og performance

# Oversættelse af generel betinget tildeling (Ved brug af forgrening)

## C kode

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

## Goto Version

```
ntest = !Test;  
if (ntest) goto Else;  
val = Then_Expr;  
goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

- Strategi a la if-then-else
- Skriv separate kode blokke for then & else udtryk
- Udfør den rigtige



# Anvendelse af betinget tildeling “Conditional Moves”

- Conditional Move instruktioner  
**`cmovX Src, Dest`**
  - Instruktionen understøtter:  
**`if (Test) Dest ← Src`**
  - X angive betingelsen: e(qual), le, ge, ...
  - GCC forsøger at anvende dem
    - Men kun når den ved at det er sikkert
- Hvorfor?
  - Forgrening via jumps er meget forstyrrende for instruktions-flow igennem processorens pipeline
  - Conditional move kan ofte undgå ændring af kontrol-flow

## C Kode

```
val = Test  
    ? Then_Expr  
    : Else_Expr;
```

## Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

NB: beregner både "then", og "else" udtryk  
Udvælger den version, der skal bruges

# Eksempel på Conditional Move

```
long absdiff
(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
#%rdi=x, %rsi=y
#%rax=result
```

```
absdiff:
    movq    %rdi, %rax    # x
    subq    %rsi, %rax    # result = x-y
    movq    %rsi, %rdx    # y
    subq    %rdi, %rdx    # eval = y-x
    cmpq    %rsi, %rdi    # x:y
    cmovle  %rdx, %rax    # if <=, result = eval
    ret
```

# Hvor Conditional Move IKKE bør anvendes

## Dyre beregninger

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Begge værdier beregnes!
- Kun gavnligt når beregningerne er simple

## Risikable Beregninger

```
val = p ? *p : 0;
```

- Begge værdier beregnes!
- Kan have uønskede side-effekter

## Beregninger med side effekter

```
val = x > 0 ? x*=7 : x+=3;
```

- Begge værdier beregnes!
- Skal være fri for side-effekter

Iteration

# General oversættelse af "Do-While"

## C kode

```
do  
    Body  
while (Test) ;
```

- Body: {  
    Statement<sub>1</sub>;  
    Statement<sub>2</sub>;  
    ...  
    Statement<sub>n</sub>;  
}

## Goto Version

```
loop:  
    Body  
    if (Test)  
        goto loop
```

# “Do-While” Eksempel

## C kode

```
long pcount_do
(unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

## Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

## ASM

```
#%rdi=x
#%rax=result
movl    $0, %eax    # result = 0
.L2:                                # loop:
movq    %rdi, %rdx   # t= x
andl    $1,%edx      # t= x & 0x1
addq    %rdx, %rax   # result += t
shrq    %rdi         # x >>= 1
jne     .L2          # if (x) goto loop
ret
```

<u>jne</u>	~ZF	Not Equal / Not Zero
------------	-----	----------------------

- Tæl antallet af 1-bits i argument  $x$  (“popcount”):  $0110.1110_2 \Rightarrow 5$
- Betinget forgrening bestemmer om løkken skal fortsætte eller stoppe

# Generel oversættelse af “While” #1

- “Jump-to-middle” oversættelse
- Bruges med -Og

## While version

```
while (Test)  
    Body
```



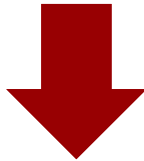
## Goto Version

```
    goto test;  
loop:  
    Body  
test:  
    if (Test)  
        goto loop;  
done:
```

# General oversættelse af “While” #2

## While version

```
while (Test)  
    Body
```



## Do-While Version

```
if (!Test)  
    goto done;  
do  
    Body  
    while (Test);  
done:
```



## Goto Version

```
if (!Test)  
    goto done;  
loop:  
    Body  
    if (Test)  
        goto loop;  
done:
```

- “Guarded do-while” konvertering
- Brugt med -O1

Tillader at compiler ofte kan optimere løkken, da det er et faktum at !test holdt.  
Se simpelt eksempel i bog



