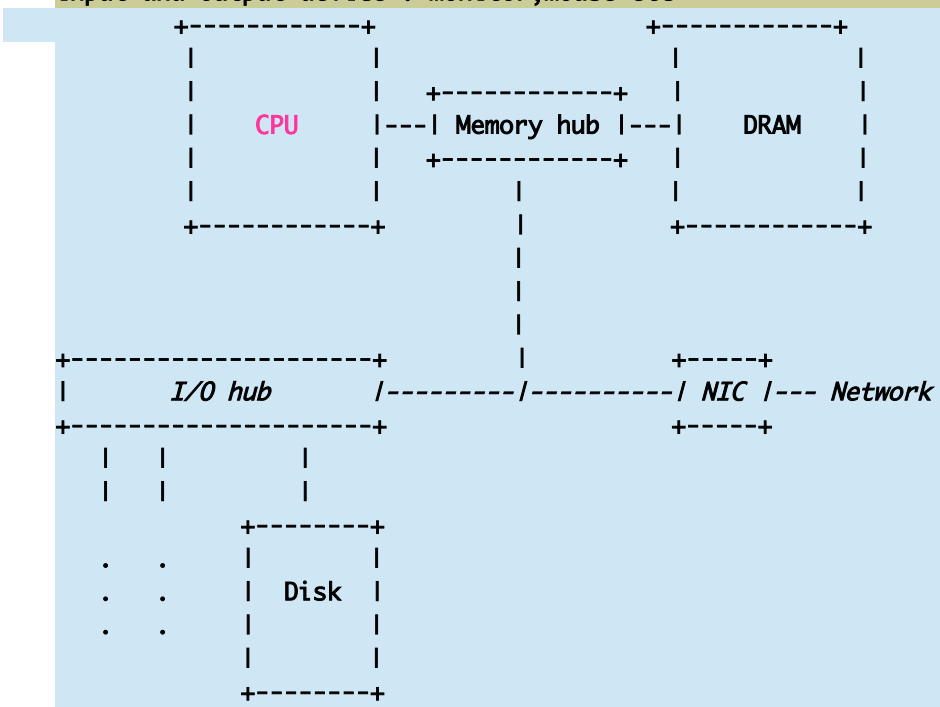
# Lecture 1  Computer Organization and Design

## General Introduction

*Uniprocessor (circa 2003):* A uniprocessor system is defined as a computer system that has a single central processing unit that is used to execute computer tasks.

CPU : receives and executes instructions
main memory : holds instructions that are about to be processed
secondary storage device : harddrive,DVD etc
input and output device : monitor,mouse etc




Every1 starts with 3 components of a computer: processor,memory,storage.  Some start with realistic interconnection network,which shows how signals are transmitted among 3 components.Main interconnection idea: while processor talks directly to memory, it must move data from storage to memory before any information is available to it.2nd idea:processor-memory communication must be very fast, while (indirect)processor-storage communication is necessarily much slower.

General,e.g,in larger computers,interconnect betn processor & memory bank is a network;when this network move data betn processor & memory at high data rate, we say
computer is blessed with _high-bandwidth interconnect, or simply is an
(uncommon) _high-bandwidth computer_.

major components of uniprocessor:
1)processor,
2)memory sys(DRAM),
3)storage sys (single disk),
3)other I/O peripheral devices(not shown),
4)wires(here,highest-level interconnect) that allow these components to communicate.

