

Hardware

Thomas Ericsson
Chalmers

thomas@math.chalmers.se
<http://www.math.chalmers.se/~thomas>
<http://www.math.chalmers.se/~thomas/PDC>

PDC Summer School 2000

1

Why this lecture?	3
CISC - Complex Instruction Set Computer	4
RISC - Reduced Instruction Set Computer	8
Pipelining	10
Superscalar CPUs	15
Memory is the problem - caches	17
An example; the MIPS R10000-processor (SGI)	22
RISC and elementary functions	23
Parallel computers	26
Virtual memory	32
Some important points	34

2

Why this lecture?

Knowledge about hardware is necessary:

- to understand the behaviour of programs
- in order to pick the most efficient algorithm
- to be able to write efficient programs, and to use common resources (e.g PDC) in a good way
- to know what computer to run on
(what type of architecture is your code best suited for)
- to read (some) articles in numerical analysis
- when looking for the next computer to buy
(and to understand those PC-ads., L2-cache, MHz, RISC...)

The change of computer architecture has made it necessary to re-design software, e.g Linpack \Rightarrow Lapack.

This lecture starts with a short historical perspective and then follows a discussion of modern computers and processors.

3

For more details about computer architecture, see for example:

- John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach, Second Edition, 1995, 760 pages, ISBN 1-55860-329-8.
http://www.mkp.com/books_catalog/1-55860-329-8.asp
- Kevin Dowd & Charles Severance, High Performance Computing (2nd ed.), O'Reilly & Associates, 1998.
<http://www.oreilly.com/catalog/hpc2/>
Details about code optimization as well.

CISC - Complex Instruction Set Computer

year 0: "The Nine Chapters on the Mathematical Art"
(Jiuzhang Suanshu),
"Gaussian elimination" using counting rods

...

June, 1984: Motorola 68020 chip available

Sun3 workstation, Macintosh II (March 1987) had this chip

August 1986, Intel ships the 80386 and the same year came the 80387 FPU.

...

Stephen G. Nash (ed.), A History of Scientific Computing, ACM Press, 1990.

4



The CPU contains the ALU, arithmetic and logic unit and the control unit. The ALU performs operations such as +, -, *, / of integers and Boolean operations.

The control unit is responsible for fetching, decoding and executing instructions.

The memory stores programs and data.

I/O-devices are disks, keyboards etc.

The CPU contains several registers, such as:

- PC, program counter, contains the address of the next instruction to be executed
- IR, instruction register, the executing instruction
- address registers (the 68020 had 8 32-bit registers)
- data registers (the 68020 had 8 32-bit registers)

The memory bus usually consist of one address bus and one data bus. The data bus may be 64 bits wide and the address bus may be 32 bits wide.

All operations in the computer are synchronized by (several) clocks. A modern workstation may run at 300 MHz (clock frequency) or more. The buses are usually much slower, e.g Compaq Deskpro, 667 MHz Pentium III, 133 MHz bus.

5

ways to address the operands) and can work on seven data types (essentially bits and integers).

The instructions take different time to execute on the 68020.

We count the time in clock cycles.

The 68020 could run at 12 MHz or 16 MHz.

On the Mac II both the CPU and memory-bus ran at 16 MHz.

Just to compute the address of an operand using a complicated addressing mode could take 24 cycles.

The 68020 instructions are at least one word (16 bits) long but may be up to 11 words.

Not a very uniform design, in other words.

The instructions were often implemented using microcode, "programs" controlling the logic on low level. Usually stored in ROM (the VAX 11/780 had > 400000 bits of microcode).

How about floating point computation?

- On a Macintosh one could use SANE, Standard Apple Numerics Environment, a software library for +, -, *, /, sin, exp etc.
- Faster was the MC68881 FPU, floating-point unit, co-processor

Execution times, using the 68881, for some operations:

FADD 35 cycles (operands in the floating point registers).

FMUL 71, **FDIV** 103, **FSIN** 391, **FSINH**, 687.

On a modern Sun **FDIV**/**FMUL** can be as high as 22.

6

Why CISC?

For a more detailed history, see the literature.

- Advanced instructions simplified programming (writing compilers, assembly language programming). Software was expensive.
- Memory was limited and slow so short programs were good. (Complex instructions ⇒ compact program.)