# “For” Loop → While Loop

For Version

```
for (Init; Test; Update )  
    Body
```



While Version

```
Init;  
while (Test) {  
    Body  
    Update;  
}
```

# Switch med “Jump Tabeller” (indirekte jmp)

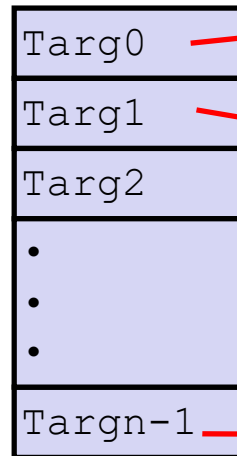
## Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
    . . .  
  case val_n-1:  
    Block n-1  
}
```

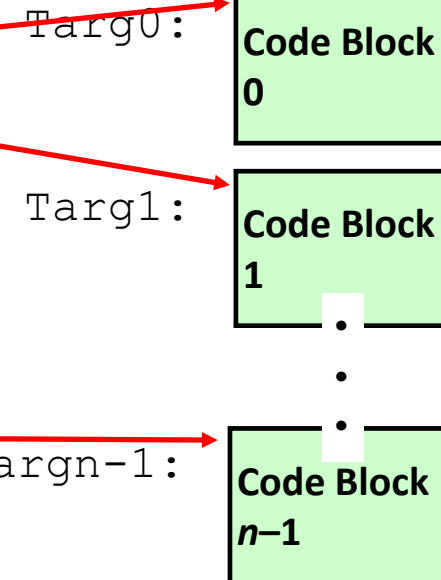
## Jump Table

(array af adresser)

.jtab:



## Jump Targets



## Translation (Extended C)

```
goto *JTab[x];
```

*Indirekte  
hop*



## ASM

```
switch_eg:  
  #x in %rdi  
  ...  
  jmp      *.jtab(,%rdi,8) # goto *JTab[x]
```

Resultat:  $O(1)$

If-then-else kæde kræver  $O(n)$  evalueringer

# Arrays

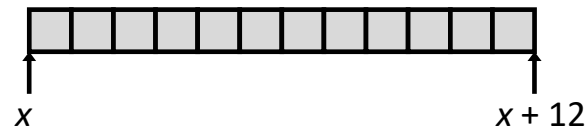
# Array Allokering

- Grundlæggende princip

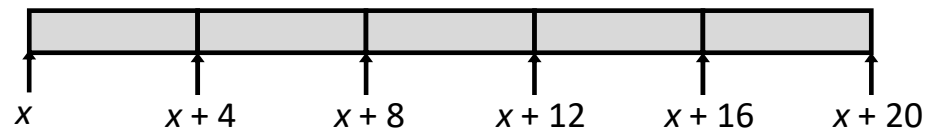
$T$   $a[L];$

- Array med navnet  $a$  af data typen  $T$  og længde  $L$
- Sammenhængende allokeret blok af størrelsen  $L * \text{sizeof}(T)$  bytes i hukommelsen
- $a$  udpeger startadressen på blokken

`char string[12];`



`int val[5];`



`double a[3];`



`char *p[3];`

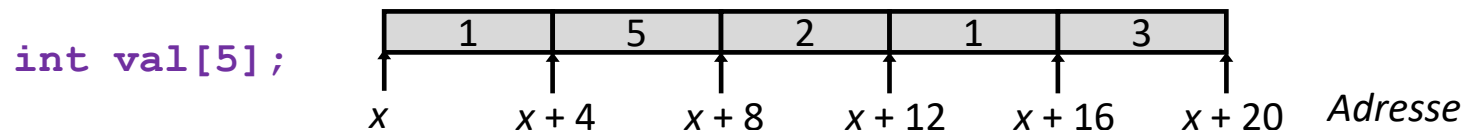


# Array indeksering

- Grundlæggende princip for array af data typen  $T$ , længde  $L$

$T a[L];$

- Identifier  $a$  kan anvendes som pointer til array element 0: Type  $T^*$



- C-Reference Type      Værdi

<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	$x$
<code>val+1</code>	<code>int *</code>	$x+4$
<code>&amp;val[2]</code>	<code>int *</code>	$x+8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	$x+4i$

//NB! Pointer Aritmetik!

$a[i]$  ækvivalent med  $*(a+i)$  //  $M[a+i*\text{sizeof}(a)]$

$\&a[i] = a+i$  // effektiv adresse  $= a + \text{sizeof}(a)*i$

# Eksempel på array indeksering

*Zip codes=post numre  
CMU=Carnegie Mellon Uni*

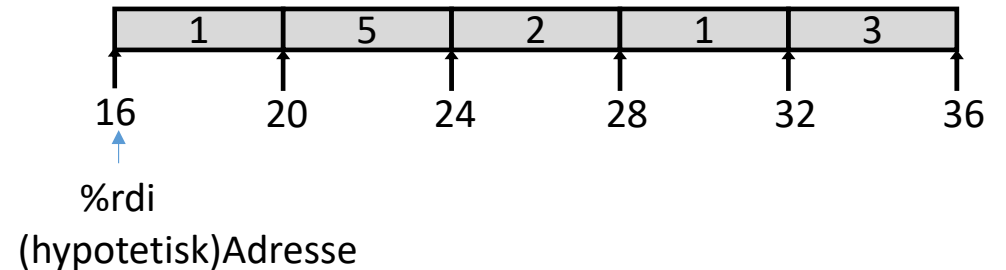
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
```

```
int get_digit (zip_dig z, int digit){
    return z[digit];
}
```