Elaboration: Many low-end computers benefit from simplicity & economy
of having a single shared bus that connects all system components.In contrast,
high-end, as well as more modern,computers generally use point-to-point
connectivity for greater speed and bandwidth.bus is a set ofwires that functions as a shared

communication link, and connects multiple
subsystems.I generally ignore buses and assume that all important
communication takes place over a point-to-point interconnection network.
As such, I call every communication link a wire,Collectively forming interconnect.
Note 1: Obviously, there is interconnect at everyspace scale.
Note 2: I/O devices are distinguished by whether they serve as auxilliary storage devices or as human-facing input/output devices.

high-bandwidth interconnect betn processor & memory.low-bandwidth interconnect (here, actual bus) shared among disk(storage device) and other I/O peripheral devices.The number and diversity of these I/O devices forces this bus organization.The network-interface chip is high bandwidth,so it connects directly to high-bandwidth interconnect.The chip in the middle of the high-bandwidth interconnect is the memory controller hub,which has been integrated onto the processor *die* in more modern processors. The bus chip leading to the low-bandwidth interconnect,i.e.,to set of wires connecting the zoo of I/O devices, is the I/O controller hub.Finally,the network interface chip_ connects the computer to a computer network of some kind.

Note:Computer optimized to process Big Data requires a much higher-quality interconnect to its storage.

many classes of comp(laptop,desktop,server,supercomputer,warehouse-scale data center,embedded, etc.). Are all computers basically same?(No! A CPU is not a GPU is not a DSP.Example,let's examine whether a server built from Intel Xeon chips is really all that different from one built from, IBM Power8 chips). One obvious difference is that large computers are aggregates of uniprocessors,& thus are called multiprocessors.Are all multiprocessors basically same?Are all uniprocessors basically same?

>*From 1979 to 2003, so-called "killer micros" (RISC microprocessors)* basically drove competing processor designs out of business.So,almost all computers today are powered by one or more killer micros.MOBILE has a cool killer micro,probably ARM.A server has one or more hot killer micros,probably Intel Xeons. These killer micros got steadily better(increasing performance) up until 2003.However,computers with same processor could still differ by the quality of their memory system or interconnect.

Users want to solve problems algorithmically by computer.i.e. write programs.Before designing computers, vendors ask:What does customer want? What kinds of programs does he wish to write?This affects design & optimization.

Computer design,like all engineering design,is a series of trade-offs.Now,are all programs same?NO! Programs differ in their memory-accessing patterns.like,programs may differ depending on whether they use predominantly short-range or long-range memory accessing.Programs can also differ in their arithmetic intensity,degree of data reuse,predictability of their memory-accessing patterns.
And all trivial parallel programs are embarrassingly local.These differences among programs essentially divide the space of computer users into different markets.

Computer vendors optimize designs to needs of programs of the largest class of users.To this day, other user classes complain and continue to be frustrated, but there isn't a whole lot they can do about it. Existing computers are essentially mass-market computers or at least computers constructed by aggregating mass-market components,such as killer micros (RISC microprocessors).

Although computers on offer from major hardware vendors are remarkably architecturally similar,none of them are general-purpose computer because each has been optimized to provide good performance only for a certain class of applications.

2003,killer micros met their Waterloo( crushing defeat):basically chips were getting too hot.Moore's law hadn't been repealed,but cooling & energy supply problems meant that business as usual had come to an abrupt end.Intel surrendered in 2004.Vendors adopted a new game plan:Instead of putting 1 hot, high-

performance processor on a chip,they put many cool,low-performance processors on a single "processor" chip.called multicore.However,even in 2016,there is no general agreement about the best way to design a multicore chip.Progress has been slow(Intel adds two cores per generation;this is linear,not exponential).The big question is,what memory system will allow us to increase no. Of cores we can profitably put on a single multiprocessor chip?This is bcz inadequate memory bandwidth may cancel benefit of the increased arithmetic capability that multicore provides.If you are puzzled,don't worry; trade-off betn more cores but less bandwidth per core is a very complicated subject.

Btw,when no. of cores becums sufficiently large,we hav to b retrained to master parallel programming.

Is there a clear distinction betn architecture & organization?same question as,is there a clear distinction betn an architecture & its implementation?Think of a computer as a black box.Vendor has given a contract that specifies externally visible behavior of box. Perhaps contract describes the behavior of every machine instruction,and also way the machine has been optimized.(In a programming language, such a black box would be called an "abstract data type", or a "module").

