/etc/nologin

/etc/login /etc/motd

SHELLS

- Shells are programs that read commands and execute them
- Logging in creates your initial *shell*
- Entering a command forks a process which may be a new *shell*
- Entering *csh* at the *shell* prompt creates a new *shell*
- Executing a C Shell script creates a new *shell*

%	Login *shell*
% csh	Requests another *shell* (2nd-level)
% my.script	Forks a *shell* (3rd-level)
%	Script completes (returns to 2nd-level)
% ^D	Terminates 2nd-level *shell*
%	Back to original login *shell*

if ($user = mueller) then
    (bell false
endif

The C Shell

Put the line:

echo "I am a new shell"

in your *.cshrc* file. When you fork a new *shell*, the line *I am a new shell* displays. Verify that this is true by forking a new *shell* with csh. Terminate the *shell* with CTRL-d. Your entry (after editing your *.cshrc* file):

```
% csh
I am a new shell
%%
```

When you type CTRL-d, it does not display, but a second *shell* prompt is displayed. Remove the line from the *.cshrc* file so that it is not displayed each time you execute a script.

C SHELL SCRIPTS

A C shell script:

- Is created with a text editor
- Contains a # in column 1 of line 1 which indicates that the C shell should be used to execute the commands; if # is not included, the Bourne Shell is assumed

  Use the line `#!/bin/csh -f` (in place of #) for fast startup; *.cshrc* file is not read
- Performs a set of UNIX commands contained in a file
- Must be executable to perform the commands
  - Change the protection mode to be executable:

    % `chmod 755` *scriptname*

    or

    % `chmod a+x` *scriptname*
- Can be executed by typing the filename

  % *scriptname* [*arguments*]
  % `sample.script`

Creating C Shell Scripts

You will be working with a variety of scripts throughout this module. Create the following *data.script* file with your favorite text editor; change the mode to executable.

```
#!/bin/csh -f
echo "This script prints information related to your login shell"
echo "The files in your current directory are:"
ls
echo "The environmental variables that are set are:"
printenv
```

Your entry to make the file executable:

```
% chmod 755 data.script
%
```

Now execute the script. Your entry looks similar to:

```
% data.script
This script prints information related to your login shell
The files in your current directory are:
example.1     example.3     sample.2      test1         test3
example.2     sample.1      test          test2
The environmental variables that are set are:
HOME=/trn/snoopy
SHELL=/bin/csh
PATH=.:/trn/snoopy/bin:/usr/convex:/usr/ucb:/bin:/usr/bin:/usr/local/bin:
TERM=vt100n
USER=snoopy
PAGER=less
EDITOR=/usr/convex/emacs
PRINTER=trip
%
```

C SHELL VARIABLES

- A variable name begins with a letter and consists of letters, digits, and underscores

- To define the contents (value) of a variable use the *set* command followed by an equal sign and the value

```
set vname=value
set sample=1
```

- To show the variables that are set, type: `set` at the *shell* prompt

- Refer to variable values by preceding their names with a dollar sign ($)

```
% set sample=1
% echo $sample
1
```

Using the previous information in a script:

```
#!/bin/csh -f
set sample=1
echo $sample
```

Execute the script:

```
% sample.script
1
```

C Shell Variables

Set the following variables:

```
set sample = 1
set sample2 = hi
set sample3 = date
```

Display a listing of all the variables that are set. Finally display individual variable values using the *echo* command.

Your entry looks similar to:

```
% set sample = 1
% set sample2 = hi
% set sample3 = date
% set
argv      ()
autologout 0
cdpath    /mnt/username
history   20
home      /mnt/username
. . .
sample    1
sample2   hi
sample3   date
% echo $sample $sample2 $sample3
1 hi date
%
```

SETTING STRING VARIABLES

- To assign the contents literally, use single forward quotes:

```
% set a = 'Hi there'
% echo $a
Hi there
```

- Strings not contained in quotes may expand incorrectly:

```
% set b = hi there
% echo $b
hi
```

This actually sets b = hi and there = (null)

- Single or double quotes cause the contents to be considered as one element

- To determine the number of elements, use $#*variable_name*

```
% echo $#a
1
```

- Using double quotes causes all variables to be expanded except filenames

Setting String Variables

Write a script named *var.script* that sets the following variables:

```
sample4 = Hi there
sample5 = "Hi there"
sample6 = 'Hi there'
```

and displays each of their values and number of elements when the script is executed. Your script contents looks similar to:

```
#!/bin/csh -f
set sample4 = Hi there
set sample5 = "Hi there"
set sample6 = 'Hi there'
echo $sample4
echo $sample5
echo $sample6
echo $#sample4
echo $#sample5
echo $#sample6
```

Your executed results look similar to (remember to change the mode to executable):

```
% chmod 755 var.script
% var.script
Hi
Hi there
Hi there
1
1
1
```

Notice in the output that your value for *sample4* is incorrect; only the first item is read and displayed. Thus, you can see that quotation marks are necessary for string values.

SETTING ARRAY VARIABLES

- To assign more than one element, use parentheses around the elements; a space separates the elements:

```
% set b = (This is four elements)
                    or
% set b = ('This' 'is' 'four' 'elements')
% echo $#b
4

% set c = "This is one element"
% echo $#c
1
```

- To specify array elements, use the form:

$*variable_name*[*element_number-element_number*]

The following example illustrates using the value of the first three elements (range) of *b*:

```
% echo $b[1-3]
This is four
```

To use the value of specific elements of the variable *b*:

```
% echo $b[1] $b[4]
This elements
```

Setting Array Variables

Write a script named *test.script* that set variable *elements* to the following:

(How many elements does this contain?)

Your script looks similar to:

```
#!/bin/csh -f
set elements = (How many elements does this contain\?)
echo $#elements
```

Execute the script:

```
% chmod 755 test.script
% test.script
6
%
```

If you can't get your script to execute properly, put a back slash (\) before the question mark.

VARIABLE EXPANSION WITH QUOTATION MARKS

- Backward single quote (‘):

```
%set i = ls
%echo $i
ls
% echo ‘$i‘
file1 file2 file3
% set i = ‘ls‘
% echo $i
file1 file2 file3
```

- Forward single quote (’):

Prevents variable expansion, wildcard expansion, or alias expansion

```
% set a = ls
% echo '$a'
$a
% set i = sample
% echo 'Here is $i'
Here is $i
```

Variable Expansion with Single Quotes

Experiment with using single quotes with a command. Enter:

set d = date

Now echo the value of variable *d* using no quotes, using single backward quotes, and single forward quotes. Note the differences in *d*'s value.

Your entry:

```
% set d = date
% echo $d
date
% echo `$d`
Wed Dec 17 13:19:18 CST 1986
% echo '$d'
$d
%
```

VARIABLE EXPANSION WITH QUOTATION MARKS (cont.)

- Double quotes (”):

Groups characters into a single argument

```
% set i = "Hi there"
% echo $i
Hi there
```

Permits variable expansion to take place

```
% set i = "my sample"
% echo "Here's $i"
Here's my sample
```

No filename wildcard expansion takes place

```
% set i = "ls file*"
% echo "$i"
ls file*
```

Filename wildcard expansion does take place if you do not use the double quotes:

```
% echo $i
ls file1  file2
file3
```

Variable Expansion with Double Quotes

Move to your home directory if you are not there. Write a *quote.script* that prints: **Here is my sample of files ending with script.** Assign **ls *script** to the variable *list*; assign **my sample** to the variable *s*.

Your script looks similar to:

```
#!/bin/csh -f
set s = "my sample"
set list = `ls *script`
echo "Here is $s of files ending with script"
echo $list
```

Verify that your script performs correctly. Your entry:

```
% quote.script
Here is my sample of files ending with script
data.script pre.script test.script var.script ...
%
```

FILENAME EXPANSION

file*	Matches *file* followed by zero or more characters `file*` matches *file*, *file1*, and *file.out*
file?	Matches *file* followed by any one character `file?` matches *file1*, *fileA*, and *filed*
file[123]	Matches any single character contained in the brackets `file[123]` matches only *file1*, *file2*, and *file3*.
file[a-z]	Matches *file* followed by any lowercase letter `file[a-z]` matches *filea*, *filef*, and *filez*

Filename Expansion

Change your directory to *samples*. Write a script named *list.script*:

1. that displays a message that says the files being listed are found in your current directory.
2. that lists all the files ending with *script* in your current directory.

Your script looks similar to:

```
#!/bin/csh -f
echo "The files in your current directory that end with script are:"
ls *script
```

Execute the script:

```
% list.script
The files in your current directory that end with script are:
data.script  test.script
pre.script   var.script
%
```

PREDEFINED VARIABLES

The C shell has the following predefined variables:

$ user	Current user `echo $user`

$ home	Home directory
	`set i = $home`
	`echo $i`

$ shell	Shell you are currently running `echo $shell`
$ path	Your search path `echo $path`
$ term	Your terminal type `if ($term==adm3a) then`
$ status	Exit status of the last command `if ($status==1) then`
$ cwd	Current working directory `echo $cwd`

Predefined Variables

Write and execute a script name *pre.script* that tells you your home directory, your current working directory, and the *shell* you are running under. Your script looks similar to:

```
#!/bin/csh -f
echo $home
echo $cwd
echo $shell
```

Execute the script:

```
% chmod 755 pre.script
% pre.script
/mnt/username
/mnt/username/your_directory_name
/bin/csh
%
```

Make a directory called *samples* and move to that directory. Write a script named *dir.script* that displays the name of your current working directory and your home directory, changes to your home directory and lists all the files in your home directory.

Your script looks similar to:

```
#!/bin/csh -f
echo $cwd
echo $home
cd $home
ls
```

Verify that your script performs correctly:

```
% chmod 755 dir.script
% dir.script
/mnt/username/samples
/mnt/user/
data data.script pre.script test.script var.script
%
```

Change to your home directory.

SPECIAL VARIABLE FORMS

The C shell recognizes the following variables:

$?name — Replaced by 0 if name is not set
Replaced by 1 if name is set

```
set i = (a b c)
echo $?i
1
```

$#name — Replaced by number of elements in variable *name*

```
set i = (a b c)
echo $#i
3
```

$name[1-3] — Replaced by first 3 elements of array *name*

```
set i = (a b c)
echo $i[1-3]
a b c
```

$name[2-] — Replaced by 2nd through last element of *name*

```
set i = (a b c)
echo $i[2-]
b c
```

$name[*] — Replaced by all elements of *name*

```
set i = (a b c)
echo $i[*]
a b c
```

Special Variable Forms

At your terminal set the following variables:

a = my.sample
b = (c d e f g h)

Use the *echo* command to display the following:

1. Number of elements in each variable
2. First 3 elements of variable *b*
3. Last 2 elements of variable *b*
4. All the elements for each variable
5. Determine if *j* is set

Your entry looks similar to:

```
% echo $# a $# b
1 6
% echo $b[1-3]
c d e
% echo $b[5-]
g h
% echo $a[*] $b[*]
my.sample c d e f g h
% echo $?j
0
%
```

SPECIAL VARIABLE FORMS (cont.)

The C shell recognizes the following variables when used in a script:

$0	Replaced by name of script being executed
$1	Replaced by first argument to script
$2	Replaced by second argument to script
$*	Replaced by all arguments to script
$$	Replaced by process number of current shell
$<	Reads 1 line of input from terminal

If *my.script* contains:

```
#!/bin/csh -f
echo $0                          # gives names of script
echo $1 $2                       # gives first and second elements
echo $*                          # gives all elements
echo $$                          # gives the process ID
mv $1 $1.old                     # renames first argument
echo "Moved $1 to $1.old"        # displays first element renamed
echo 'Who are you'               # displays contents at terminal
set a = ($<)                     # reads line of terminal input
echo 'I am done' $a              # displays contents at terminal
```

Executing the script produces:

```
% my.script file1 file2 file3
my.script
file1 file2
file1 file2 file3
13369
Moved file1 to file1.old
Who are you
user1
I am done user1
```


set a = $<

Script Variables

Write a script named *back.script*:

1. that displays a message telling the user what the script does; include the name of the script in the message.
2. that copies a file entered as the first script argument to *file*.**back**.
3. displays a message that tells the user the task is completed.

Your script looks similar to:

```
#!/bin/csh -f
echo "$0 will copy $1 to $1.back"
cp $1 $1.back
echo "$1 is copied to $1.back"
```

Execute the script (did you make it executable?); your entry:

```
% back.script file1
back.script will copy file1 to file1.back
file1 is copied to file1.back
%
```

THE *argv* VARIABLE

- The *argv* variable receives the arguments sent to the script:

    script_name `argv[1] argv[2] argv[3] ...`

argv receives its input from the variables that are included after the script name:

```
% my.script a b c
```

If *my.script* contains:

```
#!/bin/csh -f
echo $argv[1]       # first element ($1)
echo $argv[2]       # second element ($2)
echo $argv[3]       # third element ($3)
echo $argv          # all elements ($*)
echo $#argv         # number of elements
echo 'I am done'
```

Executing the script produces:

```
% my.script a b c
a
b
c
a b c
3
I am done
```

The *argv* Variable

Modify your *back.script* to use the *argv* variable. Your modified script looks similar to:

```
#!/bin/csh -f
echo "$0 will copy $argv[1] to $argv[1].back"
cp $argv[1] $argv[1].back
echo "$argv[1] is copied to $argv[1].back"
```

Execute the script (did you make it executable?); your entry:

```
% back.script file2
back.script will copy file2 to file2.back
file2 is copied to file2.back
%
```

PE 3 + 5

3+5 = 8

MATHEMATICAL OPERATIONS

Using the @command:

- Similar to the *set* command
- Used with mathematical calculations (+, -, /, *, etc.)

Examples:

```
% set i = 10
% set j = 2
% @ k = ($i + $j)
% echo $k
12
% @ z = ($i / $j)
% echo $z
5
```

- Use for incrementing:

```
% set z = 1
% @z = ($z + 1)
% echo $z
2
```

Mathematical Operations

Write *add.script* that does the following:

1. displays the message: "This scripts adds 2 integers".
2. asks the user for an integer.
3. asks user for a second integer.
4. adds the two integers.
5. displays the message: "Total is:" followed by the total.