## Assembler

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax # z[digit]
```



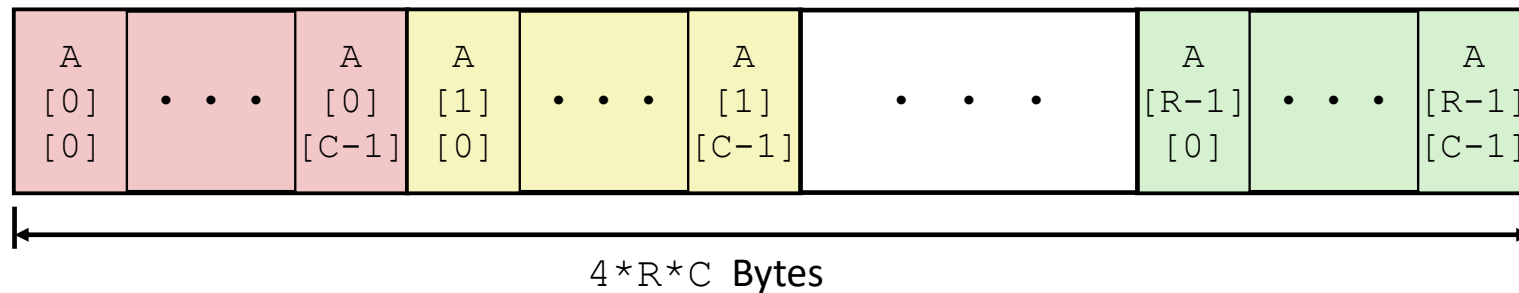
- Register `%rdi` indeholder start adresse på array
- Register `%rsi` indeholder array indeks
- Ønsket ciffer på  $\%rdi + 4 * \%rsi$
- Brug adresseringen  $(\%rdi, \%rsi, 4)$

# Multi-dimensionelle (Nestede) Arrays

- Erklæres i C med
  - $T \text{ } \mathbf{A}[R][C];$
  - 2D array af data typen  $T$
  - $R$  rækker,  $C$  kolonner
  - Type  $T$  element kræver  $K$  bytes
- Array Størrelse
  - $R * C * K$  bytes
- Hukommelseslayout
  - Række-baseret
  - "Row-Major"

$$\begin{bmatrix} A[0][0] & \cdot & \cdot & \cdot & A[0][C-1] \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ A[R-1][0] & \cdot & \cdot & \cdot & A[R-1][C-1] \end{bmatrix}$$