Some drawbacks:

- complicated construction could imply a lower clock frequency
- instruction pipelines hard to implement
- long design cycles
- many design errors
- only a small part of the instructions was used
According to Sun: Sun's C-compiler uses about 30% of the available 68020-instructions. Studies show that approximately 80% of the computations for a typical program requires only 20% of a processor's instruction set.

When memory became cheaper and faster the decode and execution on the instructions became limiting.

Studies showed that it was possible to improve performance with a simple instruction set and where instructions would execute in one cycle.

This lead to RISC:

7

RISC - Reduced Instruction Set Computer

- IBM 801, 1979 (publ. 1982)
- 1980, David Patterson, Berkeley, RISC-I, RISC-II
- 1981, John Hennessy, Stanford, MIPS, (Microprocessor without Interlocked Pipeline Stages)
- ≈ 1986, commercial processors

A processor whose design is based on the rapid execution of a sequence of simple instructions rather than on the provision of a large variety of complex instructions.

Some RISC-characteristics:

- load/store architecture; **C = A + B**

```

LOAD  R1,A
LOAD  R2,B
ADD   R1,R2,R3
STORE C,R3
  
```

- fixed-format instructions (the op-code is always in the same bit positions in each instruction which is always one word long)
- a (large) homogeneous register set, allowing any register to be used in any context and simplifying compiler design
- simple addressing modes with more complex modes replaced by sequences of simple arithmetic instructions
- one instruction/cycle
- hardwired instructions and not microcode
- efficient pipelining
- simple FPUs; only +, -, *, / and √-. sin, exp etc. are done in software, C-code.

8

produce, shorter design cycles, faster execution, easier to write optimizing compilers (easier to optimize many simple instructions than a few complicated with dependencies between each other).

CISC - short programs using complex instructions.
RISC - longer programs using simple instructions.

So why is RISC faster?

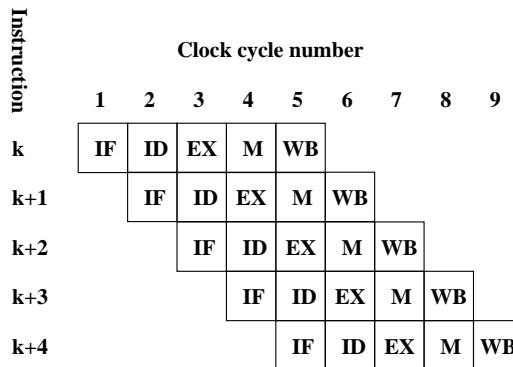
The simplicity and uniformity of the instructions make it possible to use pipelining, a higher clock frequency and to write optimizing compilers.

Will now look at some techniques used in all RISC-computers:

- instruction pipelining
work on the fetching, execution etc. of instructions in parallel
- cache memories
small and fast memories between the main memory and the CPU registers
- superscalar execution
parallel execution of instructions

Were used before RISC (but not so efficiently). The 68020 had a three stage instruction pipeline and a 2 kbyte direct mapped instruction cache, for example.

Analogy: building cars using an assembly line in a factory.



Typically (Hennessy & Patterson) but simplified:

IF : Instruction fetch cycle. Fetch next instruction from memory to IR and increase PC.

ID : Instruction decode.

EX : Execution/effective address cycle. Some examples:
ALU performs an operation.
Compute effective address of an operand.
Compute branch address.

M : Memory access/branch completion cycle. Load or store from/to memory or branch (update PC).

WB : Write-back cycle. Write results from ALU or memory fetch into the register file.

So one instruction completed per cycle once the pipeline is filled.

Not so simple in real life: different kind of hazards, causing stalls (wait cycles) in the pipeline.

- Structural hazards arise from resource conflicts, e.g.
 - two instructions need to access the system bus,
 - not fully pipelined functional units (division on a Sun takes 22 cycles, for example).
- Data hazards arise when an instruction depends on the results of a previous instruction (will look at some cases in later lectures) e.g.
$$a = b + c$$

$$d = a + e \quad d \text{ depends on } a$$
- Control hazards arise from the pipelining of branches (if-statements).

An example of a control hazard:

```
if ( a > b - c * d ) then
do something
else
do something else
end if
```

Must wait for the evaluation of the logical expression.

If-statements in loops may cause poor performance.

