# Numerical Modelling in FORTRAN day 7

Paul Tackley, 2020

# Today's Goals

1. **Makefiles**
2. **Intrinsic functions**
3. **Pointers**
4. **Optimisation**: Making your code run as fast as possible

# Projects: start thinking about

Agree topic with me
before final lecture

# Makefiles

- If your programs are split over several files, it may be easiest to write instructions for compiling them in a **makefile** (this applies mainly to unix-like systems)
- This also makes sure that the various options you use remain the same
- The makefile defines dependencies, so it only recompiles source files that have changed since the last compilation
- See manual pages for full information

# Example makefile

```
% example makefile

FFLAGS = -O2
OBJECTS = main.o modules.o

myproject: $(OBJECTS)
	gfortran -o myproject $(OBJECTS)

main.o : main.f90 stuff.mod
	gfortran -c main.f90

stuff.mod : module.o

modules.o : modules.f90
	gfortran -c modules.f90
```

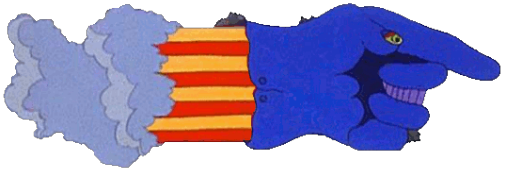
FFLAGS lists flags to be used when compiling fortran

: gives dependencies

USAGE: Just type in “make” or “make myproject”

# Intrinsic functions

- There are many more intrinsic functions than we have used!
- It is good to browse a list so that you know what is available, for example at
- http://www.nsc.liu.se/~boein/f77to90/a5.html
- This doesn't list cpu_time(), which is useful for checking how long parts of the code take



# Pointers

- Point to either
  - (i) data stored by a variable declared as a **target**, or
  - (ii) an area of allocated memory
  - (iii) a function
- (i) is mainly useful for creating interesting data structures (like linked lists); for us (ii)-(iii) will be the most useful.
- "**=>**" is used to point a pointer at something
- See examples next slide
- Functions can also return pointers ("**pointer functions**")

# Pointers to scalar variables

```
program pointless
  implicit none
  real,pointer:: p
  real,target:: a=3.1,b=4.7

  p=>a                ! point p to a
  print*,p            ! prints 3.1 (value of a)
  p=4.0               ! changes a to 4.0
  print*,a            ! prints 4.0
  p=>b                ! now points to b
  print*,b            ! prints 4.7
  print*,associated (p) ! test status: prints true
  nullify(p)          ! points p at nothing
  print*,associated (p) ! prints false

end program pointless
```

# Association status

- 3 values:
  - Undefined : status after meing declared
  - Associated : pointing to something
  - Unassociated: not pointing
- Avoid “undefined” status! Initialise pointers to null():
  - Real, pointer :: p1 => null()
  - Null() can be used anywhere instead of nullify(p1)

# Pointing pointers to pointers

- p2=>p1 sets p2 to point to the same thing as p1, but if p1 is re-pointed, p2 is not:

```
program example

  real,pointer:: p1,p2
  real,target:: t1=1.25, t2=3.5

  p1=>t1
  p2=>p1
  print*,p1,p2
  p1=>t2
  print*,p1,p2

end program example
```

```
 1.25000000      1.25000000
 3.50000000      1.25000000
```

# Pointers to array or array section

```
program pointarray
  implicit none
  integer,parameter:: n=7
  real,target:: T(n,n)
  real,pointer:: bot_boundary(:), &
       top_boundary(:), center4(:,:)

  top_boundary => T(:,1)
  bot_boundary => T(:,n)
  center4 => T(n/2:n/2+1,n/2:n/2+1)

  call random_number(T)
  top_boundary = 0.0  ! easy way to do
  bot_boundary = 1.0  ! boundary conditions
  center4 = 1.0

  print*,T
end program pointarray
```

# Pointers in derived types

similar to allocatable array (no 'target' needed, fortran creates the target array)

```
program pointertype
  implicit none

  type array2D
     real,pointer:: a(:,:)   ! 2D array
  end type array2D

  type(array2D),allocatable:: T(:) ! 1D array of 2D arrays
  integer i,n,ng

  print '(a,$)',"Number of multigrid levels:"
  read*,ng
  allocate (T(ng))  ! allocate number of grids
  do i = 1,ng
     n = 2**(i+1)         ! #grid points in this grid
     print*,'Allocating grid',n,n
     allocate (T(i)%a(n,n))  ! each grid
  end do

  !.... rest of program

end program pointertype
```