Your script looks similar to:

```
#!/bin/csh -f
echo "This script adds 2 integers:"
echo "Type your first integer and press RETURN"
set a = ($<)
echo "Type your second integer and press RETURN"
set b = ($<)
@ total = $a + $b
echo "The total is: $total"
```

Verify that the script executes correctly. Your entry:

```
% add.script
I add two numbers
Type your first number and press RETURN
345
Type your second number and press RETURN
23
The total is: 368
%
```

foreach i (1 2 3 4 5)
echo $i
end

FLOW CONTROL - FOREACH STATEMENTS

- foreach statement format:

```
foreach i (*.f)        foreach i (*.f)
  ...                    ...
  break                  continue
end                    end
```

Example script:

```
#!/bin/csh -f
#################################################
# this script does the following:
#  compiles every FORTRAN file in the
#    directory which ends in .f
#  the object files *.o are built
#    but due to the -c option,
#    no executable file is built
#################################################
foreach i (*.f)
  fc -O2 -c $i
end
```

foreach Statements

Enhance your *back.script* to handle more than one file at a time. Include the messages:

back.script is copying *filename* to *filename*.back
***filename* is copied to *filename*.back**

Your script looks similar to:

```
#!/bin/csh -f
foreach i ($*)
echo "$0 is copying $i to $i.back"
cp $i $i.back
echo "$i is copied to $i.back"
end
```

Verify that your script executes correctly. Your entry looks similar to:

```
% back.script file1 file2 file3
back.script is copying file1 to file1.back
file1 is copied to file1.back
back.script is copying file2 to file2.back
file2 is copied to file2.back
back.script is copying file3 to file3.back
file3 is copied to file3.back
%
```

FLOW CONTROL - IF STATEMENTS

Several commands exist that can be used to regulate flow control:

- if statement format:

```
if ($i == test) then
  . . .
else
  . . .
endif
```

Nested if statements are permitted

Example:

```
#!/bin/csh -f
set a = 5
set b = 6
if ( $a > $b) then
    echo $a is greater than $b
else
    echo $a is not greater than $b
endif
```

Execute the script:

```
% sample.script
5 is not greater than 6
```

if Statements

Write a *cmp.script* to perform the following:

1. Asks the user for two files to compare.
2. If two filenames are not entered, display the message: "You didn't give me two filenames! I am quitting."
3. If two filenames are entered, then display the message: "I will make a comparison of *file1 file2*.
4. Compare the files; use the option that returns only the status with the *cmp* command.
5. If the status equals zero, then display the message: "Identical files; I will remove *file1*." Then remove *file1*.
6. If the status returns non-zero, then display the message: "Not identical; I won't remove either file."

Your script looks similar to:

```
#!/bin/csh -f
#
echo "Enter the name of two files to compare"
set ans = $<
set num = ($ans)
if ($#num == 2) then
        echo "I will make a comparison of"
        echo $num[1]  $num[2]
        cmp -s $num[1] $num[2]
          if ($status == 0) then
            echo "identical files; I will remove $num[1]"
            rm $num[1]
          else
            echo "Not identical; I won't remove either file"
          endif
else
echo "You didn't give me two filenames"
echo "I am quitting."
endif
```

Verify that your script performs correctly.
Your entry:

```
% cmp.script
Enter the name of two files to compare"
test1 test.s
I will make a comparison of
test1 test.s
Not identical; I won't remove either file
% cmp.script
Enter the name of two files to compare
test
You didn't give me two file names!
I am quitting.
% cmp.script
Enter the name of two files to compare
sunny ugly
I will make a comparison of
sunny ugly
identical files; I will remove sunny
```

COMPARISON OPERATORS

=	Assignment operator `set a = 1`
==	Equal to `if ($i == 10) then`
!=	Not equal to `if ($1 != 0) then`
&&	Boolean AND
><	Greater than; less than `if ($i > 1 && $i < 10) then`
||	Boolean OR `if ($i == 0 || $i == 1) then`
!	Boolean NOT `if ( !($i <= 10) ) then`
=~	Similar To `if ($i =~ test?) then`
!~	Not Similar To `if ($i !~ test?) then`

EXAMPLE:

```
% set i = (a b c)
% if ($#i == 3) echo There are three
There are three
% if ($#i == 2) echo There are two
%
```

(Nothing is displayed as the expression is false)

Comparison Operators

Write a *assign.script* that

1. assigns 6 to *a*
2. assigns 10 to *b*
3. assigns 15 to *c*
4. uses Boolean AND to compare the value of *c* to both *a* and *b*
5. prints a message "C is greater than A or B" if it is

Your script looks similar to:

```
#!/bin/csh -f
set a = 6
set b = 10
set c = 15
if ($c > $a && $c > $b) echo C is greater than A or B
```

Execute your script:

```
% assign.script
C is greater than A or B
%
```

FLOW CONTROL - WHILE STATEMENTS

- **while statement format:**

```
while ($i != 13)          while ($i != 13)
  ...                       ...
break                     continue
end                       end
```

Example script:

```
#!/bin/csh -f
echo "Guess what number I am thinking of"
echo "Enter the number and press return"
set ans = $<
while ($ans != 5)
   echo "Enter another number"
   set ans = $<
end
echo "You guessed it"
```

while Statements

Write *cal.script* that prints a calendar for a year. If the user does not specify a year, ask for the year. Also, display a message that tells what year calendar is being printed.

Your script looks similar to:

```
#!/bin/csh -f
if ($# argv != 1) then
        echo "enter the desired year:"
        set j = ($<)
else
        set j = $1
endif
echo "The desired year is $j"
set i = 1
while ($i != 13)
        cal $i $j
        @ i = ($i + 1)
end
echo "Calendar is complete"
```

Execute your script:

```
% cal.script
Enter the desired year:
1987
The desired year is 1987
   January 1987
 S  M Tu  W Th  F  S
              1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
   (continues)
Calendar is complete
%
```

FLOW CONTROL - CASE STATEMENTS

- switch statement format:

```
switch ($variable)
case string:
  . . .
  breaksw
  . . .
default:
  . . .
endsw
```

Example script for selecting a printer:

```
#!/bin/csh -f
echo "This does an iprx on the file"
echo "Enter the file name"
set fn = ($<)
echo "Select the number of the printer you want"
echo "1 for swip, 2 for vaxip, or 3 for tacip"
set p = ($<)
switch ($p)
case 1:
  iprx -Pswip $fn
  breaksw
case 2:
  iprx -Pip $fn
  breaksw
case 3:
  iprx -Ptacip $fn
  breaksw
default:
echo "Nothing printed; invalid printer number"
endsw
```

case Statements

Write *case.script* that:

1. displays the message: "This script allows you to copy or move a file".
2. displays the message: "Enter the item number and press RETURN".
3. Asks the user to select 1 to copy a file or select 2 to move the file.
4. Displays the message(s): asking for the filename to copy or move.
5. Displays a message telling the user that file has been copied or moved.
6. Displays a message if the user enters any number other than 1 or 2.

Your script looks similar to:

```
#!/bin/csh -f
echo "This script allows you to copy or move a file"
echo "Enter the item number and press RETURN"
echo "1 for copy a file or 2 to move a file"
set ans = ($<)
switch ($ans)
case 1:
    echo "Enter the name of the file to be copied and press RETURN"
    set fn = ($<)
    echo "Enter the name of the file to copy to and press RETURN"
    set cf = ($<)
    cp $fn $cf
    echo "$fn copied to $cf"
    breaksw
case 2:
    echo "Enter the name of the file to be moved and press RETURN"
    set mf = ($<)
    echo "Enter the new name of file and press RETURN"
    set nf = ($<)
    mv $mf $nf
    echo "$mf is moved to $nf"
    breaksw
default:
  echo "You didn't give me a valid number; aborted"
endsw
```

Test your script:

```
% case.script
This script allows you to copy or move a file
Enter the item number and press RETURN
1 for copy a file or 2 to move a file
1
Enter the name of the file to be copied and press RETURN
file1
Enter the name of the file to copy to and press RETURN
file1.sample
file1 copied to file1.sample
% case.script
This script allows you to copy or move a file
Enter the item number and press RETURN
1 for copy a file or 2 to move a file
5
You didn't give me a valid number; aborted
%
```

FLOW CONTROL - GOTO STATEMENTS

goto statement format:

```
goto label
...
label: . . .
```

Example script for selecting a printer:

```
#!/bin/csh -f
echo "This does an iprx on the file"
echo "Enter the file name"
set fn = ($<)
echo "Select the number of the printer you want"
again:
echo "1 for swip, 2 for vaxip, or 3 for tacip"
set p = ($<)
echo $p
switch ($p)
case 1:
  iprx -Pswip $fn
  breaksw
case 2:
  iprx -Pip $fn
  breaksw
case 3:
  iprx -Ptacip $fn
  breaksw
default:
echo "Nothing printed; invalid printer number"
echo "Please renter your selection"
  goto again
endsw
```

goto Statements

Modify your *add.script* to perform the following:

1. Ask the operator "Do you want to try again (y or n)?"
2. If the answer is *y*, repeat sequence.
3. If the answer is *n*, clear the screen and display "Thank you"

Your script looks similar to:

```
#!/bin/csh -f
echo "I add two numbers"
again:
echo "Type your first number and press RETURN"
set a = ($<)
echo "Type your second number and press RETURN"
set b = ($<)
@ total = $a + $b
echo "The total is:  $total"
echo "Do you want to do this again?"
echo "Enter y or n and press RETURN"
set r = ($<)
   if ($r == y) then
      goto again
   endif
   if ($r == n) then
     clear
     echo "Thank you"
   endif
```

Test your script:

```
% add.script
I add two numbers
Type your first number and press RETURN
5
Type your second number and press RETURN
978
The total is:  983
Do you want to do this again?
Enter y or n and press RETURN
y
Type your first number and press RETURN
78
Type your second number and press RETURN
89
The total is:  167
Do you want to do this again
Enter y or n and press RETURN
n
(screen clears)
Thank you
%
```

FLOW CONTROL – SHIFT WITH A LOOP

- shift causes members of *argv* to be shifted to the left, discarding *argv[1]*

```
#
# This shows how shift works in a loop
set zed = (Test this again)
echo "Before using shift, there are $#zed arguments"
while ($#zed > 0)
   echo "another arg is $zed[1]"
   shift zed
end
echo "After using shift, there are $#zed arguments"
```

Execute the script:

```
% shift.script
Before using shift, there are 3 arguments
another arg is Test
another arg is this
another arg is again
After using shift, there are 0 arguments
```

Shift with a Loop

Write a *shift.script* that performs the following:

1. Uses the *shift* command.
2. *vi* or *emacs* on all files named on the command line.
3. includes a message that explain what the script does.

Your scripts looks similar to:

```
#!/bin/csh -f
echo "This script allows you to edit all of the files you named on the command line"
while ($#argv > 0)
     emacs $1
     shift
end
```

Test your script:

```
% shift.script pre.script data
#
echo $home
echo $cwd
echo $shell
(save the file)
John Foo 456
Sally Jones 234
Fred Smith 123
Edgar Thompson 345
(save the file)
%
```

FILENAME MODIFIERS

To extract a portion of a file pathname, use filename modifiers:

:r Retrieves the root of the filename

:e Retrieves the extension of the filename

:h Retrieves the head of the filename

:t Retrieves the tail of the filename

EXAMPLES:

```
% set p=/mnt/jsmith/sample/my.script
% echo $p:r
/mnt/jsmith/sample/my
% echo $p:e
script
% echo $p:h
/mnt/jsmith/sample
% echo $p:t
my.script
```

To build a new filename:

```
% echo $p:r.newfile
/mnt/jsmith/sample/my.newfile
```

Note: You can only use 1 : operator per command.

Filename Modifiers

Using the filename */mnt/foo/sample.text*, write a script name *ext.script* that:

1. asks the user for the absolute pathname.
2. displays the absolute pathname of the file and a message explaining what is being displayed.
3. displays the root of the filename and a message explaining what is being displayed.
4. displays the head of the filename and a message explaining what is being displayed.
5. displays the tail of the filename and a message explaining what is being displayed.

Your script looks similar to:

```
#!/bin/csh -f
echo "What is the absolute pathname of the file to be examined?"
set ans=($<)
echo "The absolute pathname is $ans"
echo "The root of the filename you gave me is $ans:r"
echo "The extension of the filename you gave me is $ans:e"
echo "The head of the filename you gave me is $ans:h"
echo "The tail of the filename you gave me is $ans:t"
```

Execute your script:

```
% ext.script
What is the absolute pathname of the file to be examined?
/mnt/foo/sample.text
The absolute pathname is /mnt/foo/sample.text
The root of the filename you gave me is /mnt/foo/sample
The extension of the filename you gave me is text
The head of the filename you gave me is /mnt/foo
The tail of the filename you gave me is sample.text
%
```

(Note: These filename modifiers will not work with $0.)

FILE OPERATORS

-d *filename*	True if *filename* is a directory `if (-d /mnt/usr/) then`
-e *filename*	True if *filename* exists `if (-e /mnt/usr/myfile) then`
-f *filename*	True if *filename* is a regular file `if (-f /mnt/usr/myfile) then`
-o *filename*	True if you are the owner of *filename* `if (-o /mnt/usr/myfile) then`
-r *filename*	True if *filename* is readable `if (-r myfile) then`
-w *filename*	True if *filename* is writable `if (! -w myfile) then`
-x *filename*	True if *filename* is executable `if (-x a.out) then`
-z *filename*	True if *filename* is empty `if (! -z` *myfile*`) then`

EXAMPLE:

```
% if (-e view.script) echo yes
yes
% if (-e poem) echo yes
%
```

(Nothing is displayed because the file does not exist)

File Operators

Write a *check.script* that:

1. asks the user for a filename.
2. prints a message when the file exists in the home directory.
3. prints a message if the file is executable.
4. prints a message if the file is writable.

Your script looks similar to:

```
#!/bin/csh -f
echo "What file do you want me to check?"
set a = ($<)
cd $home
if (-e $a) echo "$a exists"
if (-x $a) echo "$a is executable"
if (-r $a) echo "$a is readable"
if (-w $a) echo "$a is writable"
```

Verify that the script works correctly. Your entry:

```
% check.script
What file do you want me to check?
data
data exists
data is readable
data is writable
%
```

Supply the name of a file that does not exist. As you have not used any flow control (if-else statements), the response is the *shell* prompt.

DEBUGGING SCRIPTS

Use any of the following to debug script output:

-x Causes commands to be echoed just prior to execution

-X Causes commands, including *.cshrc* commands, to be echoed just prior to execution

-v Causes the command to be echoed after history substitution

-V Causes commands, including *.cshrc* commands, to be echoed after history substitution

The options can be combined:

```
% csh -x -v scriptname arg1 arg2
```

Debugging C Shell Scripts

Experiment with the debugging options. Determine the difference in using *-x* and *-v*. Use these options with your *back.script*. Your entry looks similar to:

```
% csh -x back.script file1 file2
foreach i ( file1 file2 )
echo back.script is copying file1 to file1.back
back.script is copying file1 to file1.back
cp -i file1 file1.back
echo file1 is copied to file1.back
file1 is copied to file1.back
end
echo back.script is copying file2 to file2.back
back.script is copying file2 to file2.back
cp -i file2 file2.back
echo file2 is copied to file2.back
file2 is copied to file2.back
end
% csh -v back.script file1 file2
foreach i ( $* )
echo "$0 is copying $i to $i.back"
back.script is copying file1 to file1.back
cp $i $i.back
echo "$i is copied to $i.back"
file1 is copied to file1.back
end
echo "$0 is copying $i to $i.back"
back.script is copying file2 to file2.back
cp $i $i.back
echo "$i is copied to $i.back"
file2 is copied to file2.back
end
%
```

BUILT-IN COMMANDS

echo -n — Writes contents of *string* to standard output, suppressing a newline

```
echo -n "Newline suppressed"
```

exit — *shell* exits with the value of the $*status* variable - any number other than 0 indicates an error

```
if ($1 == "") then
  echo "no username specified"
  exit
endif
```

exit *val* — The *shell* exits with the value specified as *val*; allows you to determine where a script failed

```
if ($1 == "") then
  echo "no username specified"
  exit 1
endif
set colon = ":"
grep '^$1$colon /etc/passwd > name_$$
wc -l name_$$ > lcn__SS
set ncnt = `awk' {print $1}' < lcnt_$$`
if ($ncnt == 0) then
  echo "user name is not known"
  rm -r name_$4
  rm -r lcnt_$$
  exit 2
endif
. . .
```

echo -n "gib mir einen Namen: " $<
set name = $<

Built-in Commands

Modify the *case.script*:

1. Give the user the opportunity to copy or move additional file(s) before the script "quits" by adding a message that asks if another file should be copied or moved.

   a. Repeat the sequence if yes
   b. If no, display the message: "Goodbye".
   c. If invalid answer, display the message: "Invalid answer; aborted".