Several techniques to minimize hazards (look in the literature for details). Some examples:

Structural hazard:

Add hardware. If the memory has only one port `LOAD adr,R1` will stall the pipeline (the fetch of data will conflict with a later instruction fetch). Add a memory port (separate data and instruction caches).

Data hazards:

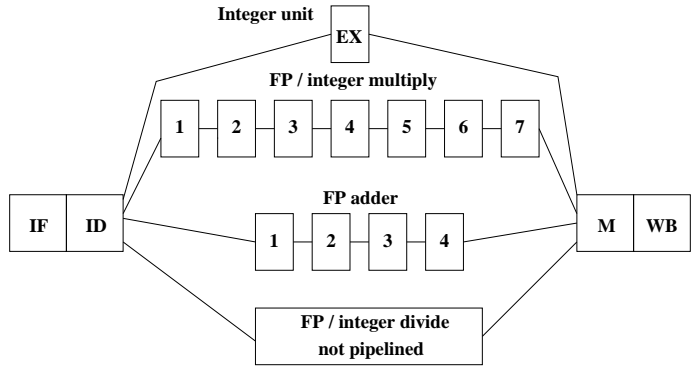
- Forwarding: `b + c` available after EX, special hardware "forwards" the result to the `a + e` computation (to the ALU without involving the CPU-registers).
- Instruction scheduling. The compiler can try and rearrange the order of instruction to minimize stalls. (changing the order between instructions).

```
a = b + c
d = a + e
```

```
load b
load c
add b + c    has to wait for load c to complete
```

```
load b
load c
load e      give the load c time to complete
add b + c   overlap with load e
```

usually take more than one clock cycle (unless we slow the clock down or build expensive hardware).



```

MULTD  IF  ID  mul1 mul2 mul3 mul4 mul5 mul6 mul7 M  WB
ADDD   IF  ID  add1 add2 add3 add4 M    WB
MULTD   IF  ID  mul1 mul2 mul3 mul4 mul5 mul6

MULTD  xxxxxxxxxxxx
MULTD  xxxxxxxxxxxx
MULTD  xxxxxxxxxxxx

Compare division; each xxxxxxxxxxxx is 22 cycles (on Sun):

DIVD  xxxxxxxxxxxx
DIVD   xxxxxxxxxxxx
DIVD   xxxxxxxxxxxx
    
```

Superscalar CPUs

We saw that additions and multiplications can be worked on in parallel.

Superscalar: more than one operation/clock cycle.

Can fetch, issue and complete several (e.g. four) instructions per clock. Not only floating point but also other instructions (a suitable mix in each cycle is necessary).

Will concentrate on floating point operations.

flop = floating point operation.

flops = plural of flop or flop / second.

Top floating point speed =

of processors × flop / s =

of processors × # flop / clock cycle × clock frequency

Some examples of processors:

CPU	+ * per cycle	clock f. MHz	top speed Mflops
IBM RS6000 (SP2)	4	160	640
Mips R10000 (SGI)	2	195	390
Sun Ultra	2	250	500

Why do we often only get a small percentage of these speeds?

Is it possible to reach the top speed (and how)?

Example on a 167 MHz Sun; top speed 334 Mflops:

Instr. fl. p. registers

```

fmuld  %f4,%f2,%f6          faddd  %f4,%f2,%f6
faddd  %f8,%f10,%f12       faddd  %f8,%f10,%f12
fmuld  %f26,%f28,%f30     faddd  %f4,%f2,%f6
faddd  %f14,%f16,%f18     faddd  %f8,%f10,%f12
fmuld  %f4,%f2,%f6         faddd  %f4,%f2,%f6
faddd  %f8,%f10,%f12     faddd  %f8,%f10,%f12
...
    
```

331.6 Mflops

166.1 Mflops

```

fdivd  %f4,%f2,%f6          fdivd  %f4,%f2,%f6
faddd  %f8,%f10,%f12       fdivd  %f8,%f10,%f12
fdivd  %f4,%f2,%f6         fdivd  %f4,%f2,%f6
faddd  %f8,%f10,%f12     fdivd  %f8,%f10,%f12
...
    
```