# Swapping arrays with pointers
## Avoids array copying

```
program swap_arrays

  real,dimension(5,5),target :: a=1., b=2., tmp
  real,dimension(:,:),pointer:: pa, pb, ptmp

  pa => a; pb => b

  ! non-pointer version: array copying required
  tmp = a
  a   = b
  b   = tmp

  ! pointer version: no array copying
  ptmp => pa
  pa   => pb
  pb   => ptmp

end program swap_arrays
```

```
program function_pointer
  implicit none
  procedure(xsq), pointer:: p
  real:: z=2.
  p => xsq
  print*,p(z)
  p => Gaussian
  print*,p(z)

contains

  real function xsq(x)
    implicit none
    real,intent(in):: x
    xsq = x**2
  end function xsq

  real function Gaussian(x)
    implicit none
    real,intent(in):: x
    real,parameter:: pi=4.0*atan(1.0)
    Gaussian = 1./sqrt(2*pi) * exp(-0.5*x**2)
  end function Gaussian

end program function_pointer
```

# Pointer to function

Output:

```
4.00000000
5.39909676E-02
```

# Pointer to function: analysis

procedure (functionname),pointer:: p

- Sets up a pointer to a function with the arguments (number and types) of functionname, but can be used to point to any function with the same arguments

# Linked list

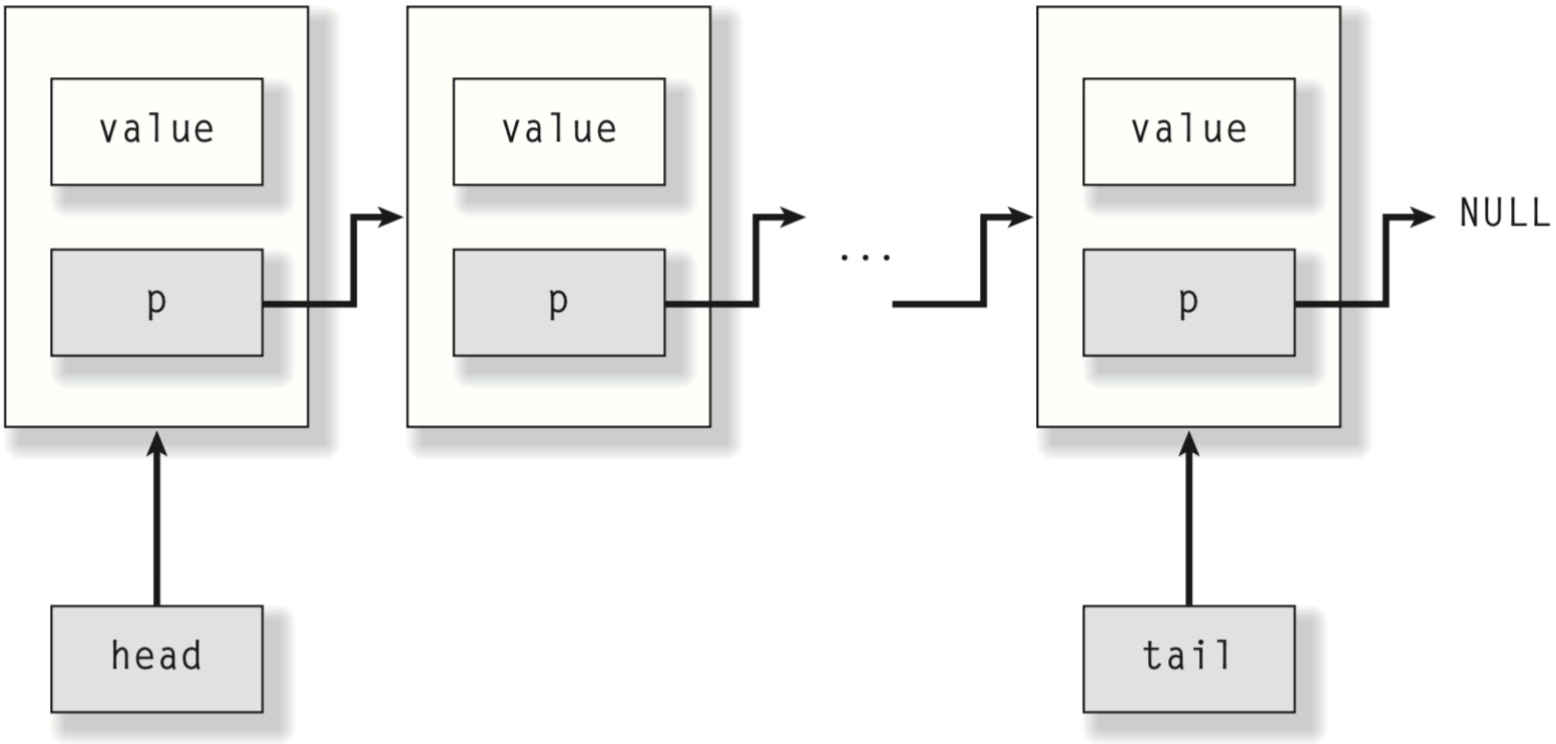



**FIGURE 15-12**
A typical linked list. Note that pointer in each variable points to the next variable in the list.

*Chapman, Fortran for Scientists & Engineers (2017)*

# Linked list

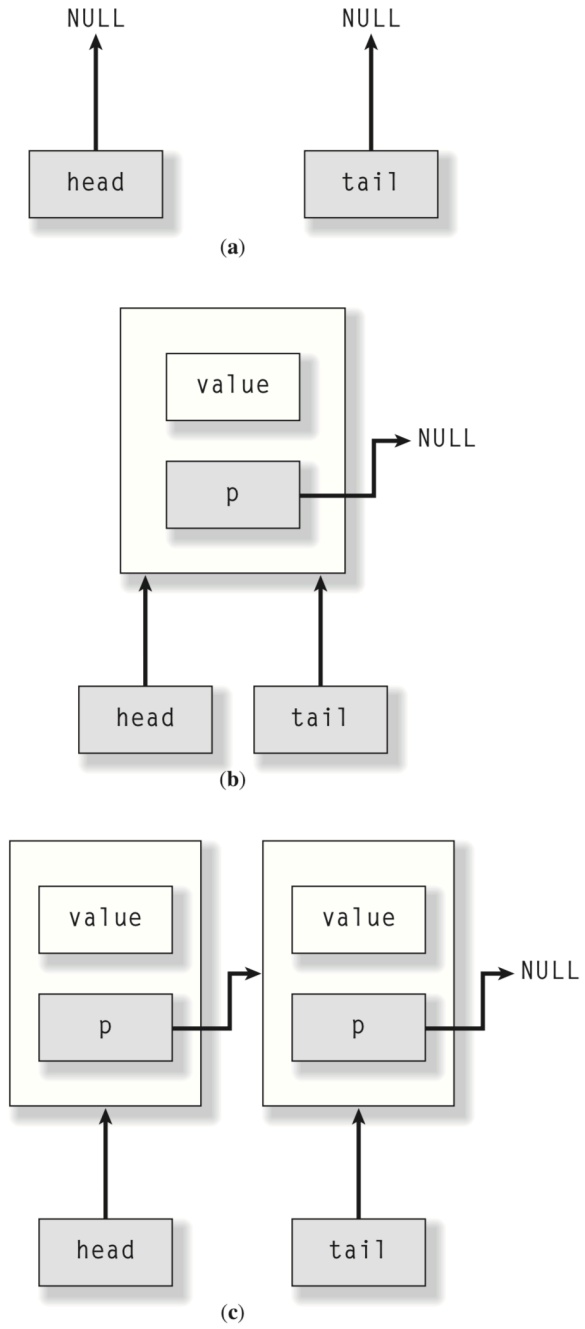



**FIGURE 15-13**
Building a linked list: (*a*) The initial situation with an empty list. (*b*) After adding one value to the list. (*c*) After adding a second value to the list.

*Chapman, Fortran for Scientists & Engineers (2017)*