2. Retain the message: "You didn't give me a valid number; aborted"; (be sure the script actually "quits" if 1 or 2 is not entered)
3. Replies to each question should be placed on the same line as the question.

Your script looks similar to:

```
#!/bin/csh -f
echo "This script allows you to copy or move a file"
again:
echo "Enter the item number and press RETURN"
echo -n "1 for copy a file or 2 to move a file: "
set ans = ($<)
switch ($ans)
case 1:
    echo -n "Enter the name of the file to be copied and press RETURN: "
    set fn = ($<)
    echo -n "Enter the name of the file to copy to and press RETURN: "
    set cf = ($<)
      cp $fn $cf
    echo "$fn copied to $cf"
breaksw
case 2:
    echo -n "Enter the name of the file to be moved and press RETURN: "
    set mf = ($<)
    echo -n "Enter the new name of file and press RETURN: "
    set nf = ($<)
      mv $mf $nf
    echo "$mf is moved to $nf"
breaksw
default:
  echo "You didn't give me a valid number; aborted"
  exit
endsw
echo "Do you want to move or copy another file"
echo -n "Enter y or n and press RETURN: "
 set r = ($<)
if ($r == y) goto again
    if ($r == n) then
        echo "Goodbye"
    else
        echo "Invalid answer; aborted"
    endif
```

Verify that your script performs correctly for all three cases, i.e, asks you to copy or move another file, if "yes", repeats sequence; if "no", displays "Goodbye" or aborts (displaying an appropriate message) when an incorrect reply is given.

BUILT-IN COMMANDS (cont.)

onintr - Ignore all interrupts during the execution of the *shell* script; (must use *kill -9* to terminate)

```
#
onintr -
loop:
echo "Try to kill me with ^C"
goto loop
```

onintr Restores the default action by the *shell* for detected interrupts

```
#
onintr -
echo "You can't get me here"
sleep 2
echo "You can't get me here"
onintr
loop:
  sleep 2
  echo "But you can here"
goto loop
```

onintr *label* Go to the section of the script labeled *label* upon detection of an interrupt

```
#
onintr intr
loop:
 echo "Stop me with ^C"
goto loop
intr:
  echo "You did it"
```

Working with Interrupts

Modify the *add.script* to display the message: **interrupt detected; aborted** when an interrupt (CTRL-c) is detected.

Your script looks similar to:

```
#!/bin/csh -f
onintr intr
echo "I add two numbers"
again:
echo "Type your first number and press RETURN"
set a = ($<)
echo "Type your second number and press RETURN"
set b = ($<)
@ total = $a + $b
echo "The total is:  $total"
echo "Do you want to do this again?"
echo "Enter y or n and press RETURN"
set r = ($<)
   if ($r == y) then
      goto again
   endif
   if ($r == n) then
     clear
     echo "Thank you"
     exit
   endif
intr:
   echo "Interrupt detected; aborted"
```

Test your script:

```
% add.script
I add two numbers
Type your first number and press RETURN
5
Type your second number and press RETURN
^C
Interrupt detected; aborted
%
```

BUILT-IN COMMANDS

dirs — Displays the directory stack

```
% dirs
/usr  /etc
```

pushd *dirname* — Keeps a stack of directories; pushes the current directory onto a directory stack and changes the current directory to *dirname*

```
% pushd /usr/bin
/usr/bin  /usr  /etc
```

pushd — Exchanges the top 2 elements of the directory stack

```
% dirs
/usr/bin  /usr  /etc
% pushd
/usr  /usr/bin  /etc
```

pushd +*n* — Rotates the stack *n* items

```
% dirs
/usr  /usr/bin  /etc
% pushd +2
/etc  /usr  /usr/bin
```

popd — Discards the top of the directory stack and changes the current *dir* to the next element of the stack

```
% dirs
/etc  /usr  /usr/bin
% popd
/usr  /usr/bin
```

Directory Stack

Experiment with the directory stack commands. First change directories to */usr*, */etc*, and */usr/bin* using ***pushd***. Display the directory stack. Then exchange the top 2 elements of the directory stack. Finally discard the top of the directory stack.

Your entry looks similar to:

```
% pushd /usr
/usr ~
% pushd /etc
/etc /usr ~
% pushd /usr/bin
/usr/bin /etc  /usr~
% dirs
/usr/bin /etc /usr ~
% pushd
/etc /usr/bin /usr ~
% popd
/usr/bin /usr ~
%
```

BUILT-IN COMMANDS (cont.)

eval — Executes a command built from variables using *eval*

```
% set a = ls
% echo $a
ls
% eval $a
file1  file2  file3
```

time *command* — Prints a summary of the time used for *command*

```
% time ls
file1  file2  file3
0.2u 01s 0:01 33% 1+2k 1+1io 1pf + Ow
```

0.2 seconds of user time
0.1 seconds of system time
0:01 seconds of elapsed real time
33% of the cpu used
1+2k memory + stack used
1+1io 1 each input and output (disk)
1pf+Ow one page faulted in; none written

source -h — Reads commands from a file and appends them to your history list

```
% source -h .cshrc
% history
4 source -h ~/.cshrc
5 set cdpath = ( ~ )
6 set history = 20
7 set notify
8 alias cd 'set old=$cwd; chdir \!*'
```

Using *eval*, *time*, and *source -h*

Assign *date* to the variable *d*. First echo the variable's value; then execute the command using the *eval* command. Your entry looks similar to:

```
% set d = date
% echo $d
date
% eval $d
Tue Dec 23 10:06:16 CST 1986
```

Execute the command assigned to *d* and display a summary of the time used to execute it. Your entry:

```
% time eval $d
Tue Dec 23 10:14:26 CST 1986
0.0u 0.0s 0:00 50% 0+0k 0+2io 1pf+0w
%
```

The following exercise demonstrates how commands can be read from a file and appended to the history list. For this exercise, read in the commands you have set in your *.logout* file; then display the history list. Your entry looks similar to:

```
%  source -h .logout
% history
     1 source -h /tac/username/.logout
     2 clear
     3 set path = /usr/games
     4 fortune
     5 /usr/local/bin/micom
     6 echo See you tomorrow!
     7 history
%
```

Exercises for Chapter 3

1. Write a script that displays: `Hi UNIX user. Here are the date and time.` Then display the output of the *date* command. Test the script.

2. Write a script that sets the variable *first* to the value of your first name; the variable *last* to the value of your last name. Using these variables, display the following line as standard output:

   `Your name is` *first last.* Verify the script.

3. Write a script named *printer* that can be executed by any user to determine the names of the printers available on the current system. Include directions on how to print a file using these printers. Have another user test your script.

4. Write a script that displays: `The date is` *display the actual date from the* **date** *command. The next line should display:* `The time is` *display the actual time as provided by the* **date** *command.* Test the script.

5. Modify the previous script so that it displays as the first line `Have a good day, Mr./Ms.` *username.* The user name should be obtained from the predefined user variable. Have another use verify the accuracy of your script.

6. Write a script that can be executed by anyone on the system. This script will display the message: `Your login name is` *username.* Your home directory is *home directory.* You have *number_of_files* in your home directory. Have some of your colleagues execute the script.

7. Write a script that displays the first argument entered by the user and the total number of arguments entered. Your script will display the message: `The first argument you entered is` *arg* and the total number of arguments you entered is *number_of_args.*

8. Make a script that changes a file to executable mode. Test the script on files in your home directory.

9. Write a script using a *foreach* statement that reads a word list from a file named *wdlist* and prints a line for each word in the word list: `I know a little girl who swallowed a` *word from wdlist.*

   The file *wdlist* contains the following list of words: *bug, frog, nickel, pizza.* Test the script.

10. Modify the previous script to use the arguments on the command line as the word list. Print the same reply for each word as you did in the previous exercise.

11. Write a script that will supply phone numbers for specified persons. First create a data base file containing the names and phone numbers (John Foo 234-9080). Your script will ask the user: `Whose phone number to you want? Enter only the first name:"`

    The script should then print out the first and last name of the person, as well as the phone number. Verify that your script works correctly.

12. Write a script that will determine the day (Monday, Tuesday, etc.) and if it is Friday, display the message `Time cards are due today!` If it is not Friday, display the message: `Time cards are due on Friday.` Test the script.

13. Using a *while* loop, write a script that asks `What do you get when it snows on Easter?`. Accept the answer *snow bunnies*. Test your script.

14. Write a script that copies one filename (argument) to *file.old*. Verify that the script works correctly.

15. Enhance the the previous script. Modify the script so that it checks to make sure that only one argument is given; use the *exit* statement if this isn't the case. If more than one argument is given, display a message that says "aborting, more than one argument given." Verify that the script works correctly.

16. Now enhance the previous script to handle more than one file at a time. If no argument is given, display a message that no argument was given and exit. Have one of your colleagues verify that your script is accurate.

17. Using the same script, check to see if *file.old* exists; if it does, refuse to copy over the existing file (but don't exit until all *file* (arguments) have been processed). Also, make sure the *file* supplied as the argument actually exists before copying. Test the script.

18. Modify the same script again. If *file.old* already exists, ask the user if it should be overwritten. Accept any answer given except "yes" as being "no." Test the script.

19. Now modify the script so that it gets the extension for the new filename from the script's name. That is, if a link is added to the script called "new," then *new_file* should copy *file* to *file.new*. (Note: Setting a variable to $0:t will not work!)

20. Write a script that renames all your fortran (.for) files to file name .f. If the name of the file is *main.for*, the new name will be *main.f*. Assume that you have several files in the directory ~/programsfc that need to be renamed. (You can make test files by using the command *touch main.for sub1.for*, etc.)

21. Write a script that displays a menu of choices for the user: copy a file; move a file; remove a file; print a file. For each menu item, make sure the script takes appropriate action. Ask another user to execute the script.

22. Write a script that:

    a. Clears the screen.
    b. Asks the user for a number.
    c. Accepts the first number and requests another number.
    d. Adds the two numbers.
    e. Display the message: `The total is:` followed by the total.
    f. Exit the program with a message to the user: `That is all for now; bye.`
    g. Verify the correctness of your script.

23. Write a script that:

    a. Clears the screen.
    b. Asks the user for two numbers.
    c. Adds the two numbers.
    d. Displays the message: `The total is:` followed by the total.
    e. Displays the message: `Do you want to try again?`
    f. If the answer is *yes*, repeat the sequence. If the answer is *no*, clear the screen and exit the program with message to the user: `Thank you!`
    g. Test the script.

24. Write a script that:

    a. Generates a menu that allows the user to select a mathematical operation. Use the code: A = add; S = subtract; M = multiply; D = divide.
    b. Ask the user to make a selection from the menu.
    c. Ask the user for 2 numbers.
    d. Verify that the user entered just two numbers. If two numbers were not given, ask the for the numbers again.
    e. Perform the selected option.
    f. Display the result of the operation.
    g. Ask the user if he/she wants to do another calculation. If *yes*, repeat the preceding sequence. If *no*, exit with a message to the user: `Calculations complete. Exiting.`
    h. Verify that the script works.

4

The *fc* Compiler

Objectives for Section 4

Upon completion of Section 4 you will be able to:

1. Understand the Compilation Process
2. Call the *fc* FORTRAN Compiler
3. Use the Compiler Options
4. Create Listings and Cross-Reference Tables
5. Execute and Redirect I/O
6. Use *fpp* Preprocessor Statements
7. Use Compiler Directives
8. Understand CONVEX Extensions to the ANSI Standard

fc FEATURES

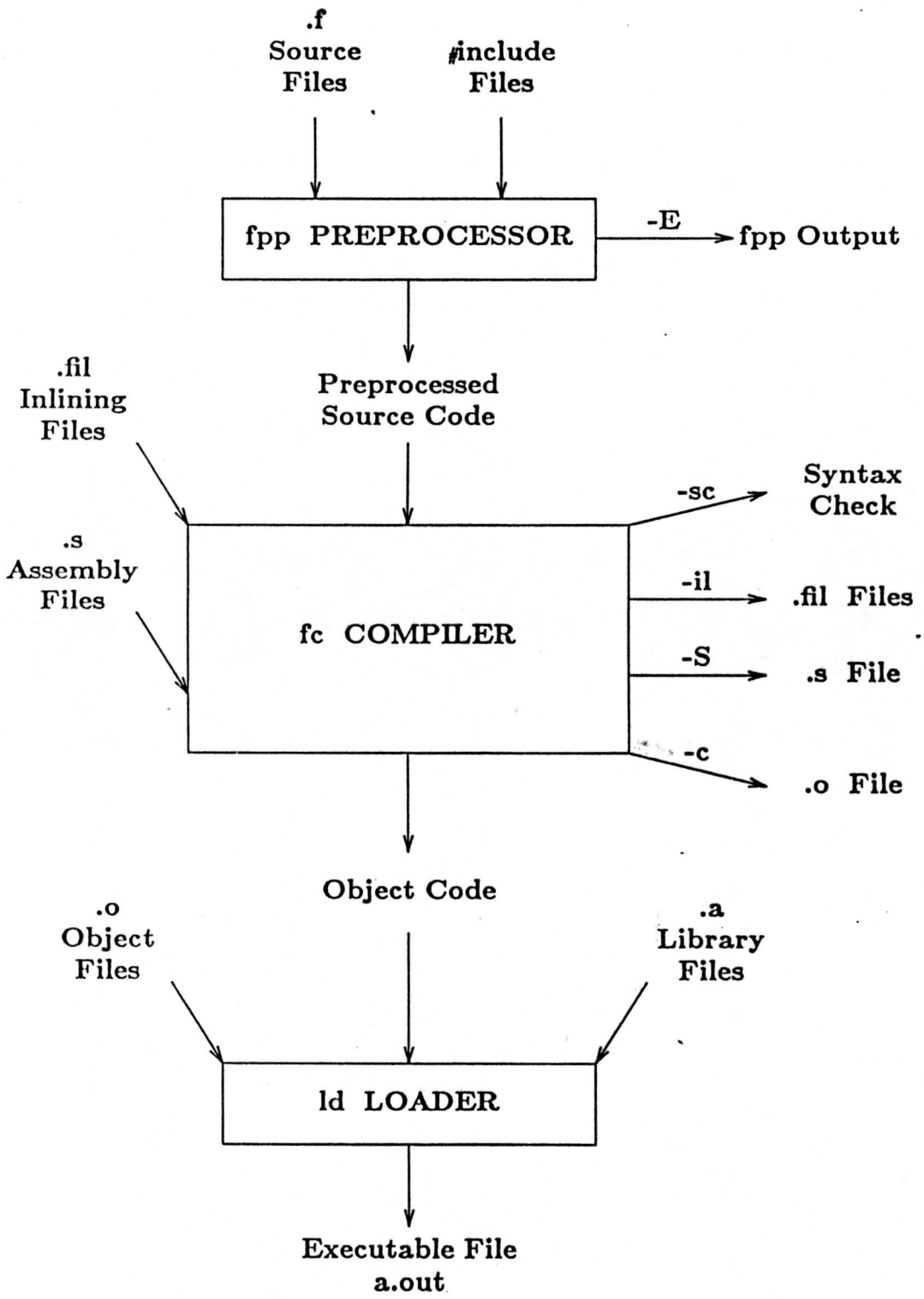
- Conforms to the ANSI FORTRAN 77 Standard
- Provides close compatibility with VMS FORTRAN
- Supports Cray compatibility
- Supports the IEEE floating-point data format
- Supports FORTRAN 66 compatibility
- Supports Automatic Inline Substitution
- Interfaces with the *prof*, *gprof*, and *bprof* performance analyzers
- Produces highly optimized code:
  - Machine-Dependent Optimizations
  - Local Scalar Optimizations
  - Global Scalar Optimizations
  - Vectorization
  - Parallelization

