Lecture 1 Computer Organization and Design   DKP/Fall 2020

____

General Introduction

____

A picture of a (standalone) uniprocessor (circa 2003):

```
         +------------+                    +------------+
         |            |                    |            |
         |            |   +------------+   |            |
         |    CPU     |---| Memory hub |---|    DRAM    |
         |            |   +------------+   |            |
         |            |        |\          |            |
         +------------+        | \         +------------+
                               |  \
                               |   \
+---------------------+        |    \      +-----+
|       I/O hub       |--------|     \----| NIC |--- Network
+---------------------+                   +-----+
   |   |         |
   |   |         |
                +--------+
   .   .        |        |
   .   .        |  Disk  |
   .   .        |        |
   .   .        |        |
                |        |
                +--------+
```

Consider three familiar components of a computer: the processor, the memory, and the storage. Also, consider a realistic _interconnection network_, which shows how signals are transmitted among these three components. (This interconnection network is the _fourth_ component). The processor can talk directly to the memory, but data must be moved from the storage to the memory before the processor can access them. Processor-memory communication, as well as processor-network communication, must be fast. However, processor-storage communication is, of necessity, slow. Question: Where should coprocessors (accelerators) fit into this scheme?

As shown, the four major components of a uniprocessor are:

1) the processor,

2) the memory system (here, DRAM),

3) the storage system (here, a single disk)

---the other peripheral devices are not shown, and

4) the wires (here, the highest-level interconnect) that allow the other components to communicate.

In larger, multiprocessor computers, the global system interconnect between processors and memories is an elaborate network. To achieve a high peak data-transfer rate between its processors and memories, a computer needs a _high-bandwidth_ communication fabric (interconnection network). It also needs a high-bandwidth memory system that can feed data values rapidly into the communication fabric. A processor that is able, in the general case, to sustain a significant fraction of this network/memory bandwidth---to keep itself supplied with useful data---is called a _latency-tolerant_ processor. At present, there are few, if any, of these. The idea of implementing processors with general-purpose latency tolerance---for a wide range of long-latency events---is only slowly being recognized by the computer industry as a scalable path to higher performance.

Modern computers use direct point-to-point links between compatible devices for greater speed and bandwidth. These links are called _wires_. In contrast, a _bus_ is a _shared_ communication link (party line) that connects multiple subsystems.

The point-to-point interconnect between processors and memories can be designed to provide high peak bandwidth. The switch in the point-to-point network (the so-called _memory controller hub_) is not a bottleneck. In contrast, the switch in the bus network (the so-called _I/O controller hub_) is literally a bus and is a _terrible_ bottleneck. The bus interconnect between a processor and its peripherals necessarily has low bandwidth because of the number and diversity of the I/O devices. For performance reasons, the network-interface chip is on the point-to-point network.

There are many kinds of computer (embedded controllers, laptops, desktops, servers, high-performance computers, Google-style constellations, etc.). What if I suggested that all processor building blocks are basically the same? (What?! Of course not. A CPU is not a GPU, nor a DSP, nor an FPGA, and certainly not a VPU!). But let's explore this wild idea. Clearly, large computers are _aggregates_ of uniprocessors, and hence are called _multiprocessors_, although a better name might be _multicomputers_. But is a large-scale, high-end computer built from Intel Xeon chips all that different from a large-scale, high-end server built from IBM Power9 chips?

There is an astonishing commonality here: across the planet, almost all high-end CPU processors, _including_ their embedded cores, are minor variants of the same pipelined RISC 2.0 processor microarchitecture.

Update 1: In 2016, Chinese engineers built a large-scale computer with a peak performance of 125 PFs/s, 10 M cores, a power efficiency of 6 GFs/s/W, 1.3 PBs of memory, and a power rating of 15.4 MWs. This remarkable achievement was made possible by _rejecting_ all of the optimizations used in the complex RISC 2.0 pipelines that first saw the light of day in the 1990s.