# Linked list

```
program linked_list  ! Based on a program in S. J. Chapman's book
  implicit none

  type:: real_value
     real :: value
     type(real_value),pointer :: next_value
  end type real_value

  type(real_value),pointer :: head, tail, p

  integer istat
  real a_in

  open(1,file='numbers.dat',status='old') ! read #s from file
  do
     read(1,*,iostat=istat) a_in ; if (istat/=0) exit
     if (.not.associated(head)) then  ! No values in list
        allocate (head)               ! Allocate new value
        tail => head                  ! Point tail to new value
     else                             ! Values already in list
        allocate (tail%next_value)    ! Allocate new value
        tail => tail%next_value       ! Point tail to new value
     end if
     tail%value = a_in              ! Store number
     nullify (tail%next_value)      ! Nullify next value pointer
  end do
  close(1)

  p => head      ! write #s to stdout
  do
     if (.not.associated(p)) exit
     print*, p%value
     p => p%next_value
  end do

end program linked_list
```

e.g.

```
1.230000
2.340000
3.450000
4.560000
5.670000
6.780000
```

# Speed and optimization

- Running large simulations can take a long time => speed is important. **Optimization**=making it run as fast as possible
- First consideration: use the most efficient algorithm, e.g., multigrid
- Then: get code working using code that is easy-to-read and debug
- Finally: Find out which part(s) of the code are taking the most time, and rewrite those to optimize speed
- Code written for maximum speed may not be the most legible or compact!