Using this contract,write & optimize programs.Resulting object code is client of this architecture.If contract,which specifies the hardware/software interface, has been properly written,then vendor can make arbitrary improvements to his implementation,and ur old fast programs will still be faster than ur old slow programs.The contract constrains both parties.You agree to rely only on it while developing programs,while vendor agrees to support the contract even if he changes implementation.Contracts are not made in heaven:after a while,you & the vendor may agree that a new contract(i.e.,new architecture) is necessary.

In one(possibly narrow)sense,killer micros temporarily killed evolution of architecture.If you look at instruction-set architectures(ISAs) of various machines,u will see that Intel and AMD are still peddling x86,and that the pure RISC processors have more or less identical ISAs.In 2016,instruction-set design is a dead horse.Note: It is  dead horse as far as the design space of conventional processors is concerned. However,in the design spaces of multicore and GPUs, it is still worth considering.But maybe,continuing evolution of multicore,together with continued evolution of GPUs,will come together to create a viable new architecture for general-purpose computing,which can only be general-purpose parallel computing. Many smart people suggest this.I personally don't see much progress towards building a computer that could execute massively parallel code with wholly unpredictable memory-accessing patterns.

We spoke of architecture as contract specifying hardware/software interface.In reality,computer is a tower of interfaces.Going down,we have:architecture, organization (a/k/a microarchitecture), and hardware(e.g.,logic design).But consider going up.A high-level programming language is an interface. You write program.compiler translates it into machine instructions (object code).computer executes object code.A (low-level) assembly language is another interface (practically extinct).You write program.An assembler translates it into object code.The computer executes object code.

Both operating sys & runtime sys are actors in this tower of interfaces.We will consider neither.An operating sys might handle file operations for ur program.A runtime might print an intelligible error message if you divide by zero.Runtime sys are increasingly merging with compilers,but this is outside our scope.

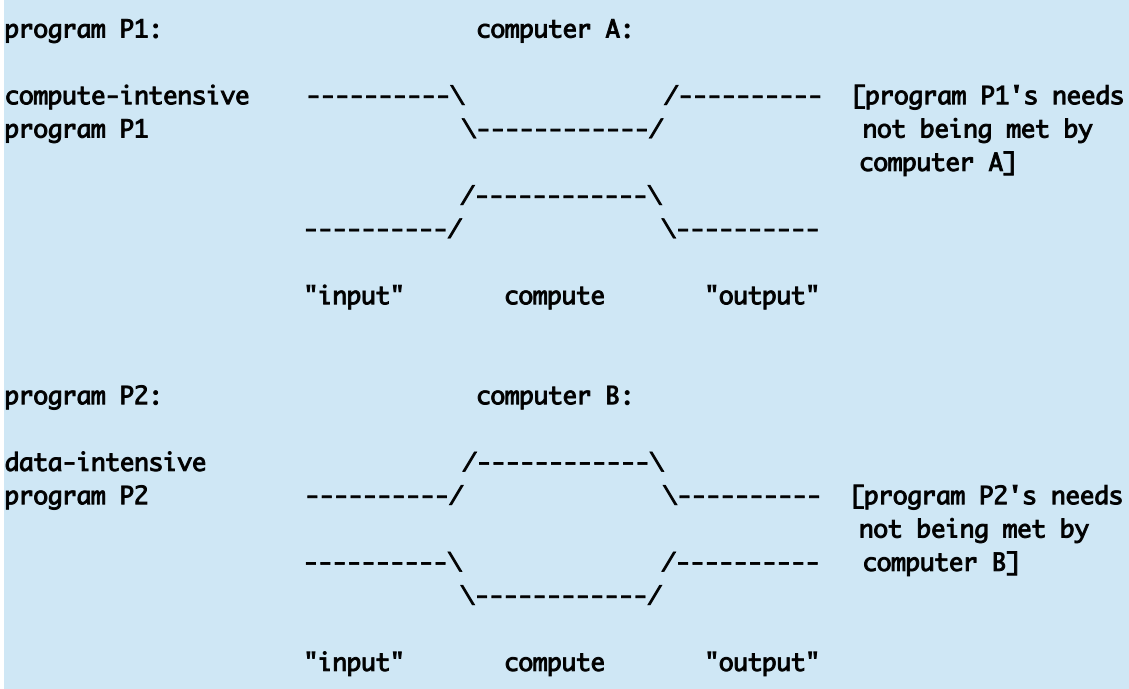
*review some basic facts:*

Computer can be many things. Ex: embedded,laptop,workstation,server,mainframe,distributed system, parallel computer,supercomputer.