Update 2: In 2020, Japanese engineers built a large-scale computer with a peak performance of 500 PFs/s, 7.6 M cores, a power efficiency of 14.7 GFs/s/W, 4.85 PBs of memory, and a power rating of 28.3 MWs. Among the innovations that made this possible is the Fujitsu 48-core ARM A64FX CPU, which has no real need for accelerators, but could add them. In my opinion, the Fugaku supercomputer is the first large computer to take nearby-memory bandwidth seriously.

Full disclosure: The A64FX retains some of the RISC 2.0 ideas.

I may introduce the RISC 2.0 dynamic-scheduling canon (in-order dispatch, out-of-order issue, register renaming, branch prediction, etc.) that industry converged to after 1990, but began to doubt in 2005. RISC 2.0 processors are normally called OOO superscalar processors.

>From 1979 to 2003, these 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. A cellphone has a cool killer micro, probably an ARM. A server has one or more hot killer micros, probably Intel Xeons. These killer micros got steadily better (had steadily increasing performance) up until 2003. Of course, computers with the same processor could still differ in the quality of their memory system or the quality of their interconnect.

Users want to solve problems algorithmically by computer. To do this, they must write programs. Before designing computers, hardware vendors ask: What do my best customers want? What _kinds_ of programs do they wish to write?

This affects design and optimization.

Computer design, like all engineering design, is a series of trade-offs. Now, are all programs basically the same? Absolutely not! To start with, programs differ in their memory-accessing patterns.

Example: One program never touches memory and one program always touches memory. More generally, programs can be quite different depending on whether they use predominantly _short-range_ or _long-range_ communication. Programs can differ in their _arithmetic intensity_. They can differ in their degree of _data reuse_.

They can differ in their _memory stride_. They can differ in the _predictability_ of their memory-accessing patterns. And so on. All together, these differences among programs essentially divide the space of computer users into different _markets_. (And this is only their memory differences!).

In the parallel world, there is a division between _task-parallel_ and _data-parallel_ programs. Issue: To what extent are the threads, coming from program decomposition, independent, i.e., to what extent do they communicate and synchronize? When they do neither, we call the programs they come from _embarrassingly localizable_. GPUs have been optimized to run embarrassingly localizable data-parallel programs. While GPUs can do data parallel fast, only CPUs can do task parallel fast.

Computer vendors naturally optimize their designs to make them match the needs of the 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, i.e., computers constructed by aggregating mass-market components, such as killer micros (RISC microprocessors).

Fact: computers on offer from major hardware vendors are remarkably architecturally similar. Fact: none of them deserves to be called a general-purpose computer because each has been optimized to provide good performance only for some _restricted_ class of applications.

In 2003, killer micros met their first Waterloo: basically, chips were getting too hot. Moore's law hadn't been repealed, but cooling and 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 _one_ hot, high-performance processor on a chip, vendors put _many_ cool, low-performance processors on a single "processor" chip. This organization is called _multicore_. However, even in 2020, there is no consensus 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 ramp up the _number_ of cores we can profitably put on a single multiprocessor chip? This is because inadequate memory bandwidth may cancel the benefit of the increased arithmetic capability that multicore provides. Clearly, core area and core power efficiency are also major factors in multicore scalability.

By the way, are scalable multicore and RISC 2.0 basically incompatible? I strongly believe so. But multicore disappointment is multifactorial.

But see the two updates above. Neither machine is standard RISC 2.0.

Other big questions: Are the pins (required for off-chip I/O) and caches (including cache coherence) a stranglehold on the future effectiveness of multicore? Also, do we require new programming models (and languages) and new system software to be able to exploit multicore efficiently? Finally, is multicore rebranding the CPU as a data-parallel device?

By the way, if and when the number of cores becomes sufficiently large, all of you have to be retrained as parallel programmers. Unfortunately, we don't yet know how to teach this! Still, some of us think that, in the future, all programmers will _only_ write parallel programs. In 2020, merchant multicore seems to have stalled---it's hard to tell, and our efforts to teach parallel programming are too embarrassing for words.