15.1 Mflops

7.5 Mflops

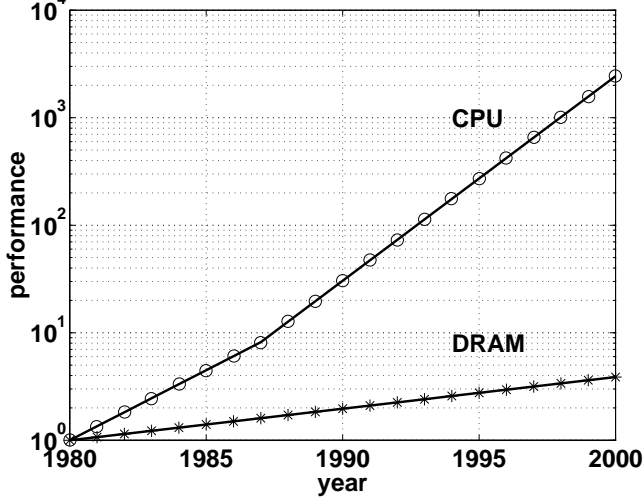
Addition and multiplication are pipelined. Division is not pipelined (so divides do not overlap) and takes 22 cycles for double precision.

$$\frac{167 \cdot 10^6 s^{-1}}{22} \approx 7.6 \cdot 10^6 / s$$

So, the answer is sometimes

- provided we have a suitable instruction mix and that
- we do not access memory too often

Performance of CPUs and DRAMs (Patterson & Hennessy)

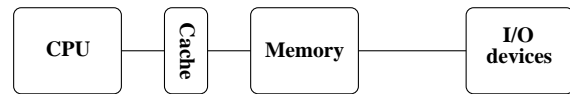


CPU: increase 1.35 improvement/year until 1986, and a 1.55 improvement/year thereafter.

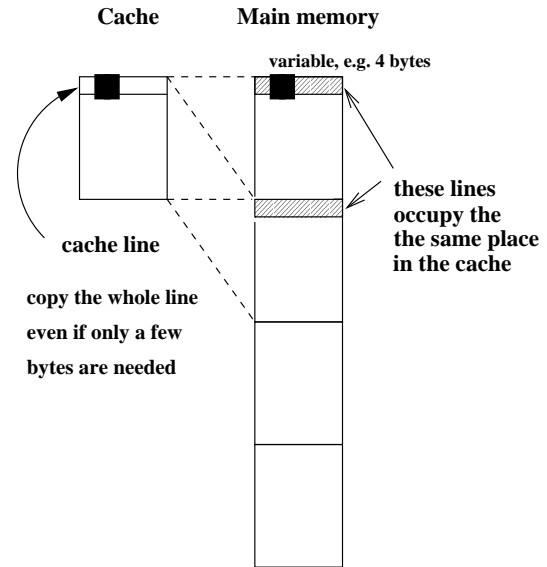
DRAM (dynamic random access memory), slow and cheap, 1.07 improvement/year.

Use SRAM (static random access memory) fast & expensive for cache.

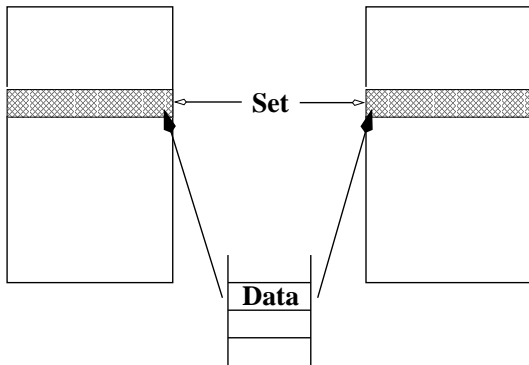
This is the simplest form of cache-construction.



The cache is a fast and small memory used for both instructions and data.

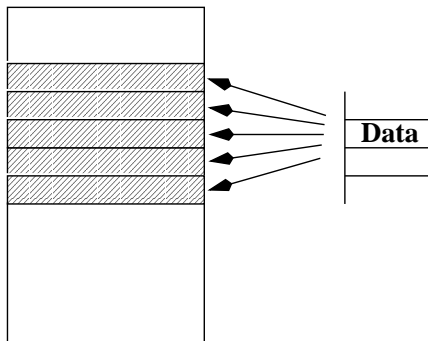


There are more general cache constructions. This is a two-way set associative cache:



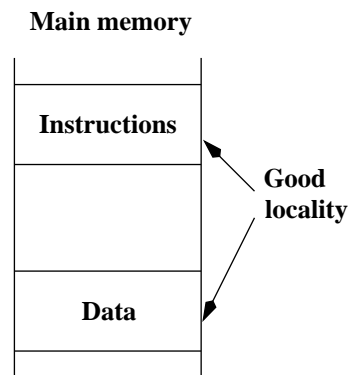
A direct mapped cache is one-way set associative.