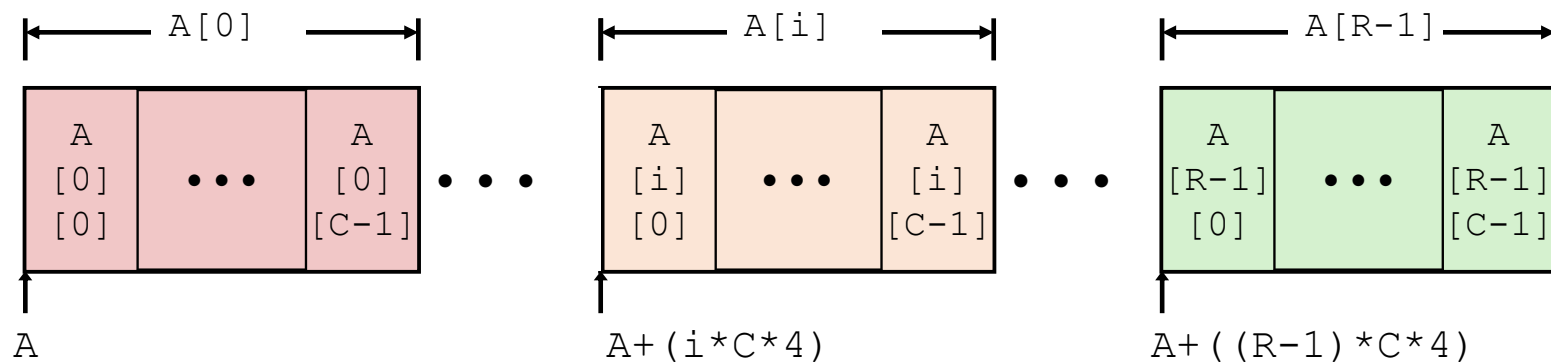
```
int A[R][C];
```



# Indeksering af rækker i nested array

- Rækkevektorer
  - $\mathbf{A}[i]$  er et array af  $C$  elementer
  - Hver element af type  $T$  kræver  $K$  bytes
  - Start adresse  $\mathbf{A} + i * (C * K)$

```
int A[R][C];
```





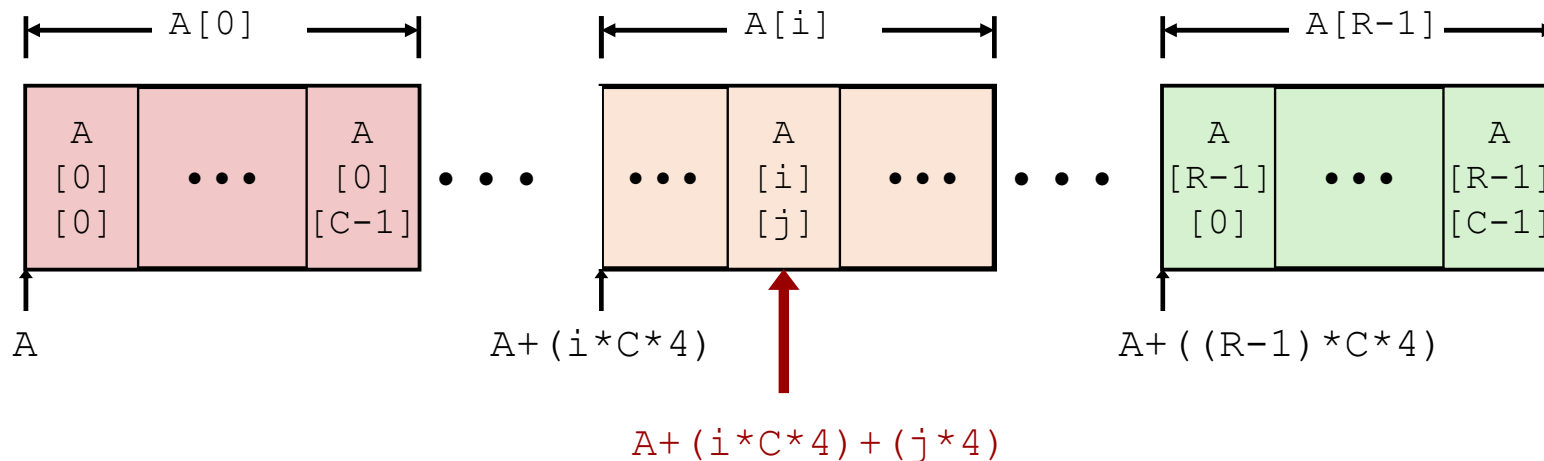
# Adgang til element i Nested Array

- Array elementer
  - `T A[R][C]; //K=sizeof(T)`
  - `A[i][j]` er et element af type `T`, som optager `K` bytes
  - Adresse  $A + i * (C * K) + j * K = A + (i * C + j) * K$

Adresse af element  $i, j$ :

start + skip  $i$  rækker af  $C$  elementer af størrelse  $K$  plus  $j$  kolonner af størrelsen  $K$