*Differences?*Many: no. of users,amount of software,technical support &storage,main-memory size,memory bandwidth, I/O bandwidth, network bandwidth, reliability,power consumption, performance, price, ... .

*Similarities?*many.Processor (CPU) does the arithmetic.It reads and writes data from/to memory. It also reads and writes data from/to peripheral storage devices, via the memory. This latter activity is called I/O.(Reading and writing from human-interface input/output devices is also called I/O, but we will normally ignore this). Again, programs differ in what they require from a computer. One important example of this is the distinction between compute-intensive and data-intensive programs. A _compute-intensive_ program does asymptotically more processing than data movement (e.g., performing many arithmetic operations for each word transferred). In contrast, a _data-intensive_ program does asymptotically less processing than data movement (e.g., performing one or fewer arithmetic operations for each word transferred). Therefore, a compute-intensive program will suffer from a bottleneck if the computer has inadequate processing resources, relatively speaking. Similarly, a data-intensive program will suffer from a bottleneck if the computer has inadequate data-movement resources, relatively speaking.

A pipeline analogy shows that imbalance between processing power and data-movement capabilities can lead to a performance bottleneck. The analogy is symmetric, but practically one case concerns us more.




Here, computer A is nice to programs with low arithmetic intensity, but computer B only performs well on programs with high arithmetic intensity. We could fix the imbalance shown by running program P1 on computer B and program P2 on computer A. By the way, GPUs have some characteristics of computer A, but all CPUs are very much like computer B.

Note that there are _two_ possible sources of imbalance, viz., either 1) between _compute bandwidth_ and _memory bandwidth_, or 2) between _compute bandwidth_ and _I/O_ bandwidth (i.e., storage bandwidth). Or both.

Today, most computers are much better at computation than they are at communication, which means that low-bandwidth communication can be the principle performance bottleneck. More programs than you might imagine are data intensive

in the sense that their _arithmetic intensity_ is low. Only highly compute-intensive programs are well matched to today's computers. To fully understand this last statement, we need the concept of a program's _working set_, which we will only introduce much later.

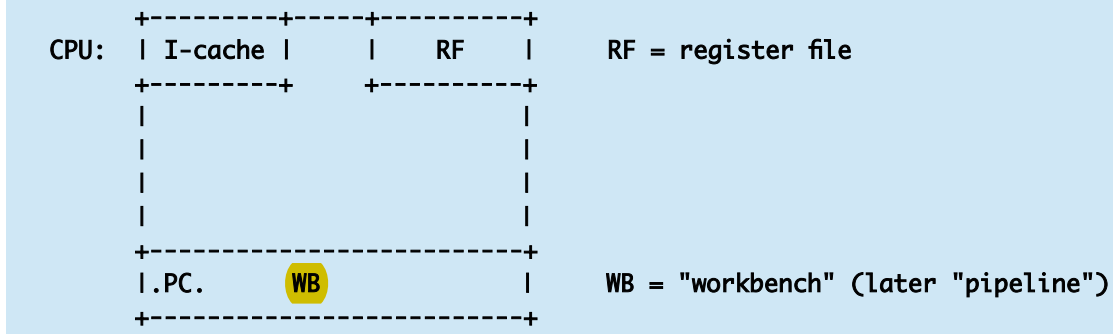
In fact, computation today is limited by communication, not arithmetic. Floating-point computation is essentially free, in time and energy. In contrast, off-chip bandwidth is limited to a few GWs/s and each word transferred consumes enormous energy. Feeding the FPUs with data is expensive, not the FPUs themselves. Since communication capability is so limited, we should try to i) exploit locality however possible to reduce our need for communications bandwidth, and ii) tolerate memory latency through some form of parallelism to keep the limited bandwidth resources in our high-latency memory systems busy. That is, we want to extract the highest delivered operand bandwidth possible from these limited bandwidth resources.