FILES PROCESSED BY *fc*

file.f	**FORTRAN Source Code**
file.o	**Unlinked Object Module**
file.s	**Assembly Language Source Code**
FILE.fil	**Intermediate Code for Inlining**
file.a	**Library of Object Modules**

- ***fc* will also process files specified in *#include* statements**

THE COMPILATION PROCESS



fc DEFAULTS

- ANSI 77 FORTRAN is assumed
- The specified files are run through:
  - o *fpp*
  - o *fc*
  - o *ld* creating *a.out*
- Optimization Level ~~-O0~~ -o0 is used (local scalar optimization)
- All types of messages (errors, warnings, advisories, and summaries) are generated and directed to *stderr*
- Declarations with unspecified lengths default to:

INTEGER	==>	INTEGER*4
LOGICAL	==>	LOGICAL*4
REAL	==>	REAL*4

- Statements with a 'D' in the first column are treated as comments

INVOKING THE *fc* FORTRAN COMPILER

Calling Syntax:

% `fc` [*options*] *files* [*loader options*]

- *options* consist of compiler, preprocessor, and assembler options
- Beware, certain *options* can be overridden by an *OPTIONS* statement within a program unit
- *files* consist of the list of files to be processed
- All source files must include the *.f* extension
- *loader options* are used to tell the compiler what nonstandard libraries with which to link

fc DEFAULTS

- No assembly language file (*.s* file) is saved
- Object files (*.o* files) are saved only if multiple source files are compiled and linked
- No information is produced for the *csd* debugger
- The execution profiling capability is not enabled
- No source code listing is produced
- Array subscript bounds checking is not enabled
- Intermediate language files (*.fil* files) are not generated
- A stack backtrace will be generated if the executable aborts at runtime

COMPILER OPTIONS

Controlling Messages and Listings:

-E	Runs only *fpp* and sends the preprocessed code to *stdout*
-S	Generates assembly code (.s files) for each program unit. Does not assemble the code or produce an executable file
-na	Suppresses all advisory diagnostic messages
-nv	Suppresses all vectorization summary messages
-nw	Suppresses all warning diagnostic messages
-vn	Identifies the compiler version being used
-xr	Generates cross-reference tables and sends them to *stdout*
-iw *n*	(Implies *-xr*) *n* specifies a column width for identifiers (8..32). (Default column width is 16)
-sl	(Implies *-xr*) Produces a source listing with line numbers
-pw *n*	(Implies *-xr*) *n* specifies the logical page width used by the output formatter (Default page width is 132)
-xr1	Generates one cross-reference table as opposed to separate tables for each object class

COMPILER OPTIONS

Invoking Debugging and Profiling Tools:

-cs	Checks constant subscripts at compile time and generates code that checks at runtime whether subscripts used in array references are within the array bounds
-dc	Treats lines of code with a 'D' in the first column as true source lines instead of comments
-db	Produces the information needed by the *csd* source code debugger
-sc	Provides a syntax check without taking the time to execute a full compilation
-p	Enables the execution profiling capability that counts the number of times each routine is called
-pb	Enables the execution profiling capability that counts the number of times each source line is executed
-pg	Enables the execution profiling capability that is similar to *-p* but keeps more extensive statistics
-tl *n*	*n* specifies the maximum number of CPU minutes that will be allowed for the compilation

COMPILER OPTIONS

Controlling the Optimization Level:

-O*n*	Performs optimizations at level *n*: 0 local scalar optimization 1 level 0 plus global scalar optimization 2 level 1 plus vectorization 3 level 2 plus parallelization
-no	Performs no machine independent optimizations
-uo	Performs (potentially) unsafe optimizations
-il	Generates intermediate language (*.fil*) files in preparation for inline substitution
-is *dir*	Performs inline substitution using the *.fil* files located in *dir*
-ss *n*	Reduces the optimization window to approximately *n* executable statements

COMPILER OPTIONS

Increasing Compatibility:

-B[*dir*]	Uses a substitute compiler that resides in *dir*. With no *dir*, *fc* looks in */usr/convex/oldfc* for a substitute compiler
-F66	Uses FORTRAN 66 interpretation rules
-cft~~c~~	Uses the Cray language definition
-a1	Tells the compiler to treat non-character arrays declared in type or dimension statements that have a last dimension of 1 as if they were declared to be assumed-size arrays
-i*n*	Tells the compiler to interpret INTEGER and LOGICAL variables with unspecified lengths to be *n* bytes long *n* can be 2, 4, or 8 (Default is 4)
-r*n*	Tells the compiler to interpret REAL variables with an unspecified length to be *n* bytes long *n* can be 4 or 8 (Default is 4)
-sa	Forces the compiler to put all arguments on the stack as opposed to producing argument packets in the text segment. This is useful when calling C routines from FORTRAN
-72	Truncates source lines at column 72
-fi	Uses IEEE floating-point format
-fn	Uses native CONVEX floating-point format
-fx	Allows programming in dual mode

COMPILER OPTIONS

Controlling *fpp* and Object Files:

-D*name*[=*def*]	Defines *name* to *fpp* If no *def* is given, *name* is defined as 1
-E	Runs only *fpp* and sends the preprocessed code to *stdout*
-I*dir*	Tells *fpp* where to search for *#include* files
-U*name*	Removes any initial definition of *name*
-c	Creates object code (.o files) does not invoke the loader
-o *name*	Creates executable file *name* instead of *a.out*

COMPILER OPTIONS

Obsolete Option	New Option
-F	-uo
-V	-vn
-a	-na
-d	-dc
-g	-db -no
-kb	-cs
-q	-72
-v	-nv
-w	-nw

USING THE *OPTIONS* STATEMENT

- The *OPTIONS* statement must be the first statement in a program unit
- The compiler options specified in the *OPTIONS* statement will override those supplied on the *fc* command line
- Syntax:

OPTIONS *options*

options can be any combination of:

-F66
-O*n*
-cs
-db
-i*n*
-na
-nv
-nw
-r*n*
-cft

FCOPTIONS ENVIRONMENT VARIABLE

- **Set the *FCOPTIONS* variable on the UNIX command line or in your *.cshrc* file:**

```
setenv FCOPTIONS 'options'
```

- **Compiler options chosen with *FCOPTIONS* will be included whenever *fc* is invoked**

- ***FCOPTIONS* will be overridden by *fc* command line flags and *OPTIONS* statements within program units**

- **Precedence for selection of options:**

  1.) *OPTIONS* Statement

  2.) *fc* Command Line

  3.) *FCOPTIONS* Environment Variable

COMPILER MESSAGES

- Messages generated during compilation:

fpp	error messages
fpp	warning messages
fc	error messages
fc	warning messages
fc	advisory messages
fc	vectorization summary messages
fc	internal compiler error messages

- *fpp* messages have the form:

`fpp:` *filename*`:` *line_number*`:` *message*

- *fc* messages have the form:

`fc:` *type* `on line` *line_number.pos* `of` *filename*`:` *message*

type	error, warning, or advisory
pos	the character position after compression by *fpp*

COMPILER MESSAGES

- Vectorization Summaries are generated for each program unit:

  - o Line number of loop start
  - o Iteration variable
  - o FORTRAN statement label for loop
  - o Start, stop, and step values for the iteration variable
  - o Vectorization message

- Internal compiler errors are clearly distinguished:

  `Compiler Error:` *message*

SOURCE CODE LISTINGS

- *-sl* option will produce a cross-reference table and a source code listing
- The *-sl* option implies the *-xr* option (*fc* calls *fxref* to get both listings and tables)
- Use `cat -n` *filename.f* `> listing` to produce a listing with line numbers without having to invoke *fxref*
- Use `fc` *filename.f* `>>& listing` to append messages to the bottom of the listing
- The *error* utility inserts compiler messages as comments in your source file
- Run *fc* on a copy of your source file and pipe *stderr* through the *error* utility to get a listing without disturbing the original source

CROSS-REFERENCE TABLES

The *fxref* cross-reference generator is invoked by including any of the following options on the *fc* command line:

-xr	Calls *fxref* with all defaults
-xr1	Produces a single cross-reference table instead of one table per object class
-sl	Produces a source code listing
-iw *n*	Controls column width for identifiers (8..32) Default is 16
-pw *n*	Controls page width Default is 132

- *fxref* can also be called independently:

  `fxref [`*options*`]` *files*

- Executing *fxref* on a syntactically incorrect FORTRAN program will give you unpredictable results.

CROSS-REFERENCE TABLE FORMAT

Column	Contents
1..16	Identifier Name
18..32	Program Unit in which the Identifier is located
35..42	Data Type of Identifier: R - REAL I - INTEGER L - LOGICAL C - CHARACTER Z - COMPLEX
44..46	Object Class of Identifier: ARY - array BLK - block data COM - common block DO - DO loop head ENT - entry point EXT - external FUN - function ITR - intrinsic LAB - statment label NML - namelist PAR - parameter PRG - program STF - statement function SUB - subroutine VAR - variable
48+	Line Numbers and Usage Types: *blank* - referenced d - defined i - initialized a - assigned p - passed as an argument

EXECUTING *a.out* AND REDIRECTING I/O

- ***stdin*, *stdout*, and *stderr* can be addressed from within a FORTRAN Program:**

stderr	**FORTRAN UNIT 0**
stdin	**FORTRAN UNIT 5**
stdout	**FORTRAN UNIT 6**

- `a.out` **with no redirection will send *stderr* and *stdout* to the screen and read *stdin* from the keyboard**

- `a.out > outfile` **will redirect FORTRAN UNIT 6 to *outfile***

- `a.out < infile` **will read FORTRAN UNIT 5 from *infile***

- `a.out >& errfile` **will redirect both FORTRAN UNIT 6 and FORTRAN UNIT 0 to *errfile***

SEPARATE COMPILATION

- Routines with external references can be compiled separately using the *-c* or the *-S* option on the *fc* command line
- Individual source files can be compiled with different options and then linked at a later time
- The *-c* option causes the compiler to generate unlinked object files with the *.o* extension
- The *-S* option causes the compiler to generate assembly code with the *.s* extension
- Invoking *fc* with *.o* and/or *.s* files will assemble the code if necessary, link the object code, and produce the executable file *a.out*

INTERFACE WITH THE ASSEMBLER

- The `-S` option on the *fc* command line:
  - o Saves the assembly code produced by the compiler in *filename.s*
  - o Prevents the compiler from generating object code
- Assembly code comments indicate:
  - o Correspondence between FORTRAN source lines and assembly code lines
  - o FORTRAN variables accessed in each assembly code line
  - o Decimal values of constants
- Invoking the *as* assembler:

  as [*-o outfile*] [*-l*] *file*

  - o The *-l* option produces an assembly listing with line numbers and the location counter value for each line

INTERFACE WITH THE LOADER

- *fc* automatically invokes the loader with the necessary libraries and object modules in the correct order
- The `-c` option on the *fc* command line:
  - o Generates object modules (*.o* files)
  - o Prevents the compiler from invoking *ld*
- Manually invoking *ld* with precompiled FORTRAN object modules:

```
ld /lib/crt0.o files.o -lU77 -lF77 -lI77 -lm -lc
```

- For more information on *ld* refer to:
  - o *man* page for *ld*
  - o *CONVEX Loader User's Guide*

GENERATED NAMES

UNIX-FORTRAN Interface	==>	_MAIN__
PROGRAM PROG1	==>	_prog1_
SUBROUTINE SUB1(X)	==>	_sub1_
FUNCTION FUNC1(X)	==>	_func1_
ENTRY ENT1(X)	==>	_ent1_
COMMON // X	==>	__blnk_
COMMON /NAMED/ X	==>	__named_
FORTRAN Library Routine	==>	_for$*nnnn*
FORTRAN Math Routine	==>	_mth$*nnnn*

- Generated names appear in:
  - o Assembly Language Listings (*.s* files)
  - o Execution Profiles
  - o Stack Backtraces

THE *fpp* PREPROCESSOR

- The *fpp* preprocessor converts FORTRAN source code for use by the *fc* compiler:

  - o Eliminates tabs, blanks, and comments
  - o Converts characters to upper case except string constants and Holleriths
  - o Handles continuation lines
  - o Handles `D` in column one
  - o Processes compiler directives
  - o Processes *fpp* statements

- The *-E* option on the *fc* command line sends *fpp* output to *stdout* and stops the compilation process after *fpp* is run

- *fpp* statements facilitate:

  - o Macro Substitution
  - o Inclusion of Files
  - o Conditional Compilation

fpp STATEMENTS

- **Preprocessor statements must be in lower case and begin with #**

```
#include file

#define identifier string

#define identifier(identifier,...) string

#undef identifier

#if expression

#ifdef identifier

#ifndef identifier

#else

#endif
```

- **Use a \ to continue *fpp* statements on the next line**

! Dir force_vector

COMPILER DIRECTIVES

- **Format of a Compiler Directive:**

  C$DIR *directive* [, *directive*]

- *directive* **can be one of the following:**

  - o NO_SIDE_EFFECTS [(*func* [, *func*])]
  - o NO_RECURRENCE
  - o SCALAR
    UNROLL

- *SCALAR* **and** *NO_RECURRENCE*:

  - o Must immediately precede the loop
  - o Affect only that loop

- **Use the** *NO_SIDE_EFFECTS* **directive on a function only if that function:**

  - o Does not perform reads or writes
  - o Does not modify a common block
  - o Does not call another routine
  - o Does not modify its parameters

- **The** *NO_SIDE_EFFECTS* **directive allows the compiler to generate more efficient calling code or remove the call entirely**

CONVEX I/O EXTENSIONS

ACCEPT	Reads sequential files from *stdin*
CLOSE	Disconnects a file from a unit
DECODE	Performs internal READ
ENCODE	Performs internal WRITE
FIND	Positions a direct-access file to a particular record
INQUIRE	Returns properties of a file connection
NAMELIST	Provides a format for performing IO on a group of items with a single symbolic name
OPEN	Connects a file with a logical unit
TYPE	Writes sequential files to *stdout*

- Several OPEN statement KEYWORD extensions are also provided

OTHER CONVEX EXTENSIONS

- The computed GOTO statement allows the selector to be a non-integer value

```
REAL X
X = 2.7
GOTO(10, 20, 30) X
```