# Manual versus automatic optimisation

- Many steps can be done automatically by the compiler. Use appropriate compiler options (see documentation), e.g.,
  - -O2, -O3 -Ofast: selects a bunch of optimisations
  - -unroll:  unroll loops
  - etc. (see compiler documentation)
- Some need to be done manually. In general, try to write code in such a way that the compiler can optimise it!

# Example of compiler optimisation

- Solution AdvDif code, test case, 32-bit, gfortran, iMac. (use “time a.out”)

| Options | Time (s) |
|---|---|
| Recommended debugging | 27.05 |
| none | 15.27 |
| -O1 | 3.371 |
| -O2 | 3.368 |
| -O3 | 2.754 |
| -O3 –ffast-math –ftree-vectorize | 2.688 |
| -Ofast | 2.652 |

# Example of compiler optimisation

- Solution convection code (this week), test case, 32-bit, gfortran, macbook pro

| Options | Time (s) |
|---|---|
| None | 31.57 |
| -O1 | 24.741 |
| -O2 | 23.578 |
| -O3 | 23.625 |
| -O2 –ffast-math | 20.958 |
| -O2 –ffast-math –ftree-vectorize | 20.552 |

# Manual optimization step 1: Identify bottlenecks

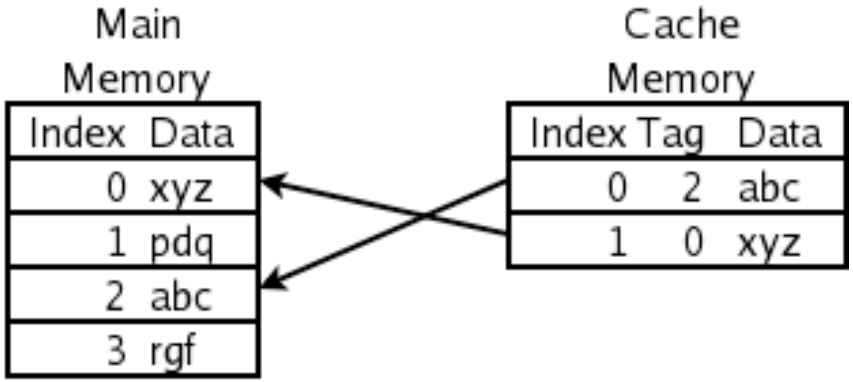
- The 90/10 law: 90% of the time is spent in 10% of the code. Find this 10% and work on that!
- e.g., Use a profiler. Or put cpu_time() statements in to time different subroutines or loops
- Usually, most of the time is spent in loops. In our multigrid code it is probably the loops that update the field and calculate the residue. Optimization of loops is the most important consideration.