```
int A[R][C];
```



# Eksempel på adgang til element i Nested Array

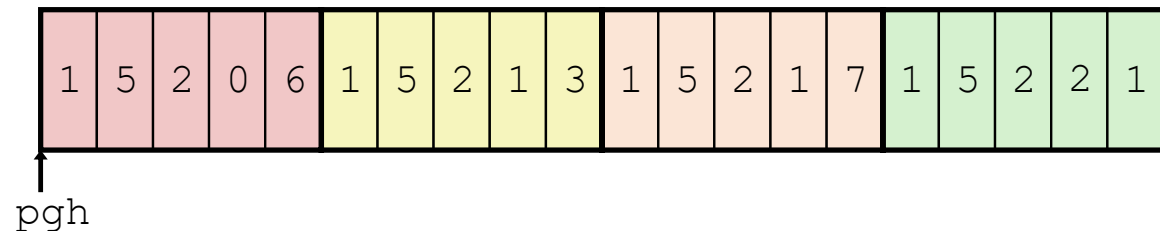
```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
//samme som int pgh[4][5]
```

```
int get_pgh_digit
(int index, int dig)
{
    return pgh[index][dig];
}
```

```
# %rdi = index, %rsi=digit
leaq    (%rdi,%rdi,4), %rax    # 5*index
addl    %rax, %rsi             # 5*index+dig
movl    pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

*Zip codes=post numre*  
*PGH= Pittsburgh (CMU)*

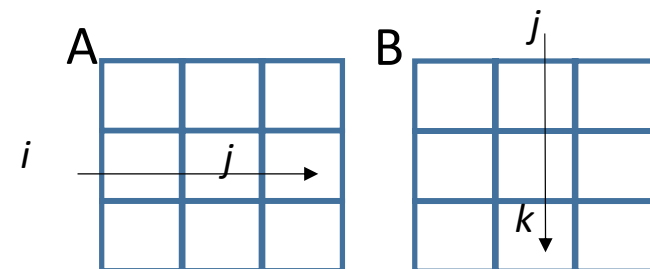
“Row-Major” layout i hukommelsen



- **pgh[index][dig]** er en **int**
- Adresse: skip index rækker med 5 elementer+dig elementer af 4 bytes
  - = **pgh + 4\*(5\*index + dig)**