  - o Transfers control to the statement at label 20

- A DO WHILE structure is provided
- An END DO statement can be used to terminate DO and DO WHILE loops
- The IMPLICIT NONE statement overrides all implicit defaults
- Type declarations can include an initialization portion:

```
INTEGER IVAL/500/
```

CONVEX PARAMETER STATEMENT

- Standard form:

```
PARAMETER  (p = c [, p = c ...])
```

- Extensions:

  - o Previously defined parameter constants can be used in the definition of COMPLEX parameter constants

  - o Certain intrinsics can be used in the parameter definition if the intrinsic is passed a constant argument:

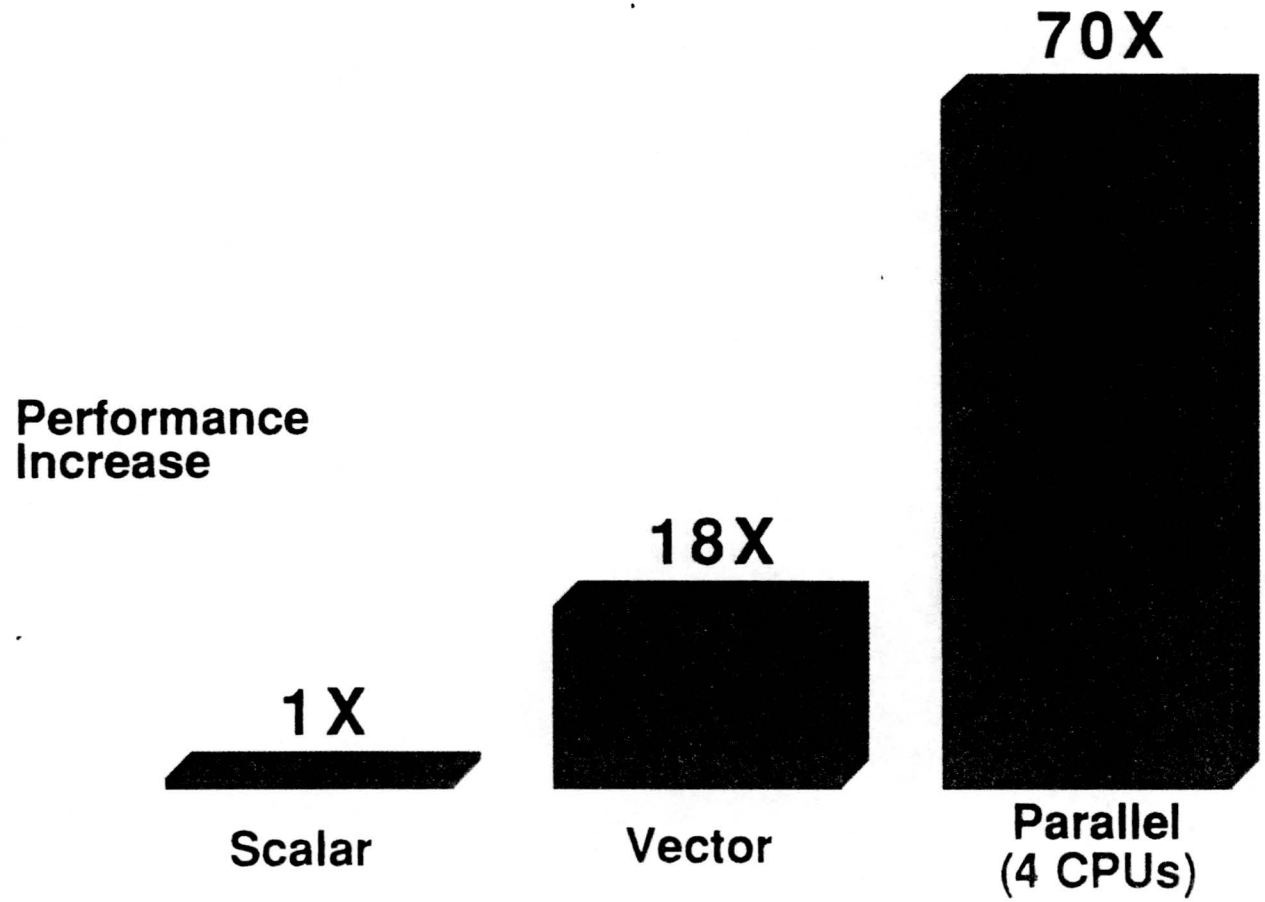
IAND	IOR	NOT	IEOR	DIM
ISHFT	LGE	LGT	LLE	DPROD
LLT	CHAR	MIN	MAX	CMPLX
ABS	MOD	ICHAR	NINT	CONJG

- Alternate form:

```
PARAMETER  p = c,  p = c [, p = c ...]
```

  - o At least two constants must be defined; a single constant definition will be mistakenly parsed as an assignment statement

Performance Through Vectorization and Parallelization



C3200 FORTRAN 400 x 400 Matrix Multiply

BLAS and EISPACK Performance

Symmetric (Hermitian) Matrices

Critical look at BLAS 2 +3 : FORTRAN / VECLIB

Use of those routines within EISPACK

What can we do better ?

Who will benefit from improved routines ?

BLAS and EISPACK Performance

Symmetric (Hermitian) Matrices

CONVEX

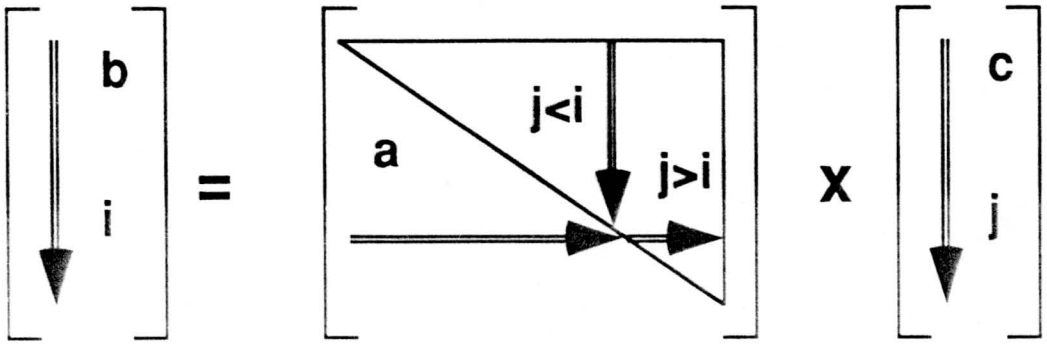
Matrix x Vector

```
parameter(lda=1001,n=1000)
dimension a(lda,n),b(n),c(n)

do i=1,n
  b(i)=0.d0
    do j=1,n
      b(i)=b(i)+a(i,j)*c(j)
    enddo
enddo
```

Normal Case

50 Mflops




```
do i=1,n
  b(i)=0.d0
    do j=1,i
      b(i)=b(i)+a(j,i)*c(j)
    enddo
enddo
```

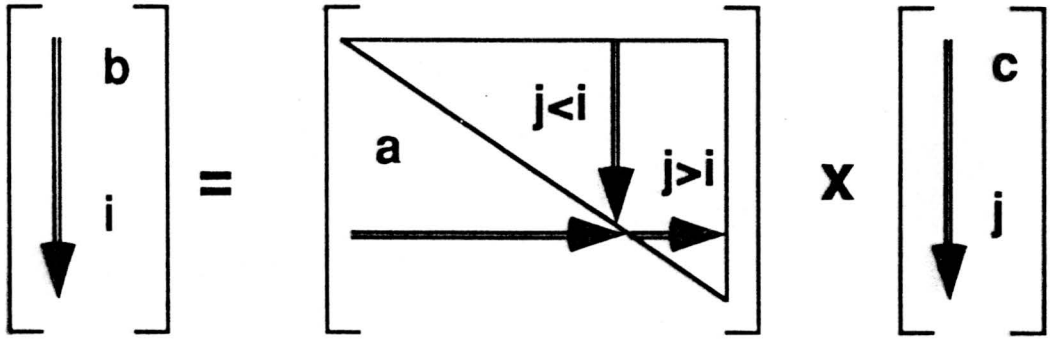
Symmetric Case

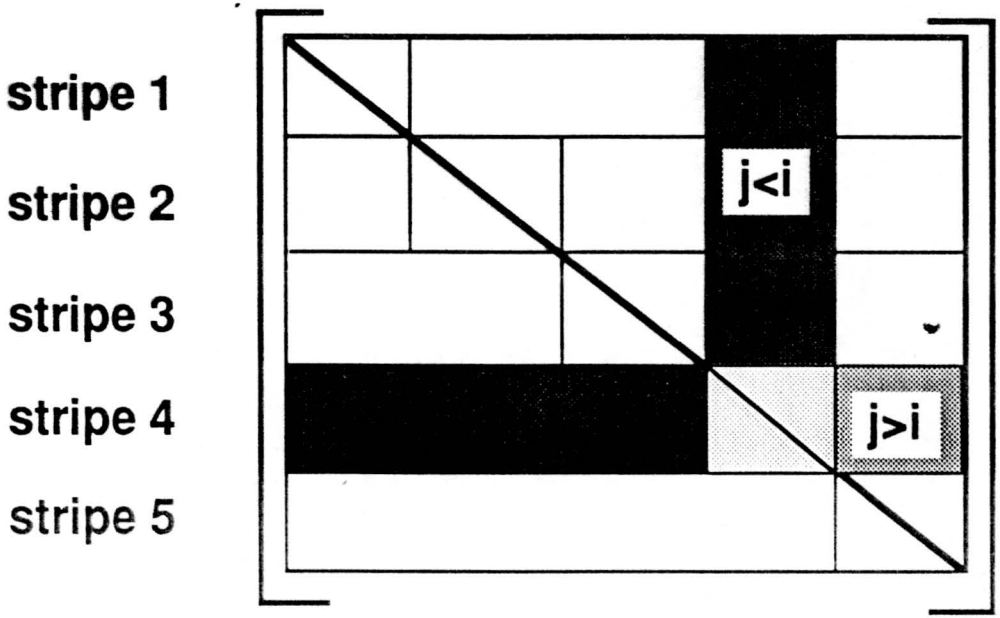
25 Mflops

```
do i=1,n
  do j=i+1,n
    b(i)=b(i)+a(i,j)*c(j)
  enddo
enddo
```

BLAS and EISPACK Performance

Symmetric (Hermitian) Matrices





Performance evaluation

1) Full Matrix reference
P(n) = P0 = 50 Mflops

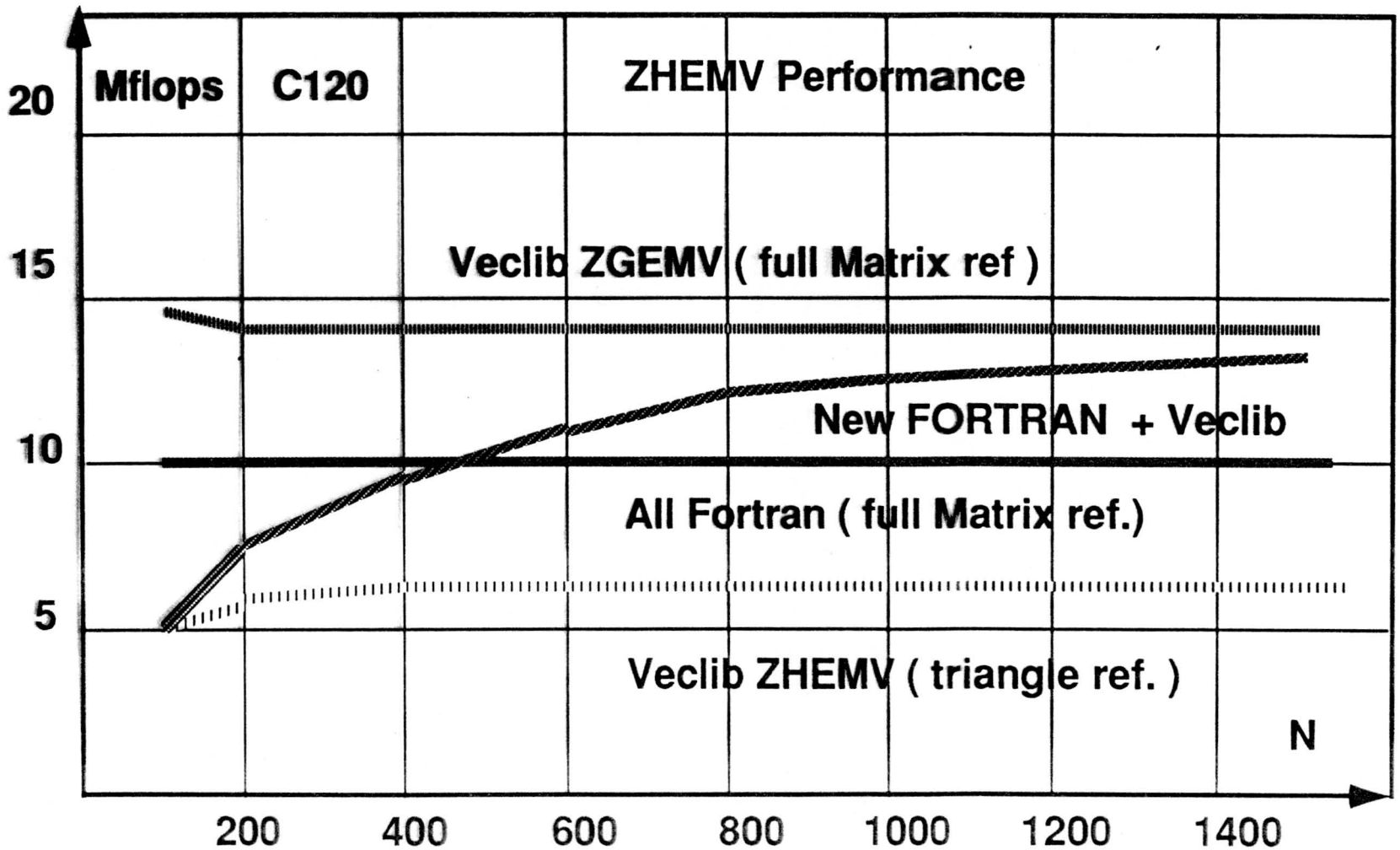
2) Normal Triangle ref.
P(n)=P0 / 2 = 25 Mflops

3) Modified Triangle ref.
P(N)=P0 (1-128 / 2N)