What are the figures of merit for DRAM memory (I tend to always answer "memory bandwidth", but that's not the whole story). As DRAM improves, memory bandwidth improves by at least the square of the improvement in memory latency. So, when do we care about _memory latency_ and when do we care about _memory bandwidth_? Imagine a processor that is able to sustain a high memory-request bandwidth of 'b' memory requests per processor cycle. This processor would benefit from a DRAM memory that is able to sustain a high memory-reply bandwidth of 'b' memory replies per processor cycle. In contrast, imagine a processor that issues a single memory request, and must wait for a reply before being able to issue its next memory request. This processor would benefit from a DRAM memory that is able to reply quickly to a single memory request; the time to do so is called the _memory latency_. CPUs, but not GPUs, have large, deep memory hierarchies to try to keep their processors busy. Everything works well when the programs have sufficient _arithmetic intensity_. But the whole story is very, very complicated. Fact: Neither CPUs nor GPUs are very successful at running programs with irregular, unpredictable memory accessing and massive parallelism opportunities.

Recall that a program's data is stored in the memory but can only be processed in the CPU. A program might make a memory reference to retrieve some operand and then be forced to wait for the operand to arrive. If the whole processor sits idle while waiting, this is not good. (Not utilizing an arithmetic functional unit is less tragic, because of the low cost of the unit). Some processors are very good at maintaining processor activity, including asking memory for more data, even if some of the programs or threads they are running are waiting for data to arrive. This rare (and good) form of processor is called _a latency-tolerant processor_ because its utilization does not degrade in the face of memory latency. Other processors---in fact, most processors---are not very good at doing this, so they depend on having _latency-avoidance mechanisms_, i.e., they try to keep operands that will be used in the near future close to the processor. This is the main justification for processor caches. Note: CPUs are latency intolerant because of their monothreading; GPUs are partly latency tolerant because of their limited multithreading.

Killer micros only perform well when the program's memory-accessing pattern can be exploited by the memory hierarchy, of which caches, at all levels, are the most important part. Lower-level caches are relevant to the storage-accessing pattern.

Let's fill in a few components of the processor, so that we can follow the
execution of one machine instruction.




The processor's "workbench" (the active part of the datapath) is a pipeline.
One relevant component (the so-called "ALU") is not pictured.

| PC reg | instruction cache | register file | ALU | register file |
|---|---|---|---|---|

Consider executing the machine instruction 'mul.d f0,f2,f4', where 'f0', 'f2',
and 'f4' are (floating-point) processor _registers_. Another processor register,
'PC' (program counter) points---or pointed---to this instruction, i.e., the
multiply instruction, which must be fetched before it can be executed.

Note: At the same time we fetch the multiply instruction, we update the PC to
point to the next instruction that will be fetched.

When we execute the multiply instruction, we first retrieve 'f2' and 'f4' from
the register file. We deliver both to the ALU. We then take the ALU's output
and put it back in the register file (in register 'f0'). Think of the
instruction cache as just a bag of machine instructions that can be fetched
using their address. The names on top are therefore names of resources the
workbench uses.

In this particular instruction, there are no _memory references_, i.e., no
_loads_ or _stores_. In other words, we have discussed a _register-register
instruction_.