In a fully associative cache data can be placed anywhere.



To use a cache efficiently locality is important.

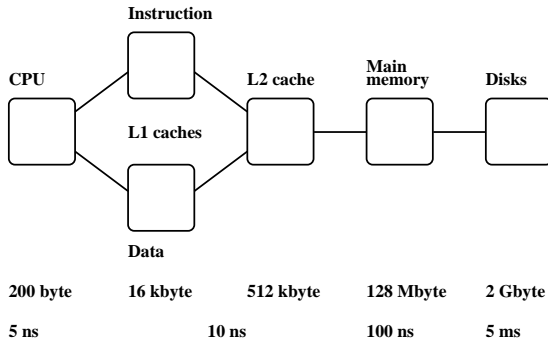
- instructions: small loops, for example
- data: use part of a matrix (blocking)



Not necessarily good locality together.

Make separate caches for data and instructions.

Can read instructions and data in parallel.



On the SGI: cycle time $1 / 195 \text{ MHz} \approx 5 \text{ ns}$.

- registers: 1 cycle
- L1 cache: 2 or 3 clock cycles
- L2 cache: 8-10 clock cycles
- main memory: ranges from 60-220 clock cycles depending on the number of router hops to the memory in which the data are stored

Memory hierarchy

Four-way superscalar RISC processor. It can fetch and decode four instructions per cycle to be run on its independent, pipelined execution units:

1. load store unit
2. two 64-bit integer ALUs
3. a 32-/64-bit pipelined floating point adder
4. a 32-/64-bit pipelined floating point multiplier

Asymmetric ALUs. Both +, - and logical operations. ALU 1 handles shifts, conditional branch and move instructions. ALU 2 integer multiplies and divides.

The FP adder handles +, -, | round, truncate, ceiling, floor, ... FP multiplier handles *, /, $\sqrt{\quad}$, ...

The two units can be chained together to perform multiply-add and multiply-subtract operations.

One roundoff when computing $a*b+c$.

Cache Architecture

On-chip L1 caches are 32 kB, two-way set associative.

Instruction cache uses a line size of 64 byte.

Data cache has a line size of 32 bytes.

Off-chip L2-cache is two-way set associative, may range in size from 512 kB to 16 MB. Line size 128 byte.

All caches use a least recently used replacement policy.

There is also a translation look-aside buffer (TLB) for holding mappings between virtual and physical addresses. This cache is fully associative and holds 64 entries.

RISC and elementary functions

A typical RISC-chip can handle +, *, -, / $\sqrt{\quad}$.

Elementary functions are done in software.

The Sun math library for double precision is available from:

<http://www.netlib.org/fdlibm/index.html>

(Freely Distributable Math Library.)

Prescribed for Java. <http://web3.javasoft.com/docs/books/jls/html/javalang.doc10.html#47547>

An example: $\sin x$. `sin` first calls `__ieee754_rem_pio2` and `__kernel_rem_pio2` if argument reduction is necessary. When x has been transformed to $[-\pi/4, \pi/4]$ `__kernel_sin` is called (listed in part below):

```

/* @(#)k_sin.c 1.3 95/01/18 */
/*
 * =====
 * Copyright (C) 1993 by Sun Microsystems,
 * Inc. All rights reserved.
 *
 * Developed at SunSoft, a Sun Microsystems, Inc. business.
 * Permission to use, copy, modify, and distribute this
 * software is freely granted, provided that this notice
 * is preserved.
 * =====
 */

/* __kernel_sin( x, y, iy)
 * kernel sin function on [-pi/4, pi/4], pi/4 ~ 0.7854
 * Input x is assumed to be bounded by ~pi/4 in magnitude.
 * Input y is the tail of x.
 * Input iy indicates whether y is 0. (if iy=0, y assume to be 0).
 */
/* Algorithm
 * 1. Since sin(-x) = -sin(x), we need only to consider
 *    positive x.
 * 2. if x < 2^-27 (hx<0x3e400000 0), return x with inexact
 *    if x!=0.