# Optimeret adgang

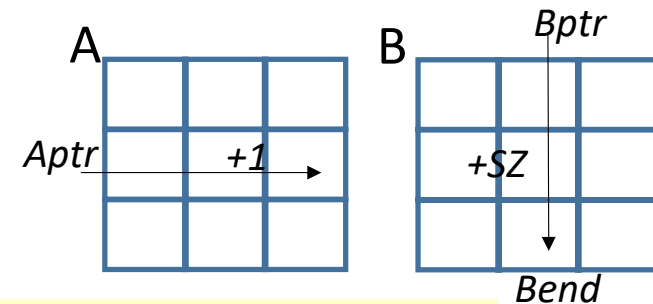
- Indre produkt af A række  $i$  og B kolonne  $j$



```
#define SZ 16;
typedef int matrix[SZ][SZ]; //define type 16*16matrix

int fix_prod_ele(matrix A, matrix B, long i, long k){
    long j;
    int result=0;
    for (j = 0; j < SZ; j++)
        result +=A[i][j]*B[j][k];
    return result;
}
```

# Optimieret adgang ved brug af pointers



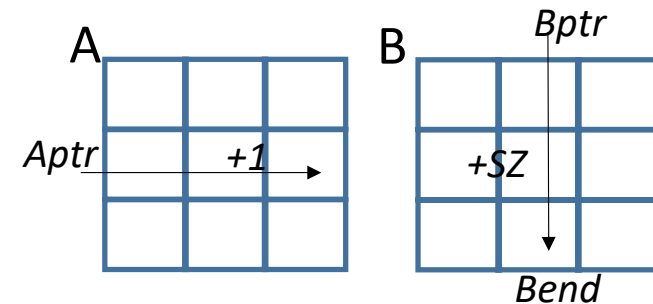
```
int fix_prod_ele(matrix A, matrix B, long i, long k) {
    int* Aptr=&A[i][0];          //points to elements in A row i
    int* Bptr=&B[0][k];           //points to elements in B col k
    int* Bend =&B[SZ][k];        //marks stopping point for Bptr

    int result=0;
    do {
        result+= (*Aptr) * (*Bptr); //compute sum of products
        Aptr++;          //advance to next A col
        Bptr+=SZ;         //advance to next B row
    } while(Bptr!=Bend) //test stopping point

    return result;
}
```

Anvendes af compiler med "-O1" når den kan gennemskue gennemløbet.

# Optimeret adgang “-O1”



```
#%rdi A, %rsi B, %rdx i, %rcx k
```

```
fix_prod_ele:
```

```
    salq $6, %rdx
```

```
    addq %rdx, %rdi
```

```
    leaq (%rsi,%rcx,4), %rcx
```

```
    leaq 1024(%rcx), %rsi
```

```
    movl $0, %eax
```

```
.L3:
```

```
    movl (%rdi), %edx
```

```
    imull (%rcx), %edx
```

```
    addl %edx, %eax
```

```
    addq $4, %rdi
```

```
    addq $64, %rcx
```

```
    cmpq %rsi, %rcx
```

```
    jne .L3
```

```
    rep ret
```

```
#64*i: skip i rows of 16*4 bytes (SZ=16)
```

```
#Aptr=A+64i
```

```
#Bptr=B+4k
```

```
#Bend=Bptr+16*16*4
```

```
#res=0
```

```
#do {
```

```
#read *Aptr
```

```
#mult *Aptr by *Pptr
```

```
#add to res
```

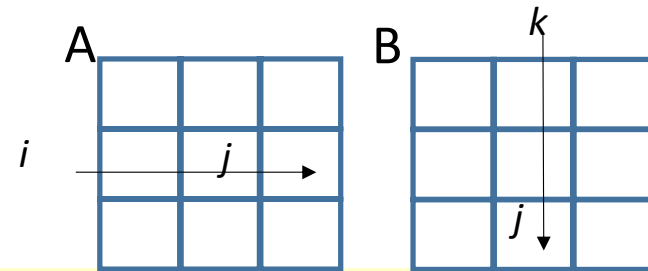
```
#incr Aptr by one col=4 bytes
```

```
#incr Bptr by 1 row=4*16
```

```
#compare Bptr w. Bend
```

```
#While != goto L3
```