Recall that any large parallel computer is necessarily a multiprocessor or multicomputer built using multicore processors. I dream of an accessible,

composable parallel-programming model that would span the entire range

from multicore uniprocessors to the largest shared-memory multiprocessors.

Is there a clear distinction between _architecture_ and _organization_? This is the same question as, is there a clear distinction between an _architecture_ and its _implementation_? Think of a computer as a black box. The vendor has given you a _contract_ that specifies the externally visible behavior of this box. Perhaps the contract describes the behavior of every machine instruction, and also describes the 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, you write and optimize programs. The resulting object code is the _client_ of this architecture. If the contract, which specifies the hardware/software interface, has been properly written, then the vendor can make arbitrary improvements to his implementation, and your old fast programs will still be faster than your old slow programs. The contract constrains both parties. You agree to rely _only_ on it while developing your programs, while the vendor agrees to support the contract even if he changes the implementation. Contracts are not made in heaven: after a while, you and the vendor may agree that a new contract (i.e., a new architecture) is necessary.

In one sense, evolving killer micros temporarily killed the _general_ evolution of architectural diversity. If you look at the instruction-set architectures (ISAs) of various machines, you will see that Intel and AMD are still peddling x86, and that the pure RISC processors have more or less identical ISAs. However, it is wrong to say that instruction-set design is a dead horse. The recent RISC-V ISA has reopened the topic.

As well, in the design spaces of multicore and GPUs, it is still worth considering. Of course, the real problem here is the dominance of utterly similar RISC implementations of conventional CPUs across the computer industry. Sun (now Oracle) is an exception. And GPUs are only coprocessors.

What technological breakthrough, what architectural innovation, will restart general-purpose _parallel_ computing? I've lost faith in the alleged benefits of CPU/GPU convergence, although this would be a good thing. For now, I don't see much progress towards building a computer that could execute massively parallel code with wholly unpredictable memory-accessing patterns. By the way, this is the major potential bottleneck of GPU coprocessors. Salvation will only come with radically new processor microarchictures, radically new communication fabrics, and radically new main memories. Computers are a package deal.

Suggestions:

1) Use high-degree multithreading to design cores with _strong_ latency tolerance.

2) Design pure-optical communication fabrics using optoelectronic devices to connect chips and fabric.

We spoke of _architecture_ as the contract specifying the hardware/software interface. In reality, a 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 your program. A compiler translates it into machine instructions (object code). The computer executes the object code. A (low-level) assembly language is another interface (practically extinct). You write your program. An assembler translates it into object code. The computer executes the object code.

Both the operating system and the runtime system are actors in this tower of interfaces. Runtime systems are increasingly merging with compilers, but this is outside our scope. We will consider neither.

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.

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 more data movement than processing (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 relative to its bandwidth capabilities. Similarly, a data-intensive program will suffer from a bottleneck if the computer has inadequate data-movement resources relative to its compute power.

A frustrated compute-intensive program is said to be _compute bound_. A frustrated data-intensive program is said to be _bandwidth bound_.

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

```
program P1:                      computer A:  <high-bandwidth interconnect>

compute-intensive    ----------\              /----------  [program P1's needs
program P1                      \------------/              not being met by
                                                            computer A]

                                /------------\
                     ----------/              \----------

                     "input"       compute       "output"


program P2:                      computer B:  <low-bandwidth interconnect>

data-intensive                  /------------\
program P2           ----------/              \----------  [program P2's needs
                                                            not being met by
                     ----------\              /----------   computer B]
                                \------------/

                     "input"       compute       "output"
```

Here, computer A only performs well on programs with low arithmetic intensity, and 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).

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.