```

```

* 3. sin(x) is approximated by a polynomial of degree 13 on
*    [0,pi/4]
*
*          3          13
*    sin(x) ~ x + S1*x + ... + S6*x
*    where
*
*    |sin(x)          2      4      6      8      10      12      |   -58
*    |----- - (1+S1*x +S2*x +S3*x +S4*x +S5*x +S6*x ) | <= 2
*    | x
*
* 4. sin(x+y) = sin(x) + sin'(x')*y
*              ~ sin(x) + (1-x*x/2)*y
* For better accuracy, let
*
*          3      2      2      2      2
*    r = x *(S2+x *(S3+x *(S4+x *(S5+x *S6)))
*    then
*
*          3      2
*    sin(x) = x + (S1*x + (x *(r-y/2)+y)
*/

```

```

#include "fdlibm.h"

#ifdef __STDC__
static const double
#else
static double
#endif
half = 5.000000000000000000000000e-01, /* 0x3FE00000, 0x00000000 */
S1 = -1.6666666666666666324348e-01, /* 0xBFC55555, 0x55555549 */
S2 = 8.33333333332248946124e-03, /* 0x3F811111, 0x1110F8A6 */
S3 = -1.98412698298579493134e-04, /* 0xBF2A01A0, 0x19C161D5 */
S4 = 2.75573137070700676789e-06, /* 0x3EC71DE3, 0x57B1FE7D */
S5 = -2.50507602534068634195e-08, /* 0xBE5AE5E6, 0x8A2B9CEB */
S6 = 1.58969099521155010221e-10; /* 0x3DE5D93A, 0x5ACFD57C */

#ifdef __STDC__
double __kernel_sin(double x, double y, int iy)
#else
double __kernel_sin(x, y, iy)
double x,y; int iy; /* iy=0 if y is zero */
#endif
{
    double z,r,v;
    etc. etc.
}

```

x	Sun, IBM, Dec	HP	SGI	Linux, g77
1.0e+16	7.7969e-01	2.2068e-01	0.00E+00	7.7968E-01
2.0e+16	-9.7643e-01	-4.3049e-01	0.00E+00	-9.7644E-01
3.0e+16	4.4313e-01	-6.1830e-01	0.00E+00	4.4317E-01
4.0e+16	4.2148e-01	-7.7711e-01	0.00E+00	4.2143E-01
5.0e+16	-9.7097e-01	6.1421e+00	0.00E+00	-9.7095E-01
6.0e+16	7.9450e-01	-9.7186e-01	0.00E+00	7.9455E-01
7.0e+16	-2.4011e-02	1.1346e+01	0.00E+00	-2.4101E-02
8.0e+16	-7.6443e-01	-9.7769e-01	0.00E+00	-7.6436E-01
9.0e+16	9.8133e-01	2.0734e+01	0.00E+00	9.8135E-01
1.0e+17	-4.6453e-01	1.5483e+03	0.00E+00	-4.6464E-01
2.5e+17	-3.5511e-01	4.1442e+07	0.00E+00	-3.5481E-01

Note that all $10^p, p = 0, 1, \dots, 22$ can be stored exactly in IEEE double precision.

```
kallsup: libm-70 : UNRECOVERABLE Scalar COS() is called
with ABS(argument) greater than or equal to 2**25. Abort
```

```
selma: jwe0203i-e
In dsin(dx) or dcos(dx),dabs(dx).ge. 3.53d+15 error
occurs at g_dsin loc 000742c8 offset fffee4e8 (0,0)
g_dsin (f) at loc 00085de0 called from loc 00000378
in MAIN__
```

It is common to construct parallel computers by connecting fast workstation CPUs by a network.

Distributed memory computers

Memory is distributed on different boards and the programmer has to take this into account (using message passing, MPI, PVM).

Original SP2. Beowulf: Pentium & Linux.

<http://www.cacr.caltech.edu/beowulf/>

<http://www.beowulf.org/>

(Beowulf, the highest achievement of Old English literature; Scandinavian hero. According to Britannica.)

ASCI Option Red Supercomputer.

MP-LINPACK benchmark 1.34 Tflops, peak 1.8 Tflops.