# Direkte oversættelse “-Og”



```
#%rdi A, %rsi B, %rdx i, %rcx k
```

```
fix_prod_ele:
```

```
    movl    $0, %eax           #res=0
    movl    $0, %r8d           #j=0
    salq    $6, %rdx           #i=i*64
    addq    %rdx, %rdi         #A+64i A[i,0]
    jmp     .L2                #while
```

```
.L3:
```

```
    movq    %r8, %r9           #j
    salq    $6, %r9           #j*64
    addq    %rsi, %r9          #B+64j
    movl    (%r9,%rcx,4), %r9d  #*(B+64k+4k)
    imull   (%rdi,%r8,4), %r9d  #Mult by *(A+64i+4*j)
    addl    %r9d, %eax         #res+=product
    addq    $1, %r8            #j++
```

```
.L2:
```

```
    cmpq    $15, %r8           #compare 15:j
    jle     .L3                #while (j<=15) goto L3
    rep     ret
```

```
result +=
A[i][j] * B[j][k];

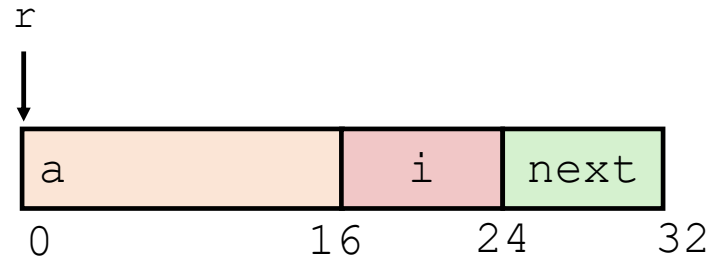
A + 4(16i + j) = A + 64i + 4j
B + 4(16j + k) = B + 64j + 4k
```

7 instruktioner i løkken vs. 5  
~2/7 = 29% forbedring

Strukturer – heterogene datatyper

# Repræsentation af Structs

```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```

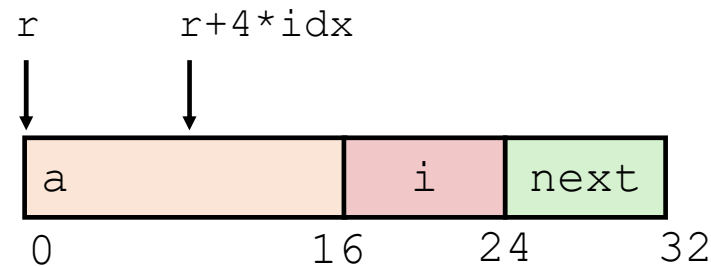


- Strukturer repræsenteres som en sammenhængende blok hukommelse
  - **Stor nok til at indeholde alle medlemmer**
- Medlemmerne udlægges i same rækkefølge som deres erklæring
  - **Selvom anden ordning kunne give mere kompakt repræsentation**
- Compiler udregner størrelse + positioner af medlemmer
  - **Maskin niveau programmer kender intet til structs fra kildekode, kun medlemmernes (relative) start position**



# Generering af pointer til struktur medlem

```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- Hvert medlems "offset" (relativ position til start på struct) kan bestemmes på compile tid
- Generering af pointer til array element `a`
  - Offset 0
  - Beregn som:  $r + 4 \cdot idx$
- Dereferering af "next" pointer
  - Offset 24
  - `r=r->next`

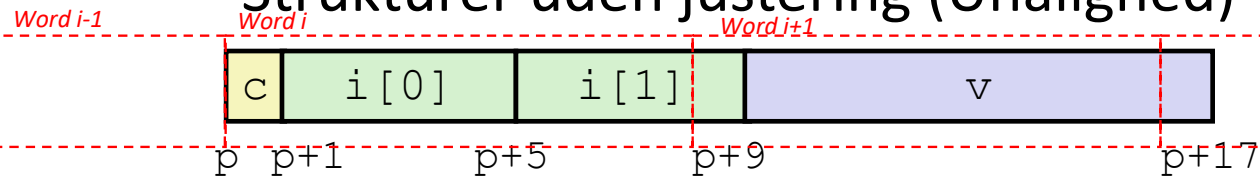
```
int *get_ap  
(struct rec *r, size_t idx)  
{  
    return &r->a[idx];  
}
```

```
# r in %rdi, idx in %rsi  
leaq  (%rdi,%rsi,4), %rax  
ret
```

```
movq  24(%rdi), %rdi
```