# To understand optimization it is important to understand how the CPU works

- Two aspects are particularly important:
  - Cache
  - Pipelining

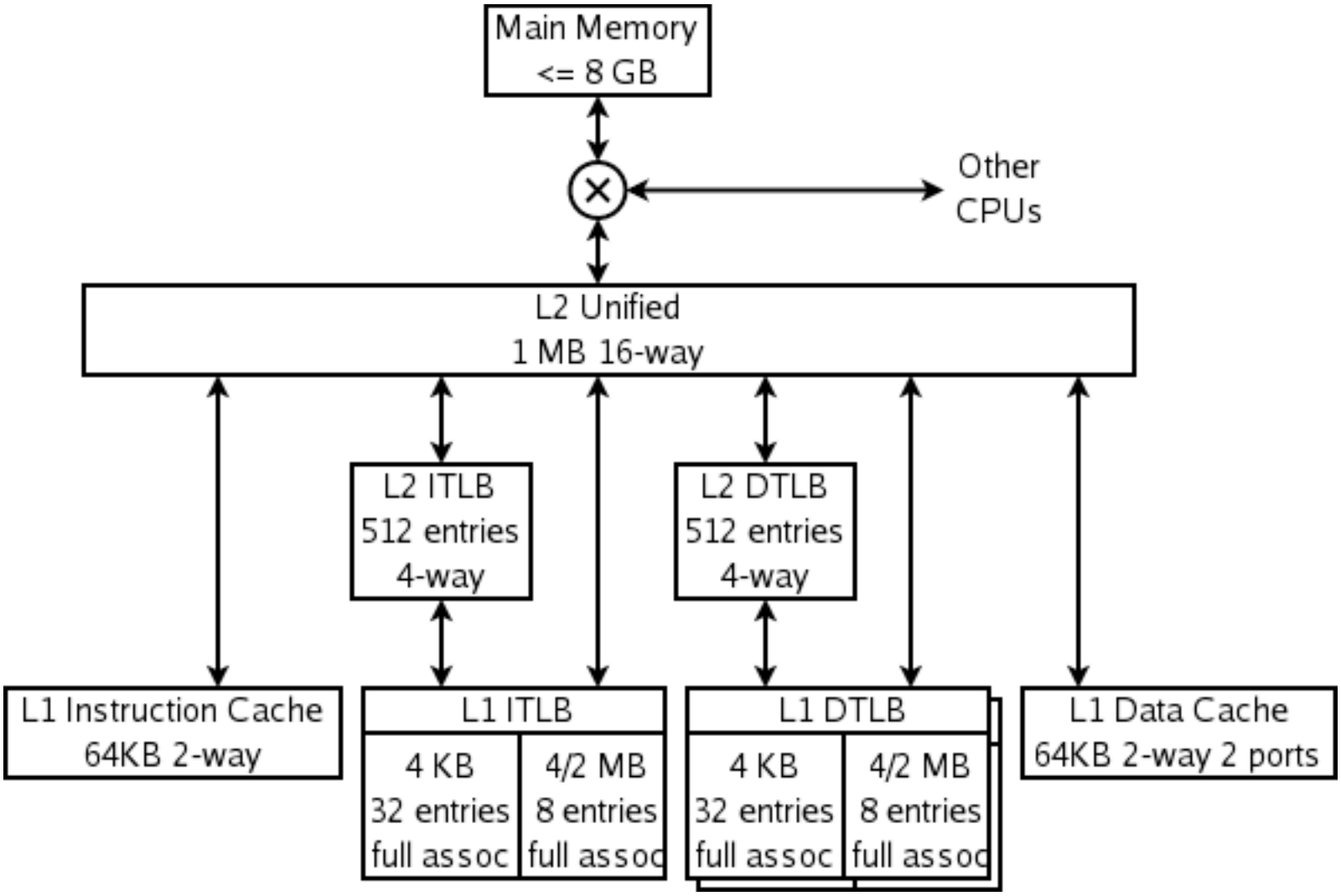
# Cache memory

- Fast, small memory close to a CPU that stores copies of data in the main memory




Substantially reduces latency of memory accesses.
Designing code such that data fits in cache can greatly improve speed. Good design includes memory locality, and not-too-large size of arrays.