9,216 Pentium Pro processors with 596 Gbytes of RAM,

640 disks, 1540 power supplies, 616 interconnection facility.

<http://developer.intel.com/technology/itj/q11998.htm>

(Distributed) shared memory computers

The memory resides on different CPU-boards, but the programmer sees one big memory. Program using OpenMP and multi-threading, for example.

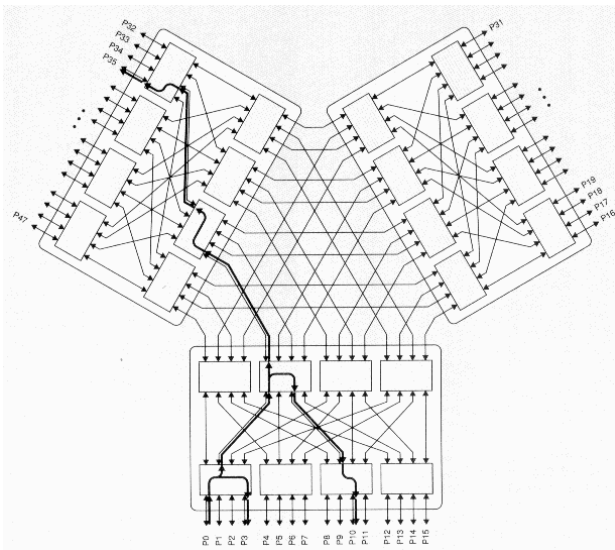
Some boards on Strindberg, SGI-Origin-machines,

“The ACL Nirvana Machine consists of 16 SGI Origin 2000s each with 128 250MHz R10K processors and 32GBytes of memory.”

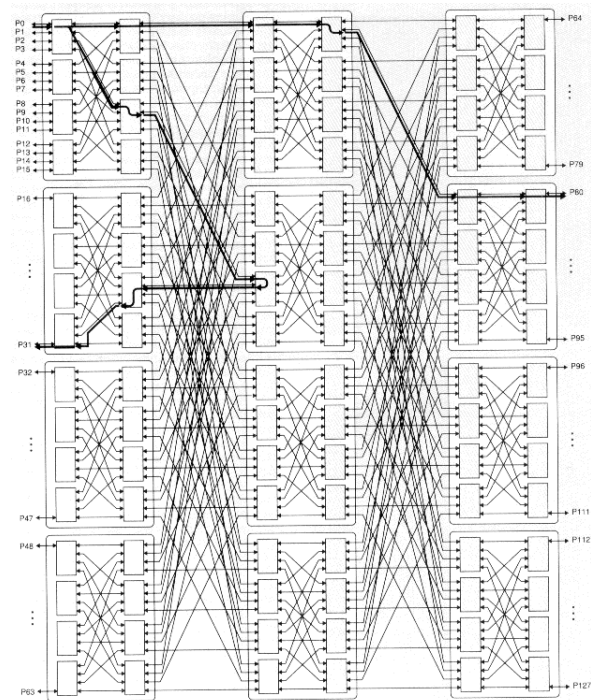
<http://www.acl.lanl.gov/news/releases/99-001.html>

This images shows a 48 processor SP2. Each “block” (a Vulcan switch chip) connects 8 inputs (one byte wide) to 8 outputs through a crossbar. The chip has a buffer which is used if there is contention.

If we communicate between CPUs on the same switch board the message may have to pass three Vulcan chips. If the CPUs are on different switch boards, four chips may be involved. For any pair of CPUs on different switch boards there are eight paths. Messages between CPUs are routed (shortest path + use links and switches in a balanced manner).

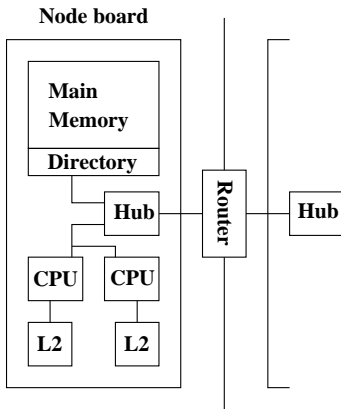


This images shows a 128 processor SP2. Note that we add communication hardware when constructing larger systems.