BLAS and EISPACK Performance

Symmetric (Hermitian) Matrices

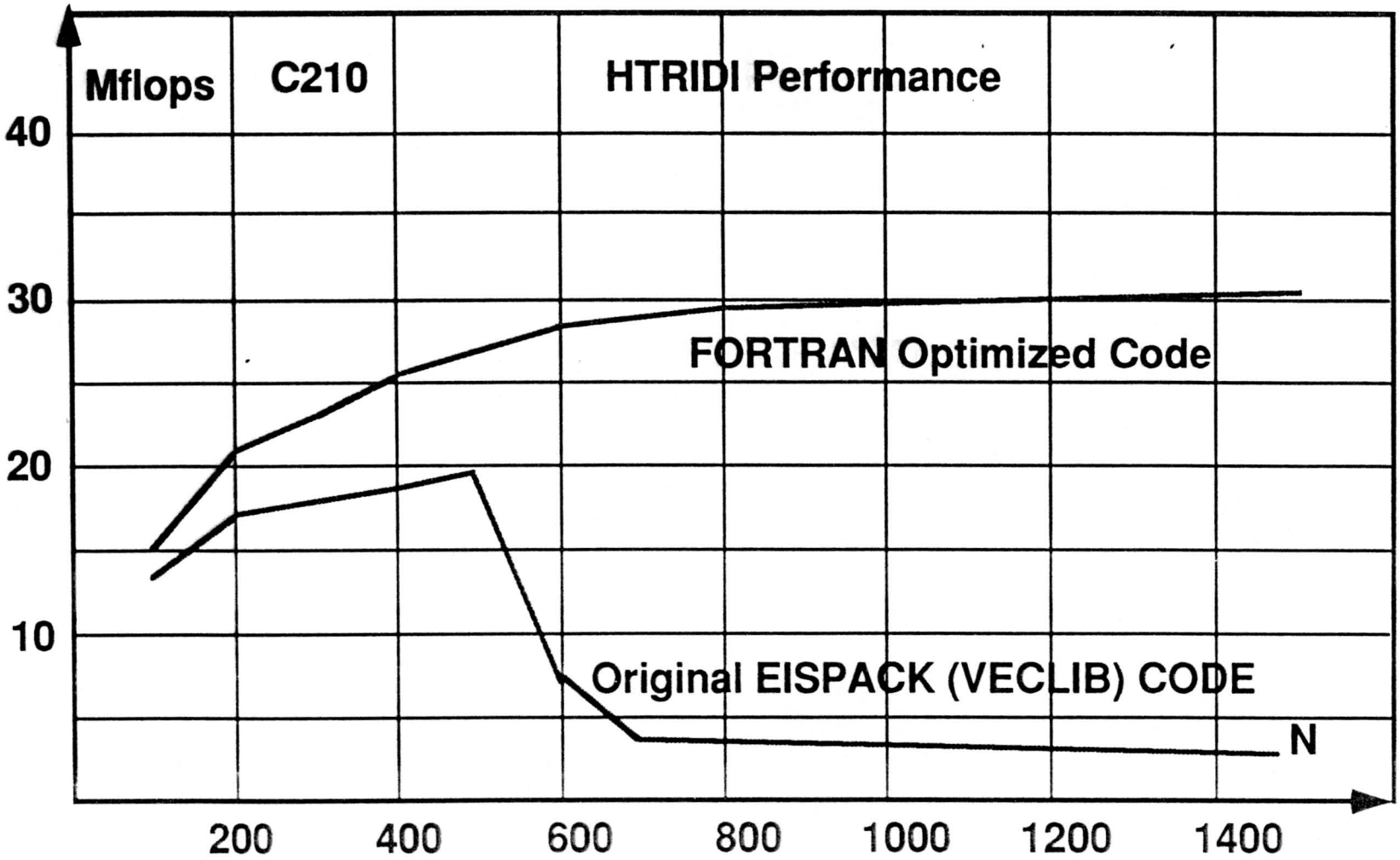
CONVEX



BLAS and EISPACK Performance

Symmetric (Hermitian) Matrices

CONVEX



BLAS and EISPACK Performance

Symmetric (Hermitian) Matrices

Solving a COMPLEX HERMITIAN Eigenvalue Problem

1) Use similarity Transformation to reduce A to real symmetric tridiagonal form

$$A x = s x$$

$$P^{-1} A P = T$$

2) Solve the eigenvalue problem for T

$$T y = s y$$

3) Backtransform the eigenvectors

$$x = P y$$

Step 1 and 3 are the most time consuming parts

computational effort : $\sim N^3$

BLAS and EISPACK Performance

Symmetric (Hermitian) Matrices

Basic steps within HTRIDI

```
do i=n,2,-1
  l= i-1
  u(j)=a(j,i)  j=1,l
  sigma= < u | u >
  h=sigma+dsqrt(sigma)*dabs(a(l,i))
  p = A * u / h
  k= < u | p > /2h
  q = p - k * u
  A = A - u * q+ - q * u+
  d(i)=A(i,i)
  e(i) = -dsqrt(sigma)
enddo
```

BLAS and EISPACK Performance

Symmetric (Hermitian) Matrices

Who will benefit from improved routines for complex hermitian matrices ?

- MPI - Stuttgart
- Fritz-Haber Institut Berlin
- TU-Berlin
- FU - Berlin
- BASF
- Bayer Leverkusen

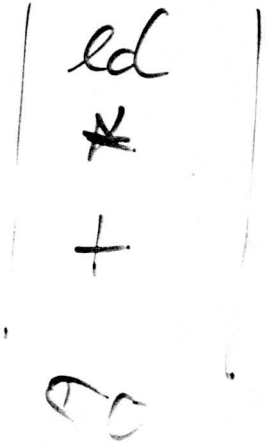
All other People working in the field of comp. chemistry and comp. solid state physics

a(3,3)

a(1,1)
a(2,1)
a(3,1)
a(1,2)
a(2,2)
a(3,2)
a(1,3)
a(2,3).
a(3,3)

0x ff

0x ff+1



Optimization by Loop for Routine ROTMAT

Line Num.	Iter. Var.	Reordering Transformation	Optimizing / Special Transformation	Exec. Mode
5	I	FULL VECTOR Inter		
6	J	None		
11	K	FULL VECTOR Inter		
12	J	None		
18	K	FULL VECTOR Inter		
19	IROT	None		
21	JROT	None	Unroll	

Line Num.	Iter. Var.	Analysis
5	I	Interchanged to innermost
11	K	Interchanged to innermost
18	K	Interchanged to innermost
21	JROT	Replicated: completely unrolled

Optimization by Loop for Routine ROTMAT

Line Num.	Iter. Var.	Reordering Transformation	Optimizing / Special Transformation	Exec. Mode
5	I	FULL VECTOR Inter		
6	J	None		
11	K	FULL VECTOR Inter		
12	J	None		
18	K	None		
19	IROT	None		
21	JROT	Scalar		

Line Num.	Iter. Var.	Analysis
5	I	Interchanged to innermost
11	K	Interchanged to innermost
18	K	Unable to distribute
19	IROT	Unable to distribute
21	JROT	No vectorization- SCALAR directive

Optimization by Loop for Routine ROTMAT

Line Num.	Iter. Var.	Reordering Transformation	Optimizing / Special Transformation	Exec. Mode
5	I	FULL VECTOR Inter		
6	J	None		
11	K	FULL VECTOR Inter		
12	J	None		
18	K	None		
19	IROT	None		
21	JROT	FULL VECTOR	Reduction	

```
fc -O0 rotmat.f -lveclib
cputime =     8.83972

fc -O1 rotmat.f -lveclib
cputime =     5.13938

fc -O2 rotmat.f -lveclib
cputime =     6.14164

fc -O2 rotmat.f -lveclib    : scalar directive
cputime =     4.95827

fc -O2 rotmat.f -lveclib    : unroll directive
cputime =     0.33650
```

```
      program rotmat
      parameter(n=500000)
      dimension g(n,3),rot(3,3),f(n,3),z(3)
c
      do i=1,3
      do j=1,3
      rot(i,j)=i+j-1
      enddo
      enddo
c
      do k=1,n
      do j=1,3
      g(k,j)=k+j
      enddo
      enddo
c
      do k=1,n
            do irot=1,3
c$dir unroll
                  do jrot=1,3
                  f(k,irot)=f(k,irot)+rot(irot,jrot)*g(k,jrot)
                  enddo
            f(k,irot)=f(k,irot)-z(irot)
            enddo
      enddo
      time=cputime(0.)-time
c
      write(6,100) time
  100 format(1h ,' cputime = ',f10.5)
c
      stop
      end
```

FORTRAN TRAINING

HAHN-MEITNER INSTITUT

RAINER GILLERT

CONVEX BERLIN

Worum geht's ?

1. Einführung

2. Hardware Überblick

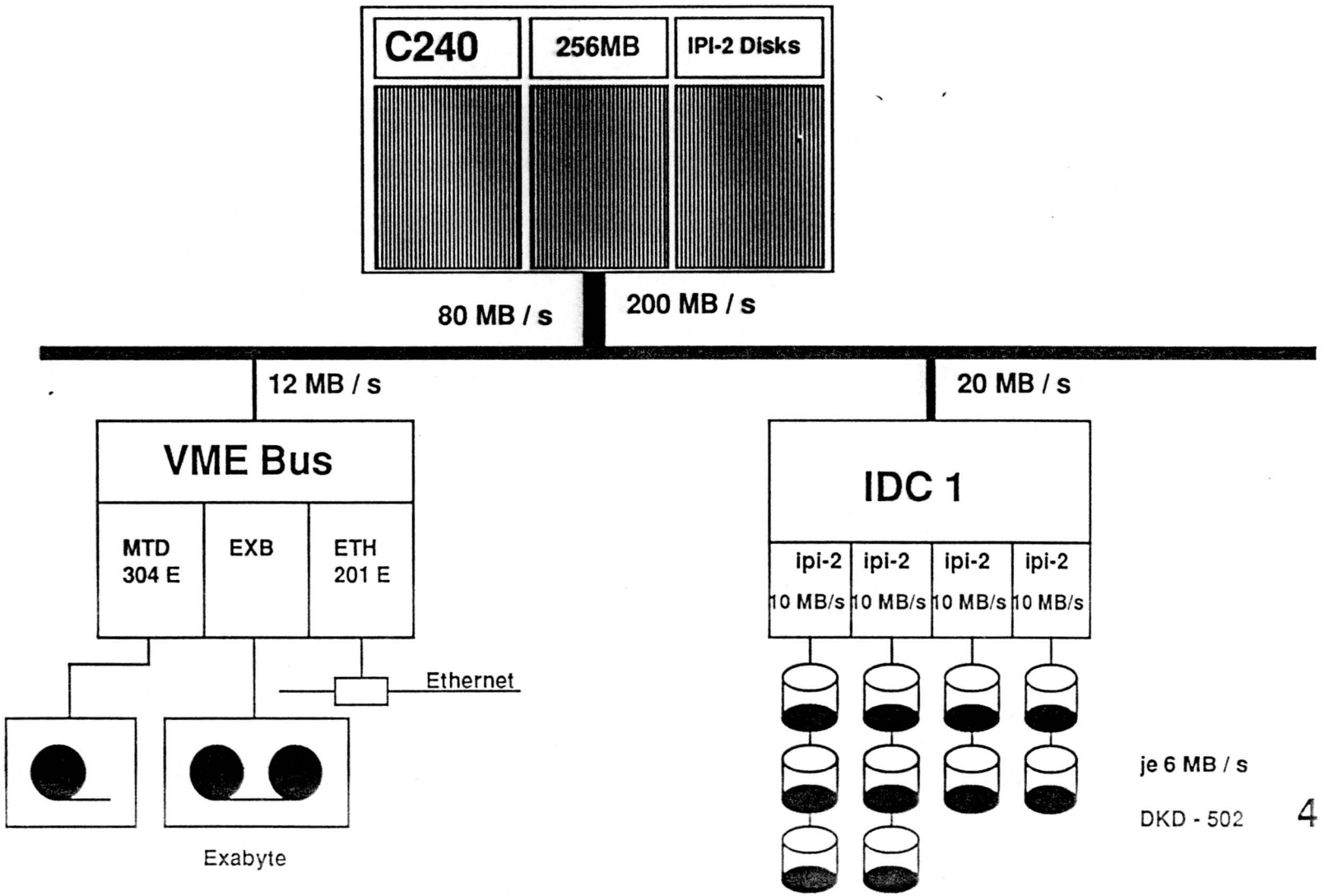
3. FORTRAN Compiler

4. Programm Portierung

5. FORTRAN Optimierung

CONVEX

C240 Konfiguration



Was sind Mflops ?

Mflops = Million Floating Point Operations per second

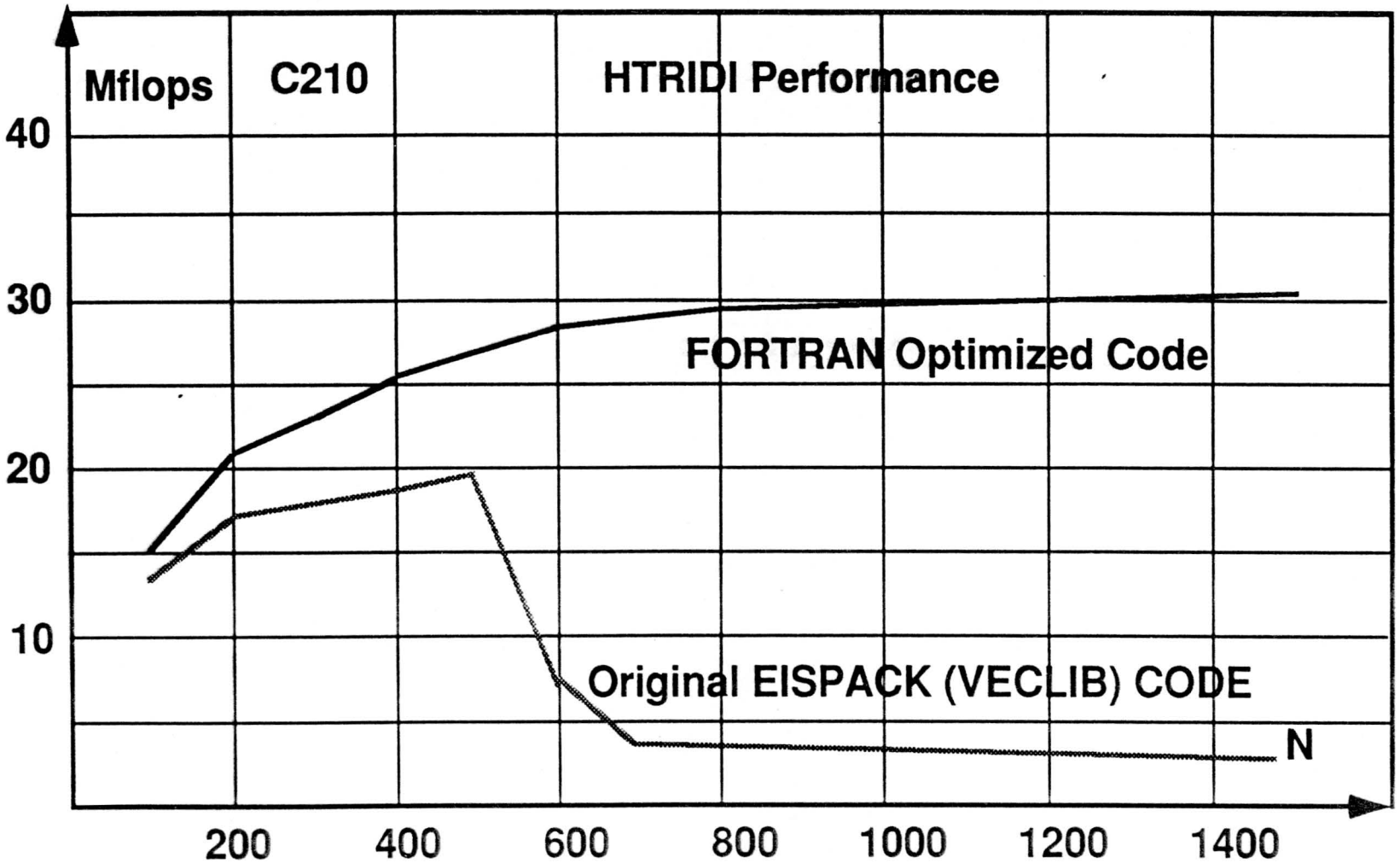
CONVEX C2 : 50 Mflops pro Prozessor = M
CONVEX C2 : 40 ns Zykluszeit = T
CONVEX C2 : 2 arithmetische Einheiten (add,mul) = F