# Athlon64 multiple caches

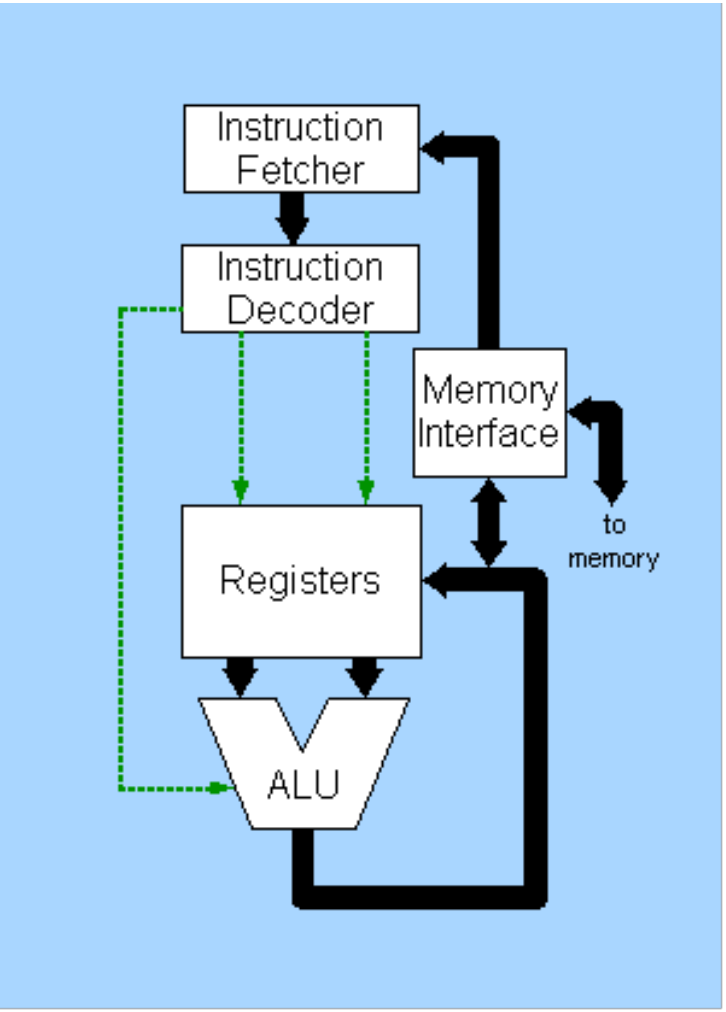


# Tips to improve cache usage

- Memory locality: data within each block should be close together (small stride). Appropriate data structures and ordering of nested loops. (see later)
- Arrays shouldn't be too large. e.g., for a matrix*matrix multiply, split each matrix into blocks and treat blocks separately

# CPU architecture and pipelining

(images from http://en.wikipedia.org/wiki/Central_processing_unit)

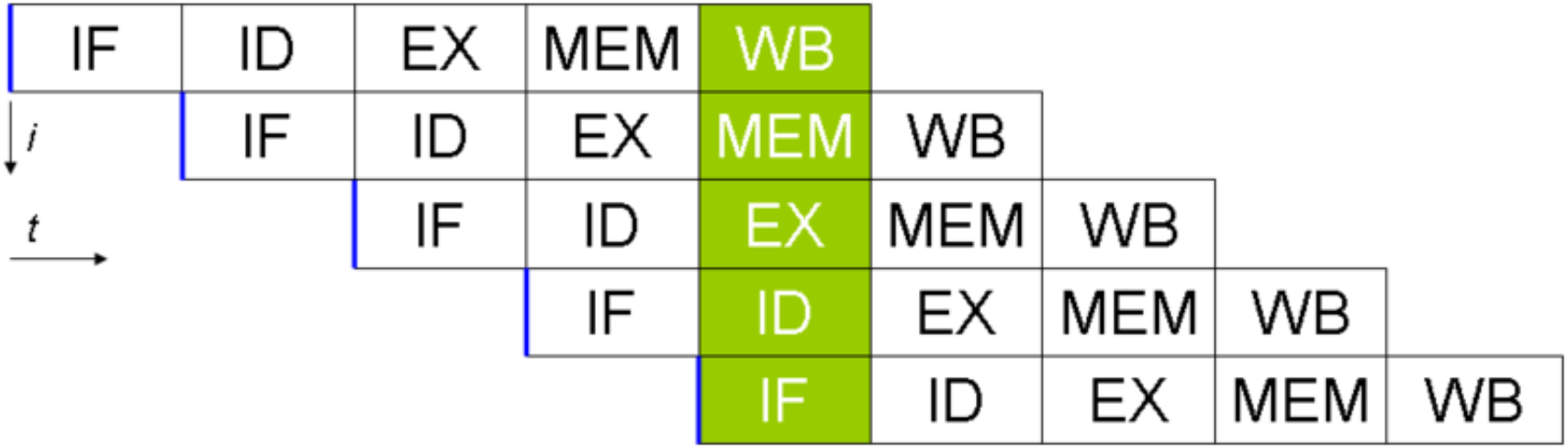



Executing an instruction takes several steps. In the simplest case these are done sequentially, e.g.,


15 cycles to perform 3 instructions!