My architectural credo (one paragraph long)

____

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 – Floating Point Units – with data is expensive, not the FPUs themselves. Since communication capability is so limited, we should try to

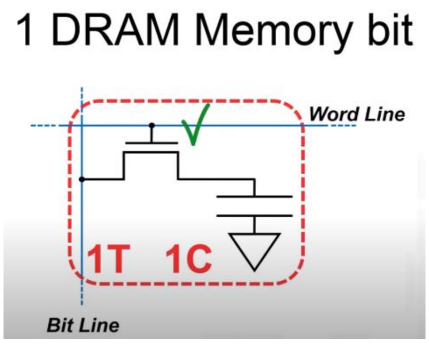
i) exploit locality whenever possible to reduce our need for communication bandwidth
ii) tolerate sufficient network/memory latency through some form of parallelism to keep the limited bandwidth resources in our high-latency network/memory systems _usefully_ busy.

That is, we want to extract the highest possible sustained operand bandwidth from our 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) ?

DRAM: Dynamic Random-Access Memory



This is a DRAM cell, 1 transistor and 1 capacitor makes-up 1 memory bit. This is 2 elements per cell. Flash Memory is 1 element per cell.

Unlike Flash, but like SRAM, DRAM is volatile. DRAM is very fast and has symmetrical read/write speed.

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 weakly 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.

Elaboration: Roughly speaking, there are two broad computer families, those that are _latency sensitive_ and those that are _bandwidth sensitive_.

I) LS: Monothreaded scalar processors have individual threads that regularly issue long-latency operations, for example, memory references to off-chip DRAM memory. As a result, these threads---and their processors---would normally spend the majority of their time stalled waiting for these operations to complete. Therefore, these processors critically depend on some powerful latency-avoidance mechanism, typically a cache hierarchy, to keep the waiting times within reasonable limits. Blithely trusting this latency-avoidance mechanism to always work, the LS vendors made no investment in bandwidth.

II) BS: Through put-oriented processors, such as vector processors, GPUs, and multithreaded processors, use parallelism of various kinds to keep many long-latency operations outstanding at all times, in such a way that the processor never stalls. However, to make this work requires a major investment in bandwidth, as well as using the limited bandwidth available as frugally as possible. These various parallelism mechanisms often depend on particular program properties being present, which affects the architecture's ability to exploit very large amounts of parallelism. For example, not all programs are vectorizable. In the GPU world, an additional concern is that irregular programs tend to clash with the implicit top-level memory hierarchy.

All indications are that only massively multithreaded (scalar) processors with exceptional bandwidth can hope to achieve _general-purpose_ latency tolerance.

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

```
      +---------+-----+----------+-----+---------+
CPU:  | I-cache |     |    RF    |     | D-cache |   RF = register file
      +---------+     +----------+     +---------+
      |                                          |
      |                                          |
      |            control circuitry             |
      |                                          |
      +------------------------------------------+
      |.PC.               PL                     |   PL = pipelined
      +------------------------------------------+        datapath
```

The processor's workbench is organized as a pipeline. (Strictly speaking, we should separate the pipeline into a passive _datapath_ and active _control_ circuitry). Other than PC, no pipeline component, including the so-called "ALU", is pictured.

Consider executing the (register-register) machine instruction 'mul.d f0,f2,f4', where 'f0', 'f2', and 'f4' are (floating-point) processor _registers_. Another (invisible) processor register, 'PC' (program counter), points---or recently pointed---to this instruction, i.e., the multiply instruction, which has to be fetched from "memory" 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 is to be fetched.

Assuming the multiply instruction has been fetched and decoded, we move on to actual execution. We localize 'f2' and 'f4' to the pipeline by retrieving them from the register file. We deliver both values to the ALU. We take the ALU's output and write it to the register file (to register 'f0'). Think of the instruction cache as just a bag of machine instructions that can be fetched using their addresses. The names on top are names of resources the pipeline uses. Since 'mul.d' is not a memory reference, the D-cache is not used.

Again, this instruction is not a _memory reference_, i.e., it is neither a _load_ nor a _store_. 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 pipeline, we need some place to put it. For this reason, every processor contains a large collection of _externally invisible_, or _non-ISA_, registers. Most of these are inside the pipeline.