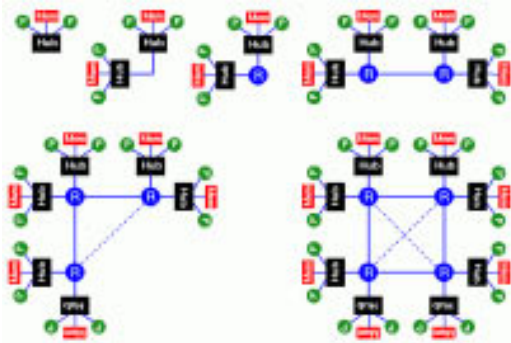


For more details: IBM Systems Journal, Vol. 34, No. 2, 1995.

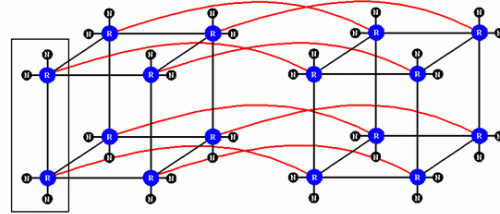
connected by a fast network.



The hub manages each processor's access to memory (both local and remote) and I/O. Local memory accesses can be done independently of each other. Accessing remote memory is more complicated and takes more time.



Hypercube configuration. When the system grows, add communication hardware for scalability.



Directly connect two 32-node systems via Craylink cables using the one free link on each router

The CPU always fetches and stores data in its cache. When the CPU refers to memory that is not present in the cache, there is a delay while a copy of the data is fetched from memory into the cache.

There can be as many independent copies of a memory location as there are CPUs in the system. If every CPU refers to the same memory address, every CPU's cache will have a copy. The "original" data has its home on one node and the directory, one the home-node, keeps track of which L2-caches have copies of data from memory.

But what if one CPU then modifies that location?

Cache coherence — how to ensure that all caches reflect the true state of memory.

For details:

"Performance Tuning Optimization for Origin2000 and Onyx2":
<http://techpubs.sgi.com/library/manuals/3000/007-3511-001/html/O2000Tuning.0.html>

NUMA - Non Uniform Memory Access, but:

- number of router hops grows as the 2-logarithm of number of CPUs
- even if data resides on a node far away, once it has been fetched to the local caches access is fast
- the programmer can influence the placement of data

For local memory accesses, the maximum latency is 313 nsec.

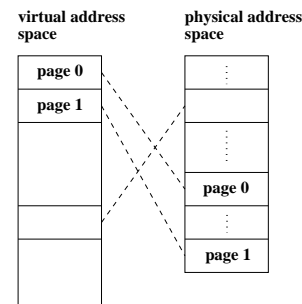
For a hub-to-hub direct connection (i.e., two-nodes)

the maximum latency is 497 nsec.

For larger configurations, the maximum latency grows approximately 100 nsec for each router hop.

Virtual memory

Use disk to "simulate" a larger memory. The virtual address space is divided into pages e.g. 4 kbytes. A virtual address is translated to the corresponding physical address by hardware; address translation.



A page is copied from disk to memory when an attempt is made to access it and it is not already present (page fault). When the main memory is full, pages must be stored on disk (e.g. the least recently used page since the previous page fault). Paging. (Swapping; moving entire processes between disk and memory.) Some advantages of virtual memory:

- allows large programs to run on little memory
- only used sections of programs need be present in memory
- several programs may reside in memory
- simplifies programming (e.g. large data structures where only a part is used)

Virtual memory requires locality (re-use of pages) to work well, or thrashing may occur.



Suppose a disk rotates 7200 rpm, so 120 revolutions/s.
We have to wait on average half a revolution (rotational delay) for the read/write head to be located in the correct sector (radius), which takes:

$$\frac{1}{240} \text{ s} \approx 4.2 \text{ ms}$$

A fast CPU can, at top speed, perform say 720 Mflops/s, so

$$\frac{720 \cdot 10^6}{240} = 3 \cdot 10^6$$

flops during this half revolution.

We have not considered seek time (moving the arm), transfer time (reading the data), controller time (getting the data to main memory).

Paging takes time!

33

- RISC - CISC; Reduced Instruction Set Computer
- pipelining, hazards: data dependencies and branches
- memory is slow: caches
- LOCALITY
- superscalar; parallel execution
- top speed
- shared memory, cache coherency
- virtual memory, disks are slow

34