M = F / T

Leistungsangaben werden in Mflops angegeben

CONVEX

Fallstudie : Matrixdiagonalisierung



Hardware Überblick

1. Einführung

2. Hardware Überblick

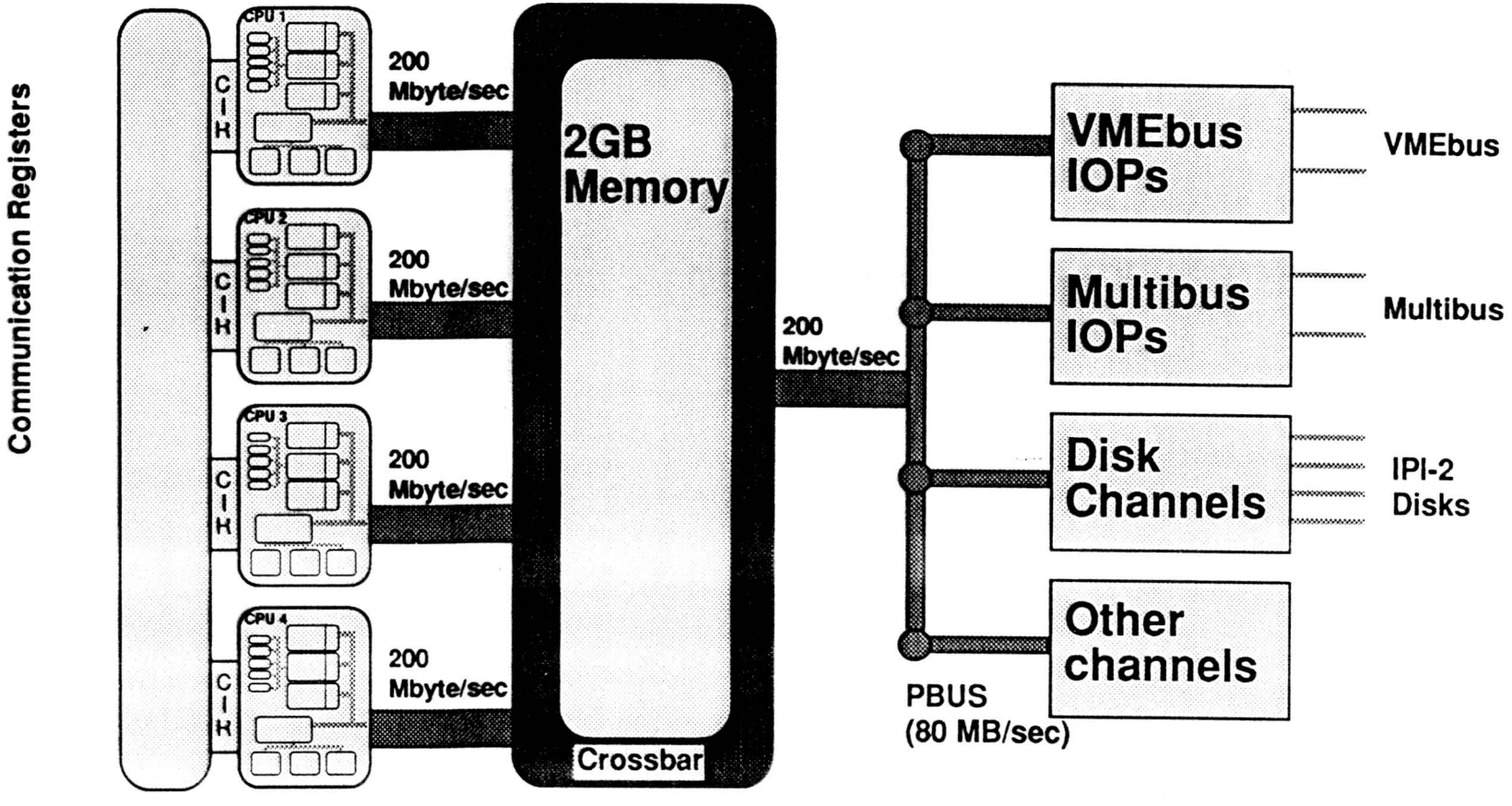
3. FORTRAN Compiler

4. Programm Portierung

5. FORTRAN Optimierung

CONVEX

C210-C240 Architektur



Was ist ein Supercomputer ?

Hohe Taktfrequenz oder kurze Zykluszeit
Unabhängige Funktionseinheiten
Segmentierte Funktionseinheiten
Vektor-Funktionseinheiten
Großer Hauptspeicher (100 MB - 2GB)
Mehrere, parallel arbeitende Cpu's
Balanciertes I/O Subsystem
Hochoptimierende Compiler

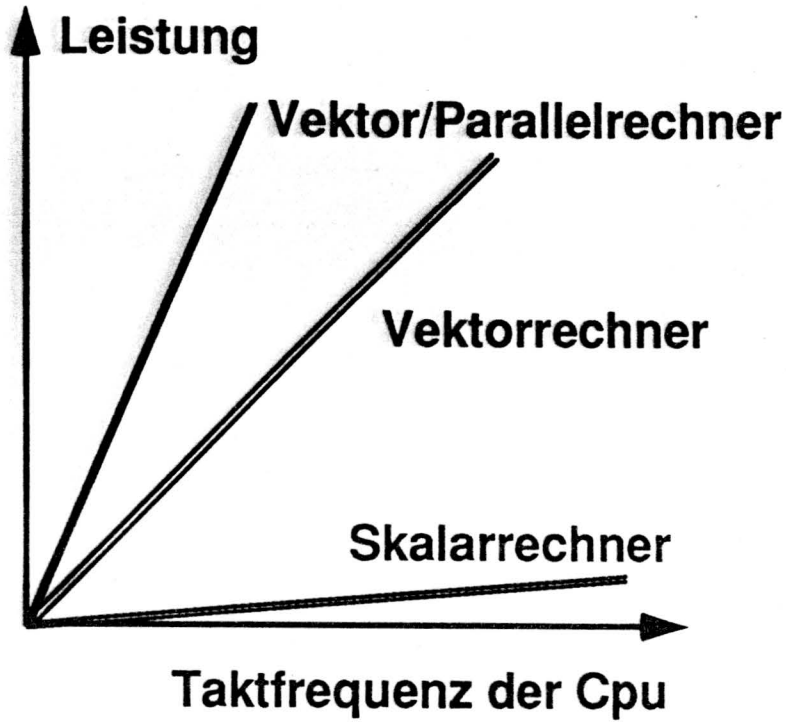
CONVEX

Rechnerarchitekturen

Skalarrechner

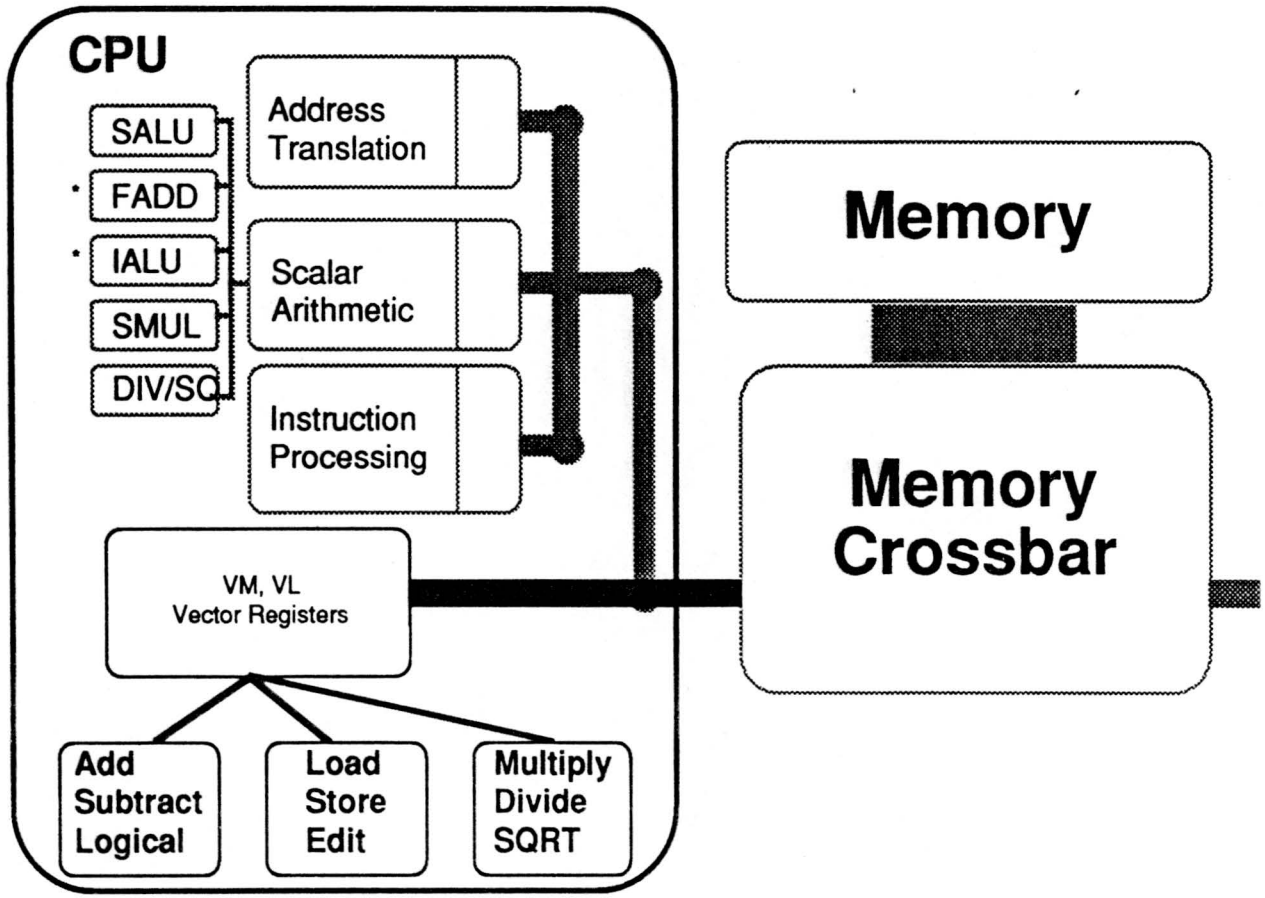
Vektorrechner

Vekor/Parallel-rechner



CONVEX

CONVEX Cpu



Pipelining

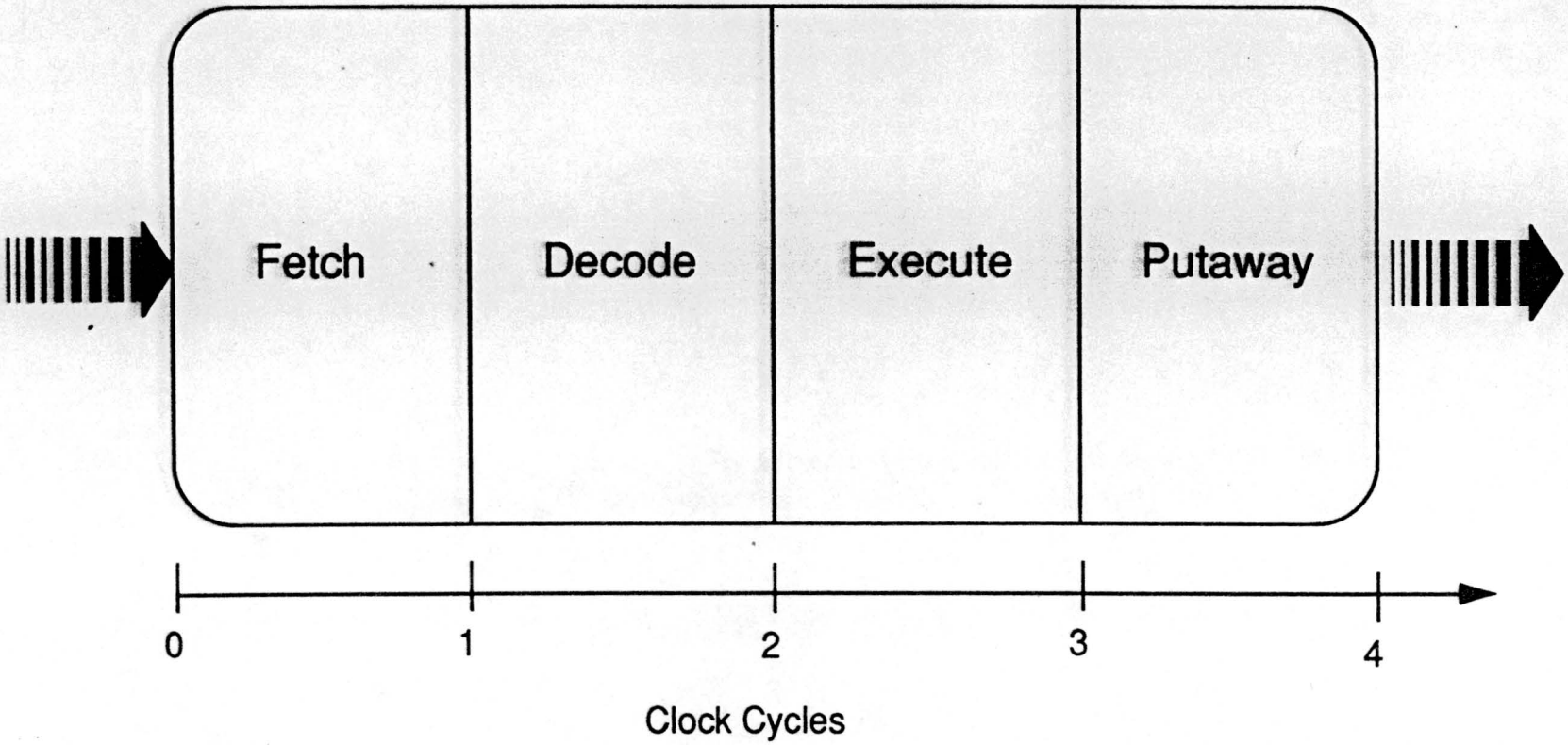
Aufteilung einer Funktionseinheit in n Untereinheiten.

Jede Untereinheit benötigt zur Ausführung 1 Taktzyklus.

n Aktionen können sich in dieser Funktionseinheit überlappen.