The most important non-ISA register is the program counter (PC). (Elaboration: The ISA---or _instruction-set architecture_---defines much of the processor's actual _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 von Neumann computational model. It started from von Neumann's idea to use a single sequence of instructions to automatically control all aspects of a computer's behavior, including both its control structures and its arithmetic operations. The von Neumann machine model requires that there be pure _control-flow_ scheduling of arithmetic instructions, taken from the current thread, using the program counter (instruction pointer). In contrast, the _dataflow_ computational model schedules instructions as soon as their operands become available and does not use threads. Over the years, the von Neumann model has grown by accretion.

For example, interrupts are now part of the model. Today, it is taken to also mean that each processor should execute precisely one thread at a time, and that single-thread performance matters greatly. The most problematic rule in the von Neumann model is: "Every processor (or core) should have precisely one program counter". This may appear reasonable, but it is not. I will criticize this rule later.

Once we have instruction streams, we can use memory to store a program's instructions and supply them in program order to the processor, using their memory addresses. We can also use memory to store the program's data. Today, essentially all CPUs are von Neumann in that they have one _program counter_ (PC) per processor. Note that one program counter per processor implies one register file per processor.

PC is an externally invisible register that holds the address of the next instruction to be fetched. To implement the von Neumann _fetch-execute cycle_, we need a dedicated adder to increment the PC. We also need to implement branches.

Consider a program segment that is 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_).

Couldn't the processor simply switch to another program when the program it is currently running blocks? That depends on the context. If the program will be blocked for a 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 for memory references to complete.

The von Neumann machine model has also 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 thus essentially 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 should keep program counters and throw variables---for the most part---in the garbage can. What's wrong with variables? They are not suitable for parallel programming because of data races.

Basics (rehash)

______

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 already dominates transistor delay. More importantly, wire power already dominates transistor power. This double inversion was caused by shrinking feature sizes. If you think about it, computer design should be refocused to become interconnect design, at all space scales and inside all components.

When the relative value of cost parameters changes, 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. This assumption is no longer true.

1) General-purpose register machines may be divided into two families:

a) load-store architectures, including notably RISC machines, and

b) CISC architectures, including the DEC Vax and the IBM System/360, but also some of the earlier Intel x86s. 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 multicore processor, interconnect links cores and caches.

D) In the pipeline, 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. Moreover, communication is the major source of time spent and energy consumed. Programs differ in whether they engage in short-range or long-range communication. The interconnect may or may not have the capability to move whatever data needs to be moved fast enough at the space scale in question. As noted, communication determines power and performance.

3) 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
      subi   r1,r1,8     ; j := j - 1
      bne    r1,r2,loop  ; if r1 /= r2 then repeat

This emulates the high level code:

      'for' j := last 'downto' first 'do'
         a[j] := a[j] + c
      'od'

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

powers of ten

-24	yocto	ys	+24	yotta	YFs/s
-21	zepto	zs	+21	zetta	ZFs/s
-18	atto	as	+18	exa	EFs/s
-15	femto	fs	+15	peta	PFs/s
-12	pico	ps	+12	tera	TFs/s
-9	nano	ns	+9	giga	GFs/s
-6	micro	us	+6	mega	MFs/s
-3	milli	ms	+3	kilo	KFs/s

In column 1, 's' stands for second.

```
1 fs    = 1 femtosecond        = 10^-15 s
2 fs    = 2 femtoseconds       = 2 * 10^-15 s
```

In column 2, 'F' stands for flop and 'Fs' stands for flops.

```
1 PF/s  = 1 petaflop/second = 10^15 Fs/s
3 PFs/s = 3 petaflops/second = 3 * 10^15 Fs/s.
```

powers of two

+80	yotta	YBs
+70	zetta	ZBs
+60	exa	EBs
+50	peta	PBs
+40	tera	TBs
+30	giga	GBs
+20	mega	MBs
+10	kilo	KBs

'B' stands for byte and 'Bs' stands for bytes.

```
1 GB    = 1 gigabyte    = 2^30 Bs

3 GBs   = 3 gigabytes   = 3 *2^30 Bs.
```

All this is because 2^10 is roughly equal to 10^3.