Thinking about this example, we see a new distinction. The register file
contains _externally visible_, or _ISA_, registers, i.e., ones that can appear
in assembly-language statements. But, if we retrieve a value from an ISA
register, and bring it into the datapath, we need some place to put it. For
this reason, every processor contains a large collection of _externally
invisible_, or _nonISA_, registers. Most of these are inside the datapath.
The most important nonISA register is the program counter (PC). (Elaboration:
The ISA defines much of the processor's _architecture_, i.e., its externally
visible behavior. In contrast, the PC is part of the architecture's
_implementation_. A fundamental idea in computer science is that the
implementation of an architecture may change rapidly, but it should implement
the same architecture for much larger stretches of time).

von Neumann model

---

This brings us to the famous von Neumann computational model. It started with the idea of a stored-program computer. In advanced language, the von Neumann _machine model_ states that there must be pure control-flow scheduling of individual threads. It also states that processors should execute single threads.

An introduction to computer organization must explain the von Neumann machine model. The slogan that captures its essence is: "Every processor shall have precisely one program counter". Let's develop this idea. In a stored-program computer, we need a memory unit to store the instructions of a program and supply instructions to the processor given a memory address. The memory unit also stores the program's data. von Neumann machines have a single _program counter_ (PC) per processor. The PC is an internal register that holds the address of the next instruction to be fetched. To implement the _fetch-execute cycle_, we need an adder to increment the PC to contain the address of the next instruction.

Consider a program segment that has given rise to some straight-line object code. The dynamic sequence of machine instructions in an execution of the object code is called the _program order_. ("Straight-line" means no branches). In straight-line code, the fetch-execute cycle steps through the sequence of machine instructions in program order. This is the simplest form of _control-flow_ scheduling of instructions. The general form includes branches. Of course, we still have program order when we include branches. It is a slightly more interesting order because of the presence of _loops_ and _if statements_.

Thus, abstractly, program order is the dynamic control-flow sequence of machine instructions in some execution of the object code.

This has enormous performance consequences. Consider a floating-point multiply somewhere in the sequence. Presumably, loads appear earlier in the program to bring the two operands of the multiply from memory. Suppose they haven't arrived yet. In that case, the multiply cannot start execution. Since we are moving through the program in program order, the whole program blocks. Since the processor is running precisely one program, the whole processor blocks. This is not good for performance (we say the processor _stalls_).

You may ask if the processor couldn't simply switch to another program when the program it is currently running blocks. That depends on the context. If the first program will be blocked for a very long time, then the cost of _context switching_ will be worth it. (Example: a program that blocks for disk I/O). However, if the program will be blocked for a much shorter time, then it makes sense just to stall the processor. Modern CPUs spin wait.

The von Neumann machine model also has had enormous programming consequences.

In the von Neumann model, we store the program and the data in the memory (programs are like data in being representable by bit patterns). We fetch instructions and data from the memory, perform computation in the processor, and push the result back to memory. Again, the central idea of the von Neumann model is that each processor should have precisely _one_ program

counter.  A von Neumann computer is a sequential computer.  And von Neumann computation becomes "rearranging the furniture in memory".

In high-level languages, this computational model gave rise to the notion of a _variable_ (i.e., a named memory location whose value can be changed).  A variable is of course a _multiple-assignment variable_.

Computing then becomes scheduling values into variables, i.e., deciding in which order which values will be "assigned" to variables.  This is the basic programming abstraction behind all von Neumann computing.  This idea is called the von Neumann _programming model_.

In truth, we can keep program counters and throw variables---for the most part---in the garbage can.

Basics (rehash)
_____

The processor die contains the processor, the L1$, and the L2$.  A separate chip used to hold the L3$.  (In 2016, it is on chip).  In many designs, the processor talks to the "bridge" chip.  That chip talks directly to memory, directly to the NIC, and directly to the I/O bus, off of which hang the I/O devices.  I/O buses are SLOW.

Consider a floating-point multiplier, which we can think of as part of the datapath.  If you give a FP multiplier two real numbers, it will multiply them together and return their product.

Two questions arise.

 - How fast is the multiplier?

 - How easy is it to keep it continually supplied with operand data?

We could say that computers consist of a) places to put data, b) operators that move data, c) functional units that compute data, and d) wires along which to move data.

Computer designs are not immutable.  The relative cost and speed of things change.  For example, even on the (small) space scale of a processor chip, wire delay is starting to dominate transistor delay.