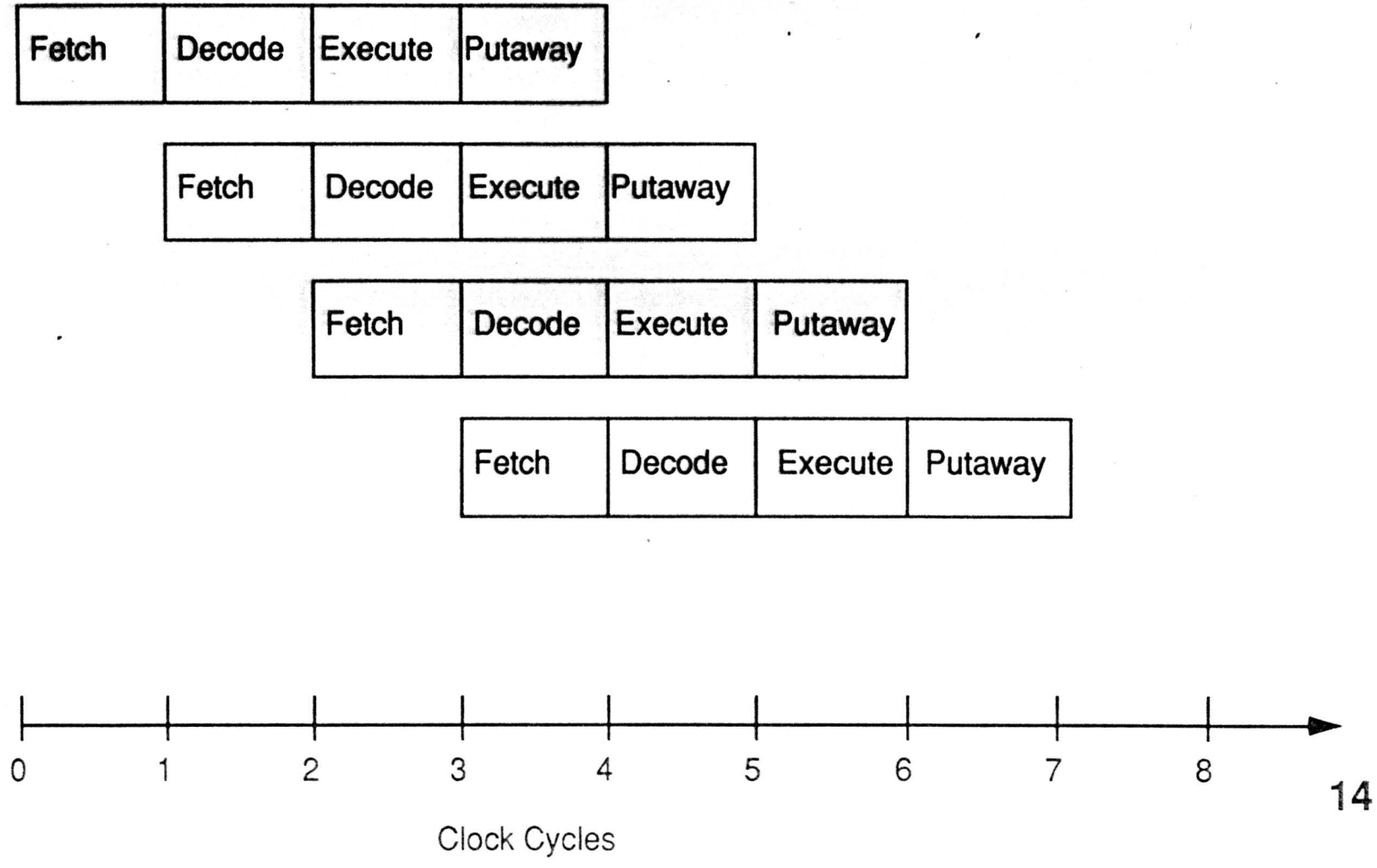
Fließbandverarbeitung.

Pipelining



CONVEX

Pipelining : Instruction Processing Unit



Vektorisierung

Ein Befehl wird auf viele gleich strukturierte Operandenpaare angewendet.

Pipeline-Funktionseinheiten

Pro Taktzyklus wird ein Resultat geliefert .

Voraussetzung:

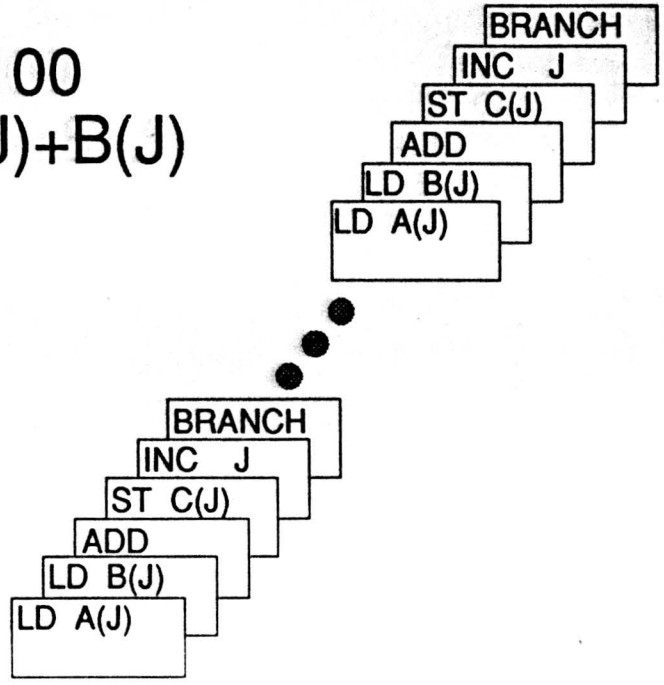
Die Funktionseinheiten müssen genügend schnell mit Daten versorgt werden.

Es müssen genügend Operandenpaare zur Verarbeitung vorhanden sein.

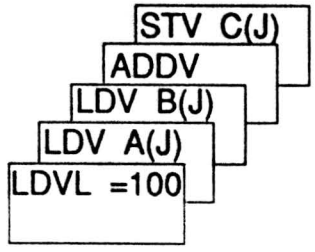
CONVEX

VeKtor Verarbeitung vs Skalarverarbeitung

```
DO  J=1,100
  C(J)=A(J)+B(J)
ENDDO
```



Scalar:
~600 Instructions



Vector:
5 Instructions

Vektor Cpu

3 Funktionale Einheiten **Load/Store** **add/subract** **multiply/divide**

8 Vektorregister **128 Elemente lang** **64 bit breit**

Vektorlängenregister

Vektormergeregister **128 Elemente lang** **1 bit breit**

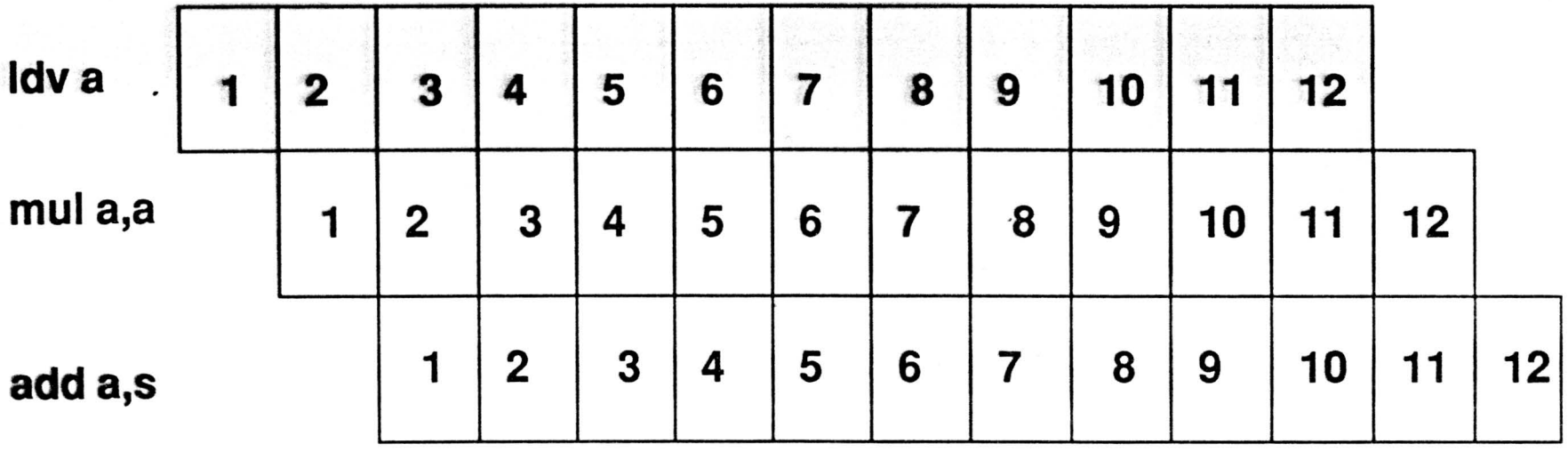
Chaining : Hintereinanderschalten von Vektorfunktionseinheiten

ohne Zwischenspeichern der Ergebnisse

CONVEX

Chaining

```
do i = 1 , 100
  s = s + a(i) * a(i)
enddo
```



CONVEX

Aufbau des Hauptspeichers

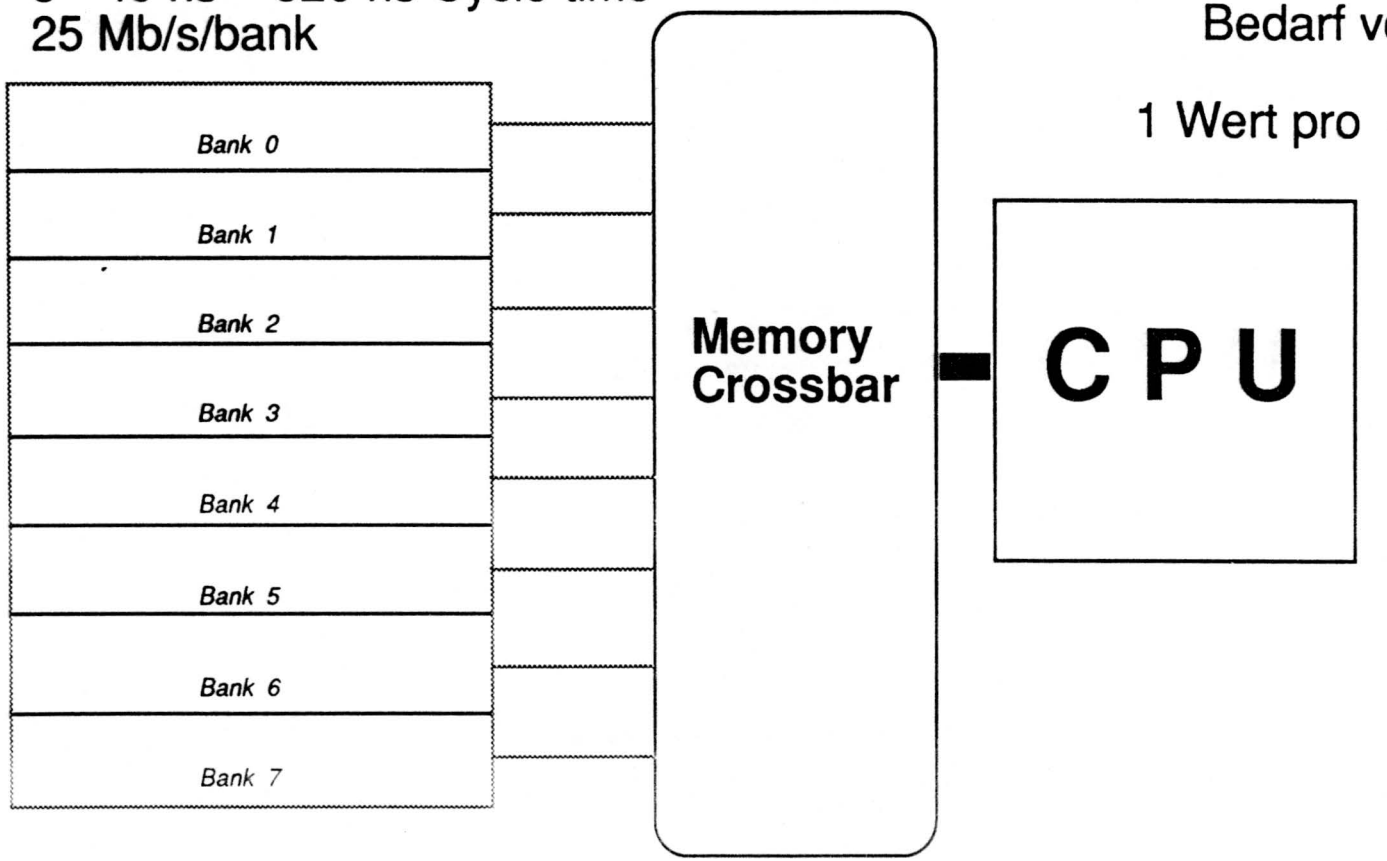
Do c=1,4
S = S+a(c)

Do c=1,4,8
S=S+a(c)

Segmentierter
Hauptspeicher
8 * 40 ns = 320 ns Cycle time
25 Mb/s/bank

Bedarf von a(256,256)

1 Wert pro 40 ns



FORTRAN Compiler

1. Einführung

2. Hardware Überblick

3. FORTRAN Compiler

4. Programm Portierung

5. FORTRAN Optimierung

Programm Portierung

1. Einführung

2. Hardware Überblick

3. FORTRAN Compiler

4. Programm Portierung

5. FORTRAN Optimierung

Vorbereitungen

Annahme : Programm mit 5000 Zeilen Source Code, 40 Subroutinen, mehrere Eingabefiles

Wie erzeuge ich ein CONVEX lauffähiges Programm ?

1) tr A-Z a-z < inpfile > outfile für alle Files

2) fsplit fortran-file : zerlegt das Programm in Module

3) Erstelle Makefile

4) Testläufe

Matrix x Vector

```
Subroutine matvec1(nm,n,a,b,c)
real *8 a(nm,n),b(n),c(n)

do i=1,n
sum = 0.d0
  do j=1,n
    sum = sum +a(i,j)*c(j)
  enddo
enddo

return
end
```

```
subroutine matvec2(nm,n,a,b,c)
real *8 a(nm,n),b(n),c(n)

do i=1,n
b(i)=0.d0
  do j=1,n
    b(i) = b(i)+a(i,j)*c(j)
  enddo
enddo

return
end
```

Compiler Strategy

```
original code
do i=1,n
b(i)=0.d0
  do j=1,n
    b(i) = b(i)+a(i,j)*c(j)
  enddo
enddo
```

```
step 1 : loop distribution
do i=1,n
b(i)=0.d0
enddo
do i=1,n
  do j=1,n
b(i)=b(i)+a(i,j)*c(j)
enddo
enddo
```

```
step 2 : loop interchange
do j=1,n
   do i=1,n
     b(i)=b(i)+a(i,j)*c(j)
    enddo
enddo
```

```
step 3 : strip mining
do j=1,n
  do iout =1,n,128
     do i=iout,iout+128
       b(i)=b(i)+a(i,j)*c(j)
     enddo
  enddo
enddo
```

Compiler Strategy

```
step 4 : vectorization
do j=1,n
  do iout=1,n,128
    v0=b(iout:iout+128)
    v1=a(iout:iout+128,j)
    s0= c(j)
    v0=v0+v1*s0
  enddo
enddo
```

```
step 5 : loop interchange
do iout=1,n,128
  do j=1,n
    v0=b(iout:iout+128)
    v1=a(iout:iout+128,j)
    s0= c(j)
    v0=v0+v1*s0
    b(iout:iout+128)=v0
  enddo
enddo
```

```
step 6 : hoisting+sinking
do iout=1.n,128
  v0=b(iout:iout+128)
    do j=1,n
     s0 = c(j)
     v1=a(iout:iout+128,j)
     v0=v0+v1*s0
    enddo
  b(iout:iout+128) = v0
enddo
```

Ergebnis : Peak performance !!