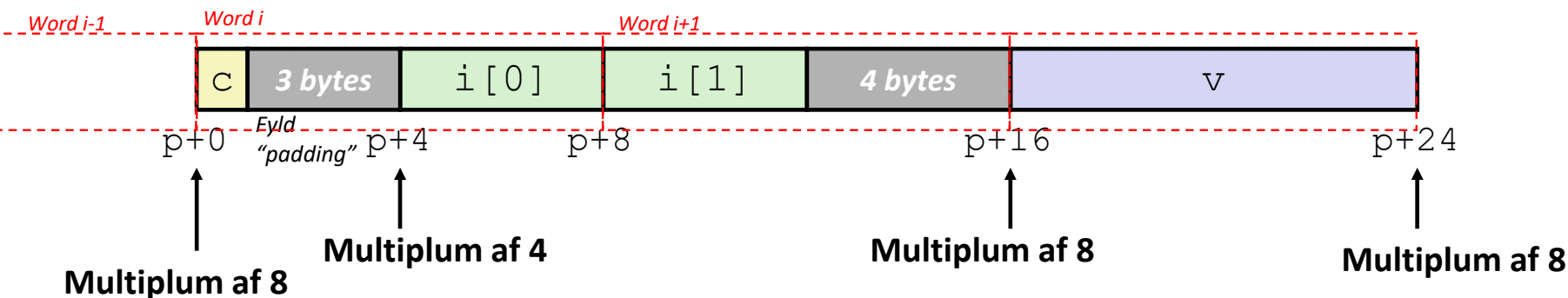
# Strukturer & Justering (Word Alignment)

- Strukturer uden justering (Unaligned)



```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

- "Skæv" adressering
- Justeret (Aligned)
  - En primitive data type, der kræver  $K$  bytes, skal udlægges på adresse, som er et multiplum af  $K$
  - Start adresse & struktur længde skal være multiplum af største  $K$



# Justeringsprincipper (Alignment)

- Justeret Data
  - Kræves på nogle arkitekturer; **tilrådeligt på x86-64**
- Motivation for justering:
  - Data overføres fra hukommelsen i (aligned) bidder på 4 el- 8 bytes (system afhængigt)
    - Ineffektivt at hente / gemme data som deler sig over (quad) word grænser
    - Virtual hukommelse også trickier når et data-element strækker sig over 2 pages
- Compiler
  - Indsætter "fyld" (padding) i strukturen for at sikre korrekt justering af medlemmerne

# Simpelt trick til pladsoptimering

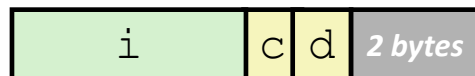
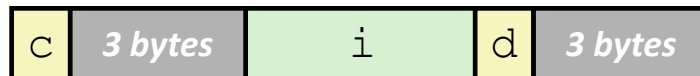
- Programmør oplister de største medlemmer først!

```
struct S4 {  
    char c;  
    int i;  
    char d;  
} *p;
```



```
struct S5 {  
    int i;  
    char c;  
    char d;  
} *p;
```

- Effekt (K=4)



- Effekt ved Arrays

```
struct S4 arr1[1000000]; 12MB  
struct S5 arr2[1000000]; 8MB = 33% mindre
```

Demo alignment.c

# Fín

- Betingelser
- Kontrolstrukturer
  - If-then-else
  - Betinget assignment
  - Do-while
  - While-do
  - For
  - Switch
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
- Data Structures
  - Allokering
  - Adgang
  - Alignment