When the relative values of cost parameters change, what was a good design may become a bad design (and vice versa).

For example, traditional designs assume it is basically free to move data from anywhere on a processor chip to anywhere else on the same chip.

1) General-purpose register machines may be divided into two families:
   a) load-store architectures, including notably RISC machines, and
   b) CISC architectures, including certainly the (extinct) DEC Vax and IBM System/360, but arguably also the Intel 80x86, whose registers are only _somewhat_ general purpose.  The debate between RISC and CISC was originally about what percent of the processor chip should

be dedicated to hardwired control. RISC vs. CISC isn't important these days (RISC won), and all computers are load-store architectures, even when they pretend otherwise.

2) Surprisingly, the various interconnects---at all scales---are the most important components of a computer. A) In a large-scale parallel computer, global system interconnect links perhaps thousands of _nodes_, each containing one or more processors and (local) memory. B) In a node, intranode interconnect links processors and local memory, as well as providing a path to (external) I/O devices and the global system _interconnection network_. C) In a processor, intraprocessor interconnect links the control unit and the datapath. In a many-core processor, interconnect is used to link cores and caches. D) In the datapath, more fine-grained interconnect links the registers and the ALU (i.e., the arithmetic and logical functional units). And so on.

   Interconnect is so important because all computations must engage in communication, at whatever scale. Programs differ in whether they engage in short-range or long-range communication. The interconnect may or may not have the capacity to move whatever data needs to be moved at the space scale in question. Communication determines power and performance (the former is bad; the latter is good).

3) Node organization includes the arrangement of the following components: a) the processor chip, with its on-chip caches; b) historically, the L3$ was off chip; c) the intranode (north) bridge chip; d) the DRAM memory chips; and e) the network-interface chip. Each node embodies the upper levels of the memory hierarchy (registers, cache, local memory, ...). Note that the storage hierarchy is both a latency hierarchy and a bandwidth hierarchy.

4) The ISA defines the assembly language, the instruction format, the addressing modes, and the programming model. Well, the ISA determines the _functional_ aspects of the programming model, but not the _performance_ aspects.

5) We studied a fragment of MIPS code. We had 64-bit floating-point registers (but only 32-bit words). We had a memory array of floating-point numbers. We used 'r1' as an address register. We saw a load instruction, an add instruction, a store instruction, and an integer-subtract instruction used to change 'r1' to point to the next floating-point number. A conditional branch sent us back to the top of the loop as long as there were more floating-point numbers to process.

appendix to first lecture
_____ 1 W = 64 bits

Bandwidth, latency, and friends in a typical memory hierarchy
_____

Level BW (W/cyc) Latency (cyc) Capacity (W) Granularity (W)

```
Registers   12               1              32             1
L1 Cache     2               3              2K             1
L2 Cache     1               8             16K            16
L3 Cache     0.5            20            512K            16
DRAM         0.25          200              1G            16
Other Node   0.001 - 0.05   500 - 10,000     1T            16 - 512


MIPS code
---------

loop: l.d    f0,0(r1)    ; f0 := a[j]
      add.d  f4,f0,f2    ; f4 := a[j] + c
      s.d    f4,0(r1)    ; a[j] := f4
      daddiu r1,r1,#-8   ; j := j - 1
      bne    r1,r2,loop  ; if r1 <> r2 then repeat

mini-MIPS
---------

loop: l.d    f0,Mem[r1:8]  ; f0 := a[j]
      add.d  f4,f0,f2      ; f4 := a[j] + c
      s.d    f4,Mem[r1:8]  ; a[j] := f4
      sub    r1,r1,#8      ; j := j - 1
      bne    r1,r2,loop    ; if r1 <> r2 then repeat

Floating-point array in memory:              +---+---+---+---+---+
                                             |   |   |   |   |   |
                                             |   |   |   |   |   |
                                             +---+---+---+---+---+
                                         ^                   ^
                                         |                <-|
                                         r2                 r1

endarticle


threads
```