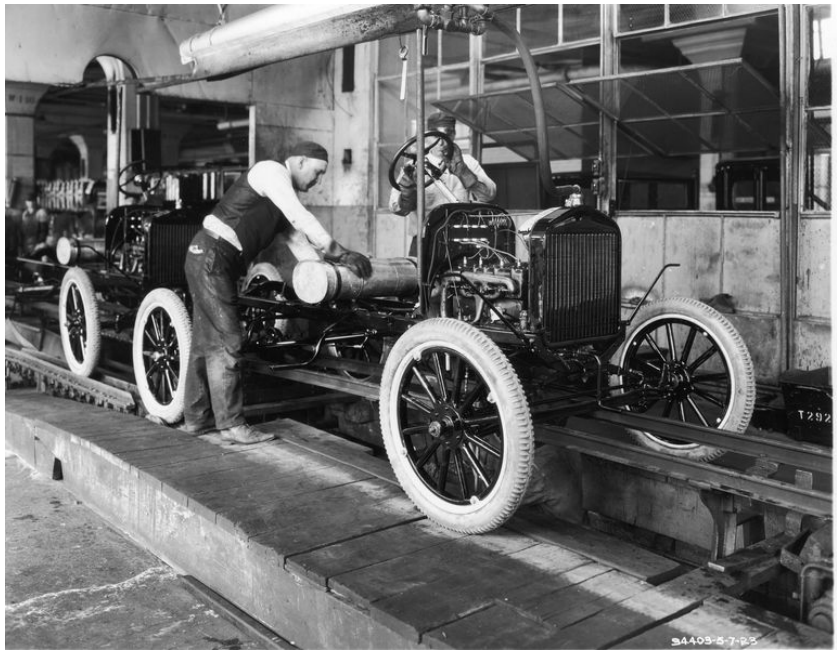
# Pipelining

If each step can be done independently, then up to 1 instruction/cycle can be sustained => 5* faster




Basic 5-stage pipline. Like an assembly line.
Several cycles are needed to start and end the pipeline.

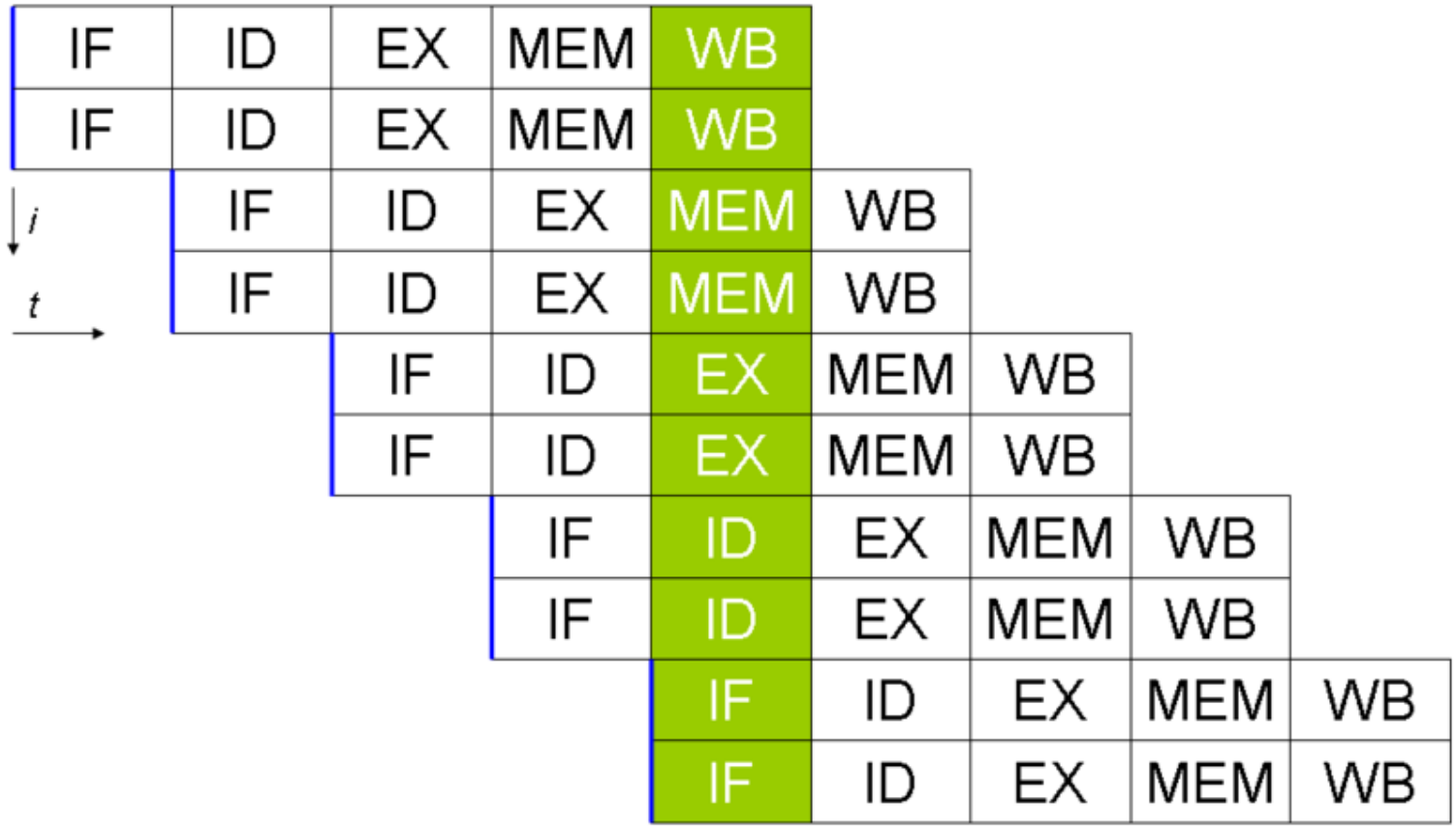
# Like a car assembly line



1913: sped up assembly of Model T Ford from 12 hours to <3 hours!

# Superscalar pipeline

More than 1 instruction per cycle (2 in the example below)



# Pipelining in practice

- Done by the compiler, but must write code to maximize success
- Branches (e.g., “if”) cause the pipeline to flush and have to restart
- Avoid branching inside loops!
- Helps if data is in cache

# Summary

- Goal is to maximize use of cache and pipelining.
- Design code to reuse data in cache as much as possible, and to stream data efficiently through the CPU (pipeline / vectorisation)

# More information

- Wikipedia pages
  - http://en.wikipedia.org/wiki/Loop_optimization
  - http://en.wikipedia.org/wiki/Optimization_(computer_science)
  - http://en.wikipedia.org/wiki/Memory_locality
  - http://en.wikipedia.org/wiki/Software_pipelining
- http://www.ibiblio.org/pub/languages/fortran/ch1-9.html
- http://www.azillionmonkeys.com/qed/optimize.html
- ETH course “How to write fast numerical code” (Frühjahrssemester)

# Time taken for various operations

- Slowest: sin, cos, **, etc.
- sqrt
- /
- *
- fastest: + -
- Simplify equations in loops to minimize number of operators, particularly slow ones!

# Loop optimization (1)

- Remove conditional statements from loops!

SLOW

```
do i=1,n
   if (condition) then
      a(i) = b(i)+c(i)
   else
      a(i) = b(i)-c(i)
   end if
end do
```

FAST

```
if (condition) then
   do i=1,n
      a(i) = b(i)+c(i)
   end do
else
   do i=1,n
      a(i) = b(i)-c(i)
   end do
end if
```

# Loop optimization (2)

- Data locality: fastest if processing nearby (e.g., consecutive) locations in memory
- Fortran arrays: first index accesses consecutive locations (opposite in C)
- Order loops such that first index loop is innermost, 2nd index loop is next, etc.

SLOW

```
do i=1,n
   do j=1,m
      a(i,j) = b(i,j)+c(i,j)
   end do
end do
```

FAST

```
do j=1,m
   do i=1,n
      a(i,j) = b(i,j)+c(i,j)
   end do
end do
```

# Loop optimization (3)

- Unrolling: eliminate loop overhead by writing loops as lots of separate operations
- Partial unrolling: reduces number of cycles, reducing loop overhead

Original

```
do i=1,n
   a(i)=b(i)*c(i)
end do
```

Unrolled by factor 4

```
do i=1,n,4
   a(i)  =b(i)  *c(i)
   a(i+1)=b(i+1)*c(i+1)
   a(i+2)=b(i+2)*c(i+2)
   a(i+3)=b(i+3)*c(i+3)
end do
```

This can be done automatically by the compiler

# Loop optimization (4)

- fusing + unrolling: see below

Original loop

```
do j=1,2*n
   do i=1,m
      a(i)=a(i)+1./real(i+j)
   end do
end do
```

Partial unrolling

```
do j=1,2*n,2
   do i=1,m
      a(i)=a(i)+1./real(i+j)
      a(i)=a(i)+1./real(i+j+1)
   end do
end do
```

+fusion (reduces number of writes by factor 2)

```
do j=1,2*n,2
   do i=1,m
      a(i)=a(i)+1./real(i+j)+1./real(i+j+1)
   end do
end do
```

# Loop optimization (5): other things

- simplify calculated indices
- use registers for temporary results
- Put invariant expressions (things that don’t change each iteration) outside the loop
- Loop blocking/tiling: splitting a big loop or nested loops into smaller ones in order to fit into cache.

# Other Optimizations

- Use binary I/O not ascii
- Avoid splitting code into excessive procedures.
  - Overhead associated with calling functions/subroutines
  - Reduces ability of compiler to do global optimizations
- Use procedure inlining (done by compiler): compiler inserts a copy of the function/subroutine each time it is called
- Use simple data structures in major loops to aid compiler optimizations (derived types may slow things down)